

Microservicesアーキテクチャのトランザクション管理の現実

Shingo Yamanari

Principal Cloud Solution Engineer
Oracle Corporation Japan, Solution Architect
May 17th, 2022





Agenda

- 1. マイクロサービスの世界観
- 2. マイクロサービス間の一貫性の仕組み
- 3. 既存システムのマイクロサービス化の現実
- 4. オラクルのトランザクション管理ソリューション





マイクロサービスの世界観

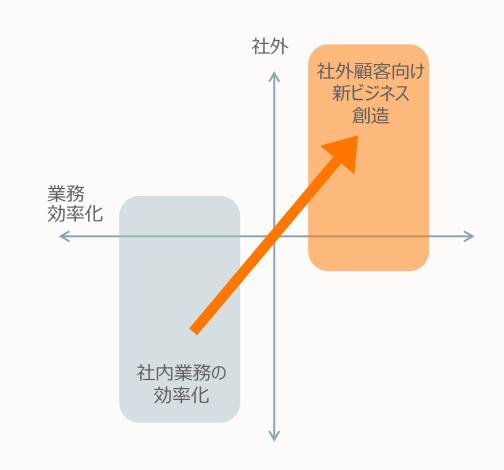
企業システムに求められるニーズの変化 デジタルトランスフォーメーション | 社内業務の効率化から新ビジネスの創造へ

ビジネスにもとめられる進化のスピードと品質

- ✓サービスの早期リリース
- ✓市場の動向/反応を早期フィードバックして対応
- ✓サービス停止による機会損失をなくす

システムは変化するビジネス要件に合わせ **短い時間で高頻度**にリリースする事が求められる

- ✓アプリの変更をすぐに本番環境に適用
- ✓変化を許容できるシステムを構築
- √停止時間の短縮と運用作業の効率化

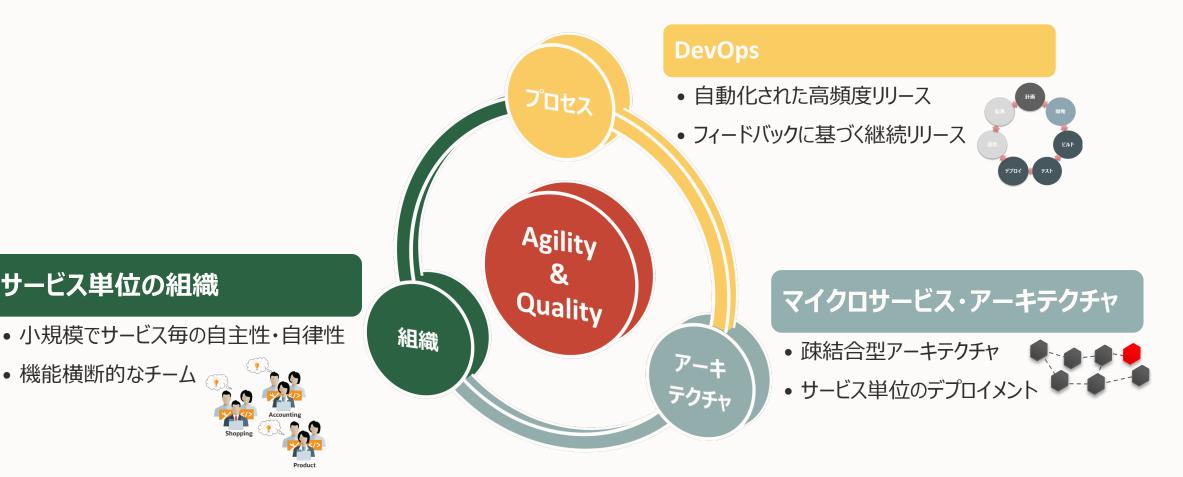


マイクロサービスの世界観

サービス単位の組織

• 機能横断的なチーム

迅速に高品質なシステムを育て続けるための戦略





マイクロサービス・アーキテクチャとはサービス間の影響を極小化しシステムの変更容易性を高めるアーキテクチャ



保守とテストの容易性

- 分割したサービス毎に組織を編成し開発・運用の自由度を高める
- 更新単位を最小限にすることでテスト規模を最小化



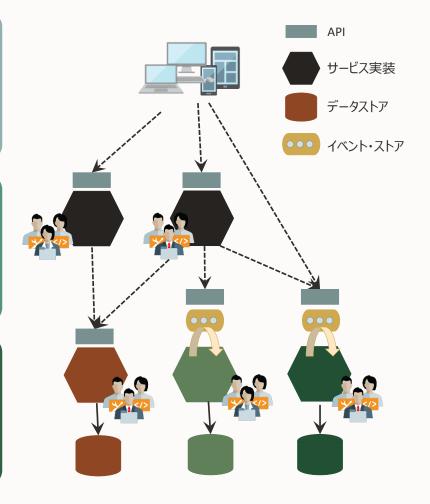
疎結合

- API化や非同期化によりサービス間の結合度を低減
- •変更による他の稼働中のサービスにへの影響を極小化



独立してデプロイ可能

- データソースやアプリケーション・モジュールをサービス毎で占有
- デプロイやスケールの変更の単位サービス毎で任意に最適化



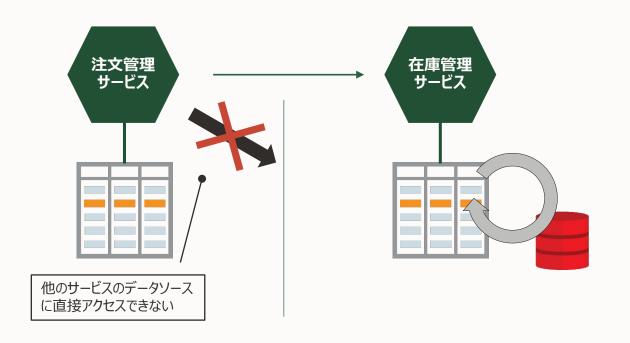


マイクロサービスにおける一貫性の考え方

結果整合性による一貫性

相手のサービスのデータソースに直接アクセスできない

- トランザクションによる一貫性が利用できない
- 自サービスと相手のサービスそれぞれの「結果整合性」により全体の一貫性を保つ





マイクロサービス間の一貫性の仕組み

マイクロサービスで検討される代表的な一貫性の仕組み

サービス間をまたがるトランザクションの仕組み

Sagaパターン

補償トランザクションによる(事後)結果整合性

- 処理の成功を前提とした楽観的な呼出し手法
- 一部の処理が失敗した場合に、既に完了した処理を 取り消す「補償トランザクション」により整合性を取る

Insert Delete

Delete

Insert

Insert

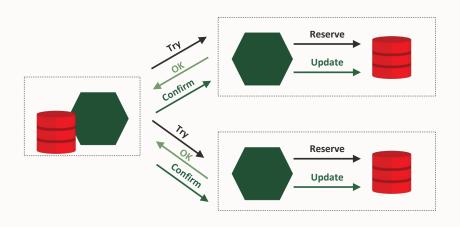
Delete

Insert

Insert

TCC (Try/Confirm/Cancel) パターン 予約ベースの(事前)結果整合性

- ・ 処理完了の可否を事前に確認した上で処理を実施
- Tryフェーズにより不整合の生じる処理を行わないことで整合性を担保





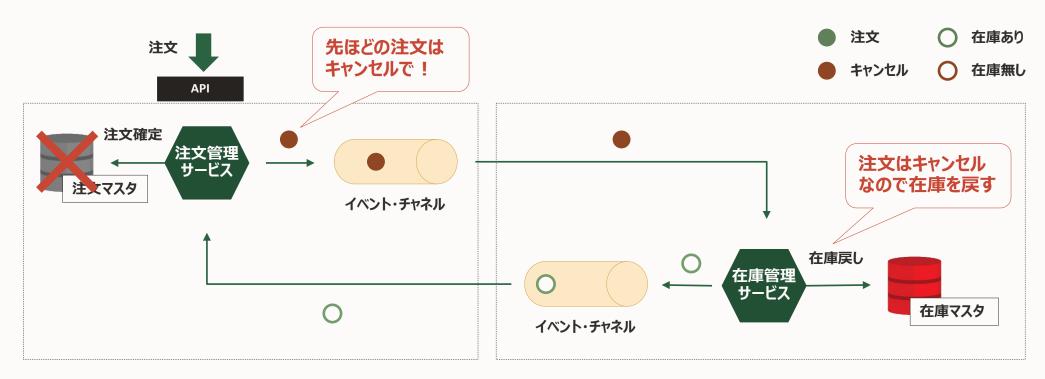
Sagaパターンの補償トランザクション

障害時などにビジネス・トランザクションとして回復する

何らかの障害で処理を完了できない場合に、処理を取り消して整合性を合わせる。

例:注文マスタの更新に失敗→在庫修正を元に戻してもらう(=注文のキャンセルを発行)

複数個所での障害など、補償トランザクションの設計が煩雑になりやすい。。。→ 長いトランザクションは控えるのがbetter

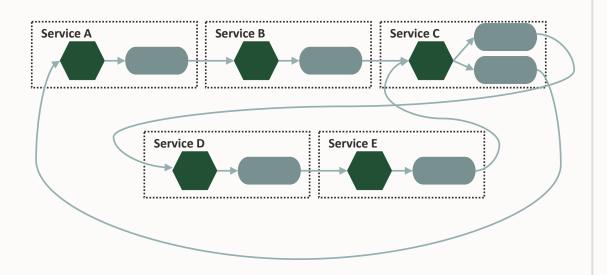


2つのSagaパターン

Choreography-based Saga & Orchestration-based Saga

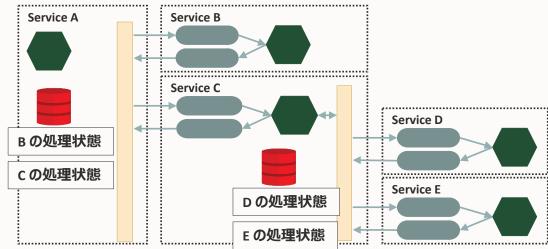
Choreography-based Saga

- 呼出し元のイベントにしか興味を持たない
- 呼出し先に対する責任を持たない
 - 全体のトランザクションに無責任になりがち
- いくらでも連鎖できるためトランザクションが長くなり易い



Orchestration-based Saga

- 呼出し先を呼出し元が管理する
- ネスト型の構造が可能で責任分界点を明確にできる
- 呼出し元が呼出し相手の結果などのステートを管理する必要がある
 - →ステート管理のためのストアが必要





TCC パターンの障害ケース

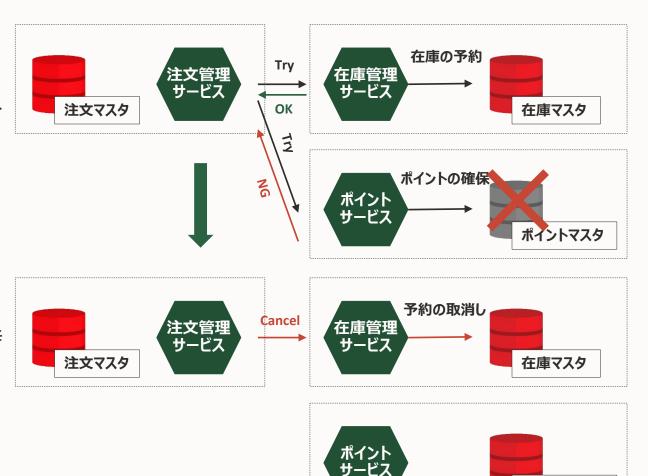
予約していたリソースを解放するのみで対応可能

Tryフェーズで予約したリソースを解放する処理

- 予約が成功したサービスに対して取消を発行
 - 各サービスへの予約の状態は同一のコンテキスト (例:注文番号など)での管理が必要
 - 予約が成功した相手のみを抽出してCancel

サービス自体の障害を考慮して、イベント・ストア等を活用した自動リトライの仕組みも要検討

- Try に対する応答時に呼出し元がダウン
- Try時は稼働していた呼出し先のサービスがCancel時にダウン





ポイントマスタ

従来のデータソース間の一貫性の担保のしくみ

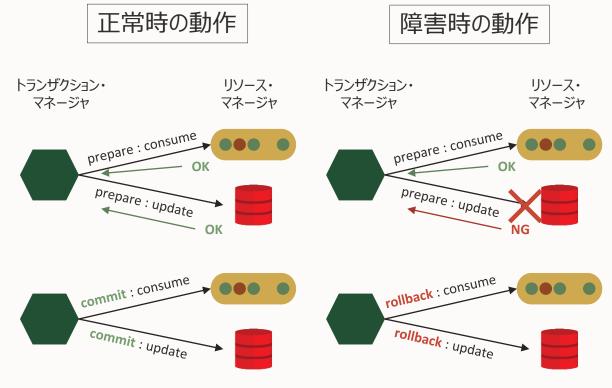
XAトランザクションを利用した一貫性の担保

X/Open XA

- X/Open社による分散トランザクションの標準規格
- 2フェーズ・コミットをベースとした一貫性担保のしくみ
- 2フェーズ・コミットの仕組み
- 更新対象のデータを排他ロック
- 更新可能かどうかを事前にチェック
- 全員更新可能であれば一斉にコミット、一人でもNGなら 全員ロールバック

XAのメリットとデメリット

- メリット
 - 複数のデータソースに跨る一貫性を確実に担保でき 実装も容易
- デメリット
 - 広範囲の排他ロックにより性能ボトルネックになり易い





Saga、TCC及びXAのトランザクション制御の特性

| | Sagaパターン | тссパターン | ХА |
|----------------|--|--|--|
| 整合性の確実さ | 補償処理の失敗やトランザクション間における 同一データに対する競合により一貫性が損 なわれやすい | トランザクション間でのデータ競合を防ぐことで一貫性を確実に保証 | データセットに対する 排他によりトランザクショ ンの一貫性を確実に保証 |
| 整合性制御の複雑さ | トランザクション参加者の処理順序や結果を 踏まえた補償処理の制御が必要 | 失敗時の回復を含め トランザクション内に閉じて容易に制御 | 異なるトランザクション間でのデータを 完全に 排他することによりシンプルに制御 |
| ビジネス・データに対する排他 | データアクセス単位の排他のみ | 予約により異なるトランザクション間で データ が 競合しない | 処理対象となるデータセットに対する 排他ロッ ク |
| サービス間のデータソース分離 | トランザクション参加者ごとに データソースを 分離 | トランザクション参加者ごとに データソースを 分離 | トランザクション・イニシエータが 他のサービス のデータソースを直接参照 |
| 主なデメリット | トランザクション間での競合などを加味した補償処理の確実な自動化が困難 | 一定期間の予約状態によりビジネス・データ が遊休状態になる | 完全な排他により、他のトランザクションへの レイテンシや並列度に影響が生じやすい |
| 利用が想定される領域 | 他のトランザクションと分離されやすく、補償処理に人手が関与しやすい領域 ・ ユーザ単位で行われる受付業務 ・ カート→チェックアウト→配送のような長期的なトランザクション、etc. | 確実な一貫性が求められ、業務的に一時的な予約が許容される領域 ・ 複数商材の一括決済、在庫を要する修理手配 ・ デポジット型のポイント決済、etc. | ACID特性の順守が必要な領域 ・ 勘定系 ・ システム制御マスタ、etc. |





既存システムのマイクロサービス化の現実

エンタープライズ・システムのマイクロサービス化の現実

既存業務・システムの制約を前提としたマイクロサービス化

既存の業務に求められるトランザクションの性質

- トランザクションに人手が介在する業務 (電話による申込み確認など)
- 確実性が重要視される業務 (=補償処理が許容されない)
- ACIDが要求される業務 (金銭の移動など)
 - ACIDが保証されない場合の影響が推測できない業務・システムなど

マイクロサービス化が必要となる既存システムの特性

- Agilityを重視すべき箇所から部分的にマイクロサービス化
- マイクロサービス化の対象にならない (または不可能な) 既存システムとの連携
 - 連携対象の既存システムに対する変更が許容されない
 - そもそも既存システムに連携手法が用意されていない → **直接既存のデータソースを参照する必要**がある
 - マイクロサービス側では**既存システムに対する依存度を排除**したい

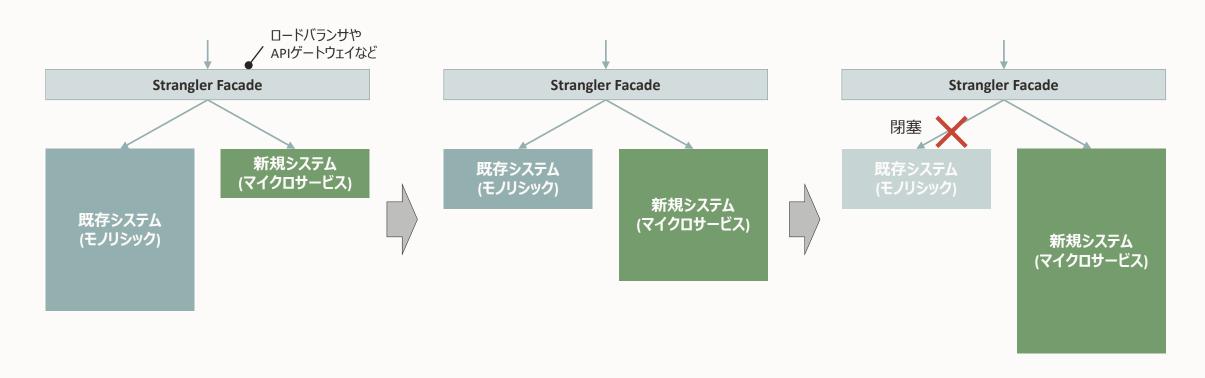


基本的なマイクロサービス化の戦略 (Strangler Pattern)

必要な箇所から切り出してマイクロサービス化

Strangler Pattern = 既存システムを段階的に移行しながら作り変える際のプラクティス

- 既存システムと新規システムをロードバランサなどでルーティングして移行期間の並行稼働
- アプリケーションのバージョンアップやアーキテクチャの見直しなど、幅広く使われる常套手段

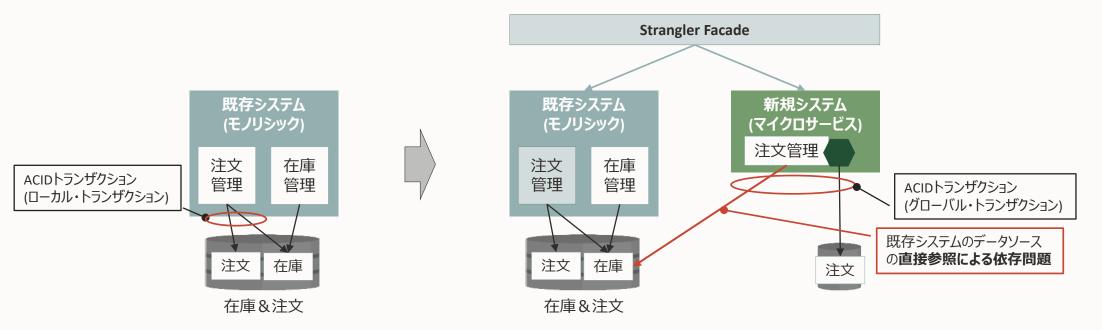


既存システムとマイクロサービスが共存する際の課題

既存業務のトランザクション要件を遵守

【マイクロサービス化の想定ケース】

- **業務的に「ACID遵守を前提」**とした在庫管理 + 注文管理
- 注文管理からマイクロサービス化し新規の注文管理用のデータソースごと分離する 【想定される実装案】
- 新規マイクロサービスから既存システムのデータソースを参照してXAを適用 → 既存システムへの依存度が高くなる

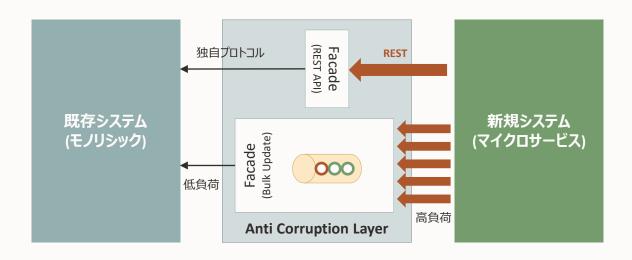


既存システムに対する依存度を切り離す戦略 (Anti Corruption Layer Pattern)

依存関係を吸収するファサードの配置

Antic Corruption Layer Pattern = 異なるシステム間の依存関係を切り離すためのプラクティス

- システム間に緩衝レイヤを設け、相互を隠蔽しつつに対する依存性・影響を吸収する
 - 既存システムの存在を隠蔽する
 - 既存システムの制約を解消する(独自プロトコルの標準化、実装言語の変換、etc.)
 - 既存システムに対する影響を吸収する(参照キャッシュの配置、高トラフィック処理のバルク処理化、etc.)

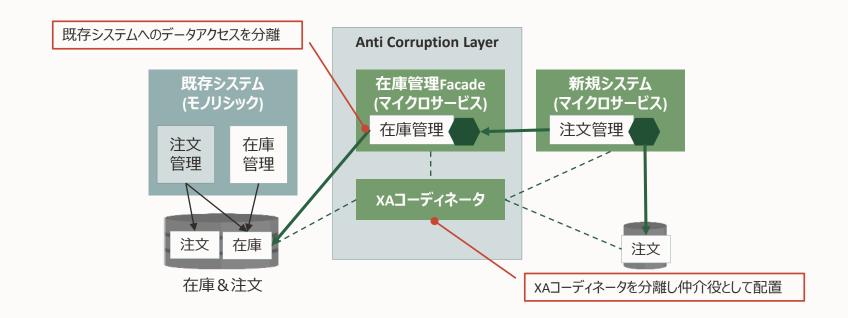


Anti Corruption Layer Patternによる既存システムのマイクロサービス化

既存システムのデータソースを隠蔽したXAによる連携

既存システムへのデータソース・アクセスとXA連携をAnti Corruption Layerで分離

- 既存システムのデータ構造に依存するデータアクセスをFacadeとして分離
- 既存システムと新規システムのデータソース間のXAコーディネータを新規システムから分離し仲介役として配置
 - XAコーディネータをアプリケーションから切り離すことでマイクロサービスの独立性を高める



マイクロサービス間のトランザクション管理の在り方

マイクロサービス間で共通する Cross-cutting concernsの在り方

高度なトランザクション管理を独自に実装するリスク

- 各サービスごとに異なる実装 → 全体のトランザクション管理の品質が劣化
- 実装難易度の高い開発を行うエンジニアの確保 → 人的リソースの浪費

サービス間のトランザクション管理の実現で死守すべき要点

- 全サービスで横断的に利用可能な「統一された手法」
- サービス実装と分離可能な「フレームワーク」化された機構

マイクロサービス間の主要なトランザクション管理を提供するトランザクション・コーディネータ・フレームワーク

Oracle Transaction Manager for Microservices (OTMM)



オラクルのトランザクション管理ソリューション



Oracle Transaction Manager for Microservices (OTMM)

マイクロサービスにおける分散トランザクション管理コーディネータ・フレームワーク

マイクロサービス・アーキテクチャの分散トランザクション管理を提供するトランザクション・コーディネータ

- TCCパターン、Sagaパターン (MicroProfile LRA準拠)
- 既存システム連携等に活躍するXAコーディネータも提供

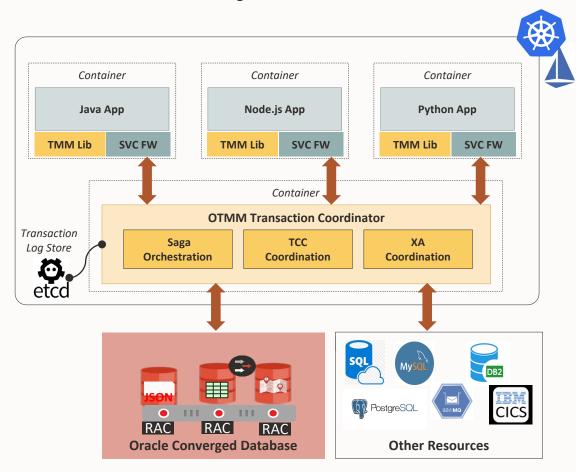
複雑な分散トランザクション制御の実装が不要

- アプリケーションと分離されたトランザクション・コーディネータを別コンテナで起動
- トランザクション・コーディネータとの通信をフレームワーク化したライブラリを各言語向けに提供

多種のプログラミング言語に対応

Java, JavaScript, Python, C/C++, Go

Oracle Transaction Manager for Microservices アーキテクチャ





Saga、TCC及びXAのトランザクション制御の特性 データソースを分離したマイクロサービス間でのXAトランザクションを実現

| | Sagaパターン | тссパターン | ХА |
|----------------|--|--|--|
| 整合性の確実さ | 補償処理の失敗やトランザクション間における 同一データに対する競合により一貫性が損 なわれやすい | トランザクション間でのデータ競合を防ぐことで一貫性を確実に保証 | データセットに対する 排他によりトランザクショ ンの一貫性を確実に保証 |
| 整合性制御の複雑さ | トランザクション参加者の処理順序や結果を 踏まえた補償処理の制御が必要 | 失敗時の回復を含めトランザクション内に閉じて容易に制御 | 異なるトランザクション間でのデータを 完全に 排他することによりシンプルに制御 |
| ビジネス・データに対する排他 | データアクセス単位の排他のみ | 予約により異なるトランザクション間で データ が競合しない | 処理対象となるデータセットに対する 排他ロッ ク |
| サービス間のデータソース分離 | トランザクション参加者ごとに データソースを 分離 | トランザクション参加者ごとに データソースを 分離 | トランサクション イニシエータが 他のリービス のデータリースを直接参照 |
| 主なデメリット | トランザクション間での競合などを加味した補償処理の確実な自動化が困難 | 一定期間の予約状態によりビジネス・データ が遊休状態になる | 完全な排他により、他のトランザクションへの レイテンシや並列度に影響が生じやすい |
| 利用が想定される領域 | 他のトランザクションと分離されやすく、補償処理に人手が関与しやすい領域 ・ ユーザ単位で行われる受付業務 ・ カート→チェックアウト→配送のような長期的なトランザクション、etc. | 確実な一貫性が求められ、業務的に一時的な予約が許容される領域 ・ 複数商材の一括決済、在庫を要する修理手配 ・ デポジット型のポイント決済、etc. | トランザクション・コーディネータを分離することで、トランザクション参加者ごとに デー タソースを分離 |

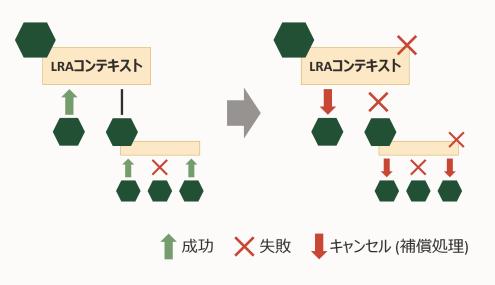


MicroProfile LRA (Long Running Action)

Sagaパターンにおける補償トランザクションのJava標準仕様

連携するサービス間で、処理状態を判別して補償処理を調停するフレームワーク仕様

- LRA (=複数のサービスによる一連の処理)内の1つのサービスが失敗すると、他のサービスの補償処理を呼び出す
- JAX-RSで呼び出されるメソッドにアノテーションを付与して調停が必要な処理をマーク
 - LRAの開始/終了/参加/離脱、補償処理、正常終了、etc.
 - LRAのコンテキストIDがHTTPへッダで伝播される



```
@Path("/")
@ApplicationScoped
public class SimpleLRAParticipant
 @LRA(LRA.Type.REQUIRES NEW)
                                              呼び出されるとLRAが開始される
 @Path("/cdi")
 @PUT
 public void doInTransaction(@HeaderParam(LRA HTTP CONTEXT HEADER) URI IraId) {...}
 @Complete •
                         LRAが完了すると呼び出される(リソースのクリーンアップなどに使う)
 @Path("/complete")
 @PUT
 public Response completeWork(@HeaderParam(LRA HTTP CONTEXT HEADER) URI Iraid, String data) {...}
 @Compensate
                                LRAがキャンセルされるとと呼び出される (補償処理に使う)
 @Path("/compensate")
 public Response compensateWork(@HeaderParam(LRA HTTP CONTEXT HEADER) URI Irald, String data) {...}
```



Project Helidon

標準仕様に対応したオラクル発のマイクロサービス・アプリケーション・フレームワーク



OracleがホストするOSSプロジェクト

- GitHubでソースコードを公開: https://github.com/oracle/helidon
- Helidonの商用サポートはWebLogic Server/Coherence/Verrazzanoのサポート契約に含まれる

マイクロサービスアプリケーションが必要とする機能を提供するJavaライブラリの集合体

- 単体のJVMとして動作し、アプリケーションサーバ不要、容易なコンテナ化
- 必要なコンポーネントを追加して拡張することも可能

マイクロサービスの開発・運用を支援する機能を提供

- OpenMetrics(監視)、OpenTracing(追跡)、OpenAPI(API公開)
- 耐障害性/回復性: ヘルスチェック、サーキット・ブレーカ







kubernetes





2つのプログラミングモデルを提供

- Helidon MP: MicroProfile標準仕様準拠の宣言的記法(Java EE開発者フレンドリー)
- Helidon SE: 関数的型記法

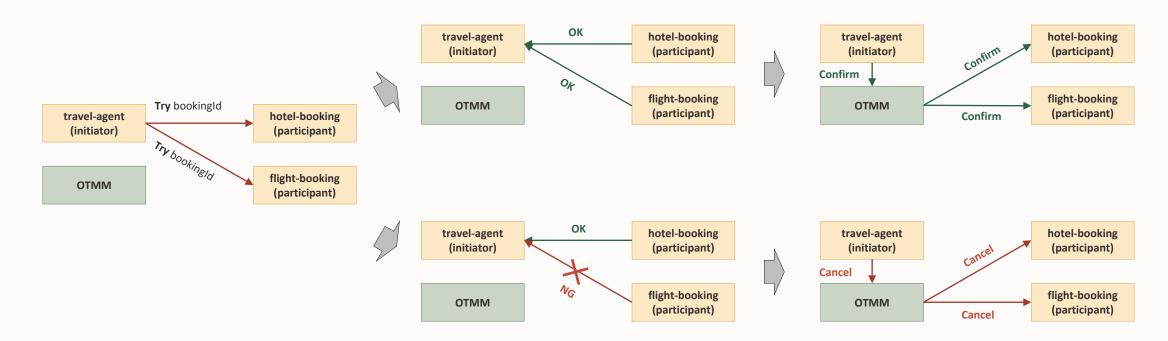


デモ) OTMMによるTCCトランザクション

ホテルとフライトを予約する

【TCCのサンプルケース】

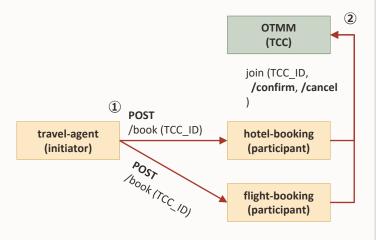
- ホテルとフライトを同時に予約するケース
 - 両方とも仮予約できない限り予約は成功しない
 - 処理が失敗した場合は仮予約を解放



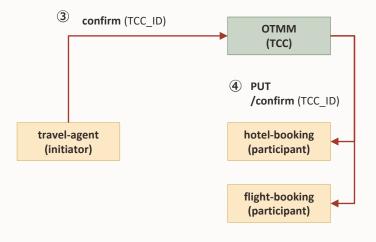
OTMMにおけるTCCトランザクションの仕組み

トランザクションをREST経由で伝播

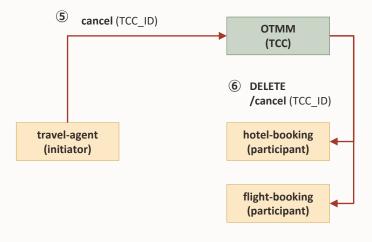
- ① トランザクションIDをHTTPへッダに 付与して各参加者のTry用のURI にPOST
- ② トランザクション参加者はOTMM に対してトランザクションIDに対す る Confirm/Cancel 用のURIを登 録



- ③ 双方のTryが成功の場合は、イニシエータがOTMMに対してトランザクションIDを指定してConfirm
- ④ OTMMは登録された参加者の Confirm用のURIに対してPUT

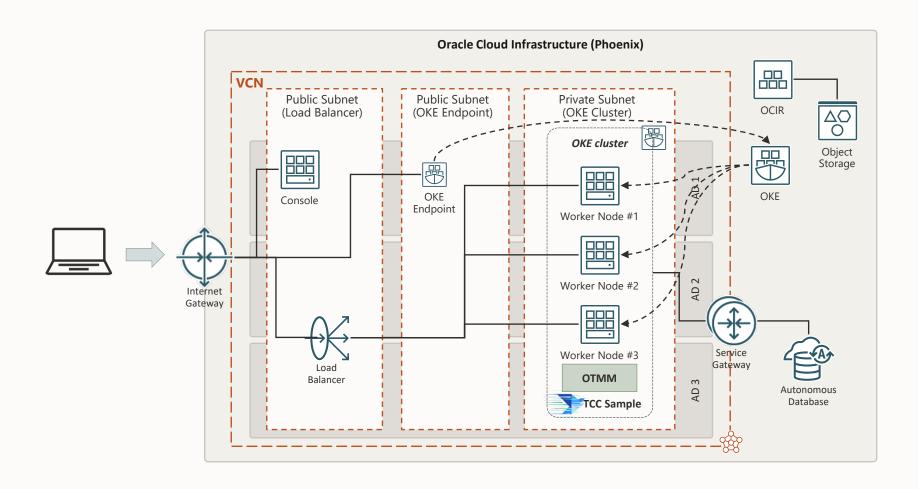


- ⑤ いずれかのTryが失敗の場合は、 イニシエータがOTMMに対して Cancel
- ⑥ OTMMは登録されたCancel用の URIに対してDELETE





デモ環境の構成





OTMMによるTCCトランザクションの記述例 #1

イニシエータ側の記述

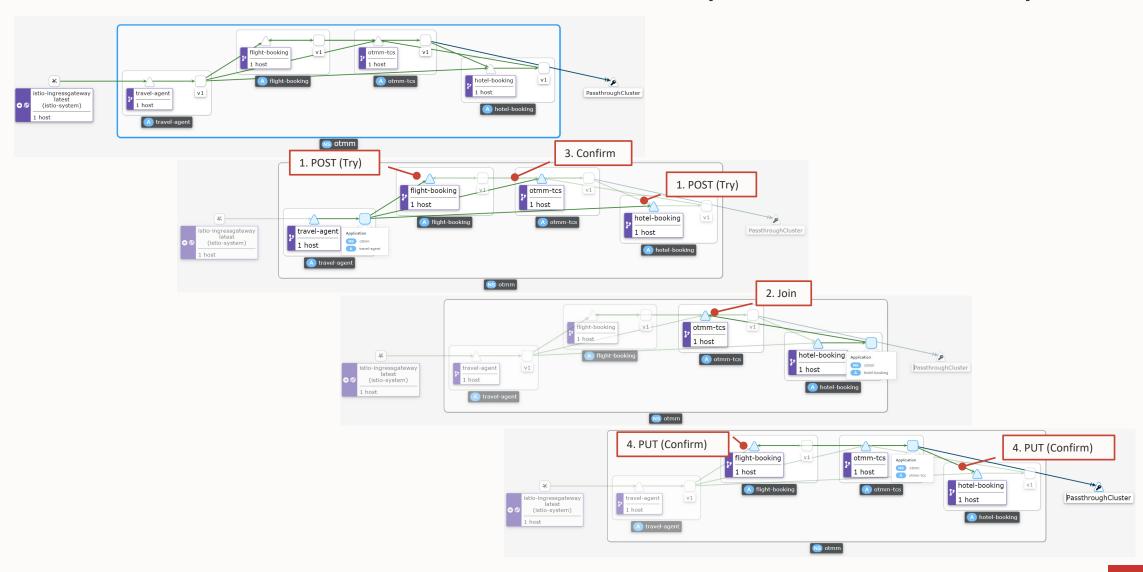
```
@TCC アノテーションを付与することでクラス内からのREST呼
@Path("api/")
                     出しをインターセプトしてTCCトランザクション情報を付与
@RequestScoped
@TCC timeLimit = 120, timeUnit = ChronoUnit.SECONDS)
public class BookingResource {
   @POST
   @Path("booking")
   @Consumes (MediaType.APPLICATION JSON)
   @Produces (MediaType.APPLICATION JSON)
   public Response booking() {
       try
           String bookingId = UUID.randomUUID().toString();
           Client svcClient = ClientBuilder.newClient();
                                                                                                       各参加者のREST APIに対してTry
           String svcEndpoint = hotelBookingAppUrl + "/create";
           Response respHotel = svcClient.target(svcEndpoint).request(MediaType.APPLICATION JSON).post(Entity.json(bookingId));
           svcEndpoint = flightBookingAppUrl + "/create";
           Response respFlight = svcClient.target(svcEndpoint).request(MediaType.APPLICATION JSON).post(Entity.json(payload));
           int respStatus = resp.getStatus();
           if ( (respHotel.getStatus() == Response.Status.OK.getStatusCode()) &&
                (respFlight.getStatus() == Response.Status.OK.getStatusCode()) ) {
               // 成功時の処理
               ConfirmResponse confirmResponse = TccClient.confirm();
                                                                          OTMM API経由でOTMMに対してConfirm
            } else {
               // 失敗時の処理
               CancelResponse cancelResponse = TccClient.cancel();
                                                                      OTMM API経由でOTMMに対してCancel
```

OTMMによるTCCトランザクションの記述例 #2

参加者側の記述

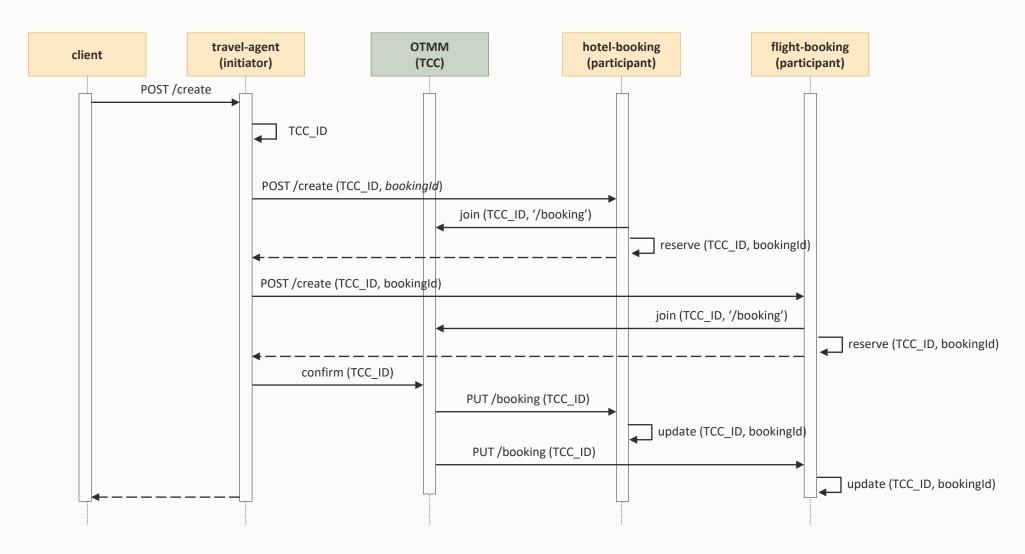
```
@TCC アノテーションを付与することで、このクラスに対するREST
@Path("api/")
                     呼出しをインターセプトしてTCCトランザクション情報を引き継ぎ
@ApplicationScoped
@TCC(timeLimit = 120, timeUnit = ChronoUnit.SECONDS)
public class HotelBookingResource extends Application {
   @POST
   @Path("create")
   @Consumes (MediaType.APPLICATION JSON)
                                                            REST経由で伝播されたトランザクション情報を取得
   public Response create() {
       int tid = TccClient.getTransactionId();
       // トランザクションごとの予約処理 (トランザクションIDに対する予約のマッピングなど)
       return Response.ok().entity(resultJson).build();
   @PUT
                       Confirm用のREST APIとしてOTMMに登録
   @Path("booking")
   @Consumes (MediaType.APPLICATION JSON)
   @Confirm _
   public Response confirm() throws TccUnknownTransactionException {
       int tid = TccClient.getTransactionId();
       // 予約から確定に変更する処理
       return Response.ok().entity(resultJson).build();
   @ DELETE
                       Cancel用のREST APIとしてOTMMに登録
   @Path("booking")
   @Consumes (MediaType.APPLICATION JSON)
   @Cancel 
   public Response cancel() throws TccUnknownTransactionException {
       int tid = TccClient.getTransactionId();
       // 予約リリース処理
       return Response.ok().entity(resultJson).build();
```

Kiali から見たOTMMによるTCCトランザクションの流れ (成功パターン: Confirm)



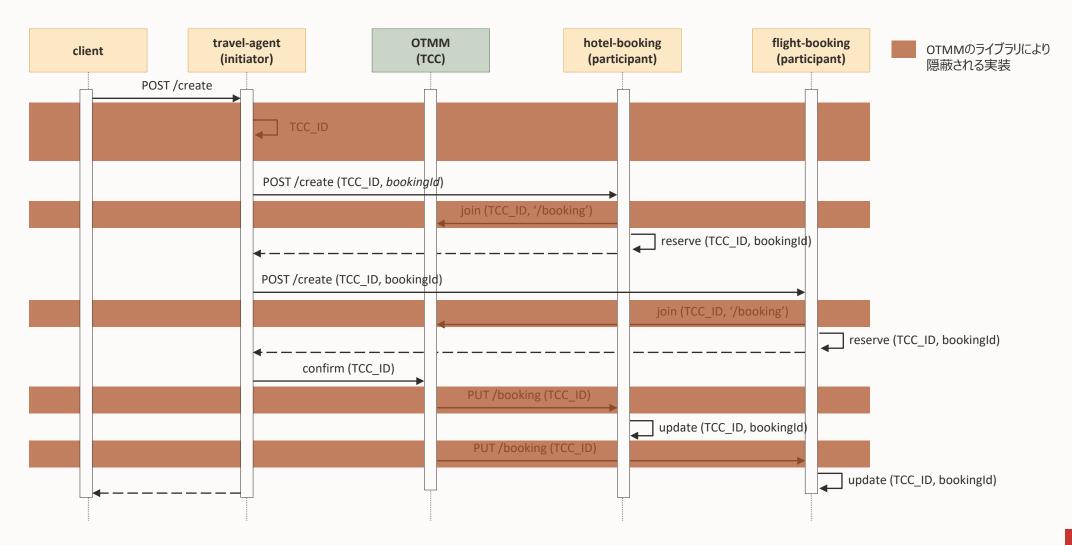
OTMM TCC Sample (成功パターン: Confirm)

OTMMによるTCCパターン実装時の一連のシーケンス

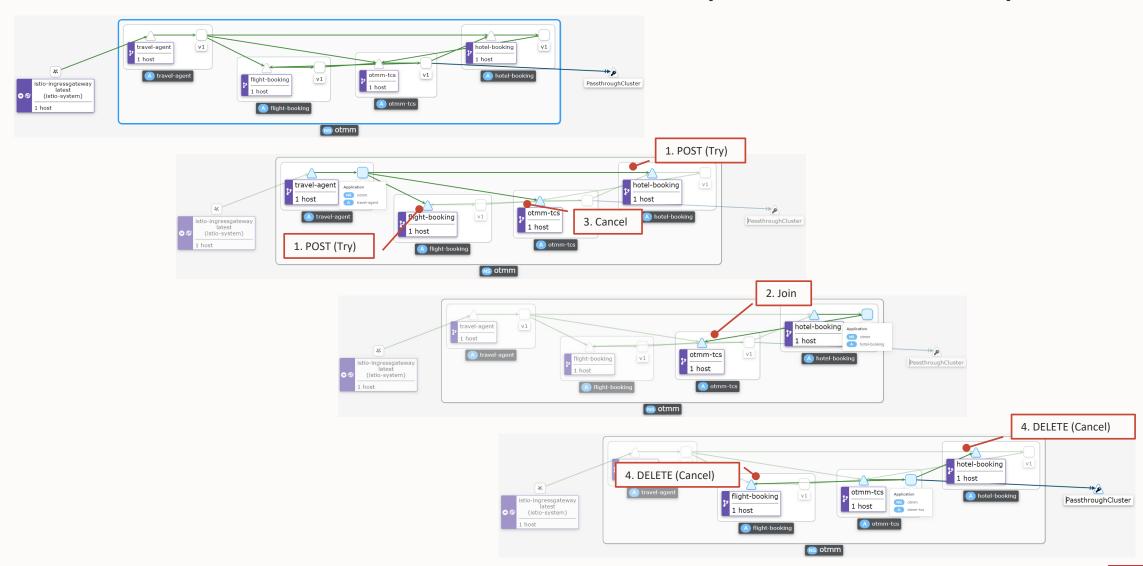


OTMM TCC Sample (成功パターン: Confirm)

OTMMによるTCCパターン実装箇所

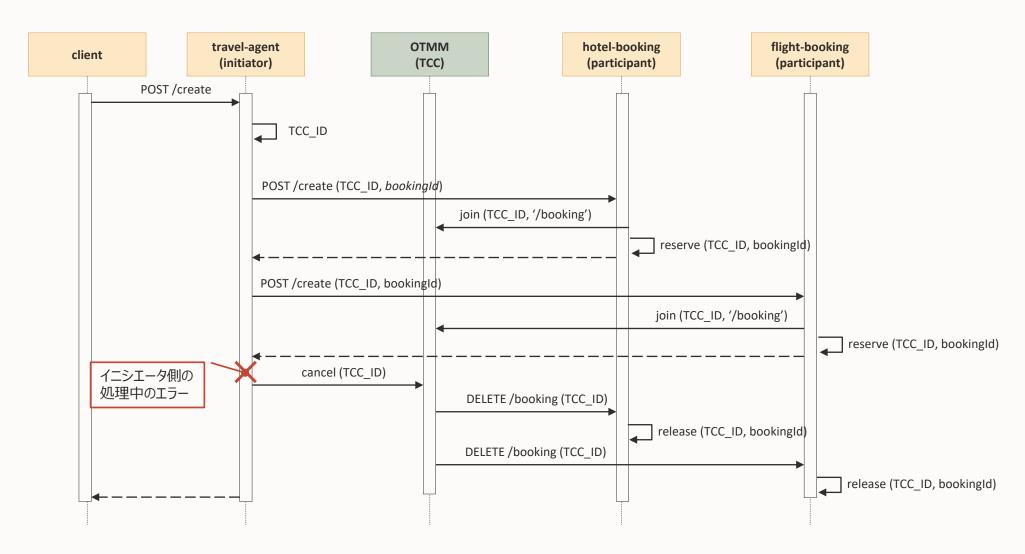


Kiali から見たOTMMによるTCCトランザクションの流れ (失敗パターン: Cancel)



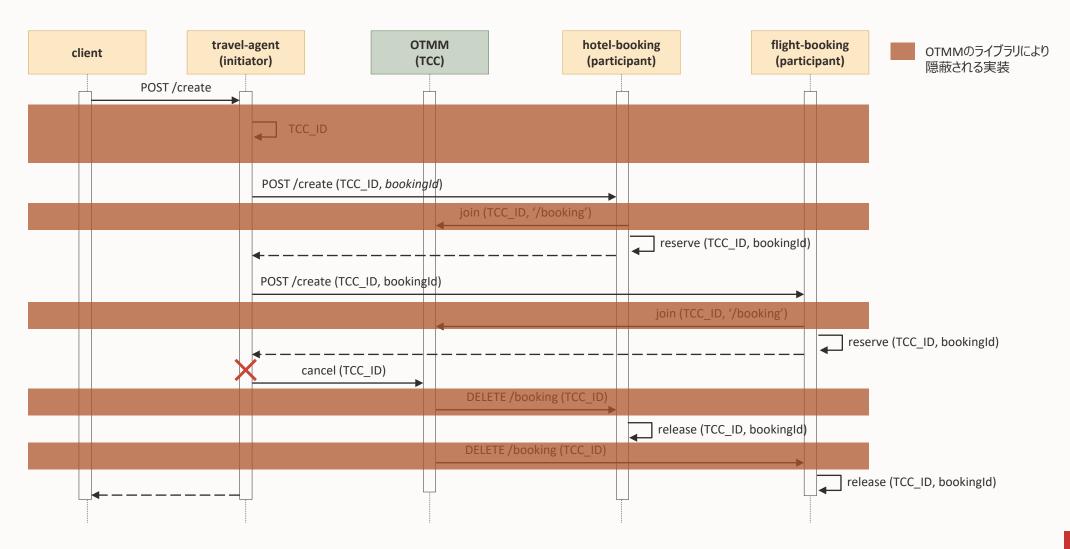
OTMM TCC Sample (失敗パターン: Cancel)

OTMMによるTCCパターン実装時の一連のシーケンス



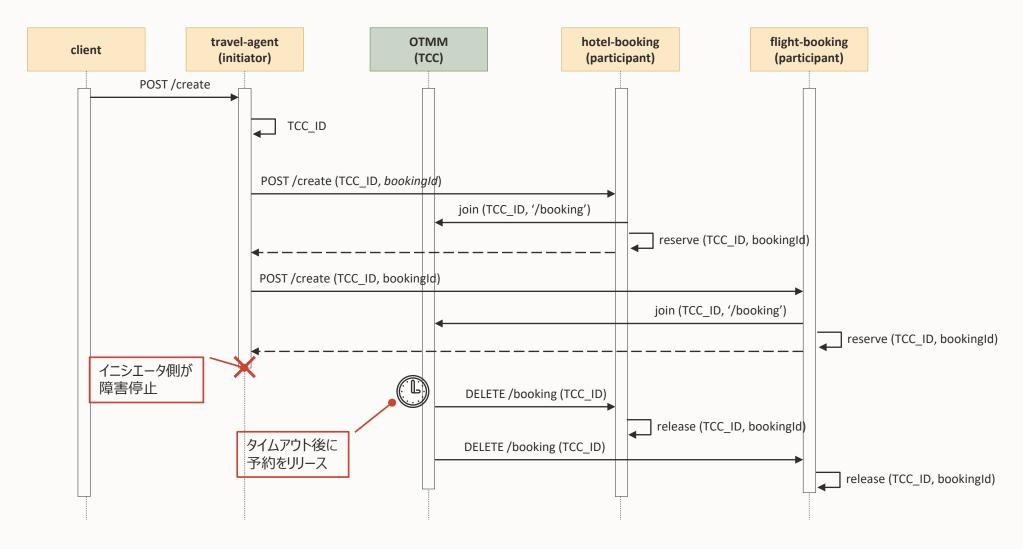
OTMM TCC Sample (失敗パターン: Cancel)

OTMMによるTCCパターン実装箇所



ご参考) OTMM TCC Sample (失敗パターン: Cancel)

トランザクションがタイムアウトするケース



まとめ) 現実を見据えたマイクロサービス化を考える

ビジネス/システム要件を見極めたトランザクション・パターンの選択

- ACIDが必要 → コーディネータ分離型のXAアーキテクチャ
- 確実に整合性が取れればよい予約可能なビジネス・モデル **⇒ TCCで確実かつスケーラブル**に
- ・ 分離された長期のビジネス・トランザクション → Sagaで人手を含めた柔軟な結果整合性を考慮

既存システムをマイクロサービス化するためのアプローチ

- **既存システムと連携**したマイクロサービス化、及びACID制約が必須なケースの存在
- Anti Corruption Layer を利用したXAによる既存システムとの連携

Oracle Transaction Manager for Microservices の特長

- Saga, TCC, XAのトランザクション・コーディネータをアプリケーションから分離させた仕組み
- データソースを含むサービス間の独立性を保ちつつ、標準準拠のトランザクション管理制御をフレームワーク化
- 複雑なトランザクション制御の実装から開発者を解放

