長年運用されているサービスの主要データ移行をサービス停止せず安全にリリースしました

2024年 12月

STORES 株式会社

目次

自己紹介

STORES 株式会社

テクノロジー部門 リテール GTM B グループ
エンジニアリングマネジャー

山下 隼人 (Hayato Yamashita)

@phayaceII

今日のプログラム

最初に質問です

50,000,000 件 +α

答えは？

STORES ネットショップ
注文データの件数

この注文データのデータ構造を移行する（書き込み先を切り替える）プロジェクトのお話です

今日のプログラムで持ち帰れること（結論）

1. アプリケーションでやるカナリアリリース
2. 新旧データ構造の差分チェックの有用性
3. テスト不足でも安全にリリースする方法

12年 サービス運用しています

データはたくさんあります
主要となる注文データの件数は先述

B2C 事業者の顧客データを中心とした
フロントオフィス プラットフォーム

提供しているサービスには、以下のものがあります

STORES ネットショップの略歴（レジを含む）

2012年8月
　　STORES ネットショップ、サービスローンチ

2020年5月
　　注文データの件数が 10,000,000 を超える

2021年6月
　　STORES レジ、サービスローンチ

2024年5月
　　注文データの件数が 50,000,000 を超える

STORES は長年運用しているサービスです

長年運用しているサービスのデータには、いくつかの特性があります

- とにかくデータ量が増えていくもの
- 書き込み回数の多いもの
- 読み込み回数の多いもの
- マスタデータとしてほとんど変化しないもの

などなど……

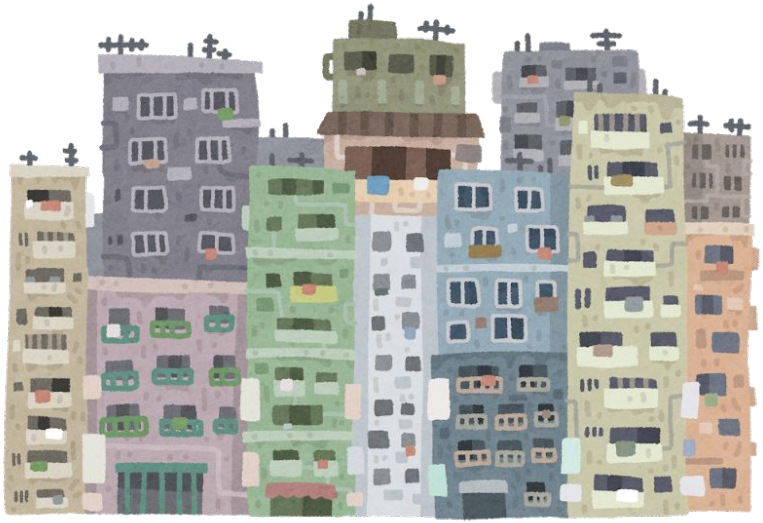
STORES の注文データは

注文データには、以下の特性があります

- とにかくデータ量が増えていくもの
- 書き込み回数の多いもの
- 読み込み回数の多いもの
- マスタデータとしてほとんど変化しないもの

当時の最善を尽くしたデータ構造設計
改修に次ぐ改修が重ねられ
だんだんと扱いづらく……

まるで九龍城砦のよう



そこで、我々はこうしました

そうだ、新しいデータ構造に移行しよう

移行プロジェクトでやりたいこと

データ構造の最適化

これまでの積み重ねられた改修を踏まえたい
STORES レジにも耐えられるデータ構造へ

レジもネットショップと同じ DB / テーブルで注文データを扱いたい

ネットショップからはじまった注文データの構造では、
より複雑な店頭販売の注文データの構造に耐えられない……

だから、データ構造を新しく作ろう

Not schema migrate
Generate new TABLE

考えられる選択肢

1. 古いデータを新しいデータ構造に移行する
2. 古いデータは移行せず、放置する
3. 古いデータは移行せず、全部削除する

考えられる選択肢から、どうやって選ぼう？

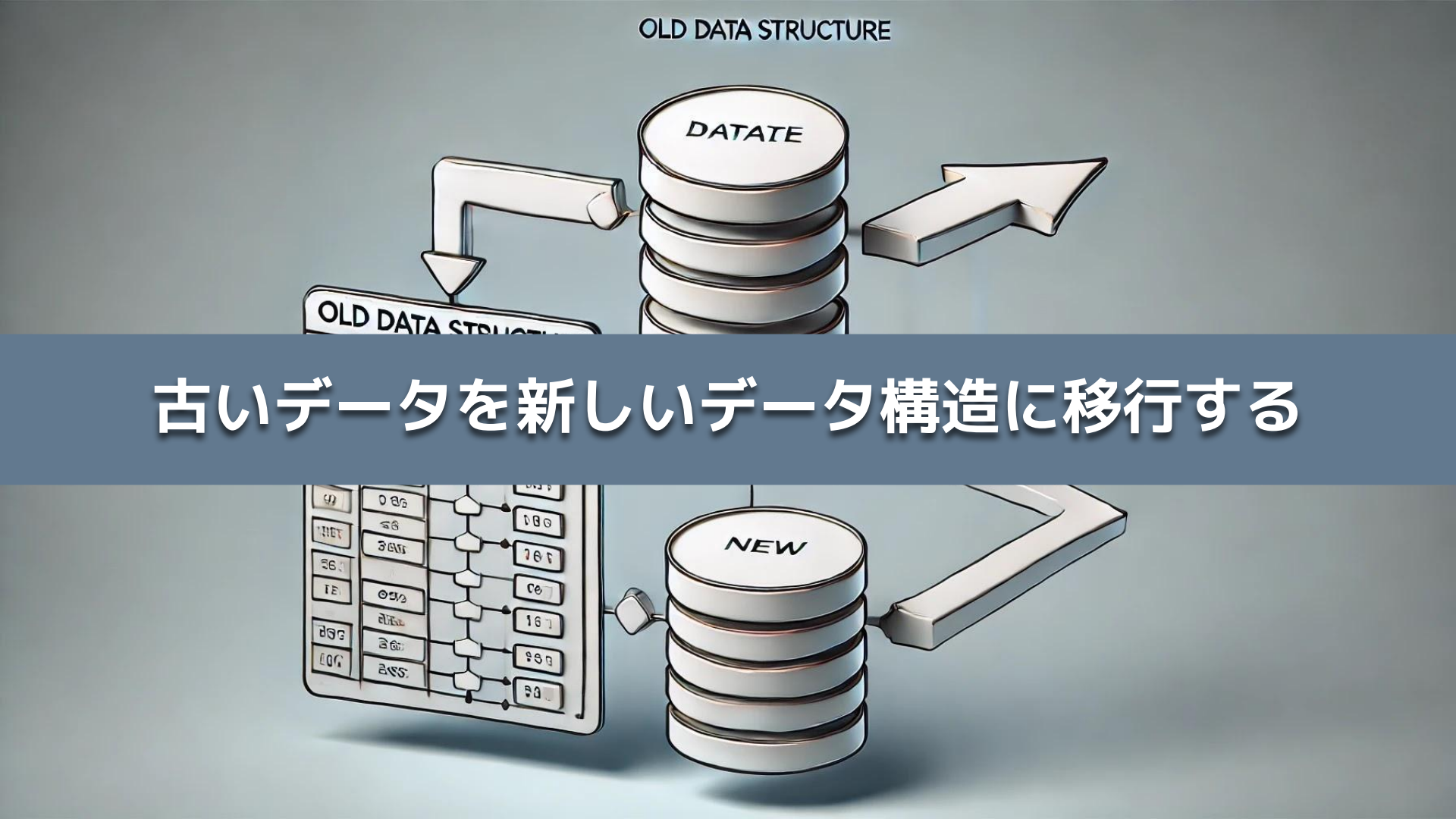
改修頻度は高い？ → YES

書き込み頻度は高い？ → YES

読み込み頻度は高い？ → YES

過去データも参照され続ける？ → YES

なので、我々はこうしました



古いデータを新しいデータ構造に移行する

移行のやり方

基本方針：ご安全に

動き続けているサービスを止めたくない
できるなら、サービス停止もしたくない

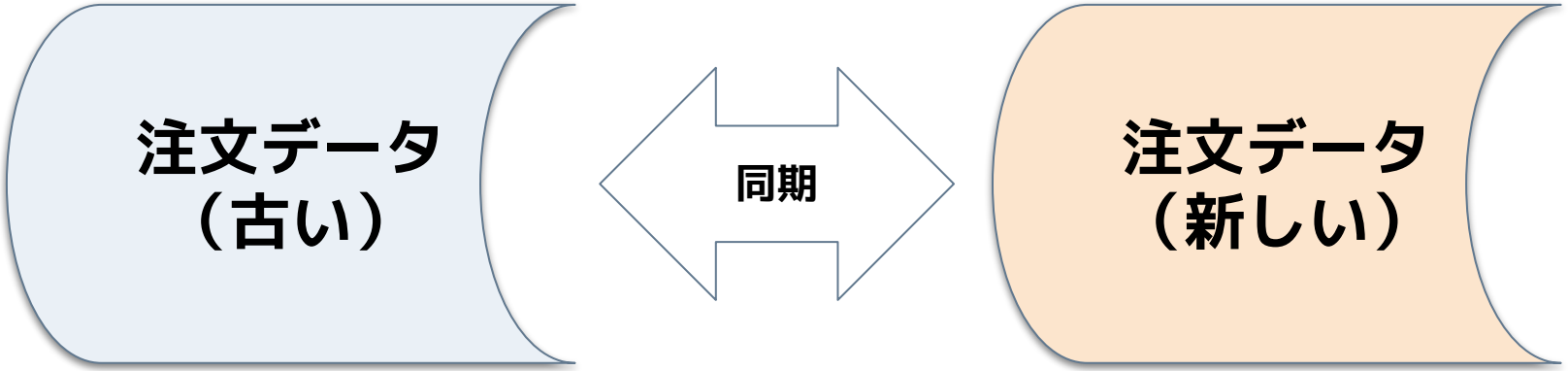
大量のデータを一気に移行することはせず、
少しずつ切り替えていく

安全なデータ移行の手順の、全体像

1. 新しいデータ構造を作る
2. 新しいデータ構造にも書き込む（二重書き込み）
3. 新しいデータ構造から書き込む（書き込み先の変更）
4. 新しいデータ構造を参照する（読み込み先の変更）
5. 古いデータ構造をやめる

安全なデータ移行の手順の、ここまではできている（前提）

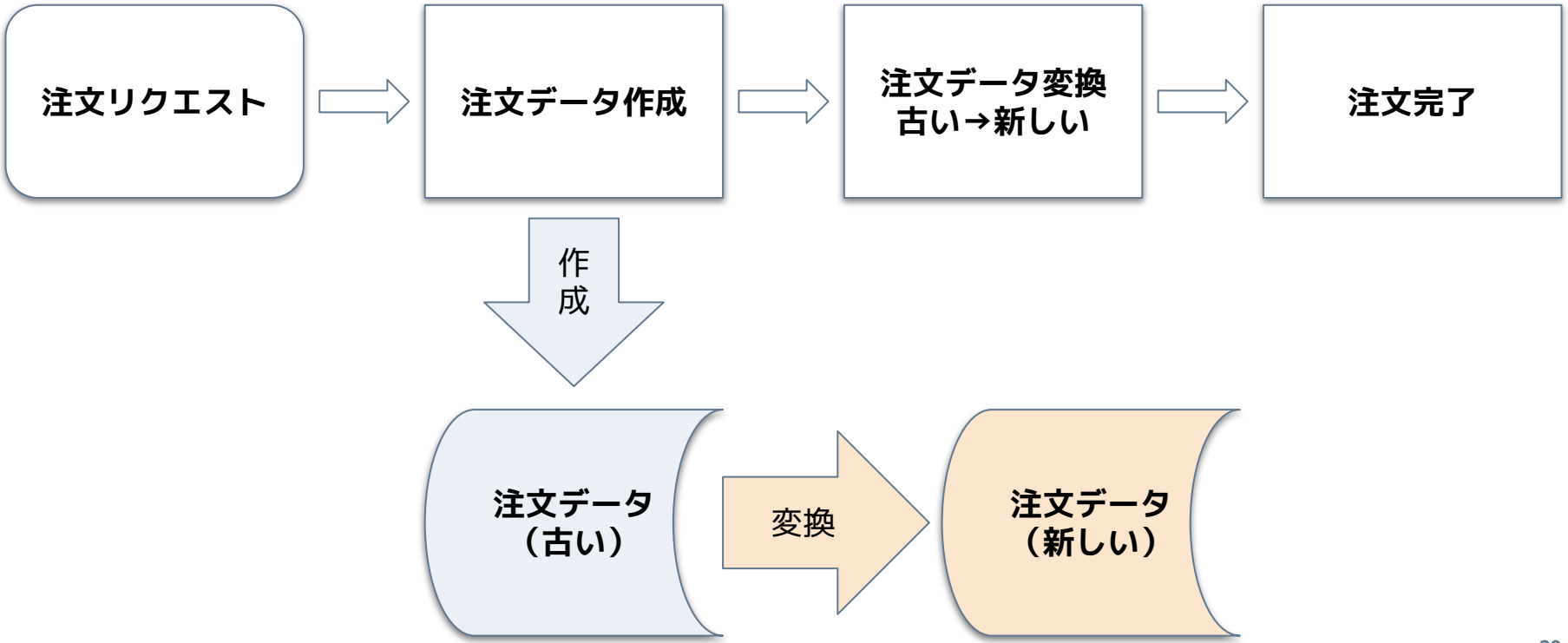
1. 新しいデータ構造を作る
2. 新しいデータ構造にも書き込む（二重書き込み）



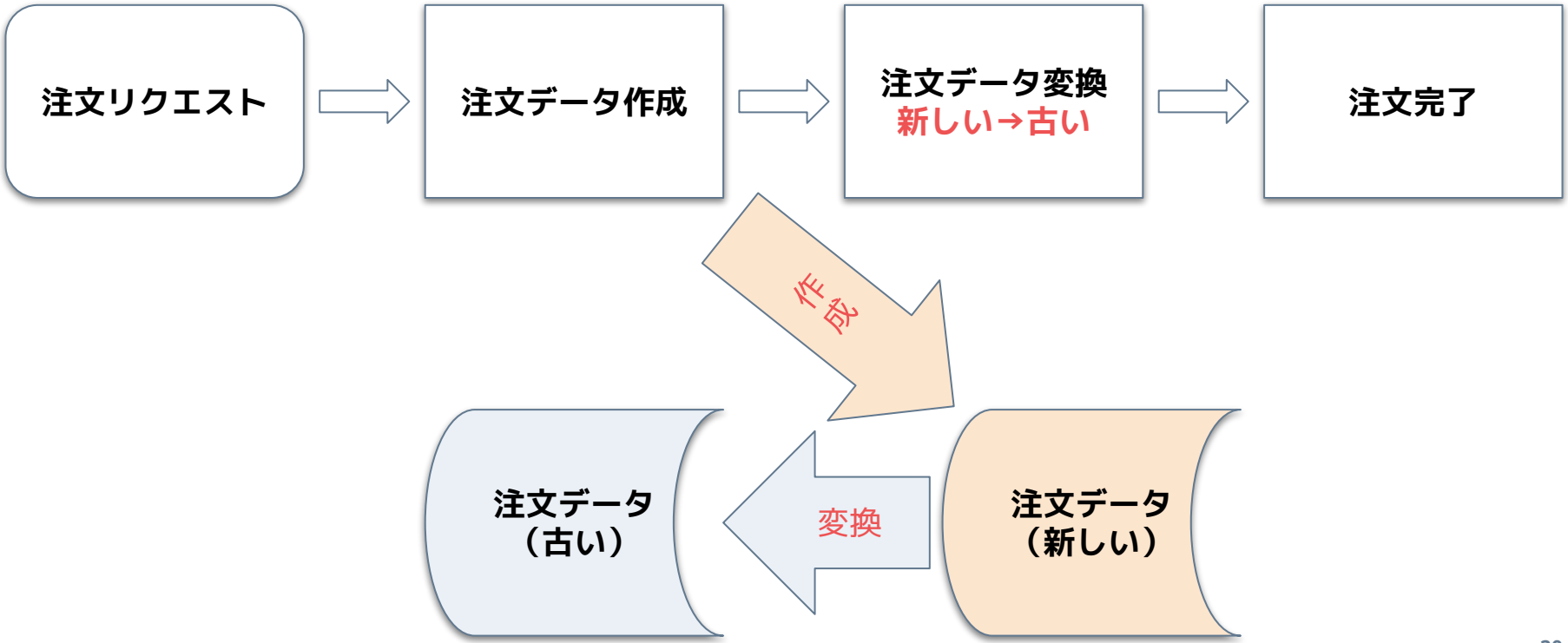
安全なデータ移行の手順の、ここをやる！

1. 新しいデータ構造を作る
2. 新しいデータ構造にも書き込む（二重書き込み）
3. **新しいデータ構造から書き込む（書き込み先の変更）**
4. 新しいデータ構造を参照する（読み込み先の変更）
5. 古いデータ構造をやめる

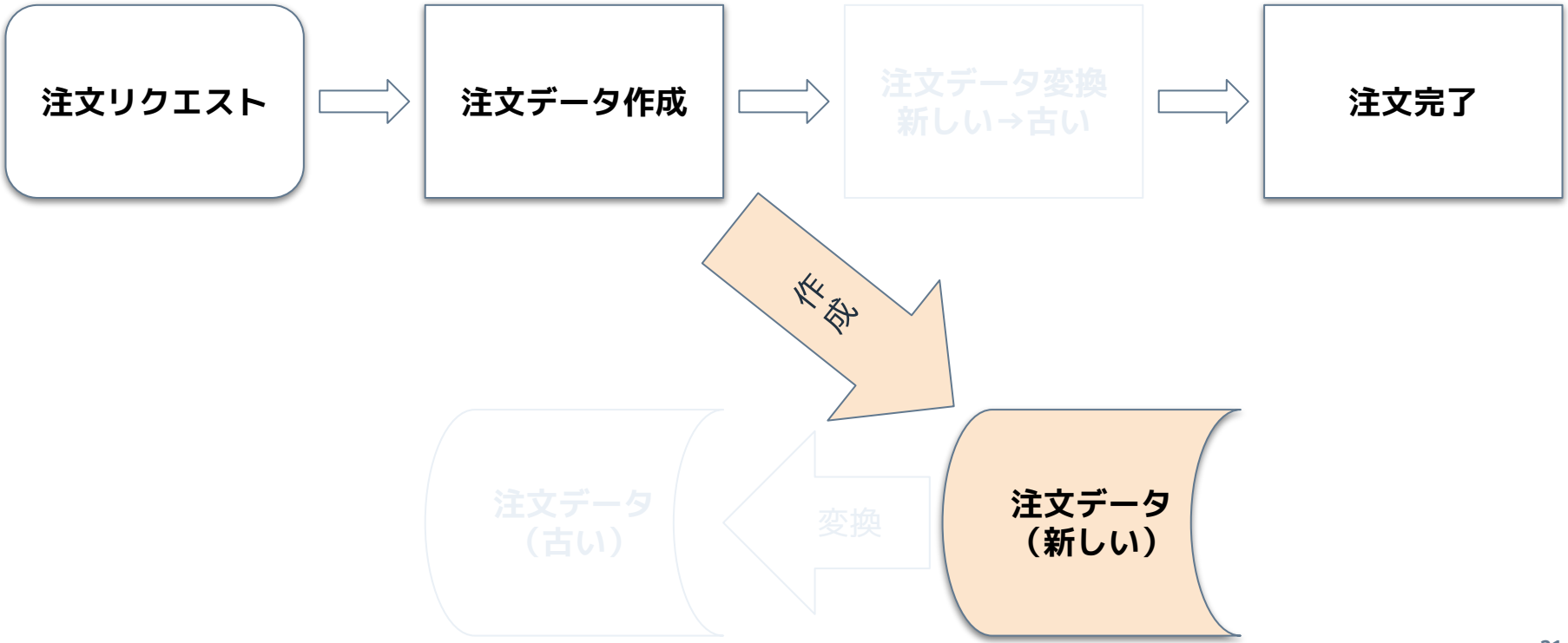
二重書き込みのイメージ



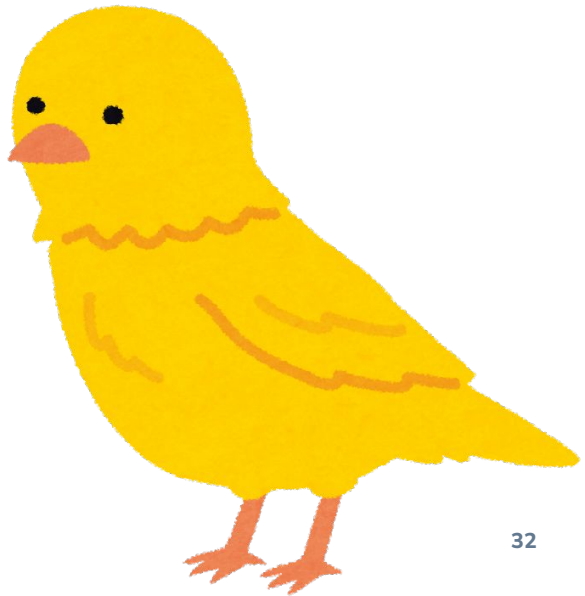
二重書き込みのイメージを、こうしたい！（書き込み先の変更）



二重書き込みのイメージを、最終的にはこうしたい！（古いデータ構造をやめる）



そして、少しずつ切り替えるため
カナリアリリースをしよう



カナリアリリースとは

*A **canary release** (or canary launch or canary deployment) allows developers to have features incrementally tested by a small set of developers.*
Feature flags like an alternate way to do canary launches and allow targeting by geographic locations or even user attributes.
If a feature's performance is not satisfactory, then it can be rolled back without any adverse effects.
It is named after the use of canaries to warn miners of toxic gases (Miner's canary).

ref. Feature toggle#Canary_release - Wikipedia

カナリアリリースの、メリットは？

少しずつ新しい作成処理（カナリア）
を動かすことができる

もしバグが見つかれば、
被害を最小限にするためリリースを中断

これまで動いていた元の処理（ベースライン）
に戻すことができる

ロードバランサーでやることが多い
カナリアリリース

今回の移行プロジェクトでは、
これをアプリケーションでやります

（厳密には段階的リリースかも）

ひとつのデータに対するリクエストが
複数のリクエストに分かれても

一貫してカナリア・ベースラインの
どちらで処理するか、振り分けができる

注文データ作成は、
ひとつのリクエストで完結しない

ユーザからの「注文する」ボタン押下後、実は redirect & callback が発生する場合がある（〇〇ポイントが貯まる決済手段とか）

ひとつの注文データの処理が完了するまでは、そのデータに対する処理をベースライン or カナリアを統一させたい（じゃないと調査が大変）

AWS CodeDeploy や LB を使ったカナリアリリースでは、これが難しい

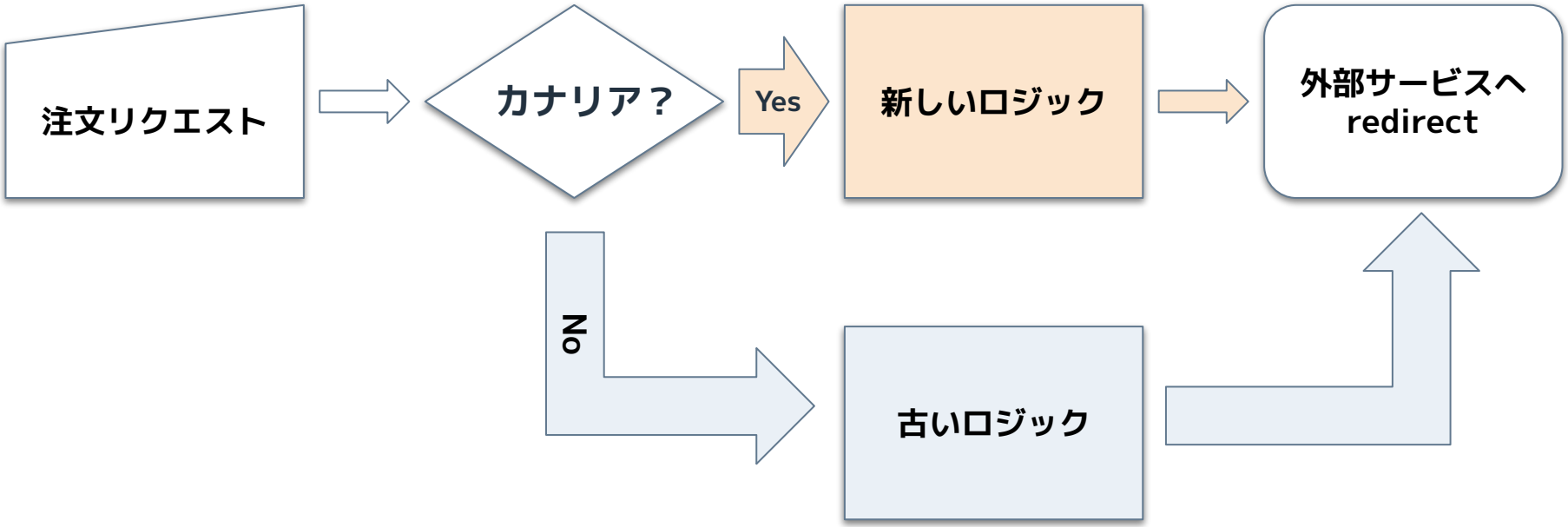
なので、アプリケーションでカナリアリリースを管理する

カナリアで作成したデータに
「カナリアで処理された」とわかる情報を残す（フラグで良い）

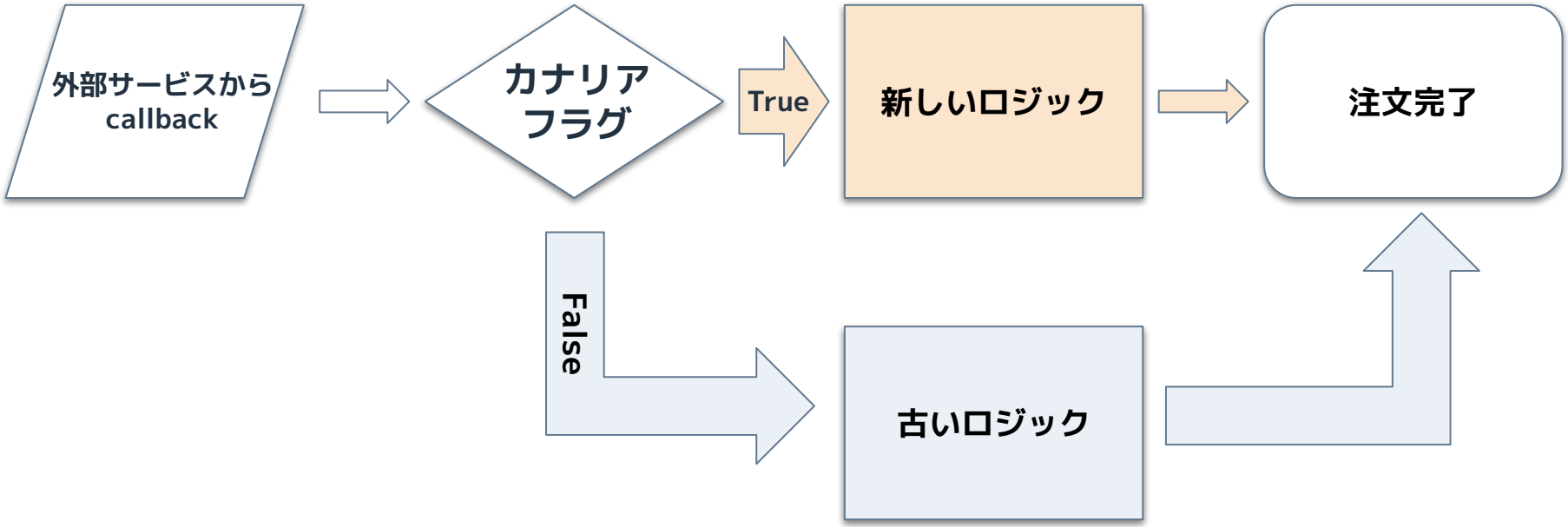
redirect & callback でリクエストが複数に分かれても、
上記を参照してベースライン or カナリアのロジックを判別できる

さらに、バグ発見時の調査も容易になった（実際に判断しやすかった）

注文データ作成の流れ ①



注文データ作成の流れ ②



書き込み先の変更
アプリケーションによるカナリアリリース

イメージできたし、だいたいできたよね？

書き込み先の変更
アプリケーションによるカナリアリリース

イメージできたし、だいたいできたよね？

いいえ 

増改築されてきたロジック・カラム

カバレッジ不足のテストコード

増改築されてきたロジック・カラム

- 膨大なパターンになるロジック
- 増えてきたカラムの数も膨大に……

カバレッジ不足のテストコード

- テストカバレッジが少ない RSpec files
- 動いているコードが正義の状態……

増改築されてきたロジック・カラム

- 膨大なパターンになるロジック
- 増えてきたカラム 膨大に……

カバレッジ不足のテストコード

- テストカバレッジが少ない RSpec files
- 動いているコードが正義の状態……

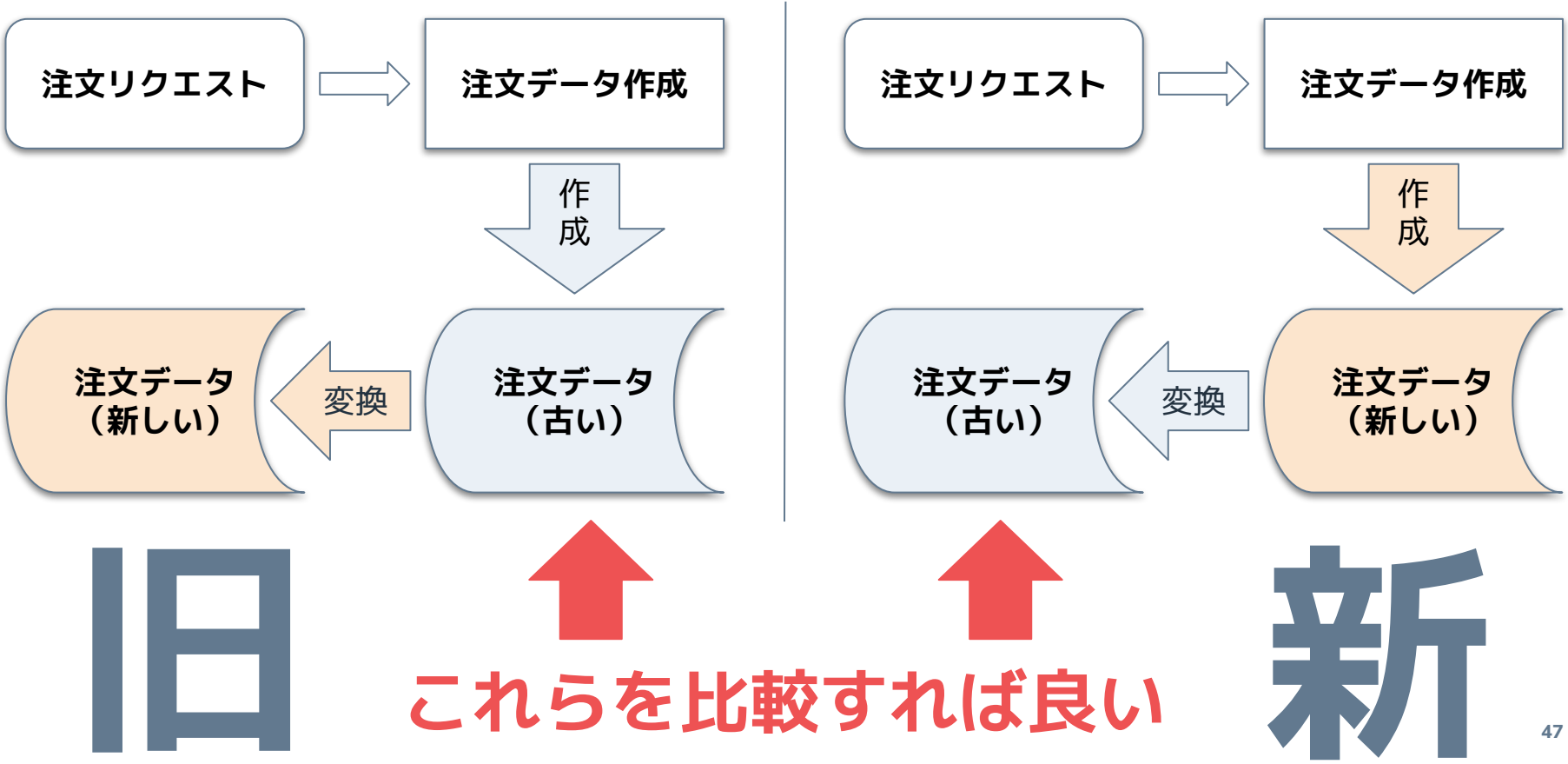
動いているコードを品質の担保にしよう

1. 移行前のコード（ベースライン）で作成した注文データ
2. 移行後のコード（カナリア）で作成した注文データ

理屈上、同じ注文データができるはず

これらを比較すれば良い！

これらを比較すれば良いのイメージ



つまり、こういうこと（実際よりかんたんにしています）

```
# ベースラインの注文データ作成
post '/orders', params:
old_order = Order.find(response.parsed_body['id'])

# カナリアの注文データ作成
create_feature_flag
post '/orders', params:
new_order = Order.find(response.parsed_body['id'])

expect(old_order).to eq new_order
```

リリース後のバグ検知も重要

カナリアで新しいデータ構造の注文データを作成した後、
古いデータ構造の注文データに変換する（二重書き込み）

それを新しいデータ構造の注文データに再変換
再変換結果で差分が出てきたら、バグの可能性あり！

エラートラッキングツールで通知をする（Sentry を使用）

つまり、こういうこと（実際よりかんたんにしています）

```
# カナリアで注文データ作成 => 古いデータ構造に変換
old_structure_order = Converter.inverse(new_structure_order)

# 差分検証
diff = Comparator.call(
  old_structure_order,
  new_structure_order,
)
Sentry.capture_message(diff:) if diff.present?
```

これで安全にリリースできる

これで安全にリリースできる、はず？

長くなったので、ここまでのまとめ

- 新しいデータ構造を作って、二重書き込み
- 書き込み先の変更は、以下で品質を担保
  - アプリケーションのカナリアリリース
  - 新旧データ構造の差分検証をするテスト
- リリース後のバグ検知も忘れない

移行リリースしてみて

リリースの割合を刻むのは、
適用する利用事業者の数

最初は数事業者から入れていき、
段階的に割合を増やして、全体適用を目指す

移行リリースの方針：初期→経過観察

一通りの処理（決済完了から返金まで）が
カナリアで動いたことを確認

一通りが動いたかどうかについては、
新旧ロジックのどちらかを使ったのかを記録済み

その注文データのフラグを見れば、
カナリア・ベースラインのどちらかがわかる

次はランダム選定で数%まで割合を増やし、
一定期間の様子を見る

あとは全事業者に適用するまで広げていく――

あった 

カナリアに切り替えた途端、
一定時間後に動く非同期処理のパフォーマンスが悪化したり

想定していなかったエッジケースで、
カナリアで作った注文データに差分が見つかったり

失敗：でも、サービス停止せず

feature flags を DB で管理していたので、
切り戻すのはフラグを折るだけですぐに終わる

フラグを折ると、以降の注文は
信頼と実績のベースラインで動き、すぐに止血完了

差分が出てしまったデータはあるべき状態になるように修正
新旧の変換・逆変換ができるので、正しい側に合わせる

その後の対策

パフォーマンスの悪化
- 原因箇所を特定
- それをチューニングして検証・再リリース

想定していなかったエッジケース
- エッジケースを考慮したテストを追加
- 周辺コードを読み直して精査・再リリース

これらを繰り返して、
全部の利用事業者で、

新ロジック（カナリア）に切り替えた 🎉

まとめ

今日のプログラムで持ち帰れること（結論・再掲）

1. アプリケーションでやるカナリアリリース
2. 新旧データ構造の差分チェックの有用性
3. テスト不足でも安全にリリースする方法

1. アプリケーションでやるカナリアリリース

ロードバランサーでは得られないメリット

- リクエスト単位とは異なる別の単位で、
  意図的にリリース後の挙動に切り替えられる
- 同じコード上に feature flags で分岐しているので、
  新旧ロジックを使ったテストで活用できる

2. 新旧データ構造の差分チェックの有用性

新旧データ構造を相互変換できることで得られたメリット

- 多くのカラムがあるテーブルでも、
  同じオブジェクトに対する比較でまとめて検証できる
- データ構造の変換・再変換で、
  おかしなデータになっていないかを差分で検証できる

3. テスト不足でも安全にリリースする方法

先述のふたつを組み合わせることで得られたメリット

- アプリケーションでやるカナリアリリースだから、
  新旧ロジックの両方を同じテストで動かせる
- 新旧データ構造の相互変換ができるから、
  同じオブジェクトを比較して全カラムの検証ができる

おわりに

長年運用しているサービスで扱う主要データは
いつか古くなって運用に耐えづらくなり、
新しいデータ構造に置き換えたくなることがあるでしょう

今後も長くサービスを運用していくため、
いまの実態に合わせたデータ構造へ
安全に切り替えてみるのはどうでしょうか？