目次

第 1 章　まえがき　1

第 2 章　導入　3
- 2.1　Rust は誰のためのものなの　3
- 2.2　この本は誰のためのものなの　5
- 2.3　この本の使い方　5
- 2.4　ソースコード　6

第 3 章　事始め　7
- 3.1　インストール　7
- 3.2　Hello, World!　10
- 3.3　Hello, Cargo!　13
- 3.4　まとめ　17

第 4 章　数当てゲームをプログラムする　19
- 4.1　新規プロジェクトの立ち上げ　19
- 4.2　予想を処理する　20
- 4.3　秘密の数字を生成する　25
- 4.4　予想と秘密の数字を比較する　29
- 4.5　ループで複数回の予想を可能にする　33
- 4.6　まとめ　37

第 5 章　一般的なプログラミングの概念　38
- 5.1　変数と可変性　38
- 5.2　データ型　42
- 5.3　関数　49

5.4 コメント 55
5.5 フロー制御 56
5.6 まとめ 63

第 6 章 所有権を理解する 64
6.1 所有権とは？ 64
6.2 参照と借用 74
6.3 スライス型 80
6.4 まとめ 86

第 7 章 構造体を使用して関係のあるデータを構造化する 87
7.1 構造体を定義し、インスタンス化する 87
7.2 構造体を使ったプログラム例 93
7.3 メソッド記法 98
7.4 まとめ 103

第 8 章 Enum とパターンマッチング 104
8.1 Enum を定義する 104
8.2 `match` フロー制御演算子 111
8.3 `if let` で簡潔なフロー制御 117
8.4 まとめ 118

第 9 章 モジュールを使用してコードを体系化し、再利用する 120
9.1 `mod` とファイルシステム 120
9.2 `pub` で公開するか制御する 129
9.3 異なるモジュールの名前を参照する 134
9.4 まとめ 139

第 10 章 一般的なコレクション 140
10.1 ベクタで一連の値を保持する 140
10.2 文字列で UTF-8 でエンコードされたテキストを保持する 145
10.3 ハッシュマップに値に紐づいたキーを格納する 153
10.4 まとめ 158

第 11 章 エラー処理 160
11.1 `panic!` で回復不能なエラー 160
11.2 `Result` で回復可能なエラー 164
11.3 `panic!` すべきかするまいか 172
11.4 まとめ 177

第 12 章 ジェネリック型、トレイト、ライフタイム 178
12.1 関数を抽出することで重複を取り除く . . . 178
12.2 ジェネリックなデータ型 . . . 181
12.3 トレイト: 共通の振る舞いを定義する . . . 189
12.4 ライフタイムで参照を有効化する . . . 198
12.5 ジェネリックな型引数、トレイト境界、ライフタイムを一度に . . . 211
12.6 総括 . . . 211

第 13 章 自動テストを書く 212
13.1 テストの記述法 . . . 213
13.2 テスト関数の解剖 . . . 213
13.3 テストの実行され方を制御する . . . 227
13.4 テストの体系化 . . . 232
13.5 まとめ . . . 238

第 14 章 入出力プロジェクト: コマンドラインプログラムを構築する 239
14.1 コマンドライン引数を受け付ける . . . 240
14.2 ファイルを読み込む . . . 243
14.3 リファクタリングしてモジュール性の向上とエラー処理 . . . 245
14.4 テスト駆動開発でライブラリの機能を開発する . . . 257
14.5 環境変数を取り扱う . . . 263
14.6 標準出力ではなく標準エラーにエラーメッセージを書き込む . . . 268
14.7 まとめ . . . 270

第 15 章 関数型言語の機能: イテレータとクロージャ 271
15.1 クロージャ: 環境をキャプチャできる匿名関数 . . . 271

第 16 章 ### クロージャで環境をキャプチャする 284
16.1 一連の要素をイテレータで処理する . . . 287
16.2 他のイテレータを生成するメソッド . . . 289
16.3 入出力プロジェクトを改善する . . . 295
16.4 パフォーマンス比較: ループ VS イテレータ . . . 299
16.5 まとめ . . . 301

第 17 章 Cargo と Crates.io についてより詳しく 302
17.1 リリースプロファイルでビルドをカスタマイズする . . . 302
17.2 Crates.io にクレートを公開する . . . 304
17.3 Cargo のワークスペース . . . 313
17.4 `cargo install` で Crates.io からバイナリをインストールする . . . 319

17.5 独自のコマンドで Cargo を拡張する . . . 319
17.6 まとめ . . . 320

第 18 章 スマートポインタ 321
18.1 ヒープのデータを指す Box<T>を使用する . . . 322
18.2 Deref トレイトでスマートポインタを普通の参照のように扱う . . . 327
18.3 Drop トレイトで片付け時にコードを走らせる . . . 333
18.4 Rc<T>は、参照カウント方式のスマートポインタ . . . 337
18.5 RefCell<T>と内部可変性パターン . . . 340
18.6 循環参照してメモリをリークすることもある . . . 349
18.7 まとめ . . . 357

第 19 章 恐れるな！ 非同期処理 359
19.1 スレッドを使用してコードを同時に走らせる . . . 360
19.2 メッセージ受け渡しを使ってスレッド間でデータを転送する . . . 367
19.3 状態共有非同期処理 . . . 373
19.4 Sync と Send トレイトで拡張可能な非同期処理 . . . 381
19.5 まとめ . . . 383

第 20 章 Rust のオブジェクト指向プログラミング機能 384
20.1 オブジェクト指向言語の特徴 . . . 384
20.2 トレイトオブジェクトで異なる型の値を許可する . . . 388
20.3 オブジェクト指向デザインパターンを実装する . . . 395
20.4 総括 . . . 408

第 21 章 パターンとマッチング 409
21.1 パターンが使用されることのある箇所全部 . . . 409
21.2 論駁可能性: パターンが合致しないか . . . 414
21.3 パターン記法全部 . . . 416
21.4 まとめ . . . 432

第 22 章 Advanced Features 433
22.1 Unsafe Rust . . . 433
22.2 Advanced Lifetimes . . . 443
22.3 Advanced Traits . . . 451
22.4 Advanced Types . . . 463
22.5 Advanced Functions and Closures . . . 469
22.6 Summary . . . 471

第 23 章 Final Project: Building a Multithreaded Web Server 472
23.1 Building a Single Threaded Web Server 473
23.2 Turning Our Single Threaded Server into a Multithreaded Server 483
23.3 Graceful Shutdown and Cleanup 505
23.4 Summary 515

第 24 章 Appendix 516
24.1 Appendix A: Keywords 516
24.2 Appendix B: Operators and Symbols 518
24.3 Appendix C: Derivable Traits 522
24.4 Appendix D: Macros 526
24.5 Appendix E: Translations of the Book 535

第 25 章 Appendix F - Newest Features 536
25.1 Field init shorthand 536
25.2 Returning from loops 537
25.3 Nested groups in `use` declarations 537
25.4 Inclusive ranges 538
25.5 128-bit integers 538

第 26 章 Appendix G - How Rust is Made and “Nightly Rust” 539

第 1 章

まえがき

常に明確だったわけではありませんが、Rust プログラミング言語は、根本的には、エンパワーメント (empowerment) に関してです: どんな種類のコードを現在書いているにせよ、Rust は、以前よりも幅広い領域で自信を持ってプログラムを組むこと、より遠くへ到達する原動力になります。

例えば、メモリ管理やデータ表現、非同期などの低レベルな詳細を扱う「システムレベル」の作業を例にとってください。伝統的にこの範囲のプログラミングは、難解と見なされ、有名な落とし穴を回避することを学ぶのに必要な年月を捧げた選ばれし者にだけアクセス可能です。鍛錬を積んだ者でさえコードが脅威やクラッシュ、頽廃 (たいはい) に開かれないように、注意深く行うのです。

Rust は、古い落とし穴を排除し、その過程で助けになる親しみ深い洗練された一連のツールを提供することで、これらの障壁を破壊します。低レベルな制御に「下がる」必要があるプログラマは、クラッシュやセキュリティホールの慣例的なリスクを負わず、変わりやすいツールチェーンのいい部分を学ぶ必要なく Rust でそうすることができます。さらにいいことに、言語は、スピードとメモリ使用の観点で効率的な信頼性の高いコードへと自然に導くよう設計されています。

既に低レベルコードに取り組んでいるプログラマは、Rust を使用して野望を高めることができます。例えば、Rust で並行性を導入することは、比較的低リスクな処理です: コンパイラが伝統的なミスを捕捉してくれるのです。そして、クラッシュや脅威を誤って導入しない自信とともにコードの積極的な最適化に取り組むことができるのです。

ですが、Rust は低レベルなシステムプログラミングに限定されているわけではありません。十分表現力豊かでエルゴノミックなので、コマンドラインアプリや Web サーバ、他の種類の書くのが好ましいコードを作れます。この本の後に両者の単純な例が見つかるでしょう。Rust と作業をすることで 1 つの領域から他の領域へと移動する技術を身につけることができます; ウェブアプリを書くことで Rust を学び、それからその同じ技術をラズベリーパイを対象にして適用できるのです。

この本は、ユーザの動力源となる Rust のポテンシャルを全て抱えています。あなたの Rust の知識のみをレベルアップさせるだけでなく、プログラマとして全般的なリーチや自信をもレベルアップさせる手助けをすることを意図した親しみ深いアプローチ可能なテキストです。従って、飛び込んで学

ぶ準備をしてください。Rust コミュニティへようこそ！

- ニコラス・マットサキス (Nicholas Matsakis) とアーロン・チューロン (Aaron Turon)

第 2 章

導入

The Rust Programming Language へようこそ。Rust に関する入門書です。

Rust プログラミング言語は、高速で信頼できるソフトウェアを書く手助けをしてくれます。高レベルのエルゴノミクス (脚注: ergonomics とは、人間工学的という意味。砕いて言えば、人間に優しいということ) と低レベルの制御は、しばしばプログラミング言語の設計においてトレードオフの関係になります; Rust は、その衝突に挑戦しています。バランスのとれた強力な技術の許容量と素晴らしい開発者経験を通して、Rust は伝統的にそれらの制御と紐付いていた困難全てなしに低レベルの詳細 (メモリ使用など) を制御する選択肢を与えてくれます。

2.1 Rust は誰のためのものなの

Rust は、様々な理由により多くの人にとって理想的です。いくつか最も重要なグループを見ていきましょう。

2.1.1 開発者チーム

Rust は、いろんなレベルのシステムプログラミングの知識を持つ開発者の巨大なチームとコラボレートするのに生産的なツールであると証明してきています。低レベルコードは様々な種類の微細なバグを抱える傾向があり、そのようなバグは他の言語だと広範なテストと、経験豊富な開発者による注意深いコードレビューによって捕捉されるだけのものです。Rust においては、コンパイラが非同期のバグも含めたこのようなとらえどころのないバグのあるコードをコンパイルするのを拒むことで、門番の役割を担います。コンパイラとともに取り組むことで、チームはバグを追いかけるよりもプログラムのロジックに集中することに、より時間を費やせるのです。

Rust はまた、現代的な開発ツールをシステムプログラミング世界に導入します。

- Cargo は、付属の依存マネージャ兼ビルドツールで、依存を追加、コンパイル、管理すること

を楽かつ、Rust エコシステムを通じて矛盾させません。
- Rustfmt は開発者の間で矛盾のないコーディングスタイルを保証します。
- Rust Language Server は IDE(Intefrated Development Environment) にコード補完とインラインのエラーメッセージの統合の源となります。

これらや他のツールを Rust のエコシステムで使用することで、開発者はシステムレベルのコードを記述しつつ、生産的になれます。

2.1.2　学生

Rust は、学生やシステムの概念を学ぶことに興味のある方向けです。Rust を使用して、多くの人が OS 開発などの話題を学んできました。コミュニティはとても暖かく、喜んで学生の質問に答えてくれます。この本のような努力を通じて、Rust チームはシステムの概念を多くの人、特にプログラミング初心者にとってアクセス可能にしたいと考えています。

2.1.3　企業

数百の企業が、大企業、中小企業を問わず、様々なタスクにプロダクションで Rust を使用しています。そのタスクには、コマンドラインツール、Web サービス、DevOps ツール、組み込みデバイス、オーディオとビデオの解析および変換、暗号通貨、生物情報学、サーチエンジン、IoT アプリケーション、機械学習、Firefox ウェブブラウザの主要部分さえ含まれます。

2.1.4　オープンソース開発者

Rust は、Rust プログラミング言語やコミュニティ、開発者ツール、ライブラリを開発したい方向けです。あなたが Rust 言語に貢献されることを心よりお待ちしております。

2.1.5　スピードと安定性に価値を見出す方

Rust は、スピードと安定性を言語に渇望する方向けです。ここでいうスピードとは、Rust で作れるプログラムのスピードとソースコードを書くスピードのことです。Rust コンパイラのチェックにより、このようなチェックがない言語において開発者が変更するのを恐れてしまう脆いレガシーコードとは対照的に、機能の追加とリファクタリングを通して安定性を保証してくれます。ゼロコスト抽象化と、手で書いたコードと同等の速度を誇る低レベルコードにコンパイルされる高レベル機能を努力することで、Rust は安全なコードを高速なコードにもしようと努力しています。

Rust 言語がサポートを望んでいる方全員の完全なリストを提供はしていないものの、名前を出した方は、最大の出資者の一部です。総合すると、Rust の最大の野望は、プログラマが数十年間受け入れてきた代償の二分を排除することです: つまり、安全性と生産性、スピードとエルゴノミクスです。Rust を試してみて、その選択が自分に合っているか確かめてください。

2.2　この本は誰のためのものなの

この本は、あなたが他のプログラミング言語でコードを書いたことがあることを想定していますが、具体的にどの言語かという想定はしません。幅広い分野のプログラミング背景からの人にとってこの資料を広くアクセスできるようにしようとしてきました。プログラミングとはなんなのかやそれについて考える方法について多くを語るつもりはありません。もし、完全なプログラミング初心者であれば、プログラミング入門を特に行う本を読むことでよりよく提供されるでしょう。

2.3　この本の使い方

一般的に、この本は、順番に読み進めていくことを前提にしています。後の章は、前の章の概念の上に成り立ち、前の章では、ある話題にさほど深入りしない可能性があります; 典型的に後ほどの章で同じ話題を再度しています。

この本には 2 種類の章があるとわかるでしょう: 概念の章とプロジェクトの章です。概念の章では、Rust の一面を学ぶでしょう。プロジェクトの章では、それまでに学んだことを適用して一緒に小さなプログラムを構築します。2、12、20 章がプロジェクトの章です。つまり、残りは概念の章です。

加えて第 2 章は、Rust 言語への実践的な導入です。概念を高度に講義し、後ほどの章で追加の詳細を提供します。今すぐ Rust の世界に飛び込みたいなら、第 2 章こそがそのためのものです。第 3 章は他のプログラミング言語の機能に似た Rust の機能を講義していますが、最初その 3 章すら飛ばして、まっすぐに第 4 章に向かい、Rust の所有権システムについて学びたくなる可能性があります。しかしながら、あなたが次に進む前に全ての詳細を学ぶことを好む特別に几帳面な学習者なら、第 2 章を飛ばして真っ先に第 3 章に行き、その詳細を適用するプロジェクトに取り組みたくなった時に第 2 章に戻りたくなる可能性があります。

第 5 章は、構造体とメソッドについて議論し、第 6 章は enum、`match`式、`if let`フロー制御構文を講義します。構造体と enum を使用して Rust において独自の型を作成します。

第 7 章では、Rust のモジュールシステムと自分のコードとその公開された API(Application Programming Interface) を体系化するプライバシー規則について学びます。第 8 章では、ベクタ、文字列、ハッシュマップなどの標準ライブラリが提供する一般的なコレクションデータ構造の一部を議論します。第 9 章では、Rust のエラー処理哲学とテクニックを探求します。

第 10 章ではジェネリクス、トレイト、ライフタイムについて深入りし、これらは複数の型に適用されるコードを定義する力をくれます。第 11 章は、完全にテストに関してで、Rust のプログラムのロジックが正しいことを保証する安全性保証があってもなお、必要になります。第 12 章では、ファイル内のテキストを検索する `grep`コマンドラインツールの一部の機能を自身で構築します。このために、以前の章で議論した多くの概念を使用します。

第 13 章はクロージャとイテレータを探求します。これらは、関数型プログラミング言語由来の Rust の機能です。第 14 章では、Cargo をより詳しく調査し、他人と自分のライブラリを共有する最

善の策について語ります。第 15 章では、標準ライブラリが提供するスマートポインタとその機能を可能にするトレイトを議論します。

第 16 章では、非同期プログラミングの異なるモデルを見ていき、Rust が恐れなしに複数のスレッドでプログラムする手助けをする方法を語ります。第 17 章では、馴染み深い可能性のあるオブジェクト指向プログラミングの原則と Rust のイディオムがどう比較されるかに目を向けます。

第 18 章は、パターンとパターンマッチングのリファレンスであり、これらは Rust プログラムを通して、考えを表現する強力な方法になります。第 19 章は、unsafe Rust やライフタイム、トレイト、型、関数、クロージャの詳細を含む、興味のある高度な話題のスモーガスボード (脚注: 日本でいうバイキングのこと) を含みます。

第 20 章では、低レベルなマルチスレッドの Web サーバを実装するプロジェクトを完成させます！

最後に、言語についての有用な情報をよりリファレンスっぽい形式で含むおまけがあります。おまけ A は Rust のキーワードを講義します。おまけ B は、Rust の演算子とシンボル、おまけ C は、標準ライブラリが提供する継承可能なトレイト、おまけ D はマクロを講義します。

この本を読む間違った方法なんてありません: 飛ばしたければ、どうぞご自由に！ 混乱したら、前の章に戻らなければならない可能性もあります。ですが、自分に合った方法でどうぞ。

Rust を学ぶ過程で重要な部分は、コンパイラが表示するエラーメッセージを読む方法を学ぶことです: それは動くコードへと導いてくれます。そのため、各場面でコンパイラが表示するエラーメッセージとともに、コンパイルできないコードの例を多く提供します。適当に例を選んで走らせたら、コンパイルできないかもしれないことを知ってください！ 周りのテキストを読んで実行しようとしている例がエラーになることを意図しているのか確認することを確かめてください。ほとんどの場合、コンパイルできないあらゆるコードの正しいバージョンへと導きます。

2.4　ソースコード

この本が生成されるソースファイルは、GitHub で見つかります。

この本はオープンソースです。間違いを見つけたら、[Github で]issue を送ったり、プルリクエストを送ったりするのを躊躇 (ためら) わないでください。詳しくは、[CONTRIBUTING.md] をご参照ください

第 3 章

事始め

Rust の旅を始めましょう！ この章では、以下のことを議論します:

- Rust を Linux、macOS、Windows にインストールする
- ”Hello, world!” と出力するプログラムを書く
- `cargo`という Rust のパッケージマネージャ兼ビルドシステムを使用する

3.1 インストール

最初の手順は、Rust をインストールすることです。Rust は、`rustup`という Rust のバージョンと関連するツールを管理するコマンドラインツールを使用して、ダウンロードします。ダウンロードするには、インターネット接続が必要でしょう。

以下の手順で最新の安定版の Rust コンパイラをインストールします。この本の例と出力は全て、安定版の Rust1.21.0 を使用しています。Rust の安定性保証により、現在この本の例でコンパイルできるものは、新しいバージョンになってもコンパイルでき続けることを保証します。出力は、バージョンによって多少異なる可能性があります。Rust は頻繁にエラーメッセージと警告を改善しているからです。言い換えると、どんな新しいバージョンでもこの手順に従ってインストールした安定版なら、この本の内容で想定通りに動くはずです。

3.1.1 コマンドライン表記

この章及び、本を通して、端末で使用するなんらかのコマンドを示すことがあります。読者が入れるべき行は、全て`$`で始まります。`$`文字を入れる必要はありません; 各コマンドの開始を示しているだけです。多くのチュートリアルで普通のユーザとして実行するコマンドには`$`、管理者として実行するコマンドには`#`を使用するという習慣を用います。`$`で始まらない行は、典

型的には直前のコマンドの出力を示します。また、PowerShell 限定の例は、$ではなく、>を使用します。

3.1.2　Linux と macOS に Rustup をインストールする

Linux か macOS を使用しているなら、端末を開き、以下のコマンドを入力してください:

```
$ curl https://sh.rustup.rs -sSf | sh
```

このコマンドはスクリプトをダウンロードし、rustupツールのインストールを開始し、Rust の最新の安定版をインストールします。パスワードを求められる可能性があります。インストールがうまく行けば、以下の行が出現するでしょう:

```
Rust is installed now. Great!
```

もちろん、curl URL | shを使用してソフトをインストールすることを信用できないのなら、お好きなようにスクリプトをダウンロードし、調べ、走らせることができます。

インストールスクリプトは、Rust を次回のログイン後にシステムの PATH に自動的に追加します。端末を再起動するのではなく、いますぐに Rust を使用し始めたいのなら、シェルで以下のコマンドを実行して Rust をシステムの PATH に手動で追加します:

```
$ source $HOME/.cargo/env
```

また、以下の行を*~/.bash_profile*に追加することもできます:

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

さらに、なんらかの類のリンカーが必要になるでしょう。既にインストールされている可能性が高いものの、Rust プログラムのコンパイルを試みて、リンカーが実行できないというエラーが出たら、インストールする必要があるでしょう。C コンパイラをインストールすれば、正しいリンカーを普通インストールすることができるでしょう。自分のプラットフォームのドキュメンテーションを見て C コンパイラのインストール方法を確認してください。一般的な Rust パッケージの中には、C コードに依存し、C コンパイラが必要になるものもあるので、いずれにせよ今インストールする価値はあるかもしれません。

3.1.3　Windows で Rustup をインストールする

Windows では、https://www.rust-lang.org/en-US/install.html に行き、手順に従って Rust をインストールしてください。インストールの途中で、Visual Studio2013 以降用の C++ ビルドツールも必要になるという旨のメッセージが出るでしょう。ビルドツールを取得する最も簡単な方法は、Visual Studio 2017 用のビルドツールをインストールすることです。ツールは、他のツール及びフレームワークのセクションにあります。

これ以降、*cmd.exe* と PowerShell の両方で動くコマンドを使用します。特定の違いがあったら、どちらを使用すべきか説明します。

3.1.4　Rustup なしでカスタムインストール

なんらかの理由で `rustup`を使用しないことを好むのなら、他の選択肢を求めて、Rust のインストールページをご覧ください。

3.1.5　更新及びアンインストール

`rustup`経由で Rust をインストールしたら、最新版への更新は、簡単になります。シェルから、以下の更新スクリプトを実行してください:

```
$ rustup update
```

Rust と `rustup`をアンインストールするには、シェルから以下のアンインストールスクリプトを実行してください:

```
$ rustup self uninstall
```

3.1.6　トラブルシューティング

Rust が正常にインストールされているか確かめるには、シェルを開いて以下の行を入力してください:

```
$ rustc --version
```

バージョンナンバー、コミットハッシュ、最新の安定版がリリースされたコミット日時が以下のフォーマットで表示されるのを目撃するはずです。

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

この情報が見れたら、Rust のインストールに成功しました！ この情報が出ず、Windows を使っているなら、Rust が `%PATH%`システム環境変数にあることを確認してください。全て正常で、それでも Rust が動かないなら、助力を得られる場所はたくさんあります。最も簡単なのが irc.mozilla.org の#rust IRC チャンネルで、Mibbit を通してアクセスできます。そのアドレスで、助けてくれる他の Rustacean(自分たちを呼ぶバカなニックネーム) とチャットできます。他の素晴らしいリソースには、ユーザ・フォーラムと Stack Overflow が含まれます。

3.1.7　ローカルのドキュメンテーション

インストーラは、ドキュメンテーションの複製もローカルに含んでいるので、オフラインで閲覧することができます。ブラウザでローカルのドキュメンテーションを開くには、`rustup doc`を実行してください。

標準ライブラリにより型や関数が提供され、それがなんなのかや使用方法に確信が持てない度に、API ドキュメンテーションを使用して探してください！

3.2　Hello, World!

Rust をインストールしたので、最初の Rust プログラムを書きましょう。新しい言語を学ぶ際に、”Hello, world!” というテキストを画面に出力する小さなプログラムを書くことは伝統的なことなので、ここでも同じようにしましょう！

> 注釈: この本は、コマンドラインに基本的な馴染みがあることを前提にしています。Rust は、編集やツール、どこにコードがあるかについて特定の要求をしないので、コマンドラインではなく IDE を使用することを好むのなら、どうぞご自由にお気に入りの IDE を使用してください。今では、多くの IDE がなんらかの形で Rust をサポートしています; 詳しくは、IDE のドキュメンテーションをご覧ください。最近、Rust チームは優れた IDE サポートを有効にすることに注力し、その前線で急激に成果があがっています！

3.2.1　プロジェクトのディレクトリを作成する

Rust コードを格納するディレクトリを作ることから始めましょう。Rust にとって、コードがどこにあるかは問題ではありませんが、この本の練習とプロジェクトのために、ホームディレクトリに *projects* ディレクトリを作成してプロジェクトを全てそこに保管することを推奨します。

端末を開いて以下のコマンドを入力し、*projects* ディレクトリと *projects* ディレクトリ内に、”Hello, world!” プロジェクトのディレクトリを作成してください。

Linux と macOS なら、こう入力してください:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Windows の cmd なら、こう:

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

Windows の PowerShell なら、こう:

```
> mkdir $env:USERPROFILE\projects
> cd $env:USERPROFILE\projects
> mkdir hello_world
> cd hello_world
```

3.2.2　Rust プログラムを書いて走らせる

次にソースファイルを作り、*main.rs* と呼んでください。Rust のファイルは常に*.rs* という拡張子で終わります。ファイル名に 2 単語以上使っているなら、アンダースコアで区切ってください。例えば、*helloworld.rs* ではなく、*hello_world.rs*を使用してください。

さて、作ったばかりの *main.rs* ファイルを開き、リスト 1-1 のコードを入力してください。

ファイル名: main.rs

```
fn main() {
    println!("Hello, world!");
}
```

リスト 1-1: ”Hello, world!” と出力するプログラム

ファイルを保存し、端末ウィンドウに戻ってください。Linux か macOS なら、以下のコマンドを打ってファイルをコンパイルし、実行してください:

```
$ rustc main.rs
$ ./main
Hello, world!
```

Windows なら、`./main`の代わりに`.\main.exe`と打ちます。

```
> rustc main.rs
> .\main.exe
Hello, world!
```

OS に関わらず、`Hello, world!`という文字列が端末に出力されるはずです。この出力が見れないなら、「トラブルシューティング」節に立ち戻って、助けを得る方法を参照してください。

`Hello, world!`が確かに出力されたら、おめでとうございます！ 正式に Rust プログラムを書きました。Rust プログラマになったのです！ ようこそ！

3.2.3　Rust プログラムの解剖

”Hello, world!” プログラムでちょうど何が起こったのか詳しく確認しましょう。こちらがパズルの最初のピースです:

```
fn main() {

}
```

これらの行で Rust で**関数**を定義しています。`main`関数は特別です: 常に全ての実行可能な Rust プログラムで走る最初のコードになります。1 行目は、引数がなく、何も返さない `main`という関数を宣言しています。引数があるなら、かっこ ((と)) の内部に入ります。

また、関数の本体が波括弧 ({と}) に包まれていることにも注目してください。Rust では、全ての関数本体の周りにこれらが必要になります。スペースを 1 つあけて、開き波括弧を関数宣言と同じ行に配置するのがいいスタイルです。

これを執筆している時点では、rustfmtと呼ばれる自動整形ツールは開発中です。複数の Rust プロジェクトに渡って、標準的なスタイルに固執したいなら、rustfmtは特定のスタイルにコードを整形してくれます。Rust チームは、最終的に rustcのように標準的な Rust の配布に含むことを計画しています。従って、この本を読んだ時期によっては、既にコンピュータにインストールされている可能性もあります！ 詳細は、オンラインのドキュメンテーションを確認してください。

main関数内には、こんなコードがあります:

```
1     println!("Hello, world!");
```

この行が、この小さなプログラムの全作業をしています: テキストを画面に出力するのです。ここで気付くべき重要な詳細が 4 つあります。まず、Rust のスタイルは、タブではなく、4 スペースでインデントするということです。

2 番目に println!は Rust の**マクロ**を呼び出すということです。代わりに関数を呼んでいたら、println(!なし) と入力されているでしょう。Rust のマクロについて詳しくは、おまけ D で議論します。とりあえず、!を使用すると、普通の関数ではなくマクロを呼んでいるのだということを知っておくだけでいいでしょう。

3 番目に、"Hello, world!"**文字列**が見えます。この文字列を引数として println!に渡し、この文字列が画面に表示されているのです。

4 番目にこの行をセミコロン、;で終え、この式が終わり、次の式の準備ができていると示唆していることです。Rust コードのほとんどの行は、セミコロンで終わります。

3.2.4　コンパイルと実行は個別のステップ

新しく作成したプログラムをちょうど実行したので、その途中の手順を調査しましょう。

Rust プログラムを実行する前に、以下のように、rustcコマンドを入力し、ソースファイルの名前を渡すことで、Rust コンパイラを使用してコンパイルしなければなりません。

```
$ rustc main.rs
```

あなたに C や C++ の背景があるなら、これは gccや clangと似ていると気付くでしょう。コンパイルに成功後、Rust はバイナリの実行可能ファイルを出力します。

Linux、macOS、Windows の PowerShell なら、シェルで以下のように lsコマンドを入力することで実行可能ファイルを見られます:

```
$ ls
main  main.rs
```

Windows の CMD なら、以下のように入力するでしょう:

```
> dir /B %= the /B option says to only show the file names =%
        %= /Bオプションは、ファイル名だけを表示することを宣言する =%
main.exe
main.pdb
main.rs
```

これは、*.rs* 拡張子のソースコードファイル、実行可能ファイル (Windows なら *main.exe*、他のプラットフォームでは、*main*)、そして、CMD を使用しているなら、*.pdb* 拡張子のデバッグ情報を含むファイルを表示します。ここから、*main* か *main.exe* を走らせます。このように:

```
$ ./main # or .\main.exe on Windows
        # または、Widnowsなら.\main.exe
```

main.rs が”Hello, world!” プログラムなら、この行は `Hello, world!` と端末に出力するでしょう。

Ruby や Python、JavaScript などの動的言語により造詣が深いなら、プログラムのコンパイルと実行を個別の手順で行うことに慣れていない可能性があります。Rust は *AOT* **コンパイル** (ahead-of-time; 脚注: 予め) 言語です。つまり、プログラムをコンパイルし、実行可能ファイルを誰かにあげ、あげた人が Rust をインストールしなくても実行できるわけです。誰かに*.rb*、*.py*、*.js* ファイルをあげたら、それぞれ Ruby、Python、JavaScript の実装がインストールされている必要があります。ですが、そのような言語では、プログラムをコンパイルし実行するには、1 コマンドしか必要ないのです。全ては言語設計においてトレードオフなのです。

簡単なプログラムなら `rustc`でコンパイルするだけでも十分ですが、プロジェクトが肥大化してくると、オプションを全て管理し、自分のコードを簡単に共有したくなるでしょう。次は、Cargo ツールを紹介します。これは、現実世界の Rust プログラムを書く手助けをしてくれるでしょう。

3.3　Hello, Cargo!

Cargo は、Rust のビルドシステム兼、パッケージマネージャです。ほとんどの Rustacean はこのツールを使用して、Rust プロジェクトの管理をしています。Cargo は、コードのビルドやコードが依存しているライブラリのダウンロード、それらのライブラリのビルド (コードが必要とするライブラリを我々は、**依存**と呼んでいます) などの多くの仕事を扱ってくれるからです。

今までに書いたような最も単純な Rust プログラムは、依存がありません。従って、”Hello, world!” プロジェクトを Cargo を使ってビルドしても、Cargo のコードをビルドする部分しか使用しないでしょう。より複雑な Rust プログラムを書くにつれて、依存を追加し、Cargo でプロジェクトを開始したら、依存の追加は、遥かに簡単になるのです。

Rust プロジェクトの大多数が Cargo を使用しているので、これ以降この本では、あなたも Cargo を使用していることを想定します。Cargo は、「インストール」節で議論した公式のインストーラを使用していれば、勝手にインストールされます。Rust を他の何らかの手段でインストールした場合、以下のコマンドを端末に入れて Cargo がインストールされているか確かめてください:

```
$ cargo --version
```

バージョンナンバーが見えたら、インストールされています！ command not foundなどのエラーが見えたら、自分のインストール方法を求めてドキュメンテーションを見、Cargo を個別にインストールする方法を決定してください。

3.3.1　Cargo でプロジェクトを作成する

Cargo を使用して新しいプロジェクトを作成し、元の”Hello, world!” プロジェクトとどう違うかを見ましょう。*projects* ディレクトリ (あるいはコードを格納すると決めた場所) に戻ってください。それから、OS に関わらず、以下を実行してください:

```
$ cargo new hello_cargo --bin
$ cd hello_cargo
```

最初のコマンドは、*hello*cargo という新しいバイナリの実行可能ファイルを作成します。*cargo new* に渡した*--bin*引数が、ライブラリとは対照的に実行可能なアプリケーション *(*よく単にバイナリと呼ばれる*)* を作成します。プロジェクトを hello*cargo* と名付け、Cargo は、そのファイルを同名のディレクトリに作成します。

*hello_cargo ディレクトリに行き、ファイルを列挙してください。*Cargo* が *2* つのファイルと *1* つのディレクトリを生成してくれたことがわかるでしょう:* Cargo.toml ファイルと、中に main.rs ファイルがある src ディレクトリです。また、.gitignore*ファイルと共に、新しい Git リポジトリも初期化しています。

> 注釈: Git は一般的なバージョンコントロールシステムです。cargo newを変更して、異なるバージョンコントロールシステムを使用したり、--vcsフラグを使用して何もバージョンコントロールシステムを使用しないようにもできます。cargo new --helpを走らせて、利用可能なオプションを確認してください。

お好きなテキストエディタで *Cargo.toml* を開いてください。リスト 1-2 のコードのような見た目のはずです。

ファイル名: Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

リスト 1-2: cargo newで生成される *Cargo.toml* の中身

このファイルは TOML(Tom’s Obvious, Minimal Language) フォーマットで、Cargo の設定フォーマットです。

最初の行の [package]は、後の文がパッケージを設定していることを示すセクションヘッダーです。もっと情報を追加すれば、別のセクションも追加するでしょう。

その後の 3 行が、Cargo がプログラムをコンパイルするのに必要な設定情報をセットします: 名前、バージョン、誰が書いたかです。Cargo は名前と E メールの情報を環境から取得するので、その情報が正しくなければ、今修正してそれから保存してください。

最後の行の [dependencies]は、プロジェクトの依存を列挙するためのセクションの始まりです。Rust では、パッケージのコードは**クレート**として参照されます。このプロジェクトでは何も他のクレートは必要ありませんが、第 2 章の最初のプロジェクトでは必要なので、その時にはこの依存セクションを使用するでしょう。

では、*src/main.rs* を開いて覗いてみてください:

ファイル名: src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

ちょうどリスト 1-1 で書いたように、Cargo は”Hello, world!” プログラムを生成してくれています。ここまでで、前のプロジェクトと Cargo が生成したプロジェクトの違いは、Cargo が *src* ディレクトリにコードを配置し、最上位のディレクトリに *Cargo.toml* 設定ファイルがあることです。

Cargo は、ソースファイルが *src* ディレクトリにあることを期待します。プロジェクトの最上位のディレクトリは、README ファイル、ライセンス情報、設定ファイル、あるいは、他のコードに関連しないもののためのものです。Cargo を使用すると、プロジェクトを体系化する手助けをしてくれます。全てのものには収まるべき場所があり、全てがその場所にあるのです。

Hello, world!プロジェクトのように、Cargo を使用しないプロジェクトを開始したら、実際に Cargo を使用するプロジェクトに変換することができます。プロジェクトのコードを *src* ディレクトリに移動し、適切な *Cargo.toml* ファイルを作成してください。

3.3.2　Cargo プロジェクトをビルドし、実行する

さて、Cargo で”Hello, world!” プログラムをビルドし、実行する時の違いに目を向けましょう！*hello_cargo*ディレクトリから、以下のコマンドを入力してプロジェクトをビルドしてください:

```
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

このコマンドは、カレントディレクトリではなく、*target/debug/hello*cargo*(あるいは Windows* なら、target/debug/hello*cargo.exe*) の実行可能ファイルを作成します。以下のコマンドで実行可能ファイルを実行できます:

```
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe on Windows
                             # あるいは、Windowsなら、.\target\debug\hello_cargo.exe
Hello, world!
```

全てがうまくいけば、Hello, world!が端末に出力されるはずです。初めて cargo buildを実行すると、Cargo が最上位に新しいファイルも作成します: *Cargo.lock* です。このファイルは、自分のプロジェクトの正確な依存のバージョンを追いかけます。このプロジェクトには依存がないので、ファイルはやや空っぽです。絶対にこのファイルを手動で変更する必要はないでしょう; Cargo が中身を管理してくれるのです。

cargo buildでプロジェクトをビルドし、./target/debug/hello_cargoで実行したばかりですが、cargo runを使用して、コードをコンパイルし、それから吐かれた実行可能ファイルを全部 1 コマンドで実行することもできます:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

今回は、Cargo が hello_cargoをコンパイルしていることを示唆する出力がないことに気付いてください。Cargo はファイルが変更されていないことを推論したので、単純にバイナリを実行したのです。ソースコードを変更していたら、Cargo は実行前にプロジェクトを再ビルドし、こんな出力を目の当たりにしたでしょう:

```
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

Cargo は cargo checkというコマンドも提供しています。このコマンドは、迅速にコードを確認し、コンパイルできることを確かめますが、実行可能ファイルは生成しません:

```
$ cargo check
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

何故、実行可能ファイルが欲しくないのでしょうか？ しばしば、cargo checkは、cargo buildよりも遥かに速くなります。実行可能ファイルを生成する手順を飛ばすからです。コードを書いている際に継続的に自分の作業を確認するのなら、cargo checkを使用すると、その過程が高速化されます！そのため、多くの Rustacean は、プログラムを書く際にコンパイルできるか確かめるために定期的にcargo checkを実行します。そして、実行可能ファイルを使用できる状態になったら、cargo buildを走らせるのです。

ここまでに Cargo について学んだことをおさらいします:

- cargo buildか cargo checkでプロジェクトをビルドできる。
- プロジェクトのビルドと実行を 1 ステップで cargo runでできる。
- ビルドの結果をコードと同じディレクトリに保存するのではなく、Cargo は *target/debug* ディレクトリに格納する。

Cargo を使用する追加の利点は、使用している OS に関わらず、同じコマンドが使用できることです。故にこの時点で、Windows と Linux 及び macOS で特定の手順を提供することは最早なくなります。

3.3.3　リリースビルドを行う

プロジェクトを最終的にリリースする準備ができたら、`cargo build --release`を使用して、最適化を行なってコンパイルすることができます。このコマンドは、*target/debug* ではなく、*target/release* に実行可能ファイルを作成します。最適化は、Rust コードの実行を速くしてくれますが、オンにするとプログラムをコンパイルする時間が延びます。このため、2 つの異なるプロファイルがあるのです: 頻繁に再ビルドをかけたい開発用と、繰り返し再ビルドすることはなく、できるだけ高速に動いてユーザにあげる最終的なプログラムをビルドする用のです。コードの実行時間をベンチマークするなら、`cargo build --release`を確実に実行し、*target/release* の実行可能ファイルでベンチマークしてください。

3.3.4　習慣としての Cargo

単純なプロジェクトでは、Cargo は単に `rustc`を使用する以上の価値を生みませんが、プログラムが複雑になるにつれて、その価値を証明するでしょう。複数のクレートからなる複雑なプロジェクトでは、Cargo にビルドを調整してもらうのが遥かに簡単です。

`hello_cargo`プロジェクトは単純ではありますが、今では、Rust のキャリアを通じて使用するであろう本物のツールを多く使用するようになりました。事実、既存のどんなプロジェクトに取り組むにも、以下のコマンドを使用して、Git でコードをチェックアウトし、そのプロジェクトのディレクトリに移動し、ビルドできます:

```
$ git clone someurl.com/someproject
$ cd someproject
$ cargo build
```

Cargo についてより詳しく知るには、ドキュメンテーションを確認してください。

3.4　まとめ

既に Rust の旅の素晴らしいスタートを切っています！ この章では、以下の方法を学びました:

- `rustup`で最新の安定版の Rust をインストールする方法
- 新しい Rust のバージョンに更新する方法
- ローカルにインストールされたドキュメンテーションを開く方法
- 直接 `rustc`を使用して”Hello, world!” プログラムを書き、実行する方法
- Cargo の慣習を使用して新しいプロジェクトを作成し、実行する方法

より中身のあるプログラムをビルドし、Rust コードの読み書きに慣れるいいタイミングです。故に、次の章では、数当てゲームを構築します。むしろ一般的なプログラミングの概念が Rust でどう動くのか学ぶことから始めたいのであれば、第 3 章を見て、それから第 2 章に戻ってください。

第 4 章

数当てゲームをプログラムする

実物のプロジェクトに一緒に取り組むことで、Rust の世界へ飛び込みましょう！ この章では、実際のプログラム内で使用しながらいくつかの一般的な Rust の概念に触れます。let 文、match 式、メソッド、関連関数、外部クレートの使用などについて学ぶでしょう！ 後ほどの章でこれらの概念について深く知ることになります。この章では、基礎部分だけにしましょう。

古典的な初心者向けのプログラミング問題を実装してみましょう: 数当てゲームです。これは以下のように動作します: プログラムは 1 から 100 までの乱数整数を生成します。そしてプレーヤーに予想を入力するよう促します。予想を入力したら、プログラムは、その予想が少なすぎたか多すぎたかを出力します。予想が当たっていれば、ゲームは祝福メッセージを表示し、終了します。

4.1 新規プロジェクトの立ち上げ

新規プロジェクトを立ち上げるには、第 1 章で作成した *projects* ディレクトリに行き、Cargo を使って以下のように新規プロジェクトを作成します。

```
$ cargo new guessing_game --bin
$ cd guessing_game
```

最初のコマンド `cargo new`は、プロジェクト名を第 1 引数に取ります (`guessing_game`ですね)。`--bin`というフラグは、Cargo にバイナリ生成プロジェクトを作成させます。第 1 章のものと似ていますね。2 番目のコマンドで新規プロジェクトのディレクトリに移動します。

生成された *Cargo.toml* ファイルを見てください:

ファイル名: Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["名前 <you@example.com>"]

[dependencies]
```

もし、Cargo があなたの環境から取得した作者情報が間違っていたら、ファイルを編集して保存し直してください。

第 1 章でも見かけたように、`cargo new`コマンドは、”Hello, world!” プログラムを生成してくれます。*src/main.rs* ファイルをチェックしてみましょう:

ファイル名: src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

さて、この”Hello, world!” プログラムをコンパイルし、`cargo run`コマンドを使用して、以前と同じように動かしてみましょう:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
     Running `target/debug/guessing_game`
Hello, world!
```

`run`コマンドは、プロジェクトに迅速に段階を踏んで取り掛かる必要がある場合に有用であり、次のステップに進む前に各段階を急速にテストして、このゲームではそれを行います。

再度 *src/main.rs* ファイルを開きましょう。ここにすべてのコードを書いていきます。

4.2　予想を処理する

数当てプログラムの最初の部分は、ユーザに入力を求め、その入力を処理し、予期した形式になっていることを確認します。手始めにプレーヤーが予想を入力できるようにしましょう。リスト 2-1 のコードを *src/main.rs* に入力してください。

ファイル名: src/main.rs

```
use std::io;

fn main() {
    println!("Guess the number!");          // 数を当ててごらん

    println!("Please input your guess.");   // ほら、予想を入力してね

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");     // 行の読み込みに失敗しました

    println!("You guessed: {}", guess);     // 次のように予想しました: {}
}
```

リスト 2-1: ユーザに予想を入力してもらい、それを出力するコード

> 注釈: The programming language Rust 第 1 版の翻訳者によると、ソースコードのコメント中以外に日本語文字があるとコンパイルに失敗することがあるそうなので、文字列の英語は、コメントに和訳を載せます。また、重複する内容の場合には、最初の 1 回だけ掲載するようにします。

このコードには、たくさんの情報が詰め込まれてますね。なので、行ごとに見ていきましょう。ユーザ入力を受け付け、結果を出力するためには、`io`(入/出力) ライブラリをスコープに導入する必要があります。`io`ライブラリは、標準ライブラリ (`std`として知られています) に存在します:

```
use std::io;
```

デフォルトでは、*prelude* に存在するいくつかの型のみ使えます。もし、使用したい型が prelude にない場合は、`use`文で明示的にその型をスコープに導入する必要があります。`std::io`ライブラリを使用することで、ユーザ入力を受け付ける能力などの実用的な機能の多くを使用することができます。

第 1 章で見た通り、`main`関数がプログラムへのエントリーポイント (スタート地点) になります:

```
fn main() {
```

`fn`構文が関数を新しく宣言し、かっこの `()`は引数がないことを示し、波括弧の`{`が関数本体のスタート地点になります。

また、第 1 章で学んだように、`println!`は、文字列を画面に表示するマクロになります:

```
println!("Guess the number!");

println!("Please input your guess.");
```

このコードは、このゲームが何かを出力し、ユーザに入力を求めています。

4.2.1　値を変数に保持する

次に、ユーザ入力を保持する場所を作りましょう。こんな感じに:

```
let mut guess = String::new();
```

さあ、プログラムが面白くなってきましたね。このたった 1 行でいろんなことが起きています。これが`let`文であることに注目してください。これを使用して**変数**を生成しています。こちらは、別の例です:

```
let foo = bar;
```

この行では、`foo`という名前の新しい変数を作成し、`bar`の値に束縛しています。Rust では、変数は標準で不変 (immutable) です。この概念について詳しくは、第 3 章の「変数と可変性」節で議論します。以下の例には、変数名の前に `mut`修飾子をつけて変数を可変にする方法が示されています:

```
let foo = 5; // immutable
let mut bar = 5; // mutable
```

注釈: //という記法は、行末まで続くコメントを記述します。コンパイラは、コメントを一切無視し、これについても第 3 章で詳しく議論します。

さあ、let mut guessが guessという名前の可変変数を導入するとわかりましたね。イコール記号 (=) の逆側には、変数 guessが束縛される値があります。この値は、String::new関数の呼び出し結果であり、この関数は、String型のオブジェクトを返します。String型は、標準ライブラリによって提供される文字列型で、サイズ可変、UTF-8 エンコードされたテキスト破片になります。

::new行にある::という記法は、newが String型の**関連付け関数**であることを表しています。関連関数とは、String型の特定のオブジェクトよりも型 (この場合は String) に対して実装された関数のことであり、**静的メソッド**と呼ばれる言語もあります。

この new関数は、新しく空の文字列を生成します。new関数は、いろんな型に見られます。なぜなら、何らかの新規値を生成する関数にとってありふれた名前だからです。

まとめると、let mut guess = String::new();という行は、現在、新たに空の Stringオブジェクトに束縛されている可変変数を作っているわけです。ふう！

プログラムの 1 行目で、use std::ioとして、標準ライブラリから入/出力機能を取り込んだことを思い出してください。今度は、io型の stdin関連関数を呼び出しましょう:

```
io::stdin().read_line(&mut guess)
    .expect("Failed to read line");
```

仮に、プログラムの冒頭で use std::ioとしていなければ、この関数呼び出しは、std::io::stdinと記述していたでしょう。この stdin関数は、std::io::Stdinオブジェクトを返し、この型は、ターミナルの標準入力へのハンドルを表す型になります。

その次のコード破片、.read_line(&mut guess)は、標準入力ハンドルの [read_line][read_line] メソッドを呼び出して、ユーザから入力を受け付けます。また、read_lineメソッドに対して、&mut guessという引数を一つ渡していますね.

«««< HEAD [read_line]: ../../std/io/struct.Stdin.html#method.read_line

read_lineメソッドの仕事は、ユーザが標準入力したものすべてを取り出し、文字列に格納することなので、格納する文字列を引数として取ります。この文字列引数は、可変である必要があります。メソッドがユーザ入力を追記して、文字列の中身を変えられるようにってことですね。

&という記号は、この引数が**参照**であることを表し、これのおかげで、データを複数回メモリにコピーせずとも、コードの複数箇所で同じデータにアクセスできるようになるわけです。参照は複雑な機能であり、とても安全かつ簡単に参照を使うことができることは、Rust の主要な利点の一つでもあります。そのような詳細を知らなくても、このプログラムを完成させることはできます: 現時点では、変数のように、参照も標準で不変であることを知っておけばいいでしょう。故に、&guessと書くので

はなく、&mut guessと書いて、可変にする必要があるのです。(第 4 章で参照について詳しく説明します)

4.2.2　Result 型で失敗の可能性を扱う

まだ、この行は終わりではありませんよ。ここまでに議論したのはテキストでは 1 行ですが、コードとしての論理行としては、まだ所詮最初の部分でしかないのです。2 番目の部分はこのメソッドです:

```
1 .expect("Failed to read line");
```

.foo()という記法で、メソッドを呼び出す時、改行と空白で長い行を分割するのは賢明なことです。今回の場合、こう書くこともできますよね:

```
1 io::stdin().read_line(&mut guess).expect("Failed to read line");
```

しかし、長い行は読みづらいものです。なので、分割しましょう: 2 回のメソッド呼び出しに、2 行です。さて、この行が何をしているのかについて議論しましょうか。

以前にも述べたように、read_lineメソッドは、渡された文字列にユーザが入力したものを入れ込むだけでなく、値も返します (今回は io::Resultです)。Rust には Resultと名のついた型が標準ライブラリにたくさんあります: 汎用の Resultの他、io::Resultなどのサブモジュール用に特化したものまで。

この Result型は、**列挙型**であり、普通、*enum*(イーナム) と呼ばれます。列挙型とは、固定された種類の値を持つ型のことであり、それらの値は、enum の**バリアント** (variant) と呼ばれます。enum については、第 6 章で詳しく解説します。

Result型に関しては、取りうる型の値 (バリアント) は Okか Errです。Ok列挙子は、処理が成功したことを表し、中に生成された値を保持します。Err列挙子は、処理が失敗したことを意味し、Errは、処理が失敗した過程や、理由などの情報を保有します。

これら Result型の目的は、エラー処理の情報をコード化することです。Result型の値も、他の型同様、メソッドが定義されています。io::Resultオブジェクトには、呼び出し可能な expectメソッドがあります。この io::Resultオブジェクトが Err値の場合、expectメソッドはプログラムをクラッシュさせ、引数として渡されたメッセージを表示します。read_lineメソッドが Errを返したら、恐らく根底にある OS によるエラーに起因するのでしょう。この io::Resultオブジェクトが Ok値の場合、expectメソッドは、Okバリアントが保持する返り値を取り出して、ただその値を返すので、これを使用することができるでしょう。今回の場合、その返り値とは、ユーザが標準入力に入力したデータのバイト数になります。

もし、expectメソッドを呼び出さなかったら、コンパイルは通るものの、警告が出るでしょう:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `std::result::Result` which must be used
(警告: 使用されなければならない結果が使用されていません)
```

```
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by default
```

コンパイラは、私たちが read_lineメソッドから返ってきた Result値を使用していないと警告してきており、これは、プログラムがエラーの可能性に対処していないことを示します。

警告を抑制する正しい手段は、実際にエラー対処コードを書くことですが、今は、問題が起きた時にプロラグムをクラッシュさせたいので、expectを使用できるわけです。エラーから復旧する方法については、第 9 章で学ぶでしょう。

4.2.3　println!マクロのプレースホルダーで値を出力する

閉じ波かっこを除けば、ここまでに追加されたコードのうち議論すべきものは、残り 1 行であり、それは以下の通りです:

```
1 println!("You guessed: {}", guess);
```

この行は、ユーザ入力を保存した文字列の中身を出力します。1 組の波括弧の{}は、プレースホルダーの役目を果たします: {}は値を所定の場所に保持する小さなカニのはさみと考えてください。波括弧を使って一つ以上の値を出力できます: 最初の波括弧の組は、フォーマット文字列の後に列挙された最初の値に対応し、2 組目は、2 つ目の値、とそんな感じで続いていきます。1 回の println!の呼び出しで複数値を出力するコードは、以下のような感じになります:

```
1 let x = 5;
2 let y = 10;
3
4 println!("x = {} and y = {}", x, y);
```

このコードは、x = 5 and y = 10と出力するでしょう.

4.2.4　最初の部分をテストする

数当てゲームの最初の部分をテストしてみましょう。cargo runでプログラムを走らせてください:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
     Running `target/debug/guessing_game`
Guess the number!           (数を当ててごらん)
Please input your guess.    (ほら、予想を入力して)
6
You guessed(次のように予想したよ): 6
```

ここまでで、ゲームの最初の部分は完成になります: キーボードからの入力を受け付け、出力できるようになりました。

4.3　秘密の数字を生成する

次に、ユーザが数当てに挑戦する秘密の数字を生成する必要があります。毎回この秘密の数字は、変わるべきです。ゲームが何回も楽しめるようにですね。ゲームが難しくなりすぎないように、1 から 100 までの乱数を使用しましょう。Rust の標準ライブラリには、乱数機能はまだ含まれていません。ですが、実は、Rust の開発チームが randクレートを用意してくれています。

4.3.1　クレートを使用して機能を追加する

クレートは Rust コードのパッケージであることを思い出してください。私たちがここまで作ってきたプロジェクトは、**バイナリークレート**であり、これは実行可能形式になります。randクレートは**ライブラリクレート**であり、他のプログラムで使用するためのコードが含まれています。

Cargo を使って外部クレートを使用すると、Cargo がとても輝きます。randを使ったコードを書ける前に、*Cargo.toml* ファイルを編集して、randクレートを依存ファイルとして取り込む必要があります。今このファイルを開いて、以下の行を Cargo が自動生成した [dependencies]セクションヘッダーの一番下に追記しましょう:

ファイル名: Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

Cargo.toml ファイルにおいて、ヘッダーに続くものは全て、他のセクションが始まるまで続くセクションの一部になります。[dependecies]セクションは、プロジェクトが依存する外部クレートと必要とするバージョンを記述するところです。今は、randクレートで、セマンティックバージョンには 0.3.14を指定します。Cargo はバージョンナンバー記述の標準規格であるセマンティックバージョニング (時に *SemVer* と呼ばれる) を理解します。0.3.14という数字は、実際には^0.3.14の省略記法で、これは、「バージョン 0.3.14 と互換性のある公開 API を持つ任意のバージョン」を意味します。

さて、コードは一切変えずに、リスト 2-2 のようにプロジェクトをビルドしましょう:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index` (レジストリを更新
        しています)
 Downloading rand v0.3.14                                              (rand v0.3.14をダ
     ウンロードしています)
 Downloading libc v0.2.14                                              (libc v0.2.14をダ
     ウンロードしています)
   Compiling libc v0.2.14                                              (libc v0.2.14をコ
       ンパイルしています)
   Compiling rand v0.3.14                                              (rand v0.3.14をコ
       ンパイルしています)
```

```
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)   (guessing_game v0
      .1.0をコンパイルしています)
   Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

リスト 2-2: rand クレートを依存として追加した後の `cargo build`コマンドの出力

もしかしたら、バージョンナンバーは違うかもしれません (でも、互換性はあります、SemVer のおかげでね！)。そして、行の出力順序も違うかもしれません。

今や、外部依存を持つようになったので、Cargo は**レジストリ** (registry、登録所) から最新バージョンを拾ってきます。**レジストリ**とは、Crates.io のデータのコピーです. Crates.io とは、Rust のエコシステムにいる人間が、他の人も使えるように自分のオープンソースの Rust プロジェクトを投稿する場所です。

レジストリの更新後、Cargo は `[dependencies]`セクションをチェックし、まだ取得していないクレートを全部ダウンロードします。今回の場合、`rand`しか依存ファイルには列挙していませんが、Cargo は `libc`のコピーも拾ってきます。`rand`クレートが `libc`に依存しているからですね。クレートのダウンロード完了後、コンパイラは依存ファイル、そして、依存が利用可能な状態でプロジェクトをコンパイルします。

何も変更せず即座に `cargo build`コマンドを走らせたら、`Finished`行を除いて何も出力されないでしょう。Cargo は、すでに全ての依存をダウンロードしてコンパイル済みであることも、あなたが *Cargo.toml* ファイルを弄ってないことも知っているからです。さらに、Cargo はプログラマがコードを変更していないことも検知するので、再度コンパイルすることもありません。することがないので、ただ単に終了します。

src/main.rs ファイルを開き、些細な変更をし、保存して再度ビルドを行えば、2 行だけ出力があるでしょう:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

この行は、Cargo が *src/main.rs* ファイルへの取るに足らない変更に対してビルドを更新していることを示しています。依存は変更していないので、Cargo は、すでにダウンロードしてコンパイルまで済ませてある依存を使用できると検知します。自分で書いたコードのみ再ビルドをかけるわけです。

4.3.1.1　*Cargo.lock* ファイルで再現可能なビルドを保証する

Cargo は、プログラマが自分のコードを更新するたびに同じ生成物を再構成することを保証してくれるメカニズムを備えています: Cargo は、プログラマが明示するまで、指定したバージョンの依存のみを使用します。例として、`rand`クレートの次週のバージョン `v0.3.15`が登場し、重要なバグ修正がなされているけれども、自分のコードを破壊してしまう互換性破壊があった場合はどうなるでしょう？

この問題に対する回答は、*Cargo.lock* ファイルであり、このファイルは、初めて `cargo build`コマンドを走らせた時に生成され、*guessing_game ディレクトリに存在しています。プロジェクトを初

めてビルドする際に、*Cargo* は判断基準 *(criteria)* に合致するよう全ての依存のバージョンを計算し、Cargo.lock ファイルに記述します。次にプロジェクトをビルドする際には、*Cargo* は Cargo.lock ファイルが存在することを確かめ、再度バージョンの計算の作業を行うのではなく、そこに指定されているバージョンを使用します。このことにより、自動的に再現可能なビルドを構成できるのです。つまり、明示的にアップグレードしない限り、プロジェクトが使用するバージョンは *0.3.14*に保たれるのです。Cargo.lock*ファイルのおかげでね。

4.3.1.2　*Cargo.lock* ファイルで再現可能なビルドを保証する

クレートを本当にアップグレードする必要が出てきたら、Cargo の別のコマンド (update) を使用しましょう。これは、*Cargo.lock* ファイルを無視して *Cargo.toml* ファイル内の全ての指定に合致する最新バージョンを計算します。それがうまくいったら、Cargo はそれらのバージョンを *Cargo.lock* ファイルに記述します。

しかし標準で Cargo は、0.3.0以上、0.4.0未満のバージョンのみを検索します。randクレートの新バージョンが 2 つリリースされていたら (0.3.15と 0.4.0だとします)、cargo updateコマンドを走らせた時に以下のようなメッセージを目の当たりにするでしょう:

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    (レジストリ`https://github.com/rust-lang/crates-io-index`を更新しています)
    Updating rand v0.3.14 -> v0.3.15
    (randクレートをv0.3.14 -> v0.3.15に更新しています)
```

この時点で、*Cargo.lock* ファイルに書かれている現在使用している randクレートのバージョンが、0.3.15になっていることにも気付くでしょう。

randのバージョン 0.4.0または、0.4.xシリーズのどれかを使用したかったら、代わりに *Cargo.toml* ファイルを以下のように更新しなければならないでしょう:

```
[dependencies]

rand = "0.4.0"
```

次回、cargo buildコマンドを走らせたら、Cargo は利用可能なクレートのレジストリを更新し、randクレートの必要条件を指定した新しいバージョンに再評価します。

まだ第 14 章で議論する Cargo とそのエコシステムについては述べたいことが山ほどありますが、とりあえずは、これで知っておくべきことは全てです。Cargo のおかげでライブラリはとても簡単に再利用ができるので、Rustacean（Rust ユーザのこと）は数多くのパッケージから構成された小規模のプロジェクトを書くことができるのです。

4.3.2　乱数を生成する

Cargo.toml に randクレートを追加したので、randクレートを使用開始しましょう。次のステップは、リスト 2-3 のように *src/main.rs* ファイルを更新することです:

ファイル名: src/main.rs

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);    //秘密の数字は次の通り: {}

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");                    //行の読み込みに失敗しました

    println!("You guessed: {}", guess);
}
```

リスト 2-3: 乱数を生成するコードの追加

まず、コンパイラに randクレートを外部依存として使用することを知らせる行を追加しています。これにより、use randを呼ぶのと同じ効果が得られるので、randクレートのものを rand:: という接頭辞をつけて呼び出せるようになりました。

次に、別の use行を追加しています: use rand::Rngですね。Rngトレイトは乱数生成器が実装するメソッドを定義していて、このトレイトがスコープにないと、メソッドを使用できないのです。トレイトについて詳しくは、第 10 章で解説します。

また、途中に 2 行追加もしています。rand::thread_rng関数は、これから使う特定の乱数生成器を返してくれます: この乱数生成器は、実行スレッドに固有で、OS により、シード値を与えられています。次に、この乱数生成器の gen_rangeメソッドを呼び出しています。このメソッドは、use rand::Rng文でスコープに導入した Rngトレイトで定義されています。gen_rangeメソッドは二つの数字を引数に取り、それらの間の乱数を生成してくれます。範囲は下限値を含み、上限値を含まないため、1と 101と指定しないと 1 から 100 の範囲の数字は得られません。

> 単純に使用すべきトレイトとクレートからどの関数とメソッドを呼び出すか知っているわけではないでしょう。クレートの使用方法は、各クレートのドキュメントにあります。Cargo の別の素晴しい機能は、cargo doc --openコマンドを走らせてローカルに存在する依存すべてのドキュメントをビルドし、ブラウザで閲覧できる機能です。例えば、randクレートの他の機能に興味があるなら、cargo doc --openコマンドを走らせて、左側のサイドバーから randをクリックしてください。

コードに追加した 2 行目は、秘密の数字を出力してくれます。これは、プログラムを開発中にはテストするのに役立ちますが、最終版からは削除する予定です。プログラムがスタートと同時に答えを出力しちゃったら、ゲームになりませんからね！

試しに何回かプログラムを走らせてみてください:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
     Running `target/debug/guessing_game`
Guess the number!                          (何回も出ているので、ここでは和訳は省略します
    )
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

毎回異なる乱数が出て、その数字はすべて 1 から 100 の範囲になるはずです。よくやりました！

4.4　予想と秘密の数字を比較する

今や、ユーザ入力と乱数生成ができるようになったので、比較することができますね。このステップはリスト 2-4 に示されています。このコードは現状ではコンパイルできないので、説明することに注意してください。

ファイル名: src/main.rs

```
1  extern crate rand;
2
3  use std::io;
4  use std::cmp::Ordering;
5  use rand::Rng;
6
7  fn main() {
8      // ---snip---
9
10     println!("You guessed: {}", guess);
11
12     match guess.cmp(&secret_number) {
13         Ordering::Less => println!("Too small!"),       //小さすぎ！
14         Ordering::Greater => println!("Too big!"),      //大きすぎ！
15         Ordering::Equal => println!("You win!"),        //やったね！
16     }
17 }
```

リスト 2-4: 2 値比較の可能性のある返り値を処理する

最初の新しい点は、別の use文です。これで、std::cmp::Orderingという型を標準ライブラリからスコープに導入しています。Resultと同じく Orderingも enum です。ただ、Orderingの列挙子は、Less、Greaterそして、Equalです。これらは、2 値比較した時に発生しうる 3 種類の結果です。

それから、一番下に 5 行追加して Ordering型を使用しています。

```
match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
```

cmpメソッドは、2 値を比較し、比較できるものに対してならなんに対しても呼び出せます。このメソッドは、比較したいものへの参照を取ります: ここでは、guess変数と secret_number変数を比較しています。それからこのメソッドは use文でスコープに導入した Ordering列挙型の値を返します。match式を使用して、guess変数と secret_numberを cmpに渡して返ってきた Orderingの列挙子に基づき、次の動作を決定しています。

match式は、複数の**アーム** (腕) からできています。一つのアームは、パターンとそのパターンにmatch式の冒頭で与えた値がマッチした時に走るコードから構成されています。Rust は、matchに与えられた値を取り、各アームのパターンを順番に照合していきます。match式とパターンは、コードを書く際に出 (で) 会 (くわ) す様々なシチュエーションを表現させてくれ、すべてのシチュエーションに対処していることを保証するのを手助けしてくれる Rust の強力な機能です。これらの機能は、それぞれ、第 6 章と第 18 章で詳しく解説することにします。

Let's walk through an example of what would happen with the match expression used here. Say that the user has guessed 50 and the randomly generated secret number this time is 38. When the code compares 50 to 38, the cmp method will return Ordering::Greater, because 50 is greater than 38. The match expression gets the Ordering::Greater value and starts checking each arm's pattern. It looks at the first arm's pattern, Ordering::Less, and sees that the value Ordering::Greater does not match Ordering::Less, so it ignores the code in that arm and moves to the next arm. The next arm's pattern, Ordering::Greater, *does* match Ordering::Greater ! The associated code in that arm will execute and print Too big! to the screen. The match expression ends because it has no need to look at the last arm in this scenario.

ここで使われている match式でどんなことが起こるかの例をじっくり観察してみましょう！ 例えば、ユーザは 50 と予想し、ランダム生成された秘密の数字は今回、38 だったとしましょう。コードが 50 と 38 を比較すると、cmpメソッドは Ordering::Greaterを返します。50 は 38 よりも大きいからですね。match式に、Ordering::Greaterが与えられ、各アームのパターンを吟味し始めます。まず、最初のアームのパターンと照合します (Ordering::Lessですね)。しかし、値の Ordering::Greaterと Ordering::Lessはマッチしないため、このアームのコードは無視され、次のアームに移ります。次のアームのパターン、Ordering::Greaterは**見事に** Ordering::Greaterとマッチします！ このアームに

紐づけられたコードが実行され、画面に Too big!が表示されます。これで match式の実行は終わりになります。この筋書きでは、最後のアームと照合する必要はもうないからですね。

ところが、リスト 2-4 のコードは、まだコンパイルが通りません。試してみましょう:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types          (型が合いません)
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^ expected struct `std::string::String`, found
       integral variable
   |                                     (構造体`std::string::String`を予期したけど、整
       数型変数が見つかりました)
   |
   = note: expected type `&std::string::String`
   = note:    found type `&{integer}`

error: aborting due to previous error   (先のエラーのため、処理を中断します)
Could not compile `guessing_game`.      (`guessing_game`をコンパイルできませんでした)
```

このエラーの核は、**型の不一致**があると言っています。Rust は、強い静的型システムを持っています。しかし、型推論にも対応しています。let guess = String::new()と書いた時、コンパイラは、guessが String型であるはずと推論してくれ、その型を明示させられることはありませんでした。一方で、secret_number変数は、数値型です。1 から 100 を表すことができる数値型はいくつかあります: i32は 32 ビットの数字; u32は 32 ビットの非負数字; i64は 64 ビットの数字; などです。Rust での標準は、i32型であり、型情報をどこかに追加して、コンパイラに異なる数値型だと推論させない限り、secret_numberの型はこれになります。エラーの原因は、Rust では、文字列と数値型を比較できないことです。

究極的には、プログラムが入力として読み込む String型を現実の数値型に変換し、予想と数値として比較できるようにしたいわけです。これは、以下の 2 行を main関数の本体に追記することでできます:

ファイル名: src/main.rs

```
1  // --snip--
2
3      let mut guess = String::new();
4
5      io::stdin().read_line(&mut guess)
6          .expect("Failed to read line");
7
8      let guess: u32 = guess.trim().parse()
9          .expect("Please type a number!");                //数値を入力してください！
10
11     println!("You guessed: {}", guess);
12
13     match guess.cmp(&secret_number) {
14         Ordering::Less => println!("Too small!"),
```

```
15          Ordering::Greater => println!("Too big!"),
16          Ordering::Equal => println!("You win!"),
17      }
18  }
```

その 2 行とは:

```
1  let guess: u32 = guess.trim().parse()
2      .expect("Please type a number!");
```

guessという名前の変数を生成しています。あれ、でも待って。もうプログラムには guessという名前の変数がありませんでしたっけ？ 確かにありますが、Rust では、新しい値で guessの値を**覆い隠す** (shadow) ことが許されているのです。この機能は、今回のような、値を別の型に変換したいシチュエーションでよく使われます。シャドーイング (shadowing) のおかげで別々の変数を 2 つ作らされることなく、guessという変数名を再利用することができるのです。guess_strと guessみたいなね (シャドーイングについては、第 3 章でもっと掘り下げます)。

guessを guess.trim().parse()という式に束縛しています。この式中の guessは、入力が入った Stringの元々の guessを指しています。Stringオブジェクトの trimメソッドは、両端の空白をすべて除去します。u32型は、数字しか含むことができませんが、ユーザは、read_lineの処理を終えるためにエンターを押さなければなりません。ユーザがエンターを押したら、改行文字が文字列に追加されます。具体例として、ユーザが 5 を入力して、エンターを押せば、guessは次のようになります: 5\n。この\nが「改行」、つまりエンターキーを押した結果を表しているわけです。trimメソッドは、\nを削除するので、ただの 5になります。

文字列の parseメソッドは、文字列をパースして何らかの数値にします。このメソッドは、いろんな数値型をパースできるので、let guess: u32としてコンパイラに私たちが求めている型をズバリ示唆する必要があるのです。guessの後のコロン (:) がコンパイラに変数の型を注釈する合図になります。Rust には、組み込みの数値型がいくつかあります; ここの u32型は、32 ビットの非負整数です。u32型は小さな非負整数のデフォルトの選択肢として丁度良いです。他の数値型については、第 3 章で学ぶでしょう。付け加えると、このサンプルプログラムの u32という注釈と secret_number変数との比較は、secret_number変数も u32型であるとコンパイラが推論することを意味します。従って、今では比較が同じ型の 2 つの値で行われることになるわけです！

parseメソッドの呼び出しは、エラーになりやすいです。例としては、文字列が A%を含んでいたら、数値に変換できるわけがありません。失敗する可能性があるので、parseメソッドは、Result型を返すわけです。ちょうど、(「Result 型で失敗する可能性に対処する」節で先ほど議論した)read_lineメソッドのようにというわけですね。今回も、expectメソッドを使用して Result型を同じように扱います。この Resultを expectメソッドを再度使用して、同じように扱います。もし、文字列から数値を生成できなかったために、parseメソッドが Result型の Err値を返したら、expectメソッドの呼び出しは、ゲームをクラッシュさせ、与えたメッセージを表示します。もし、parseメソッドが文字列の数値への変換に成功したら、Result型の Ok値を返し、expectメソッドは、Ok値から必要な数値を返してくれます。

さあ、プログラムを走らせましょう！

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

いいですね！ 予想の前にスペースを追加したにもかかわらず、プログラムはちゃんとユーザが 76 と予想したことを導き出しました。プログラムを何回か走らせて、異なる入力の色々な振る舞いを確認してください: つまり、数字を正しく言い当てたり、大きすぎる値を予想したり、低すぎる数字を入力したりということです。

ここまでで大方ゲームはうまく動くようになりましたが、まだユーザは 1 回しか予想できません。ループを追加して、その部分を変更しましょう！

4.5　ループで複数回の予想を可能にする

loopキーワードは、無限ループを作り出します。これを追加して、ユーザが何回も予想できるようにしましょう:

ファイル名: src/main.rs

```
// --snip--

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
    }
}
```

見てわかる通り、予想入力部分以降をループに入れ込みました。ループ内の行にインデントを追加するのを忘れないようにして、またプログラムを走らせてみましょう。新たな問題が発生したことに気付いてください。プログラムが教えた通りに動作しているからですね: 永遠に予想入力を求めるわけです！ これでは、ユーザが終了できないようです！

ユーザは、ctrl-c というキーボードショートカットを使って、いつでもプログラムを強制終了させられます。しかし、「予想を秘密の数字と比較する」節の parseメソッドに関する議論で触れたように、この貪欲なモンスターを回避する別の方法があります: ユーザが数字以外の答えを入力すれば、プログラムはクラッシュするのです。ユーザは、その利点を活かして、終了することができます。以下のようにね:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind: InvalidDigit
    }', src/libcore/result.rs:785
(スレッド'main'は'数字を入力してください！: ParseIntError { kind: InvalidDigit }', src/
    libcore/result.rs:785でパニックしました)
note: Run with `RUST_BACKTRACE=1` for a backtrace.
(注釈: `RUST_BACKTRACE=1`で走らせるとバックトレースを見れます)
error: Process didn't exit successfully: `target/debug/guess` (exit code: 101)
(エラー: プロセスは予期なく終了しました)
```

quitと入力すれば、実際にゲームを終了できるわけですが、別に他の数字以外の入力でもそうなります。しかしながら、これは最低限度と言えるでしょう。正しい数字が予想されたら、自動的にゲームが停止してほしいわけです。

4.5.1　正しい予想をした後に終了する

break文を追加して、ユーザが勝った時にゲームが終了するようにプログラムしましょう:

ファイル名: src/main.rs

```
// --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
```

```
7                   println!("You win!");
8                   break;
9               }
10          }
11      }
12  }
```

break文の 1 行を You win!の後に追記することで、ユーザが秘密の数字を正確に予想した時に、プログラムはループを抜けるようになりました。ついでに、ループを抜けることは、プログラムを終了することを意味します。ループが main関数の最後の部分だからですね。

4.5.2　不正な入力を処理する

さらにゲームの振る舞いを改善するために、ユーザが数値以外を入力した時にプログラムをクラッシュさせるのではなく、非数値を無視してユーザが数当てを続けられるようにしましょう！ これは、guessが String型から u32型に変換される行を改変することで達成できます:

```
1  let guess: u32 = match guess.trim().parse() {
2      Ok(num) => num,
3      Err(_) => continue,
4  };
```

expectメソッドの呼び出しから match式に切り替えることは、エラーでクラッシュする動作からエラー処理を行う処理へ変更する一般的な手段になります。parseメソッドは、Result型を返し、Result は Okか Errの値を取りうる enum であることを思い出してください。ここでは match式を使っています。cmpメソッドの Orderingという結果のような感じですね。

parseメソッドは、文字列から数値への変換に成功したら、結果の数値を保持する Ok値を返します。この Ok値は、最初のアームのパターンにマッチし、この match式は parseメソッドが生成し、Ok値に格納した numの値を返すだけです。その数値が最終的に生成した新しい guess変数の欲しい場所に含まれます。

parseメソッドは、文字列から数値への変換に失敗したら、エラーに関する情報を多く含む Err値を返します。この Err値は、最初の matchアームの Ok(num)というパターンにはマッチしないものの、2 番目のアームの Err(_)というパターンにはマッチするわけです。この_は、包括値です; この例では、保持している情報がどんなものでもいいから全ての Err値にマッチさせたいと宣言しています。従って、プログラムは 2 番目のアームのコードを実行し (continueですね)、これは、loopの次の段階に移り、再度予想入力を求めることを意味します。故に実質的には、プログラムは parseメソッドが遭遇しうる全てのエラーを無視するようになります！

さて、プログラムの全てがうまく予想通りに動くはずです。試しましょう:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
```

```
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

素晴らしい！ 最後にひとつまみ変更を加えて、数当てゲームを完了にしましょう。プログラムが未だに秘密の数字を出力していることを思い出してください。テスト中はうまく動くけど、ゲームを台無しにしてしまいます。秘密の数字を出力する `println!`を削除しましょう。リスト 2-5 が成果物のコードです:

ファイル名: src/main.rs

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
```

```
31                  println!("You win!");
32                  break;
33              }
34          }
35      }
36  }
```

リスト 2-5: 数当てゲームの完全なコード

4.6　まとめ

ここまでで、数当てゲームの作成に成功しました！ おめでとうございます！

このプロジェクトは、たくさんの新しい Rust の概念に触れる実践的な方法でした: `let`文、`match`式、メソッド、関連関数、外部クレートの使用などなど。以降の数章で、これらの概念についてより深く学ぶことになるでしょう。第 3 章では、ほとんどのプログラミング言語が持っている、変数、データ型、関数などの概念について解説し、それらの Rust での使用方法について示します。第 4 章では、所有権について見ます。これにより、Rust は他の言語とかけ離れた存在になっています。第 5 章では、構造体とメソッド記法について議論し、第 6 章では enum の動作法を説明します。

第 5 章

一般的なプログラミングの概念

この章では、ほとんど全てのプログラミング言語で見られる概念を解説し、それらが Rust において、どう動作するかを見ていきます。多くのプログラミング言語は、その核心において、いろいろなものを共有しています。この章で提示する概念は、全て Rust に固有のものではありませんが、Rust の文脈で議論し、これらの概念を使用することにまつわる仕様を説明します。

具体的には、変数、基本的な型、関数、コメント、そしてフロー制御について学びます。これらの基礎は全ての Rust プログラムに存在するものであり、それらを早期に学ぶことは強力な基礎を築くことになるでしょう。

5.0.1 キーワード

Rust 言語にも他の言語同様、キーワードが存在し、これらは言語だけが使用できるようになっています。これらの単語は、変数や関数名には使えないことを弁えておいてください。ほとんどのキーワードは、特別な意味を持っており、自らの Rust プログラムにおいて、様々な作業をこなすために使用することができます; いくつかは、紐付けられた機能がないものの、将来 Rust に追加されるかもしれない機能用に予約されています。キーワードの一覧は、付録 A で確認できます。

5.1 変数と可変性

第 2 章で触れた通り、変数は標準で不変になります。これは、Rust が提供する安全性や簡潔な並列プログラミングの利点を享受する形でコードを書くために与えられた一押しです。ところが、まだ変数を可変にするという選択肢も残されています。不変性を好むようコンパイラが推奨する手段と理由および、それと違う道を選びたくなる理由を見ていきましょう。

変数が不変であると、値が一旦名前に束縛されたら、その値を変えることができません。これを具

体化するために、*projects* ディレクトリに `cargo new --bin variables`コマンドを使って、*variables* という名前のプロジェクトを生成しましょう。

それから、新規作成した *variables* ディレクトリで、*src/main.rs* ファイルを開き、まだコンパイルできないけれど、その中身を以下のコードに置き換えましょう:

ファイル名: src/main.rs

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);     // xの値は{}です
    x = 6;
    println!("The value of x is: {}", x);
}
```

これを保存し、`cargo run`コマンドでプログラムを走らせてください。次の出力に示されているようなエラーメッセージを受け取るはずです:

```
error[E0384]: cannot assgin twice immutable variable `x`
              (不変変数`x`に2回代入できません)
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         - first assignment to `x`
  |         (`x`への最初の代入)
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```

この例では、コンパイラがプログラムに潜むエラーを見つけ出す手助けをしてくれることが示されています。コンパイルエラーは、イライラすることもあるものですが、まだプログラムにしてほしいことを安全に行えていないだけということなのです; エラーが出るからといって、あなたがいいプログラマではないという意味では**ありません**！ 経験豊富な Rust 市民でも、コンパイルエラーを出すことはあります。

このエラーは、エラーの原因が不変変数xに2回代入できないであると示しています。不変な xという変数に第 2 段階の値を代入しようとしたからです。

以前に不変と指定された値を変えようとした時に、コンパイルエラーが出るのは重要なことです。なぜなら、この状況はまさしく、バグに繋がるからです。コードのある部分は、値が変わることはないという前提のもとに処理を行い、別の部分がその値を変更していたら、最初の部分が目論見通りに動いていない可能性があるのです。このようなバグの発生は、事実 (脚注:実際にプログラムを走らせた結果のことと思われる) の後には追いかけづらいものです。特に第 2 のコード破片が、値を**時々**しか変えない場合尚更です。

Rust では、値が不変であると宣言したら、本当に変わらないことをコンパイラが担保してくれます。つまり、コードを読み書きする際に、どこでどうやって値が変化しているかを追いかける必要がなくなります。故にコードを通して推測することが簡単になるのです。

しかし、可変性は時として非常に有益なこともあります。変数は、標準でのみ、不変です。つまり、

第 2 章のように変数名の前に mutキーワードを付けることで、可変にできるわけです。この値が変化できるようにするとともに、mutにより、未来の読者に対してコードの別の部分がこの変数の値を変える可能性を示すことで、その意図を汲ませることができるのです。

例として、*src/main.rs* ファイルを以下のように書き換えてください:

ファイル名: src/main.rs

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

今、このプログラムを走らせると、以下のような出力が得られます:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
     Running `target/debug/variables`
The value of x is: 5   (xの値は5です)
The value of x is: 6
```

mutキーワードを使われると、xが束縛している値を 5から 6に変更できます。変数を可変にする方が、不変変数だけがあるよりも書きやすくなるので、変数を可変にしたくなることもあるでしょう。

考えるべきトレードオフはバグの阻止以外にも、いくつかあります。例えば、大きなデータ構造を使う場合などです。インスタンスを可変にして変更できるようにする方が、いちいちインスタンスをコピーして新しくメモリ割り当てされたインスタンスを返すよりも速くなります。小規模なデータ構造なら、新規インスタンスを生成して、もっと関数型っぽいコードを書く方が通して考えやすくなるため、低パフォーマンスは、その簡潔性を得るのに足りうるペナルティになるかもしれません。

5.1.1　変数と定数 (constants) の違い

変数の値を変更できないようにするといえば、他の多くの言語も持っている別のプログラミング概念を思い浮かべるかもしれません: **定数**です。不変変数のように、定数は名前に紐付き、変更することが叶わない値のことですが、定数と変数の間にはいくつかの違いがあります。

まず、定数には mutキーワードは使えません: 定数は標準で不変であるだけでなく、常に不変なのです。

定数は letキーワードの代わりに、constキーワードで宣言し、値の型は**必ず**注釈しなければなりません。型と型注釈については次のセクション、「データ型」で解説する予定なので、その詳細については気にする必要はありません。ただ単に型は常に注釈しなければならないのだと思っていてください。

定数はどんなスコープでも定義できます。グローバルスコープも含めてね。なので、いろんなところで使用される可能性のある値を定義するのに役に立ちます。

最後の違いは、定数は定数式にしかセットすることが叶わないことです。関数呼び出し結果や、実行時に評価される値にはセットできません。

定数の名前が MAX_POINTSで、値が 100,000 にセットされた定数定義の例をご覧ください。(Rust の定数の命名規則は、全て大文字でアンダースコアで単語区切りすることです):

```
const MAX_POINTS: u32 = 100_000;
```

定数は、プログラムが走る期間、定義されたスコープ内でずっと有効です。従って、プログラムのいろんなところで使用される可能性のあるアプリケーション空間の値を定義するのに有益な選択肢になります。例えば、ゲームでプレイヤーが取得可能なポイントの最高値や、光速度などですね。

プログラム中で使用されるハードコードされた値に対して、定数として名前付けすることは、コードの将来的な管理者にとって値の意味を汲むのに役に立ちます。将来、ハードコードされた値を変える必要が出た時に、たった 1 箇所を変更するだけで済むようにもしてくれます。

5.1.2　(変数の) 多重定義 (shadowing)

第 2 章の「秘密の数字と予想を比較する」節で見たように、前に定義した変数と同じ名前の変数を新しく宣言でき、新しい変数は、前の変数を上書き (shadow) します。Rust 市民はこれを最初の変数は、2 番目の変数に**上書き**されたと言い、この変数を使用した際に、2 番目の変数の値が現れるということです。以下のようにして、同じ変数名を用いて変数を上書きし、letキーワードの使用を繰り返します:

ファイル名: src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    let x = x * 2;

    println!("The value of x is: {}", x);
}
```

このプログラムはまず、xを 5という値に束縛します。それから let x =を繰り返すことで xを上書きし、元の値に 1を加えることになるので、xの値は、6になります。3 番目の letも xを上書きし、以前の値に 2をかけることになるので、xの最終的な値は 12になります。このプログラムを走らせたら、以下のように出力するでしょう:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running `target/debug/variables`
The value of x is: 12
```

シャドーイングは、変数を mutにするのとは違います。なぜなら、letキーワードを使わずに、誤ってこの変数に再代入を試みようものなら、コンパイルエラーが出るからです。letを使うことで、値にちょっとした加工は加えられますが、その加工が終わったら、変数は不変になるわけです。

mutと上書きのもう一つの違いは、再度 letキーワードを使用したら、実効的には新しい変数を生成していることになるので、値の型を変えつつ、同じ変数名を使いまわせることです。例えば、プログラムがユーザに何らかのテキストに対して空白文字を入力することで何個分のスペースを表示したいかを尋ねるとします。ただ、実際にはこの入力を数値として保持したいとしましょう:

```
let spaces = "   ";
let spaces = spaces.len();
```

この文法要素は、容認されます。というのも、最初の spaces変数は文字列型であり、2 番目の spaces変数は、たまたま最初の変数と同じ名前になったまっさらな変数のわけですが、数値型になるからです。故に、上書きのおかげで、異なる名前を思いつく必要がなくなるわけです。spaces_strと spaces_numなどですね; 代わりに、よりシンプルな spacesという名前を再利用できるわけです。一方で、この場合に mutを使おうとすると、以下に示した通りですが、コンパイルエラーになるわけです:

```
let mut spaces = "   ";
spaces = spaces.len();
```

変数の型を可変にすることは許されていないと言われているわけです:

```
error[E0308]: mismatched types          (型が合いません)
 --> src/main.rs:3:14
  |
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected &str, found usize
  |                           (&str型を予期しましたが、usizeが見つかりました)
  |
  = note: expected type `&str`
             found type `usize`
```

さあ、変数が動作する方法を見てきたので、今度は変数が取りうるデータ型について見ていきましょう。

5.2　データ型

Rust における値は全て、何らかの**型**になり、コンパイラがどんなデータが指定されているか知れるので、そのデータの取り扱い方も把握できるというわけです。2 種のデータ型のサブセットを見ましょう: スカラー型と複合型です。

Rust は**静的型付け**言語であることを弁えておいてください。つまり、コンパイル時に全ての変数の型が判明している必要があるということです。コンパイラは通常、値と使用方法に基づいて、使用したい型を推論してくれます。複数の型が推論される可能性がある場合、そう例えば、第 2 章の「秘密

の数字と予想を比較する」節で parseメソッドを使って String型を数値型に変換した時のような場合には、型注釈をつけなければいけません。以下のようにね:

```
let guess: u32 = "42".parse().expect("Not a number!");    // 数字ではありません！
```

ここで型注釈を付けなければ、コンパイラは以下のエラーを表示し、これは可能性のある型のうち、どの型を使用したいのかを知るのに、コンパイラがプログラマからもっと情報を得る必要があることを意味します:

```
error[E0282]: type annotations needed
              (型注釈が必要です)
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^ cannot infer type for `_`
  |               (`_`の型が推論できません)
  |
  = note: type annotations or generic parameter binding required
    (注釈: 型注釈、またはジェネリクス引数束縛が必要です)
```

他のデータ型についても、様々な型注釈を目撃することになるでしょう。

5.2.1　スカラー型

スカラー型は、単独の値を表します。Rust には主に 4 つのスカラー型があります: 整数、浮動小数点数、論理値、最後に文字です。他のプログラミング言語でも、これらの型を見かけたことはあるでしょう。Rust での動作方法に飛び込みましょう。

5.2.1.1　整数型

整数とは、小数部分のない数値のことです。第 2 章で一つの整数型を使用しました。u32型です。この型定義は、紐付けられる値が、符号なし整数 (符号付き整数は uではなく、iで始まります) になり、これは、32 ビット分のサイズを取ります。表 3-1 は、Rust の組み込み整数型を表示しています。符号付きと符号なし欄の各バリアント (例: i16) を使用して、整数値の型を宣言することができます。

表 3-1: Rust の整数型

大きさ	符号付き	符号なし
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
arch	isize	usize

各バリアントは、符号付きか符号なしかを選べ、明示的なサイズを持ちます。**符号付きと符号なし**は、数値が正負を持つかどうかを示します。つまり、数値が符号を持つ必要があるかどうか (符号付き)、または、絶対に正数にしかならず符号なしで表現できるかどうか (符号なし) です。これは、数値を紙に書き下すのと似ています: 符号が問題になるなら、数値はプラス記号、またはマイナス記号とともに表示されます; しかしながら、その数値が正数であると仮定することが安全なら、符号なしで表示できるわけです。符号付き数値は、2 の補数表現で保持されます (これが何なのか確信を持てないのであれば、ネットで検索することができます。まあ要するに、この解説は、この本の範疇外というわけです)。

各符号付きバリアントは、-(2n - 1) から 2n - 1 - 1 までの数値を保持でき、ここで n はこのバリアントが使用するビット数です。以上から、`i8`型は-(27) から 27 - 1 まで、つまり、-128 から 127 までを保持できます。符号なしバリアントは、0 から 2n - 1 までを保持できるので、`u8`型は、0 から 28 - 1 までの値、つまり、0 から 255 までを保持できることになります。

加えて、`isize`と `usize`型は、プログラムが動作しているコンピュータの種類に依存します: 64 ビットアーキテクチャなら、64 ビットですし、32 ビットアーキテクチャなら、32 ビットになります。

整数リテラル (脚注: リテラルとは、見たまんまの値ということ) は、表 3-2 に示すどの形態でも記述することができます。バイトリテラルを除く数値リテラルは全て、型接尾辞を付加すること (例えば、`57u8`) と`_`を見た目の区切り記号 (例えば、`1_000`) にできます。

表 3-2: Rust の整数リテラル

数値リテラル	例
10 進数	`98_222`
16 進数	`0xff`
8 進数	`0o77`
2 進数	`0b1111_0000`
バイト (`u8`だけ)	`b'A'`

では、どの整数型を使うべきかはどう把握すればいいのでしょうか？ もし確信が持てないのならば、Rust の基準型は一般的にいい選択肢になります。整数型の基準は `i32`型です: 64 ビットシステム上でも、この型が普通最速になります。`isize`と `usize`を使う主な状況は、何らかのコレクションにアクセスすることです。

5.2.1.2　浮動小数点型

Rust にはさらに、**浮動小数点数**に対しても、2 種類の基本型があり、浮動小数点数とは 10 進小数のことです。Rust の浮動小数点型は、`f32`と `f64`で、それぞれ 32 ビットと 64 ビットサイズです。基準型は `f64`です。なぜなら、現代の CPU では、`f32`とほぼ同スピードにもかかわらず、より精度が高くなるからです。

実際に動作している浮動小数点数の例をご覧ください:

ファイル名: src/main.rs

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

浮動小数点数は、IEEE-754 規格に従って表現されています。`f32`が単精度浮動小数点数、`f64`が倍精度浮動小数点数です。

5.2.1.3　数値演算

Rust にも期待されうる標準的な数学演算が全数値型に対して用意されています: 足し算、引き算、掛け算、割り算、余りです。以下の例では、`let`文での各演算の使用方法をご覧になれます:

ファイル名: src/main.rs

```
fn main() {
    // 足し算
    let sum = 5 + 10;

    // 引き算
    let difference = 95.5 - 4.3;

    // 掛け算
    let product = 4 * 30;

    // 割り算
    let quotient = 56.7 / 32.2;

    // 余り
    let remainder = 43 % 5;
}
```

これらの文の各式は、数学演算子を使用しており、一つの値に評価され、変数に束縛されます。付録 B に Rust で使える演算子の一覧が載っています。

5.2.1.4　論理値型

他の多くの言語同様、Rust の論理値型も取りうる値は二つしかありません: `true`と `false`です。Rust の論理値型は、`bool`と指定されます。例です:

ファイル名: src/main.rs

```
fn main() {
    let t = true;

    let f: bool = false; // 明示的型注釈付きで
}
```

論理値を消費する主な手段は、条件式です。例えば、if式などですね。if式の Rust での動作方法については、「フロー制御」節で解説します。

5.2.1.5　文字型

ここまで、数値型のみ扱ってきましたが、Rust には文字も用意されています。Rust の char型は、言語の最も基本的なアルファベット型であり、以下のコードでその使用方法の一例を見ることができます。(charは、ダブルクォーテーションマークを使用する文字列に対して、シングルクォートで指定されることに注意してください。)

ファイル名: src/main.rs

```
fn main() {
   let c = 'z';
   let z = '  ';
   let heart_eyed_cat = ' ';     //ハート目の猫
}
```

Rust の char型は、ユニコードのスカラー値を表します。これはつまり、アスキーよりもずっとたくさんのものを表せるということです。アクセント文字; 中国語、日本語、韓国語表意文字 (脚注: 漢字のことだと思われる); 絵文字; ゼロ幅スペースは、全て Rust では、有効な char型になります。ユニコードスカラー値は、U+0000から U+D7FFまでと U+E0000から U+10FFFFまでの範囲になります。ところが、「文字」は実はユニコードの概念ではないので、文字とは何かという人間としての直観は、Rust における char型が何かとは合致しない可能性があります。この話題については、第 8 章の「文字列」で詳しく議論しましょう。

5.2.2　複合型

複合型により、複数の値を一つの型にまとめることができます。Rust には、2 種類の基本的な複合型があります: タプルと配列です。

5.2.2.1　タプル型

タプルは、複数の型の何らかの値を一つの複合型にまとめ上げる一般的な手段です。

タプルは、丸かっこの中にカンマ区切りの値リストを書くことで生成します。タプルの各位置には型が紐付けられ、タプルの値は全てが同じ型である必要はありません。今回の例では、型注釈をあえて追加してみました:

ファイル名: src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

変数 tupは、タプル全体に束縛されています。なぜなら、タプルは、一つの複合要素と考えられる

からです。タプルから個々の値を取り出すには、パターンマッチングを使用して分解することができます。以下のように:

ファイル名: src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

このプログラムは、まずタプルを生成し、それを変数 tupに束縛しています。それから letx文とパターンを使って tup変数の中身を 3 つの個別の変数 (x、y、zですね) に変換しています。この過程は、**分解**と呼ばれます。単独のタプルを破壊して三分割しているからです。最後に、プログラムは y変数の値を出力し、6.4と表示されます。

パターンマッチングを通しての分解の他にも、アクセスしたい値の番号をピリオド (.) に続けて書くことで、タプルの要素に直接アクセスすることもできます。例です:

ファイル名: src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

このプログラムは、新しいタプル xを作成し、添え字アクセスで各要素に対して新しい変数も作成しています。多くのプログラミング言語同様、タプルの最初の添え字は 0 です。

5.2.2.2　配列型

配列によっても、複数の値のコレクションを得ることができます。タプルと異なり、配列の全要素は、同じ型でなければなりません。Rust の配列は、他の言語と異なっています。Rust の配列は、固定長なのです: 一度宣言されたら、サイズを伸ばすことも縮めることもできません。

Rust では、配列に入れる要素は、角かっこ内にカンマ区切りリストとして記述します:

ファイル名: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

配列は、ヒープよりもスタック (スタックとヒープについては第 4 章で詳 (つまび) らかに議論します) にデータのメモリを確保したい時、または、常に固定長の要素があることを確認したい時に有効です。ただ、配列は、ベクタ型ほど柔軟ではありません。ベクタも、標準ライブラリによって提供されている似たようなコレクション型で、こちらは、サイズを伸縮させることができます。配列とベクタ型、どちらを使うべきか確信が持てない時は、おそらくベクタ型を使うべきです。第 8 章でベクタについて詳細に議論します。

ベクタ型よりも配列を使いたくなるかもしれない例は、1 年の月の名前を扱うプログラムです。そのようなプログラムで、月を追加したり削除したりすることはほぼ稀なので、配列を使用できます。常に 12 個要素があることもわかってますしね:

```
let months = ["January", "February", "March", "April", "May", "June", "July",
              "August", "September", "October", "November", "December"];
```

■配列の要素にアクセスする　配列は、スタック上に確保される一塊のメモリです。添え字によって、配列の要素にアクセスすることができます。こんな感じ:

ファイル名: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

この例では、`first`という名前の変数には `1`という値が格納されます。配列の `[0]`番目にある値が、それだからですね。`second`という名前の変数には、配列の `[1]`番目の値 `2`が格納されます。

■配列要素への無効なアクセス　配列の終端を越えて要素にアクセスしようとしたら、どうなるでしょうか？ 先ほどの例を以下のように変えたとすると、コンパイルは通りますが、実行するとエラーで終了します:

ファイル名: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

    println!("The value of element is: {}", element);   // 要素の値は{}です
}
```

このコードを `cargo run`で走らせると、以下のような結果になります:

```
$ cargo run
   Compiling arrays v0.1.0 (file:///projects/arrays)
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running `target/debug/arrays`
thread '<main>' panicked at 'index out of bounds: the len is 5 but the index is
 10', src/main.rs:6
スレッド'<main>'は'範囲外アクセス: 長さは5ですが、添え字は10でした', src/main.rs:6
でパニックしました
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

コンパイルでは何もエラーが出なかったものの、プログラムは**実行時**エラーに陥り、正常終了しませんでした。要素に添え字アクセスを試みると、言語は、指定されたその添え字が配列長よりも小さいかを確認してくれます。添え字が配列長よりも大きければ、言語は**パニック**します。パニックとは、プログラムがエラーで終了したことを表す Rust 用語です。

これは、実際に稼働している Rust の安全機構の最初の例になります。低レベル言語の多くでは、この種のチェックは行われないため、間違った添え字を与えると、無効なメモリにアクセスできてしまいます。Rust では、メモリアクセスを許可し、処理を継続する代わりに即座にプログラムを終了することで、この種のエラーからプログラマを保護しています。Rust のエラー処理については、第 9 章でもっと議論します。

5.3　関数

関数は、Rust のコードにおいてよく見かける存在です。すでに、言語において最も重要な関数のうちの一つを目撃していますね: そう、`main`関数です。これは、多くのプログラムのエントリーポイント (脚注: プログラム実行時に最初に走る関数のこと) になります。`fn`キーワードもすでに見かけましたね。これによって新しい関数を宣言することができます。

Rust の関数と変数の命名規則は、**スネークケース** (脚注: some_variable のような命名規則) です。スネークケースとは、全文字を小文字にし、単語区切りにアンダースコアを使うことです。以下のプログラムで、サンプルの関数定義をご覧ください:

ファイル名: src/main.rs

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");  // 別の関数
}
```

Rust において関数定義は、`fn`キーワードで始まり、関数名の後に丸かっこの組が続きます。波かっこの位置が、コンパイラが関数本体の位置と判断する場所です。

定義した関数は、名前に丸かっこの組を続けることで呼び出すことができます。`another_function`関数がプログラム内で定義されているので、`main`関数内から呼び出すことができるわけです。ソース

コード中で another_functionを mainの**後**に定義していることに注目してください; 勿論、main 関数の前に定義することもできます。コンパイラは、関数がどこで定義されているかは気にしません。どこかで定義されていることのみ気にします。

functions という名前の新しいバイナリ生成プロジェクトを始めて、関数についてさらに深く探求していきましょう。another_functionの例を *src/main.rs* ファイルに配置して、走らせてください。以下のような出力が得られるはずです:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28 secs
     Running `target/debug/functions`
Hello, world!
Another function.
```

行出力は、main関数内に書かれた順序で実行されています。最初に、”Hello, world” メッセージが出、それから another_functionが呼ばれて、こちらのメッセージが出力されています。

5.3.1　関数の引数

関数は、引数を持つようにも定義できます。引数とは、関数シグニチャの一部になる特別な変数のことです。関数に引数があると、引数の位置に実際の値を与えることができます。技術的にはこの実際の値は**実引数**と呼ばれますが、普段の会話では、仮引数 (”parameter”) と実引数 (”argument”) を関数定義の変数と関数呼び出し時に渡す実際の値、両方の意味に区別なく使います (脚注: 日本語では、どちらも単に引数と呼ぶことが多いでしょう)。

以下の書き直した another_functionでは、Rust の仮引数がどんな見た目なのかを示しています:

ファイル名: src/main.rs

```
1 fn main() {
2     another_function(5);
3 }
4
5 fn another_function(x: i32) {
6     println!("The value of x is: {}", x);   // xの値は{}です
7 }
```

このプログラムを走らせてみてください; 以下のような出力が得られるはずです:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 1.21 secs
     Running `target/debug/functions`
The value of x is: 5
```

another_functionの宣言には、xという名前の仮引数があります。xの型は、i32と指定されています。値 5が another_functionに渡されると、println!マクロにより、フォーマット文字列中の 1 組の波かっこがある位置に値 5が出力されます。

関数シグニチャにおいて、各仮引数の型を宣言しなければ**なりません**。これは、Rust の設計において、意図的な判断です: 関数定義で型注釈が必要不可欠ということは、コンパイラがその意図するところを推し量るのに、コードの他の箇所で使用する必要がないということを意味します。

関数に複数の仮引数をもたせたいときは、仮引数定義をカンマで区切ってください。こんな感じです:

ファイル名: src/main.rs

```
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
```

この例では、2 引数の関数を生成しています。そして、引数はどちらも `i32`型です。それからこの関数は、仮引数の値を両方出力します。関数引数は、全てが同じ型である必要はありません。今回は、偶然同じになっただけです。

このコードを走らせてみましょう。今、*function* プロジェクトの *src/main.rs* ファイルに記載されているプログラムを先ほどの例と置き換えて、`cargo run`で走らせてください:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running `target/debug/functions`
The value of x is: 5
The value of y is: 6
```

`x`に対して値 `5`、`y`に対して値 `6`を渡して関数を呼び出したので、この二つの文字列は、この値で出力されました。

5.3.2　関数本体

関数本体は、文が並び、最後に式を置くか文を置くという形で形成されます。現在までには、式で終わらない関数だけを見てきたわけですが、式が文の一部になっているものなら見かけましたね。Rust は、式指向言語なので、これは理解しておくべき重要な差異になります。他の言語にこの差異はありませんので、文と式がなんなのかと、その違いが関数本体にどんな影響を与えるかを見ていきましょう。

5.3.3　文と式

実のところ、もう文と式は使っています。**文**とは、なんらかの動作をして値を返さない命令です。**式**は結果値に評価されます。ちょっと例を眺めてみましょう。

letキーワードを使用して変数を生成し、値を代入することは文になります。リスト 3-1 で let y = 6;は文です:

ファイル名: src/main.rs

```
fn main() {
    let y = 6;
}
```

リスト 3-1: 1 文を含む main関数宣言

関数定義も文になります。つまり、先の例は全体としても文になるわけです。

文は値を返しません。故に、let文を他の変数に代入することはできません。以下のコードでは試みてますけどね; エラーになります:

ファイル名: src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```

このプログラムを実行すると、以下のようなエラーが出るでしょう:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
(エラー: 式を予期しましたが、文が見つかりました (`let`))
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
  |
  = note: variable declaration using `let` is a statement
    (注釈: `let`を使う変数宣言は、文です)
```

この let y = 6という文は値を返さないので、xを束縛する相手がないわけです。これは、C や Ruby などの言語とは異なる動作です。C や Ruby では、代入は代入値を返します。これらの言語では、x = y = 6と書いて、xも yも値 6 になるようにできるのですが、Rust においては、そうは問屋が卸さないわけです。

式は何かに評価され、これからあなたが書く Rust コードの多くを構成します。簡単な数学演算 (5 + 6など) を思い浮かべましょう。この例は、値 11に評価される式です。式は文の一部になりえます: リスト 3-1 において、let y = 6という文の 6は値 6に評価される式です。関数呼び出しも式です。マクロ呼び出しも式です。新しいスコープを作る際に使用するブロック ({}) も式です:

ファイル名: src/main.rs

```
fn main() {
    let x = 5;

    let y = {
        let x = 3;
```

```
        x + 1
    };

    println!("The value of y is: {}", y);
}
```

以下の式は:

```
{
    let x = 3;
    x + 1
}
```

今回の場合、4に評価されるブロックです。その値が、let文の一部として yに束縛されます。今まで見かけてきた行と異なり、文末にセミコロンがついていない x + 1の行に気をつけてください。式は終端にセミコロンを含みません。式の終端にセミコロンを付けたら、文に変えてしまいます。そして、文は値を返しません。次に関数の戻り値や式を見ていく際にこのことを肝に命じておいてください。

5.3.4　戻り値のある関数

関数は、それを呼び出したコードに値を返すことができます。戻り値に名前付けはできませんが、矢印 (->) の後に型を書いて宣言します。Rust では、関数の戻り値は、関数本体ブロックの最後の式の値と同義です。returnキーワードで関数から早期リターンし、値を指定することもできますが、多くの関数は最後の式を暗黙的に返します。こちらが、値を返す関数の例です:

ファイル名: src/main.rs

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {}", x);
}
```

five関数内には、関数呼び出しもマクロ呼び出しも、let文でさえ存在しません。数字の 5 が単独であるだけです。これは、Rust において、完璧に問題ない関数です。関数の戻り値型が-> i32と指定されていることにも注目してください。このコードを実行してみましょう; 出力はこんな感じになるはずです:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
     Running `target/debug/functions`
The value of x is: 5
```

five内の 5が関数の戻り値です。だから、戻り値型が i32なのです。これについてもっと深く考察しましょう。重要な箇所は 2 つあります: まず、let x = five()という行は、関数の戻り値を使って変数を初期化していることを示しています。関数 fiveは 5を返すので、この行は以下のように書くのと同義です:

```
let x = 5;
```

2 番目に、five関数は仮引数をもたず、戻り値型を定義していますが、関数本体はセミコロンなしの 5単独です。なぜなら、これが返したい値になる式だからです。

もう一つ別の例を見ましょう:

ファイル名: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

このコードを走らせると、The value of x is: 6と出力されるでしょう。しかし、x + 1を含む行の終端にセミコロンを付けて、式から文に変えたら、エラーになるでしょう:

ファイル名: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

このコードを実行すると、以下のようにエラーが出ます:

```
error[E0308]: mismatched types
              (型が合いません)
 --> src/main.rs:7:28
  |
7 |   fn plus_one(x: i32) -> i32 {
  |  ____________________________^
8 | |     x + 1;
  | |          - help: consider removing this semicolon
9 | | }
  | |_^ expected i32, found ()
  |     (i32を予期したのに、()型が見つかりました)
```

```
  |
  = note: expected type `i32`
             found type `()`
```

メインのエラーメッセージである「型が合いません」でこのコードの根本的な問題が明らかになるでしょう。関数 plus_oneの定義では、i32型を返すと言っているのに、文は値に評価されないからです。このことは、()、つまり空のタプルとして表現されています。それゆえに、何も戻り値がなく、これが関数定義と矛盾するので、結果としてエラーになるわけです。この出力内で、コンパイラは問題を修正する手助けになりそうなメッセージも出していますね: セミコロンを削除するよう提言しています。そして、そうすれば、エラーは直るわけです。

5.4　コメント

全プログラマは、自分のコードがわかりやすくなるよう努めますが、時として追加の説明が許されることもあります。このような場合、プログラマは注釈または**コメント**をソースコードに残し、コメントをコンパイラは無視しますが、ソースコードを読む人間には有益なものと思えるでしょう。

こちらが単純なコメントです:

```
// Hello, world.
```

Rust では、コメントは 2 連スラッシュで始め、行の終わりまで続きます。コメントが複数行にまたがる場合、各行に//を含める必要があります。こんな感じに:

```
// So we're doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment will
// explain what's going on.
// ここで何か複雑なことをしていて、長すぎるから複数行のコメントが必要なんだ。
// ふう！願わくば、このコメントで何が起きているか説明されていると嬉しい。
```

コメントは、コードが書かれた行の末尾にも配置することができます:

Filename: src/main.rs

```
fn main() {
    let lucky_number = 7; // I'm feeling lucky today.(今日はラッキーな気がするよ)
}
```

しかし、こちらの形式のコメントの方が見かける機会は多いでしょう。注釈しようとしているコードの 1 行上に書く形式です:

ファイル名: src/main.rs

```
fn main() {
    // I'm feeling lucky today.
    // 今日はラッキーな気がするよ
    let lucky_number = 7;
}
```

Rust には他の種類のコメント、ドキュメントコメントもあり、それについては第 14 章で議論します。

5.5　フロー制御

条件が真かどうかによってコードを走らせるかどうかを決定したり、条件が真の間繰り返しコードを走らせるか決定したりすることは、多くのプログラミング言語において、基本的な構成ブロックです。Rust コードの実行フローを制御する最も一般的な文法要素は、`if`式とループです。

5.5.1　`if` 式

if 式によって、条件に依存して枝分かれをさせることができます。条件を与え、以下のように宣言します。「もし条件が合ったら、この一連のコードを実行しろ。条件に合わなければ、この一連のコードは実行するな」と。

projects ディレクトリに *branches* という名のプロジェクトを作って `if`式について掘り下げていきましょう。*src/main.rs* ファイルに、以下のように入力してください:

ファイル名: src/main.rs

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");        // 条件は真です
    } else {
        println!("condition was false");       // 条件は偽です
    }
}
```

`if`式は全て、キーワードの `if`から始め、条件式を続けます。今回の場合、条件式は変数 `number`が 5 未満の値になっているかどうかをチェックします。条件が真の時に実行したい一連のコードを条件式の直後に波かっこで包んで配置します。`if`式の条件式と紐付けられる一連のコードは、時として**アーム**と呼ばれることがあります。第 2 章の「予想と秘密の数字を比較する」の節で議論した `match`式のアームのようですね。

オプションとして、`else`式を含むこともでき (ここではそうしています)、これによりプログラムは、条件式が偽になった時に実行するコードを与えられることになります。仮に、`else`式を与えずに条件式が偽になったら、プログラムは単に `if`ブロックを無視して次のコードを実行しにいきます。

このコードを走らせてみましょう; 以下のような出力を目の当たりにするはずです:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running `target/debug/branches`
condition was true
```

numberの値を条件が falseになるような値に変更してどうなるか確かめてみましょう:

```
let number = 7;
```

再度プログラムを実行して、出力に注目してください:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running `target/debug/branches`
condition was false
```

このコード内の条件式は、bool型で**なければならない**ことにも触れる価値があります。条件式が、bool型でない時は、エラーになります。例です:

ファイル名: src/main.rs

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");      // 数値は3です
    }
}
```

今回、ifの条件式は 3という値に評価され、コンパイラがエラーを投げます:

```
error[E0308]: mismatched types
              (型が合いません)
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected bool, found integral variable
  |                (bool型を予期したのに、整数変数が見つかりました)
  |
  = note: expected type `bool`
             found type `{integer}`
```

このエラーは、コンパイラは bool型を予期していたのに、整数だったことを示唆しています。Ruby や JavaScript などの言語とは異なり、Rust では、論理値以外の値が、自動的に論理値に変換されることはありません。明示的に必ず ifには条件式として、論理値を与えなければなりません。例えば、数値が 0以外の時だけ ifのコードを走らせたいなら、以下のように if式を変更することができます:

ファイル名: src/main.rs

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");   // 数値は0以外の何かです
    }
}
```

このコードを実行したら、`number was something other than zero`と表示されるでしょう。

5.5.1.1　`else if` で複数の条件を扱う

`if`と `else`を組み合わせて `else if`式にすることで複数の条件を持たせることもできます。例です:

ファイル名: src/main.rs

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        // 数値は4で割り切れます
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        // 数値は3で割り切れます
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        // 数値は2で割り切れます
        println!("number is divisible by 2");
    } else {
        // 数値は4、3、2で割り切れません
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

このプログラムには、通り道が 4 つあります。実行後、以下のような出力を目の当たりにするはずです:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running `target/debug/branches`
number is divisible by 3
```

このプログラムを実行すると、`if`式が順番に吟味され、最初に条件が真になった本体が実行されます。6 は 2 で割り切れるものの、`number is devisible by 2`や、`else`ブロックの `number is not divisible by 4, 3, or 2`という出力はされないことに注目してください。それは、言語が最初の真条件のブロックのみを実行し、条件に合ったものが見つかったら、残りはチェックすらしないからです。

`else if`式を使いすぎると、コードがめちゃくちゃになってしまうので、1 つ以上あるなら、コードをリファクタリングしたくなるかもしれません。これらのケースに有用な `match`と呼ばれる、強力な Rust の枝分かれ文法要素については第 6 章で解説します。

5.5.1.2　`let` 文内で `if` 式を使う

`if`は式なので、`let`文の右辺に持ってくることができます。リスト 3-2 のようにね:

ファイル名: src/main.rs

```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        6
    };

    // numberの値は、{}です
    println!("The value of number is: {}", number);
}
```

リスト 3-2: if式の結果を変数に代入する

この number変数は、if式の結果に基づいた値に束縛されます。このコードを走らせてどうなるか確かめてください:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
     Running `target/debug/branches`
The value of number is: 5
```

一連のコードは、そのうちの最後の式に評価され、数値はそれ単独でも式になることを思い出してください。今回の場合、この if式全体の値は、どのブロックのコードが実行されるかに基づきます。これはつまり、ifの各アームの結果になる可能性がある値は、同じ型でなければならないということになります; リスト 3-4 で、ifアームも elseアームも結果は、i32の整数でした。以下の例のように、型が合わない時には、エラーになるでしょう:

ファイル名: src/main.rs

```
fn main() {
    let condition = true;

    let number = if condition {
        5
    } else {
        "six"
    };

    println!("The value of number is: {}", number);
}
```

このコードを走らせようとすると、エラーになります。ifと elseアームは互換性のない値の型になり、コンパイラがプログラム内で問題の見つかった箇所をスバリ指摘してくれます:

```
error[E0308]: if and else have incompatible types
              (ifとelseの型に互換性がありません)
 --> src/main.rs:4:18
  |
4 |       let number = if condition {
```

```
  |  ___________________^
5 | |         5
6 | |     } else {
7 | |         "six"
8 | |     };
  | |_____^ expected integral variable, found &str
  |         (整数変数を予期しましたが、&strが見つかりました)
  |
  = note: expected type `{integer}`
             found type `&str`
```

ifブロックの式は整数に評価され、elseブロックの式は文字列に評価されます。これでは動作しません。変数は単独の型でなければならないからです。コンパイラは、コンパイル時に number変数の型を確実に把握する必要があるため、コンパイル時に numberが使われている箇所全部で型が有効かどうか検査することができるのです。numberの型が実行時にしか決まらないのであれば、コンパイラはそれを実行することができなくなってしまいます; どの変数に対しても、架空の複数の型があることを追いかけなければならないのであれば、コンパイラはより複雑になり、コードに対して行える保証が少なくなってしまうでしょう。

5.5.2　ループでの繰り返し

一連のコードを 1 回以上実行できると、しばしば役に立ちます。この作業用に、Rust にはいくつかの**ループ**が用意されています。ループは、本体内のコードを最後まで実行し、直後にまた最初から処理を開始します。ループを試してみるのに、*loops* という名の新プロジェクトを作りましょう。

Rust には 3 種類のループが存在します: loopと whileと forです。それぞれ試してみましょう。

5.5.2.1　loop でコードを繰り返す

loopキーワードを使用すると、同じコードを何回も何回も永遠に明示的にやめさせるまで実行します。

例として、*loops* ディレクトリの *src/main.rs* ファイルを以下のような感じに書き換えましょう:

ファイル名: src/main.rs

```
fn main() {
    loop {
        println!("again!");   // また
    }
}
```

このプログラムを実行すると、プログラムを手動で止めるまで、何度も何度も続けて again!と出力するでしょう。ほとんどのターミナルで ctrl-c というショートカットが使え、永久ループに囚われてしまったプログラムを終了させられます。試しにやってみましょう:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.29 secs
     Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

^Cという記号が出た場所が、ctrl-c を押した場所です。^Cの後には again!と表示されたり、されなかったりします。ストップシグナルをコードが受け取った時にループのどこにいたかによります。

幸いなことに、Rust にはループを抜け出す別のより信頼できる手段があります。ループ内に break キーワードを配置することでプログラムに実行を終了すべきタイミングを教えることができます。第 2 章の「正しい予想をした後に終了する」節の数当てゲーム内でこれをして、ユーザが予想を的中させ、ゲームに勝った時にプログラムを終了させたことを思い出してください。

5.5.2.2　while で条件付きループ

プログラムにとってループ内で条件式を評価できると、有益なことがしばしばあります。条件が真の間、ループが走るわけです。条件が真でなくなった時にプログラムは breakを呼び出し、ループを終了します。このタイプのループは、loop、if、else、breakを組み合わせることでも実装できます; お望みなら、プログラムで試してみるのもいいでしょう。

しかし、このパターンは頻出するので、Rust にはそれ用の文法要素が用意されていて、whileループと呼ばれます。リスト 3-3 は、whileを使用しています: プログラムは 3 回ループし、それぞれカウントダウンします。それから、ループ後に別のメッセージを表示して終了します:

ファイル名: src/main.rs

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);

        number = number - 1;
    }

    // 発射！
    println!("LIFTOFF!!!");
}
```

リスト 3-3: 条件が真の間、コードを走らせる whileループを使用する

この文法要素により、loop、if、else、breakを使った時に必要になるネストがなくなり、より明確になります。条件が真の間、コードは実行されます; そうでなければ、ループを抜けます.

5.5.2.3　for でコレクションを覗き見る

while要素を使って配列などのコレクションの要素を覗き見ることができます。例えば、リスト 3-4 を見ましょう:

ファイル名: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        // 値は{}です
        println!("the value is: {}", a[index]);

        index = index + 1;
    }
}
```

リスト 3-4: whileループでコレクションの各要素を覗き見る

ここで、コードは配列の要素を順番にカウントアップして覗いています。番号 0 から始まり、配列の最終番号に到達するまでループします (つまり、index < 5が真でなくなる時です)。このコードを走らせると、配列内の全要素が出力されます:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
     Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

予想通り、配列の 5 つの要素が全てターミナルに出力されています。index変数の値はどこかで 5という値になるものの、配列から 6 番目の値を拾おうとする前にループは実行を終了します。

しかし、このアプローチは間違いが発生しやすいです; 添え字の長さが間違っていれば、プログラムはパニックしてしまいます。また遅いです。コンパイラが実行時にループの各回ごとに境界値チェックを行うようなコードを追加するからです。

より効率的な対立案として、forループを使ってコレクションの各アイテムに対してコードを実行することができます。forループはリスト 3-5 のこんな見た目です:

ファイル名: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        // 値は{}です
```

```
        println!("the value is: {}", element);
    }
}
```

リスト 3-4: forループを使ってコレクションの各要素を覗き見る

このコードを走らせたら、リスト 3-4 と同じ出力が得られるでしょう。より重要なのは、コードの安全性を向上させ、配列の終端を超えてアクセスしたり、終端に届く前にループを終えてアイテムを見逃してしまったりするバグの可能性を完全に排除したことです。

例えば、リスト 3-4 のコードで、a配列からアイテムを 1 つ削除したのに、条件式を `while index < 4` にするのを忘れていたら、コードはパニックします。forループを使っていれば、配列の要素数を変えても、他のコードをいじることを覚えておく必要はなくなるわけです。

forループのこの安全性と簡潔性により、Rust で使用頻度の最も高いループになっています。リスト 3-5 で whileループを使ったカウントダウンサンプルのように、一定の回数、同じコードを実行したいような状況であっても、多くの Rust 市民は、forループを使うでしょう。どうやってやるかといえば、Range型を使うのです。Range 型は、標準ライブラリで提供される片方の数字から始まって、もう片方の数字未満の数値を順番に生成する型です。

forループを使い、まだ話していない別のメソッド revを使って範囲を逆順にしたカウントダウンはこうなります:

ファイル名: src/main.rs

```
fn main() {
    for number in (1..4).rev() {
        println!("{}!", number);
    }
    println!("LIFTOFF!!!");
}
```

こちらのコードの方が少しいいでしょう？

5.6　まとめ

やりましたね！ 結構長い章でした: 変数とスカラー値、複合データ型、関数、コメント、if式、そして、ループについて学びました！ この章で議論した概念について経験を積みたいのであれば、以下のことをするプログラムを組んでみてください:

- 温度を華氏と摂氏で変換する。
- フィボナッチ数列の n 番目を生成する。
- クリスマスキャロルの定番、”The Twelve Days of Christmas” の歌詞を、曲の反復性を利用して出力する。

次に進む準備ができたら、他の言語にはあまり存在しない Rust の概念について話しましょう: 所有権です。

第6章

所有権を理解する

所有権はRustの最もユニークな機能であり、これのおかげでガベージコレクタなしで安全性担保を行うことができるのです。故に、Rustにおいて、所有権がどう動作するのかを理解するのは重要です。この章では、所有権以外にも、関連する機能をいくつか話していきます: 借用、スライス、そして、コンパイラがデータをメモリにどう配置するかです。«««< HEAD

6.1 所有権とは？

Rustの中心的な機能は、**所有権**です。機能は説明するのに単純なのですが、言語の残りの機能全てにかかるほど深い裏の意味を含んでいるのです。

全てのプログラムは、実行中にコンピュータのメモリの使用方法を管理する必要があります。プログラムが動作するにつれて、定期的に使用されていないメモリを検索するガベージコレクションを持つ言語もありますが、他の言語では、プログラマが明示的にメモリを確保したり、解放したりしなければなりません。Rustでは第3の選択肢を取っています: メモリは、コンパイラがコンパイル時にチェックする一定の規則とともに所有権システムを通じて管理されています。どの所有権機能も、実行中にプログラムの動作を遅くすることはありません。

所有権は多くのプログラマにとって新しい概念なので、慣れるまでに時間がかかります。Rustと所有権システムの規則と経験を積むにつれて、自然に安全かつ効率的なコードを構築できるようになることは、素晴らしいお知らせです。その調子でいきましょう！

所有権を理解した時、Rustを際立たせる機能の理解に対する強固な礎を得ることになるでしょう。この章では、非常に一般的なデータ構造に着目した例を取り扱うことで所有権を学んでいくでしょう: 文字列です。

6.1.1　スタックとヒープ

多くのプログラミング言語において、スタックとヒープについて考える機会はそう多くないでしょう。しかし、Rust のようなシステムプログラミング言語においては、値がスタックに載るかヒープに載るかは、言語の振る舞い方や、特定の決断を下す理由などに影響以上のものを与えるのです。この章の後半でスタックとヒープを絡めて所有権に一部は解説されるので、ここでちょっと予行演習をしておきましょう。

スタックもヒープも、実行時にコードが使用できるメモリの一部になりますが、異なる手段で構成されています。スタックは、得た順番に値を並べ、逆の順で値を取り除いていきます。これは、*last in, first out*(脚注: あえて日本語にするなら、けつ入れ頭出しといったところでしょうか) と呼ばれます。お皿の山を思い浮かべてください: お皿を追加する時には、山の一番上に置き、お皿が必要になったら、一番上から 1 枚を取り去りますよね。途中や一番下に追加したり、取り除いたりすることは同じようにはできません。データを追加することは、**スタックに *push* する**といい、データを取り除くことは、**スタックから *pop* する**と表現します (脚注: これらの動作に対する画一的な日本語訳を見かけたことはありません)。

データへのアクセス方法のおかげで、スタックは高速です: 新データを置いたり、データを取得する場所を探す必要が絶対にないわけです。というのも、その場所は常に一番上だからですね。スタックを高速にする特性は、他にもあり、それはスタック上のデータは全て既知の固定サイズにならなければならないということです。

コンパイル時にサイズがわからなかったり、サイズが可変のデータについては、代わりにヒープに格納することができます。ヒープは、もっとごちゃごちゃしています: ヒープにデータを置く時、あるサイズのスペースを求めます。OS はヒープ上に十分な大きさの空の領域を見つけ、使用中にし、**ポインタ**を返してきます。ポインタとは、その場所へのアドレスです。この過程は、**ヒープに領域を確保する**と呼ばれ、時としてそのフレーズを単に *allocate* **する**などと省略したりします。(脚注: こちらもこなれた日本語訳はないでしょう。allocate はメモリを確保すると訳したいところですが) スタックに値を載せることは、メモリ確保とは考えられません。ポインタは、既知の固定サイズなので、スタックに保管することができますが、実データが必要になったら、ポインタを追いかける必要があります。

レストランで席を確保することを考えましょう。入店したら、グループの人数を告げ、店員が全員座れる空いている席を探し、そこまで誘導します。もしグループの誰かが遅れて来るのなら、着いた席の場所を尋ねてあなたを発見することができます。

ヒープへのデータアクセスは、スタックのデータへのアクセスよりも低速です。ポインタを追って目的の場所に到達しなければならないからです。現代のプロセッサは、メモリをあちこち行き来しなければ、より速くなります。似た例えを続けましょう。レストランで多くのテーブルから注文を受ける給仕人を考えましょう。最も効率的なのは、次のテーブルに移らずに、一つのテーブルで全部の注文を受け付けてしまうことです。テーブル A で注文を受け、それか

らテーブル B の注文、さらにまた A、それからまた B と渡り歩くのは、かなり低速な過程になってしまうでしょう。同じ意味で、プロセッサは、データが隔離されている (ヒープではそうなっている可能性がある) よりも近くにある (スタックではこうなる) ほうが、仕事をうまくこなせるのです。ヒープに大きな領域を確保する行為も時間がかかることがあります。
コードが関数を呼び出すと、関数に渡された値 (ヒープのデータへのポインタも含まれる可能性あり) と、関数のローカル変数がスタックに載ります。関数の実行が終了すると、それらの値はスタックから取り除かれます。
どの部分のコードがどのヒープ上のデータを使用しているか把握すること、ヒープ上の重複するデータを最小化すること、メモリ不足にならないようにヒープ上の未使用のデータを掃除することは全て、所有権が解決する問題です。一度所有権を理解したら、あまり頻繁にスタックとヒープに関して考える必要はなくなるでしょうが、ヒープデータを管理することが所有権の存在する理由だと知っていると、所有権がありのままで動作する理由を説明するのに役立つこともあります。

6.1.2　所有権規則

まず、所有権のルールについて見ていきましょう。この規則を具体化する例を扱っていく間もこれらのルールを肝に命じておいてください:

1. Rust の各値は、**所有者**と呼ばれる変数と対応している。
2. いかなる時も所有者は一つである。
3. 所有者がスコープから外れたら、値は破棄される。

6.1.3　変数スコープ

第 2 章で、Rust プログラムの例はすでに見ています。もう基本的な記法は通り過ぎたので、`fn main() {`というコードはもう例に含みません。従って、例をなぞっているなら、これからの例は `main` 関数に手動で入れ込まなければいけなくなるでしょう。結果的に、例は少々簡潔になり、定型コードよりも具体的な詳細に集中しやすくなります。

所有権の最初の例として、何らかの変数の**スコープ**について見ていきましょう。スコープとは、要素が有効になるプログラム内の範囲のことです。以下のような変数があるとしましょう:

```
let s = "hello";
```

変数 sは、文字列リテラルを参照し、ここでは、文字列の値はプログラムのテキストとしてハードコードされています。この変数は、宣言された地点から、現在の**スコープ**の終わりまで有効になります。リスト 4-1 には、変数 sが有効な場所に関する注釈がコメントで付記されています:

```
{                      // sは、ここでは有効ではない。まだ宣言されていない
```

```
    let s = "hello";   // sは、ここから有効になる

    // sで作業をする
}                      // このスコープは終わり。もうsは有効ではない
```

リスト 4-1: 変数と有効なスコープ

言い換えると、ここまでに重要な点は二つあります:

1. sがスコープに入ると、有効になる
2. スコープを抜けるまで、有効なまま

ここで、スコープと変数が有効になる期間の関係は、他の言語に類似しています。さて、この理解のもとに、String型を導入して構築していきましょう。

6.1.4　String 型

所有権の規則を具体化するには、第 3 章の「データ型」節で講義したものよりも、より複雑なデータ型が必要になります。以前講義した型は全てスタックに保管され、スコープが終わるとスタックから取り除かれますが、ヒープに確保されるデータ型を観察して、コンパイラがどうそのデータを掃除すべきタイミングを把握しているかを掘り下げていきたいです。

ここでは、例として String型を使用し、String型の所有権にまつわる部分に着目しましょう。また、この観点は、標準ライブラリや自分で生成する他の複雑なデータ型にも適用されます。String型については、第 8 章でより深く議論します。

すでに文字列リテラルは見かけましたね。文字列リテラルでは、文字列の値はプログラムにハードコードされます。文字列リテラルは便利ですが、テキストを使いたいかもしれない場面全てに最適なわけではありません。一因は、文字列リテラルが不変であることに起因します。別の原因は、コードを書く際に、全ての文字列値が判明するわけではないからです: 例えば、ユーザ入力を受け付け、それを保持したいとしたらどうでしょうか？ このような場面用に、Rust には、2 種類目の文字列型、String型があります。この型はヒープにメモリを確保するので、コンパイル時にはサイズが不明なテキストも保持することができるのです。from関数を使用して、文字列リテラルから String型を生成できます。以下のように:

```
let s = String::from("hello");
```

この二重コロンは、string_fromなどの名前を使うのではなく、String型直下の from関数を特定する働きをする演算子です。この記法について詳しくは、第 5 章の「メソッド記法」節と、第 7 章の「モジュール定義」でモジュールを使った名前空間分けについて話をするときに議論します。

この種の文字列は、可変化することができます:

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str()関数は、リテラルをStringに付け加える
```

```
4
5 println!("{}", s); // これは`hello, world!`と出力する
```

では、ここでの違いは何でしょうか？ なぜ、String型は可変化できるのに、リテラルはできないのでしょうか？ 違いは、これら二つの型がメモリを扱う方法にあります。

6.1.5　メモリと確保

文字列リテラルの場合、中身はコンパイル時に判明しているので、テキストは最終的なバイナリファイルに直接ハードコードされます。このため、文字列リテラルは、高速で効率的になるのです。しかし、これらの特性は、その文字列リテラルの不変性にのみ端を発するものです。残念なことに、コンパイル時にサイズが不明だったり、プログラム実行に合わせてサイズが可変なテキスト片用に一塊のメモリをバイナリに確保しておくことは不可能です。

String型では、可変かつ伸長可能なテキスト破片をサポートするために、コンパイル時には不明な量のメモリをヒープに確保して内容を保持します。つまり:

- メモリは、実行時に OS に要求される。
- String型を使用し終わったら、OS にこのメモリを返還する方法が必要である。

この最初の部分は、すでにしています: String::from関数を呼んだら、その実装が必要なメモリを要求するのです。これは、プログラミング言語において、極めて普遍的です。

しかしながら、2 番目の部分は異なります。**ガベージコレクタ** (GC) 付きの言語では、GC がこれ以上、使用されないメモリを検知して片付けるため、プログラマは、そのことを考慮する必要はありません。GC がないなら、メモリがもう使用されないことを見計らって、明示的に返還するコードを呼び出すのは、プログラマの責任になります。ちょうど要求の際にしたようにですね。これを正確にすることは、歴史的にも難しいプログラミング問題の一つであり続けています。もし、忘れていたら、メモリを無駄にします。タイミングが早すぎたら、無効な変数を作ってしまいます。2 回解放してしまっても、バグになるわけです。allocateと freeは完璧に 1 対 1 対応にしなければならないのです。

Rust は、異なる道を歩んでいます: ひとたび、メモリを所有している変数がスコープを抜けたら、メモリは自動的に返還されます。こちらの例は、リスト 4-1 のスコープ例を文字列リテラルからString型を使うものに変更したバージョンになります:

```
1 {
2     let s = String::from("hello"); // sはここから有効になる
3
4     // sで作業をする
5 }                                   // このスコープはここでおしまい。sは
6                                     // もう有効ではない
```

String型が必要とするメモリを OS に返還することが自然な地点があります: s変数がスコープを抜ける時です。変数がスコープを抜ける時、Rust は特別な関数を呼んでくれます。この関数は、drop

と呼ばれ、ここに String型の書き手はメモリ返還するコードを配置することができます。Rust は、閉じ}で自動的に drop関数を呼び出します。

> 注釈: C++ では、要素の生存期間の終了地点でリソースを解放するこのパターンを時に、*RAII*(Resource Aquisition Is Initialization: リソースの獲得は、初期化である) と呼んだりします。Rust の drop関数は、あなたが RAII パターンを使ったことがあれば、馴染み深いものでしょう。

このパターンは、Rust コードの書かれ方に甚大な影響をもたらします。現状は簡単そうに見えるかもしれませんが、ヒープ上に確保されたデータを複数の変数に使用させるようなもっと複雑な場面では、コードの振る舞いは、予期しないものになる可能性もあります。これから、そのような場面を掘り下げてみましょう。

6.1.5.1　変数とデータの相互作用法: ムーブ

Rust においては、複数の変数が同じデータに対して異なる手段で相互作用することができます。整数を使用したリスト 4-2 の例を見てみましょう:

```
let x = 5;
let y = x;
```

リスト 4-2: 変数 xの整数値を yに代入する

もしかしたら、何をしているのか予想することができるでしょう:「値 5を xに束縛する; それから x の値をコピーして yに束縛する。」これで、二つの変数 (xと y) が存在し、両方、値は 5になりました。これは確かに起こっている現象を説明しています。なぜなら、整数は既知の固定サイズの単純な値で、これら二つの 5という値は、スタックに積まれるからです。

では、Stringバージョンを見ていきましょう:

```
let s1 = String::from("hello");
let s2 = s1;
```

このコードは先ほどのコードに酷似していますので、動作方法も同じだと思い込んでしまうかもしれません: 要するに、2 行目で s1の値をコピーし、s2に束縛するということです。ところが、これは全く起こることを言い当てていません。

図 4-1 を見て、ベールの下で Stringに何が起きているかを確かめてください。String型は、左側に示されているように、3 つの部品でできています: 文字列の中身を保持するメモリへのポインタと長さ、そして、許容量です。この種のデータは、スタックに保持されます。右側には、中身を保持したヒープ上のメモリがあります。

図 4-1: s1に束縛された"hello"という値を保持する Stringのメモリ上の表現

長さは、String型の中身が現在使用しているメモリ量をバイトで表したものです。許容量は、String型が OS から受け取った全メモリ量をバイトで表したものです。長さと許容量の違いは問題になることですが、この文脈では違うので、とりあえずは、許容量を無視しても構わないでしょう。

s1を s2に代入すると、String型のデータがコピーされます。つまり、スタックにあるポインタ、長さ、許容量をコピーするということです。ポインタが指すヒープ上のデータはコピーしません。言い換えると、メモリ上のデータ表現は図 4-4 のようになるということです。

図 4-2: s1のポインタ、長さ、許容量のコピーを保持する変数 s2のメモリ上での表現

メモリ上の表現は、図 4-5 のようにはなりません。これは、Rust が代わりにヒープデータもコピーするという選択をしていた場合のメモリ表現ですね。Rust がこれをしていたら、ヒープ上のデータが大きい時に s2 = s1という処理の実行時性能がとても悪くなっていた可能性があるでしょう。

図 4-3: Rust がヒープデータもコピーしていた場合に s2 = s1という処理が行なった可能性のあること

先ほど、変数がスコープを抜けたら、Rust は自動的に drop関数を呼び出し、その変数が使っていたヒープメモリを片付けると述べました。しかし、図 4-4 は、両方のデータポインタが同じ場所を指していることを示しています。これは問題です: s2と s1がスコープを抜けたら、両方とも同じメモリを解放しようとします。これは**二重解放**エラーとして知られ、以前触れたメモリ安全性上のバグの一つになります。メモリを 2 回解放することは、メモリの退廃につながり、さらにセキュリティ上の脆弱性を生む可能性があります。

メモリ安全性を保証するために、Rust においてこの場面で知っておきたい起こる事の詳細がもう一つあります。確保されたメモリをコピーしようとする代わりに、コンパイラは、s1が最早有効ではないと考え、故に s1がスコープを抜けた際に何も解放する必要がなくなるわけです。s2の生成後に s1を使用しようとしたら、どうなるかを確認してみましょう。動かないでしょう:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

コンパイラが無効化された参照は使用させてくれないので、以下のようなエラーが出るでしょう:

```
error[E0382]: use of moved value: `s1`
              (ムーブされた値の使用: `s1`)
 --> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
  |                               (ムーブ後にここで使用されています)
  |
  = note: move occurs because `s1` has type `std::string::String`, which does
  not implement the `Copy` trait
    (注釈: ムーブが起きたのは、`s1`が`std::string::String`という
    `Copy`トレイトを実装していない型だからです)
```

他の言語を触っている間に”shallow copy” と”deep copy” という用語を耳にしたことがあるなら、データのコピーなしにポインタと長さ、許容量をコピーするという概念は、shallow copy のように思

えるかもしれません。ですが、コンパイラは最初の変数をも無効化するので、shallow copy と呼ばれる代わりに、ムーブとして知られているわけです。ここでは、s1は s2にムーブされたと解読します。以上より、実際に起きることを図 4-4 に示してみました。

図 4-4: s1が無効化された後のメモリ表現

これにて一件落着です。s2だけが有効なので、スコープを抜けたら、それだけがメモリを解放して、終わりになります。

付け加えると、これにより暗示される設計上の選択があります: Rust では、自動的にデータの”deep copy” が行われることは絶対にないわけです。それ故に、あらゆる自動コピーは、実行時性能の観点で言うと、悪くないと考えてよいことになります。

6.1.5.2　変数とデータの相互作用法: クローン

仮に、スタック上のデータだけでなく、本当に String型のヒープデータの deep copy が必要ならば、cloneと呼ばれるよくあるメソッドを使うことができます。メソッド記法については第 5 章で議論しますが、メソッドは多くのプログラミング言語に見られる機能なので、以前に見かけたこともあるんじゃないでしょうか。

こちらで、cloneメソッドの稼働例をご覧ください:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

これは単純にうまく動き、図 4-3 で示した動作を明示的に生み出します。ここでは、ヒープデータが実際にコピーされています。

cloneメソッドの呼び出しを見かけたら、何らかの任意のコードが実行され、その実行コストは高いと把握できます。何か違うことが起こっているなと見た目でわかるわけです。

6.1.5.3　スタックのみのデータ: コピー

まだ話題にしていない別のしわ (脚注:「気になるもの」程度の意味と思われる) の話があります。この整数を使用したコードは、一部を先ほどリスト 4-2 で示しましたが、うまく動作し、有効です:

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

ですが、このコードは一見、今学んだことと矛盾しているように見えます: cloneメソッドの呼び出しがないのに、xは有効で、yにムーブされませんでした。

その理由は、整数のようなコンパイル時に既知のサイズを持つ型は、スタック上にすっぽり保持されるので、実際の値をコピーするのも高速だからです。これは、変数 yを生成した後にも xを無効化したくなる理由がないことを意味します。換言すると、ここでは、shallow copy と deep copy の違い

がないことになり、cloneメソッドを呼び出しても、一般的な shallow copy 以上のことをしなくなり、そのまま放置しておけるということです。

Rust には Copyトレイトと呼ばれる特別な注釈があり、整数のようなスタックに保持される型に対して配置することができます (トレイトについては第 10 章でもっと詳しく話します)。型が Copyトレイトに適合していれば、代入後も古い変数が使用可能になります。コンパイラは、型やその一部分でも Dropトレイトを実装している場合、Copyトレイトによる注釈をさせてくれません。型の値がスコープを外れた時に何か特別なことを起こす必要がある場合に、Copy注釈を追加すると、コンパイルエラーが出ます。型に Copy注釈をつける方法について学ぶには、付録 C の「継承可能トレイト」をご覧ください。

では、どの型が Copyなのでしょうか？ ある型について、ドキュメントをチェックすればいいのですが、一般規則として、単純なスカラー値の集合は何でも Copyであり、メモリ確保が必要だったり、何らかの形態のリソースだったりするものは Copyではありません。ここに Copyの型を並べておきます。

- あらゆる整数型。u32など。
- 論理値型、bool、trueと falseという値がある。
- あらゆる浮動小数点型、f64など。
- 文字型、char。
- タプル。ただ、Copyの型だけを含む場合。例えば、(i32, i32)は Copyだが、(i32, String)は違う。

6.1.6　所有権と関数

意味論的に、関数に値を渡すことと、値を変数に代入することは似ています。関数に変数を渡すと、代入のようにムーブやコピーされます。リスト 4-7 は変数がスコープに入ったり、抜けたりする地点について注釈してある例です:

ファイル名: src/main.rs

```
fn main() {
    let s = String::from("hello");  // sがスコープに入る

    takes_ownership(s);             // sの値が関数にムーブされ...
                                    // ... ここではもう有効ではない

    let x = 5;                      // xがスコープに入る

    makes_copy(x);                  // xも関数にムーブされるが、
                                    // i32はCopyなので、この後にxを使っても
                                    // 大丈夫

} // ここでxがスコープを抜け、sも。だけど、sの値はムーブされてるので、何も特別なことはない。
  //

```

```
fn takes_ownership(some_string: String) { // some_stringがスコープに入る。
    println!("{}", some_string);
} // ここでsome_stringがスコープを抜け、`drop`が呼ばれる。後ろ盾してたメモリが解放される。
  //

fn makes_copy(some_integer: i32) { // some_integerがスコープに入る
    println!("{}", some_integer);
} // ここでsome_integerがスコープを抜ける。何も特別なことはない。
```

リスト 4-3: 所有権とスコープが注釈された関数群

takes_ownershipの呼び出し後に sを呼び出そうとすると、コンパイラは、コンパイルエラーを投げるでしょう。これらの静的チェックにより、ミスを犯さないでいられます。sや xを使用するコードをmainに追加してみて、どこで使えて、そして、所有権規則により、どこで使えないかを確認してください。

6.1.7　戻り値とスコープ

値を返すことでも、所有権は移動します。リスト 4-4 は、リスト 4-3 と似た注釈のついた例です:

ファイル名: src/main.rs

```
fn main() {
    let s1 = gives_ownership();         // gives_ownershipは、戻り値をs1に
                                        // ムーブする

    let s2 = String::from("hello");     // s2がスコープに入る

    let s3 = takes_and_gives_back(s2);  // s2はtakes_and_gives_backにムーブされ
                                        // 戻り値もs3にムーブされる
} // ここで、s3はスコープを抜け、ドロップされる。s2もスコープを抜けるが、ムーブされているので、
  // 何も起きない。s1もスコープを抜け、ドロップされる。

fn gives_ownership() -> String {             // gives_ownershipは、戻り値を
                                             // 呼び出した関数にムーブする

    let some_string = String::from("hello"); // some_stringがスコープに入る

    some_string                              // some_stringが返され、呼び出し元関数に
                                             // ムーブされる。
}

// takes_and_gives_backは、Stringを一つ受け取り、返す。
fn takes_and_gives_back(a_string: String) -> String { // a_stringがスコープに入る。

    a_string  // a_stringが返され、呼び出し元関数にムーブされる
}
```

リスト 4-4: 戻り値の所有権を移動する

変数の所有権は、毎回同じパターンを辿っています: 別の変数に値を代入すると、ムーブされます。ヒープにデータを含む変数がスコープを抜けると、データが別の変数に所有されるようムーブされていない限り、dropにより片付けられるでしょう。

所有権を得ては返すを全ての関数でしていたら、ちょっとめんどくさいですね。関数に値を使わせたいけど、所有権は保持させたくない場合はどうすればいいのでしょうか？ 返したいと思うかもしれない関数本体で発生したあらゆるデータとともに再利用したかったら、渡されたものをまた返さなきゃいけないのは、非常に煩わしいことです。

タプルで、複数の値を返すことは可能です。リスト 4-5 のようにね:

ファイル名: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    //'{}'の長さは、{}です
    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len()メソッドは、Stringの長さを返します

    (s, length)
}
```

リスト 4-5: 引数の所有権を返す

でも、これでは、大袈裟すぎますし、ありふれているはずの概念に対して、作業量が多すぎます。私たちにとって幸運なことに、Rust にはこの概念に対する機能があり、**参照**と呼ばれます。

6.2　参照と借用

リスト 4-5 のタプルコードの問題は、String型を呼び出し元の関数に戻さないと、calculate_lengthを呼び出した後に、Stringオブジェクトが使えなくなることであり、これは Stringオブジェクトがcalculate_lengthにムーブされてしまうためでした。

ここで、値の所有権をもらう代わりに引数としてオブジェクトへの参照を取る calculate_length関数を定義し、使う方法を見てみましょう:

ファイル名: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    // '{}'の長さは、{}です
    println!("The length of '{}' is {}.", s1, len);
```

```
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

まず、変数宣言と関数の戻り値にあったタプルコードは全てなくなったことに気付いてください。2番目に、`&s1`を `calcuate_length`に渡し、その定義では、`String`型ではなく、`&String`を受け取っていることに注目してください。

これらのアンド記号が参照であり、これのおかげで所有権をもらうことなく値を参照することができるのです。図 4-5 はその図解です。

図 4-5: `String s1`を指す`&String`の図表

> 注釈: `&`による参照の逆は、**参照外し**であり、参照外し演算子の`*`で達成できます。第 8 章で参照外し演算子の使用例を眺め、第 15 章で参照外しについて詳しく議論します。

ここの関数呼び出しについて、もっと詳しく見てみましょう:

```
# fn calculate_length(s: &String) -> usize {
#     s.len()
# }
let s1 = String::from("hello");

let len = calculate_length(&s1);
```

この`&s1`という記法により、`s1`の値を**参照する**参照を生成することができますが、これを所有することはありません。所有してないということは、指している値は、参照がスコープを抜けてもドロップされないということです。

同様に、関数のシグニチャでも、`&`を使用して引数 `s`の型が参照であることを示しています。説明的な注釈を加えてみましょう:

```
fn calculate_length(s: &String) -> usize { // sはStringへの参照
    s.len()
} // ここで、sはスコープ外になる。けど、参照しているものの所有権を持っているわけではないので
  // 何も起こらない
```

変数 `s`が有効なスコープは、あらゆる関数の引数のものと同じですが、所有権はないので、`s`がスコープを抜けても、参照が指しているものをドロップすることはありません。関数が実際の値の代わりに参照を引数に取ると、所有権をもらわないので、所有権を返す目的で値を返す必要はありません。

関数の引数に参照を取ることを**借用**と呼びます。現実生活のように、誰かが何かを所有していたら、それを借りることができます。用が済んだら、返さなきゃいけないわけです。

では、借用した何かを変更しようとしたら、どうなるのでしょうか？ リスト 4-6 のコードを試してください。ネタバレ注意: 動きません！

ファイル名: src/main.rs

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

リスト 4-6: 借用した値を変更しようと試みる

これがエラーです:

```
error[E0596]: cannot borrow immutable borrowed content `*some_string` as mutable
(エラー: 不変な借用をした中身`*some_string`を可変で借用できません)
 --> error.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- use `&mut String` here to make mutable
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ cannot borrow as mutable
```

変数が標準で不変なのと全く同様に、参照も不変なのです。参照している何かを変更することは叶わないわけです。

6.2.1　可変な参照

一捻り加えるだけでリスト 4-6 のコードのエラーは解決します:

ファイル名: src/main.rs

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

始めに、sを mutに変えなければなりませんでした。そして、&mut sで可変な参照を生成し、some_string: &mut Stringで可変な参照を受け入れなければなりませんでした。

ところが、可変な参照には大きな制約が一つあります: 特定のスコープである特定のデータに対しては、一つしか可変な参照を持てないことです。こちらのコードは失敗します:

ファイル名: src/main.rs

```
let mut s = String::from("hello");
```

```
2
3 let r1 = &mut s;
4 let r2 = &mut s;
```

これがエラーです:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
(エラー: 一度に`s`を可変として2回以上借用することはできません)
 --> borrow_twice.rs:5:19
  |
4 |     let r1 = &mut s;
  |                   - first mutable borrow occurs here
  |                    (最初の可変な参照はここ)
5 |     let r2 = &mut s;
  |                   ^ second mutable borrow occurs here
  |                    (二つ目の可変な参照はここ)
6 | }
  | - first borrow ends here
  |   (最初の借用はここで終わり)
```

この制約は、可変化を許可するものの、それを非常に統制の取れた形で行えます。これは、新たな Rust 市民にとっては、壁です。なぜなら、多くの言語では、いつでも好きな時に可変化できるからです。

この制約がある利点は、コンパイラがコンパイル時にデータ競合を防ぐことができる点です。データ競合とは、競合条件と類似していて、これら 3 つの振る舞いが起きる時に発生します:

- 2 つ以上のポインタが同じデータに同時にアクセスする。
- 少なくとも一つのポインタがデータに書き込みを行っている。
- データへのアクセスを同期する機構が使用されていない。

データ競合は未定義の振る舞いを引き起こし、実行時に追いかけようとした時に特定し解決するのが難しい問題です。しかし、Rust は、データ競合が起こるコードをコンパイルさえしないので、この問題が発生しないようにしてくれるわけです。

いつものように、波かっこを使って新しいスコープを生成し、**同時並行**なものでなく、複数の可変な参照を作ることができます。

```
1 let mut s = String::from("hello");
2
3 {
4     let r1 = &mut s;
5
6 } // r1はここでスコープを抜けるので、問題なく新しい参照を作ることができる
7
8 let r2 = &mut s;
```

可変と不変な参照を組み合わせることに関しても、似たような規則が存在しています。このコードはエラーになります:

```
let mut s = String::from("hello");

let r1 = &s; // 問題なし
let r2 = &s; // 問題なし
let r3 = &mut s; // 大問題！
```

これがエラーです:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
(エラー: `s`は不変で借用されているので、可変で借用できません)
 --> borrow_thrice.rs:6:19
  |
4 |     let r1 = &s; // no problem
  |              - immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |                   ^ mutable borrow occurs here
7 | }
  | - immutable borrow ends here
```

ふう！ さらに不変な参照をしている間は、可変な参照をすることはできません。不変参照の使用者は、それ以降に値が突然変わることなんて予想してません！ しかしながら、複数の不変参照をすることは可能です。データを読み込んでいるだけの人に、他人がデータを読み込むことに対して影響を与える能力はないからです。

これらのエラーは、時としてイライラするものではありますが、Rust コンパイラがバグの可能性を早期に指摘してくれ (それも実行時ではなくコンパイル時に)、問題の発生箇所をズバリ示してくれるのだと覚えておいてください。そうして想定通りにデータが変わらない理由を追いかける必要がなくなります。

6.2.2　宙に浮いた参照

ポインタのある言語では、誤ってダングリングポインタを生成してしまいやすいです。ダングリングポインタとは、他人に渡されてしまった可能性のあるメモリを指すポインタのことであり、その箇所へのポインタを保持している間に、メモリを解放してしまうことで発生します。対照的に Rust では、コンパイラが、参照がダングリング参照に絶対ならないよう保証してくれます:つまり、何らかのデータへの参照があったら、コンパイラは参照がスコープを抜けるまで、データがスコープを抜けることがないよう確認してくれるわけです。

ダングリング参照作りを試してみますが、コンパイラはこれをコンパイルエラーで阻止します:

ファイル名: src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

```

```
fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

こちらがエラーです:

```
error[E0106]: missing lifetime specifier
(エラー: ライフタイム指定子がありません)
 --> dangle.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no
    value for it to be borrowed from
    (助言: この関数の戻り値型は、借用した値を含んでいますが、借用される値がどこにもありま
    せん)
  = help: consider giving it a 'static lifetime
  ('staticライフタイムを与えることを考慮してみてください)
```

このエラーメッセージは、まだ解説していない機能について触れています: **ライフタイム**です。ライフタイムについては第 10 章で詳しく議論しますが、ライフタイムに関する部分を無視すれば、このメッセージは、このコードが問題になる理由に関する鍵を握っています。

```
this function's return type contains a borrowed value, but there is no value
for it to be borrowed from.
```

dangleコードの各段階で一体何が起きているのかを詳しく見ていきましょう:

```
fn dangle() -> &String { // dangleはStringへの参照を返す

    let s = String::from("hello"); // sは新しいString

    &s // String sへの参照を返す
} // ここで、sはスコープを抜け、ドロップされる。そのメモリは吹き飛ばされる。
  // 危険だ
```

sは、dangle内で生成されているので、dangleのコードが終わったら、sは解放されてしまいますが、そこへの参照を返そうとしました。つまり、この参照は無効な Stringを指していると思われるのです。よくないことです！ コンパイラは、これを阻止してくれるのです。

ここでの解決策は、Stringを直接返すことです:

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

これは何の問題もなく動きます。所有権はムーブされ、何も解放されることはありません。

6.2.3　参照の規則

参照について議論したことを再確認しましょう:

- 任意のタイミングで、一つの可変参照か不変な参照いくつでものどちらかを行える
- 参照は常に有効でなければならない

次は、違う種類の参照を見ていきましょう: スライスです。

6.3　スライス型

所有権のない別のデータ型は、**スライス**です。スライスにより、コレクション全体というより、その内の一連の要素を参照することができます。

ここに小さなプログラミング問題があります: 文字列を受け取って、その文字列中の最初の単語を返す関数を書いてください。関数が文字列中に空白を見つけなかったら、文字列全体が一つの単語に違いないので、文字列全体が返されるべきです。

この関数のシグニチャについて考えてみましょう:

```
fn first_word(s: &String) -> ?
```

この関数、`first_word`は引数に`&String`をとります。所有権はいらないので、これで十分です。ですが、何を返すべきでしょうか？ 文字列の**一部**について語る方法が全くありません。しかし、単語の終端の番号を返すことができますね。リスト 4-7 に示したように、その方法を試してみましょう:

ファイル名: src/main.rs

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

リスト 4-7: `String`引数へのバイト数で表された番号を返す `first_word`関数

`String`の値を要素ごとに見て、空白かどうかを確かめる必要があるので、`as_bytes`メソッドを使って、`String`オブジェクトをバイト配列に変換しています。

```
let bytes = s.as_bytes();
```

次に、そのバイト配列に対して、`iter`メソッドを使用してイテレータを生成しています:

```
for (i, &item) in bytes.iter().enumerate() {
```

イテレータについて詳しくは、第 13 章で議論します。今は、iterは、コレクション内の各要素を返すメソッドであること、enumerateが iterの結果を包んで、代わりにタプルの一部として各要素を返すことを知っておいてください。enumerateから返ってくるタプルの第 1 要素は、番号であり、2 番目の要素は、(コレクションの) 要素への参照になります。これは、手動で番号を計算するよりも少しだけ便利です。

enumerateメソッドがタプルを返すので、Rust のあらゆる場所同様、パターンを使って、そのタプルを分解できます。従って、forループ内で、タプルの番号に対する iとタプルの 1 バイトに対応する &itemを含むパターンを指定しています。.iter().enumerate()から要素への参照を取得するので、パターンに&を使っています。

forループ内で、バイトリテラル表記を使用して空白を表すバイトを検索しています。空白が見つかったら、その位置を返します。それ以外の場合、s.len()を使って文字列の長さを返します。

```
    if item == b' ' {
        return i;
    }
}
s.len()
```

さて、文字列内の最初の単語の終端の番号を見つけ出せるようになりましたが、問題があります。usize型を単独で返していますが、これは&Stringの文脈でのみ意味を持つ数値です。言い換えると、Stringから切り離された値なので、将来的にも有効である保証がないのです。リスト 4-7 の first_word関数を使用するリスト 4-8 のプログラムを考えてください:

ファイル名: src/main.rs

```
# fn first_word(s: &String) -> usize {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return i;
#         }
#     }
#
#     s.len()
# }
#
fn main() {
    let mut s = String::from("hello world");

<<<<<<< HEAD
    let word = first_word(&s); // wordの中身は、値5になる。

    s.clear(); // Stringを空にする。つまり、""と等しくする。
=======
```

```
    let word = first_word(&s); // word will get the value 5

    s.clear(); // This empties the String, making it equal to ""
>>>>>>> fork_master_master

    // wordはまだ値5を保持しているが、もうこの値を有効に使用できる文字列は存在しない。
    // wordは完全に無効なのだ！
}
```

リスト 4-8: first_word関数の呼び出し結果を保持し、Stringの中身を変更する

このプログラムは何のエラーもなくコンパイルが通り、wordを s.clear()の呼び出し後に使用しても、コンパイルが通ります。wordは sの状態に全く関連づけられていないので、その中身はまだ値 5 のままです。その値 5を変数 sに使用し、最初の単語を取り出そうとすることはできますが、これはバグでしょう。というのも、sの中身は、5を wordに保存してから変わってしまったからです。

word内の番号が sに格納されたデータと同期されなくなるのを心配することは、面倒ですし間違いになりやすいです！ これらの番号を管理するのは、second_word関数を書いたら、さらに脆くなります。そのシグニチャは以下のようにならなければおかしいです:

```
fn second_word(s: &String) -> (usize, usize) {
```

今、私たちは開始と終端の番号を追うようになりました。特定の状態のデータから計算されたけど、その状態に全く紐付かない値が増えました。同期を取る必要のある宙に浮いた関連性のない変数が 3 つになってしまいました。

運のいいことに、Rust にはこの問題への解決策が用意されています: 文字列スライスです。

6.3.1　文字列スライス

文字列スライスとは、Stringの一部への参照で、こんな見た目をしています:

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

これは、String全体への参照を取ることに似ていますが、余計な [0..5]という部分が付いています。String全体への参照というよりも、Stringの一部への参照です。開始..終点という記法は、開始から始まり、終点未満までずっと続く範囲です。

[starting_index..ending_index]と指定することで、角かっこに範囲を使い、スライスを生成できます。ここで、starting_indexはスライスの最初の位置、ending_indexはスライスの終端位置よりも、1 大きくなります。内部的には、スライスデータ構造は、開始地点とスライスの長さを保持しており、スライスの長さは ending_indexから starting_indexを引いたものに対応します。以上より、let world = &s[6..11];の場合には、worldは sの 6 バイト目へのポインタと 5 という長さを保持するスライスになるでしょう。

図 4-6 は、これを図解しています。

図 4-6: Stringオブジェクトの一部を参照する文字列スライス

Rust の..という範囲記法で、最初の番号 (ゼロ) から始めたければ、2 連ピリオドの前に値を書かなければいいのです。換言すれば、これらは等価です:

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

同様の意味で、Stringの最後のバイトをスライスが含むのならば、末尾の数値を書かなければいいのです。つまり、これらは等価になります:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

さらに、両方の値を省略すると、文字列全体のスライスを得られます。故に、これらは等価です:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

> 注釈: 文字列スライスの範囲インデックスは、有効な UTF-8 文字境界に置かなければなりません。マルチバイト文字の真ん中で文字列スライスを生成しようとしたら、エラーでプログラムは落ちるでしょう。文字列スライスを導入する目的で、この節では ASCII のみを想定しています; UTF-8 に関する より徹底した議論は、第 8 章の「文字列」節で行います。

これら全ての情報を心に留めて、first_wordを書き直してスライスを返すようにしましょう。文字列スライスを意味する型は、&strと記述します:

ファイル名: src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
```

```
}
```

リスト 4-7 で取った手段と同じ方法で単語の終端番号を取得しています。つまり、最初の空白を探すことです。空白を発見したら、文字列の最初と、空白の番号を開始、終了地点として使用して文字列スライスを返しています。

これで、first_wordを呼び出すと、元のデータに紐付けられた単独の値を得られるようになりました。この値は、スライスの開始地点への参照とスライス中の要素数から構成されています。

second_word関数についても、スライスを返すことでうまくいくでしょう:

```
fn second_word(s: &String) -> &str {
```

これで、ずっと混乱しにくい素直な API になりました。なぜなら、Stringへの参照が有効なままであることをコンパイラが、保証してくれるからです。最初の単語の終端番号を得た時に、文字列を空っぽにして先ほどの番号が無効になってしまったリスト 4-8 のプログラムのバグを覚えていますか？ そのコードは、論理的に正しくないのですが、即座にエラーにはなりませんでした。問題は後になってから発生し、それは空の文字列に対して、最初の単語の番号を使用し続けようとした時でした。スライスならこんなバグはあり得ず、コードに問題があるなら、もっと迅速に判明します。スライスバージョンの first_wordを使用すると、コンパイルエラーが発生します:

ファイル名: src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // Error!
}
```

こちらがコンパイルエラーです:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
(エラー: 不変として借用されているので、`s`を可変で借用できません)
 --> src/main.rs:6:5
  |
4 |     let word = first_word(&s);
  |                           - immutable borrow occurs here (不変借用はここで起きています)
5 |
6 |     s.clear(); // Error!
  |     ^ mutable borrow occurs here (可変借用はここで起きています)
7 | }
  | - immutable borrow ends here (不変借用はここで終わっています)
```

借用規則から、何かへの不変な参照がある時、さらに可変な参照を得ることはできないことを思い出してください。clearが Stringを切り詰める必要があるので、可変な参照を得ようとして失敗して

いるわけです。Rust のおかげで API が使いやすくなるだけでなく、ある種のエラー全てを完全にコンパイル時に排除してくれるのです！

6.3.1.1　文字列リテラルはスライスである

文字列は、バイナリに埋め込まれると話したことを思い出してください。今やスライスのことを知ったので、文字列リテラルを正しく理解することができます。

```
let s = "Hello, world!";
```

ここでの sの型は、&strです: バイナリのその特定の位置を指すスライスです。これは、文字列が不変である理由にもなっています。要するに、&strは不変な参照なのです。

6.3.1.2　引数としての文字列スライス

リテラルや Stringのスライスを得ることができると知ると、first_wordに対して、もう一つ改善点を見出すことができます。シグニチャです:

```
fn first_word(s: &String) -> &str {
```

もっと経験を積んだ Rust 市民なら、代わりに以下のように書くでしょう。というのも、こうすると、同じ関数を Stringと&str両方に使えるようになるからです:

```
fn first_word(s: &str) -> &str {
```

もし、文字列スライスがあるなら、それを直接渡せます。Stringオブジェクトがあるなら、その String全体のスライスを渡せます。Stringへの参照の代わりに文字列スライスを取るよう関数を定義すると、何も機能を失うことなく API をより一般的で有益なものにできるのです。

Filename: src/main.rs

```
# fn first_word(s: &str) -> &str {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return &s[0..i];
#         }
#     }
#
#     &s[..]
# }
fn main() {
    let my_string = String::from("hello world");

    // first_wordは`String`のスライスに対して機能する
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";
```

```
19
20      // first_wordは文字列リテラルのスライスに対して機能する
21      let word = first_word(&my_string_literal[..]);
22
23      // 文字列リテラルは、すでに文字列スライス*な*ので、
24      // スライス記法なしでも機能するのだ！
25      let word = first_word(my_string_literal);
26  }
```

6.3.2　他のスライス

文字列リテラルは、想像通り、文字列に特化したものです。ですが、もっと一般的なスライス型も存在します。この配列を考えてください:

```
1  let a = [1, 2, 3, 4, 5];
```

文字列の一部を参照したくなる可能性があるのと同様、配列の一部を参照したくなる可能性もあります。以下のようにすれば、参照することができます:

```
1  let a = [1, 2, 3, 4, 5];
2
3  let slice = &a[1..3];
```

このスライスは、`&[i32]`という型になります。これも文字列スライスと全く同じように動作します。つまり、最初の要素への参照と長さを保持することです。他のすべての種類のコレクションに対して、この種のスライスは使用することができるでしょう。これらのコレクションについて詳しくは、第 8 章でベクタ型について話すときに議論します。

6.4　まとめ

所有権、借用、スライスの概念は、コンパイル時に Rust プログラムにおいて、メモリ安全性を保証します。Rust 言語も他のシステムプログラミング言語と同じように、メモリの使用法について制御させてくれるわけですが、所有者がスコープを抜けたときにデータの所有者に自動的にデータを片付けさせることは、この制御を得るために、余計なコードを書いてデバッグする必要がないことを意味します。

所有権は、Rust の他のいろんな部分が動作する方法に影響を与えるので、これ以降もこれらの概念についてさらに語っていく予定です。第 5 章に移って、`struct`でデータをグループ化することについて見ていきましょう。

第 7 章

構造体を使用して関係のあるデータを構造化する

struct または、**構造体**は、意味のあるグループを形成する複数の関連した値をまとめ、名前付けできる独自のデータ型です。オブジェクト指向言語に造詣が深いなら、*struct* はオブジェクトのデータ属性みたいなものです。この章では、タプルと構造体を対照的に比較し、構造体の使用法をデモし、メソッドや関連関数を定義して、構造体のデータに紐付く振る舞いを指定する方法について議論します。構造体と *enum*(第 6 章で議論します) は、自分のプログラム領域で新しい型を定義し、Rust のコンパイル時型精査機能をフル活用する構成要素になります。

7.1 構造体を定義し、インスタンス化する

構造体は第 3 章で議論したタプルと似ています。タプル同様、構造体の一部を異なる型にできます。一方タプルとは違って、各データ片には名前をつけるので、値の意味が明確になります。この名前のおかげで、構造体はタプルに比して、より柔軟になるわけです: データの順番に頼って、インスタンスの値を指定したり、アクセスしたりする必要がないのです。

構造体の定義は、`struct`キーワードを入れ、構造体全体に名前を付けます。構造体名は、一つにグループ化されるデータ片の意義を表すものであるべきです。そして、波かっこ内に、データ片の名前と型を定義し、これは**フィールド**と呼ばれます。例えば、リスト 5-1 では、ユーザアカウントに関する情報を保持する構造体を示しています:

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

リスト 5-1: User構造体定義

構造体を定義した後に使用するには、各フィールドに対して具体的な値を指定して構造体のインスタンスを生成します。インスタンスは、構造体名を記述し、key: valueペアを含む波かっこを付け加えることで生成します。ここで、キーはフィールド名、値はそのフィールドに格納したいデータになります。フィールドは、構造体で宣言した通りの順番に指定する必要はありません。換言すると、構造体定義とは、型に対する一般的な雛形のようなものであり、インスタンスは、その雛形を特定のデータで埋め、その型の値を生成するわけです。例えば、リスト 5-2 で示されたように特定のユーザを宣言することができます。

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

リスト 5-2: User構造体のインスタンスを生成する

構造体から特定の値を得るには、ドット記法が使えます。このユーザの E メールアドレスだけが欲しいなら、この値を使いたかった場所全部で user1.emailが使えます。インスタンスが可変であれば、ドット記法を使い特定のフィールドに代入することで値を変更できます。リスト 5-3 では、可変な Userインスタンスの emailフィールド値を変更する方法を示しています:

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@example.com");
```

リスト 5-3: ある Userインスタンスの emailフィールド値を変更する

インスタンス全体が可変でなければならないことに注意してください; Rust では、一部のフィール

ドのみを可変にすることはできないのです。

また、あらゆる式同様、構造体の新規インスタンスを関数本体の最後の式として生成して、そのインスタンスを返すことを暗示できます。リスト 5-4 は、与えられた email と username で User インスタンスを生成する build_user関数を示しています。activeフィールドには true値が入り、sign_in_countには値 1が入ります。

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

リスト 5-4: E メールとユーザ名を取り、Userインスタンスを返す build_user関数

構造体のフィールドと同じ名前を関数の引数にもつけることは筋が通っていますが、emailとusernameというフィールド名と変数を繰り返さなきゃいけないのは、ちょっと面倒です。構造体にもっとフィールドがあれば、名前を繰り返すことはさらに煩わしくなるでしょう。幸運なことに、便利な省略記法があります！

7.1.1　フィールドと変数が同名の時にフィールド初期化省略記法を使う

仮引数名と構造体のフィールド名がリスト 5-4 では、全く一緒なので、**フィールド初期化省略**記法を使って build_userを書き換えても、振る舞いは全く同じにしつつ、リスト 5-5 に示したようにemailと usernameを繰り返さなくてもよくなります。

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
```

```
}
```

リスト 5-5: emailと username引数が構造体のフィールドと同名なので、フィールド初期化省略法を使用する build_user関数

ここで、emailというフィールドを持つ User構造体の新規インスタンスを生成しています。email フィールドを build_user関数の email引数の値にセットしたいわけです。emailフィールドと email 引数は同じ名前なので、email: emailと書くよりも、emailと書くだけで済むのです。

7.1.2　構造体更新記法で他のインスタンスからインスタンスを生成する

多くは前のインスタンスの値を使用しつつ、変更する箇所もある形で新しいインスタンスを生成できるとしばしば有用です。**構造体更新記法**でそうすることができます。

まず、リスト 5-6 では、更新記法なしで user2に新しい Userインスタンスを生成する方法を示しています。emailと usernameには新しい値をセットしていますが、それ以外にはリスト 5-2 で生成した user1の値を使用しています:

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
# let user1 = User {
#     email: String::from("someone@example.com"),
#     username: String::from("someusername123"),
#     active: true,
#     sign_in_count: 1,
# };
#
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    active: user1.active,
    sign_in_count: user1.sign_in_count,
};
```

リスト 5-6: user1の一部の値を使用しつつ、新しい Userインスタンスを生成する

構造体更新記法を使用すると、リスト 5-7 に示したように、コード量を減らしつつ、同じ効果を達成できます。..という記法により、明示的にセットされていない残りのフィールドが、与えられたインスタンスのフィールドと同じ値になるように指定します。

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
```

```
#     active: bool,
# }
#
# let user1 = User {
#     email: String::from("someone@example.com"),
#     username: String::from("someusername123"),
#     active: true,
#     sign_in_count: 1,
# };
#
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

リスト 5-7: 構造体更新記法を使用して、新しい Userインスタンス用の値に新しい emailと username をセットしつつ、残りの値は、user1変数のフィールド値を使う

リスト 5-7 のコードも、emailと usernameについては異なる値、activeと sign_in_countフィールドについては、user1と同じ値になるインスタンスを user2に生成します。

7.1.3　異なる型を生成する名前付きフィールドのないタプル構造体

構造体名により追加の意味を含むものの、フィールドに紐づけられた名前がなく、むしろフィールドの型だけの**タプル構造体**と呼ばれる、タプルに似た構造体を定義することもできます。タプル構造体は、構造体名が提供する追加の意味は含むものの、フィールドに紐付けられた名前はありません; むしろ、フィールドの型だけが存在します。タプル構造体は、タプル全体に名前をつけ、そのタプルを他のタプルとは異なる型にしたい場合に有用ですが、普通の構造体のように各フィールド名を与えるのは、冗長、または余計になるでしょう。

タプル構造体を定義するには、structキーワードの後に構造体名、さらにタプルに含まれる型を続けます。例えば、こちらは、Colorと Pointという 2 種類のタプル構造体の定義と使用法です:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

blackと originの値は、違う型であることに注目してください。これらは、異なるタプル構造体のインスタンスだからですね。定義された各構造体は、構造体内のフィールドが同じ型であっても、それ自身が独自の型になります。例えば、Color型を引数に取る関数は、Pointを引数に取ることはできません。たとえ、両者の型が、3 つの i32値からできているにもかかわらずです。それ以外については、タプル構造体のインスタンスは、タプルと同じように振る舞います: 分解して個々の部品にしたり、.と番号を使用して個々の値にアクセスするなどです。

7.1.4　フィールドのないユニット様 (よう) 構造体

また、一切フィールドのない構造体を定義することもできます！ これらは、()、ユニット型と似たような振る舞いをすることから、**ユニット様構造体**と呼ばれます。ユニット様構造体は、ある型にトレイトを実装するけれども、型自体に保持させるデータは一切ない場面に有効になります。トレイトについては第 10 章で議論します。

7.1.5　構造体データの所有権

リスト 5-1 の `User`構造体定義において、`&str`文字列スライス型ではなく、所有権のある `String` 型を使用しました。これは意図的な選択です。というのも、この構造体のインスタンスには全データを所有してもらう必要があり、このデータは、構造体全体が有効な間はずっと有効である必要があるのです。

構造体に、他の何かに所有されたデータへの参照を保持させることもできますが、そうするには**ライフタイム**という第 10 章で議論する Rust の機能を使用しなければなりません。ライフタイムのおかげで構造体に参照されたデータが、構造体自体が有効な間、ずっと有効であることを保証してくれるのです。ライフタイムを指定せずに構造体に参照を保持させようとしたとしましょう。以下の通りですが、これは動きません:

ファイル名: src/main.rs

```
struct User {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

コンパイラは、ライフタイム指定子が必要だと怒るでしょう:

```
error[E0106]: missing lifetime specifier
(エラー: ライフタイム指定子がありません)
 -->
  |
2 |     username: &str,
  |               ^ expected lifetime parameter
```

```
                    (ライフタイム引数を予期しました)

error[E0106]: missing lifetime specifier
 -->
  |
3 |     email: &str,
  |            ^ expected lifetime parameter
```

第 10 章で、これらのエラーを解消して構造体に参照を保持する方法について議論しますが、当面、今回のようなエラーは、`&str`のような参照の代わりに、`String`のような所有された型を使うことで解消します。

7.2　構造体を使ったプログラム例

構造体を使用したくなる可能性のあるケースを理解するために、四角形の面積を求めるプログラムを書きましょう。単一の変数から始め、代わりに構造体を使うようにプログラムをリファクタリングします。

Cargo で *rectangles* という新規バイナリプロジェクトを作成しましょう。このプロジェクトは、四角形の幅と高さをピクセルで指定し、その面積を求めます。リスト 5-8 に、プロジェクトの *src/main.rs* で、正にそうする一例を短いプログラムとして示しました:

ファイル名: src/main.rs

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    // 四角形の面積は、{}平方ピクセルです
    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

リスト 5-8: 個別の幅と高さ変数を指定して四角形の面積を求める

では、`cargo run`でこのプログラムを走らせてください:

```
The area of the rectangle is 1500 square pixels.
(四角形の面積は、1500平方ピクセルです)
```

7.2.1　タプルでリファクタリングする

リスト 5-8 のコードはうまく動き、各次元で area関数を呼び出すことで四角形の面積を割り出しますが、改善点があります。幅と高さは、組み合わせると一つの四角形を表すので、相互に関係があるわけです。

このコードの問題点は、areaのシグニチャから明らかです:

```
1 fn area(width: u32, height: u32) -> u32 {
```

area関数は、1 四角形の面積を求めるものと考えられますが、今書いた関数には、引数が 2 つあります。引数は関連性があるのに、このプログラム内のどこにもそのことは表現されていません。幅と高さを一緒にグループ化する方が、より読みやすく、扱いやすくなるでしょう。それをする一つの方法については、第 3 章の「タプル型」節ですでに議論しました: タプルを使うのです。

7.2.2　タプルでリファクタリングする

リスト 5-9 は、タプルを使う別バージョンのプログラムを示しています:

ファイル名: src/main.rs

```
1  fn main() {
2      let rect1 = (30, 50);
3
4      println!(
5          "The area of the rectangle is {} square pixels.",
6          area(rect1)
7      );
8  }
9
10 fn area(dimensions: (u32, u32)) -> u32 {
11     dimensions.0 * dimensions.1
12 }
```

リスト 5-9: タプルで四角形の幅と高さを指定する

ある意味では、このプログラムはマシです。タプルのおかげで少し構造的になり、一引数を渡すだけになりました。しかし別の意味では、このバージョンは明確性を失っています: タプルは要素に名前を付けないので、計算が不明瞭になったのです。なぜなら、タプルの一部に添え字アクセスする必要があるからです。

面積計算で幅と高さを混在させるのなら問題はないのですが、四角形を画面に描画したいとなると、問題になるのです！ タプルの添え字 0が幅で、添え字 1が高さであることを肝に命じておかなければなりません。他人がこのコードをいじることになったら、このことを割り出し、同様に肝に命じなければならないでしょう。容易く、このことを忘れたり、これらの値を混ぜこぜにしたりしてエラーを発生させてしまうでしょう。データの意味をコードに載せていないからです。

7.2.3　構造体でリファクタリングする: より意味付けする

データにラベル付けをして意味付けを行い、構造体を使います。現在使用しているタプルを全体と一部に名前のあるデータ型に、変形することができます。そう、リスト 5-10 に示したように:

ファイル名: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

リスト 5-10: Rectangle構造体を定義する

ここでは、構造体を定義し、Rectangleという名前にしています。波括弧の中で widthと heightというフィールドを定義し、u32という型にしました。それから main内で Rectangleの特定のインスタンスを生成し、幅を 30、高さを 50 にしました。

これで area関数は引数が一つになり、この引数は名前が rectangle、型は Rectangle構造体インスタンスへの不変借用になりました。第 4 章で触れたように、構造体の所有権を奪うよりも借用する必要があります。こうすることで mainは所有権を保って、rect1を使用し続けることができ、そのために関数シグニチャと関数呼び出し時に&を使っているわけです。

area関数は、Rectangleインスタンスの widthと heightフィールドにアクセスしています。これで、areaの関数シグニチャは、我々の意図をズバリ示すようになりました: widthと heightフィールドを使って、Rectangleの面積を計算します。これにより、幅と高さが相互に関係していることが伝わり、タプルの 0や 1という添え字を使うのではなく、値に説明的な名前を与えられるのです。簡潔性を勝ち取ったわけですね。

7.2.4　トレイトの継承で有用な機能を追加する

プログラムのデバッグをし、フィールドの値を調べている間に Rectangleのインスタンスを出力できると、素晴らしいわけです。リスト 5-11 では、以前の章のように、println!マクロを試しに使用しようとしていますが、動きません:

ファイル名: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    // rect1は{}です
    println!("rect1 is {}", rect1);
}
```

リスト 5-11: Rectangleのインスタンスを出力しようとする

このコードを走らせると、こんな感じのエラーが出ます:

```
error[E0277]: the trait bound `Rectangle: std::fmt::Display` is not satisfied
(エラー: トレイト境界`Rectangle: std::fmt::Display`が満たされていません)
```

println!マクロには、様々な整形があり、標準では、波括弧は Displayとして知られる整形をするよう、println!に指示するのです: 直接エンドユーザ向けの出力です。これまでに見てきた基本型は、標準で Displayを実装しています。というのも、1や他の基本型をユーザに見せる方法は一つしかないからです。しかし構造体では、println!が出力を整形する方法は自明ではなくなります。出力方法がいくつもあるからです: カンマは必要なの？ 波かっこを出力する必要はある？ 全フィールドが見えるべき？ この曖昧性のため、Rust は必要なものを推測しようとせず、構造体には Display実装が提供されないのです。

エラーを読み下すと、こんな有益な注意書きがあります:

```
`Rectangle` cannot be formatted with the default formatter; try using
`:?` instead if you are using a format string
(注釈: `Rectangle`は、デフォルト整形機では、整形できません; フォーマット文字列を使うのなら
代わりに`:?`を試してみてください)
```

試してみましょう！ pritnln!マクロ呼び出しは、println!("rect1 is {:?}", rect1);という見た目になるでしょう。波括弧内に:?という指定子を書くと、println!に Debugと呼ばれる出力整形を使いたいと指示するのです。Debugとは、開発者にとって有用な方法で構造体を出力させてくれるトレイトなので、コードをデバッグしている最中に、値を確認することができます。

変更してコードを走らせてください。なに！ まだエラーが出ます:

```
error[E0277]: the trait bound `Rectangle: std::fmt::Debug` is not satisfied
(エラー: トレイト境界`Rectangle: std::fmt::Debug`は満たされていません)
```

しかし今回も、コンパイラは有益な注意書きを残してくれています:

```
`Rectangle` cannot be formatted using `:?`; if it is defined in your
crate, add `#[derive(Debug)]` or manually implement it
```

```
(注釈: `Rectangle`は`:?`を使って整形できません; 自分のクレートで定義しているのなら
`#[derive(Debug)]`を追加するか、手動で実装してください)
```

確かに Rust にはデバッグ用の情報を出力する機能が備わっていますが、この機能を構造体で使えるようにするには、明示的な選択をしなければならないのです。そうするには、構造体定義の直前に`#[derive(Debug)]`という注釈を追加します。そう、リスト 5-12 で示されている通りです:

ファイル名: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!("rect1 is {:?}", rect1);
}
```

リスト 5-12: `Debug`トレイトを継承する注釈を追加し、`Rectangle`インスタンスをデバッグ用整形機で出力する

これでプログラムを実行すれば、エラーは出ず、以下のような出力が得られるでしょう:

```
rect1 is Rectangle { width: 30, height: 50 }
```

素晴らしい！ 最善の出力ではないものの、このインスタンスの全フィールドの値を出力しているので、デバッグ中には間違いなく役に立つでしょう。より大きな構造体があるなら、もう少し読みやすい出力の方が有用です; そのような場合には、`println!`文字列中の`{:?}`の代わりに`{:#?}`を使うことができます。この例で`{:#?}`というスタイルを使用したら、出力は以下のようになるでしょう:

```
rect1 is Rectangle {
    width: 30,
    height: 50
}
```

«««< HEAD

Rust には、`derive`注釈で使えるトレイトが多く提供されており、独自の型に有用な振る舞いを追加することができます。そのようなトレイトとその振る舞いは、おまけ C、「継承可能トレイト」で一覧になっています。これらのトレイトを独自の動作とともに実装する方法だけでなく、独自のトレイトを生成する方法については、第 10 章で解説します。

`area`関数は、非常に特殊です: 四角形の面積を算出するだけです。`Rectangle`構造体とこの動作をより緊密に結び付けられると、役に立つでしょう。なぜなら、他のどんな型でもうまく動作しなくなるからです。`area`関数を`Rectangle`型に定義された`area`**メソッド**に変形することで、このコードをリファクタリングし続けられる方法について見ていきましょう。======= Rust has provided a number

of traits for us to use with the `derive` annotation that can add useful behavior to our custom types. Those traits and their behaviors are listed in Appendix C, “Derivable Traits.” We’ll cover how to implement these traits with custom behavior as well as how to create your own traits in Chapter 10.

Our `area` function is very specific: it only computes the area of rectangles. It would be helpful to tie this behavior more closely to our `Rectangle` struct, because it won’t work with any other type. Let’s look at how we can continue to refactor this code by turning the `area` function into an `area` *method* defined on our `Rectangle` type.

fork*master*master

7.3　メソッド記法

メソッドは関数に似ています: `fn`キーワードと名前で宣言されるし、引数と返り値があるし、どこか別の場所で呼び出された時に実行されるコードを含みます。ところが、メソッドは構造体の文脈 (あるいは enum かトレイトオブジェクトの。これらについては各々第 6 章と 17 章で解説します) で定義されるという点で、関数とは異なり、最初の引数は必ず `self`になり、これはメソッドが呼び出されている構造体インスタンスを表します。

7.3.1　メソッドを定義する

`Rectangle`インスタンスを引数に取る `area`関数を変え、代わりに `Rectangle`構造体上に `area`メソッドを作りましょう。リスト 5-13 に示した通りですね:

ファイル名: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
```

```
20 }
```

««««< HEAD

リスト 5-13: Rectangle構造体上に areaメソッドを定義する

Rectangleの文脈内で関数を定義するには、impl(implementation; 実装) ブロックを始めます。それから area関数を implの波かっこ内に移動させ、最初の (今回は唯一の) 引数をシグニチャ内と本体内全てで selfに変えます。area関数を呼び出し、rect1を引数として渡す mainでは、代替としてメソッド記法を使用して、Rectangleインスタンスの areaメソッドを呼び出せます。メソッド記法は、インスタンスの後に続きます: ドット、メソッド名、かっこ、そして引数と続くわけです。

areaのシグニチャでは、rectangle: &Rectangleの代わりに&selfを使用しています。というのも、コンパイラは、このメソッドが impl Rectangleという文脈内に存在するために、selfの型が Rectangleであると把握しているからです。&Rectangleと同様に、selfの直前に&を使用していることに注意してください。メソッドは、selfの所有権を奪ったり、ここでしているように不変で selfを借用したり、可変で selfを借用したりできるのです。他の引数と全く同じですね。

ここで&selfを選んでいるのは、関数バージョンで&Rectangleを使用していたのと同様の理由です: 所有権はいらず、構造体のデータを読み込みたいだけで、書き込む必要はないわけです。メソッドの一部でメソッドを呼び出したインスタンスを変更したかったら、第 1 引数に&mut selfを使用するでしょう。selfだけを第 1 引数にしてインスタンスの所有権を奪うメソッドを定義することは稀です; このテクニックは、通常メソッドが selfを何か別のものに変形し、変形後に呼び出し元が元のインスタンスを使用できないようにしたい場合に使用されます。

関数の代替としてメソッドを使う主な利点は、メソッド記法を使用して全メソッドのシグニチャで selfの型を繰り返す必要がなくなる以外だと、体系化です。コードの将来的な利用者に Rectangleの機能を提供しているライブラリ内の各所でその機能を探させるのではなく、この型のインスタンスでできることを一つの implブロックにまとめあげています。

7.3.2　->演算子はどこに行ったの？

C と C++ では、メソッド呼び出しには 2 種類の異なる演算子が使用されます: オブジェクトに対して直接メソッドを呼び出すのなら、.を使用するし、オブジェクトのポインタに対してメソッドを呼び出し、先にポインタを参照外しする必要があるなら、->を使用するわけです。言い換えると、objectがポインタなら、object->something()は、(⋆object).something()と同等なのです。

Rust には->演算子の代わりとなるようなものはありません; その代わり、Rust には、**自動参照および参照外し**という機能があります。Rust においてメソッド呼び出しは、この動作が行われる数少ない箇所なのです。

動作方法はこうです: object.something()とメソッドを呼び出すと、コンパイラは objectがメソッドのシグニチャと合致するように、自動で&か&mut、⋆を付与するのです。要するに、以

下のコードは同じものです:

```
# #[derive(Debug,Copy,Clone)]
# struct Point {
#     x: f64,
#     y: f64,
# }
#
# impl Point {
#    fn distance(&self, other: &Point) -> f64 {
#        let x_squared = f64::powi(other.x - self.x, 2);
#        let y_squared = f64::powi(other.y - self.y, 2);
#
#        f64::sqrt(x_squared + y_squared)
#    }
# }
# let p1 = Point { x: 0.0, y: 0.0 };
# let p2 = Point { x: 5.0, y: 6.5 };
p1.distance(&p2);
(&p1).distance(&p2);
```

前者の方がずっと明確です。メソッドには自明な受け手 (selfの型) がいるので、この自動参照機能は動作するのです。受け手とメソッド名が与えられれば、コンパイラは確実にメソッドが読み込み専用 (&self) か、書き込みもする (&mut self) のか、所有権を奪う (self) のか判断できるわけです。Rust において、メソッドの受け手に関して借用が明示されないという事実は、所有権を現実世界でプログラマフレンドリーにさせる大部分を占めているのです。

7.3.3　より引数の多いメソッド

Rectangle構造体に 2 番目のメソッドを実装して、メソッドを使う鍛錬をしましょう。今回は、Rectangleのインスタンスに、別の Rectangleのインスタンスを取らせ、2 番目の Rectangleが self に完全にはめ込まれたら、trueを返すようにしたいのです; そうでなければ、falseを返すべきです。つまり、一旦 can_holdメソッドを定義したら、リスト 5-14 のようなプログラムを書けるようになりたいのです:

ファイル名: src/main.rs

```
fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 10, height: 40 };
    let rect3 = Rectangle { width: 60, height: 45 };

    // rect1にrect2ははまり込む？
    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

リスト 5-14: 未完成の can_holdを使用する

そして、予期される出力は以下のようになります。なぜなら、rect2の各次元は rect1よりも小さいものの、rect3は rect1より幅が広いからです:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

メソッドを定義したいことはわかっているので、impl Rectangleブロック内での話になります。メソッド名は、can_holdになり、引数として別の Rectangleを不変借用で取るでしょう。メソッドを呼び出すコードを見れば、引数の型が何になるかわかります: rect1.can_hold(&rect2)は、&rect2、Rectangleのインスタンスである rect2への不変借用を渡しています。これは道理が通っています。なぜなら、rect2を読み込む (書き込みではなく。この場合、可変借用が必要になります) だけでよく、can_holdメソッドを呼び出した後にも rect2が使えるよう、所有権を mainに残したままにしたいからです。can_holdの返り値は、論理型になり、メソッドの中身は、selfの幅と高さがもう一つの Rectangleの幅と高さよりも、それぞれ大きいことを確認します。リスト 5-13 の implブロックに新しい can_holdメソッドを追記しましょう。リスト 5-15 に示した通りです:

ファイル名: src/main.rs

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

リスト 5-15: 別の Rectangleのインスタンスを引数として取る can_holdメソッドを、Rectangleに実装する

このコードをリスト 5-14 の main関数と合わせて実行すると、望み通りの出力が得られます。メソッドは、self引数の後にシグニチャに追加した引数を複数取ることができ、その引数は、関数の引数と同様に動作するのです。

7.3.4　関連関数

implブロックの別の有益な機能は、implブロック内に selfを引数に取らない関数を定義できることです。これは、構造体に関連付けられているので、**関連関数**と呼ばれます。それでも、関連関数は関数であり、メソッドではありません。というのも、対象となる構造体のインスタンスが存在しないからです。もう String::fromという関連関数を使用したことがありますね。

関連関数は、構造体の新規インスタンスを返すコンストラクタによく使用されます。例えば、一次元の引数を取り、長さと幅両方に使用する関連関数を提供することができ、その結果、同じ値を 2 回指定する必要なく、正方形の Rectangleを生成しやすくすることができます。

ファイル名: src/main.rs

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { width: size, height: size }
    }
}
```

この関連関数を呼び出すために、構造体名と一緒に::記法を使用します; 一例は let sq = Rectangle::square(3);です。この関数は、構造体によって名前空間分けされています: ::という記法は、関連関数とモジュールによって作り出される名前空間両方に使用されます。モジュールについては第 7 章で議論します。

7.3.5　複数の impl ブロック

各構造体には、複数の implブロックを存在させることができます。例えば、リスト 5-15 はリスト 5-16 に示したコードと等価で、リスト 5-16 では、各メソッドごとに implブロックを用意しています:

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

リスト 5-16: 複数の implブロックを使用してリスト 5-15 を書き直す

ここでこれらのメソッドを個々の implブロックに分ける理由はないのですが、有効な書き方です。複数の implブロックが有用になるケースは第 10 章で見ますが、そこではジェネリクスのある型と、

トレイトについて議論します。

7.4　まとめ

構造体により、自分の領域で意味のある独自の型を作成することができます。構造体を使用することで、関連のあるデータ片を相互に結合させたままにし、各部品に名前を付け、コードを明確にすることができます。メソッドにより、構造体のインスタンスが行う動作を指定することができ、関連関数により、構造体に特有の機能をインスタンスを利用することなく、名前空間分けすることができます。

しかし、構造体だけが独自の型を作成する手段ではありません: Rust の enum 機能に目を向けて、別の道具を道具箱に追加しましょう。

第 8 章

Enum とパターンマッチング

この章では、**列挙型**について見ていきます。列挙型は、*enum* とも称されます。enum は、取りうる値を列挙することで、型を定義させてくれます。最初に、enum を定義し、使用して、enum がデータとともに意味をコード化する方法を示します。次に、特別に有用な enum である `Option`について掘り下げていきましょう。この型は、値が何かかなんでもないかを表現します。それから、`match`式のパターンマッチングにより、どう enum の色々な値に対して異なるコードを走らせやすくなるかを見ます。最後に、`if let`文法要素も、如何 (いか) に enum をコードで扱う際に使用可能な便利で簡潔な慣用句であるかを解説します。

enum は多くの言語に存在する機能ですが、その能力は言語ごとに異なります。Rust の enum は、F#、OCaml、Haskell などの、関数型言語に存在する**数学的データ型**に最も酷似しています。

8.1 Enum を定義する

コードで表現したくなるかもしれない場面に目を向けて、enum が有効でこの場合、構造体よりも適切である理由を確認しましょう。IP アドレスを扱う必要が出たとしましょう。現在、IP アドレスの規格は二つあります: バージョン 4 とバージョン 6 です。これらは、プログラムが遭遇する IP アドレスのすべての可能性です: 列挙型は、取りうる値をすべて**列挙**でき、これが列挙型の名前の由来です。

どんな IP アドレスも、バージョン 4 かバージョン 6 のどちらかになりますが、同時に両方にはなり得ません。IP アドレスのその特性により enum データ構造が適切なものになります。というのも、enum の値は、そのバリアントのいずれか一つにしかなり得ないからです。バージョン 4 とバージョン 6 のアドレスは、どちらも根源的には IP アドレスですから、コードがいかなる種類の IP アドレスにも適用される場面を扱う際には、同じ型として扱われるべきです。

この概念をコードでは、`IpAddrKind`列挙型を定義し、IP アドレスがなりうる種類、`V4`と `V6`を列挙することで、表現できます。これらは、enum の**列挙子**として知られています:

```
1 enum IpAddrKind {
2     V4,
```

```
    V6,
}
```

これで、IpAddrKindはコードの他の場所で使用できる独自のデータ型になります。

8.1.1　Enum の値

以下のようにして、IpAddrKindの各バリアントのインスタンスは生成できます:

```
# enum IpAddrKind {
#     V4,
#     V6,
# }
#
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

enum の列挙子は、その識別子の元に名前空間分けされていることと、2 連コロンを使ってその二つを区別していることに注意してください。これが有効な理由は、こうすることで、値 IpAddrKind::V4 と IpAddrKind::V6という値は両方とも、同じ型 IpAddrKindになったからです。そうしたら、例えば、どんな IpAddrKindを取る関数も定義できるようになります。

```
# enum IpAddrKind {
#     V4,
#     V6,
# }
#
fn route(ip_type: IpAddrKind) { }
```

そして、この関数をどちらのバリアントに対しても呼び出せます:

```
# enum IpAddrKind {
#     V4,
#     V6,
# }
#
# fn route(ip_type: IpAddrKind) { }
#
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

enum の利用には、さらなる利点さえもあります。この IP アドレス型についてもっと考えてみると、現状では、実際の IP アドレスのデータを保持する方法がありません。つまり、どんな種類であるかを知っているだけです。構造体について第 5 章で学んだばっかりとすると、この問題に対しては、リスト 6-1 のように対処するかもしれません。

```
enum IpAddrKind {
    V4,
```

```
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

リスト 6-1: IP アドレスのデータと IpAddrKindのバリアントを structを使って保持する

ここでは、二つのフィールドを持つ IpAddrという構造体を定義しています: IpAddrKind型 (先ほど定義した enum ですね) の kindフィールドと、String型の addressフィールドです。この構造体のインスタンスが 2 つあります。最初のインスタンス、homeには kindとして IpAddrKind::V4があり、紐付けられたアドレスデータは 127.0.0.1です。2 番目のインスタンス、loopbackには、kindの値として、IpAddrKindのもう一つの列挙子、V6があり、アドレス::1が紐付いています。構造体を使って kindと address値を一緒に包んだので、もう列挙子は値と紐付けられています。

各 enum の列挙子に直接データを格納して、enum を構造体内に使うというよりも enum だけを使って、同じ概念をもっと簡潔な方法で表現することができます。この新しい IpAddrの定義は、V4と V6列挙子両方に String値が紐付けられていることを述べています。

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

enum の各列挙子にデータを直接添付できるので、余計な構造体を作る必要は全くありません。

構造体よりも enum を使うことには、別の利点もあります: 各列挙子に紐付けるデータの型と量は、異なってもいいのです。バージョン 4 の IP アドレスには、常に 0 から 255 の値を持つ 4 つの数値があります。V4のアドレスは、4 つの u8型の値として、格納するけれども、V6のアドレスは引き続き、単独の String型の値で格納したかったとしても、構造体では不可能です。enum なら、こんな場合も容易に対応できます:

```
enum IpAddr {
```

```
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

バージョン 4 とバージョン 6 の IP アドレスを格納するデータ構造を定義する複数の異なる方法を示してきました。しかしながら、蓋を開けてみれば、IP アドレスを格納してその種類をコード化したくなるということは一般的なので、標準ライブラリに使用可能な定義があります！ 標準ライブラリでの `IpAddr`の定義のされ方を見てみましょう: 私たちが定義し、使用したのと全く同じ enum と列挙子がありますが、アドレスデータを二種の異なる構造体の形で列挙子に埋め込み、この構造体は各列挙子用に異なる形で定義されています。

```
struct Ipv4Addr {
<<<<<<< HEAD
    // 省略
}

struct Ipv6Addr {
    // 省略
=======
    // --snip--
}

struct Ipv6Addr {
    // --snip--
>>>>>>> fork_master_master
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

このコードは、enum 列挙子内にいかなる種類のデータでも格納できることを描き出しています: 例を挙げれば、文字列、数値型、構造体などです。他の enum を含むことさえできます！ また、標準ライブラリの型は、あなたが思い付いたよりも複雑ではないことがしばしばあります。

標準ライブラリに `IpAddr`に対する定義は含まれるものの、標準ライブラリの定義をスコープに導入していないので、まだ、干渉することなく自分自身の定義を生成して使用できることに注意してください。型をスコープに導入することについては、第 7 章でもっと詳しく言及します。

リスト 6-2 で enum の別の例を見てみましょう: 今回のコードは、幅広い種類の型が列挙子に埋め込まれています:

```
enum Message {
    Quit,
```

```
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

リスト 6-2: 列挙子各々が異なる型と量の値を格納する Messageenum

この enum には、異なる型の列挙子が 4 つあります:

- Quitには紐付けられたデータは全くなし。
- Moveは、中に匿名構造体を含む。
- Writeは、単独の Stringオブジェクトを含む。
- ChangeColorは、3 つの i32値を含む。

リスト 6-2 のような列挙子を含む enum を定義することは、enum の場合、structキーワードを使わず、全部の列挙子が Message型の元に分類される点を除いて、異なる種類の構造体定義を定義するのと類似しています。以下の構造体も、先ほどの enum の列挙子が保持しているのと同じデータを格納することができるでしょう:

```
struct QuitMessage; // ユニット構造体
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // タプル構造体
struct ChangeColorMessage(i32, i32, i32); // タプル構造体
```

ですが、異なる構造体を使っていたら、各々、それ自身の型があるので、単独の型になるリスト 6-2 で定義した Messageenum ほど、これらの種のメッセージいずれもとる関数を簡単に定義することはできないでしょう。

enum と構造体にはもう 1 点似通っているところがあります: implを使って構造体にメソッドを定義できるのと全く同様に、enum にもメソッドを定義することができるのです。こちらは、Messageenum 上に定義できる callという名前のメソッドです:

```
# enum Message {
#     Quit,
#     Move { x: i32, y: i32 },
#     Write(String),
#     ChangeColor(i32, i32, i32),
# }
#
impl Message {
    fn call(&self) {
        // method body would be defined here
        // メソッド本体はここに定義される
    }
}

```

```
let m = Message::Write(String::from("hello"));
m.call();
```

メソッドの本体では、selfを使用して、メソッドを呼び出した相手の値を取得できるでしょう。この例では、Message::Write(String::from("hello"))という値を持つ、変数 mを生成したので、これが m.call()を走らせた時に、callメソッドの本体内で selfが表す値になります。

非常に一般的で有用な別の標準ライブラリの enum を見てみましょう: Optionです。

8.1.2　Optionenum と Null 値に勝る利点

前節で、IpAddrenum が Rust の型システムを使用して、プログラムにデータ以上の情報をコード化できる方法を目撃しました。この節では、Optionのケーススタディを掘り下げていきます。この型も標準ライブラリにより定義されている enum です。この Option型はいろんな箇所で使用されます。なぜなら、値が何かかそうでないかという非常に一般的な筋書きをコード化するからです。この概念を型システムの観点で表現することは、コンパイラが、プログラマが処理すべき場面全てを処理していることをチェックできることを意味します; この機能は、他の言語において、究極的にありふれたバグを阻止することができます。

プログラミング言語のデザインは、しばしばどの機能を入れるかという観点で考えられるが、除いた機能も重要なのです。Rust には、他の多くの言語にはある null 機能がありません。*null* とはそこに何も値がないことを意味する値です。null のある言語において、変数は常に二者択一どちらかの状態になります: null かそうでないかです。

null の開発者である Tony Hoare の 2009 年のプレゼンテーション、”Null References: The Billion Dollar Mistake” では、こんなことが語られています。

> 私はそれを 10 億ドルの失敗と呼んでいます。その頃、私は、オブジェクト指向言語の参照に対する、最初のわかりやすい型システムを設計していました。私の目標は、どんな参照の使用も全て完全に安全であるべきことを、コンパイラにそのチェックを自動で行ってもらって保証することだったのです。しかし、null 参照を入れるという誘惑に打ち勝つことができませんでした。それは、単純に実装が非常に容易だったからです。これが無数のエラーや脆弱性、システムクラッシュにつながり、過去 40 年で 10 億ドルの苦痛や損害を引き起こしたであろうということなのです。

null 値の問題は、null の値を null でない値のように使用しようとしたら、何らかの種類のエラーが出ることです。この null かそうでないかという特性は広く存在するので、この種の間違いを大変犯しやすいのです。

しかしながら、null が表現しようとしている概念は、それでも役に立つものです: null は、何らかの理由で現在無効、または存在しない値のことなのです。

問題は、全く概念にあるのではなく、特定の実装にあるのです。そんな感じなので、Rust には null がありませんが、値が存在するか不在かという概念をコード化する enum ならあります。この enum

が Option<T>で、以下のように標準ライブラリに定義されています。

```
enum Option<T> {
    Some(T),
    None,
}
```

Option<T>は有益すぎて、初期化処理 (prelude) にさえ含まれています。つまり、明示的にスコープに導入する必要がないのです。さらに、列挙子もそうなっています: Someと Noneを Option::の接頭辞なしに直接使えるわけです。ただ、Option<T>はそうは言っても、普通の enum であり、Some(T)と Noneも Option<T>型のただの列挙子です。

<T>という記法は、まだ語っていない Rust の機能です。これは、ジェネリック型引数であり、ジェネリクスについて詳しくは、第 10 章で解説します。とりあえず、知っておく必要があることは、<T>は、Optionenum の Some列挙子が、あらゆる型のデータを 1 つだけ持つことができることを意味していることだけです。こちらは、Option値を使って、数値型や文字列型を保持する例です。

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

Someではなく、Noneを使ったら、コンパイラに Option<T>の型が何になるかを教えなければいけません。というのも、None値を見ただけでは、Some列挙子が保持する型をコンパイラが推論できないからです。

Some値がある時、値が存在するとわかり、その値は、Someに保持されています。None値がある場合、ある意味、null と同じことを意図します: 有効な値がないのです。では、なぜ Option<T>の方が、null よりも少しでも好ましいのでしょうか？

簡潔に述べると、Option<T>と T(ここで Tはどんな型でもいい) は異なる型なので、コンパイラが Option<T>値を確実に有効な値かのようには使用させてくれません。例えば、このコードは i8を Option<i8>に足そうとしているので、コンパイルできません。

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

このコードを動かしたら、以下のようなエラーメッセージが出ます。

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option<i8>>` is
not satisfied
(エラー: `i8: std::ops::Add<std::option::Option<i8>>`というトレイト境界が満たされていません)
 -->
  |
5 |     let sum = x + y;
  |                   ^ no implementation for `i8 + std::option::Option<i8>`
```

```
|
```

なんて強烈な！ 実際に、このエラーメッセージは、`i8`と `Option<i8>`が異なる型なので、足し合わせる方法がコンパイラにはわからないことを意味します。Rust において、`i8`のような型の値がある場合、コンパイラが常に有効な値であることを確認してくれます。この値を使う前に null であることをチェックする必要なく、自信を持って先に進むことができるのです。`Option<i8>`がある時 (あるいはどんな型を扱おうとしていても) のみ、値を保持していない可能性を心配する必要はないわけであり、コンパイラはプログラマが値を使用する前にそのような場面を扱っているか確かめてくれます。

言い換えると、`T`型の処理を行う前には、`Option<T>`を `T`に変換する必要があるわけです。一般的に、これにより、null の最もありふれた問題の一つを捕捉する一助になります: 実際には null なのに、そうでないかのように想定することです。

不正確に null でない値を想定する心配をしなくてもよいということは、コード内でより自信を持てることになります。null になる可能性のある値を保持するには、その値の型を `Option<T>`にすることで明示的に同意しなければなりません。それからその値を使用する際には、値が null である場合を明示的に処理する必要があります。値が `Option<T>`以外の型であるとこ全てにおいて、値が null でないと安全に想定することができます。これは、Rust にとって、意図的な設計上の決定であり、null の普遍性を制限し、Rust コードの安全性を向上させます。

では、`Option<T>`型の値がある時、その値を使えるようにするには、どのように `Some`列挙子から `T`型の値を取り出せばいいのでしょうか？ `Option<T>`には様々な場面で有効に活用できる非常に多くのメソッドが用意されています; ドキュメントでそれらを確認できます。`Option<T>`のメソッドに馴染むと、Rust の旅が極めて有益になるでしょう。

In general, in order to use an `Option<T>` value, you want to have code that will handle each variant. You want some code that will run only when you have a `Some(T)` value, and this code is allowed to use the inner `T`. You want some other code to run if you have a `None` value, and that code doesn't have a `T` value available. The `match` expression is a control flow construct that does just this when used with enums: it will run different code depending on which variant of the enum it has, and that code can use the data inside the matching value.

一般的に、`Option<T>`値を使うには、各列挙子を処理するコードが欲しくなります。`Some(T)`値がある時だけ走る何らかのコードが欲しくなり、このコードが内部の `T`を使用できます。`None`値があった場合に走る別のコードが欲しくなり、そちらのコードは `T`値は使用できない状態になります。`match`式が、enum とともに使用した時にこれだけの動作をするフロー制御文法要素になります: enum の列挙子によって、違うコードが走り、そのコードがマッチした値の中のデータを使用できるのです。

8.2　match フロー制御演算子

Rust には、一連のパターンに対して値を比較し、マッチしたパターンに応じてコードを実行させてくれる `match`と呼ばれる、非常に強力なフロー制御演算子があります。パターンは、リテラル値、変

数名、ワイルドカードやその他多数のもので構成することができます; 第 18 章で、全ての種類のパターンと、その目的については解説します。matchのパワーは、パターンの表現力とコンパイラが全てのありうるパターンを処理しているかを確認してくれるという事実に由来します。

match式をコイン並べ替え装置のようなものと考えてください: コインは、様々なサイズの穴が空いた通路を流れ落ち、各コインは、サイズのあった最初の穴に落ちます。同様に、値は matchの各パターンを通り抜け、値が「適合する」最初のパターンで、値は紐付けられたコードブロックに落ち、実行中に使用されるわけです。

コインについて話したので、それを matchを使用する例にとってみましょう！ 数え上げ装置と同じ要領で未知のアメリカコインを一枚取り、どの種類のコインなのか決定し、その価値をセントで返す関数をリスト 6-3 で示したように記述することができます:

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

リスト 6-3: enum とその enum の列挙子をパターンにした match式

value_in_cents関数内の matchを噛み砕きましょう。まず、matchキーワードに続けて式を並べています。この式は今回の場合、値 coinです。ifで使用した式と非常に酷似しているみたいですね。しかし、大きな違いがあります: ifでは、式は論理値を返す必要がありますが、ここでは、どんな型でも構いません。この例における coinの型は、1 行目で定義した Coinenum です。

次は、matchアームです。一本のアームには 2 つの部品があります: パターンと何らかのコードです。今回の最初のアームは Coin::Pennyという値のパターンであり、パターンと動作するコードを区別する=>演算子が続きます。この場合のコードは、ただの値 1です。各アームは次のアームとカンマで区切られています。

この match式が実行されると、結果の値を各アームのパターンと順番に比較します。パターンに値がマッチしたら、そのコードに紐付けられたコードが実行されます。パターンが値にマッチしなければ、コイン並べ替え装置と全く同じように、次のアームが継続して実行されます。必要なだけパターンは存在できます: リスト 6-3 では、matchには 4 本のアームがあります。

各アームに紐付けられるコードは式であり、マッチしたアームの式の結果が match式全体の戻り値になります。

典型的に、アームのコードが短い場合、波かっこは使用されません。リスト 6-3 では、各アームが値を返すだけなので、これに倣っています。マッチのアームで複数行のコードを走らせたいのなら、波かっこを使用することができます。例えば、以下のコードは、メソッドが `Coin::Penny`とともに呼び出されるたびに「ラッキーなペニー」と表示しつつ、ブロックの最後の値、`1`を返すでしょう。

```
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter,
# }
#
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

8.2.1　値に束縛されるパターン

マッチのアームの別の有益な機能は、パターンにマッチした値の一部に束縛できる点です。こうして、enum の列挙子から値を取り出すことができます。

例として、enum の列挙子の一つを中にデータを保持するように変えましょう。1999 年から 2008 年まで、アメリカは、片側に 50 の州それぞれで異なるデザインをしたクォーターコインを鋳造していました。他のコインは州のデザインがなされることはなかったので、クォーターだけがこのおまけの値を保持します。`Quarter`列挙子を変更して、`UsState`値が中に保持されるようにすることで enum にこの情報を追加でき、それをしたのがリスト 6-4 のコードになります:

```
#[derive(Debug)] // すぐに州を点検できるように
enum UsState {
    Alabama,
    Alaska,
    // ... などなど
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

リスト 6-4: Quarter列挙子が UsStateの値も保持する Coinenum

友人の一人が 50 州全部のクォーターコインを収集しようとしているところを想像しましょう。コインの種類で緩い変更を並べ替える間、各クォーターに関連した州の名前を叫ぶことになるので、友人が持っていない種類だったら、コレクションに追加することができます。

このコードの match 式では、Coin::Quarter列挙子の値にマッチする stateという名の変数をパターンに追加します。Coin::Quarterがマッチすると、state変数はそのクォーターの state の値に束縛されます。それから、stateをそのアームのコードで使用できます。以下のようにですね:

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
# }
#
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter(UsState),
# }
#
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        },
    }
}
```

value_in_cents(Coin::Quarter(UsState::Alaska))と呼び出すつもりだったなら、coinは Coin::Quarter(UsState::Alaska)になります。その値を match の各アームと比較すると、Coin::Quater(state)に到達するまで、どれにもマッチしません。その時に、stateに束縛されるのは、UsState::Alaskaという値です。そして、println!式でその束縛を使用することができ、そのため、Coinenum の列挙子から Quarterに対する中身の state の値を取得できたわけです。

8.2.2　Option<T>とのマッチ

前節では、Option<T>を使用する際に、Someケースから中身の Tの値を取得したくなりました。要するに、Coinenum に対して行ったように、matchを使って Option<T>を扱うこともできるというわけです！ コインを比較する代わりに、Option<T>の列挙子を比較するのですが、match式の動作の仕方は同じままです。

Option<i32>を取る関数を書きたくなったとし、中に値があったら、その値に 1 を足すことにしま

しょう。中に値がなければ、関数は None値を返し、何も処理を試みるべきではありません。

matchのおかげで、この関数は大変書きやすく、リスト 6-5 のような見た目になります:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

リスト 6-5: Option<i32>に match式を使う関数

plus_oneの最初の実行についてもっと詳しく検証しましょう。plus_one(five)と呼び出した時、plus_oneの本体の変数 xは Some(5)になります。そして、これをマッチの各アームと比較します。

```
None => None,
```

Some(5)という値は、Noneというパターンにはマッチしませんので、次のアームに処理が移ります。

```
Some(i) => Some(i + 1),
```

Some(5)は Some(i)にマッチしますか？ えっと、します！ 列挙子が同じです。iは Someに含まれる値に束縛されるので、iは値 5になります。それから、このマッチのアームのコードが実行されるので、iの値に 1 を足し、合計の 6を中身にした新しい Some値を生成します。

さて、xが Noneになるリスト 6-5 の 2 回目の plus_pneの呼び出しを考えましょう。matchに入り、最初のアームと比較します。

```
None => None,
```

マッチします！ 足し算する値がないので、プログラムは停止し、=>の右辺にある None値が返ります。最初のアームがマッチしたため、他のアームは比較されません。

matchと enum の組み合わせは、多くの場面で有効です。Rust コードにおいて、このパターンはよく見かけるでしょう: enum に対し matchし、内部のデータに変数を束縛させ、それに基づいたコードを実行します。最初はちょっと巧妙ですが、一旦慣れてしまえば、全ての言語にあってほしいと願うことになるでしょう。一貫してユーザのお気に入りなのです。

8.2.3　マッチは包括的

もう一つ議論する必要のある matchの観点があります。1 点バグがありコンパイルできないこんなバージョンの plus_one関数を考えてください:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

Noneの場合を扱っていないため、このコードはバグを生みます。幸い、コンパイラが捕捉できるバグです。このコードのコンパイルを試みると、こんなエラーが出ます:

```
error[E0004]: non-exhaustive patterns: `None` not covered
(エラー: 包括的でないパターン: `None`がカバーされてません)
 -->
  |
6 |         match x {
  |               ^ pattern `None` not covered
```

全可能性を網羅していないことをコンパイラは検知しています。もっと言えば、どのパターンを忘れているかさえ知っているのです。Rust におけるマッチは、**包括的**です: 全てのあらゆる可能性を網羅し尽くさなければ、コードは有効にならないのです。特に Option<T>の場合には、コンパイラが明示的に Noneの場合を扱うのを忘れないようにする時、null になるかもしれない値があることを想定しないように、故に、前に議論した 10 億ドルの失敗を犯さないよう、保護してくれるわけです。

8.2.4　_というプレースホルダー

Rust には、全ての可能性を列挙したくない時に使用できるパターンもあります。例えば、u8は、有効な値として、0 から 255 までを取ります。1、3、5、7 の値にだけ興味があったら、0、2、4、6、8、9 と 255 までの数値を列挙する必要に迫られたくはないです。幸運なことに、する必要はありません: 代わりに特別なパターンの_を使用できます:

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

_というパターンは、どんな値にもマッチします。他のアームの後に記述することで、_は、それまでに指定されていない全ての可能性にマッチします。()は、ただのユニット値なので、_の場合には、何も起こりません。結果として、_プレースホルダーの前に列挙していない可能性全てに対しては、何もしたくないと言えるわけです。

ですが、一つのケースにしか興味がないような場面では、match式はちょっと長ったらしすぎます。このような場面用に、Rust には、if letが用意されています。

8.3　if let で簡潔なフロー制御

if let記法で ifと letをより冗長性の少ない方法で組み合わせ、残りを無視しつつ、一つのパターンにマッチする値を扱うことができます。Option<u8>にマッチするけれど、値が 3 の時にだけコードを実行したい、リスト 6-6 のプログラムを考えてください:

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

リスト 6-6: 値が Some(3)の時だけコードを実行する match

Some(3)にマッチした時だけ何かをし、他の Some<u8>値や None値の時には何もしたくありません。match式を満たすためには、列挙子を一つだけ処理した後に_ => ()を追加しなければなりません。これでは、追加すべき定型コードが多すぎます。

その代わり、if letを使用してもっと短く書くことができます。以下のコードは、リスト 6-6 の matchと全く同じように振る舞います:

```
# let some_u8_value = Some(0u8);
if let Some(3) = some_u8_value {
    println!("three");
}
```

if letという記法は=で区切られたパターンと式を取り、式が matchに与えられ、パターンが最初のアームになった matchと、同じ動作をします。

if letを使うと、タイプ数が減り、インデントも少なくなり、定型コードも減ります。しかしながら、matchでは強制された包括性チェックを失ってしまいます。matchか if letかの選択は、特定の場面でどんなことをしたいかと簡潔性を得ることが包括性チェックを失うのに適切な代償となるかによります。

言い換えると、if letは値が一つのパターンにマッチした時にコードを走らせ、他は無視する match への糖衣構文と考えることができます。

if letでは、elseを含むこともできます。elseに入るコードブロックは、if letと elseに等価な match式の_の場合に入るコードブロックと同じになります。リスト 6-4 の Coinenum 定義を思い出してください。ここでは、Quarter列挙子は、UsStateの値も保持していましたね。クォーターコインの状態を告げつつ、見かけたクォーター以外のコインの枚数を数えたいなら、以下のように match式で実現することができるでしょう:

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
```

```
# }
#
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter(UsState),
# }
# let coin = Coin::Penny;
let mut count = 0;
match coin {
    // {:?}州のクォーターコイン
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

または、以下のように `if let`と `else`を使うこともできるでしょう:

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
# }
#
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter(UsState),
# }
# let coin = Coin::Penny;
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

`match`を使って表現するには冗長的すぎるロジックがプログラムにあるようなシチュエーションに遭遇したら、`if let`も Rust 道具箱にあることを思い出してください。

8.4　まとめ

これで、enum を使用してワンセットの列挙された値のどれかになりうる独自の型を生成する方法を解説しました。標準ライブラリの `Option<T>`が型システムを使用して、エラーを回避する際に役立つ方法についても示しました。enum の値がデータを内部に含む場合、処理すべきケースの数に応じて、`match`か `if let`を使用して値を取り出し、使用できます。

もう Rust プログラムで構造体と enum を使用して、自分の領域の概念を表現できます。API 内で

使用するために独自の型を生成することで、型安全性を保証することができます: コンパイラが、各関数の予期する型の値のみを関数が得ることを確かめてくれるのです。

使用するのに率直な整理整頓された API をユーザに提供し、ユーザが必要とするものだけを公開するために、今度は、Rust のモジュールに目を向けてみましょう。

第 9 章

モジュールを使用してコードを体系化し、再利用する

Rust でのプログラミングをし始めた頃は、コードは全て `main`関数内に収まったかもしれません。コードが肥大化するにつれ、最終的に機能を別の関数に移して再利用性とまとまりを高めるでしょう。コードを細切りにすることで、個々のコード片をそれだけで理解しやすくします。しかし、あまりにも多くの関数があったらどうなるでしょうか？ Rust には、コードの再利用を体系化された形で行うことのできるモジュールシステムが組み込まれています。

コードを関数に抽出するのと同様に、関数 (や他のコード、構造体や enum など) を異なるモジュールに抽出することができます。**モジュール**とは、関数や型定義を含む名前空間のことで、それらの定義がモジュール外からも見えるようにするか (public) 否か (private) は、選択することができます。以下が、モジュールの動作法の概要です:

- `mod`キーワードで新規モジュールを宣言します。モジュール内のコードは、この宣言の直後の波かっこ内か、別のファイルに存在します。
- 標準では、関数、型、定数、モジュールは非公開です。`pub`キーワードで要素は公開され、名前空間の外からも見えるようになります。
- `use`キーワードでモジュールやモジュール内の定義をスコープに入れることができるので、参照するのが楽になります。

この各部品を見て、それらが全体にどうはまり込むかを理解します。

9.1 `mod` とファイルシステム

モジュールの例を Cargo で新規プロジェクトを生成することから始めるが、バイナリクレートの代わりに、ライブラリクレートを作成します: 他人が依存として自分のプロジェクトに引き込めるプロ

ジェクトです。例を挙げると、第 2 章で議論した randクレートは、数当てゲームプロジェクトで依存に使用したライブラリクレートです。

何らかの一般的なネットワーク機能を提供するライブラリの骨格を作成します; モジュールと関数の体系化に集中し、関数の本体にどんなコードが入るかについては気にかけません。このライブラリを communicatorと呼びましょう。ライブラリを生成するために、--binの代わりに--libオプションを渡してください:

```
$ cargo new communicator --lib
$ cd communicator
```

Cargo が *src/main.rs* の代わりに *src/lib.rs* を生成したことに注目してください。*src/lib.rs* には、以下のような記述があります:

ファイル名: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Cargo は、--binオプションを使った時に得られる”Hello, world!” バイナリではなく、空のテストを生成して、ライブラリの事始めをしてくれました。#[]と mod testsという記法については、この章の後ほど、「superを使用して親モジュールにアクセスする」節で見ますが、今のところは、このコードを *src/lib.rs* の最後に残しておきましょう。

src/main.rs ファイルがないので、cargo runコマンドで Cargo が実行できるものは何もないわけです。従って、cargo buildコマンドを使用してライブラリクレートのコードをコンパイルします。

コードの意図によって、いろんなシチュエーションで最適になるライブラリコードを体系化する別のオプションをお目にかけます。

9.1.1　モジュール定義

communicatorネットワークライブラリについて、まずは connectという関数定義を含む networkという名前のモジュールを定義します。Rust において、モジュール定義は全て、modキーワードから開始します。このコードを *src/lib.rs* ファイルの頭、テストコードの上に追記してください。

ファイル名: src/lib.rs

```
mod network {
    fn connect() {
    }
}
```

modキーワードに続いて、モジュール名の network、さらに一連のコードを波かっこ内に記述します。このブロック内に存在するものは全て、networkという名前空間に属します。今回の場合、connectという単独の関数があります。この関数を networkモジュール外のスクリプトから呼び出したい場合、モジュールを指定し、以下のように名前空間記法の::を使用する必要があるでしょう: network::connect()。

同じ *src/lib.rs* ファイル内に複数のモジュールを並べることもできます。例として、connectという関数を含む clientモジュールも用意するには、リスト 7-1 に示したように追記すればいいわけです:

ファイル名: src/lib.rs

```
mod network {
    fn connect() {
    }
}

mod client {
    fn connect() {
    }
}
```

リスト 7-1: *src/lib.rs* に並べて定義された networkモジュールと clientモジュール

これで、network::connect関数と client::connect関数が用意できました。これらは全く異なる機能を有する可能性があり、異なるモジュールに存在するので、関数名がお互いに衝突することはありません。

今回の場合、ライブラリを構成しているので、ライブラリビルド時にエントリーポイントとなるファイルは、*src/lib.rs* になります。しかし、モジュールを作成するという点に関しては、*src/lib.rs* には何も特別なことはありません。ライブラリクレートに対して *src/lib.rs* にモジュールを生成するのと全く同様に、バイナリクレートに対して *src/main.rs* にモジュールを生成することもできます。実は、モジュール内にモジュールを書くこともでき、モジュールが肥大化するにつれて、関連のある機能を一緒くたにし、機能を切り離すのに有用なのです。コードを体系化すると選択する方法は、コードの部分部分の関連性に対する考え方によって選択することになります。例ですが、clientコードとその connect関数は、リスト 7-2 のように、代わりに network名前空間内に存在したら、ライブラリの使用者にとって意味のあるものになるかもしれません。

ファイル名: src/lib.rs

```
mod network {
    fn connect() {
    }

    mod client {
        fn connect() {
        }
    }
}
```

リスト 7-2: clientモジュールを networkモジュール内に移動させる

src/lib.rs ファイル内で、すでにある mod networkと mod clientの定義をリスト 7-2 のものと置き換えると、clientモジュールは networkの内部モジュールになるわけです。関数、network::connect と network::client::connectはどちらも connectという名前ですが、異なる名前空間にあるので、互いに干渉することはありません。

このように、モジュールは階層構造を形成します。*src/lib.rs* の中身が頂点に立ち、サブモジュールが子供になるわけです。リスト 7-1 の例を階層構造という観点で見たときの構造は、以下のような感じになります:

```
communicator
  ├── network
  └── client
```

さらに、リスト 7-2 の例に対応する階層構造は、以下の通りです:

```
communicator
  └── network
      └── client
```

この階層構造は、リスト 7-2 において、clientモジュールは networkモジュールの兄弟というよりも、子供になっていることを示しています。より複雑なプロジェクトなら、たくさんのモジュールが存在し、把握するのに論理的に体系化しておく必要があるでしょう。プロジェクト内で「論理的」とは、あなた次第であり、ライブラリ作成者と使用者がプロジェクトの領域についてどう考えるか次第でもあるわけです。こちらで示したテクニックを使用して、並列したモジュールや、ネストしたモジュールなど、どんな構造のモジュールでも、作成してください。

9.1.2　モジュールを別ファイルに移す

モジュールは階層構造をなす……コンピュータにおいて、もっと見慣れた構造に似ていませんか: そう、ファイルシステムです！ Rust のモジュールシステムを複数のファイルで使用して、プロジェクトを分割するので、全部が *src/lib.rs* や *src/main.rs* に存在することにはならなくなります。これの例として、リスト 7-3 のようなコードから始めましょう:

ファイル名: src/lib.rs

```
mod client {
    fn connect() {
    }
}

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
```

```
    }
}
```

リスト 7-3: 全て *src/lib.rs* に定義された三つのモジュール、client、network、network::server

src/lib.rs ファイルのモジュール階層は、こうなっています:

```
communicator
 ├── client
 └── network
     └── server
```

これらのモジュールが多数の関数を含み、その関数が長ったらしくなってきたら、このファイルをスクロールして、弄りたいコードを探すのが困難になるでしょう。関数が一つ以上の mod ブロックにネストされているので、関数の中身となるコードも長ったらしくなってしまうのです。これだけで、client、network、serverモジュールを *src/lib.rs* から分け、単独のファイルに配置するには十分でしょう。

最初に、clientモジュールのコードを clientモジュールの宣言だけに置き換えましょう。すると、*src/lib.rs* はリスト 7-4 のコードのようになります:

ファイル名: src/lib.rs

```
mod client;

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

リスト 7-4: clientモジュールの中身を抽出するが、宣言は *src/lib.rs* に残したまま

一応、clientモジュールをここで**宣言**していますが、ブロックをセミコロンで置換したことで、clientモジュールのスコープのコードは別の場所を探すようにコンパイラに指示しているわけです。言い換えると、mod client;の行は、以下のような意味になります:

```
mod client {
    // contents of client.rs
}
```

さて、このモジュール名の外部ファイルを作成する必要が出てきました。*src/*ディレクトリ内に *client.rs* ファイルを作成し、開いてください。それから以下のように入力してください。前段階で削除した clientモジュールの connect関数です:

ファイル名: src/client.rs

```
fn connect() {
}
```

このファイルには、mod宣言が必要ないことに着目してください。なぜなら、*src/lib.rs* に modを使って、もう clientモジュールを宣言しているからです。このファイルは、clientモジュールの**中身**を提供するだけなのです。ここにも mod clientを記述したら、clientに clientという名前のサブモジュールを与えることになってしまいます！

コンパイラは、標準で *src/lib.rs* だけを検索します。プロジェクトにもっとファイルを追加したかったら、*src/lib.rs* で他のファイルも検索するよう、コンパイラに指示する必要があるのです; このため、mod clientを *src/lib.rs* に定義し、*src/client.rs* には定義できなかったのです。

これでプロジェクトは問題なくコンパイルできるはずです。まあ、警告がいくつか出るんですが。cargo runではなく、cargo buildを使うことを忘れないでください。バイナリクレートではなく、ライブラリクレートだからですね:

```
$ cargo build
   Compiling communicator v0.1.0 (file:///projects/communicator)

warning: function is never used: `connect`
(警告: 関数は使用されていません: `connect`)
 --> src/client.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default

warning: function is never used: `connect`
 --> src/lib.rs:4:5
  |
4 | /     fn connect() {
5 | |     }
  | |_____^

warning: function is never used: `connect`
 --> src/lib.rs:8:9
  |
8 | /         fn connect() {
9 | |         }
  | |_________^
```

これらの警告は、全く使用されていない関数があると忠告してくれています。今は、警告を危惧する必要はありません; この章の後ほど、「pubで公開するか制御する」節で特定します。嬉しいことにただの警告です; プロジェクトはビルドに成功しました！

次に、同様のパターンを使用して networkモジュールも単独のファイルに抽出しましょう。*src/lib.rs* で、networkモジュールの本体を削除し、宣言にセミコロンを付加してください。こんな感じです:

ファイル名: src/lib.rs

```
mod client;

mod network;
```

それから新しい *src/network.rs* ファイルを作成して、以下のように入力してください:

ファイル名: src/network.rs

```
fn connect() {
}

mod server {
    fn connect() {
    }
}
```

このモジュールファイル内にもまだ `mod`宣言があることに気付いてください; `server`はまだ `network` のサブモジュールにしたいからです。

再度 `cargo build`してください。成功！ 抽出すべきモジュールがもう 1 個あります: `server`です。これはサブモジュール (つまり、モジュール内のモジュール) なので、モジュール名に倣ったファイルにモジュールを抽出するという今の手法は、通用しません。いずれにしても、エラーが確認できるように、試してみましょう。まず、*src/network.rs* ファイルを `server`モジュールの中身を含む代わりに、`mod server;`となるように変更してください。

ファイル名: src/network.rs

```
fn connect() {
}

mod server;
```

そして、*src/server.rs* ファイルを作成し、抽出した `server`モジュールの中身を入力してください:

ファイル名: src/server.rs

```
fn connect() {
}
```

`cargo build`を試すと、リスト 7-4 に示したようなエラーが出ます:

```
$ cargo build
   Compiling communicator v0.1.0 (file:///projects/communicator)
error: cannot declare a new module at this location
(エラー: この箇所では新規モジュールを宣言できません)
 --> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
  |
```

```
note: maybe move this module `src/network.rs` to its own directory via `src/network/mod
    .rs`
(注釈: もしかして、`src/network.rs`というこのモジュールを`src/network/mod.rs`経由で独自
    のディレクトリに移すの)
 --> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
note: ... or maybe `use` the module `server` instead of possibly redeclaring it
(注釈: それとも、再度宣言する可能性はなく、`server`というモジュールを`use`したの)
 --> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
```

リスト 7-5: serverサブモジュールを *src/server.rs* に抽出しようとしたときのエラー

エラーは、この箇所では新規モジュールを宣言できませんと忠告し、*src/network.rs* の `mod server;`行を指し示しています。故に、*src/network.rs* は、*src/lib.rs* と何かしら違うのです: 理由を知るために読み進めましょう。

リスト 7-5 の真ん中の注釈は、非常に有用です。というのも、まだ話題にしていないことを指摘しているからです。

```
note: maybe move this module `network` to its own directory via
`network/mod.rs`
```

以前行ったファイル命名パターンに従い続けるのではなく、注釈が提言していることをすることができます:

1. 親モジュール名である *network* という名前の新規**ディレクトリ**を作成する。
2. *src/network.rs* ファイルを *network* ディレクトリに移し、*src/network/mod.rs* と名前を変える。
3. サブモジュールファイルの *src/server.rs* を *network* ディレクトリに移す。

以下が、これを実行するコマンドです:

```
$ mkdir src/network
$ mv src/network.rs src/network/mod.rs
$ mv src/server.rs src/network
```

`cargo build`を走らせたら、ようやくコンパイルは通ります (まだ警告はありますけどね)。それでも、モジュールの配置は、リスト 7-3 で *src/lib.rs* に全てのコードを収めていたときと全く同じになります:

```
communicator
  ├── client
  └── network
      └── server
```

対応するファイルの配置は、以下のようになっています:

```
└── src
    ├── client.rs
    ├── lib.rs
    └── network
        ├── mod.rs
        └── server.rs
```

では、network::serverモジュールを抽出したかったときに、なぜ、*src/network.rs* ファイルを *src/network/mod.rs* ファイルに変更し、network::serverのコードを *network* ディレクトリ内の *src/network/server.rs* に置かなければならなかったのでしょうか？ なぜ、単に network::serverモジュールを *src/server.rs* に抽出できなかったのでしょうか？ 理由は、*server.rs* ファイルが *src* ディレクトリにあると、コンパイラが、serverは networkのサブモジュールと考えられることを検知できないからです。ここでのコンパイラの動作をはっきりさせるために、以下のようなモジュール階層をもつ別の例を考えましょう。こちらでは、定義は全て *src/lib.rs* に存在します。

```
communicator
 ├── client
 └── network
     └── client
```

この例でも、モジュールは 3 つあります: client、network、network::clientです。以前と同じ段階を経てモジュールをファイルに抽出すると、clientモジュール用に *src/client.rs* を作成することになるでしょう。networkモジュールに関しては、*src/network.rs* を作成します。しかし、network::client モジュールを *src/client.rs* ファイルに抽出することはできません。もうトップ階層の clientモジュールとして存在するからです！ clientと network::client**双方**のコードを *src/client.rs* ファイルに書くことができたら、コンパイラは、コードが client用なのか、network::client用なのか知る術を失ってしまいます。

従って、networkモジュールの network::clientサブモジュールをファイルに抽出するには、*src/network.rs* ファイルではなく、networkモジュールのディレクトリを作成する必要があったわけです。そうすれば、networkモジュールのコードは、*src/network/mod.rs* ファイルに移ることになり、network::clientというサブモジュールは専用の src/network/client.rsファイルを持てるわけです。これで、頂点にある *src/client.rs* は間違いなく、clientモジュールに属するコードになるわけです。

9.1.3　モジュールファイルシステムの規則

ファイルに関するモジュール規則をまとめましょう:

- fooという名前のモジュールにサブモジュールがなければ、fooの定義は、*foo.rs* というファイルに書くべきです。
- fooというモジュールに本当にサブモジュールがあったら、fooの定義は、*foo/mod.rs* というファイルに書くべきです。

これらのルールは再帰的に適用されるので、fooというモジュールに barというサブモジュールがあり、barにはサブモジュールがなければ、*src* ディレクトリには以下のようなファイルが存在するはずです:

```
├── foo
│   ├── bar.rs (contains the declarations in `foo::bar`)
│   │          (`foo::bar`内の定義を含む)
│   └── mod.rs (contains the declarations in `foo`, including `mod bar`)
               (`mod bar`を含む、`foo`内の定義を含む)
```

モジュールは、親モジュールのファイル内で modキーワードを使って宣言されるべきなのです。

次は、pubキーワードについて話し、警告を取り除きます！

9.2　pub で公開するか制御する

リスト 7-5 に示したエラーメッセージを networkと network::serverのコードを、*src/network/mod.rs* と *src/network/server.rs* ファイルにそれぞれ移動することで解決しました。その時点で cargo build はプロジェクトをビルドできましたが、client::connectと network::connectと network::server::connect関数が、使用されていないという警告メッセージが出ていました:

```
warning: function is never used: `connect`
 --> src/client.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default

warning: function is never used: `connect`
 --> src/network/mod.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^

warning: function is never used: `connect`
 --> src/network/server.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
```

では、何故このような警告を受けているのでしょうか？ 結局のところ、必ずしも自分のプロジェクト内ではなく、**利用者**に利用されることを想定した関数を含むライブラリを構成しているので、これらの connect関数が使用されていかないということは、問題になるはずはありません。これらの関数を生成することの要点は、自分ではなく、他のプロジェクトで使用することにあるのです。

このプログラムがこのような警告を引き起こす理由を理解するために、外部から connectライブラ

リを呼び出して、他のプロジェクトからこれを使用してみましょう。そうするには、以下のようなコードを含む *src/main.rs* を作成して、ライブラリクレートと同じディレクトリにバイナリクレートを作成します。

ファイル名: src/main.rs

```
extern crate communicator;

fn main() {
    communicator::client::connect();
}
```

`extern crate`コマンドを使用して、`communicator`ライブラリクレートをスコープに導入しています。パッケージには *2* つのクレートが含まれるようになりました。Cargo は、*src/main.rs* をバイナリクレートのルートファイルとして扱い、これはルートファイルが *src/lib.rs* になる既存のライブラリクレートとは区別されます。このパターンは、実行形式プロジェクトで非常に一般的です: ほとんどの機能はライブラリクレートにあり、バイナリクレートはそれを使用するわけです。結果として、他のプログラムもまたこのライブラリクレートを使用でき、良い責任の分離になるわけです。

`communicator`ライブラリの外部のクレートが検索するという観点から言えば、これまでに作ってきたモジュールは全て、`communicator`というクレートと同じ名前を持つモジュール内にあります。クレートのトップ階層のモジュールを、**ルートモジュール**と呼びます。

プロジェクトのサブモジュール内で外部クレートを使用しているとしても、`extern crate`はルートモジュール (つまり、*src/main.rs*、または *src/lib.rs*) に書くべきということにも、注目してください。それから、サブモジュールで外部クレートの要素をトップ階層のモジュールかのように参照できるわけです。

現状、バイナリクレートは、`client`モジュールからライブラリの `connect`関数を呼び出しているだけです。ところが、`cargo build`を呼び出すと、警告の後にエラーが発生します:

```
error[E0603]: module `client` is private
(エラー: `client`モジュールは非公開です)
 --> src/main.rs:4:5
  |
4 |     communicator::client::connect();
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

ああ！ このエラーは、`client`モジュールが非公開であることを教えてくれ、それが警告の肝だったわけです。Rust の文脈において、**公開**とか**非公開**という概念にぶち当たったのは、これが初めてでもあります。全コードの初期状態は、非公開です: 誰も他の人はコードを使用できないわけです。プログラム内で非公開の関数を使用していなければ、自分のプログラムだけがその関数を使用することを許可された唯一のコードなので、コンパイラは関数が未使用と警告してくるのです。

`client::connect`のような関数を公開にすると指定した後は、バイナリクレートからその関数への呼び出しが許可されるだけでなく、関数が未使用であるという警告も消え去るわけです。関数を公開にすれば、コンパイラは、関数が自分のプログラム外のコードからも使用されることがあると知りま

す。コンパイラは、関数が「使用されている」という架空の外部使用の可能性を考慮してくれます。それ故に、関数が公開とマークされれば、コンパイラはそれが自分のプログラムで使用されるべきという要求をなくし、その関数が未使用という警告も止めるのです。

9.2.1　関数を公開にする

コンパイラに何かを公開すると指示するには、定義の先頭に pubキーワードを追記します。今は、client::connectが未使用であるとする警告とバイナリークレートのモジュール`client`が非公開であるエラーの解消に努めます。*src/lib.rs* を弄って、clientモジュールを公開にしてください。そう、こんな感じに:

ファイル名: src/lib.rs

```
pub mod client;

mod network;
```

pubキーワードは、modの直前に配置されています。再度ビルドしてみましょう:

```
error[E0603]: function `connect` is private
 --> src/main.rs:4:5
  |
4 |     communicator::client::connect();
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

やった！ 違うエラーになりました！ そうです、別のエラーメッセージは、祝杯を上げる理由になるのです。新エラーは、関数`connect`は非公開ですと示しているので、*src/client.rs* を編集して、client::connectも公開にしましょう:

ファイル名: src/client.rs

```
pub fn connect() {
}
```

さて、再び、cargo buildを走らせてください:

```
warning: function is never used: `connect`
 --> src/network/mod.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default

warning: function is never used: `connect`
 --> src/network/server.rs:1:1
  |
1 | / fn connect() {
2 | | }
```

```
  | |_^
```

コードのコンパイルが通り、client:connectが使用されていないという警告はなくなりました！

コード未使用警告が必ずしも、コード内の要素を公開にしなければならないことを示唆しているわけではありません: これらの関数を公開 API の一部にしたくなかったら、未使用コード警告がもう必要なく、安全に削除できるコードに注意を向けてくれている可能性もあります。また未使用コード警告は、ライブラリ内でこの関数を呼び出している箇所全てを誤って削除した場合に、バグに目を向けさせてくれている可能性もあります。

しかし今回は、本当に他の 2 つの関数もクレートの公開 API にしたいので、これも pubとマークして残りの警告を除去しましょう。*src/network/mod.rs* を変更して以下のようにしてください:

ファイル名: src/network/mod.rs

```
pub fn connect() {
}

mod server;
```

そして、コードをコンパイルします:

```
warning: function is never used: `connect`
 --> src/network/mod.rs:1:1
  |
1 | / pub fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default

warning: function is never used: `connect`
 --> src/network/server.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
```

んんー、nework::connectは pubに設定されたにもかかわらず、まだ未使用関数警告が出ます。その理由は、関数はモジュール内で公開になったものの、関数が存在する networkモジュールは公開ではないからです。今回は、ライブラリの内部から外に向けて作業をした一方、client::connectでは、外から内へ作業をしていました。*src/lib.rs* を変えて networkも公開にする必要があります。以下のように:

ファイル名: src/lib.rs

```
pub mod client;

pub mod network;
```

これでコンパイルすれば、あの警告はなくなります:

```
warning: function is never used: `connect`
 --> src/network/server.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default
```

残る警告は 1 つなので、自分で解消してみてください！

9.2.2　プライバシー規則

まとめると、要素の公開性は以下のようなルールになります:

- 要素が公開なら、どの親モジュールを通してもアクセス可能です。
- 要素が非公開なら、直接の親モジュールとその親の子モジュールのみアクセスできます。

9.2.3　プライバシー例

もうちょっと鍛錬を得るために、もういくつかプライバシー例を見てみましょう。新しいライブラリプロジェクトを作成し、リスト 7-5 のコードを新規プロジェクトの *src/lib.rs* に入力してください:

ファイル名: src/lib.rs

```rust
mod outermost {
    pub fn middle_function() {}

    fn middle_secret_function() {}

    mod inside {
        pub fn inner_function() {}

        fn secret_function() {}
    }
}

fn try_me() {
    outermost::middle_function();
    outermost::middle_secret_function();
    outermost::inside::inner_function();
    outermost::inside::secret_function();
}
```

リスト 7-6: 公開と非公開関数の例。不正なものもあります

このコードをコンパイルする前に、`try_me`関数のどの行がエラーになるか当ててみてください。それからコンパイルを試して、合ってたかどうか確かめ、エラーの議論を求めて読み進めてください！

9.2.3.1　エラーを確かめる

try_me関数は、プロジェクトのルートモジュールに存在しています。outermostという名前のモジュールは非公開ですが、プライバシー規則の 2 番目にある通り、try_meそのままに、outermostは現在 (ルート) のモジュールなので、try_me関数は、outermostモジュールにアクセスすることを許可されるのです。

middle_functionは公開なので、outermost::middle_functionという呼び出しも動作し、try_meはmiddle_functionにその親モジュールの outermostを通してアクセスしています。前の段落でこのモジュールは、アクセス可能と決定しました。

outermost::middle_secret_functionの呼び出しは、コンパイルエラーになるでしょう。middle_secret_functionは非公開なので、2 番目の規則が適用されます。ルートモジュールは、middle_secret_functionの現在のモジュール (outermostがそうです) でも、middle_secret_functionの現在のモジュールの子供でもないのです。

insideという名前のモジュールは非公開で子モジュールを持たないので、現在のモジュールであるoutermostからのみアクセスできます。つまり、try_me関数は、outermost::inside::inner_functionも outermost::inside::secret_functionも呼び出すことを許されないのです。

9.2.3.2　エラーを修正する

エラーを修正しようとする過程でできるコード変更案は、以下の通りです。各々試してみる前に、エラーを解消できるか当ててみてください。それからコンパイルして正しかったか間違っていたか確かめ、プライバシー規則を使用して理由を理解してください。もっと実験を企てて試してみるのもご自由に！

- insideモジュールが公開だったらどうだろうか？
- outermostが公開で、insideが非公開ならどうだろうか？
- inner_functionの本体で::outermost::middle_secret_function()を呼び出したらどうだろうか？ (頭の二つのコロンは、ルートモジュールから初めてモジュールを参照したいということを意味します)

今度は、useキーワードで要素をスコープに導入する話をしましょう。

9.3　異なるモジュールの名前を参照する

モジュール名を呼び出しの一部に使用して、モジュール内に定義された関数の呼び出し方法を解説しました。リスト 7-7 に示した nested_modules関数の呼び出しのような感じですね:

ファイル名: src/main.rs

```
pub mod a {
    pub mod series {
        pub mod of {
```

```
            pub fn nested_modules() {}
        }
    }
}

fn main() {
    a::series::of::nested_modules();
}
```

リスト 7-7: 囲まれたモジュールをフルパス指定して関数を呼び出す

見てお分かりの通り、フルパス指定した名前を参照すると非常に長ったらしくなります。幸い、Rust には、これらの呼び出しをもっと簡潔にするキーワードが用意されています。

9.3.1　use キーワードで名前をスコープに導入する

Rust の useキーワードは、呼び出したい関数のモジュールをスコープに導入することで、長ったらしい関数呼び出しを短縮します。以下は、a::series::ofモジュールをバイナリクレートのルートスコープに持ってくる例です:

Filename: src/main.rs

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

use a::series::of;

fn main() {
    of::nested_modules();
}
```

use a::series::of;の行は、ofモジュールを参照したい箇所全部でフルパスの a::series::ofを使用するのではなく、ofを利用できることを意味しています。

この useキーワードは、指定したものだけをスコープに入れます: モジュールの子供はスコープに導入しないのです。そのため、nested_modules関数を呼び出したい際に、それでもまだ of::nested_modulesを使わなければならないのです。

以下のように、代わりに useで関数を指定して、関数をスコープに入れることもできました:

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
```

```
}

use a::series::of::nested_modules;

fn main() {
    nested_modules();
}
```

そうすれば、モジュールをすべて取り除き、関数を直接参照することができます。

enum もモジュールのようにある種の名前空間をなすので、enum のバリアントを useでスコープに導入することもできます。どんな use文に関しても、一つの名前空間から複数の要素をスコープに導入する場合、波かっことお尻にカンマを使用することで列挙できます。こんな感じで:

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::{Red, Yellow};

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = TrafficLight::Green;
}
```

Greenを use文に含んでいないので、まだ Greenバリアント用に TrafficLight名前空間を指定しています。

9.3.2　Glob で全ての名前をスコープに導入する

ある名前空間の要素を全て一度にスコープに導入するには、*表記が使用でき、これは glob(塊) 演算子と呼ばれます。この例は、ある enum の列挙子を各々を列挙せずに全てスコープに導入しています:

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::*;

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = Green;
}
```

`*`は `TrafficLight`名前空間に存在する全て公開要素をスコープに導入します。あまり glob は使用するべきではありません: 便利ではありますが、glob は予想以上の要素を引き込んで、名前衝突を引き起こす可能性があるのです。

9.3.3　super を使用して親モジュールにアクセスする

この章の頭で見かけたように、ライブラリクレートを作成する際、Cargo は `tests`モジュールを用意してくれました。今からそれについて詳しく掘り下げていくことにしましょう。`communicator`プロジェクトで *src/lib.rs* を開いてください:

ファイル名: src/lib.rs

```
pub mod client;

pub mod network;

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

第 11 章でテストについて詳しく説明しますが、これでこの例の一部が持つ意味がわかったのではないでしょうか: 他のモジュールに隣接する `tests`という名前のモジュールがあり、このモジュールは `it_works`という名前の関数を含んでいます。特別な注釈があるものの、`tests`モジュールもただのモジュールです！ よって、モジュール階層は以下のような見た目になります:

```
communicator
  ├── client
  ├── network
  |   └── client
  └── tests
```

テストは、ライブラリ内でコードの準備運動を行うためのものなので、この `it_works`関数から `client::connect`関数を呼び出してみましょう。まあ、尤も今のところ、機能の検査は何もしないんですけどね。これはまだ動きません:

ファイル名: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        client::connect();
    }
```

```
}
```

cargo testコマンドを呼び出してテストを実行してください:

```
$ cargo test
   Compiling communicator v0.1.0 (file:///projects/communicator)
error[E0433]: failed to resolve. Use of undeclared type or module `client`
(エラー: 解決に失敗しました。未定義の型、またはモジュール`client`を使用しています)
 --> src/lib.rs:9:9
  |
9 |         client::connect();
  |         ^^^^^^ Use of undeclared type or module `client`
```

コンパイルが失敗しましたが、なぜでしょうか？*src/main.rs* のように、関数の直前に communicator ::を配置する必要はありません。なぜなら、間違いなくここでは、communicatorライブラリクレート内にいるからです。原因は、パスが常に現在のモジュールに対して相対的になり、ここでは testsになっているからです。唯一の例外は、use文内であり、パスは標準でクレートのルートに相対的になります。testsモジュールは、clientモジュールがスコープに存在する必要があるのです！

では、どうやってモジュール階層を一つ上がり、testsモジュールの client::connect関数を呼び出すのでしょうか？ testsモジュールにおいて、先頭にコロンを使用して、コンパイラにルートから始めて、フルパスを列挙したいと知らせることもできます。こんな感じで:

```
::client::connect();
```

あるいは、superを使用して現在のモジュールからモジュール階層を一つ上がることもできます。以下のように:

```
super::client::connect();
```

この例では、これら二つの選択はそれほど異なるようには見えませんが、モジュール階層がもっと深ければ、常にルートから書き始めるのは、コードを冗長にする原因になります。そのような場合、superを使用して現在のモジュールから兄弟のモジュールに辿り着くのは、いいショートカットになります。さらに、コードのいろんなところでルートからパスを指定し、モジュール構造を変化させた場合、複数箇所でパスを更新する必要が陥り、面倒なことになるでしょう。

各テストで super::と入力しなければならないのも不快なことですが、それを解決してくれる道具をもう見かけています: useです！ super::の機能は、useに与えるパスを変更するので、ルートモジュールではなく、親モジュールに対して相対的になります。

このような理由から、ことに testsモジュールにおいて use super::somthingは通常、最善策になるわけです。故に、今ではテストはこんな見た目になりました:

ファイル名: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::client;
```

```
4
5       #[test]
6       fn it_works() {
7           client::connect();
8       }
9   }
```

再度 cargo testを実行すると、テストは通り、テスト結果出力の最初の部分は以下のようになるでしょう:

```
$ cargo test
   Compiling communicator v0.1.0 (file:///projects/communicator)
     Running target/debug/communicator-92007ddb5330fa5a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

9.4　まとめ

これでコードを体系化する新しいテクニックを知りましたね！ これらのテクニックを使用して、関連のある機能をまとめ上げ、ファイルが長くなりすぎるのを防ぎ、ライブラリの使用者に整理整頓された公開 API を提供してください。

次は、自分の素晴らしく綺麗なコードで使用できる標準ライブラリのコレクションデータ構造について見ていきましょう。

第 10 章

一般的なコレクション

Rust の標準ライブラリは、**コレクション**と呼ばれる多くの非常に有益なデータ構造を含んでいます。他の多くのデータ型は、ある一つの値を表しますが、コレクションは複数の値を含むことができます。組み込みの配列とタプル型とは異なり、これらのコレクションが指すデータはヒープに確保され、データ量はコンパイル時にわかる必要はなく、プログラムの実行にあわせて、伸縮可能であることになります。各種のコレクションには異なる能力とコストが存在し、自分の現在の状況に最適なものを選び取るスキルは、時間とともに育っていきます。この章では、Rust のプログラムにおいて、非常に頻繁に使用される 3 つのコレクションについて議論しましょう。

- **ベクタ型**は、可変長の値を並べて保持できる。
- **文字列**は、文字のコレクションである。以前、`String`型について触れたが、この章ではより掘り下げていく。
- **ハッシュマップ**は、値を特定のキーと紐付けさせてくれる。より一般的なデータ構造である、**マップ**の特定の実装である。

標準ライブラリで提供されている他の種のコレクションについて学ぶには、ドキュメントを参照されたし。

ベクタ型、文字列、ハッシュマップの生成と更新方法や、各々が特別な点について議論していきましょう。

10.1 ベクタで一連の値を保持する

最初に見るコレクションは、`Vec<T>`であり、**ベクタ**としても知られています。ベクタは、メモリ上に値を隣り合わせに並べる単独のデータ構造に 2 つ以上の値を保持させてくれます。ベクタには、同じ型の値しか保持できません。要素のリストがある場合に有用です。例えば、テキストファイルの各行とか、ショッピングカートのアイテムの価格などです。

10.1.1　新しいベクタを生成する

新しい空のベクタを作るには、リスト 8-1 に示されたように、Vec::new関数を呼ぶことができます。

```
let v: Vec<i32> = Vec::new();
```

リスト 8-1: 新しい空のベクタを生成して i32型の値を保持する

ここでは、型注釈を付け足したことに注目してください。このベクタに対して、何も値を挿入していないので、コンパイラには、どんなデータを保持させるつもりなのかわからないのです。これは重要な点です。ベクタは、ジェネリクスを使用して実装されているのです; 独自の型でジェネリクスを使用する方法については、第 10 章で解説します。今は、標準ライブラリにより提供されている Vec<T> 型は、どんな型でも保持でき、特定のベクタが特定の型を保持するとき、その型は山かっこ内に指定されることを知っておいてください。リスト 8-1 では、コンパイラに vの Vec<T>は、i32型の要素を保持すると指示しました。

より現実的なコードでは、一旦値を挿入したら、コンパイラは保持させたい値の型をしばしば推論できるので、この型注釈をすることは滅多にありません。初期値のある Vec<T>を生成する方が一般的ですし、Rust には、利便性のために vec!というマクロも用意されています。このマクロは、与えた値を保持する新しいベクタ型を生成します。リスト 8-2 では、1、2、3という値を持つ新しい Vec<i32> を生成しています:

```
let v = vec![1, 2, 3];
```

リスト 8-2: 値を含む新しいベクタを生成する

初期値の i32値を与えたので、コンパイラは、vの型が Vec<i32>であると推論でき、型注釈は必要なくなりました。次は、ベクタを変更する方法を見ましょう。

10.1.2　ベクタを更新する

ベクタを生成し、それから要素を追加するには、リスト 8-3 に示したように、pushメソッドを使用できます。

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

リスト 8-3: pushメソッドを使用してベクタ型に値を追加する

あらゆる変数同様、第 3 章で議論したように、値を変化させたかったら、mutキーワードで可変にする必要があります。中に配置する数値は全て i32型であり、コンパイラはこのことをデータから推論するので、Vec<i32>という注釈は必要なくなります。

10.1.3　ベクタをドロップすれば、要素もドロップする

他のあらゆる構造体同様、ベクタもスコープを抜ければ、解放されます。リスト 8-4 に注釈したようにね:

```
{
    let v = vec![1, 2, 3, 4];

    // vで作業をする

} // <- vはここでスコープを抜け、解放される
```

リスト 8-4: ベクタとその要素がドロップされる箇所を示す

ベクタがドロップされると、その中身もドロップされます。つまり、保持されていた整数値が、片付けられるということです。これは一見単純な点に見えるかもしれませんが、ベクタの要素への参照を導入した途端、もうちょっと複雑になる可能性を秘めています。次は、それに挑んでいきましょう！

10.1.4　ベクタの要素を読む

もうベクタを生成し、更新し、破壊する方法を知ったので、コンテンツを読む方法を知るのはいいステップアップです。ベクタに保持された値を参照する方法は 2 つあります。例では、さらなる明瞭性を求めて、これらの関数から返る値の型を注釈しました。

リスト 8-5 に示したのは、両メソッドがベクタの値に対して、添字記法と getメソッドによりアクセスするところです:

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
let third: Option<&i32> = v.get(2);
```

リスト 8-5: 添字記法か getメソッドを使用してベクタの要素にアクセスする

ここでは、2 つのことに気付いてください。まず、3 番目の要素を得るのに 2という添え字の値を使用していることです: ベクタは、数値により順序付けされ、添え字は 0 から始まります。2 番目に、3 番目の要素を得る 2 つの方法は、&と []を使用して参照を得るものと、番号を引数として getメソッドに渡して、Option<&T>を得るものということです。

Rust には要素を参照する方法が 2 通りあるので、ベクタに要素が含まれない番号の値を使用しようとした時に、プログラムの振る舞いを選択できます。例として、ベクタに 5 つ要素があり、番号 100 の要素にアクセスを試みた場合、プログラムがすることを確認しましょう。リスト 8-6 に示したようにね:

```
let v = vec![1, 2, 3, 4, 5];

let does_not_exist = &v[100];
```

```
let does_not_exist = v.get(100);
```

リスト 8-6: 5 つの要素を含むベクタの 100 番目の要素にアクセスしようとする

このコードを走らせると、最初の []メソッドはプログラムをパニックさせます。存在しない要素を参照しているからです。このメソッドは、ベクタの終端を超えて要素にアクセスしようした時にプログラムをクラッシュさせたい場合に最適です。

getメソッドがベクタ外の番号を渡されると、パニックすることなく Noneを返します。普通の状態でも、ベクタの範囲外にアクセスする可能性がある場合に、このメソッドを使用することになるでしょう。そうしたら、コードには Some(&element)か Noneを扱うロジックが存在することになります。そう、第 6 章で議論したように。例えば、番号は人間に数値を入力してもらうことで得ることもできます。もし大きすぎる値を誤って入力し、プログラムが None値を得てしまったら、現在ベクタに幾つ要素があるかをユーザに教え、再度正しい値を入力してもらうことができるでしょう。その方が、タイプミスでプログラムをクラッシュさせるより、ユーザに優しくなるでしょう。

プログラムに有効な参照がある場合、borrow checker(借用精査機) は (第 4 章で解説しましたが)、所有権と借用規則を強制し、ベクタ型の中身へのこの参照や他のいかなる参照も有効であり続けることを保証してくれます。同一スコープ上では、可変と不変な参照を同時には存在させられないというルールを思い出してください。このルールはリスト 8-7 にも適用され、リスト 8-7 ではベクタの最初の要素への不変参照を保持し、終端に要素を追加しようとしていますが、動きません:

```
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);
```

リスト 8-7: 要素への参照を保持しつつ、ベクタに要素を追加しようとする

このコードをコンパイルすると、こんなエラーになります:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
(エラー: 不変としても借用されているので、`v`を可変で借用できません)
  |
4 |     let first = &v[0];
  |                  - immutable borrow occurs here
  |                  (不変借用はここで発生しています)
5 |
6 |     v.push(6);
  |     ^ mutable borrow occurs here
  |      (可変借用は、ここで発生しています)
7 | }
  | - immutable borrow ends here
  |   (不変借用はここで終了しています)
```

リスト 8-7 のコードは、一見動くはずのように見えるかもしれません: なぜ、最初の要素への参照が、ベクタの終端への変更を気にかける必要があるのでしょうか？ このエラーは、ベクタの動作法の

せいです: 新規要素をベクタの終端に追加すると、ベクタが現在存在する位置に隣り合って要素を入れるだけの領域がなかった場合に、メモリの新規確保をして古い要素を新しいスペースにコピーする必要があるかもしれないからです。その場合、最初の要素を指す参照は、解放されたメモリを指すことになるでしょう。借用規則により、そのような場面に落ち着かないよう回避されるのです。

> 注釈: `Vec<T>`の実装に関する詳細については、`https://doc.rust-lang.org/stable/nomicon/vec.html`の、”The Rustonomicon” を参照されたし。

10.1.5　ベクタの値を走査する

ベクタの要素に順番にアクセスしたいなら、添え字で 1 回に 1 要素にアクセスするのではなく、全要素を走査することができます。リスト 8-8 で `for`ループを使い、`i32`のベクタの各要素に対する不変な参照を得て、それらを出力する方法を示しています:

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

リスト 8-8: `for`ループで要素を走査し、ベクタの各要素を出力する

全要素に変更を加える目的で、可変なベクタの各要素への可変な参照を走査することもできます。リスト 8-9 の `for`ループでは、各要素に `50`を足しています:

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

リスト 8-9: ベクタの要素への可変な参照を走査する

可変参照が参照している値を変更するには、`+=`演算子を使用する前に、参照外し演算子 (`*`) を使用して `i`の値に辿り着かないといけません。

10.1.6　Enum を使って複数の型を保持する

この章の冒頭で、ベクタは同じ型の値しか保持できないと述べました。これは不便に考えられることもあります; 異なる型の要素を保持する必要性が出てくるユースケースも確かにあるわけです。幸運なことに、enum の列挙子は、同じ enum の型の元に定義されるので、ベクタに異なる型の要素を保持する必要が出たら、enum を定義して使用することができます！

例えば、スプレッドシートの行から値を得たくなったとしましょう。ここで行の列には、整数を含むものや、浮動小数点数を含むもの、文字列を含むものがあります。列挙子が異なる値の型を保持する enum を定義できます。そして、この enum の列挙子は全て同じ型: enum の型と考えられるわけ

です。それからその enum を保持するベクタを生成でき、結果的に異なる型を保持できるようになるわけです。リスト 8-10 でこれを模擬しています。

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

リスト 8-10: enumを定義して、一つのベクタに異なる型の値を保持する

各要素を保持するのにヒープ上でズバリどれくらいのメモリが必要になるかをわかるように、コンパイラがコンパイル時にベクタに入る型を知る必要があります。副次的な利点は、このベクタではどんな型が許容されるのか明示できることです。もし Rust でベクタがどんな型でも保持できたら、ベクタの要素に対して行われる処理に対して一つ以上の型がエラーを引き起こす可能性があったでしょう。enum に加えて match式を使うことは、第 6 章で議論した通り、コンパイル時にありうる場合全てに対処していることをコンパイラが、確認できることを意味します。

プログラム記述時にプログラムがベクタに実行時に保持するありとあらゆる一連の型をプログラマが知らない場合、この enum テクニックはうまく動かないでしょう。代わりにトレイトオブジェクトを使用することができ、こちらは第 17 章で解説します。

今や、ベクタを使用するべきありふれた方法について議論したので、標準ライブラリで Vec<T>に定義されている多くの有益なメソッドについては、API ドキュメントを確認することを心得てください。例として、pushに加えて、popメソッドは最後の要素を削除して返します。次のコレクション型に移りましょう: Stringです！

10.2　文字列で UTF-8 でエンコードされたテキストを保持する

第 4 章で文字列について語りましたが、今度はより掘り下げていきましょう。新参者の Rust 市民は、3 つの概念の組み合わせにより、文字列でよく行き詰まります: Rust のありうるエラーを晒す性質、多くのプログラマが思っている以上に文字列が複雑なデータ構造であること、そして UTF-8 です。これらの要因が、他のプログラミング言語から移ってきた場合、一見困難に見えるように絡み合うわけです。

コレクションの文脈で文字列を議論することは、有用なことです。なぜなら、文字列はテキストとして解釈された時に有用になる機能を提供するメソッドと、バイトの塊で実装されているからです。この節では、生成、更新、読み込みのような全コレクションが持つ Stringの処理について語ります。また、Stringが他のコレクションと異なる点についても議論します。具体的には、人間とコンピュー

タが Stringデータを解釈する方法の差異により、Stringに添え字アクセスする方法がどう複雑なのかということです。

10.2.1　文字列とは？

まずは、**文字列**という用語の意味を定義しましょう。Rust には、言語の核として 1 種類しか文字列型を持ちません。文字列スライスの strで、通常借用された形態&strで見かけます。第 4 章で、**文字列スライス**について語りました。これは、別の場所に保持された UTF-8 エンコードされた文字列データへの参照です。例えば、文字列リテラルは、プログラムのバイナリ出力に保持されるので、文字列スライスになります。

String型は、言語の核として組み込まれるのではなく、Rust の標準ライブラリで提供されますが、伸長可能、可変、所有権のある UTF-8 エンコードされた文字列型です。Rust 市民が Rust において「文字列」を指したら、どちらかではなく、Stringと文字列スライスの&strのことを通常意味します。この節は、大方、Stringについてですが、どちらの型も Rust の標準ライブラリで重宝されており、どちらも UTF-8 エンコードされています。

また、Rust の標準ライブラリには、他の文字列型も含まれています。OsString、OsStr、CString、CStrなどです。ライブラリクレートにより、文字列データを保持する選択肢はさらに増えます。それらの名前が全て Stringか Strで終わっているのがわかりますか？ 所有権ありと借用されたバージョンを指しているのです。ちょうど以前見かけた Stringと&strのようですね。例えば、これらの文字列型は、異なるエンコード方法でテキストを保持していたり、メモリ上の表現が異なったりします。この章では、これらの他の種類の文字列については議論しません; 使用方法やどれが最適かについては、API ドキュメントを参照してください。

10.2.2　新規文字列を生成する

Vec<T>で使用可能な処理の多くが Stringでも使用できます。文字列を生成する new関数から始めましょうか。リスト 8-11 に示したようにね:

```
let mut s = String::new();
```

リスト 8-11: 新しい空の Stringを生成する

この行は、新しい空の sという文字列を生成しています。それからここにデータを読み込むことができるわけです。だいたい、文字列の初期値を決めるデータがあるでしょう。そのために、to_string メソッドを使用します。このメソッドは、文字列リテラルがしているように、Displayトレイトを実装する型ならなんでも使用できます。リスト 8-12 に 2 例、示しています:

```
let data = "initial contents";

let s = data.to_string();

// the method also works on a literal directly:
```

```
let s = "initial contents".to_string();
```

リスト 8-12: to_stringメソッドを使用して文字列リテラルから Stringを生成する

このコードは、initial contents(初期値) を含む文字列を生成します。

さらに、String::from関数を使っても、文字列リテラルから Stringを生成することができます。リスト 8-11 のコードは、to_stringを使用するリスト 8-10 のコードと等価です:

```
let s = String::from("initial contents");
```

リスト 8-13: String::from関数を使って文字列リテラルから Stringを作る

文字列は、非常に多くのものに使用されるので、多くの異なる一般的な API を使用でき、たくさんの選択肢があるわけです。冗長に思われるものもありますが、適材適所です！ 今回の場合、String::fromと to_stringは全く同じことをします。従って、どちらを選ぶかは、スタイル次第です。

文字列は UTF-8 エンコードされていることを覚えていますか？ 要するに文字列には、適切にエンコードされていればどんなものでも含めます。リスト 8-14 に示したように:

```
let hello = String::from("            ");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("       ");
let hello = String::from("      ");
let hello = String::from("こんにちは");
let hello = String::from("          ");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

リスト 8-14: いろんな言語の挨拶を文字列に保持する

これらは全て、有効な Stringの値です。

10.2.3　文字列を更新する

Stringは、サイズを伸ばすことができ、中身も変化します。追加のデータをプッシュすれば、Vec<T>の中身のようですね。付け加えると、String値を連結する +演算子や、format!マクロを便利に使用することができます。

10.2.3.1　push_str と push で文字列に追加する

push_strメソッドで文字列スライスを追記することで、Stringを伸ばすことができます。リスト 8-15 の通りです:

```
let mut s = String::from("foo");
s.push_str("bar");
```

リスト 8-15: push_strメソッドで Stringに文字列スライスを追記する

この 2 行の後、sは foobarを含むことになります。push_strメソッドは、必ずしも引数の所有権を得なくていいので、文字列スライスを取ります。例えば、リスト 8-16 のコードは、中身を s1に追加した後、s2を使えなかったら不幸だということを示しています:

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {}", s2);
```

リスト 8-16: 中身を Stringに追加した後に、文字列スライスを使用する

もし、push_strメソッドが s2の所有権を奪っていたら、最後の行でその値を出力することは不可能でしょう。ところが、このコードは予想通りに動きます！

pushメソッドは、1 文字を引数として取り、Stringに追加します。リスト 8-15 は、pushメソッドで l を Stringに追加するコードを呈示しています。

```
let mut s = String::from("lo");
s.push('l');
```

リスト 8-17: pushで String値に 1 文字を追加する

このコードの結果、sは lolを含むことになるでしょう。

> 編者注: lolは laughing out loud(大笑いする) の頭文字からできたスラングです。日本語の wwwみたいなものですね。

10.2.3.2　+ 演算子、または format!マクロで連結

2 つのすでにある文字列を組み合わせたくなることがよくあります。リスト 8-18 に示したように、一つ目の方法は、+演算子を使用することです:

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // s1はムーブされ、もう使用できないことに注意
```

リスト 8-18: +演算子を使用して二つの String値を新しい String値にする

このコードの結果、s3という文字列は、Hello, world!を含むことになるでしょう。追記の後、s1がもう有効でなくなった理由と、s2への参照を使用した理由は、+演算子を使用した時に呼ばれるメソッドのシグニチャと関係があります。+演算子は、addメソッドを使用し、そのシグニチャは以下のような感じです:

```
fn add(self, s: &str) -> String {
```

これは、標準ライブラリにあるシグニチャそのものではありません: 標準ライブラリでは、addはジェネリクスで定義されています。ここでは、ジェネリックな型を具体的な型に置き換えた addの

シグニチャを見ており、これは、このメソッドを String値とともに呼び出した時に起こることです。ジェネリクスについては、第 10 章で議論します。このシグニチャが、+演算子の巧妙な部分を理解するのに必要な手がかりになるのです。

まず、s2には&がついてます。つまり、add関数の s引数のために最初の文字列に 2 番目の文字列の参照を追加するということです: Stringには&strを追加することしかできません。要するに 2 つのString値を追加することはできないのです。でも待ってください。addの第 2 引数で指定されているように、&s2の型は、&strではなく、&Stringではないですか。では、なぜ、リスト 8-18 は、コンパイルできるのでしょうか？

add呼び出しで&s2を使える理由は、コンパイラが&String引数を&strに**型強制**してくれるためです。addメソッド呼び出しの際、コンパイラは、**参照外し型強制**というものを使用し、ここでは、&s2を&s2[..]に変えるものと考えることができます。参照外し型強制について詳しくは、第 15 章で議論します。addが s引数の所有権を奪わないので、この処理後も s2が有効な Stringになるわけです。

2 番目に、シグニチャから addは selfの所有権をもらうことがわかります。selfには&がついていないからです。これはつまり、リスト 8-18 において s1は add呼び出しにムーブされ、その後は有効ではなくなるということです。故に、s3 = s1 + &s2;は両文字列をコピーして新しいものを作るように見えますが、この文は実際には s1の所有権を奪い、s2の中身のコピーを追記し、結果の所有権を返すのです。言い換えると、たくさんのコピーをしているように見えますが、違います; 実装は、コピーよりも効率的です。

複数の文字列を連結する必要が出ると、+演算子の振る舞いは扱いにくくなります:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

ここで、sは tic-tac-toeになるでしょう。+と"文字のせいで何が起きているのかわかりにくいです。もっと複雑な文字列の連結には、format!マクロを使用することができます:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}-{}-{}", s1, s2, s3);
```

このコードでも、sは tic-tac-toeになります。format!マクロは、println!と同様の動作をしますが、出力をスクリーンに行う代わりに、中身を Stringで返すのです。format!を使用したコードの方がはるかに読みやすく、引数の所有権を奪いません。

10.2.4　文字列に添え字アクセスする

他の多くのプログラミング言語では、文字列中の文字に、番号で参照してアクセスすることは、有効なコードであり、一般的な処理です。しかしながら、Rust において、添え字記法で `String`の一部にアクセスしようとすると、エラーが発生するでしょう。リスト 8-19 の無効なコードを考えてください:

```
let s1 = String::from("hello");
let h = s1[0];
```

リスト 8-19: 文字列に対して添え字記法を試みる

このコードは、以下のようなエラーに落ち着きます:

```
error[E0277]: the trait bound `std::string::String: std::ops::Index<{Integer}>` is not
    satisfied
(エラー: トレイト境界`std::string::String: std::ops::Index<{Integer}>`が満たされていま
    せん)
  |>
3 |>     let h = s1[0];
  |>             ^^^^^ the type `std::string::String` cannot be indexed by `{Integer}`
  |>                   (型`std::string::String`は`{Integer}`で添え字アクセスできません)
  = help: the trait `std::ops::Index<{Integer}>` is not implemented for `std::string::
      String`
  (ヘルプ: `std::ops::Index<{Integer}>`というトレイトが`std::string::String`に対して実
      装されていません)
```

エラーと注釈が全てを物語っています: Rust の文字列は、添え字アクセスをサポートしていないのです。でも、なぜでしょうか？ その疑問に答えるには、Rust がメモリにどのように文字列を保持しているかについて議論する必要があります。

10.2.4.1　内部表現

`String`は `Vec<u8>`のラッパです。リスト 8-14 から適切に UTF-8 でエンコードされた文字列の例をご覧ください。まずは、これ:

```
let len = String::from("Hola").len();
```

この場合、`len`は 4 になり、これは、文字列”Hola” を保持するベクタの長さが 4 バイトであることを意味します。これらの各文字は、UTF-8 でエンコードすると、1 バイトになるのです。しかし、以下の行ではどうでしょうか？ (この行は大文字のキリル文字 Ze で始まり、アラビア数字の 3 では始まっていないことに注意してください)

```
let len = String::from("Здравствуйте").len();
```

文字列の長さはと問われたら、あなたは 12 と答えるかもしれません。ところが、Rust の答えは、24 です: “Здравствуйте” を UTF-8 でエンコードすると、この長さになります。各 Unicode

スカラー値は、2 バイトの領域を取るからです。それ故に、文字列のバイト番号は、必ずしも有効な Unicode のスカラー値とは相互に関係しないのです。デモ用に、こんな無効な Rust コードを考えてください:

```
let hello = "З д р а в с т в у й т е";
let answer = &hello[0];
```

answerの値は何になるべきでしょうか？ 最初の文字の3になるべきでしょうか？ UTF-8 エンコードされた時、3の最初のバイトは 208、2 番目は 151になるので、answerは実際、208になるべきですが、208は単独では有効な文字ではありません。この文字列の最初の文字を求めている場合、208を返すことは、ユーザの望んでいるものではないでしょう; しかしながら、Rust には、番号 0 の位置には、そのデータしかないのです。文字列がラテン文字のみを含む場合でも、ユーザは一般的にバイト値が返ることを望みません: &"hello"[0]がバイト値を返す有効なコードだったら、hではなく、104を返すでしょう。予期しない値を返し、すぐには判明しないバグを引き起こさないために、Rust はこのコードを全くコンパイルせず、開発過程の早い段階で誤解を防いでくれるのです。

10.2.4.2　バイトとスカラー値と書記素クラスタ！ なんてこった！

UTF-8 について別の要点は、実際 Rust の観点から文字列を見るには 3 つの関連した方法があるということです: バイトとして、スカラー値として、そして、書記素クラスタ (人間が**文字**と呼ぶものに一番近い) としてです。

ヒンディー語の単語、“” をデーヴァナーガリー (脚注: サンスクリット語とヒンディー語を書くときに使われる書記法) で表記したものを見たら、以下のような見た目の u8値のベクタとして保持されます:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

18 バイトになり、このようにしてコンピュータは最終的にこのデータを保持しているわけです。これを Unicode スカラー値として見たら、Rust の char型はこれなのですが、このバイトは以下のような見た目になります:

```
[' ', ' ', ' ', ' ', ' ', ' ']
```

ここでは、6 つ char値がありますが、4 番目と 6 番目は文字ではありません: 単独では意味をなさないダイアクリティックです。最後に、書記素クラスタとして見たら、このヒンディー語の単語を作り上げる人間が 4 文字と呼ぶであろうものが得られます:

```
[" ", " ", "  ", "  "]
```

Rust には、データが表す自然言語に関わらず、各プログラムが必要な解釈方法を選択できるように、コンピュータが保持する生の文字列データを解釈する方法がいろいろ用意されています。

Rust で文字を得るのに Stringに添え字アクセスすることが許されない最後の理由は、添え字アクセスという処理が常に定数時間 (O(1)) になると期待されるからです。しかし、Stringでそのパフォー

マンスを保証することはできません。というのも、有効な文字がいくつあるか決定するのに、最初から番号まで中身を走査する必要があるからです。

10.2.5　文字列をスライスする

文字列に添え字アクセスするのは、しばしば悪い考えです。文字列添え字処理の戻り値の型が明瞭ではないからです: バイト値、文字、書記素クラスタ、あるいは文字列スライスにもなります。故に、文字列スライスを生成するのに、添え字を使う必要が本当に出た場合にコンパイラは、もっと特定するよう求めてきます。添え字アクセスを特定し、文字列スライスが欲しいと示唆するためには、[]で 1 つの数値により添え字アクセスするのではなく、範囲とともに []を使って、特定のバイトを含む文字列スライスを作ることができます:

```
let hello = "З д р а в с т в у й т е";

let s = &hello[0..4];
```

ここで、sは文字列の最初の 4 バイトを含む&strになります。先ほど、これらの文字は各々 2 バイトになると指摘しましたから、sはЗдになります。

&hello[0..1]と使用したら、何が起きるでしょうか？ 答え: Rust はベクタの無効な番号にアクセスしたかのように、実行時にパニックするでしょう:

```
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside 'З' (
    bytes 0..2) of `З д р а в с т в у й т е`', src/libcore/str/mod.rs:2188:4
('main'スレッドは「バイト番号1は文字の境界ではありません; `З д р а в с т в у й т е`の'
    З'(バイト番号0から2)の中です」でパニックしました)
```

範囲を使用して文字列スライスを作る際にはプログラムをクラッシュさせることがあるので、気をつけるべきです。

10.2.6　文字列を走査するメソッド群

幸いなことに、他の方法でも文字列の要素にアクセスすることができます。

もし、個々の Unicode スカラー値に対して処理を行う必要があったら、最適な方法は charsメソッドを使用するものです。“” に対して charsを呼び出したら、分解して 6 つの char型の値を返すので、各要素にアクセスするには、その結果を走査すればいいわけです:

```
for c in "      ".chars() {
    println!("{}", c);
}
```

このコードは、以下のように出力します:

```

```

```
```

bytesメソッドは、各バイトをそのまま返すので、最適になることもあるかもしれません:

```
for b in "      ".bytes() {
    println!("{}", b);
}
```

このコードは、Stringをなす 18 バイトを出力します:

```
224
164
// --snip--
165
135
```

ですが、有効な Unicode スカラー値は、2 バイト以上からなる場合もあることは心得ておいてください。

書記素クラスタを文字列から得る方法は複雑なので、この機能は標準ライブラリでは提供されていません。この機能が必要なら、crates.io でクレートを入手可能です。

10.2.7　文字列はそう単純じゃない

まとめると、文字列は込み入っています。プログラミング言語ごとにこの複雑性をプログラマに提示する方法は違います。Rust では、Stringデータを正しく扱うことが、全ての Rust プログラムにとっての規定動作になっているわけであり、これは、プログラマが UTF-8 データを素直に扱う際に、より思考しないといけないことを意味します。このトレードオフにより、他のプログラミング言語で見えるよりも文字列の複雑性がより露出していますが、ASCII 以外の文字に関するエラーを開発の後半で扱わなければならない可能性が排除されているのです。

もう少し複雑でないものに切り替えていきましょう: ハッシュマップです！

10.3　ハッシュマップに値に紐づいたキーを格納する

一般的なコレクションのトリを飾るのは、**ハッシュマップ**です。型 HashMap<K, V>は、K型のキーとV型の値の対応関係を保持します。これを**ハッシュ関数**を介して行います。ハッシュ関数は、キーと値のメモリ配置方法を決めるものです。多くのプログラミング言語でもこの種のデータ構造はサポートされていますが、しばしば名前が違います。hash、map、object、ハッシュテーブル、連想配列など、枚挙に暇 (いとま) はありません。

ハッシュマップは、ベクタで可能なように番号ではなく、どんな型にもなりうるキーを使ってデータを参照したいときに有用です。例えば、ゲームにおいて、各チームのスコアをハッシュマップで追

いかけることができます。ここで、各キーはチーム名、値が各チームのスコアになります。チーム名が与えられれば、スコアを扱うことができるわけです。

この節でマッシュマップの基礎的な API を見ていきますが、より多くのグッズが標準ライブラリにより、HashMap<K, V>上に定義された関数に隠されています。いつものように、もっと情報が欲しければ、標準ライブラリのドキュメントをチェックしてください。

10.3.1　新規ハッシュマップを生成する

空のハッシュマップを newで作り、要素を insertで追加することができます。リスト 8-20 では、名前がブルーとイエローの 2 チームのスコアを追いかけています。ブルーチームは 10 点から、イエローチームは 50 点から始まります:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

リスト 8-20: ハッシュマップを生成してキーと値を挿入する

最初に標準ライブラリのコレクション部分から HashMapを useする必要があることに注意してください。今までの 3 つの一般的なコレクションの内、これが最も使用頻度が低いので、初期化処理で自動的にスコープに導入される機能には含まれていません。また、標準ライブラリからのサポートもハッシュマップは少ないです; 例えば、生成するための組み込みマクロがありません。

ベクタと全く同様に、ハッシュマップはデータをヒープに保持します。この HashMapはキーが String型、値は i32型です。ベクタのように、ハッシュマップは均質です: キーは全て同じ型でなければならず、値も全て同じ型でなければなりません。

ハッシュマップを生成する別の方法は、タプルのベクタに対して collectメソッドを使用するものです。ここで、各タプルは、キーと値から構成されています。collectメソッドはいろんなコレクション型にデータをまとめ上げ、そこには HashMapも含まれています。例として、チーム名と初期スコアが別々のベクタに含まれていたら、zipメソッドを使ってタプルのベクタを作り上げることができ、そこでは「ブルー」は 10 とペアになるなどします。リスト 8-21 に示したように、それから collectメソッドを使って、そのタプルのベクタをハッシュマップに変換することができるわけです:

```
use std::collections::HashMap;

let teams  = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
```

リスト 8-21: チームのリストとスコアのリストからハッシュマップを作る

ここでは、HashMap<_, _>という型注釈が必要になります。なぜなら、いろんなデータ構造にまとめ上げることができ、コンパイラは指定しない限り、どれがいるのかわからないからです。ところが、キーと値の型引数については、アンダースコアを使用しており、コンパイラはベクタのデータ型に基づいてハッシュマップが含む型を推論することができるのです。

10.3.2　ハッシュマップと所有権

i32のような Copyトレイトを実装する型について、値はハッシュマップにコピーされます。String のような所有権のある値なら、値はムーブされ、リスト 8-22 でデモされているように、ハッシュマップはそれらの値の所有者になるでしょう:

```
use std::collections::HashMap;

let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name and field_value are invalid at this point, try using them and
// see what compiler error you get!
// field_nameとfield_valueはこの時点で無効になる。試しに使ってみて
// どんなコンパイルエラーが出るか確認してみて！
```

リスト 8-22: 一旦挿入されたら、キーと値はハッシュマップに所有されることを示す

insertを呼び出して field_nameと field_valueがハッシュマップにムーブされた後は、これらの変数を使用することは叶いません。

値への参照をハッシュマップに挿入したら、値はハッシュマップにムーブされません。参照が指している値は、最低でもハッシュマップが有効な間は、有効でなければなりません。これらの問題について詳細には、第 10 章の「ライフタイムで参照を有効化する」節で語ります。

10.3.3　ハッシュマップの値にアクセスする

リスト 8-23 に示したように、キーを getメソッドに提供することで、ハッシュマップから値を取り出すことができます:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

リスト 8-23: ハッシュマップに保持されたブルーチームのスコアにアクセスする

ここで、scoreはブルーチームに紐づけられた値になり、結果は Some(&10)となるでしょう。結果は Someに包まれます。というのも、getは Option<&V>を返すからです; キーに対応する値がハッシュマップになかったら、getは Noneを返すでしょう。プログラムは、この Optionを第 6 章で解説した方法のどれかで扱う必要があるでしょう。

ベクタのように、forループでハッシュマップのキーと値のペアを走査することができます:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

このコードは、各ペアを任意の順番で出力します:

```
Yellow: 50
Blue: 10
```

10.3.4　ハッシュマップを更新する

キーと値の数は伸長可能なものの、各キーには 1 回に 1 つの値しか紐づけることができません。ハッシュマップ内のデータを変えたい時は、すでにキーに値が紐づいている場合の扱い方を決めなければなりません。古い値を新しい値で置き換えて、古い値を完全に無視することもできます。古い値を保持して、新しい値を無視し、キーにまだ値がない場合に新しい値を追加するだけにすることもできます。あるいは、古い値と新しい値を組み合わせることもできます。各方法について見ていきましょう！

10.3.4.1　値を上書きする

キーと値をハッシュマップに挿入し、同じキーを異なる値で挿入したら、そのキーに紐づけられている値は置換されます。リスト 8-24 のコードは、insertを二度呼んでいるものの、ハッシュマップには一つのキーと値の組しか含まれません。なぜなら、ブルーチームキーに対する値を 2 回とも挿入しているからです:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

```

```
println!("{:?}", scores);
```

リスト 8-24: 特定のキーで保持された値を置き換える

このコードは、`{"Blue": 25}`と出力するでしょう。`10`という元の値は上書きされたのです。

10.3.4.2　キーに値がなかった時のみ値を挿入する

特定のキーに値があるか確認することは一般的であり、存在しない時に値を挿入することも一般的です。ハッシュマップには、これを行う `entry`と呼ばれる特別な API があり、これは、引数としてチェックしたいキーを取ります。この `entry`関数の戻り値は、`Entry`と呼ばれる enum であり、これは存在したりしなかったりするかもしれない値を表します。イエローチームに対するキーに値が紐づけられているか否か確認したくなったとしましょう。存在しなかったら、50 という値を挿入したく、ブルーチームに対しても同様です。`entry`API を使用すれば、コードはリスト 8-25 のようになります:

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{:?}", scores);
```

リスト 8-25: `entry`メソッドを使ってキーに値がない場合だけ挿入する

`Entry`上の `or_insert`メソッドは、対応する `Entry`キーが存在した時にそのキーに対する値への可変参照を返すために定義されており、もしなかったら、引数をこのキーの新しい値として挿入し、新しい値への可変参照をを返します。このテクニックの方が、そのロジックを自分で書くよりもはるかに綺麗な上に、borrow checker とも親和性が高くなります。

リスト 8-25 のコードを実行すると、`{"Yellow": 50, "Blue": 10}`と出力するでしょう。最初の `entry`呼び出しは、まだイエローチームに対する値がないので、値 `50`でイエローチームのキーを挿入します。`entry`の 2 回目の呼び出しはハッシュマップを変更しません。なぜなら、ブルーチームにはすでに `10`という値があるからです。

10.3.4.3　古い値に基づいて値を更新する

ハッシュマップの別の一般的なユースケースは、キーの値を探し、古い値に基づいてそれを更新することです。例えば、リスト 8-26 は、各単語があるテキストに何回出現するかを数え上げるコードを示しています。キーに単語を入れたハッシュマップを使用し、その単語を何回見かけたか追いかけるために値を増やします。ある単語を見かけたのが最初だったら、まず `0`という値を挿入します:

```
use std::collections::HashMap;

let text = "hello world wonderful world";
```

```
4
5  let mut map = HashMap::new();
6
7  for word in text.split_whitespace() {
8      let count = map.entry(word).or_insert(0);
9      *count += 1;
10 }
11
12 println!("{:?}", map);
```

リスト 8-26: 単語とカウントを保持するハッシュマップを使って単語の出現数をカウントする

このコードは、`{"world": 2, "hello": 1, "wonderful": 1}`と出力するでしょう。`or_insert`関数は実際、このキーに対する値への可変参照 (`&mut V`) を返すのです。ここでその可変参照を `count`変数に保持しているので、その値に代入するには、まずアスタリスク (`*`) で `count`を参照外ししなければならないのです。この可変参照は、`for`ループの終端でスコープを抜けるので、これらの変更は全て安全であり、借用規則により許可されるのです。

10.3.5　ハッシュ関数

標準では、`HashMap`はサービス拒否 (DoS) アタックに対して抵抗を示す暗号学的に安全なハッシュ関数を使用します。これは、利用可能な最速のハッシュアルゴリズムではありませんが、パフォーマンスの欠落と引き換えに安全性を得るというトレードオフは、価値があります。自分のコードをプロファイリングして、自分の目的では標準のハッシュ関数は遅すぎると判明したら、異なる *hasher* を指定することで別の関数に切り替えることができます。hasher とは、`BuildHasher`トレイトを実装する型のことです。トレイトについてとその実装方法については、第 10 章で語ります。必ずしも独自の hasher を 1 から作り上げる必要はありません; crates.io には、他の Rust ユーザによって共有された多くの一般的なハッシュアルゴリズムを実装した hasher を提供するライブラリがあります。

10.4　まとめ

ベクタ、文字列、ハッシュマップはデータを保持し、アクセスし、変更する必要のあるプログラムで必要になる、多くの機能を提供してくれるでしょう。今なら解決可能なはずの練習問題を用意しました:

- 整数のリストが与えられ、ベクタを使って mean(平均値)、median(ソートされた時に真ん中に来る値)、mode(最も頻繁に出現する値; ハッシュマップがここでは有効活用できるでしょう) を返してください。
- 文字列をピッグ・ラテン (脚注: 英語の言葉遊びの一つ) に変換してください。各単語の最初の子音は、単語の終端に移り、”ay” が足されます。従って、”first” は”irst-fay” になります。ただし、母音で始まる単語には、お尻に”hay” が付け足されます (”apple” は”apple-hay” になります)。UTF-8 エンコードに関する詳細を心に留めておいてください！

- ハッシュマップとベクタを使用して、ユーザに会社の部署に雇用者の名前を追加させられるテキストインターフェイスを作ってください。例えば、”Add Sally to Engineering”(開発部門にサリーを追加) や”Add Amir to Sales”(販売部門にアミールを追加) などです。それからユーザにある部署にいる人間の一覧や部署ごとにアルファベット順で並べ替えられた会社の全人間の一覧を、扱わせてあげてください。

標準ライブラリの API ドキュメントには、この練習問題に有用な、ベクタ、文字列、ハッシュマップのメソッドが解説されています。

処理が失敗する可能性のあるような、より複雑なプログラムに入り込んできています; ということは、エラーの処理法について議論するのにぴったりということです。次にそれをします！

第 11 章

エラー処理

Rust の信頼性への傾倒は、エラー処理にも及びます。ソフトウェアにおいて、エラーは生きている証しです。従って、Rust には何かがおかしくなる場面を扱う機能がたくさんあります。多くの場面で、コンパイラは、エラーの可能性を知り、コードのコンパイルが通るまでに何かしら対応を行うことを要求してきます。この要求により、エラーを発見し、コードを実用に供する前に適切に対処していることを確認することでプログラムを頑健なものにしてくれるのです！

Rust では、エラーは大きく二つに分類されます: **回復可能**と**回復不能**なエラーです。ファイルが見つからないなどの回復可能なエラーには、問題をユーザに報告し、処理を再試行することが合理的になります。回復不能なエラーは、常にバグの兆候です。例えば、配列の境界を超えた箇所にアクセスしようとすることなどです。

多くの言語では、この 2 種のエラーを区別することはなく、例外などの機構を使用して同様に扱います。Rust には例外が存在しません。代わりに、回復可能なエラーには `Result<T, E>`値があり、プログラムが回復不能なエラーに遭遇した時に、実行を中止する `panic!`マクロがあります。この章では、まず `panic!`の呼び出しを講義し、それから `Result<T, E>`を戻り値にする話をします。加えて、エラーからの回復を試みるか、実行を中止するか決定する際に考慮すべき事項についても、掘り下げましょう。

11.1 `panic!`で回復不能なエラー

時として、コードで悪いことが起きるものです。そして、それに対してできることは何もありません。このような場面で、Rust には `panic!`マクロが用意されています。`panic!`マクロが実行されると、プログラムは失敗のメッセージを表示し、スタックを巻き戻し掃除して、終了します。これが最もありふれて起こるのは、何らかのバグが検出された時であり、プログラマには、どうエラーを処理すればいいか明確ではありません。

11.1.1　パニックに対してスタックを巻き戻すか異常終了するか

標準では、パニックが発生すると、プログラムは**巻き戻し**を始めます。つまり、言語がスタックを遡り、遭遇した各関数のデータを片付けるということです。しかし、この遡りと片付けはすべきことが多くなります。対立案は、即座に異常終了し、片付けをせずにプログラムを終了させることです。そうなると、プログラムが使用していたメモリは、OS が片付ける必要があります。プロジェクトにおいて、実行可能ファイルを極力小さくする必要があれば、*Cargo.toml* ファイルの適切な `[profile]`欄に `panic = 'abort'`を追記することで、パニック時に巻き戻しから異常終了するように切り替えることができます。例として、リリースモード時に異常終了するようにしたければ、以下を追記してください:

```
[profile.release]
panic = 'abort'
```

単純なプログラムで `panic!`の呼び出しを試してみましょう:

ファイル名: src/main.rs

```
1 fn main() {
2     panic!("crash and burn");  //クラッシュして炎上
3 }
```

このプログラムを実行すると、以下のような出力を目の当たりにするでしょう:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25 secs
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:4
('main'スレッドはsrc/main.rs:2:4の「クラッシュして炎上」でパニックしました)
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

`panic!`の呼び出しが、最後の 3 行に含まれるエラーメッセージを発生させているのです。1 行目にパニックメッセージとソースコード中でパニックが発生した箇所を示唆しています: *src/main.rs:2:4* は、*src/main.rs* ファイルの 2 行目 4 文字目であることを示しています。

この場合、示唆される行は、自分のコードの一部で、その箇所を見に行けば、`panic!`マクロ呼び出しがあるわけです。`panic!`呼び出しが、自分のコードが呼び出しているコードの一部になっている可能性もあるわけです。エラーメッセージで報告されるファイル名と行番号が、結果的に `panic!`呼び出しに導いた自分のコードの行ではなく、`panic!`マクロが呼び出されている他人のコードになるでしょう。`panic!`呼び出しの発生元である関数のバックトレースを使用して、問題を起こしている自分のコードの箇所を割り出すことができます。バックトレースがどんなものか、次に議論しましょう。

11.1.2　panic!バックトレースを使用する

別の例を眺めて、自分のコードでマクロを直接呼び出す代わりに、コードに存在するバグにより、ライブラリで panic!呼び出しが発生するとどんな感じなのか確かめてみましょう。リスト 9-1 は、添え字でベクタの要素にアクセスを試みるあるコードです:

ファイル名: src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

リスト 9-1: ベクタの境界を超えて要素へのアクセスを試み、panic!を発生させる

ここでは、ベクタの 100 番目の要素 (添字は 0 始まりなので添字 99) にアクセスを試みていますが、ベクタには 3 つしか要素がありません。この場面では、Rust はパニックします。[]の使用は、要素を返すと想定されるものの、無効な番号を渡せば、ここで Rust が返せて正しいと思われる要素は何もないわけです。

他の言語 (C など) では、この場面で欲しいものではないにもかかわらず、まさしく要求したものを返そうとしてきます: メモリがベクタに属していないにもかかわらず、ベクタ内のその要素に対応するメモリ上の箇所にあるものを何か返してくるのです。これは、**バッファー外読み出し** (buffer overread; 脚注: 初めて見かけた表現。Rust 独自？) と呼ばれ、攻撃者が、配列の後に格納された読めるべきでないデータを読み出せるように添え字を操作できたら、セキュリティ脆弱性につながる可能性があります。

この種の脆弱性からプログラムを保護するために、存在しない番号の要素を読もうとしたら、Rust は実行を中止し、継続を拒みます。試して確認してみましょう:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', /checkout/src/liballoc/vec.rs:1555:10
('main'スレッドは、/checkout/src/liballoc/vec.rs:1555:10の
「境界外番号: 長さは3なのに、添え字は99です」でパニックしました)
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

このエラーは、自分のファイルではない *vec.rs* ファイルを指しています。標準ライブラリの Vec<T> の実装です。ベクタ vに対して []を使った時に走るコードは、*vec.rs* に存在し、ここで実際に panic! が発生しているのです。

その次の注釈行は、RUST_BACKTRACE環境変数をセットして、まさしく何が起き、エラーが発生したのかのバックトレースを得られることを教えてくれています。**バックトレース**とは、ここに至るまでに呼び出された全関数の一覧です。Rust のバックトレースも、他の言語同様に動作します: バックト

レースを読むコツは、頭からスタートして自分のファイルを見つけるまで読むことです。そこが、問題の根源になるのです。自分のファイルを言及している箇所以前は、自分のコードで呼び出したコードになります; 以後は、自分のコードを呼び出しているコードになります。これらの行には、Rust の核となるコード、標準ライブラリのコード、使用しているクレートなどが含まれるかもしれません。RUST_BACKTRACE環境変数を 0 以外の値にセットして、バックトレースを出力してみましょう。リスト 9-2 のような出力が得られるでしょう:

```
$ RUST_BACKTRACE=1 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', /
    checkout/src/liballoc/vec.rs:1555:10
stack backtrace:
   0: std::sys::imp::backtrace::tracing::imp::unwind_backtrace
             at /checkout/src/libstd/sys/unix/backtrace/tracing/gcc_s.rs:49
   1: std::sys_common::backtrace::_print
             at /checkout/src/libstd/sys_common/backtrace.rs:71
   2: std::panicking::default_hook::{{closure}}
             at /checkout/src/libstd/sys_common/backtrace.rs:60
             at /checkout/src/libstd/panicking.rs:381
   3: std::panicking::default_hook
             at /checkout/src/libstd/panicking.rs:397
   4: std::panicking::rust_panic_with_hook
             at /checkout/src/libstd/panicking.rs:611
   5: std::panicking::begin_panic
             at /checkout/src/libstd/panicking.rs:572
   6: std::panicking::begin_panic_fmt
             at /checkout/src/libstd/panicking.rs:522
   7: rust_begin_unwind
             at /checkout/src/libstd/panicking.rs:498
   8: core::panicking::panic_fmt
             at /checkout/src/libcore/panicking.rs:71
   9: core::panicking::panic_bounds_check
             at /checkout/src/libcore/panicking.rs:58
  10: <alloc::vec::Vec<T> as core::ops::index::Index<usize>>::index
             at /checkout/src/liballoc/vec.rs:1555
  11: panic::main
             at src/main.rs:4
  12: __rust_maybe_catch_panic
             at /checkout/src/libpanic_unwind/lib.rs:99
  13: std::rt::lang_start
             at /checkout/src/libstd/panicking.rs:459
             at /checkout/src/libstd/panic.rs:361
             at /checkout/src/libstd/rt.rs:61
  14: main
  15: __libc_start_main
  16: <unknown>
```

リスト 9-2: RUST_BACKTRACE環境変数をセットした時に表示される、panic!呼び出しが生成するバックトレース

出力が多いですね！ OS や Rust のバージョンによって、出力の詳細は変わる可能性があります。この情報とともに、バックトレースを得るには、デバッグシンボルを有効にしなければなりません。デバッグシンボルは、`--release`オプションなしで `cargo build`や `cargo run`を使用していれば、標準で有効になり、ここではそうなっています。

リスト 9-2 の出力で、バックトレースの 11 行目が問題発生箇所を指し示しています: *src/main.rs* の 4 行目です。プログラムにパニックしてほしくなければ、自分のファイルについて言及している最初の行で示されている箇所が、どのようにパニックを引き起こす値でこの箇所にたどり着いたか割り出すために調査を開始すべき箇所になります。バックトレースの使用法を模擬するためにわざとパニックするコードを書いたリスト 9-1 において、パニックを解消する方法は、3 つしか要素のないベクタの添字 99 の要素を要求しないことです。将来コードがパニックしたら、パニックを引き起こすどんな値でコードがどんな動作をしているのかと、代わりにコードは何をすべきなのかを算出する必要があるでしょう。

また、この章の後ほど、「`panic!`するか `panic!`するまいか」節で `panic!`とエラー状態を扱うのに `panic!`を使うべき時と使わぬべき時に戻ってきます。次は、`Result`を使用してエラーから回復する方法を見ましょう。

11.2　Result で回復可能なエラー

多くのエラーは、プログラムを完全にストップさせるほど深刻ではありません。時々、関数が失敗すると、容易に解釈し、対応できる理由によることがあります。例えば、ファイルを開こうとして、ファイルが存在しないために処理が失敗したら、プロセスを殺すのではなく、ファイルを作成したいことがあります。

第 2 章の「`Result`型で失敗する可能性に対処する」で `Result`enum が以下のように、`Ok`と `Err`の 2 値からなるよう定義されていることを思い出してください:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`T`と `E`は、ジェネリックな型引数です: ジェネリクスについて詳しくは、第 10 章で議論します。たった今知っておく必要があることは、`T`が成功した時に `Ok`バリアントに含まれて返される値の型を表すことと、`E`が失敗した時に `Err`バリアントに含まれて返されるエラーの型を表すことです。`Result`はこのようなジェネリックな型引数を含むので、標準ライブラリ上に定義されている `Result`型や関数などを、成功した時とエラーの値が異なるような様々な場面で使用できるのです。

関数が失敗する可能性があるために `Result`値を返す関数を呼び出しましょう: リスト 9-3 では、ファイルを開こうとしています:

ファイル名: src/main.rs

```
use std::fs::File;
```

```
2
3 fn main() {
4     let f = File::open("hello.txt");
5 }
```

リスト 9-3: ファイルを開く

File::openが Resultを返すとどう知るのでしょうか？ 標準ライブラリの API ドキュメントを参照することもできますし、コンパイラに尋ねることもできます！ fに関数の戻り値ではないと判明している型注釈を与えて、コードのコンパイルを試みれば、コンパイラは型が合わないと教えてくれるでしょう。そして、エラーメッセージは、fの実際の型を教えてくれるでしょう。試してみましょう！ File::openの戻り値の型は u32ではないと判明しているので、let f文を以下のように変更しましょう:

```
1 let f: u32 = File::open("hello.txt");
```

コンパイルしようとすると、以下のような出力が得られます:

```
error[E0308]: mismatched types
(エラー: 型が合いません)
 --> src/main.rs:4:18
  |
4 |     let f: u32 = File::open("hello.txt");
  |                  ^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
`std::result::Result`
  |
  = note: expected type `u32`
  (注釈: 予期した型は`u32`です)
             found type `std::result::Result<std::fs::File, std::io::Error>`
  (実際の型は`std::result::Result<std::fs::File, std::io::Error>`です)
```

これにより、File::open関数の戻り値の型は、Result<T, E>であることがわかります。ジェネリック引数の Tは、ここでは成功値の型 std::fs::Fileで埋められていて、これはファイルハンドルです。エラー値で使用されている Eの型は、std::io::Errorです。

この戻り値型は、File::openの呼び出しが成功し、読み込みと書き込みを行えるファイルハンドルを返す可能性があることを意味します。また、関数呼び出しは失敗もする可能性があります: 例えば、ファイルが存在しない可能性、ファイルへのアクセス権限がない可能性です。File::openには成功したか失敗したかを知らせる方法とファイルハンドルまたは、エラー情報を与える方法が必要なのです。この情報こそが Resultenum が伝達するものなのです。

File::openが成功した場合、変数 fの値はファイルハンドルを含む Okインスタンスになります。失敗した場合には、発生したエラーの種類に関する情報をより多く含む Errインスタンスが fの値になります。

リスト 9-3 のコードに追記をして File::openが返す値に応じて異なる動作をする必要があります。リスト 9-4 に基礎的な道具を使って Resultを扱う方法を一つ示しています。第 6 章で議論した match 式です。

ファイル名: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            // ファイルを開く際に問題がありました
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
```

リスト 9-4: match式を使用して返却される可能性のある Result列挙子を処理する

Optionenum のように、Resultenum とそのバリアントは、初期化処理でインポートされているので、matchアーム内で Okと Errバリアントの前に Result::を指定する必要がないことに注目してください。

ここでは、結果が Okの時に、Ok列挙子から中身の file値を返すように指示し、それからそのファイルハンドル値を変数 fに代入しています。matchの後には、ファイルハンドルを使用して読み込んだり書き込むことができるわけです。

matchのもう一つのアームは、File::openから Err値が得られたケースを処理しています。この例では、panic!マクロを呼び出すことを選択しています。カレントディレクトリに *hello.txt* というファイルがなく、このコードを走らせたら、panic!マクロからの以下のような出力を目の当たりにするでしょう:

```
thread 'main' panicked at 'There was a problem opening the file: Error { repr:
Os { code: 2, message: "No such file or directory" } }', src/main.rs:9:12
('main'スレッドは、src/main.rs:9:12の「ファイルを開く際に問題がありました: Error{ repr:
Os { code: 2, message: "そのような名前のファイルまたはディレクトリはありません"}}」でパ
    ニックしました)
```

通常通り、この出力は、一体何がおかしくなったのかを物語っています。

11.2.1　色々なエラーにマッチする

リスト 9-4 のコードは、File::openが失敗した理由にかかわらず panic!します。代わりにしたいことは、失敗理由によって動作を変えることです: ファイルが存在しないために File::openが失敗したら、ファイルを作成し、その新しいファイルへのハンドルを返したいです。他の理由 (例えばファイルを開く権限がなかったなど) で、File::openが失敗したら、リスト 9-4 のようにコードには panic!してほしいのです。リスト 9-5 を眺めてください。ここでは matchに別のアームを追加しています:

ファイル名: src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(ref error) if error.kind() == ErrorKind::NotFound => {
            match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => {
                    //ファイルを作成しようとしましたが、問題がありました
                    panic!(
                        "Tried to create file but there was a problem: {:?}",
                        e
                    )
                },
            }
        },
        Err(error) => {
            panic!(
                "There was a problem opening the file: {:?}",
                error
            )
        },
    };
}
```

リスト 9-5: 色々な種類のエラーを異なる方法で扱う

`File::open`が`Err`列挙子に含めて返す値の型は、`io::Error`であり、これは標準ライブラリで提供されている構造体です。この構造体には、呼び出すと`io::ErrorKind`値が得られる`kind`メソッドがあります。`io::ErrorKind`という enum は、標準ライブラリで提供されていて、`io`処理の結果発生する可能性のある色々な種類のエラーを表す列挙子があります。使用したい列挙子は、`ErrorKind::NotFound`で、これは開こうとしているファイルがまだ存在しないことを示唆します。

`if error.kind() == ErrorKind::Notfound`という条件式は、**マッチガード**と呼ばれます: アームのパターンをさらに洗練する`match`アーム上のおまけの条件式です。この条件式は、そのアームのコードが実行されるには真でなければいけないのです; そうでなければ、パターンマッチングは継続し、`match`の次のアームを考慮します。パターンの`ref`は、`error`がガード条件式にムーブされないように必要ですが、ただガード式に参照されます。`ref`を使用して`&`の代わりにパターン内で参照を作っている理由は、第 18 章で詳しく解説されるでしょう。手短に言えば、パターンの文脈において、`&`は参照にマッチし、その値を返すが、`ref`は値にマッチし、それへの参照を返すということなのです。

マッチガードで精査したい条件は、`error.kind()`により返る値が、`ErrorKind`enum の`NotFound`列挙子であるかということです。もしそうなら、`File::create`でファイル作成を試みます。ところが、`File::create`も失敗する可能性があるので、内部にも`match`文を追加する必要があるのです。ファイ

ルが開けないなら、異なるエラーメッセージが出力されるでしょう。外側の matchの最後のアームは同じままなので、ファイルが行方不明のエラー以外ならプログラムはパニックします。

11.2.2　エラー時にパニックするショートカット: unwrap と expect

matchの使用は、十分に仕事をしてくれますが、いささか冗長になり得る上、必ずしも意図をよく伝えるとは限りません。Result<T, E>型には、色々な作業をするヘルパーメソッドが多く定義されています。それらの関数の一つは、unwrapと呼ばれますが、リスト 9-4 で書いた match文と同じように実装された短絡メソッドです。Result値が Ok列挙子なら、unwrapは Okの中身を返します。Resultが Err列挙子なら、unwrapは panic!マクロを呼んでくれます。こちらが実際に動作している unwrapの例です:

ファイル名: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

このコードを *hello.txt* ファイルなしで走らせたら、unwrapメソッドが行う panic!呼び出しからのエラーメッセージを目の当たりにするでしょう:

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error {
repr: Os { code: 2, message: "No such file or directory" } }',
src/libcore/result.rs:906:4
('main'スレッドは、src/libcore/result.rs:906:4の
「`Err`値に対して`Result::unwrap()`が呼び出されました: Error{
repr: Os { code: 2, message: "そのようなファイルまたはディレクトリはありません" } }」で
    パニックしました)
```

別のメソッド expectは、unwrapに似ていますが、panic!のエラーメッセージも選択させてくれます。unwrapの代わりに expectを使用していいエラーメッセージを提供することは、意図を伝え、パニックの原因をたどりやすくしてくれます。expectの表記はこんな感じです:

ファイル名: src/main.rs

```
use std::fs::File;

fn main() {
    // hello.txtを開くのに失敗しました
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

expectを unwrapと同じように使用してます: ファイルハンドルを返したり、panic!マクロを呼び出しています。expectが panic!呼び出しで使用するエラーメッセージは、unwrapが使用するデフォルトの panic!メッセージではなく、expectに渡した引数になります。以下のようになります:

```
thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code:
2, message: "No such file or directory" } }', src/libcore/result.rs:906:4
```

このエラーメッセージは、指定したテキストの hello.txtを開くのに失敗しましたで始まっているので、コード内のどこでエラーメッセージが出力されたのかより見つけやすくなるでしょう。複数箇所でunwrapを使用していたら、ズバリどの unwrapがパニックを引き起こしているのか算出するのは、より時間がかかる可能性があります。パニックする unwrap呼び出しは全て、同じメッセージを出力するからです。

11.2.3　エラーを委譲する

失敗する可能性のある何かを呼び出す実装をした関数を書く際、関数内でエラーを処理する代わりに、呼び出し元がどうするかを決められるようにエラーを返すことができます。これはエラーの**委譲**として認知され、呼び出し元に自分のコードの文脈で利用可能なものよりも、エラーの処理法を規定する情報やロジックがより多くある箇所を制御してもらいます。

例えば、リスト 9-6 の関数は、ファイルからユーザ名を読み取ります。ファイルが存在しなかったり、読み込みできなければ、この関数はそのようなエラーを呼び出し元のコードに返します:

ファイル名: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

リスト 9-6: matchでエラーを呼び出し元のコードに返す関数

まずは、関数の戻り値型に注目しましょう: Result<String, io::Error>です。つまり、この関数は、Result<T, E>型の値を返しているということです。ここでジェネリック引数の Tは、具体型 Stringで埋められ、ジェネリック引数の Eは具体型 io::Errorで埋められています。この関数が何の問題もなく成功すれば、この関数を呼び出したコードは、String(関数がファイルから読み取ったユーザ

名) を保持する Ok値を受け取ります。この関数が何か問題に行き当たったら、呼び出し元のコードは io::Errorのインスタンスを保持する Err値を受け取り、この io::Errorは問題の内容に関する情報をより多く含んでいます。関数の戻り値の型に io::Errorを選んだのは、この関数本体で呼び出している失敗する可能性のある処理が両方ともたまたまこの型をエラー値として返すからです: File::open関数と read_to_stringメソッドです。

関数の本体は、File::open関数を呼び出すところから始まります。そして、リスト 9-4 の matchに似た matchで返ってくる Result値を扱い、Errケースに panic!を呼び出す代わりだけですが、この関数から早期リターンしてこの関数のエラー値として、File::openから得たエラー値を呼び出し元に渡し返します。File::openが成功すれば、ファイルハンドルを変数 fに保管して継続します。

さらに、変数 sに新規 Stringを生成し、fのファイルハンドルに対して read_to_stringを呼び出して、ファイルの中身を sに読み出します。File::openが成功しても、失敗する可能性があるので、read_to_stringメソッドも、Resultを返却します。その Resultを処理するために別の matchが必要になります: read_to_stringが成功したら、関数は成功し、今は Okに包まれた sに入っているファイルのユーザ名を返却します。read_to_stringが失敗したら、File::openの戻り値を扱った matchでエラー値を返したように、エラー値を返します。しかし、明示的に returnを述べる必要はありません。これが関数の最後の式だからです。

そうしたら、呼び出し元のコードは、ユーザ名を含む Ok値か、io::Errorを含む Err値を得て扱います。呼び出し元のコードがそれらの値をどうするかはわかりません。呼び出しコードが Err値を得たら、例えば、panic!を呼び出してプログラムをクラッシュさせたり、デフォルトのユーザ名を使ったり、ファイル以外の場所からユーザ名を検索したりできるでしょう。呼び出し元のコードが実際に何をしようとするかについて、十分な情報がないので、成功や失敗情報を全て委譲して適切に扱えるようにするのです。

Rust において、この種のエラー委譲は非常に一般的なので、Rust にはこれをしやすくする?演算子が用意されています。

11.2.3.1　エラー委譲のショートカット: ?演算子

リスト 9-7 もリスト 9-6 と同じ機能を有する read_username_from_fileの実装ですが、こちらははてなマーク演算子を使用しています:

ファイル名: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

リスト 9-7: ?でエラーを呼び出し元に返す関数

Result値の直後に置かれた?は、リスト 9-6 で Result値を処理するために定義した match式とほぼ同じように動作します。Resultの値が Okなら、Okの中身がこの式から返ってきて、プログラムは継続します。値が Errなら、returnキーワードを使ったかのように関数全体から Errの中身が返ってくるので、エラー値は呼び出し元のコードに委譲されます。

リスト 9-6 の match式と?には違いがあります: ?を使ったエラー値は、標準ライブラリの Fromトレイトで定義され、エラーの型を別のものに変換する from関数を通ることです。?が from関数を呼び出すと、受け取ったエラー型が現在の関数の戻り値型で定義されているエラー型に変換されます。これは、個々がいろんな理由で失敗する可能性があるのにも関わらず、関数が失敗する可能性を全て一つのエラー型で表現して返す時に有用です。各エラー型が from関数を実装して返り値のエラー型への変換を定義している限り、?が変換の面倒を自動的に見てくれます。

リスト 9-7 の文脈では、File::open呼び出し末尾の?は Okの中身を変数 fに返します。エラーが発生したら、?により関数全体から早期リターンし、あらゆる Err値を呼び出し元に与えます。同じ法則が read_to_string呼び出し末尾の?にも適用されます。

?演算子により定型コードの多くが排除され、この関数の実装を単純にしてくれます。リスト 9-8 で示したように、?の直後のメソッド呼び出しを連結することでさらにこのコードを短くすることさえもできます:

ファイル名: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

リスト 9-8: ?の後のメソッド呼び出しを連結する

sの新規 Stringの生成を関数の冒頭に移動しました; その部分は変化していません。変数 fを生成する代わりに、read_to_stringの呼び出しを直接 File::open("hello.txt")?の結果に連結させました。それでも、read_to_string呼び出しの末尾には?があり、File::openと read_to_string両方が成功したら、エラーを返すというよりもまだ sにユーザ名を含む Ok値を返します。機能もまたリスト 9-6 及び、9-7 と同じです; ただ単に異なるバージョンのよりプログラマフレンドリーな書き方なのです。

11.2.3.2　?演算子は、Result を返す関数でしか使用できない

?演算子は戻り値に Resultを持つ関数でしか使用できません。というのも、リスト 9-6 で定義した match式と同様に動作するよう、定義されているからです。Resultの戻り値型を要求する matchの部品

は、return Err(e)なので、関数の戻り値はこの returnと互換性を保つために Resultでなければならないのです。

main関数で?を使用したらどうなるか見てみましょう。main関数は、戻り値が ()でしたね:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt")?;
}
```

このコードをコンパイルすると、以下のようなエラーメッセージが得られます:

```
error[E0277]: the trait bound `(): std::ops::Try` is not satisfied
(エラー: `(): std::ops::Try`というトレイト境界が満たされていません)
 --> src/main.rs:4:13
  |
4 |     let f = File::open("hello.txt")?;
  |             ------------------------
  |             |
  |             the `?` operator can only be used in a function that returns
  `Result` (or another type that implements `std::ops::Try`)
  |             in this macro invocation
  |             (このマクロ呼び出しの`Result`(かまたは`std::ops::Try`を実装する他の型)
      を返す関数でしか`?`演算子は使用できません)
  |
  = help: the trait `std::ops::Try` is not implemented for `()`
  (助言: `std::ops::Try`トレイトは`()`には実装されていません)
  = note: required by `std::ops::Try::from_error`
  (注釈: `std::ops::Try::from_error`で要求されています)
```

このエラーは、?は Resultを返す関数でしか使用が許可されないと指摘しています。Resultを返さない関数では、Resultを返す別の関数を呼び出した時、?を使用してエラーを呼び出し元に委譲する可能性を生み出す代わりに、matchか Resultのメソッドのどれかを使う必要があるでしょう。

さて、panic!呼び出しや Resultを返す詳細について議論し終えたので、どんな場合にどちらを使うのが適切か決める方法についての話に戻りましょう。

11.3　panic!すべきかするまいか

では、panic!すべき時と Resultを返すべき時はどう決定すればいいのでしょうか？ コードがパニックしたら、回復する手段はありません。回復する可能性のある手段の有る無しに関わらず、どんなエラー場面でも panic!を呼ぶことはできますが、そうすると、呼び出す側のコードの立場に立ってこの場面は回復不能だという決定を下すことになります。Result値を返す決定をすると、決断を下すのではなく、呼び出し側に選択肢を与えることになります。呼び出し側は、場面に合わせて回復を試みることを決定したり、この場合の Err値は回復不能と断定して、panic!を呼び出し、回復可能だったエラーを回復不能に変換することもできます。故に、Resultを返却することは、失敗する可能性のある関数を定義する際には、いい第 1 選択肢になります。

稀な場面では、Resultを返すよりもパニックするコードを書く方がより適切になることもあります。例やプロトタイプコード、テストでパニックするのが適切な理由を探求しましょう。それからコンパイラは失敗することはないとわからないけれど、人間はできる場面を議論しましょう。そして、ライブラリコードでパニックするか決定する方法についての一般的なガイドラインで結論づけましょう。

11.3.1　例、プロトタイプコード、テスト

例を記述して何らかの概念を具体化している時、頑健なエラー処理コードも例に含むことは、例の明瞭さを欠くことになりかねません。例において、unwrapなどのパニックする可能性のあるメソッド呼び出しは、アプリケーションにエラーを処理してほしい方法へのプレースホルダーを意味していると理解され、これは残りのコードがしていることによって異なる可能性があります。

同様に、unwrapや expectメソッドは、エラーの処理法を決定する準備ができる前、プロトタイプの段階では、非常に便利です。それらにより、コードにプログラムをより頑健にする時の明らかなマーカーが残されるわけです。

メソッド呼び出しがテスト内で失敗したら、そのメソッドがテスト下に置かれた機能ではなかったとしても、テスト全体が失敗してほしいでしょう。panic!が、テストが失敗と印づけられる手段なので、unwrapや expectの呼び出しはスバリ起こるべきことです。

11.3.2　コンパイラよりもプログラマがより情報を持っている場合

Resultが Ok値であると確認する何らかの別のロジックがある場合、unwrapを呼び出すことは適切でしょうが、コンパイラは、そのロジックを理解はしません。それでも、処理する必要のある Resultは存在するでしょう: 呼び出している処理が何であれ、自分の特定の場面では論理的に起こり得なくても、一般的にまだ失敗する可能性はあるわけです。手動でコードを調査して Err列挙子は存在しないと確認できたら、unwrapを呼び出すことは完全に受容できることです。こちらが例です:

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1".parse().unwrap();
```

ハードコードされた文字列を構文解析することで IpAddrインスタンスを生成しています。プログラマには 127.0.0.1が有効な IP アドレスであることがわかるので、ここで unwrapを使用することは、受容可能なことです。しかしながら、ハードコードされた有効な文字列が存在することは、parseメソッドの戻り値型を変えることにはなりません: それでも得られるのは、Result値であり、コンパイラはまだ Err列挙子になる可能性があるかのように Resultを処理することを強制してきます。コンパイラは、この文字列が常に有効な IP アドレスであると把握できるほど利口ではないからです。プログラムにハードコードされるのではなく、IP アドレス文字列がユーザ起源でそれ故に**確かに**失敗する可能性がある場合、絶対に Resultをもっと頑健な方法で代わりに処理する必要があるでしょう。

11.3.3　エラー処理のガイドライン

コードが悪い状態に陥る可能性があるときにパニックさせるのは、アドバイスされることです。この文脈において、**悪い状態**とは、何らかの前提、保証、契約、不変性が破られたことを言い、例を挙げれば、無効な値、矛盾する値、行方不明な値がコードに渡されることと、さらに以下のいずれか一つ以上の状態であります:

- 悪い状態がときに起こるとは**想定**されないとき
- この時点以降、この悪い状態にないことを頼りにコードが書かれている場合
- 使用している型にこの情報をコード化するいい手段がないとき

誰かが自分のコードを呼び出して筋の通らない値を渡してきたら、最善の選択肢は `panic!`し、開発段階で修正できるように自分たちのコードにバグがあることをライブラリ使用者に通知することかもしれません。同様に自分の制御下にない外部コードを呼び出し、修正しようのない無効な状態を返すときに `panic!`はしばしば適切です。

悪い状態に達すると、どんなにコードをうまく書いても起こると予想されるが、`panic!`呼び出しをするよりもまだ、`Result`を返すほうがより適切です。例には、不正なデータを渡されたパーサーとか、訪問制限に引っかかったことを示唆するステータスを返す HTTP リクエストなどが挙げられます。このような場合には、呼び出し側が問題の処理方法を決定できるように `Result`を返してこの悪い状態を委譲して、失敗が予想される可能性であることを示唆するべきです。`panic!`呼び出すことは、これらのケースでは最善策ではないでしょう。

コードが値に対して処理を行う場合、コードはまず値が有効であることを確認し、値が有効でなければパニックするべきです。これはほぼ安全性上の理由によるものです: 不正なデータの処理を試みると、コードを脆弱性に晒す可能性があります。これが、境界外へのメモリアクセスを試みたときに標準ライブラリが `panic!`を呼び出す主な理由です: 現在のデータ構造に属しないメモリにアクセスを試みることは、ありふれたセキュリティ問題なのです。関数にはしばしば**契約**が伴います: 入力が特定の条件を満たすときのみ、振る舞いが保証されるのです。契約が侵されたときにパニックすることは、道理が通っています。なぜなら、契約侵害は常に呼び出し側のバグを示唆し、呼び出し側に明示的に処理してもらう必要のある種類のエラーではないからです。実際に、呼び出し側が回復する合理的な手段はありません; 呼び出し側の**プログラマ**がコードを修正する必要があるのです。関数の契約は、特に侵害がパニックを引き起こす際には、関数の API ドキュメント内で説明されているべきです。

ですが、全ての関数でたくさんのエラーチェックを行うことは冗長で煩わしいことでしょう。幸運にも、Rust の型システム (故にコンパイラが行う型精査) を使用して多くの精査を行ってもらうことができます。関数の引数に特定の型があるなら、有効な値があるとコンパイラがすでに確認していることを把握して、コードのロジックに進むことができます。例えば、`Option`以外の型がある場合、プログラムは、**何もない**ではなく**何かある**と想定します。そうしたらコードは、`Some`と `None`列挙子の 2 つの場合を処理する必要がなくなるわけです: 確実に値があるという可能性しかありません。関数に

何もないことを渡そうとしてくるコードは、コンパイルが通りもしませんので、その場合を実行時に精査する必要はないわけです。別の例は、u32のような符号なし整数を使うことであり、この場合、引数は負には絶対にならないことが確認されます。

11.3.4　有効化のために独自の型を作る

Rust の型システムを使用して有効な値があると確認するというアイディアを一歩先に進め、有効化のために独自の型を作ることに目を向けましょう。第 2 章の数当てゲームで、コードがユーザに 1 から 100 までの数字を推測するよう求めたことを思い出してください。秘密の数字と照合する前にユーザの推測がそれらの値の範囲にあることを全く確認しませんでした; 推測が正であることしか確認しませんでした。この場合、結果はそれほど悲惨なものではありませんでした:「大きすぎ」、「小さすぎ」という出力は、それでも正しかったでしょう。ユーザを有効な推測に導き、ユーザが範囲外の数字を推測したり、例えばユーザが文字を代わりに入力したりしたときに別の挙動をするようにしたら、有益な改善になるでしょう。

これをする一つの方法は、ただの u32の代わりに i32として推測をパースし、負の数になる可能性を許可し、それから数字が範囲に収まっているというチェックを追加することでしょう。そう、以下のように:

```
loop {
    // --snip--

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("The secret number will be between 1 and 100.");
        continue;
    }

    match guess.cmp(&secret_number) {
    // --snip--
}
```

この if式が、値が範囲外かどうかをチェックし、ユーザに問題を告知し、continueを呼び出してループの次の繰り返しを始め、別の推測を求めます。if式の後、guessは 1 から 100 の範囲にあると把握して、guessと秘密の数字の比較に進むことができます。

ところが、これは理想的な解決策ではありません: プログラムが 1 から 100 の範囲の値しか処理しないことが間違いなく、肝要であり、この要求がある関数の数が多ければ、このようなチェックを全関数で行うことは、面倒でパフォーマンスにも影響を及ぼす可能性があるでしょう。

代わりに、新しい型を作ってバリデーションを関数内に閉じ込め、バリデーションを全箇所で繰り返すのではなく、その型のインスタンスを生成することができます。そうすれば、関数がその新しい

型をシグニチャに用い、受け取った値を自信を持って使用することは安全になります。リスト 9-9 に、new関数が 1 から 100 までの値を受け取った時のみ、Guessのインスタンスを生成する Guess型を定義する一つの方法を示しました:

```
pub struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }

    pub fn value(&self) -> u32 {
        self.value
    }
}
```

リスト 9-9: 値が 1 から 100 の場合のみ処理を継続する Guess型

まず、u32型の valueをフィールドに持つ Guessという名前の構造体を定義しています。ここに数値が保管されます。

それから Guessに Guess値のインスタンスを生成する newという名前の関連関数を実装しています。new関数は、u32型の valueという引数を取り、Guessを返すように定義されています。new関数の本体のコードは、valueをふるいにかけ、1 から 100 の範囲であることを確かめます。valueがふるいに引っかかったら、panic!呼び出しを行います。これにより、呼び出しコードを書いているプログラマに、修正すべきバグがあると警告します。というのも、この範囲外の valueで Guessを生成することは、Guess::newが頼りにしている契約を侵害するからです。Guess::newがパニックするかもしれない条件は、公開されている API ドキュメントで議論されるべきでしょう; あなたが作成する API ドキュメントで panic!の可能性を示唆する、ドキュメントの規約は、第 14 章で解説します。valueが確かにふるいを通ったら、valueフィールドが value引数にセットされた新しい Guessを作成して返します。

次に、selfを借用し、他に引数はなく、u32を返す valueというメソッドを実装します。この類のメソッドは時に**ゲッター**と呼ばれます。目的がフィールドから何らかのデータを得て返すことだからです。この公開メソッドは、Guess構造体の valueフィールドが非公開なので、必要になります。valueフィールドが非公開なことは重要であり、そのために Guess構造体を使用するコードは、直接 valueをセットすることが叶わないのです: モジュール外のコードは、Guess::new関数を使用して Guessのインスタンスを生成し**なければならず**、それにより、Guess::new関数の条件式でチェックされていない valueが Guessに存在する手段はないことが保証されるわけです。

そうしたら、引数を一つ持つか、1 から 100 の範囲の数値のみを返す関数は、シグニチャで u32で

はなく、Guessを取るか返し、本体内で追加の確認を行う必要はなくなると宣言できるでしょう。

11.4　まとめ

Rust のエラー処理機能は、プログラマがより頑健なコードを書く手助けをするように設計されています。panic!マクロは、プログラムが処理できない状態にあり、無効だったり不正な値で処理を継続するのではなく、プロセスに処理を中止するよう指示することを通知します。Resultenum は、Rust の型システムを使用して、コードが回復可能な方法で処理が失敗するかもしれないことを示唆します。Resultを使用して、呼び出し側のコードに成功や失敗する可能性を処理する必要があることも教えます。適切な場面で panic!や Resultを使用することで、必然的な問題の眼前でコードの信頼性を上げてくれます。

今や、標準ライブラリが Optionや Resultenum などでジェネリクスを有効活用するところを目の当たりにしたので、ジェネリクスの動作法と自分のコードでの使用方法について語りましょう。

第 12 章

ジェネリック型、トレイト、ライフタイム

全てのプログラミング言語には、概念の重複を効率的に扱う道具があります。Rust において、そのような道具の一つが**ジェネリクス**です。ジェネリクスは、具体型や他のプロパティの抽象的な代役です。コード記述の際、コンパイルやコード実行時に、ジェネリクスの位置に何が入るかを知ることなく、ジェネリクスの振る舞いや他のジェネリクスとの関係を表現できるのです。

関数が未知の値の引数を取り、同じコードを複数の具体的な値に対して走らせるように、`i32`や`String`などの具体的な型の代わりに何かジェネリックな型の引数を取ることができます。実際、第 6 章で `Option<T>`、第 8 章で `Vec<T>`と `HashMap<K, V>`、第 9 章で `Result<T, E>`を既に使用しました。この章では、独自の型、関数、メソッドをジェネリクスとともに定義する方法を探求します！

まず、関数を抽出して、コードの重複を減らす方法を確認しましょう。次に同じテクニックを活用して、引数の型のみが異なる 2 つの関数からジェネリックな関数を生成します。また、ジェネリックな型を構造体や enum 定義で使用する方法も説明します。

それから、トレイトを使用して、ジェネリックな方法で振る舞いを定義する方法を学びます。すると、ジェネリックな型のあるトレイトを組み合わせてただ単にどんな型に対してもとは対照的に、ジェネリックな型を特定の振る舞いのある型のみに制限することができます。

最後に、ライフタイムを議論します。ライフタイムとは、コンパイラに参照がお互いにどう関係しているかの情報を与える 1 種のジェネリクスです。ライフタイムのおかげでコンパイラに参照が有効であることを確認してもらうことを可能にしつつ、多くの場面で値を借用できます。

12.1 関数を抽出することで重複を取り除く

ジェネリクスの記法に飛び込む前にまずは、関数を抽出することでジェネリックな型が関わらない重複を取り除く方法を見ましょう。そして、このテクニックを適用してジェネリックな関数を抽出す

るのです！ 重複したコードを認識して関数に抽出できるのと同じように、ジェネリクスを使用できる重複コードも認識し始めるでしょう。

リスト 10-1 に示したように、リスト内の最大値を求める短いプログラムを考えてください:

ファイル名: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    // 最大値は{}です
    println!("The largest number is {}", largest);
#   assert_eq!(largest, 100);
}
```

リスト 10-1: 数字のリストから最大値を求めるコード

このコードは、整数のリストを変数 number_listに格納し、リストの最初の数字を largestという変数に配置しています。それからリストの数字全部を走査し、現在の数字が largestに格納された数値よりも大きければ、その変数の値を置き換えます。ですが、現在の数値が今まで見た最大値よりも小さければ、変数は変わらず、コードはリストの次の数値に移っていきます。リストの数値全てを吟味した後、largestは最大値を保持しているはずで、今回は 100 になります。

2 つの異なる数値のリストから最大値を発見するには、リスト 10-1 のコードを複製し、プログラムの異なる 2 箇所で同じロジックを使用できます。リスト 10-2 のようにね:

ファイル名: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = number_list[0];

```

```
    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

リスト 10-2: 2 つの数値のリストから最大値を探すコード

このコードは動くものの、コードを複製することは退屈ですし、間違いも起きやすいです。また、複数箇所のコードを更新しないと変更できません。

この重複を排除するには、引数で与えられた整数のどんなリストに対しても処理が行える関数を定義して抽象化できます。この解決策によりコードがより明確になり、リストの最大値を探すという概念を抽象的に表現させてくれます。

リスト 10-3 では、最大値を探すコードを `largest`という関数に抽出しました。リスト 10-1 のコードは、たった 1 つの特定のリストからだけ最大値を探せますが、それとは異なり、このプログラムは 2 つの異なるリストから最大値を探せます:

ファイル名: src/main.rs

```
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
#    assert_eq!(result, 100);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
#    assert_eq!(result, 6000);
}
```

リスト 10-3: 2 つのリストから最大値を探す抽象化されたコード

`largest`関数には `list`と呼ばれる引数があり、これは、関数に渡す可能性のあるあらゆる `i32`値の

具体的なスライスを示します。結果的に、関数呼び出しの際、コードは渡した特定の値に対して走るのです。

まとめとして、こちらがリスト 10-2 のコードからリスト 10-3 に変更するのに要したステップです:

1. 重複したコードを認識する。
2. 重複コードを関数本体に抽出し、コードの入力と戻り値を関数シグニチャで指定する。
3. 重複したコードの 2 つの実体を代わりに関数を呼び出すように更新する。

次は、この同じ手順をジェネリクスでも踏んで異なる方法でコードの重複を減らします。関数本体が特定の値ではなく抽象的な `list`に対して処理できたのと同様に、ジェネリクスは抽象的な型に対して処理するコードを可能にしてくれます。

例えば、関数が 2 つあるとしましょう: 1 つは `i32`値のスライスから最大の要素を探し、1 つは `char`値のスライスから最大要素を探します。この重複はどう排除するのでしょうか？ 答えを見つけましょう！

12.2　ジェネリックなデータ型

関数シグニチャや構造体などの要素の定義を生成するのにジェネリクスを使用することができ、それはさらに他の多くの具体的なデータ型と使用することもできます。まずは、ジェネリクスで関数、構造体、enum、メソッドを定義する方法を見ましょう。それから、ジェネリクスがコードのパフォーマンスに与える影響を議論します。

12.2.1　関数定義では

ジェネリクスを使用する関数を定義する時、通常、引数や戻り値のデータ型を指定する関数のシグニチャにジェネリクスを配置します。そうすることでコードがより柔軟になり、コードの重複を阻止しつつ、関数の呼び出し元により多くの機能を提供します。

`largest`関数を続けます。リスト 10-4 はどちらもスライスから最大値を探す 2 つの関数を示しています:

ファイル名: src/main.rs

```
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

```

```
fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);
#    assert_eq!(result, 100);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
#    assert_eq!(result, 'y');
}
```

リスト 10-4: 名前とシグニチャの型のみが異なる 2 つの関数

`largest_i32`関数は、リスト 10-3 で抽出したスライスから最大の `i32`を探す関数です。`largest_char`関数は、スライスから最大の `char`を探します。関数本体には同じコードがあるので、単独の関数にジェネリックな型引数を導入してこの重複を排除しましょう。

これから定義する新しい関数の型を引数にするには、ちょうど関数の値引数のように型引数に名前をつける必要があります。型引数の名前にはどんな識別子も使用できますが、`T`を使用します。というのも、慣習では、Rust の引数名は短く (しばしばたった 1 文字になります)、Rust の型の命名規則がキャメルケースだからです。”type” の省略形なので、`T`が多くの Rust プログラマの規定の選択なのです。

関数の本体で引数を使用すると、コンパイラがその名前の意味を知れるようにシグニチャでその引数名を宣言しなければなりません。同様に、型引数名を関数シグニチャで使用する際には、使用する前に型引数名を宣言しなければなりません。ジェネリックな `largest`関数を定義するために、型名宣言を山カッコ (<>) 内、関数名と引数リストの間に配置してください。こんな感じに:

```
fn largest<T>(list: &[T]) -> T {
```

この定義は以下のように解読します: 関数 `largest`は、なんらかの型 `T`に関してジェネリックであると。この関数には `list`という引数が 1 つあり、これは型 `T`の値のスライスです。`largest`関数は同じ`T`型の値を返します。

リスト 10-5 は、シグニチャにジェネリックなデータ型を使用して largest関数定義を組み合わせたものを示しています。このリストはさらに、この関数を i32値か char値のどちらかで呼べる方法も表示しています。このコードはまだコンパイルできないことに注意してください。ですが、この章の後ほど修正します。

ファイル名: src/main.rs

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

リスト 10-5: ジェネリックな型引数を使用するものの、まだコンパイルできない largest関数の定義

直ちにこのコードをコンパイルしたら、以下のようなエラーが出ます:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
(エラー: 2項演算`>`は、型`T`に適用できません)
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
  = note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
  (注釈: `std::cmp""PartialOrd`の実装が`T`に対して不足しています)
```

注釈が std::cmp::PartialOrdに触れていて、これは、**トレイト**です。トレイトについては、次の節で語ります。とりあえず、このエラーは、largestの本体は、Tがなりうる全ての可能性のある型に対して動作しないと述べています。本体で型 Tの値を比較したいので、値が順序付け可能な型のみしか使用できないのです。比較を可能にするために、標準ライブラリには型に実装できる std::cmp::PartialOrd トレイトがあります (このトレイトについて詳しくはおまけ C「継承可能トレイト」を参照されたし)。

ジェネリックな型が特定のトレイトを持つと指定する方法は「トレイト境界」節で習うでしょうが、先にジェネリックな型引数を使用する他の方法を探求しましょう。

12.2.2　構造体定義では

また、構造体を定義して<>記法で 1 つ以上のフィールドにジェネリックな型引数を使用することもできます。リスト 10-6 は、Point<T>構造体を定義してあらゆる型の xと y座標を保持する方法を示しています:

ファイル名: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

リスト 10-6: 型 Tの xと y値を保持する Point<T>構造体

構造体定義でジェネリクスを使用する記法は、関数定義のものと似ています。まず、山カッコ内に型引数の名前を構造体名の直後に宣言します。そして、そうしていなければ具体的なデータ型を記述する構造体定義の箇所にジェネリックな型を使用できます。

1 つのジェネリックな型を使用して Point<T>を定義したので、この定義は、Point<T>構造体がなんらかの型 Tに関して、ジェネリックであると述べていてその型がなんであれ、xと yのフィールドは**両方**その同じ型になっていることに注意してください。リスト 10-7 のように、異なる型の値のある Point<T>のインスタンスを生成すれば、コードはコンパイルできません:

ファイル名: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

リスト 10-7: どちらも同じジェネリックなデータ型 Tなので、xと yというフィールドは同じ型でなければならない

この例で、xに整数値 5を代入すると、この Point<T>のインスタンスに対するジェネリックな型 Tは整数になるとコンパイラに知らせます。それから yに 4.0を指定する時に、このフィールドは xと同じ型と定義したはずなので、このように型不一致エラーが出ます:

```
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integral variable, found
floating-point variable
  |
  = note: expected type `{integer}`
             found type `{float}`
```

xと yが両方ジェネリックだけれども、異なる型になり得る Point構造体を定義するには、複数のジェネリックな型引数を使用できます。例えば、リスト 10-8 では、Pointの定義を変更して、型 Tと Uに関してジェネリックにし、xが型 Tで、yが型 Uになります:

ファイル名: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

リスト 10-8: Point<T, U>は 2 つの型に関してジェネリックなので、xと yは異なる型の値になり得る

もう、示された Pointインスタンスは全部許可されます！ 所望の数だけ定義でジェネリックな型引数を使用できますが、数個以上使用すると、コードが読みづらくなります。コードで多くのジェネリックな型が必要な時は、コードの小分けが必要なサインかもしれません。

12.2.3　enum 定義では

構造体のように、列挙子にジェネリックなデータ型を保持する enum を定義することができます。標準ライブラリが提供し、第 6 章で使用した Option<T>enum に別の見方をしましょう:

```
enum Option<T> {
    Some(T),
    None,
}
```

この定義はもう、あなたにとって道理が通っているはずです。ご覧の通り、Option<T>は、型 Tに関してジェネリックで 2 つの列挙子を持つ enum です: その列挙子は、型 Tの値を保持する Someと、値を何も保持しない Noneです。Option<T>enum を使用することで、オプショナルな値があるという抽

象的な概念を表現でき、Option<T>はジェネリックなので、オプショナルな値の型に関わらず、この抽象を使用できます。

enum も複数のジェネリックな型を使用できます。第 9 章で使用した Resultenum の定義が 1 例です:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Resultenum は 2 つの型 T、Eに関してジェネリックで、2 つの列挙子があります: 型 Tの値を保持する Okと、型 Eの値を保持する Errです。この定義により、Resultenum を成功する (なんらかの型 T の値を返す) か、失敗する (なんらかの型 Eのエラーを返す) 可能性のある処理があるあらゆる箇所に使用するのが便利になります。事実、ファイルを開くのに成功した時に Tに型 std::fs::Fileが入り、ファイルを開く際に問題があった時に Eに型 std::io::Errorが入ったものが、リスト 9-3 でファイルを開くのに使用したものです。

自分のコード内で保持している値の型のみが異なる構造体や enum 定義の場面を認識したら、代わりにジェネリックな型を使用することで重複を避けることができます。

12.2.4　メソッド定義では

第 5 章のように、定義にジェネリックな型を含むメソッドを構造体や enum に実装できます。リスト 10-9 は、リスト 10-6 で定義した Point<T>構造体に xというメソッドを実装したものを示しています:

ファイル名: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

リスト 10-9: 型 Tの xフィールドへの参照を返す xというメソッドを Point<T>構造体に実装する

ここで、フィールド xのデータへの参照を返す xというメソッドを Point<T>に定義しました。

型 Point<T>にメソッドを実装していると指定するために Tを使用できるよう implの直後に宣言しなければならないことに注意してください。implの後に Tをジェネリックな型として宣言することで、コンパイラは、Pointの山カッコ内の型が、具体的な型ではなくジェネリックな型であることを認識できるのです。

例えば、あらゆるジェネリックな型とともに Point<T>インスタンスではなく、Point<f32>だけにメソッドを実装することもできるでしょう。リスト 10-10 では、具体的な型 f32を使用しています。つまり、implの後に型を宣言しません:

```
# struct Point<T> {
#     x: T,
#     y: T,
# }
#
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

リスト 10-10: ジェネリックな型引数 Tに対して特定の具体的な型がある構造体にのみ適用される implブロック

このコードは、Point<f32>には distance_from_originというメソッドが存在するが、Tが f32ではない Point<T>の他のインスタンスにはこのメソッドが定義されないことを意味します。このメソッドは、この点が座標 (0.0, 0.0) の点からどれだけ離れているかを測定し、浮動小数点数にのみ利用可能な数学的処理を使用します。

構造体定義のジェネリックな型引数は、必ずしもその構造体のメソッドシグニチャで使用するものと同じにはなりません。例を挙げれば、リスト 10-11 は、リスト 10-8 の Point<T, U>にメソッド mixupを定義しています。このメソッドは、他の Pointを引数として取り、この引数は mixupを呼び出している selfの Pointとは異なる型の可能性があります。このメソッドは、(型 Tの)selfの Pointの x値と渡した (型 Wの)Pointの y値から新しい Pointインスタンスを生成します:

ファイル名: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

```

```
fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c'};

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

リスト 10-11: 構造体定義とは異なるジェネリックな型を使用するメソッド

mainで、x(値は 5) に i32、y(値は 10.4) に f64を持つ Pointを定義しました。p2変数は、x(値は "Hello") に文字列スライス、y(値は c) に charを持つ Point構造体です。引数 p2で p1に mixupを呼び出すと、p3が得られ、xは i32になります。xは p1由来だからです。p3変数は、yに charになります。y は p2由来だからです。println!マクロの呼び出しは、p3.x = 5, p3.y = cと出力するでしょう。

この例の目的は、一部のジェネリックな引数は implで宣言され、他の一部はメソッド定義で宣言される場面をデモすることです。ここで、ジェネリックな引数 Tと Uは implの後に宣言されています。構造体定義にはまるからです。ジェネリックな引数 Vと Wは fn mixupの後に宣言されています。何故なら、このメソッドにしか関係ないからです。

12.2.5　ジェネリクスを使用したコードのパフォーマンス

ジェネリックな型引数を使用すると、実行時にコストが発生するかあなたは不思議に思っている可能性があります。コンパイラが、ジェネリクスを具体的な型がある時よりもジェネリックな型を使用したコードを実行するのが遅くならないように実装しているのは、嬉しいお知らせです。

コンパイラはこれをジェネリクスを使用しているコードの単相化をコンパイル時に行うことで達成しています。**単相化** (monomorphization) は、コンパイル時に使用されている具体的な型を入れることで、ジェネリックなコードを特定のコードに変換する過程のことです。

この過程において、コンパイラは、リスト 10-5 でジェネリックな関数を生成するために使用した手順と真逆のことをしています: コンパイラは、ジェネリックなコードが呼び出されている箇所全部を見て、ジェネリックなコードが呼び出されている具体的な型のコードを生成するのです。

標準ライブラリの Option<T>enum を使用する例でこれが動作する方法を見ましょう:

```
let integer = Some(5);
let float = Some(5.0);
```

コンパイラがこのコードをコンパイルすると、単相化を行います。その過程で、コンパイラは Option<T>のインスタンスに使用された値を読み取り、2 種類の Option<T>を識別します: 一方は i32 で、もう片方は f64です。そのように、コンパイラは、Option<T>のジェネリックな定義を Option_i32 と Option_f64に展開し、それにより、ジェネリックな定義を特定の定義と置き換えます。

単相化されたバージョンのコードは、以下のようになります。ジェネリックな Option<T>が、コンパイラが生成した特定の定義に置き換えられています:

ファイル名: src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Rust では、ジェネリックなコードを各インスタンスで型を指定したコードにコンパイルするので、ジェネリクスを使用することに対して実行時コストを払うことはありません。コードを実行すると、それぞれの定義を手作業で複製した時のように振る舞います。単相化の過程により、Rust のジェネリクスは実行時に究極的に効率的になるのです。

12.3　トレイト: 共通の振る舞いを定義する

トレイトにより、Rust コンパイラに特定の型に存在し、他の型と共有できる機能について知らせます。トレイトを使用して共通の振る舞いを抽象的に定義できます。トレイト境界を使用して、あるジェネリックが特定の振る舞いのあるあらゆる型になり得ることを指定できます。

> 注釈: 違いはあるものの、トレイトは他の言語でよくインターフェイスと呼ばれる機能に類似しています。

12.3.1　トレイトを定義する

型の振る舞いは、その型に対して呼び出せるメソッドから構成されます。異なる型は、それらの型全部に対して同じメソッドを呼び出せたら、同じ振る舞いを共有します。トレイト定義は、メソッドシグニチャを一緒くたにしてなんらかの目的を達成するのに必要な一連の振る舞いを定義する手段です。

例えば、いろんな種類や量のテキストを保持する複数の構造体があるとしましょう: 特定の場所で送られる新しいニュースを保持する `NewsArticle`と、新規ツイートか、リツイートか、はたまた他のツイートへのリプライなのかを示すメタデータを伴う最大で 280 文字までの `Tweet`です。

`NewsArticle`や `Tweet`インスタンスに格納される可能性のあるデータの総括を表示するメディア総括ライブラリを作成したいです。このために、各型からまとめが必要で、インスタンスに対して `summarize`メソッドを呼び出すことでそのまとめを要求する必要があります。リスト 10-12 は、この

振る舞いを表現する Summaryトレイトの定義を表示しています:

ファイル名: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

リスト 10-12: summarizeメソッドで提供される振る舞いからなる Summaryトレイト

ここでは、traitキーワード、それからトレイト名を使用してトレイトを定義していて、その名前は今回の場合、Summaryです。波括弧の中にこのトレイトを実装する型の振る舞いを記述するメソッドシグニチャを定義し、今回の場合は、fn summarize(&self) -> Stringです。

メソッドシグニチャの後に、波括弧内に実装を提供する代わりに、セミコロンを使用しています。このトレイトを実装する型はそれぞれ、メソッドの本体に独自の振る舞いを提供しなければなりません。コンパイラにより、Summaryトレイトを保持するあらゆる型に、このシグニチャと全く同じメソッド summarizeが定義されていることが、強制されます。

トレイトには、本体に複数のメソッドを含むことができます: メソッドシグニチャは行ごとに列挙され、各行はセミコロンで終止します。

12.3.2　トレイトを型に実装する

今や Summaryトレイトで欲しい振る舞いを定義したので、メディア総括機で型に実装することができます。リスト 10-13 は見出し、著者、場所を使用して summarizeの戻り値を生成する NewsArticle構造体の Summaryトレイト実装を示しています。Tweet構造体に関しては、ツイートの内容が既に 280 文字に限定されていることを想定して、summarizeをユーザ名にツイート全体のテキストが続く形で定義します。

ファイル名: src/lib.rs

```
# pub trait Summary {
#     fn summarize(&self) -> String;
# }
#
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
```

```
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

リスト 10-13: Summaryトレイトを NewsArticleと Tweet型に実装する

型にトレイトを実装することは、普通のメソッドを実装することに似ています。違いは、implの後に、実装したいトレイトの名前を置き、それから forキーワード、さらにトレイトの実装対象の型の名前を指定することです。implブロック内に、トレイト定義で定義したメソッドシグニチャを置きます。各シグニチャの後にセミコロンを追記するのではなく、波括弧を使用し、メソッド本体に特定の型のトレイトのメソッドに欲しい特定の振る舞いを入れます。

トレイトを実装後、普通の関数同様に NewsArticleや Tweetのインスタンスに対してメソッドを呼び出せます。こんな感じで:

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    // もちろん、ご存知かもしれないように ね、みなさん
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

このコードは、1 new tweet: horse_ebooks: of course, as you probably already know, peopleと出力します。

リスト 10-13 で Summaryトレイトと NewArticle、Tweet型を同じ *lib.rs* に定義したので、全部同じスコープにあることに注目してください。この *lib.rs* を aggregatorと呼ばれるクレート専用にして、誰か他の人が私たちのクレートの機能を活用して自分のライブラリのスコープに定義された構造体に Summaryトレイトを実装したいとしましょう。まず、トレイトをスコープにインポートする必要があるでしょう。use aggregator::Summary;と指定してそれを行い、これにより、自分の型に Summaryを実装することが可能になります。Summaryトレイトは、他のクレートが実装するためには、公開トレイトである必要があり、ここでは、リスト 10-12 の traitの前に、pubキーワードを置いたのでそうなっています。

トレイト実装で注意すべき制限の 1 つは、トレイトか対象の型が自分のクレートにローカルである時のみ、型に対してトレイトを実装できるということです。例えば、Displayのような標準ライブラリのトレイトを aggregatorクレートの機能の一部として、Tweetのような独自の型に実装できます。型

Tweetが aggregatorクレートにローカルだからです。また、Summaryを aggregatorクレートで Vec<T>に対して実装することもできます。トレイト Summaryは、aggregatorクレートにローカルだからです。

しかし、外部のトレイトを外部の型に対して実装することはできません。例として、aggregatorクレート内で Vec<T>に対して Displayトレイトを実装することはできません。Displayと Vec<T>は標準ライブラリで定義され、aggregatorクレートにローカルではないからです。この制限は、**コヒーレンス** (coherence) あるいは、具体的に**オーファンルール** (orphan rule) と呼ばれるプログラムの特性の一部で、親の型が存在しないためにそう命名されました。この規則により、他の人のコードが自分のコードを壊したり、その逆が起きないことを保証してくれます。この規則がなければ、2 つのクレートが同じ型に対して同じトレイトを実装できてしまい、コンパイラはどちらの実装を使うべきかわからなくなってしまうでしょう。

12.3.3　デフォルト実装

時として、全ての型の全メソッドに対して実装を必要とするのではなく、トレイトの全てあるいは一部のメソッドに対してデフォルトの振る舞いがあると有用です。そうすれば、特定の型にトレイトを実装する際、各メソッドのデフォルト実装を保持するかオーバーライドできるわけです。

リスト 10-14 は、リスト 10-12 のように、メソッドシグニチャだけを定義するのではなく、Summaryトレイトの summarizeメソッドにデフォルトの文字列を指定する方法を示しています:

ファイル名: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
        // (もっと読む)
        String::from("(Read more...)")
    }
}
```

リスト 10-14: summarizeメソッドのデフォルト実装がある Summaryトレイトの定義

デフォルト実装を使用して独自の実装を定義するのではなく、NewsArticleのインスタンスをまとめるには、impl Summary for NewsArticle {}と空の implブロックを指定します。

たとえ、最早 NewsArticleに直接 summarizeメソッドを定義することはなくても、デフォルト実装を提供し、NewsArticleは Summaryトレイトを実装すると指定しました。結果的に、それでも、NewsArticleのインスタンスに対して summarizeメソッドを呼び出すことができます。このように:

```
let article = NewsArticle {
    // ペンギンチームがスタンレーカップチャンピオンシップを勝ち取る！
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    // ピッツバーグ・ペンギンが再度NHLで最強のホッケーチームになった
    content: String::from("The Pittsburgh Penguins once again are the best
    hockey team in the NHL."),
};

```

```
// 新しい記事が利用可能です！ {}
println!("New article available! {}", article.summarize());
```

このコードは、New article available! (Read more...)と出力します。

summarizeにデフォルト実装を作っても、リスト 10-13 の Tweetの Summary実装を変える必要はありません。理由は、デフォルト実装をオーバーライドする記法がデフォルト実装のないトレイトメソッドを実装する記法と同じだからです。

デフォルト実装は、他のデフォルト実装がないメソッドでも呼び出すことができます。このように、トレイトは多くの有用な機能を提供しつつ、実装者に僅かな部分だけ指定してもらう必要しかないのです。例えば、Summaryトレイトを実装が必須の summarize_authorメソッドを持つように定義し、それから summarize_authorメソッドを呼び出すデフォルト実装のある summarizeメソッドを定義することもできます:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        // {}からもっと読む
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

このバージョンの Summaryを使用するには、型にトレイトを実装する際に summarize_authorを定義する必要だけあります:

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

summarize_author定義後、Tweet構造体のインスタンスに対して summarizeを呼び出せ、summarizeのデフォルト実装は、提供済みの summarize_authorの定義を呼び出すでしょう。summarize_authorを実装したので、追加のコードを書く必要なく、Summaryトレイトは、summarizeメソッドの振る舞いを与えてくれました。

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

このコードは、1 new tweet: (Read more from @horse_ebooks...)と出力します。

同じメソッドのオーバーライドした実装からは、デフォルト実装を呼び出すことができないことに注意してください。

12.3.4　トレイト境界

これでトレイトの定義とトレイトを型に実装する方法を知ったので、ジェネリックな型引数でトレイトを使用する方法を探求できます。**トレイト境界**を使用してジェネリックな型を制限し、型が特定のトレイトや振る舞いを実装するものに制限されることを保証できます。

例として、リスト 10-13 で、Summaryトレイトを型 NewsArticleと Tweetに実装しました。引数 itemに対して summarizeメソッドを呼び出す関数 notifyを定義でき、この引数はジェネリックな型 Tです。ジェネリックな型 Tがメソッド summarizeを実装しないというエラーを出さずに itemに summarizeを呼び出せるために、Tに対してトレイト境界を使用して itemは、Summaryトレイトを実装する型でなければならないと指定できます:

```
pub fn notify<T: Summary>(item: T) {
    // 衝撃的なニュース！ {}
    println!("Breaking news! {}", item.summarize());
}
```

トレイト境界をジェネリックな型引数宣言とともにコロンの後、山カッコ内に配置しています。Tに対するトレイト境界のため、notifyを呼び出して NewsArticleか Tweetのどんなインスタンスも渡すことができます。あらゆる他の型、Stringや i32などでこの関数を呼び出すコードは、型が Summaryを実装しないので、コンパイルできません。

+記法でジェネリックな型に対して複数のトレイト境界を指定できます。例えば、関数で Tに対してフォーマット表示と、summarizeメソッドを使用するには、T: Summary + Displayを使用して、Tは Summaryと Displayを実装するどんな型にもなると宣言できます。

しかしながら、トレイト境界が多すぎると欠点もあります。各ジェネリックには、特有のトレイト境界があります; 故に、複数のジェネリックな型引数がある関数には、関数名と引数リストの間に多くのトレイト境界の情報が付くこともあり、関数シグニチャが読みづらくなる原因になります。このため、Rust には関数シグニチャの後、where節内にトレイト境界を指定する対立的な記法があります。従って、こう書く代わりに:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

こんな感じに where節を活用できます:

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

この関数シグニチャは、多くのトレイト境界のない関数のように、関数名、引数リスト、戻り値の型が一緒になって近いという点でごちゃごちゃしていません。

12.3.5　トレイト境界で largest 関数を修正する

ジェネリックな型引数の境界で使用したい振る舞いを指定する方法を知ったので、リスト 10-5 に戻って、ジェネリックな型引数を使用する largest関数の定義を修正しましょう！ 最後にそのコードを実行しようとしたら、こんなエラーが出ました:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
  = note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

largestの本体で、大なり演算子 (>) を使用して型 Tの 2 つの値を比較したかったのです。その演算子は、標準ライブラリトレイトの std::cmp::PartialOrdでデフォルトメソッドとして定義されているので、largest関数が、比較できるあらゆる型のスライスに対して動くように Tのトレイト境界に PartialOrdを指定する必要があります。初期化処理に含まれているので、PartialOrdをスコープに導入する必要はありません。largestのシグニチャを以下のような見た目に変えてください:

```
1 fn largest<T: PartialOrd>(list: &[T]) -> T {
```

今度コードをコンパイルすると、異なる一連のエラーが出ます:

```
error[E0508]: cannot move out of type `[T]`, a non-copy slice
(エラー: `[T]`、コピーでないスライスからムーブできません。)
 --> src/main.rs:2:23
  |
2 |     let mut largest = list[0];
  |                       ^^^^^^^
  |                       |
  |                       cannot move out of here
  |                       help: consider using a reference instead: `&list[0]`
                         (助言: 代わりに参照の使用を考慮してください: `&list[0]`)

error[E0507]: cannot move out of borrowed content
(エラー: 借用された内容からムーブできません)
 --> src/main.rs:4:9
  |
4 |     for &item in list.iter() {
  |         ^----
  |         ||
  |         |hint: to prevent move, use `ref item` or `ref mut item`
  |         cannot move out of borrowed content
            (ヒント: ムーブを避けるには、`ref item`か`ref mut item`を使用してください)
```

このエラーの鍵となる行は、cannot move out of type [T], a non-copy sliceです。ジェネリックでないバージョンの largest関数では、最大の i32か charを探そうとするだけでした。第 4 章の「ス

タックだけのデータ: コピー」節で議論したように、i32や charのような既知のサイズの型は、スタックに格納できるので、Copyトレイトを実装しています。しかし、largest関数をジェネリックにすると、list引数が Copyトレイトを実装しない型を含む可能性も出てきたのです。結果として、list[0]から値をムーブできず、largestにムーブできず、このエラーに落ち着いたのです。

このコードを Copyトレイトを実装する型とだけで呼び出すには、Tのトレイト境界に Copyを追加できます！ リスト 10-15 は、関数に渡したスライスの値の型が i32や charなどのように、PartialOrdと Copyを実装する限り、コンパイルできるジェネリックな largest関数の完全なコードを示しています:

ファイル名: src/main.rs

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

リスト 10-15: PartialOrdと Copyトレイトを実装するあらゆるジェネリックな型に対して動く、largest関数の動く定義

もし largest関数を Copyを実装する型だけに制限したくなかったら、Copyではなく、T が Cloneというトレイト境界を含むと指定することもできます。そうしたら、largest関数に所有権が欲しい時にスライスの各値をクローンできます。clone関数を使用するということは、Stringのようなヒープデータを所有する型の場合にもっとヒープ確保が発生する可能性があることを意味し、大きなデータを取り扱っていたら、ヒープ確保は遅いこともあります。

largestの別の実装方法は、関数がスライスの T値への参照を返すようにすることです。戻り値の型を Tではなく&Tに変え、それにより関数の本体を参照を返すように変更したら、Cloneか Copyトレイト境界は必要なくなり、ヒープ確保も避けられるでしょう。試しにこれらの対立的な解決策もご自身で実装してみてください！

12.3.6　トレイト境界を使用して、メソッド実装を条件分けする

implブロックでジェネリックな型引数を使用するトレイト境界を活用することで、特定のトレイトを実装する型に対するメソッド実装を条件分けできます。例えば、リスト 10-16 の型 Pair<T>は、常に new関数を実装します。しかし、Pair<T>は、内部の型 Tが比較を可能にする PartialOrdトレイトと出力を可能にする Displayトレイトを実装している時のみ、cmp_displayメソッドを実装します:

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

リスト 10-16: トレイト境界によってジェネリックな型に対するメソッド実装を条件分けする

また、別のトレイトを実装するあらゆる型に対するトレイト実装を条件分けすることもできます。トレイト境界を満たすあらゆる型にトレイトを実装することは、**ブランケット実装** (blanket implementation) と呼ばれ、Rust の標準ライブラリで広く使用されています。例を挙げれば、標準ライブラリは、Displayトレイトを実装するあらゆる型に ToStringトレイトを実装しています。標準ライブラリの implブロックは以下のような見た目です:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

標準ライブラリにはこのブランケット実装があるので、Displayトレイトを実装する任意の型に対して、ToStringトレイトで定義された to_stringメソッドを呼び出せるのです。例えば、整数は Displayを実装するので、このように整数値を対応する String値に変換できます:

```
let s = 3.to_string();
```

ブランケット実装は、「実装したもの」節のトレイトのドキュメンテーションに出現します。

トレイトとトレイト境界により、ジェネリックな型引数を使用して重複を減らしつつ、コンパイラに対して、そのジェネリックな型に特定の振る舞いが欲しいことを指定するコードを書くことができます。それからコンパイラは、トレイト境界の情報を活用してコードに使用された具体的な型が正しい振る舞いを提供しているか確認できます。動的型付け言語では、型が実装しない型のメソッドを呼び出せば、実行時にエラーが出るでしょう。しかし、Rust はこの種のエラーをコンパイル時に移したので、コードが動かせるようにさえなる以前に問題を修正することを強制されるのです。加えて、コンパイル時に既に確認したので、実行時に振る舞いがあるかどう確認するコードを書かなくても済みます。そうすることでジェネリクスの柔軟性を諦める必要なく、パフォーマンスを向上させます。

もう使用したことのある別の種のジェネリクスは、ライフタイムと呼ばれます。型が欲しい振る舞いを保持していることを保証するのではなく、必要な間だけ参照が有効であることをライフタイムは保証します。ライフタイムがどうやってそれを行うかを見ましょう。

12.4　ライフタイムで参照を有効化する

第 4 章の「参照と借用」節で議論しなかった詳細の 1 つは、Rust において参照は全てライフタイムを保持することであり、ライフタイムとは、その参照が有効になるスコープのことです。多くの場合、型が推論されるように、多くの場合、ライフタイムも暗黙的に推論されます。複数の型の可能性があるときには、型を注釈しなければなりません。同様に、参照のライフタイムがいくつか異なる方法で関係することがある場合には注釈しなければなりません。コンパイラは、ジェネリックライフタイム引数を使用して関係を注釈し、実行時に実際の参照が確かに有効であることを保証することを要求するのです。

ライフタイムの概念は、ほかのプログラミング言語の道具とはどこか異なり、議論はあるでしょうが、ライフタイムが Rust で一番際立った機能になっています。この章では、ライフタイムの全てを講義しないものの、ライフタイム記法と遭遇する可能性のある一般的な手段を議論するので、概念に馴染めます。もっと詳しく知るには、第 19 章の「高度なライフタイム」節を参照されたし。

12.4.1　ライフタイムはダングリング参照を回避する

ライフタイムの主な目的は、ダングリング参照を回避することであり、ダングリング参照によりプログラムは、参照することを意図したデータ以外のデータを参照してしまいます。リスト 10-17 のプログラムを考えてください。これには、外側のスコープと内側のスコープが含まれています:

```
{
    let r;

    {
```

```
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

リスト 10-17: 値がスコープを抜けてしまった参照を使用しようとする

注釈: リスト 10-17 や今後数例では、変数に初期値を与えずに宣言しているので、変数名は外側のスコープに存在します。初見では、これは Rust には null 値が存在しないということと衝突しているように見える可能性があります。しかしながら、値を与える前に変数を使用しようとすれば、コンパイルエラーになり、確かに Rust では null 値は許可されないことを示します。

外側のスコープで初期値なしの rという変数を宣言し、内側のスコープで初期値 5の xという変数を宣言しています。内側のスコープ内で、rの値を xへの参照にセットしようとしています。それから内側のスコープが終わり、rの値を出力しようとしています。rが参照している値が使おうとする前にスコープを抜けるので、このコードはコンパイルできません。こちらがエラーメッセージです:

```
error[E0597]: `x` does not live long enough
(エラー: `x`の生存期間が短すぎます)
  --> src/main.rs:7:5
   |
6  |         r = &x;
   |             - borrow occurs here
   |             (借用はここで起きています)
7  |     }
   |     ^ `x` dropped here while still borrowed
   |     (`x`は借用されている間にここでドロップされました)
...
10 | }
   | - borrowed value needs to live until here
   | (借用された値はここまで生きる必要があります)
```

変数 xの「生存期間が短すぎます。」原因は内側のスコープが 7 行目で終わった時点で xがスコープを抜けるからです。ですが、rはまだ、外側のスコープに対して有効です; スコープが大きいので、「長生きする」と言います。Rust で、このコードが動くことを許可していたら、rは xがスコープを抜けた時に解放されるメモリを参照していることになり、rで行おうとするいかなることもちゃんと動かないでしょう。では、どうやってコンパイラはこのコードが無効であると決定しているのでしょうか？借用精査機 (borrow checker) を使用しています。

12.4.2　借用精査機

Rust コンパイラには、スコープを比較して全ての借用が有効であることを決定する**借用精査機**があります。リスト 10-18 は、リスト 10-17 と同じコードを示していますが、変数のライフタイムを表示する注釈が付いています:

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

リスト 10-18: それぞれ'aと'bと名付けられた rと xのライフタイムの注釈

ここで、rのライフタイムは'a、xのライフタイムは'bで注釈しました。ご覧の通り、内側の'bブロックの方が、外側の'aライフタイムブロックよりはるかに小さいです。コンパイル時に、コンパイラは 2 つのライフタイムのサイズを比較し、rは'aのライフタイムだけれども、'bのライフタイムのメモリを参照していると確認します。'bは'aよりも短いので、プログラムは拒否されます: 参照の被写体が参照ほど長生きしないのです。

リスト 10-19 でコードを修正したので、ダングリング参照はなくなり、エラーなくコンパイルできます:

```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

リスト 10-19: データのライフタイムが参照より長いので、有効な参照

ここで xのライフタイムは'bになり、今回の場合'aよりも大きいです。つまり、コンパイラは xが有効な間、rの参照も常に有効になることを把握しているので、rは xを参照できます。

今や、参照のライフタイムがどこにあり、コンパイラがライフタイムを解析して参照が常に有効であることを保証する仕組みがわかったので、関数の文脈でジェネリックな引数と戻り値のライフタイムを探求しましょう。

12.4.3　関数のジェネリックなライフタイム

2 つの文字列スライスのうち、長い方を返す関数を書きましょう。この関数は、2 つの文字列スライスを取り、1 つの文字列スライスを返します。longest関数の実装完了後、リスト 10-20 のコードは、The logest string is abcdと出力するはずです:

ファイル名: src/main.rs

```
fn main() {
```

```
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    // 最長の文字列は、{}です
    println!("The longest string is {}", result);
}
```

リスト 10-20: longest関数を呼び出して 2 つの文字列スライスのうち長い方を探す main関数

関数に取ってほしい引数が文字列スライス、つまり参照であることに注意してください。何故なら、longest関数に引数の所有権を奪ってほしくないからです。この関数に Stringのスライス (変数 string1に格納されている型) と文字列リテラル (変数 string2が含むもの) を受け取らせたいのです。

リスト 10-20 で使用している引数が求めているものである理由についてもっと詳しい議論は、第 4 章の「引数としての文字列スライス」節をご参照ください。

リスト 10-21 に示したように longetst関数を実装しようとしたら、コンパイルできないでしょう:

ファイル名: src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

リスト 10-21: 2 つの文字列スライスのうち長い方を返すけれども、コンパイルできない longest関数の実装

代わりに、以下のようなライフタイムに言及するエラーが出ます:

```
error[E0106]: missing lifetime specifier
(エラー: ライフタイム指定子が不足しています)
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |                                 ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `x` or `y`
  (助言: この関数の戻り値型は借用された値を含んでいますが、
シグニチャは、それが`x`か`y`由来のものなのか宣言していません)
```

助言テキストが戻り値の型はジェネリックなライフタイム引数である必要があると明かしています。というのも、返している参照が xか yを参照しているかコンパイラにはわからないからです。この関数の本体の ifブロックは xへの参照を返し、elseブロックは yへの参照を返すので、実際、どちらか私たちにもわかりません！

この関数を定義する際、この関数に渡される具体的な値がわからないので、ifケースか、elseケースが実行されるか、わからないのです。また、渡される参照の具体的なライフタイムもわからないの

で、リスト 10-18 と 10-19 で、返す参照が常に有効であることを決定したようにスコープを見ることもできないのです。借用精査機もこれを決定することはできません。xと yのライフタイムがどう戻り値のライフタイムと関係するかわからないからです。このエラーを修正するには、借用精査機が解析を実行できるように、参照間の関係を定義するジェネリックなライフタイム引数を追加します。

12.4.4　ライフタイム注釈記法

ライフタイム注釈は、いかなる参照の生存期間も変えることはありません。シグニチャがジェネリックな型引数を指定している時に、関数があらゆる型を受け入れるのと全く同様に、ジェネリックなライフタイム引数を指定することで関数は、あらゆるライフタイムの参照を受け入れるのです。ライフタイム注釈は、ライフタイムに影響することなく、複数の参照のライフタイムのお互いの関係を記述します。

ライフタイム注釈は、少しだけ不自然な記法です: ライフタイム引数の名前はアポストロフィー'で始まらなければならず、通常全部小文字で、ジェネリック型のようにとても短いです。多くの人は、'aという名前を使います。ライフタイム引数注釈は、参照の&の後に配置し、注釈と参照の型を区別するために空白を 1 つ使用します。

例を挙げましょう: ライフタイム引数なしの i32への参照、'aというライフタイム引数付きの i32への参照、これもライフタイム'a付き i32への可変参照です。

```
            // (ただの)参照
&i32        // a reference
            // 明示的なライフタイム付きの参照
&'a i32     // a reference with an explicit lifetime
            // 明示的なライフタイム付きの可変参照
&'a mut i32 // a mutable reference with an explicit lifetime
```

1 つのライフタイム注釈それだけでは、大して意味はありません。注釈は、複数の参照のジェネリックなライフタイム引数が、お互いにどう関係するかをコンパイラに指示することを意図しているからです。例えば、ライフタイム'a付きの i32への参照となる引数 firstのある関数があるとしましょう。この関数にはさらに、'aのライフタイム付きの i32への別の参照となる secondという別の引数もあります。ライフタイム注釈は、firstと secondの参照がどちらもジェネリックなライフタイムと同じだけ生きることを示唆します。

12.4.5　関数シグニチャにおけるライフタイム注釈

さて、longest関数の文脈でライフタイム注釈を調査しましょう。ジェネリックな型引数同様、関数名と引数リストの間、山カッコの中にジェネリックなライフタイム引数を宣言する必要があります。このシグニチャで表現したい制約は、引数の全参照と戻り値が同じライフタイムになることです。ライフタイムを'aと名付け、それから各参照に追記します。リスト 10-22 に示したように:

ファイル名: src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

リスト 10-22: シグニチャの全参照が同じライフタイム'aになると指定した longest関数の定義

このコードはコンパイルでき、リスト 10-20 の main関数とともに使用したら、欲しい結果になるはずです。

これで関数シグニチャは、なんらかのライフタイム'aに対して、関数は 2 つの引数を取り、どちらも少なくともライフタイム'aと同じだけ生きる文字列スライスであるとコンパイラに教えています。また、この関数シグニチャは、関数から返る文字列スライスも少なくともライフタイム'aと同じだけ生きると、コンパイラに教えています。これらの制約は、コンパイラに強制してほしいものです。この関数シグニチャでライフタイム引数を指定する時、渡されたり、返したりしたいかなる値のライフタイムも変更していないことを思い出してください。むしろ、借用精査機は、これらの制約を支持しない値全てを拒否するべきと指定しています。longest関数は、正確に xと yの生存期間を知る必要はなく、何かのスコープが'aに代替され、このシグニチャを満足することだけ知っている必要があることに注意してください。

関数でライフタイムを注釈する際、注釈は関数シグニチャに嵌 (はま) り、関数本体には嵌りません。コンパイラは、なんの助けもなく、関数内のコードを解析できます。しかしながら、関数に関数外やからの参照がある場合、コンパイラは引数や戻り値のライフタイムをそれだけではじき出すことはほとんど不可能になります。ライフタイムは、関数が呼び出される度に異なる可能性があります。このために、手動でライフタイムを注釈する必要があるのです。

具体的な参照を longestに渡すと、'aを代替する具体的なライフタイムは、yのスコープと被さる xのスコープの一部になります。言い換えると、ジェネリックなライフタイム'aは、xと yのライフタイムのうち、小さい方に等しい具体的なライフタイムになるのです。返却される参照を同じライフタイム引数'aで注釈したので、返却される参照も xか yのライフタイムの小さい方と同じだけ有効になるでしょう。

ライフタイム注釈が異なる具体的なライフタイムになる参照を渡すことで longest関数を制限する方法を見ましょう。リスト 10-23 は、率直な例です:

ファイル名: src/main.rs

```
# fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
#     if x.len() > y.len() {
#         x
#     } else {
#         y
#     }
# }
#
```

```
fn main() {
    // 長い文字列は長い
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

リスト 10-23: 異なる具体的なライフタイムの String値への参照で longest関数を使用する

この例において、string1は外側のスコープの終わりまで有効で、string2は内側のスコープの終わりまで有効、そして resultは内側のスコープの終わりまで有効な何かを参照しています。このコードを実行すると、借用精査機がこのコードに賛成するのがわかるでしょう。要するに、コンパイルでき、The longest string is long string is longと出力するのです。

次に、resultの参照のライフタイムが 2 つの引数の小さい方のライフタイムになることを示す例を試しましょう。result変数の宣言を内側のスコープの外に移すものの、result変数への代入はstring2のスコープ内に残したままにします。それから resultを使用する println!を内側のスコープの外、内側のスコープが終わった後に移動します。リスト 10-24 のコードはコンパイルできません:

ファイル名: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

リスト 10-24: string2がスコープを抜けてから resultを使用しようとする; このコードはコンパイルできない

このコードのコンパイルを試みると、こんなエラーになります:

```
error[E0597]: `string2` does not live long enough
  --> src/main.rs:15:5
   |
14 |         result = longest(string1.as_str(), string2.as_str());
   |                                            ------- borrow occurs here
15 |     }
   |     ^ `string2` dropped here while still borrowed
16 |     println!("The longest string is {}", result);
17 | }
   | - borrowed value needs to live until here
```

このエラーは、resultがprintln!文に対して有効になるために、string2が外側のスコープの終わりまで有効である必要があることを示しています。関数引数と戻り値のライフタイムを同じライフタイム引数'aで注釈したので、コンパイラはこのことを知っています。

人間からしたら、このコードを見てstring1はstring2よりも長いことが確認でき、故にresultはstring1への参照を含んでいます。まだstring1はスコープを抜けていないので、それでもstring1への参照はprintln!にとって有効でしょう。ですが、コンパイラはこの場合、参照が有効であると見なせません。longest関数から返ってくる参照のライフタイムは、渡した参照のうちの小さい方と同じだとコンパイラに指示しました。それ故に、借用精査機は、リスト 10-24 のコードを無効な参照がある可能性があるとして許可しないのです。

試しに値やlongest関数に渡される参照のライフタイムや返される参照の使用法が異なる実験をもっと企ててみてください。自分の実験がコンパイル前に借用精査機を通るかどうか仮説を立ててください; そして、正しいか確かめてください！

12.4.6　ライフタイムの観点で思考する

ライフタイム引数を指定する必要のある手段は、関数が行っていることによります。例えば、longest関数の実装を最長の文字列スライスではなく、常に最初の引数を返すように変更したら、y引数に対してライフタイムを指定する必要はなくなるでしょう。以下のコードはコンパイルできます:

ファイル名: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

この例では、引数 xと戻り値に対してライフタイム引数'aを指定しましたが、引数 yには指定していません。yのライフタイムは xや戻り値のライフタイムとは何の関係もないからです。

関数から参照を返す際、戻り値型のライフタイム引数は、引数のうちどれかのライフタイム引数と一致する必要があります。返される参照が引数のどれかを参照していなければ、この関数内で生成された値を参照しているに違いなく、これは、その値が関数の末端でスコープを抜けるので、ダングリング参照になるでしょう。コンパイルできないこの longest関数の未遂の実装を考えてください:

ファイル名: src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    // 本当に長い文字列
    let result = String::from("really long string");
    result.as_str()
}
```

ここでは、たとえ、戻り値型にライフタイム引数'aを指定していても、戻り値のライフタイムは、引数のライフタイムと全く関係がないので、この実装はコンパイルできないでしょう。こちらが、得られるエラーメッセージです:

```
error[E0597]: `result` does not live long enough
 --> src/main.rs:3:5
  |
3 |     result.as_str()
  |     ^^^^^^ does not live long enough
4 | }
  | - borrowed value only lives until here
  |
note: borrowed value must be valid for the lifetime 'a as defined on the
function body at 1:1...
(注釈: 借用された値は、関数本体1行目1文字目で定義されているようにライフタイム'aに対して
    有効でなければなりません)
 --> src/main.rs:1:1
  |
1 | / fn longest<'a>(x: &str, y: &str) -> &'a str {
2 | |     let result = String::from("really long string");
3 | |     result.as_str()
4 | | }
  | |_^
```

問題は、resultが longest関数の末端でスコープを抜け、片付けられてしまうことです。また、関数から resultを返そうともしています。ダングリング参照を変えるであろうライフタイム引数を指定する手段はなく、コンパイラは、ダングリング参照を生成させてくれません。今回の場合、最善の修正案は、呼び出し元の関数が値の片付けに責任を持てるよう、参照ではなく所有されたデータ型を返すことでしょう。

究極的にライフタイム記法は、関数のいろんな引数と戻り値のライフタイムを接続することに関するのです。一旦、繋がりができたら、メモリ安全な処理を許可するのに十分な情報がコンパイラにはあり、ダングリングポインタを生成するであろう処理を不認可し、さもなくばメモリ安全性を侵害するのです。

12.4.7　構造体定義のライフタイム注釈

ここまで、所有された型を保持する構造体だけを定義してきました。構造体に参照を保持させることもできますが、その場合、構造体定義の全参照にライフタイム注釈を付け加える必要があるでしょう。リスト 10-25 には、文字列スライスを保持する ImportantExcerptという構造体があります:

ファイル名: src/main.rs

```
1  struct ImportantExcerpt<'a> {
2      part: &'a str,
3  }
4
5  fn main() {
6      // 僕をイシュマエルとお呼び。何年か前・・・
7      let novel = String::from("Call me Ishmael. Some years ago...");
8      let first_sentence = novel.split('.')
9          .next()
10         .expect("Could not find a '.'");  // '.'が見つかりませんでした
```

```
    let i = ImportantExcerpt { part: first_sentence };
}
```

リスト 10-25: 参照を含む構造体なので、定義にライフタイム注釈が必要

この構造体には文字列スライスを保持する 1 つのフィールド、partがあり、これは参照です。ジェネリックなデータ型同様、構造体名の後、山カッコの中にジェネリックなライフタイム引数の名前を宣言するので、構造体定義の本体でライフタイム引数を使用できます。この注釈は、ImportantExcerptのインスタンスが、partフィールドに保持している参照よりも長生きしないことを意味します。

ここの main関数は、変数 novelに所有される Stringの最初の文への参照を保持するImportantExcerptインスタンスを生成しています。novelのデータは、ImportantExcerptインスタンスが作られる前に存在しています。加えて、ImportantExcerptがスコープを抜けるまで novelはスコープを抜けないので、ImportantExcerptインスタンスの参照は有効なのです。

12.4.8　ライフタイム省略

全参照にはライフタイムがあり、参照を使用する関数や構造体にはライフタイム引数を指定する必要があることを学びました。ですが、第 4 章の「文字列スライス」節にある関数があり、リスト 10-26 に再度示しましたが、これは、ライフタイム注釈なしでコンパイルできました:

ファイル名: src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

リスト 10-26: 引数と戻り値型が参照であるにも関わらず、ライフタイム注釈なしでコンパイルできた第 4 章で定義した関数

この関数がライフタイム注釈なしでコンパイルできた理由は、歴史的なものです: 昔のバージョンの Rust(1.0 以前) では、全参照に明示的なライフタイムが必要だったので、このコードはコンパイルできませんでした。その頃、関数シグニチャはこのように記述されていたのです:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

多くの Rust コードを書いた後、Rust チームは、Rust プログラマが特定の場面では、何度も何度も同じライフタイム注釈に入ることを発見しました。これらの場面は予測可能で、いくつかの決定的なパターンに従っていました。開発者はこのパターンをコンパイラのコードに落とし込んだので、こ

のような場面には借用精査機がライフタイムを推論できるようになり、明示的な注釈を必要としなくなったのです。

他の決定的なパターンが出現し、コンパイラに追加されることもあり得るので、この Rust の歴史は関係があります。将来的に、さらに少数のライフタイム注釈しか必要にならない可能性もあります。

コンパイラの参照解析に落とし込まれたパターンは、**ライフタイム省略規則**と呼ばれます。これらはプログラマが従う規則ではありません; コンパイラが考慮する一連の特定のケースであり、自分のコードがこのケースに当てはまれば、ライフタイムを明示的に書く必要はなくなります。

省略規則は、完全な推論を提供しません。コンパイラが決定的に規則を適用できるけれども、参照がなるライフタイムに関してそれでも曖昧性があるなら、コンパイラは、残りの参照がなるべきライフタイムを推論しません。この場合、推論ではなく、コンパイラは、参照がお互いにどう関係するかを指定するライフタイム注釈を追記することで、解決できるエラーを与えます。

関数やメソッドの引数のライフタイムは、**入力ライフタイム**と呼ばれ、戻り値のライフタイムは**出力ライフタイム**と称されます。

コンパイラは 3 つの規則を活用し、明示的な注釈がない時に、参照がどんなライフタイムになるかを計算します。最初の規則は入力ライフタイムに適用され、2 番目と 3 番目の規則は出力ライフタイムに適用されます。コンパイラが 3 つの規則の最後まで到達し、それでもライフタイムを割り出せない参照があったら、コンパイラはエラーで停止します。

最初の規則は、参照である各引数は、独自のライフタイム引数を得るというものです。換言すれば、1 引数の関数は、1 つのライフタイム引数を得るということです: `fn foo<'a>(x: &'a i32)`; 2 つ引数のある関数は、2 つの個別のライフタイム引数を得ます: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`; 以下同様。

2 番目の規則は、1 つだけ入力ライフタイム引数があるなら、そのライフタイムが全ての出力ライフタイム引数に代入されるというものです: `fn foo<'a>(x: &'a i32) -> &'a i32`。

3 番目の規則は、複数の入力ライフタイム引数があるけれども、メソッドなのでそのうちの一つが`&self`や`&mut self`だったら、`self`のライフタイムが全出力ライフタイム引数に代入されるというものです。この 3 番目の規則により、必要なシンボルの数が減るので、メソッドが遥かに読み書きしやすくなります。

コンパイラになってみましょう。これらの規則を適用して、リスト 10-26 の `first_word`関数のシグニチャの参照のライフタイムが何か計算します。シグニチャは、参照に紐付けられるライフタイムがない状態から始まります:

```
1 fn first_word(s: &str) -> &str {
```

そうして、コンパイラは最初の規則を適用し、各引数が独自のライフタイムを得ると指定します。それを通常通り`'a`と呼ぶので、シグニチャはこうなります:

```
1 fn first_word<'a>(s: &'a str) -> &str {
```

1 つだけ入力ライフタイムがあるので、2 番目の規則を適用します。2 番目の規則は、1 つの入力引数のライフタイムが、出力引数に代入されると指定するので、シグニチャはこうなります:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

もうこの関数シグニチャの全ての参照にライフタイムが付いたので、コンパイラは、プログラマにこの関数シグニチャのライフタムを注釈してもらう必要なく、解析を続行できます。

別の例に目を向けましょう。今回は、リスト 10-21 で取り掛かったときにはライフタイム引数はなかった `longest`関数です:

```
fn longest(x: &str, y: &str) -> &str {
```

最初の規則を適用しましょう: 各引数が独自のライフタイムを得るのです。今回は、1 つではなく 2 つ引数があるので、ライフタイムも 2 つです:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

2 つ以上入力ライフタイムがあるので、2 番目の規則は適用されないとわかります。また 3 番目の規則も適用されません。`longest`はメソッドではなく関数なので、どの引数も `self`ではないのです。3 つの規則全部を適用した後、まだ戻り値型のライフタイムが判明していません。このために、リスト 10-21 でこのコードをコンパイルしようとしてエラーになったのです: コンパイラは、ライフタイム省略規則全てを適用したけれども、シグニチャの参照全部のライフタイムを計算できなかったのです。

3 番目の規則は本当にメソッドシグニチャでしか適用されないので、次にその文脈でライフタイムを観察し、3 番目の規則が、メソッドシグニチャであまり頻繁にライフタイムを注釈しなくても済むことを意味する理由を確認します。

12.4.9　メソッド定義におけるライフタイム注釈

構造体にライフタイムのあるメソッドを実装する際、リスト 10-11 で示したジェネリックな型引数と同じ記法を使用します。ライフタイム引数を宣言し使用する場所は、構造体フィールドかメソッド引数と戻り値に関係するかによります。

構造体のフィールド用のライフタイム名は、`impl`キーワードの後に宣言する必要があり、それから構造体名の後に使用されます。そのようなライフタイムは構造体の型の一部になるからです。

`impl`ブロック内のメソッドシグニチャでは、参照は構造体のフィールドの参照のライフタイムに紐づくか、独立している可能性があります。加えて、ライフタイム省略規則により、メソッドシグニチャでライフタイム注釈が必要なくなることがよくあります。リスト 10-25 で定義した `ImportantExcerpt`という構造体を使用して、何か例を眺めましょう。

まず、唯一の引数が `self`への参照で戻り値が `i32`という何かへの参照ではない `level`というメソッドを使用します:

```
# struct ImportantExcerpt<'a> {
```

```
#     part: &'a str,
# }
#
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

impl後のライフタイム引数宣言と型名の後に使用するのは必須ですが、最初の省略規則のため、selfへの参照のライフタイムを注釈する必要はありません。

3 番目のライフタイム省略規則が適用される例はこちらです:

```
# struct ImportantExcerpt<'a> {
#     part: &'a str,
# }
#
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

2 つ入力ライフタイムがあるので、コンパイラは最初のライフタイム省略規則を適用し、&selfとannouncementに独自のライフタイムを与えます。それから、引数の 1 つが&selfなので、戻り値型は&selfのライフタイムを得て、全てのライフタイムが説明されました。

12.4.10　静的ライフタイム

議論する必要のある 1 種の特殊なライフタイムが、'staticであり、これはプログラム全体の期間を示します。文字列リテラルは全て'staticライフタイムになり、次のように注釈できます:

```
let s: &'static str = "I have a static lifetime."; //静的なライフタイムだよ。
```

この文字列のテキストは、プログラムのバイナリに直接格納され、常に利用可能です。故に、全文字列リテラルのライフタイムは、'staticなのです。

エラーメッセージで'staticライフタイムを使用することを勧められることがある可能性があります。ですが、参照に対してライフタイムとして'staticを指定する前に、今ある参照が本当にプログラムの全期間生きるかどうか考えてください。可能であっても、参照がそれだけの期間生きるかどうか考慮する可能性があります。ほとんどの場合、問題は、ダングリング参照を生成しようとしているか、利用可能なライフタイムの不一致が原因です。そのような場合、解決策はその問題を修正することであり、'staticライフタイムを指定することではありません。

12.5　ジェネリックな型引数、トレイト境界、ライフタイムを一度に

ジェネリックな型引数、トレイト境界、ライフタイムを指定する記法を全て 1 関数でちょっと眺めましょう！

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
    where T: Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

これがリスト 10-22 からの 2 つの文字列のうち長い方を返す `longest`関数ですが、ジェネリックな型 `T`の `ann`という追加の引数があり、これは `where`節で指定されているように、`Display`トレイトを実装するあらゆる型で埋めることができます。この追加の引数は、関数が文字列スライスの長さを比較する前に出力されるので、`Display`トレイト境界が必要なのです。ライフタイムは 1 種のジェネリックなので、ライフタイム引数`'a`とジェネリックな型引数 `T`が関数名の後、山カッコ内の同じリストに収まっています。

12.6　総括

いろんなことをこの章では講義しましたね！ 今やジェネリックな型引数、トレイトとトレイト境界、そしてジェネリックなライフタイム引数を知ったので、多くの異なる場面で動くコードを繰り返しなく書く準備ができました。ジェネリックな型引数により、コードを異なる型に適用させてくれます。トレイトとトレイト境界は、型がジェネリックであっても、コードが必要とする振る舞いを持つことを保証します。ライフタイム注釈を活用して、この柔軟なコードにダングリング参照が存在しないことを保証する方法を学びました。さらにこの解析は全てコンパイル時に起こり、実行時のパフォーマンスには影響しません！

信じるかどうかは自由ですが、この章で議論した話題にはもっともっと学ぶべきことがあります: 第 17 章ではトレイトオブジェクトを議論し、これはトレイトを使用する別の手段です。第 19 章では、ライフタイム注釈が関わるもっと複雑な筋書きと何か高度な型システムの機能を講義します。ですが次の章では、Rust でテストを書く方法を学ぶので、コードがあるべき通りに動いていることを確かめられます。

第13章

自動テストを書く

1972 年のエッセイ「謙虚なプログラマ」でエドガー・W・ダイクストラは以下のように述べている。「プログラムのテストは、バグの存在を示すには非常に効率的な手法であるが、バグの不在を示すには望み薄く不適切である」と。これは、できるだけテストを試みるべきではないということではありません。

プログラムの正当性は、どこまで自分のコードが意図していることをしているかなのです。Rust は、プログラムの正当性に重きを置いて設計されていますが、正当性は複雑で、単純に証明することはありません。Rust の型システムは、この重荷の多くの部分を肩代わりしてくれますが、型システムはあらゆる種類の不当性を捕捉してはくれません。ゆえに、Rust では、言語内で自動化されたソフトウェアテストを書くことをサポートしているのです。

例として、渡された何かの数値に 2 を足す `add_two`という関数を書くとしましょう。この関数のシグニチャは、引数に整数を取り、結果として整数を返します。この関数を実装してコンパイルすると、コンパイラはこれまでに学んできた型チェックと借用チェックを全て行い、例えば、`String`の値や無効な参照をこの関数に渡していないかなどを確かめるのです。ところが、コンパイラはプログラマがまさしく意図したことを関数が実行しているかどうかは確かめ**られません**。つまり、そうですね、引数に 10 を足したり、50 を引いたりするのではなく、引数に 2 を足していることです。そんな時にテストは、必要になるのです。

例えば、`add_two`関数に `3`を渡した時に、戻り値は 5 であることをアサーションするようなテストを書くことができます。コードに変更を加えた際にこれらのテストを走らせ、既存の正当な振る舞いが変わっていないことを確認できます。

テストは、煩雑なスキルです: いいテストの書き方をあらゆる方面から解説することは 1 章だけではできないのですが、Rust のテスト機構のメカニズムについて議論します。テストを書く際に利用可能になるアノテーションとマクロについて、テストを実行するのに提供されているオプションと標準の動作、さらにテストをユニットテストや統合テストに体系化する方法について語ります。

13.1　テストの記述法

テストは、非テストコードが想定された手段で機能していることを実証する Rust の関数です。テスト関数の本体は、典型的には以下の 3 つの動作を行います:

1. 必要なデータや状態をセットアップする。
2. テスト対象のコードを走らせる。
3. 結果が想定通りかアサーションする。

Rust が、特にこれらの動作を行うテストを書くために用意している機能を見ていきましょう。これには、`test`アトリビュート、いくつかのマクロ、`should_panic`アトリビュートが含まれます。

13.2　テスト関数の解剖

最も単純には、Rust におけるテストは `test`アトリビュートで注釈された関数のことです。アトリビュートとは、Rust コードの欠片に関するメタデータです; 一例を挙げれば、構造体とともに第 5 章で使用した `derive`アトリビュートです。関数をテスト関数に変えるには、`fn`の前に`#[test]`を付け加えるのです。`cargo test`コマンドでテストを実行したら、コンパイラは `test`アトリビュートで注釈された関数を走らせるテスト用バイナリをビルドし、各テスト関数が通過したか失敗したかを報告します。

第 7 章で、Cargo で新規ライブラリプロジェクトを作成した時に、テスト関数が含まれるテストモジュールが自動で生成されたことを見かけました。このモジュールのおかげでテストを書き始めることができるので、新しいプロジェクトを立ち上げる度に、テスト関数の正確な構造と記法を調べる必要がなくなるわけです。必要なだけテスト関数とテストモジュールは追加することができます。

実際にテストすることなしにテンプレートのテストが生成されるのを実験することでテストの動作法の一部を探求しましょう。それから、自分で書いた何らかのコードを呼び出し、振る舞いが正しいかアサーションする現実世界のテストを書きましょう。

`adder`という新しいライブラリプロジェクトを生成しましょう:

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

`adder`ライブラリの *src/lib.rs* ファイルの中身はリスト 11-1 のような見た目のはずです:

ファイル名: src/lib.rs

```
# fn main() {}
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
```

```
    }
}
```

リスト 11-1: cargo newで自動生成されたテストモジュールと関数

とりあえず、最初の 2 行は無視し、関数に集中してその動作法を見ましょう。fn行の#[test]アノテーションに注目してください: このアトリビュートは、これがテスト関数であることを示唆しますので、テスト実行機はこの関数をテストとして扱うとわかるのです。さらに、testsモジュール内には非テスト関数を入れ込み、一般的なシナリオをセットアップしたり、共通の処理を行う手助けをしたりもできるので、#[test]アトリビュートでどの関数がテストかを示唆する必要があるのです。

関数本体は、assert_eq!マクロを使用して、2 + 2 が 4 に等しいことをアサーションしています。このアサーションは、典型的なテストのフォーマット例をなしているわけです。走らせてこのテストが通ることを確かめましょう。

cargo testコマンドでプロジェクトにあるテストが全て実行されます。リスト 11-2 に示したようにね:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22 secs
     Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

リスト 11-2: 自動生成されたテストを走らせた出力

Cargo がテストをコンパイルし、走らせました。Compiling, Finished, Runningの行の後に running 1 testの行があります。次行が、生成されたテスト関数の it_worksという名前とこのテストの実行結果、okを示しています。テスト実行の総合的なまとめが次に出現します。test result:ok.というテキストは、全テストが通ったことを意味し、1 passed; 0 failedと読める部分は、通過または失敗したテストの数を合計しているのです。

無視すると指定したテストは何もなかったため、まとめは 0 ignoredと示しています。また、実行するテストにフィルタをかけもしなかったので、まとめの最後に 0 filtered outと表示されています。テストを無視することとフィルタすることに関しては次の節、「テストの実行され方を制御する」で語ります。

0 measuredという統計は、パフォーマンスを測定するベンチマークテスト用です。ベンチマークテストは、本書記述の時点では、ナイトリー版の Rust でのみ利用可能です。詳しくは、ベンチマーク

テストのドキュメンテーションを参照されたし。

テスト出力の次の部分、つまり Doc-tests adderで始まる部分は、ドキュメンテーションテストの結果用のものです。まだドキュメンテーションテストは何もないものの、コンパイラは、API ドキュメントに現れたどんなコード例もコンパイルできます。この機能により、ドキュメントとコードを同期することができるわけです。ドキュメンテーションテストの書き方については、第 14 章の「ドキュメンテーションコメント」節で議論しましょう。今は、Doc-tests出力は無視します。

テストの名前を変更してどうテスト出力が変わるか確かめましょう。it_works関数を違う名前、explorationなどに変えてください。そう、以下のように:

ファイル名: src/lib.rs

```
# fn main() {}
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

そして、cargo testを再度走らせます。これで出力が it_worksの代わりに explorationと表示しています:

```
running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

別のテストを追加しますが、今回は失敗するテストにしましょう！ テスト関数内の何かがパニックすると、テストは失敗します。各テストは、新規スレッドで実行され、メインスレッドが、テストスレッドが死んだと確認した時、テストは失敗と印づけられます。第 9 章でパニックを引き起こす最も単純な方法について語りました。要するに、panic!マクロを呼び出すことです。*src/lib.rs* ファイルがリスト 11-3 のような見た目になるよう、新しいテスト anotherを入力してください:

ファイル名: src/lib.rs

```
# fn main() {}
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        //このテストを失敗させる
        panic!("Make this test fail");
    }
```

```
14 }
```

リスト 11-3: panic!マクロを呼び出したために失敗する 2 番目のテストを追加する

cargo testで再度テストを走らせてください。出力はリスト 11-4 のようになるはずであり、explorationテストは通り、anotherは失敗したと表示されます:

```
running 2 tests
test tests::exploration ... ok
test tests::another ... FAILED

failures:

---- tests::another stdout ----
    thread 'tests::another' panicked at 'Make this test fail', src/lib.rs:10:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed
```

リスト 11-4: 1 つのテストが通り、失敗するときのテスト結果

okの代わりに test test::anotherの行は、FAILEDを表示しています。個々の結果とまとめの間に、2 つ新たな区域ができました: 最初の区域は、失敗したテスト各々の具体的な理由を表示しています。今回の場合、anotherは'Make this test fail'でパニックしたために失敗し、これは、*src/lib.rs* ファイルの 10 行で起きました。次の区域は失敗したテストの名前だけを列挙し、テストがたくさんあり、失敗したテストの詳細がたくさん表示されるときに有用になります。失敗したテストの名前を使用してそのテストだけを実行し、より簡単にデバッグすることができます。ただし、テストの実行方法については、「テストの実行され方を制御する」節でもっと語りましょう。

総括の行が最後に出力されています: 総合的に言うと、テスト結果は失敗でした。一つのテストが通り、一つが失敗したわけです。

異なるシナリオでのテスト結果がどんな風になるか見てきたので、テストを行う際に有用になるpanic!以外のマクロに目を向けましょう。

13.2.1　assert!マクロで結果を確認する

assert!マクロは、標準ライブラリで提供されていますが、テスト内の何らかの条件が trueと評価されることを確かめたいときに有効です。assert!マクロには、論理値に評価される引数を与えます。その値が trueなら、assert!は何もせず、テストは通ります。その値が falseなら、assert!マクロはpanic!マクロを呼び出し、テストは失敗します。assert!マクロを使用することで、コードが意図した通りに機能していることを確認する助けになるわけです。

第 5 章のリスト 5-15 で、Rectangle構造体と can_holdメソッドを使用しました。リスト 11-5 でもそれを繰り返しています。このコードを *src/lib.rs* ファイルに放り込み、assert!マクロでそれ用のテストを何か書いてみましょう。

ファイル名: src/lib.rs

```
# fn main() {}
#[derive(Debug)]
pub struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

リスト 11-5: 第 5 章から Rectangle構造体とその can_holdメソッドを使用する

can_holdメソッドは論理値を返すので、assert!マクロの完璧なユースケースになるわけです。リスト 11-6 で、長さが 8、幅が 7 の Rectangleインスタンスを生成し、これが長さ 5、幅 1 の別の Rectangleインスタンスを保持できるとアサーションすることで can_holdを用いるテストを書きます:

ファイル名: src/lib.rs

```
# fn main() {}
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        assert!(larger.can_hold(&smaller));
    }
}
```

リスト 11-6: より大きな四角形がより小さな四角形を確かに保持できるかを確認する can_hold用のテスト

testsモジュール内に新しい行を加えたことに注目してください: use super::*です。testsモジュールは、第 7 章の「プライバシー規則」節で解説した通常の公開ルールに従う普通のモジュールです。testsモジュールは、内部モジュールなので、外部モジュール内のテスト配下にあるコードを内部モジュールのスコープに持っていく必要があります。ここでは glob を使用して、外部モジュールで定義したもの全てがこの testsモジュールでも使用可能になるようにしています。

テストは larger_can_hold_smallerと名付け、必要な Rectangleインスタンスを 2 つ生成していま

す。そして、assert!マクロを呼び出し、larger.can_hold(&smaller)の呼び出し結果を渡しました。この式は、trueを返すと考えられるので、テストは通るはずです。確かめましょう！

```
running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

確かに！ 別のテストを追加しましょう。今回は、小さい四角形は、より大きな四角形を保持できないことをアサーションします。

ファイル名: src/lib.rs

```
# fn main() {}
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        assert!(!smaller.can_hold(&larger));
    }
}
```

今回の場合、can_hold関数の正しい結果は falseなので、その結果を assert!マクロに渡す前に反転させる必要があります。結果として、can_holdが falseを返せば、テストは通ります。

```
running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

通るテストが 2 つ！ さて、コードにバグを導入したらテスト結果がどうなるか確認してみましょう。長さを比較する大なり記号を小なり記号で置き換えて can_holdメソッドの実装を変更しましょう:

```
# fn main() {}
# #[derive(Debug)]
# pub struct Rectangle {
#     length: u32,
#     width: u32,
# }
// --snip--

```

```
impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length < other.length && self.width > other.width
    }
}
```

テストを実行すると、以下のような出力をします:

```
running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... FAILED

failures:

---- tests::larger_can_hold_smaller stdout ----
    thread 'tests::larger_can_hold_smaller' panicked at 'assertion failed:
    larger.can_hold(&smaller)', src/lib.rs:22:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

テストによりバグが捕捉されました！ larger.lengthが 8、smaller.lengthが 5 なので、can_hold 内の長さの比較が今は falseを返すようになったのです: 8 は 5 より小さくないですからね。

13.2.2　assert_eq!と assert_ne!マクロで等値性をテストする

機能をテストする一般的な方法は、テスト下にあるコードの結果をコードが返すと期待される値と比較して、等しいと確かめることです。これを assertマクロを使用して==演算子を使用した式を渡すことで行うこともできます。しかしながら、これはありふれたテストなので、標準ライブラリには 1 組のマクロ (assert_eq!と assert_ne!) が提供され、このテストをより便利に行うことができます。これらのマクロはそれぞれ、二つの引数を等値性と非等値性のために比較します。また、アサーションが失敗したら二つの値の出力もし、テストが失敗した**原因**を確認しやすくなります。一方で assert!マクロは、==式の値が false値になったことしか示唆せず、false値に導いた値は出力しません。

リスト 11-7 において、引数に 2を加えて結果を返す add_twoという名前の関数を書いています。そして、assert_eq!マクロでこの関数をテストしています。

ファイル名: src/lib.rs

```
# fn main() {}
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
```

```
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

リスト 11-7: assert_eq!マクロで add_two関数をテストする

テストが通ることを確認しましょう！

```
running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

assert_eq!マクロに与えた第 1 引数の 4は、add_two(2)の呼び出し結果と等しいです。このテストの行は test tests::it_adds_two ... okであり、okというテキストはテストが通ったことを示しています！

コードにバグを仕込んで、assert_eq!を使ったテストが失敗した時にどんな見た目になるのか確認してみましょう。add_two関数の実装を代わりに 3を足すように変えてください:

```
# fn main() {}
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

テストを再度実行します:

```
running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
        thread 'tests::it_adds_two' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `5`', src/lib.rs:11:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

テストがバグを捕捉しました！ it_adds_twoのテストは失敗し、assertion failed: `(left == right)`というメッセージを表示し、leftは 4で、rightは 5だったと示しています。このメッセージは有用で、デバッグを開始する助けになります: assert_eq!の left引数は 4だったが、add_two(2)がある right引数は 5だったことを意味しています。

ある言語やテストフレームワークでは、二つの値が等しいとアサーションを行う関数の引数は、expectedと actualと呼ばれ、引数を指定する順序が問題になることに注意してください。ですが Rust では、leftと rightと呼ばれ、期待する値とテスト下のコードが生成する値を指定する順序は、問題になりません。assert_eq!(add_two(2), 4)と今回のテストのアサーションを書くこともでき、そうすると失敗メッセージは、assertion failed: `(left == right)`と、leftが 5で rightが 4と表示されるわけです。

assert_ne!マクロは、与えた 2 つの値が等しくなければ通り、等しければ失敗します。このマクロは、値が何になるだろうか確信が持てないけれども、コードが意図した通りに動いていれば、確実にこの値にはならないだろうとわかっているような場合に最も有用になります。例えば、入力を何らかの手段で変えることが保障されているけれども、入力が変更される方法がテストを実行する曜日に依存する関数をテストしているなら、アサーションすべき最善の事柄は、関数の出力が入力と等しくないことかもしれません。

表面下では、assert_eq!と assert_ne!マクロはそれぞれ、==と!=演算子を使用しています。アサーションが失敗すると、これらのマクロは引数をデバッグフォーマットを使用して出力するので、比較対象の値は PartialEqと Debugトレイトを実装していなければなりません。組み込み型の全部と、標準ライブラリの型はほぼ全てこれらのトレイトを実装しています。自分で定義した構造体と enum については、PartialEqを実装して、その型の値が等しいか等しくないかアサーションする必要があるでしょう。Debugを実装して、アサーションが失敗した時に値を出力する必要もあるでしょう。第 5 章のリスト 5-12 で触れたように、どちらのトレイトも継承可能トレイトなので、これは通常、構造体や enum 定義に#[derive(PartialEq, Debug)]というアノテーションを追加するくらい率直になります。これらや他の継承可能トレイトに関する詳細については、おまけ C、「継承可能トレイト」をご覧ください。

13.2.3　カスタムの失敗メッセージを追加する

さらに、assert!、assert_eq!、assert_ne!の追加引数として、失敗メッセージと共にカスタムのメッセージが表示されるよう、追加することもできます。assert!の 1 つの必須引数、あるいは assert_eq!と assert_ne!の 2 つの必須引数の後に指定された引数はどれも format!マクロに明け渡されるので、(format!マクロについては第 8 章の「+演算子または、format!マクロで連結する」節で議論しました)、{}プレースホルダーを含むフォーマット文字列とこのプレースホルダーに置き換えられる値を渡すことができます。カスタムメッセージは、アサーションがどんな意味を持つかドキュメント化するのに役に立ちます; テストが失敗した時、問題が何なのかコードと共により良い考えを持てるでしょう。

例として、人々に名前で挨拶をする関数があり、関数に渡した名前が出力に出現することをテストしたいとしましょう:

ファイル名: src/lib.rs

```
# fn main() {}
```

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

このプログラムの必要事項はまだ合意が得られておらず、挨拶の先頭の `Hello`というテキストは変わるだろうということは確かです。要件が変わった時にテストを更新しなくてもよいようにしたいと決定したので、`greeting`関数から返る値と正確な等値性を確認するのではなく、出力が入力引数のテキストを含むことをアサーションするだけにします。

`greeting`が `name`を含まないように変更してこのコードにバグを仕込み、このテストの失敗がどんな見た目になるのか確かめましょう:

```
# fn main() {}
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

このテストを実行すると、以下のように出力されます:

```
running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
        thread 'tests::greeting_contains_name' panicked at 'assertion failed:
result.contains("Carol")', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::greeting_contains_name
```

この結果は、アサーションが失敗し、どの行にアサーションがあるかを示しているだけです。より有用な失敗メッセージは今回の場合、`greeting`関数から得た値を出力することでしょう。`greeting`関数から得た実際の値で埋められるプレースホルダーを含むフォーマット文字列からなるカスタムの失敗メッセージを与え、テスト関数を変更しましょう:

```
#[test]
fn greeting_contains_name() {
```

```
    let result = greeting("Carol");
    //挨拶は名前を含んでいません。値は`{}`でした
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`", result
    );
}
```

これでテストを実行したら、より有益なエラーメッセージが得られるでしょう:

```
---- tests::greeting_contains_name stdout ----
        thread 'tests::greeting_contains_name' panicked at 'Greeting did not
contain name, value was `Hello!`', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

実際に得られた値がテスト出力に見られ、起こると想定していたものではなく、起こったものをデバッグするのに役に立ちます。

13.2.4　should_panic でパニックを確認する

期待する正しい値をコードが返すことを確認することに加えて、想定通りにコードがエラー状態を扱っていることを確認するのも重要です。例えば、第 9 章のリスト 9-9 で生成した Guess型を考えてください。Guessを使用する他のコードは、Guessのインスタンスは 1 から 100 の範囲の値しか含まないという保証に依存しています。その範囲外の値で Guessインスタンスを生成しようとするとパニックすることを保証するテストを書くことができます。

これは、テスト関数に should_panicという別のアトリビュートを追加することで達成できます。このアトリビュートは、関数内のコードがパニックしたら、テストを通過させます。つまり、関数内のコードがパニックしなかったら、テストは失敗するわけです。

リスト 11-8 は、想定した時に Guess::newのエラー状態が発生していることを確認するテストを示しています:

ファイル名: src/lib.rs

```
# fn main() {}
pub struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 || value > 100 {
            //予想値は1から100の間でなければなりません
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
```

```
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

リスト 11-8: 状況が panic!を引き起こすとテストする

#[test]アトリビュートの後、適用するテスト関数の前に#[should_panic]アトリビュートを配置しています。このテストが通るときの結果を見ましょう:

```
running 1 test
test tests::greater_than_100 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

よさそうですね！ では、値が 100 より大きいときに new関数がパニックするという条件を除去することでコードにバグを導入しましょう:

```
# fn main() {}
# pub struct Guess {
#     value: u32,
# }
#
// --snip--

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1  {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }
}
```

リスト 11-8 のテストを実行すると、失敗するでしょう:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:
```

```
failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

この場合、それほど役に立つメッセージは得られませんが、テスト関数に目を向ければ、#[should_panic]で注釈されていることがわかります。得られた失敗は、テスト関数のコードがパニックを引き起こさなかったことを意味するのです。

should_panicを使用するテストは不正確なこともあります。なぜなら、コードが何らかのパニックを起こしたことしか示さないからです。should_panicのテストは、起きると想定していたもの以外の理由でテストがパニックしても通ってしまうのです。should_panicのテストの正確を期すために、should_panicアトリビュートの省略可能な expected引数を追加できます。このテストの拘束具が、失敗メッセージに与えられたテキストが含まれていることを確かめてくれるでしょう。例えば、リスト 11-9 の Guessの変更されたコードを考えてください。ここでは、new関数は、値の大小によって異なるメッセージでパニックします。

ファイル名: src/lib.rs

```
# fn main() {}
# pub struct Guess {
#     value: u32,
# }
#
// --snip--

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 {
            //予想値は、1以上でなければなりません
            panic!("Guess value must be greater than or equal to 1, got {}.",
                   value);
        } else if value > 100 {
            //予想値は100以下でなければなりません
            panic!("Guess value must be less than or equal to 100, got {}.",
                   value);
        }

        Guess {
            value
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    // 予想値は100以下でなければなりません
```

```
32      #[should_panic(expected = "Guess value must be less than or equal to 100")]
33      fn greater_than_100() {
34          Guess::new(200);
35      }
36  }
```

リスト 11-9: 状況が特定のパニックメッセージで panic!を引き起こすことをテストする

should_panicアトリビュートの expected引数に置いた値が Guess::new関数がパニックしたメッセージの一部になっているので、このテストは通ります。予想されるパニックメッセージ全体を指定することもでき、そうすれば今回の場合、Guess value must be less than or equal to 100, got 200.となります。should_panicの予想される引数に指定すると決めたものは、パニックメッセージの固有性や活動性、テストの正確性によります。今回の場合、パニックメッセージの一部でも、テスト関数内のコードが、else if value > 100ケースを実行していると確認するのに事足りるのです。

expectedメッセージありの should_panicテストが失敗すると何が起きるのが確かめるために、if value < 1と else if value > 100ブロックの本体を入れ替えることで再度コードにバグを仕込みましょう:

```
1  if value < 1 {
2      panic!("Guess value must be less than or equal to 100, got {}.", value);
3  } else if value > 100 {
4      panic!("Guess value must be greater than or equal to 1, got {}.", value);
5  }
```

should_panicテストを実行すると、今回は失敗するでしょう:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

---- tests::greater_than_100 stdout ----
        thread 'tests::greater_than_100' panicked at 'Guess value must be
greater than or equal to 1, got 200.', src/lib.rs:11:12
note: Run with `RUST_BACKTRACE=1` for a backtrace.
note: Panic did not include expected string 'Guess value must be less than or
equal to 100'

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

この失敗メッセージは、このテストが確かにまさしく予想通りパニックしたことを示唆していますが、パニックメッセージは、予想される文字列の'Guess value must be less than or equal to 100'を含んでいませんでした。実際に得られたパニックメッセージは今回の場合、Guess value must be greater than or equal to 1, got 200でした。そうしてバグの所在地を割り出し始めることができるわけです！

今やテスト記法を複数知ったので、テストを走らせる際に起きていることに目を向け、cargo test で使用できる色んなオプションを探求しましょう。

13.3　テストの実行され方を制御する

cargo runがコードをコンパイルし、出来上がったバイナリを走らせるのと全く同様に、cargo test はコードをテストモードでコンパイルし、出来上がったテストバイナリを実行します。コマンドラインオプションを指定して cargo testの規定動作を変更することができます。例えば、cargo testで生成されるバイナリの規定動作は、テストを全て並行に実行し、テスト実行中に生成された出力をキャプチャして出力が表示されるのを防ぎ、テスト結果に関係する出力を読みやすくすることです。

コマンドラインオプションの中には cargo testにかかるものや、出来上がったテストバイナリにかかるものがあります。この 2 種の引数を区別するために、cargo testにかかる引数を--という区分記号の後に列挙し、それからテストバイナリにかかる引数を列挙します。cargo test --helpを走らせると、cargo testで使用できるオプションが表示され、cargo test -- --helpを走らせると、--という区分記号の後に使えるオプションが表示されます。

13.3.1　テストを並行または連続して実行する

複数のテストを実行するとき、標準では、スレッドを使用して並行に走ります。これはつまり、テストが早く実行し終わり、コードが機能しているいかんにかかわらず、反応をより早く得られることを意味します。テストは同時に実行されているので、テストが相互や共有された環境を含む他の共通の状態に依存してないことを確かめてください。現在の作業対象ディレクトリや環境変数などですね。

例えば、各テストがディスクに*test_output.txt*というファイルを作成し、何らかのデータを書き込むコードを走らせるとしてください。そして、各テストはそのファイルのデータを読み取り、ファイルが特定の値を含んでいるとアサーションし、その値は各テストで異なります。テストが同時に走るので、あるテストが、他のテストが書き込んだり読み込んだりする間隙にファイルを上書きするかもしれません。それから 2 番目のテストが失敗します。コードが不正だからではなく、並行に実行されている間にテストがお互いに邪魔をしてしまったせいです。各テストが異なるファイルに書き込むことを確かめるのが一つの解決策です; 別の解決策では、一度に一つのテストを実行します。

並行にテストを実行したくなかったり、使用されるスレッド数をよりきめ細かく制御したい場合、--test-threadsフラグと使用したいスレッド数をテストバイナリに送ることができます。以下の例に目を向けてください:

```
$ cargo test -- --test-threads=1
```

テストスレッドの数を 1にセットし、並行性を使用しないようにプログラムに指示しています。1 スレッドのみを使用してテストを実行すると、並行に実行するより時間がかかりますが、状態を共有していても、お互いに邪魔をすることはありません。

13.3.2　関数の出力を表示する

標準では、テストが通ると、Rust のテストライブラリは標準出力に出力されたものを全てキャプチャします。例えば、テストで println! を呼び出してテストが通ると、println! の出力は、ターミナルに表示されません; テストが通ったことを示す行しか見れないでしょう。テストが失敗すれば、残りの失敗メッセージと共に、標準出力に出力されたものが全て見えるでしょう。

例として、リスト 11-10 は引数の値を出力し、10 を返す馬鹿げた関数と通過するテスト 1 つ、失敗するテスト 1 つです。

ファイル名: src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    //{}という値を得た
    println!("I got the value {}", a);
    10
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

リスト 11-10: println! を呼び出す関数用のテスト

これらのテストを cargo test で実行すると、以下のような出力を目の当たりにするでしょう:

```
running 2 tests
test tests::this_test_will_pass ... ok
test tests::this_test_will_fail ... FAILED

failures:

---- tests::this_test_will_fail stdout ----
        I got the value 8
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

```
failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

この出力のどこにも I got the value 4という出力が見られないことに注意してください。この出力は、通るテストが走る際に出力されるものです。この出力はキャプチャされてしまったのです。失敗したテストからの出力である I got the value 8はテスト総括出力の区域に出現し、ここには、テスト失敗の原因も表示されています。

通過するテストについても出力される値が見たかったら、出力キャプチャ機能を--nocaptureフラグで無効化することができます:

```
$ cargo test -- --nocapture
```

リスト 11-10 のテストを--nocaptureフラグと共に再度実行したら、以下のような出力を目の当たりにします:

```
running 2 tests
I got the value 4
I got the value 8
test tests::this_test_will_pass ... ok
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
test tests::this_test_will_fail ... FAILED

failures:

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

テスト用の出力とテスト結果の出力がまぜこぜになっていることに注意してください; その理由は、前節で語ったようにテストが並行に実行されているからです。-test-threads=1オプションと--nocaptureフラグを使ってみて、出力がどうなるか確かめてください！

13.3.3　名前でテストの一部を実行する

時々、全テストを実行すると時間がかかってしまうことがあります。特定の部分のコードしか対象にしていない場合、そのコードに関わるテストのみを走らせたいかもしれません。cargo testに走らせたいテストの名前を引数として渡すことで、実行するテストを選ぶことができます。

テストの一部を走らせる方法を模擬するために、リスト 11-11 に示したように、add_two関数用に 3 つテストを作成し、走らせるテストを選択します:

ファイル名: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

リスト 11-11: 3 つの異なる名前の 3 つのテスト

以前見かけたように、引数なしでテストを走らせたら、全テストが並行に走ります:

```
running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

13.3.3.1　単独のテストを走らせる

あらゆるテスト関数の名前を `cargo test`に渡して、そのテストのみを実行することができます:

```
$ cargo test one_hundred
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-06a75b4a1f2515e9

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out
```

`one_hundred`という名前のテストだけが走りました; 他の 2 つのテストはその名前に合致しなかったのです。まとめ行の最後に `2 filtered out`と表示することでテスト出力は、このコマンドが走らせた以上のテストがあることを知らせてくれています。

この方法では、複数のテストの名前を指定することはできません; cargo testに与えられた最初の値のみが使われるのです。ですが、複数のテストを走らせる方法もあります。

13.3.3.2　複数のテストを実行するようフィルターをかける

テスト名の一部を指定でき、その値に合致するあらゆるテストが走ります。例えば、我々のテストの 2 つが addという名前を含むので、cargo test addを実行することで、その二つを走らせることができます:

```
$ cargo test add
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-06a75b4a1f2515e9

running 2 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

このコマンドは名前に addを含むテストを全て実行し、one_hundredという名前のテストを除外しました。また、テストが出現するモジュールがテスト名の一部になっていて、モジュール名でフィルターをかけることで、あるモジュール内のテスト全てを実行できることに注目してください。

13.3.4　特に希望のない限りテストを無視する

時として、いくつかの特定のテストが実行するのに非常に時間がかかることがあり、cargo testの実行のほとんどで除外したくなるかもしれません。引数として確かに実行したいテストを全て列挙するのではなく、ここに示したように代わりに時間のかかるテストを ignoreアトリビュートで除外すると注釈することができます。

ファイル名: src/lib.rs

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // 実行に1時間かかるコード
    // code that takes an hour to run
}
```

#[test]の後の除外したいテストに#[ignore]行を追加しています。これで、テストを実行したら、it_worksは実行されるものの、expensive_testは実行されません:

```
$ cargo test
```

```
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs
     Running target/debug/deps/adder-ce99bcc2479f4607

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out
```

expensive_test関数は、ignoredと列挙されています。無視されるテストのみを実行したかったら、cargo test -- --ignoredを使うことができます:

```
$ cargo test -- --ignored
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

どのテストを走らせるか制御することで、結果が早く出ることを確かめることができるのです。ignoredテストの結果を確認することが道理に合い、結果を待つだけの時間ができたときに、代わりに cargo test -- --ignoredを走らせることができます。

13.4　テストの体系化

章の初めで触れたように、テストは複雑な鍛錬であり、人によって専門用語や体系化が異なります。Rust のコミュニティでは、テストを 2 つの大きなカテゴリで捉えています: **単体テスト**と**結合テスト**です。単体テストは小規模でより集中していて、個別に 1 回に 1 モジュールをテストし、非公開のインターフェイスもテストする可能性があります。結合テストは、完全にライブラリ外になり、他の外部コード同様に自分のコードを使用し、公開インターフェイスのみ使用し、1 テストにつき複数のモジュールを用いることもあります。

どちらのテストを書くのも、ライブラリの一部が個別かつ共同でしてほしいことをしていることを確認するのに重要なのです。

13.4.1　単体テスト

単体テストの目的は、残りのコードから切り離して各単位のコードをテストし、コードが想定通り、動いたり動いていなかったりする箇所を急速に特定することです。単体テストは、テスト対象となるコードと共に、*src* ディレクトリの各ファイルに置きます。慣習は、各ファイルに testsという名前のモジュールを作り、テスト関数を含ませ、そのモジュールを cfg(test)で注釈することです。

13.4.1.1　テストモジュールと#[cfg(test)]

testsモジュールの#[cfg(test)]という注釈は、コンパイラにcargo buildを走らせた時ではなく、cargo testを走らせた時にだけ、テストコードをコンパイルし走らせるよう指示します。これにより、ライブラリをビルドしたいだけの時にコンパイルタイムを節約し、テストが含まれないので、コンパイル後の成果物のサイズも節約します。結合テストは別のディレクトリに存在することになるので、#[cfg(test)]注釈は必要ないとわかるでしょう。しかしながら、単体テストはコードと同じファイルに存在するので、#[cfg(test)]を使用してコンパイル結果に含まれないよう指定するのです。

この章の最初の節で新しいadderプロジェクトを生成した時に、Cargoがこのコードも生成してくれたことを思い出してください:

ファイル名: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

このコードが自動生成されたテストモジュールです。cfgというアトリビュートは、*configuration*を表していて、コンパイラに続く要素が、ある特定の設定オプションを与えて含まれるように指示します。今回の場合、設定オプションは、testであり、言語によって提供されているテストをコンパイルし、走らせるためのものです。cfgアトリビュートを使用することで、cargo testで積極的にテストを実行した場合のみ、Cargoがテストコードをコンパイルします。これには、このモジュールに含まれるかもしれないヘルパー関数全ても含まれ、#[test]で注釈された関数だけにはなりません。

13.4.1.2　非公開関数をテストする

テストコミュニティ内で非公開関数を直接テストするべきかについては議論があり、他の言語では非公開関数をテストするのは困難だったり、不可能だったりします。あなたがどちらのテストイデオロギーを支持しているかに関わらず、Rustの公開性規則により、非公開関数をテストすることが確かに可能です。リスト 11-12 の非公開関数internal_adderを含むコードを考えてください:

ファイル名: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
```

```
mod tests {
    use super::*;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

リスト 11-12: 非公開関数をテストする

`internal_adder`関数は `pub`とマークされていないものの、テストも単なる Rust のコードであり、`tests`モジュールもただのモジュールでしかないので、テスト内で `internal_adder`を普通にインポートし呼び出すことができます。非公開関数はテストするべきではないとお考えなら、Rust にはそれを強制するものは何もありません。

13.4.2　結合テスト

Rust において、結合テストは完全にライブラリ外のものです。他のコードがするのと全く同様にあなたのライブラリを使用するので、ライブラリの公開 API の一部である関数しか呼び出すことはできません。その目的は、ライブラリのいろんな部分が共同で正常に動作しているかをテストすることです。単体では正常に動くコードも、結合した状態だと問題を孕む可能性もあるので、結合したコードのテストの範囲も同様に重要になるのです。結合テストを作成するには、まず *tests* ディレクトリが必要になります。

13.4.2.1　*tests* ディレクトリ

プロジェクトディレクトリのトップ階層、*src* の隣に *tests* ディレクトリを作成します。Cargo は、このディレクトリに結合テストのファイルを探すことを把握しています。そして、このディレクトリ内にいくらでもテストファイルを作成することができ、Cargo はそれぞれのファイルを個別のクレートとしてコンパイルします。

結合テストを作成しましょう。リスト 11-12 のコードが *src/lib.rs* ファイルにあるまま、*tests* ディレクトリを作成し、*tests/integration_test.rs*という名前の新しいファイルを生成し、リスト 11-13 のコードを入力してください:

ファイル名: tests/integration_test.rs

```
extern crate adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

リスト 11-13: `adder`クレートの関数の結合テスト

コードの頂点に `extern crate adder`を追記しましたが、これは単体テストでは必要なかったものです。理由は、`tests`ディレクトリのテストはそれぞれ個別のクレートであるため、各々ライブラリをインポートする必要があるためです。

*tests/integration_test.rs*のどんなコードも`#[cfg(test)]`で注釈する必要はありません。Cargo は `tests`ディレクトリを特別に扱い、`cargo test`を走らせた時にのみこのディレクトリのファイルをコンパイルするのです。さあ、`cargo test`を実行してください:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running target/debug/deps/adder-abcabcabc

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

     Running target/debug/deps/integration_test-ce99bcc2479f4607

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

3 つの区域の出力が単体テスト、結合テスト、ドックテストを含んでいます。単体テスト用の最初の区域は、今まで見てきたものと同じです: 各単体テストに 1 行 (リスト 11-12 で追加した `internal`という名前のもの) と、単体テストの総括行です。

結合テストの区域は、`Running target/debug/deps/integration-test-ce99bcc2479f4607`という行で始まっています (最後のハッシュは違うでしょう)。次に、この結合テストの各テスト関数用の行があり、`Doc-tests adder`区域が始まる直前に、結合テストの結果用の総括行があります。

単体テスト関数を追加すると、単体テスト区域のテスト結果の行が増えることに似て、作成した結合テストファイルにもっとテスト関数を追加すると、そのファイルの区域に行が増えることになります。結合テストファイルはそれぞれ独自の区域があるため、*tests* ディレクトリにさらにファイルを追加すれば、結合テストの区域が増えることになるでしょう。

それでも、テスト関数の名前を引数として`cargo test`に指定することで、特定の結合テスト関数を走らせることができます。特定の結合テストファイルにあるテストを全て走らせるには、`cargo test`に`--test`引数、その後にファイル名を続けて使用してください:

```
$ cargo test --test integration_test
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
```

```
     Running target/debug/integration_test-952a27e0126bb565

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

このコマンドは、*tests/integration_test.rs*ファイルにあるテストのみを実行します。

13.4.2.2　結合テスト内のサブモジュール

結合テストを追加するにつれて、*tests* ディレクトリに 2 つ以上のファイルを作成して体系化したくなるかもしれません; 例えば、テスト対象となる機能でテスト関数をグループ化することができます。前記したように、*tests* ディレクトリの各ファイルは、個別のクレートとしてコンパイルされます。

各結合テストファイルをそれ自身のクレートとして扱うと、エンドユーザが読者のクレートを使用するかのような個別のスコープを生成するのに役立ちます。ですが、これは *tests* ディレクトリのファイルが、コードをモジュールとファイルに分ける方法に関して第 7 章で学んだように、*src* のファイルとは同じ振る舞いを共有しないことを意味します。

tests ディレクトリのファイルの異なる振る舞いは、複数の結合テストファイルで役に立ちそうなヘルパー関数ができ、第 7 章の「モジュールを別のファイルに移動する」節の手順に従って共通モジュールに抽出しようとした時に最も気付きやすくなります。例えば、*tests/common.rs* を作成し、そこに `setup`という名前の関数を配置したら、複数のテストファイルの複数のテスト関数から呼び出したい `setup`に何らかのコードを追加することができます:

ファイル名: tests/common.rs

```
pub fn setup() {
    // ここにライブラリテスト固有のコードが来る
    // setup code specific to your library's tests would go here
}
```

再度テストを実行すると、*common.rs* ファイルは何もテスト関数を含んだり、`setup`関数をどこかから呼んだりしてないのに、テスト出力に *common.rs* 用の区域が見えるでしょう。

```
running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

     Running target/debug/deps/common-b8b07b6f1be2db70

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

     Running target/debug/deps/integration_test-d993c68b431d39df

running 1 test
```

```
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

commonが running 0 testsとテスト結果に表示されるのは、望んだ結果ではありません。ただ単に他の結合テストファイルと何らかのコードを共有したかっただけです。

commonがテスト出力に出現するのを防ぐには、*tests/common.rs* を作成する代わりに、*tests/common/mod.rs* を作成します。第 7 章の「モジュールファイルシステムの規則」節において、*module_name/mod.rs という命名規則をサブモジュールのあるモジュールのファイルに使用しました。ここでは *common*にサブモジュールはありませんが、このように命名することでコンパイラに *common*モジュールを結合テストファイルとして扱わないように指示します。*setup*関数のコードを tests/common/mod.rs に移動し、tests/common.rs ファイルを削除すると、テスト出力に区域はもう表示されなくなります。tests*ディレクトリのサブディレクトリ内のファイルは個別クレートとしてコンパイルされたり、テスト出力に区域が表示されることがないのです。

tests/common.mod.rs を作成した後、それをどの結合テストファイルからもモジュールとして使用することができます。こちらは、*tests/integration_test.rs*内の it_adds_twoテストから setup関数を呼び出す例です:

ファイル名: tests/integration_test.rs

```
extern crate adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

mod common;という宣言は、リスト 7-4 で模擬したモジュール宣言と同じであることに注意してください。それから、テスト関数内で common::setup()関数を呼び出すことができます。

13.4.2.3　バイナリクレート用の結合テスト

もしもプロジェクトが *src/main.rs* ファイルのみを含み、*src/lib.rs* ファイルを持たないバイナリクレートだったら、*tests* ディレクトリに結合テストを作成し、extern crateを使用して *src/main.rs* ファイルに定義された関数をインポートすることはできません。ライブラリクレートのみが、他のクレートが呼び出して使用できる関数を晒け出せるのです; バイナリクレートはそれ単体で実行するという意味を持っています。

これは、バイナリを提供する Rust のプロジェクトに、*src/lib.rs* ファイルに存在するロジックを呼び出す素直な *src/main.rs* ファイルがある一因になっています。この構造を使用して結合テストは、`extern crate`を使用して重要な機能を用いることでライブラリクレートをテストすることができます。この重要な機能が動作すれば、*src/main.rs* ファイルの少量のコードも動作し、その少量のコードはテストする必要がないわけです。

13.5　まとめ

Rust のテスト機能は、変更を加えた後でさえ想定通りにコードが機能し続けることを保証して、コードが機能すべき方法を指定する手段を提供します。単体テストはライブラリの異なる部分を個別に用い、非公開の実装詳細をテストすることができます。結合テストは、ライブラリのいろんな部分が共同で正常に動作することを確認し、ライブラリの公開 API を使用して外部コードが使用するのと同じ方法でコードをテストします。Rust の型システムと所有権ルールにより防がれるバグの種類もあるものの、それでもテストは、コードの振る舞い方に関するロジックのバグを減らすのに重要なのです。

この章と以前の章で学んだ知識を結集して、とあるプロジェクトに取り掛かりましょう！

第 14 章

入出力プロジェクト: コマンドラインプログラムを構築する

この章は、ここまでに学んできた多くのスキルを思い出すきっかけであり、もういくつか標準ライブラリの機能も探求します。ファイルやコマンドラインの入出力と相互作用するコマンドラインツールを構築し、今やあなたの支配下にある Rust の概念の一部を練習していきます。

Rust の速度、安全性、単バイナリ出力、クロスプラットフォームサポートにより、コマンドラインツールを作るのにふさわしい言語なので、このプロジェクトでは、独自の伝統的なコマンドラインツールの `grep`(**g**lobally search a **r**egular **e**xpression and **p**rint: 正規表現をグローバルで検索し表示する) を作成していきます。最も単純な使用法では、`grep`は指定したファイルから指定した文字列を検索します。そうするには、`grep`は引数としてファイル名と文字列を受け取ります。それからファイルを読み込んでそのファイル内で文字列引数を含む行を探し、検索した行を出力するのです。

その過程で、多くのコマンドラインツールが使用している端末の機能を使用させる方法を示します。環境変数の値を読み取ってユーザがこのツールの振る舞いを設定できるようにします。また、標準出力 (`stdout`) の代わりに、標準エラー出力 (`stderr`) するので、例えば、ユーザはエラーメッセージはスクリーン上で確認しつつ、成功した出力はファイルにリダイレクトできます。

ある Rust コミュニティのメンバであるアンドリュー・ガラント (Andrew Gallant) が既に全機能装備の非常に高速な `grep`、`ripgrep`と呼ばれるものを作成しました。比較対象として、我々の `grep`はとても単純ですが、この章により、`ripgrep`のような現実世界のプロジェクトを理解するのに必要な背景知識の一部を身に付けられるでしょう。

この `grep`プロジェクトは、ここまでに学んできた多くの概念を集結させます:

- コードを体系化する (モジュール、第 7 章で学んだことを使用)
- ベクタ型と文字列を使用する (コレクション、第 8 章)
- エラーを処理する (第 9 章)
- 適切な箇所でトレイトとライフタイムを使用する (第 10 章)

- テストを記述する (第 11 章)

さらに、クロージャ、イテレータ、トレイトオブジェクトなど、第 13 章、17 章で詳しく解説するものもちょっとだけ紹介します。

14.1　コマンドライン引数を受け付ける

いつものように、cargo newで新しいプロジェクトを作りましょう。プロジェクトを minigrepと名付けて、既に自分のシステムに存在するかもしれない grepツールと区別しましょう。

```
$ cargo new --bin minigrep
     Created binary (application) `minigrep` project
$ cd minigrep
```

最初の仕事は、minigrepを二つの引数を受け付けるようにすることです: ファイル名と検索する文字列ですね。つまり、cargo runで検索文字列と検索を行うファイルへのパスと共にプログラムを実行できるようになりたいということです。こんな感じにね:

```
$ cargo run searchstring example-filename.txt
```

今現在は、cargo newで生成されたプログラムは、与えた引数を処理できません。Crates.io に存在するある既存のライブラリは、コマンドライン引数を受け付けるプログラムを書く手助けをしてくれますが、ちょうどこの概念を学んでいる最中なので、この能力を自分で実装しましょう。

14.1.1　引数の値を読み取る

minigrepが渡したコマンドライン引数の値を読み取れるようにするために、Rust の標準ライブラリで提供されている関数が必要になり、それは、std::env::argsです。この関数は、minigrepに与えられたコマンドライン引数の**イテレータ**を返します。イテレータについてはまだ議論していません (完全には第 13 章で解説します) が、とりあえずイテレータに関しては、2 つの詳細のみ知っていればいいです: イテレータは一連の値を生成することと、イテレータに対して collect関数を呼び出し、イテレータが生成する要素全部を含むベクタ型などのコレクションに変えられることです。

リスト 12-1 のコードを使用して minigrepプログラムに渡されたあらゆるコマンドライン引数を読み取れるようにし、それからその値をベクタ型として集結させてください:

ファイル名: src/main.rs

```
1 use std::env;
2
3 fn main() {
4     let args: Vec<String> = env::args().collect();
5     println!("{:?}", args);
6 }
```

リスト 12-1: コマンドライン引数をベクタ型に集結させ、出力する

まず、std::envモジュールを use文でスコープに導入したので、args関数が使用できます。std::env::args関数は、2 レベルモジュールがネストされていることに気付いてください。第 7 章で議論したように、希望の関数が 2 モジュール以上ネストされている場合、関数ではなく親モジュールをスコープに導入するのが因習的です。そうすることで、std::envから別の関数も容易に使用することができます。また、use std::env::argsを追記し、関数を argsとするだけで呼び出すのに比べて曖昧でもありません。というのも、argsは現在のモジュールに定義されている関数と容易に見間違えられるかもしれないからです。

14.1.2　args 関数と不正なユニコード

引数のどれかが不正なユニコードを含んでいたら、std::env::argsはパニックすることに注意してください。プログラムが不正なユニコードを含む引数を受け付ける必要があるなら、代わりに std::env::args_osを使用してください。この関数は、String値ではなく、OsString値を生成するイテレータを返します。ここでは、簡潔性のために std::env::argsを使うことを選択しました。なぜなら、OsString値はプラットフォームごとに異なり、String値に比べて取り扱いが煩雑だからです。

mainの最初の行で env::argsを呼び出し、そして即座に collectを使用して、イテレータをイテレータが生成する値全てを含むベクタ型に変換しています。collect関数を使用して多くの種類のコレクションを生成することができるので、argsの型を明示的に注釈して文字列のベクタが欲しいのだと指定しています。Rust において、型を注釈しなければならない頻度は非常に少ないのですが、collectはよく確かに注釈が必要になる一つの関数なのです。コンパイラには、あなたが欲しているコレクションの種類が推論できないからです。

最後に、デバッグ整形機の:?を使用してベクタを出力しています。引数なしでコードを走らせてみて、それから引数二つで試してみましょう:

```
$ cargo run
--snip--
["target/debug/minigrep"]

$ cargo run needle haystack
--snip--
["target/debug/minigrep", "needle", "haystack"]
```

ベクタの最初の値は"target/debug/minigrep"であり、これはバイナリの名前であることに気付いてください。これは C の引数リストの振る舞いと合致し、実行時に呼び出された名前をプログラムに使わせてくれるわけです。メッセージで出力したり、プログラムを起動するのに使用されたコマンドラインエイリアスによってプログラムの振る舞いを変えたい場合に、プログラム名にアクセスするのにしばしば便利です。ですが、この章の目的には、これを無視し、必要な二つの引数のみを保存します。

14.1.3　引数の値を変数に保存する

引数のベクタの値を出力すると、プログラムはコマンドライン引数として指定された値にアクセスできることが具現化されました。さて、プログラムの残りを通して使用できるように、二つの引数の値を変数に保存する必要があります。それをしているのがリスト 12-2 です:

ファイル名: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let filename = &args[2];

    // {}を探しています
    println!("Searching for {}", query);
    // {}というファイルの中
    println!("In file {}", filename);
}
```

リスト 12-2: クエリ引数とファイル名引数を保持する変数を生成

ベクタを出力した時に確認したように、プログラム名がベクタの最初の値、`args[0]`を占めているので、添字 `1`から始めます。`minigrep`が取る最初の引数は、検索する文字列なので、最初の引数への参照を変数 `query`に置きました。2 番目の引数はファイル名でしょうから、2 番目の引数への参照は変数 `filename`に置きました。

一時的にこれら変数の値を出力して、コードが意図通りに動いていることを証明しています。再度このプログラムを `test`と `sample.txt`という引数で実行しましょう:

```
$ cargo run test sample.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

素晴らしい、プログラムは動作しています！ 必要な引数の値が、正しい変数に保存されています。後ほど、何らかのエラー処理を加えて、特定の可能性のあるエラー状況に対処します。ユーザが引数を提供しなかった場合などです; 今は、そのような状況はないものとし、代わりにファイル読み取り能力を追加することに取り組みます。

14.2　ファイルを読み込む

では、filenameコマンドライン引数で指定されたファイルを読み込む機能を追加しましょう。まず、テスト実行するためのサンプルファイルが必要ですね: minigrepが動作していることを確かめるために使用すべき好適なファイルは、複数行にわたって同じ単語の繰り返しのある少量のテキストです。リスト 12-3 は、うまくいくであろうエミリー・ディキンソン (Emily Dickinson) の詩です！ プロジェクトのルート階層に *poem.txt* というファイルを作成し、この詩「私は誰でもない！ あなたは誰？」を入力してください。

ファイル名: poem.txt

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us — don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!

私は誰でもない！あなたは誰？
あなたも誰でもないの？
なら、私たちは組だね、何も言わないで！
あの人たちは、私たちを追放するでしょう。わかりますよね？

誰かでいるなんて侘しいじゃない！
カエルみたいで公すぎるじゃない。
自分の名を長い1日に告げるのなんて。
感服するような沼地にね！
```

リスト 12-3: エミリー・ディキンソンの詩は、いいテストケースになる

テキストを適当な場所に置いて、*src/main.rs* を編集し、ファイルを開くコードを追加してください。リスト 12-4 に示したようにね:

ファイル名: src/main.rs

```
use std::env;
use std::fs::File;
use std::io::prelude::*;

fn main() {
#     let args: Vec<String> = env::args().collect();
#
#     let query = &args[1];
#     let filename = &args[2];
#
#     println!("Searching for {}", query);
    // --snip--
    println!("In file {}", filename);
```

```
14
15      // ファイルが見つかりませんでした
16      let mut f = File::open(filename).expect("file not found");
17
18      let mut contents = String::new();
19      // ファイルの読み込み中に問題がありました
20      f.read_to_string(&mut contents)
21          .expect("something went wrong reading the file");
22
23      // テキストは\n{}です
24      println!("With text:\n{}", contents);
25  }
```

リスト 12-4: 第 2 引数で指定されたファイルの中身を読み込む

最初に、もう何個か use文を追記して、標準ライブラリの関係のある箇所を持ってきています: ファイルを扱うのに std::fs::Fileが必要ですし、std::io::prelude::*はファイル入出力を含む入出力処理をするのに有用なトレイトを色々含んでいます。言語が一般的な初期化処理で特定の型や関数を自動的にスコープに導入するように、std::ioモジュールにはそれ独自の共通の型や関数の初期化処理があり、入出力を行う際に必要になるわけです。標準の初期化処理とは異なり、std::ioの初期化処理には明示的に use文を加えなければなりません。

mainで 3 文を追記しました: 一つ目が、File::open関数を呼んで filename変数の値に渡して、ファイルへの可変なハンドルを得る処理です。二つ目が、contentsという名の変数を生成して、可変で空の Stringを割り当てる処理です。この変数が、ファイル読み込み後に中身を保持します。三つ目が、ファイルハンドルに対して read_to_stringを呼び出し、引数として contentsへの可変参照を渡す処理です。

それらの処理の後に、今回もファイル読み込み後に contentsの値を出力する一時的な println!文を追記したので、ここまでプログラムがきちんと動作していることを確認できます。

第 1 コマンドライン引数には適当な文字列 (まだ検索する箇所は実装してませんからね) を、第 2 引数に *poem.txt* ファイルを入れて、このコードを実行しましょう:

```
$ cargo run the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us — don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

素晴らしい！ コードがファイルの中身を読み込み、出力するようになりました。しかし、このコードにはいくつか欠陥があります。main関数が複数の責任を受け持っています: 一般に、各関数がただ一つの責任だけを持つようになれば、関数は明確に管理しやすくなります。もう一つの問題点は、できうる限りのエラー処理を怠っていることです。まだプログラムが小規模なので、これらの欠陥は大きな問題にはなりませんが、プログラムが大規模になるにつれ、それを綺麗に解消するのは困難になっていきます。プログラムを開発する際に早い段階でリファクタングを行うのは、良い傾向です。リファクタリングするコードの量が少なければ、簡単になりますからね。次は、それを行いましょう。

14.3　リファクタリングしてモジュール性の向上とエラー処理

プログラムを改善するために、プログラムの構造と起こりうるエラーに対処する方法に関連する 4 つの問題を解消していきましょう。

1 番目は、main関数が 2 つの仕事を受け持っていることです: 引数を解析し、ファイルを開いています。このような小さな関数なら、これは、大した問題ではありませんが、main内でプログラムを巨大化させ続けたら、main関数が扱う個別の仕事の数も増えていきます。関数が責任を受け持つごとに、内容が把握しにくくなり、テストも行いづらくなり、機能を壊さずに変更するのも困難になっていきます。機能を小分けして、各関数が 1 つの仕事のみに責任を持つようにするのが最善です。

この問題は、2 番目の問題にも紐付いています: queryと filenameはプログラムの設定用変数ですが、fや contentsといった変数は、プログラムのロジックを担っています。mainが長くなるほど、スコープに入れるべき変数も増えます。そして、スコープにある変数が増えれば、各々の目的を追うのも大変になるわけです。設定用変数を一つの構造に押し込め、目的を明瞭化するのが最善です。

3 番目の問題は、ファイルを開き損ねた時に expectを使ってエラーメッセージを出力しているのに、エラーメッセージがファイルが見つかりませんでしたとしか表示しないことです。ファイルを開く行為は、ファイルが存在しない以外にもいろんな方法で失敗することがあります: 例えば、ファイルは存在するかもしれないけれど、開く権限がないかもしれないなどです。現時点では、そのような状況になった時、「ファイルが見つかりませんでした」というエラーメッセージを出力し、これはユーザに間違った情報を与えるのです。

4 番目は、異なるエラーを処理するのに expectを繰り返し使用しているので、ユーザが十分な数の引数を渡さずにプログラムを起動した時に、問題を明確に説明しない「範囲外アクセス (index out of bounds)」というエラーが Rust から得られることです。エラー処理のコードが全て 1 箇所に存在し、将来エラー処理ロジックが変更になった時に、メンテナンス者が 1 箇所のコードのみを考慮すればいいようにするのが最善でしょう。エラー処理コードが 1 箇所にあれば、エンドユーザにとって意味のあるメッセージを出力していることを確認することにもつながります。

プロジェクトをリファクタリングして、これら 4 つの問題を扱いましょう。

14.3.1　バイナリプロジェクトの責任の分離

main関数に複数の仕事の責任を押し付けるという構造上の問題は、多くのバイナリプロジェクトでありふれています。結果として、mainが肥大化し始めた際にバイナリプログラムの個別の責任を分割するためにガイドラインとして活用できる工程を Rust コミニュティは、開発しました。この工程は、以下のような手順になっています:

- プログラムを *main.rs* と *lib.rs* に分け、ロジックを *lib.rs* に移動する。
- コマンドライン引数の解析ロジックが小規模な限り、*main.rs* に置いても良い。
- コマンドライン引数の解析ロジックが複雑化の様相を呈し始めたら、*main.rs* から抽出して *lib.rs* に移動する。
- この工程の後に main関数に残る責任は以下に限定される:
  - 引数の値でコマンドライン引数の解析ロジックを呼び出す
  - 他のあらゆる設定を行う
  - *lib.rs* の run関数を呼び出す
  - runがエラーを返した時に処理する

このパターンは、責任の分離についてです: *main.rs* はプログラムの実行を行い、そして、*lib.rs* が手にある仕事のロジック全てを扱います。main関数を直接テストすることはできないので、この構造により、プログラムのロジック全てを *lib.rs* の関数に移すことでテストできるようになります。*main.rs* に残る唯一のコードは、読めばその正当性が評価できるだけ小規模になるでしょう。この工程に従って、プログラムのやり直しをしましょう。

14.3.1.1　引数解析器を抽出する

引数解析の機能を mainが呼び出す関数に抽出して、コマンドライン引数解析ロジックを *src/lib.rs* に移動する準備をします。リスト 12-5 に新しい関数 parse_configを呼び出す mainの冒頭部を示し、この新しい関数は今だけ *src/main.rs* に定義します。

ファイル名: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, filename) = parse_config(&args);

    // --snip--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
```

```
}
```

リスト 12-5: mainから parse_config関数を抽出する

それでもまだ、コマンドライン引数をベクタ型に集結させていますが、main関数内で引数の値の添字 1を変数 queryに、添字 2を変数 filenameに代入する代わりに、ベクタ全体を parse_config関数に渡しています。そして、parse_config関数にはどの引数がどの変数に入り、それらの値を mainに返すというロジックが存在します。まだ main内で queryと filenameという変数を生成していますが、もう mainは、コマンドライン引数と変数がどう対応するかを決定する責任は持ちません。

This rework may seem like overkill for our small program, but we're refactoring in small, incremental steps. After making this change, run the program again to verify that the argument parsing still works. It's good to check your progress often, to help identify the cause of problems when they occur.

このやり直しは、私たちの小規模なプログラムにはやりすぎに思えるかもしれませんが、少しずつ段階的にリファクタリングしているのです。この変更後、プログラムを再度実行して、引数解析がまだ動作していることを実証してください。問題が発生した時に原因を特定する助けにするために頻繁に進捗を確認するのはいいことです。

14.3.1.2　設定値をまとめる

もう少し parse_config関数を改善することができます。現時点では、タプルを返していますが、即座にタプルを分解して再度個別の値にしています。これは、正しい抽象化をまだできていないかもしれない兆候です。

まだ改善の余地があると示してくれる他の徴候は、parse_configの configの部分であり、返却している二つの値は関係があり、一つの設定値の一部にどちらもなることを暗示しています。現状では、一つのタプルにまとめていること以外、この意味をデータの構造に載せていません; この二つの値を 1 構造体に置き換え、構造体のフィールドそれぞれに意味のある名前をつけることもできるでしょう。そうすることで将来このコードのメンテナンス者が、異なる値が相互に関係する仕方や、目的を理解しやすくできるでしょう。

> 注釈: この複雑型 (complex type) がより適切な時に組み込みの値を使うアンチパターンを、*primitive obsession*(脚注: 初めて聞いた表現。**組み込み型強迫観念**といったところだろうか) と呼ぶ人もいます。

リスト 12-6 は、Configという名前の構造体が queryと filenameというフィールドを持つよう定義された追加分を示しています。また、parse_config関数を Config構造体のインスタンスを返すように変え、mainも個別の変数ではなく構造体のフィールドを使うように更新しました。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs::File;
#
```

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    let mut f = File::open(config.filename).expect("file not found");

    // --snip--
}

struct Config {
    query: String,
    filename: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let filename = args[2].clone();

    Config { query, filename }
}
```

リスト 12-6: parse_configをリファクタリングして Config構造体のインスタンスを返すようにした

parse_configのシグニチャは、これで Config値を返すと示すようになりました。parse_configの本体では、以前は argsの String値を参照する文字列スライスを返していましたが、今では所有する String値を含むように Configを定義しています。mainの args変数は引数値の所有者であり、parse_config関数だけに借用させていますが、これは Configが argsの値の所有権を奪おうとしたら、Rust の借用規則に違反してしまうことを意味します。

Stringのデータは、多くの異なる手法で管理できますが、最も単純だけれどもどこか非効率的な手段は、値に対して cloneメソッドを呼び出すことです。これにより、Configインスタンスが所有するデータの総コピーが生成されるので、文字列データへの参照を保持するよりも時間とメモリを消費します。ですが、データをクローンすることで、コードがとても素直にもなります。というのも、参照のライフタイムを管理する必要がないからです。つまり、この場面において、少々のパフォーマンスを犠牲にして単純性を得るのは、価値のある交換です。

14.3.2　clone を使用するトレードオフ

実行時コストのために cloneを使用して所有権問題を解消するのを避ける傾向が多くの Rust 市民にあります。第 13 章で、この種の状況においてより効率的なメソッドの使用法を学ぶでしょう。ですがとりあえずは、これらのコピーをするのは 1 回だけですし、ファイル名とクエ

> リ文字列は非常に小さなものなので、いくつかの文字列をコピーして進捗するのは良しとしましょう。最初の通り道でコードを究極的に効率化しようとするよりも、ちょっと非効率的でも動くプログラムを用意する方がいいでしょう。もっと Rust の経験を積めば、最も効率的な解決法から開始することも簡単になるでしょうが、今は、cloneを呼び出すのは完璧に受け入れられることです。

mainを更新したので、parse_configから返された Configのインスタンスを configという変数に置くようになり、以前は個別の queryと filename変数を使用していたコードを更新したので、代わりに Config構造体のフィールドを使用するようになりました。

これでコードは queryと filenameが関連していることと、その目的がプログラムの振る舞い方を設定するということをより明確に伝えます。これらの値を使用するあらゆるコードは、configインスタンスの目的の名前を冠したフィールドにそれらを発見することを把握しています。

14.3.2.1　Config のコンストラクタを作成する

ここまで、コマンドライン引数を解析する責任を負ったロジックを mainから抽出し、parse_config関数に配置しました。そうすることで queryと filenameの値が関連し、その関係性がコードに載っていることを確認する助けになりました。それから Config構造体を追加して queryと filenameの関係する目的を名前付けし、構造体のフィールド名として parse_config関数からその値の名前を返すことができています。

したがって、今や parse_config関数の目的は Configインスタンスを生成することになったので、parse_configをただの関数から Config構造体に紐付く newという関数に変えることができます。この変更を行うことで、コードがより慣用的になります。Stringなどの標準ライブラリの型のインスタンスを、String::newを呼び出すことで生成できます。同様に、parse_configを Configに紐付く new関数に変えれば、Config::newを呼び出すことで Configのインスタンスを生成できるようになります。リスト 12-7 が、行う必要のある変更を示しています:

ファイル名: src/main.rs

```
# use std::env;
#
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);

    // --snip--
}

# struct Config {
#     query: String,
#     filename: String,
# }
#
// --snip--
```

```
17
18 impl Config {
19     fn new(args: &[String]) -> Config {
20         let query = args[1].clone();
21         let filename = args[2].clone();
22
23         Config { query, filename }
24     }
25 }
```

リスト 12-7: parse_configを Config::newに変える

parse_configを呼び出していた mainを代わりに Config::newを呼び出すように更新しました。parse_configの名前を newに変え、implブロックに入れ込んだので、new関数と Configが紐付くようになりました。再度このコードをコンパイルしてみて、動作することを確かめてください。

14.3.3　エラー処理を修正する

さて、エラー処理の修正に取り掛かりましょう。ベクタが 2 個以下の要素しか含んでいないときにargsベクタの添字 1か 2にアクセスしようとすると、プログラムがパニックすることを思い出してください。試しに引数なしでプログラムを実行してください。すると、こんな感じになります:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', src/main.rs:29:21
(スレッド'main'は、「境界外アクセス: 長さは1なのに添字も1です」でパニックしました)
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

境界外アクセス: 長さは1なのに添字も1ですという行は、プログラマ向けのエラーメッセージです。エンドユーザが起きたことと代わりにすべきことを理解する手助けにはならないでしょう。これを今修正しましょう。

14.3.3.1　エラーメッセージを改善する

リスト 12-8 で、new関数に添字 1と 2にアクセスする前にスライスが十分長いことを実証するチェックを追加しています。スライスの長さが十分でなければ、プログラムはパニックし、境界外アクセスよりもいいエラーメッセージを表示します。

ファイル名: src/main.rs

```
1 // --snip--
2 fn new(args: &[String]) -> Config {
3     if args.len() < 3 {
4         // 引数の数が足りません
5         panic!("not enough arguments");
6     }
```

```
7       // --snip--
```

リスト 12-8: 引数の数のチェックを追加する

このコードは、リスト 9-9 で記述した value引数が正常な値の範囲外だった時に panic!を呼び出した Guess::new関数と似ています。ここでは、値の範囲を確かめる代わりに、argsの長さが少なくとも3であることを確かめていて、関数の残りの部分は、この条件が満たされているという仮定のもとで処理を行うことができます。argsに 2 要素以下しかなければ、この条件は真になり、panic!マクロを呼び出して、即座にプログラムを終了させます。

では、newのこのおまけの数行がある状態で、再度引数なしでプログラムを走らせ、エラーがどんな見た目か確かめましょう:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:30:12
(スレッド'main'は「引数が足りません」でパニックしました)
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

この出力の方がマシです: これでエラーメッセージが合理的になりました。ですが、ユーザに与えたくない追加の情報も含まれてしまっています。おそらく、ここではリスト 9-9 で使用したテクニックを使用するのは最善ではありません: panic!の呼び出しは、第 9 章で議論したように、使用の問題よりもプログラミング上の問題により適しています。代わりに、第 9 章で学んだもう一つのテクニックを使用することができます。成功か失敗かを示唆する Resultを返すことです。

14.3.3.2　panic!を呼び出す代わりに new から Result を返す

代わりに、成功時には Configインスタンスを含み、エラー時には問題に言及する Result値を返すことができます。Config::newが mainと対話する時、Result型を使用して問題があったとシグナルを送ることができます。それから mainを変更して、panic!呼び出しが引き起こしていた thread 'main' と RUST_BACKTRACEに関する周囲のテキストがない、ユーザ向けのより実用的なエラーに Errバリアントを変換することができます。

リスト 12-9 は、Config::newの戻り値に必要な変更と Resultを返すのに必要な関数の本体を示しています。mainも更新するまで、これはコンパイルできないことに注意してください。その更新は次のリストで行います。

ファイル名: src/main.rs

```
1  impl Config {
2      fn new(args: &[String]) -> Result<Config, &'static str> {
3          if args.len() < 3 {
4              return Err("not enough arguments");
5          }
6
7          let query = args[1].clone();
```

```
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}
```

リスト 12-9: Config::newから Resultを返却する

new関数は、今や、成功時には Configインスタンスを、エラー時には&'static strを伴う Resultを返すようになりました。第 10 章の「静的ライフタイム」節から&'static strは文字列リテラルの型であることを思い出してください。これは、今はエラーメッセージの型になっています。

new関数の本体で 2 つ変更を行いました: 十分な数の引数をユーザが渡さなかった場合に panic!を呼び出す代わりに、今は Err値を返し、Config戻り値を Okに包んでいます。これらの変更により、関数が新しい型シグニチャに適合するわけです。

Config::newから Err値を返すことにより、main関数は、new関数から返ってくる Result値を処理し、エラー時により綺麗にプロセスから抜け出すことができます。

14.3.3.3　Config::new を呼び出し、エラーを処理する

エラーケースを処理し、ユーザフレンドリーなメッセージを出力するために、mainを更新して、リスト 12-10 に示したように Config::newから返されている Resultを処理する必要があります。また、panic!からコマンドラインツールを 0 以外のエラーコードで抜け出す責任も奪い取り、手作業でそれも実装します。0 以外の終了コードは、我々のプログラムを呼び出したプロセスにプログラムがエラー状態で終了したことを通知する慣習です。

ファイル名: src/main.rs

```
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        // 引数解析時に問題
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
```

リスト 12-10: 新しい Config作成に失敗したら、エラーコードで終了する

このリストにおいて、以前には講義していないメソッドを使用しました: unwrap_or_elseです。これは標準ライブラリで Result<T, E>に定義されています。unwrap_or_elseを使うことで、panic!ではない何らか独自のエラー処理を定義できるのです。この Resultが Ok値だったら、このメソッドの振る舞いは unwrapに似ています: Okが包んでいる中身の値を返すのです。しかし、値が Err値なら、このメソッドは、**クロージャ**内でコードを呼び出し、クロージャは定義し、引数として unwrap_or_elseに

渡す匿名関数です。クロージャについては第 13 章で詳しく解説します。とりあえず、unwrap_or_elseは、今回リスト 12-9 で追加した not enough argumentsという静的文字列の Errの中身を、縦棒の間に出現する err引数のクロージャに渡していることだけ知っておく必要があります。クロージャのコードはそれから、実行された時に err値を使用できます。

新規 use行を追加して標準ライブラリから processをインポートしました。クロージャ内のエラー時に走るコードは、たった 2 行です: errの値を出力し、それから process::exitを呼び出します。process::exit関数は、即座にプログラムを停止させ、渡された数字を終了コードとして返します。これは、リスト 12-8 で使用した panic!ベースの処理と似ていますが、もう余計な出力はされません。試しましょう:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs
     Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

素晴らしい！ この出力の方が遥かにユーザに優しいです。

14.3.4　main からロジックを抽出する

ようやく設定解析のリファクタリングが終了したので、プログラムのロジックに目を向けましょう。「バイナリプロジェクトの責任の分離」で述べたように、現在 main関数に存在する設定セットやエラー処理に関わらない全てのロジックを保持することになる runという関数を抽出します。やり終わったら、mainは簡潔かつ視察で確かめやすくなり、他のロジック全部に対してテストを書くことができるでしょう。

リスト 12-11 は、抜き出した run関数を示しています。今は少しずつ段階的に関数を抽出する改善を行っています。それでも、*src/main.rs* に関数を定義していきます。

ファイル名: src/main.rs

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    run(config);
}

fn run(config: Config) {
    let mut f = File::open(config.filename).expect("file not found");

    let mut contents = String::new();
    f.read_to_string(&mut contents)
        .expect("something went wrong reading the file");

    println!("With text:\n{}", contents);
```

```
}

// --snip--
```

リスト 12-11: 残りのプログラムロジックを含む run関数を抽出する

これで run関数は、ファイル読み込みから始まる main関数の残りのロジック全てを含むようになりました。この run関数は、引数に Configインスタンスを取ります。

14.3.4.1　run 関数からエラーを返す

残りのプログラムロジックが run関数に隔離されたので、リスト 12-9 の Config::newのように、エラー処理を改善することができます。expectを呼び出してプログラムにパニックさせる代わりに、run関数は、何か問題が起きた時に Result<T, E>を返します。これにより、さらにエラー処理周りのロジックをユーザに優しい形で mainに統合することができます。リスト 12-12 にシグニチャと run本体に必要な変更を示しています:

ファイル名: src/main.rs

```
use std::error::Error;

// --snip--

fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    println!("With text:\n{}", contents);

    Ok(())
}
```

リスト 12-12: run関数を変更して Resultを返す

ここでは、3 つの大きな変更を行いました。まず、run関数の戻り値を Result<(), Box<Error>>に変えました。この関数は、前はユニット型、()を返していて、それを Okの場合に返される値として残しました。

エラー型については、**トレイトオブジェクト**の Box<Error>を使用しました (同時に冒頭で use文により、std::error::Errorをスコープに導入しています)。トレイトオブジェクトについては、第 17 章で講義します。とりあえず、Box<Error>は、関数が Errorトレイトを実装する型を返すことを意味しますが、戻り値の型を具体的に指定しなくても良いことを知っておいてください。これにより、エラーケースによって異なる型のエラー値を返す柔軟性を得ます。

2 番目に、expectの呼び出しよりも?を選択して取り除きました。第 9 章で語りましたね。エラーでパニックするのではなく、?は呼び出し元が処理できるように、現在の関数からエラー値を返します。

3 番目に、run関数は今、成功時に Ok値を返すようになりました。run関数の成功型は、シグニチャ

で ()と定義したので、ユニット型の値を Ok値に包む必要があります。最初は、この Ok(())という記法は奇妙に見えるかもしれませんが、このように ()を使うことは、runを副作用のためだけに呼び出していると示唆する慣習的な方法です; 必要な値は返しません。

このコードを実行すると、コンパイルは通るものの、警告が表示されるでしょう:

```
warning: unused `std::result::Result` which must be used
(警告: 使用されなければならない`std::result::Result`が未使用です)
  --> src/main.rs:18:5
   |
18 |     run(config);
   |     ^^^^^^^^^^^^
= note: #[warn(unused_must_use)] on by default
```

コンパイラは、コードが Result値を無視していると教えてくれて、この Result値は、エラーが発生したと示唆しているかもしれません。しかし、エラーがあったか確認するつもりはありませんが、コンパイラは、ここにエラー処理コードを書くつもりだったんじゃないかと思い出させてくれています！ 今、その問題を改修しましょう。

14.3.4.2　main で run から返ってきたエラーを処理する

リスト 12-10 の Config::newに対して行った方法に似たテクニックを使用してエラーを確認し、扱いますが、少し違いがあります:

ファイル名: src/main.rs

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);

        process::exit(1);
    }
}
```

unwrap_or_elseではなく、if letで runが Err値を返したかどうかを確認し、そうなら process::exit(1)を呼び出しています。run関数は、Config::newが Configインスタンスを返すのと同じようにunwrapしたい値を返すことはありません。runは成功時に ()を返すので、エラーを検知することにのみ興味があり、()でしかないので、unwrap_or_elseに包まれた値を返してもらう必要はないのです。

if letを unwrap_or_else関数の中身はどちらも同じです: エラーを出力して終了します。

14.3.5　コードをライブラリクレートに分割する

ここまで minigrepは良さそうですね！ では、テストを行い、*src/main.rs* ファイルの責任が減らせるように、*src/main.rs* ファイルを分割し、一部のコードを *src/lib.rs* ファイルに置きましょう。

main関数以外のコード全部を *src/main.rs* から *src/lib.rs* に移動しましょう:

- run関数定義
- 関係する use文
- Configの定義
- Config::new関数定義

src/lib.rs の中身にはリスト 12-13 に示したようなシグニチャがあるはずです (関数の本体は簡潔性のために省略しました)。この後に *src/main.rs* に変更を加えるまでこのコードはコンパイルできないことに注意してください。

ファイル名: src/lib.rs

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;

pub struct Config {
    pub query: String,
    pub filename: String,
}

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        // --snip--
    }
}

pub fn run(config: Config) -> Result<(), Box<Error>> {
    // --snip--
}
```

リスト 12-13: Configと runを *src/lib.rs* に移動する

ここでは、寛大に pubを使用しています: Configのフィールドと newメソッドと run関数です。これでテスト可能な公開 API のあるライブラリクレートができました！

さて、*src/lib.rs* に移動したコードを *src/main.rs* のバイナリクレートのスコープに持っていく必要があります。リスト 12-14 に示したようにね:

ファイル名: src/main.rs

```
extern crate minigrep;

use std::env;
```

```
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    if let Err(e) = minigrep::run(config) {
        // --snip--
    }
}
```

リスト 12-14: minigrepクレートを *src/main.rs* のスコープに持っていく

ライブラリクレートをバイナリクレートに持っていくのに、extern crate minigrepを使用しています。それから use minigrep::Config行を追加して Config型をスコープに持っていき、run関数にクレート名を接頭辞として付けます。これで全機能が連結され、動くはずです。cargo runでプログラムを走らせて、すべてがうまくいっていることを確かめてください。

ふう！ 作業量が多かったですね。ですが、将来成功する準備はできています。もう、エラー処理は遥かに楽になり、コードのモジュール化もできました。ここから先の作業は、ほぼ *src/lib.rs* で完結するでしょう。

古いコードでは大変だけれども、新しいコードでは楽なことをして新発見のモジュール性を活用しましょう: テストを書くのです！

14.4　テスト駆動開発でライブラリの機能を開発する

今や、ロジックを *src/lib.rs* に抜き出し、引数集めとエラー処理を *src/main.rs* に残したので、コードの核となる機能のテストを書くのが非常に容易になりました。いろんな引数で関数を直接呼び出し、コマンドラインからバイナリを呼び出す必要なく戻り値を確認できます。ご自由に Config::new や run関数の機能のテストは、自身でお書きください。

この節では、テスト駆動開発 (TDD) 過程を使用して minigrepプログラムに検索ロジックを追加します。このソフトウェア開発テクニックは、以下の手順に従います:

1. 失敗するテストを書き、走らせて想定通りの理由で失敗することを確かめる。
2. 十分な量のコードを書くか変更して新しいテストを通過するようにする。
3. 追加または変更したばかりのコードをリファクタリングし、テストが通り続けることを確認する。
4. 手順 1 から繰り返す！

この過程は、ソフトウェアを書く多くの方法のうちの一つに過ぎませんが、TDD によりコードデザインも駆動することができます。テストを通過させるコードを書く前にテストを書くことで、過程を通して高いテストカバー率を保つ助けになります。

実際にクエリ文字列の検索を行う機能の実装をテスト駆動し、クエリに合致する行のリストを生成

します。この機能を searchという関数に追加しましょう。

14.4.1　失敗するテストを記述する

もう必要ないので、プログラムの振る舞いを確認していた println!文を *src/lib.rs* と *src/main.rs* から削除しましょう。それから *src/lib.rs* で、テスト関数のある testモジュールを追加します。第 11 章のようにね。このテスト関数が search関数に欲しい振る舞いを指定します: クエリとそれを検索するテキストを受け取り、クエリを含む行だけをテキストから返します。リスト 12-15 にこのテストを示していますが、まだコンパイルは通りません:

ファイル名: src/lib.rs

```
# fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     vec![]
# }
#
#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        // Rustは
        // 安全で速く生産性も高い。
        // 3つ選んで。
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }
}
```

リスト 12-15: こうだったらいいなという search関数の失敗するテストを作成する

このテストは、"duct."という文字列を検索します。検索対象の文字列は 3 行で、うち 1 行だけが"duct."を含みます。search関数から返る値が想定している行だけを含むことをアサーションします。

このテストを走らせ、失敗するところを観察することはできません。このテストはコンパイルもできないからです: まだ search関数が存在していません！ ゆえに今度は、空のベクタを常に返す search関数の定義を追加することで、テストをコンパイルし走らせるだけのコードを追記します。リスト 12-16 に示したようにね。そうすれば、テストはコンパイルでき、失敗するはずです。なぜなら、空のベクタは、"safe, fast, productive."という行を含むベクタとは合致しないからです。

ファイル名: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

リスト 12-16: テストがコンパイルできるのに十分なだけ search関数を定義する

明示的なライフタイムの'aが searchのシグニチャで定義され、contents引数と戻り値で使用されていることに気付いてください。第 10 章からライフタイム仮引数は、どの実引数のライフタイムが戻り値のライフタイムに関連づけられているかを指定することを思い出してください。この場合、返却されるベクタは、(query引数ではなく)contents引数のスライスを参照する文字列スライスを含むべきと示唆しています。

言い換えると、コンパイラに search関数に返されるデータは、search関数に contents引数で渡されているデータと同期間生きることを教えています。これは重要なことです！ スライスに参照されるデータは、参照が有効になるために有効である必要があるのです; コンパイラが contentsではなく queryの文字列スライスを生成すると想定してしまったら、安全性チェックを間違って行うことになってしまいます。

ライフタイム注釈を忘れてこの関数をコンパイルしようとすると、こんなエラーが出ます:

```
error[E0106]: missing lifetime specifier
(エラー: ライフタイム指定子が欠けています)
 --> src/lib.rs:5:51
  |
5 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
  |                                                   ^ expected lifetime
parameter
  |
  = help: this function's return type contains a borrowed value, but the
  signature does not say whether it is borrowed from `query` or `contents`
  (助言: この関数の戻り値は、借用された値を含んでいますが、シグニチャにはそれが、
  `query`か`contents`から借用されたものであるかが示されていません)
```

コンパイラには、二つの引数のどちらが必要なのか知る由がないので、教えてあげる必要があるのです。contentsがテキストを全て含む引数で、合致するそのテキストの一部を返したいので、contentsがライフタイム記法で戻り値に関連づくはずの引数であることをプログラマは知っています。

他のプログラミング言語では、シグニチャで引数と戻り値を関連づける必要はありません。従って、これは奇妙に思えるかもしれませんが、時間とともに楽になっていきます。この例を第 10 章、「ライフタイムで参照を有効化する」節と比較したくなるかもしれません。

さあ、テストを実行しましょう:

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
--warnings--
    Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
     Running target/debug/deps/minigrep-abcabcabc

running 1 test
```

```
test test::one_result ... FAILED

failures:

---- test::one_result stdout ----
        thread 'test::one_result' panicked at 'assertion failed: `(left ==
right)`
left: `["safe, fast, productive."]`,
right: `[]`)', src/lib.rs:48:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    test::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed, to rerun pass '--lib'
```

素晴らしい。テストは全く想定通りに失敗しています。テストが通るようにしましょう！

14.4.2　テストを通過させるコードを書く

空のベクタを常に返しているために、現状テストは失敗しています。それを解消し、searchを実装するには、プログラムは以下の手順に従う必要があります:

- 中身を各行ごとに繰り返す。
- 行にクエリ文字列が含まれるか確認する。
- するなら、それを返却する値のリストに追加する。
- しないなら、何もしない。
- 一致する結果のリストを返す。

各行を繰り返す作業から、この手順に順に取り掛かりましょう。

14.4.2.1　linesメソッドで各行を繰り返す

Rust には、文字列を行ごとに繰り返す役立つメソッドがあり、利便性のために linesと名付けられ、リスト 12-17 のように動作します。まだこれはコンパイルできないことに注意してください:

ファイル名: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // 行に対して何かする
        // do something with line
    }
}
```

リスト 12-17: contentsの各行を繰り返す

linesメソッドはイテレータを返します。イテレータについて詳しくは、第 13 章で話しますが、リスト 3-5 でこのようなイテレータの使用法は見かけたことを思い出してください。そこでは、イテレータに forループを使用してコレクションの各要素に対して何らかのコードを走らせていました。

14.4.2.2　クエリを求めて各行を検索する

次に現在の行がクエリ文字列を含むか確認します。幸運なことに、文字列にはこれを行ってくれるcontainsという役に立つメソッドがあります！ search関数に、containsメソッドの呼び出しを追加してください。リスト 12-18 のようにね。これもまだコンパイルできないことに注意してください:

ファイル名: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

リスト 12-18: 行が queryの文字列を含むか確認する機能を追加する

14.4.2.3　合致した行を保存する

また、クエリ文字列を含む行を保存する方法が必要です。そのために、forループの前に可変なベクタを生成し、pushメソッドを呼び出して lineをベクタに保存することができます。forループの後でベクタを返却します。リスト 12-19 のようにね:

ファイル名: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

リスト 12-19: 合致する行を保存したので、返すことができる

これで search関数は、queryを含む行だけを返すはずであり、テストも通るはずです。テストを実行しましょう:

```
$ cargo test
--snip--
running 1 test
test test::one_result ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

テストが通り、動いていることがわかりました！

ここで、テストが通過するよう保ったまま、同じ機能を保持しながら、検索関数の実装をリファクタリングする機会を考えることもできます。検索関数のコードは悪すぎるわけではありませんが、イテレータの有効な機能の一部を活用していません。この例には第 13 章で再度触れ、そこでは、イテレータをより深く探求しますが、さらに改善する方法に目を向けます。

14.4.2.4　run 関数内で search 関数を使用する

search関数が動きテストできたので、run関数から searchを呼び出す必要があります。condif.queryの値と、ファイルから runが読み込む contentsの値を search関数に渡す必要があります。それから runは、searchから返ってきた各行を出力するでしょう:

ファイル名: src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    for line in search(&config.query, &contents) {
        println!("{}", line);
    }

    Ok(())
}
```

それでも forループで searchから各行を返し、出力しています。

さて、プログラム全体が動くはずです！ 試してみましょう。まずはエミリー・ディキンソンの詩から、ちょうど 1 行だけを返すはずの言葉から。”frog” です:

```
$ cargo run frog poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
     Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

かっこいい！ 今度は、複数行にマッチするであろう言葉を試しましょう。”body” とかね:

```
$ cargo run body poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

そして最後に、詩のどこにも現れない単語を探したときに、何も出力がないことを確かめましょう。”monomorphization” などね:

```
$ cargo run monomorphization poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep monomorphization poem.txt`
```

最高です！ 古典的なツールの独自のミニバージョンを構築し、アプリケーションを構造化する方法を多く学びました。また、ファイル入出力、ライフタイム、テスト、コマンドライン引数の解析についても、少し学びました。

このプロジェクトをまとめ上げるために、環境変数を扱う方法と標準エラー出力に出力する方法を少しだけデモします。これらはどちらも、コマンドラインプログラムを書く際に有用です。

14.5　環境変数を取り扱う

おまけの機能を追加して minigrepを改善します: 環境変数でユーザがオンにできる大文字小文字無視の検索用のオプションです。この機能をコマンドラインオプションにして、適用したい度にユーザが入力しなければならないようにすることもできますが、代わりに環境変数を使用します。そうすることでユーザは 1 回環境変数をセットすれば、そのターミナルセッションの間は、大文字小文字無視の検索を行うことができるようになるわけです。

14.5.1　大文字小文字を区別しない search 関数用に失敗するテストを書く

環境変数がオンの場合に呼び出す search_case_insensitive関数を新しく追加したいです。テスト駆動開発の過程に従い続けるので、最初の手順は、今回も失敗するテストを書くことです。新しい search``_case_insensitive関数用の新規テストを追加し、古いテストを one_resultから case_sensitiveに名前変更して、二つのテストの差異を明確化します。リスト 12-20 に示したようにね:

ファイル名: src/lib.rs

```
#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(
            vec!["safe, fast, productive."],
```

```
            search(query, contents)
        );
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}
```

リスト 12-20: 追加しようとしている大文字小文字を区別しない関数用の失敗するテストを新しく追加する

古いテストの contentsも変更していることに注意してください。大文字小文字を区別する検索を行う際に、”duct” というクエリに合致しないはずの大文字 D を使用した"Duct tape"という新しい行を追加しました。このように古いテストを変更することで、すでに実装済みの大文字小文字を区別する検索機能を誤って壊してしまわないことを保証する助けになります。このテストはもう通り、大文字小文字を区別しない検索に取り掛かっても通り続けるはずです。

大文字小文字を区別しない検索の新しいテストは、クエリに”rUsT” を使用しています。追加直前の search_case_insensitive関数では、”rUsT” というクエリは、両方ともクエリとは大文字小文字が異なるのに、大文字 R の”Rust:” を含む行と、“Trust me.”という行にもマッチするはずです。これが失敗するテストであり、まだ search_case_insensitive関数を定義していないので、コンパイルは失敗するでしょう。リスト 12-16 の search関数で行ったように空のベクタを常に返す実装の骨格を追加して、ご自由にテストがコンパイルされ、失敗する様を確認してください。

14.5.2　search_case_insensitive 関数を実装する

search_case_insensitive関数は、リスト 12-21 に示しましたが、search関数とほぼ同じです。唯一の違いは、queryと各 lineを小文字化していることなので、入力引数の大文字小文字によらず、行がクエリを含んでいるか確認する際には、同じになるわけです。

ファイル名: src/lib.rs

```
fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

```

```
    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

リスト 12-21: 比較する前にクエリと行を小文字化するよう、search_case_insensitive関数を定義する

まず、query文字列を小文字化し、同じ名前の上書きされた変数に保存します。ユーザのクエリが"rust"や"RUST"、"Rust"、"rUsT"などだったりしても、"rust"であり、大文字小文字を区別しないかのようにクエリを扱えるように、to_lowercaseをクエリに対して呼び出すことは必須です。

queryは最早、文字列スライスではなく Stringであることに注意してください。というのも、to_lowercaseを呼び出すと、既存のデータを参照するというよりも、新しいデータを作成するからです。例として、クエリは"rUsT"としましょう: その文字列スライスは、小文字の uや tを使えるように含んでいないので、"rust"を含む新しい Stringのメモリを確保しなければならないのです。今、containsメソッドに引数として queryを渡すと、アンド記号を追加する必要があります。containsのシグニチャは、文字列スライスを取るよう定義されているからです。

次に、各 lineが queryを含むか確かめる前に to_lowercaseの呼び出しを追加し、全文字を小文字化しています。今や lineと queryを小文字に変換したので、クエリが大文字であろうと小文字であろうとマッチを検索するでしょう。

この実装がテストを通過するか確認しましょう:

```
running 2 tests
test test::case_insensitive ... ok
test test::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

素晴らしい！ 通りました。では、run関数から新しい search_case_insensitive関数を呼び出しましょう。1 番目に大文字小文字の区別を切り替えられるよう、Config構造体に設定オプションを追加します。まだどこでも、このフィールドの初期化をしていないので、追加するとコンパイラエラーが起きます:

ファイル名: src/lib.rs

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool,
}
```

論理値を持つ case_sensitiveフィールドを追加したことに注意してください。次に、run関数に、case_sensitiveフィールドの値を確認し、search関数か search_case_insensitive関数を呼ぶかを決定するのに使ってもらう必要があります。リスト 12-22 のようにね。それでも、これはまだコンパイルできないことに注意してください:

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs::File;
# use std::io::prelude::*;
#
# fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     vec![]
# }
#
# fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     vec![]
# }
#
# struct Config {
#     query: String,
#     filename: String,
#     case_sensitive: bool,
# }
#
pub fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```

リスト 12-22: config.case_sensitiveの値に基づいて searchか search_case_insensitiveを呼び出す

最後に、環境変数を確認する必要があります。環境変数を扱う関数は、標準ライブラリの envモジュールにあるので、use std::env;行で *src/lib.rs* の冒頭でそのモジュールをスコープに持ってくる必要があります。そして、envモジュールから varメソッドを使用して CASE_INSENSITIVEという環境変数のチェックを行います。リスト 12-23 のようにね:

ファイル名: src/lib.rs

```
use std::env;
# struct Config {
#     query: String,
#     filename: String,
#     case_sensitive: bool,
# }

// --snip--

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

リスト 12-23: CASE_INSENSITIVEという環境変数のチェックを行う

ここで、case_sensitiveという新しい変数を生成しています。その値をセットするために、env::var関数を呼び出し、CASE_INSENSITIVE環境変数の名前を渡しています。env::varメソッドは、環境変数がセットされていたら、環境変数の値を含む Okバリアントの成功値になる Resultを返します。環境変数がセットされていなければ、Errバリアントを返すでしょう。

Resultの is_errメソッドを使用して、エラーでありゆえに、セットされていないことを確認しています。これは大文字小文字を区別する検索をすべきことを意味します。CASE_INSENSITIVE環境変数が何かにセットされていれば、is_errは false を返し、プログラムは大文字小文字を区別しない検索を実行するでしょう。環境変数の値はどうでもよく、セットされているかどうかだけ気にするので、unwrapや expectあるいは、他のここまで見かけた Resultのメソッドではなく、is_errをチェックしています。

case_sensitive変数の値を Configインスタンスに渡しているので、リスト 12-22 で実装したように、run関数はその値を読み取り、searchか search_case_insensitiveを呼び出すか決定できるのです。

試行してみましょう！ まず、環境変数をセットせずにクエリは、toでプログラムを実行し、この時は全て小文字で”to” という言葉を含むあらゆる行が合致するはずです。

```
$ cargo run to poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep to poem.txt`
```

```
Are you nobody, too?
How dreary to be somebody!
```

まだ機能しているようです！ では、CASE_INSENSITIVEを 1 にしつつ、同じクエリの toでプログラムを実行しましょう。

PowerShell を使用しているなら、1 コマンドではなく、2 コマンドで環境変数をセットし、プログラムを実行する必要があるでしょう:

```
$ $env:CASE_INSENSITIVE=1
$ cargo run to poem.txt
```

大文字も含む可能性のある”to” を含有する行が得られるはずです:

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

素晴らしい、”To” を含む行も出てきましたね！ minigrepプログラムはこれで、環境変数によって制御できる大文字小文字を区別しない検索も行えるようになりました。もうコマンドライン引数か、環境変数を使ってオプションを管理する方法も知りましたね。

引数と環境変数で同じ設定を行うことができるプログラムもあります。そのような場合、プログラムはどちらが優先されるか決定します。自身の別の練習として、コマンドライン引数か、環境変数で大文字小文字の区別を制御できるようにしてみてください。片方は大文字小文字を区別するようにセットされ、もう片方は区別しないようにセットしてプログラムが実行された時に、コマンドライン引数と環境変数のどちらの優先度が高くなるかを決めてください。

std::envモジュールは、環境変数を扱うもっと多くの有用な機能を有しています: ドキュメントを確認して、何が利用可能か確かめてください。

14.6　標準出力ではなく標準エラーにエラーメッセージを書き込む

現時点では、すべての出力を println!関数を使用してターミナルに書き込んでいます。多くのターミナルは、2 種類の出力を提供します: 普通の情報用の*標準**出力 *(stdout)* とエラーメッセージ用の標準エラー出力*(stderr) です。この差異のおかげで、ユーザは、エラーメッセージを画面に表示しつつ、プログラムの成功した出力をファイルにリダイレクトすることを選択できます。

println!関数は、標準出力に出力する能力しかないので、標準エラーに出力するには他のものを使用しなければなりません。

14.6.1　エラーが書き込まれる場所を確認する

まず、minigrepに出力される中身が、代わりに標準エラーに書き込みたいいかなるエラーメッセージも含め、どのように標準出力に書き込まれているかを観察しましょう。意図的にエラーを起こしつつ、ファイルに標準出力ストリームをリダイレクトすることでそうします。標準エラーストリームはリダイレクトしないので、標準エラーに送られる内容は、すべて画面に表示され続けます。

コマンドラインプログラムは、エラーメッセージを標準出力に送信していると期待されているので、標準出力ストリームをファイルにリダイレクトしても、画面にエラーメッセージが見られます。我々のプログラムは、現状、いい振る舞いをしていません: 代わりにファイルにエラーメッセージ出力を保存するところを、目撃するところです！

この動作をデモする方法は、>と標準出力ストリームをリダイレクトする先のファイル名、*output.txt* でプログラムを走らせることによります。引数は何も渡しませんが、そうするとエラーが起きるはずです:

```
$ cargo run > output.txt
```

>記法により、標準出力の中身を画面の代わりに *output.txt* に書き込むようシェルは指示されます。画面に出力されると期待していたエラーメッセージは見られないので、ファイルに入っているということでしょう。以下が *output.txt* が含んでいる内容です:

```
Problem parsing arguments: not enough arguments
```

そうです。エラーメッセージは標準出力に出力されているのです。このようなエラーメッセージは標準エラーに出力され、成功した状態のデータのみがファイルに残ると遥かに有用です。それを変更します。

14.6.2　エラーを標準エラーに出力する

リスト 12-24 のコードを使用して、エラーメッセージの出力の仕方を変更します。この章の前で行ったリファクタリングのため、エラーメッセージを出力するコードはすべて 1 関数、mainにあります。標準ライブラリは、標準エラーストリームに出力する eprintln!マクロを提供しているので、println!を呼び出してエラーを出力していた 2 箇所を代わりに eprintln!を使うように変更しましょう。

ファイル名: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

```

```
    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {}", e);

        process::exit(1);
    }
}
```

リスト 12-24: eprintln!を使って標準出力ではなく、標準エラーにエラーメッセージを書き込む

println!を eprintln!に変えてから、再度同じようにプログラムを実行しましょう。引数なしかつ、標準出力を>でリダイレクトしてね:

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

これで、エラーは画面に見えつつ、*output.txt* は何も含まなくなり、これはコマンドラインプログラムに期待する動作です。

再度、エラーは起こさずに標準出力をファイルにリダイレクトする引数でプログラムを走らせましょう。以下のようにね:

```
$ cargo run to poem.txt > output.txt
```

ターミナルには出力は見られず、*output.txt* に結果が含まれます:

ファイル名: output.txt

```
Are you nobody, too?
How dreary to be somebody!
```

これは、今や成功した出力には標準出力を、エラー出力には標準エラーを適切に使用していることをデモしています。

14.7　まとめ

この章では、ここまでに学んできた主要な概念の一部を想起させ、Rust で入出力処理を行う方法を講義しました。コマンドライン引数、ファイル、環境変数、そしてエラー出力に eprintln!マクロを使用することで、もう、コマンドラインアプリケーションを書く準備ができています。以前の章の概念を使用することで、コードはうまく体系化され、適切なデータ構造に効率的にデータを保存し、エラーをうまく扱い、よくテストされるでしょう。

次は、関数型言語に影響された Rust 機能を一部探求します: クロージャとイテレータです。

第 15 章

関数型言語の機能: イテレータとクロージャ

Rust の設計は、多くの既存の言語やテクニックにインスピレーションを得ていて、その一つの大きな影響が**関数型プログラミング**です。関数型でのプログラミングには、しばしば、引数で渡すことで関数を値として使用したり、関数から関数を返したり、関数を後ほど使用するために変数に代入することなどが含まれます。

この章では、関数型プログラミングがどんなものであったり、なかったりするかという問題については議論しませんが、代わりに関数型とよく言及される多くの言語の機能に似た Rust の機能の一部について議論しましょう。

具体的には、以下を講義します:

- **クロージャ**、変数に保存できる関数に似た文法要素
- **イテレータ**、一連の要素を処理する方法
- これら 2 つの機能を使用して第 12 章の入出力プロジェクトを改善する方法
- これら 2 つの機能のパフォーマンス (ネタバレ: 思ったよりも速いです)

パターンマッチングや enum など、他の Rust の機能も関数型に影響されていますが、他の章で講義します。クロージャとイテレータをマスターすることは、慣用的で速い Rust コードを書く重要な部分なので、この章を丸ごと捧げます。

15.1 クロージャ: 環境をキャプチャできる匿名関数

Rust の**クロージャ**は、変数に保存したり、引数として他の関数に渡すことのできる匿名関数です。ある場所でクロージャを生成し、それから別の文脈でクロージャを呼び出して評価することができます。関数と異なり、呼び出されたスコープの値をクロージャは、キャプチャすることができます。こ

れらのクロージャの機能がコードの再利用や、動作のカスタマイズを行わせてくれる方法を模擬しましょう。

15.1.1　クロージャで動作の抽象化を生成する

クロージャを保存して後々使用できるようにするのが有効な場面の例に取り掛かりましょう。その過程で、クロージャの記法、型推論、トレイトについて語ります。

以下のような架空の場面を考えてください: カスタマイズされたエクササイズのトレーニングプランを生成するアプリを作る立ち上げにかかることになりました。バックエンドは Rust で記述され、トレーニングプランを生成するアルゴリズムは、アプリユーザの年齢や、BMI、運動の好み、最近のトレーニング、指定された強弱値などの多くの要因を考慮します。実際に使用されるアルゴリズムは、この例では重要ではありません; 重要なのは、この計算が数秒要することです。必要なときだけこのアルゴリズムを呼び出し、1 回だけ呼び出したいので、必要以上にユーザを待たせないことになります。

リスト 13-1 に示した `simulated_expensive_calculation`関数でこの仮定のアルゴリズムを呼び出すことをシミュレートし、この関数は `calculating slowly`と出力し、2 秒待ってから、渡した数値をなんでも返します:

ファイル名: src/main.rs

```
use std::thread;
use std::time::Duration;

fn simulated_expensive_calculation(intensity: u32) -> u32 {
    // ゆっくり計算します
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    intensity
}
```

リスト 13-1: 実行に約 2 秒かかる架空の計算の代役を務める関数

次は、この例で重要なトレーニングアプリの部分を含む `main`関数です。この関数は、ユーザがトレーニングプランを要求した時にアプリが呼び出すコードを表します。アプリのフロントエンドと相互作用する部分は、クロージャの使用と関係ないので、プログラムへの入力を表す値をハードコードし、その出力を出力します。

必要な入力は以下の通りです:

- ユーザの強弱値、これはユーザがトレーニングを要求して、低強度のトレーニングか、高強度のトレーニングがしたいかを示したときに指定されます。
- 乱数、これはトレーニングプランにバリエーションを起こします。

出力は、推奨されるトレーニングプランになります。リスト 13-2 は使用する `main`関数を示しています:

ファイル名: src/main.rs

```
fn main() {
    let simulated_user_specified_value = 10;
    let simulated_random_number = 7;

    generate_workout(
        simulated_user_specified_value,
        simulated_random_number
    );
}
# fn generate_workout(intensity: u32, random_number: u32) {}
```

リスト 13-2: ユーザ入力や乱数生成をシミュレートするハードコードされた値がある main関数

簡潔性のために、変数 simulated_user_specified_valueは 10 と、変数 simulated_random_number は 7 とハードコードしました; 実際のプログラムにおいては、強弱値はアプリのフロントエンドから取得し、乱数の生成には、randクレートを使用します。第 2 章の数当てゲームの例みたいにね。main関数は、シミュレートされた入力値とともに generate_workout関数を呼び出します。

今や文脈ができたので、アルゴリズムに取り掛かりましょう。リスト 13-3 の generate_workout関数は、この例で最も気にかかるアプリのビジネスロジックを含んでいます。この例での残りの変更は、この関数に対して行われるでしょう:

ファイル名: src/main.rs

```
# use std::thread;
# use std::time::Duration;
#
# fn simulated_expensive_calculation(num: u32) -> u32 {
#     println!("calculating slowly...");
#     thread::sleep(Duration::from_secs(2));
#     num
# }
#
fn generate_workout(intensity: u32, random_number: u32) {
    if intensity < 25 {
        // 今日は{}回腕立て伏せをしてください！
        println!(
            "Today, do {} pushups!",
            simulated_expensive_calculation(intensity)
        );
        // 次に、{}回腹筋をしてください！
        println!(
            "Next, do {} situps!",
            simulated_expensive_calculation(intensity)
        );
    } else {
        if random_number == 3 {
            // 今日は休憩してください！水分補給を忘れずに！
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            // 今日は、{}分間走ってください！
            println!(
```

```
                "Today, run for {} minutes!",
                simulated_expensive_calculation(intensity)
            );
        }
    }
}
```

リスト 13-3:　入力に基づいてトレーニングプランを出力するビジネスロジックと、simulated_expensive_calculation関数の呼び出し

リスト 13-3 のコードには、遅い計算を行う関数への呼び出しが複数あります。最初の ifブロックが、simulated_expensive_calculationを 2 回呼び出し、外側の else内の ifは全く呼び出さず、2 番目の elseケースの内側にあるコードは 1 回呼び出しています。

generate_workout関数の期待される振る舞いは、まずユーザが低強度のトレーニング (25 より小さい数値で表される) か、高強度のトレーニング (25 以上の数値) を欲しているか確認することです。

低強度のトレーニングプランは、シミュレーションしようとしている複雑なアルゴリズムに基づいて、多くの腕立て伏せや腹筋運動を推奨してきます。

ユーザが高強度のトレーニングを欲していれば、追加のロジックがあります: アプリが生成した乱数がたまたま 3 なら、アプリは休憩と水分補給を勧めます。そうでなければ、ユーザは複雑なアルゴリズムに基づいて数分間のランニングをします。

このコードは、ビジネスのほしいままに動くでしょうが、データサイエンスチームが、simulated_expensive_calculation関数を呼び出す方法に何らかの変更を加える必要があると決定したとしましょう。そのような変更が起きた時に更新を簡略化するため、simulated_expensive_calculation関数を 1 回だけ呼び出すように、このコードをリファクタリングしたいです。また、過程でその関数への呼び出しを増やすことなく無駄に 2 回、この関数を現時点で呼んでいるところを切り捨てたくもあります。要するに、結果が必要なければ関数を呼び出したくなく、それでも 1 回だけ呼び出したいのです。

15.1.1.1　関数でリファクタリング

多くの方法でトレーニングプログラムを再構築することもできます。1 番目に expensive_expensive_calculation関数への重複した呼び出しを変数に抽出しようとしましょう。リスト 13-4 に示したように:

ファイル名: src/main.rs

```
# use std::thread;
# use std::time::Duration;
#
# fn simulated_expensive_calculation(num: u32) -> u32 {
#     println!("calculating slowly...");
#     thread::sleep(Duration::from_secs(2));
#     num
# }
#
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_result =
```

```
        simulated_expensive_calculation(intensity);

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result
        );
        println!(
            "Next, do {} situps!",
            expensive_result
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result
            );
        }
    }
}
```

リスト 13-4: 複数の simulated_expensive_calculationの呼び出しを 1 箇所に抽出し、結果をexpensive_result変数に保存する

この変更により simulated_expensive_calculationの呼び出しが単一化され、最初の ifブロックが無駄に関数を 2 回呼んでいた問題を解消します。不幸なことに、これでは、あらゆる場合にこの関数を呼び出し、その結果を待つことになり、結果値を全く使用しない内側の ifブロックでもそうしてしまいます。

プログラムの 1 箇所でコードを定義したいですが、結果が本当に必要なところでだけコードを実行します。これは、クロージャのユースケースです！

15.1.1.2　クロージャでリファクタリングして、コードを保存する

ifブロックの前にいつも simulated_expensive_calculation関数を呼び出す代わりに、クロージャを定義し、関数呼び出しの結果を保存するのではなく、そのクロージャを変数に保存できます。リスト 13-5 のようにね。simulated_expensive_calculationの本体全体を実際に、ここで導入しているクロージャ内に移すことができます:

ファイル名: src/main.rs

```
# use std::thread;
# use std::time::Duration;
#
let expensive_closure = |num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

```
# expensive_closure(5);
```

リスト 13-5: クロージャを定義し、expensive_closure変数に保存する

クロージャ定義が=に続き、変数 expensive_closureに代入しています。クロージャを定義するには、1 組の縦棒から始め、その内部にクロージャの仮引数を指定します; この記法は、Smalltalk や Ruby のクロージャ定義と類似していることから、選択されました。このクロージャには、numという引数が 1 つあります: 2 つ以上引数があるなら、|param1, param2|のように、カンマで区切ります。

引数の後に、クロージャの本体を保持する波括弧を配置します (これはクロージャ本体が式一つなら省略可能です)。波括弧の後、クロージャのお尻には、セミコロンが必要で、let文を完成させます。クロージャ本体の最後の行から返る値 (num) が、呼び出された時にクロージャから返る値になります。その行がセミコロンで終わっていないからです; 関数の本体みたいですね。

この let文は、expensive_closureが、匿名関数を呼び出した**結果の値**ではなく、匿名関数の**定義**を含むことを意味することに注意してください。コードを定義して、1 箇所で呼び出し、そのコードを保存し、後々、それを呼び出したいためにクロージャを使用していることを思い出してください; 呼び出したいコードは、現在、expensive_closureに保存されています。

クロージャが定義されたので、ifブロックのコードを変更して、そのコードを実行するクロージャを呼び出し、結果値を得ることができます。クロージャは、関数のように呼び出せます: クロージャ定義を含む変数名を指定し、使用したい引数値を含むかっこを続けます。リスト 13-6 に示したようにね:

ファイル名: src/main.rs

```
# use std::thread;
# use std::time::Duration;
#
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_closure = |num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_closure(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
```

```
                "Today, run for {} minutes!",
                expensive_closure(intensity)
            );
        }
    }
}
```

リスト 13-6: 定義した expensive_closureを呼び出す

今では、重い計算はたった 1 箇所でのみ呼び出され、結果が必要なところでそのコードを実行するだけになりました。

ところが、リスト 13-3 の問題の一つを再浮上させてしまいました: それでも、最初の ifブロックでクロージャを 2 回呼んでいて、そうすると、重いコードを 2 回呼び出し、必要な分の 2 倍ユーザを待たせてしまいます。その ifブロックのみに属する変数を生成して、クロージャの呼び出し結果を保持するその ifブロックに固有の変数を生成することでこの問題を解消することもできますが、クロージャは他の解決法も用意してくれます。その解決策については、もう少し先で語りましょう。でもまずは、クロージャ定義に型注釈がない理由とクロージャに関わるトレイトについて話しましょう。

15.1.2　クロージャの型推論と注釈

クロージャでは、fn関数のように引数の型や戻り値の型を注釈する必要はありません。関数では、型注釈は必要です。ユーザに露出する明示的なインターフェイスの一部だからです。このインターフェイスを堅実に定義することは、関数が使用したり、返したりする値の型についてみんなが合意していることを保証するために重要なのです。しかし、クロージャはこのような露出するインターフェイスには使用されません: 変数に保存され、名前付けしたり、ライブラリの使用者に晒されることなく、使用されます。

クロージャは通常短く、あらゆる任意のシナリオではなく、狭い文脈でのみ関係します。このような限定された文脈内では、コンパイラは、多くの変数の型を推論できるのに似て、引数や戻り値の型を頼もしく推論することができます。

このような小さく、匿名の関数で型をプログラマに注釈させることは、煩わしくコンパイラがすでに利用可能な情報と、大筋で余分でしょう。

変数のように、厳格に必要な以上に冗長になることと引き換えに、明示性と明瞭性を向上させたいなら、型注釈を加えることができます; リスト 13-4 で定義したクロージャに型を注釈するなら、リスト 13-7 に示した定義のようになるでしょう:

ファイル名: src/main.rs

```
# use std::thread;
# use std::time::Duration;
#
let expensive_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
```

```
};
```

リスト 13-7: クロージャの引数と戻り値の省略可能な型注釈を追加する

クロージャと記法は、型注釈があると関数の記法に酷似して見えます。以下が、引数に 1 を加える関数の定義と、同じ振る舞いをするクロージャの定義の記法を縦に比べたものです。空白を追加して、関連のある部分を並べています。これにより、縦棒の使用と省略可能な記法の量を除いて、クロージャ記法が関数記法に似ているところを具体化しています。

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

1 行目が関数定義を示し、2 行目がフルに注釈したクロージャ定義を示しています。3 行目は、クロージャ定義から型注釈を取り除き、4 行目は、かっこを取り除いていて、これはクロージャの本体がただ 1 つの式からなるので、省略可能です。これらは全て、呼び出された時に同じ振る舞いになる有効な定義です。

クロージャ定義には、引数それぞれと戻り値に対して推論される具体的な型が一つあります。例えば、リスト 13-8 に引数として受け取った値を返すだけの短いクロージャの定義を示しました。このクロージャは、この例での目的以外には有用ではありません。この定義には、何も型注釈を加えていないことに注意してください: それから 1 回目に `String`を引数に、2 回目に `u32`を引数に使用してこのクロージャを 2 回呼び出そうとしたら、エラーになります。

ファイル名: src/main.rs

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

リスト 13-8: 2 つの異なる型で型が推論されるクロージャの呼び出しを試みる

コンパイラは、次のエラーを返します:

```
error[E0308]: mismatched types
 --> src/main.rs
  |
  | let n = example_closure(5);
  |                         ^ expected struct `std::string::String`, found
  integral variable
  |
  = note: expected type `std::string::String`
             found type `{integer}`
```

`String`値で `example_closure`を呼び出した最初の時点で、コンパイラは `x`とクロージャの戻り値の型を `String`と推論します。そして、その型が `example_closure`のクロージャに閉じ込められ、同じクロージャを異なる型で使用しようとすると、型エラーが出るのです。

15.1.3　ジェネリック引数と Fn トレイトを使用してクロージャを保存する

トレーニング生成アプリに戻りましょう。リスト 13-6 において、まだコードは必要以上の回数、重い計算のクロージャを呼んでいました。この問題を解決する一つの選択肢は、重いクロージャの結果を再利用できるように変数に保存し、クロージャを再度呼ぶ代わりに、結果が必要になる箇所それぞれでその変数を使用することです。しかしながら、この方法は同じコードを大量に繰り返す可能性があります。

運のいいことに、別の解決策もあります。クロージャやクロージャの呼び出し結果の値を保持する構造体を作れるのです。結果の値が必要な場合のみにその構造体はクロージャを実行し、その結果の値をキャッシュするので、残りのコードは、結果を保存し、再利用する責任を負わなくて済むのです。このパターンは、**メモ化** (memoization) または、**遅延実行** (lazy evaluation) として知っているかもしれません。

クロージャを保持する構造体を作成するために、クロージャの型を指定する必要があります。構造体定義は、各フィールドの型を把握しておく必要がありますからね。各クロージャインスタンスには、独自の匿名の型があります: つまり、2 つのクロージャが全く同じシグニチャでも、その型はそれでも違うものと考えられるということです。クロージャを使用する構造体、enum、関数引数を定義するには、第 10 章で議論したように、ジェネリクスとトレイト境界を使用します。

Fnトレイトは、標準ライブラリで用意されています。全てのクロージャは、そのトレイトのどれかを実装しています: Fn、FnMutまたは、FnOnceです。「クロージャで環境をキャプチャする」節で、これらのトレイト間の差異を議論します; この例では、Fnトレイトを使えます。

Fnトレイト境界への型を追加して、クロージャがこのトレイト境界を合致させるために持っていなければならない引数と戻り値の型を表します。今回の場合、クロージャは、引数の型が u32で、u32を返すので、指定するトレイト境界は、Fn(u32) => u32です。

リスト 13-9 は、クロージャとオプションの結果値を保持する Cacher構造体の定義を示しています:

ファイル名: src/main.rs

```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}
```

リスト 13-9: クロージャを calculationに、オプションの結果値を valueに保持する Cacher構造体を定義する

Cacher構造体には、ジェネリックな型 Tの calculationフィールドがあります。Tに関するトレイト境界は、Fnトレイトを使うことでクロージャであると指定しています。calculationフィールドに保存したいクロージャは全て、1 つの u32引数 (Fnの後の括弧内で指定されている) を取り、u32(->の後に指定されている) を返さなければなりません。

> 注釈: 関数も 3 つの Fnトレイト全部を実装します。したいことに環境から値をキャプチャすることが必要ないなら、Fnトレイトを実装する何かが必要になるクロージャではなく、関数を使用できます。

valueフィールドの型は、Option<u32>です。クロージャを実行する前に、valueは Noneになるでしょう。Cacherを使用するコードがクロージャの結果を求めてきたら、その時点で Cacherはクロージャを実行し、その結果を valueフィールドの Someバリアントに保存します。それから、コードが再度クロージャの結果を求めたら、クロージャを再実行するのではなく、Cacherは Someバリアントに保持された結果を返すでしょう。

たった今説明した valueフィールド周りのロジックは、リスト 13-10 で定義されています:

ファイル名: src/main.rs

```
# struct Cacher<T>
#     where T: Fn(u32) -> u32
# {
#     calculation: T,
#     value: Option<u32>,
# }
#
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}
```

リスト 13-10: Cacherのキャッシュ機構

呼び出し元のコードにこれらのフィールドの値を直接変えてもらうのではなく、Cacherに構造体のフィールドの値を管理してほしいので、これらのフィールドは非公開になっています。

Cacher::new関数は、ジェネリックな引数の Tを取り、これは、Cacher構造体と同じトレイト境界を持つと定義しました。それから calculationフィールドに指定されたクロージャと、valueフィールドに None値を保持する Cacherインスタンスを Cacher::newは返します。まだクロージャを実行してい

ないからですね。

呼び出し元のコードがクロージャの評価結果を必要としたら、クロージャを直接呼ぶ代わりに、valueメソッドを呼びます。このメソッドは、結果の値が self.valueの Someにすでにあるかどうか確認します; そうなら、クロージャを再度実行することなく Some内の値を返します。

self.valueが Noneなら、コードは self.calculationに保存されたクロージャを呼び出し、結果を将来使えるように self.valueに保存し、その値を返しもします。

リスト 13-11 は、リスト 13-6 の関数 generate_workoutでこの Cacher構造体を使用する方法を示しています:

ファイル名: src/main.rs

```
# use std::thread;
# use std::time::Duration;
#
# struct Cacher<T>
#     where T: Fn(u32) -> u32
# {
#     calculation: T,
#     value: Option<u32>,
# }
#
# impl<T> Cacher<T>
#     where T: Fn(u32) -> u32
# {
#     fn new(calculation: T) -> Cacher<T> {
#         Cacher {
#             calculation,
#             value: None,
#         }
#     }
#
#     fn value(&mut self, arg: u32) -> u32 {
#         match self.value {
#             Some(v) => v,
#             None => {
#                 let v = (self.calculation)(arg);
#                 self.value = Some(v);
#                 v
#             },
#         }
#     }
# }
#
fn generate_workout(intensity: u32, random_number: u32) {
    let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if intensity < 25 {
```

```
        println!(
            "Today, do {} pushups!",
            expensive_result.value(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_result.value(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result.value(intensity)
            );
        }
    }
}
```

リスト 13-11: generate_workout関数内で Cacherを使用し、キャッシュ機構を抽象化する

クロージャを変数に直接保存する代わりに、クロージャを保持する Cacherの新規インスタンスを保存しています。そして、結果が必要な場所それぞれで、その Cacherインスタンスに対して valueメソッドを呼び出しています。必要なだけ valueメソッドを呼び出したり、全く呼び出さないこともでき、重い計算は最大でも 1 回しか走りません。

リスト 13-2 の main関数とともにこのプログラムを走らせてみてください。simulated_user_specified_value と simulated_random_number変数の値を変えて、いろんな ifや elseブロックの場合全てで、calculating slowlyは 1 回だけ、必要な時にのみ出現することを実証してください。必要以上に重い計算を呼び出さないことを保証するのに必要なロジックの面倒を Cacherは見るので、generate_workoutはビジネスロジックに集中できるのです。

15.1.4　Cacher 実装の限界

値をキャッシュすることは、コードの他の部分でも異なるクロージャで行いたくなるかもしれない一般的に有用な振る舞いです。しかし、現在の Cacherの実装には、他の文脈で再利用することを困難にしてしまう問題が 2 つあります。

1 番目の問題は、Cacherインスタンスが、常に valueメソッドの引数 argに対して同じ値になると想定していることです。言い換えると、Cacherのこのテストは、失敗するでしょう:

```
#[test]
fn call_with_different_values() {
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    let v2 = c.value(2);

```

```
8       assert_eq!(v2, 2);
9   }
```

このテストは、渡された値を返すクロージャを伴う Cacherインスタンスを新しく生成しています。この Cacherインスタンスに対して 1 という arg値で呼び出し、それから 2 という arg値で呼び出し、2 という arg値の value呼び出しには 2 が返るはずと期待しています。

このテストをリスト 13-9 とリスト 13-10 の Cacher実装で動かすと、assert_eqからこんなメッセージが出て、このテストは失敗します:

```
thread 'call_with_different_values' panicked at 'assertion failed: `(left == right)`
  left: `1`,
 right: `2`', src/main.rs
```

問題は、初めて c.valueを 1 で呼び出した時に、Cacherインスタンスは self.valueに Some(1)を保存したことです。その後 valueメソッドに何を渡しても、常に 1 を返すわけです。

単独の値ではなく、ハッシュマップを保持するように Cacherを改変してみてください。ハッシュマップのキーは、渡される arg値になり、ハッシュマップの値は、そのキーでクロージャを呼び出した結果になるでしょう。self.valueが直接 Someか None値であることを調べる代わりに、value関数はハッシュマップの argを調べ、存在するならその値を返します。存在しないなら、Cacherはクロージャを呼び出し、arg値に紐づけてハッシュマップに結果の値を保存します。

現在の Cacher実装の 2 番目の問題は、引数の型に u32を一つ取り、u32を返すクロージャしか受け付けないことです。例えば、文字列スライスを取り、usizeを返すクロージャの結果をキャッシュしたくなるかもしれません。この問題を解決するには、Cacher機能の柔軟性を向上させるためによりジェネリックな引数を導入してみてください。

«««< HEAD

第 16 章

クロージャで環境をキャプチャする

Listing 13-12 has an example of a closure stored in the `equal_to_x` variable that uses the `x` variable from the closure's surrounding environment:

fork*master*master

トレーニング生成の例において、クロージャをインラインの匿名関数として使っただけでした。しかし、クロージャには、関数にはない追加の能力があります: 環境をキャプチャし、自分が定義されたスコープの変数にアクセスできるのです。

リスト 13-12 は、`equal_to_x`変数に保持されたクロージャを囲む環境から `x`変数を使用するクロージャの例です:

ファイル名: src/main.rs

```
fn main() {
    let x = 4;

    let equal_to_x = |z| z == x;

    let y = 4;

    assert!(equal_to_x(y));
}
```

リスト 13-12: 内包するスコープの変数を参照するクロージャの例

ここで、`x`は `equal_to_x`の引数でもないのに、`equal_to_x`が定義されているのと同じスコープで定義されている `x`変数を `equal_to_x`クロージャは使用できています。

同じことを関数では行うことができません; 以下の例で試したら、コードはコンパイルできません:

ファイル名: src/main.rs

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x }

    let y = 4;

    assert!(equal_to_x(y));
}
```

エラーが出ます:

```
error[E0434]: can't capture dynamic environment in a fn item; use the || { ...
} closure form instead
(エラー: fn要素では動的な環境をキャプチャできません; 代わりに|| { ... }のクロージャ形式
    を
使用してください)
 --> src/main.rs
  |
4 |     fn equal_to_x(z: i32) -> bool { z == x }
  |                                         ^
```

コンパイラは、この形式はクロージャでのみ動作することにも触れています！

クロージャが環境から値をキャプチャすると、メモリを使用してクロージャ本体で使用できるようにその値を保存します。このメモリ使用は、環境をキャプチャしないコードを実行するようなもっと一般的な場合には払いたくないオーバーヘッドです。関数は、絶対に環境をキャプチャすることが許可されていないので、関数を定義して使えば、このオーバーヘッドを招くことは絶対にありません。

クロージャは、3 つの方法で環境から値をキャプチャでき、この方法は関数が引数を取れる 3 つの方法に直に対応します: 所有権を奪う、可変で借用する、不変で借用するです。これらは、以下のように 3 つの `Fn`トレイトでコード化されています:

- `FnOnce`は内包されたスコープからキャプチャした変数を消費し、これがクロージャの**環境**として知られています。キャプチャした変数を消費するために、定義された際にクロージャはこれらの変数の所有権を奪い、自身にムーブするのです。名前のうち、`Once`の部分は、このクロージャは同じ変数の所有権を 2 回以上奪うことができないという事実を表しているので、1 回しか呼ぶことができないのです。
- `FnMut`は、可変で値を借用するので、環境を変更することができます。
- `Fn`は、環境から値を不変で借用します。

クロージャを生成する時、クロージャが環境を使用する方法に基づいて、コンパイラはどのトレイトを使用するか推論します。少なくとも 1 回は呼び出されるので、全てのクロージャは `FnOnce`を実装しています。キャプチャした変数をムーブしないクロージャは、`FnMut`も実装し、キャプチャした変数に可変でアクセスする必要のないクロージャは、`Fn`も実装しています。リスト 13-12 では、`equal_to_x`クロージャは `x`を不変で借用しています (ゆえに `equal_to_x`は `Fn`トレイトです)。クロー

ジャの本体は、xを読む必要しかないからです。

環境でクロージャが使用している値の所有権を奪うことをクロージャに強制したいなら、引数リストの前に moveキーワードを使用できます。このテクニックは、新しいスレッドにデータが所有されるように、クロージャを新しいスレッドに渡して、データをムーブする際に大概は有用です。

並行性について語る第 16 章で、moveクロージャの例はもっと多く出てきます。とりあえず、こちらが moveキーワードがクロージャ定義に追加され、整数の代わりにベクタを使用するリスト 13-12 からのコードです。整数はムーブではなく、コピーされてしまいますからね; このコードはまだコンパイルできないことに注意してください。

ファイル名: src/main.rs

```
fn main() {
    let x = vec![1, 2, 3];

    let equal_to_x = move |z| z == x;

    // ここでは、xを使用できません: {:?}
    println!("can't use x here: {:?}", x);

    let y = vec![1, 2, 3];

    assert!(equal_to_x(y));
}
```

以下のようなエラーを受けます:

```
error[E0382]: use of moved value: `x`
(エラー: ムーブされた値の使用: `x`)
 --> src/main.rs:6:40
  |
4 |     let equal_to_x = move |z| z == x;
  |                      -------- value moved (into closure) here
  |                               (値はここで(クロージャに)ムーブされた)
5 |
6 |     println!("can't use x here: {:?}", x);
  |                                        ^ value used here after move
  |                                          (ムーブ後、値はここで使用された)
  |
  = note: move occurs because `x` has type `std::vec::Vec<i32>`, which does not
  implement the `Copy` trait
  (注釈: `x`が`std::vec::Vec<i32>`という`Copy`トレイトを実装しない型のため、ムーブが起
      きました)
```

クロージャが定義された際に、クロージャに xの値はムーブされています。moveキーワードを追加したからです。そして、クロージャは xの所有権を持ち、mainは println!で xを使うことはもう叶わないのです。println!を取り除けば、この例は修正されます。

Fnトレイトのどれかを指定するほとんどの場合、Fnから始めると、コンパイラがクロージャ本体内で起こっていることにより、FnMutや FnOnceが必要な場合、教えてくれるでしょう。

環境をキャプチャできるクロージャが関数の引数として有用な場面を具体化するために、次のトピックに移りましょう: イテレータです。

16.1　一連の要素をイテレータで処理する

イテレータパターンにより、一連の要素に順番に何らかの作業を行うことができます。イテレータは、各要素を繰り返し、シーケンスが終わったことを決定するロジックの責任を負います。イテレータを使用すると、自身でそのロジックを再実装する必要がなくなるのです。

Rust において、イテレータは**怠惰**です。つまり、イテレータを使い込んで消費するメソッドを呼ぶまで何の効果もないということです。例えば、リスト 13-13 のコードは、Vecに定義された iterメソッドを呼ぶことで v1ベクタの要素に対するイテレータを生成しています。このコード単独では、何も有用なことはしません。

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();
```

リスト 13-13: イテレータを生成する

一旦イテレータを生成したら、いろんな手段で使用することができます。第 3 章のリスト 3-5 では、ここまで iterの呼び出しが何をするかごまかしてきましたが、forループでイテレータを使い、各要素に何かコードを実行しています。

リスト 13-14 の例は、イテレータの生成と forループでイテレータを使用することを区別しています。イテレータは、v1_iter変数に保存され、その時には繰り返しは起きていません。v1_iterのイテレータで、forループが呼び出された時に、イテレータの各要素がループの繰り返しで使用され、各値が出力されます。

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    // {}でした
    println!("Got: {}", val);
}
```

リスト 13-14: forループでイテレータを使用する

標準ライブラリにより提供されるイテレータが存在しない言語では、変数を添字 0 から始め、その変数でベクタを覗き見て値を得て、ベクタの総要素数に到達するまでループでその変数の値をインクリメントすることで、この同じ機能を書く可能性が高いでしょう。

イテレータはそのロジック全てを処理してくれるので、めちゃくちゃにしてしまう可能性のあるコードの繰り返しを減らしてくれます。イテレータにより、添字を使えるデータ構造、ベクタなどだけではなく、多くの異なるシーケンスに対して同じロジックを使う柔軟性も得られます。イテレータ

がそれをする方法を調査しましょう。

16.1.1　Iterator トレイトと next メソッド

全てのイテレータは、標準ライブラリで定義されている Iteratorというトレイトを実装しています。このトレイトの定義は、以下のようになっています:

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // デフォルト実装のあるメソッドは省略
    // methods with default implementations elided
}
```

この定義は、何か新しい記法を使用していることに気付いてください: type Itemと Self::Itemで、これらはこのトレイトとの**関連型** (associated type) を定義しています。関連型についての詳細は、第 19 章で語ります。とりあえず、知っておく必要があることは、このコードが Iteratorトレイトを実装するには、Item型も定義する必要があり、そしてこの Item型が nextメソッドの戻り値の型に使われていると述べていることです。換言すれば、Item型がイテレータから返ってくる型になるだろうということです。

Iteratorトレイトは、一つのメソッドを定義することを実装者に要求することだけします: nextメソッドで、これは 1 度に Someに包まれたイテレータの 1 要素を返し、繰り返しが終わったら、Noneを返します。

イテレータに対して直接 nextメソッドを呼び出すこともできます; リスト 13-15 は、ベクタから生成されたイテレータの nextを繰り返し呼び出した時にどんな値が返るかを模擬しています:

ファイル名: src/lib.rs

```
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

リスト 13-15: イテレータに対して nextメソッドを呼び出す

v1_iterを可変にする必要があったことに注目してください: イテレータの nextメソッドを呼び出すと、今シーケンスのどこにいるかを追いかけるためにイテレータが使用している内部の状態が変わります。つまり、このコードはイテレータを**消費**、または使い込むのです。nextの各呼び出しは、イ

テレータの要素を一つ、食います。forループを使用した時には、v1_iterを可変にする必要はありませんでした。というのも、ループがv1_iterの所有権を奪い、陰で可変にしていたからです。

また、nextの呼び出しで得られる値は、ベクタの値への不変な参照であることにも注目してください。iterメソッドは、不変参照へのイテレータを生成します。v1の所有権を奪い、所有された値を返すイテレータを生成したいなら、iterではなくinto_iterを呼び出すことができます。同様に、可変参照を繰り返したいなら、iterではなくiter_mutを呼び出せます。

16.1.2　イテレータを消費するメソッド

Iteratorトレイトには、標準ライブラリが提供してくれているデフォルト実装のある多くの異なるメソッドがあります; Iteratorトレイトの標準ライブラリの API ドキュメントを検索することで、これらのメソッドについて知ることができます。これらのメソッドの中には、定義内でnextメソッドを呼ぶものもあり、故にIteratorトレイトを実装する際には、nextメソッドを実装する必要があるのです。

nextを呼び出すメソッドは、**消費アダプタ** (consuming adaptors) と呼ばれます。呼び出しがイテレータの使い込みになるからです。一例は、sumメソッドで、これはイテレータの所有権を奪い、nextを繰り返し呼び出すことで要素を繰り返し、故にイテレータを消費するのです。繰り返しが進むごとに、各要素を一時的な合計に追加し、繰り返しが完了したら、その合計を返します。リスト 13-16 は、sumの使用を具体化したテストです:

ファイル名: src/lib.rs

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

リスト 13-16: sumメソッドを呼び出してイテレータの全要素の合計を得る

sumは呼び出し対象のイテレータの所有権を奪うので、sum呼び出し後にv1_iterを使用することはできません。

16.2　他のイテレータを生成するメソッド

Iteratorトレイトに定義された他のメソッドは、**イテレータアダプタ** (iterator adaptors) として知られていますが、イテレータを別の種類のイテレータに変えさせてくれます。イテレータアダプタを複数回呼ぶ呼び出しを連結して、複雑な動作を読みやすい形で行うことができます。ですが、全て

のイテレータは怠惰なので、消費アダプタメソッドのどれかを呼び出し、イテレータアダプタの呼び出しから結果を得なければなりません。

リスト 13-17 は、イテレータアダプタメソッドの mapの呼び出し例を示し、各要素に対して呼び出すクロージャを取り、新しいイテレータを生成します。ここのクロージャは、ベクタの各要素が 1 インクリメントされる新しいイテレータを作成します。ところが、このコードは警告を発します:

ファイル名: src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

リスト 13-17: イテレータアダプタの mapを呼び出して新規イテレータを作成する

出る警告は以下の通りです:

```
warning: unused `std::iter::Map` which must be used: iterator adaptors are lazy
and do nothing unless consumed
(警告: 使用されねばならない`std::iter::Map`が未使用です: イテレータアダプタは怠惰で、
消費されるまで何もしません)
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(unused_must_use)] on by default
```

リスト 13-17 のコードは何もしません; 指定したクロージャは、決して呼ばれないのです。警告が理由に触れています: イテレータアダプタは怠惰で、ここでイテレータを消費する必要があるのです。

これを解消し、イテレータを消費するには、collectメソッドを使用しますが、これは第 12 章のリスト 12-1 で env::argsとともに使用しました。このメソッドはイテレータを消費し、結果の値をコレクションデータ型に集結させます。

リスト 13-18 において、map呼び出しから返ってきたイテレータを繰り返した結果をベクタに集結させています。このベクタは、最終的に元のベクタの各要素に 1 を足したものが含まれます。

ファイル名: src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

リスト 13-18: mapメソッドを呼び出して新規イテレータを作成し、それから collectメソッドを呼び出してその新規イテレータを消費し、ベクタを生成する

mapはクロージャを取るので、各要素に対して行いたいどんな処理も指定することができます。これは、Iteratorトレイトが提供する繰り返し動作を再利用しつつ、クロージャにより一部の動作をカスタマイズできる好例になっています。

16.2.1　環境をキャプチャするクロージャを使用する

イテレータが出てきたので、filterイテレータアダプタを使って環境をキャプチャするクロージャの一般的な使用をデモすることができます。イテレータの filterメソッドは、イテレータの各要素を取り、論理値を返すクロージャを取ります。このクロージャが trueを返せば、filterが生成するイテレータにその値が含まれます。クロージャが falseを返したら、結果のイテレータにその値は含まれません。

リスト 13-19 では、環境から shoe_size変数をキャプチャするクロージャで filterを使って、Shoe構造体インスタンスのコレクションを繰り返しています。指定したサイズの靴だけを返すわけです。

ファイル名: src/lib.rs

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter()
        .filter(|s| s.size == shoe_size)
        .collect()
}

#[test]
fn filters_by_size() {
    let shoes = vec![
        Shoe { size: 10, style: String::from("sneaker") },
        Shoe { size: 13, style: String::from("sandal") },
        Shoe { size: 10, style: String::from("boot") },
    ];

    let in_my_size = shoes_in_my_size(shoes, 10);

    assert_eq!(
        in_my_size,
        vec![
            Shoe { size: 10, style: String::from("sneaker") },
            Shoe { size: 10, style: String::from("boot") },
        ]
    );
}
```

リスト 13-19: shoe_sizeをキャプチャするクロージャで fileterメソッドを使用する

shoes_in_my_size関数は、引数として靴のベクタとサイズの所有権を奪います。指定されたサイズの靴だけを含むベクタを返します。

shoes_in_my_sizeの本体で、into_iterを呼び出してベクタの所有権を奪うイテレータを作成して

います。そして、filterを呼び出してそのイテレータをクロージャが trueを返した要素だけを含む新しいイテレータに適合させます。

クロージャは、環境から shoe_size引数をキャプチャし、指定されたサイズの靴だけを保持しながら、その値を各靴のサイズと比較します。最後に、collectを呼び出すと、関数により返ってきたベクタに適合させたイテレータから返ってきた値が集まるのです。

shoes_in_my_sizeを呼び出した時に、指定した値と同じサイズの靴だけが得られることをテストは示しています。

16.2.2　Iterator トレイトで独自のイテレータを作成する

ベクタに対し、iter、into_iter、iter_mutを呼び出すことでイテレータを作成できることを示してきました。ハッシュマップなどの標準ライブラリの他のコレクション型からもイテレータを作成できます。Iteratorトレイトを自分で実装することで、したいことを何でもするイテレータを作成することもできます。前述の通り、定義を提供する必要のある唯一のメソッドは、nextメソッドなのです。一旦、そうしてしまえば、Iteratorトレイトが用意しているデフォルト実装のある他の全てのメソッドを使うことができるのです！

デモ用に、1 から 5 を絶対にカウントするだけのイテレータを作成しましょう。まず、値を保持する構造体を生成し、Iteratorトレイトを実装することでこの構造体をイテレータにし、その実装内の値を使用します。

リスト 13-20 は、Counter構造体と Counterのインスタンスを作る new関連関数の定義です:

ファイル名: src/lib.rs

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

リスト 13-20: Counter構造体と countに対して 0 という初期値で Counterのインスタンスを作る new関数を定義する

Counter構造体には、countというフィールドがあります。このフィールドは、1 から 5 までの繰り返しのどこにいるかを追いかける u32値を保持しています。Counterの実装にその値を管理してほしいので、countフィールドは非公開です。countフィールドは常に 0 という値から新規インスタンスを開始するという動作を new関数は強要します。

次に、nextメソッドの本体をこのイテレータが使用された際に起きてほしいことを指定するように定義して、Counter型に対して Iteratorトレイトを実装します。リスト 13-21 のようにね:

ファイル名: src/lib.rs

```
# struct Counter {
#     count: u32,
# }
#
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

リスト 13-21: Counter構造体に Iteratorトレイトを実装する

イテレータの Item関連型を u32に設定しました。つまり、イテレータは、u32の値を返します。ここでも、まだ関連型について心配しないでください。第 19 章で講義します。

イテレータに現在の状態に 1 を足してほしいので、まず 1 を返すように 0 に countを初期化しました。countの値が 5 以下なら、nextは Someに包まれた現在の値を返しますが、countが 6 以上なら、イテレータは Noneを返します。

16.2.2.1　Counter イテレータの next メソッドを使用する

一旦 Iteratorトレイトを実装し終わったら、イテレータの出来上がりです！ リスト 13-22 は、リスト 13-15 のベクタから生成したイテレータと全く同様に、直接 nextメソッドを呼び出すことで、Counter構造体のイテレータ機能を使用できることをデモするテストを示しています。

ファイル名: src/lib.rs

```
# struct Counter {
#     count: u32,
# }
#
# impl Iterator for Counter {
#     type Item = u32;
#
#     fn next(&mut self) -> Option<Self::Item> {
#         self.count += 1;
#
#         if self.count < 6 {
#             Some(self.count)
#         } else {
#             None
#         }
#     }
```

```
# }
#
#[test]
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

リスト 13-22: nextメソッド実装の機能をテストする

このテストは、counter変数に新しい Counterインスタンスを生成し、それからイテレータにほしい動作が実装し終わっていることを実証しながら、nextを繰り返し呼び出しています: 1 から 5 の値を返すことです。

16.2.2.2　他の Iterator トレイトメソッドを使用する

nextメソッドを定義して Iteratorトレイトを実装したので、今では、標準ライブラリで定義されているように、どんな Iteratorトレイトメソッドのデフォルト実装も使えるようになりました。全てnextメソッドの機能を使っているからです。

例えば、何らかの理由で、Counterインスタンスが生成する値を取り、最初の値を飛ばしてから、別の Counterインスタンスが生成する値と一組にし、各ペアを掛け算し、3 で割り切れる結果だけを残し、全結果の値を足し合わせたくなったら、リスト 13-23 のテストに示したように、そうすることができます:

ファイル名: src/lib.rs

```
# struct Counter {
#     count: u32,
# }
#
# impl Counter {
#     fn new() -> Counter {
#         Counter { count: 0 }
#     }
# }
#
# impl Iterator for Counter {
#     // このイテレータはu32を生成します
#     // Our iterator will produce u32s
#     type Item = u32;
#
#     fn next(&mut self) -> Option<Self::Item> {
#         // カウントをインクリメントする。故に0から始まる
#         // increment our count. This is why we started at zero.
```

```
#         self.count += 1;
#
#         // カウントが終わったかどうか確認する
#         // check to see if we've finished counting or not.
#         if self.count < 6 {
#             Some(self.count)
#         } else {
#             None
#         }
#     }
# }
#
#[test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Counter::new().skip(1))
                                 .map(|(a, b)| a * b)
                                 .filter(|x| x % 3 == 0)
                                 .sum();
    assert_eq!(18, sum);
}
```

リスト 13-23: Counterイテレータに対していろんな Iteratorトレイトのメソッドを使用する

zipは 4 組しか生成しないことに注意してください; 理論的な 5 番目の組の (5, None)は、入力イテレータのどちらかが Noneを返したら、zipは Noneを返却するため、決して生成されることはありません。

nextメソッドの動作方法を指定し、標準ライブラリが nextを呼び出す他のメソッドにデフォルト実装を提供しているので、これらのメソッド呼び出しは全てあり得ます。

16.3　入出力プロジェクトを改善する

このイテレータに関する新しい知識があれば、イテレータを使用してコードのいろんな場所をより明確で簡潔にすることで、第 12 章の入出力プロジェクトを改善することができます。イテレータが Config::new関数と search関数の実装を改善する方法に目を向けましょう。

16.3.1　イテレータを使用して clone を取り除く

リスト 12-6 において、スライスに添字アクセスして値をクローンすることで、Config構造体に値を所有させながら、String値のスライスを取り、Config構造体のインスタンスを作るコードを追記しました。リスト 13-24 では、リスト 12-23 のような Config::newの実装を再現しました。

ファイル名: src/lib.rs

```
impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }
```

```
6
7           let query = args[1].clone();
8           let filename = args[2].clone();
9
10          let case_sensitive = env::var("CASE_INSENSITIVE").is_err();
11
12          Ok(Config { query, filename, case_sensitive })
13      }
14  }
```

リスト 13-24: リスト 12-23 から Config::new関数の再現

その際、将来的に除去する予定なので、非効率的な clone呼び出しを憂慮するなと述べました。えっと、その時は今です！

引数 argsに String要素のスライスがあるためにここで cloneが必要だったのですが、new関数は argsを所有していません。Configインスタンスの所有権を返すためには、Configインスタンスがその値を所有できるように、Configの queryと filenameフィールドから値をクローンしなければなりませんでした。

イテレータについての新しい知識があれば、new関数をスライスを借用する代わりに、引数としてイテレータの所有権を奪うように変更することができます。スライスの長さを確認し、特定の場所に添字アクセスするコードの代わりにイテレータの機能を使います。これにより、イテレータは値にアクセスするので、Config::new関数がすることが明確化します。

ひとたび、Config::newがイテレータの所有権を奪い、借用する添字アクセス処理をやめたら、clone を呼び出して新しくメモリ確保するのではなく、イテレータからの String値を Configにムーブできます。

16.3.1.1　返却されるイテレータを直接使う

入出力プロジェクトの *src/main.rs* ファイルを開いてください。こんな見た目のはずです:

ファイル名: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
}
```

リスト 12-24 のような main関数の冒頭をリスト 13-25 のコードに変更します。これは、Config::new も更新するまでコンパイルできません。

ファイル名: src/main.rs

```
fn main() {
    let config = Config::new(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
}
```

リスト 13-25: env::argsの戻り値を Config::newに渡す

env::args関数は、イテレータを返します！ イテレータの値をベクタに集結させ、それからスライスを Config::newに渡すのではなく、今では env::argsから返ってくるイテレータの所有権を直接 Config::newに渡しています。

次に、Config::newの定義を更新する必要があります。入出力プロジェクトの *src/lib.rs* ファイルで、Config::newのシグニチャをリスト 13-26 のように変えましょう。関数本体を更新する必要があるので、それでもコンパイルはできません。

ファイル名: src/lib.rs

```
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {
        // --snip--
```

リスト 13-26: Config::newのシグニチャをイテレータを期待するように更新する

env::args関数の標準ライブラリドキュメントは、自身が返すイテレータの型は、std::env::Argsであると表示しています。Config::new関数のシグニチャを更新したので、引数 argsの型は、&[String]ではなく、std::env::Argsになりました。argsの所有権を奪い、繰り返しを行うことで argsを可変化する予定なので、args引数の仕様に mutキーワードを追記し、可変にできます。

16.3.1.2　添字の代わりに Iterator トレイトのメソッドを使用する

次に、Config::newの本体を修正しましょう。標準ライブラリのドキュメントは、std::env::Argsが Iteratorトレイトを実装していることにも言及しているので、それに対して nextメソッドを呼び出せることがわかります！ リスト 13-27 は、リスト 13-23 のコードを nextメソッドを使用するように更新したものです:

ファイル名: src/lib.rs

```
# fn main() {}
# use std::env;
#
# struct Config {
#     query: String,
#     filename: String,
#     case_sensitive: bool,
# }
#
```

```
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            // クエリ文字列を得られませんでした
            None => return Err("Didn't get a query string"),
        };

        let filename = match args.next() {
            Some(arg) => arg,
            // ファイル名を得られませんでした
            None => return Err("Didn't get a file name"),
        };

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

リスト 13-27: Config::newの本体をイテレータメソッドを使うように変更する

env::argsの戻り値の 1 番目の値は、プログラム名であることを思い出してください。それは無視し、次の値を取得したいので、まず nextを呼び出し、戻り値に対して何もしません。2 番目に、nextを呼び出して Configの queryフィールドに置きたい値を得ます。nextが Someを返したら、matchを使用してその値を抜き出します。Noneを返したら、十分な引数が与えられなかったということなので、Err値で早期リターンします。filename値に対しても同じことをします。

16.3.2　イテレータアダプタでコードをより明確にする

入出力プロジェクトの search関数でも、イテレータを活用することができ、その関数は、リスト 12-19 のように、ここリスト 13-28 に再現しました。

ファイル名: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

リスト 13-28: リスト 12-19 の search関数の実装

イテレータアダプタメソッドを使用して、このコードをもっと簡潔に書くことができます。そうすれば、可変な中間の `results`ベクタをなくすこともできます。関数型プログラミングスタイルは、可変な状態の量を最小化することを好み、コードを明瞭化します。可変な状態を除去すると、検索を同時並行に行うという将来的な改善をするのが、可能になる可能性があります。なぜなら、`results`ベクタへの同時アクセスを管理する必要がなくなるからです。リスト 13-29 は、この変更を示しています:

ファイル名: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents.lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

リスト 13-29: `search`関数の実装でイテレータアダプタのメソッドを使用する

`search`関数の目的は、`query`を含む `contents`の行全てを返すことであることを思い出してください。リスト 13-19 の `filter`例に酷似して、このコードは `filter`アダプタを使用して `line.contains(query)`が真を返す行だけを残すことができます。それから、合致した行を別のベクタに `collect`で集結させます。ずっと単純です！ ご自由に、同じ変更を行い、`search_case_insensitive`関数でもイテレータメソッドを使うようにしてください。

次の論理的な疑問は、自身のコードでどちらのスタイルを選ぶかと理由です: リスト 13-28 の元の実装とリスト 13-29 のイテレータを使用するバージョンです。多くの Rust プログラマは、イテレータスタイルを好みます。とっかかりが少し困難ですが、いろんなイテレータアダプタとそれがすることの感覚を一度掴めれば、イテレータの方が理解しやすいこともあるでしょう。いろんなループを少しずつもてあそんだり、新しいベクタを構築する代わりに、コードは、ループの高難度の目的に集中できるのです。これは、ありふれたコードの一部を抽象化するので、イテレータの各要素が通過しなければならないふるい条件など、このコードに独特の概念を理解しやすくなります。

ですが、本当に 2 つの実装は等価なのでしょうか？ 直観的な仮説は、より低レベルのループの方がより高速ということかもしれません。パフォーマンスに触れましょう。

16.4　パフォーマンス比較: ループ VS イテレータ

ループを使うべきかイテレータを使うべきか決定するために、`search`関数のうち、どちらのバージョンが速いか知る必要があります: 明示的な `for`ループがあるバージョンと、イテレータのバージョンです。

サー・アーサー・コナン・ドイル (Sir Arthur Conan Doyle) の、**シャーロックホームズの冒険** (The Adventures of Sherlock Homes) 全体を `String`に読み込み、そのコンテンツで *the* という単語を検索することでベンチマークを行いました。こちらが、`for`を使用した `search`関数のバージョンと、イテレータを使用したバージョンに関するベンチマーク結果です。

```
test bench_search_for  ... bench:  19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench:  19,234,900 ns/iter (+/- 657,200)
```

イテレータバージョンの方が些か高速ですね！ ここでは、ベンチマークのコードは説明しません。なぜなら、要点は、2 つのバージョンが等価であることを証明することではなく、これら 2 つの実装がパフォーマンス的にどう比較されるかを大まかに把握することだからです。

より理解しやすいベンチマークには、いろんなサイズの様々なテキストを `contents`として、異なる単語、異なる長さの単語を `query`として、他のあらゆる種類のバリエーションを確認するべきです。重要なのは: イテレータは、高度な抽象化にも関わらず、低レベルのコードを自身で書いているかのように、ほぼ同じコードにコンパイルされることです。イテレータは、Rust の**ゼロ代償抽象化**の一つであり、これは、抽象化を使うことが追加の実行時オーバーヘッドを生まないことを意味しています。このことは、C++ の元の設計者であり実装者のビャーネ・ストルヴストルップ (Bjarne Stroustrup) が、**ゼロオーバーヘッド**を「C++ の基礎 (2012)」で定義したのと類似しています。

> 一般的に、C++ の実装は、ゼロオーバーヘッド原則を遵守します: 使用しないものには、支払わなくてよい。さらに: 実際に使っているものに対して、コードをそれ以上うまく渡すことはできない。

別の例として、以下のコードは、オーディオデコーダから取ってきました。デコードアルゴリズムは、線形予測数学演算を使用して、以前のサンプルの線形関数に基づいて未来の値を予測します。このコードは、イテレータ連結をしてスコープにある 3 つの変数に計算を行っています: `buffer`というデータのスライス、12 の `coefficients`の配列、`qlp_shift`でデータをシフトする量です。この例の中で変数を宣言しましたが、値は与えていません; このコードは、文脈の外では大して意味を持ちませんが、それでも Rust が高レベルな考えを低レベルなコードに翻訳する簡潔で現実的な例になっています:

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
                                 .zip(&buffer[i - 12..i])
                                 .map(|(&c, &s)| c * s as i64)
                                 .sum::<i64>() >> qlp_shift;
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

`prediction`の値を算出するために、このコードは、`coefficients`の 12 の値を繰り返し、`zip`メソッドを使用して、係数値を前の `buffer`の 12 の値と組にします。それから各組について、その値をかけ合わせ、結果を全て合計し、合計のビットを `qlp_shift`ビット分だけ右にシフトさせます。

オーディオデコーダのようなアプリケーションの計算は、しばしばパフォーマンスに最も重きを置きます。ここでは、イテレータを作成し、2 つのアダプタを使用し、それから値を消費しています。こ

の Rust コードは、どんな機械語コードにコンパイルされるのでしょうか？ えー、執筆時点では、手作業で書いたものと同じ機械語にコンパイルされます。`coefficients`の値の繰り返しに対応するループは全く存在しません: コンパイラは、12 回繰り返しがあることを把握しているので、ループを「展開」します。**ループの展開**は、ループ制御コードのオーバーヘッドを除去し、代わりにループの繰り返しごとに同じコードを生成する最適化です。

係数は全てレジスタに保存されます。つまり、値に非常に高速にアクセスします。実行時に配列の境界チェックをすることもありません。コンパイラが適用可能なこれらの最適化全てにより、結果のコードは究極的に効率化されます。このことがわかったので、イテレータとクロージャを恐れなしに使用することができますね！ それらのおかげでコードは、高レベルだけれども、そうすることに対して実行時のパフォーマンスを犠牲にしないようになります。

16.5　まとめ

クロージャとイテレータは、関数型言語の考えに着想を得た Rust の機能です。低レベルのパフォーマンスで、高レベルの考えを明確に表現するという Rust の能力に貢献しています。クロージャとイテレータの実装は、実行時のパフォーマンスが影響されないようなものです。これは、ゼロ代償抽象化を提供するのに努力を惜しまない Rust の目標の一部です。

今や入出力プロジェクトの表現力を改善したので、プロジェクトを世界と共有するのに役に立つ`cargo`の機能にもっと目を向けましょう。

第 17 章

Cargo と Crates.io についてより詳しく

今まで Cargo のビルド、実行、コードのテストを行うという最も基礎的な機能のみを使ってきましたが、他にもできることはたくさんあります。この章では、そのような他のより高度な機能の一部を議論し、以下のことをする方法をお見せしましょう:

- リリースプロファイルでビルドをカスタマイズする
- crates.io でライブラリを公開する
- ワークスペースで巨大なプロジェクトを体系化する
- crates.io からバイナリをインストールする
- 独自のコマンドを使用して Cargo を拡張する

また、Cargo はこの章で講義する以上のこともできるので、機能の全解説を見るには、ドキュメンテーションを参照されたし。

17.1 リリースプロファイルでビルドをカスタマイズする

Rust において、**リリースプロファイル**とは、プログラマがコードのコンパイルオプションについて制御可能な異なる設定を持つ、定義済みのカスタマイズ可能なプロファイルです。各プロファイルは、それぞれ個別で設定されます。

Cargo には 2 つの主なプロファイルが存在します: `dev`プロファイルは、`cargo build`コマンドを実行したときに使用され、`release`プロファイルは、`cargo build --release`コマンドを実行したときに使用されます。`dev`プロファイルは、開発中に役に立つデフォルト設定がなされており、`release`プロファイルは、リリース用の設定がなされています。

これらのプロファイル名は、ビルドの出力で馴染みのあるものかもしれません:

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
$ cargo build --release
```

```
    Finished release [optimized] target(s) in 0.0 secs
```

このビルド出力で表示されている devと releaseは、コンパイラが異なるプロファイルを使用していることを示しています。

プロジェクトの *Cargo.toml* ファイルに [profile.*]セクションが存在しない際に適用される各プロファイル用のデフォルト設定が、Cargo には存在します。カスタマイズしたいプロファイル用の[profile.*]セクションを追加することで、デフォルト設定の一部を上書きすることができます。例えば、こちらが devと releaseプロファイルの opt-level設定のデフォルト値です:

ファイル名: Cargo.toml

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

opt-level設定は、0 から 3 の範囲でコンパイラがコードに適用する最適化の度合いを制御します。最適化を多くかけると、コンパイル時間が延びるので、開発中に頻繁にコードをコンパイルするのなら、たとえ出力結果のコードの動作速度が遅くなっても早くコンパイルが済んでほしいですよね。これが、devの opt-levelのデフォルト設定が 0になっている唯一の理由です。コードのリリース準備ができたら、より長い時間をコンパイルにかけるのが最善の策です。リリースモードでコンパイルするのはたった 1 回ですが、コンパイル結果のプログラムは何度も実行するので、リリースモードでは、長いコンパイル時間と引き換えに、生成したコードが速く動作します。これが releaseの opt-levelのデフォルト設定が 3になっている唯一の理由です。

デフォルト設定に対して Cargo.tomlで異なる値を追加すれば、上書きすることができます。例として、開発用プロファイルで最適化レベル 1 を使用したければ、以下の 2 行をプロジェクトの *Cargo.toml* ファイルに追加できます:

ファイル名: Cargo.toml

```
[profile.dev]
opt-level = 1
```

このコードは、デフォルト設定の 0を上書きします。こうすると、cargo buildを実行したときに、devプロファイル用のデフォルト設定に加えて、Cargo は opt-levelの変更を適用します。opt-level を 1に設定したので、Cargo はデフォルトよりは最適化を行いますが、リリースビルドほどではありません。

設定の選択肢と各プロファイルのデフォルト設定の一覧は、Cargo のドキュメンテーションを参照されたし。

17.2　Crates.io にクレートを公開する

プロジェクトの依存として crates.io のパッケージを使用しましたが、自分のパッケージを公開することで他の人とコードを共有することもできます。crates.io のクレート登録所は、自分のパッケージのソースコードを配布するので、主にオープンソースのコードをホストします。

Rust と Cargo は、公開したパッケージを人が使用し、まず見つけやすくしてくれる機能を有しています。これらの機能の一部を次に語り、そして、パッケージの公開方法を説明します。

17.2.1　役に立つドキュメンテーションコメントを行う

パッケージを正確にドキュメントすることで、他のユーザがパッケージを使用する方法や、いつ使用すべきかを理解する手助けをすることになるので、ドキュメンテーションを書くことに時間を費やす価値があります。第 3 章で、2 スラッシュ、//で Rust のコードにコメントをつける方法を議論しました。Rust には、ドキュメンテーション用のコメントも用意されていて、都合のいいことに**ドキュメンテーションコメント**として知られ、HTML ドキュメントを生成します。クレートの**実装**法とは対照的にクレートの**使用**法を知ることに興味のあるプログラマ向けの、公開 API 用のドキュメンテーションコメントの中身をこの HTML は表示します。

ドキュメンテーションコメントは、2 つではなく、3 スラッシュ、///を使用し、テキストを整形する Markdown 記法もサポートしています。ドキュメント対象の要素の直前にドキュメンテーションコメントを配置してください。リスト 14-1 は、my_crateという名のクレートの add_one関数用のドキュメンテーションコメントを示しています:

ファイル名: src/lib.rs

```
/// Adds one to the number given.
///
/// 与えられた数値に1を足す。
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, my_crate::add_one(5));
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

リスト 14-1: 関数のドキュメンテーションコメント

ここで、add_one関数がすることの説明を与え、Examplesというタイトルでセクションを開始し、add_one関数の使用法を模擬するコードを提供しています。このドキュメンテーションコメントから cargo docを実行することで、HTML ドキュメントを生成することができます。このコマンドは

コンパイラとともに配布されている rustdocツールを実行し、生成された HTML ドキュメントを *target/doc* ディレクトリに配置します。

利便性のために、cargo doc --openを走らせれば、現在のクレートのドキュメント用の HTML(と、自分のクレートが依存している全てのドキュメント) を構築し、その結果を Web ブラウザで開きます。add_one関数まで下り、図 14-1 に示したように、ドキュメンテーションコメントのテキストがどう描画されるかを確認しましょう:

図 14-1: add_one関数の HTML ドキュメント

17.2.1.1　よく使われるセクション

Examplesマークダウンのタイトルをリスト 14-1 で使用し、「例」というタイトルのセクションを HTML に生成しました。こちらがこれ以外にドキュメントでよくクレート筆者が使用するセクションです:

- **Panics**: ドキュメント対象の関数が panic!する可能性のある筋書きです。プログラムをパニックさせたくない関数の使用者は、これらの状況で関数が呼ばれないことを確かめる必要があります。
- **Errors**: 関数が Resultを返すなら、起きうるエラーの種類とどんな条件がそれらのエラーを引き起こす可能性があるのか解説すると、呼び出し側の役に立つので、エラーの種類によって処理するコードを変えて書くことができます。
- **Safety**: 関数が呼び出すのに unsafe(unsafe については第 19 章で議論します) なら、関数が unsafe な理由を説明し、関数が呼び出し元に保持していると期待する不変条件を講義するセクションがあるべきです。

多くのドキュメンテーションコメントでは、これら全てのセクションが必要になることはありませんが、これは自分のコードを呼び出している人が知りたいと思うコードの方向性を思い出させてくれるいいチェックリストになります。

17.2.1.2　テストとしてのドキュメンテーションコメント

ドキュメンテーションコメントに例のコードブロックを追加すると、ライブラリの使用方法のデモに役立ち、おまけもついてきます: cargo testを走らせると、ドキュメントのコード例をテストとして実行するのです！ 例付きのドキュメントに上回るものはありません。しかし、ドキュメントが書かれてからコードが変更されたがために、動かない例がついているよりも悪いものもありません。リスト 14-1 から add_one関数のドキュメンテーションとともに、cargo testを走らせたら、テスト結果に以下のような区域が見られます:

```
   Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

さて、例の assert_eq!がパニックするように、関数か例を変更し、再度 cargo testを実行したら、doc テストが、例とコードがお互いに同期されていないことを捕捉するところを目撃するでしょう！

17.2.1.3　含まれている要素にコメントする

doc コメントの別スタイル、//!は、コメントに続く要素にドキュメンテーションを付け加えるのではなく、コマンドを含む要素にドキュメンテーションを付け加えます。典型的には、クレートのルートファイル (規定では、*src/lib.rs*) 内部や、モジュールの内部で使用して、クレートやモジュール全体にドキュメントをつけます。

例えば、add_one関数を含む my_crateクレートの目的を解説するドキュメンテーションを追加したいのなら、//!で始まるドキュメンテーションコメントを *src/lib.rs* ファイルの先頭につけることができます。リスト 14-2 に示したようにね:

ファイル名: src/lib.rs

```
1  //! # My Crate
2  //!
3  //! `my_crate` is a collection of utilities to make performing certain
4  //! calculations more convenient.
5
6  //! #自分のクレート
7  //!
8  //! `my_crate`は、ユーティリティの集まりであり、特定の計算をより便利に行うことができます。
9
10 /// Adds one to the number given.
11 // --snip--
```

リスト 14-2: 全体として my_crateクレートにドキュメントをつける

//!で始まる最後の行以降には、コードが何もないことに気付いてください。///ではなく、//!でコメントを開始しているので、このコメントに続く要素ではなく、このコメントを含む要素にドキュメントをつけているわけです。今回の場合、このコメントを含む要素は *src/lib.rs* ファイルであり、クレートのルートです。これらのコメントは、クレート全体を解説しています。

cargo doc --openを実行すると、これらのコメントは、my_crateのドキュメントの最初のページ、クレートの公開要素のリストの上部に表示されます。図 14-2 のようにね:

図 14-2: クレート全体を解説するコメントを含む my_crateの描画されたドキュメンテーション

要素内のドキュメンテーションコメントは、特にクレートやモジュールを解説するのに有用です。コンテナの全体の目的を説明し、クレートの使用者がクレートの体系を理解する手助けをするのに使用してください。

17.2.2　pub use で便利な公開 API をエクスポートする

第 7 章において、modキーワードを使用してモジュールにコードを体系化する方法、pubキーワードで要素を公開にする方法、useキーワードで要素をスコープに導入する方法について講義しました。しかしながら、クレートの開発中に、自分にとって意味のある構造は、ユーザにはあまり便利ではない可能性があります。複数階層を含むヒエラルキーで、自分の構造体を体系化したくなるかもしれませんが、それからヒエラルキーの深いところで定義した型を使用したい人は、型が存在することを見つけ出すのに困難を伴う可能性もあります。また、そのような人は、use my_crate::UsefulTypeの代わりに use my_crate::some_module::another_module::UsefulType;と入力するのを煩わしく感じる可能性もあります。

自分の公開 API の構造は、クレートを公開する際に考慮すべき点です。自分のクレートを使用したい人は、自分よりもその構造に馴染みがないですし、クレートのモジュール階層が大きければ、使用したい部分を見つけるのが困難になる可能性があります。

嬉しいお知らせは、構造が他人が他のライブラリから使用するのに便利ではない場合、内部的な体系を再構築する必要はないということです: 代わりに、要素を再エクスポートし、pub useで自分の非公開構造とは異なる公開構造にできます。再エクスポートは、ある場所の公開要素を一つ取り、別の場所で定義されているかのように別の場所で公開します。

例えば、芸術的な概念をモデル化するために artという名のライブラリを作ったとしましょう。このライブラリ内には、2 つのモジュールがあります: PrimaryColorと SecondaryColorという名前の 2 つの eunm を含む、kindsモジュールと mixという関数を含む utilsモジュールです。リスト 14-3 のようにね:

ファイル名: src/lib.rs

```
//! # Art
//!
//! A library for modeling artistic concepts.
//! #芸術
//!
//! 芸術的な概念をモデル化するライブラリ。

pub mod kinds {
    /// The primary colors according to the RYB color model.
    /// RYBカラーモデルによる主色
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    /// RYBカラーモデルによる副色
    pub enum SecondaryColor {
        Orange,
```

```
        Green,
        Purple,
    }
}

pub mod utils {
    use kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    ///2つの主色を同じ割合で混合し、副色にする
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
    }
}
```

リスト 14-3: kindsと utilsモジュールに体系化される要素を含む artライブラリ

図 14-3 は、cargo docにより生成されるこのクレートのドキュメンテーションの最初のページがどんな見た目になるか示しています:

図 14-3: kindsと utilsモジュールを列挙する artのドキュメンテーションのトップページ

PrimaryColorも SecondaryColor型も、mix関数もトップページには列挙されていないことに注意してください。kindsと utilsをクリックしなければ、参照することができません。

このライブラリに依存する別のクレートは、現在定義されているモジュール構造を指定して、artの要素をインポートする use文が必要になるでしょう。リスト 14-4 は、artクレートから PrimaryColorと mix要素を使用するクレートの例を示しています:

ファイル名: src/main.rs

```
extern crate art;

use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

リスト 14-4: 内部構造もエクスポートされて artクレートの要素を使用するクレート

リスト 14-4 は artクレートを使用していますが、このコードの筆者は、PrimaryColorが kindsモジュールにあり、mixが utilsモジュールにあることを割り出さなければなりませんでした。artクレートのモジュール構造は、artクレートの使用者よりも、artクレートに取り組む開発者などに関係が深いです。クレートの一部を kindsモジュールと utilsモジュールに体系化する内部構造は、artクレートの使用方法を理解しようとする人には、何も役に立つ情報を含んでいません。代わりに、開発者がどこを見るべきか割り出す必要があるので、artクレートのモジュール構造は混乱を招き、また、

開発者はモジュール名を use文で指定しなければならないので、この構造は不便です。

公開 API から内部体系を除去するために、リスト 14-3 の artクレートコードを変更し、pub use文を追加して、最上位で要素を再エクスポートすることができます。リスト 14-5 みたいにね:

ファイル名: src/lib.rs

```
//! # Art
//!
//! A library for modeling artistic concepts.

pub use kinds::PrimaryColor;
pub use kinds::SecondaryColor;
pub use utils::mix;

pub mod kinds {
    // --snip--
}

pub mod utils {
    // --snip--
}
```

リスト 14-5: pub use文を追加して要素を再エクスポートする

このクレートに対して cargo docが生成する API ドキュメンテーションは、これで図 14-4 のようにトップページに再エクスポートを列挙しリンクするので、PrimaryColorと SecondaryColor型と mix関数を見つけやすくします。

図 14-4: 再エクスポートを列挙する artのドキュメンテーションのトップページ

artクレートのユーザは、それでも、リスト 14-4 にデモされているように、リスト 14-3 の内部構造を見て使用することもできますし、リスト 14-5 のより便利な構造を使用することもできます。リスト 14-6 に示したようにね:

ファイル名: src/main.rs

```
extern crate art;

use art::PrimaryColor;
use art::mix;

fn main() {
    // --snip--
}
```

リスト 14-6: artクレートの再エクスポートされた要素を使用するプログラム

ネストされたモジュールがたくさんあるような場合、最上位階層で pub useにより型を再エクスポートすることは、クレートの使用者の経験に大きな違いを生みます。

役に立つ API 構造を作ることは、科学というよりも芸術の領域であり、ユーザにとって何が最善の API なのか、探求するために繰り返してみることができます。pub useは、内部的なクレート構造に

柔軟性をもたらし、その内部構造をユーザに提示する構造から切り離してくれます。インストールしてある他のクレートを見て、内部構造が公開 API と異なっているか確認してみてください。

17.2.3　Crates.io のアカウントをセットアップする

クレートを公開する前に、crates.io のアカウントを作成し、API トークンを取得する必要があります。そうするには、crates.io のホームページを訪れ、Github アカウントでログインしてください。(現状は、Github アカウントがなければなりませんが、いずれは他の方法でもアカウントを作成できるようになる可能性があります。) ログインしたら、`https://crates.io/me/` で自分のアカウントの設定に行き、API キーを取り扱ってください。そして、`cargo login`コマンドを API キーとともに実行してください。以下のようにね:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

このコマンドは、Cargo に API トークンを知らせ、*~/.cargo/credentials* にローカルに保存します。このトークンは、**秘密**です: 他人とは共有しないでください。なんらかの理由で他人と共有してしまったら、古いものを破棄して crates.io で新しいトークンを生成するべきです。

17.2.4　新しいクレートにメタデータを追加する

アカウントはできたので、公開したいクレートがあるとしましょう。公開前に、*Cargo.toml* ファイルの `[package]`セクションに追加することでクレートにメタデータを追加する必要があるでしょう。

クレートには、独自の名前が必要でしょう。クレートをローカルで作成している間、クレートの名前はなんでもいい状態でした。ところが、crates.io のクレート名は、最初に来たもの勝ちの精神で付与されていますので、一旦クレート名が取られてしまったら、その名前のクレートを他の人が公開することは絶対できません。もう使われているか、サイトで使いたい名前を検索してください。まだなら、*Cargo.toml* ファイルの `[package]`以下の名前を編集して、名前を公開用に使ってください。以下のように:

ファイル名: Cargo.toml

```
[package]
name = "guessing_game"
```

たとえ、独自の名前を選択していたとしても、この時点で `cargo publish`を実行すると、警告とエラーが出ます:

```
$ cargo publish
    Updating registry `https://github.com/rust-lang/crates.io-index`
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
(警告: マニフェストに説明、ライセンス、ライセンスファイル、ドキュメンテーション、ホーム
    ページ、
リポジトリのいずれかがありません)
--snip--
```

```
error: api errors: missing or empty metadata fields: description, license.
(エラー: APIエラー: 存在しないメタデータフィールド: description, license)
```

原因は、大事な情報を一部入れていないからです: 説明とライセンスは、他の人が自分のクレートは何をし、どんな条件の元で使っていいのかを知るために必要なのです。このエラーを解消するには、*Cargo.toml* ファイルにこの情報を入れ込む必要があります。

1 文か 2 文程度の説明をつけてください。これは、検索結果に表示されますからね。`license`フィールドには、**ライセンス識別子**を与える必要があります。Linux 団体の Software Package Data Exchange(SPDX) に、この値に使用できる識別子が列挙されています。例えば、自分のクレートを MIT ライセンスでライセンスするためには、`MIT`識別子を追加してください:

ファイル名: Cargo.toml

```
[package]
name = "guessing_game"
license = "MIT"
```

SPDX に出現しないライセンスを使用したい場合、そのライセンスをファイルに配置し、プロジェクトにそのファイルを含め、それから `license`キーを使う代わりに、そのファイルの名前を指定するのに `license-file`を使う必要があります。

どのライセンスが自分のプロジェクトに相 (ふ) 応 (さわ) しいというガイドは、この本の範疇を超えています。Rust コミュニティの多くの人間は、`MIT OR Apache-2.0`のデュアルライセンスを使用することで、Rust 自体と同じようにプロジェクトをライセンスします。この実践は、`OR`で区切られる複数のライセンス識別子を指定して、プロジェクトに複数のライセンスを持たせることもできることを模擬しています。

独自の名前、バージョン、クレート作成時に `cargo new`が追加した筆者の詳細、説明、ライセンスが追加され、公開準備のできたプロジェクト用の `Cargo.toml`ファイルは以下のような見た目になっていることでしょう:

ファイル名: Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
description = "A fun game where you guess what number the computer has chosen."
              (コンピュータが選択した数字を言い当てる面白いゲーム)
license = "MIT OR Apache-2.0"

[dependencies]
```

Cargo のドキュメンテーションには、指定して他人が発見しより容易くクレートを使用できることを保証する他のメタデータが解説されています。

17.2.5　Crates.io に公開する

アカウントを作成し、API トークンを保存し、クレートの名前を決め、必要なメタデータを指定したので、公開する準備が整いました！ クレートを公開すると、特定のバージョンが、crates.io に他の人が使用できるようにアップロードされます。

公開は**永久**なので、クレートの公開時には気をつけてください。バージョンは絶対に上書きできず、コードも削除できません。crates.io の一つの主な目標が、crates.io のクレートに依存している全てのプロジェクトのビルドが、動き続けるようにコードの永久アーカイブとして機能することなのです。バージョン削除を可能にしてしまうと、その目標を達成するのが不可能になってしまいます。ですが、公開できるクレートバージョンの数には制限はありません。

再度 `cargo publish`コマンドを実行してください。今度は成功するはずです:

```
$ cargo publish
 Updating registry `https://github.com/rust-lang/crates.io-index`
Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
 Finished dev [unoptimized + debuginfo] target(s) in 0.19 secs
Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

おめでとうございます！ Rust コミュニティとコードを共有し、誰でも自分のクレートを依存として簡単に追加できます。

17.2.6　既存のクレートの新バージョンを公開する

クレートに変更を行い、新バージョンをリリースする準備ができたら、*Cargo.toml* ファイルに指定された `version`の値を変更し、再公開します。セマンティックバージョンルールを使用して加えた変更の種類に基づいて次の適切なバージョン番号を決定してください。そして、`cargo publish`を実行し、新バージョンをアップロードします。

17.2.7　cargo yank で Crates.io からバージョンを削除する

以前のバージョンのクレートを削除することはできないものの、新しい依存として将来的にプロジェクトに追加することを防ぐことはできます。ある理由により、クレートバージョンが壊れている場合に有用です。そのような場面において、Cargo はクレートバージョンの**取り下げ**をサポートしています。

バージョンを取り下げると、他の既存のプロジェクトには、引き続きダウンロードし、そのバージョンに依存させ続けつつ、新規プロジェクトが新しくそのバージョンに依存しだすことを防ぎます。本質的に取り下げは、*Cargo.lock* が存在するプロジェクトは全て壊れないことを意味し、将来的に *Cargo.lock* ファイルが生成されるものは、取り下げられたバージョンを使わないのです。

あるバージョンのクレートを取り下げるには、cargo yankを実行し、取り下げたいバージョンを指定します:

```
$ cargo yank --vers 1.0.1
```

--undoをコマンドに付与することで、取り下げを取り消し、再度あるバージョンにプロジェクトを依存させ始めることもできます:

```
$ cargo yank --vers 1.0.1 --undo
```

取り下げは、コードの削除は一切しません。例として、取り下げ機能は、誤ってアップロードされた秘密鍵を削除するためのものではありません。もしそうなってしまったら、即座に秘密鍵をリセットしなければなりません。

17.3　Cargo のワークスペース

第 12 章で、バイナリクレートとライブラリクレートを含むパッケージを構築しました。プロジェクトの開発が進むにつれて、ライブラリクレートの肥大化が続き、さらに複数のライブラリクレートにパッケージを分割したくなることでしょう。この場面において、Cargo はワークスペースという協調して開発された関連のある複数のパッケージを管理するのに役立つ機能を提供しています。

17.3.1　Creating a Workspace

17.3.2　ワークスペースを生成する

ワークスペースは、同じ *Cargo.lock* と出力ディレクトリを共有する一連のパッケージです。ワークスペースを使用したプロジェクトを作成し、ワークスペースの構造に集中できるよう、瑣末なコードを使用しましょう。ワークスペースを構築する方法は複数ありますが、一般的な方法を提示しましょう。バイナリ 1 つとライブラリ 2 つを含むワークスペースを作ります。バイナリは、主要な機能を提供しますが、2 つのライブラリに依存しています。一方のライブラリは、add_one関数を提供し、2 番目のライブラリは、add_two関数を提供します。これら 3 つのクレートが同じワークスペースの一部になります。ワークスペース用の新しいディレクトリを作ることから始めましょう:

```
$ mkdir add
$ cd add
```

次に *add* ディレクトリにワークスペース全体を設定する *Cargo.toml* ファイルを作成します。このファイルには、他の *Cargo.toml* ファイルで見かけるような [package]セクションやメタデータはありません。代わりにバイナリクレートへのパスを指定することでワークスペースにメンバを追加させてくれる [workspace]セクションから開始します; 今回の場合、そのパスは *adder* です:

ファイル名: Cargo.toml

```
[workspace]
```

```
members = [
    "adder",
]
```

次に、*add* ディレクトリ内で `cargo new`を実行することで `adder`バイナリクレートを作成しましょう:

```
$ cargo new --bin adder
     Created binary (application) `adder` project
```

この時点で、`cargo build`を走らせるとワークスペースを構築できます。*add* ディレクトリに存在するファイルは、以下のようになるはずです:

```
├── Cargo.lock
├── Cargo.toml
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

ワークスペースには、コンパイルした生成物を置けるように最上位に**ターゲット**のディレクトリがあります; `adder`クレートには**ターゲット**ディレクトリはありません。*adder* ディレクトリ内部から `cargo build`を走らせることになっていたとしても、コンパイルされる生成物は、*add/adder/target* ではなく、*add/target* に落ち着くでしょう。ワークスペースのクレートは、お互いに依存しあうことを意味するので、Cargo はワークスペースの**ターゲット**ディレクトリをこのように構成します。各クレートが**ターゲット**ディレクトリを持っていたら、各クレートがワークスペースの他のクレートを再コンパイルし、**ターゲット**ディレクトリに生成物がある状態にしなければならないでしょう。一つの**ターゲット**ディレクトリを共有することで、クレートは不必要な再ビルドを回避できるのです。

17.3.3　ワークスペース内に 2 番目のクレートを作成する

次に、ワークスペースに別のメンバクレートを作成し、`add-one`と呼びましょう。最上位の *Cargo.toml* を変更して `members`リストで *add-one* パスを指定するようにしてください:

ファイル名: Cargo.toml

```
[workspace]

members = [
    "adder",
    "add-one",
]
```

それから、`add-one`という名前のライブラリクレートを生成してください:

```
$ cargo new add-one
     Created library `add-one` project
```

これで *add* ディレクトリには、以下のディレクトリやファイルが存在するはずです:

```
├── Cargo.lock
├── Cargo.toml
├── add-one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

add-one/src/lib.rs ファイルに `add_one`関数を追加しましょう:

ファイル名: add-one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

ワークスペースにライブラリクレートが存在するようになったので、バイナリクレート `adder`をライブラリクレートの `add-one`に依存させられます。まず、`add-one`へのパス依存を *adder/Cargo.toml* に追加する必要があります:

ファイル名: adder/Cargo.toml

```
[dependencies]

add-one = { path = "../add-one" }
```

Cargo はワークスペースのクレートが、お互いに依存しているとは想定していないので、クレート間の依存関係について明示する必要があります。

次に、`adder`クレートの `add-one`クレートから `add_one`関数を使用しましょう。*adder/src/main.rs* ファイルを開き、冒頭に `extern crate`行を追加して新しい `add-one`ライブラリクレートをスコープに導入してください。それから `main`関数を変更し、`add_one`関数を呼び出します。リスト 14-7 のようにね:

ファイル名: adder/src/main.rs

```
extern crate add_one;

fn main() {
    let num = 10;
    // こんにちは世界！{}+1は{}!
    println!("Hello, world! {} plus one is {}!", num, add_one::add_one(num));
}
```

リスト 14-7: `adder`クレートから `add-one`ライブラリクレートを使用する

最上位の *add* ディレクトリで `cargo build`を実行することでワークスペースをビルドしましょう！

```
$ cargo build
   Compiling add-one v0.1.0 (file:///projects/add/add-one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.68 secs
```

add ディレクトリからバイナリクレートを実行するには、-p引数とパッケージ名を cargo runと共に使用して、使用したいワークスペースのパッケージを指定する必要があります:

```
$ cargo run -p adder
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

これにより、*adder/src/main.rs* のコードが実行され、これは add_oneクレートに依存しています。

17.3.3.1　ワークスペースの外部クレートに依存する

ワークスペースには、各クレートのディレクトリそれぞれに *Cargo.lock* が存在するのではなく、ワークスペースの最上位階層にただ一つの *Cargo.lock* が存在するだけのことに注目してください。これにより、全クレートが全依存の同じバージョンを使用していることが確認されます。randクレートを *adder/Cargo.toml* と *add-one/Cargo.toml* ファイルに追加すると、Cargo は両者をあるバージョンの randに解決し、それを一つの *Cargo.lock* に記録します。ワークスペースの全クレートに同じ依存を使用させるということは、ワークスペースのクレートが相互に互換性を常に維持するということになります。*add-one/Cargo.toml* ファイルの [dependencies]セクションに randクレートを追加して、add-oneクレートで randクレートを使用できます:

ファイル名: add-one/Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

これで、*add-one/src/lib.rs* ファイルに extern crate rand;を追加でき、*add* ディレクトリで cargo buildを実行することでワークスペース全体をビルドすると、randクレートを持ってきてコンパイルするでしょう:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.14
   --snip--
   Compiling rand v0.3.14
   Compiling add-one v0.1.0 (file:///projects/add/add-one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 10.18 secs
```

さて、最上位の *Cargo.lock* は、randに対する add-oneの依存の情報を含むようになりました。ですが、randはワークスペースのどこかで使用されているにも関わらず、それぞれの *Cargo.toml* ファイルにも、randを追加しない限り、ワークスペースの他のクレートでそれを使用することはできません。

例えば、adderクレートの *adder/src/main.rs* ファイルに extern crate rand;を追加すると、エラーが出ます:

```
$ cargo build
   Compiling adder v0.1.0 (file:///projects/add/adder)
error: use of unstable library feature 'rand': use `rand` from crates.io (see
issue #27703)
(エラー: 不安定なライブラリの機能'rand'を使用しています: crates.ioの`rand`を使用してください)
 --> adder/src/main.rs:1:1
  |
1 | extern crate rand;
```

これを修正するには、adderクレートの *Cargo.toml* ファイルを編集し、同様にそのクレートが randに依存していることを示してください。adderクレートをビルドすると、randを *Cargo.lock* の adderの依存一覧に追加しますが、randのファイルが追加でダウンロードされることはありません。Cargoが、ワークスペースの randを使用するどのクレートも、同じバージョンを使っていることを確かめてくれるのです。ワークスペース全体で randの同じバージョンを使用することにより、複数のコピーが存在しないのでスペースを節約し、ワークスペースのクレートが相互に互換性を維持することを確かめます。

17.3.3.2　ワークスペースにテストを追加する

さらなる改善として、add_oneクレート内に add_one::add_one関数のテストを追加しましょう:

ファイル名: add-one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

では、最上位の *add* ディレクトリで cargo testを実行してください:

```
$ cargo test
   Compiling add-one v0.1.0 (file:///projects/add/add-one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
     Running target/debug/deps/add_one-f0253159197f7841

running 1 test
test tests::it_works ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

     Running target/debug/deps/adder-f88af9d2cc175a5e

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

出力の最初の区域が、add-oneクレートの it_worksテストが通ったことを示しています。次の区域には、adderクレートにはテストが見つなかったことが示され、さらに最後の区域には、add-oneクレートにドキュメンテーションテストは見つからなかったと表示されています。このような構造をしたワークスペースで cargo testを走らせると、ワークスペースの全クレートのテストを実行します。

-pフラグを使用し、テストしたいクレートの名前を指定することで最上位ディレクトリから、ワークスペースのある特定のクレート用のテストを実行することもできます:

```
$ cargo test -p add-one
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/add_one-b3235fea9a156f74

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

この出力は、cargo testが add-oneクレートのテストのみを実行し、adderクレートのテストは実行しなかったことを示しています。

ワークスペースのクレートを*https://crates.io/*に公開したら、ワークスペースのクレートは個別に公開される必要
cargo publishコマンドには--allフラグや-pフラグはないので、各クレートのディレクトリに移動して、ワークスペースの各クレートを cargo publishして、公開しなければなりません。

鍛錬を積むために、add-oneクレートと同様の方法でワークスペースに add-twoクレートを追加してください！

プロジェクトが肥大化してきたら、ワークスペースの使用を考えてみてください: 大きな一つのコードの塊よりも、微細で個別のコンポーネントの方が理解しやすいです。またワークスペースにクレートを保持することは、同時に変更されることが多いのなら、協調しやすくなることにも繋がり

ます。

17.4　cargo install で Crates.io からバイナリをインストールする

cargo installコマンドにより、バイナリクレートをローカルにインストールし、使用することができます。これは、システムパッケージを置き換えることを意図したものではありません。即 (すなわ) ち、Rust の開発者が、他人が crates.io に共有したツールをインストールするのに便利な方法を意味するのです。バイナリターゲットを持つパッケージのみインストールできることに注意してください。バイナリターゲットとは、クレートが *src/main.rs* ファイルやバイナリとして指定された他のファイルを持つ場合に生成される実行可能なプログラムのことであり、単独では実行不可能なものの、他のプログラムに含むのには適しているライブラリターゲットとは一線を画します。通常、クレートには、*README* ファイルに、クレートがライブラリかバイナリターゲットか、両方をもつかという情報があります。

cargo installでインストールされるバイナリは全て、インストールのルートの *bin* フォルダに保持されます。Rust を *rustup.rs* を使用し、独自の設定を何も行なっていなければ、このディレクトは、*$HOME/.cargo/bin* になります。cargo installでインストールしたプログラムを実行できるようにするためには、そのディレクトリが$PATHに含まれていることを確かめてください。

例えば、第 12 章で、ファイルを検索する ripgrepという grepツールの Rust 版があることに触れました。ripgrepをインストールしたかったら、以下を実行することができます:

```
$ cargo install ripgrep
Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading ripgrep v0.3.2
 --snip--
   Compiling ripgrep v0.3.2
    Finished release [optimized + debuginfo] target(s) in 97.91 secs
  Installing ~/.cargo/bin/rg
```

出力の最後の行が、インストールされたバイナリの位置と名前を示していて、ripgrepの場合、rgです。インストールディレクトリが$PATHに存在する限り、前述したように、rg --helpを走らせて、より高速で Rust らしいファイル検索ツールを使用し始めることができます！

17.5　独自のコマンドで Cargo を拡張する

Cargo は変更する必要なく、新しいサブコマンドで拡張できるように設計されています。$PATHにあるバイナリが cargo-somethingという名前なら、cargo somethingを実行することで、Cargo のサブコマンドであるかのように実行することができます。このような独自のコマンドは、cargo --listを実行すると、列挙もされます。cargo installを使用して拡張をインストールし、それから組み込みの Cargo ツール同様に実行できることは、Cargo の設計上の非常に便利な恩恵です！

17.6　まとめ

Cargo で crates.io とコードを共有することは、Rust のエコシステムを多くの異なる作業に有用にするものの一部です。Rust の標準ライブラリは、小さく安定的ですが、クレートは共有および使用しやすく、言語とは異なるタイムラインで進化します。積極的に crates.io で自分にとって有用なコードを共有してください; 他の誰かにとっても、役に立つものであることでしょう！

第18章

スマートポインタ

ポインタは、メモリのアドレスを含む変数の一般的な概念です。このアドレスは、何らかの他のデータを参照、または「指します」。Rust において、最もありふれた種類のポインタは、参照であり、第 4 章で習いましたね。参照は、&記号で示唆され、指している値を借用します。データを参照すること以外に特別な能力は何もありません。また、オーバーヘッドもなく、最も頻繁に使われる種類のポインタです。

一方、**スマートポインタ**は、ポインタのように振る舞うだけでなく、追加のメタデータと能力があるデータ構造です。スマートポインタという概念は、Rust に固有のものではありません: スマートポインタは、C++ に端を発し、他の言語にも存在しています。Rust では、標準ライブラリに定義された色々なスマートポインタが、参照以上の機能を提供します。この章で探究する一つの例が、**参照カウント**方式のスマートポインタ型です。このポインタにより、所有者の数を追いかけることでデータに複数の所有者を持たせることができ、所有者がいなくなったら、データの片付けをしてくれます。

所有権と借用の概念を使う Rust で、参照とスマートポインタの別の差異は、参照はデータを借用するだけのポインタであることです; 対照的に多くの場合、スマートポインタは指しているデータを**所有**します。

当時は、スマートポインタとは呼ばなかったものの、第 8 章の `String`や `Vec<T>`のように、この本の中でいくつかのスマートポインタに遭遇してきました。これらの型はどちらも、あるメモリを所有し、それを弄ることができるので、スマートポインタに数えられます。また、メタデータ (キャパシティなど) や追加の能力、あるいは保証 (`String`ならデータが常に有効な UTF-8 であると保証することなど) もあります。

スマートポインタは普通、構造体を使用して実装されています。スマートポインタを通常の構造体と区別する特徴は、スマートポインタは、`Deref`と `Drop`トレイトを実装していることです。`Deref`トレイトにより、スマートポインタ構造体のインスタンスは、参照のように振る舞うことができるので、参照あるいはスマートポインタのどちらとも動作するコードを書くことができるのです。`Drop`トレイトにより、スマートポインタのインスタンスがスコープを外れた時に走るコードをカスタマイズする

ことができます。この章では、どちらのトレイトについても議論し、これらのトレイトがスマートポインタにとって重要な理由をデモします。

スマートポインタパターンが Rust においてよく使われる一般的なデザインパターンだとして、この章では、全ての既存のスマートポインタを講義しません。多くのライブラリに独自のスマートポインタがあり、自分だけのスマートポインタを書くことさえできます。標準ライブラリの最もありふれたスマートポインタを講義します:

- ヒープに値を確保する `Box<T>`
- 複数の所有権を可能にする参照カウント型の `Rc<T>`
- `RefCell<T>`を通してアクセスされ、コンパイル時ではなく実行時に借用ルールを強制する型の `Ref<T>`と `RefMut<T>`

さらに、不変な型が、内部の値を可変化する API を晒す**内部可変性**パターンについても講義します。また、**循環参照**についても議論します: 循環参照により、メモリがリークする方法とそれを回避する方法です。

さあ、飛び込みましょう！

18.1　ヒープのデータを指す Box<T>を使用する

最も素直なスマートポインタは**ボックス**であり、その型は `Box<T>`と記述されます。ボックスにより、スタックではなくヒープにデータを格納することができます。スタックに残るのは、ヒープデータへのポインタです。スタックとヒープの違いを再確認するには、第 4 章を参照されたし。

ボックスは、データをスタックの代わりにヒープに格納する以外は、パフォーマンスのオーバーヘッドはありません。しかし、多くのおまけの能力もありません。以下のような場面で最もよく使用するでしょう:

- コンパイル時にはサイズを知ることができない型があり、正確なサイズを要求する文脈でその型の値を使用する時
- 多くのデータがあり、所有権を転送したいが、そうする時にデータがコピーされないことを確認する時
- 値を所有する必要があり、特定の型ではなく特定のトレイトを実装する型であることのみ気にかけている時

この節では、「ボックスで再帰的型を可能にする」節の最初の場面を模擬します。2 番目の場合、多くのデータの所有権を転送するには、データがスタック上でコピーされるので、長い時間がかかり得ます。この場面でパフォーマンスを向上させるには、多くのデータをヒープ上にボックスとして格納することができます。そして、参照しているデータはヒープ上の 1 箇所に留まりつつ、少量のポインタのデータのみをスタック上でコピーするのです。3 番目のケースは、**トレイトオブジェクト**として知られ、第 17 章は、その話題だけに「異なる型の値を許容するトレイトオブジェクトを使用する」1

節全体を捧げています。従って、ここで学ぶことは、第 17 章でまた適用するでしょう！

18.1.1　Box<T>を使ってヒープにデータを格納する

Box<T>のこのユースケースを議論する前に、記法と Box<T>内に格納された値と相互作用する方法について講義しましょう。

リスト 15-1 は、ボックスを使用してヒープに i32の値を格納する方法を示しています:

ファイル名: src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

リスト 15-1: ボックスを使用して i32の値をヒープに格納する

変数 bを定義して値 5を指す Boxの値を持っていて、この値はヒープに確保されています。このプログラムは、b = 5と出力するでしょう; この場合、このデータがスタックにあるのと同じような方法でボックスのデータにアクセスできます。あらゆる所有された値同様、bが mainの終わりでするようにボックスがスコープを抜けたら、メモリから解放されます。メモリの解放は (スタックに格納されている) ボックスと (ヒープに格納されている) 指しているデータに対して起きます。

ヒープに単独の値を置くことはあまり有用ではないので、このように単独でボックスを使用することはあまりありません。単独の i32のような値を規定で格納される場所であるスタックに置くことが、大多数の場合にはより適切です。ボックスがなかったら定義することの叶わない型をボックスが定義させてくれる場合を見ましょう。

18.1.2　ボックスで再帰的な型を可能にする

コンパイル時に、コンパイラは、ある型が取る領域を知る必要があります。コンパイル時にサイズがわからない型の 1 つは、**再帰的な型**であり、これは、型の一部として同じ型の他の値を持つものです。この値のネストは、理論的には無限に続く可能性があるので、コンパイラは再帰的な型の値が必要とする領域を知ることができないのです。しかしながら、ボックスは既知のサイズなので、再帰的な型の定義にボックスを挟むことで再帰的な型を存在させることができるのです。

コンスリストは関数型プログラミング言語では一般的なデータ型ですが、これを再帰的な型の例として探求しましょう。我々が定義するコンスリストは、再帰を除いて素直です; 故に、これから取り掛かる例の概念は、再帰的な型が関わるもっと複雑な場面に遭遇したら必ず役に立つでしょう。

18.1.2.1　コンスリストについてもっと詳しく

コンスリストは、Lisp プログラミング言語とその方言に由来するデータ構造です。Lisp では、cons 関数 (”construct function” の省略形です) が 2 つの引数から新しいペアを構成し、この引数は通常、単独の値と別のペアからなります。これらのペアを含むペアがリストをなすのです。

cons 関数の概念は、より一般的な関数型プログラミングのスラングにもなっています: ”to cons x onto y” は、俗に要素 x をこの新しいコンテナの初めに置き、コンテナ y を従わせて新しいコンテナのインスタンスを生成することを意味します。

コンスリストの各要素は、2 つの要素を含みます: 現在の要素の値と次の要素です。リストの最後の要素は、次の要素なしに `Nil`と呼ばれる値だけを含みます。コンスリストは、繰り返し `cons`関数を呼び出すことで生成されます。繰り返しの規範事例を意味する標準的な名前は、`Nil`です。これは第 6 章の”null” や”nil” の概念とは異なることに注意してください。”null” や”nil” は、無効だったり存在しない値です。

関数型プログラミング言語は、頻繁にコンスリストを使用するものの、Rust ではあまり使用されないデータ構造です。Rust で要素のリストがある場合はほとんどの場合、`Vec<T>`を使用するのがよりよい選択になります。他のより複雑な再帰的なデータ型は、様々な場面で役に立ちますが、コンスリストから始めることで、大して気を散らすことなく再帰的なデータ型をボックスが定義させてくれる方法を探求することができます。

リスト 15-2 には、コンスリストの enum 定義が含まれています。このコードは、`List`型が既知のサイズではないため、まだコンパイルできないことに注意してください。既知のサイズがないことをこれから模擬します。

ファイル名: src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
```

リスト 15-2: `i32`値のコンスリストデータ構造を表す enum を定義する最初の試行

> 注釈: この例のためだけに `i32`値をだけを保持するコンスリストを実装します。第 10 章で議論したように、ジェネリクスを使用してどんな型の値も格納できるコンスリストを定義して実装することもできました。

この `List`型を使用してリスト `1`, `2`, `3`を格納すると、リスト 15-3 のコードのような見た目になるでしょう:

ファイル名: src/main.rs

```
use List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

リスト 15-3: `List`enum を使用してリスト `1`, `2`, `3`を格納する

最初の `Cons`値は、`1`と別の `List`値を保持しています。この `List`値は、`2`とまた別の `List`値を保持する別の `Cons`値です。この `List`値は、`3`と、ついにリストの終端を通知する非再帰的なバリアントの

Nilになる List値を保持するまたまた別の Cons値です。

リスト 15-3 のコードをコンパイルしようとすると、リスト 15-4 に示したエラーが出ます:

```
error[E0072]: recursive type `List` has infinite size
(エラー: 再帰的な型`List`は無限のサイズです)
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ----- recursive without indirection
  |
  = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
  make `List` representable
  (助言: 間接参照(例: `Box`、`Rc`、あるいは`&`)をどこかに挿入して、`List`を表現可能にし
      てください)
```

リスト 15-4: 再帰的な enum を定義しようとすると得られるエラー

エラーは、この型は「無限のサイズである」と表示しています。理由は、再帰的な列挙子を含む List を定義したからです: 自身の別の値を直接保持しているのです。結果として、コンパイラは、List値を格納するのに必要な領域が計算できないのです。このエラーが得られた理由を少し噛み砕きましょう。まず、非再帰的な型の値を格納するのに必要な領域をどうコンパイラが決定しているかを見ましょう。

18.1.2.2　非再帰的な型のサイズを計算する

第 6 章で enum 定義を議論した時にリスト 6-2 で定義した Messageenum を思い出してください:

```
1 enum Message {
2     Quit,
3     Move { x: i32, y: i32 },
4     Write(String),
5     ChangeColor(i32, i32, i32),
6 }
```

Message値一つにメモリを確保するために必要な領域を決定するために、コンパイラは、各列挙子を見てどの列挙子が最も領域を必要とするかを確認します。コンパイラは、Message::Quitは全く領域を必要とせず、Message::Moveは i32値を 2 つ格納するのに十分な領域が必要などと確かめます。ただ 1 つの列挙子しか使用されないので、Message値一つが必要とする最大の領域は、最大の列挙子を格納するのに必要になる領域です。

これをコンパイラがリスト 15-2 の Listenum のような再帰的な型が必要とする領域を決定しようとする時に起こることと比較してください。コンパイラは、Cons列挙子を見ることから始め、この列挙子には、型 i32値が一つと型 Listの値が一つ保持されます。故に、Consは 1 つの i32と Listのサイズに等しい領域を必要とします。Listが必要とするメモリ量を計算するのに、コンパイラは Cons列挙子から列挙子を観察します。Cons列挙子は型 i32を 1 つと型 Listの値 1 つを保持し、この過程は無限に続きます。図 15-1 のようにね。

図 15-1: 無限の Cons列挙子からなる無限の List

18.1.2.3　Box<T>で既知のサイズの再帰的な型を得る

コンパイラは、再帰的に定義された型に必要なメモリ量を計算できないので、リスト 15-4 ではエラーを返します。しかし、エラーには確かにこの役に立つ提言が含まれています:

```
= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
make `List` representable
```

この提言において、「間接参照」は、値を直接格納する代わりに、データ構造を変更して値へのポインタを代わりに格納することで、値を間接的に格納することを意味します。

Box<T>はポインタなので、コンパイラには Box<T>が必要とする領域が必ずわかります: ポインタのサイズは、指しているデータの量によって変わることはありません。つまり、別の List値を直接置く代わりに、Cons列挙子の中に Box<T>を配置することができます。Box<T>は、Cons列挙子の中ではなく、ヒープに置かれる次の List値を指します。概念的には、それでも他のリストを「保持する」リストとともに作られたリストがありますが、この実装は今では、要素はお互いの中にあるというよりも、隣り合って配置するような感じになります。

リスト 15-2 の Listenum の定義とリスト 15-3 の Listの使用をリスト 15-5 のコードに変更することができ、これはコンパイルが通ります:

ファイル名: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```

リスト 15-5: 既知のサイズを得るために Box<T>を使用する Listの定義

Cons列挙子は、1 つの i32のサイズプラスボックスのポインタデータを格納する領域を必要とするでしょう。Nil列挙子は、値を格納しないので、Cons列挙子よりも必要な領域は小さいです。これで、どんな List値も i321 つのサイズプラスボックスのポインタデータのサイズを必要とすることがわかりました。ボックスを使うことで、無限の再帰的な繰り返しを破壊したので、コンパイラは、List値を格納するのに必要なサイズを計算できます。図 15-2 は、Cons列挙子の今の見た目を示しています。

図 15-2: Consが Boxを保持しているので、無限にサイズがあるわけではない List

ボックスは、間接参照とヒープメモリ確保だけを提供します; 他のスマートポインタ型で目撃するような、他の特別な能力は何もありません。これらの特別な能力が招くパフォーマンスのオーバー

ヘッドもないので、間接参照だけが必要になる唯一の機能であるコンスリストのような場合に有用になり得ます。より多くのボックスのユースケースは第 17 章でもお見かけするでしょう。

Box<T>型は、Derefトレイトを実装しているので、スマートポインタであり、このトレイトによりBox<T>の値を参照のように扱うことができます。Box<T>値がスコープを抜けると、Dropトレイト実装によりボックスが参照しているヒープデータも片付けられます。これら 2 つのトレイトをより詳しく探求しましょう。これら 2 つのトレイトは、この章の残りで議論する他のスマートポインタ型で提供される機能にとってさらに重要でしょう。

18.2　Deref トレイトでスマートポインタを普通の参照のように扱う

Derefトレイトを実装することで**参照外し演算子**の*(掛け算やグロブ演算子とは対照的に) の振る舞いをカスタマイズすることができます。スマートポインタを普通の参照のように扱えるようにDerefを実装することで、参照に対して処理を行うコードを書き、そのコードをスマートポインタとともに使用することもできます。

まずは、参照外し演算子が普通の参照に対して動作するところを見ましょう。それからBox<T>のように振る舞う独自の型を定義し、参照外し演算子が新しく定義した型に対して参照のように動作しない理由を確認しましょう。Derefトレイトを実装することでスマートポインタが参照と似た方法で動作するようにできる方法を探求します。そして、Rust の**参照外し型強制**機能と、それにより参照やスマートポインタに取り掛かる方法を見ます。

18.2.1　参照外し演算子で値までポインタを追いかける

普通の参照は 1 種のポインタであり、ポインタの捉え方の一つが、どこか他の場所に格納された値への矢印としてです。リスト 15-6 で、i32値への参照を生成し、それから参照外し演算子を使用して参照をデータまで追いかけています:

ファイル名: src/main.rs

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

リスト 15-6: 参照外し演算子を使用して参照を i32値まで追いかける

変数 xは i32値の 5を保持しています。yを xへの参照にセットします。xは 5に等しいとアサートできます。しかしながら、yの値に関するアサートを行いたい場合、*yを使用して参照を指している値まで追いかけなければなりません (そのため**参照外し**です)。一旦、yを参照外ししたら、yが指している 5と比較できる整数値にアクセスできます。

代わりに assert_eq!(5, y);と書こうとしたら、こんなコンパイルエラーが出るでしょう:

```
error[E0277]: the trait bound `{integer}: std::cmp::PartialEq<&{integer}>` is
not satisfied
(エラー: トレイト境界`{integer}: std::cmp::PartialEq<&{integer}>`は満たされていません)
 --> src/main.rs:6:5
  |
6 |     assert_eq!(5, y);
  |     ^^^^^^^^^^^^^^^^ can't compare `{integer}` with `&{integer}`
  |
  = help: the trait `std::cmp::PartialEq<&{integer}>` is not implemented for
  `{integer}`
  (助言: トレイト`std::cmp::PartialEq<&{integer}>`は`{integer}`に対して実装されていませ
      ん)
```

参照と数値は異なる型なので、比較することは許容されていません。参照外し演算子を使用して、参照を指している値まで追いかけなければならないのです。

18.2.2　Box<T>を参照のように使う

リスト 15-6 のコードを参照の代わりに Box<T>を使うように書き直すことができます; 参照外し演算子は、リスト 15-7 に示したように動くでしょう:

ファイル名: src/main.rs

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

リスト 15-7: Box<i32>に対して参照外し演算子を使用する

リスト 15-7 とリスト 15-6 の唯一の違いは、ここでは、xの値を指す参照ではなく、xの値を指すボックスのインスタンスに yをセットしていることです。最後のアサートで参照外し演算子を使用して yが参照だった時のようにボックスのポインタを追いかけることができます。次に、独自のボックス型を定義することで参照外し演算子を使用させてくれる Box<T>について何が特別なのかを探求します。

18.2.3　独自のスマートポインタを定義する

標準ライブラリが提供している Box<T>型に似たスマートポインタを構築して、スマートポインタは規定で、どう異なって参照に比べて振る舞うのか経験しましょう。それから、参照外し演算子を使う能力を追加する方法に目を向けましょう。

Box<T>型は究極的に 1 要素のタプル構造体として定義されているので、リスト 15-8 は、同じように MyBox<T>型を定義しています。また、Box<T>に定義された new関数と合致する new関数も定義しています。

ファイル名: src/main.rs

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

リスト 15-8: MyBox<T>型を定義する

MyBoxという構造体を定義し、ジェネリック引数の Tを宣言しています。自分の型にどんな型の値も保持させたいからです。MyBox型は、型 Tを 1 要素持つタプル構造体です。MyBox::new関数は型 Tの引数を 1 つ取り、渡した値を保持する MyBoxインスタンスを返します。

試しにリスト 15-7 の main関数をリスト 15-8 に追加し、Box<T>の代わりに定義した MyBox<T>型を使うよう変更してみてください。コンパイラは MyBoxを参照外しする方法がわからないので、リスト 15-9 のコードはコンパイルできません。

ファイル名: src/main.rs

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

リスト 15-9: 参照と Box<T>を使ったのと同じように MyBox<T>を使おうとする

こちらが結果として出るコンパイルエラーです:

```
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
(エラー: 型`MyBox<{integer}>`は参照外しできません)
  --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^
```

MyBox<T>に参照外しの能力を実装していないので、参照外しできません。*演算子で参照外しできるようにするには、Derefトレイトを実装します。

18.2.4　Deref トレイトを実装して型を参照のように扱う

第 10 章で議論したように、トレイトを実装するには、トレイトの必須メソッドに実装を提供する必要があります。Derefトレイトは標準ライブラリで提供されていますが、selfを借用し、内部のデータへの参照を返す derefという 1 つのメソッドを実装する必要があります。リスト 15-10 には、MyBox の定義に追記する Derefの実装が含まれています:

ファイル名: src/main.rs

```
use std::ops::Deref;

# struct MyBox<T>(T);
impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

リスト 15-10: `MyBox<T>`に `Deref`を実装する

`type Target = T;`という記法は、`Deref`トレイトが使用する関連型を定義しています。関連型は、ジェネリック引数を宣言する少しだけ異なる方法ですが、今は気にする必要はありません; 第 19 章でより詳しく講義します。

`deref`メソッドの本体を`&self.0`で埋めているので、`deref`は`*`演算子でアクセスしたい値への参照を返します。リスト 15-9 の `MyBox<T>`に`*`を呼び出す `main`関数はこれでコンパイルでき、アサートも通ります！

`Deref`がなければ、コンパイラは`&`参照しか参照外しできなくなります。`deref`メソッドによりコンパイラは、`Deref`を実装するあらゆる型の値を取り、`deref`メソッドを呼び出して参照外しの仕方を知っている`&`参照を得る能力を獲得するのです。

リスト 15-9 に`*y`を入力した時、水面下でコンパイラは、実際にはこのようなコードを走らせていました:

```
*(y.deref())
```

コンパイラは、`*`演算子を `deref`メソッド、それから何の変哲もない参照外しの呼び出しに置き換えるので、`deref`メソッドを呼び出す必要があるかどうかを考える必要はないわけです。この Rust の機能により、普通の参照か `Deref`を実装した型があるかどうかと等しく機能するコードを書くことができます。

`deref`メソッドが値への参照を返し、`*(y.deref())`のかっこの外の何の変哲もない参照外しがそれでも必要な理由は、所有権システムです。`deref`メソッドが値への参照ではなく、値を直接返したら、値は `self`から外にムーブされてしまいます。今回の場合や、参照外し演算子を使用する多くの場合には `MyBox<T>`の中の値の所有権を奪いたくはありません。

`*`演算子は `deref`メソッドの呼び出し 1 回とコードで`*`打つたび、ただ 1 回の`*`演算子の呼び出しに置き換えられることに注意してください。`*`演算子の置き換えは、無限に繰り返されないので、型 `i32`に行き着き、リスト 15-9 で `assert_eq!`の `5`と合致します。

18.2.5　関数やメソッドで暗黙的な参照外し型強制

参照外し型強制は、コンパイラが関数やメソッドの実引数に行う便利なものです。参照外し型強制は、Derefを実装する型への参照を Derefが元の型を変換できる型への参照に変換します。参照外し型強制は、特定の型の値への参照を関数やメソッド定義の引数型と一致しない引数として関数やメソッドに渡すときに自動的に発生します。一連の derefメソッドの呼び出しが、提供した型を引数が必要とする型に変換します。

参照外し型強制は、関数やメソッド呼び出しを書くプログラマが&や*で多くの明示的な参照や参照外しとして追加する必要がないように、Rust に追加されました。また、参照外し型強制のおかげで参照あるいはスマートポインタのどちらかで動くコードをもっと書くことができます。

参照外し型強制が実際に動いていることを確認するため、リスト 15-8 で定義した MyBox<T>と、リスト 15-10 で追加した Derefの実装を使用しましょう。リスト 15-11 は、文字列スライス引数のある関数の定義を示しています:

ファイル名: src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}
```

リスト 15-11: 型&strの引数 nameのある hello関数

hello関数は、文字列スライスを引数として呼び出すことができます。例えば、hello("Rust")などです。参照外し型強制により、helloを型 MyBox<String>の値への参照とともに呼び出すことができます。リスト 15-12 のようにね:

ファイル名: src/main.rs

```
# use std::ops::Deref;
#
# struct MyBox<T>(T);
#
# impl<T> MyBox<T> {
#     fn new(x: T) -> MyBox<T> {
#         MyBox(x)
#     }
# }
#
# impl<T> Deref for MyBox<T> {
#     type Target = T;
#
#     fn deref(&self) -> &T {
#         &self.0
#     }
# }
#
# fn hello(name: &str) {
#     println!("Hello, {}!", name);
```

```
# }
#
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

リスト 15-12: helloを MyBox<String>値とともに呼び出し、参照外し型強制のおかげで動く

ここで、hello関数を引数&mとともに呼び出しています。この引数は、MyBox<String>値への参照です。リスト 15-10 で MyBox<T>に Derefトレイトを実装したので、コンパイラは derefを呼び出すことで、&MyBox<String>を&Stringに変換できるのです。標準ライブラリは、Stringに文字列スライスを返す Derefの実装を提供していて、この実装は、Derefの API ドキュメンテーションに載っています。コンパイラはさらに derefを呼び出して、&Stringを&strに変換し、これは hello関数の定義と合致します。

Rust に参照外し型強制が実装されていなかったら、リスト 15-12 のコードの代わりにリスト 15-13 のコードを書き、型&MyBox<String>の値で helloを呼び出さなければならなかったでしょう。

ファイル名: src/main.rs

```
# use std::ops::Deref;
#
# struct MyBox<T>(T);
#
# impl<T> MyBox<T> {
#     fn new(x: T) -> MyBox<T> {
#         MyBox(x)
#     }
# }
#
# impl<T> Deref for MyBox<T> {
#     type Target = T;
#
#     fn deref(&self) -> &T {
#         &self.0
#     }
# }
#
# fn hello(name: &str) {
#     println!("Hello, {}!", name);
# }
#
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

リスト 15-13: Rust に参照外し型強制がなかった場合に書かなければならないであろうコード

(*m)が MyBox<String>を Stringに参照外ししています。そして、&と [..]により、文字列全体と等しい Stringの文字列スライスを取り、helloのシグニチャと一致するわけです。参照外し型強制のな

いコードは、これらの記号が関係するので、読むのも書くのも理解するのもより難しくなります。参照外し型強制により、コンパイラはこれらの変換を自動的に扱えるのです。

`Deref`トレイトが関係する型に定義されていると、コンパイラは、型を分析し必要なだけ`Deref::deref`を使用して、参照を得、引数の型と一致させます。`Deref::deref`が挿入される必要のある回数は、コンパイル時に解決されるので、参照外し型強制を活用する実行時の代償は何もありません。

18.2.6　参照外し型強制が可変性と相互作用する方法

`Deref`トレイトを使用して不変参照に対して$\star$をオーバーライドするように、R `DerefMut`トレイトを使用して可変参照の$\star$演算子をオーバーライドできます。

以下の 3 つの場合に型やトレイト実装を見つけた時にコンパイラは、参照外し型強制を行います:

- `T: Deref<Target=U>`の時、`&T`から`&U`
- `T: DerefMut<Target=U>`の時、`&mut T`から`&mut U`
- `T: Deref<Target=U>`の時、`&mut T`から`&U`

前者 2 つは、可変性を除いて一緒です。最初のケースは、`&T`があり、`T`が何らかの型 `U`への `Deref`を実装しているなら、透過的に`&U`を得られると述べています。2 番目のケースは、同じ参照外し型強制が可変参照についても起こることを述べています。

3 番目のケースはもっと巧妙です: Rust はさらに、可変参照を不変参照にも型強制するのです。ですが、逆はできま**せん**: 不変参照は、絶対に可変参照に型強制されないのです。借用ルールにより、可変参照があるなら、その可変参照がそのデータへの唯一の参照に違いありません (でなければ、プログラムはコンパイルできません)。1 つの可変参照を 1 つの不変参照に変換することは、借用ルールを絶対に破壊しません。不変参照を可変参照にするには、そのデータへの不変参照がたった 1 つしかないことが必要ですが、借用ルールはそれを保証してくれません。故に、不変参照を可変参照に変換することが可能であるという前提を敷けません。

18.3　Drop トレイトで片付け時にコードを走らせる

スマートポインタパターンにとって重要な 2 番目のトレイトは、`Drop`であり、これのおかげで値がスコープを抜けそうになった時に起こることをカスタマイズできます。どんな型に対しても `Drop`トレイトの実装を提供することができ、指定したコードは、ファイルやネットワーク接続などのリソースを解放するのに活用できます。`Drop`をスマートポインタの文脈で導入しています。`Drop`トレイトの機能は、ほぼ常にスマートポインタを実装する時に使われるからです。例えば、`Box<T>`は `Drop`をカスタマイズしてボックスが指しているヒープの領域を解放しています。

ある言語では、プログラマがスマートポインタのインスタンスを使い終わる度にメモリやリソースを解放するコードを呼ばなければなりません。忘れてしまったら、システムは詰め込みすぎになりクラッシュする可能性があります。Rust では、値がスコープを抜ける度に特定のコードが走るよう指定

でき、コンパイラはこのコードを自動的に挿入します。結果として、特定の型のインスタンスを使い終わったプログラムの箇所全部にクリーンアップコードを配置するのに配慮する必要はありません。それでもリソースをリークすることはありません。

Dropトレイトを実装することで値がスコープを抜けた時に走るコードを指定してください。Dropトレイトは、selfへの可変参照を取るdropという 1 つのメソッドを実装する必要があります。いつ Rust が dropを呼ぶのか確認するために、今は println!文のある dropを実装しましょう。

リスト 15-14 は、唯一の独自の機能が、インスタンスがスコープを抜ける時に Dropping CustomSmartPointer!と出力するだけの、CustomSmartPointer構造体です。この例は、コンパイラがいつ drop関数を走らせるかをデモしています。

ファイル名: src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        // CustomSmartPointerをデータ`{}`とともにドロップするよ
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer { data: String::from("my stuff") };       // 俺のもの
    let d = CustomSmartPointer { data: String::from("other stuff") };    // 別のもの
    println!("CustomSmartPointers created.");                            //
    CustomSmartPointerが生成された
}
```

リスト 15-14: クリーンアップコードを配置する Dropトレイトを実装する CustomSmartPointer構造体

Dropトレイトは、初期化処理に含まれるので、インポートする必要はありません。CustomSmartPointerに Dropトレイトを実装し、println!を呼び出す dropメソッドの実装を提供しています。drop関数の本体は、自分の型のインスタンスがスコープを抜ける時に走らせたいあらゆるロジックを配置する場所です。ここで何らかのテキストを出力し、コンパイラがいつ dropを呼ぶのかデモしています。

mainで、CustomSmartPointerのインスタンスを 2 つ作り、それから CustomSmartPointers created.と出力しています。mainの最後で、CustomSmartPointerのインスタンスはスコープを抜け、コンパイラは最後のメッセージを出力しながら、dropメソッドに置いたコードを呼び出します。dropメソッドを明示的に呼び出す必要はなかったことに注意してください。

このプログラムを実行すると、以下のような出力が出ます:

```
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
```

```
Dropping CustomSmartPointer with data `my stuff`!
```

インスタンスがスコープを抜けた時に指定したコードを呼び出しながらコンパイラは、dropを自動的に呼び出してくれました。変数は、生成されたのと逆の順序でドロップされるので、dは cより先にドロップされました。この例は、dropメソッドの動き方を見た目で案内するだけですが、通常は、メッセージ出力ではなく、自分の型が走らせる必要のあるクリーンアップコードを指定するでしょう。

18.3.1　std::mem::drop で早期に値をドロップする

残念ながら、自動的な drop機能を無効化することは、素直ではありません。通常、dropを無効化する必要はありません; Dropトレイトの重要な目的は、自動的に考慮されることです。ですが、時として、値を早期に片付けたくなる可能性があります。1 例は、ロックを管理するスマートポインタを使用する時です: 同じスコープの他のコードがロックを獲得できるように、ロックを解放する dropメソッドを強制的に走らせたくなる可能性があります。Rust は、Dropトレイトの dropメソッドを手動で呼ばせてくれません; スコープが終わる前に値を強制的にドロップさせたいなら、標準ライブラリが提供する std::mem:drop関数を呼ばなければなりません。

リスト 15-14 の main関数を変更して手動で Dropトレイトの dropメソッドを呼び出そうとしたら、コンパイルエラーになるでしょう。リスト 15-15 のようにね:

ファイル名: src/main.rs

```
fn main() {
    let c = CustomSmartPointer { data: String::from("some data") };
    println!("CustomSmartPointer created.");
    c.drop();
    // mainの終端の前にCustomSmartPointerがドロップされた
    println!("CustomSmartPointer dropped before the end of main.");
}
```

リスト 15-15: Dropトレイトから dropメソッドを手動で呼び出し、早期に片付けようとする

このコードをコンパイルしてみようとすると、こんなエラーが出ます:

```
error[E0040]: explicit use of destructor method
(エラー: デストラクタメソッドを明示的に使用しています)
  --> src/main.rs:14:7
   |
14 |     c.drop();
   |       ^^^^ explicit destructor calls not allowed
```

明示的に dropを呼び出すことは許されていないことをこのエラーメッセージは述べています。エラーメッセージは**デストラクタ**という専門用語を使っていて、これは、インスタンスを片付ける関数の一般的なプログラミング専門用語です。**デストラクタ**は、**コンストラクタ**に類似していて、これはインスタンスを生成します。Rust の drop関数は、1 種の特定のデストラクタです。

コンパイラはそれでも、mainの終端で値に対して自動的に dropを呼び出すので、dropを明示的に呼ばせてくれません。コンパイラが 2 回同じ値を片付けようとするので、これは**二重解放**エラーになる

でしょう。

値がスコープを抜けるときに dropが自動的に挿入されるのを無効化できず、dropメソッドを明示的に呼ぶこともできません。よって、値を早期に片付けさせる必要があるなら、std::mem::drop関数を使用できます。

std::mem::drop関数は、Dropトレイトの dropメソッドとは異なります。早期に強制的にドロップさせたい値を引数で渡すことで呼びます。この関数は初期化処理に含まれているので、リスト 15-14 の mainを変更して drop関数を呼び出せます。リスト 15-16 のようにね:

ファイル名: src/main.rs

```
# struct CustomSmartPointer {
#     data: String,
# }
#
# impl Drop for CustomSmartPointer {
#     fn drop(&mut self) {
#         println!("Dropping CustomSmartPointer!");
#     }
# }
#
fn main() {
    let c = CustomSmartPointer { data: String::from("some data") };
    println!("CustomSmartPointer created.");
    drop(c);
    // CustomSmartPointerはmainが終わる前にドロップされた
    println!("CustomSmartPointer dropped before the end of main.");
}
```

リスト 15-16: 値がスコープを抜ける前に明示的にドロップするために std::mem::dropを呼び出す

このコードを実行すると、以下のように出力されます:

```
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

Dropping CustomSmartPointer with data `some data`!というテキストが、CustomSmartPointer created.と CustomSmartPointer dropped before the end of mainテキストの間に出力されるので、dropメソッドのコードがその時点で呼び出されて cをドロップしたことを示しています。

Dropトレイト実装で指定されたコードをいろんな方法で使用し、片付けを便利で安全にすることができます: 例を挙げれば、これを使用して独自のメモリアロケータを作ることもできるでしょう。Dropトレイトと Rust の所有権システムがあれば、コンパイラが自動的に行うので、片付けを覚えておく必要はなくなります。

まだ使用中の値を間違って片付けてしまうことに起因する問題を心配もしなくて済みます: 参照が常に有効であると確認してくれる所有権システムが、値が最早使用されなくなった時に dropが 1 回だけ呼ばれることを保証してくれるのです。

これで Box<T>とスマートポインタの特徴の一部を調査したので、標準ライブラリに定義されている

他のスマートポインタをいくつか見ましょう。

18.4　Rc<T>は、参照カウント方式のスマートポインタ

大多数の場合、所有権は明らかです: 一体どの変数が与えられた値を所有しているかわかるのです。ところが、単独の値が複数の所有者を持つ可能性のある場合もあります。例えば、グラフデータ構造では、複数の辺が同じノードを指す可能性があり、概念的にそのノードはそれを指す全ての辺に所有されるわけです。指す辺がなくならない限り、ノードは片付けられるべきではありません。

複数の所有権を可能にするため、Rust には Rc<T>という型があり、これは、*reference counting* の省略形です。Rc<T>型は、値がまだ使用中かどうか決定する値への参照の数を追跡します。値への参照が 0 なら、どの参照も無効にすることなく、値は片付けられます。

Rc<T>を家族部屋のテレビと想像してください。1 人がテレビを見に部屋に入ったら、テレビをつけます。他の人も部屋に入ってテレビを観ることができます。最後の人が部屋を離れる時、もう使用されていないので、テレビを消します。他の人がまだ観ているのに誰かがテレビを消したら、残りのテレビ視聴者が騒ぐでしょう！

ヒープにプログラムの複数箇所で読む何らかのデータを確保したい時に Rc<T>型を使用し、コンパイル時には、どの部分が最後にデータを使用し終わるか決定できません。どの部分が最後に使用し終わるかわかれば、単にその部分をデータの所有者にして、コンパイル時に強制される普通の所有権ルールが効果を発揮するでしょう。

Rc<T>は、シングルスレッドシナリオで使用するためだけのものであることに注意してください。第 16 章で非同期処理について議論する時に、マルチスレッドプログラムで参照カウントをする方法を講義します。

18.4.1　Rc<T>でデータを共有する

リスト 15-5 のコンスリストの例に回帰しましょう。Box<T>を使って定義したことを思い出してください。今回は、両方とも 3 番目のリストの所有権を共有する 2 つのリストを作成します。これは概念的には図 15-3 のような見た目になります:

図 15-3: 3 番目のリスト、aの所有権を共有する 2 つのリスト、bと c

5 と 10 を含むリスト aを作ります。さらにもう 2 つリストを作ります: 3 で始まる bと 4 で始まる cです。bと cのどちらもそれから 5 と 10 を含む最初の aリストに続きます。換言すれば、どちらのリストも 5 と 10 を含む最初のリストを共有しています。

Listの定義を使用して Box<T>とともにこの筋書きを実装しようとしても、うまくいきません。リスト 15-17 のようにね:

ファイル名: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
```

```
}

use List::{Cons, Nil};

fn main() {
    let a = Cons(5,
        Box::new(Cons(10,
            Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

リスト 15-17: 3 番目のリストの所有権を共有しようとする Box<T>を使った 2 つのリストを存在させることはできないとデモする

このコードをコンパイルすると、こんなエラーが出ます:

```
error[E0382]: use of moved value: `a`
  --> src/main.rs:13:30
   |
12 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
13 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move
   |
   = note: move occurs because `a` has type `List`, which does not implement
   the `Copy` trait
```

Cons列挙子は、保持しているデータを所有するので、bリストを作成する時に、aが bにムーブされ、bが aを所有します。それから cを作る際に再度 aを使用しようとすると、aはムーブ済みなので、できないわけです。

Cons定義を代わりに参照を保持するように変更することもできますが、そうしたら、ライフタイム引数を指定しなければなりません。ライフタイム引数を指定することで、リストの各要素が最低でもリスト全体と同じ期間だけ生きることを指定することになります。例えば、借用精査機は let a = Cons(10, &Nil);をコンパイルさせてくれません。一時的な Nil値が、aが参照を得られるより前にドロップされてしまうからです。

代わりに、Listの定義をリスト 15-18 のように、Box<T>の箇所に Rc<T>を使うように変更します。これで各 Cons列挙子は、値と Listを指す Rc<T>を保持するようになりました。bを作る際、aの所有権を奪うのではなく、aが保持している Rc<List>をクローンします。それによって、参照の数が 1 から 2 に増え、aと bにその Rc<List>にあるデータの所有権を共有させます。また、cを生成する際にも aをクローンするので、参照の数は 2 から 3 になります。Rc::cloneを呼ぶ度に、Rc<List>内のデータの参照カウントが増え、参照が 0 にならない限りデータは片付けられません。

ファイル名: src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
```

```
}

use List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

リスト 15-18: Rc<T>を使用する Listの定義

初期化処理に含まれていないので、use文を追加して Rc<T>をスコープに導入する必要があります。mainで 5 と 10 を保持するリストを作成し、aの新しい Rc<List>に格納しています。それから、bと cを作成する際に、Rc::clone関数を呼び出し、引数として aの Rc<List>への参照を渡しています。

Rc::clone(&a)ではなく、a.clone()を呼ぶこともできますが、Rust のしきたりは、この場合 Rc::cloneを使うことです。Rc::cloneの実装は、多くの型の clone実装がするように、全てのデータのディープコピーをすることではありません。Rc::cloneの呼び出しは、参照カウントをインクリメントするだけであり、時間はかかりません。データのディープコピーは時間がかかることもあります。参照カウントに Rc::cloneを使うことで、視覚的にディープコピーをする類のクローンと参照カウントを増やす種類のクローンを区別することができます。コード内でパフォーマンスの問題を探す際、ディープコピーのクローンだけを考慮し、Rc::cloneの呼び出しを無視できるのです。

18.4.2　Rc<T>をクローンすると、参照カウントが増える

aの Rc<List>への参照を作ったりドロップする毎に参照カウントが変化するのが確かめられるように、リスト 15-18 の動く例を変更しましょう。

リスト 15-19 で、リスト cを囲む内側のスコープができるよう mainを変更します; そうすれば、cがスコープを抜けるときに参照カウントがどう変化するか確認できます。

ファイル名: src/main.rs

```
# enum List {
#     Cons(i32, Rc<List>),
#     Nil,
# }
#
# use List::{Cons, Nil};
# use std::rc::Rc;
#
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    // a生成後のカウント = {}
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    // b生成後のカウント = {}
    println!("count after creating b = {}", Rc::strong_count(&a));
```

```
16      {
17          let c = Cons(4, Rc::clone(&a));
18          // c生成後のカウント = {}
19          println!("count after creating c = {}", Rc::strong_count(&a));
20      }
21      // cがスコープを抜けた後のカウント = {}
22      println!("count after c goes out of scope = {}", Rc::strong_count(&a));
23  }
```

リスト 15-19: 参照カウントを出力する

プログラム内で参照カウントが変更される度に、参照カウントを出力します。参照カウントは、Rc::strong_count関数を呼び出すことで得られます。Rc<T>型には weak_countもあるので、この関数は countではなく strong_countと命名されています; weak_countの使用目的は、「循環参照を回避する」節で確かめます。

このコードは、以下の出力をします:

```
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

aの Rc<List>は最初 1 という参照カウントであることがわかります; そして、cloneを呼び出す度に、カウントは 1 ずつ上がります。cがスコープを抜けると、カウントは 1 下がります。参照カウントを増やすのに、Rc::cloneを呼ばなければいけなかったみたいに参照カウントを減らすのに関数を呼び出す必要はありません: Rc<T>値がスコープを抜けるときに Dropトレイトの実装が自動的に参照カウントを減らします。

この例ではわからないことは、bそして aが、mainの終端でスコープを抜ける時に、カウントが 0 になり、その時点で Rc<List>が完全に片付けられることです。Rc<T>で単独の値に複数の所有者を持たせることができ、所有者のいずれも存在している限り、値が有効であり続けることをカウントは保証します。

不変参照経由で、Rc<T>は読み取り専用にプログラムの複数箇所間でデータを共有させてくれます。Rc<T>が複数の可変参照を存在させることも許可してくれたら、第 4 章で議論した借用ルールの 1 つを侵害する恐れがあります: 同じ場所への複数の可変借用は、データ競合や矛盾を引き起こすことがあるのです。しかし、データを可変化する能力はとても有用です！ 次の節では、内部可変性パターンと、Rc<T>と絡めて使用してこの不変性制限を手がけられる RefCell<T>型について議論します。

18.5　RefCell<T>と内部可変性パターン

内部可変性は、そのデータへの不変参照がある時でさえもデータを可変化できる Rust でのデザインパターンです: 普通、この行動は借用ルールにより許可されません。データを可変化するために、このパターンは、データ構造内で unsafeコードを使用して、可変性と借用を支配する Rust の通常のルールを捻じ曲げています。まだ、unsafe コードについては講義していません; 第 19 章で行います。

たとえ、コンパイラが保証できなくても、借用ルールに実行時に従うことが保証できる時、内部可変性パターンを使用した型を使用できます。関係する unsafeコードはそうしたら、安全な API にラップされ、外側の型は、不変です。

内部可変性パターンに従う RefCell<T>型を眺めてこの概念を探求しましょう。

18.5.1　RefCell<T>で実行時に借用ルールを強制する

Rc<T>と異なり、RefCell<T>型は、保持するデータに対して単独の所有権を表します。では、どうして RefCell<T>が Box<T>のような型と異なるのでしょうか？ 第 4 章で学んだ借用ルールを思い出してください:

- いかなる時も (以下の両方ではなく、)1 つの可変参照かいくつもの不変参照のどちらかが可能になる
- 参照は常に有効でなければならない。

参照と Box<T>では、借用ルールの不変条件は、コンパイル時に強制されています。RefCell<T>では、これらの不変条件は、実行時に強制されます。参照でこれらの規則を破ったら、コンパイルエラーになりました。RefCell<T>でこれらの規則を破ったら、プログラムはパニックし、終了します。

コンパイル時に借用ルールを精査することの利点は、エラーが開発過程の早い段階で捕捉されることと、あらかじめ全ての分析が終わるので、実行パフォーマンスへの影響がないことです。それらの理由により、多くの場合でコンパイル時に借用ルールを精査することが最善の選択肢であり、これが Rust の規定になっているのです。

借用ルールを実行時に代わりに精査する利点は、コンパイル時の精査では許容されない特定のメモリ安全な筋書きが許容されることです。Rust コンパイラのような静的解析は、本質的に保守的です。コードの特性には、コードを解析するだけでは検知できないものもあります: 最も有名な例は停止性問題であり、この本の範疇を超えていますが、調べると面白い話題です。

不可能な分析もあるので、Rust のコンパイラが、コードが所有権ルールに応じていると確証を得られない場合、正しいプログラムを拒否する可能性があります; このように、保守的なのです。コンパイラが不正なプログラムを受け入れたら、ユーザは、コンパイラが行う保証を信じることはできなくなるでしょう。しかしながら、コンパイラが正当なプログラムを拒否するのなら、プログラマは不便に思うでしょうが、悲劇的なことは何も起こり得ません。コードが借用ルールに従っていると確証を得られる時に RefCell<T>型は有用ですが、コンパイラはそれを理解し、保証できません。

Rc<T>と類似して、RefCell<T>もシングルスレッドシナリオで使用するためのものであり、試しにマルチスレッドの文脈で使ってみようとすると、コンパイルエラーを出します。RefCell<T>の機能をマルチスレッドのプログラムで得る方法については、第 16 章で語ります。

こちらに Box<T>, Rc<T>, RefCell<T>を選択する理由を要約しておきます:

- Rc<T>は、同じデータに複数の所有者を持たせてくれる; Box<T>と RefCell<T>は単独の所有者。
- Box<T>は、不変または可変借用をコンパイル時に精査してくれる; Rc<T>は不変借用のみをコン

パイル時に精査してくれる; RefCell<T>は、不変または可変借用を実行時に精査してくれる。

- RefCell<T>は実行時に精査される可変借用を許可するので、RefCell<T>が不変でも、RefCell<T>内の値を可変化できる。

不変な値の中の値を可変化することは、**内部可変性**パターンです。内部可変性が有用になる場面を見て、それが可能になる方法を調査しましょう。

18.5.2　内部可変性: 不変値への可変借用

借用ルールの結果は、不変値がある時、可変で借用することはできないということです。例えば、このコードはコンパイルできません:

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

このコードをコンパイルしようとしたら、以下のようなエラーが出るでしょう:

```
error[E0596]: cannot borrow immutable local variable `x` as mutable
(エラー: ローカル変数`x`を可変で借用することはできません)
 --> src/main.rs:3:18
  |
2 |     let x = 5;
  |         - consider changing this to `mut x`
3 |     let y = &mut x;
  |                  ^ cannot borrow mutably
```

ですが、メソッド内で値が自身を可変化するけれども、他のコードにとっては、不変に見えることが有用な場面もあります。その値のメソッドの外のコードは、その値を可変化することはできないでしょう。RefCell<T>を使うことは、内部可変性を取得する能力を得る 1 つの方法です。しかし、RefCell<T>は借用ルールを完全に回避するものではありません: コンパイラの借用精査機は、内部可変性を許可し、借用ルールは代わりに実行時に精査されます。この規則を侵害したら、コンパイルエラーではなく panic!になるでしょう。

RefCell<T>を使用して不変値を可変化する実践的な例に取り組み、それが役に立つ理由を確認しましょう。

18.5.2.1　内部可変性のユースケース: モックオブジェクト

テストダブルは、テスト中に別の型の代わりに使用される型の一般的なプログラミングの概念です。**モックオブジェクト**は、テスト中に起きることを記録するテストダブルの特定の型なので、正しい動作が起きたことをアサートできます。

> 編注: テストダブルとは、ソフトウェアテストにおいて、テスト対象が依存しているコンポーネントを置き換える代用品のこと

Rust には、他の言語でいうオブジェクトは存在せず、また、他の言語のように標準ライブラリにモックオブジェクトの機能が組み込まれてもいません。ですが、同じ目的をモックオブジェクトとして提供する構造体を作成することは確かにできます。

以下が、テストを行う筋書きです: 値を最大値に対して追跡し、現在値が最大値に近い程度に基づいてメッセージを送信するライブラリを作成します。このライブラリは、ユーザが行うことのできる API コールの数の割り当てを追跡するのに使用することができるでしょう。

作成するライブラリは、値がどれくらい最大に近いかと、いつどんなメッセージになるべきかを追いかける機能を提供するだけです。このライブラリを使用するアプリケーションは、メッセージを送信する機構を提供すると期待されるでしょう: アプリケーションは、アプリケーションにメッセージを置いたり、メールを送ったり、テキストメッセージを送るなどできるでしょう。ライブラリはその詳細を知る必要はありません。必要なのは、提供する `Messenger`と呼ばれるトレイトを実装している何かなのです。リスト 15-20 は、ライブラリのコードを示しています:

ファイル名: src/lib.rs

```
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: 'a + Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
    where T: Messenger {
    pub fn new(messenger: &T, max: usize) -> LimitTracker<T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 0.75 && percentage_of_max < 0.9 {
            // 警告: 割り当ての75%以上を使用してしまいました
            self.messenger.send("Warning: You've used up over 75% of your quota!");
        } else if percentage_of_max >= 0.9 && percentage_of_max < 1.0 {
            // 切迫した警告: 割り当ての90%以上を使用してしまいました
            self.messenger.send("Urgent warning: You've used up over 90% of your quota!
    ");
        } else if percentage_of_max >= 1.0 {
            // エラー: 割り当てを超えています
```

```
            self.messenger.send("Error: You are over your quota!");
        }
    }
}
```

リスト 15-20: 値が最大値にどれくらい近いかを追跡し、任意のレベルの時に警告するライブラリ

このコードの重要な部分の 1 つは、Messengerトレイトには、selfへの不変参照とメッセージのテキストを取る sendというメソッドが 1 つあることです。これが、モックオブジェクトが持つ必要のあるインターフェイスなのです。もう 1 つの重要な部分は、LimitTrackerの set_valueメソッドの振る舞いをテストしたいということです。value引数に渡すものを変えることができますが、set_valueはアサートを行えるものは何も返してくれません。LimitTrackerを Messengerトレイトを実装する何かと、maxの特定の値で生成したら、valueに異なる数値を渡した時にメッセンジャーは適切なメッセージを送ると指示されると言えるようになりたいです。

sendを呼び出す時にメールやテキストメッセージを送る代わりに送ると指示されたメッセージを追跡するだけのモックオブジェクトが必要です。モックオブジェクトの新規インスタンスを生成し、モックオブジェクトを使用する LimitTrackerを生成し、LimitTrackerの set_valueを呼び出し、それからモックオブジェクトに期待しているメッセージがあることを確認できます。リスト 15-21 は、それだけをするモックオブジェクトを実装しようとするところを示しますが、借用精査機が許可してくません:

ファイル名: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages: vec![] }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);
```

```
27
28          assert_eq!(mock_messenger.sent_messages.len(), 1);
29      }
30  }
```

リスト 15-21: 借用精査機が許可してくれない MockMessangerを実装しようとする

このテストコードは Stringの Vecで送信すると指示されたメッセージを追跡する sent_messages フィールドのある MockMessenger構造体を定義しています。また、空のメッセージリストから始まる新しい MockMessenger値を作るのを便利にしてくれる関連関数の newも定義しています。それから MockMessengerに Messengerトレイトを実装しているので、LimitTrackerに MockMessenger を与えられます。sendメソッドの定義で引数として渡されたメッセージを取り、sent_messagesの MockMessengerリストに格納しています。

テストでは、max値の 75% 以上になる何かに valueをセットしろと LimitTrackerが指示される時に起きることをテストしています。まず、新しい MockMessengerを生成し、空のメッセージリストから始まります。そして、新しい LimitTrackerを生成し、新しい MockMessengerの参照と 100 という max値を与えます。LimitTrackerの set_valueメソッドは 80 という値で呼び出し、これは 100 の 75% を上回っています。そして、MockMessengerが追いかけているメッセージのリストが今は 1 つのメッセージを含んでいるはずとアサートします。

ところが、以下のようにこのテストには 1 つ問題があります:

```
error[E0596]: cannot borrow immutable field `self.sent_messages` as mutable
(エラー: 不変なフィールド`self.sent_messages`を可変で借用できません)
  --> src/lib.rs:52:13
   |
51 |         fn send(&self, message: &str) {
   |                 ----- use `&mut self` here to make mutable
52 |             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^ cannot mutably borrow immutable field
```

sendメソッドは selfへの不変参照を取るので、MockMessengerを変更してメッセージを追跡できないのです。代わりに&mut selfを使用するというエラーテキストからの提言を選ぶこともできないのです。そうしたら、sendのシグニチャが、Messengerトレイト定義のシグニチャと一致しなくなるからです (気軽に試してエラーメッセージを確認してください)。

これは、内部可変性が役に立つ場面なのです！ sent_messagesを RefCell<T>内部に格納し、そして sendメッセージは、sent_messagesを変更して見かけたメッセージを格納できるようになるでしょう。リスト 15-22 は、それがどんな感じかを示しています:

ファイル名: src/lib.rs

```
1  #[cfg(test)]
2  mod tests {
3      use super::*;
4      use std::cell::RefCell;
5
6      struct MockMessenger {
```

```
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages: RefCell::new(vec![]) }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--
#         let mock_messenger = MockMessenger::new();
#         let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);
#         limit_tracker.set_value(75);

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

リスト 15-22: 外側の値は不変と考えられる一方で RefCell<T>で内部の値を可変化する

さて、sent_messagesフィールドは、Vec<String>ではなく、型 RefCell<Vec<String>>になりました。new関数で、空のベクタの周りに RefCell<Vec<String>>を新しく作成しています。

sendメソッドの実装については、最初の引数はそれでも selfへの不変借用で、トレイト定義と合致しています。RefCell<Vec<String>>の borrow_mutを self.sent_messagesに呼び出し、RefCell<Vec<String>>の中の値への可変参照を得て、これはベクタになります。それからベクタへの可変参照に pushを呼び出して、テスト中に送られるメッセージを追跡しています。

行わなければならない最後の変更は、アサート内部にあります: 内部のベクタにある要素の数を確認するため、RefCell<Vec<String>>に borrowを呼び出し、ベクタへの不変参照を得ています。

RefCell<T>の使用法を見かけたので、動作法を深掘りしましょう！

18.5.2.2　RefCell<T>で実行時に借用を追いかける

不変および可変参照を作成する時、それぞれ&と&mut記法を使用します。RefCell<T>では、borrow と borrow_mutメソッドを使用し、これらは RefCell<T>に所属する安全な API の一部です。borrowメソッドは、スマートポインタ型の Ref<T>を返し、borrow_mutはスマートポインタ型の RefMut<T>を返します。どちらの型も Derefを実装しているので、普通の参照のように扱うことができます。

RefCell<T>は、現在活動中の Ref<T>と RefMut<T>スマートポインタの数を追いかけます。borrowを呼び出す度に、RefCell<T>は活動中の不変参照の数を増やします。Ref<T>の値がスコープを抜けた

ら、不変参照の数は 1 下がります。コンパイル時の借用ルールと全く同じように、RefCell<T>はいかなる時も、複数の不変借用または 1 つの可変借用を持たせてくれるのです。

これらの規則を侵害しようとすれば、参照のようにコンパイルエラーになるのではなく、RefCell<T>の実装は実行時にパニックするでしょう。リスト 15-23 は、リスト 15-22 の send実装に対する変更を示しています。同じスコープで 2 つの可変借用が活動するようわざと生成し、RefCell<T>が実行時にこれをすることを阻止してくれるところを具体化しています。

ファイル名: src/lib.rs

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut();

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));
    }
}
```

リスト 15-23: 同じスコープで 2 つの可変参照を生成して RefCell<T>がパニックすることを確かめる

borrow_mutから返ってきた RefMut<T>スマートポインタに対して変数 one_borrowを生成しています。そして、同様にして変数 two_borrowにも別の可変借用を生成しています。これにより同じスコープで 2 つの可変参照ができ、これは許可されないことです。このテストを自分のライブラリ用に走らせると、リスト 15-23 のコードはエラーなくコンパイルできますが、テストは失敗するでしょう:

```
---- tests::it_sends_an_over_75_percent_warning_message stdout ----
    thread 'tests::it_sends_an_over_75_percent_warning_message' panicked at
'already borrowed: BorrowMutError', src/libcore/result.rs:906:4
  (スレッド'tests::it_sends_an_over_75_percent_warning_message'は、
'すでに借用されています: BorrowMutError', src/libcore/result.rs:906:4でパニックしました
    )
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

コードは、already borrowed: BorrowMutErrorというメッセージとともにパニックしたことに気付いてください。このようにして RefCell<T>は実行時に借用ルールの侵害を扱うのです。

コンパイル時ではなく実行時に借用エラーをキャッチするということは、開発過程の遅い段階でコードのミスを発見し、コードをプロダクションにデプロイする時まで発見しないことを意味します。また、コンパイル時ではなく、実行時に借用を追いかける結果として、少し実行時にパフォーマンスを犠牲にするでしょう。しかしながら、RefCell<T>を使うことで不変値のみが許可される文脈で使用しつつ、自身を変更して見かけたメッセージを追跡するモックオブジェクトを書くことを可能にしてくれます。代償にも関わらず RefCell<T>を使用して、普通の参照よりも多くの機能を得ることができるわけです。

18.5.3　Rc<T>と RefCell<T>を組み合わせることで可変なデータに複数の所有者を持たせる

RefCell<T>の一般的な使用法は、Rc<T>と組み合わせることにあります。Rc<T>は何らかのデータに複数の所有者を持たせてくれるけれども、そのデータに不変のアクセスしかさせてくれないことを思い出してください。RefCell<T>を抱える Rc<T>があれば、複数の所有者を持ちそして、可変化できる値を得ることができるのです。

例を挙げれば、Rc<T>を使用して複数のリストに別のリストの所有権を共有させたリスト 15-18 のコンスリストの例を思い出してください。Rc<T>は不変値だけを抱えるので、一旦生成したら、リストの値はどれも変更できません。RefCell<T>を含めて、リストの値を変更する能力を得ましょう。RefCell<T>を Cons定義で使用することで、リスト全てに格納されている値を変更できることをリスト 15-24 は示しています:

ファイル名: src/main.rs

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

リスト 15-24: Rc<RefCell<i32>>で可変化できる Listを生成する

Rc<RefCell<i32>>のインスタンスの値を生成し、valueという名前の変数に格納しているので、直接後ほどアクセスすることができます。そして、aに valueを持つ Cons列挙子で Listを生成しています。valueから aに所有権を移したり、aが valueから借用するのではなく、aと valueどちらにも中の 5の値の所有権を持たせるよう、valueをクローンする必要があります。

リスト aを Rc<T>に包んでいるので、リスト bと cを生成する時に、どちらも aを参照できます。リスト 15-18 ではそうしていました。

a、b、cのリストを作成した後、valueの値に 10 を足しています。これを valueの borrow_mutを呼び出すことで行い、これは、第 5 章で議論した自動参照外し機能 (「->演算子はどこに行ったの？」節をご覧ください) を使用して、Rc<T>を内部の RefCell<T>値に参照外ししています。borrow_mutメソッドは、RefMut<T>スマートポインタを返し、それに対して参照外し演算子を使用し、中の値を変更します。

a、b、cを出力すると、全て 5 ではなく、変更された 15 という値になっていることがわかります。

```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

このテクニックは非常に綺麗です！ RefCell<T>を使用することで表面上は不変な List値を持てます。しかし、内部可変性へのアクセスを提供する RefCell<T>のメソッドを使用できるので、必要な時にはデータを変更できます。借用ルールを実行時に精査することでデータ競合を防ぎ、時としてデータ構造でちょっとのスピードを犠牲にこの柔軟性を得るのは価値があります。

標準ライブラリには、Cell<T>などの内部可変性を提供する他の型もあり、この型は、内部値への参照を与える代わりに、値は Cell<T>の内部や外部へコピーされる点を除き似ています。また Mutex<T>もあり、これはスレッド間で使用するのが安全な内部可変性を提供します; 第 16 章で使用することを議論しましょう。標準ライブラリのドキュメンテーションをチェックして、これらの型の違いを詳しく知ってください。

18.6　循環参照してメモリをリークすることもある

Rust のメモリ安全保証により誤って絶対に片付けられることのないメモリ (**メモリリーク**として知られています) を生成してしまうことが困難にはなりますが、不可能にはなりません。コンパイル時にデータ競合を防ぐのと同じようにメモリリークを完全に回避することは、Rust の保証の一つではなく、メモリリークは Rust においてはメモリ安全であることを意味します。Rust では、Rc<T>と RefCell<T>を使用してメモリリークを許可するとわかります: 要素がお互いに循環して参照する参照を生成することも可能ということです。循環の各要素の参照カウントが絶対に 0 にならないので、これはメモリリークを起こし、値は絶対にドロップされません。

18.6.1　循環参照させる

リスト 15-25 の Listenum の定義と tailメソッドから始めて、循環参照が起こる可能性のある方法とその回避策を見ましょう:

ファイル名: src/main.rs

```
# fn main() {}
use std::rc::Rc;
```

```
use std::cell::RefCell;
use List::{Cons, Nil};

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match *self {
            Cons(_, ref item) => Some(item),
            Nil => None,
        }
    }
}
```

リスト 15-25: Cons列挙子が参照しているものを変更できるように RefCell<T>を抱えているコンスリストの定義

リスト 15-5 の List定義の別バリエーションを使用しています。Cons列挙子の 2 番目の要素はこれで RefCell<Rc<List>>になり、リスト 15-24 のように i32値を変更する能力があるのではなく、Cons列挙子が指している List値の先を変えたいということです。また、tailメソッドを追加して Cons列挙子があるときに 2 番目の要素にアクセスするのが便利になるようにしています。

リスト 15-26 でリスト 15-25 の定義を使用する main関数を追加しています。このコードは、aにリストを、bに aのリストを指すリストを作成します。それから aのリストを変更して bを指し、循環参照させます。その流れの中に過程のいろんな場所での参照カウントを示す println!文が存在しています。

ファイル名: src/main.rs

```
# use List::{Cons, Nil};
# use std::rc::Rc;
# use std::cell::RefCell;
# #[derive(Debug)]
# enum List {
#     Cons(i32, RefCell<Rc<List>>),
#     Nil,
# }
#
# impl List {
#     fn tail(&self) -> Option<&RefCell<Rc<List>>> {
#         match *self {
#             Cons(_, ref item) => Some(item),
#             Nil => None,
#         }
#     }
# }
#
fn main() {
```

```
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    // aの最初の参照カウント = {}
    println!("a initial rc count = {}", Rc::strong_count(&a));
    // aの次の要素は = {:?}
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    // b作成後のaの参照カウント = {}
    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    // bの最初の参照カウント = {}
    println!("b initial rc count = {}", Rc::strong_count(&b));
    // bの次の要素 = {:?}
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    // aを変更後のbの参照カウント = {}
    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    // aを変更後のaの参照カウント = {}
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // 次の行のコメントを外して循環していると確認してください; スタックオーバーフローします
    // println!("a next item = {:?}", a.tail());        // aの次の要素 = {:?}
}
```

リスト 15-26: 2 つの `List`値がお互いを指して循環参照する

最初のリストが `5`, `Nil`の `List`値を保持する `Rc<List>`インスタンスを変数 `a`に生成します。そして、値 10 と `a`のリストを指す別の `List`値を保持する `Rc<List>`インスタンスを変数 `b`に生成します。

`a`が `Nil`ではなく `b`を指すように変更して、循環させます。`tail`メソッドを使用して、`a`の `RefCell<Rc<List>>`への参照を得ることで循環させて、この参照は変数 `link`に配置します。それから `RefCell<Rc<List>>`の `borrow_mut`メソッドを使用して中の値を `Nil`値を持つ `Rc<List>`から、`b`の `Rc<List>`に変更します。

最後の `println!`を今だけコメントアウトしたまま、このコードを実行すると、こんな出力が得られます:

```
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
```

aのリストを bを指すように変更した後の aと bの Rc<List>インスタンスの参照カウントは 2 です。mainの終端で、コンパイラはまず bをドロップしようとし、aと bの各 Rc<List>インスタンスのカウントを 1 減らします。

しかしながら、それでも aは bにあった Rc<List>を参照しているので、その Rc<List>のカウントは 0 ではなく 1 になり、その Rc<List>がヒープに確保していたメモリはドロップされません。メモリはただ、カウント 1 のままそこに永遠に居座るのです。この循環参照を可視化するために、図 15-4 に図式を作成しました:

図 15-4: お互いを指すリスト aと bの循環参照

最後の println!のコメントを外してプログラムを実行したら、aが bを指して、bが aを指してと、スタックがオーバーフローするまでコンパイラはこの循環を出力しようとするでしょう。

この場合、循環参照を作る直後にプログラムは終了します。この循環の結果は、それほど悲壮なものではありません。しかしながら、より複雑なプログラムが多くのメモリを循環で確保し長い間その状態を保ったら、プログラムは必要以上のメモリを使用し、使用可能なメモリを枯渇させてシステムを参らせてしまう可能性があります。

循環参照は簡単にできることではありませんが、不可能というわけでもありません。Rc<T>値を含む RefCell<T>値があるなどの内部可変性と参照カウントのある型がネストして組み合わさっていたら、循環していないことを保証しなければなりません; コンパイラがそれを捕捉することを信頼できないのです。循環参照をするのは、自動化テストやコードレビューなどの他のソフトウェア開発手段を使用して最小化すべきプログラム上のロジックバグでしょう。

循環参照を回避する別の解決策は、ある参照は所有権を表現して他の参照はしないというようにデータ構造を再構成することです。結果として所有権のある関係と所有権のない関係からなる循環ができ、所有権のある関係だけが値がドロップされうるかどうかに影響します。リスト 15-25 では、常に Cons列挙子にリストを所有してほしいので、データ構造を再構成することはできません。親ノードと子ノードからなるグラフを使った例に目を向けて、どんな時に所有権のない関係が循環参照を回避するのに適切な方法になるか確認しましょう。

18.6.2　循環参照を回避する: Rc<T>を Weak<T>に変換する

ここまで、Rc::cloneを呼び出すと Rc<T>インスタンスの strong_countが増えることと、strong_countが 0 になった時に Rc<T>インスタンスは片付けられることをデモしてきました。Rc::downgradeを呼び出し、Rc<T>への参照を渡すことで、Rc<T>インスタンス内部の値への**弱い参照** (weak reference) を作ることもできます。Rc::downgradeを呼び出すと、型 Weak<T>のスマートポインタが得られます。Rc<T>インスタンスの strong_countを 1 増やす代わりに、Rc::downgradeを呼び出すと、weak_countが 1 増えます。strong_count同様、Rc<T>型は weak_countを使用して、幾つの Weak<T>参照が存在しているかを追跡します。違いは、Rc<T>が片付けられるのに、weak_countが 0 である必要はないということです。

強い参照は、Rc<T>インスタンスの所有権を共有する方法です。弱い参照は、所有権関係を表現しま

せん。ひとたび、関係する値の強い参照カウントが 0 になれば、弱い参照が関わる循環はなんでも破壊されるので、循環参照にはなりません。

Weak<T>が参照する値はドロップされてしまう可能性があるので、Weak<T>が指す値に何かをするには、値がまだ存在することを確認しなければなりません。Weak<T>の upgradeメソッドを呼び出すことでこれをしてください。このメソッドは Option<Rc<T>>を返します。Rc<T>値がまだドロップされていなければ、Someの結果が、Rc<T>値がドロップ済みなら、Noneの結果が得られます。upgradeが Option<T>を返すので、コンパイラは、Someケースと Noneケースが扱われていることを確かめてくれ、無効なポインタは存在しません。

例として、要素が次の要素を知っているだけのリストを使うのではなく、要素が子要素と親要素を知っている木を作りましょう。

18.6.2.1　木データ構造を作る: 子ノードのある Node

手始めに子ノードを知っているノードのある木を構成します。独自の i32値と子供の Node値への参照を抱える Nodeという構造体を作ります:

ファイル名: src/main.rs

```
use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}
```

Nodeに子供を所有してほしく、木の各 Nodeに直接アクセスできるよう、その所有権を変数と共有したいです。こうするために、Vec<T>要素を型 Rc<Node>の値になるよう定義しています。どのノードが他のノードの子供になるかも変更したいので、Vec<Rc<Node>>の周りの childrenを RefCell<T>にしています。

次にこの構造体定義を使って値 3 と子供なしの leafという 1 つの Nodeインスタンスと、値 5 と leafを子要素の一つとして持つ branchという別のインスタンスを作成します。リスト 15-27 のようにね:

ファイル名: src/main.rs

```
# use std::rc::Rc;
# use std::cell::RefCell;
#
# #[derive(Debug)]
# struct Node {
#     value: i32,
#     children: RefCell<Vec<Rc<Node>>>,
# }
#
fn main() {
```

```
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

リスト 15-27: 子供なしの leafノードと leafを子要素に持つ branchノードを作る

leafの Rc<Node>をクローンし、branchに格納しているので、leafの Nodeは leafと branchという 2 つの所有者を持つことになります。branch.childrenを通して branchから leafへ辿ることはできるものの、leafから branchへ辿る方法はありません。理由は、leafには branchへの参照がなく、関係していることを知らないからです。leafに branchが親であることを知ってほしいです。次はそれを行います。

18.6.2.2　子供から親に参照を追加する

子供に親の存在を気付かせるために、Node構造体定義に parentフィールドを追加する必要があります。parentの型を決める際に困ったことになります。Rc<T>を含むことができないのはわかります。そうしたら、leaf.parentが branchを指し、branch.childrenが leafを指して循環参照になり、strong_count値が絶対に 0 にならなくなってしまうからです。

この関係を別の方法で捉えると、親ノードは子供を所有すべきです: 親ノードがドロップされたら、子ノードもドロップされるべきなのです。ですが、子供は親を所有するべきではありません: 子ノードをドロップしても、親はまだ存在するべきです。弱い参照を使う場面ですね！

従って、Rc<T>の代わりに parentの型を Weak<T>を使ったもの、具体的には RefCell<Weak<Node>>にします。さあ、Node構造体定義はこんな見た目になりました:

ファイル名: src/main.rs

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

ノードは親ノードを参照できるものの、所有はしないでしょう。リスト 15-28 で、leafノードが親の branchを参照できるよう、この新しい定義を使用するように mainを更新します:

ファイル名: src/main.rs

```
# use std::rc::{Rc, Weak};
# use std::cell::RefCell;
#
# #[derive(Debug)]
# struct Node {
#     value: i32,
#     parent: RefCell<Weak<Node>>,
#     children: RefCell<Vec<Rc<Node>>>,
# }
#
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    // leafの親 = {:?}
    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

リスト 15-28: 親ノードの branchへの弱い参照がある leafノード

leafノードを作成することは、parentフィールドの例外を除いてリスト 15-27 での leafノードの作成法に似ています: leafは親なしで始まるので、新しく空の Weak<Node>参照インスタンスを作ります。

この時点で upgradeメソッドを使用して leafの親への参照を得ようとすると、None値になります。このことは、最初の println!文の出力でわかります:

```
leaf parent = None
```

branchノードを作る際、branchには親ノードがないので、こちらも parentフィールドには新しい Weak<Node>参照が入ります。それでも、leafは branchの子供になっています。一旦 branchに Nodeインスタンスができたら、leafを変更して親への Weak<Node>参照を与えることができます。leafの parentフィールドには、RefCell<Weak<Node>>の borrow_mutメソッドを使用して、それから Rc::downgrade関数を使用して、branchの Rc<Node>から branchへの Weak<Node>参照を作ります。

再度 leafの親を出力すると、今度は branchを保持する Some列挙子が得られます: これで leafが親にアクセスできるようになったのです！ leafを出力すると、リスト 15-26 で起こっていたような最

終的にスタックオーバーフローに行き着く循環を避けることもできます; Weak<Node>参照は、(Weak)と出力されます:

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },
children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak) },
children: RefCell { value: [] } }] } })
```

無限の出力が欠けているということは、このコードは循環参照しないことを示唆します。このことは、Rc::strong_countと Rc::weak_countを呼び出すことで得られる値を見てもわかります。

18.6.2.3　strong_count と weak_count への変更を可視化する

新しい内部スコープを作り、branchの作成をそのスコープに移動することで、Rc<Node>インスタンスの strong_countと weak_count値がどう変化するかを眺めましょう。そうすることで、branchが作成され、それからスコープを抜けてドロップされる時に起こることが確認できます。変更は、リスト 15-29 に示してあります:

ファイル名: src/main.rs

```
# use std::rc::{Rc, Weak};
# use std::cell::RefCell;
#
# #[derive(Debug)]
# struct Node {
#     value: i32,
#     parent: RefCell<Weak<Node>>,
#     children: RefCell<Vec<Rc<Node>>>,
# }
#
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    // leafのstrong_count = {}, weak_count = {}
    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

```

```
        // branchのstrong_count = {}, weak_count = {}
        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf strong = {}, weak = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```

リスト 15-29: 内側のスコープで branchを作成し、強弱参照カウントを調査する

leaf作成後、その Rc<Node>の強カウントは 1、弱カウントは 0 になります。内側のスコープで branchを作成し、leafに紐付け、この時点でカウントを出力すると、branchの Rc<Node>の強カウントは 1、弱カウントも 1 になります (leaf.parentが Weak<Node>で branchを指しているため)。leafのカウントを出力すると、強カウントが 2 になっていることがわかります。branchが今は、branch.childrenに格納された leafの Rc<Node>のクローンを持っているからですが、それでも弱カウントは 0 でしょう。

内側のスコープが終わると、branchはスコープを抜け、Rc<Node>の強カウントは 0 に減るので、この Nodeはドロップされます。leaf.parentからの弱カウント 1 は、Nodeがドロップされるか否かには関係ないので、メモリリークはしないのです！

このスコープの終端以後に leafの親にアクセスしようとしたら、再び Noneが得られます。プログラムの終端で leafの Rc<Node>の強カウントは 1、弱カウントは 0 です。変数 leafが今では Rc<Node>への唯一の参照に再度なったからです。

カウントや値のドロップを管理するロジックは全て、Rc<T>や Weak<T>とその Dropトレイトの実装に組み込まれています。Nodeの定義で子供から親への関係は Weak<T>参照になるべきと指定することで、循環参照やメモリリークを引き起こさずに親ノードに子ノードを参照させたり、その逆を行うことができます。

18.7　まとめ

この章は、スマートポインタを使用して Rust が規定で普通の参照に対して行うのと異なる保証や代償を行う方法を講義しました。Box<T>型は、既知のサイズで、ヒープに確保されたデータを指しま

す。Rc<T>型は、ヒープのデータへの参照の数を追跡するので、データは複数の所有者を保有できます。内部可変性のある RefCell<T>型は、不変型が必要だけれども、その型の中の値を変更する必要がある時に使用できる型を与えてくれます; また、コンパイル時ではなく実行時に借用ルールを強制します。

Derefと Dropトレイトについても議論しましたね。これらは、スマートポインタの多くの機能を可能にしてくれます。メモリリークを引き起こす循環参照と Weak<T>でそれを回避する方法も探求しました。

この章で興味をそそられ、独自のスマートポインタを実装したくなったら、もっと役に立つ情報を求めて、“The Rustonomicon” をチェックしてください。

次は、Rust での非同期処理について語ります。もういくつか新しいスマートポインタについてさえも学ぶでしょう。

第 19 章

恐れるな！ 非同期処理

非同期処理を安全かつ効率的に扱うことは、Rust の別の主な目標です。非同期プログラミングは、プログラムの異なる部分が個別に実行することであり、並行プログラミングはプログラムの異なる部分が同時に実行することですが、多くのコンピュータが複数のプロセッサの利点を生かすようになるにつれ、重要度を増しています。歴史的に、これらの文脈で行うプログラミングは困難で、エラーが起きやすいものでした: Rust はこれを変えると期待されています。

当初、Rust チームは、メモリ安全性を期することと、非同期処理問題を回避することは、異なる方法で解決すべき 2 つの個別の挑戦だと考えていました。時間とともに、チームは、所有権と型システムは、メモリ安全性と非同期処理問題を管理する役に立つ一連の強力な道具であることを発見しました。所有権と型チェックに影響させることで、多くの非同期処理エラーは、実行時エラーではなくコンパイルエラーになります。故に、実行時に非同期処理のバグが起きた状況と全く同じ状況を再現しようと時間を浪費させるよりも、不正なコードはコンパイルを拒み、問題を説明するエラーを提示するでしょう。結果として、プロダクトになってから可能性があるというよりも、コードに取り組んでいる間に修正できます。Rust のこの方向性を**恐れるな！ 非同期処理**とニックネーム付けしました。これにより、潜在的なバグがなく、新しいバグを導入することなく簡単にリファクタリングできるコードを書くことができます。

> 注釈: 簡潔性のため、非同期または並行と述べることで正確を期するのではなく、多くの問題を**非同期**と割り切ってしまいます。この本がもし**非同期あるいは並行性**に関した本ならば、もっと特定していたでしょう。この章に対しては、**非同期**を使ったら、脳内で**非同期または並行性**と置き換えてください。

多くの言語は、自分が提供する非同期処理問題を扱う解決策について独断的です。例えば、Erlang には、メッセージ受け渡しの非同期処理に関する素晴らしい機能がありますが、スレッド間で状態を共有することに関しては、曖昧な方法しかありません。可能な解決策の一部のみをサポートすることは、高級言語にとっては合理的な施策です。なぜなら、高級言語は一部の制御を失う代わりに抽象化

することから恩恵を受けるからです。ところが、低級言語は、どんな場面でも最高のパフォーマンスで解決策を提供すると想定され、ハードウェアに関してほとんど抽象化はしません。そのため、Rust は、自分の状況と必要性に適した方法が何であれ、問題をモデル化するためのいろんな道具を備えています。

こちらが、この章で講義する話題です:

- スレッドを生成して、複数のコードを同時に走らせる方法
- チャンネルがスレッド間でメッセージを送る**メッセージ受け渡し**非同期処理
- 複数のスレッドが何らかのデータにアクセスする**状態共有**非同期処理
- Rust の非同期処理保証をユーザ定義型と標準ライブラリが提供する型にも及ばせる `Sync`と `Send`トレイト

19.1　スレッドを使用してコードを同時に走らせる

多くの現代の OS では、実行中のプログラムのコードは**プロセス**で走り、OS は同時に複数のプロセスを管理します。自分のプログラム内で、同時に実行する部分を作ることもできます。これらの個別の部分を走らせる機能を**スレッド**と呼びます。

プログラム内の計算を複数のスレッドに分けると、パフォーマンスが改善します。プログラムが同時に複数の作業をするからですが、複雑度も増します。スレッドは同時に走らせることができるので、異なるスレッドのコードが走る順番に関して、本来の保証はありません。これは問題を招きます。例えば:

- スレッドがデータやリソースに矛盾した順番でアクセスする競合状態
- 2 つのスレッドがお互いにもう一方が持っているリソースを使用し終わるのを待ち、両者が継続するのを防ぐデッドロック
- 特定の状況でのみ起き、再現や信頼して修正が困難なバグ

Rust は、スレッド使用のマイナスの効果を軽減しようとしていますが、それでも、マルチスレッドの文脈でのプログラミングは、注意深い思考とシングルスレッドで走るプログラムとは異なるコード構造を必要とします。

プログラミング言語によってスレッドはいくつかの方法で実装されています。多くの OS で、新規スレッドを生成する API が提供されています。言語が OS の API を呼び出してスレッドを生成するこのモデルを時に *1:1* と呼び、1 つの OS スレッドに対して 1 つの言語スレッドを意味します。

多くのプログラミング言語がスレッドの独自の特別な実装を提供しています。プログラミング言語が提供するスレッドは、**グリーン**スレッドとして知られ、このグリーンスレッドを使用する言語は、それを異なる数の OS スレッドの文脈で実行します。このため、グリーンスレッドのモデルは *M:N* モデルと呼ばれます: `M`個のグリーンスレッドに対して、`N`個の OS スレッドがあり、`M`と `N`は必ずしも同じ数字ではありません。

各モデルには、それだけの利点と妥協点があり、Rust にとって最も重要な妥協点は、ランタイムのサポートです。**ランタイム**は、混乱しやすい用語で文脈によって意味も変わります。

この文脈では、ランタイムとは、言語によって全てのバイナリに含まれるコードのことを意味します。言語によってこのコードの大小は決まりますが、非アセンブリ言語は全てある量の実行時コードを含みます。そのため、口語的に誰かが「ノーランタイム」と言ったら、「小さいランタイム」のことを意味することがしばしばあります。ランタイムが小さいと機能も少ないですが、バイナリのサイズも小さくなるという利点があり、その言語を他の言語とより多くの文脈で混ぜることが容易になります。多くの言語では、より多くの機能と引き換えにランタイムのサイズが膨れ上がるのは、受け入れられることですが、Rust にはほとんどゼロのランタイムが必要でパフォーマンスを維持するために C コードを呼び出せることを妥協できないのです。

M:N のグリーンスレッドモデルは、スレッドを管理するのにより大きな言語ランタイムが必要です。よって、Rust の標準ライブラリは、1:1 スレッドの実装のみを提供しています。Rust はそのような低級言語なので、例えば、どのスレッドがいつ走るか制御する方向性や、文脈切り替えの低コストを求めるオーバーヘッドと引き換えるなら、M:N スレッドの実装をしたクレートもあります。

今や Rust におけるスレッドを定義したので、標準ライブラリで提供されているスレッド関連の API の使用法を探求しましょう。

19.1.1　spawn で新規スレッドを生成する

新規スレッドを生成するには、`thread::spawn`関数を呼び出し、新規スレッドで走らせたいコードを含むクロージャ (クロージャについては第 13 章で語りました) を渡します。リスト 16-1 の例は、メインスレッドと新規スレッドからテキストを出力します:

ファイル名: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            // やあ！立ち上げたスレッドから数字{}だよ！
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        // メインスレッドから数字{}だよ！
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

リスト 16-1: メインスレッドが別のものを出力する間に新規スレッドを生成して何かを出力する

この関数では、実行が終わったかどうかにかかわらず、新しいスレッドは、メインスレッドが終了したら停止することに注意してください。このプログラムからの出力は毎回少々異なる可能性がありますが、だいたい以下のような感じでしょう:

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

`thread::sleep`を呼び出すと、少々の間、スレッドの実行を止め、違うスレッドを走らせることができます。スレッドは順番待ちをしますが、保証はありません: OS がスレッドのスケジュールを行う方法によります。この実行では、立ち上げられたスレッドの print 文がコードでは先に出現しているのに、メインスレッドがまず出力しています。また、立ち上げたスレッドには `i`が 9 になるまで出力するよう指示しているのに、メインスレッドが終了する前の 5 までしか到達していません。

このコードを実行してメインスレッドの出力しか目の当たりにできなかったり、オーバーラップがなければ、範囲の値を増やして OS がスレッド切り替えを行う機会を増やしてみてください。

19.1.2　join ハンドルで全スレッドの終了を待つ

リスト 16-1 のコードは、メインスレッドが終了するためにほとんどの場合、新規スレッドが未完で終わるだけでなく、新規スレッドが実行されるかどうかも保証できません。原因は、スレッドの実行順に保証がないからです。

`thread::spawn`の戻り値を変数に保存することで、立ち上げたスレッドが実行されなかったり、完全に実行されなかったりする問題を修正することができます。`thread:spawn`の戻り値の型は `JoinHandle`です。`JoinHandle`は、`join`メソッドを呼び出したときにスレッドの終了を待つ所有された値です。リスト 16-2 は、リスト 16-1 で生成したスレッドの `JoinHandle`を使用し、`join`を呼び出して、立ち上げたスレッドが、`main`が終了する前に完了することを確認する方法を示しています:

ファイル名: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
```

```
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

リスト 16-2: thread::spawnの JoinHandleを保存してスレッドが完了するのを保証する

ハンドルに対して joinを呼び出すと、現在実行中のスレッドをハンドルが表すスレッドが停止するまでブロックします。スレッドを**ブロック**するとは、そのスレッドが作業をしたり、終了することを防ぐことです。joinの呼び出しをメインスレッドの forループの後に配置したので、リスト 16-2 を実行すると、以下のような出力が出るはずです:

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

2 つのスレッドが代わる代わる実行されていますが、handle.join()呼び出しのためにメインスレッドは待機し、立ち上げたスレッドが終了するまで終わりません。

ですが、代わりに handle.join()を forループの前に移動したらどうなるのか確認しましょう。こんな感じに:

ファイル名: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
```

```
}
```

メインスレッドが、立ち上げたスレッドが終了するまで待ち、それから forループを実行するので、以下のように出力はもう混じりません:

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

どこで joinを呼ぶかのような小さな詳細が、スレッドが同時に走るかどうかに影響することもあります。

19.1.3　スレッドで move クロージャを使用する

moveクロージャは、thread::spawnとともによく使用されます。あるスレッドから別のスレッドにデータを使用させてくれるからです。

第 13 章で、クロージャの引数リストの前に moveキーワードを使用して、クロージャに環境で使用している値の所有権を強制的に奪わせることができると述べました。このテクニックは、あるスレッドから別のスレッドに値の所有権を移すために新しいスレッドを生成する際に特に有用です。

新規スレッドを立ち上げているので、クロージャに値をキャプチャすることについて語りましょう。

リスト 16-1 において、thread::spawnに渡したクロージャには引数がなかったことに気付いてください: 立ち上げたスレッドのコードでメインスレッドからのデータは何も使用していないのです。立ち上げたスレッドでメインスレッドのデータを使用するには、立ち上げるスレッドのクロージャは、必要な値をキャプチャしなければなりません。リスト 16-3 は、メインスレッドでベクタを生成し、立ち上げたスレッドで使用する試みを示しています。しかしながら、すぐにわかるように、これはまだ動きません:

ファイル名: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        // こちらがベクタ: {:?}
        println!("Here's a vector: {:?}", v);
```

```
    });

    handle.join().unwrap();
}
```

リスト 16-3: 別のスレッドでメインスレッドが生成したベクタを使用しようとする

クロージャは vを使用しているので、vをキャプチャし、環境の一部にしています。thread::spawn はこのクロージャを新しいスレッドで走らせるので、その新しいスレッド内で vにアクセスできるはずです。しかし、このコードをコンパイルすると、以下のようなエラーが出ます:

```
error[E0373]: closure may outlive the current function, but it borrows `v`,
which is owned by the current function
(エラー: クロージャは現在の関数よりも長生きするかもしれませんが、現在の関数が所有してい
    る
`v`を借用しています)
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
  |                                            - `v` is borrowed here
  |
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
(ヘルプ: `v`(や他の参照されている変数)の所有権をクロージャに奪わせるには、`move`キー
    ワードを使用してください)
  |
6 |     let handle = thread::spawn(move || {
  |                                ^^^^^^^
```

Rust は vのキャプチャ方法を**推論**し、println!は vへの参照のみを必要とするので、クロージャは、vを借用しようとします。ですが、問題があります: コンパイラには、立ち上げたスレッドがどのくらいの期間走るのかわからないので、vへの参照が常に有効であるか把握できないのです。

リスト 16-4 は、vへの参照がより有効でなさそうな筋書きです:

ファイル名: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    // いや～！
    drop(v); // oh no!

    handle.join().unwrap();
}
```

リスト 16-4: vをドロップするメインスレッドから vへの参照をキャプチャしようとするクロージャとスレッド

このコードを実行することが叶うのなら、立ち上げたスレッドが実行されることなく即座にバックグラウンドに置かれる可能性があります。立ち上げたスレッドは内部に vへの参照を保持していますが、メインスレッドは、第 15 章で議論した drop関数を使用して、即座に vをドロップしています。そして、立ち上げたスレッドが実行を開始する時には、vはもう有効ではなく、参照も不正になるのです。あちゃー！

リスト 16-3 のコンパイルエラーを修正するには、エラーメッセージのアドバイスを活用できます:

```
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ^^^^^^^
```

クロージャの前に moveキーワードを付することで、コンパイラに値を借用すべきと推論させるのではなく、クロージャに使用している値の所有権を強制的に奪わせます。リスト 16-5 に示したリスト 16-3 に対する変更は、コンパイルでき、意図通りに動きます:

ファイル名: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

リスト 16-5: moveキーワードを使用してクロージャに使用している値の所有権を強制的に奪わせる

moveクロージャを使用していたら、メインスレッドが dropを呼び出すリスト 16-4 のコードはどうなるのでしょうか？ moveで解決するのでしょうか？ 残念ながら、違います; リスト 16-4 が試みていることは別の理由によりできないので、違うエラーが出ます。クロージャに moveを付与したら、vをクロージャの環境にムーブするので、最早メインスレッドで dropを呼び出すことは叶わなくなるでしょう。代わりにこのようなコンパイルエラーが出るでしょう:

```
error[E0382]: use of moved value: `v`
(エラー: ムーブされた値の使用: `v`)
  --> src/main.rs:10:10
   |
6  |     let handle = thread::spawn(move || {
```

```
   |                                 ------- value moved (into closure) here
...
10 |     drop(v); // oh no!
   |          ^ value used here after move
   |
   = note: move occurs because `v` has type `std::vec::Vec<i32>`, which does
   not implement the `Copy` trait
   (注釈: `v`の型が`std::vec::Vec<i32>`のためムーブが起きました。この型は、`Copy`トレイ
       トを実装していません)
```

再三 Rust の所有権ルールが救ってくれました！ リスト 16-3 のコードはエラーになりました。コンパイラが一時的に保守的になり、スレッドに対して vを借用しただけだったからで、これは、メインスレッドは理論上、立ち上げたスレッドの参照を不正化する可能性があることを意味します。vの所有権を立ち上げたスレッドに移動するとコンパイラに指示することで、メインスレッドはもう vを使用しないとコンパイラに保証しているのです。リスト 16-4 も同様に変更したら、メインスレッドで vを使用しようとする際に所有権ルールを侵害することになります。moveキーワードにより、デフォルトで借用するという Rust の保守性が上書きされるのです; 所有権ルールを侵害させてはくれないのです。

スレッドとスレッド API の基礎知識が入ったので、スレッドで**できる**ことを見ていきましょう。

19.2　メッセージ受け渡しを使ってスレッド間でデータを転送する

人気度を増してきている安全な非同期処理を保証する一つのアプローチが**メッセージ受け渡し**で、スレッドやアクターがデータを含むメッセージを相互に送り合うことでやり取りします。こちらが、Go 言語のドキュメンテーションのスローガンにある考えです: メモリを共有することでやり取りするな; 代わりにやり取りすることでメモリを共有しろ。

メッセージ送信非同期処理を達成するために Rust に存在する一つの主な道具は、**チャンネル**で、Rust の標準ライブラリが実装を提供しているプログラミング概念です。プログラミングのチャンネルは、水の流れのように考えることができます。小川とか川ですね。アヒルのおもちゃやボートみたいなものを流れに置いたら、水路の終端まで下流に流れていきます。

プログラミングにおけるチャンネルは、2 分割できます: 転送機と受信機です。転送機はアヒルのおもちゃを川に置く上流になり、受信機は、アヒルのおもちゃが行き着く下流になります。コードのある箇所が送信したいデータとともに転送機のメソッドを呼び出し、別の部分がメッセージが到着していないか受信側を調べます。転送機と受信機のどちらかがドロップされると、チャンネルは**閉じられた**と言います。

ここで、1 つのスレッドが値を生成し、それをチャンネルに送信し、別のスレッドがその値を受け取り、出力するプログラムに取り掛かります。チャンネルを使用してスレッド間に単純な値を送り、機能の具体化を行います。一旦、そのテクニックに慣れてしまえば、チャンネルを使用してチャットシステムや、多くのスレッドが計算の一部を担い、結果をまとめる 1 つのスレッドにその部分を送るようなシステムを実装できるでしょう。

まず、リスト 16-6 において、チャンネルを生成するものの、何もしません。チャンネル越しにどんな型の値を送りたいのかコンパイラがわからないため、これはまだコンパイルできないことに注意してください。

ファイル名: src/main.rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
#     tx.send(()).unwrap();
}
```

リスト 16-6: チャンネルを生成し、2 つの部品を txと rxに代入する

mpsc::channel関数で新しいチャンネルを生成しています; mpscは *multiple producer, single consumer* を表しています。簡潔に言えば、Rust の標準ライブラリがチャンネルを実装している方法は、1 つのチャンネルが値を生成する複数の**送信**側と、その値を消費するたった 1 つの**受信**側を持つことができるということを意味します。複数の小川が互いに合わさって 1 つの大きな川になるところを想像してください: どの川を通っても、送られたものは最終的に 1 つの川に行き着きます。今は、1 つの生成器から始めますが、この例が動作するようになったら、複数の生成器を追加します。

mpsc::channel関数はタプルを返し、1 つ目の要素は、送信側、2 つ目の要素は受信側になります。txと rxという略称は、多くの分野で伝統的に**転送機**と**受信機**にそれぞれ使用されているので、変数をそのように名付けて、各終端を示します。タプルを分解するパターンを伴う let文を使用しています; let文でパターンを使用することと分解については、第 18 章で議論しましょう。このように let文を使うと、mpsc::channelで返ってくるタプルの部品を抽出するのが便利になります。

立ち上げたスレッドがメインスレッドとやり取りするように、転送機を立ち上げたスレッドに移動し、1 文字列を送らせましょう。リスト 16-7 のようにね。川の上流にアヒルのおもちゃを置いたり、チャットのメッセージをあるスレッドから別のスレッドに送るみたいですね。

ファイル名: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

リスト 16-7: txを立ち上げたスレッドに移動し、「やあ」を送る

今回も、thread::spawnを使用して新しいスレッドを生成し、それから moveを使用して、立ち上げたスレッドが txを所有するようにクロージャに txをムーブしています。立ち上げたスレッドは、メッ

セージをチャンネルを通して送信できるようにチャンネルの送信側を所有する必要があります。

転送側には、送信したい値を取る sendメソッドがあります。sendメソッドは Result<T, E>型を返すので、すでに受信側がドロップされ、値を送信する場所がなければ、送信処理はエラーを返します。この例では、エラーの場合には、パニックするように unwrapを呼び出しています。ですが、実際のアプリケーションでは、ちゃんと扱うでしょう: 第 9 章に戻ってちゃんとしたエラー処理の方法を再確認してください。

リスト 16-8 において、メインスレッドのチャンネルの受信側から値を得ます。アヒルのおもちゃを川の終端で水から回収したり、チャットメッセージを取得するみたいですね。

ファイル名: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    // 値は{}です
    println!("Got: {}", received);
}
```

リスト 16-8:「やあ」の値をメインスレッドで受け取り、出力する

チャンネルの受信側には有用なメソッドが 2 つあります: recvと try_recvです。*receive* の省略形である recvを使っていて、これは、メインスレッドの実行をブロックし、値がチャンネルを流れてくるまで待機します。一旦値が送信されたら、recvはそれを Result<T, E>に含んで返します。チャンネルの送信側が閉じたら、recvはエラーを返し、もう値は来ないと通知します。

try_recvメソッドはブロックせず、代わりに即座に Result<T, E>を返します: メッセージがあったら、それを含む Ok値、今回は何もメッセージがなければ、Err値です。メッセージを待つ間にこのスレッドにすることが他にあれば、try_recvは有用です: try_recvを頻繁に呼び出し、メッセージがあったら処理し、それ以外の場合は、再度チェックするまでちょっとの間他の作業をするループを書くことができるでしょう。

この例では、簡潔性のために recvを使用しました; メッセージを待つこと以外にメインスレッドがすべき作業はないので、メインスレッドをブロックするのは適切です。

リスト 16-8 のコードを実行したら、メインスレッドから値が出力されます:

```
Got: hi
```

完璧です！

19.2.1　チャンネルと所有権の転送

安全な非同期コードを書く手助けをしてくれるので、所有権ルールは、メッセージ送信で重要な役割を担っています。非同期プログラミングでエラーを回避することは、Rust プログラム全体で所有権について考える利点です。実験をしてチャンネルと所有権がともに動いてどう問題を回避するかをお見せしましょう: val値を立ち上げたスレッドで、チャンネルに送った後に使用を試みます。リスト 16-9 のコードのコンパイルを試みてこのコードが許容されない理由を確認してください:

ファイル名: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        // valは{}
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

リスト 16-9: チャンネルに送信後に valの使用を試みる

ここで、tx.send経由でチャンネルに送信後に valを出力しようとしています。これを許可するのは、悪い考えです: 一旦、値が他のスレッドに送信されたら、そのスレッドが再度値を使用しようとする前に変更したりドロップできてしまいます。可能性として、その別のスレッドの変更により、矛盾していたり存在しないデータのせいでエラーが発生したり、予期しない結果になるでしょう。ですが、リスト 16-9 のコードのコンパイルを試みると、Rust はエラーを返します:

```
error[E0382]: use of moved value: `val`
  --> src/main.rs:10:31
   |
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val is {}", val);
   |                               ^^^ value used here after move
   |
   = note: move occurs because `val` has type `std::string::String`, which does
not implement the `Copy` trait
```

非同期処理のミスがコンパイルエラーを招きました。send関数は引数の所有権を奪い、値がムーブされると、受信側が所有権を得るのです。これにより、送信後に誤って再度値を使用するのを防いで

くれます; 所有権システムが、万事問題ないことを確認してくれます。

19.2.2　複数の値を送信し、受信側が待機するのを確かめる

リスト 16-8 のコードはコンパイルでき、動きましたが、2 つの個別のスレッドがお互いにチャンネル越しに会話していることは、明瞭に示されませんでした。リスト 16-10 において、リスト 16-8 のコードが非同期で動いていることを証明する変更を行いました: 立ち上げたスレッドは、複数のメッセージを送信し、各メッセージ間で、1 秒待機します。

ファイル名: src/main.rs

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        // スレッドからやあ(hi from the thread)
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

リスト 16-10: 複数のメッセージを送信し、メッセージ間で停止する

今回は、メインスレッドに送信したい文字列のベクタを立ち上げたスレッドが持っています。それらをイテレートし、各々個別に送信し、`Duration`の値 1 秒とともに `thread::sleep`関数を呼び出すことで、メッセージ間で停止します。

メインスレッドにおいて、最早 `recv`関数を明示的に呼んではいません: 代わりに、`rx`をイテレータとして扱っています。受信した値それぞれを出力します。チャンネルが閉じられると、繰り返しも終わります。

リスト 16-10 のコードを走らせると、各行の間に 1 秒の待機をしつつ、以下のような出力を目の当たりにするはずです:

```
Got: hi
Got: from
Got: the
Got: thread
```

メインスレッドの forループには停止したり、遅れせたりするコードは何もないので、メインスレッドが立ち上げたスレッドから値を受け取るのを待機していることがわかります。

19.2.3　転送機をクローンして複数の生成器を作成する

mpscは、*mutiple producer, single consumer* の頭字語であると前述しました。mpscを使用に移し、リスト 16-10 のコードを拡張して全てが値を同じ受信機に送信する複数のスレッドを生成しましょう。チャンネルの転送の片割れをクローンすることでそうすることができます。リスト 16-11 のようにね:

ファイル名: src/main.rs

```
# use std::thread;
# use std::sync::mpsc;
# use std::time::Duration;
#
# fn main() {
// --snip--

let (tx, rx) = mpsc::channel();

let tx1 = mpsc::Sender::clone(&tx);
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    // 君のためにもっとメッセージを(more messages for you)
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];

```

```
34      for val in vals {
35          tx.send(val).unwrap();
36          thread::sleep(Duration::from_secs(1));
37      }
38  });
39
40  for received in rx {
41      println!("Got: {}", received);
42  }
43
44  // --snip--
45  # }
```

リスト 16-11: 複数の生成器から複数のメッセージを送信する

今回、最初のスレッドを立ち上げる前に、チャンネルの送信側に対して `clone`を呼び出しています。これにより、最初に立ち上げたスレッドに渡せる新しい送信ハンドルが得られます。元のチャンネルの送信側は、2 番目に立ち上げたスレッドに渡します。これにより 2 つスレッドが得られ、それぞれチャンネルの受信側に異なるメッセージを送信します。

コードを実行すると、出力は以下のようなものになるはずです:

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

別の順番で値が出る可能性もあります; システム次第です。非同期処理が面白いと同時に難しい部分でもあります。異なるスレッドで色々な値を与えて `thread::sleep`で実験をしたら、走らせるたびにより非決定的になり、毎回異なる出力をするでしょう。

チャンネルの動作方法を見たので、他の非同期処理に目を向けましょう。

19.3　状態共有非同期処理

メッセージ受け渡しは、非同期処理を扱う素晴らしい方法ですが、唯一の方法ではありません。Go 言語ドキュメンテーションのスローガンのこの部分を再び考えてください:「メモリを共有することでやり取りする。」

メモリを共有することでやり取りするとはどんな感じなのでしょうか？ さらに、なぜメッセージ受け渡しに熱狂的な人は、それを使わず、代わりに全く反対のことをするのでしょうか？

ある意味では、どんなプログラミング言語のチャンネルも単独の所有権に類似しています。一旦チャンネルに値を転送したら、その値は最早使用することがないからです。メモリ共有非同期処理は、複数の所有権に似ています: 複数のスレッドが同時に同じメモリ位置にアクセスできるのです。第 15

章でスマートポインタが複数の所有権を可能にするのを目の当たりにしたように、異なる所有者を管理する必要があるので、複数の所有権は複雑度を増させます。Rust の型システムと所有権ルールにより、この管理を正当に行う大きな助けになります。例を挙げれば、メモリ共有を行うより一般的な非同期処理の基本型の一つであるミューテックスを見ましょう。

19.3.1 ミューテックスを使用して一度に 1 つのスレッドからデータにアクセスすることを許可する

ミューテックスは、どんな時も 1 つのスレッドにしかなんらかのデータへのアクセスを許可しないというように、”mutual exclusion”(相互排他) の省略形です。ミューテックスにあるデータにアクセスするには、ミューテックスのロックを所望することでアクセスしたいことをまず、スレッドは通知しなければなりません。ロックとは、現在誰がデータへの排他的アクセスを行なっているかを追跡するミューテックの一部をなすデータ構造です。故に、ミューテックスはロックシステム経由で保持しているデータを**死守する** (guarding) と解説されます。

ミューテックスは、2 つの規則を覚えておく必要があるため、難しいという評判があります:

- データを使用する前にロックの獲得を試みなければならない。
- ミューテックスが死守しているデータの使用が終わったら、他のスレッドがロックを獲得できるように、データをアンロックしなければならない。

ミューテックスを現実世界の物で例えるなら、マイクが 1 つしかない会議のパネルディスカッションを思い浮かべてください。パネリストが発言できる前に、マイクを使用したいと申し出たり、通知しなければなりません。マイクを受け取ったら、話したいだけ話し、それから次に発言を申し出たパネリストにマイクを手渡します。パネリストが発言し終わった時に、マイクを手渡すのを忘れていたら、誰も他の人は発言できません。共有されているマイクの管理がうまくいかなければ、パネルは予定通りに動かないでしょう！

ミューテックスの管理は、正しく行うのに著しく巧妙なことがあるので、多くの人がチャンネルに熱狂的になるわけです。しかしながら、Rust の型システムと所有権ルールのおかげで、ロックとアンロックを間違えることはありません。

19.3.1.1 `Mutex<T>`の API

ミューテックスの使用方法の例として、ミューテックスをシングルスレッドの文脈で使うことから始めましょう。リスト 16-12 のようにね:

ファイル名: src/main.rs

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
```

```
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

リスト 16-12: 簡潔性のために Mutex<T>の API をシングルスレッドの文脈で探求する

多くの型同様、newという関連関数を使用して Mutex<T>を生成します。ミューテックス内部のデータにアクセスするには、lockメソッドを使用してロックを獲得します。この呼び出しは、現在のスレッドをブロックするので、ロックを得られる順番が来るまで何も作業はできません。

ロックを保持している他のスレッドがパニックしたら、lockの呼び出しは失敗するでしょう。その場合、誰もロックを取得することは叶わないので、unwrapすると決定し、そのような状況になったら、このスレッドをパニックさせます。

ロックを獲得した後、今回の場合、numと名付けられていますが、戻り値を中に入っているデータへの可変参照として扱うことができます。型システムにより、mの値を使用する前にロックを獲得していることが確認されます: Mutex<i32>は i32ではないので、i32を使用できるようにするには、ロックを獲得し**なければならない**のです。忘れることはあり得ません; 型システムにより、それ以外の場合に内部の i32にアクセスすることは許されません。

疑っているかもしれませんが、Mutex<T>はスマートポインタです。より正確を期すなら、lockの呼び出しが MutexGuardというスマートポインタを**返却**します。このスマートポインタが、内部のデータを指す Derefを実装しています; このスマートポインタはさらに MutexGuardがスコープを外れた時に、自動的にロックを解除する Drop実装もしていて、これがリスト 16-12 の内部スコープの終わりで発生します。結果として、ロックの解除が自動的に行われるので、ロックの解除を忘れ、ミューテックスが他のスレッドで使用されるのを阻害するリスクを負いません。

ロックをドロップした後、ミューテックスの値を出力し、内部の i32の値を 6 に変更できたことが確かめられるのです。

19.3.1.2　複数のスレッド間で Mutex<T>を共有する

さて、Mutex<T>を使って複数のスレッド間で値を共有してみましょう。10 個のスレッドを立ち上げ、各々カウンタの値を 1 ずつインクリメントさせるので、カウンタは 0 から 10 まで上がります。以下の数例は、コンパイルエラーになることに注意し、そのエラーを使用して Mutex<T>の使用法と、コンパイラがそれを正しく活用する手助けをしてくれる方法について学びます。リスト 16-13 が最初の例です:

ファイル名: src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
```

```
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

リスト 16-13: Mutex<T>により死守されているカウンタを 10 個のスレッドがそれぞれインクリメントする

リスト 16-12 のように、counter変数を生成して Mutex<T>の内部に i32を保持しています。次に、数値の範囲をマッピングして 10 個のスレッドを生成しています。thread::spawnを使用して、全スレッドに同じクロージャを与えています。このクロージャは、スレッド内にカウンタをムーブし、lockメソッドを呼ぶことで Mutex<T>のロックを獲得し、それからミューテックスの値に 1 を足します。スレッドがクロージャを実行し終わったら、numはスコープ外に出てロックを解除するので、他のスレッドが獲得できるわけです。

メインスレッドで全ての join ハンドルを収集します。それからリスト 16-2 のように、各々に対して joinを呼び出し、全スレッドが終了するのを確かめています。その時点で、メインスレッドはロックを獲得し、このプログラムの結果を出力します。

この例はコンパイルできないでしょうと仄めかしました。では、理由を探りましょう！

```
error[E0382]: capture of moved value: `counter`
(エラー: ムーブされた値をキャプチャしています: `counter`)
  --> src/main.rs:10:27
   |
9  |         let handle = thread::spawn(move || {
   |                                    ------- value moved (into closure) here
10 |             let mut num = counter.lock().unwrap();
   |                           ^^^^^^^ value captured here after move
   |
   = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
   which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
  --> src/main.rs:21:29
   |
9  |         let handle = thread::spawn(move || {
   |                                    ------- value moved (into closure) here
```

```
...
21 |     println!("Result: {}", *counter.lock().unwrap());
   |                             ^^^^^^^ value used here after move
   |
   = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
   which does not implement the `Copy` trait

error: aborting due to 2 previous errors
(エラー: 前述の2つのエラーにより アボート)
```

エラーメッセージは、counter値はクロージャにムーブされ、それから lockを呼び出したときにキャプチャされていると述べています。その説明は、所望した動作のように聞こえますが、許可されていないのです！

プログラムを単純化してこれを理解しましょう。forループで 10 個スレッドを生成する代わりに、ループなしで 2 つのスレッドを作るだけにしてどうなるか確認しましょう。リスト 16-13 の最初のforループを代わりにこのコードと置き換えてください:

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();

        *num += 1;
    });
    handles.push(handle);

    let handle2 = thread::spawn(move || {
        let mut num2 = counter.lock().unwrap();

        *num2 += 1;
    });
    handles.push(handle2);

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

2 つのスレッドを生成し、2 番目のスレッドの変数名を handle2と num2に変更しています。今回このコードを走らせると、コンパイラは以下の出力をします:

```
error[E0382]: capture of moved value: `counter`
  --> src/main.rs:16:24
```

```
   |
8  |     let handle = thread::spawn(move || {
   |                                ------- value moved (into closure) here
...
16 |         let mut num2 = counter.lock().unwrap();
   |                        ^^^^^^^ value captured here after move
   |
   = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
   which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
  --> src/main.rs:26:29
   |
8  |     let handle = thread::spawn(move || {
   |                                ------- value moved (into closure) here
...
26 |     println!("Result: {}", *counter.lock().unwrap());
   |                             ^^^^^^^ value used here after move
   |
   = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
   which does not implement the `Copy` trait

error: aborting due to 2 previous errors
```

なるほど！ 最初のメッセージは、handleに紐付けられたスレッドのクロージャに counterがムーブされていることを示唆しています。そのムーブにより、それに対して lockを呼び出し、結果を 2 番目のスレッドの num2に保持しようとした時に、counterをキャプチャすることを妨げています。ゆえに、コンパイラは、counterの所有権を複数のスレッドに移すことはできないと教えてくれています。これは、以前では確認しづらかったことです。なぜなら、スレッドはループの中にあり、ループの違う繰り返しにある違うスレッドをコンパイラは指し示せないからです。第 15 章で議論した複数所有権メソッドによりコンパイルエラーを修正しましょう。

19.3.1.3　複数のスレッドで複数の所有権

第 15 章で、スマートポインタの Rc<T>を使用して参照カウントの値を作ることで、1 つの値に複数の所有者を与えました。同じことをここでもして、どうなるか見ましょう。リスト 16-14 で Rc<T>に Mutex<T>を包含し、所有権をスレッドに移す前に Rc<T>をクローンします。今やエラーを確認したので、forループの使用に立ち戻り、クロージャに moveキーワードを使用し続けます。

ファイル名: src/main.rs

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
```

```
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

リスト 16-14: Rc<T>を使用して複数のスレッドに Mutex<T>を所有させようとする

再三、コンパイルし……別のエラーが出ました！ コンパイラはいろんなことを教えてくれています。

```
error[E0277]: the trait bound `std::rc::Rc<std::sync::Mutex<i32>>:
std::marker::Send` is not satisfied in `[closure@src/main.rs:11:36:
15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`
(エラー: トレイト境界`std::rc::Rc<std::sync::Mutex<i32>>:
std::marker::Send`は`[closure@src/main.rs:11:36:15:10
counter:std::rc::Rc<std::sync::Mutex<i32>>]`で満たされていません)
  --> src/main.rs:11:22
   |
11 |         let handle = thread::spawn(move || {
   |                      ^^^^^^^^^^^^^ `std::rc::Rc<std::sync::Mutex<i32>>`
cannot be sent between threads safely
                      (`std::rc::Rc<std::sync::Mutex<i32>>`は、スレッド間で安全に送
                          信できません)
   |
   = help: within `[closure@src/main.rs:11:36: 15:10
counter:std::rc::Rc<std::sync::Mutex<i32>>]`, the trait `std::marker::Send` is
not implemented for `std::rc::Rc<std::sync::Mutex<i32>>`
     (ヘルプ: `[closure@src/main.rs:11:36 15:10
     counter:std::rc::Rc<std::sync::Mutex<i32>>]`内でトレイト`std::marker::Send`は、
     `std::rc::Rc<std::sync::Mutex<i32>>`に対して実装されていません)
   = note: required because it appears within the type
`[closure@src/main.rs:11:36: 15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`
     (注釈: 型`[closure@src/main.rs:11:36 15:10
     counter:std::rc::Rc<std::sync::Mutex<i32>>]`内に出現するので必要です)
   = note: required by `std::thread::spawn`
     (注釈: `std::thread::spawn`により必要とされています)
```

おお、このエラーメッセージはとても長ったらしいですね！ こちらが、注目すべき重要な部分です: 最初のインラインエラーは`std::rc::Rc<std::sync::Mutex<i32>>` cannot be sent between threads safelyと述べています。この理由は、次に注目すべき重要な部分、エラーメッセージにあります。蒸留されたエラーメッセージは、the trait bound `Send` is not satisfiedと述べています。Sendについては、次の節で語ります: スレッドとともに使用している型が非同期処理の場面で使

われることを意図したものであることを保証するトレイトの 1 つです。

残念ながら、Rc<T>はスレッド間で共有するには安全ではないのです。Rc<T>が参照カウントを管理する際、cloneが呼び出されるたびにカウントを追加し、クローンがドロップされるたびにカウントを差し引きます。しかし、非同期基本型を使用してカウントの変更が別のスレッドに妨害されないことを確認していないのです。これは間違ったカウントにつながる可能性があり、今度はメモリリークや、使用し終わる前に値がドロップされることにつながる可能性のある潜在的なバグです。必要なのは、全く Rc<T>のようだけれども、参照カウントへの変更をスレッドセーフに行うものです。

19.3.1.4　Arc<T>で原子的な参照カウント

幸いなことに、Arc<T>は Rc<T>のような非同期処理の状況で安全に使用できる型です。*a* は *atomic* を表し、原子的に参照カウントする型を意味します。アトミックは、ここでは詳しく講義しない非同期処理の別の基本型です: 詳細は、std::sync::atomicの標準ライブラリドキュメンテーションを参照されたし。現時点では、アトミックは、基本型のように動くけれども、スレッド間で共有しても安全なことだけ知っていればいいです。

そうしたら、なぜ全ての基本型がアトミックでなく、標準ライブラリの型も標準で Arc<T>を使って実装されていないのか疑問に思う可能性があります。その理由は、本当に必要な時だけ支払いたいパフォーマンスの犠牲とともにスレッド安全性が得られるものだからです。シングルスレッドで値に処理を施すだけなら、アトミックが提供する保証を強制する必要がなければコードはより速く走るのです。

例に回帰しましょう: Arc<T>と Rc<T>の API は同じなので、use行と newと cloneの呼び出しを変更して、プログラムを修正します。リスト 16-15 は、ようやくコンパイルでき、動作します:

ファイル名: src/main.rs

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

```

```
    println!("Result: {}", *counter.lock().unwrap());
}
```

リスト 16-15: Arc<T>を使用して Mutex<T>をラップし、所有権を複数のスレッド間で共有できるようにする

このコードは、以下のように出力します:

```
Result: 10
```

やりました！ 0 から 10 まで数え上げました。これは、あまり印象的ではないように思えるかもしれませんが、本当に Mutex<T>とスレッド安全性についていろんなことを教えてくれました。このプログラムの構造を使用して、カウンタをインクリメントする以上の複雑な処理を行うこともできるでしょう。この手法を使えば、計算を独立した部分に小分けにし、その部分をスレッドに分割し、それから Mutex<T>を使用して、各スレッドに最終結果を更新させることができます。

19.3.2　RefCell<T>/Rc<T>と Mutex<T>/Arc<T>の類似性

counterは不変なのに、その内部にある値への可変参照を得ることができたことに気付いたでしょうか; つまり、Mutex<T>は、Cell系のように内部可変性 (interior mutability) を提供するわけです。第 15 章で RefCell<T>を使用して Rc<T>の内容を可変化できるようにしたのと同様に、Mutex<T>を使用して Arc<T>の内容を可変化しているのです。

また、気付いておくべき別の詳細は、Mutex<T>を使用する際にあらゆる種類のロジックエラーからは、コンパイラは保護してくれないということです。第 15 章で Rc<T>は、循環参照を生成してしまうリスクを伴い、2 つの Rc<T>の値がお互いを参照し合い、メモリリークを引き起こしてしまうことを思い出してください。同様に、Mutex<T>は**デッドロック**を生成するリスクを伴っています。これは、処理が 2 つのリソースをロックする必要があり、2 つのスレッドがそれぞれにロックを 1 つ獲得して永久にお互いを待ちあってしまうときに起こります。デッドロックに興味があるのなら、デッドロックのある Rust プログラムを組んでみてください; それからどんな言語でもいいので、ミューテックスに対してデッドロックを緩和する方法を調べて、Rust でそれを実装してみてください。Mutex<T>と MutexGuardに関する標準ライブラリの API ドキュメンテーションは、役に立つ情報を提供してくれます。

Sendと Syncトレイトと、それらを独自の型で使用する方法について語って、この章を締めくくります。

19.4　Sync と Send トレイトで拡張可能な非同期処理

面白いことに、Rust 言語には、寡少な非同期機能があります。この章でここまでに語った非同期処理のほとんどは、標準ライブラリの一部であり、言語ではありません。非同期を扱う選択肢は、言語や標準ライブラリに制限されません; 独自の非同期処理機能を書いたり、他人が書いたものを利用し

たりできるのです。

ですが、2 つの非同期処理概念が言語に埋め込まれています: `std::marker`トレイトの `Sync`と `Send`です。

19.4.1　Send でスレッド間の所有権の転送を許可する

`Send`マーカートレイトは、`Send`を実装した型の所有権をスレッド間で転送できることを示唆します。Rust のほとんどの型は `Send`ですが、`Rc<T>`を含めて一部例外があります: この型は、`Rc<T>`の値をクローンし、クローンしたものの所有権を別のスレッドに転送しようとしたら、両方のスレッドが同時に参照カウントを更新できてしまうので、`Send`になり得ません。このため、`Rc<T>`はスレッド安全性のためのパフォーマンスの犠牲を支払わなくても済む、シングルスレッド環境で使用するために実装されているわけです。

故に、Rust の型システムとトレイト境界により、`Rc<T>`の値を不安全にスレッド間で誤って送信することが絶対ないよう保証してくれるのです。リスト 16-14 でこれを試みた時には、`the trait Send is not implemented for Rc<Mutex<i32>>`というエラーが出ました。`Send`の `Arc<T>`に切り替えたら、コードはコンパイルできたわけです。

完全に `Send`の型からなる型も全て自動的に `Send`と印付けされます。生ポインタを除くほとんどの基本型も `Send`で、生ポインタについては第 19 章で議論します。

19.4.2　Sync で複数のスレッドからのアクセスを許可する

`Sync`マーカートレイトは、`Sync`を実装した型は、複数のスレッドから参照されても安全であることを示唆します。言い換えると、`&T`(`T`への参照) が `Send`なら、型 `T`は `Sync`であり、参照が他のスレッドに安全に送信できることを意味します。`Send`同様、基本型は `Sync`であり、`Sync`の型からのみ構成される型もまた `Sync`です。

`Send`ではなかったのと同じ理由で、スマートポインタの `Rc<T>`もまた `Sync`ではありません。`RefCell<T>`型 (これについては第 15 章で話しました) と `Cell<T>`系についても `Sync`ではありません。`RefCell<T>`が実行時に行う借用チェックの実装は、スレッド安全ではないのです。スマートポインタの `Mutex<T>`は `Sync`で、「複数のスレッド間で `Mutex<T>`を共有する」節で見たように、複数のスレッドでアクセスを共有するのに使用することができます。

19.4.3　Send と Sync を手動で実装するのは非安全である

`Send`と `Sync`トレイトから構成される型は自動的に `Send`と `Sync`にもなるので、それらのトレイトを手動で実装する必要はありません。マーカートレイトとして、実装すべきメソッドさえも何もありません。非同期処理に関連する不変条件を強制することに有効なだけなのです。

これらのトレイトを手動で実装すると、unsafe な Rust コードを実装することが関わってきます。unsafe な Rust コードを使用することについては第 19 章で語ります; とりあえず、重要な情報は、

`Send`と `Sync`ではない部品からなる新しい非同期型を構成するには、安全性保証を保持するために、注意深い思考が必要になるということです。The Rustonomicon には、これらの保証とそれを保持する方法についての情報がより多くあります。

19.5　まとめ

この本において非同期処理を見かけるのは、これで最後ではありません: 第 20 章のプロジェクトでは、この章の概念をここで議論した微小な例よりもより現実的な場面で使用するでしょう。

前述のように、Rust が非同期処理を扱うごく一部が言語の一部なので、多くの非同期処理解決策は、クレートとして実装されています。これらは標準ライブラリよりも迅速に進化するので、確実にオンラインでマルチスレッド環境で使用する現在の最先端のクレートを検索してください。

Rust の標準ライブラリは、メッセージ受け渡しにチャンネルを、非同期処理の文脈で安全に使用できる、`Mutex<T>`や `Arc<T>`などのスマートポインタ型を提供しています。型システムと借用精査機により、これらの解決策を使用するコードがデータ競合や無効な参照に行き着かないことを保証してくれます。一旦コードをコンパイルすることができたら、他の言語ではありふれている追跡困難なバグなしに、複数のスレッドでも喜んで動くので安心できます。非同期プログラミングは、もはや恐れるべき概念ではありません: 進んでプログラムを非同期にして恐れないでください！

次は、Rust プログラムが肥大化するにつれて問題をモデル化し、解決策を構造化する慣例的な方法について話します。さらに、Rust のイディオムがオブジェクト指向プログラミングで馴染み深いかもしれないイディオムに関連する方法についても議論します。

第 20 章

Rust のオブジェクト指向プログラミング機能

オブジェクト指向プログラミング (OOP) は、プログラムをモデル化する手段です。オブジェクトは、1960 年代の Simula で導入されました。このオブジェクトは、お互いにメッセージを渡し合うというアラン・ケイ (Alan Kay) のプログラミングアーキテクチャに影響を及ぼしました。彼は、このアーキテクチャを解説するために、**オブジェクト指向プログラミング**という用語を造語しました。多くの競合する定義が OOP が何かを解説しています; Rust をオブジェクト指向と区分する定義もありますし、しない定義もあります。この章では、広くオブジェクト指向と捉えられる特定の特徴と、それらの特徴がこなれた Rust でどう表現されるかを探求します。それからオブジェクト指向のデザインパターンを Rust で実装する方法を示し、そうすることと Rust の強みを活用して代わりの解決策を実装する方法の代償を議論します。

20.1　オブジェクト指向言語の特徴

言語がオブジェクト指向と考えられるのになければならない機能について、プログラミングコミュニティ内での総意はありません。Rust は OOP を含めた多くのプログラミングパラダイムに影響を受けています; 例えば、第 13 章で関数型プログラミングに由来する機能を探求しました。議論はあるかもしれないが、OOP 言語は特定の一般的な特徴を共有しています。具体的には、オブジェクトやカプセル化、継承などです。それらの個々の特徴が意味するものと Rust がサポートしているかを見ましょう。

20.1.1　オブジェクトは、データと振る舞いを含む

エーリヒ・ガンマ (Enoch Gamma)、リチャード・ヘルム (Richard Helm)、ラルフ・ジョンソン (Ralph Johnson)、ジョン・ブリシディース (John Vlissides)(アディソン・ワズリー・プロ) により、

1994 に書かれた**デザインパターン: 再利用可能なオブジェクト指向ソフトウェアの要素**という本は、俗に *4 人のギャングの本* (脚注: the Gang of Four book; GoF とよく略される) と呼ばれ、オブジェクト指向デザインパターンのカタログです。そこでは、OOP は以下のように定義されています:

> オブジェクト指向プログラムは、オブジェクトで構成される。オブジェクトは、データとそのデータを処理するプロシージャを梱包している。このプロシージャは、典型的に**メソッド**または**オペレーション**と呼ばれる。

この定義を使用すれば、Rust はオブジェクト指向です: 構造体と enum にはデータがありますし、`impl`ブロックが構造体と enum にメソッドを提供します。メソッドのある構造体と enum は、オブジェクトとは呼ばれないものの、GoF のオブジェクト定義によると、同じ機能を提供します。

20.1.2　カプセル化は、実装詳細を隠蔽する

OOP とよく紐づけられる別の側面は、カプセル化の思想です。これは、オブジェクトの実装詳細は、そのオブジェクトを使用するコードにはアクセスできないことを意味します。故に、オブジェクトと相互作用する唯一の手段は、その公開 API を通してです; オブジェクトを使用するコードは、オブジェクトの内部に到達して、データや振る舞いを直接変更できるべきではありません。このために、プログラマはオブジェクトの内部をオブジェクトを使用するコードを変更する必要なく、変更しリファクタリングできます。

カプセル化を制御する方法は、第 7 章で議論しました: `pub`キーワードを使用して、自分のコードのどのモジュールや型、関数、メソッドを公開するか決められ、規定ではそれ以外のものは全て非公開になります。例えば、`i32`値のベクタを含むフィールドのある `AveragedCollection`という構造体を定義できます。この構造体はさらに、ベクタの値の平均を含むフィールドを持てます。つまり、平均は誰かが必要とする度に、オンデマンドで計算する必要はないということです。言い換えれば、`AveragedCollection`は、計算した平均をキャッシュしてくれるわけです。リスト 17-1 には、`AveragedCollection`構造体の定義があります:

ファイル名: src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

リスト 17-1: 整数のリストとコレクションの要素の平均を管理する `AveragedCollection`構造体

構造体は、他のコードが使用できるように `pub`で印づけされていますが、構造体のフィールドは非公開のままです。値が追加されたりリストから削除される度に、平均も更新されることを保証したいので、今回の場合重要です。`add`や `remove`、`average`メソッドを構造体に実装することでこれをします。リスト 17-2 のようにね:

ファイル名: src/lib.rs

```
# pub struct AveragedCollection {
#     list: Vec<i32>,
#     average: f64,
# }
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            },
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

リスト 17-2: AveragedCollectionの add、remove、average公開メソッドの実装

add、remove、averageの公開メソッドが AveragedCollectionのインスタンスを変更する唯一の方法になります。要素が addメソッドを使用して listに追加されたり、removeメソッドを使用して削除されたりすると、各メソッドの実装が averageフィールドの更新を扱う非公開の update_averageメソッドも呼び出します。

listと averageフィールドを非公開のままにしているので、外部コードが要素を listフィールドに直接追加したり削除したりする方法はありません; そうでなければ、averageフィールドは、listが変更された時に同期されなくなる可能性があります。averageメソッドは averageフィールドの値を返し、外部コードに averageを読ませるものの、変更は許可しません。

構造体 AveragedCollectionの実装詳細をカプセル化したので、データ構造などの側面を将来容易に変更することができます。例を挙げれば、listフィールドに Vecではなく HashSetを使うこともできます。add、remove、average公開メソッドのシグニチャが同じである限り、AveragedCollectionを使用するコードは変更する必要がないでしょう。代わりに listを公開にしたら、必ずしもこうはならないでしょう: HashSetと Vecは、要素の追加と削除に異なるメソッドを持っているので、外部コードが直接 listを変更しているなら、外部コードも変更しなければならない可能性が高いでしょう。

カプセル化が、言語がオブジェクト指向と考えられるのに必要な側面ならば、Rust はその条件を満たしています。コードの異なる部分で pubを使用するかしないかという選択肢のおかげで、実装詳細をカプセル化することが可能になります。

20.1.3　型システム、およびコード共有としての継承

継承は、それによってオブジェクトが他のオブジェクトの定義から受け継ぐことができる機構であり、それ故に、再定義する必要なく、親オブジェクトのデータと振る舞いを得ます。

言語がオブジェクト指向言語であるために継承がなければならないのならば、Rust は違います。親構造体のフィールドとメソッドの実装を受け継ぐ構造体を定義する方法はありません。しかしながら、継承がプログラミング道具箱にあることに慣れていれば、そもそも継承に手を伸ばす理由によって、Rust で他の解決策を使用することができます。

継承を選択する理由は 2 つあります。1 つ目は、コードの再利用です: ある型に特定の振る舞いを実装し、継承により、その実装を他の型にも再利用できるわけです。デフォルトのトレイトメソッド実装を代わりに使用して、Rust コードを共有でき、これは、リスト 10-14 で Summaryトレイトに summarizeメソッドのデフォルト実装を追加した時に見かけました。Summaryトレイトを実装する型は全て、追加のコードなく summarizeメソッドが使用できます。これは、親クラスにメソッドの実装があり、継承した子クラスにもそのメソッドの実装があることと似ています。また、Summaryトレイトを実装する時に、summarizeメソッドのデフォルト実装を上書きすることもでき、これは、親クラスから継承したメソッドの実装を子クラスが上書きすることに似ています。

継承を使用するもう 1 つの理由は、型システムに関連しています: 親の型と同じ箇所で子供の型を使用できるようにです。これは、**多相性** (polymorphism) とも呼ばれ、複数のオブジェクトが特定の特徴を共有しているなら、実行時にお互いに代用できることを意味します。

20.1.4　多相性

多くの人にとって、多相性は、継承の同義語です。ですが、実際には複数の型のデータを取り扱えるコードを指すより一般的な概念です。継承について言えば、それらの型は一般的にはサブクラスです。

Rust は代わりにジェネリクスを使用して様々な可能性のある型を抽象化し、トレイト境界を使用してそれらの型が提供するものに制約を課します。これは時に、**パラメータ境界多相性** (bounded parametric polymorphism) と呼ばれます。

継承は、最近、多くのプログラミング言語において、プログラムの設計解決策としては軽んじられています。というのも、しばしば必要以上にコードを共有してしまう危険性があるからです。サブクラスは、必ずしも親クラスの特徴を全て共有するべきではないのに、継承ではそうなってしまうのです。これにより、プログラムの設計の柔軟性がなくなります。また道理に合わなかったり、メソッドがサブクラスには適用されないために、エラーを発生させるサブクラスのメソッドの呼び出しを引き

起こす可能性が出てくるのです。さらに、サブクラスに 1 つのクラスからだけ継承させる言語もあり、さらにプログラムの設計の柔軟性が制限されます。

これらの理由により、継承ではなくトレイトオブジェクトを使用して Rust は異なるアプローチを取っています。Rust において、トレイトオブジェクトがどう多相性を可能にするかを見ましょう。

20.2　トレイトオブジェクトで異なる型の値を許可する

第 8 章で、ベクタの 1 つの制限は、たった 1 つの型の要素を保持することしかできないことだと述べました。リスト 8-10 で整数、浮動小数点数、テキストを保持する列挙子のある `SpreadsheetCell`enum を定義して、これを回避しました。つまり、各セルに異なる型のデータを格納しつつ、1 行のセルを表すベクタを保持するということです。コンパイル時にわかるある固定されたセットの型にしか取り替え可能な要素がならない場合には、完璧な解決策です。

ところが、時として、ライブラリの使用者が特定の場面で有効になる型のセットを拡張できるようにしたくなることがあります。これをどう実現する可能性があるか示すために、各アイテムに `draw`メソッドを呼び出してスクリーンに描画するという、GUI ツールで一般的なテクニックをしてあるリストの要素を走査する例の GUI ツールを作ります。GUI ライブラリの構造を含む `gui`と呼ばれるライブラリクレートを作成します。このクレートには、他人が使用できる `Button`や `TextField`などの型が包含される可能性があります。さらに、`gui`の使用者は、描画可能な独自の型を作成したくなるでしょう: 例えば、ある人は `Image`を追加し、別の人は `SelectBox`を追加する可能性があります。

この例のために本格的な GUI ライブラリは実装するつもりはありませんが、部品がどう組み合わさるかは示します。ライブラリの記述時点では、他のプログラマが作成したくなる可能性のある型全てを知る由も、定義することもできません。しかし、`gui`は異なる型の多くの値を追いかけ、この異なる型の値に対して `draw`メソッドを呼び出す必要があることは、確かにわかっています。`draw`メソッドを呼び出した時に正確に何が起きるかを知っている必要はありません。値にそのメソッドが呼び出せるようあることだけわかっていればいいのです。

継承のある言語でこれを行うには、`draw`という名前のメソッドがある `Component`というクラスを定義する可能性があります。`Button`、`Image`、`SelectBox`などの他のクラスは、`Component`を継承し、故に `draw`メソッドを継承します。個々に `draw`メソッドをオーバーライドして、独自の振る舞いを定義するものの、フレームワークは、`Component`インスタンスであるかのようにその型全部を扱い、この型に対して `draw`を呼び出します。ですが、Rust に継承は存在しないので、使用者に新しい型で拡張してもらうために `gui`ライブラリを構成する他の方法が必要です。

20.2.1　一般的な振る舞いにトレイトを定義する

`gui`に欲しい振る舞いを実装するには、`draw`という 1 つのメソッドを持つ `Draw`というトレイトを定義します。それから**トレイトオブジェクト**を取るベクタを定義できます。トレイトオブジェクトは、指定したトレイトを実装するある型のインスタンスを指します。`&`参照や `Box<T>`スマートポインタな

どの、何らかのポインタを指定し、それから関係のあるトレイトを指定する (トレイトオブジェクトがポインタを使用しなければならない理由については、第 19 章の「動的サイズ型と Sized」節で語ります) ことでトレイトオブジェクトを作成します。ジェネリックまたは具体的な型があるところにトレイトオブジェクトは使用できます。どこでトレイトオブジェクトを使用しようと、Rust の型システムは、コンパイル時にその文脈で使用されているあらゆる値がそのトレイトオブジェクトのトレイトを実装していることを保証します。結果的にコンパイル時に可能性のある型を全て知る必要はなくなるのです。

Rust では、構造体と enum を他の言語のオブジェクトと区別するために「オブジェクト」と呼ぶことを避けていることに触れましたね。構造体や enum において、構造体のフィールドのデータや `impl` ブロックの振る舞いは区分けされているものの、他の言語では 1 つの概念に押し込められるデータと振る舞いにはしばしばオブジェクトと分類されます。しかしながら、トレイトオブジェクトは、データと振る舞いをごちゃ混ぜにするという観点で他の言語のオブジェクトに近いです。しかし、トレイトオブジェクトは、データを追加できないという点で伝統的なオブジェクトと異なっています。トレイトオブジェクトは、他の言語のオブジェクトほど一般的に有用ではありません: その目的は、共通の振る舞いに対して抽象化を行うことです。

リスト 17-3 は、`draw`という 1 つのメソッドを持つ `Draw`というトレイトを定義する方法を示しています:

ファイル名: src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

リスト 17-3: `Draw`トレイトの定義

この記法は、第 10 章のトレイトの定義方法に関する議論で馴染み深いはずです。その次は、新しい記法です: リスト 17-4 では、`components`というベクタを保持する `Screen`という名前の構造体を定義しています。このベクタの型は `Box<Draw>`で、これはトレイトオブジェクトです; `Draw`トレイトを実装する `Box`内部の任意の型に対する代役です。

ファイル名: src/lib.rs

```
# pub trait Draw {
#     fn draw(&self);
# }
#
pub struct Screen {
    pub components: Vec<Box<Draw>>,
}
```

リスト 17-4: `Draw`トレイトを実装するトレイトオブジェクトのベクタを保持する `components`フィールドがある `Screen`構造体の定義

`Screen`構造体に、`components`の各要素に対して `draw`メソッドを呼び出す `run`というメソッドを定義します。リスト 17-5 のようにね:

ファイル名: src/lib.rs

```
# pub trait Draw {
#     fn draw(&self);
# }
#
# pub struct Screen {
#     pub components: Vec<Box<Draw>>,
# }
#
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

リスト 17-5: 各コンポーネントに対して drawメソッドを呼び出す Screenの runメソッド

これは、トレイト境界を含むジェネリックな型引数を使用する構造体を定義するのとは異なる動作をします。ジェネリックな型引数は、一度に 1 つの具体型にしか置き換えられないのに対して、トレイトオブジェクトは、実行時にトレイトオブジェクトに対して複数の具体型で埋めることができます。例として、ジェネリックな型とトレイト境界を使用してリスト 17-6 のように Screen構造体を定義することもできました:

ファイル名: src/lib.rs

```
# pub trait Draw {
#     fn draw(&self);
# }
#
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
    where T: Draw {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

リスト 17-6: ジェネリクスとトレイト境界を使用した Screen構造体と runメソッドの対立的な実装

こうすると、全てのコンポーネントの型が Buttonだったり、TextFieldだったりする Screenのインスタンスに制限されてしまいます。同種のコレクションしか持つ予定がないのなら、ジェネリクスとトレイト境界は、定義がコンパイル時に具体的な型を使用するように単相化されるので、望ましいです。

一方で、メソッドがトレイトオブジェクトを使用すると、1 つの Screenインスタンスが、Box<Button>と Box<TextField>を含む Vecを保持できます。この動作方法を見、それから実行時性能の裏の意味について語りましょう。

20.2.2　トレイトを実装する

さて、Drawトレイトを実装する型を追加しましょう。Button型を提供します。ここも、実際に GUI ライブラリを実装することは、この本の範疇を超えているので、drawメソッドの本体は、何も有用な実装はしません。実装がどんな感じになるか想像するために、Button構造体は、width、height、labelフィールドを持っている可能性があります。リスト 17-7 に示したようにね:

ファイル名: src/lib.rs

```
# pub trait Draw {
#     fn draw(&self);
# }
#
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
        // 実際にボタンを描画するコード
    }
}
```

リスト 17-7: Drawトレイトを実装するある Button構造体

Buttonの width、height、labelフィールドは、TextField型のように、それらのフィールドプラス placeholderフィールドを代わりに持つ可能性のある他のコンポーネントのフィールドとは異なるでしょう。スクリーンに描画したい型のコンポーネントはそれぞれ Drawトレイトを実装しますが、Buttonがここでしているように、drawメソッドでは異なるコードを使用してその特定の型を描画する方法を定義しています (実際の GUI コードは、この章の範疇を超えるのでありませんが)。例えば、Buttonには、ユーザがボタンをクリックした時に起こることに関連するメソッドを含む、追加の implブロックがある可能性があります。この種のメソッドは、TextFieldのような型には適用されません。

ライブラリの使用者が、width、height、optionsフィールドのある SelectBox構造体を実装しようと決めたら、SelectBox型に Drawトレイトも実装します。リスト 17-8 のようにね:

ファイル名: src/main.rs

```
extern crate gui;
use gui::Draw;

struct SelectBox {
```

```
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
        //セレクトボックスを実際に描画するコード
    }
}
```

リスト 17-8: guiを使用し、SelectBox構造体に Drawトレイトを実装する別のクレート

ライブラリの使用者はもう、main関数を書き、Screenインスタンスを生成できます。Screenインスタンスには、それぞれを Box<T>に放り込んでトレイトオブジェクト化して SelectBoxと Buttonを追加できます。それから Screenインスタンスに対して runメソッドを呼び出すことができ、そうすると各コンポーネントの drawが呼び出されます。リスト 17-9 は、この実装を示しています:

ファイル名: src/main.rs

```
use gui::{Screen, Button};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    // はい
                    String::from("Yes"),
                    // 多分
                    String::from("Maybe"),
                    // いいえ
                    String::from("No")
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                // 了解
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

リスト 17-9: トレイトオブジェクトを使って同じトレイトを実装する異なる型の値を格納する

ライブラリを記述した時点では、誰かが SelectBox型を追加する可能性があるなんて知りませんでしたが、Screenの実装は、新しい型を処理し、描画することができました。何故なら、SelectBoxはDraw型、つまり、drawメソッドを実装しているからです。

この値の具体的な型ではなく、値が応答したメッセージにのみ関係するという概念は、動的型付け言語の**ダックタイピング**に似た概念です: アヒルのように歩き、鳴くならば、アヒルに違いないのです！ リスト 17-5 の Screenの runの実装では、runは、各コンポーネントの実際の型がなんであるか知る必要はありません。コンポーネントが、Buttonや SelectBoxのインスタンスであるかを確認することはなく、コンポーネントの drawメソッドを呼び出すだけです。componentsベクタで Box<Draw>を値の型として指定することで、Screenを、drawメソッドを呼び出せる値を必要とするように定義できたのです。

トレイトオブジェクトと Rust の型システムを使用してダックタイピングを活用したコードに似たコードを書くことの利点は、実行時に値が特定のメソッドを実装しているか確認したり、値がメソッドを実装していない時にエラーになることを心配したりする必要は絶対になく、とにかく呼び出せることです。コンパイラは、値が、トレイトオブジェクトが必要としているトレイトを実装していなければ、コンパイルを通さないのです。

例えば、リスト 17-10 は、コンポーネントに Stringのある Screenを作成しようとした時に起こることを示しています:

ファイル名: src/main.rs

```
extern crate gui;
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(String::from("Hi")),
        ],
    };

    screen.run();
}
```

リスト 17-10: トレイトオブジェクトのトレイトを実装しない型の使用を試みる

Stringは Drawトレイトを実装していないので、このようなエラーが出ます:

```
error[E0277]: the trait bound `std::string::String: gui::Draw` is not satisfied
  --> src/main.rs:7:13
   |
 7 |             Box::new(String::from("Hi")),
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait gui::Draw is not
   implemented for `std::string::String`
   |
   = note: required for the cast to the object type `gui::Draw`
```

このエラーは、渡すことを意図していないものを Screenに渡しているので、異なる型を渡すべき

か、Screenが drawを呼び出せるように Stringに Drawを実装するべきのどちらかであることを知らせてくれています。

20.2.3　トレイトオブジェクトは、ダイナミックディスパッチを行う

第 10 章の「ジェネリクスを使用したコードのパフォーマンス」節でジェネリクスに対してトレイト境界を使用した時に、コンパイラが行う単相化過程の議論を思い出してください: コンパイラは、関数やメソッドのジェネリックでない実装を、ジェネリックな型引数の箇所に使用している具体的な型に対して生成するんでした。単相化の結果吐かれるコードは、**スタティックディスパッチ**を行い、これは、コンパイル時にコンパイラがどのメソッドを呼び出しているかわかる時のことです。これは、**ダイナミックディスパッチ**とは対照的で、この時、コンパイラは、コンパイル時にどのメソッドを呼び出しているのかわかりません。ダイナミックディスパッチの場合、コンパイラは、実行時にどのメソッドを呼び出すか弾き出すコードを生成します。

トレイトオブジェクトを使用すると、コンパイラはダイナミックディスパッチを使用しなければなりません。コンパイラは、トレイトオブジェクトを使用しているコードで使用される可能性のある型全てを把握しないので、どの型に実装されたどのメソッドを呼び出すかわからないのです。代わりに実行時に、トレイトオブジェクト内でポインタを使用して、コンパイラは、どのメソッドを呼ぶか知ります。スタティックディスパッチでは行われないこの検索が起きる時には、実行時コストがあります。また、ダイナミックディスパッチは、コンパイラがメソッドのコードをインライン化することも妨げ、そのためある種の最適化が不可能になります。ですが、リスト 17-5 で記述し、リスト 17-9 ではサポートできたコードで追加の柔軟性を確かに得られたので、考慮すべき代償です。

20.2.4　トレイトオブジェクトには、オブジェクト安全性が必要

トレイトオブジェクトには、**オブジェクト安全**なトレイトしか作成できません。トレイトオブジェクトを安全にする特性全てを司る複雑な規則がありますが、実際には、2 つの規則だけが関係があります。トレイトは、トレイト内で定義されているメソッド全てに以下の特性があれば、オブジェクト安全になります。

- 戻り値の型が Selfでない。
- ジェネリックな型引数がない。

Selfキーワードは、トレイトやメソッドを実装しようとしている型の別名です。トレイトオブジェクトは、一旦、トレイトオブジェクトを使用したら、コンパイラにはそのトレイトを実装している具体的な型を知りようがないので、オブジェクト安全でなければなりません。トレイトメソッドが具体的な Self型を返すのに、トレイトオブジェクトが Selfの具体的な型を忘れてしまったら、メソッドが元の具体的な型を使用できる手段はなくなってしまいます。同じことがトレイトを使用する時に具体的な型引数で埋められるジェネリックな型引数に対しても言えます: 具体的な型がトレイトを実装する型の一部になるのです。トレイトオブジェクトの使用を通して型が忘却されたら、そのジェネリッ

クな型引数を埋める型がなんなのか知る術はないのです。

メソッドがオブジェクト安全でないトレイトの例は、標準ライブラリの Cloneトレイトです。Clone トレイトの cloneメソッドのシグニチャは以下のような感じです:

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

String型は Cloneトレイトを実装していて、Stringのインスタンスに対して cloneメソッドを呼び出すと、Stringのインスタンスが返ってきます。同様に、Vecのインスタンスに対して cloneを呼び出すと、Vecのインスタンスが返ってきます。cloneのシグニチャは、Selfの代わりに入る型を知る必要があります。それが、戻り値の型になるからです。

コンパイラは、トレイトオブジェクトに関していつオブジェクト安全の規則を侵害するようなことを試みているかを示唆します。例えば、リスト 17-4 で Screen構造体を実装して Drawトレイトではなく、Cloneトレイトを実装した型を保持しようとしたとしましょう。こんな感じで:

```
pub struct Screen {
    pub components: Vec<Box<Clone>>,
}
```

こんなエラーになります:

```
error[E0038]: the trait `std::clone::Clone` cannot be made into an object
(エラー: `std::clone::Clone`トレイトは、オブジェクトにすることはできません)
 --> src/lib.rs:2:5
  |
2 |     pub components: Vec<Box<Clone>>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::clone::Clone` cannot be
made into an object
  |
  = note: the trait cannot require that `Self : Sized`
  (注釈: このトレイトは、`Self : Sized`を満たせません)
```

このエラーは、このようにこのトレイトをトレイトオブジェクトとして使用することはできないことを意味しています。オブジェクト安全性についてより詳しく興味があるのなら、Rust RFC 255 を参照されたし。

20.3　オブジェクト指向デザインパターンを実装する

ステートパターンは、オブジェクト指向デザインパターンの 1 つです。このパターンの肝は、値が一連の**ステートオブジェクト**で表されるなんらかの内部状態を持ち、その内部の状態に基づいて値の振る舞いが変化するというものです。ステートオブジェクトは、機能を共有します: Rust では、もちろん、オブジェクトと継承ではなく、構造体とトレイトを使用します。各ステートオブジェクトは、自身の振る舞いと別の状態に変化すべき時を司ることに責任を持ちます。ステートオブジェクトを保持する値は、状態ごとの異なる振る舞いや、いつ状態が移行するかについては何も知りません。

ステートパターンを使用することは、プログラムの業務用件が変わる時、状態を保持する値のコードや、値を使用するコードを変更する必要はないことを意味します。ステートオブジェクトの 1 つのコードを更新して、規則を変更したり、あるいはおそらくステートオブジェクトを追加する必要しかないのです。ステートデザインパターンの例と、その Rust での使用方法を見ましょう。

ブログ記事のワークフローを少しずつ実装していきます。ブログの最終的な機能は以下のような感じになるでしょう:

1. ブログ記事は、空の草稿から始まる。
2. 草稿ができたら、査読が要求される。
3. 記事が承認されたら、公開される。
4. 公開されたブログ記事だけが表示する内容を返すので、未承認の記事は、誤って公開されない。

それ以外の記事に対する変更は、効果を持つべきではありません。例えば、査読を要求する前にブログ記事の草稿を承認しようとしたら、記事は、非公開の草稿のままになるべきです。

リスト 17-11 は、このワークフローをコードの形で示しています: これは、`blog`というライブラリクレートに実装する API の使用例です。まだ `blog`クレートを実装していないので、コンパイルはできません。

ファイル名: src/main.rs

```
extern crate blog;
use blog::Post;

fn main() {
    let mut post = Post::new();

    // 今日はお昼にサラダを食べた
    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}
```

リスト 17-11: `blog`クレートに欲しい振る舞いをデモするコード

ユーザが `Post::new`で新しいブログ記事の草稿を作成できるようにしたいです。それから、草稿状態の間にブログ記事にテキストを追加できるようにしたいです。承認前に記事の内容を即座に得ようとしたら、記事はまだ草稿なので、何も起きるべきではありません。デモするためにコードに `assert_eq!`を追加しました。これに対する素晴らしい単体テストは、ブログ記事の草稿が `content`メソッドから空の文字列を返すことをアサートすることでしょうが、この例に対してテストを書くつもりはありません。

次に、記事の査読を要求できるようにしたく、また査読を待機している間は `content`に空の文字列

を返してほしいです。記事が承認を受けたら、公開されるべきです。つまり、contentを呼んだ時に記事のテキストが返されるということです。

クレートから相互作用している唯一の型は、Postだけであることに注意してください。この型はステートパターンを使用し、記事がなり得る種々の状態を表す 3 つのステートオブジェクトのうちの 1 つになる値を保持します。草稿、査読待ち、公開中です。1 つの状態から別の状態への変更は、Post型内部で管理されます。Postインスタンスのライブラリ使用者が呼び出すメソッドに呼応して状態は変化しますが、状態の変化を直接管理する必要はありません。また、ユーザは、査読前に記事を公開するなど状態を誤ることはありません。

20.3.1　Post を定義し、草稿状態で新しいインスタンスを生成する

ライブラリの実装に取り掛かりましょう！ なんらかの内容を保持する公開の Post構造体が必要なことはわかるので、構造体の定義と、関連する公開の Postインスタンスを生成する new関数から始めましょう。リスト 17-12 のようにね。また、非公開の Stateトレイトも作成します。それから、Post は stateという非公開のフィールドに、Optionで Box<State>のトレイトオブジェクトを保持します。Optionが必要な理由はすぐわかります。

ファイル名: src/lib.rs

```
pub struct Post {
    state: Option<Box<State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

リスト 17-12: Post構造体、新規 Postインスタンスを生成する new関数、Stateトレイト、Draft構造体の定義

Stateトレイトは、異なる記事の状態で共有される振る舞いを定義し、Draft、PendingReview、Published状態は全て、Stateトレイトを実装します。今は、トレイトにメソッドは何もなく、Draft が記事の初期状態にしたい状態なので、その状態だけを定義することから始めます。

新しい Postを作る時、stateフィールドは、Boxを保持する Some値にセットします。この Boxが

Draft構造体の新しいインスタンスを指します。これにより、新しい Postを作る度に、草稿から始まることが保証されます。Postの stateフィールドは非公開なので、Postを他の状態で作成する方法はないのです！ Post::new関数では、contentフィールドを新しい空の Stringにセットしています。

20.3.2　記事の内容のテキストを格納する

Post::new関数では、contentフィールドを新しい空の Stringにセットしています。リスト 17-11 は、add_textというメソッドを呼び出し、ブログ記事のテキスト内容に追加される&strを渡せるようになりたいことを示しました。これを contentフィールドを pubにして晒すのではなく、メソッドとして実装しています。これは、後ほど contentフィールドデータの読まれ方を制御するメソッドを実装できることを意味しています。add_textメソッドは非常に素直なので、リスト 17-13 の実装を impl Postブロックに追加しましょう:

ファイル名: src/lib.rs

```
# pub struct Post {
#     content: String,
# }
#
impl Post {
    // --snip--
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

リスト 17-13: 記事の contentにテキストを追加する add_textメソッドを実装する

add_textメソッドは、selfへの可変参照を取ります。というのも、add_textを呼び出した Postインスタンスを変更しているからです。それから contentの Stringに対して push_strを呼び出し、text引数を渡して保存された contentに追加しています。この振る舞いは、記事の状態によらないので、ステートパターンの一部ではありません。add_textメソッドは、stateフィールドと全く相互作用しませんが、サポートしたい振る舞いの一部ではあります。

20.3.3　草稿の記事の内容は空であることを保証する

add_textを呼び出して記事に内容を追加して後でさえ、記事はまだ草稿状態なので、それでも contentメソッドには空の文字列スライスを返してほしいです。リスト 17-11 の 8 行目で示したようにね。とりあえず、この要求を実現する最も単純な方法で contentメソッドを実装しましょう: 常に空の文字列スライスを返すことです。一旦、記事の状態を変更する能力を実装したら、公開できるように、これを後ほど変更します。ここまで、記事は草稿状態にしかなり得ないので、記事の内容は常に空のはずです。リスト 17-14 は、この仮の実装を表示しています:

ファイル名: src/lib.rs

```
# pub struct Post {
#     content: String,
# }
#
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        ""
    }
}
```

リスト 17-14: Postに常に空の文字列スライスを返す contentの仮の実装を追加する

この追加された contentメソッドとともに、リスト 17-11 の 8 行目までのコードは、想定通り動きます。

20.3.4　記事の査読を要求すると、状態が変化する

次に、記事の査読を要求する機能を追加する必要があり、これをすると、状態が Draftから PendingReviewに変わるはずです。リスト 17-15 はこのコードを示しています:

ファイル名: src/lib.rs

```
# pub struct Post {
#     state: Option<Box<State>>,
#     content: String,
# }
#
impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
```

```
30      fn request_review(self: Box<Self>) -> Box<State> {
31          self
32      }
33  }
```

リスト 17-15: Postと Stateトレイトに request_reviewメソッドを実装する

Postに selfへの可変参照を取る request_reviewという公開メソッドを与えます。それから、Post の現在の状態に対して内部の request_reviewメソッドを呼び出し、この 2 番目の request_reviewが現在の状態を消費し、新しい状態を返します。

Stateトレイトに request_reviewメソッドを追加しました; このトレイトを実装する型は全て、これで request_reviewメソッドを実装する必要があります。メソッドの第 1 引数に self、&self、&mut selfではなく、self: Box<Self>としていることに注意してください。この記法は、型を保持する Boxに対して呼ばれた時、このメソッドが有効のみになることを意味しています。この記法は、Box<Self>の所有権を奪い、古い状態を無効化するので、Postの状態値は、新しい状態に変形できます。

古い状態を消費するために、request_reviewメソッドは、状態値の所有権を奪う必要があります。ここで Postの stateフィールドの Optionが問題になるのです: takeメソッドを呼び出して、state フィールドから Some値を取り出し、その箇所に Noneを残します。なぜなら、Rust は、構造体に未代入のフィールドを持たせてくれないからです。これにより、借用するのではなく、Postの state値をムーブすることができます。それから、記事の state値をこの処理の結果にセットするのです。

self.state = self.state.request_review();のようなコードで直接 state値の所有権を得るよう設定するのではなく、一時的に Noneに stateをセットする必要があります。これにより、新しい状態に変形した後に、Postが古い state値を使えないことが保証されるのです。

Draftの request_reviewメソッドは、新しい PendingReview構造体の新しいボックスのインスタンスを返す必要があり、これが、記事が査読待ちの時の状態を表します。PendingReview構造体も request_reviewメソッドを実装しますが、何も変形はしません。むしろ、自身を返します。というのも、既に PendingReview状態にある記事の査読を要求したら、PendingReview状態に留まるべきだからです。

ようやくステートパターンの利点が見えてき始めました: state値が何であれ、Postの request_reviewメソッドは同じです。各状態は、独自の規則にのみ責任を持ちます。

Postの contentメソッドを空の文字列スライスを返すというようにそのままにします。これで Post は PendingReviewと Draft状態になり得ますが、PendingReview状態でも、同じ振る舞いが欲しいです。もうリスト 17-11 は 11 行目まで動くようになりました！

20.3.5　content の振る舞いを変化させる approve メソッドを追加する

approveメソッドは、request_reviewメソッドと類似するでしょう: 状態が承認された時に、現在の状態があるべきと言う値に stateをセットします。リスト 17-16 のようにね:

ファイル名: src/lib.rs

```
# pub struct Post {
#     state: Option<Box<State>>,
#     content: String,
# }
#
impl Post {
    // --snip--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<State>;
    fn approve(self: Box<Self>) -> Box<State>;
}

struct Draft {}

impl State for Draft {
#     fn request_review(self: Box<Self>) -> Box<State> {
#         Box::new(PendingReview {})
#     }
#
    // --snip--
    fn approve(self: Box<Self>) -> Box<State> {
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
#     fn request_review(self: Box<Self>) -> Box<State> {
#         self
#     }
#
    // --snip--
    fn approve(self: Box<Self>) -> Box<State> {
        Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) -> Box<State> {
        self
    }

```

```
    fn approve(self: Box<Self>) -> Box<State> {
        self
    }
}
```

リスト 17-16: Postと Stateトレイトに approveメソッドを実装する

Stateトレイトに approveメソッドを追加し、Published状態という Stateを実装する新しい構造体を追加します。

request_reviewのように、Draftに対して approveメソッドを呼び出したら、selfを返すので、何も効果はありません。PendingReviewに対して approveを呼び出すと、Published構造体の新しいボックス化されたインスタンスを返します。Published構造体は Stateトレイトを実装し、request_reviewメソッドと approveメソッド両方に対して、自身を返します。そのような場合には記事は、Published状態に留まるべきだからです。

さて、Postの contentメソッドを更新する必要が出てきました: 状態が Publishedなら、記事の contentフィールドの値を返したいのです; それ以外なら、空の文字列スライスを返したいです。リスト 17-17 のようにね:

ファイル名: src/lib.rs

```
# trait State {
#     fn content<'a>(&self, post: &'a Post) -> &'a str;
# }
# pub struct Post {
#     state: Option<Box<State>>,
#     content: String,
# }
#
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(&self)
    }
    // --snip--
}
```

リスト 17-17: Postの contentメソッドを更新して Stateの contentメソッドに委譲する

目的は、これらの規則全てを Stateを実装する構造体の内部に押し留めることなので、stateの値に対して contentメソッドを呼び出し、記事のインスタンス (要するに、self) を引数として渡します。そして、state値の contentメソッドを使用したことから返ってきた値を返します。

Optionに対して as_refメソッドを呼び出します。値の所有権ではなく、Option内部の値への参照が欲しいからです。stateは Option<Box<State>>なので、as_refを呼び出すと、Option<&Box<State>>が返ってきます。as_refを呼ばなければ、stateを関数引数の借用した&selfからムーブできないので、エラーになるでしょう。

さらに unwrapメソッドを呼び出し、これは絶対にパニックしないことがわかっています。何故なら、Postのメソッドが、それらのメソッドが完了した際に stateは常に Some値を含んでいることを保

証するからです。これは、コンパイラには理解不能であるものの、None値が絶対にあり得ないとわかる第 9 章の「コンパイラよりも情報を握っている場合」節で語った 1 例です。

この時点で、&Box<State>に対して contentを呼び出すと、参照外し型強制が&と Boxに働くので、究極的に contentメソッドが Stateトレイトを実装する型に対して呼び出されることになります。つまり、contentを Stateトレイト定義に追加する必要があり、そこが現在の状態に応じてどの内容を返すべきかというロジックを配置する場所です。リスト 17-18 のようにね:

ファイル名: src/lib.rs

```
# pub struct Post {
#     content: String
# }
trait State {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        ""
    }
}

// --snip--
struct Published {}

impl State for Published {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        &post.content
    }
}
```

リスト 17-18: Stateトレイトに contentメソッドを追加する

空の文字列スライスを返すデフォルト実装を contentメソッドに追加しています。これにより、Draftと PendingReview構造体に contentを実装する必要はありません。Published構造体は、contentメソッドをオーバーライドし、post.contentの値を返します。

第 10 章で議論したように、このメソッドにはライフタイム注釈が必要なことに注意してください。postへの参照を引数として取り、その postの一部への参照を返しているので、返却される参照のライフタイムは、post引数のライフタイムに関連します。

出来上がりました。要するに、リスト 17-11 はもう動くようになったのです！ ブログ記事ワークフローの規則でステートパターンを実装しました。その規則に関連するロジックは、Post中に散乱するのではなく、ステートオブジェクトに息づいています。

20.3.6　ステートパターンの代償

オブジェクト指向のステートパターンを実装して各状態の時に記事がなり得る異なる種類の振る舞いをカプセル化する能力が、Rust にあることを示しました。Postのメソッドは、種々の振る舞いについては何も知りません。コードを体系化した方法、公開された記事が振る舞うことのある様々な方法

を知るには、1 箇所のみを調べればいいのです: Published構造体の Stateトレイトの実装です。

ステートパターンを使用しない対立的な実装を作ることになったら、代わりに Postのメソッドか、あるいは記事の状態を確認し、それらの箇所 (編注: Postのメソッドのことか) の振る舞いを変更する mainコードでさえ、match式を使用したかもしれません。そうなると、複数個所を調べて記事が公開状態にあることの裏の意味全てを理解しなければならなくなります！ これは、追加した状態が増えれば、さらに上がるだけでしょう: 各 match式には、別のアームが必要になるのです。

ステートパターンでは、Postのメソッドと Postを使用する箇所で、match式が必要になることはなく、新しい状態を追加するのにも、新しい構造体を追加し、その 1 つの構造体にトレイトメソッドを実装するだけでいいわけです。

ステートパターンを使用した実装は、拡張して機能を増やすことが容易です。ステートパターンを使用するコードの管理の単純さを確認するために、以下の提言を試してみてください:

- 記事の状態を PendingReviewから Draftに戻す rejectメソッドを追加する。
- 状態が Publishedに変化させられる前に approveを 2 回呼び出す必要があるようにする。
- 記事が Draft状態の時のみテキスト内容をユーザが追加できるようにする。ヒント: ステートオブジェクトに内容について変わる可能性のあるものの責任を持たせつつも、Postを変更することには責任を持たせない。

ステートパターンの欠点の 1 つは、状態が状態間の遷移を実装しているので、状態の一部が密に結合した状態になってしまうことです。PendingReviewと Publishedの間に、Scheduledのような別の状態を追加したら、代わりに PendingReviewのコードを Scheduledに遷移するように変更しなければならないでしょう。状態が追加されても PendingReviewを変更する必要がなければ、作業が減りますが、そうすれば別のデザインパターンに切り替えることになります。

別の欠点は、ロジックの一部を重複させてしまうことです。重複を除くためには、Stateトレイトの request_reviewと approveメソッドに selfを返すデフォルト実装を試みる可能性があります; ですが、これはオブジェクト安全性を侵害するでしょう。というのも、具体的な selfが一体なんなのかトレイトには知りようがないからです。Stateをトレイトオブジェクトとして使用できるようにしたいので、メソッドにはオブジェクト安全になってもらう必要があるのです。

他の重複には、Postの request_reviewと approveメソッドの実装が似ていることが含まれます。両メソッドは Optionの stateの値に対する同じメソッドの実装に委譲していて、stateフィールドの新しい値を結果にセットします。このパターンに従う Postのメソッドが多くあれば、マクロを定義して繰り返しを排除することも考える可能性があります (マクロについては付録 D を参照)。

オブジェクト指向言語で定義されている通り忠実にステートパターンを実装することで、Rust の強みをできるだけ発揮していません。blogクレートに対して行える無効な状態と遷移をコンパイルエラーにできる変更に目を向けましょう。

20.3.6.1　状態と振る舞いを型としてコード化する

ステートパターンを再考して別の代償を得る方法をお見せします。状態と遷移を完全にカプセル化して、外部のコードに知らせないようにするよりも、状態を異なる型にコード化します。結果的に、Rust の型検査システムが、公開記事のみが許可される箇所で草稿記事の使用を試みることをコンパイルエラーを発して阻止します。

リスト 17-11 の `main`の最初の部分を考えましょう:

ファイル名: src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
}
```

それでも、`Post::new`で草稿状態の新しい記事を生成することと記事の内容にテキストを追加する能力は可能にします。しかし、空の文字列を返す草稿記事の `content`メソッドを保持する代わりに、草稿記事は、`content`メソッドを全く持たないようにします。そうすると、草稿記事の内容を得ようとしたら、メソッドが存在しないというコンパイルエラーになるでしょう。その結果、誤ってプロダクションコードで草稿記事の内容を表示することが不可能になります。そのようなコードは、コンパイルさえできないからです。リスト 17-19 は `Post`構造体、`DraftPost`構造体、さらにメソッドの定義を示しています:

ファイル名: src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
```

```
        self.content.push_str(text);
    }
}
```

リスト 17-19: contentメソッドのある Postと contentメソッドのない DraftPost

Postと DraftPost構造体どちらにもブログ記事のテキストを格納する非公開の contentフィールドがあります。状態のコード化を構造体の型に移動したので、この構造体は最早 stateフィールドを持ちません。Postは公開された記事を表し、contentを返す contentメソッドがあります。

それでも Post::new関数はありますが、Postのインスタンスを返すのではなく、DraftPostのインスタンスを返します。contentは非公開であり、Postを返す関数も存在しないので、現状 Postのインスタンスを生成することは不可能です。

DraftPost構造体には、以前のようにテキストを contentに追加できるよう add_textメソッドがありますが、DraftPostには contentメソッドが定義されていないことに注目してください！ 従って、これでプログラムは、全ての記事が草稿記事から始まり、草稿記事は表示できる内容がないことを保証します。この制限をかいくぐる試みは、全てコンパイルエラーに落ち着くでしょう。

20.3.6.2　遷移を異なる型への変形として実装する

では、どうやって公開された記事を得るのでしょうか？ 公開される前に草稿記事は査読され、承認されなければならないという規則を強制したいです。査読待ち状態の記事は、それでも内容を表示するべきではありません。別の構造体 PendingReviewPostを追加し、DraftPostに PendingReviewPostを返す request_reviewメソッドを定義し、PendingReviewPostに Postを返す approveメソッドを定義してこれらの制限を実装しましょう。リスト 17-20 のようにね:

ファイル名: src/lib.rs

```
# pub struct Post {
#     content: String,
# }
#
# pub struct DraftPost {
#     content: String,
# }
#
impl DraftPost {
    // --snip--

    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}
```

```
22
23 impl PendingReviewPost {
24     pub fn approve(self) -> Post {
25         Post {
26             content: self.content,
27         }
28     }
29 }
```

リスト 17-20: DraftPostの request_reviewを呼び出すことで生成される PendingReviewPostと、PendingReviewPostを公開された Postに変換する approveメソッド

request_reviewと approveメソッドは selfの所有権を奪い、故に DraftPostと PendingReviewPostインスタンスを消費し、それぞれ PendingReviewPostと公開された Postに変形します。このように、DraftPostインスタンスに request_reviewを呼んだ後には、DraftPostインスタンスは生きながらえず、以下同様です。PendingReviewPost構造体には、contentメソッドが定義されていないので、DraftPost同様に、その内容を読もうとするとコンパイルエラーに落ち着きます。contentメソッドが確かに定義された公開された Postインスタンスを得る唯一の方法が、PendingReviewPostに対して approveを呼び出すことであり、PendingReviewPostを得る唯一の方法が、DraftPostに request_reviewを呼び出すことなので、これでブログ記事のワークフローを型システムにコード化しました。

ですが、さらに mainにも多少小さな変更を行わなければなりません。request_reviewと approveメソッドは、呼ばれた構造体を変更するのではなく、新しいインスタンスを返すので、let post =という上書き代入をもっと追加し、返却されたインスタンスを保存する必要があります。また、草稿と査読待ち記事の内容を空の文字列でアサートすることも、する必要もありません: 最早、その状態にある記事の内容を使用しようとするコードはコンパイル不可能だからです。mainの更新されたコードは、リスト 17-21 に示されています:

ファイル名: src/main.rs

```
1  extern crate blog;
2  use blog::Post;
3
4  fn main() {
5      let mut post = Post::new();
6
7      post.add_text("I ate a salad for lunch today");
8
9      let post = post.request_review();
10
11     let post = post.approve();
12
13     assert_eq!("I ate a salad for lunch today", post.content());
14 }
```

リスト 17-21: ブログ記事ワークフローの新しい実装を使う mainの変更

postを再代入するためにmainに行う必要のあった変更は、この実装が最早、全くオブジェクト指向のステートパターンに沿っていないことを意味します: 状態間の変形は最早、Post実装内に完全にカプセル化されていません。ですが、型システムとコンパイル時に起きる型チェックのおかげでもう無効な状態があり得なくなりました。これにより、未公開の記事の内容が表示されるなどの特定のバグが、プロダクションコードに移る前に発見されることが保証されます。

blogクレートに関してこの節の冒頭で触れた追加の要求に提言される作業をそのままリスト 17-20 の後に試してみて、このバージョンのコードについてどう思うか確かめてください。この設計では、既に作業の一部が達成されている可能性があることに注意してください。

Rust は、オブジェクト指向のデザインパターンを実装する能力があるものの、状態を型システムにコード化するなどの他のパターンも、Rust では利用可能なことを確かめました。これらのパターンには、異なる代償があります。オブジェクト指向のパターンには馴染み深い可能性があるものの、問題を再考して Rust の機能の強みを活かすと、コンパイル時に一部のバグを回避できるなどの利益が得られることもあります。オブジェクト指向のパターンは、オブジェクト指向言語にはない所有権などの特定の機能により Rust では、必ずしも最善の解決策ではないでしょう。

20.4　総括

この章読了後に、あなたが Rust はオブジェクト指向言語であるか考えるかどうかに関わらず、もうトレイトオブジェクトを使用して Rust でオブジェクト指向の機能の一部を得ることができると知っています。ダイナミックディスパッチは、多少の実行時性能と引き換えにコードに柔軟性を齎 (もたら) してくれます。この柔軟性を利用してコードのメンテナンス性に寄与するオブジェクト指向パターンを実装することができます。Rust にはまた、オブジェクト指向言語にはない所有権などの他の機能もあります。オブジェクト指向パターンは、必ずしも Rust の強みを活かす最善の方法にはなりませんが、選択肢の 1 つではあります。

次は、パターンを見ます。パターンも多くの柔軟性を可能にする Rust の別の機能です。本全体を通して僅かに見かけましたが、まだその全能力は目の当たりにしていません。さあ、行きましょう！

第 21 章

パターンとマッチング

パターンは、Rust で複雑であれ、単純であれ、型の構造に一致する特別な記法です。match式や他の構文と組み合わせてパターンを使用すると、プログラムの制御フローをよりコントロールできます。パターンは、以下の組み合わせで構成されます:

- リテラル
- 分解された配列、enum、構造体、タプル
- 変数
- ワイルドカード
- プレースホルダー

これらの要素が取り組んでいるデータの形を説明し、それから値に対してマッチを行い、プログラムに正しい値があって特定のコードを実行し続けられるかどうかを決定します。

パターンを使用するには、なんらかの値と比較します。パターンが値に合致したら、コードで値の部分を使用します。コイン並び替えマシンの例のような第 6 章でパターンを使用した match式を思い出してください。値がパターンの形に当てはまったら、名前のある部品を使用できます。当てはまらなければ、パターンに紐付いたコードは実行されません。

この章は、パターンに関連するあらゆるものの参照です。パターンを使用するのが合法な箇所、**論駁** *(ろんばく)* **可能**と**論駁不可能**なパターンの違い、目撃する可能性のある色々な種類のパターン記法を講義します。章の終わりまでに、パターンを使用して多くの概念をはっきり表現する方法を知るでしょう。

21.1　パターンが使用されることのある箇所全部

Rust において、パターンはいろんな箇所に出現し、そうと気づかないうちにたくさん使用してきました！ この節は、パターンが合法な箇所全部の参照になります。

21.1.1　match アーム

第 6 章で議論したように、パターンを match式のアームで使います。正式には、match式はキーワード match、マッチ対象の値、パターンとそのアームのパターンに値が合致したら実行される式からなる 1 つ以上のマッチアームとして定義されます。以下のように:

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

match式の必須事項の 1 つは、match式の値の可能性全てが考慮されなければならないという意味で**網羅的**である必要があることです。全可能性をカバーしていると保証する 1 つの手段は、最後のアームに包括的なパターンを入れることです: 例えば、どんな値にも合致する変数名は失敗することがあり得ないので、故に残りの全ケースをカバーできます。

_という特定のパターンは何にでもマッチしますが、変数には束縛されないので、よく最後のマッチアームに使用されます。例えば、_パターンは、指定されていないあらゆる値を無視したい時に有用です。_パターンについて詳しくは、この章の後ほど、「パターンで値を無視する」節で講義します。

21.1.2　条件分岐 if let 式

第 6 章で主に if let式を 1 つの場合にしか合致しない matchと同様のものを書く省略法として使用する方法を議論しました。オプションとして、if letには if letのパターンが合致しない時に走るコードを含む対応する elseも用意できます。

リスト 18-1 は、if let、else if、else if let式を混ぜて合致させることもできることを示しています。そうすると、パターンと 1 つの値しか比較することを表現できない match式よりも柔軟性が高くなります。また、一連の if let、else if、else if letアームの条件は、お互いに関連している必要はありません。

リスト 18-1 のコードは、背景色が何になるべきかを決定するいくつかの異なる条件を連なって確認するところを示しています。この例には、実際のプログラムではユーザ入力を受け付ける可能性のある変数をハードコードされた値で生成しています。

ユーザがお気に入りの色を指定したら、その色が背景色になります。今日が火曜日なら、背景色は緑です。ユーザが年齢を文字列で指定し、数値として解析することができたら、背景色は、その数値の値によって紫かオレンジになります。どの条件も適用できなければ、背景色は青になります:

ファイル名: src/main.rs

```
1 fn main() {
2     let favorite_color: Option<&str> = None;
3     let is_tuesday = false;
4     let age: Result<u8, _> = "34".parse();
5
```

```
    if let Some(color) = favorite_color {
        // あなたのお気に入りの色、{}を背景色に使用します
        println!("Using your favorite color, {}, as the background", color);
    } else if is_tuesday {
        // 火曜日は緑の日！
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            // 紫を背景色に使用します
            println!("Using purple as the background color");
        } else {
            // オレンジを背景色に使用します
            println!("Using orange as the background color");
        }
    } else {
        // 青を背景色に使用します
        println!("Using blue as the background color");
    }
}
```

リスト 18-1: `if let`、`else if`、`else if let`、`else`を混ぜる

この条件分岐構造により、複雑な要件をサポートさせてくれます。ここにあるハードコードされた値では、この例は`Using purple as the background color`と出力するでしょう。

`match`アームのように`if let`もシャドウイングされた変数を導入できることがわかります: `if let Ok(age) = age`の行は、`Ok`列挙子の中の値を含むシャドウイングされた新しい`age`変数を導入します。つまり、`if age > 30`という条件は、そのブロック内に配置する必要があります: これら 2 つの条件を組み合わせて、`if let Ok(age) = age && age > 30`とすることはできません。30 と比較したいシャドウイングされた`age`は、波括弧で新しいスコープが始まるまで有効にならないのです。

`if let`式を使うことの欠点は、コンパイラが網羅性を確認してくれないことです。一方で`match`式ではしてくれます。最後の`else`ブロックを省略して故に、扱い忘れたケースがあっても、コンパイラは、ロジックバグの可能性を指摘してくれないでしょう。

21.1.3　`while let`条件分岐ループ

`if let`と構成が似て、`while let`条件分岐ループは、パターンが合致し続ける限り、`while`ループを走らせます。リスト 18-2 の例は、ベクタをスタックとして使用する`while let`ループを示し、ベクタの値をプッシュしたのとは逆順に出力します:

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
```

```
}
```

リスト 18-2: while letループを使って stack.pop()が Someを返す限り値を出力する

この例は、3, 2, そして 1 と出力します。popメソッドはベクタの最後の要素を取り出して Some(value)を返します。ベクタが空なら、popは Noneを返します。whileループは popが Someを返す限り、ブロックのコードを実行し続けます。popが Noneを返すと、ループは停止します。while letを使用してスタックから全ての要素を取り出せるのです。

21.1.4　for ループ

第 3 章で、Rust コードにおいては、forループが最もよく使われるループ構造だと述べましたが、forが取るパターンについてはまだ議論してませんでした。forループにおいて、直接キーワード for に続く値がパターンなので、for x in yでは、xがパターンになります。

リスト 18-3 は forループでパターンを使用して forループの一部としてタプルを分解あるいは、分離する方法をデモしています:

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{} is at index {}", value, index);
}
```

リスト 18-3: forループでパターンを使用してタプルを分解する

リスト 18-3 のコードは、以下のように出力するでしょう:

```
a is at index 0
b is at index 1
c is at index 2
```

enumerateメソッドを使用してイテレータを改造し、値とその値のイテレータでの番号をタプルに配置して生成しています。enumerateの最初の呼び出しは、タプル (0, 'a')を生成します。この値がパターン (index, value)とマッチさせられると、indexは 0、valueは'a'になり、出力の最初の行を出力するのです。

21.1.5　let 文

この章に先駆けて、matchと if letでパターンを使用することだけ明示的に議論してきましたが、実は letを含む他の箇所でもパターンを使用してきたのです。例として、この letでの率直な変数代入を考えてください:

```
let x = 5;
```

この本を通してこのような letを何百回も使用してきて、お気付きではなかったかもしれませんが、パターンを使用していたのです！ より正式には、let文はこんな見た目をしています:

```
let PATTERN = EXPRESSION;
```

`let x = 5;`のような変数名が `PATTERN`スロットにある文で、変数名は、ただ特に単純な形態のパターンなのです。Rust は式をパターンと比較し、見つかったあらゆる名前を代入します。故に、`let x = 5;`の例では、`x`は「ここでマッチしたものを変数 `x`に束縛する」ことを意味するパターンです。名前 `x`がパターンの全容なので、このパターンは実質的に「値が何であれ、全てを変数 `x`に束縛しろ」を意味します。

`let`のパターンマッチングの観点をよりはっきり確認するためにリスト 18-4 を考えてください。これは `let`でパターンを使用し、タプルを分解します:

```
let (x, y, z) = (1, 2, 3);
```

リスト 18-4: パターンを使用してタプルを分解し、3 つの変数を一度に生成する

ここでタプルに対してパターンをマッチさせています。Rust は値 `(1, 2, 3)`をパターン `(x, y, z)`と比較し、値がパターンに合致すると確認するので、`1`を `x`に、`2`を `y`に、`3`を `z`に束縛します。このタプルパターンを個別の 3 つの変数パターンが内部にネストされていると考えることもできます。

パターンの要素数がタプルの要素数と一致しない場合、全体の型が一致せず、コンパイルエラーになるでしょう。例えば、リスト 18-5 は、3 要素のタプルを 2 つの変数に分解しようとしているところを表示していて、動きません:

```
let (x, y) = (1, 2, 3);
```

リスト 18-5: 変数がタプルの要素数と一致しないパターンを間違って構成する

このコードのコンパイルを試みると、このような型エラーに落ち着きます:

```
error[E0308]: mismatched types
 --> src/main.rs:2:9
  |
2 |     let (x, y) = (1, 2, 3);
  |         ^^^^^^ expected a tuple with 3 elements, found one with 2 elements
  |
  = note: expected type `({integer}, {integer}, {integer})`
             found type `(_, _)`
```

タプルの値のうち 1 つ以上を無視したかったら、「パターンで値を無視する」節で見かけるように、`_`か`..`を使用できるでしょう。パターンに変数が多すぎるというのが問題なら、変数の数がタプルの要素数と一致するように変数を減らすことで、型を一致させることが解決策です。

21.1.6　関数の引数

関数の引数もパターンにできます。リスト 18-6 のコードは、型 `i32`の `x`という引数 1 つを取る `foo`という関数を宣言していますが、これまでに馴染み深くなっているはずです:

```
fn foo(x: i32) {
    // コードがここに来る
    // code goes here
}
```

リスト 18-6: 関数シグニチャが引数にパターンを使用している

xの部分がパターンです！ letのように、関数の引数でパターンにタプルを合致させられるでしょう。リスト 18-7 では、タプルを関数に渡したのでその中の値を分離しています:

ファイル名: src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    // 現在の位置: ({}, {})
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

リスト 18-7: タプルを分解する引数を伴う関数

このコードは `Current location: (3, 5)`と出力します。値`&(3, 5)`はパターン`&(x, y)`と合致するので、xは値 3、yは値 5になります。

また、クロージャの引数リストでも、関数の引数リストのようにパターンを使用することができます。第 13 章で議論したように、クロージャは関数に似ているからです。

この時点で、パターンを使用する方法をいくつか見てきましたが、パターンを使用できる箇所全部で同じ動作をするわけではありません。パターンが**論駁不可能**でなければならない箇所もあり、つまり、パターンが与えられたあらゆる値に合致しなければならないということです。他の状況では、論駁可能にもなり得ます。この 2 つの概念を次に議論しましょう。

21.2　論駁可能性: パターンが合致しないか

パターンには 2 つの形態があります: 論駁可能なものと論駁不可能なものです。渡される可能性のあるあらゆる値に合致するパターンは、**論駁不可能**なものです。xは何にでも合致し、故に合致に失敗することがあり得ないので、文 `let x = 5;`の xは一例です。なんらかの可能性のある値に対して合致しないことがあるパターンは、**論駁可能**なものです。一例は、式 `if let Some(x) = a_value`の `Some(x)`になるでしょう; a_value変数の値が Someではなく、Noneなら、`Some(x)`パターンは合致しないでしょう。

関数の引数、let文、forループは、値が合致しなかったら何も意味のあることをプログラムが実行できないので、論駁不可能なパターンしか受け付けられません。`if let`と `while let`式は、定義により失敗する可能性を処理することを意図したものなので、論駁可能なパターンのみを受け付けます:

条件式の機能は、成功か失敗によって異なる振る舞いをする能力にあるのです。

一般的に、論駁可能と論駁不可能なパターンの差異について心配しなくてもいいはずです; しかしながら、エラーメッセージで見かけた際に対応できるように、論駁可能性の概念に確かに慣れておく必要があります。そのような場合には、コードの意図した振る舞いに応じて、パターンかパターンを使用している構文を変える必要があるでしょう。

コンパイラが論駁不可能なパターンを必要とする箇所で論駁可能なパターンを使用しようとしたら、何が起きるかとその逆の例を見ましょう。リスト 18-8 は `let`文を示していますが、パターンには `Some(x)`と指定し、論駁可能なパターンです。予想しているかもしれませんが、このコードはエラーになります:

```
let Some(x) = some_option_value;
```

リスト 18-8: `let`で論駁可能なパターンを使用しようとする

`some_option_value`が `None`値だったなら、パターン `Some(x)`に合致しないことになり、パターンが論駁可能であることを意味します。ですが、`let`文は論駁不可能なパターンしか受け付けられません。`None`値に対してコードができる合法なことは何もないからです。コンパイル時にコンパイラは、論駁不可能なパターンが必要な箇所に論駁可能なパターンを使用しようとしたと文句を言うでしょう:

```
error[E0005]: refutable pattern in local binding: `None` not covered
(エラー: ローカル束縛に論駁可能なパターン: `None`がカバーされていません)
 -->
  |
3 | let Some(x) = some_option_value;
  |     ^^^^^^^ pattern `None` not covered
```

パターン `Some(x)`で全ての合法な値をカバーしなかった (できません！) ので、コンパイラは当然、コンパイルエラーを生成します。

論駁不可能なパターンが必要な箇所に論駁可能なパターンがある問題を修正するには、パターンを使用するコードを変えればいいのです: `let`の代わりに `if let`を使用できます。そして、パターンが合致しなかったら、コードは合法に継続する手段を残して、波括弧内のコードを飛ばすだけでしょう。リスト 18-9 は、リスト 18-8 のコードの修正方法を示しています:

```
# let some_option_value: Option<i32> = None;
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

リスト 18-9: `let`ではなく、`if let`と論駁可能なパターンを含むブロックを使用する

コードに逃げ道を与えました！ このコードは完全に合法ですが、エラーを受け取らないで論駁不可能なパターンを使用することはできないことを意味します。リスト 18-10 のように、`x`のような常にマッチするパターンを `if let`に与えたら、エラーになるでしょう:

```
if let x = 5 {
    println!("{}", x);
```

```
};
```

リスト 18-10: if letで論駁不可能なパターンを使用してみる

コンパイラは、論駁不可能なパターンと if letを使用するなんて道理が通らないと文句を言います:

```
error[E0162]: irrefutable if-let pattern
(エラー: 論駁不可能なif-letパターン)
 --> <anon>:2:8
  |
2 | if let x = 5 {
  |        ^ irrefutable pattern
```

このため、マッチアームは、論駁不可能なパターンで残りのあらゆる値に合致すべき最後のアームを除いて、論駁可能なパターンを使用しなければなりません。コンパイラは、たった 1 つしかアームのない matchで論駁不可能なパターンを使用させてくれますが、この記法は特別有用なわけではなく、より単純な letを文に置き換えることもできるでしょう。

今やパターンを使用すべき箇所と論駁可能と論駁不可能なパターンの違いを知ったので、パターンを生成するために使用できる全ての記法を講義しましょう。

21.3　パターン記法全部

本全体で、多くの異なる種類のパターンの例を見かけてきました。この節では、パターンで合法な記法全てを集め、それぞれを使用したくなる可能性がある理由について議論します。

21.3.1　リテラルにマッチする

第 6 章で目撃したように、パターンを直接リテラルに合致させられます。以下のコードが例を挙げています:

```
let x = 1;

match x {
    1 => println!("one"),       // 1
    2 => println!("two"),       // 2
    3 => println!("three"),     // 3
    _ => println!("anything"),  // なんでも
}
```

このコードは、xの値が 1 なので、oneを出力します。この記法は、コードが特定の具体的な値を得た時に行動を起こしてほしい時に有用です。

21.3.2　名前付き変数にマッチする

名前付き変数はどんな値にも合致する論駁不可能なパターンであり、この本の中で何度も使用してきました。ですが、名前付き変数を match式で使うと、厄介な問題があります。matchは新しいスコー

プを開始するので、match式内のパターンの一部として宣言された変数は、あらゆる変数同様に match構文外部の同じ名前の変数を覆い隠します。リスト 18-11 で、値 Some(5)の xという変数と値 10の変数 yを宣言しています。それから値 xに対して match式を生成します。マッチアームのパターンと最後の println!を見て、このコードを実行したり、先まで読み進める前にこのコードが何を出力するか推測してみてください:

ファイル名: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        // 50だったよ
        Some(50) => println!("Got 50"),
        // マッチしたよ
        Some(y) => println!("Matched, y = {:?}", y),
        // 規定のケース
        _ => println!("Default case, x = {:?}", x),
    }

    // 最後にはx = {}, y = {}
    println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

リスト 18-11: シャドウイングされた変数 yを導入するアームのある match式

match式を実行した時に起こることを見ていきましょう。最初のマッチアームのパターンは、xの定義された値に合致しないので、コードは継続します。

2 番目のマッチアームのパターンは、Some値内部のあらゆる値に合致する新しい yという変数を導入します。match式内の新しいスコープ内にいるので、これは新しい y変数であり、最初に値 10 で宣言した yではありません。この新しい y束縛は、Some内のあらゆる値に合致し、xにあるものはこれです。故に、この新しい yは、xの中身の値に束縛されます。その値は 5なので、そのアームの式が実行され、Matched, y = 5と出力されます。

xが Some(5)ではなく None値だったなら、最初の 2 つのアームのパターンはマッチしなかったので、値はアンダースコアに合致したでしょう。アンダースコアのアームのパターンでは x変数を導入しなかったので、その式の xは、まだシャドウイングされない外側の xのままです。この架空の場合、match は Default case, x = Noneと出力するでしょう。

match式が完了すると、スコープが終わるので、中の yのスコープも終わります。最後の println! は at the end: x = Some(5), y = 10を生成します。

シャドウイングされた変数を導入するのではなく、外側の xと yの値を比較する match式を生成するには、代わりにマッチガード条件式を使用する必要があるでしょう。マッチガードについては、後ほど、「マッチガードで追加の条件式」節で語ります。

21.3.3　複数のパターン

match式で|記法で複数のパターンに合致させることができ、これは *or* を意味します。例えば、以下のコードは xの値をマッチアームに合致させ、最初のマッチアームには *or* 選択肢があり、xの値がそのアームのどちらかの値に合致したら、そのアームのコードが走ることを意味します:

```
let x = 1;

match x {
    // 1か2
    1 | 2 => println!("one or two"),
    // 3
    3 => println!("three"),
    // なんでも
    _ => println!("anything"),
}
```

このコードは、one or twoを出力します。

21.3.4　... で値の範囲に合致させる

...記法により、限度値を含む値の範囲にマッチさせることができます。以下のコードでは、パターンが範囲内のどれかの値に合致すると、そのアームが実行されます:

```
let x = 5;

match x {
    // 1から5まで
    1 ... 5 => println!("one through five"),
    // それ以外
    _ => println!("something else"),
}
```

xが 1、2、3、4 か 5 なら、最初のアームが合致します。この記法は、|演算子を使用して同じ考えを表現するより便利です; 1 ... 5ではなく、|を使用したら、1 | 2 | 3 | 4 | 5と指定しなければならないでしょう。範囲を指定する方が遥かに短いです。特に 1 から 1000 までの値と合致させたいとかなら！

範囲は、数値か char値でのみ許可されます。コンパイラがコンパイル時に範囲が空でないことを確認しているからです。範囲が空かそうでないかコンパイラにわかる唯一の型が charか数値なのです。

こちらは、char値の範囲を使用する例です:

```
let x = 'c';

match x {
    // ASCII文字前半
    'a' ... 'j' => println!("early ASCII letter"),
```

```
    // ASCII文字後半
    'k' ... 'z' => println!("late ASCII letter"),
    // それ以外
    _ => println!("something else"),
}
```

コンパイラには cが最初のパターンの範囲にあることがわかり、early ASCII letterと出力されます。

21.3.5　分解して値を分離する

またパターンを使用して構造体、enum、タプル、参照を分解しこれらの値の異なる部分を使用することもできます。各値を見ていきましょう。

21.3.5.1　構造体を分解する

リスト 18-12 は、let文でパターンを使用して分離できる 2 つのフィールド xと yのある Point構造体を示しています:

ファイル名: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

リスト 18-12: 構造体のフィールドを個別の変数に分解する

このコードは、p変数の xと yフィールドの値に合致する変数 aと bを生成します。この例は、パターンの変数の名前は、構造体のフィールド名と合致する必要はないことを示しています。しかし、変数名をフィールド名と一致させてどの変数がどのフィールド由来のものなのか覚えやすくしたくなることは一般的なことです。

変数名をフィールドに一致させることは一般的であり、let Point{ x: x, y: y } = p;と書くことは多くの重複を含むので、構造体のフィールドと一致するパターンには省略法があります: 構造体のフィールドの名前を列挙するだけで、パターンから生成される変数は同じ名前になるのです。リスト 18-13 は、リスト 18-12 と同じ振る舞いをするコードを表示していますが、letパターンで生成される変数は aと bではなく、xと yです:

ファイル名: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

リスト 18-13: 構造体フィールド省略法で構造体のフィールドを分解する

このコードは、p変数の xと yフィールドに一致する変数 xと yを生成します。結果は、変数 xと yが p構造体の値を含むというものです。

また全フィールドに対して変数を生成するのではなく、リテラル値を構造体パターンの一部にして分解することもできます。そうすることで他のフィールドは分解して変数を生成しつつ、一部のフィールドは特定の値と一致するか確認できます。

リスト 18-14 は、Point値を 3 つの場合に区別する match式を表示しています: x軸上の点 (y = 0ならそうなる)、y軸上の点 (x = 0)、あるいはどちらでもないです:

ファイル名: src/main.rs

```
# struct Point {
#     x: i32,
#     y: i32,
# }
#
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        // x軸上の{}
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        // y軸上の{}
        Point { x: 0, y } => println!("On the y axis at {}", y),
        // どちらの軸上でもない: ({}, {})
        Point { x, y } => println!("On neither axis: ({}, {})", x, y),
    }
}
```

リスト 18-14: 分解とリテラル値との一致を 1 つのパターンで

最初のアームは、yフィールドの値がリテラル 0と一致するならマッチすると指定することで、x軸上にあるどんな点とも一致します。このパターンはそれでも、このアームのコードで使用できる x変数を生成します。

同様に、2 番目のアームは、xフィールドが 0ならマッチすると指定することで y軸上のどんな点と

も一致し、yフィールドの値には変数 yを生成します。3 番目のアームは何もリテラルを指定しないので、それ以外のあらゆる Pointに合致し、xと yフィールド両方に変数を生成します。

この例で、値 pは 0 を含む xの力で 2 番目のアームに一致するので、このコードは On the y axis at 7と出力します。

21.3.5.2　enum を分解する

例えば、第 6 章のリスト 6-5 で Option<i32>を分解するなどこの本の前半で enum を分解しました。明示的に触れなかった詳細の 1 つは、enum を分解するパターンは、enum 内に格納されているデータが定義されている手段に対応すべきということです。例として、リスト 18-15 では、リスト 6-2 から Messageenum を使用し、内部の値それぞれを分解するパターンを伴う matchを書いています:

ファイル名: src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            // Quit列挙子には分解すべきデータがない
            println!("The Quit variant has no data to destructure.")
        },
        Message::Move { x, y } => {
            println!(
                // x方向に{}、y方向に{}だけ動く
                "Move in the x direction {} and in the y direction {}",
                x,
                y
            );
        }
        // テキストメッセージ: {}
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => {
            println!(
                // 色を(R, G, B) = ({}, {}, {})に変更
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
    }
}
```

リスト 18-15: 異なる種類の値を保持する enum の列挙子を分解する

このコードは、`Change the color to red 0, green 160, blue 255`と出力します。試しに `msg`の値を変更して、他のアームのコードが走るところを確認してください。

`Message::Quit`のようなデータのない enum 列挙子については、それ以上値を分解することができません。リテラル `Message::Quit`値にマッチするだけで、変数はそのパターンに存在しません。

`Message::Move`のような構造体に似た enum の列挙子については、構造体と一致させるために指定するパターンと似たパターンを使用できます。列挙子の名前の後に波括弧を配置し、それから変数とともにフィールドを列挙するので、部品を分離してこのアームのコードで使用します。ここでは、リスト 18-13 のように省略形態を使用しています。

1 要素タプルを保持する `Message::Write`や、3 要素タプルを保持する `Message::ChangeColor`のようなタプルに似た enum の列挙子について、パターンは、タプルと一致させるために指定するパターンと類似しています。パターンの変数の数は、マッチ対象の列挙子の要素数と一致しなければなりません。

21.3.5.3　参照を分解する

パターンとマッチさせている値に参照が含まれる場合、値から参照を分解する必要があり、パターンに`&`を指定することでそうすることができます。そうすることで参照を保持する変数を得るのではなく、参照が指している値を保持する変数が得られます。このテクニックは特に、参照を走査するイテレータがあるけれども、参照ではなく、クロージャで値を使用したいクロージャで特に役に立ちます。

リスト 18-16 の例は、ベクタの `Point`インスタンスへの参照を走査し、`x`と `y`値に簡単に計算を行えるように、参照と構造体を分解します:

```
# struct Point {
#     x: i32,
#     y: i32,
# }
#
let points = vec![
    Point { x: 0, y: 0 },
    Point { x: 1, y: 5 },
    Point { x: 10, y: -3 },
];

let sum_of_squares: i32 = points
    .iter()
    .map(|&Point { x, y }| x * x + y * y)
    .sum();
```

リスト 18-16: 構造体への参照を構造体のフィールド値に分解する

このコードは、値 135 を保持する変数 `sum_of_squares`を返してきて、これは、`x`値と `y`値を 2 乗し、足し合わせ、`points`ベクタの `Point`それぞれの結果を足して 1 つの数値にした結果です。

`&Point { x, y }`に`&`が含まれていなかったら、型不一致エラーが発生していたでしょう。`iter`はそ

うして、実際の値ではなく、ベクタの要素への参照を走査するからです。そのエラーはこんな見た目でしょう:

```
error[E0308]: mismatched types
  -->
   |
14 |         .map(|Point { x, y }| x * x + y * y)
   |               ^^^^^^^^^^^^ expected &Point, found struct `Point`
   |
   = note: expected type `&Point`
              found type `Point`
```

このエラーは、コンパイラがクロージャに`&Point`と一致することを期待しているのに、`Point`への参照ではなく、`Point`値に直接一致させようとしたことを示唆しています。

21.3.5.4　構造体とタプルを分解する

分解パターンをさらに複雑な方法で混ぜて一致させ、ネストすることができます。以下の例は、構造体とタプルをタプルにネストし、全ての基本的な値を取り出している複雑な分解を表示しています:

```
# struct Point {
#     x: i32,
#     y: i32,
# }
#
let ((feet, inches), Point {x, y}) = ((3, 10), Point { x: 3, y: -10 });
```

このコードは、複雑な型を構成する部品に分解させてくれるので、興味のある値を個別に使用できます。

パターンで分解することは、構造体の各フィールドからの値のように、一部の値を他と区別して使用する便利な方法です。

21.3.6　パターンの値を無視する

`match`の最後のアームのように、パターンの値を無視して実際には何もしないけれども、残りの全ての値の可能性を考慮する包括的なものを得ることは時として有用であると認識しましたね。値全体やパターンの一部の値を無視する方法はいくつかあります: _パターンを使用すること (もう見かけました)、他のパターン内で_パターンを使用すること、アンダースコアで始まる名前を使用すること、..を使用して値の残りの部分を無視することです。これらのパターンそれぞれを使用する方法と理由を探求しましょう。

21.3.6.1　_で値全体を無視する

どんな値にも一致するけれども、値を束縛しないワイルドカードパターンとしてアンダースコア、_を使用しました。アンダースコア、_パターンは特に `match`式の最後のアームとして役に立ちますが、関数の引数も含めてあらゆるパターンで使えます。リスト 18-17 に示したようにね:

ファイル名: src/main.rs

```
fn foo(_: i32, y: i32) {
    // このコードは、y引数を使うだけです: {}
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

リスト 18-17: 関数シグニチャで_を使用する

このコードは、最初の引数として渡された値 3を完全に無視し、`This code only uses the y parameter: 4`と出力します。

特定の関数の引数が最早必要ないほとんどの場合、未使用の引数が含まれないようにシグニチャを変更するでしょう。関数の引数を無視することが特に有用なケースもあります: 例えば、トレイトを実装する際、特定の型シグニチャが必要だけれども、自分の実装の関数本体では引数の 1 つが必要ない時などです。そうすれば、代わりに名前を使った場合のようには、未使用関数引数についてコンパイラが警告することはないでしょう。

21.3.6.2　ネストされた_で値の一部を無視する

また、他のパターンの内部で_を使用して、値の一部だけを無視することもできます: 例えば、値の一部を確認したいけれども、走らせたい対応するコードでは他の部分を使用することがない時などです。リスト 18-18 は、設定の値を管理する責任を負ったコードを示しています。業務要件は、ユーザが既存の設定の変更を上書きすることはできないべきだけれども、設定をリセットし、現在設定がされてなければ設定に値を与えられるというものです。

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        // 既存の値の変更を上書きできません
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

// 設定は{:?}です
println!("setting is {:?}", setting_value);
```

リスト 18-18: Some内の値を使用する必要がない時に Some列挙子と合致するパターンでアンダースコアを使用する

このコードは、Can't overwrite an existing customized value、そして setting is Some(5)と出力するでしょう。最初のマッチアームで、どちらの Some列挙子内部の値にも合致させたり、使用する必要はありませんが、setting_valueと new_setting_valueが Some列挙子の場合を確かに確認する必要があります。その場合、何故 setting_valueを変更しないかを出力し、変更しません。

2 番目のアームの_パターンで表現される他のあらゆる場合 (setting_valueと new_setting_value どちらかが Noneなら) には、new_setting_valueには setting_valueになってほしいです。

また、1 つのパターンの複数箇所でアンダースコアを使用して特定の値を無視することもできます。リスト 18-19 は、5 要素のタプルで 2 番目と 4 番目の値を無視する例です:

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        // 何かの数値: {}, {}, {}
        println!("Some numbers: {}, {}, {}", first, third, fifth)
    },
}
```

リスト 18-19: タプルの複数の部分を無視する

このコードは、Some numbers: 2, 8, 32と出力し、値 4 と 16 は無視されます。

21.3.6.3　名前をアンダースコアで始めて未使用の変数を無視する

変数を作っているのにどこでも使用していなければ、バグかもしれないのでコンパイラは通常、警告を発します。しかし時として、まだ使用しない変数を作るのが有用なこともあります。プロトタイプを開発したり、プロジェクトを始めた直後だったりなどです。このような場面では、変数名をアンダースコアで始めることで、コンパイラに未使用変数について警告しないよう指示することができます。リスト 18-20 で 2 つの未使用変数を生成していますが、このコードを実行すると、そのうちの 1 つにしか警告が出ないはずです。

ファイル名: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

リスト 18-20: アンダースコアで変数名を始めて未使用変数警告が出るのを回避する

ここで、変数 yを使用していないことに対して警告が出ていますが、アンダースコアが接頭辞になっている変数には、使用していないという警告が出ていません。

_だけを使うのとアンダースコアで始まる名前を使うことには微妙な違いがあることに注意してください。_x記法はそれでも、値を変数に束縛する一方で、_は全く束縛しません。この差異が問題になる場合を示すために、リスト 18-21 はエラーを提示するでしょう:

```
// やあ！
```

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    // 文字列が見つかりました
    println!("found a string");
}

println!("{:?}", s);
```

リスト 18-21: それでも、アンダースコアで始まる未使用の変数は値を束縛し、値の所有権を奪う可能性がある

それでも s値は_sにムーブされ、再度 sを使用できなくするので、エラーを受け取るでしょう。ですが、アンダースコアを単独で使用すれば、値を束縛することは全くありません。sが_にムーブされないので、リスト 18-22 はエラーなくコンパイルできます:

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

println!("{:?}", s);
```

リスト 18-22: アンダースコアを使用すると、値を束縛しない

このコードは、sを何にも束縛しないので、ただ単に上手く動きます。つまり、ムーブされないのです。

21.3.6.4　.. で値の残りの部分を無視する

多くの部分がある値では、..記法を使用していくつかの部分だけを使用して残りを無視し、無視する値それぞれにアンダースコアを列挙する必要性を回避できます。..パターンは、パターンの残りで明示的にマッチさせていない値のどんな部分も無視します。リスト 18-23 では、3 次元空間で座標を保持する Point構造体があります。match式で x座標のみ処理し、yと zフィールドの値は無視したいです:

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}
```

リスト 18-23: ..で x以外の Pointのフィールド全てを無視する

x値を列挙し、それから..パターンを含んでいるだけです。これは、y: _や z: _と列挙しなければいけないのに比べて、手っ取り早いです。特に 1 つや 2 つのフィールドのみが関連する場面で多くのフィールドがある構造体に取り掛かっている時には。

..記法は、必要な数だけ値に展開されます。リスト 18-24 は、タプルで..を使用する方法を表示しています:

ファイル名: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        },
    }
}
```

リスト 18-24: タプルの最初と最後の値にだけ合致し、他の値を無視する

このコードにおいて、最初と最後の値は firstと lastに合致します。..は、途中のもの全部に合致し、無視します。

しかしながら、..を使うのは明確でなければなりません。どの値がマッチしてどの値が無視されるべきかが不明瞭なら、コンパイラはエラーを出します。リスト 18-25 は、..を曖昧に使用する例なので、コンパイルできません:

ファイル名: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {}", second)
        },
    }
}
```

リスト 18-25: ..を曖昧に使用しようとする試み

この例をコンパイルすると、こんなエラーが出ます:

```
error: `..` can only be used once per tuple or tuple struct pattern
(エラー: `..`は、タプルやタプル構造体パターン1つにつき、1回しか使用できません)
 --> src/main.rs:5:22
  |
5 |         (.., second, ..) => {
  |                      ^^
```

コンパイラが、secondの値に合致する前にタプルの幾つの値を無視し、それからそれによってさらに幾つの値を無視するかを決めることは不可能です。このコードは、2を無視し、secondに 4を束縛し、それから 8、16、32を無視したり、2と 4を無視して secondに 8を束縛し、それから 16と 32を無視するなどを意味することもあるでしょう。変数名の secondは、コンパイラにとってなんの特別な意味もないため、このように 2 箇所で..を使うのは曖昧なので、コンパイルエラーになります。

21.3.7　パターンで参照を生成する ref と ref mut

refを使用して値の所有権がパターンの変数にムーブされないように、参照を生成することに目を向けましょう。通常、パターンにマッチさせると、パターンで導入された変数は値に束縛されます。Rust の所有権規則は、その値が matchなどパターンを使用しているあらゆる場所にムーブされることを意味します。リスト 18-26 は、変数があるパターンとそれから matchの後に値全体を println!文で後ほど使用する matchの例を示しています。このコードはコンパイルに失敗します。robot_name値の一部の所有権が、最初の matchアームのパターンの name変数に移るからです。

```
let robot_name = Some(String::from("Bors"));

match robot_name {
    // 名前が見つかりました: {}
    Some(name) => println!("Found a name: {}", name),
    None => (),
}

// robot_nameは: {:?}
println!("robot_name is: {:?}", robot_name);
```

リスト 18-26: matchアームパターンで変数を生成すると、値の所有権が奪われる

robot_nameの一部の所有権が nameにムーブされたので、robot_nameに最早所有権がないために、matchの後に println!で最早 robot_nameを使用することは叶いません。

このコードを修正するために、Some(name)パターンに所有権を奪わせるのではなく、robot_nameのその部分を借用させたいです。パターンの外なら、値を借用する手段は、&で参照を生成することだと既にご認識でしょうから、解決策は Some(name)を Some(&name)に変えることだとお考えかもしれませんね。

しかしながら、「分解して値を分離する」節で見かけたように、パターンにおける&記法は参照を**生成**せず、値の既存の参照に**マッチ**します。パターンにおいて&には既にその意味があるので、&を使用してパターンで参照を生成することはできません。

その代わりに、パターンで参照を生成するには、リスト 18-27 のように、新しい変数の前に refキーワードを使用します:

```
let robot_name = Some(String::from("Bors"));

match robot_name {
    Some(ref name) => println!("Found a name: {}", name),
```

```
    None => (),
}

println!("robot_name is: {:?}", robot_name);
```

リスト 18-27: パターンの変数が値の所有権を奪わないように参照を生成する

robot_nameの Some列挙子の値が matchにムーブされないので、この例はコンパイルできます; match はムーブするのではなく、robot_nameのデータへの参照を取っただけなのです。

パターンで合致した値を可変化できるように可変参照を生成するには、&mutの代わりに ref mutを使用します。理由は今度も、パターンにおいて、前者は既存の可変参照にマッチするためにあり、新しい参照を生成しないからです。リスト 18-28 は、可変参照を生成するパターンの例です:

```
let mut robot_name = Some(String::from("Bors"));

match robot_name {
    // 別の名前
    Some(ref mut name) => *name = String::from("Another name"),
    None => (),
}

println!("robot_name is: {:?}", robot_name);
```

リスト 18-28: ref mutを使用してパターンの一部として値への可変参照を生成する

この例はコンパイルが通り、robot_name is: Some("Another name")と出力するでしょう。nameは可変参照なので、マッチアーム内で*演算子を使用して値を可変化するために参照外しする必要があります。

21.3.8　マッチガードで追加の条件式

マッチガードは、matchアームのパターンの後に指定されるパターンマッチングとともに、そのアームが選択されるのにマッチしなければならない追加の if条件です。マッチガードは、1 つのパターン単独でできるよりも複雑な考えを表現するのに役に立ちます。

この条件は、パターンで生成された変数を使用できます。リスト 18-29 は、最初のアームにパターン Some(x)と if x < 5というマッチガードもある matchを示しています:

```
let num = Some(4);

match num {
    // 5未満です: {}
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}", x),
    None => (),
}
```

リスト 18-29: パターンにマッチガードを追記する

この例は、`less than five: 4`と出力します。numが最初のアームのパターンと比較されると、`Some(4)`は`Some(x)`に一致するので、マッチします。そして、マッチガードがxの値が5未満か確認し、そうなっているので、最初のアームが選択されます。

代わりにnumが`Some(10)`だったなら、最初のアームのマッチガードは偽になったでしょう。10は5未満ではないからです。Rustはそうしたら2番目のアームに移動し、マッチするでしょう。2番目のアームにはマッチガードがなく、それ故にあらゆるSome列挙子に一致するからです。

パターン内で`if x < 5`という条件を表現する方法はありませんので、マッチガードにより、この論理を表現する能力が得られるのです。

リスト 18-11 において、マッチガードを使用すれば、パターンがシャドウイングする問題を解決できると述べました。matchの外側の変数を使用するのではなく、match式のパターン内部では新しい変数が作られることを思い出してください。その新しい変数は、外側の変数の値と比較することができないことを意味しました。リスト 18-30 は、マッチガードを使ってこの問題を修正する方法を表示しています:

ファイル名: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {:?}", n),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

リスト 18-30: マッチガードを使用して外側の変数と等しいか確認する

このコードは今度は、`Default case, x = Some(5)`と出力するでしょう。2番目のマッチアームのパターンは、外側のyを覆い隠してしまう新しい変数yを導入せず、マッチガード内で外側のyを使用できることを意味します。外側のyを覆い隠してしまう`Some(y)`としてパターンを指定するのではなく、`Some(n)`を指定しています。これにより、何も覆い隠さない新しい変数nが生成されます。matchの外側にはn変数は存在しないからです。

マッチガードの`if n == y`はパターンではなく、故に新しい変数を導入しません。このyは、新しいシャドウイングされたyではなく、外側のyであり、nとyを比較することで、外側のyと同じ値を探すことができます。

また、マッチガードで *or* 演算子の|を使用して複数のパターンを指定することもできます; マッチガードの条件は全てのパターンに適用されます。リスト 18-31 は、|を使用するパターンとマッチガードを組み合わせる優先度を示しています。この例で重要な部分は、`if y`は6にしか適用されないように見えるのに、`if y`マッチガードが4、5、そして6に適用されることです:

```
let x = 4;
let y = false;

match x {
    // はい
    4 | 5 | 6 if y => println!("yes"),
    // いいえ
    _ => println!("no"),
}
```

リスト 18-31: 複数のパターンとマッチガードを組み合わせる

マッチの条件は、xの値が 4、5、6に等しくかつ yが trueの場合だけにアームがマッチすると宣言しています。このコードが走ると、最初のアームのパターンは xが 4なので、合致しますが、マッチガード if yは偽なので、最初のアームは選ばれません。コードは 2 番目のアームに移動し、これがマッチし、このプログラムは noと出力します。理由は、if条件が最後の値の 6だけでなく、パターン全体 4 | 5 | 6に適用されるからです。言い換えると、パターンと関わるマッチガードの優先度は、以下のように振る舞います:

```
(4 | 5 | 6) if y => ...
```

以下のようにではありません:

```
4 | 5 | (6 if y) => ...
```

コードを実行後には、優先度の動作は明らかになります: マッチガードが|演算子で指定される値のリストの最後の値にしか適用されないなら、アームはマッチし、プログラムは yesと出力したでしょう。

21.3.9　@束縛

at 演算子@により、値を保持する変数を生成するのと同時にその値がパターンに一致するかを調べることができます。リスト 18-32 は、Message::Helloの idフィールドが範囲 3...7にあるかを確かめたいという例です。しかし、アームに紐付いたコードで使用できるように変数 id_variableに値を束縛もしたいです。この変数をフィールドと同じ、idと名付けることもできますが、この例では異なる名前にします:

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3...7 } => {
        // 範囲内のidが見つかりました: {}
        println!("Found an id in range: {}", id_variable)
    },
```

```
    Message::Hello { id: 10...12 } => {
        // 別の範囲内のidが見つかりました
        println!("Found an id in another range")
    },
    Message::Hello { id } => {
        // それ以外のidが見つかりました
        println!("Found some other id: {}", id)
    },
}
```

@を使用してテストしつつパターンの値に束縛する

この例は、`Found an id in range: 5`と出力します。範囲`3...7`の前に`id_variable @`と指定することで、値が範囲パターンに一致することを確認しつつ、範囲にマッチしたどんな値も捕捉しています。

パターンで範囲しか指定していない 2 番目のアームでは、アームに紐付いたコードに`id`フィールドの実際の値を含む変数はありません。`id`フィールドの値は 10、11、12 だった可能性もありますが、そのパターンに当てはまるコードは、どれなのかわかりません。パターンのコードは`id`フィールドの値を使用することは叶いません。`id`の値を変数に保存していないからです。

範囲なしに変数を指定している最後のアームでは、確かにアームのコードで使用可能な値が`id`という変数にあります。理由は、構造体フィールド省略記法を使ったからです。しかし、このアームで`id`フィールドの値に対して、最初の 2 つのアームのようには、確認を行っていません: どんな値でも、このパターンに一致するでしょう。

@を使用することで、値を調べつつ、1 つのパターン内で変数に保存させてくれるのです。

21.4　まとめ

Rust のパターンは、異なる種類のデータを区別するのに役立つという点でとても有用です。`match`式で使用されると、コンパイラはパターンが全ての可能性を網羅しているか保証し、そうでなければプログラムはコンパイルできません。`let`文や関数の引数のパターンは、その構文をより有用にし、値を分解して小さな部品にすると同時に変数に代入できるようにしてくれます。単純だったり複雑だったするパターンを生成してニーズに合わせることができます。

次の本の末尾から 2 番目の章では、Rust のいろんな機能の高度な視点に目を向けます。

第 22 章

Advanced Features

By now, you've learned the most commonly used parts of the Rust programming language. Before we do one more project in Chapter 20, we'll look at a few aspects of the language you might run into every once in a while. You can use this chapter as a reference for when you encounter any unknowns when using Rust. The features you'll learn to use in this chapter are useful in very specific situations. Although you might not reach for them often, we want to make sure you have a grasp of all the features Rust has to offer.

In this chapter, we'll cover:

- Unsafe Rust: How to opt out of some of Rust's guarantees and take responsibility for manually upholding those guarantees
- Advanced lifetimes: Syntax for complex lifetime situations
- Advanced traits: Associated types, default type parameters, fully qualified syntax, supertraits, and the newtype pattern in relation to traits
- Advanced types: More about the newtype pattern, type aliases, the *never* type, and dynamically sized types
- Advanced functions and closures: Function pointers and returning closures

It's a panoply of Rust features with something for everyone! Let's dive in!

22.1 Unsafe Rust

All the code we've discussed so far has had Rust's memory safety guarantees enforced at compile time. However, Rust has a second language hidden inside it that doesn't enforce these memory safety guarantees: it's called *unsafe Rust* and works just like regular Rust, but gives us extra superpowers.

Unsafe Rust exists because, by nature, static analysis is conservative. When the compiler tries to determine whether or not code upholds the guarantees, it's better for it to reject some valid programs rather than accepting some invalid programs. Although the code might be okay, as far as Rust is able to tell, it's not! In these cases, we can use unsafe code to tell the compiler, "trust me, I know what I'm doing." The downside is that we use it at our own risk: if we use unsafe code incorrectly, problems due to memory unsafety, such as null pointer dereferencing, can occur.

Another reason Rust has an unsafe alter ego is that the underlying computer hardware is inherently unsafe. If Rust didn't let us do unsafe operations, we couldn't do certain tasks. Rust needs to allow us to do low-level systems programming, such as directly interacting with the operating system or even writing our own operating system. Working with low-level systems programming is one of the goals of the language. Let's explore what we can do with unsafe Rust and how to do it.

22.1.1 Unsafe Superpowers

To switch to unsafe Rust, we use the `unsafe` keyword, and then start a new block that holds the unsafe code. We can take four actions in unsafe Rust, which we call *unsafe superpowers*, that we can't in safe Rust. Those superpowers include the ability to:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait

It's important to understand that `unsafe` doesn't turn off the borrow checker or disable any other of Rust's safety checks: if you use a reference in unsafe code, it will still be checked. The `unsafe` keyword only gives us access to these four features that are then not checked by the compiler for memory safety. We still get some degree of safety inside of an unsafe block.

In addition, `unsafe` does not mean the code inside the block is necessarily dangerous or that it will definitely have memory safety problems: the intent is that as the programmer, we'll ensure the code inside an `unsafe` block will access memory in a valid way.

People are fallible, and mistakes will happen, but by requiring these four unsafe operations to be inside blocks annotated with `unsafe` we'll know that any errors related to memory safety must be within an `unsafe` block. Keep `unsafe` blocks small; you'll be thankful later when you investigate memory bugs.

To isolate unsafe code as much as possible, it's best to enclose unsafe code within a safe abstraction and provide a safe API, which we'll discuss later in the chapter when we examine

unsafe functions and methods. Parts of the standard library are implemented as safe abstractions over unsafe code that has been audited. Wrapping unsafe code in a safe abstraction prevents uses of `unsafe` from leaking out into all the places that you or your users might want to use the functionality implemented with `unsafe` code, because using a safe abstraction is safe.

Let's look at each of the four unsafe superpowers in turn: we'll also look at some abstractions that provide a safe interface to unsafe code.

22.1.2 Dereferencing a Raw Pointer

In Chapter 4, in the "Dangling References" section, we mentioned that the compiler ensures references are always valid. Unsafe Rust has two new types called *raw pointers* that are similar to references. As with references, raw pointers can be immutable or mutable and are written as `*const T` and `*mut T`, respectively. The asterisk isn't the dereference operator; it's part of the type name. In the context of raw pointers, "immutable" means that the pointer can't be directly assigned to after being dereferenced.

Different from references and smart pointers, keep in mind that raw pointers:

- Are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location
- Aren't guaranteed to point to valid memory
- Are allowed to be null
- Don't implement any automatic cleanup

By opting out of having Rust enforce these guarantees, we can make the trade-off of giving up guaranteed safety to gain performance or the ability to interface with another language or hardware where Rust's guarantees don't apply.

Listing 19-1 shows how to create an immutable and a mutable raw pointer from references:

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Listing 19-1: Creating raw pointers from references

Notice that we don't include the `unsafe` keyword in this code. We can create raw pointers in safe code; we just can't dereference raw pointers outside an unsafe block, as you'll see in a bit.

We've created raw pointers by using `as` to cast an immutable and a mutable reference into their corresponding raw pointer types. Because we created them directly from references guaranteed to be valid, we know these particular raw pointers are valid, but we can't make that assumption about just any raw pointer.

Next, we'll create a raw pointer whose validity we can't be so certain of. Listing 19-2 shows how to create a raw pointer to an arbitrary location in memory. Trying to use arbitrary memory is undefined: there might be data at that address or there might not, the compiler might optimize the code so there is no memory access, or the program might error with a segmentation fault. Usually, there is no good reason to write code like this, but it is possible:

```
let address = 0x012345usize;
let r = address as *const i32;
```

Listing 19-2: Creating a raw pointer to an arbitrary memory address

Recall that we can create raw pointers in safe code, but we can't *dereference* raw pointers and read the data being pointed to. In Listing 19-3, we use the dereference operator `*` on a raw pointer that requires an `unsafe` block:

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Listing 19-3: Dereferencing raw pointers within an `unsafe` block

Creating a pointer does no harm; it's only when we try to access the value that it points at that we might end up dealing with an invalid value.

Note also that in Listing 19-1 and 19-3 we created `*const i32` and `*mut i32` raw pointers that both pointed to the same memory location, where `num` is stored. If we instead tried to create an immutable and a mutable reference to `num`, the code would not have compiled because Rust's ownership rules don't allow a mutable reference at the same time as any immutable references. With raw pointers, we can create a mutable pointer and an immutable pointer to the same location, and change data through the mutable pointer, potentially creating a data race. Be careful!

With all of these dangers, why would we ever use raw pointers? One major use case is when interfacing with C code, as you'll see in the next section, "Calling an Unsafe Function or Method." Another case is when building up safe abstractions that the borrow checker doesn't understand. We'll introduce unsafe functions and then look at an example of a safe abstraction that uses unsafe code.

22.1.3　Calling an Unsafe Function or Method

The second type of operation that requires an unsafe block is calls to unsafe functions. Unsafe functions and methods look exactly like regular functions and methods, but they have an extra `unsafe` before the rest of the definition. The `unsafe` keyword in this context indicates the function has requirements we need to uphold when we call this function, because Rust can't guarantee we've met these requirements. By calling an unsafe function within an `unsafe` block, we're saying that we've read this function's documentation and take responsibility for upholding the function's contracts.

Here is an unsafe function named `dangerous` that doesn't do anything in its body:

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

We must call the `dangerous` function within a separate `unsafe` block. If we try to call `dangerous` without the `unsafe` block, we'll get an error:

```
error[E0133]: call to unsafe function requires unsafe function or block
 -->
  |
4 |     dangerous();
  |     ^^^^^^^^^^ call to unsafe function
```

By inserting the `unsafe` block around our call to `dangerous`, we're asserting to Rust that we've read the function's documentation, we understand how to use it properly, and we've verified that we're fulfilling the contract of the function.

Bodies of unsafe functions are effectively `unsafe` blocks, so to perform other unsafe operations within an unsafe function, we don't need to add another `unsafe` block.

22.1.3.1　Creating a Safe Abstraction over Unsafe Code

Just because a function contains unsafe code doesn't mean we need to mark the entire function as unsafe. In fact, wrapping unsafe code in a safe function is a common abstraction. As an example, let's study a function from the standard library, `split_at_mut`, that requires some unsafe code and explore how we might implement it. This safe method is defined on mutable slices: it takes one slice and makes it two by splitting the slice at the index given as an argument. Listing 19-4 shows how to use `split_at_mut`:

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];
```

```
4
5  let (a, b) = r.split_at_mut(3);
6
7  assert_eq!(a, &mut [1, 2, 3]);
8  assert_eq!(b, &mut [4, 5, 6]);
```

Listing 19-4: Using the safe `split_at_mut` function

We can't implement this function using only safe Rust. An attempt might look something like Listing 19-5, which won't compile. For simplicity, we'll implement `split_at_mut` as a function rather than a method and only for slices of `i32` values rather than for a generic type `T`.

```
1  fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
2      let len = slice.len();
3
4      assert!(mid <= len);
5
6      (&mut slice[..mid],
7       &mut slice[mid..])
8  }
```

Listing 19-5: An attempted implementation of `split_at_mut` using only safe Rust

This function first gets the total length of the slice, then it asserts that the index given as a parameter is within the slice by checking that it's less than or equal to the length. The assertion means that if we pass an index that is greater than the index to split the slice at, the function will panic before it attempts to use that index.

Then we return two mutable slices in a tuple: one from the start of the original slice to the `mid` index and another from `mid` to the end of the slice.

When we try to compile the code in Listing 19-5, we'll get an error:

```
error[E0499]: cannot borrow `*slice` as mutable more than once at a time
 -->
  |
6 |     (&mut slice[..mid],
  |           ----- first mutable borrow occurs here
7 |      &mut slice[mid..])
  |           ^^^^^ second mutable borrow occurs here
8 | }
  | - first borrow ends here
```

Rust's borrow checker can't understand that we're borrowing different parts of the slice; it only knows that we're borrowing from the same slice twice. Borrowing different parts of a slice is fundamentally okay because the two slices aren't overlapping, but Rust isn't smart enough to know this. When we know code is okay, but Rust doesn't, it's time to reach for unsafe code.

Listing 19-6 shows how to use an `unsafe` block, a raw pointer, and some calls to unsafe functions to make the implementation of `split_at_mut` work:

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

Listing 19-6: Using unsafe code in the implementation of the `split_at_mut` function

Recall from the "Slices" section in Chapter 4 that slices are a pointer to some data and the length of the slice. We use the `len` method to get the length of a slice and the `as_mut_ptr` method to access the raw pointer of a slice. In this case, because we have a mutable slice to `i32` values, `as_mut_ptr` returns a raw pointer with the type `*mut i32`, which we've stored in the variable `ptr`.

We keep the assertion that the `mid` index is within the slice. Then we get to the unsafe code: the `slice::from_raw_parts_mut` function takes a raw pointer and a length, and creates a slice. We use this function to create a slice that starts from `ptr` and is `mid` items long. Then we call the `offset` method on `ptr` with `mid` as an argument to get a raw pointer that starts at `mid`, and we create a slice using that pointer and the remaining number of items after `mid` as the length.

The function `slice::from_raw_parts_mut` is unsafe because it takes a raw pointer and must trust that this pointer is valid. The `offset` method on raw pointers is also unsafe, because it must trust that the offset location is also a valid pointer. Therefore, we had to put an `unsafe` block around our calls to `slice::from_raw_parts_mut` and `offset` so we could call them. By looking at the code and by adding the assertion that `mid` must be less than or equal to `len`, we can tell that all the raw pointers used within the `unsafe` block will be valid pointers to data within the slice. This is an acceptable and appropriate use of `unsafe`.

Note that we don't need to mark the resulting `split_at_mut` function as `unsafe`, and we can call this function from safe Rust. We've created a safe abstraction to the unsafe code with an implementation of the function that uses `unsafe` code in a safe way, because it creates only valid pointers from the data this function has access to.

In contrast, the use of `slice::from_raw_parts_mut` in Listing 19-7 would likely crash when the slice is used. This code takes an arbitrary memory location and creates a slice ten thousand items long:

```
use std::slice;

let address = 0x012345usize;
```

```
let r = address as *mut i32;

let slice = unsafe {
    slice::from_raw_parts_mut(r, 10000)
};
```

Listing 19-7: Creating a slice from an arbitrary memory location

We don't own the memory at this arbitrary location, and there is no guarantee that the slice this code creates contains valid `i32` values. Attempting to use `slice` as though it's a valid slice results in undefined behavior.

22.1.3.2　Using `extern` Functions to Call External Code

Sometimes, your Rust code might need to interact with code written in another language. For this, Rust has a keyword, `extern`, that facilitates the creation and use of a *Foreign Function Interface (FFI)*. An FFI is a way for a programming language to define functions and enable a different (foreign) programming language to call those functions.

Listing 19-8 demonstrates how to set up an integration with the `abs` function from the C standard library. Functions declared within `extern` blocks are always unsafe to call from Rust code. The reason is that other languages don't enforce Rust's rules and guarantees, and Rust can't check them, so responsibility falls on the programmer to ensure safety.

Filename: src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

Listing 19-8: Declaring and calling an `extern` function defined in another language

Within the `extern "C"` block, we list the names and signatures of external functions from another language we want to call. The `"C"` part defines which *application binary interface (ABI)* the external function uses: the ABI defines how to call the function at the assembly level. The `"C"` ABI is the most common and follows the C programming language's ABI.

22.1.3.3　Calling Rust Functions from Other Languages

We can also use `extern` to create an interface that allows other languages to call Rust functions. Instead of an `extern` block, we add the `extern` keyword and specify the ABI to use just before the `fn` keyword. We also need to add a `#[no_mangle]` annotation to tell the Rust compiler not to

mangle the name of this function. *Mangling* is when a compiler changes the name we've given a function to a different name that contains more information for other parts of the compilation process to consume but is less human readable. Every programming language compiler mangles names slightly differently, so for a Rust function to be nameable by other languages, we must disable the Rust compiler's name mangling.

In the following example, we make the `call_from_c` function accessible from C code, after it's compiled to a shared library and linked from C:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

This usage of `extern` does not require `unsafe`.

22.1.4　Accessing or Modifying a Mutable Static Variable

Until now, we've not talked about *global variables*, which Rust does support but can be problematic with Rust's ownership rules. If two threads are accessing the same mutable global variable, it can cause a data race.

In Rust, global variables are called *static* variables. Listing 19-9 shows an example declaration and use of a static variable with a string slice as a value:

Filename: src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

Listing 19-9: Defining and using an immutable static variable

Static variables are similar to constants, which we discussed in the "Differences Between Variables and Constants" section in Chapter 3. The names of static variables are in `SCREAMING_SNAKE_CASE` by convention, and we *must* annotate the variable's type, which is `&'static str` in this example. Static variables can only store references with the `'static` lifetime, which means the Rust compiler can figure out the lifetime; we don't need to annotate it explicitly. Accessing an immutable static variable is safe.

Constants and immutable static variables might seem similar, but a subtle difference is that values in a static variable have a fixed address in memory. Using the value will always access the same data. Constants, on the other hand, are allowed to duplicate their data whenever they're used.

Another difference between constants and static variables is that static variables can be mutable. Accessing and modifying mutable static variables is *unsafe*. Listing 19-10 shows how to declare, access, and modify a mutable static variable named `COUNTER`:

Filename: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

Listing 19-10: Reading from or writing to a mutable static variable is unsafe

As with regular variables, we specify mutability using the `mut` keyword. Any code that reads or writes from `COUNTER` must be within an `unsafe` block. This code compiles and prints `COUNTER: 3` as we would expect because it's single threaded. Having multiple threads access `COUNTER` would likely result in data races.

With mutable data that is globally accessible, it's difficult to ensure there are no data races, which is why Rust considers mutable static variables to be unsafe. Where possible, it's preferable to use the concurrency techniques and thread-safe smart pointers we discussed in Chapter 16, so the compiler checks that data accessed from different threads is done safely.

22.1.5　Implementing an Unsafe Trait

The final action that only works with `unsafe` is implementing an unsafe trait. A trait is unsafe when at least one of its methods has some invariant that the compiler can't verify. We can declare that a trait is `unsafe` by adding the `unsafe` keyword before `trait`; then implementation of the trait must be marked as `unsafe` too, as shown in Listing 19-11:

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}
```

Listing 19-11: Defining and implementing an unsafe trait

By using `unsafe impl`, we're promising that we'll uphold the invariants that the compiler can't verify.

As an example, recall the `Sync` and `Send` marker traits we discussed in the "Extensible Concurrency with the `Sync` and `Send` Traits" section in Chapter 16: the compiler implements these traits automatically if our types are composed entirely of `Send` and `Sync` types. If we implement a type that contains a type that is not `Send` or `Sync`, such as raw pointers, and we want to mark that type as `Send` or `Sync`, we must use `unsafe`. Rust can't verify that our type upholds the guarantees that it can be safely sent across threads or accessed from multiple threads; therefore, we need to do those checks manually and indicate as such with `unsafe`.

22.1.6 When to Use Unsafe Code

Using `unsafe` to take one of the four actions (superpowers) just discussed isn't wrong or even frowned upon. But it is trickier to get `unsafe` code correct because the compiler can't help uphold memory safety. When you have a reason to use `unsafe` code, you can do so, and having the explicit `unsafe` annotation makes it easier to track down the source of problems if they occur.

22.2 Advanced Lifetimes

In Chapter 10 in the "Validating References with Lifetimes" section, you learned how to annotate references with lifetime parameters to tell Rust how lifetimes of different references relate. You saw how every reference has a lifetime, but most of the time, Rust will let you elide lifetimes. Now we'll look at three advanced features of lifetimes that we haven't covered yet:

- Lifetime subtyping: Ensures that one lifetime outlives another lifetime
- Lifetime bounds: Specifies a lifetime for a reference to a generic type
- Inference of trait object lifetimes: How the compiler infers trait object lifetimes and when they need to be specified

22.2.1 Lifetime Subtyping Ensures One Lifetime Outlives Another

Lifetime subtyping specifies that one lifetime should outlive another lifetime. To explore lifetime subtyping, imagine we want to write a parser. We'll use a structure called `Context` that holds a reference to the string we're parsing. We'll write a parser that will parse this string and return success or failure. The parser will need to borrow the `Context` to do the parsing. Listing 19-12 implements this parser code, except the code doesn't have the required lifetime annotations, so it won't compile:

Filename: src/lib.rs

```
struct Context(&str);

struct Parser {
    context: &Context,
}

impl Parser {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Listing 19-12: Defining a parser without lifetime annotations

Compiling the code results in errors because Rust expects lifetime parameters on the string slice in `Context` and the reference to a `Context` in `Parser`.

For simplicity's sake, the `parse` function returns `Result<(), &str>`. That is, the function will do nothing on success, and on failure will return the part of the string slice that didn't parse correctly. A real implementation would provide more error information and would return a structured data type when parsing succeeds. We won't be discussing those details because they aren't relevant to the lifetimes part of this example.

To keep this code simple, we won't write any parsing logic. However, it's very likely that somewhere in the parsing logic we would handle invalid input by returning an error that references the part of the input that is invalid; this reference is what makes the code example interesting in regard to lifetimes. Let's pretend that the logic of our parser is that the input is invalid after the first byte. Note that this code might panic if the first byte is not on a valid character boundary; again, we're simplifying the example to focus on the lifetimes involved.

To get this code to compile, we need to fill in the lifetime parameters for the string slice in `Context` and the reference to the `Context` in `Parser`. The most straightforward way to do this is to use the same lifetime everywhere, as shown in Listing 19-13:

Filename: src/lib.rs

```
struct Context<'a>(&'a str);

struct Parser<'a> {
    context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Listing 19-13: Annotating all references in `Context` and `Parser` with the same lifetime parameter

This code compiles just fine. It tells Rust that a `Parser` holds a reference to a `Context` with lifetime `'a`, and that `Context` holds a string slice that also lives as long as the reference to the `Context` in `Parser`. Rust's compiler error message stated that lifetime parameters were required for these references, and we've now added lifetime parameters.

Next, in Listing 19-14, we'll add a function that takes an instance of `Context`, uses a `Parser` to parse that context, and returns what `parse` returns. This code doesn't quite work:

Filename: src/lib.rs

```
fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}
```

Listing 19-14: An attempt to add a `parse_context` function that takes a `Context` and uses a `Parser`

We get two verbose errors when we try to compile the code with the addition of the `parse_context` function:

```
error[E0597]: borrowed value does not live long enough
  --> src/lib.rs:14:5
   |
14 |     Parser { context: &context }.parse()
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ does not live long enough
15 | }
   | - temporary value only lives until here
   |
note: borrowed value must be valid for the anonymous lifetime #1 defined on the
    function body at 13:1...
  --> src/lib.rs:13:1
   |
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
   | |_^

error[E0597]: `context` does not live long enough
  --> src/lib.rs:14:24
   |
14 |     Parser { context: &context }.parse()
   |                       ^^^^^^^ does not live long enough
15 | }
   | - borrowed value only lives until here
   |
note: borrowed value must be valid for the anonymous lifetime #1 defined on the
    function body at 13:1...
  --> src/lib.rs:13:1
   |
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
```

```
15 | | }
   | |_^
```

These errors state that the `Parser` instance that is created and the `context` parameter live only until the end of the `parse_context` function. But they both need to live for the entire lifetime of the function.

In other words, `Parser` and `context` need to *outlive* the entire function and be valid before the function starts as well as after it ends for all the references in this code to always be valid. The `Parser` we're creating and the `context` parameter go out of scope at the end of the function, because `parse_context` takes ownership of `context`.

To figure out why these errors occur, let's look at the definitions in Listing 19-13 again, specifically the references in the signature of the `parse` method:

```
1    fn parse(&self) -> Result<(), &str> {
```

Remember the elision rules? If we annotate the lifetimes of the references rather than eliding, the signature would be as follows:

```
1    fn parse<'a>(&'a self) -> Result<(), &'a str> {
```

That is, the error part of the return value of `parse` has a lifetime that is tied to the lifetime of the `Parser` instance (that of `&self` in the `parse` method signature). That makes sense: the returned string slice references the string slice in the `Context` instance held by the `Parser`, and the definition of the `Parser` struct specifies that the lifetime of the reference to `Context` and the lifetime of the string slice that `Context` holds should be the same.

The problem is that the `parse_context` function returns the value returned from `parse`, so the lifetime of the return value of `parse_context` is tied to the lifetime of the `Parser` as well. But the `Parser` instance created in the `parse_context` function won't live past the end of the function (it's temporary), and `context` will go out of scope at the end of the function (`parse_context` takes ownership of it).

Rust thinks we're trying to return a reference to a value that goes out of scope at the end of the function, because we annotated all the lifetimes with the same lifetime parameter. The annotations told Rust the lifetime of the string slice that `Context` holds is the same as that of the lifetime of the reference to `Context` that `Parser` holds.

The `parse_context` function can't see that within the `parse` function, the string slice returned will outlive `Context` and `Parser`, and that the reference `parse_context` returns refers to the string slice, not to `Context` or `Parser`.

By knowing what the implementation of `parse` does, we know that the only reason the return value of `parse` is tied to the `Parser` is because it's referencing the `Parser`'s `Context`, which is referencing the string slice. So, it's really the lifetime of the string slice that `parse_context` needs

to care about. We need a way to tell Rust that the string slice in `Context` and the reference to the `Context` in `Parser` have different lifetimes and that the return value of `parse_context` is tied to the lifetime of the string slice in `Context`.

First, we'll try giving `Parser` and `Context` different lifetime parameters, as shown in Listing 19-15. We'll use `'s` and `'c` as lifetime parameter names to clarify which lifetime goes with the string slice in `Context` and which goes with the reference to `Context` in `Parser`. Note that this solution won't completely fix the problem, but it's a start. We'll look at why this fix isn't sufficient when we try to compile.

Filename: src/lib.rs

```
struct Context<'s>(&'s str);

struct Parser<'c, 's> {
    context: &'c Context<'s>,
}

impl<'c, 's> Parser<'c, 's> {
    fn parse(&self) -> Result<(), &'s str> {
        Err(&self.context.0[1..])
    }
}

fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}
```

Listing 19-15: Specifying different lifetime parameters for the references to the string slice and to `Context`

We've annotated the lifetimes of the references in all the same places that we annotated them in Listing 19-13. But this time we used different parameters depending on whether the reference goes with the string slice or with `Context`. We've also added an annotation to the string slice part of the return value of `parse` to indicate that it goes with the lifetime of the string slice in `Context`.

When we try to compile now, we get the following error:

```
error[E0491]: in type `&'c Context<'s>`, reference has a longer lifetime than the data
    it references
 --> src/lib.rs:4:5
  |
4 |     context: &'c Context<'s>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^
  |
note: the pointer is valid for the lifetime 'c as defined on the struct at 3:1
 --> src/lib.rs:3:1
  |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
```

```
5 | | }
  | |_^
note: but the referenced data is only valid for the lifetime 's as defined on the
    struct at 3:1
 --> src/lib.rs:3:1
  |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
  | |_^
```

Rust doesn't know of any relationship between `'c` and `'s`. To be valid, the referenced data in `Context` with lifetime `'s` needs to be constrained to guarantee that it lives longer than the reference with lifetime `'c`. If `'s` is not longer than `'c`, the reference to `Context` might not be valid.

Now we get to the point of this section: the Rust feature *lifetime subtyping* specifies that one lifetime parameter lives at least as long as another one. In the angle brackets where we declare lifetime parameters, we can declare a lifetime `'a` as usual and declare a lifetime `'b` that lives at least as long as `'a` by declaring `'b` using the syntax `'b: 'a`.

In our definition of `Parser`, to say that `'s` (the lifetime of the string slice) is guaranteed to live at least as long as `'c` (the lifetime of the reference to `Context`), we change the lifetime declarations to look like this:

Filename: src/lib.rs

```
# struct Context<'a>(&'a str);
#
struct Parser<'c, 's: 'c> {
    context: &'c Context<'s>,
}
```

Now the reference to `Context` in the `Parser` and the reference to the string slice in the `Context` have different lifetimes; we've ensured that the lifetime of the string slice is longer than the reference to the `Context`.

That was a very long-winded example, but as we mentioned at the start of this chapter, Rust's advanced features are very specific. You won't often need the syntax we described in this example, but in such situations, you'll know how to refer to something you have a reference to.

22.2.2　Lifetime Bounds on References to Generic Types

In the "Trait Bounds" section in Chapter 10, we discussed using trait bounds on generic types. We can also add lifetime parameters as constraints on generic types; these are called *lifetime bounds*. Lifetime bounds help Rust verify that references in generic types won't outlive the data they're referencing.

As an example, consider a type that is a wrapper over references. Recall the `RefCell<T>` type from the "`RefCell<T>` and the Interior Mutability Pattern" section in Chapter 15: its `borrow` and `borrow_mut` methods return the types `Ref` and `RefMut`, respectively. These types are wrappers over references that keep track of the borrowing rules at runtime. The definition of the `Ref` struct is shown in Listing 19-16, without lifetime bounds for now:

Filename: src/lib.rs

```
struct Ref<'a, T>(&'a T);
```

Listing 19-16: Defining a struct to wrap a reference to a generic type, without lifetime bounds to start

Without explicitly constraining the lifetime `'a` in relation to the generic parameter `T`, Rust will error because it doesn't know how long the generic type `T` will live:

```
error[E0309]: the parameter type `T` may not live long enough
 --> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  |                   ^^^^^^
  |
  = help: consider adding an explicit lifetime bound `T: 'a`...
note: ...so that the reference type `&'a T` does not outlive the data it points at
 --> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  |                   ^^^^^^
```

Because `T` can be any type, `T` could be a reference or a type that holds one or more references, each of which could have their own lifetimes. Rust can't be sure `T` will live as long as `'a`.

Fortunately, the error provides helpful advice on how to specify the lifetime bound in this case:

```
consider adding an explicit lifetime bound `T: 'a` so that the reference type
`&'a T` does not outlive the data it points at
```

Listing 19-17 shows how to apply this advice by specifying the lifetime bound when we declare the generic type `T`:

```
struct Ref<'a, T: 'a>(&'a T);
```

Listing 19-17: Adding lifetime bounds on `T` to specify that any references in `T` live at least as long as `'a`

This code now compiles because the `T: 'a` syntax specifies that `T` can be any type, but if it contains any references, the references must live at least as long as `'a`.

We could solve this problem in a different way, as shown in the definition of a `StaticRef` struct in Listing 19-18, by adding the `'static` lifetime bound on `T`. This means if `T` contains any references, they must have the `'static` lifetime:

```
struct StaticRef<T: 'static>(&'static T);
```

Listing 19-18: Adding a `'static` lifetime bound to `T` to constrain `T` to types that have only `'static` references or no references

Because `'static` means the reference must live as long as the entire program, a type that contains no references meets the criteria of all references living as long as the entire program (because there are no references). For the borrow checker concerned about references living long enough, there is no real distinction between a type that has no references and a type that has references that live forever: both are the same for determining whether or not a reference has a shorter lifetime than what it refers to.

22.2.3 Inference of Trait Object Lifetimes

In Chapter 17 in the "Using Trait Objects that Allow for Values of Different Types" section, we discussed trait objects, consisting of a trait behind a reference, that allow us to use dynamic dispatch. We haven't yet discussed what happens if the type implementing the trait in the trait object has a lifetime of its own. Consider Listing 19-19 where we have a trait `Red` and a struct `Ball`. The `Ball` struct holds a reference (and thus has a lifetime parameter) and also implements trait `Red`. We want to use an instance of `Ball` as the trait object `Box<Red>`:

Filename: src/main.rs

```
trait Red { }

struct Ball<'a> {
    diameter: &'a i32,
}

impl<'a> Red for Ball<'a> { }

fn main() {
    let num = 5;

    let obj = Box::new(Ball { diameter: &num }) as Box<Red>;
}
```

Listing 19-19: Using a type that has a lifetime parameter with a trait object

This code compiles without any errors, even though we haven't explicitly annotated the lifetimes involved in `obj`. This code works because there are rules for working with lifetimes and trait objects:

- The default lifetime of a trait object is `'static`.
- With `&'a Trait` or `&'a mut Trait`, the default lifetime of the trait object is `'a`.

- With a single `T: 'a` clause, the default lifetime of the trait object is `'a`.
- With multiple `T: 'a`-like clauses, there is no default lifetime; we must be explicit.

When we must be explicit, we can add a lifetime bound on a trait object like `Box<Red>` using the syntax `Box<Red + 'static>` or `Box<Red + 'a>`, depending on whether the reference lives for the entire program or not. As with the other bounds, the syntax adding a lifetime bound means that any implementor of the `Red` trait that has references inside the type must have the same lifetime specified in the trait object bounds as those references.

Next, let's look at some other advanced features that manage traits.

22.3 Advanced Traits

We first covered traits in the "Traits: Defining Shared Behavior" section of Chapter 10, but as with lifetimes, we didn't discuss the more advanced details. Now that you know more about Rust, we can get into the nitty-gritty.

22.3.1 Associated Types Specify Placeholder Types in Trait Definitions

Associated types connect a type placeholder with a trait such that the trait method definitions can use these placeholder types in their signatures. The implementor of a trait will specify the concrete type to be used in this type's place for the particular implementation. That way, we can define a trait that uses some types without needing to know exactly what those types are until the trait is implemented.

We've described most of the advanced features in this chapter as being rarely needed. Associated types are somewhere in the middle: they're used more rarely than features explained in the rest of the book, but more commonly than many of the other features discussed in this chapter.

One example of a trait with an associated type is the `Iterator` trait that the standard library provides. The associated type is named `Item` and stands in for the type of the values the type implementing the `Iterator` trait is iterating over. In "The `Iterator` Trait and the `next` Method" section of Chapter 13, we mentioned that the definition of the `Iterator` trait is as shown in Listing 19-20:

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Listing 19-20: The definition of the `Iterator` trait that has an associated type `Item`

The type `Item` is a placeholder type, and the `next` method's definition shows that it will return values of type `Option<Self::Item>`. Implementors of the `Iterator` trait will specify the concrete

type for `Item`, and the `next` method will return an `Option` containing a value of that concrete type.

22.3.1.1 Associated Types vs. Generics

Associated types might seem like a similar concept to generics, in that they allow us to define a function without specifying what types it can handle. So why use associated types?

Let's examine the difference between the two concepts with an example from Chapter 13 that implements the `Iterator` trait on the `Counter` struct. In Listing 13-21, we specified that the `Item` type was `u32`:

Filename: src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
```

This syntax seems comparable to generics. So why not just define the `Iterator` trait with generics, as shown in Listing 19-21?

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

Listing 19-21: A hypothetical definition of the `Iterator` trait using generics

The difference is that when using generics, as in Listing 19-21, we must annotate the types in each implementation. The reason is that we can also implement `Iterator<String> for Counter` or any other type, which would give us multiple implementations of `Iterator` for `Counter`. In other words, when a trait has a generic parameter, it can be implemented for a type multiple times, changing the concrete types of the generic type parameters each time. When we use the `next` method on `Counter`, we would have to provide type annotations to indicate which implementation of `Iterator` we want to use.

With associated types, we don't need to annotate types because we can't implement a trait on a type multiple times. In Listing 19-20 with the definition that uses associated types, we can only choose what the type of `Item` will be once, because there can only be one `impl Iterator for Counter`. We don't have to specify that we want an iterator of `u32` values everywhere that we call `next` on `Counter`.

22.3.2 Default Generic Type Parameters and Operator Overloading

When we use generic type parameters, we can specify a default concrete type for the generic type. This eliminates the need for implementors of the trait to specify a concrete type if the

default type works. The syntax for specifying a default type for a generic type is `<PlaceholderType =ConcreteType>` when declaring the generic type.

A great example of a situation where this technique is useful is with operator overloading. *Operator overloading* is customizing the behavior of an operator (such as `+`) in particular situations.

Rust doesn't allow you to create your own operators or overload arbitrary operators. But you can overload the operations and corresponding traits listed in `std::ops` by implementing the traits associated with the operator. For example, in Listing 19-22 we overload the `+` operator to add two `Point` instances together. We do this by implementing the `Add` trait on a `Point` struct:

Filename: src/main.rs

```
use std::ops::Add;

#[derive(Debug,PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
               Point { x: 3, y: 3 });
}
```

Listing 19-22: Implementing the `Add` trait to overload the `+` operator for `Point` instances

The `add` method adds the `x` values of two `Point` instances and the `y` values of two `Point` instances to create a new `Point`. The `Add` trait has an associated type named `Output` that determines the type returned from the `add` method.

The default generic type in this code is within the `Add` trait. Here is its definition:

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

This code should look generally familiar: a trait with one method and an associated type. The

new part is `RHS=Self` in the angle brackets: this syntax is called *default type parameters*. The `RHS` generic type parameter (short for "right hand side") defines the type of the `rhs` parameter in the `add` method. If we don't specify a concrete type for `RHS` when we implement the `Add` trait, the type of `RHS` will default to `Self`, which will be the type we're implementing `Add` on.

When we implemented `Add` for `Point`, we used the default for `RHS` because we wanted to add two `Point` instances. Let's look at an example of implementing the `Add` trait where we want to customize the `RHS` type rather than using the default.

We have two structs holding values in different units, `Millimeters` and `Meters`. We want to add values in millimeters to values in meters and have the implementation of `Add` do the conversion correctly. We can implement `Add` for `Millimeters` with `Meters` as the `RHS`, as shown in Listing 19-23:

Filename: src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Listing 19-23: Implementing the `Add` trait on `Millimeters` to add `Millimeters` to `Meters`

To add `Millimeters` and `Meters`, we specify `impl Add<Meters>` to set the value of the `RHS` type parameter instead of using the default of `Self`.

We use default type parameters in two main ways:

- To extend a type without breaking existing code
- To allow customization in specific cases most users won't need

The standard library's `Add` trait is an example of the second purpose: usually, you'll add two like types, but the `Add` trait provides the ability for customizing beyond that. Using a default type parameter in the `Add` trait definition means you don't have to specify the extra parameter most of the time. In other words, a bit of implementation boilerplate isn't needed, making it easier to use the trait.

The first purpose is similar to the second but in reverse: if we want to add a type parameter to an existing trait, we can give it a default to let us extend the functionality of the trait without breaking the existing implementation code.

22.3.3 Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name

Nothing in Rust prevents a trait from having a method with the same name as another trait's method, nor does Rust prevent us from implementing both traits on one type. It's also possible to implement a method directly on the type with the same name as methods from traits.

When calling methods with the same name, we need to tell Rust which one we want to use. Consider the code in Listing 19-24 where we've defined two traits, `Pilot` and `Wizard`, that both have a method called `fly`. We then implement both traits on a type `Human` that already has a method named `fly` implemented on it. Each `fly` method does something different:

Filename: src/main.rs

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

Listing 19-24: Two traits defined to have a `fly` method and implementations of those traits on the `Human` type in addition to a `fly` method on `Human` directly

When we call `fly` on an instance of `Human`, the compiler defaults to calling the method that is directly implemented on the type, as shown in Listing 19-25:

Filename: src/main.rs

```
# trait Pilot {
#     fn fly(&self);
# }
#
# trait Wizard {
#     fn fly(&self);
# }
#
# struct Human;
#
# impl Pilot for Human {
#     fn fly(&self) {
#         println!("This is your captain speaking.");
#     }
# }
#
# impl Wizard for Human {
#     fn fly(&self) {
#         println!("Up!");
#     }
# }
#
# impl Human {
#     fn fly(&self) {
#         println!("*waving arms furiously*");
#     }
# }
#
fn main() {
    let person = Human;
    person.fly();
}
```

Listing 19-25: Calling `fly` on an instance of `Human`

Running this code will print `*waving arms furiously*`, which shows that Rust called the `fly` method implemented on `Human` directly.

To call the `fly` methods from either the `Pilot` trait or the `Wizard` trait, we need to use more explicit syntax to specify which `fly` method we mean. Listing 19-26 demonstrates this syntax:

Filename: src/main.rs

```
# trait Pilot {
#     fn fly(&self);
# }
#
# trait Wizard {
#     fn fly(&self);
# }
#
# struct Human;
#
# impl Pilot for Human {
```

```
#     fn fly(&self) {
#         println!("This is your captain speaking.");
#     }
# }
#
# impl Wizard for Human {
#     fn fly(&self) {
#         println!("Up!");
#     }
# }
#
# impl Human {
#     fn fly(&self) {
#         println!("*waving arms furiously*");
#     }
# }
#
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Listing 19-26: Specifying which trait's `fly` method we want to call

Specifying the trait name before the method name clarifies to Rust which implementation of `fly` we want to call. We could also write `Human::fly(&person)`, which is equivalent to `person.fly()` that we used in Listing 19-26 but is a bit longer to write if we don't need to disambiguate.

Running this code prints the following:

```
This is your captain speaking.
Up!
*waving arms furiously*
```

Because the `fly` method takes a `self` parameter, if we had two *types* that both implement one *trait*, Rust can figure out which implementation of a trait to use based on the type of `self`.

However, associated functions that are part of traits don't have a `self` parameter. When two types in the same scope implement that trait, Rust can't figure out which type we mean unless we use *fully qualified syntax*. For example, the `Animal` trait in Listing 19-27 has the associated function `baby_name`, the implementation of `Animal` for the struct `Dog`, and the associated function `baby_name` defined on `Dog` directly:

Filename: src/main.rs

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;
```

```
6
7  impl Dog {
8      fn baby_name() -> String {
9          String::from("Spot")
10     }
11 }
12
13 impl Animal for Dog {
14     fn baby_name() -> String {
15         String::from("puppy")
16     }
17 }
18
19 fn main() {
20     println!("A baby dog is called a {}", Dog::baby_name());
21 }
```

Listing 19-27: A trait with an associated function and a type that has an associated function with the same name that also implements the trait

This code is for an animal shelter that wants to name all puppies Spot, which is implemented in the `baby_name` associated function that is defined on `Dog`. The `Dog` type also implements the trait `Animal`, which describes characteristics that all animals have. Baby dogs are called puppies, and that is expressed in the implementation of the `Animal` trait on `Dog` in the `baby_name` function associated with the `Animal` trait.

In `main`, we call the `Dog::baby_name` function, which calls the associated function defined on `Dog` directly. This code prints the following:

```
A baby dog is called a Spot
```

This output isn't what we wanted. We want to call the `baby_name` function that is part of the `Animal` trait that we implemented on `Dog` so the code prints `A baby dog is called a puppy`. The technique of specifying the trait name that we used in Listing 19-26 doesn't help here; if we change `main` to the code in Listing 19-28, we'll get a compilation error:

Filename: src/main.rs

```
1 fn main() {
2     println!("A baby dog is called a {}", Animal::baby_name());
3 }
```

Listing 19-28: Attempting to call the `baby_name` function from the `Animal` trait, but Rust doesn't know which implementation to use

Because `Animal::baby_name` is an associated function rather than a method, and thus doesn't have a `self` parameter, Rust can't figure out which implementation of `Animal::baby_name` we want. We'll get this compiler error:

```
error[E0283]: type annotations required: cannot resolve `_: Animal`
  --> src/main.rs:20:43
   |
20 |     println!("A baby dog is called a {}", Animal::baby_name());
   |                                           ^^^^^^^^^^^^^^^^
   |
   = note: required by `Animal::baby_name`
```

To disambiguate and tell Rust that we want to use the implementation of `Animal` for `Dog`, we need to use *fully qualified syntax*, which is the most specific we can be when calling a function. Listing 19-29 demonstrates how to use fully qualified syntax:

Filename: src/main.rs

```
# trait Animal {
#     fn baby_name() -> String;
# }
#
# struct Dog;
#
# impl Dog {
#     fn baby_name() -> String {
#         String::from("Spot")
#     }
# }
#
# impl Animal for Dog {
#     fn baby_name() -> String {
#         String::from("puppy")
#     }
# }
#
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

Listing 19-29: Using fully qualified syntax to specify that we want to call the `baby_name` function from the `Animal` trait as implemented on `Dog`

We're providing Rust with a type annotation within the angle brackets, which indicates we want to call the `baby_name` method from the `Animal` trait as implemented on `Dog` by saying that we want to treat the `Dog` type as an `Animal` for this function call. This code will now print what we want:

```
A baby dog is called a puppy
```

In general, fully qualified syntax is defined as follows:

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

For associated functions, there would not be a `receiver`: there would only be the list of other arguments. We could use fully qualified syntax everywhere that we call functions or methods. However, we're allowed to omit any part of this syntax that Rust can figure out from other information in the program. We only need to use this more verbose syntax in cases where there are multiple implementations that use the same name and Rust needs help to identify which implementation we want to call.

22.3.4　Using Supertraits to Require One Trait's Functionality Within Another Trait

Sometimes, we might need one trait to use another trait's functionality. In this case, we need to rely on the dependent trait also being implemented. The trait we're relying on is a *supertrait* of the trait we're implementing.

For example, let's say we want to make an `OutlinePrint` trait with an `outline_print` method that will print a value framed in asterisks. That is, given a `Point` struct that implements `Display` to result in `(x, y)`, when we call `outline_print` on a `Point` instance that has `1` for `x` and `3` for `y`, it should print the following:

```
**********
*        *
* (1, 3) *
*        *
**********
```

In the implementation of `outline_print`, we want to use the `Display` trait's functionality. Therefore, we need to specify that the `OutlinePrint` trait will only work for types that also implement `Display` and provide the functionality that `OutlinePrint` needs. We can do that in the trait definition by specifying `OutlinePrint: Display`. This technique is similar to adding a trait bound to the trait. Listing 19-30 shows an implementation of the `OutlinePrint` trait:

Filename: src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

Listing 19-30: Implementing the `OutlinePrint` trait that requires the functionality from `Display`

Because we've specified that `OutlinePrint` requires the `Display` trait, we can use the `to_string` function that is automatically implemented for any type that implements `Display`. If we tried to use `to_string` without adding `: Display` after the trait name, we'd get an error saying that no method named `to_string` was found for the type `&Self` in the current scope.

Let's see what happens when we try to implement `OutlinePrint` on a type that doesn't implement `Display`, such as the `Point` struct:

Filename: src/main.rs

```
# trait OutlinePrint {}
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

We get an error saying that `Display` is required but not implemented:

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied
  --> src/main.rs:20:6
   |
20 | impl OutlinePrint for Point {}
   |      ^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter;
   try using `:?` instead if you are using a format string
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
```

To fix this, we implement `Display` on `Point` and satisfy the constraint that `OutlinePrint` requires, like so:

Filename: src/main.rs

```
# struct Point {
#     x: i32,
#     y: i32,
# }
#
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Then implementing the `OutlinePrint` trait on `Point` will compile successfully, and we can call `outline_print` on a `Point` instance to display it within an outline of asterisks.

22.3.5　The Newtype Pattern to Implement External Traits on External Types

In Chapter 10 in the "Implementing a Trait on a Type" section, we mentioned the orphan rule that states we're allowed to implement a trait on a type as long as either the trait or the type are local to our crate. It's possible to get around this restriction using the *newtype pattern*, which involves creating a new type in a tuple struct. (We covered tuple structs in the "Tuple Structs without Named Fields to Create Different Types" section of Chapter 5.) The tuple struct will have one field and be a thin wrapper around the type we want to implement a trait for. Then the wrapper type is local to our crate, and we can implement the trait on the wrapper. *Newtype* is a term that originates from the Haskell programming language. There is no runtime performance penalty for using this pattern, and the wrapper type is elided at compile time.

As an example, let's say we want to implement `Display` on `Vec`, which the orphan rule prevents us from doing directly because the `Display` trait and the `Vec` type are defined outside our crate. We can make a `Wrapper` struct that holds an instance of `Vec`; then we can implement `Display` on `Wrapper` and use the `Vec` value, as shown in Listing 19-31:

Filename: src/main.rs

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}
```

Listing 19-31: Creating a `Wrapper` type around `Vec<String>` to implement `Display`

The implementation of `Display` uses `self.0` to access the inner `Vec`, because `Wrapper` is a tuple struct and `Vec` is the item at index 0 in the tuple. Then we can use the functionality of the `Display` type on `Wrapper`.

The downside of using this technique is that `Wrapper` is a new type, so it doesn't have the methods of the value it's holding. We would have to implement all the methods of `Vec` directly on `Wrapper` so it can delegate to `self.0`, allowing us to treat `Wrapper` exactly like a `Vec`. If we wanted the new type to have every method the inner type has, implementing the `Deref` trait (discussed in Chapter 15 in the "Treating Smart Pointers like Regular References with the `Deref`

Trait" section) on the `Wrapper` to return the inner type would be a solution. If we don't want the `Wrapper` type to have all the methods of the inner type, in order to restrict the `Wrapper` type's behavior for example, we would have to implement just the methods we do want manually.

Now you know how the newtype pattern is used in relation to traits; it's also a useful pattern even when traits are not involved. Let's switch focus and look at some advanced ways to interact with Rust's type system.

22.4 Advanced Types

The Rust type system has some features that we've mentioned in this book but haven't yet discussed. We'll start by discussing newtypes in general as we examine why newtypes are useful as types. Then we'll move on to type aliases, a feature similar to newtypes but with slightly different semantics. We'll also discuss the `!` type and dynamically sized types.

22.4.1 Using the Newtype Pattern for Type Safety and Abstraction

> Note: This section assumes you've read the earlier section "The Newtype Pattern to Implement External Traits on External Types".

The newtype pattern is useful for other tasks beyond what we've discussed so far, including statically enforcing that values are never confused and as an indication of the units of a value. You saw an example of using newtypes to indicate units in Listing 19-23: recall that the `Millimeters` and `Meters` structs wrapped `u32` values in a newtype. If we wrote a function with a parameter of type `Millimeters`, we couldn't compile a program that accidentally tried to call that function with a value of type `Meters` or a plain `u32`.

Another use of the newtype pattern is in abstracting away some implementation details of a type: the new type can expose a public API that is different from the API of the private inner type if we used the new type directly to restrict the available functionality, for example.

Newtypes can also hide internal implementation. For example, we could provide a `People` type to wrap a `HashMap<i32, String>` that stores a person's ID associated with their name. Code using `People` would only interact with the public API we provide, such as a method to add a name string to the `People` collection; that code wouldn't need to know that we assign an `i32` ID to names internally. The newtype pattern is a lightweight way to achieve encapsulation to hide implementation details, which we discussed in the "Encapsulation that Hides Implementation Details" section of Chapter 17.

22.4.2　Type Aliases Create Type Synonyms

Along with the newtype pattern, Rust provides the ability to declare a *type alias* to give an existing type another name. For this we use the `type` keyword. For example, we can create the alias `Kilometers` to `i32` like so:

```
type Kilometers = i32;
```

Now, the alias `Kilometers` is a *synonym* for `i32`; unlike the `Millimeters` and `Meters` types we created in Listing 19-23, `Kilometers` is not a separate, new type. Values that have the type `Kilometers` will be treated the same as values of type `i32`:

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

Because `Kilometers` and `i32` are the same type, we can add values of both types and we can pass `Kilometers` values to functions that take `i32` parameters. However, using this method, we don't get the type checking benefits that we get from the newtype pattern discussed earlier.

The main use case for type synonyms is to reduce repetition. For example, we might have a lengthy type like this:

```
Box<Fn() + Send + 'static>
```

Writing this lengthy type in function signatures and as type annotations all over the code can be tiresome and error prone. Imagine having a project full of code like that in Listing 19-32:

```
let f: Box<Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<Fn() + Send + 'static>) {
    // --snip--
}

fn returns_long_type() -> Box<Fn() + Send + 'static> {
    // --snip--
#     Box::new(|| ())
}
```

Listing 19-32: Using a long type in many places

A type alias makes this code more manageable by reducing the repetition. In Listing 19-33, we've introduced an alias named `Thunk` for the verbose type and can replace all uses of the type with the shorter alias `Thunk`:

```
type Thunk = Box<Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
#     Box::new(|| ())
}
```

Listing 19-33: Introducing a type alias `Thunk` to reduce repetition

This code is much easier to read and write! Choosing a meaningful name for a type alias can help communicate your intent as well (*thunk* is a word for code to be evaluated at a later time, so it's an appropriate name for a closure that gets stored).

Type aliases are also commonly used with the `Result<T, E>` type for reducing repetition. Consider the `std::io` module in the standard library. I/O operations often return a `Result<T, E>` to handle situations when operations fail to work. This library has a `std::io::Error` struct that represents all possible I/O errors. Many of the functions in `std::io` will be returning `Result<T, E>` where the `E` is `std::io::Error`, such as these functions in the `Write` trait:

```
use std::io::Error;
use std::fmt;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

The `Result<..., Error>` is repeated a lot. As such, `std::io` has this type of alias declaration:

```
type Result<T> = Result<T, std::io::Error>;
```

Because this declaration is in the `std::io` module, we can use the fully qualified alias `std::io::Result<T>`; that is, a `Result<T, E>` with the `E` filled in as `std::io::Error`. The `Write` trait function signatures end up looking like this:

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
```

```
    fn write_fmt(&mut self, fmt: Arguments) -> Result<()>;
}
```

The type alias helps in two ways: it makes code easier to write *and* it gives us a consistent interface across all of `std::io`. Because it's an alias, it's just another `Result<T, E>`, which means we can use any methods that work on `Result<T, E>` with it, as well as special syntax like `?`.

22.4.3　The ! Never Type that Never Returns

Rust has a special type named `!` that's known in type theory lingo as the *empty type* because it has no values. We prefer to call it the *never type* because it stands in the place of the return type when a function will never return. Here is an example:

```
fn bar() -> ! {
    // --snip--
}
```

This code is read as "the function `bar` returns never." Functions that return never are called *diverging functions*. We can't create values of the type `!` so `bar` can never possibly return.

But what use is a type you can never create values for? Recall the code in Chapter 2 that we added in the "Handling Invalid Input" section; we've reproduced it here in Listing 19-34:

```
# let guess = "3";
# loop {
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
# break;
# }
```

Listing 19-34: A `match` with an arm that ends in `continue`

At the time, we skipped over some details in this code. In Chapter 6 in "The `match` Control Flow Operator" section, we discussed that `match` arms must all return the same type. So, for example, the following code doesn't work:

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
}
```

The type of `guess` in this code would have to be an integer *and* a string, and Rust requires that `guess` can only have one type. So what does `continue` return? How were we allowed to return a `u32` from one arm and have another arm that ends with `continue` in Listing 19-34?

As you might have guessed, `continue` has a `!` value. That is, when Rust computes the type of `guess`, it looks at both match arms, the former with a value of `u32` and the latter with a `!` value. Because `!` can never have a value, Rust decides that the type of `guess` is `u32`.

The formal way of describing this behavior is that expressions of type `!` can be coerced into any other type. We're allowed to end this `match` arm with `continue` because `continue` doesn't return a value; instead, it moves control back to the top of the loop, so in the `Err` case, we never assign a value to `guess`.

The never type is useful with the `panic!` macro as well. Remember the `unwrap` function that we call on `Option<T>` values to produce a value or panic? Here is its definition:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

In this code, the same thing happens as in the `match` in Listing 19-34: Rust sees that `val` has the type `T` and `panic!` has the type `!` so the result of the overall `match` expression is `T`. This code works because `panic!` doesn't produce a value; it ends the program. In the `None` case, we won't be returning a value from `unwrap`, so this code is valid.

One final expression that has the type `!` is a `loop`:

```
print!("forever ");

loop {
    print!("and ever ");
}
```

Here, the loop never ends, so `!` is the value of the expression. However, this wouldn't be true if we included a `break`, because the loop would terminate when it got to the `break`.

22.4.4　Dynamically Sized Types and `Sized`

Due to Rust's need to know certain details, such as how much space to allocate for a value of a particular type, there is a corner of its type system that can be confusing: the concept of *dynamically sized types*. Sometimes referred to as *DSTs* or *unsized types*, these types let us write code using values whose size we can only know at runtime.

Let's dig into the details of a dynamically sized type called `str`, which we've been using throughout the book. That's right, not `&str`, but `str` on its own, is a DST. We can't know how

long the string is until runtime, meaning we can't create a variable of type `str`, nor can we take an argument of type `str`. Consider the following code, which does not work:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

Rust needs to know how much memory to allocate for any value of a particular type, and all values of a type must use the same amount of memory. If Rust allowed us to write this code, these two `str` values would need to take up the same amount of space. But they have different lengths: `s1` needs 12 bytes of storage and `s2` needs 15. This is why it's not possible to create a variable holding a dynamically sized type.

So what do we do? In this case, you already know the answer: we make the types of `s1` and `s2` a `&str` rather than a `str`. Recall that in the "String Slices" section of Chapter 4 we said the slice data structure stores the starting position and the length of the slice.

So although a `&T` is a single value that stores the memory address of where the `T` is located, a `&str` is *two* values: the address of the `str` and its length. As such, we can know the size of a `&str` value at compile time: it's two times the size of a `usize` in length. That is, we always know the size of a `&str`, no matter how long the string it refers to is. In general, this is the way in which dynamically sized types are used in Rust: they have an extra bit of metadata that stores the size of the dynamic information. The golden rule of dynamically sized types is that we must always put values of dynamically sized types behind a pointer of some kind.

We can combine `str` with all kinds of pointers: for example, `Box<str>` or `Rc<str>`. In fact, you've seen this before but with a different dynamically sized type: traits. Every trait is a dynamically sized type we can refer to by using the name of the trait. In Chapter 17 in the "Using Trait Objects that Allow for Values of Different Types" section, we mentioned that to use traits as trait objects, we must put them behind a pointer, such as `&Trait` or `Box<Trait>` (`Rc<Trait>` would work too).

To work with DSTs, Rust has a particular trait called the `Sized` trait to determine whether or not a type's size is known at compile time. This trait is automatically implemented for everything whose size is known at compile time. In addition, Rust implicitly adds a bound on `Sized` to every generic function. That is, a generic function definition like this:

```
fn generic<T>(t: T) {
    // --snip--
}
```

is actually treated as though we had written this:

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

By default, generic functions will only work on types that have a known size at compile time. However, you can use the following special syntax to relax this restriction:

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

A trait bound on `?Sized` is the opposite of a trait bound on `Sized`: we would read this as "`T` may or may not be `Sized`." This syntax is only available for `Sized`, not any other traits.

Also note that we switched the type of the `t` parameter from `T` to `&T`. Because the type might not be `Sized`, we need to use it behind some kind of pointer. In this case, we've chosen a reference.

Next, we'll talk about functions and closures!

22.5　Advanced Functions and Closures

Finally, we'll explore some advanced features related to functions and closures, which include function pointers and returning closures.

22.5.1　Function Pointers

We've talked about how to pass closures to functions; you can also pass regular functions to functions! This technique is useful when we want to pass a function we've already defined rather than defining a new closure. We do this using function pointers to allow us to use functions as arguments to other functions. Functions coerce to the type `fn` (with a lowercase f), not to be confused with the `Fn` closure trait. The `fn` type is called a function pointer. The syntax for specifying that a parameter is a function pointer is similar to that of closures, as shown in Listing 19-35:

Filename: src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

Listing 19-35: Using the `fn` type to accept a function pointer as an argument

This code prints `The answer is: 12`. We specify that the parameter `f` in `do_twice` is an `fn` that takes one parameter of type `i32` and returns an `i32`. We can then call `f` in the body of `do_twice`. In `main`, we can pass the function name `add_one` as the first argument to `do_twice`.

Unlike closures, `fn` is a type rather than a trait, so we specify `fn` as the parameter type directly rather than declaring a generic type parameter with one of the `Fn` traits as a trait bound.

Function pointers implement all three of the closure traits (`Fn`, `FnMut`, and `FnOnce`), so we can always pass a function pointer as an argument for a function that expects a closure. It's best to write functions using a generic type and one of the closure traits so your functions can accept either functions or closures.

An example of where you would want to only accept `fn` and not closures is when interfacing with external code that doesn't have closures: C functions can accept functions as arguments, but C doesn't have closures.

As an example of where we can use either a closure defined inline or a named function, let's look at a use of `map`. To use the `map` function to turn a vector of numbers into a vector of strings, we could use a closure, like this:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(|i| i.to_string())
    .collect();
```

Or we could name a function as the argument to `map` instead of the closure, like this:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(ToString::to_string)
    .collect();
```

Note that we must use the fully qualified syntax that we talked about earlier in the "Advanced Traits" section because there are multiple functions available named `to_string`. Here, we're using the `to_string` function defined in the `ToString` trait, which the standard library has implemented for any type that implements `Display`.

Some people prefer this style, and some people prefer to use closures. They end up compiling to the same code, so use whichever style is clearer to you.

22.5.2　Returning Closures

Closures are represented by traits, which means we can't return closures directly. In most cases where we might want to return a trait, we can instead use the concrete type that implements the trait as the return value of the function. But we can't do that with closures because they

don't have a concrete type that is returnable; we're not allowed to use the function pointer `fn` as a return type, for example.

The following code tries to return a closure directly, but it won't compile:

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
```

The compiler error is as follows:

```
error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static:
std::marker::Sized` is not satisfied
 -->
  |
1 | fn returns_closure() -> Fn(i32) -> i32 {
  |                         ^^^^^^^^^^^^^^ `std::ops::Fn(i32) -> i32 + 'static`
  does not have a constant size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for
  `std::ops::Fn(i32) -> i32 + 'static`
  = note: the return type of a function must have a statically known size
```

The error references the `Sized` trait again! Rust doesn't know how much space it will need to store the closure. We saw a solution to this problem earlier. We can use a trait object:

```
fn returns_closure() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

This code will compile just fine. For more about trait objects, refer to the "Trait Objects" section in Chapter 17.

22.6 Summary

Whew! Now you have some features of Rust in your toolbox that you won't use often, but you'll know they're available in very particular circumstances. We've introduced several complex topics so that when you encounter them in error message suggestions or in other peoples' code, you'll be able to recognize these concepts and syntax. Use this chapter as a reference to guide you to solutions.

Next, we'll put everything we've discussed throughout the book into practice and do one more project!

第 23 章

Final Project: Building a Multithreaded Web Server

It's been a long journey, but we've reached the end of the book. In this chapter, we'll build one more project together to demonstrate some of the concepts we covered in the final chapters, as well as recap some earlier lessons.

For our final project, we'll make a web server that only says "hello" and looks like Figure 20-1 in a web browser.

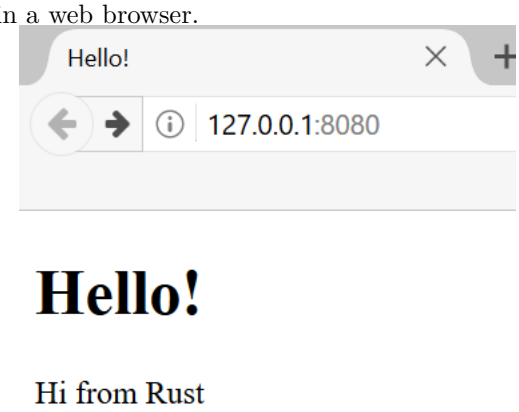


Figure 20-1: Our final shared project

Here is the plan to build the web server:

1. Learn a bit about TCP and HTTP
2. Listen for TCP connections on a socket
3. Parse a small number of HTTP requests
4. Create a proper HTTP response

5. Improve the throughput of our server with a thread pool

But before we get started, we should mention one detail: the method we'll use won't be the best way to build a web server with Rust. A number of production-ready crates are available on *`https://crates.io/*` that provide more complete web server and thread pool implementations than we'll build.

However, our intention in this chapter is to help you learn, not to take the easy route. Because Rust is a systems programming language, we can choose the level of abstraction we want to work with and can go to a lower level than is possible or practical in other languages. We'll write the basic HTTP server and thread pool manually so you can learn the general ideas and techniques behind the crates you might use in the future.

23.1 Building a Single Threaded Web Server

We'll start by getting a single threaded web server working. Before we begin, let's look at a quick overview of the protocols involved in building web servers. The details of these protocols are beyond the scope of this book, but a brief overview will give you the information you need.

The two main protocols involved in web servers are the *Hypertext Transfer Protocol (HTTP)* and the *Transmission Control Protocol (TCP)*. Both protocols are *request-response* protocols, meaning a *client* initiates requests, and a *server* listens to the requests and provides a response to the client. The contents of those requests and responses are defined by the protocols.

TCP is the lower-level protocol that describes the details of how information gets from one server to another but doesn't specify what that information is. HTTP builds on top of TCP by defining the contents of the requests and responses. It's technically possible to use HTTP with other protocols, but in the vast majority of cases, HTTP sends its data over TCP. We'll work with the raw bytes of TCP and HTTP requests and responses.

23.1.1 Listening to the TCP Connection

Our web server needs to listen to a TCP connection, so that's the first part we'll work on. The standard library offers a `std::net` module that lets us do this. Let's make a new project in the usual fashion:

```
$ cargo new hello --bin
     Created binary (application) `hello` project
$ cd hello
```

Now enter the code in Listing 20-1 in *src/main.rs* to start. This code will listen at the address `127.0.0.1:7878` for incoming TCP streams. When it gets an incoming stream, it will print `Connection established!`.

Filename: src/main.rs

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("Connection established!");
    }
}
```

Listing 20-1: Listening for incoming streams and printing a message when we receive a stream

Using `TcpListener`, we can listen for TCP connections at the address `127.0.0.1:7878`. In the address, the section before the colon is an IP address representing your computer (this is the same on every computer and doesn't represent the authors' computer specifically), and `7878` is the port. We've chosen this port for two reasons: HTTP is normally accepted on this port, and 7878 is "rust" typed on a telephone. Note that connecting to port 80 requires administrator privileges; nonadministrators can only listen on ports higher than 1024.

The `bind` function in this scenario works like the `new` function in that it will return a new `TcpListener` instance. The reason the function is called `bind` is that in networking, connecting to a port to listen to is known as "binding to a port."

The `bind` function returns a `Result<T, E>`, which indicates that binding might fail. For example, if we tried to connect to port 80 without being an administrator or if we ran two instances of our program and so had two programs listening to the same port, binding wouldn't work. Because we're writing a basic server just for learning purposes, we won't worry about handling these kinds of errors; instead, we use `unwrap` to stop the program if errors happen.

The `incoming` method on `TcpListener` returns an iterator that gives us a sequence of streams (more specifically, streams of type `TcpStream`). A single *stream* represents an open connection between the client and the server. A *connection* is the name for the full request and response process in which a client connects to the server, the server generates a response, and the server closes the connection. As such, `TcpStream` will read from itself to see what the client sent, and then allow us to write our response to the stream. Overall, this `for` loop will process each connection in turn and produce a series of streams for us to handle.

For now, our handling of the stream consists of calling `unwrap` to terminate our program if the stream has any errors; if there aren't any errors, the program prints a message. We'll add more functionality for the success case in the next listing. The reason we might receive errors from the `incoming` method when a client connects to the server is that we're not actually iterating over connections, we're iterating over *connection attempts*. The connection might not

be successful for a number of reasons, many of them operating system specific. For example, many operating systems have a limit to the number of simultaneous open connections they can support; new connection attempts beyond that number will produce an error until some of the open connections are closed.

Let's try running this code! Invoke `cargo run` in the terminal, and then load `127.0.0.1:7878` in a web browser. The browser should show an error message like "Connection reset," because the server isn't currently sending back any data. But when you look at your terminal, you should see several messages that were printed when the browser connected to the server!

```
     Running `target/debug/hello`
Connection established!
Connection established!
Connection established!
```

Sometimes, you'll see multiple messages printed for one browser request; the reason might be that the browser is making a request for the page as well as a request for other resources, like the `favicon.ico` icon that appears in the browser tab.

It could also be that the browser is trying to connect to the server multiple times because the server isn't responding with any data. When `stream` goes out of scope and is dropped at the end of the loop, the connection is closed as part of the `drop` implementation. Browsers sometimes deal with closed connections by retrying, because the problem might be temporary. The important factor is that we've successfully gotten a handle to a TCP connection!

Remember to stop the program by pressing ctrl-c when you're done running a particular version of the code. Then restart `cargo run` after you've made each set of code changes to make sure you're running the newest code.

23.1.2 Reading the Request

Let's implement the functionality to read the request from the browser! To separate the concerns of first getting a connection and then taking some action with the connection, we'll start a new function for processing connections. In this new `handle_connection` function, we'll read data from the TCP stream and print it so we can see the data being sent from the browser. Change the code to look like Listing 20-2.

Filename: src/main.rs

```
use std::io::prelude::*;
use std::net::TcpStream;
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
```

```
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

    println!("Request: {}", String::from_utf8_lossy(&buffer[..]));
}
```

Listing 20-2: Reading from the `TcpStream` and printing the data

We bring `std::io::prelude` into scope to get access to certain traits that let us read from and write to the stream. In the `for` loop in the `main` function, instead of printing a message that says we made a connection, we now call the new `handle_connection` function and pass the `stream` to it.

In the `handle_connection` function, we've made the `stream` parameter mutable. The reason is that the `TcpStream` instance keeps track of what data it returns to us internally. It might read more data than we asked for and save that data for the next time we ask for data. It therefore needs to be `mut` because its internal state might change; usually, we think of "reading" as not needing mutation, but in this case we need the `mut` keyword.

Next, we need to actually read from the stream. We do this in two steps: first, we declare a `buffer` on the stack to hold the data that is read in. We've made the buffer 512 bytes in size, which is big enough to hold the data of a basic request and sufficient for our purposes in this chapter. If we wanted to handle requests of an arbitrary size, buffer management would need to be more complicated; we'll keep it simple for now. We pass the buffer to `stream.read`, which will read bytes from the `TcpStream` and put them in the buffer.

Second, we convert the bytes in the buffer to a string and print that string. The `String::from_utf8_lossy` function takes a `&[u8]` and produces a `String` from it. The "lossy" part of the name indicates the behavior of this function when it sees an invalid UTF-8 sequence: it will replace the invalid sequence with `?`, the `U+FFFD REPLACEMENT CHARACTER`. You might see replacement characters for characters in the buffer that aren't filled by request data.

Let's try this code! Start the program and make a request in a web browser again. Note that we'll still get an error page in the browser, but our program's output in the terminal will now look similar to this:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42 secs
     Running `target/debug/hello`
```

```
Request: GET / HTTP/1.1
Host: 127.0.0.1:7878
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101
Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
??????????????????????????????????????
```

Depending on your browser, you might get slightly different output. Now that we're printing the request data, we can see why we get multiple connections from one browser request by looking at the path after `Request: GET`. If the repeated connections are all requesting `/`, we know the browser is trying to fetch `/` repeatedly because it's not getting a response from our program.

Let's break down this request data to understand what the browser is asking of our program.

23.1.3 A Closer Look at an HTTP Request

HTTP is a text-based protocol, and a request takes this format:

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

The first line is the *request line* that holds information about what the client is requesting. The first part of the request line indicates the *method* being used, such as `GET` or `POST`, which describes how the client is making this request. Our client used a `GET` request.

The next part of the request line is `/`, which indicates the *Uniform Resource Identifier (URI)* the client is requesting: a URI is almost, but not quite, the same as a *Uniform Resource Locator (URL)*. The difference between URIs and URLs isn't important for our purposes in this chapter, but the HTTP spec uses the term URI, so we can just mentally substitute URL for URI here.

The last part is the HTTP version the client uses, and then the request line ends in a CRLF sequence. The CRLF sequence can also be written as `\r\n`: the `\r` part is a *carriage return* and `\n` is a *line feed*. (These terms come from the typewriter days!) The CRLF sequence separates the request line from the rest of the request data. Note that when the CRLF is printed, we see a new line start rather than `\r\n`.

Looking at the request line data we received from running our program so far, we see that `GET` is the method, `/` is the request URI, and `HTTP/1.1` is the version.

After the request line, the remaining lines starting from `Host:` onward are headers. `GET` requests have no body.

Try making a request from a different browser or asking for a different address, such as `127.0.0.1:7878/test`, to see how the request data changes.

Now that we know what the browser is asking for, let's send back some data!

23.1.4 Writing a Response

Now we'll implement sending data in response to a client request. Responses have the following format:

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

The first line is a *status line* that contains the HTTP version used in the response, a numeric status code that summarizes the result of the request, and a reason phrase that provides a text description of the status code. After the CRLF sequence are any headers, another CRLF sequence, and the body of the response.

Here is an example response that uses HTTP version 1.1, has a status code of 200, an OK reason phrase, no headers, and no body:

```
HTTP/1.1 200 OK\r\n\r\n
```

The status code 200 is the standard success response. The text is a tiny successful HTTP response. Let's write this to the stream as our response to a successful request! From the `handle_connection` function, remove the `println!` that was printing the request data and replace it with the code in Listing 20-3.

Filename: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

    let response = "HTTP/1.1 200 OK\r\n\r\n";

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Listing 20-3: Writing a tiny successful HTTP response to the stream

The first new line defines the `response` variable that holds the success message's data. Then we call `as_bytes` on our `response` to convert the string data to bytes. The `write` method on `stream` takes a `&[u8]` and sends those bytes directly down the connection.

Because the `write` operation could fail, we use `unwrap` on any error result as before. Again, in a real application you would add error-handling here. Finally, `flush` will wait and prevent the

program from continuing until all the bytes are written to the connection; `TcpStream` contains an internal buffer to minimize calls to the underlying operating system.

With these changes, let's run our code and make a request. We're no longer printing any data to the terminal, so we won't see any output other than the output from Cargo. When you load `127.0.0.1:7878` in a web browser, you should get a blank page instead of an error. You've just hand-coded an HTTP request and response!

23.1.5 Returning Real HTML

Let's implement the functionality for returning more than a blank page. Create a new file, *hello.html*, in the root of your project directory, not in the *src* directory. You can input any HTML you want; Listing 20-4 shows one possibility.

Filename: hello.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

Listing 20-4: A sample HTML file to return in a response

This is a minimal HTML5 document with a heading and some text. To return this from the server when a request is received, we'll modify `handle_connection` as shown in Listing 20-5 to read the HTML file, add it to the response as a body, and send it.

Filename: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
use std::fs::File;
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let mut file = File::open("hello.html").unwrap();

    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();

    let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);
```

```
16
17      stream.write(response.as_bytes()).unwrap();
18      stream.flush().unwrap();
19  }
```

Listing 20-5: Sending the contents of *hello.html* as the body of the response

We've added a line at the top to bring the standard library's `File` into scope. The code for opening a file and reading the contents should look familiar; we used it in Chapter 12 when we read the contents of a file for our I/O project in Listing 12-4.

Next, we use `format!` to add the file's contents as the body of the success response.

Run this code with `cargo run` and load `127.0.0.1:7878` in your browser; you should see your HTML rendered!

Currently, we're ignoring the request data in `buffer` and just sending back the contents of the HTML file unconditionally. That means if you try requesting `127.0.0.1:7878/something-else` in your browser, you'll still get back this same HTML response. Our server is very limited and is not what most web servers do. We want to customize our responses depending on the request, and only send back the HTML file for a well-formed request to `/`.

23.1.6　Validating the Request and Selectively Responding

Right now, our web server will return the HTML in the file no matter what the client requested. Let's add functionality to check that the browser is requesting `/` before returning the HTML file, and return an error if the browser requests anything else. For this we need to modify `handle_connection` as shown in Listing 20-6. This new code checks the content of the request received against what we know a request for `/` looks like and adds `if` and `else` blocks to treat requests differently.

Filename: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
# use std::fs::File;
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";

    if buffer.starts_with(get) {
        let mut file = File::open("hello.html").unwrap();

        let mut contents = String::new();
        file.read_to_string(&mut contents).unwrap();
```

```
17
18          let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);
19
20          stream.write(response.as_bytes()).unwrap();
21          stream.flush().unwrap();
22      } else {
23          // some other request
24      }
25  }
```

Listing 20-6: Matching the request and handling requests to `/` differently than other requests

First, we hardcode the data corresponding to the `/` request into the `get` variable. Because we're reading raw bytes into the buffer, we transform `get` into a byte string by adding the `b""` byte string syntax at the start of the content data. Then we check if `buffer` starts with the bytes in `get`. If it does, it means we've received a well-formed request to `/`, which is the success case we'll handle in the `if` block that returns the contents of our HTML file.

If `buffer` does *not* start with the bytes in `get`, it means we've received some other request. We'll add code to the `else` block in a moment to respond to all other requests.

Run this code now and request `127.0.0.1:7878`; you should get the HTML in *hello.html*. If you make any other request, such as `127.0.0.1:7878/something-else`, you'll get a connection error like you saw when running the code in Listing 20-1 and Listing 20-2.

Now let's add the code in Listing 20-7 to the `else` block to return a response with the status code `404`, which signals that the content for the request was not found. We'll also return some HTML for a page to render in the browser indicating as such to the end user.

Filename: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
# use std::fs::File;
# fn handle_connection(mut stream: TcpStream) {
# if true {
// --snip--

} else {
    let status_line = "HTTP/1.1 404 NOT FOUND\r\n\r\n";
    let mut file = File::open("404.html").unwrap();
    let mut contents = String::new();

    file.read_to_string(&mut contents).unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
# }
```

Listing 20-7: Responding with status code `404` and an error page if anything other than `/` was requested

Here, our response has a status line with status code `404` and the reason phrase `NOT FOUND`. We're still not returning headers, and the body of the response will be the HTML in the file *404.html*. You'll need to create a *404.html* file next to *hello.html* for the error page; again feel free to use any HTML you want or use the example HTML in Listing 20-8.

Filename: 404.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

Listing 20-8: Sample content for the page to send back with any `404` response

With these changes, run your server again. Requesting `127.0.0.1:7878` should return the contents of *hello.html*, and any other request, like `127.0.0.1:7878/foo`, should return the error HTML from *404.html*.

23.1.7　A Touch of Refactoring

At the moment the `if` and `else` blocks have a lot of repetition: they're both reading files and writing the contents of the files to the stream. The only differences are the status line and the filename. Let's make the code more concise by pulling out those differences into separate `if` and `else` lines that will assign the values of the status line and the filename to variables; we can then use those variables unconditionally in the code to read the file and write the response. Listing 20-9 shows the resulting code after replacing the large `if` and `else` blocks.

Filename: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
# use std::fs::File;
// --snip--

fn handle_connection(mut stream: TcpStream) {
#     let mut buffer = [0; 512];
#     stream.read(&mut buffer).unwrap();
#
#     let get = b"GET / HTTP/1.1\r\n";
    // --snip--
```

```
12
13      let (status_line, filename) = if buffer.starts_with(get) {
14          ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
15      } else {
16          ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
17      };
18
19      let mut file = File::open(filename).unwrap();
20      let mut contents = String::new();
21
22      file.read_to_string(&mut contents).unwrap();
23
24      let response = format!("{}{}", status_line, contents);
25
26      stream.write(response.as_bytes()).unwrap();
27      stream.flush().unwrap();
28  }
```

Listing 20-9: Refactoring the `if` and `else` blocks to contain only the code that differs between the two cases

Now the `if` and `else` blocks only return the appropriate values for the status line and filename in a tuple; we then use destructuring to assign these two values to `status_line` and `filename` using a pattern in the `let` statement, as discussed in Chapter 18.

The previously duplicated code is now outside the `if` and `else` blocks, and uses the `status_line` and `filename` variables. This makes it easier to see the difference between the two cases, and means we have only one place to update the code if we want to change how the file reading and response writing works. The behavior of the code in Listing 20-9 will be the same as that in Listing 20-8.

Awesome! We now have a simple web server in approximately 40 lines of Rust code that responds to one request with a page of content and responds to all other requests with a `404` response.

Currently, our server runs in a single thread, meaning it can only serve one request at a time. Let's examine how that can be a problem by simulating some slow requests, and then fix it so our server can handle multiple requests at once.

23.2　Turning Our Single Threaded Server into a Multithreaded Server

Right now, the server will process each request in turn, meaning it won't process a second connection until the first is finished processing. If the server received more and more requests, this serial execution would be less and less optimal. If the server receives a request that takes a long time to process, subsequent requests will have to wait until the long request is finished, even if the new requests can be processed quickly. We'll need to fix this, but first, we'll look at

the problem in action.

23.2.1 Simulating a Slow Request in the Current Server Implementation

We'll look at how a slow-processing request can affect other requests made to our current server implementation. Listing 20-10 implements handling a request to `/sleep` with a simulated slow response that will cause the server to sleep for five seconds before responding.

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;
# use std::io::prelude::*;
# use std::net::TcpStream;
# use std::fs::File;
// --snip--

fn handle_connection(mut stream: TcpStream) {
#     let mut buffer = [0; 512];
#     stream.read(&mut buffer).unwrap();
    // --snip--

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };

    // --snip--
}
```

Listing 20-10: Simulating a slow request by recognizing `/sleep` and sleeping for five seconds

This code is a bit messy, but it's good enough for simulation purposes. We created a second request `sleep`, whose data our server recognizes. We added an `else if` after the `if` block to check for the request to `/sleep`. When that request is received, the server will sleep for five seconds before rendering the successful HTML page.

You can see how primitive our server is: real libraries would handle the recognition of multiple requests in a much less verbose way!

Start the server using `cargo run`, and then open two browser windows: one for `http://localhost:7878/` and the other for `http://localhost:7878/sleep`. If you enter the `/` URI a few times, as before, you'll see it respond quickly. But if you enter `/sleep`, and then load `/`,

you'll see that `/` waits until `sleep` has slept for its full five seconds before loading.

There are multiple ways we could change how our web server works to avoid having all requests back up behind a slow request; the one we'll implement is a thread pool.

23.2.2 Improving Throughput with a Thread Pool

A *thread pool* is a group of spawned threads that are waiting and ready to handle a task. When the program receives a new task, it assigns one of the threads in the pool to the task, and that thread will process the task. The remaining threads in the pool are available to handle any other tasks that come in while the first thread is processing. When the first thread is done processing its task, it's returned to the pool of idle threads ready to handle a new task. A thread pool will allow us to process connections concurrently, increasing the throughput of our server.

We'll limit the number of threads in the pool to a small number to protect us from Denial of Service (DoS) attacks; if we had our program create a new thread for each request as it comes in, someone making ten million requests to our server could create havoc by using up all our server's resources and grinding the processing of all requests to a halt.

Rather than spawning unlimited threads, we'll have a fixed number of threads waiting in the pool. As requests come in, they'll be sent to the pool for processing. The pool will maintain a queue of incoming requests. Each of the threads in the pool will pop off a request from this queue, handle the request, and then ask the queue for another request. With this design, we can process `N` requests concurrently, where `N` is the number of threads. If each thread is responding to a long-running request, subsequent requests can still back up in the queue, but we've increased the number of long-running requests we can handle before that point.

This technique is just one of many ways to improve the throughput of our web server. Other options you might explore are the fork/join model and the single threaded async I/O model. If you're interested in this topic, you can read more about other solutions and try to implement them in Rust; with a low-level language like Rust, all of these options are possible.

Before we begin implementing a thread pool, let's talk about what using the pool should look like. When you're trying to design code, writing the client interface first can help guide your design. Write the API of the code so it's structured in the way you want to call it, and then implement the functionality within that structure rather than implementing the functionality and then designing the public API.

Similar to how we used Test Driven Development in the project in Chapter 12, we'll use Compiler Driven Development here. We'll write the code that calls the functions we want, and then we'll look at errors from the compiler to determine what we should change next to get the code to work.

23.2.2.1 Code Structure If We Could Spawn a Thread for Each Request

First, let's explore how our code might look if it did create a new thread for every connection. As mentioned earlier, this isn't our final plan due to the problems with potentially spawning an unlimited number of threads, but it is a starting point. Listing 20-11 shows the changes to make to `main` to spawn a new thread to handle each stream within the `for` loop.

Filename: src/main.rs

```
# use std::thread;
# use std::io::prelude::*;
# use std::net::TcpListener;
# use std::net::TcpStream;
#
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}
# fn handle_connection(mut stream: TcpStream) {}
```

Listing 20-11: Spawning a new thread for each stream

As you learned in Chapter 16, `thread::spawn` will create a new thread and then run the code in the closure in the new thread. If you run this code and load `/sleep` in your browser, then `/` in two more browser tabs, you'll indeed see that the requests to `/` don't have to wait for `/sleep` to finish. But as we mentioned, this will eventually overwhelm the system because we're making new threads without any limit.

23.2.2.2 Creating a Similar Interface for a Finite Number of Threads

We want our thread pool to work in a similar, familiar way so switching from threads to a thread pool doesn't require large changes to the code that uses our API. Listing 20-12 shows the hypothetical interface for a `ThreadPool` struct we want to use instead of `thread::spawn`.

Filename: src/main.rs

```
# use std::thread;
# use std::io::prelude::*;
# use std::net::TcpListener;
# use std::net::TcpStream;
# struct ThreadPool;
# impl ThreadPool {
#     fn new(size: u32) -> ThreadPool { ThreadPool }
```

```
#    fn execute<F>(&self, f: F)
#        where F: FnOnce() + Send + 'static {}
# }
#
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
# fn handle_connection(mut stream: TcpStream) {}
```

Listing 20-12: Our ideal `ThreadPool` interface

We use `ThreadPool::new` to create a new thread pool with a configurable number of threads, in this case four. Then, in the `for` loop, `pool.execute` has a similar interface as `thread::spawn` in that it takes a closure the pool should run for each stream. We need to implement `pool.execute` so it takes the closure and gives it to a thread in the pool to run. This code won't yet compile, but we'll try so the compiler can guide us in how to fix it.

23.2.2.3 Building the `ThreadPool` Struct Using Compiler Driven Development

Make the changes in Listing 20-12 to *src/main.rs*, and then let's use the compiler errors from `cargo check` to drive our development. Here is the first error we get:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve. Use of undeclared type or module `ThreadPool`
  --> src\main.rs:10:16
   |
10 |     let pool = ThreadPool::new(4);
   |                ^^^^^^^^^^^^^^^^ Use of undeclared type or module
   `ThreadPool`

error: aborting due to previous error
```

Great, this error tells us we need a `ThreadPool` type or module, so we'll build one now. Our `ThreadPool` implementation will be independent of the kind of work our web server is doing. So, let's switch the `hello` crate from a binary crate to a library crate to hold our `ThreadPool` implementation. After we change to a library crate, we could also use the separate thread pool library for any work we want to do using a thread pool, not just for serving web requests.

Create a *src/lib.rs* that contains the following, which is the simplest definition of a `ThreadPool`

struct that we can have for now:

Filename: src/lib.rs

```
pub struct ThreadPool;
```

Then create a new directory, *src/bin*, and move the binary crate rooted in *src/main.rs* into *src/bin/main.rs*. Doing so will make the library crate the primary crate in the *hello* directory; we can still run the binary in *src/bin/main.rs* using `cargo run`. After moving the *main.rs* file, edit it to bring the library crate in and bring `ThreadPool` into scope by adding the following code to the top of *src/bin/main.rs*:

Filename: src/bin/main.rs

```
extern crate hello;
use hello::ThreadPool;
```

This code still won't work, but let's check it again to get the next error that we need to address:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for type
`hello::ThreadPool` in the current scope
 --> src/bin/main.rs:13:16
   |
13 |     let pool = ThreadPool::new(4);
   |                ^^^^^^^^^^^^^^^ function or associated item not found in
   `hello::ThreadPool`
```

This error indicates that next we need to create an associated function named `new` for `ThreadPool`. We also know that `new` needs to have one parameter that can accept `4` as an argument and should return a `ThreadPool` instance. Let's implement the simplest `new` function that will have those characteristics:

Filename: src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

We chose `usize` as the type of the `size` parameter, because we know that a negative number of threads doesn't make any sense. We also know we'll use this 4 as the number of elements in a collection of threads, which is what the `usize` type is for, as discussed in the "Integer Types" section of Chapter 3.

Let's check the code again:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
 --> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |                ^^^^
  |
  = note: #[warn(unused_variables)] on by default
  = note: to avoid this warning, consider using `_size` instead

error[E0599]: no method named `execute` found for type `hello::ThreadPool` in the
    current scope
  --> src/bin/main.rs:18:14
   |
18 |         pool.execute(|| {
   |              ^^^^^^^
```

Now we get a warning and an error. Ignoring the warning for a moment, the error occurs because we don't have an `execute` method on `ThreadPool`. Recall from the "Creating a Similar Interface for a Finite Number of Threads" section that we decided our thread pool should have an interface similar to `thread::spawn`. In addition, we'll implement the `execute` function so it takes the closure it's given and gives it to an idle thread in the pool to run.

We'll define the `execute` method on `ThreadPool` to take a closure as a parameter. Recall from the "Storing Closures Using Generic Parameters and the `Fn` Traits" section in Chapter 13 that we can take closures as parameters with three different traits: `Fn`, `FnMut`, and `FnOnce`. We need to decide which kind of closure to use here. We know we'll end up doing something similar to the standard library `thread::spawn` implementation, so we can look at what bounds the signature of `thread::spawn` has on its parameter. The documentation shows us the following:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T + Send + 'static,
        T: Send + 'static
```

The `F` type parameter is the one we're concerned with here; the `T` type parameter is related to the return value and we're not concerned with that. We can see that `spawn` uses `FnOnce` as the trait bound on `F`. This is probably what we want as well, because we'll eventually pass the argument we get in `execute` to `spawn`. We can be further confident that `FnOnce` is the trait we want to use because the thread for running a request will only execute that request's closure one time, which matches the `Once` in `FnOnce`.

The `F` type parameter also has the trait bound `Send` and the lifetime bound `'static`, which are useful in our situation: we need `Send` to transfer the closure from one thread to another and `'static` because we don't know how long the thread will take to execute. Let's create an `execute`

method on `ThreadPool` that will take a generic parameter of type `F` with these bounds:

Filename: src/lib.rs

```
# pub struct ThreadPool;
impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {

    }
}
```

We still use the `()` after `FnOnce` because this `FnOnce` represents a closure that takes no parameters and doesn't return a value. Just like function definitions, the return type can be omitted from the signature, but even if we have no parameters, we still need the parentheses.

Again, this is the simplest implementation of the `execute` method: it does nothing, but we're trying only to make our code compile. Let's check it again:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
 --> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |                ^^^^
  |
  = note: #[warn(unused_variables)] on by default
  = note: to avoid this warning, consider using `_size` instead

warning: unused variable: `f`
 --> src/lib.rs:8:30
  |
8 |     pub fn execute<F>(&self, f: F)
  |                              ^
  |
  = note: to avoid this warning, consider using `_f` instead
```

We're receiving only warnings now, which means it compiles! But note that if you try `cargo run` and make a request in the browser, you'll see the errors in the browser that we saw at the beginning of the chapter. Our library isn't actually calling the closure passed to `execute` yet!

> Note: A saying you might hear about languages with strict compilers, such as Haskell and Rust, is "if the code compiles, it works." But this saying is not universally true. Our project compiles, but it does absolutely nothing! If we were building a real, complete project, this would be a good time to start writing unit tests to check that the code compiles *and* has

the behavior we want.

23.2.2.4　Validating the Number of Threads in `new`

We'll continue to get warnings because we aren't doing anything with the parameters to `new` and `execute`. Let's implement the bodies of these functions with the behavior we want. To start, let's think about `new`. Earlier we chose an unsigned type for the `size` parameter, because a pool with a negative number of threads makes no sense. However, a pool with zero threads also makes no sense, yet zero is a perfectly valid `usize`. We'll add code to check that `size` is greater than zero before we return a `ThreadPool` instance and have the program panic if it receives a zero by using the `assert!` macro, as shown in Listing 20-13.

Filename: src/lib.rs

```
# pub struct ThreadPool;
impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }

    // --snip--
}
```

Listing 20-13: Implementing `ThreadPool::new` to panic if `size` is zero

We've added some documentation for our `ThreadPool` with doc comments. Note that we followed good documentation practices by adding a section that calls out the situations in which our function can panic, as discussed in Chapter 14. Try running `cargo doc --open` and clicking the `ThreadPool` struct to see what the generated docs for `new` look like!

Instead of adding the `assert!` macro as we've done here, we could make `new` return a `Result` like we did with `Config::new` in the I/O project in Listing 12-9. But we've decided in this case that trying to create a thread pool without any threads should be an unrecoverable error. If you're feeling ambitious, try to write a version of `new` with the following signature to compare both versions:

```
pub fn new(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

23.2.2.5 Creating Space to Store the Threads

Now that we have a way to know we have a valid number of threads to store in the pool, we can create those threads and store them in the `ThreadPool` struct before returning it. But how do we "store" a thread? Let's take another look at the `thread::spawn` signature:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T + Send + 'static,
        T: Send + 'static
```

The `spawn` function returns a `JoinHandle<T>`, where `T` is the type that the closure returns. Let's try using `JoinHandle` too and see what happens. In our case, the closures we're passing to the thread pool will handle the connection and not return anything, so `T` will be the unit type `()`.

The code in Listing 20-14 will compile but doesn't create any threads yet. We've changed the definition of `ThreadPool` to hold a vector of `thread::JoinHandle<()>` instances, initialized the vector with a capacity of `size`, set up a `for` loop that will run some code to create the threads, and returned a `ThreadPool` instance containing them.

Filename: src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // create some threads and store them in the vector
        }

        ThreadPool {
            threads
        }
    }

    // --snip--
}
```

Listing 20-14: Creating a vector for `ThreadPool` to hold the threads

We've brought `std::thread` into scope in the library crate, because we're using `thread::JoinHandle` as the type of the items in the vector in `ThreadPool`.

Once a valid size is received, our `ThreadPool` creates a new vector that can hold `size` items. We haven't used the `with_capacity` function in this book yet, which performs the same task as `Vec::new` but with an important difference: it preallocates space in the vector. Because we know we need to store `size` elements in the vector, doing this allocation up front is slightly more efficient than using `Vec::new`, which resizes itself as elements are inserted.

When you run `cargo check` again, you'll get a few more warnings, but it should succeed.

23.2.2.6 A `Worker` Struct Responsible for Sending Code from the `ThreadPool` to a Thread

We left a comment in the `for` loop in Listing 20-14 regarding the creation of threads. Here, we'll look at how we actually create threads. The standard library provides `thread::spawn` as a way to create threads, and `thread::spawn` expects to get some code the thread should run as soon as the thread is created. However, in our case we want to create the threads and have them *wait* for code that we'll send later. The standard library's implementation of threads doesn't include any way to do that; we have to implement it manually.

We'll implement this behavior by introducing a new data structure between the `ThreadPool` and the threads that will manage this new behavior. We'll call this data structure `Worker`, which is a common term in pooling implementations. Think of people working in the kitchen at a restaurant: the workers wait until orders come in from customers, and then they're responsible for taking those orders and filling them.

Instead of storing a vector of `JoinHandle<()>` instances in the thread pool, we'll store instances of the `Worker` struct. Each `Worker` will store a single `JoinHandle<()>` instance. Then we'll implement a method on `Worker` that will take a closure of code to run and send it to the already running thread for execution. We'll also give each worker an `id` so we can distinguish between the different workers in the pool when logging or debugging.

Let's make the following changes to what happens when we create a `ThreadPool`. We'll implement the code that sends the closure to the thread after we have `Worker` set up in this way:

1. Define a `Worker` struct that holds an `id` and a `JoinHandle<()>`.
2. Change `ThreadPool` to hold a vector of `Worker` instances.
3. Define a `Worker::new` function that takes an `id` number and returns a `Worker` instance that holds the `id` and a thread spawned with an empty closure.
4. In `ThreadPool::new`, use the `for` loop counter to generate an `id`, create a new `Worker` with that `id`, and store the worker in the vector.

If you're up for a challenge, try implementing these changes on your own before looking at the code in Listing 20-15.

Ready? Here is Listing 20-15 with one way to make the preceding modifications.

Filename: src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers
        }
    }
    // --snip--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-15: Modifying `ThreadPool` to hold `Worker` instances instead of holding threads directly

We've changed the name of the field on `ThreadPool` from `threads` to `workers` because it's now holding `Worker` instances instead of `JoinHandle<()>` instances. We use the counter in the `for` loop as an argument to `Worker::new`, and we store each new `Worker` in the vector named `workers`.

External code (like our server in *src/bin/main.rs*) doesn't need to know the implementation details regarding using a `Worker` struct within `ThreadPool`, so we make the `Worker` struct and its

`new` function private. The `Worker::new` function uses the `id` we give it and stores a `JoinHandle<()>` instance that is created by spawning a new thread using an empty closure.

This code will compile and will store the number of `Worker` instances we specified as an argument to `ThreadPool::new`. But we're *still* not processing the closure that we get in `execute`. Let's look at how to do that next.

23.2.2.7 Sending Requests to Threads via Channels

Now we'll tackle the problem that the closures given to `thread::spawn` do absolutely nothing. Currently, we get the closure we want to execute in the `execute` method. But we need to give `thread::spawn` a closure to run when we create each `Worker` during the creation of the `ThreadPool`.

We want the `Worker` structs that we just created to fetch code to run from a queue held in the `ThreadPool` and send that code to its thread to run.

In Chapter 16, you learned about *channels*—a simple way to communicate between two threads—that would be perfect for this use case. We'll use a channel to function as the queue of jobs, and `execute` will send a job from the `ThreadPool` to the `Worker` instances, which will send the job to its thread. Here is the plan:

1. The `ThreadPool` will create a channel and hold on to the sending side of the channel.
2. Each `Worker` will hold on to the receiving side of the channel.
3. We'll create a new `Job` struct that will hold the closures we want to send down the channel.
4. The `execute` method will send the job it wants to execute down the sending side of the channel.
5. In its thread, the `Worker` will loop over its receiving side of the channel and execute the closures of any jobs it receives.

Let's start by creating a channel in `ThreadPool::new` and holding the sending side in the `ThreadPool` instance, as shown in Listing 20-16. The `Job` struct doesn't hold anything for now but will be the type of item we're sending down the channel.

Filename: src/lib.rs

```
# use std::thread;
// --snip--
use std::sync::mpsc;

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
```

```
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}
#
# struct Worker {
#     id: usize,
#     thread: thread::JoinHandle<()>,
# }
#
# impl Worker {
#     fn new(id: usize) -> Worker {
#         let thread = thread::spawn(|| {});
#
#         Worker {
#             id,
#             thread,
#         }
#     }
# }
```

Listing 20-16: Modifying `ThreadPool` to store the sending end of a channel that sends `Job` instances

In `ThreadPool::new`, we create our new channel and have the pool hold the sending end. This will successfully compile, still with warnings.

Let's try passing a receiving end of the channel into each worker as the thread pool creates them. We know we want to use the receiving end in the thread that the workers spawn, so we'll reference the `receiver` parameter in the closure. The code in Listing 20-17 won't quite compile yet.

Filename: src/lib.rs

```
impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
```

```
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-17: Passing the receiving end of the channel to the workers

We've made some small and straightforward changes: we pass the receiving end of the channel into `Worker::new`, and then we use it inside the closure.

When we try to check this code, we get this error:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
  --> src/lib.rs:27:42
   |
27 |             workers.push(Worker::new(id, receiver));
   |                                          ^^^^^^^^ value moved here in
   previous iteration of loop
   |
   = note: move occurs because `receiver` has type
   `std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
```

The code is trying to pass `receiver` to multiple `Worker` instances. This won't work, as you'll

recall from Chapter 16: the channel implementation that Rust provides is multiple *producer*, single *consumer*. This means we can't just clone the consuming end of the channel to fix this code. Even if we could, that is not the technique we would want to use; instead, we want to distribute the jobs across threads by sharing the single `receiver` between all the workers.

Additionally, taking a job off the channel queue involves mutating the `receiver`, so the threads need a safe way to share and modify `receiver`; otherwise, we might get race conditions (as covered in Chapter 16).

Recall the thread-safe smart pointers discussed in Chapter 16: to share ownership across multiple threads and allow the threads to mutate the value, we need to use `Arc<Mutex<T>>`. The `Arc` type will let multiple workers own the receiver, and `Mutex` will ensure that only one worker gets a job from the receiver at a time. Listing 20-18 shows the changes we need to make.

Filename: src/lib.rs

```
# use std::thread;
# use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;
// --snip--

# pub struct ThreadPool {
#     workers: Vec<Worker>,
#     sender: mpsc::Sender<Job>,
# }
# struct Job;
#
impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender,
        }
    }

    // --snip--
}

```

```
# struct Worker {
#     id: usize,
#     thread: thread::JoinHandle<()>,
# }
#
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--
#         let thread = thread::spawn(|| {
#             receiver;
#         });
#
#         Worker {
#             id,
#             thread,
#         }
    }
}
```

Listing 20-18: Sharing the receiving end of the channel between the workers using `Arc` and `Mutex`

In `ThreadPool::new`, we put the receiving end of the channel in an `Arc` and a `Mutex`. For each new worker, we clone the `Arc` to bump the reference count so the workers can share ownership of the receiving end.

With these changes, the code compiles! We're getting there!

23.2.2.8　Implementing the `execute` Method

Let's finally implement the `execute` method on `ThreadPool`. We'll also change `Job` from a struct to a type alias for a trait object that holds the type of closure that `execute` receives. As discussed the "Type Aliases Create Type Synonyms" section of Chapter 19, type aliases allow us to make long types shorter. Look at Listing 20-19.

Filename: src/lib.rs

```
// --snip--
# pub struct ThreadPool {
#     workers: Vec<Worker>,
#     sender: mpsc::Sender<Job>,
# }
# use std::sync::mpsc;
# struct Worker {}

type Job = Box<FnOnce() + Send + 'static>;

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
```

```
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --snip--
```

Listing 20-19: Creating a `Job` type alias for a `Box` that holds each closure and then sending the job down the channel

After creating a new `Job` instance using the closure we get in `execute`, we send that job down the sending end of the channel. We're calling `unwrap` on `send` for the case that sending fails, which might happen if, for example, we stop all our threads from executing, meaning the receiving end has stopped receiving new messages. At the moment, we can't stop our threads from executing: our threads continue executing as long as the pool exists. The reason we use `unwrap` is that we know the failure case won't happen, but the compiler doesn't know that.

But we're not quite done yet! In the worker, our closure being passed to `thread::spawn` still only *references* the receiving end of the channel. Instead, we need the closure to loop forever, asking the receiving end of the channel for a job and running the job when it gets one. Let's make the change shown in Listing 20-20 to `Worker::new`.

Filename: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Worker {} got a job; executing.", id);

                (*job)();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-20: Receiving and executing the jobs in the worker's thread

Here, we first call `lock` on the `receiver` to acquire the mutex, and then call `unwrap` to panic on any errors. Acquiring a lock might fail if the mutex is in a *poisoned* state, which can happen if some other thread panicked while holding the lock rather than releasing the lock. In this situation, calling `unwrap` to have this thread panic is the correct action to take. Feel free to change this `unwrap` to an `expect` with an error message that is meaningful to you.

If we get the lock on the mutex, we call `recv` to receive a `Job` from the channel. A final `unwrap` moves past any errors here as well, which might occur if the thread holding the sending side of the channel has shut down, similar to how the `send` method returns `Err` if the receiving side shuts down.

The call to `recv` blocks, so if there is no job yet, the current thread will wait until a job becomes available. The `Mutex<T>` ensures that only one `Worker` thread at a time is trying to request a job.

Theoretically, this code should compile. Unfortunately, the Rust compiler isn't perfect yet, and we get this error:

```
error[E0161]: cannot move a value of type std::ops::FnOnce() +
std::marker::Send: the size of std::ops::FnOnce() + std::marker::Send cannot be
statically determined
  --> src/lib.rs:63:17
   |
63 |                 (*job)();
   |                 ^^^^^^
```

This error is fairly cryptic because the problem is fairly cryptic. To call a `FnOnce` closure that is stored in a `Box<T>` (which is what our `Job` type alias is), the closure needs to move itself *out* of the `Box<T>` because the closure takes ownership of `self` when we call it. In general, Rust doesn't allow us to move a value out of a `Box<T>` because Rust doesn't know how big the value inside the `Box<T>` will be: recall in Chapter 15 that we used `Box<T>` precisely because we had something of an unknown size that we wanted to store in a `Box<T>` to get a value of a known size.

As you saw in Listing 17-15, we can write methods that use the syntax `self: Box<Self>`, which allows the method to take ownership of a `Self` value stored in a `Box<T>`. That's exactly what we want to do here, but unfortunately Rust won't let us: the part of Rust that implements behavior when a closure is called isn't implemented using `self: Box<Self>`. So Rust doesn't yet understand that it could use `self: Box<Self>` in this situation to take ownership of the closure and move the closure out of the `Box<T>`.

Rust is still a work in progress with places where the compiler could be improved, but in the future, the code in Listing 20-20 should work just fine. People just like you are working to fix this and other issues! After you've finished this book, we would love for you to join in.

But for now, let's work around this problem using a handy trick. We can tell Rust explicitly that in this case we can take ownership of the value inside the `Box<T>` using `self: Box<Self>`; then, once we have ownership of the closure, we can call it. This involves defining a new trait

`FnBox` with the method `call_box` that will use `self: Box<Self>` in its signature, defining `FnBox` for any type that implements `FnOnce()`, changing our type alias to use the new trait, and changing `Worker` to use the `call_box` method. These changes are shown in Listing 20-21.

Filename: src/lib.rs

```
trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<FnBox + Send + 'static>;

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Worker {} got a job; executing.", id);

                job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-21: Adding a new trait `FnBox` to work around the current limitations of `Box<FnOnce()>`

First, we create a new trait named `FnBox`. This trait has the one method `call_box`, which is similar to the `call` methods on the other `Fn*` traits except that it takes `self: Box<Self>` to take ownership of `self` and move the value out of the `Box<T>`.

Next, we implement the `FnBox` trait for any type `F` that implements the `FnOnce()` trait. Effectively, this means that any `FnOnce()` closures can use our `call_box` method. The implementation of `call_box` uses `(*self)()` to move the closure out of the `Box<T>` and call the closure.

We now need our `Job` type alias to be a `Box` of anything that implements our new trait `FnBox`. This will allow us to use `call_box` in `Worker` when we get a `Job` value instead of invoking the

closure directly. Implementing the `FnBox` trait for any `FnOnce()` closure means we don't have to change anything about the actual values we're sending down the channel. Now Rust is able to recognize that what we want to do is fine.

This trick is very sneaky and complicated. Don't worry if it doesn't make perfect sense; someday, it will be completely unnecessary.

By implementing this trick, our thread pool is in a working state! Give it a `cargo run`, and make some requests:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never used: `workers`
 --> src/lib.rs:7:5
  |
7 |     workers: Vec<Worker>,
  |     ^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `id`
  --> src/lib.rs:61:5
   |
61 |     id: usize,
   |     ^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

warning: field is never used: `thread`
  --> src/lib.rs:62:5
   |
62 |     thread: thread::JoinHandle<()>,
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.99 secs
     Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

Success! We now have a thread pool that executes connections asynchronously. There are never more than four threads created, so our system won't get overloaded if the server receives a lot of requests. If we make a request to `/sleep`, the server will be able to serve other requests

by having another thread run them.

After learning about the `while let` loop in Chapter 18, you might be wondering why we didn't write the worker thread code as shown in Listing 20-22.

Filename: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            while let Ok(job) = receiver.lock().unwrap().recv() {
                println!("Worker {} got a job; executing.", id);

                job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-22: An alternative implementation of `Worker::new` using `while let`

This code compiles and runs but doesn't result in the desired threading behavior: a slow request will still cause other requests to wait to be processed. The reason is somewhat subtle: the `Mutex` struct has no public `unlock` method because the ownership of the lock is based on the lifetime of the `MutexGuard<T>` within the `LockResult<MutexGuard<T>>` that the `lock` method returns. At compile time, the borrow checker can then enforce the rule that a resource guarded by a `Mutex` cannot be accessed unless we hold the lock. But this implementation can also result in the lock being held longer than intended if we don't think carefully about the lifetime of the `MutexGuard<T>`. Because the values in the `while` expression remain in scope for the duration of the block, the lock remains held for the duration of the call to `job.call_box()`, meaning other workers cannot receive jobs.

By using `loop` instead and acquiring the lock and a job within the block rather than outside it, the `MutexGuard` returned from the `lock` method is dropped as soon as the `let job` statement ends. This ensures that the lock is held during the call to `recv`, but it is released before the call to `job.call_box()`, allowing multiple requests to be serviced concurrently.

23.3 Graceful Shutdown and Cleanup

The code in Listing 20-21 is responding to requests asynchronously through the use of a thread pool, as we intended. We get some warnings about the `workers`, `id`, and `thread` fields that we're not using in a direct way that reminds us we're not cleaning up anything. When we use the less elegant ctrl-c method to halt the main thread, all other threads are stopped immediately as well, even if they're in the middle of serving a request.

Now we'll implement the `Drop` trait to call `join` on each of the threads in the pool so they can finish the requests they're working on before closing. Then we'll implement a way to tell the threads they should stop accepting new requests and shut down. To see this code in action, we'll modify our server to only accept two requests before gracefully shutting down its thread pool.

23.3.1 Implementing the `Drop` Trait on `ThreadPool`

Let's start with implementing `Drop` on our thread pool. When the pool is dropped, our threads should all join on to make sure they finish their work. Listing 20-23 shows a first attempt at a `Drop` implementation; this code won't quite work yet.

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

Listing 20-23: Joining each thread when the thread pool goes out of scope

First, we loop through each of the thread pool `workers`. We use `&mut` for this because `self` is a mutable reference, and we also need to be able to mutate `worker`. For each worker, we print a message saying that this particular worker is shutting down, and then we call `join` on that worker's thread. If the call to `join` fails, we use `unwrap` to make Rust panic and go into an ungraceful shutdown.

Here is the error we get when we compile this code:

```
error[E0507]: cannot move out of borrowed content
  --> src/lib.rs:65:13
   |
65 |             worker.thread.join().unwrap();
   |             ^^^^^^ cannot move out of borrowed content
```

The error tells us we can't call `join` because we only have a mutable borrow of each `worker`, and `join` takes ownership of its argument. To solve this issue, we need to move the thread out of the `Worker` instance that owns `thread` so `join` can consume the thread. We did this in Listing 17-15: if `Worker` holds an `Option<thread::JoinHandle<()>` instead, we can call the `take` method on the `Option` to move the value out of the `Some` variant and leave a `None` variant in its place. In other words, a `Worker` that is running will have a `Some` variant in `thread`, and when we want to clean up a worker, we'll replace `Some` with `None` so the worker doesn't have a thread to run.

So we know we want to update the definition of `Worker` like this:

Filename: src/lib.rs

```
# use std::thread;
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

Now let's lean on the compiler to find the other places that need to change. Checking this code, we get two errors:

```
error[E0599]: no method named `join` found for type
`std::option::Option<std::thread::JoinHandle<()>>` in the current scope
  --> src/lib.rs:65:27
   |
65 |             worker.thread.join().unwrap();
   |                           ^^^^

error[E0308]: mismatched types
  --> src/lib.rs:89:13
   |
89 |             thread,
   |             ^^^^^^
   |             |
   |             expected enum `std::option::Option`, found struct
   `std::thread::JoinHandle`
   |             help: try using a variant of the expected type: `Some(thread)`
   |
   = note: expected type `std::option::Option<std::thread::JoinHandle<()>>`
              found type `std::thread::JoinHandle<_>`
```

Let's address the second error, which points to the code at the end of `Worker::new`; we need to wrap the `thread` value in `Some` when we create a new `Worker`. Make the following changes to fix this error:

Filename: src/lib.rs

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--

```

```
        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

The first error is in our `Drop` implementation. We mentioned earlier that we intended to call `take` on the `Option` value to move `thread` out of `worker`. The following changes will do so:

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

As discussed in Chapter 17, the `take` method on `Option` takes the `Some` variant out and leaves `None` in its place. We're using `if let` to destructure the `Some` and get the thread; then we call `join` on the thread. If a worker's thread is already `None`, we know that worker has already had its thread cleaned up, so nothing happens in that case.

23.3.2 Signaling to the Threads to Stop Listening for Jobs

With all the changes we've made, our code compiles without any warnings. But the bad news is this code doesn't function the way we want it to yet. The key is the logic in the closures run by the threads of the `Worker` instances: at the moment we call `join`, but that won't shut down the threads because they `loop` forever looking for jobs. If we try to drop our `ThreadPool` with our current implementation of `drop`, the main thread will block forever waiting for the first thread to finish.

To fix this problem, we'll modify the threads so they listen for either a `Job` to run or a signal that they should stop listening and exit the infinite loop. Instead of `Job` instances, our channel will send one of these two enum variants:

Filename: src/lib.rs

```
# struct Job;
enum Message {
    NewJob(Job),
    Terminate,
```

```
}
```

This `Message` enum will either be a `NewJob` variant that holds the `Job` the thread should run, or it will be a `Terminate` variant that will cause the thread to exit its loop and stop.

We need to adjust the channel to use values of type `Message` rather than type `Job`, as shown in Listing 20-24.

Filename: src/lib.rs

```
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

// --snip--

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(job)).unwrap();
    }
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
        Worker {

        let thread = thread::spawn(move ||{
            loop {
                let message = receiver.lock().unwrap().recv().unwrap();

                match message {
                    Message::NewJob(job) => {
                        println!("Worker {} got a job; executing.", id);

                        job.call_box();
                    },
                    Message::Terminate => {
                        println!("Worker {} was told to terminate.", id);

                        break;
                    },
                }
            }
```

```
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

Listing 20-24: Sending and receiving `Message` values and exiting the loop if a `Worker` receives `Message::Terminate`

To incorporate the `Message` enum, we need to change `Job` to `Message` in two places: the definition of `ThreadPool` and the signature of `Worker::new`. The `execute` method of `ThreadPool` needs to send jobs wrapped in the `Message::NewJob` variant. Then, in `Worker::new` where a `Message` is received from the channel, the job will be processed if the `NewJob` variant is received, and the thread will break out of the loop if the `Terminate` variant is received.

With these changes, the code will compile and continue to function in the same way as it did after Listing 20-21. But we'll get a warning because we aren't creating any messages of the `Terminate` variety. Let's fix this warning by changing our `Drop` implementation to look like Listing 20-25.

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate message to all workers.");

        for _ in &mut self.workers {
            self.sender.send(Message::Terminate).unwrap();
        }

        println!("Shutting down all workers.");

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

Listing 20-25: Sending `Message::Terminate` to the workers before calling `join` on each worker thread

We're now iterating over the workers twice: once to send one `Terminate` message for each worker and once to call `join` on each worker's thread. If we tried to send a message and `join`

immediately in the same loop, we couldn't guarantee that the worker in the current iteration would be the one to get the message from the channel.

To better understand why we need two separate loops, imagine a scenario with two workers. If we used a single loop to iterate through each worker, on the first iteration a terminate message would be sent down the channel and `join` called on the first worker's thread. If that first worker was busy processing a request at that moment, the second worker would pick up the terminate message from the channel and shut down. We would be left waiting on the first worker to shut down, but it never would because the second thread picked up the terminate message. Deadlock!

To prevent this scenario, we first put all of our `Terminate` messages on the channel in one loop; then we join on all the threads in another loop. Each worker will stop receiving requests on the channel once it gets a terminate message. So, we can be sure that if we send the same number of terminate messages as there are workers, each worker will receive a terminate message before `join` is called on its thread.

To see this code in action, let's modify `main` to only accept two requests before gracefully shutting down the server, as shown in Listing 20-26.

Filename: src/bin/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}
```

Listing 20-26: Shut down the server after serving two requests by exiting the loop

You wouldn't want a real-world web server to shut down after serving only two requests. This code just demonstrates that the graceful shutdown and cleanup is in working order.

The `take` method is defined in the `Iterator` trait and limits the iteration to the first two items at most. The `ThreadPool` will go out of scope at the end of `main`, and the `drop` implementation will run.

Start the server with `cargo run`, and make three requests. The third request should error, and in your terminal you should see output similar to this:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
     Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 3 got a job; executing.
Shutting down.
Sending terminate message to all workers.
Shutting down all workers.
Shutting down worker 0
Worker 1 was told to terminate.
Worker 2 was told to terminate.
Worker 0 was told to terminate.
Worker 3 was told to terminate.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

You might see a different ordering of workers and messages printed. We can see how this code works from the messages: workers zero and three got the first two requests, and then on the third request the server stopped accepting connections. When the `ThreadPool` goes out of scope at the end of `main`, its `Drop` implementation kicks in, and the pool tells all workers to terminate. The workers each print a message when they see the terminate message, and then the thread pool calls `join` to shut down each worker thread.

Notice one interesting aspect of this particular execution: the `ThreadPool` sent the terminate messages down the channel, and before any worker received the messages, we tried to join worker 0. Worker 0 had not yet received the terminate message, so the main thread blocked waiting for worker 0 to finish. In the meantime, each of the workers received the termination messages. When worker 0 finished, the main thread waited for the rest of the workers to finish. At that point, they had all received the termination message and were able to shut down.

Congrats! We've now completed our project; we have a basic web server that uses a thread pool to respond asynchronously. We're able to perform a graceful shutdown of the server, which cleans up all the threads in the pool.

Here's the full code for reference:

Filename: src/bin/main.rs

```
extern crate hello;
use hello::ThreadPool;

use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::fs::File;
use std::thread;
use std::time::Duration;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
```

```
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };

     let mut file = File::open(filename).unwrap();
     let mut contents = String::new();

     file.read_to_string(&mut contents).unwrap();

     let response = format!("{}{}", status_line, contents);

     stream.write(response.as_bytes()).unwrap();
     stream.flush().unwrap();
}
```

Filename: src/lib.rs

```
use std::thread;
use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;

enum Message {
    NewJob(Job),
    Terminate,
}

pub struct ThreadPool {
```

```
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<FnBox + Send + 'static>;

impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender,
        }
    }

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(job)).unwrap();
    }
}

```

```
65  impl Drop for ThreadPool {
66      fn drop(&mut self) {
67          println!("Sending terminate message to all workers.");
68
69          for _ in &mut self.workers {
70              self.sender.send(Message::Terminate).unwrap();
71          }
72
73          println!("Shutting down all workers.");
74
75          for worker in &mut self.workers {
76              println!("Shutting down worker {}", worker.id);
77
78              if let Some(thread) = worker.thread.take() {
79                  thread.join().unwrap();
80              }
81          }
82      }
83  }
84
85  struct Worker {
86      id: usize,
87      thread: Option<thread::JoinHandle<()>>,
88  }
89
90  impl Worker {
91      fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
92          Worker {
93
94          let thread = thread::spawn(move ||{
95              loop {
96                  let message = receiver.lock().unwrap().recv().unwrap();
97
98                  match message {
99                      Message::NewJob(job) => {
100                         println!("Worker {} got a job; executing.", id);
101
102                         job.call_box();
103                     },
104                     Message::Terminate => {
105                         println!("Worker {} was told to terminate.", id);
106
107                         break;
108                     },
109                 }
110             }
111         });
112
113         Worker {
114             id,
115             thread: Some(thread),
116         }
117     }
```

```
118 }
```

We could do more here! If you want to continue enhancing this project, here are some ideas:

- Add more documentation to `ThreadPool` and its public methods.
- Add tests of the library's functionality.
- Change calls to `unwrap` to more robust error handling.
- Use `ThreadPool` to perform some task other than serving web requests.
- Find a thread pool crate on *`https://crates.io/*` and implement a similar web server using the crate instead. Then compare its API and robustness to the thread pool we implemented.

23.4 Summary

Well done! You've made it to the end of the book! We want to thank you for joining us on this tour of Rust. You're now ready to implement your own Rust projects and help with other peoples' projects. Keep in mind that there is a welcoming community of other Rustaceans who would love to help you with any challenges you encounter on your Rust journey.

第 24 章

Appendix

The following sections contain reference material you may find useful in your Rust journey.

24.1 Appendix A: Keywords

The following list contains keywords that are reserved for current or future use by the Rust language. As such, they cannot be used as identifiers, such as names of functions, variables, parameters, struct fields, modules, crates, constants, macros, static values, attributes, types, traits, or lifetimes.

24.1.1 Keywords Currently in Use

- `as` - perform primitive casting, disambiguate the specific trait containing an item, or rename items in `use` and `extern crate` statements
- `break` - exit a loop immediately
- `const` - define constant items or constant raw pointers
- `continue` - continue to the next loop iteration
- `crate` - link an external crate or a macro variable representing the crate in which the macro is defined
- `else` - fallback for `if` and `if let` control flow constructs
- `enum` - define an enumeration
- `extern` - link an external crate, function, or variable
- `false` - Boolean false literal
- `fn` - define a function or the function pointer type
- `for` - loop over items from an iterator, implement a trait, or specify a higher-ranked lifetime
- `if` - branch based on the result of a conditional expression

- `impl` - implement inherent or trait functionality
- `in` - part of `for` loop syntax
- `let` - bind a variable
- `loop` - loop unconditionally
- `match` - match a value to patterns
- `mod` - define a module
- `move` - make a closure take ownership of all its captures
- `mut` - denote mutability in references, raw pointers, or pattern bindings
- `pub` - denote public visibility in struct fields, `impl` blocks, or modules
- `ref` - bind by reference
- `return` - return from function
- `Self` - a type alias for the type implementing a trait
- `self` - method subject or current module
- `static` - global variable or lifetime lasting the entire program execution
- `struct` - define a structure
- `super` - parent module of the current module
- `trait` - define a trait
- `true` - Boolean true literal
- `type` - define a type alias or associated type
- `unsafe` - denote unsafe code, functions, traits, or implementations
- `use` - import symbols into scope
- `where` - denote clauses that constrain a type
- `while` - loop conditionally based on the result of an expression

24.1.2 Keywords Reserved for Future Use

The following keywords do not have any functionality but are reserved by Rust for potential future use.

- `abstract`
- `alignof`
- `become`
- `box`
- `do`
- `final`
- `macro`
- `offsetof`

- override
- priv
- proc
- pure
- sizeof
- typeof
- unsized
- virtual
- yield

24.2 Appendix B: Operators and Symbols

This appendix contains a glossary of Rust's syntax, including operators and other symbols that appear by themselves or in the context of paths, generics, trait bounds, macros, attributes, comments, tuples, and brackets.

24.2.1 Operators

The following list contains the operators in Rust, an example of how the operator would appear in context, a short explanation, and whether that operator is overloadable. If an operator is overloadable, the relevant trait to use to overload that operator is listed.

- `!` (`ident!(...)`, `ident!{...}`, `ident![...]`): denotes macro expansion.
- `!` (`!expr`): bitwise or logical complement. Overloadable (`Not`).
- `!=` (`var != expr`): nonequality comparison. Overloadable (`PartialEq`).
- `%` (`expr % expr`): arithmetic remainder. Overloadable (`Rem`).
- `%=` (`var %= expr`): arithmetic remainder and assignment. Overloadable (`RemAssign`).
- `&` (`&expr`, `&mut expr`): borrow.
- `&` (`&type`, `&mut type`, `&'a type`, `&'a mut type`): borrowed pointer type.
- `&` (`expr & expr`): bitwise AND. Overloadable (`BitAnd`).
- `&=` (`var &= expr`): bitwise AND and assignment. Overloadable (`BitAndAssign`).
- `&&` (`expr && expr`): logical AND.
- `*` (`expr * expr`): arithmetic multiplication. Overloadable (`Mul`).
- `*` (`*expr`): dereference.
- `*` (`*const type`, `*mut type`): raw pointer.
- `*=` (`var *= expr`): arithmetic multiplication and assignment. Overloadable (`MulAssign`).
- `+` (`trait + trait`, `'a + trait`): compound type constraint.

- `+` (`expr + expr`): arithmetic addition. Overloadable (`Add`).
- `+=` (`var += expr`): arithmetic addition and assignment. Overloadable (`AddAssign`).
- `,`: argument and element separator.
- `-` (`- expr`): arithmetic negation. Overloadable (`Neg`).
- `-` (`expr - expr`): arithmetic subtraction. Overloadable (`Sub`).
- `-=` (`var -= expr`): arithmetic subtraction and assignment. Overloadable (`SubAssign`).
- `->` (`fn(...) -> type`, `|...| -> type`): function and closure return type.
- `.` (`expr.ident`): member access.
- `..` (`..`, `expr..`, `..expr`, `expr..expr`): right-exclusive range literal.
- `..` (`..expr`): struct literal update syntax.
- `..` (`variant(x, ..)`, `struct_type { x, .. }`): "and the rest" pattern binding.
- `...` (`expr...expr`) *in a pattern*: inclusive range pattern.
- `/` (`expr / expr`): arithmetic division. Overloadable (`Div`).
- `/=` (`var /= expr`): arithmetic division and assignment. Overloadable (`DivAssign`).
- `:` (`pat: type`, `ident: type`): constraints.
- `:` (`ident: expr`): struct field initializer.
- `:` (`'a: loop {...}`): loop label.
- `;`: statement and item terminator.
- `;` (`[...; len]`): part of fixed-size array syntax
- `<<` (`expr << expr`): left-shift. Overloadable (`Shl`).
- `<<=` (`var <<= expr`): left-shift and assignment. Overloadable (`ShlAssign`).
- `<` (`expr < expr`): less-than comparison. Overloadable (`PartialOrd`).
- `<=` (`expr <= expr`): less-than or equal-to comparison. Overloadable (`PartialOrd`).
- `=` (`var = expr`, `ident = type`): assignment/equivalence.
- `==` (`expr == expr`): equality comparison. Overloadable (`PartialEq`).
- `=>` (`pat => expr`): part of match arm syntax.
- `>` (`expr > expr`): greater-than comparison. Overloadable (`PartialOrd`).
- `>=` (`expr >= expr`): greater-than or equal-to comparison. Overloadable (`PartialOrd`).
- `>>` (`expr >> expr`): right-shift. Overloadable (`Shr`).
- `>>=` (`var >>= expr`): right-shift and assignment. Overloadable (`ShrAssign`).
- `@` (`ident @ pat`): pattern binding.
- `^` (`expr ^ expr`): bitwise exclusive OR. Overloadable (`BitXor`).
- `^=` (`var ^= expr`): bitwise exclusive OR and assignment. Overloadable (`BitXorAssign`).
- `|` (`pat | pat`): pattern alternatives.
- `|` (`|…| expr`): closures.
- `|` (`expr | expr`): bitwise OR. Overloadable (`BitOr`).
- `|=` (`var |= expr`): bitwise OR and assignment. Overloadable (`BitOrAssign`).

- `||` (`expr || expr`): logical OR.
- `_`: "ignored" pattern binding. Also used to make integer literals readable.
- `?` (`expr?`): error propagation.

24.2.2 Non-operator Symbols

The following list contains all non-letters that don't function as operators; that is, they don't behave like a function or method call.

24.2.2.1 Stand-Alone Syntax

- `'ident`: named lifetime or loop label.
- `...u8`, `...i32`, `...f64`, `...usize`, *etc.*: numeric literal of specific type.
- `"..."`: string literal.
- `r"..."`, `r#"..."#`, `r##"..."##`, *etc.*: raw string literal, escape characters are not processed.
- `b"..."`: byte string literal, constructs a `[u8]` instead of a string.
- `br"..."`, `br#"..."#`, `br##"..."##`, *etc.*: raw byte string literal, combination of raw and byte string literal.
- `'...'`: character literal.
- `b'...'`: ASCII byte literal.
- `|...| expr`: closure.
- `!`: always empty bottom type for diverging functions.

24.2.2.2 Path-Related Syntax

- `ident::ident`: namespace path.
- `::path`: path relative to the crate root (*i.e.*, an explicitly absolute path).
- `self::path`: path relative to the current module (*i.e.*, an explicitly relative path).
- `super::path`: path relative to the parent of the current module.
- `type::ident`, `<type as trait>::ident`: associated constants, functions, and types.
- `<type>::...`: associated item for a type that cannot be directly named (*e.g.*, `<&T>::...`, `<[T]>::...`, *etc.*).
- `trait::method(...)`: disambiguating a method call by naming the trait that defines it.
- `type::method(...)`: disambiguating a method call by naming the type for which it's defined.
- `<type as trait>::method(...)`: disambiguating a method call by naming the trait *and* type.

24.2.2.3 Generics

- `path<...>` (*e.g.*, `Vec<u8>`): specifies parameters to generic type *in a type*.
- `path::<...>`, `method::<...>` (*e.g.*, `"42".parse::<i32>()`): specifies parameters to generic type, function, or method *in an expression*. Often referred to as *turbofish*.
- `fn ident<...> ...`: define generic function.
- `struct ident<...> ...`: define generic structure.
- `enum ident<...> ...`: define generic enumeration.
- `impl<...> ...`: define generic implementation.
- `for<...> type`: higher-ranked lifetime bounds.
- `type<ident=type>` (*e.g.*, `Iterator<Item=T>`): a generic type where one or more associated types have specific assignments.

24.2.2.4 Trait Bound Constraints

- `T: U`: generic parameter `T` constrained to types that implement `U`.
- `T: 'a`: generic type `T` must outlive lifetime `'a`. When we say that a type "outlives" the lifetime, we mean it cannot transitively contain any references with lifetimes shorter than `'a`.
- `T : 'static`: the generic type `T` contains no borrowed references other than `'static` ones.
- `'b: 'a`: generic lifetime `'b` must outlive lifetime `'a`.
- `T: ?Sized`: allow generic type parameter to be a dynamically sized type.
- `'a + trait`, `trait + trait`: compound type constraint.

24.2.2.5 Macros and Attributes

- `#[meta]`: outer attribute.
- `#![meta]`: inner attribute.
- `$ident`: macro substitution.
- `$ident:kind`: macro capture.
- `$(…)…`: macro repetition.

24.2.2.6 Comments

- `//`: line comment.
- `//!`: inner line doc comment.
- `///`: outer line doc comment.
- `/*...*/`: block comment.

- `/*!...*/`: inner block doc comment.
- `/**...*/`: outer block doc comment.

24.2.2.7 Tuples

- `()`: empty tuple (*aka* unit), both literal and type.
- `(expr)`: parenthesized expression.
- `(expr,)`: single-element tuple expression.
- `(type,)`: single-element tuple type.
- `(expr, ...)`: tuple expression.
- `(type, ...)`: tuple type.
- `expr(expr, ...)`: function call expression. Also used to initialize tuple `struct`s and tuple `enum` variants.
- `ident!(...)`, `ident!{...}`, `ident![...]`: macro invocation.
- `expr.0`, `expr.1`, *etc.*: tuple indexing.

24.2.2.8 Curly Brackets

- `{...}`: block expression.
- `Type {...}`: `struct` literal.

24.2.2.9 Square Brackets

- `[...]`: array literal.
- `[expr; len]`: array literal containing `len` copies of `expr`.
- `[type; len]`: array type containing `len` instances of `type`.
- `expr[expr]`: collection indexing. Overloadable (`Index`, `IndexMut`).
- `expr[..]`, `expr[a..]`, `expr[..b]`, `expr[a..b]`: collection indexing pretending to be collection slicing, using `Range`, `RangeFrom`, `RangeTo`, or `RangeFull` as the "index."

24.3 Appendix C: Derivable Traits

In various places in the book, we've discussed the `derive` attribute that you can apply to a struct or enum definition.

The `derive` attribute generates code that will implement a trait with its own default implementation on the type you've annotated with the `derive` syntax. In this appendix, we provide a reference of all the traits in the standard library that you can use with `derive`. Each section covers:

- What operators and methods deriving this trait will enable
- What the implementation of the trait provided by `derive` does
- What implementing the trait signifies about the type
- The conditions in which you're allowed or not allowed to implement the trait
- Examples of operations that require the trait

If you want different behavior than that provided by the `derive` attribute, consult the standard library documentation for each trait for details on how to manually implement them.

The rest of the traits defined in the standard library can't be implemented on your types using `derive`. These traits don't have sensible default behavior, so it's up to you to implement them in the way that makes sense for what you're trying to accomplish.

An example of a trait that can't be derived is `Display`, which handles formatting for end users. You should always consider the appropriate way to display a type to an end user: for example, what parts of the type should an end user be allowed to see? What parts would they find relevant? What format of the data would be most relevant to them? The Rust compiler doesn't have this insight, so it can't provide appropriate default behavior for you.

The list of derivable traits provided in this appendix is not comprehensive: libraries can implement `derive` for their own traits, making the list of traits you can use `derive` with truly open-ended. Implementing `derive` involves using a procedural macro, which is covered in Appendix D, "Macros."

24.3.1 Debug for Programmer Output

The `Debug` trait enables debug formatting in format strings, which you indicate by adding `:?` within `{}` placeholders.

The `Debug` trait allows you to print instances of a type for debugging purposes, so you and other programmers using your type can inspect an instance at a particular point in a program's execution.

The `Debug` trait is required, for example, in use of the `assert_eq!` macro. This macro prints the values of instances given as arguments if the equality assertion fails so programmers can see why the two instances weren't equal.

24.3.2 PartialEq and Eq for Equality Comparisons

The `PartialEq` trait allows you to compare instances of a type to check for equality and enables use of the `==` and `!=` operators.

Deriving `PartialEq` implements the `eq` method. When `PartialEq` is derived on structs, two instances are equal only if *all* fields are equal and not equal if any fields are not equal. When

derived on enums, each variant is equal to itself and not equal to the other variants.

The `PartialEq` trait is required, for example, with the use of the `assert_eq!` macro, which needs to be able to compare two instances of a type for equality.

The `Eq` trait has no methods. Its purpose is to signal that for every value of the annotated type, the value is equal to itself. The `Eq` trait can only be applied to types that also implement `PartialEq`, although not all types that implement `PartialEq` can implement `Eq`. One example of this is floating point number types: the implementation of floating point numbers states that two instances of the not-a-number (`NaN`) value are not equal to each other.

An example of when `Eq` is required is for keys in a `HashMap` so the `HashMap` can tell whether two keys are the same.

24.3.3 `PartialOrd` and `Ord` for Ordering Comparisons

The `PartialOrd` trait allows you to compare instances of a type for sorting purposes. A type that implements `PartialOrd` can be used with the `<`, `>`, `<=`, and `>=` operators. You can only apply the `PartialOrd` trait to types that also implement `PartialEq`.

Deriving `PartialOrd` implements the `partial_cmp` method, which returns an `Option<Ordering>` that will be `None` when the values given don't produce an ordering. An example of a value that doesn't produce an ordering, even though most values of that type can be compared, is the not-a-number (`NaN`) floating point value. Calling `partial_cmp` with any floating point number and the `NaN` floating point value will return `None`.

When derived on structs, `PartialOrd` compares two instances by comparing the value in each field in the order in which the fields appear in the struct definition. When derived on enums, variants of the enum declared earlier in the enum definition are considered less than the variants listed later.

The `PartialOrd` trait is required, for example, for the `gen_range` method from the `rand` crate that generates a random value in the range specified by a low value and a high value.

The `Ord` trait allows you to know that for any two values of the annotated type, a valid ordering will exist. The `Ord` trait implements the `cmp` method, which returns an `Ordering` rather than an `Option<Ordering>` because a valid ordering will always be possible. You can only apply the `Ord` trait to types that also implement `PartialOrd` and `Eq` (and `Eq` requires `PartialEq`). When derived on structs and enums, `cmp` behaves the same way as the derived implementation for `partial_cmp` does with `PartialOrd`.

An example of when `Ord` is required is when storing values in a `BTreeSet<T>`, a data structure that stores data based on the sort order of the values.

24.3.4 `Clone` and `Copy` for Duplicating Values

The `Clone` trait allows you to explicitly create a deep copy of a value, and the duplication process might involve running arbitrary code and copying heap data. See the "Ways Variables and Data Interact: Clone" section in Chapter 4 for more information on `Clone`.

Deriving `Clone` implements the `clone` method, which when implemented for the whole type, calls `clone` on each of the parts of the type. This means all the fields or values in the type must also implement `Clone` to derive `Clone`.

An example of when `Clone` is required is when calling the `to_vec` method on a slice. The slice doesn't own the type instances it contains, but the vector returned from `to_vec` will need to own its instances, so `to_vec` calls `clone` on each item. Thus, the type stored in the slice must implement `Clone`.

The `Copy` trait allows you to duplicate a value by only copying bits stored on the stack; no arbitrary code is necessary. See the "Stack-Only Data: Copy" section in Chapter 4 for more information on `Copy`.

The `Copy` trait doesn't define any methods to prevent programmers from overloading those methods and violating the assumption that no arbitrary code is being run. That way, all programmers can assume that copying a value will be very fast.

You can derive `Copy` on any type whose parts all implement `Copy`. You can only apply the `Copy` trait to types that also implement `Clone`, because a type that implements `Copy` has a trivial implementation of `Clone` that performs the same task as `Copy`.

The `Copy` trait is rarely required; types that implement `Copy` have optimizations available, meaning you don't have to call `clone`, which makes the code more concise.

Everything possible with `Copy` you can also accomplish with `Clone`, but the code might be slower or have to use `clone` in places.

24.3.5 `Hash` for Mapping a Value to a Value of Fixed Size

The `Hash` trait allows you to take an instance of a type of arbitrary size and map that instance to a value of fixed size, using a hash function. Deriving `Hash` implements the `hash` method. The derived implementation of the `hash` method combines the result of calling `hash` on each of the parts of the type, meaning all fields or values must also implement `Hash` to derive `Hash`.

An example of when `Hash` is required is in storing keys in a `HashMap` to store data efficiently.

24.3.6 `Default` for Default Values

The `Default` trait allows you to create a default value for a type. Deriving `Default` implements the `default` function. The derived implementation of the `default` function calls the `default` function on each part of the type, meaning all fields or values in the type must also implement `Default` to derive `Default`.

The `Default::default` function is commonly used in combination with the struct update syntax discussed in the "Creating Instances From Other Instances With Struct Update Syntax" section in Chapter 5. You can customize a few fields of a struct and then set and use a default value for the rest of the fields by using `..Default::default()`.

The `Default` trait is required when, for example, you use the `unwrap_or_default` method on `Option<T>` instances. If the `Option<T>` is `None`, the `unwrap_or_default` method will return the result of `Default::default` for the type `T` stored in the `Option<T>`.

24.4 Appendix D: Macros

We've used macros like `println!` throughout this book but haven't fully explored what a macro is and how it works. This appendix explains macros as follows:

- What macros are and how they differ from functions
- How to define a declarative macro to do metaprogramming
- How to define a procedural macro to create custom `derive` traits

We're covering the details of macros in an appendix because they're still evolving in Rust. Macros have changed and, in the near future, will change at a quicker rate than the rest of the language and standard library since Rust 1.0, so this section is more likely to date than the rest of the book. Due to Rust's stability guarantees, the code shown here will continue to work with future versions. But there may be additional capabilities or easier ways to write macros that weren't available at the time of this publication. Bear that in mind when you try to implement anything from this appendix.

24.4.1 The Difference Between Macros and Functions

Fundamentally, macros are a way of writing code that writes other code, which is known as *metaprogramming*. In Appendix C, we discussed the `derive` attribute, which generates an implementation of various traits for you. We've also used the `println!` and `vec!` macros throughout the book. All of these macros *expand* to produce more code than the code you've written manually.

Metaprogramming is useful for reducing the amount of code you have to write and maintain,

which is also one of the roles of functions. However, macros have some additional powers that functions don't have.

A function signature must declare the number and type of parameters the function has. Macros, on the other hand, can take a variable number of parameters: we can call `println!("hello")` with one argument or `println!("hello {}", name)` with two arguments. Also, macros are expanded before the compiler interprets the meaning of the code, so a macro can, for example, implement a trait on a given type. A function can't, because it gets called at runtime and a trait needs to be implemented at compile time.

The downside to implementing a macro instead of a function is that macro definitions are more complex than function definitions because you're writing Rust code that writes Rust code. Due to this indirection, macro definitions are generally more difficult to read, understand, and maintain than function definitions.

Another difference between macros and functions is that macro definitions aren't namespaced within modules like function definitions are. To prevent unexpected name clashes when using external crates, you have to explicitly bring the macros into the scope of your project at the same time as you bring the external crate into scope, using the `#[macro_use]` annotation. The following example would bring all the macros defined in the `serde` crate into the scope of the current crate:

```
#[macro_use]
extern crate serde;
```

If `extern crate` was able to bring macros into scope by default without this explicit annotation, you would be prevented from using two crates that happened to define macros with the same name. In practice, this conflict doesn't occur often, but the more crates you use, the more likely it is.

There is one last important difference between macros and functions: you must define or bring macros into scope *before* you call them in a file, whereas you can define functions anywhere and call them anywhere.

24.4.2　Declarative Macros with `macro_rules!` for General Metaprogramming

The most widely used form of macros in Rust are *declarative macros*. These are also sometimes referred to as *macros by example*, *`macro_rules!` macros*, or just plain *macros*. At their core, declarative macros allow you to write something similar to a Rust `match` expression. As discussed in Chapter 6, `match` expressions are control structures that take an expression, compare the resulting value of the expression to patterns, and then run the code associated with the matching pattern. Macros also compare a value to patterns that have code associated with them; in this situation, the value is the literal Rust source code passed to the macro, the patterns are compared

with the structure of that source code, and the code associated with each pattern is the code that replaces the code passed to the macro. This all happens during compilation.

To define a macro, you use the `macro_rules!` construct. Let's explore how to use `macro_rules!` by looking at how the `vec!` macro is defined. Chapter 8 covered how we can use the `vec!` macro to create a new vector with particular values. For example, the following macro creates a new vector with three integers inside:

```
let v: Vec<u32> = vec![1, 2, 3];
```

We could also use the `vec!` macro to make a vector of two integers or a vector of five string slices: we wouldn't be able to use a function to do the same because we wouldn't know the number or type of values up front.

Let's look at a slightly simplified definition of the `vec!` macro in Listing D-1:

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Listing D-1: A simplified version of the `vec!` macro definition

> Note: The actual definition of the `vec!` macro in the standard library includes code to preallocate the correct amount of memory up front. That code is an optimization that we don't include here to make the example simpler.

The `#[macro_export]` annotation indicates that this macro should be made available whenever the crate in which we're defining the macro is imported. Without this annotation, even if someone depending on this crate uses the `#[macro_use]` annotation, the macro wouldn't be brought into scope.

We then start the macro definition with `macro_rules!` and the name of the macro we're defining *without* the exclamation mark. The name, in this case `vec`, is followed by curly brackets denoting the body of the macro definition.

The structure in the `vec!` body is similar to the structure of a `match` expression. Here we have one arm with the pattern `( $( $x:expr ),* )`, followed by `=>` and the block of code associated with this pattern. If the pattern matches, the associated block of code will be emitted. Given

that this is the only pattern in this macro, there is only one valid way to match; any other will be an error. More complex macros will have more than one arm.

Valid pattern syntax in macro definitions is different than the pattern syntax covered in Chapter 18 because macro patterns are matched against Rust code structure rather than values. Let's walk through what the pieces of the pattern in Listing D-1 mean; for the full macro pattern syntax, see the reference.

First, a set of parentheses encompasses the whole pattern. Next comes a dollar sign (`$`) followed by a set of parentheses, which captures values that match the pattern within the parentheses for use in the replacement code. Within `$()` is `$x:expr`, which matches any Rust expression and gives the expression the name `$x`.

The comma following `$()` indicates that a literal comma separator character could optionally appear after the code that matches the code captured in `$()`. The `*` following the comma specifies that the pattern matches zero or more of whatever precedes the `*`.

When we call this macro with `vec![1, 2, 3];`, the `$x` pattern matches three times with the three expressions `1`, `2`, and `3`.

Now let's look at the pattern in the body of the code associated with this arm: the `temp_vec.push()` code within the `$()*` part is generated for each part that matches `$()` in the pattern, zero or more times depending on how many times the pattern matches. The `$x` is replaced with each expression matched. When we call this macro with `vec![1, 2, 3];`, the code generated that replaces this macro call will be the following:

```
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
```

We've defined a macro that can take any number of arguments of any type and can generate code to create a vector containing the specified elements.

Given that most Rust programmers will *use* macros more than *write* macros, we won't discuss `macro_rules!` any further. To learn more about how to write macros, consult the online documentation or other resources, such as "The Little Book of Rust Macros".

24.4.3 Procedural Macros for Custom `derive`

The second form of macros is called *procedural macros* because they're more like functions (which are a type of procedure). Procedural macros accept some Rust code as an input, operate on that code, and produce some Rust code as an output rather than matching against patterns and replacing the code with other code as declarative macros do. At the time of this writing, you can only define procedural macros to allow your traits to be implemented on a type by specifying

the trait name in a `derive` annotation.

We'll create a crate named `hello_macro` that defines a trait named `HelloMacro` with one associated function named `hello_macro`. Rather than making our crate users implement the `HelloMacro` trait for each of their types, we'll provide a procedural macro so users can annotate their type with `#[derive(HelloMacro)]` to get a default implementation of the `hello_macro` function. The default implementation will print `Hello, Macro! My name is TypeName!` where `TypeName` is the name of the type on which this trait has been defined. In other words, we'll write a crate that enables another programmer to write code like Listing D-2 using our crate.

Filename: src/main.rs

```
extern crate hello_macro;
#[macro_use]
extern crate hello_macro_derive;

use hello_macro::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Listing D-2: The code a user of our crate will be able to write when using our procedural macro

This code will print `Hello, Macro! My name is Pancakes!` when we're done. The first step is to make a new library crate, like this:

```
$ cargo new hello_macro --lib
```

Next, we'll define the `HelloMacro` trait and its associated function:

Filename: src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

We have a trait and its function. At this point, our crate user could implement the trait to achieve the desired functionality, like so:

```
extern crate hello_macro;

use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
```

```
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

However, they would need to write the implementation block for each type they wanted to use with `hello_macro`; we want to spare them from having to do this work.

Additionally, we can't yet provide a default implementation for the `hello_macro` function that will print the name of the type the trait is implemented on: Rust doesn't have reflection capabilities, so it can't look up the type's name at runtime. We need a macro to generate code at compile time.

The next step is to define the procedural macro. At the time of this writing, procedural macros need to be in their own crate. Eventually, this restriction might be lifted. The convention for structuring crates and macro crates is as follows: for a crate named `foo`, a custom derive procedural macro crate is called `foo_derive`. Let's start a new crate called `hello_macro_derive` inside our `hello_macro` project:

```
$ cargo new hello_macro_derive --lib
```

Our two crates are tightly related, so we create the procedural macro crate within the directory of our `hello_macro` crate. If we change the trait definition in `hello_macro`, we'll have to change the implementation of the procedural macro in `hello_macro_derive` as well. The two crates will need to be published separately, and programmers using these crates will need to add both as dependencies and bring them both into scope. We could instead have the `hello_macro` crate use `hello_macro_derive` as a dependency and reexport the procedural macro code. But the way we've structured the project makes it possible for programmers to use `hello_macro` even if they don't want the `derive` functionality.

We need to declare the `hello_macro_derive` crate as a procedural macro crate. We'll also need functionality from the `syn` and `quote` crates, as you'll see in a moment, so we need to add them as dependencies. Add the following to the *Cargo.toml* file for `hello_macro_derive`:

Filename: hello*macro*derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "0.11.11"
quote = "0.3.15"
```

To start defining the procedural macro, place the code in Listing D-3 into your *src/lib.rs* file for the `hello_macro_derive` crate. Note that this code won't compile until we add a definition for the `impl_hello_macro` function.

Notice the way we've split the functions in D-3; this will be the same for almost every procedural macro crate you see or create, because it makes writing a procedural macro more convenient. What you choose to do in the place where the `impl_hello_macro` function is called will be different depending on your procedural macro's purpose.

Filename: hello*macro*derive/src/lib.rs

```
extern crate proc_macro;
extern crate syn;
#[macro_use]
extern crate quote;

use proc_macro::TokenStream;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a string representation of the type definition
    let s = input.to_string();

    // Parse the string representation
    let ast = syn::parse_derive_input(&s).unwrap();

    // Build the impl
    let gen = impl_hello_macro(&ast);

    // Return the generated impl
    gen.parse().unwrap()
}
```

Listing D-3: Code that most procedural macro crates will need to have for processing Rust code

We've introduced three new crates: `proc_macro`, `syn`, and `quote`. The `proc_macro` crate comes with Rust, so we didn't need to add that to the dependencies in *Cargo.toml*. The `proc_macro` crate allows us to convert Rust code into a string containing that Rust code. The `syn` crate parses Rust code from a string into a data structure that we can perform operations on. The `quote` crate takes `syn` data structures and turns them back into Rust code. These crates make it much simpler to parse any sort of Rust code we might want to handle: writing a full parser for Rust code is no simple task.

The `hello_macro_derive` function will get called when a user of our library specifies `#[derive (HelloMacro)]` on a type. The reason is that we've annotated the `hello_macro_derive` function here with `proc_macro_derive` and specified the name, `HelloMacro`, which matches our trait name; that's the convention most procedural macros follow.

This function first converts the `input` from a `TokenStream` to a `String` by calling `to_string`. This `String` is a string representation of the Rust code for which we are deriving `HelloMacro`. In the example in Listing D-2, `s` will have the `String` value `struct Pancakes;` because that is the Rust code we added the `#[derive(HelloMacro)]` annotation to.

> Note: At the time of this writing, you can only convert a `TokenStream` to a string. A richer API will exist in the future.

Now we need to parse the Rust code `String` into a data structure that we can then interpret and perform operations on. This is where `syn` comes into play. The `parse_derive_input` function in `syn` takes a `String` and returns a `DeriveInput` struct representing the parsed Rust code. The following code shows the relevant parts of the `DeriveInput` struct we get from parsing the string `struct Pancakes;`:

```
DeriveInput {
    // --snip--

    ident: Ident(
        "Pancakes"
    ),
    body: Struct(
        Unit
    )
}
```

The fields of this struct show that the Rust code we've parsed is a unit struct with the `ident` (identifier, meaning the name) of `Pancakes`. There are more fields on this struct for describing all sorts of Rust code; check the `syn` documentation for `DeriveInput` for more information.

At this point, we haven't defined the `impl_hello_macro` function, which is where we'll build the new Rust code we want to include. But before we do, note that the last part of this `hello_macro_derive` function uses the `parse` function from the `quote` crate to turn the output of the `impl_hello_macro` function back into a `TokenStream`. The returned `TokenStream` is added to the code that our crate users write, so when they compile their crate, they get extra functionality that we provide.

You might have noticed that we're calling `unwrap` to panic if the calls to the `parse_derive_input` or `parse` functions fail here. Panicking on errors is necessary in procedural macro code because `proc_macro_derive` functions must return `TokenStream` rather than `Result` to conform to the procedural macro API. We've chosen to simplify this example by using `unwrap`; in production code, you should provide more specific error messages about what went wrong by using `panic!` or `expect`.

Now that we have the code to turn the annotated Rust code from a `TokenStream` into a `String` and a `DeriveInput` instance, let's generate the code that implements the `HelloMacro` trait on the

annotated type:

Filename: hello*macro*derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> quote::Tokens {
    let name = &ast.ident;
    quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}", stringify!(#name));
            }
        }
    }
}
```

We get an `Ident` struct instance containing the name (identifier) of the annotated type using `ast.ident`. The code in Listing D-2 specifies that the `name` will be `Ident("Pancakes")`.

The `quote!` macro lets us write the Rust code that we want to return and convert it into `quote::Tokens`. This macro also provides some very cool templating mechanics; we can write `#name` and `quote!` will replace it with the value in the variable named `name`. You can even do some repetition similar to the way regular macros work. Check out the `quote` crate's docs for a thorough introduction.

We want our procedural macro to generate an implementation of our `HelloMacro` trait for the type the user annotated, which we can get by using `#name`. The trait implementation has one function, `hello_macro`, whose body contains the functionality we want to provide: printing `Hello, Macro! My name is` and then the name of the annotated type.

The `stringify!` macro used here is built into Rust. It takes a Rust expression, such as `1 + 2`, and at compile time turns the expression into a string literal, such as `"1 + 2"`. This is different than `format!` or `println!`, which evaluate the expression and then turn the result into a `String`. There is a possibility that the `#name` input might be an expression to print literally, so we use `stringify!`. Using `stringify!` also saves an allocation by converting `#name` to a string literal at compile time.

At this point, `cargo build` should complete successfully in both `hello_macro` and `hello_macro_derive`. Let's hook up these crates to the code in Listing D-2 to see the procedural macro in action! Create a new binary project in your *projects* directory using `cargo new --bin pancakes`. We need to add `hello_macro` and `hello_macro_derive` as dependencies in the `pancakes` crate's *Cargo.toml*. If you're publishing your versions of `hello_macro` and `hello_macro_derive` to `*https://crates.io/*`, they would be regular dependencies; if not, you can specify them as `path` dependencies as follows:

```
[dependencies]
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

Put the code from Listing D-2 into *src/main.rs*, and run `cargo run`: it should print `Hello, Macro! My name is Pancakes!` The implementation of the `HelloMacro` trait from the procedural macro was included without the `pancakes` crate needing to implement it; the `#[derive(HelloMacro)]` added the trait implementation.

24.4.4 The Future of Macros

In the future, Rust will expand declarative and procedural macros. Rust will use a better declarative macro system with the `macro` keyword and will add more types of procedural macros for more powerful tasks than just `derive`. These systems are still under development at the time of this publication; please consult the online Rust documentation for the latest information.

24.5 Appendix E: Translations of the Book

For resources in languages other than English. Most are still in progress; see the Translations label to help or let us know about a new translation!

- Português (BR)
- Português (PT)
- Tiếng việt
- 体中文, alternate
- У к р а　н с ь к а
- Español
- Italiano
- Р у с с к и й
-
- 日本語
- Français
- Polski
-
- Cebuano
- Tagalog

第 25 章

Appendix F - Newest Features

This appendix documents features that have been added to stable Rust since the main part of the book was completed.

25.1 Field init shorthand

We can initialize a data structure (struct, enum, union) with named fields, by writing `fieldname` as a shorthand for `fieldname: fieldname`. This allows a compact syntax for initialization, with less duplication:

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

fn main() {
    let name = String::from("Peter");
    let age = 27;

    // Using full syntax:
    let peter = Person { name: name, age: age };

    let name = String::from("Portia");
    let age = 27;

    // Using field init shorthand:
    let portia = Person { name, age };

    println!("{:?}", portia);
}
```

25.2　Returning from loops

One of the uses of a `loop` is to retry an operation you know can fail, such as checking if a thread completed its job. However, you might need to pass the result of that operation to the rest of your code. If you add it to the `break` expression you use to stop the loop, it will be returned by the broken loop:

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    assert_eq!(result, 20);
}
```

25.3　Nested groups in `use` declarations

If you have a complex module tree with many different submodules and you need to import a few items from each one, it might be useful to group all the imports in the same declaration to keep your code clean and avoid repeating the base modules' name.

The `use` declaration supports nesting to help you in those cases, both with simple imports and glob ones. For example this snippets imports `bar`, `Foo`, all the items in `baz` and `Bar`:

```
# #![allow(unused_imports, dead_code)]
#
# mod foo {
#     pub mod bar {
#         pub type Foo = ();
#     }
#     pub mod baz {
#         pub mod quux {
#             pub type Bar = ();
#         }
#     }
# }
#
use foo::{
    bar::{self, Foo},
    baz::{*, quux::Bar},
};
```

```
#
# fn main() {}
```

25.4　Inclusive ranges

Previously, when a range (`..` or `...`) was used as an expression, it had to be `..`, which is exclusive of the upper bound, while patterns had to use `...`, which is inclusive of the upper bound. Now, `..=` is accepted as syntax for inclusive ranges in both expression and range context:

```
fn main() {
    for i in 0 ..= 10 {
        match i {
            0 ..= 5 => println!("{}: low", i),
            6 ..= 10 => println!("{}: high", i),
            _ => println!("{}: out of range", i),
        }
    }
}
```

The `...` syntax is still accepted in matches, but it is not accepted in expressions. `..=` should be preferred.

25.5　128-bit integers

Rust 1.26.0 added 128-bit integer primitives:

- `u128`: A 128-bit unsigned integer with range [0, 2^128 - 1]
- `i128`: A 128-bit signed integer with range [-(2^127), 2^127 - 1]

These primitives are implemented efficiently via LLVM support. They are available even on platforms that don't natively support 128-bit integers and can be used like the other integer types.

These primitives can be very useful for algorithms that need to use very large integers efficiently, such as certain cryptographic algorithms.

第 26 章

Appendix G - How Rust is Made and "Nightly Rust"

This appendix is about how Rust is made and how that affects you as a Rust developer. We mentioned that the output in this book was generated by stable Rust 1.21.0, but any examples that compile should continue to compile in any stable version of Rust greater than that. This section is to explain how we ensure this is true!

26.0.1 Stability Without Stagnation

As a language, Rust cares a *lot* about the stability of your code. We want Rust to be a rock-solid foundation you can build on, and if things were constantly changing, that would be impossible. At the same time, if we can't experiment with new features, we may not find out important flaws until after their release, when we can no longer change things.

Our solution to this problem is what we call "stability without stagnation", and our guiding principle is this: you should never have to fear upgrading to a new version of stable Rust. Each upgrade should be painless, but should also bring you new features, fewer bugs, and faster compile times.

26.0.2 Choo, Choo! Release Channels and Riding the Trains

Rust development operates on a *train schedule*. That is, all development is done on the `master` branch of the Rust repository. Releases follow a software release train model, which has been used by Cisco IOS and other software projects. There are three *release channels* for Rust:

- Nightly

- Beta
- Stable

Most Rust developers primarily use the stable channel, but those who want to try out experimental new features may use nightly or beta.

Here's an example of how the development and release process works: let's assume that the Rust team is working on the release of Rust 1.5. That release happened in December of 2015, but it will provide us with realistic version numbers. A new feature is added to Rust: a new commit lands on the `master` branch. Each night, a new nightly version of Rust is produced. Every day is a release day, and these releases are created by our release infrastructure automatically. So as time passes, our releases look like this, once a night:

```
nightly: * - - * - - *
```

Every six weeks, it's time to prepare a new release! The `beta` branch of the Rust repository branches off from the `master` branch used by nightly. Now, there are two releases:

```
nightly: * - - * - - *
                    |
beta:               *
```

Most Rust users do not use beta releases actively, but test against beta in their CI system to help Rust discover possible regressions. In the meantime, there's still a nightly release every night:

```
nightly: * - - * - - * - - * - - *
                    |
beta:               *
```

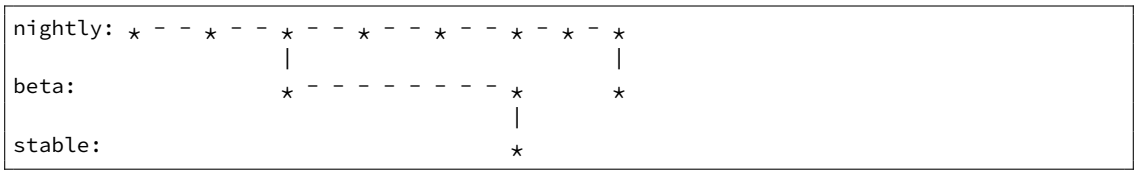
Let's say a regression is found. Good thing we had some time to test the beta release before the regression snuck into a stable release! The fix is applied to `master`, so that nightly is fixed, and then the fix is backported to the `beta` branch, and a new release of beta is produced:

```
nightly: * - - * - - * - - * - - * - - *
                    |
beta:               * - - - - - - - - *
```

Six weeks after the first beta was created, it's time for a stable release! The `stable` branch is produced from the `beta` branch:

```
nightly: * - - * - - * - - * - - * - - * - * - *
                    |
beta:               * - - - - - - - - *
                                      |
stable:                               *
```

Hooray! Rust 1.5 is done! However, we've forgotten one thing: because the six weeks have gone by, we also need a new beta of the *next* version of Rust, 1.6. So after `stable` branches off of `beta`, the next version of `beta` branches off of `nightly` again:



This is called the “train model” because every six weeks, a release “leaves the station”, but still has to take a journey through the beta channel before it arrives as a stable release.

Rust releases every six weeks, like clockwork. If you know the date of one Rust release, you can know the date of the next one: it's six weeks later. A nice aspect of having releases scheduled every six weeks is that the next train is coming soon. If a feature happens to miss a particular release, there's no need to worry: another one is happening in a short time! This helps reduce pressure to sneak possibly unpolished features in close to the release deadline.

Thanks to this process, you can always check out the next build of Rust and verify for yourself that it's easy to upgrade to: if a beta release doesn't work as expected, you can report it to the team and get it fixed before the next stable release happens! Breakage in a beta release is relatively rare, but `rustc` is still a piece of software, and bugs do exist.

26.0.3 Unstable Features

There's one more catch with this release model: unstable features. Rust uses a technique called “feature flags” to determine what features are enabled in a given release. If a new feature is under active development, it lands on `master`, and therefore, in nightly, but behind a *feature flag*. If you, as a user, wish to try out the work-in-progress feature, you can, but you must be using a nightly release of Rust and annotate your source code with the appropriate flag to opt in.

If you're using a beta or stable release of Rust, you can't use any feature flags. This is the key that allows us to get practical use with new features before we declare them stable forever. Those who wish to opt into the bleeding edge can do so, and those who want a rock-solid experience can stick with stable and know that their code won't break. Stability without stagnation.

This book only contains information about stable features, as in-progress features are still changing, and surely they'll be different between when this book was written and when they get enabled in stable builds. You can find documentation for nightly-only features online.

26.0.4 Rustup and the Role of Rust Nightly

Rustup makes it easy to change between different release channels of Rust, on a global or per-project basis. By default, you'll have stable Rust installed. To install nightly, for example:

```
$ rustup install nightly
```

You can see all of the *toolchains* (releases of Rust and associated components) you have installed with `rustup` as well. Here's an example on one of your authors' computers:

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

As you can see, the stable toolchain is the default. Most Rust users use stable most of the time. You might want to use stable most of the time, but use nightly on a specific project, because you care about a cutting-edge feature. To do so, you can use `rustup override` in that project's directory to set the nightly toolchain as the one `rustup` should use when you're in that directory:

```
$ cd ~/projects/needs-nightly
$ rustup override add nightly
```

Now, every time you call `rustc` or `cargo` inside of *~/projects/needs-nightly*, `rustup` will make sure that you are using nightly Rust, rather than your default of stable Rust. This comes in handy when you have a lot of Rust projects!

26.0.5 The RFC Process and Teams

So how do you learn about these new features? Rust's development model follows a *Request For Comments (RFC) process*. If you'd like an improvement in Rust, you can write up a proposal, called an RFC.

Anyone can write RFCs to improve Rust, and the proposals are reviewed and discussed by the Rust team, which is comprised of many topic subteams. There's a full list of the teams on Rust's website, which includes teams for each area of the project: language design, compiler implementation, infrastructure, documentation, and more. The appropriate team reads the proposal and the comments, writes some comments of their own, and eventually, there's consensus to accept or reject the feature.

If the feature is accepted, an issue is opened on the Rust repository, and someone can implement it. The person who implements it very well may not be the person who proposed the feature in the first place! When the implementation is ready, it lands on the `master` branch behind a feature gate, as we discussed in the “Unstable Features” section.

After some time, once Rust developers who use nightly releases have been able to try out the new feature, team members will discuss the feature, how it’s worked out on nightly, and decide if it should make it into stable Rust or not. If the decision is to move forward, the feature gate is removed, and the feature is now considered stable! It rides the trains into a new stable release of Rust.