2024/10/25

デプロイを任されたので、教わった通りにデプロイしたら障害になった件

～俺のやらかしを越えてゆけ～

izumitomo

techouse

Kaigi on Rails 2024

2024.10.25 (Fri.)- 26 (Sat.)

@有明セントラルタワーホール＆カンファレンス

1日目も終盤！
長丁場ですので
アイスブレイクから

有明といえば
ビッグサイトですよね！

ビッグサイトの思い出

- 営業イベントにSalesとして参加
  - 出展者がブースでサービスを宣伝し、来場者は興味のあるブースで話を聞く
- 弊社サービス『クラウドハウス』も出展
  - 生身の相手に対し、自分の言葉で自社プロダクトを売ることに挑戦

来場者でごった返す会場

弊社ブースの様子

ブースの前を通りかかった人を
何とかブースに連れ込もうとしている自分

ビッグサイトの思い出

- あの会社はああやってサービスを訴求しているのか…
- 企業の担当者はそういう課題感を持ってるのか…
- 俺が作ったあの超大作機能、相手に全然刺さらねえ…

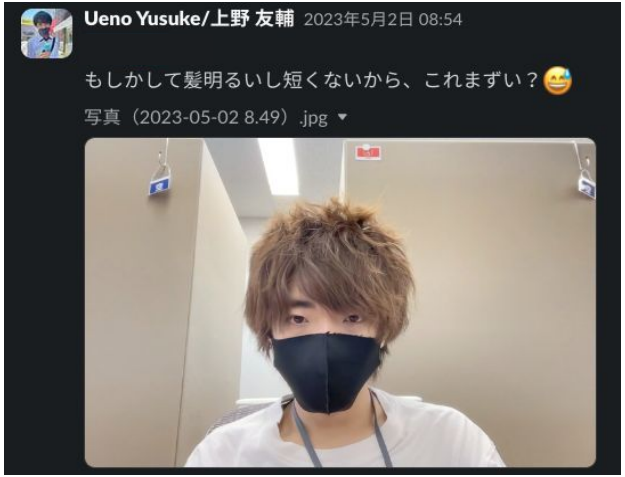
ビッグサイトの思い出

- あの会社はああやってサービスを訴求しているのか…
- 企業の担当者はそういう課題感を持ってるのか…
- 俺が作ったあの超大作機能、相手に全然刺さらねえ…

⇒これがプロダクト作りに欠かせないinsightってやつか…！

ビッグサイトの思い出　～郷に入っては郷に従え～

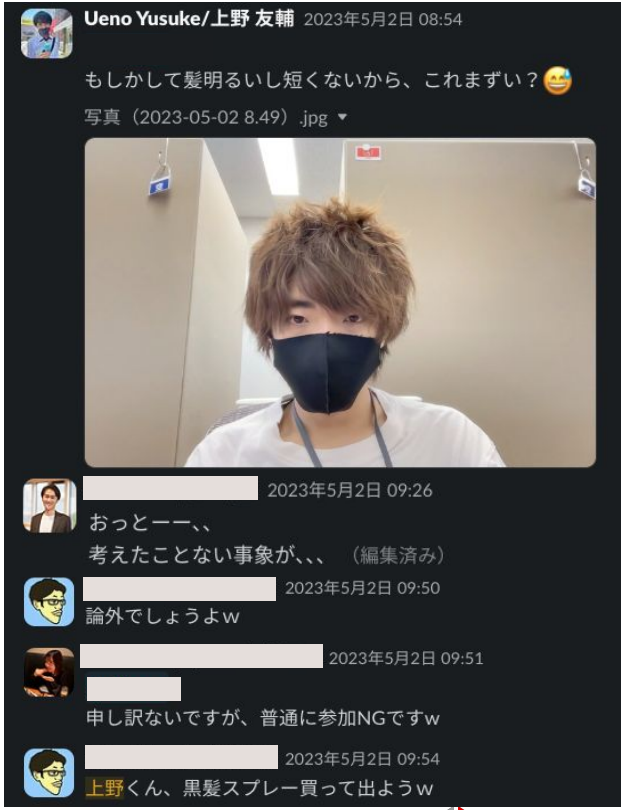
- エンジニアも顧客の前に立ってみてはいかが？
  - ただし、身なりには気をつけよう！
  - 我々エンジニアは時にあり得ない判断を平然とやってのける



実際にあった怖い話

ビッグサイトの思い出　～郷に入っては郷に従え～

- エンジニアも顧客の前に立ってみてはいかが？
  - ただし、身なりには気をつけよう！
  - 我々エンジニアは時にあり得ない判断を平然とやってのける



実際にあった怖い話

自己紹介

上野 友輔@izumitomo

- 株式会社Techouse
  - 去年ブースでガラポンやってた会社
- Rails歴 2年
  - 普段はRails x GraphQL で開発
- Kaigi on Rails初登壇🙌

デプロイしてますか？✋

そのデプロイの仕組み、
説明できますか？✋

デプロイとは

- デプロイとは開発したアプリケーションを本番環境に移行し、実際にユーザが利用できる状態にする一連のプロセス
  - 🫠 CI/CDプロセスが整備されると自然と目が向かなくなる
  - 😑 特に初学者はブラックボックスになりがち
  - 🤯 そして<u>なにかとやらかす</u>

今日話すこと・持ち帰ってもらえるもの

今日話すこと

- デプロイで実際にやらかした失敗とその対応

持ち帰ってもらえるもの

- Railsのダウンタイムのないデプロイにおける注意点
- 初学者がデプロイに目を向けることのメリット

ある日

上野くん、デプロイ任せていい？

退職間近の先輩社員

任せてください！

あの緑のボタン押すだけですよね！

Merge pull request

（新卒1年目）

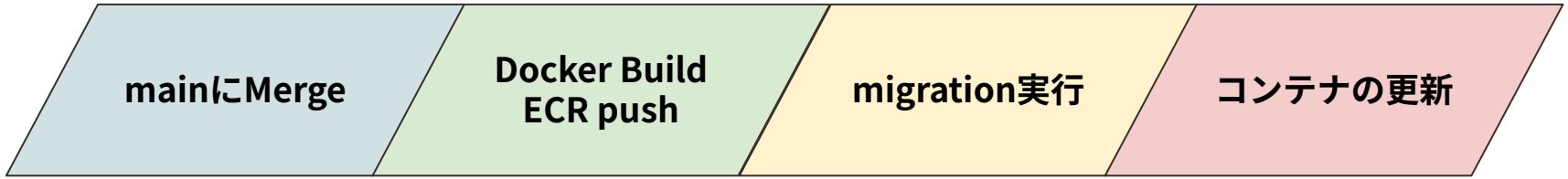
……

……

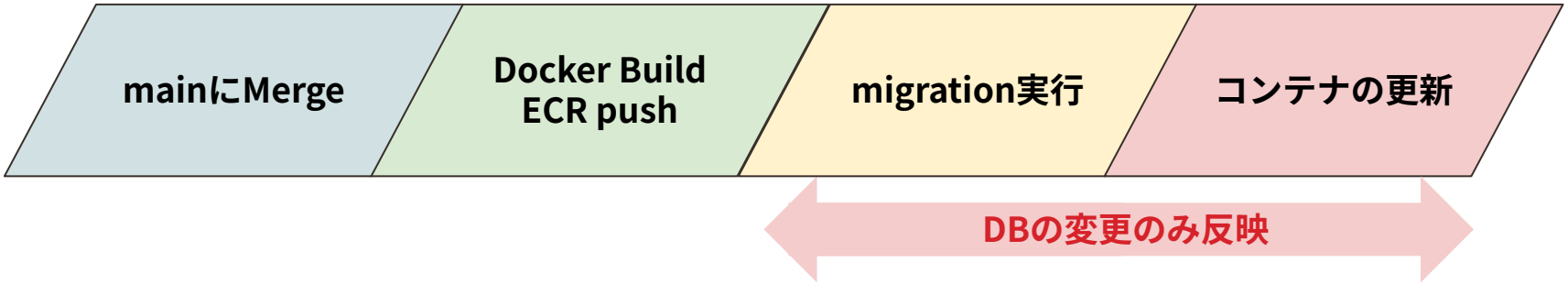
そうだよ

- あっさりデプロイを託されるも、流石に不安だったので
  どのようにデプロイされているかを軽く整理
  - RailsアプリケーションをAmazon ECS on Fargateで管理
  - GitHub Actionでスクリプトを実行してデプロイ
  - ゼロダウンタイムデプロイでサービスは閉塞しない

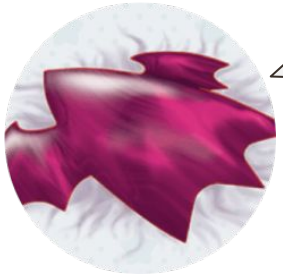
デプロイの流れ



デプロイの流れ

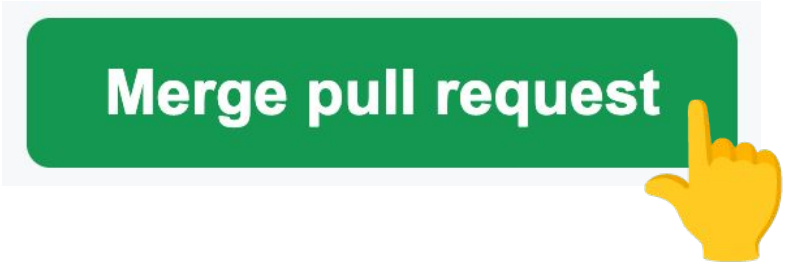


migrationが実行されてからECSコンテナの更新が完了するまでの間は、
DBの変更だけ反映され、アプリケーションコードは古いままとなる
⇒テーブルやカラムのDELETEやRENAMEなどは要注意

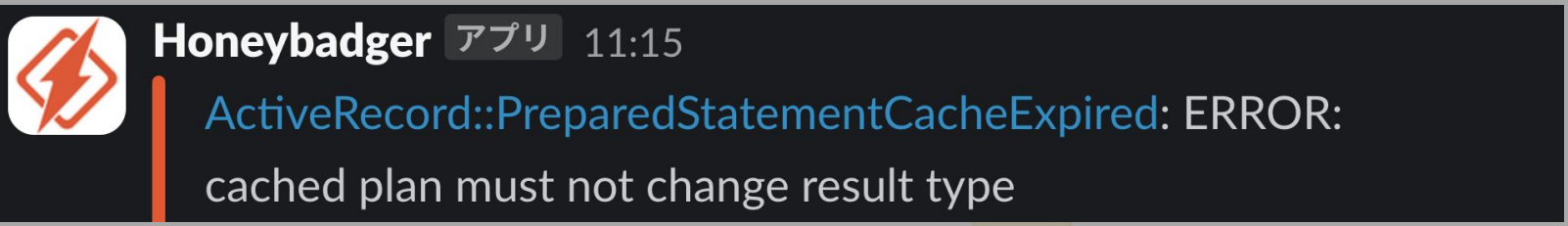


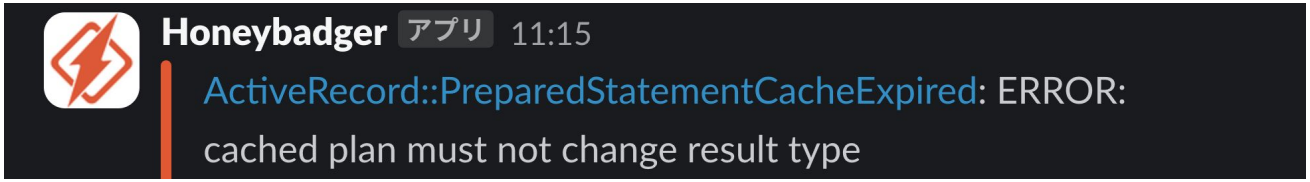
DBとアプリケーション間の不整合に
注意すればいいってことか🤔

- さっそくデプロイの機会が訪れる
  - ある機能開発でDBの変更のみを先に本番に反映しておくことに
    - テーブルにカラムを追加するだけの変更で追加したカラムに依存するアプリケーションコードは存在しない

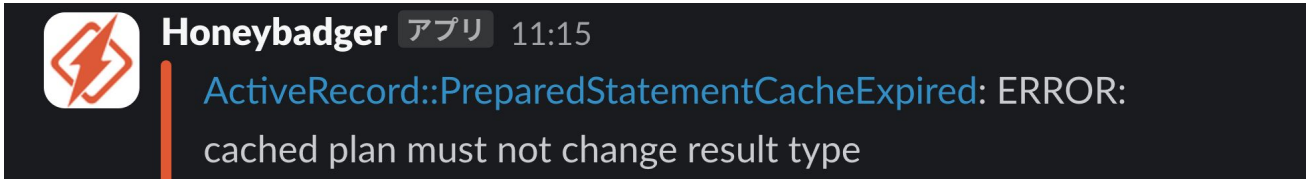


無事にデプロイ完了🙌

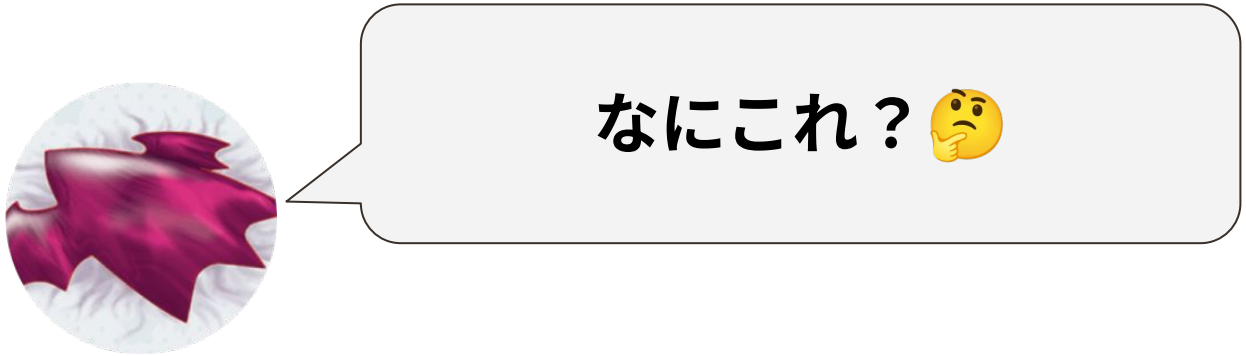




デプロイ直後、サービスの例外通知用のslackチャンネルに1件の例外が通知された



デプロイ直後、サービスの例外通知用のslackチャンネルに1件の例外が通知された



Honeybadger アプリ 11:15

ActiveRecord::PreparedStatementCacheExpired: ERROR: cached plan must not change result type

Honeybadger アプリ 11:15

ActiveRecord::PreparedStatementCacheExpired: ERROR: cached plan must not change result type

ん？🤔

ActiveRecord::PreparedStatementCacheExpired: ERROR:

cached plan must not change result type

Honeybadger アプリ 11:15

ActiveRecord::PreparedStatementCacheExpired: ERROR:

cached plan must not change result type

Honeybadger アプリ 11:15

ActiveRecord::PreparedStatementCacheExpired: ERROR:

cached plan must not change result type

Honeybadger アプリ 11:15

ActiveRecord::PreparedStatementCacheExpired: ERROR:

cached plan must not change result type

Honeybadger アプリ 11:15

ActiveRecord::PreparedStatementCacheExpired: ERROR:

cached plan must not change result type

Honeybadger アプリ 11:15

ActiveRecord::PreparedStatementCacheExpired: ERROR:

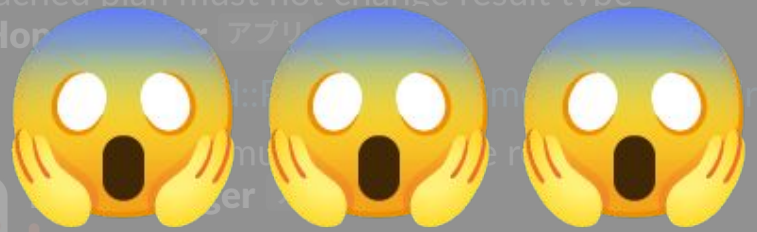
cached plan must not change result type

Honeybadger アプリ 11:15

ActiveRecord::PreparedStatementCacheExpired: ERROR:

cached plan must not change result type

死ぬほど焦る



焦るな！！
まずは落ち着いて状況整理から！

退職間近の先輩社員

原因調査をしたくなるのもわかるが
まずは関係部署に連絡と顧客影響を調査！

了解です…申し訳ないです…

影響範囲の調査

- 例外は全てSidekiqの非同期処理の中で発生していた
  - ジョブのリトライが実行され、リトライ処理で成功していた
  - 結果として幸いにもユーザ影響はなかった

誰にでも失敗はあるから気にするな！
いい経験だから後で原因調査するといいよ！

ためになるアドバイスを残し、先輩は退職

頼りになる先輩

誰にでも失敗はあるから気にするな！
いい経験だから徹底的に原因調査するといいよ！

数日後

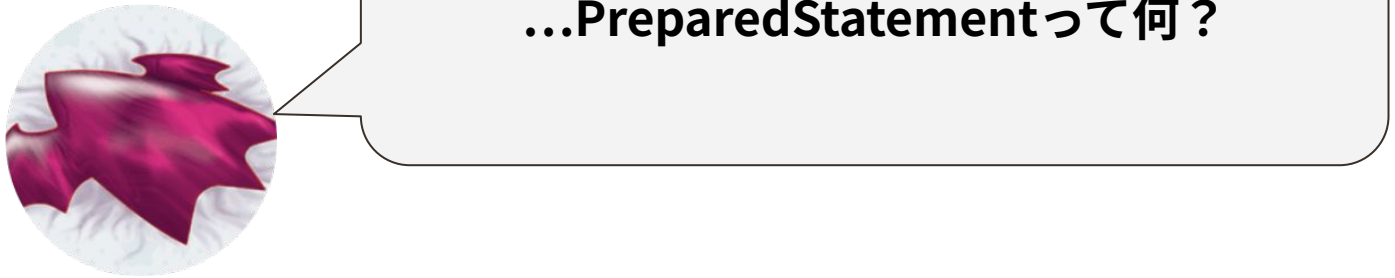
ためになるアドバイスを残し、先輩は退職

原因解明へ

この前デプロイで起こったあの問題
ちゃんと調べてみるか🤔

発生した例外クラス：**ActiveRecord::PreparedStatementCacheExpired**

メッセージ：**ERROR: cached plan must not change result type**



PreparedStatementを理解するにあたって以下の用語を軽くおさらい

- プレースホルダ
- バインド

周辺知識の整理

Railsコンソールで `User.find(1)` を実行すると以下のログが出る

```
naro(dev)> User.find(1)
  User Load (1.6ms)  SELECT "users".* FROM "users" WHERE "users"."id" = $1 LIMIT $2  [["id", 1], ["LIMIT", 1]]
=>
#<User:0x0000ffff771c91d0
 id: 1,
```

$1,$2という特殊な記号を用いて動的に値を割り当てているが、
この記号をプレースホルダと呼び、値の割り当てをバインドと呼ぶ

周辺知識の整理

- バインドのタイミングに関して2つの手法がある
  - 静的プレースホルダ
    - データベース側でバインドする
  - 動的プレースホルダ
    - アプリケーション側でバインドする

周辺知識の整理

- バインドのタイミングに関して2つの手法がある
  - 静的プレースホルダ
    - データベース側でバインドする
    - PreparedStatementに関係があるのはこっち
  - 動的プレースホルダ
    - アプリケーション側でバインドする

静的プレースホルダのバインドの流れ

1. アプリケーション側は以下の2つを送る
   a. プレースホルダが入ったSQL文

```
SELECT * from users where id = $1 AND LIMIT $2
```

   b. 実際のパラメータの値

```
$1 = 1, $2 = 1
```

2. データベース側はバインドを行い、SQL文を実行する

```
SELECT * from users where id = 1 AND LIMIT 1
```

静的プレースホルダのバインドの流れ

1. アプリケーション側は以下の2つを送る
   a. プレースホルダが入ったSQL文

```
SELECT * from users where id = $1 AND LIMIT $2
```

   SQL文の構造が確定しているのでSQL実行前に構文解析ができる

   ⇒構文解析済みのSQL文をPreparedStatementとして、コネクションが切れるまで保持する

   b. 実際のパラメータの値

2. データベース側はバインドを行い、SQL文を実行する

PreparedStatementについて

- PREPARE文で作成し、EXECUTE文で実行できる

```
naro_dev=# PREPARE stmt(bigint, bigint ) AS
             SELECT name FROM users where id = $1 LIMIT $2;
PREPARE
```

PREPARE文の実行結果
（返り値はPREPARE）

```
naro_dev=# EXECUTE stmt(1,1);
   name
-----------
 izumitomo
(1 row)
```

EXECUTE文の実行結果
（SELECT name FROM users where id = 1 LIMIT 1 が実行される）

検証環境でPreparedStatementの
中身を確認してみるか

- 『クラウドハウス』シリーズはDBにPostgreSQLを採用
  - PostgreSQLにはPrepatedStatementを確認できるpg_prepared_statementsというシステムビューがある
- 検証環境のコンテナの中に入ってRailsコンソールから確認してみる

```
# SQLを実行
irb(main):001:0> Website.find(00000000-0000-0000-0000-000000000000)
=>
#<Website:0x00007f01f5239df0
id: "00000000-0000-0000-0000-000000000000",
company_id: "11111111-1111-1111-1111-111111111111",
name: "テストサイト",
status: "inspect">

# PreparedStatementが作られたことを確認する
irb(main):002:0> ActiveRecord::Base.connection.execute('select * from pg_prepared_statements;').count
=> 1

# 中身を見てみる
irb(main):003:0> ActiveRecord::Base.connection.execute('select * from pg_prepared_statements;')[0]
=>
{"name"=>"a1",
"statement"=>"SELECT \"websites\".* FROM \"websites\" WHERE \"websites\".\"id\" = $1 LIMIT $2",
"prepare_time"=>2024-10-08 03:56:42.93202 +0000,
"parameter_types"=>"{uuid,bigint}",
"result_types"=>"{uuid,uuid,\"character varying\",website_status_enum}",
"from_sql"=>false,
"generic_plans"=>0,
"custom_plans"=>1}
```

Railsコンソールからpg_prepared_statementsを覗いてみた結果

見覚えのある単語

```
# 中身を見てみる
irb(main):003:0> ActiveRecord::Base.connection.execute('select * from pg_prepared_statements;')[0]
=>
{"name"=>"a1",
"statement"=>"SELECT \"websites\".* FROM \"websites\" WHERE \"websites\".\"id\" = $1 LIMIT $2",
"prepare_time"=>2024-10-08 03:56:42.93202 +0000,
"parameter_types"=>"{uuid,bigint}",
"result_types"=>"{uuid,uuid,\"character varying\",website_status_enum}",
"from_sql"=>false,
"generic_plans"=>0,
"custom_plans"=>1}
```

発生した例外クラス：**ActiveRecord::PreparedStatementCacheExpired**

メッセージ：**ERROR: cached plan must not change result type**

result_typeはstatementをEXECUTEした時の結果の型が入る

```
"statement"=>"SELECT \"websites\".* FROM \"websites\" WHERE \"websites\".\"id\" = $1 LIMIT $2",
"result_types"=>"{uuid,uuid,\"character varying\",website_status_enum}",
```

```
--
-- Name: websites; Type: TABLE; Schema: public; Owner: -
--

CREATE TABLE public.websites (
    id uuid DEFAULT gen_random_uuid() NOT NULL,
    company_id uuid NOT NULL,
    name character varying(50) NOT NULL,
    status public.website_status_enum NOT NULL
);
```

websitesテーブルのスキーマ（自動生成）

なるほど…そういうことか😮

再掲：静的プレースホルダのバインドの流れ

1. アプリケーション側は以下の2つを送る
   a. プレースホルダが入ったSQL文

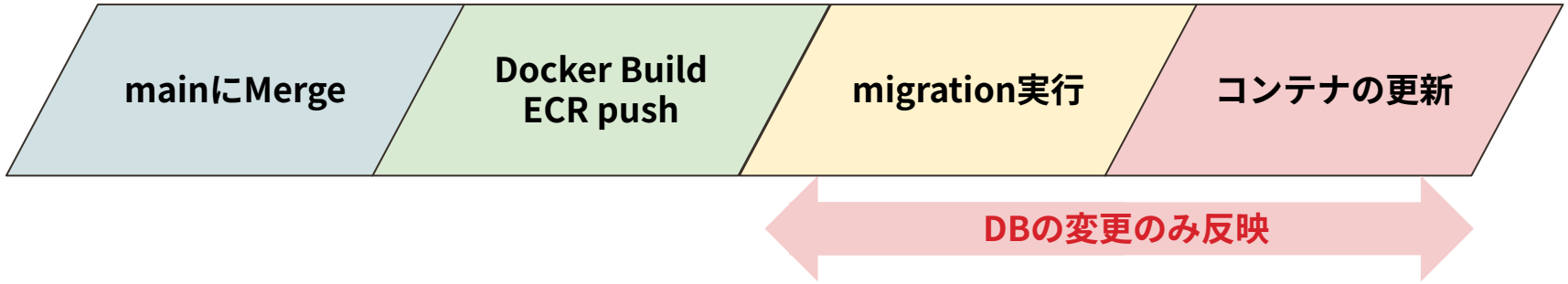
```
SELECT * from users where id = $1 AND LIMIT $2
```

   ⇒構文解析済みのSQL文をPreparedStatementとして、コネクションが切れるまで保持する

   b. 実際のパラメータの値

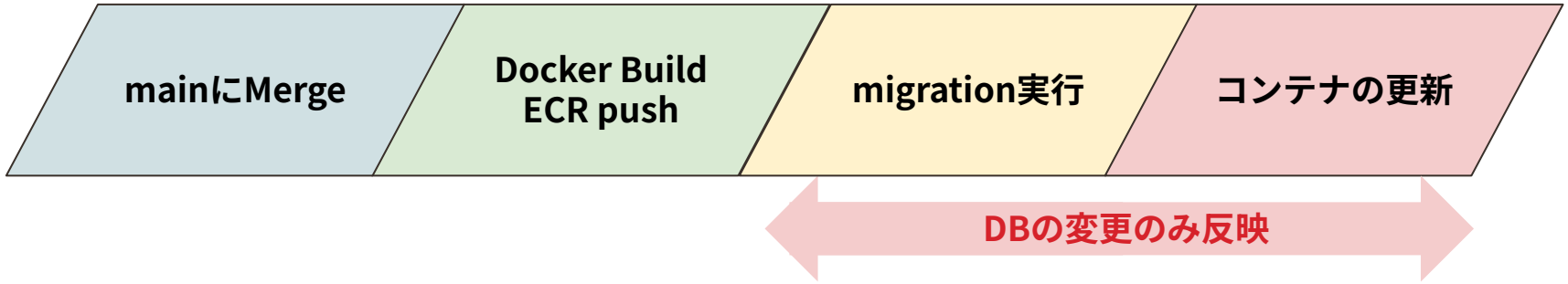
2. データベース側はバインドを行い、SQL文を実行する

再掲：整理したデプロイの流れ



migrationの実行からコンテナの更新で古いコンテナが停止するまでの間、
不整合が生まれる

再掲：整理したデプロイの流れ



migrationの実行からコンテナの更新で古いコンテナが停止するまでの間、
不整合が生まれる

⇒古いコンテナのDBコネクションには、migration実行前に作られた
古いPreparedStatementが残っている

古いPreparedStatement

テーブルにカラムが追加され、PreparedStatementが古くなると

- 構文解析時に確定したresult_types
- 実際にstatementを実行して返ってきた結果の型

が異なる状態に陥ってしまう

古いPreparedStatement

テーブルにカラムが追加され、PreparedStatementが古くなると

- 構文解析時に確定したresult_types
- 実際にstatementを実行して返ってきた結果の型

が異なる状態に陥ってしまう

発生した例外クラス：ActiveRecord::PreparedStatementCacheExpired

メッセージ：ERROR: cached plan must not change result type

ざっくり何が起こったか検討が付いたし
ソースコードを読みに行くか🧐

Railsの中を読む

- 発生した例外を元に調べるとすぐに該当のコードが見つかった

```ruby
# Nothing we can do if we are in a transaction because all commands
# will raise InFailedSQLTransaction
if in_transaction?
  raise ActiveRecord::PreparedStatementCacheExpired.new(e.cause.message, connection_pool: @pool)
else
  @lock.synchronize do
    # outside of transactions we can simply flush this query and retry
    @statements.delete sql_key(sql)
  end
  retry
end
```

activerecord/lib/active_record/connection_adapters/postgresql_adapter.rb（7.2-stable）

- トランザクションの中のみraiseするようになっている
  - トランザクション外だとstatementを削除してリトライする
  - トランザクション内ではPostgresql仕様上、リトライで解消できない

ローカルで再現

- ソースコードをもとに、ローカルで簡単に例外を再現できた
  - 2つのターミナルを準備してそれぞれ別々に動かす
    - RailsコンソールからSQLを実行するターミナル
    - カラム追加のmigrationを実行するターミナル

対応策

Rails7以降、この状況を回避するために

`config.active_record.enumerate_columns_in_select_statements`

というオプションが追加されている

対応策

これをTRUEにすると、ワイルドカードクエリを回避する

```
naro(dev)> User.first
  User Load (0.5ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" ASC
LIMIT $1  [["LIMIT", 1]]
=> #<User:0x0000ffffab5af130 id: 1, name: "izumitomo">
```

config.active_record.enumerate_columns_in_select_statements = FALSE

```
naro(dev)> User.first
  User Load (0.5ms)  SELECT "users"."id", "users"."name" FROM "users" ORDER BY
 "users"."id" ASC LIMIT $1  [["LIMIT", 1]]
=> #<User:0x0000ffffb71feb60 id: 1, name: "izumitomo">
```

config.active_record.enumerate_columns_in_select_statements = TRUE

カラム追加を行ってもresult_typesの不一致が発生しない

今回の事象について理解できたが、

スキーマを変更するデプロイは今までも行われていたはず

⇒このやらかしは初めて起こったのか？

今回の事象について理解できたが、

スキーマを変更するデプロイは今までも行われていたはず

調べた結果

⇒このやらかしは自分が本当に初めてなのか？

過去に起きてたけど

無視されてた…

誰にでも失敗はあるから気にするな！
いい経験だから後で原因調査するといいよ！

先輩の優しさに気付く

誰にでも失敗はあるから気にするな！
いい経験だから後で原因調査するといいよ！

ためになるタスクを残してくれた
先輩の優しさに感謝🥰

余談：MySQLについて

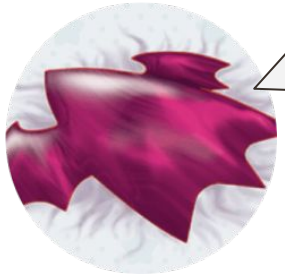
- RailsのPreparedStatementのデフォルトの利用設定について、MySQLは利用しない設定になっている
- MySQLでもRails7.2以降からデフォルトで利用する動きがあった
  - しかし[バグ報告]により現在もデフォルトは利用しない設定となっている

そして気付く

誰にでも失敗はあるからな！
時間があったら調査するといいよ！

時は流れて

以前から起こってたのでは ……？



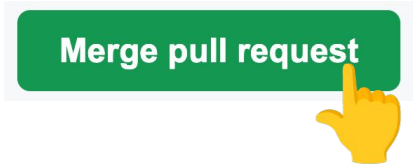
デプロイ慣れた😘

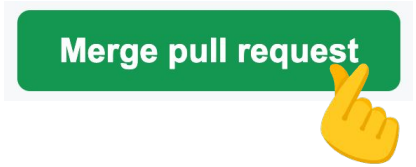
若かりし頃

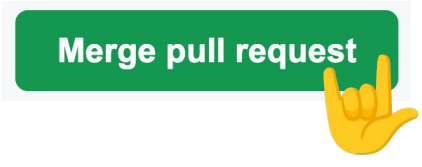
一つ一つ丁寧にデプロイしていたあの頃も今は昔

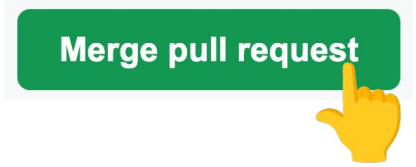
- DBとアプリケーション間の不整合は起きるか？
- 仮に起きる場合、どのような問題が発生する？
- その場合の対処は何が考えられる？
- …

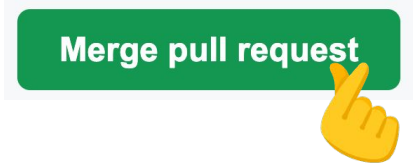
流れるようにMergeボタンを押していく日々

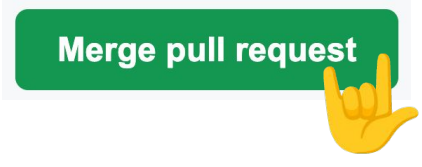












現在

流れるようにMergeボタンを押していく日々

Merge pull request

Merge pull request

Merge pull request

ある日

Merge pull request

Merge pull request

Merge pull request

「エクスポートが実行中のまま終わらない」
と先ほど顧客から連絡が来ました…

再度エクスポートを行うと
今度は成功したとのことです

原因わかりますか？

うーん…ちょっと調べてみますね

（エクスポート機能は初期に突貫で作られたらしいので、多分バグがありそう…🤔）

（やれやれ…先人のバグを直してやるか😘）

現状把握

- エクスポートは非同期処理で行われる
  - 非同期処理はSidekiqで行っている
  - エクスポートのジョブの状態や開始・完了時刻はDBに保存

現状把握

- 調べてみると確かに顧客のエクスポートのジョブのステータスが「実行中」のまま数時間が経過していた
  - 本来であればエクスポートは長くても数分で終わる
  - 既にそのジョブはSidekiqのWorker内に存在しない
    - ジョブの実行状況を監視するSidekiq Web UIというツールがある

現状把握

- ログをもとにエクスポート処理のコードを読みながら原因を探っていく
  - しかし原因がわからない…
    - ローカルで色々動かしてみても特に何も得られず
    - 異常終了しているのに例外を検知できていないのが謎

ふと、異常終了したジョブを眺めていると

ある事実に気づいた

そういや、このジョブの開始直後に
俺デプロイしたな…😶

（やれやれ…俺が先人のバグを直してやるか）

もしかしてこれ…
デプロイした俺のせい？🫣

検証環境で試してみるか

検証環境でエクスポートの非同期処理の実行中にデプロイを走らせてみる…

検証環境でエクスポートの非同期処理の実行中にデプロイを走らせてみる…

⇒「実行中」のまま止まった

状況の再現に成功（嬉しいけど嬉しくはない）

検証環境でエクスポートの非同期処理の実行中にデプロイを走らせてみる…

⇒「実行中」のまま止まった

状況の再現に成功（嬉しいけど嬉しくはない）

この瞬間、エンジニアとしての直感が働いた

一応…他の非同期処理も見とくか🤔

インポートの処理中にデプロイしてみる

インポートの処理中にデプロイしてみる

⇒「実行中」のまま止まる

インポートの処理中にデプロイしてみる

⇒「実行中」のまま止まる

メール送信の処理中にデプロイしてみる

インポートの処理中にデプロイしてみる

⇒「実行中」のまま止まる

メール送信の処理中にデプロイしてみる

⇒「実行中」のまま止まる

インポートの処理中にデプロイしてみる

⇒「実行中」のまま止まる

メール送信の処理中にデプロイしてみる

⇒「実行中」のまま止まる

SMS送信の処理中にデプロイしてみる

インポートの処理中にデプロイしてみる

⇒「実行中」のまま止まる

メール送信の処理中にデプロイしてみる

⇒「実行中」のまま止まる

SMS送信の処理中にデプロイしてみる

⇒「実行中」のまま止まる

気付いた

デプロイのたびに

ジョブ消し飛んでる🚀🚀

気付いた

調査開始

- Sidekiqのドキュメントとソースコードを読み、実行中のジョブがあるときにどのようにSidekiqが停止するかを理解
  - SIGTERMシグナルが重要な役割を果たしている

ここで一旦、シグナルのおさらい

おさらい：シグナル

- シグナルとはソフトウェア割り込みの一種
  - 特定のイベント発生時にプロセスに通知するための仕組み
  - Ctrl-CやCtrl-Zで常日頃お世話になっているアレ
- SIGTERMシグナルはプロセスの安全な終了のために使われる
  - SIGTERM受信時の動作はプログラムに委ねられている
    - 例えばプロセスの強制終了を伝えるSIGKILLを受信した場合、制御はOSに委ねられるためプログラムは関与できない

おさらい：シグナル

sleep中のプロセスにSIGTERMを送ってみる

```
yu tools$ () echo "このシェルのPID: $$ "
このシェルのPID: 98814
yu tools$ () sleep 1000
```

おさらい：シグナル

sleepしているプロセスのID（PID）をpstreeで特定してみる

```
yu tools$ () pstree -p 98814
-+= 00001 root /sbin/launchd
 \-+= 00655 yu /Applications/Visual Studio C
   \-+- 01508 yu /Applications/Visual Studio
     \-+= 98786 yu zsh (qterm)
       \-+= 98814 yu /bin/zsh --login
         \--= 25988 yu sleep 1000
```

PID25988のプロセスがsleepを実行しているので、このプロセスに対してSIGTERMを送ればよい

おさらい：シグナル

別のターミナルから

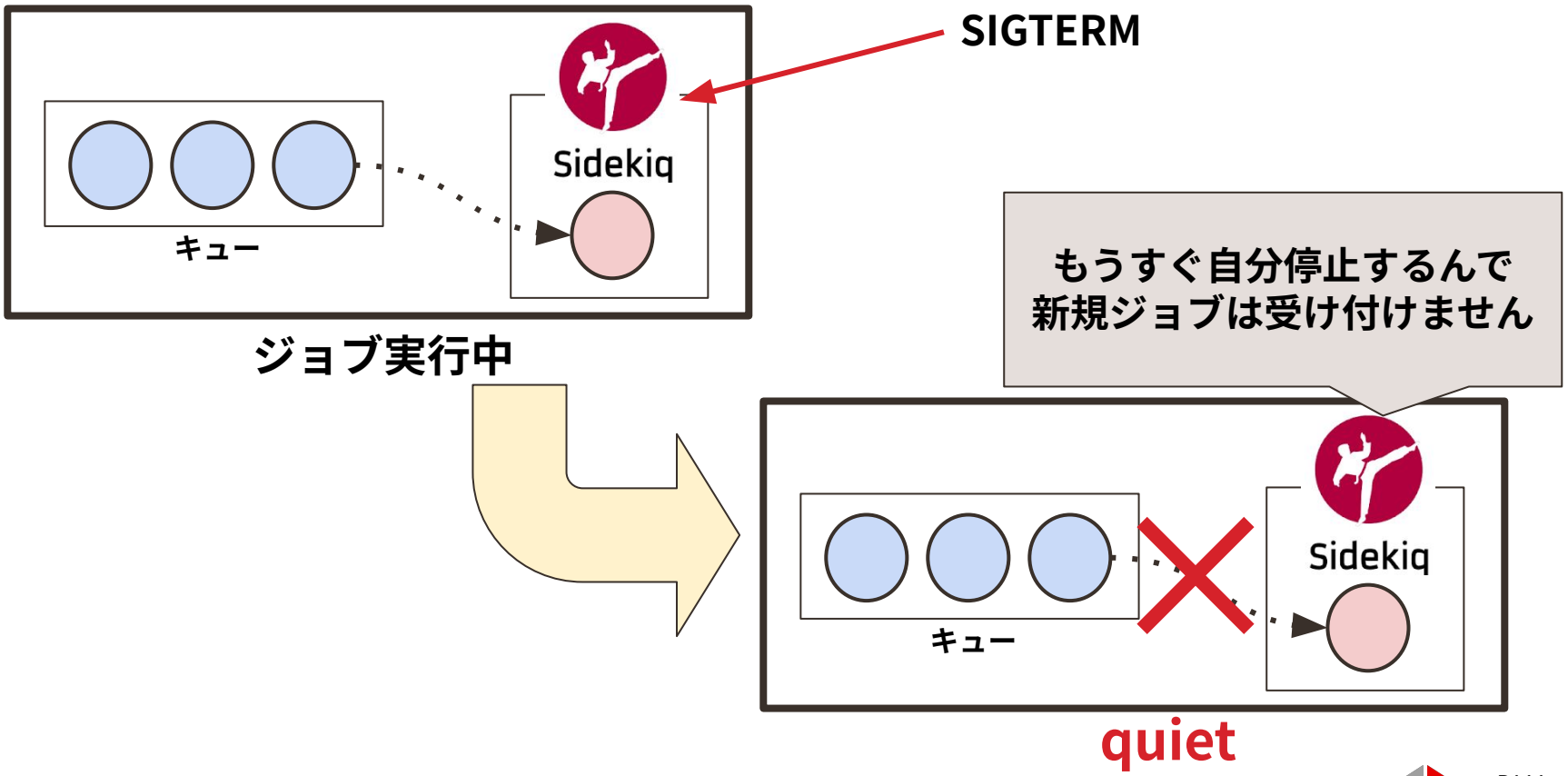
```
kill -SIGTERM 25988
```

を実行してSIGTERMを送ると、sleepは以下のように終了する

```
yu tools$ () sleep 1000
zsh: terminated  sleep 1000
yu tools$ ()
```

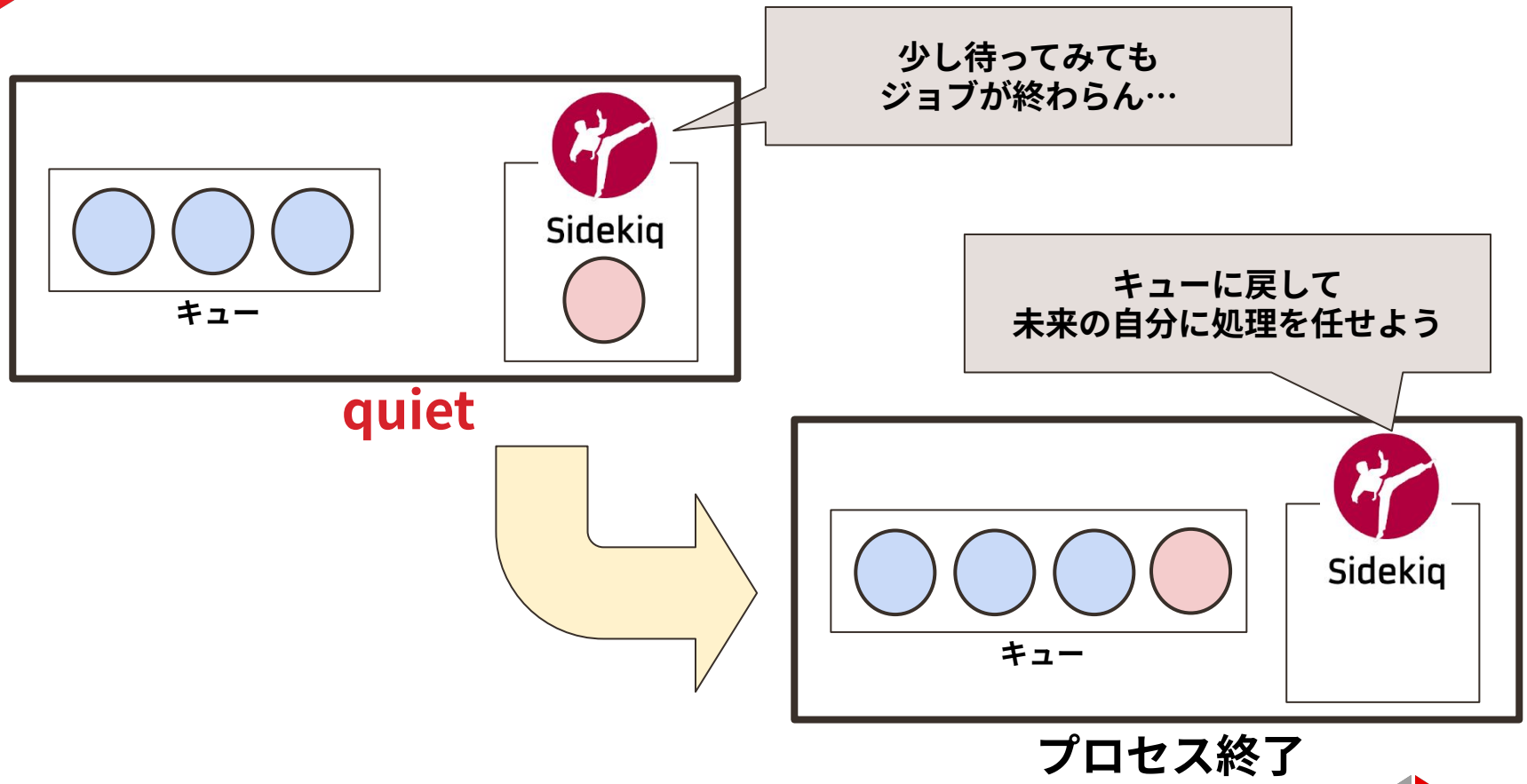
Sidekiqの仕様

- SidekiqはSIGTERMを受け取ると、現在実行中のジョブの終了を待つquietという状態に移行する
  - quietの時、新しいジョブの実行は受け付けない
  - 実行中のジョブの終了を待つ猶予時間は調整可能



Sidekiqの仕様

- 猶予時間が経っても実行中のジョブが存在する場合、そのジョブをキューに押し戻す
  - 新たにSidekiqコンテナが立ち上がると、キューに戻されたそのジョブを再び実行する



現状、明らかに仕様と
異なる挙動をしている…

まずはローカル環境で
ジョブの消失を観察するか🧐

Docker環境でSidekiqコンテナにSIGTERMを送ってみる…

Docker環境でSidekiqコンテナにSIGTERMを送ってみる…

⇒ジョブがキューに押し戻された

状況の再現に失敗（嬉しくないけど嬉しい）

Docker環境でSidekiqコンテナにSIGTERMを送ってみる…

⇒ジョブがキューに押し戻された

状況の再現に失敗（嬉しくないけど嬉しい）

ローカルで発生しないということは……？

ECS周りの設定が怪しい…！🤩

（ECS全然わからん😆）

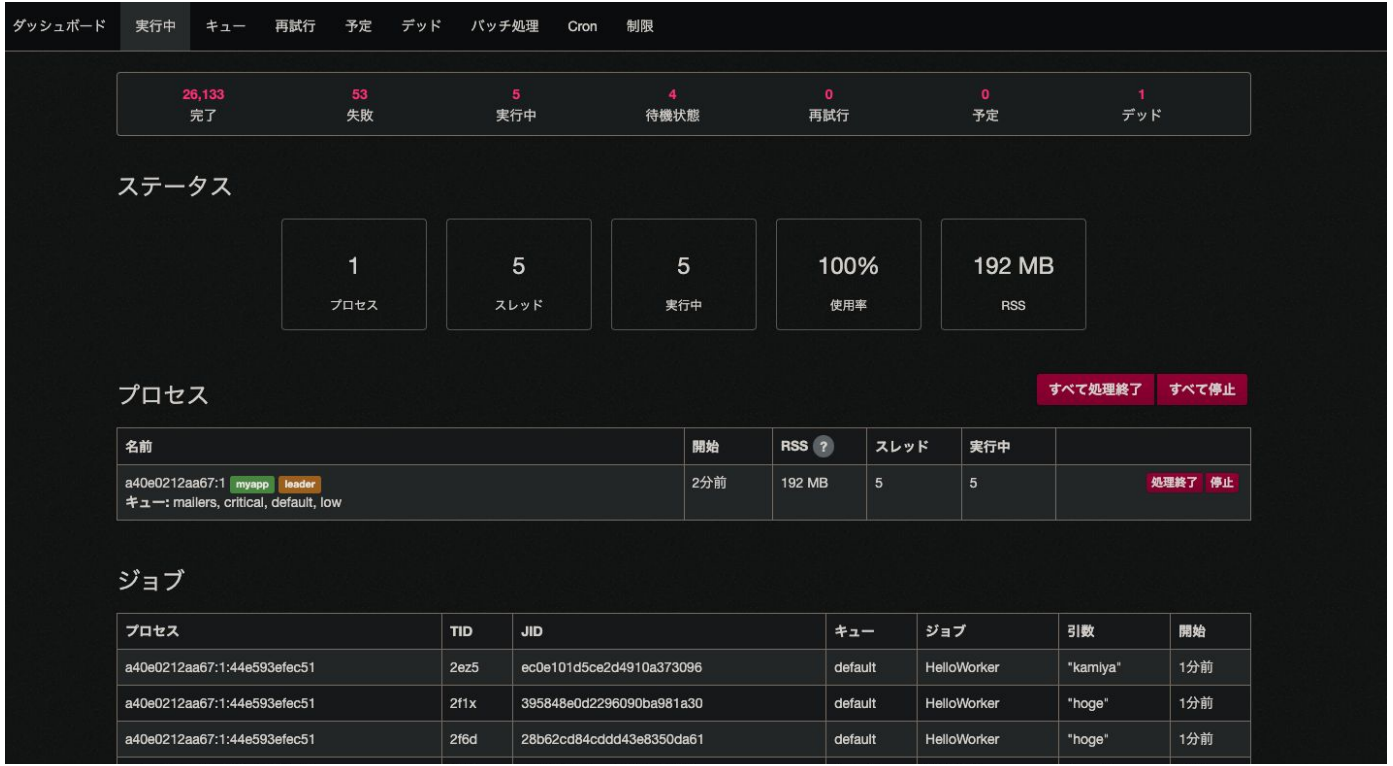
- ECSのドキュメントを読んでコンテナライフサイクルを把握
  - 停止時にコンテナのエントリプロセスはSIGTERMを受け取る
  - SIGTERM受信から一定の猶予時間の後にSIGKILLが飛んでくる
    - この猶予時間は調整可能

- つまりアプリケーションはSIGTERMを受け取ってからSIGKILLが来るまでに処理を正常に終了させるように必要がある
  - SidekiqではSIGTERMを受け取るとquietに移行して所定の時間だけ処理が完了するのを待つ

- つまりアプリケーションはSIGTERMを受け取ってからSIGKILLが来るまでに処理を正常に終了させるように必要がある
  - SidekiqではSIGTERMを受け取るとquietに移行して所定の時間だけ処理が完了するのを待つ
  - この待っている間にSIGKILLが飛んできたらジョブが消える！
    - SidekiqとECSの設定を見直す

- しかし設定は正しかった
  - Sidekiqは25秒待つ（デフォルト）
  - ECSの方は120秒待つ

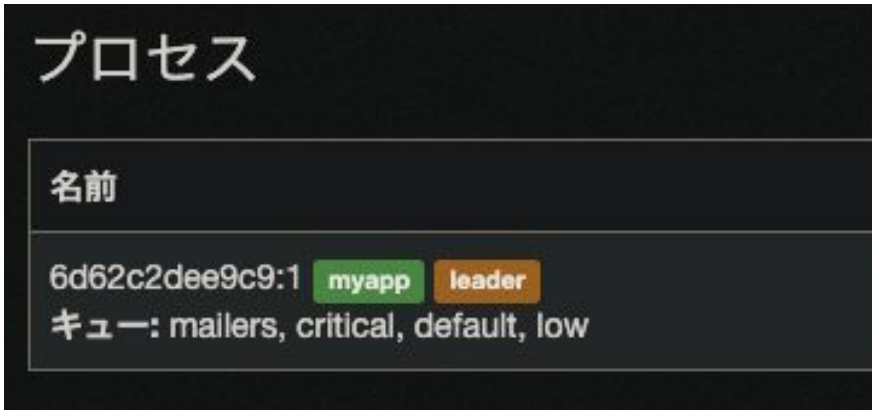
あてが外れたので、次はローカルと検証環境のSidekiq Web UIをじっくり見比べていくことに



Sidekiq Web UIの画面

ローカル

以下の稼働中のSidekiqコンテナに対し、SIGTERMを送ってみる



稼働中のSidekiqプロセス

ローカル

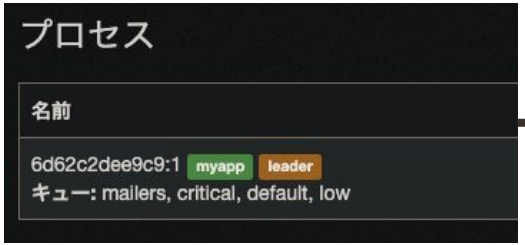
SIGTERMを送るとquietに移行した

初期状態

SIGTERMを送った後

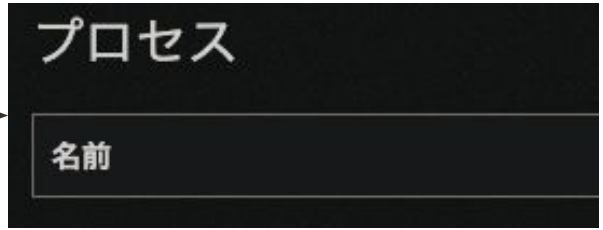
ローカル

そして猶予時間が経過すると停止した



初期状態

SIGTERMを送った後



時間経過後

検証環境

検証環境で同様にコンテナを停止させてみる

初期状態

検証環境

SidekiqコンテナがSIGTERMを受け取るはずの状態になっても

なぜかquietにならない…

初期状態

SIGTERMを受け取るはずの状態

検証環境

そしてそのまま消えた

初期状態

SIGTERMを受け取るはずの状態

ECSタスクが停止

SidekiqにSIGTERM到達してない🫣

- SIGTERMを受け取るエントリプロセスがどこになっているのかを調べる
  - ECSのタスク定義に記載されているcommandフィールドがエントリプロセスだと判明
    - DockerfileとECSタスク定義の書き方次第で変わる

```
"command": [
  "/myapp/startup_sidekiq.sh"
],
```

commandフィールド

```
#!/bin/bash -x

bundle exec sidekiq
```

startup_sidekiq.shの中身

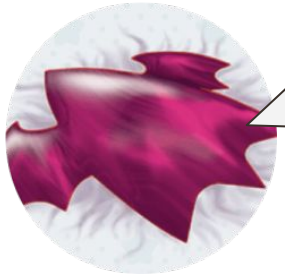
```
"command": [
  "/myapp/startup_sidekiq.sh"
],
```

commandフィールド

```
#!/bin/bash -x

bundle exec sidekiq
```

startup_sidekiq.shの中身



ぱっと見、正しそうだが…🤔

- プロセスツリーを書き出してみる（PIDは適当）

```
  PID        COMMAND
  14      bash /myapp/startup_sidekiq.sh
  25      \_ sidekiq 7.1.6  [0 of 5 busy] leader
```

- プロセスツリーを書き出してみる（PIDは適当）

- PID14がエントリプロセスとなる

- プロセスツリーを書き出してみる（PIDは適当）

```
  PID        COMMAND
  14      bash /myapp/startup_sidekiq.sh
  25      \_ sidekiq 7.1.6  [0 of 5 busy] leader
```

bashにSIGTERM送ってるな🫣

対応

- Sidekiqをエントリプロセスとして扱えばよいのでexecを使う

```
#!/bin/bash -x

bundle exec sidekiq
```



```
#!/bin/bash -x

exec bundle exec sidekiq
```

- execは今のシェルプロセスを指定したプログラムに置き換える
  - 詳細は割愛するが、以下のように置き換わる

```
PID      COMMAND
14     bash /myapp/startup_sidekiq.sh
25     \_ sidekiq 7.1.6 [0 of 5 busy] leader
```

```
PID      COMMAND
14     sidekiq 7.1.6 [0 of 5 busy] leader
```

execを付けた状態で非同期処理の実行中にデプロイしてみる…

execを付けた状態で非同期処理の実行中にデプロイしてみる…

⇒ジョブがキューに押し戻された！

execを付けた状態で非同期処理の実行中にデプロイしてみる…

⇒ジョブがキューに押し戻された！

無事解決できたが、再びエンジニアの直感が働く

インフラ周りの設定って
使い回されがちだよな…🤔

他のECSタスクも一応
確認しておくか

その結果

- 全部同じミスしてた…
  - SidekiqだけでなくPumaやgRPCなど、プロダクト内の全ECSコンテナが同じやらかしをしていた
- というわけであわせて一気に修正

- 今度こそ一件落着だが、三度エンジニアの直感が働く
  - というか途中で流石に気づいたけど、見ないふりしてた

インフラ周りの設定って
使い回されがちだよな（2回目）

他の事業部…大丈夫だよな？

その結果

驚愕の事実

全社で同じミスしてると気付く

（bashにシグナル送ってた）

その結果

全社で同じシステムを使っていると気付く

（bashにシグナル送ってた）

「もしかして…俺が浅いだけか？」

と急に不安になる新卒1年目

その結果

「もしかして、この仕事向いてない？」

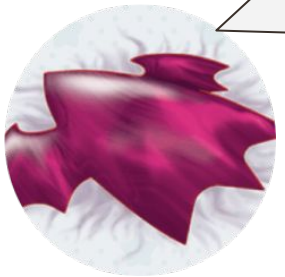
と急に不安になる新卒1年目

bashにシグナルを送る

深遠な意味を考え始める

他の事業部のDevに聞いてみることに

あの…デプロイ時にSidekiqの実行中の
ジョブ、消えてたりしませんか…？

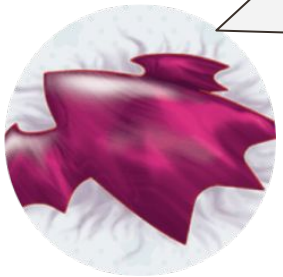


他事業部のDev

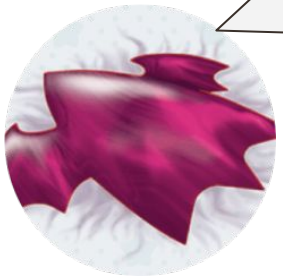
いや、特に問題ないよ？

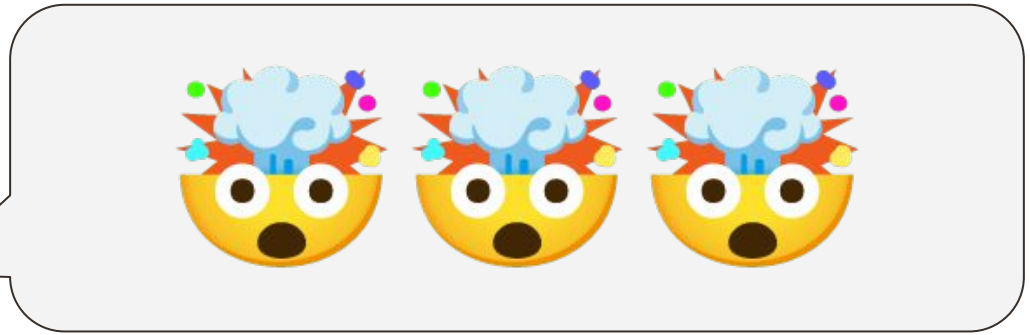
他事業部のDev

あれっ…そうですか
（やっぱ俺がなんか見落としてるのか…）



だって実行中のジョブがないか、
毎回チェックしてデプロイしてるからね



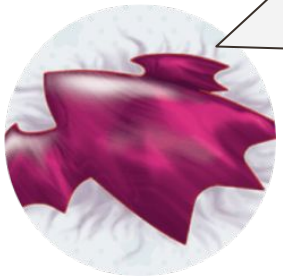


- 運用でたまたま回避できていた
  - Sidekiq Web UIを使えばプロセスをquietにできるので可能ではある
- 全事業部で修正を進めていくことになった

余談：技術責任者に報告

- ECSのタスク定義というインフラ周りの設定に手を入れたこともあり、今回の件を技術責任者のYさんに伝えておくことに
  - Yさんは社内の全プロダクトをインフラ面から手厚くサポート
  - 過去にYさんがECSタスクのオートスケールイン・スケールアウトでGraceful Shutdownを導入しようとしていた
    - もしかしたら今回の件が役に立つかもしれない

以前ECSでGraceful Shutdownを導入
しようとしてたと思うんですけど…

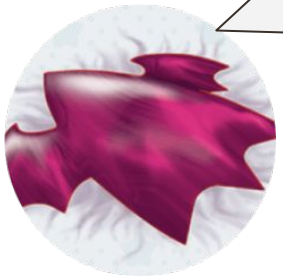


Yさん

そうなんだけどね、
上手く設定がハマってないんだよ…！

Yさん

あの…それもしかしたら



現状、原因としてはいくつか考えられて、
まずECSのstopTimeoutが短すぎるという可能性、
そしてそもそものシグナルハンドラにバグがある厄介な
パターン、別の可能性として間に挟まっているCDNのタ
イムアウトが考慮できてないのもありえて…
……だから……かもしれないし……

現状、原因としてはいくつか考えられて、
まずECSのstopTimeoutが短すぎるという可能性、
な
タ

そもそもSIGTERM届いてないかもです…

……

……

…まじで？

…まじで？

誰にでも見落としはあるので、臆せず意見することは大事

まとめ（初学者向け）

- デプロイ業務とやらかしを通じて多くの学びを得た
  - 普段の開発業務では関心すら持てていなかった技術領域に
    デプロイという自然な切り口から飛び込むことができた
  - 当時まだ駆け出しだった自分にとって大きな財産になった

- 今一度デプロイに目を向けてみてはいかが？

まとめ（初学者をフォローするミドル層向け）

- 若手の成長を促すために、時には身の丈以上の責任ある仕事を託す勇気と、心置きなくチャレンジできる環境づくりが大切
  - 全力のチャレンジにはやらかしがつきもの
    - でもそのやらかしは必ず彼らの糧になる
  - 転ばぬ先の杖というより、転んだ先の杖を用意しておく
    - 当時の自分にデプロイを託すというチームの決断の意味と、その環境を整えてくれていたことを後になって気付いた

謝辞

- デプロイを託してくれた先輩含む開発チーム
- 何度も技術的アドバイスをくれたYさん
- 全社のやらかしを晒す暴挙を許してくれた会社




皆様に感謝🥰

ご清聴ありがとうございました

デプロイを任されたので、

教わった通りにデプロイしたら障害になった件

～~~俺~~ 俺たちのやらかしを越えてゆけ～

Fin.