ORACLE

# Oracle DatabaseでのDevOps原則の実装

2023年10月、バージョン1.1



公開

ORACLE

# 本書の目的

本書では、最新のアプリケーション開発原則の概要をOracle Databaseとの関連で説明します。本書は、自動化された方法でコード変更をデプロイするためのアプリケーション開発ワークフローの変更に関するビジネス上の利点の評価を支援することのみを目的としています。開発プラクティスでは無数に近い変更が行われる可能性があるため、この技術概要では提案とベスト・プラクティスを列挙するにとどまります。具体的な導入方法は、常にお客様の要件、スキル、プロセスなどに依存します。

対象読者は、開発者、アーキテクト、およびデータベース環境の一部としてOracle Databaseを利用するチームを率いるマネージャーです。本書をできる限り分かりやすくするために全力を尽くしていますが、この技術概要を最大限に利用するにはOracle Databaseの基礎的な理解が必要です。

ORACLE

# 目次

ORACLE

## 図一覧

ORACLE

# はじめに

ソフトウェア・サービスを顧客に提供する市場にいる者は誰もが、イノベーションを起こし、結果としてソフトウェア開発プロセスを改善することのプレッシャーを感じてきました。競争は決して休まることはなく、この激しい競争の中にいるほとんどの企業は同じペースで動かないわけにはいきません。

新機能を頻繁にリリースすることは、そのリード・タイムが長い（長すぎる）場合に問題となる可能性があります。多くの場合、ソフトウェア・リリース・サイクルの頻度が低下することになります。新機能を四半期ごとにリリースすることが許容されていた場合、リリース・プロセスは自動化されていないことが非常に多く、結果としてデータベース管理者（DBA）が夜間や週末に変更を本番環境へ適用していました。

リリース日の潜在的な問題とは別に、以下のような、新リリースに関連する典型的な問題を解決するため、時間がかかりエラーが生じる可能性のある多くの人的介入が必要でした。

- 長時間実行の多数の機能ブランチをリリース（またはメイン）ブランチにマージ
- ソフトウェア・リリースに必要な変更すべてを収集
- ソフトウェア・リリースの作成（特にデータベースを含む場合）
- 有意義なインテグレーション/承認/パフォーマンス・テストを稼働前に実行

開発者が長年にわたり、フロントエンド・アプリケーションなどのソフトウェア・リリースの自動化とバンドルにおいて大きく前進できた一方で、データベース・アプリケーションは同レベルの注目と自動化を享受していません。

多くの開発者は、通常はデータベースに属する機能をアプリケーションに移動することによって、認識した欠点をバックエンド・データストアで補おうとしてきました。これは満足できる解決策ではありません。なぜなら、以下のような問題が必然的に隠されたり、先延ばしにされたりする傾向があるからです。

- データ・ガバナンスを含む問題
- アプリケーションがデータの入力に使用されない場合にデータ品質に悪影響が及ぶ可能性
- 特にやり取りが頻繁なアプリケーションの場合、ロードの増加によってスケーラビリティに影響
- データベースによって提供されるアプリケーション機能のメンテナンス
- データベースによって提供される再開発された機能の、異なるアプリケーション間での複製

データベース・アプリケーションのデプロイの自動化は、多くのソフトウェア・プロジェクトにおいて依然として初期段階にあります。ほとんどの場合、データベースへの変更（以下、データベース・リリースと呼びます）は、バージョン管理リポジトリの外部で作成されます。最悪の場合、そのようなデータベース・リリースは、電子メール経由で共有されるZIPファイルに組み込まれた複数のスクリプトで構成され、停止時間期間中にデータベースに対して手動で実行される可能性があります。

ご想像のとおり、このプロセスは（時間が経つにつれて証明される可能性はありますが）、機能をより頻繁にリリースするための要件に合わせて十分に拡張されるわけではありません。また、どの変更がいつ行われたかのトレーサビリティなしに変更をデプロイするリスクも高まります。

アプリケーションのリリース・サイクルが速いという前提がある場合、自動化以外に現実的な代替手段はありません。歓迎すべき副次効果として、自動化によって、データベース・リリースを扱うデータベース専門家の認知的負担が軽減される可能性もあります。

## 最新のソフトウェア開発ワークフローに向けて

リリース頻度を向上させる必要がある場合は、マージ・フェーズ中に、関連する問題を含み、長期間存続する多数のブランチを特徴とするワークフローに従うのではなく、異なるアプローチが必要になります。

Nicole Forsgren氏らによる画期的な調査が、2018年の有名な著書『Accelerate-the science of lean devops and devops building and scaling high performance technology organizations』で公開されました。それ以来、何千もの個人の分析結果を含むState of DevOpsレポートが毎年公開されています。

鍵となるメッセージの1つは、パフォーマンスの高いチームは**より頻繁かつ迅速にコードをデプロイ**しており、同僚よりも**エラーが少ない**ということです。念頭に置くべき鍵となるメトリックは以下のようなものです。

- 変更までのリード・タイム
- 平均リカバリ時間
- デプロイメントの頻度
- 変更の失敗率
- 信頼性

これが注目すべきなのは、長い間、業界では頻繁なリリースは必ず品質の犠牲の上に成り立つと信じられていたからです。正しく実行された場合、これは当てはまらないことが実証済みです。次のセクションでは、コードの品質を向上させるためのツールと方法について説明します。

コードをより迅速にデプロイするために鍵となるコンポーネントは、前述の書籍では、**継続的インテグレーション/継続的デプロイメント（CI/CD）**と見なされています。

**継続的インテグレーション**は、Gitのようなバージョン管理システム（VCS）でのコミットごとにコード変更が既存のコードベースに対してテストされるプロセスです。

ソフトウェアの継続的インテグレーションの自動化は、以下の前提条件に基づきます。

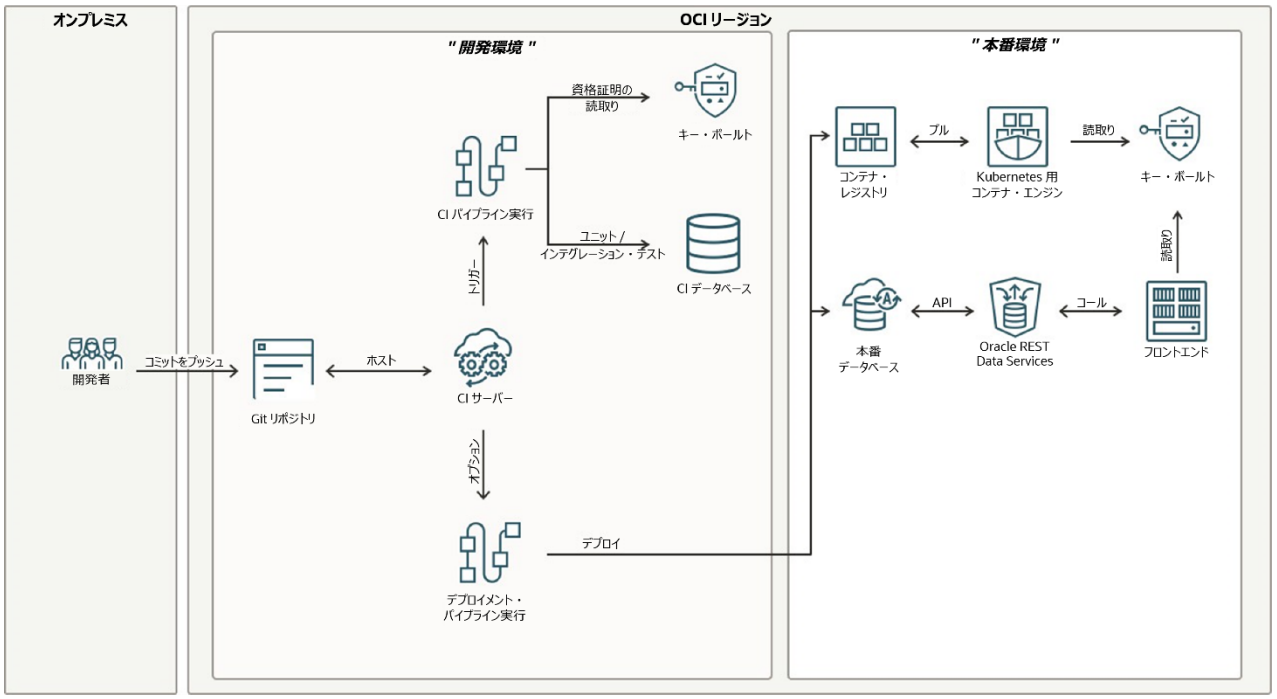
- すべてのコードがGitのようなツールを使用してバージョン管理されていること
- 小さな増分変更に焦点を当てること（“小規模なバッチでの作業”）
- 変更を何日間も何週間も機能ブランチに置いたままにするのではなく、メイン・ブランチへの統合を頻繁に実行して変更が正常にマージされるようにすること
- チームが規格に同意した後に、コード・カバレッジ、書式設定、および構文チェックを自動化すること
- テストがソフトウェア・プロジェクトの重要部分を占め、より高い層の環境へのデプロイ前に合格すること。該当するテストなしにコードを追加しないこと（“**テスト駆動開発**”）

興味深いことに、これらの要件には、技術的なものもあればそうでないものもあります。Gitのようなバージョン管理システム（VCS）ですべてを行う文化を採用するには、変更管理チームによる多大な努力が必要とされる場合があります。最終的には、VCSに基づく開発が前進のための最良の方法であることをチーム全員に確信させるように努力する価値があります。

## 継続的インテグレーションの実装

継続的インテグレーション（CI）によってチームをサポートするには、多数のツールを利用できます。アーキテクチャの中心となるのは、紛れもなくCIサーバーです。CIサーバーは、タスクの実行を調整し、ダッシュボードに結果を表示し、多くの場合、ソフトウェアの問題を管理して追跡するための方法を提供します。テストを実行する際、CIサーバーは通常、データベース変更の開発環境、ボールトやキーストアなどの資格証明支援、およびターゲット・プラットフォームとしてのKubernetesクラスタやContainer Engineランタイムなどの補助インフラストラクチャに協力を求めます。以下の図は、**Oracle Cloud Infrastructure（OCI）**でのセットアップを簡略化して説明したものです。

図1：CI/CDセットアップを大幅に簡略化した図




CIサーバーの中心要素は、CIパイプラインです。ソフトウェア・プロジェクトでは、多くの場合、パイプラインは、“リンティング”、“ビルド”、“テスト”、“ソフトウェア・リポジトリへのデプロイ”などの複数のステージで構成されます。各ステージはさらにタスクに細分されます。

“リンティング”ステージは通常、リンティング、コード・カバレッジ、およびセキュリティ脆弱性チェックで構成されます。“ビルド”フェーズには通常、アーティファクト/コンテナ・イメージなどの構築が含まれます。データベース・アプリケーションのコンテキストにおいて、ビルド・フェーズは多くの場合、テスト環境の作成、スキーマ変更のオート・デプロイ、およびユニットまたはインテグレーション・テストの実行で構成されます。

通常は、ソース・コード・リポジトリに新たにコミットすることで、パイプラインが実行されます。CIの中心的なパラダイムの1つが、“常にパイプラインを緑色に維持する”ためのルールです。これは、パイプラインのステータスがダッシュボード上の“信号”によって示されることを意味します。何らかの理由で構築が失敗すると（赤色ステータス）、開発者は急いでエラーを修正して他のワークフローが中断されないようにする必要があります。

多くの場合、CIパイプラインはYAML（GitHub、GitLabなど）や他のドメイン固有言語で定義されます。プロジェクト用に選択するCIサーバーでは、ソース・コードに沿ってパイプラインの（非バイナリ）表現を保存できるようにすることを推奨します。これで、プロジェクトのすべてがバージョン管理されるという原則に戻ります。

以下のスニペットは、GitLabで使用されているCIパイプラインの抜粋です。

```
# ------------------------------------------------------------------------------------------ global variables
variables:
   CLONE_PDB_NAME:  "t$CI_COMMIT_SHORT_SHA"
   SRC_PDB_NAME:    "SRCPDB"
   TAG_NAME:        "t$CI_COMMIT_SHORT_SHA"

# ------------------------------------------------------------------------------------------ stage definition
stages:
- linting
- build
- test
- update-ci-db
- deploy
```

```
# ---------------------------------------------------------------------------------------------- linting

# use the docker executor to get the latest node image and install
# eslint.All JavaScript source files that are part of the release
# will be linted as per .eslintrc

lint-mle-modules:
    image: node
    stage: linting
    script:
    -  npm install eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin
    -  npx eslint r*/migrations/*.ts
    tags:
    -  docker

# ... additional steps
```

## 継続的デリバリおよびデプロイメント

多くのソフトウェア・プロジェクトは、完全にテストされ、デプロイの準備が整ったアーティファクトを、インテグレーション・パイプラインですべてのテストに合格したのち、ユーザー受け入れテスト（UAT）や本番環境などの別の層に提供することで、継続的インテグレーションをさらに一歩進めています。再びパイプラインを使用しますが、今回はデプロイメント・パイプラインと呼びます。

多くの場合、**継続的デリバリと継続的デプロイメント（CD）**の間の線引きは、自動化の度合いに基づきます。**継続的デプロイメント**では、ビルドはすべてのチェックとテストに合格すると、そのまま本番環境に進むことが前提です。**継続的デリバリ**はこれに非常に似ていますが、実際のデプロイメントがパイプラインではなくオペレーターによって実行される点が異なります。継続的デリバリがない場合、継続的デプロイメントは実行できません。

継続的デプロイメントは、ステートレス・アプリケーションに対してさえも実装が難しい点に注意する必要があります。自信を持っておのおのの変更を本番環境に移行できるようにするには、多数のコミットメント、リソース、トレーニング、および堅牢なテスト・フレームワークが必要とされます。

## 継続的インテグレーションおよびデリバリの属性

CI/CDのいくつかの基本原理は、本書のコンテキストにおいて詳しく調査する価値があります。

### バージョン管理

前述のとおり、アプリケーションのテストおよびデプロイメントの自動化は、バージョン管理なしには実装できません。あらゆる種類の自動化の要件とは別に、バージョン管理システムを使用することには、他にも以下のような多数の利点があります。

- 複数の開発者による分散ワークフローを実現
- 強力な競合解消ツール
- アプリケーションの変更を過去にさかのぼってトレース可能
- 特定のデプロイメント/マイルストーン/コミットの前後のコードを比較
- 中央コード・リポジトリを使用するとコードが自動的にバックアップ

VCSシステムはインフラストラクチャにとって重要であり、フォルトトレラントな方法でセットアップする必要があります。VCSへのアクセスがダウンすると、自動化パイプラインの重要な部分に障害が生じ、本番環境に変更を加えることができなくなります。

第2章では、バージョン管理システムとしてのGitの使用についてさらに詳しく説明します。

## 高速フィードバック・ループ

ソフトウェアの長いリード・タイムに伴う問題の1つが、多くの場合、手遅れになるまでフィードバックが得られないことです。2つのチームが立て続けに何週間も別々のブランチでそれぞれの機能に取り組むケースを想像するのは難しくありません。これらの2つのブランチをリリース・ブランチに統合することになった場合、マージの競合が発生することは珍しいことではありません。時間が経過すると、おのおののブランチで発生した競合する変更を解決するのに長い時間がかかり、リリースが不必要に遅れる可能性があります。

問題が検出されるまでに何週間も費やすよりも、より頻繁にさまざまなブランチの変更を統合するほうが簡単でしょう。

高速フィードバック・ループは、パイプラインの実行に関して頻繁に言及されます。パイプラインの実行時間が長すぎると、開発者はプロセスに不満を募らせ、パイプラインの使用を避けるようになる可能性があります。これはコードの品質に深刻な影響を与えるため、CIにおいて決してあってはならないことです。そのため、DevOpsのエンジニアは、パイプラインの実行ランタイムを短時間に維持するようにしなければなりません。これは、特にテスト・データベースのプロビジョニングがパイプライン実行の一部である場合に課題となる可能性があります。

第4章では、Oracle Database環境をCIパイプラインに提供する方法に関するさまざまなオプションについて説明します。それにより、プロジェクトのCIパイプライン実行におけるリンティング、コード・カバレッジ、およびユニット/システム/インテグレーション・テストを可能な限り最短期間で完了させることができます。

## 自動テスト

コードを頻繁に統合することは、リリース頻度を高めるための1つのステップです。ただし、エラーなしでコードがマージ（およびコンパイル）されるかどうかをテストしただけでは、アプリケーションを信頼することはできません、デプロイされたアプリケーションをテストすることによってのみ、新しい機能が期待どおりに動作することを確認できます。**テスト駆動開発（TDD）**を採用している開発者は、アプリケーション・コード内の新しい機能について、それぞれ一連のテストを行う必要があることを認識しています。

## 小規模で頻繁な変更

バグが発生するリスクは、変更のサイズに比例する傾向があります。メイン・リリース・ブランチに何週間もマージされなかった大きな変更は、たとえば数時間以内にマージされた小規模な変更よりも、マージ競合を引き起こすリスクが大きくなります。

このコンテキストでのマージ競合は、チームが機能を提供できるかどうかに直接影響します。マージ競合が解決するまでは、機能を本稼働できません。しかし、CI/CDでの考えは、ソフトウェアをすぐにデプロイ可能な、常時準備が整った状態にすることです。

**トランクベース開発**などの新たな開発技術により、チームは小規模の増分変更をより安全に提供できるようになります。トランクベース開発では、頻繁に、場合によっては1日に複数回コミットすることが推奨されています。これにより、存続期間の長いブランチの作成とそれに関連する問題を回避することができます。

## 自動デプロイメント

データベース・プロジェクトには、たとえばJavaプロジェクトの場合のような単一のビルド・アーティファクトは存在しません。アプリケーションは、JARファイル、WARファイル、またはコンテナ・イメージにはビルドされません。データベース・プロジェクトでは、すべての"ビルド"にデータベースへのデプロイメントが含まれる可能性があります。

通常、データベース・プロジェクトでは少なくとも2回デプロイする必要があります。

1. 開発環境（のクローン）へのデプロイメント
2. 本番環境を含む、より高い層へのデプロイメント

CI/CDでプロジェクトを成功させるには、デプロイ先の環境にかかわらず、デプロイメント・メカニズムが同一である必要があります。デプロイメントも、安全な方法で繰り返し実行できなければなりません。言い換えると、新しいデプロイメントにおいてCIフェーズ時に検出されなかったエラーが発生しないように、環境は類似している（理想的には同一である）必要があります。大量の管理作業を手動で実行する環境では、開発環境へのデプロイメントに、UAT、統合、または本番環境とほとんど類似点がないという大きなリスクがあります。

クラウドでは、Infrastructure as Code（IaC）を用いたサポートが提供されます。TerraformやAnsibleなどのツールを使用することで、同一環境の作成や維持が大幅に簡略化されます。クラウド・バックアップは迅速にリストアでき、データベース・クローンのプロビジョニングに必要となる時間を短縮できます。

第3章では、データベース・スキーマ移行の反復可能でべき等性のあるデプロイメントを作成する方法について説明します。

## まとめ

データベース中心の多数のアプリケーション・プロジェクトは、JavaのJARファイルや他の言語の実行可能ファイルなど、アーティファクトを生成するにすぎないステートレス・ソフトウェア・プロジェクトとは別の方法で処理されます。本書は、自動化のメリット、ソース・コード管理、および最新の開発技術を紹介することで、この状況を変えることを目標としています。以下の章では、バージョン管理システムの使用から、CI/CDパイプラインでの移行ツールの使用によるスキーマ変更のデプロイまで、さまざまな側面を詳細に説明します。

# バージョン管理システムの使用

このセクションでは、**バージョン管理システム（VCS）**、特に本書の執筆時点でもっとも一般的なバージョン管理システムであるGitについて取り上げます。Gitに言及しているとしても、その特定のテクノロジーが一般に推奨されているとは解釈しないでください。本書で説明する原則は、すべてのバージョン管理システムに適用できます。

## バージョン管理システムの使用なくしてCI/CDは存在せず

前の章で、**CI（継続的インテグレーション）**サーバーが自動化アーキテクチャの中心部分であり、パイプライン経由でスクリプト、テスト、および他のすべての操作の実行を調整していることを説明しました。ほとんどすべてのCIサーバーで、ソース・コードをGitリポジトリに提供することが期待されます。

バージョン管理システムにコードを格納しないと、*CI/CD*パイプラインを開発できません。

プロジェクトのソース・コードをバージョン管理下に置くことが、開発プロセスおよび潜在的なデプロイメント・プロセスを自動化するための最初のステップです。この章では、利用可能なさまざまなオプションについて説明します。

## バージョン管理システムの使用についてチームの賛同を得る

従来、（フロントエンドの）開発者とデータベース管理者（DBA）は、別々のチームに属していました。長期間、この分離がうまく機能してきた一方で、アプローチは最新の開発モデルに適さなくなっています。実際のところ、それはここしばらくのことではありません。

開発者がアプリケーション・コードをおなじみのフェンスの向こうに投げ込むのではなく、より良い方法を使用することが可能であり、使用するべきです。**DevOps**運動では、開発者（DevOpsにおける“dev”）と運用（“ops”）間の連携を活用します。DevOpsは技術というよりは文化における変化ですが、新しいスタイルの連携を導入するには、通常、以前は手動で行っていたプロセスを自動化する必要があります。ここで、**CI**が再び役割を果たします。

DevOpsの文化の導入は困難な可能性があり、管理層は関係者全員から十分な同意を得る必要があります。さもないと、近代化プロジェクトが早い段階で必要以上に深刻な困難に直面する可能性があります。

バージョン管理システムを導入するには、プロジェクトに取り組む全員がVCSリポジトリでのコントリビューションを共有する必要があります。これによって個人のコントリビューションの可視化を強化できますが、誰もが満足できるわけではありません。VCSの操作に消極的なプロジェクト・メンバーには、以下のようなVCSのメリット（これらに限定されるものではありません）を指摘することによって納得させられる可能性があります。

- リポジトリに追加された時点からファイルの履歴を記録
- 以前の既知の状態に戻すことが可能
- 各コード変更の可視化
- 分散ワークフローの実現
- 強力な競合解消
- 中央リポジトリを使用した場合にデータ損失から保護
- コンピュータまたはネットワーク・ファイル共有にローカルにファイルを格納するよりも開発者のエクスペリエンスが大幅に向上

**CI/CDパイプライン**の使用を推進するすべてのチームにとって、実際のところVCSを使用する以外の選択肢はありません。このメッセージは、従事するチーム・メンバーの懸念を考慮し、VCSに不慣れなメンバーにサポート、トレーニング、メンタリングを提供して穏やかに伝えると最適です。全員が新しい作業方法に慣れるまで、問合せ先の中心として機能する“VCSチャンピオン”を任命することもできます。

## Gitの概要

Gitは、実績のある分散型バージョン管理システムです。Gitのおもな目的は、開発者たちがソフトウェア・プロジェクトに同時に取り組める

ようにすることです。Gitは、多数の異なるワークフローや手法をサポートしており、重要なこととしては、チームが経時的な変更を追跡できるようにします。また、非常に効率的なのは、コミットで変更されたファイルのみを格納する点です。Gitは、テキスト・ファイルで最適に動作します。バイナリ・ファイルは、Gitに格納するにはあまり適しません。またGitは、Java JARファイルやコンパイル結果（通常はビルド・アーティファクトと呼ばれます）など、ビルドの一部として作成されたものを格納しないという業界でもっとも有名な方法でもあります。

Gitには、優れた開発者エクスペリエンスに貢献する多数の機能があります。コマンドライン・インタフェースに加えて、あらゆるおもな統合開発環境（IDE）によってGitは標準でサポートされます。

## Gitの用語

プロジェクトのファイルは、（ソフトウェア）リポジトリの一部です。プロジェクトのファイル以外に、リポジトリにはGitの正しい動作に必要なメタ情報が含まれます。これには、無視すべきファイルのリスト、ローカル・ブランチ情報といった内部データや構成情報などがあります。

絶対にパスワードやセキュア・トークンなどの機密情報を*Git*に**コミットしないでください**。

Gitは、ローカル・リポジトリおよびリモート・リポジトリを操作できます。ローカル・リポジトリは通常、一元的にホストされたGitLabやGitHubなどのGitサーバー、または組織内の他のシステムからクローニングされます。開発者は、ローカル・コピーを使用して変更を行ってから、中央（“リモート”）リポジトリにアップロード（“プッシュ”）します。

リモート・リポジトリがまだ存在しない場合、管理者は最初にリモート・リポジトリを作成してから、クローンとコントリビュートに必要な権限をチーム・メンバーに付与する必要があります。

リポジトリのあらゆるファイルは、ブランチに関連付けられます。ブランチの名前は任意です。ほとんどのプロジェクトでは、独自の命名体系に従います。慣例的に、*MAIN*は安定したブランチと見なされますが、その名前を強制するルールはありません。ブランチ処理はGitの中核的な機能の1つですが、過度なブランチ処理は問題の原因となります。詳しい説明は以下を参照してください。

リポジトリに新しく追加されたファイルは、*untracked*（未追跡）ファイルとして開始されます。まだ編集されていない既存ファイルは、*unmodified*（未修正）と呼ばれます。ディレクトリのファイルへの変更が、リポジトリに自動的に保存されるわけではありません。新しいファイルは最初にリポジトリに追加する必要がある一方、変更されたファイルはそれらのファイルをステージング領域へ追加することによってステージングする必要があります。ファイルは追加またはステージングされたら、プロジェクトへコミットできます。

コミットが完了したら、リポジトリにコミットされたすべてのファイルのステータスはunmodifiedに設定されます。

コミットのたびに、コミット・メッセージを追加する必要があります。メッセージは、重要なコミット属性です。理想的には、メッセージは変更の性質を可能な限り詳細に伝えます。有益なコミット・メッセージの例は次のとおりです。

(issue #51): extend column length of t1.c1 to 50 characters

適切なコミット・メッセージは、コミット履歴を理解するのに大いに役立ちます。

## ブランチとマージ

ブランチとマージは、旧世代のVCSではリソースと時間の両方を大量に消費するプロセスで、もはやGitの関心領域ではありません。

ただし、最近の調査では、広範囲のブランチは、解決が困難で時間を消費するマージ競合の原因となる可能性があり、大幅な遅延が生じる場合があることが指摘されています。これは変更のより高速なリリースを妨げるものとなります。小規模な増分変更では、組織ははるかに頻繁にリリースを行うことができます。極端に言えば、一部のモデルでは、開発者がコードを送信する単一のブランチまたはトランクの使用を提案しています。このアプローチは、トランクベース開発として知られています。

機能の準備が整うと、開発者は通常、**プル・リクエスト（PR）/マージ・リクエスト（MR）**を作成し、機能をMAINブランチにマージします。

## プル・リクエストとマージ・リクエスト

もともと、プル・リクエスト（GitHub）およびマージ・リクエスト（GitLab）はGitの一部ではありませんでした。これらはホスト型開発プラットフォームの一部として導入され、広く採用されています。

それらの目標は、ある特定のブランチの保守者に新しい機能やホットフィックスの送信を通知することです。

通常、MR/PRに関連するメタデータには、問題の説明、コラボレーション・ツールへのリンク、他のさまざまなワークフローの詳細が含まれます。

もっとも重要なのは、ソース・ブランチからのすべてのコミットと、その中の変更されたすべてのファイルが列挙されるため、全員が影響を評価し、変更についてコメントできるようになることです。

多くの場合、特にオープンソース・プロジェクトでは、**コード・レビュー**はマージ・リクエストに基づいて実行されます（以下を参照）。

## プロジェクトのフォーク

フォークは、オープンソース・ソフトウェア（OSS）に比べると、自社ソフトウェア開発プロジェクトではそれほど一般的ではありません。OSSでは、コードへのコントリビューションが推奨されますが、明らかな理由により、これらのコントリビューションをマージする前に十分な精査を行う必要があります。

言い換えると、一般ユーザーには、GitHubまたはGitLabでホストされるパブリック・ソフトウェア・プロジェクトへの書込み権限はありません。この制限を回避するため、プロジェクトへコントリビュートしようとする開発者は、自身の名前空間でコピー、つまりフォークを作成し、自分のものであるかのようにそれを修正します。変更を元のプロジェクトに戻して統合するには、コントリビューションの準備が整ってから、プル・リクエスト（GitHub）、またはマージ・リクエスト（GitLab）を生成します。

プロジェクト保守者はコントリビューションを確認し、それらをマージするか、またはさらなる変更または拡張をリクエストすることができます。コントリビューションがマージされたら、コントリビューターのフォークは冗長化されて、アーカイブ、削除、または将来のコントリビューションのために元のプロジェクト・リポジトリと同期できます。

## データベース・プロジェクトのGitリポジトリ

ソフトウェア開発ライフサイクルのあらゆる側面と同様に、実装を開始する前に将来についてしばらく考えることが効果的です。プロジェクトのライフスパンの初期に誤りを犯すと、後で解決するのにコストがかかり、複雑になる可能性があります。

### 単一リポジトリと別個のフロントエンド/バックエンド・リポジトリ

開発者コミュニティでは、Angular、React、その他のフロントエンド・テクノロジーなどのアプリケーション・コードを単一リポジトリ内でデータベース変更コードと共存させるべきかどうかについて議論されています。最新のアプリケーション、特にマイクロサービス・パターンに従うアプリケーションでは、フロントエンドおよびバックエンド・コードを同一のリポジトリに含めることは非常に意味があります。

既存の複雑なソフトウェア・プロジェクト、特に多数のアプリケーションがデータベースにアクセスするソフトウェア・プロジェクトでは、個別のGitリポジトリを作成することが合理的かもしれません。アプリケーション・リポジトリとデータベース・リポジトリが別々だと、リリース頻度に遅れが生じるリスクがあります。アプリケーションが必要とする変更が時間内に組み込まれずに、遅延が発生する可能性があります。開発ワークフローでは、サイロ化したチーム構造が存在したり、引き継がれたりすることのないようにする必要があります。

この技術概要は、開発者がフロントエンドとバックエンドの両方を所有しており、そのためユーザー・インタフェースとデータベース・スキーマの両方の変更を単一のリポジトリに組み合わせていることを前提に書かれました。

### ディレクトリ構造

データベース・プロジェクトのディレクトリ構造では、ファイル・レイアウトの選択がとても重要です。これは、おもに変更のデプロイに使用される方法によります。

- 移行ベースのアプローチ（“delta”または“increment”メソッド）
- 状態ベースのアプローチ（“snapshot”メソッド）

**移行ベースのアプローチ**を使用した場合、開発者はデータベース・スキーマの想定される状態に基づいて変更をデプロイします。t0の時点で変更がスキーマの表をデプロイしたと仮定すると、続くデータベース変更はALTER TABLE文を使用して表を変更します。スキーマ移行は継続的プロセスであり、1つの変更は以前の変更に依存します。

**状態ベースのアプローチ**を使用した場合、開発者は表の構造などのターゲットの状態を宣言します。デプロイメント・ツールは、表の現在の状態を評価し、現在の状態からターゲットの状態に移行する一連の変更をオンザフライで作成します。

この技術概要では、両方を組み合わせた方法を推奨します。ただし、状態は参照用にのみ保存されるか、または空のスキーマを移入するために使用されます。

**ステートフル・アプリケーションでは、すべてのリリースは移行**であることに留意してください。リリースの一部として、表などの既存のスキーマ・オブジェクトは変更されます。たとえば、新しい列を追加したり、列の定義を変更したりする必要があるかもしれません。これらの操作は、ALTER … SQL文を使用して実行します。

第3章では、スクリプトが不変であるという前提に基づいて、多数のスキーマ移行ツールによってコードがデプロイされることを説明します。これは、ソース・コードが新しい要件に合うように編集される他の多くのソフトウェア・プロジェクトとは異なります。スキーマ移行ツールを使用して、通常はスキーマ・オブジェクトの既存の状態を変更するための新しい追加のスクリプトを作成します。変更は引き継がれ、既存のスクリプトが変更されることはありません。

以下のディレクトリ構造では、移行ベースと状態ベースのアプローチの組合せが可能です。フロントエンドとデータベース変更を念頭に置いて、単一リポジトリで作成されています。フロントエンド言語やフレームワーク（Spring、Flask、React、Angular、Vueなど）によって非常に多数の異なるディレクトリ構造が可能であるため、この技術概要ではデータベース部分のみに焦点を当てます。これは、プロジェクトのルートにあるsrc/databaseディレクトリに格納されます。

```
$ tree src/database
src/database/
├── controller.xml
├── r1
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
├── r2
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
├── r3
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
└── utils
    └── newRelease.sh
```

データベース・リリースごとに1つのディレクトリが存在します（例ではr1からr3）。データベース変更スクリプトは不変で、新しい要件は新しいリリース・フォルダを使用して実装されるという以前の説明を思い出してください。上記のリリース・フォルダの名前は単なる例です。データベース・リリースを明確かつ一意に識別できる限り、任意のネーミング規則を選択できます。

データベースの変更は、指定されたデータベース・リリース・フォルダの下のmigrationsフォルダ内のファイルとして保存されます。Liquibaseデプロイメントの場合で、明示的な順序が定義されていないとき、これらのファイルはアルファベット順に実行されます。

stateディレクトリには、スキーマ移行の成功時に見られるのとまったく同じように、スキーマ構造を生成するためのあらゆるオブジェクトのスキーマDDLが含まれます。このディレクトリは、参照用およびトラブルシューティングに使用できます。スキーマ変更のデプロイには**使用しないでください**。

場合によっては、ユニット・テストでテスト・データを使用する必要があります。このテスト・データセットは、必要に応じてtestdataディレクトリに格納できます。

オプションのutilsディレクトリは、リリースの一部としてデータベースに適用する必要がないユーティリティ機能を格納します。このディレクトリには通常、デプロイメント前またはデプロイメント後（あるいはその両方）の検証を実行するためのスクリプトが含まれます。このようなスクリプトは、データベース・スキーマへの変更を**実行できません**。

データベース・リリースがデータベースへ適用され次第、そのmigrationフォルダのファイルへの変更は行われなくなります。新しいリリースとは、以前のリリースからプロセスを再開し、一連の変更をスキーマに適用するものとして考えてください。

src/databaseの下にあるオプションのutilsディレクトリは、新しいデータベース・リリースの作成や検証などのプロセスを簡素化するために使用できる他の一般的なユーティリティ機能を格納するために使用できますが、スキーマ変更デプロイメントは**実行しません**。

newRelease.shという名前のヘルパー・スクリプトは、Liquibaseによって使用されるメイン・アプリケーションの変更ログ（このトピックについては後で詳述）へのリリースの追加など、必要なフォルダ構造や他の基盤となる作業を作成します。

## リリース管理

前のセクションで説明したとおり、競合を避けるために、各データベース・リリースはリポジトリの中の専用のサブディレクトリで維持されます。あるリリースから別のリリースへアプリケーションを移行するには、移行スクリプトを実行するだけで済みます。リリースrN-1からrNへの飛躍に限定されるわけではありません。たとえば、より早くrN-5で開始することもできます。前述したスキーマ移行ツールによって、まだデータベースに適用されていないスクリプトのみがデータベース・リリースの一部として適用されます（=移行）。

アプリケーション（スキーマ）が"正しい"状態にあるかどうかの判断は、このアプローチを使用すると難しい場合があります。移行が成功したかどうかをリリース管理によって確認したい場合、スキーマがどうあるべきかを正確に知る必要があります。これは、移行ベースのアプローチでは難しい可能性があります。なぜなら、少なくとも1つ、またはそれ以上のリリースについて、どの変更がスキーマ・オブジェクトに加えられたのかを特定する必要があるためです。

比較すべき参照を保存することで、そのタスクが簡素化されます。そこで、stateディレクトリが役立つ可能性があります。適した状態の各スキーマ・オブジェクトのDDL文が含まれるためです。これらのDDL文を使用すると、データベース内のスキーマ・オブジェクトを、あるべき状態と簡単に比較できます。

## 継続的インテグレーション・パイプラインとの統合

コミットがリモート・リポジトリにプッシュされた後に、プロジェクトのCIパイプラインの実行をトリガーするのは一般的なプラクティスです。したがって、適切なレベルの信頼性で動作すると予測されるコードのみをプッシュすることが非常に重要です。ローカルでのテストおよびコミット・フックは、その信頼性の構築に役立つ可能性があります。何らかの事情でパイプラインの実行が中断される場合、全員参加の状況が生じ、全員が最優先でパイプラインの修正に取り組む必要があります。

CIパイプラインの詳細については、本書の後半のセクションで説明します。

## まとめ

Gitは、もっとも一般的に使用されているバージョン管理システムです。Gitの動作方法を理解し、Gitを使用するために全員から同意を得ることが、最新のソフトウェア開発アーキテクチャを採用するために重要な最初のステップです。データベース・コードとフロントエンド・コードに別々のリポジトリを使用するかどうかは、プロジェクトと周囲の状況に完全に依存します。

データベース・プロジェクトの"正しい"ディレクトリ・レイアウトを選択することが、プロジェクトが軌道に乗った際に功を奏します。スキーマ変更は、自社開発ソリューションを維持するコストを回避するため、Liquibaseなどの専用の商用既製品（COTS）ツールを使用して適用する必要があります。

## べき等性のある反復可能なスキーマ移行の確保

これまでの章では、開発プロセスの一部としてバージョン管理システムを使用することの必要性に大きく重点を置きました。そこでは、スキーマ移行スクリプトを作成する際に使用するフォーマットの詳細についてはあえて取り上げませんでした。これについては、このセクションで扱います。

すべてのデータベース・リリースはスキーマ移行であるという第2章の内容を思い出してください。他のソフトウェア開発プロジェクトとは異なり、本章で説明する理由により、既存のデータベース・コードに触れることはできません。これはおそらく、ステートレス・アプリケーション開発とデータベース・プロジェクトの間のただ1つの最大の違いです。本章では、その理由と、この状況に対処する方法について説明します。

以下のようなディレクトリ構造を使用すると、データベース・リリースにとって有益であることが分かります。これは、アプリケーションのフロントエンド・コードとバックエンド・コードの両方で、単一のGitリポジトリが使用されるという事実に基づいています。プロジェクトのデータベースの部分は、プロジェクトのルート・ディレクトリの下のsrc/databaseにあります。これは推奨にすぎず、データベース部分が自己完結型である限り、逸脱する可能性があります。

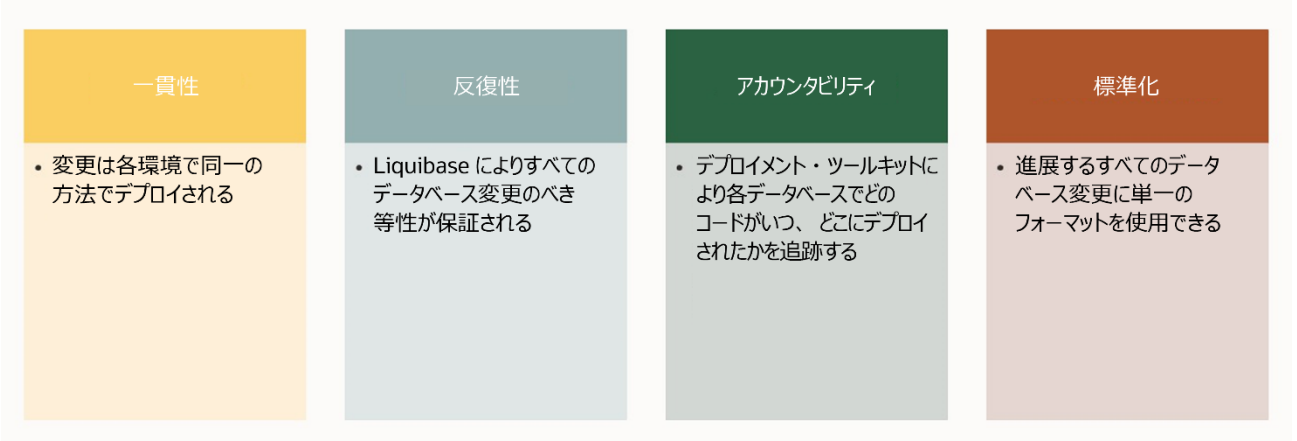
```
$ tree src/database/
src/database/
├── controller.xml
├── r1
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
├── r2
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
├── r3
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
└── utils
    └── newRelease.sh
```

最初のソフトウェア・リリースはr1ディレクトリにあります。最初のリリースがデプロイされたら、その後のデータベース・スキーマへの変更はすべて、上記の例のr2のような新しいリリース・フォルダに提供される必要があります。

## 専用のスキーマ移行ツールを使用する利点

チームの多くが、LiquibaseやFlywayなどのスキーマ移行ツールや、社内で構築された同等のスキーマ移行のデプロイ方法を使用しています。たとえば、Liquibaseには次のような利点があります。

図2：Liquibaseを使用してスキーマ変更をデプロイする利点




既製ツールを使用するほうが、社内開発ソフトウェアを使用するよりも一般的には望ましいとされます。あらゆる環境において、カスタム・ソリューションを最新に維持するために必要なメンテナンス作業は、通常は価値を付加せず、貴重な開発リソースを実際の製品の提供から引き抜く可能性があります。

## SQLclおよびLiquibaseの使用によるスキーマ移行のデプロイ

Oracle SQL Developerコマンドライン（**SQLcl**）は、Oracle Database用の無償のコマンドライン・インタフェースです。SQL、PL/SQL、およびJavaScriptをインタラクティブに、または一括で実行できます。SQLclにはインライン編集、文の完了、コマンドの再呼出しなどの豊富な機能が備わっているのと同時に、以前に記述されたSQL*Plusスクリプトもサポートします。

Liquibaseは、データベース・スキーマ変更の追跡、管理、および適用のための、データベースに依存しないオープンソースのライブラリです。

これら2つのテクノロジーを組み合わせることで、CI/CDパイプライン内でデータベース移行スクリプトをデプロイするための優れた方法が提供されます。チェックサムおよび他のメタデータを使用することで、LiquibaseやFlywayなどの他の同等のツールは、データベースに対してどのスクリプトが実行されているかを特定でき、有害となる可能性のある不必要な再デプロイメントを回避できます。

おそらくSQLclの最大の利点は、その低いストレージ・フットプリント、豊富な最新のコマンドライン機能、および組込みのLiquibase機能です。この緊密な統合は、特にOracle Autonomous Databaseなどの相互TLS暗号化を有効化したシステムを対象にしている場合、生産性を大幅に向上させます。

SQLclにおけるLiquibaseの統合については、メインのSQLclのドキュメントに記載されています。これは、拡張データタイプ、エディションベースの再定義（EBR）などのOracle Databaseの高度な機能をLiquibaseに認識させるためにOracleによって拡張された特別なものであり、Web上にあるLiquibaseオープンソース・エディションとは異なります。Oracle Databaseを操作する場合は、Liquibaseオープンソース・エディションではなく、SQLcl内の組込みLiquibaseサポートを使用することを強くお勧めします。

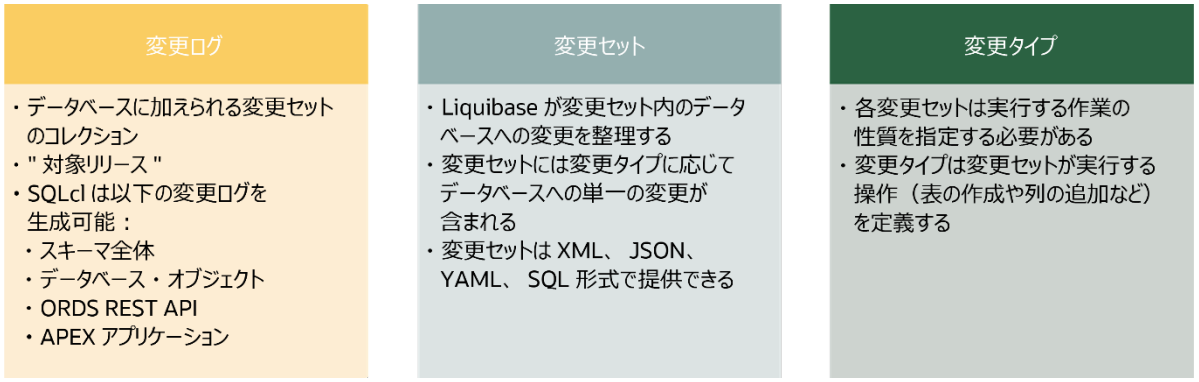
## Liquibaseの用語と基本概念

Liquibaseでの作業を開始する前に、まず基本概念を理解する必要があります。これには以下が含まれます。

- 変更ログ
- 変更セット
- 変更タイプ

以下の図では、これらについてそれぞれ詳細に説明しています。全体の詳細については、Liquibaseの公式ドキュメントを参照してください。

図3：Liquibaseの変更ログ、変更セット、および変更タイプの説明




変更セットは、開発者が新しいデータベース・リリースの一部として作成する基本のエンティティです。複数の変更セットは、データベースの変更ログと呼ばれます。変更セットには、1つまたは複数の変更タイプが含まれます。デプロイメント時の複雑さを避けるため、ベスト・プラクティスとして、変更セットあたり1つのタイプの変更に制限してください、

上記の図にあるとおり、変更セットは多数の形式で提供されます。SQLclは、リバースエンジニアリング・スキーマ・オブジェクトの場合、XML形式にデフォルト設定されます。これは、小規模なプロジェクトの場合や、SQLclとLiquibaseから開始した場合に最適です。XML形式を使用することの欠点は、CDATAタグに埋め込まれた実際のコマンドを読み取るためにほとんどのリンターを使用できないことです。プロジェクトにとってコードのリンティングが重要な場合は、代わりにSQL形式の使用を考慮する必要があります。また、SQL形式の使用は、Liquibaseの使用へ移行するためのもっとも簡単な方法になる場合があります。それは、既存のSQLスクリプトにLiquibase固有の注釈を付加すれば済むからです。

## データベース変更ログ作成の実用的な側面

開発者は通常、表の作成、表への列の追加、JavaScriptやPL/SQLでのデータベース・コードのデプロイなど、変更セットを含む複数のファイルを作成します。そして、これらの変更セットは、デプロイメントを簡素化するために変更ログに含まれます。変更ログは、“リリース”とも呼ばれます。

それはプロジェクトではどのようなものでしょうか。新しい列を既存の表へ追加する必要があり、追加される列は外部キーであると仮定します。変更ログは、次の3つの変更セットで構成されています。

1. 表への列の追加
2. 新しい列に対応する索引の作成
3. 親表への新しい列の外部キーの追加

開発者は通常、上記のそれぞれの変更セットに別個のファイルを作成します。デプロイメントを簡素化するため、追加のファイルが頻繁に作成されます。最初の3つのファイルとは対照的に、後者はメイン変更ログまたはリリース変更ログと呼ばれることが多く、スキーマ・オブジェクトを作成したり、それらとやり取りしたりするためのデータベース固有のコマンドは含まれません。このファイルは、includeディレクティブまたはincludeAllディレクティブを使用して、Liquibaseによってデプロイされるファイルをすべて列挙します。これらの詳細については、本章の後半で説明します。

## 変更ログの書式設定

Liquibaseで使用するためには、変更セットに注釈を付ける必要があります。少なくとも、著者とIDを関連付ける必要があります。ID、著者、およびファイル自体のファイル・パスの組合せによって、変更セットが一意に識別されます。これは、変更セットがデータベースに対してすでに実行されているかどうかを判断する場合に重要です。

Oracle Databaseでは、PL/SQLコードは通常、プログラムの最後にスラッシュで終了しますが、これはLiquibaseに伝える必要があります。追加属性としてendDelimiter:/を加えることで、ツールに認識させることができます。

そうしないと、PL/SQLコードのセミコロンはすべて文の終了記号として機能し、エラーの原因となります。さらに、Liquibaseでは、コメントはすべて変更セットからデフォルトで削除されます。これは望ましくない場合がほとんどです。この問題は、stripComments:false属性を加えることで解決できます。

以下の例では、SQLとして提供される変更セットにLiquibase属性を使用する方法を示しています。

```
--liquibase formatted sql
--changeset developer1:"r1-01" failOnError:true labels:r1

CREATE TABLE sessions (
    -- this is a UUID
    session_id char(32) not null,
        constraint pk_session_hit_counter
            primary key (session_id),
    browser                 varchar2(100),
    operating_system        varchar2(100),
    duration                interval day to second
);
```

この例に示すとおり、SQLファイルは、Liquibaseプリアンブル（--liquibase formatted sql)）を記載する必要があります。これはLiquibaseに、ファイル形式が（XMLやJSONではなく）SQLであり、変更セット情報自体（--changeset author:ID、たとえばdeveloper1:"r1-01"）が続くことを伝えます。変更セットのフラグを追加できますが、中でもfailOnErrorとlabelsはもっとも重要です。（SQLclのwhenever sqlerror exitコマンドと一緒に）failOnErrorをTRUEに設定することにより、エラーが発生し次第、パイプラインの実行を確実に停止できます。リリース名に対応するラベルを追加することにより、コードの実行時にラベル・フィルタを提供して、特定のラベルを含む変更セットのみを実行できます。

次に、エラーの場合にパイプラインの実行を確実に停止する方法についてさらに説明します。

PL/SQLの変更セットは、変更セットの追加の属性が必要とされる場合やそれが望ましい場合でも、同じ行に沿って記述することができます。この例では、utPLSQLに基づいたユニット・テストを示しています。utPLSQLでの処理ロジックは、コメントに大きく基づいています。（注：例の中の"↵"文字は、ファイルそのものの中ではテキストは同じ行で続いていますが、ページ幅が限られているため、以下の例では別の行に折り返されていることを示しています。）

```
--liquibase formatted sql
--changeset developer2:"r1-09" failOnError:true endDelimiter:/ stripComments:false ↵
labels:r1

create or replace package my_test_pkg as

    --%suite(unit tests for my application)

    --%test(verify that something is done right)
    --%tags(basic)
    procedure my_test_001;

end my_test_pkg;
/
```

ここでは、同じauthor:IDの組合せと、前の例のようなlabel属性が示されています。さらに、PL/SQLコード内のどのコメントも削除されないようにするために、stripComments属性がfalseに設定されています。そうしないと、utPLSQLユニット・テストが中断されます。さらに、SQLコマンドはendDelimiter属性の指示に従って/文字で終了するように定義されています。そのようにしておかないと、LiquibaseはPL/SQLパッケージ内に最初に;文字が出現した時点でSQLコマンドを終了するからです。

## 変更ログのデプロイ

コード変更は、SQLclでlb updateコマンドを使用してデプロイされます。それから、Liquibaseは、提供された変更ログのすべての変更セットを順番に読み取ります。

特定の変更セットを適用する前に、変更ログ表に対するメタデータの問合せを実行します。スクリプトが以前に実行されている場合は、スキップします（runAlways属性が設定されている場合を除きます。これは**推奨されていません**）。

Liquibaseのデプロイメント・ロジックは、最大限に使用できます。“メイン”変更ログは、プロジェクトに対して1回定義できます。これをすべてのファイル変更によって最新に維持するのではなく、以下の例に示すようにincludeAllディレクティブを使用できます。

```
<?xml version="1.0"encoding"UTF-8"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.liquibase.org/xml/ns/dbchangelog
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-latest.xsd">
  <includeAll path="r1/migrations"/>
  <includeAll path="r2/migrations"/>
</databaseChangeLog>
```

それぞれの新しいデータベース・リリース（r*N*）は、新しいincludeAllディレクティブを使用して追加されます。指示されたディレクトリにあるLiquibase変更セットはすべて、アルファベット順に実行されます。たとえば、ファイルが正しい順序で実行されるように、番号の接頭辞を付けてディレクトリにファイルを置くようにしてください。

前の例（外部キー列を表に追加）に戻ると、開発者はr*N*/migrationsに以下のファイルを作成しています。

- 01_*tablename*_add_column_*columnname*.sql
- 02_*tablename*_add_index_*indexname*.sql
- 03_*tablename*_add_foreign_key_*fkname*.sql

このようにすると、実行の順序が同一で、正しい順序であることが保証されます。r*N*/migrations/*内のすべての変更が、メイン変更ログに追加されたディレクトリを参照するエントリで実行されることを確認します。

Liquibaseにより実行されるメタデータ・チェックによって、ベースラインおよびmigrationsフォルダにある、すでに実行された変更セットはいずれも自動的にスキップされます。条件ロジックは必要ありません。メイン変更ログ・ファイル自体には変更セットが含まれず、SQLコマンド自体も含まれないため、XMLやサポートされる別の形式を使用しても構いません。

最初に実行されるメタデータ問合せに加えて、壊れたリリースを阻止する別のセーフティ・ネットが存在します。ファイルが改ざんされていないことを確認するために、MD5チェックサムが各ファイルを対象に維持されます。チェックサムは、Liquibaseのメタデータ・カタログの一部として保存されます。次にlb update が試行されると、変更が通知され、Liquibaseがその実行を中断します。

デプロイメント中に変更されたソースによって引き起こされる問題を防ぐために、変更ログを実行する前に検証することができます。lb validateコマンドは変更ログを検証し、チェックサムが変更されている場合は警告します。

ファイルを不変に保つという要件があるため、コードを修正して適切な場所に配置できないとすると、「コード・ユニットのバグのような問題はどうすれば修正できるだろうか」という興味深い疑問が生じます。Liquibaseの哲学に従えば、このようなケースではフォワード・フィックスが必要です。言い換えれば、バグを修正するために必要な修正が加えられた新しいリリースを作成することです。ソース・コード・ファイルのバグを削除して再デプロイすることは、推奨されるオプションではありません。

さらに、Liquibaseでの変更をロールバックすることもできますが、この機能はこの技術概要の範囲外です。

## デプロイメントのステータス確認

Liquibaseのデプロイメント時に使用されるメタデータは、誰もが使用できます。ターゲット・スキーマは、Liquibaseによって作成される複数の表から成ります。

- DATABASECHANGELOGLOCK

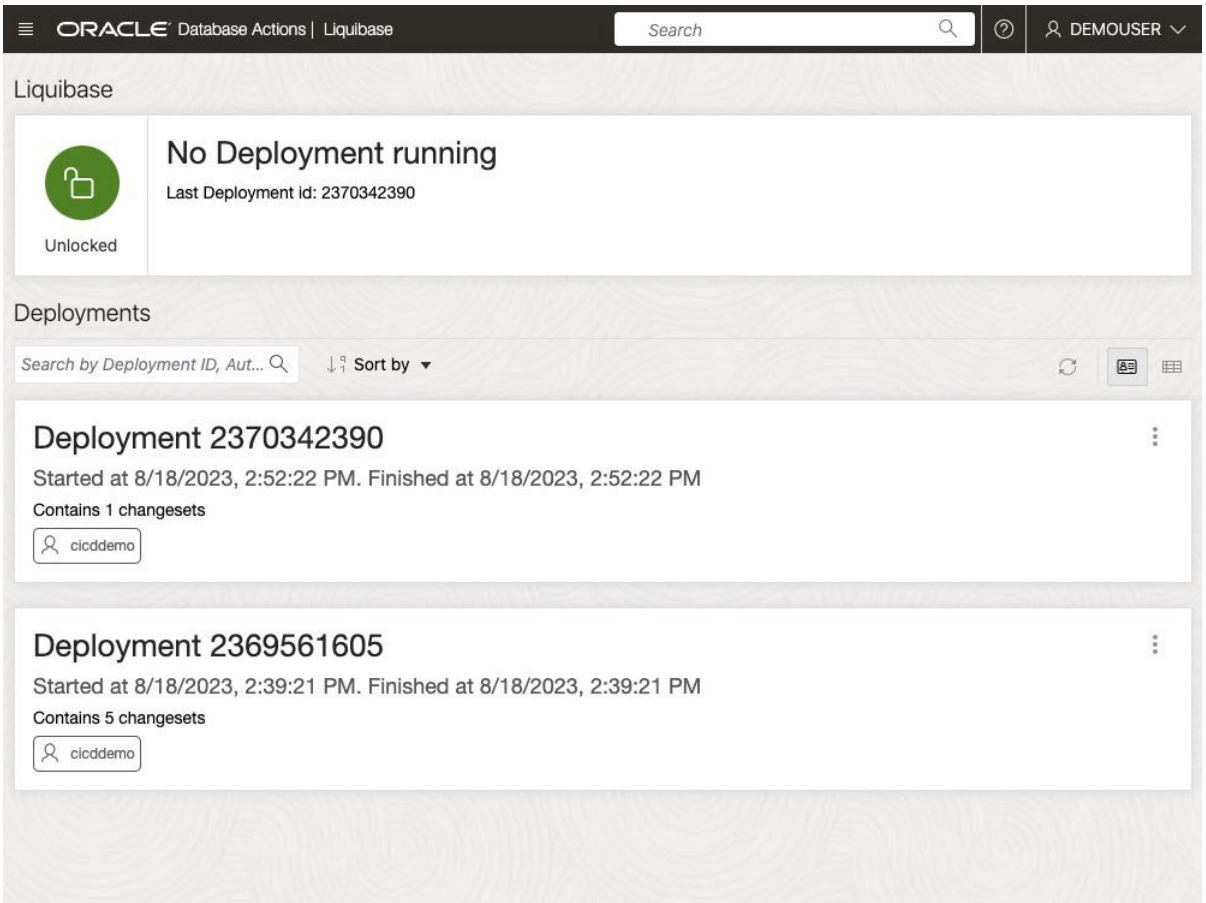
- DATABASECHANGELOG_ACTIONS
- DATABASECHANGELOG

lb historyコマンドは、これらの表の内容をリリース管理に公開します。次に、データベースに対する実際のデプロイメントの例を示します。

```
SQL> lb history
--Starting Liquibase at 14:39:27 ↵
(version 4.17.0 #0 built at 2022-11-02 21:48+0000)

Liquibase History for
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=cidb.test.oraclevcn.com)(PORT=152
1))(CONNECT_DATA=(SERVICE_NAME=tcdae8c8)))
-  Database updated at 8/18/23, 2:39 PM.Applied 5 changeset(s) in 0.058s, ↵
DeploymentId:2369561605
    r1/migrations/01_javascript_sources.sql::migration00001-01::cicddemo
    r1/migrations/02_sessions.sql::migration00001-02::cicddemo
    r1/migrations/03_hit_counts.sql::migration00001-03::cicddemo
    r1/migrations/05_hit_counter.sql::migration00001-05::cicddemo
    r2/migrations/02_hit_counter.sql::release00002-01::cicddemo

-  Database updated at 8/18/23, 2:52 PM.Applied 1 changeset(s), DeploymentId:2370342390
    r2/migrations/03_hit_counter_pkg.sql::release00002-02::cicddemo


Operation completed successfully.
```

Database Actionsでは、以下のスクリーンショットで示すように、テキストベースのバージョンへのグラフィカルなフロントエンドが提供されます。

図4：Database ActionのLiquibaseデプロイメント画面を示すスクリーンショット



デプロイメントの追跡にどちらの方法（コマンドラインの使用またはグラフィカル・ユーザー・インタフェースの使用）を選んだとしても、データベースに対してどの変更セットがいつデプロイされたかを常に認識できます。これはほとんどのユーザーにとって大きな改善点です。これにより、必要とされるチーム間の連携が大幅に少なくなるためです。データベース変更の追跡は今や、アプリケーション開発中の重要な目標ではなく副産物となっており、このようなタイプの追跡を実現するための労力はまったく必要ありません。この本質的な利点を過小評価すべきではありません。

## パイプライン障害の確認

前半の章で説明したように、データベースのデプロイメント・エラーが発生した場合、CIパイプラインの障害は避けられません。Liquibaseの場合、この目標を達成するためにいくつか属性を設定する必要があります。

最初に、常にfailOnErrorをTRUEに設定する必要があります。これにより、SQLclに現在の変更セットの実行を中断させることができます。そして、パイプラインのログを使用して、問題が発生した理由を初期の段階で見つけることが可能です。

次に、SQLclにも、エラー発生時に失敗してパイプラインの実行を中止するように指示する必要があります。SQL*Plusのように、ユーザーはwhenever sqlerror exitコマンドを指定できます。

多くのユーザーは、プロジェクトのsrc/database/utilsサブディレクトリに自身のデプロイメント・スクリプトを作成することを選択しました。たとえば、そのようなデプロイメント・スクリプトによって、デプロイメント前にデータベース内にリストア・ポイントを作成し、ロールバック操作が必要な場合にLiquibaseタグを設定してから、変更ログをデプロイできます。SQLclにはLiquibaseサポートが組み込まれているため、このすべてを他のSQLコマンドと同じように通常の*.sqlファイル内で実行できます。デプロイメント・スクリプトの例を以下に示します。

```
/*
      NAME :
            deploy.sql
      PURPOSE:
            enable liquibase deployments in CI/CD pipelines

      PARAMETERS
            (1)   tag name (used for lb tag and creating a restore point)
                  typically the COMMIT SHA.Since that can start with an invalid
                  character the value is prefixed with a t_
                  This parameter is provided during the invocation of deploy.sql
                  as part of the pipieline execution.
*/
whenever sqlerror exit

declare
      nonexistant_restore_point exception;
      pragma exception_init(nonexistant_restore_point, -38780);
begin
      if length('&1') = 0 then
            raise_application_error(
                  -20001,
                  'Please provide a valid tag name!'
            );
      end if;

      -- drop the restore point, it doesn't matter if it exists or not.
      -- note it's not possible to use bind variables in dynamic SQL
      -- executing DDL statements
      begin
            execute immediate
                  replace(
```

```
                    'drop restore point :MYRPNAME',
                    ':MYRPNAME',
                    dbms_assert.simple_sql_name('t_&1')
                );
    exception
        when nonexistant_restore_point then null;
        when others then raise;

    end;
end;
/

create restore point t_&1;
lb tag -tag t_&1
lb update -changelog-file controller.xml
drop restore point t_&1;
```

このファイルはCIパイプラインで使用できます。スキーマ移行の一部としてエラーに遭遇し次第、パイプラインは停止します。忘れずにリストア・ポイントの使用を監視し、リリースがサインオフまたは成功とマーク付けされたら、それらをクリーンアップしてください。

## まとめ

SQLclやLiquibase、Flywayなどのスキーマ移行ツールを使用すると、開発者はデータベース移行について自信を深められます。いったん開発者が各ツールに関連するワークフローを採用すれば、スキーマ移行ははるかに管理しやすくなります。頻繁なリリース、そして小規模な増分変更というモットーを組み合わせることで、メインの変更ログが何百もの変更で構成されるという状況にならずに済みます。実のところ、そのような状況は、ユーザーがパイプラインのタイムアウトに直面する可能性があるため、不具合が生じます。したがって、本番同様の量のデータに対してテストを行うことも重要です。これについては、次の章で取り上げます。

# 効率的で迅速なテスト・データベースのプロビジョニング

テスト・データベースは、**継続的インテグレーション（CI）**パイプラインに不可欠です。このコンテキストでは、データベースは多くの場合、**CIデータベース**と呼ばれます。この技術概要の第1章で説明したように、リリースがCIデータベースにデプロイされると、複数のテストが自動的に実行されます。理想的には、エンティティ（データベース・スキーマ、プラガブル・データベース（PDB）、クラウド・サービスなど）によって本番データベースが表されます。

CIパイプラインの実行は迅速に終了しなければならないという原則に従って、デプロイメント・ターゲットのプロビジョニングが完了するまでにかかる時間は、可能な限り短くする必要があります。CI/CDパイプラインを効率的に使用するためには、迅速なフィードバックが不可欠であることを忘れないでください。開発者が問題を認識する時期が早ければ早いほど、すぐに問題を解決できます。

CIデータベースの作成を短縮するには、異なるアプローチを使用することもできます。

- Autonomous Databaseクラウド・サービスのプロビジョニング
- コンテナ・イメージ（スタンドアロン/Kubernetesによるオーケストレーション）の使用
- プラガブル・データベースの作成
- Copy-On-Writeテクノロジーの使用による（プラガブル）データベースのクローニング
- データベース・スキーマのプロビジョニング

これらの手法にはそれぞれ利点と欠点があります。それについては、このセクションで説明します。

## Oracle Autonomous Database

Oracle Autonomous Databaseは、柔軟に拡張可能で高速の問合せパフォーマンスを実現する、使いやすい完全自律型のデータベースを提供します。クラウド・サービスとして、Autonomous Databaseはデータベース管理を必要としません。

Autonomous Database-Serverless（ADB-S）データベースは、既存のクラウド・フットプリントを持つお客様にとっての適切な候補です。このデータベースは、自動化のレベルが高く、さまざまなオプションを使用して作成できるため、CIパイプラインでの使用に最適です。Autonomous Databaseを作成するための一般的なオプションには次のようなものがあります。

- 空のADB-Sインスタンスの作成（あまり一般的ではない）
- 本番環境などからのADB-Sインスタンスのクローニング
- バックアップからのADB-Sインスタンスの作成

これらすべての操作は、**Terraform**、**Oracle Cloud Infrastructure**（**OCI**）コマンドライン・インタフェース（CLI）、またはプレーン**REST**コールを使用して自動化できます。以下のTerraformスニペットは、CI/CDパイプラインで使用する既存のAutonomous Databaseのクローニングに必要な最小限の情報を提供しています。

```
resource "oci_database_autonomous_database" "clone_adb_instance" {
   compartment_id              = var.compartment_ocid
   db_name                     = var.ci_database_name
   clone_type                  = "FULL"
   source                       = "DATABASE"
   source_id                   = ci_database_autonomous_database.src_instance.id
   admin_password              = base64decode(local.admin_pwd_ocid)
   cpu_core_count               = 1
   ocpu_count                  = 1
   data_storage_size_in_tbs    = 1
   nsg_ids                     = [ module.network.cicd_nsg_ocid ]
   subnet_id                   = module.network.backend_subnet_ocid
}
```

ADB-Sインスタンスはプライベート・サブネット（Terraformモジュールによって作成され、ここでは示されていません）内で作成され、CIサーバー、パイプラインのインフラストラクチャ、およびネットワーク・セキュリティ・グループ（NSG）に統合されています。このスニペットは完全クローンを作成しますが、別のクローン・タイプも使用できます。ワークロードの要件にもっとも適したものを選択する必要があります。

Autonomous Databaseのクローニングの詳細については、公式ドキュメント・セットを参照してください。

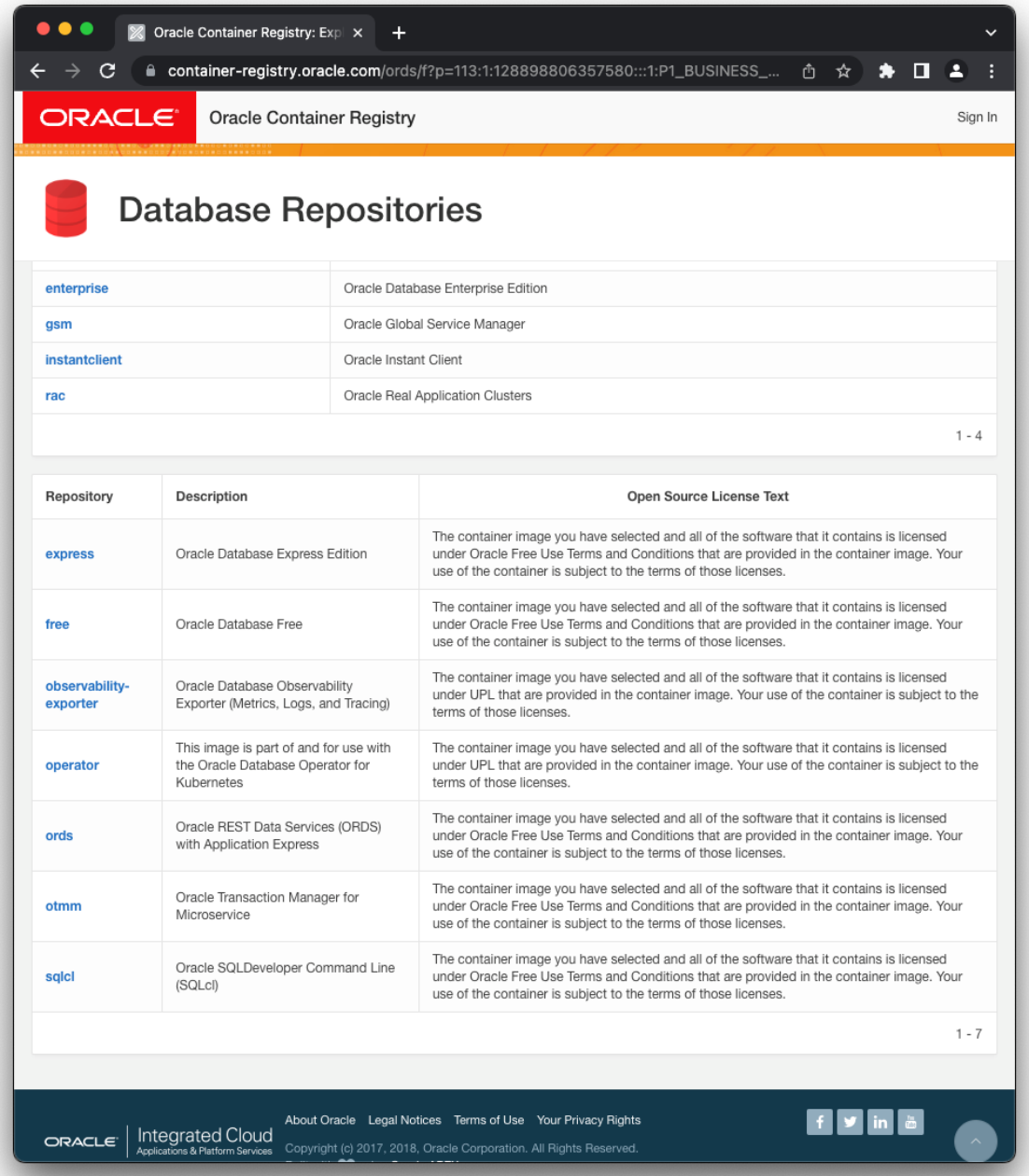
## コンテナ・イメージの使用

過去10年間で、コンテナ・テクノロジーは広く普及しました。従来の仮想マシンとは異なり、相互の独立性は低くなりますが、同時にはるかに軽量化されました。

コンテナ・テクノロジーを使用することの魅力は簡素化です。CI/CDパイプラインでは、非常に頻繁にコンテナ・イメージを構築します。これらのコンテナ・イメージは、コンテナ・ランタイムが使用できる場所であれば、開発者のラップトップから本番環境にいたるまでどこでもデプロイできます。コンテナ・イメージは自己完結型であり、ランタイム・ライブラリをアプリケーション・コードと一緒にパッケージ化するプロセスによって、デプロイメントの問題が発生する可能性が低くなります。

多数のお客様にとって、コンテナ・イメージの使用は標準となりました。アプリケーションがコンテナにデプロイされている場合、データベース・コンテナを使わない理由はないでしょう。オラクル独自のコンテナ・レジストリには、Oracle Database専用のセクションがあります。

図5：オラクルのコンテナ・レジストリ上のデータベース・コンテナ・イメージを示すスクリーンショット




さまざまなコンテナ・ランタイムのデータベース・サポートに関する詳細については、My Oracle Support *Oracle Support for Database Running on Docker*（*Doc ID 2216342.1*）を参照してください。

## PodmanまたはDockerによるコンテナ・イメージの使用

以下の例は、公式のコンテナ・イメージを使用してOracle Database 23c Freeデータベースをプロビジョニングする方法を示しています。このシナリオでは、データベースは一時的なものであることに注意してください。それ以外の場合は、必ずコンテナにボリュームを提供する必要があります。そうしないと、データ損失を被る可能性があります。例では、ディストリビューションのデフォルトのコンテナ・ランタイムであるPodmanを使用して、Oracle Linux 8上でテストを行いました。

```
podman run --rm -it \
--secret=oracle_pwd \
--name cicd-example \
--publish 1521:1521 \
```

container-registry.oracle.com/database/free:latest

上記のコマンドは、Oracle Database 23c Freeのイメージに基づいて新しいコンテナ・インスタンスを開始し、リスナー・ポート1521を開きます。このコマンドは、oracle_pwdという名前のPodmanシークレットとして保存された値に対するSYSTEMとSYSの両方のデータベース・ユーザー・パスワードを初期化します。データベースは1分以内に使用できるようになり、コンテナ・ホスト上のポート1521でアクセスできます。

CI/CDパイプラインでは、プロビジョニングされた空のデータベースをソースとして使用できること、または、本章の後半で説明するクローニング・テクノロジーを使用して既存の"ゴールデン・コピー"（プラガブル）データベースのコピーを作成できることに注意してください。後者のほうがより効率的な（つまり、時間がかからない）アプローチである可能性があります。

## Kubernetesによるコンテナ・イメージの使用

コンテナ・テクノロジーに精通したユーザーは、Kubernetesまたは同等のオーケストレーション・エンジンでデータベース・コンテナをデプロイすることをお勧めします。Oracle Databaseコンテナおよびクラスタ・リソースを手動でプロビジョニングして管理する代わりに、管理者はオープンソースのOracle Database Operator for Kubernetesを使用できます。

Oracle DatabaseをKubernetesネイティブ化（Kubernetesによる観察と運用が可能）するというオラクルのコミットメントの一部として、オラクルはOracle Database Operator for Kubernetes（OraOperator）をリリースしました。OraOperatorは、カスタム・リソースおよびコントローラでKubernetes APIを拡張することにより、Oracle Databaseのライフサイクル管理を自動化します。

現行のリリース（バージョン1.0.0）は、Autonomous Databaseのサポートを含む、多数のデータベース構成およびインフラストラクチャをサポートします。データベースおよびインフラストラクチャ・タイプ別のサポートされる操作の詳細なリストについては、ドキュメントを参照してください。

OraOperatorでは、以下のYAMLファイルを使用してOracle Database 23c Freeのインスタンスをデプロイできます。

```
#
# Copyright (c) 2023, Oracle and/or its affiliates.
# Licensed under the Universal Permissive License v 1.0 as shown at
# http://oss.oracle.com/licenses/upl.
#
apiVersion: database.oracle.com/v1alpha1
kind:SingleInstanceDatabase
metadata:
  name: freedb-sample
  namespace: default
spec:
  ## Use only alphanumeric characters for sid, always FREE for Oracle Database Free
  sid:FREE

  ## DB edition
  edition: free

  ## Secret containing SIDB password mapped to secretKey
  adminPassword:
    secretName: freedb-admin-secret
  ## Database image details
  image:
    ## Oracle Database Free is only supported from DB version 23 onwards
    pullFrom: container-registry.oracle.com/database/free:latest
    prebuiltDB: true
```

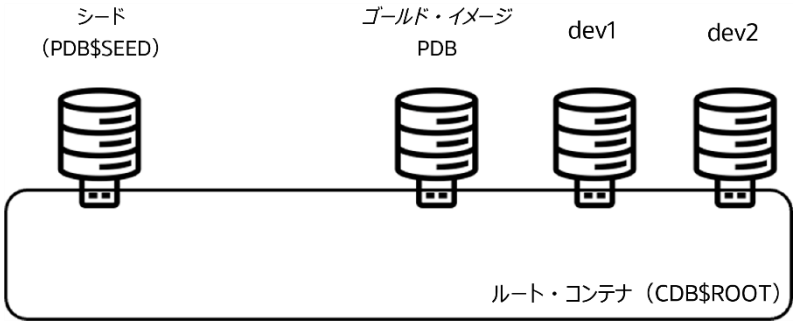
```
## Count of Database Pods.Should be 1 for Oracle Database Free or Express Edition.
replicas:1
```

YAMLファイルを適用すると、OraOperatorがデータベースの設定を行います。YAMLファイルがKubernetes APIへ送信された数分後に、データベースに接続できます。

## コンテナ・データベースの使用

コンテナ・データベース（CDB）は、Oracle 12c Release 1で導入されました。従来の非CDBアーキテクチャとは異なり、コンテナ・データベースは、オラクルが管理するシード・データベースおよび1つまたは複数のユーザー・プラガブル・データベース（PDB）で構成されます。PDBは、スキーマ、オブジェクト、および関連構造から成るユーザーが作成したセットです。これはクライアント・アプリケーションでは、論理上は別のデータベースとして認識されます。言い換えると、おのおののPDBによって名前空間の分離が提供されます。これは、ワークロードの統合や隔離された個別の開発環境の提供に最適です。

図6：コンテナ・データベースの使用によるテスト環境の迅速なプロビジョニング




追加の抽象化レベルはアプリケーション・コンテナの形式で可能ですが、これについてはこの技術概要の範囲外です。

コンテナ・データベースの使用には追加のライセンスが必要な場合があることに注意してください。詳細については、該当する特定のOracle Databaseバージョンの Database Licensing Guideを参照してください。

以降のセクションでは、CI/CDパイプラインの一部としてプラガブル・データベースを作成するための一般的なオプションについて説明してから、それらの作成や削除を自動化する方法について説明します。

### 新しい空のプラガブル・データベースの作成

新しく作成されたPDBは、作成後は“空”です。CIパイプラインでは、ユニット・テストが実行可能になる前に、まずアプリケーション全体を再デプロイできるようにする必要があります。これは、時間のかかるタスクとなる可能性があります。前述したコンテナ・イメージを使用している場合、すでに空のPDBのデプロイメント・ターゲットが存在する可能性があります。Oracle Database 23c Freeおよび旧世代のOracle Database Express Edition（XE）のどちらでも、空のPDBが標準で提供されます。これはそれぞれ、FREEPDB1またはXEPDB1と名付けられます。コンテナ・イメージに基づいたセットアップと組み合わせることで、ほとんどの場合、1分以内に起動してFREEPDB1/XEPDB1に接続できます。

### 既存のプラガブル・データベースのクローニング

スキーマ変更をデプロイするための、より洗練された、時間の消費も少ない可能性がある方法は、オラクルのPDBクローニング機能を使用することです。オラクルのMultitenantオプションが12c Release 1で開発されて以来、PDBのクローニングのための方法が次々と追加されてきました。詳細については、Oracle Database管理者ガイドの第8章で説明しています。

既存の“ゴールド・イメージ”PDBのクローニングにより、アプリケーションのデプロイにかかる時間を大幅に短縮できます。LiquibaseやFlywayなどの適切なツールを使用すると、可能な限り短時間でPDBクローンにリリースを適用できます。スキーマ変更を管理するためのツールについては、本書の第3章で説明しています。

上記のドキュメントのとおり、クローニングには多数のオプションがあります。（Copy-on-Writeテクノロジーに基づく）スパース・クローンを使用することで、クローニングされるPDBのストレージ要件は通常、大幅に減少します。

本番ロールアウト時の不測の事態を避けるために、PDBのクローンは本番環境に似せる必要があることに注意してください。ユニット・テストを本番規模のデータベース・クローン上で実行したくない場合は、インテグレーション・テストまたはパフォーマンス・テストの段階でこれについて考慮してください。

## プラガブル・データベースのライフサイクル管理の自動化

変更チケットを作成することでシステムのクローンをリクエストするという手動の方法は、ほとんどのユーザーにとってもはや実行不可能です。それは単純に時間がかかりすぎるからです。この従来型のワークフローは、CIパイプラインでの使用にも適していません。CIパイプラインの使用に向けた最初のステップは、PDBライフサイクルを自動化することです。幸い、このタスクはすでに完了しています。Oracle REST Data Services（ORDS）は、PDBの作成、クローニング、および削除の自動化に使用できるRESTエンドポイントを提供します。

以下の例は、既存のCIパイプラインから取り出したものです。curlユーティリティが"ゴールド・イメージPDB"のクローニングに使用されています。

```
clone-source-PDB:
   stage: build
   environment:
      name: testing
   script: |
      curl --fail -X POST -d '{
         "method": "clone",
         "clonePDBName": "'${CLONE_PDB_NAME}'",
         "no_data": false,
         "snapshotCopy": false,
         "tempSize":"100M",
         "totalSize":"UNLIMITED"
      }' \
      -H "Content-Type:application/json" \
      -u devops:${ORDS_PASSWORD} \
      https://${ORDS_HOST}:8443/ords/_/db-api/stable/database/pdbs/${SRC_PDB_NAME}/
```

Vaultインスタンスによって、またはローカルでCIサーバーによって維持される環境変数を使用することで、コードの再利用性とセキュリティが向上します。上記の例は非常に基本的で、スナップショット・クローニング機能を使用しません。Copy-On-Writeスナップショット（例では不使用）を作成することで、クローニングされるデータベースのストレージ・フットプリントを削減でき（対応したストレージをシステムが使用している場合）、クローニング操作の期間全体を大幅に短縮できます。

PDBライフサイクル管理コールの詳細については、Oracle REST Data Services APIドキュメントを参照してください。

## Oracle Recovery Managerのduplicateコマンドの使用

Oracle Recovery Manager（*Oracle RMAN*）はOracle Databaseと完全に統合されており、（プラガブル）データベースの複製を含む、さまざまなバックアップとリカバリのアクティビティを実行します。必要に応じて、Recovery Managerを使用して以下のソースを複製できます。

- 非コンテナ・データベース（非CDB）
- コンテナ・データベース（CDB）全体
- CDB内の個々のプラガブル・データベース（PDB）

Recovery Managerの複製ステップはCIパイプラインに簡単に統合できます。必要なRecovery Managerコマンドはスクリプトとして Gitリポジトリに格納する必要があります。この技術概要で前述したトップレベルのutilsディレクトリに配置するのが理想的です。

Recovery Managerのduplicateコマンドの使用に関連するおもな懸念事項は、経過時間です。ソースが大きくなるほど、複製が完了するまでに時間がかかります。duplicateコマンドの完了に必要な時間は、割り当てるチャネルを適切な数にしたり、高帯域幅のネットワークリンクを使用したり、Backup and Recovery User's Guideに記載されているその他の手段を使用したりすることで短縮できます。

## ブロックデバイス・クローニング・テクノロジーの使用

従来のストレージ・アレイをオンプレミスで使用するお客様およびブロック・ボリューム・サービスを使用するクラウドのお客様は、ブロック・ボリューム・クローニング・テクノロジーを使用してデータベースのコピーを迅速に作成できます。このオプションは、非CDBデータベースで使用されていたものであり、従来のOracleアーキテクチャを備えたOracle Database 19cを使用するお客様は引き続き利用できます。ブロックボリューム・クローニングのプロセスでは、データベースをバックアップ・モードにして、処理中のI/Oリクエスト（必ずしも順序どおりにデータベースに送信されません）によってコピーが破損されるのを防ぐ必要がある場合があります。

多数のストレージ・ベンダーが、ブロック・デバイスをクローニングするためのツールや手順を提供しています。これらを参照してプロセスを自動化することが、外部APIが提供されている場合にはもっとも簡単でしょう。

## Copy-On-Writeテクノロジーの使用

Copy-On-Write（COW）テクノロジー（スパース・クローンとも呼ばれます）を使用すると、管理者はソースで必要とされる領域のわずかな部分しか使わずにデータベースのフルサイズ・コピーを作成できます。かなり簡潔に言えば、ボリュームのスパース・クローンを迅速に作成できます。オペレーティング・システムの観点から見ると、スパース・クローンはソースと同じプロパティを持ちます。ただし内部では、ストレージ・ソフトウェアは、完全クローンの場合のような、ソースからターゲットへのすべてのビットのコピーを開始しません。スパース・クローンには、ソース・データ（ソース・ボリューム）へのポインタがあります。クローニングされたボリュームのデータが変更される場合にのみ、ストレージが使用されます。言い換えると、クローニングされるデータベースに必要なストレージの量は、変更の量に正比例します。

このプロセスは、通常はソース・データベースの10 %以下が変更されるCIパイプラインで極めて有益になり得ます。このアプローチの潜在的な欠点である、差分のメンテナンスによって生じるストレージ・レイヤー上のオーバーヘッドは通常、大規模なデータベースの場合は特に、ストレージの節約によって補われます。

COWテクノロジーはオラクルのMultitenantオプションが導入される前から存在するため、非CDBアーキテクチャおよびプラガブル・データベースを使用する19cデータベースで使用できます。

すべてのストレージ・エンジンおよびファイル・システムがCOWテクノロジーをサポートしているわけではないため、お使いのソリューションが（プラガブル）データベースのCOWクローニングをサポートしているかどうかについてストレージ・ベンダーとオラクルに確認してください。

## スキーマ・プロビジョニングの使用

スキーマ・プロビジョニングは、デプロイメント・ターゲットの提供に適した最後の重要なメカニズムです。スキーマ・プロビジョニングは、Multitenantおよび非CDB環境で同様に利用できます。

もっとも基本的な形式では、ユーザー作成のRESTコールが既存のOracleデータベースに新しいスキーマを作成し、CIパイプラインにパスワードを返します。次のステップで、アプリケーション全体をプロビジョニングする必要があります。前述した新しい空のプラガブル・データベース・アプローチと同じく、これは時間のかかるタスクとなる可能性があります。

PDBのクローニングに関連して前述したシナリオと同様の、よく知られ十分に定義された状態で開始するほうが速い可能性があります。Oracle Databaseのスキーマは、SQLコマンドを使用して“クローニング”することはできません。ただし、Data Pump Export / Data Pump Importを使用して複製できます。適切なエクスポート・ファイルがある場合、CIパイプラインはRESTコールを使用してData Pumpを起動できます。ORDSで一連のRESTエンドポイントが提供されるため、Data Pump Import Jobの作成が可能になります。

## まとめ

迅速なフィードバック・ループの精神に従って、CIパイプラインはデプロイメント・ターゲットが迅速にプロビジョニングされるようにする必要があります。すでにクラウドへの移行を進めているお客様は、多数のオプションを使用できます。Autonomous Database Serverless（ADB-S）

インスタンスのクローニングは、多くの条件を満たします。“本番環境同様のデータ量でのテスト”から“環境の迅速なプロビジョニング”まで、足りない部分はほとんどありません。このアプローチは、適切なデータ・マスキングや他の多数のコンプライアンス・チェックがないと本番データを使用できない、規制の厳しい環境では適さない場合があります。

代わりに、オンプレミスまたはクラウドでのクローニング環境は、多数のお客様に適している可能性があります。スパース・クローンは特に、テスト環境相当のTBをプロビジョニングしなければならないというジレンマから抜け出す方法を提供します。

どのアプローチを選択するにしても、稼働させる前に必ずパフォーマンス・テストを実行してください。本番同様のデータでのパフォーマンス・テストは、デプロイメント・パイプラインが長時間実行タスクによってタイムアウトにならないようにするための唯一の方法です。

# 効率的なCI/CDパイプラインの記述

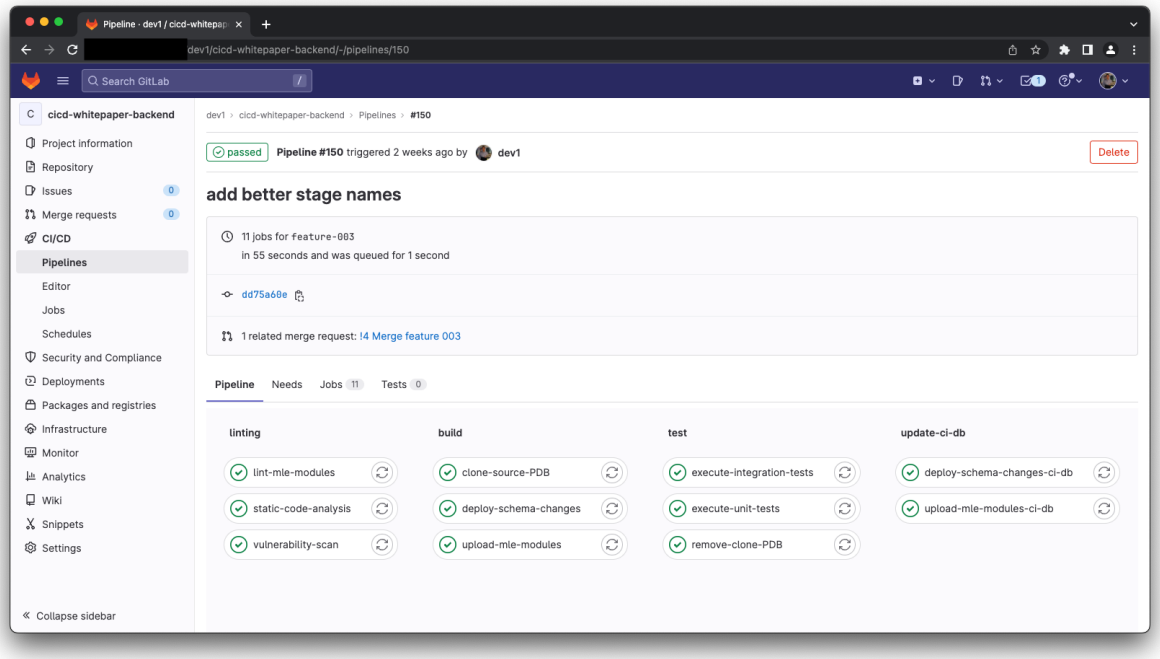
この技術概要のこれまでの章では、CI/CDパイプラインを定義する方法を理解するために必要な基礎を築こうとしてきました。この章では、GitLab Community Editionに基づく仮想CI/CDパイプラインについて説明します。この章ではGitLab CEを選択しましたが、次に説明する概念は、JenkinsからGitHub Actionsまであらゆる CIサーバーに当てはまります。GitLabの使用によって、このテクノロジーを推奨するわけではありません。

注：*GitLab*や*GitHub*などの*CI/CD*ソリューションの管理について取り上げるなら、すぐに何百ページもの説明になってしまいます。本章では、特定のテクノロジーの使用を開始するのに必要な概念とオプションについて取り上げようとしています。これは、それぞれのドキュメントの代わりにはなりません。

## CI/CDパイプラインの概要

自動化プロジェクトの中核として、CI/CDパイプラインは通常、YAMLなどのマークアップ言語で定義されます。一部のCIサーバーは、独自のドメイン固有言語を使用します。バージョン管理システムに関する前半の章で説明したように、パイプラインの定義は、アプリケーション・コードと併せて簡単に保存できる形式であることが重要です。

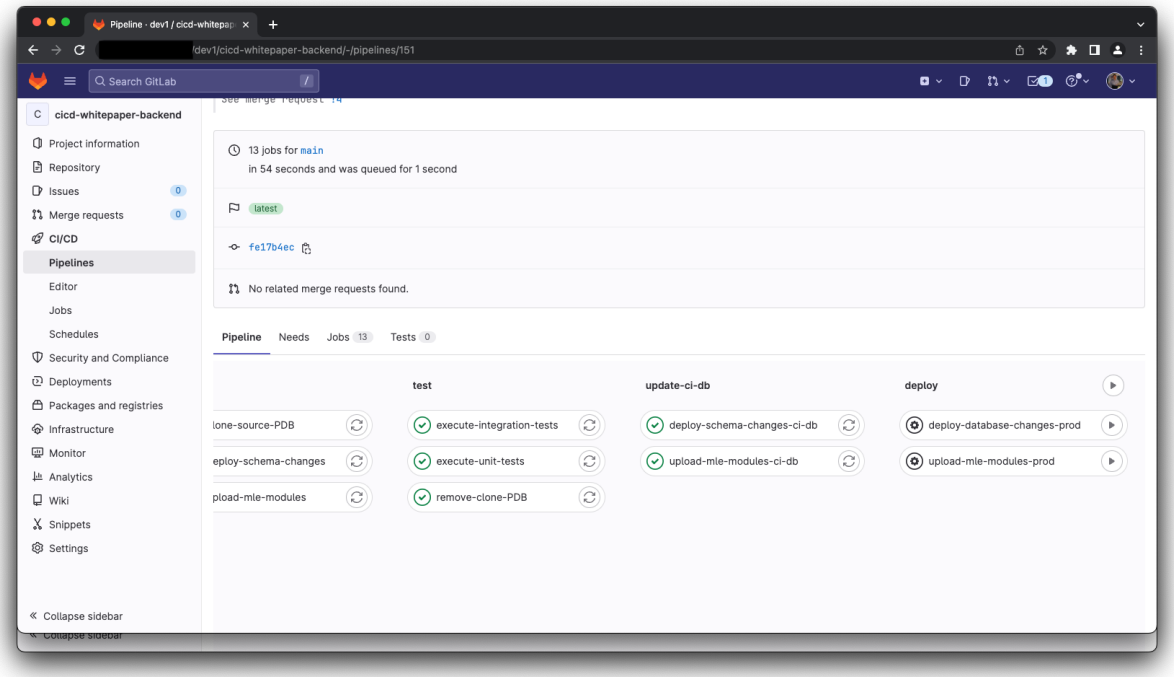
図7：GitLabでのパイプライン実行の成功例




上記のスクリーンショットは、パイプライン実行の成功を示しています。このケースでは、このパイプラインはマージ・リクエスト（GitHubでのプル・リクエスト）の一部として実行されました。GitLabでは、パイプラインはマージ・リクエストの作成時に実行され、実際のマージの一部として再度実行されます。GitLabへのコミットのプッシュは、CIパイプラインの実行をトリガーする別のオプションです。

アクティビティによっては、追加のアクティビティを開始できます。ブランチのマージは通常、特に**トランクベース開発**のモデルを適用する場合は、より複雑なアクティビティです。その場合は、アプリケーションを本番環境へデプロイするために、“trunk”または“main”へのマージを使用できます。これについて、以下の図に示します。

図8：GitLabでのマージ・パイプライン実行の成功例




デプロイメントも自動化されている継続的デプロイメントの場合とは異なり、デプロイ・ステージの“play”ボタンで分かるように、この例ではアプリケーションは本番環境に手動でデプロイされます。

これらのスクリーンショットには、CIパイプラインに関連した重要な概念が示されています。

- ステージ
- ジョブ

## CIパイプラインのステージ

ステージを使用してジョブをグループ化できます。上記の例では、リンティング、静的コード分析、および脆弱性スキャンがリンティング・ステージで実行されます。ステージの名前は完全に任意で、プロジェクトの要件に応じて選択できます。ほとんどのCIサーバーでは、以下のGitlabでの例のようにステージ名の定義が可能です。

```
# stage definition
stages:
- linting
- build
- test
- update-ci-db
- deploy
```

通常、ステージは順序どおりに完了します。ステージの設計に際しては、“早く失敗せよ”の原則を考慮する必要があります。パイプラインに障害が発生するのが早いほど、開発者はより迅速に対応できます。時間がほとんどかからないジョブを最初に実行してから、ソース・データベースのクローニングのように時間のかかる可能性のあるジョブを開始することをお薦めします。

## CIパイプラインのジョブ

ジョブは、typescript-eslint、jslint、jshint、および他のリンティング・ツールへのJavaScriptモジュールの受渡しなど、CIパイプラインが実行しなければならないものです。これには、デプロイメント・ターゲットを作成するための“ゴールデン・コピー”PDBのクローニングも含まれます。CI/CDパイプラインは、ジョブを論理的にステージにグループ化します。YAML構文を使用するシステムでは、ジョブを次のように定義できます。

```
remove-clone-PDB:
```

```
    stage: test
    environment:
    name: testing
  script: |
    curl --fail -X DELETE \
    -d '{ "action":"INCLUDING" }' \
    -H "Content-Type:application/json" \
    -u devops:${ORDS_PASSWORD} \
   https://${ORDS_DEV}/ords/_/db-api/stable/database/pdbs/${CLONE_PDB_NAME}/
```

remove-clone-PDBジョブは、“test”ステージの一部として実行され、シェルスクリプト・コードを実行します（テストの完了後にクローニングされたPDBの削除をリクエストするORDSインスタンスに対するcurlコマンドライン・ユーティリティ経由のHTTP POSTリクエスト）。

## コード品質の確保

コード品質は、デプロイメントを自動化するにあたってもっとも重要なメトリックの1つです。State of DevOps Reportの定期的な報告によると、頻繁なデプロイは失敗率の低下と密接に関連しています。これは直感に反するように聞こえるかもしれませんが、CIパイプラインまたはpre-commitフックの一部として実行されるコード品質チェックにより、達成可能な要件となっています。

### リンティング

ウィキペディアによると、リンティングは、

- プログラミング・エラー
- 多数の種類のバグ
- （プログラミング）スタイル
- その他

が検出または実行される可能性のあるプロセスに対してコンピュータ・サイエンスで使用される用語です。リンティングは、CIパイプラインの実行時に最初のタスクの1つとして実行される必要があります。リンティングのガイドラインに準拠しないコードは、デプロイして正常に動作しないことを明らかにする必要がありません。リンティング・ステージで、そのコードが失敗することが確認されます。そのため、デプロイメントをスキップでき、パイプラインのステータスは“failure”に設定されます。

リンティング・フェーズ中のエラーは、まれなはずです。ほとんどの**統合開発環境（IDE）**では、開発者は開発プロセスにリンターを含めることができます。開発者のラップトップでパイプラインと同じリンティング・ルールと定義を使用している場合、潜在的なエラーはすべてIDEによりハイライト表示され、コミットの前に修正されているはずです。

リンティングは精密科学ではなく、汎用的なものではありません。リンターがデフォルトで実行するルールの中には、プロジェクトに適用できないものもあります。このような場合、チームは通常、どのリンティング・ルールを使用すべきで、どれを無効化すべきかを判断します。

**Infrastructure as Code**などの他のすべての構成設定と同様に、リンター構成もGitリポジトリの一部である必要があります。

### ユニット・テスト

コードは、正式な要件を通過すると、**ユニット・テスト**を受けることができます。ここでは、多数の一般的なユニット・テスト・フレームワークを使用できます。utPLSQLはPL/SQLのテストに役立ち、jest、mocha、その他多数のJavaScriptユニット・テスト・フレームワークがあります。

JavaScriptユニット・テスト・フレームワークを使用すると、異なるユニット・テスト・フレームワークを学ぶ必要なしに、JavaScriptエコシステムを維持できます。Mochaとchaiは、非常に普及している組合せのユニット・テスト・ツールであることが証明されています。好みに応じて、クライアント側のnode-oracledbデータベース・ドライバによるユニット・テストを実行するか、またはデータベース内からユニット・テストを実行することができます。以下の例は、クライアント側のドライバを使用したユニット・テスト・ファイルの抜粋です。

```
import { assert, expect, should } from "chai";
```

```
import { incrementCounter } from "../src/incrementCounter.mjs";
import oracledb from "oracledb";

describe("client-side unit test suite", function() {

    describe("ensure that a session is created", function() {

        it("should insert a row into the sessions table", async function () {
            const session_id = '123456C0340C0138E063020011AC3B29';
            // action
            incrementCounter(
                session_id,
                'node',
                'macOS'
            );

            // verification
            const conn = await oracledb.getConnection(
                {
                    user:          process.env.DB_USERNAME,
                    password:      process.env.DB_PASSWORD,
                    connectionString: process.env.DB_CONNECTIONSTRING
                }
            );

            const result = await conn.execute(
                `select
                    count(*)
                from
                    sessions
                where
                    session_id = :session_id `,
                {
                    session_id: {
                        dir: oracledb.BIND_IN,
                        val: session_id,
                        type: oracledb.STRING
                    }
                },
                {
                    outFormat: oracledb.OUT_FORMAT_ARRAY
                }
            );

            // there should only be 1 entry in the sessions table for this
            // particular session_id
            assert.strictEqual(
                result.rows[0][0], 1,
                "there should only be 1 session per GUID"
            );
        });
```

```
    });
});
```

CIパイプラインを使用してこのテストを簡単に実行できます。

```
$ npx mocha

  client-side unit test suite
    ensure that a session is created
      □ should insert a row into the sessions table (220ms)

  1 passing (222ms)
```

注：あらゆるサード・パーティ・ソフトウェアと同様に、関連チームから常に同意を得て使用する必要があります。

当然ながら、ユニット・テストにPL/SQLを使用できます。以下の例は、この領域で非常に普及しているオープンソース・フレームワークであるutPLSQLを使用してユニット・テストを実行する方法を示したものです（簡略化のためLiquibase固有の注釈は省略しています）。この例は、ページ・ヒットをカウントする仮想アプリケーションから取られたものです。

```
create or replace package hit_counter_test_pkg as

    --%suite(unit tests for hit counter functionality)

    --%test(verify that a new session is created)
    --%tags(basic)
    procedure incrementCounter_test_001;

end hit_counter_test_pkg;
/

create or replace package body hit_counter_test_pkg as

    c_session_id constant varchar2(100) := 016050AA7B87051AE063020011ACAED8';
    c_browser constant varchar2(100) := 'utplsql';
    c_operating_system constant varchar2(100) := 'oracle';

    procedure incrementCounter_test_001 as
        l_cnt pls_integer;
    begin
        -- Act
        hit_counter_pkg.increment_counter(
            p_session_id => c_session_id,
            p_browser => c_browser,
            p_operating_system => c_operating_system
        );

        -- Assert
        select
            count(*)
        into
            l_cnt
        from
            hit_counts
```

```
            where
                session_id = c_session_id;

            -- there shouldn't be more than 1 rows inserted
            ut.expect( l_cnt ).to_equal( 1 );

    end incrementCounter_test_001;
end hit_counter_test_pkg;
/
```

作成が可能なユニット・テストは多数ありますが、この記事ではポイントを示すために1つのテストを取り上げます。
本番アプリケーションでは、より徹底したテストを実行する必要があります。

## パフォーマンス・テスト

パフォーマンス・テストは、時間がかかる性質があるため、必ずしもCI/CDパイプラインによって開始されるわけではありません。ただし、定期的にパフォーマンス・テストを実行することが非常に重要です。理想的には、これらのテストは本番ワークロードを対象とする必要があります。Oracle Databaseに関しては、Real Application TestingやSQL Performance Analyzerなどの多数のオプションを使用できます。ライセンス同意書によっては、これらのオプションにコストがかかる場合があります。

クライアント側のロード・ジェネレータも、アプリケーション・ロードの生成に適切に使用されています。実際の使用特性をできるだけ正確に表現できる限り、これらを使用することもパフォーマンス・テストへの実行可能なアプローチとなります。

## デプロイメント

継続的インテグレーション・パイプラインのインテグレーション部分が問題なく完了したら、次は変更をデプロイします。コンテナなどの最新のソフトウェア開発ツールによって、以前によく見られたライブラリの非互換性などのデプロイメントに関する問題が適切に処理されます。デプロイメント・パイプラインを使用して、Liquibase、Flyway、またはその他のツールを使用したデータベース変更を促進することで、データベース・アプリケーションでも同じことが実現できます。これらのツールが保持するメタデータによって、スクリプトは一度だけ実行されることが保証されます。したがって、プロジェクトのリリースのmigrationsサブディレクトリ内のすべての変更セットを参照するメイン変更ログを問題なく定義できます。

デプロイメントの自動化の度合いに関する疑問は残されたままです。純粋な形の継続的デプロイメントでは、デプロイメントがパイプラインで定義されたすべてのテストに合格したらすぐに、それらのデプロイメントが本番環境に対して実行されることを要求します。ただし、これはリスクがゼロではなく、多くの部門にとってはデプロイメントを手動でトリガーするほうが良いでしょう。あらゆるおもなCIサーバーは、パイプラインの手動のデプロイメント句をサポートしています。以下は、デプロイメント・ステップを“manual”に設定する方法を示す.gitlab-ci.ymlファイルからの抜粋です。

```
deploy-database-changes-to-prod:
  stage: deploy
  script:
  - cd src/database
  - |
    sql ${ORACLE_PROD_USER}/${ORACLE_PROD_PASSWORD}@${ORACLE_PROD_HOST}/${PROD_PDB_NAME} \
    @utils/deploy.sql ${TAG_NAME}
  environment:
    name: production
  when: manual rules:
  - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

when属性により、このステップの実行は手動で行われます。

データベース変更のデプロイメントは、CIデータベースと本番環境に限定されません。他のいずれの層も同様に重要と見なす必要があります。ユーザー受け入れテスト、インテグレーション・テスト、およびパフォーマンス・テスト環境でも、同じデプロイメント・メカニズムを使用する必要があります。ほとんどのCIサーバーでは、パイプラインを手動で実行して変数をプロセスに渡すことができます。これらは、デプロイメント先サーバーの決定に使用できます。

## CIデータベースの更新

本番環境に加えられた最新の変更によってCIデータベースを最新の状態に維持することは、CI/CDパイプラインの実行のもう1つの重要な側面です。CIデータベースは常にできる限り本番環境に近いものである必要があるため、本番環境にデプロイされた変更はCIデータベースにも反映する必要があります。そのようにしないと、将来のテストは、それらの変更が反映されていないために失敗する、あるいはさらに悪いことに、本番環境にすでに存在する変更を繰り返そうとするというリスクがある状態で実行されることになります。

LiquibaseやFlywayなどのツールのおかげで、データベースに**まだ適用されていない**すべての変更（**変更の差分**）が自動的にロールアウトされます。これは、本番環境へのデプロイメントだけでなく、テスト実行の一部としてプロビジョニングされるCIデータベースにも当てはまります。ただし、変更の差分が大きくなるほど、つまり本番環境より遅れてCIデータベースに適用される変更が多くなるほど、CIデータベースのセットアップ・フェーズにかかる時間は長くなり、最終的にパイプライン全体の実行時間が長くなります。

この問題を改善するには、CIデータベースを最新の変更で定期的に更新して、変更の差分を最小限に抑える必要があります。

デプロイメントの頻度と、継続的デリバリと継続的デプロイメントのどちらを採用しているかに基づいて、CIデータベースをいつどのように更新するかを決定する必要があります。

本番環境への日中の直接のデプロイメントを計画している場合は、変更を反映したCIデータベースの更新をデプロイメント段階の一部にするとよいでしょう。

一方、継続的デリバリを採用している場合、またはデプロイメントの頻度が低い場合は、本番環境からの“ゴールド・イメージ”を定期的にリフレッシュするだけで十分な場合があります。

## まとめ

ソフトウェア・プロジェクトの効率的なCI/CDパイプラインを作成することは、すべてのプロジェクトの要件が実装された場合に非常に有意義なものとなります。ただし、タスクは簡単ではないため、パイプラインの計画と作成には十分な時間を取る必要があります。文化が変わる可能性とチームの間に必要とされる連携の規模を過小評価すべきではありません。プロジェクトの計画段階で、不測の事態に備えてある程度の時間を取ることを推奨します。

CI/CDパイプラインは通常、YAMLまたは同等のマークアップ言語で記述されます。他のアプリケーション・アーティファクトと同様に、これらはプロジェクトのGitリポジトリの一部である必要があります。

CI/CDパイプラインにおける黄金律は、“パイプラインを緑色に保つ”、言い換えると、問題を発生させないことです。開発者は、コミットをリモート・リポジトリへプッシュする前に、ローカル・テスト（リンティング、ユニット・テスト、コード・カバレッジなど）を使用できます。多くのState of DevOpsレポートに見られるように、**テスト駆動開発（TDD）**と**トランクベース開発**の組合せは成功していることが証明されています。

ORACLE

# オンラインでのスキーマ変更の実行

これまでの章では、スキーマ変更を効率的にデプロイする方法の概要について説明しました。チームにとって適切な開発ワークフローとCI/CDパイプラインとを組み合わせることで、本番環境に障害を発生させない高い信頼度で、小規模な増分変更をアプリケーションにデプロイできます。

## 自信を持って本番環境へデプロイ

本番環境へのデプロイメントは特別です。進行中の操作を中断しないように、格別の注意を払う必要があります。多くのシステムにとって、ソフトウェア・リリースのデプロイのために本番ワークロードを停止することは長年にわたって不可能であり、そのような極端な対策はもはや必要ありません。

開発者から聞かれる懸念事項の1つが、オンラインでのスキーマ移行を実行するためにリレーショナル・データベースを使用できないこと（誤解）に関するものです。この章の目的は、実際にはアプリケーション変更をオンラインで実行できることを示して、それらの懸念事項に対処することです。

## たとえ短時間であっても停止を回避

Oracle Databaseでのスキーマ移行は、他の多くのリレーショナル・データベース管理システム（RDBMS）とは異なります。Oracle Databaseでは、過去数十年、多数の**DDL（データ定義言語）**操作をオンラインで実行可能としていました。たとえば、オンライン索引再作成は、早くもOracle 8iから利用可能でした。表への列の追加、索引の再作成、PL/SQLでのコード変更も、データベース・オブジェクトが長期間ロックされたり、ワークロードの実行がブロックされたりする結果にはなりません。

Oracle Databaseでは、アプリケーション開発に重点を置くこの章で扱うよりも、はるかに多くのオンライン操作が提供されることに注意してください。詳細については、Oracle Database Development Guide、Database Concepts、およびDatabase Administrator's Guideを参照してください。

オラクルには、**Maximum Availability Architecture（MAA）**の設計専門の完全なチームがあります。計画停止中および計画外停止中に基盤インフラストラクチャを維持する場合、このチームの仕事はとても重要です。この技術概要とある程度重複していますが、MAA技術概要も確認することをお薦めします。

## オンライン操作

オンライン操作とブロッキング操作との経過時間における違いは、特に変更対象のオブジェクトが頻繁にアクセスされる場合には顕著です。オラクルは、セグメントのコンテンツを変更しているトランザクションがアクティブな間は、表、パーティション、索引などのスキーマ・オブジェクトへの構造的な変更を適用できないことを保証します。トリガーやストアド・プロシージャなどのビジネス・ロジックが実行される場合も同じことが当てはまります。その論理的根拠は、一貫性と整合性を確保することです。これは意図されたものであり、優れた原則でもあります。

この章の関連でのオンライン操作とは、たとえロックが必要だとしても、最短期間のみロックを必要とするように最適化された操作を指します。これらの操作は通常、ONLINEというキーワードをDDLコマンドに追加するため、見つけるのは簡単です。この章で説明した一部の機能は追加のライセンスが必要になる場合があります。ご不明な点がある場合は、必ずDatabase Licensing Guideを参照してください。

Oracle SQL言語リファレンスには、リリース別のノンブロッキングDDL操作の一覧が含まれています。

## オンラインでの索引の作成

索引の作成および索引の再作成は、25年以上もの間、Oracle Databaseエンジンの一部でした。8iの期間で導入されており、開発者はONLINEキーワードを使用して、索引の作成中に表上の通常のDML操作が可能になることを示します。

索引の作成は、短期間のロックがないと完全には実行できませんが、その時間は非常に短くなければなりません。索引の作成または再作成のコマンドは、進行中のトランザクションと同様に、そのロックのキューに入ります。

最初は、表HIT_COUNTSのSESSION_IDを参照する外部キーには索引が付いていませんでした。以下のスクリプトによって、オンラインで索引が追加されます。

```
--liquibase formatted sql
--changeset developer1:"r2-02" failOnError:true labels:r2

CREATE INDEX i_hit_counts_sessions ON
      hit_counts (
            session_id
      )
            ONLINE;
```

上記のスクリプトを使用することで、ユーザーが表に対してDML操作を実行することを妨げずに索引がアプリケーションに追加されました。

## 既存の非パーティション表へのパーティション化の導入

最善の努力を尽くして計画を立てても、予期しない大量のデータにより、以前はパーティション化されていなかった表のパーティション化が必要になる場合があります。

オラクルでは、表のパーティション化を導入するためのテクノロジーを複数提供しています。以下のALTER TABLEコマンドおよびDBMS_REDEFINITION PL/SQLパッケージです。後者は別のユースケースで機能するため、後で詳しく説明します。

アプリケーションのHIT_COUNTS表のエントリを維持するための要件では、HIT_TIME列に基づくレンジごとに表のパーティション化が求められます。以下のSQLコマンドは、この操作をオンラインで実行します。同時に、以前追加された索引は、ローカル・パーティション索引に変換されます。

```
--liquibase formatted sql
--changeset developer1:"r2-03" failOnError:true labels:r2
ALTER TABLE hit_counts MODIFY
      PARTITION BY RANGE (
            hit_time
      ) INTERVAL
          ( numtoyminterval(
                1, 'MONTH'
          ) )
      ( PARTITION p1
            VALUES LESS THAN ( TO_TIMESTAMP('01.01.2000', 'dd.mm.yyyy') )
      )
      ONLINE
      UPDATE INDEXES (
            i_hit_count_session LOCAL
      );
```

上記の例は、HIT_TIMEタイムスタンプに基づいた表へのインターバル・パーティション化を示しています。データは、NUMTOYMINTERVAL()関数によって自動的に正しいパーティションへソートされます。これは、利用できる可能性の1つの例にすぎません。パーティション化スキームは、索引を含め、ほぼ望みどおりの方法で変更できます。

## オンラインでのセグメントの圧縮

表および表（サブ）パーティションは、オンラインで圧縮できます。レンジ・パーティション化をHIT_COUNTSに導入する前述の例に従って、もっとも古いセグメントをオンラインで圧縮できます。

```
--liquibase formatted sql
```

```
--changeset developer1:"r2-04" failOnError:true labels:r2
ALTER TABLE hit_counts
MOVE PARTITION p1
COMPRESS BASIC
ONLINE;
```

パーティションP1は、BASIC圧縮を使用して圧縮されました。プラットフォームによっては、**Advanced Compressionオプション**や**Hybrid Columnar Compression（HCC）**を使用して、圧縮レベルを上げることができるでしょう。

読取り専用の古いデータを圧縮するプロセスを所有したくない場合は、**自動データ最適化（ADO）**をお薦めします。ADOは、ヒートマップを使用してセグメント・アクティビティを記録します。これを使用すると、異なる表領域へのセグメントの移動やポリシー実行の一部としてのセグメントの圧縮など、**情報ライフサイクル管理（ILM）**ポリシーを定義できます。

## 表への列の追加

表へ列を追加することは、いずれの開発者にとっても一般的なタスクです。Oracle Databaseでは、新しい列を追加するプロセスを最適化しました。Oracle Database 11.2以降では、デフォルト値のない*NULL*値可能列を中断なく表に追加できます。同様に、NOT NULLとして定義されている列は、デフォルト値を使用するときにオンラインで追加できます。Oracle Database 11.2より前は、デフォルト値を含むNOT NULL列を追加するには、列の追加後に列にデフォルト値を保存するために表全体を更新する必要があったため、ストレージ・システムに大きな負荷がかかり、他のオーバーヘッドが生じました。Oracle Database 11.2では、これをメタデータのみの操作に変更し、コマンドを実行するまでの経過時間と表のサイズとの相関関係をなくしました。Oracle Database 12.1では、デフォルト値があるNULL値可能列へのサポートも追加し、表への列の追加をオンライン操作に変換しました。

ALTER TABLE ... ADD COLUMNコマンドは、ALTER TABLE ... ADD COLUMNコマンドより前のすべてのトランザクションが終了するまで待機してから、表を短時間ロックして完了します。ある意味で、ALTER TABLE ... ADD COLUMNコマンドは、ロックを保持している以前のすべてのトランザクションが終了するまでキューに入れられている他のトランザクションとまったく同じです。専用のONLINEキーワードはないことに留意してください。

## オンライン表再定義を使用したオンラインでの表構造の変更

Oracle Databaseは、他のユーザーやワークロードでの表の可用性に大きな影響を与えずに表構造の変更を行うためのメカニズムをユーザーに提供しています。このメカニズムはオンライン表再定義と呼ばれ、DBMS_REDEFINITION PL/SQLパッケージから公開されています。オンラインで表を再定義することで、手動で表を再定義する従来の方法よりも可用性が大幅に向上します。

表がオンラインで再定義される場合、ほとんどの再定義プロセス中に問合せとDML操作の両方にアクセスできます。表は通常、表のサイズや再定義の複雑さとは無関係に極めて短時間のみ、排他モードでロックされます。これはユーザーに対して完全に透過的です。ただし、再定義中に多数の同時DML操作が存在する場合は、表をロックできるようになるまで、より長く待機する必要がある場合があります。

アプリケーションのSESSIONS表は、次の4つの列で構成されるリレーショナル表として定義されます。

- SESSION_ID – フロントエンドによって提供されるGUID
- BROWSER – アプリケーションを起動するブラウザ
- OPERATING_SYSTEM – クライアントのオペレーティング・システム
- DURATION – セッションの期間

チームは、表構造を変更することにし、後者の3列をJSONドキュメントへと組み合わせました。変更は、アプリケーションがオンラインである間に実行する必要があります。ソース表を再定義できるかどうかを確認することをお薦めします。

```
begin
   sys.dbms_redefinition.can_redef_table ('DEMOUSER', 'SESSIONS'); end;
/
```

このプロセスは、上記のプロシージャによってエラーまたは例外がスローされない場合に開始できます。最初に、ターゲット表を定義します。ターゲット表は、プロセスが終了したときに、再定義する表がどのようになるかを定義します。ドキュメントでは、これを仮表と呼びます。

```
CREATE TABLE sessions_json (
      -- this is a UUID
      session_id              char(32) NOT NULL,
      session_data            json
);
```

ソース表の既存の索引は仮表に対するプロシージャの一部として自動的にコピーされるため、仮表では主キーや一意索引は定義されないことに注意してください。非常に大規模な表は、パラレルDMLやパラレル問合せから恩恵を受ける場合があります。ワークロードに余裕があり、追加の負荷に対処するのに十分なリソースがデータベース・サーバー上にある場合は、必要に応じて有効化します。

プロセスを開始できるようになりました。

```
BEGIN
 DBMS_REDEFINITION.start_redef_table(
    uname => 'DEMOUSER',
    orig_table => 'SESSIONS',
    int_table => 'SESSIONS_JSON',
    col_mapping =>
       'session_id session_id, ' ||
       'json_object(browser,operating_system,duration returning json) session_data',
    options_flag => DBMS_REDEFINITION.cons_use_pk
  );
END
;
/
```

COL_MAPPING は、このコード・スニペットの中で間違いなくもっとも重要なパラメータです。列マッピング文字列を使用して、ソース表と仮表の間で列をマッピングする方法を定義します。最初のパラメータは式で、2番目のパラメータは仮表の列の名前を示します。この例では次のとおりになります。

- SESSION_IDはSESSION_IDにマッピングされます。変更はなく、列の名前とデータタイプは同一です。
- ソース表からのリレーショナル列で構成されるJSON_OBJECT()へのコールは、仮表のSESSION_DATA列にマッピングされます。

表の依存性をコピーすることも可能です。以下の例では、一意索引と権限を仮表にコピーしますが、主キー制約やトリガーはコピーしません。COPY_STATISTICSがFALSEであるため、再定義操作後にも統計情報を手動で収集する必要があることに注意してください。これは、ソース表と仮表の表構造が異なるため、仮表の新しい列の統計がソース表に存在しないことを前提とした意識的な判断です。他のすべてのパラメータは、アプリケーションの要件に基づいて設定されます。実行中にエラーが発生した場合、IGNORE_ERRORS = FALSEにより、コード・ブロックが例外を発生させます。

```
DECLARE
   l_errors PLS_INTEGER;
```

```
BEGIN
   DBMS_REDEFINITION.copy_table_dependents(
      uname => 'DEMOUSER',
      orig_table => 'SESSIONS',
      int_table => 'SESSIONS_JSON',
      copy_indexes => DBMS_REDEFINITION.cons_orig_params,
      copy_triggers => FALSE,
      copy_constraints => FALSE,
      copy_privileges => TRUE,
      ignore_errors => FALSE,
      num_errors => l_errors,
      copy_statistics => FALSE
   );
   dbms_output.put_line('error count: ' || l_errors);

   if l_errors != 0 then
      raise_application_error(
         -20001,
         l_errors || ' encountered trying to copy table dependents'
      );
   end if; END;
/
```

コマンドが完了したら、制約を追加します。上記のケースで、追加する必要があるのは主キーだけです。

```
ALTER TABLE sessions_json
   ADD CONSTRAINT pk_sessions_json
   PRIMARY KEY ( session_id );
```

再定義プロセス中に、仮表データをソース表と同期させることができます。START_REDEF_TABLEをコールすることで再定義プロセスが開始したのち、かつ、FINISH_REDEF_TABLEがコールされてそれが終了する前に、多数のDML文がソース表で実行されている可能性があります。これに該当すると認識している場合は、仮表をソース表と定期的に同期させることを推奨します。SYNC_INTERIM_TABLE()をコールできる回数に制限はありません。

```
BEGIN
   DBMS_REDEFINITION.sync_interim_table(
      uname => 'DEMOUSER',
      orig_table => 'SESSIONS',
      int_table => 'SESSIONS_JSON',
      continue_after_errors => false
   );
END
;
/
```

開発チームは仮表に問合せを行い、構造、コンテンツ、および他の重要なプロパティが期待どおりであることを確認できます。
期待どおりであった場合は、再定義プロセスを終了できます。

```
BEGIN
   DBMS_REDEFINITION.finish_redef_table(
      uname => 'DEMOUSER',
      orig_table => 'SESSIONS',
```

```
      int_table => 'SESSIONS_JSON',
      dml_lock_timeout => 60,
      continue_after_errors => false
   );
END
;
/
```

プロンプトが返されるとすぐに、表の統計情報を収集する必要があります。

```
BEGIN
   -- table prefs define all the necessary attributes for stats gathering
   -- they are not shown here
   DBMS_STATS.gather_table_stats('DEMOUSER', 'SESSIONS');
END;
/
```

これで、表の再定義の例を終了します。

## 次のレベルの可用性：エディションベースの再定義

**エディションベースの再定義（EBR）**は、エディションを使用してアプリケーション・コードとデータ・モデル構造をバージョニングすることにより、アプリケーションの利用を中断せずにオンライン・アプリケーション・アップグレードができます。アプリケーションのアップグレードのロールアウトが完了すると、アップグレード前のバージョンのアプリケーションとアップグレード後のバージョンのアプリケーションの両方を同時にアクティブにして使用できます。

このメカニズムを使用すると、既存のセッションでは、使用が自然に終了するまでアップグレード前のアプリケーションのバージョンを継続して使用できます。また、すべての新しいセッションで、アップグレード後のアプリケーションのバージョンを使用できます。アップグレード前のアプリケーションのバージョンは、それを使用するセッションがなくなった時点で廃止できます。

EBRをこのように使用すると、アップグレード前のバージョンからアップグレード後のバージョンへのホット・ロールオーバーが、停止時間ゼロで実現します。

EBRは複数のステップで採用でき、以降のセクションで説明する最終レベルに向かって進まなくても構いません。変更に対するアプリケーションのレジリエンスを高めることができるものはすべてが成功です。

### EBRの概要

名前が示すように、**エディションベースの再定義**は、エディションに基づいています。エディションはスキーマではないオブジェクトであるため、所有者を持たず、スキーマの内部に存在することもありません。エディションは単一の名前空間で作成され、複数のエディションがデータベースに共存できます。エディションは、アプリケーションのスキーマ・オブジェクトを再定義するために必要な独立性を提供します。

パッケージ、プロシージャ、トリガー、ビューなどのデータベース・オブジェクトはすべてエディション化できます。実行の一部としてこのようなオブジェクトを使用するアプリケーションは、EBRを活用して、現在のオブジェクトを変更または削除することなく、変更されたオブジェクトを新しいエディション（またはバージョン）に導入できます。ユーザーまたはアプリケーション自体は、いつ、どのエディションを使用するかを決定でき、データベースはそれに応じてオブジェクトが正しいバージョンとなるように解決します。

表はエディション化されておらず、エディション化することはできません。表への変更が必要な場合は、代わりにエディショニング・ビューを操作します。エディショニング・ビューを作成するには、アプリケーションを停止する必要があることに注意してください。ただし、EBRをフルに活用している場合、これは、必要とされる最後の停止と考えて差し支えないでしょう。エディショニング・ビューでは、表での場合と同じようにトリガーを定義できます。ただし、一時的なcrossedition トリガー、およびINSTEAD OFトリガーを除きます。したがって、それらはエディション化できるため、エディショニング・ビューを使用すると、元表がエディション化されているかのように元表を扱うことができます。

表の構造を変更している間に他のユーザーが表内のデータを変更できるようにしなければならないシナリオでは、forward crossedition トリガーを使用して、新しい列や変更されたデータタイプがある列など、新しい構造要素にもデータを格納できます。アップグレード前の

アプリケーションとアップグレード後のアプリケーションを同時に使用（ホット・ロールオーバー）する場合は、reverse crossedition トリガーを使用して、アップグレード後のアプリケーションからのデータをアップグレード前のアプリケーションが使用できるように、古い構造要素に格納することもできます。crossedition トリガーは、アプリケーションの永続的な部分ではなく、すべてのユーザーがアップグレード後のアプリケーション・バージョンを使用するようになったら削除します。crossedition トリガーと非crossedition トリガーのもっとも重要な違いは、エディションとの相互作用の方法です。

## 採用レベル

EBRは極めて強力ですが、それゆえに使用するにあたってはある程度の複雑さを伴います。EBRはレベル別に採用でき、層が追加されるごとにアプリケーション変更に対するレジリエンスが増加します。

### レベル1

1番目の採用レベルの目標は、ライブラリ・キャッシュ・ロックを発生させずにバックエンドPL/SQL APIへの変更を可能にすることです。

PL/SQLコードなどのライブラリ・キャッシュ内のオブジェクトは、DDLロックによって変更から保護されます。現在使用中のPL/SQLストアド・プロシージャは、すべてのセッションが使用を終了するまで置換できません。データベースを静止させずにビジー状態のシステムの中核的なPL/SQL機能を置換することは、非常に困難となる可能性があります。EBRを使用すると、これははるかに簡単になります。

他のすべての変更は、EBR機能のサポートなしに実装されます。

このプロセスの例については、Oracle Database開発ガイド、セクション32.7.2を参照してください。

### レベル2

次のEBR採用レベルでは、開発者は、レベル1と同様に新しいエディションでPL/SQLの変更を実装できます。さらに、エディショニング・ビューを使用します。現行バージョンのアプリケーションは影響を受けません。言い換えると、アプリケーション・コードはエディション間のデータ・アクセスを必要とせず、アプリケーション・メンテナンス操作中に、ユーザーは再定義される表にアクセスしません。

他のすべての変更は、EBR機能のサポートなしに実装されます。

このプロセスの例については、Oracle Database開発ガイド、セクション37.7.3を参照してください。

### レベル3

レベル3の採用は、エディション間のデータ送信が必要であることを除いて、前のすべてのレベルを使用することを意味します。選択されたいくつかのケースでは、crossedition トリガーは、それらを実装するための労力が少ないところで使用されています。もっともビジーな表のみがcrossedition対応です。

他のすべての変更は、EBR機能のサポートなしに実装されます。

このプロセスの例については、Oracle Database開発ガイド、セクション37.7.4を参照してください。

### レベル4

採用レベル4のユーザーは、利用可能なEBR機能すべてを使用して、あらゆるアプリケーション変更を実行します。アプリケーションの変更を実行するために停止をまったく必要としない可能性がありますが、このマイルストーンには、テクノロジーへの確実な投資と変更管理プロセスの自動化が必要です。レベル3と4の違いは有効範囲です。レベル3を採用すると選択された少数の表のみがcrossedition対応になりますが、レベル4ではすべての表がそれに対応します。

このプロセスの例については、Oracle Database開発ガイド、セクション37.7.4を参照してください。

## 考えられるワークフロー

EBRを使用してアプリケーションをオンラインでアップグレードできるようになるには、最初に以下を準備する必要があります。

1. 適切なデータベース・ユーザーと、スキーマの適切なスキーマ・オブジェクト・タイプにエディションを有効化します。
2. アプリケーションでエディショニング・ビューを使用できるように準備します。1つまたは複数の表を使用するアプリケーションは、おのおのの表をエディショニング・ビューに対応させる必要があります。

以下の手順は、EBRを使用したアプリケーション変更をデプロイする際に考えられるワークフローを示したものです。

1. 新しいエディションを作成する。
2. 新しく作成したエディションを使用できるようにセッションを変更する。
3. アプリケーション変更をデプロイする。
4. すべてのオブジェクトが有効であることを確認する。
5. ユニット・テストおよびインテグレーション・テストを実行する。
6. 新しいエディションをすべてのユーザーに対して利用可能にし、それをデフォルトにする。

サービスを使用してOracle Databaseに接続すべきですが、すべてのサービスが同じというわけではありません。たとえば、自動生成されたサービス名は、データベース管理にのみ使用されます。すべてのアプリケーションは、アプリケーションの初期デプロイの一部として作成された専用のサービスに接続する必要があります。新しいエディションのロールアウトが完了したら、サービスのエディション・プロパティを変更して新しいエディションを指定できます。

## まとめ

Oracle Databaseには、オンラインでのスキーマ移行実行の十分な実績があります。表への列の追加、索引の作成、パーティション化の導入、およびパーティションのメンテナンス操作は、アプリケーションのアップタイムに不要な負荷をかけることなく実行できます。

エディションベースの再定義はOracle Databaseに限定された機能で、アプリケーション変更のホット・デプロイに使用できます。これを採用するのにビッグバン・アプローチは必要なく、スタガード・アプローチを使用して満足できる程度まで組み込むことができます。PL/SQL変更のみをオンラインで管理しようとする場合も、標準のアプローチと比較して膨大なメリットがある可能性があります。さらに、アプリケーションがPL/SQLで記述されたAPIを使用してフロントエンドをデータベース・バックエンドから分離している場合、それぞれのリリース・サイクル間のリンクを破棄することができます。

ORACLE

## Connect with us

**+1.800.ORACLE1**までご連絡いただくか、**oracle.com**をご覧ください。北米以外の地域では、**oracle.com/contact**で最寄りの営業所をご確認いただけます。

blogs.oracle.com facebook.com/oracle twitter.com/oracle



**著者：Oracle、Senior Principal Product Manager、Martin Bach**

貢献者：Oracle、Senior Principal Product Manager、Ludovico Caldara / Oracle、Developer Advocate、Connor McDonald / Oracle、Senior Director、Gerald Venzl