ORACLE

Oracle Database 21cで提供されている Java開発者向け機能

ミッション・クリティカルでクラウド・ネイティブな
Javaアプリケーションを設計しデプロイするための
JDBC、UCP、OJVMの新しい機能拡張

2020年12月

公開

免責事項

本文書には、ソフトウェアや印刷物など、いかなる形式のものも含め、オラクルの独占的な所有物である占有情報が含まれます。この機密文書へのアクセスと使用は、締結および遵守に同意したOracle Software License and Service Agreementの諸条件に従うものとします。本文書と本文書に含まれる情報は、オラクルの事前の書面による同意なしに、公開、複製、再作成、またはオラクルの外部に配布することはできません。本文書は、ライセンス契約の一部ではありません。また、オラクル、オラクルの子会社または関連会社との契約に組み込むことはできません。

本書は情報提供のみを目的としており、記載した製品機能の実装およびアップグレードの計画を支援することのみを意図しています。マテリアルやコード、機能の提供をコミットメント（確約）するものではなく、購買を決定する際の判断材料になさらないでください。本書に記載されている機能の開発、リリース、および時期については、弊社の裁量により決定されます。製品アーキテクチャの性質上、コードが大幅に不安定化するリスクなしに、本書に記載されているすべての機能を安全に含めることができない場合があります。

目次

はじめに

Java開発者またはアーキテクトがOracle Database 21cリリースを検討すべき理由は何でしょうか。RDBMS、組込みJVM（別名OJVM）、JDBCドライバ、Java Connection Pool、Oracle Cloud Infrastructureの新しい機能拡張は、Oracleデータベースを初めて扱う開発者のオンボーディングを容易にすること、Oracleデータベースに詳しい開発者のエクスペリエンスを改善すること、およびミッション・クリティカルでクラウド・ネイティブなJavaアプリケーションの開発とデプロイを容易にすることを目的としています。

この技術概要では、新機能によってこれらの目的がどのように達成されるかを、特に以下の点に重点を置いて説明します。

1. クラウド・データベースへの接続性に関する機能拡張
2. Java開発者のOracleデータベースに関するエクスペリエンスやオンボーディングを容易にするための、よく使用されるJavaフレームワークおよびIDEのサポート
3. クラウド・ネイティブなアプリケーション（データ駆動型マイクロサービスおよびサーバーレス機能）のサポート
4. 開発者のエクスペリエンスとミッション・クリティカルなデプロイメントを強化するための診断とトレースの機能拡張
5. ミッション・クリティカルなデプロイメント向けのパフォーマンスとスケーラビリティの機能拡張：非同期またはリアクティブの機能拡張、仮想スレッド（Project Loom）のサポート、Reactive Streams Ingestionライブラリ、GraalVMネイティブ・イメージのサポート、新機能のシャーディング・データソースなど
6. ミッション・クリティカルなデプロイメント向けの停止時間ゼロに関連する機能拡張：透過的アプリケーション・コンティニュイティ、一時停止ゼロのOJVMローリング・パッチなど

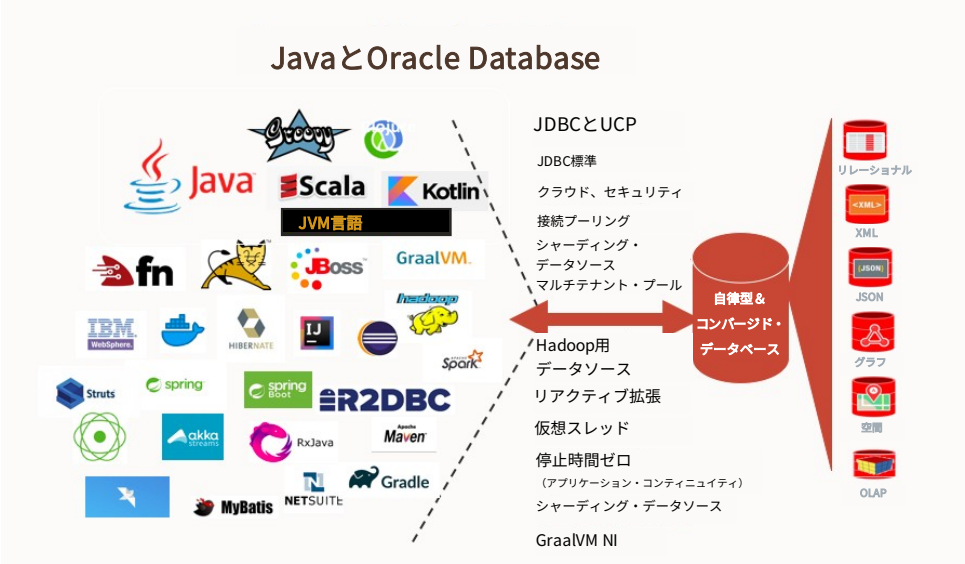

図1 JavaおよびOracleデータベースの全体像

よく使用されるJavaフレームワークおよびIDEのサポート

Javaフレームワークは多数存在しますが、オラクルは特によく使用されるものについて、Oracle JDBCドライバおよびUniversal Connection Pool（UCP）と効率的に連携させるための構成方法に着目しました。

EclipseプラグインとIntelliJ*プラグイン

Eclipseプラグインはhttps://github.com/oracle/oci-toolkit-eclipseで入手できます。詳しくは、関連するブログ記事を参照してください。

* IntelliJプラグインは本書の執筆時点でリリースに向けて作業中です。

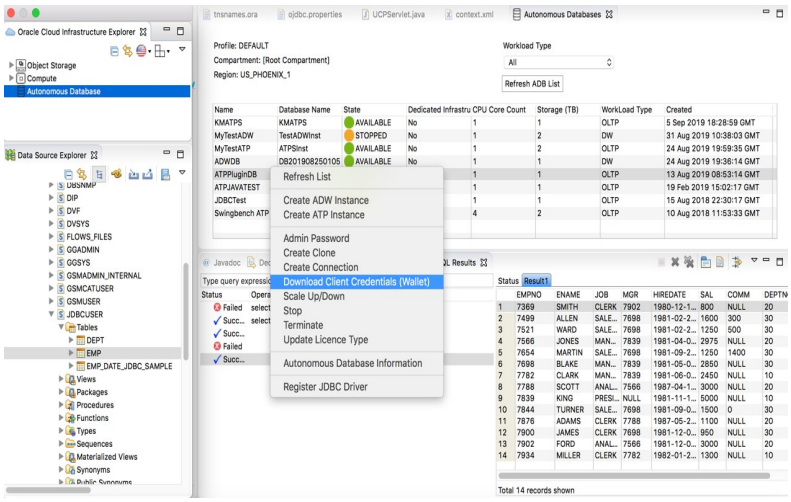

図2 Oracle Cloud InfrastructureのEclipse用ツールキット

これらのプラグインの機能セットには、新しい自律型データベースのプロビジョニング、開始/停止/クローン作成、OCPUおよびストレージのスケールアップまたはスケールダウン、管理者パスワードの変更、クラウドの資格証明のダウンロード、クラウド・データベースへの接続テスト、データベース・スキーマの参照、その他のデータベース操作の実行が含まれます。

よく使用されるフレームワークおよびJava EEアプリケーション・サーバーのサポート

1. MavenとGradleは、Java開発者によく使用されているビルド自動化フレームワークです。Oracle JDBCがMaven Central（https://bit.ly/33bpLVJ）に配置されるようになりました。詳しくは、https://bit.ly/2IBDXjJの開発者向けガイドを参照してください。また、19.8リリースより、Sprintプロジェクト（Sprint Initializr）で取得できるように、確立済みの依存関係（“フレーバーPOM”）がBOMファイルに追加されました。
2. Apache Tomcatサーブレット・エンジンの自律型データベースATPへの接続については、こちらのビデオをご覧ください。
3. Apache Hadoop：Oracle Datasource for Hadoopによって、Oracleデータベースの表をHadoop外部表に変換できます。
4. Hibernate：Hibernateの構成とUCPとの連携の方法については、https://bit.ly/32WrYUNを参照してください。
5. MyBatis：MyBatisの構成と連携の方法については、https://bit.ly/2IRgpGYを参照してください。
6. Spring：コードを追加せずに、Java接続プール（UCP）をSpringデータソースとして構成できるようになりました。Springがアプリケーションのプロパティ・ファイルから構成を取得し、それらの値をデータソースに自動的に関連付け（注入）します。

```
spring.datasource.url=jdbc:oracle:thin:@host:1521/myservice
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
spring.datasource.type=oracle.ucp.jdbc.UCPDataSource
spring.datasource.ucp.connection-factory-class-
name=oracle.jdbc.replay.OracleDataSourceImpl
spring.datasource.ucp.sql-for-validate-connection=select * from dual
spring.datasource.ucp.connection-pool-name=connectionPoolName1
spring.datasource.ucp.initial-pool-size=15
spring.datasource.ucp.min-pool-size=10
spring.datasource.ucp.max-pool-size=30
```

また、SpringBootによるUCPの構成方法については、https://bit.ly/3nDr6Mrを参照してください。

7. R2DBC：オープン・ソースのOracle-R2DBCドライバが数か月後にリリースされる予定です。
8. GraalVM：JDBCドライバがGraalVMネイティブ・イメージに対応するようになりました。詳細については後述します。

JBossデータソースとしてのUCP

このリリースでは、Java Universal Connection Pool（UCP）で、Java EEコンポーネント（サーブレット、JSP、JMS、EJBなど）が使用するためのJBossとの統合用クラスが提供されます。

- このクラスはServletContextListenerインタフェースを実装し、メソッドのオーバーライド@Override public void contextInitialized(ServletContextEvent contextEvent)をサポートします。
- このクラスは、@WebListenerアノテーションを付与するか、web.xmlで<listener>タグと<listener-class>タグを使用して手動で追加できます。
- このクラスは、アプリケーション・サーバーの起動時に実行されます。オブジェクトが構成/記述ファイルを読み取り、読み取った値を使用してデータソースを作成し、そのデータソースをJNDIアドレスにバインドし、CDIによって注入されたアプリケーション・オブジェクトにもバインドします。このJNDIアドレスとアプリケーション・オブジェクトは、コンポーネントがデータソースおよび接続を取得するために使用できます。パラメータはWeb記述子（web.xml）の<context-param>の値から、またはUCP XML構成ファイルを使用して取得されます。

以下は、ServletContextListener実装、Web記述子の一部、およびサーブレットからの使用部分に関する開発者向けサンプルです。

Web記述子の一部

```
<context-param>
        <param-name>ucp.jndiName</param-name>
        <param-value>java:/datasources/mypool_usingwl</param-value>
</context-param>
<context-param>
        <param-name>ucp.url</param-name>
        <param-value>jdbc:oracle:thin:@myhost:5521/myservice</param-value>
</context-param>
<context-param>
        <param-name>ucp.connectionFactoryClassName</param-name>
        <param-value>oracle.jdbc.replay.OracleDataSourceImpl</param-value>
</context-param>
<context-param>
        <param-name>ucp.dataSourceName</param-name>
        <param-value>myDataSource</param-value>
</context-param>
```

```
        <context-param>
                <param-name>ucp.user</param-name>
                <param-value>scott</param-value>
        </context-param>
        <context-param>
                <param-name>ucp.password</param-name>
                <param-value>*****</param-value>
        </context-param>
         . . .
</web-app>
```

UCP XML構成の使用

UCP XML構成ファイルを使用するには、JVMシステム・プロパティのoracle.ucp.jdbc.xmlConfigFileを以下のように設定します。

```
-Doracle.ucp.jdbc.xmlConfigFile=file:/Users/scott/conf/ucp_config.xml
```

次に、Web記述子（web.xml）内のucp.dataSourceNameパラメータを設定します。このパラメータは、/ucp-properties/connection-pool/data-source/data-source-name内のdata-source-attributeと一致している必要があります。

Web.xml

```
        <context-param>
                <param-name>ucp.dataSourceName</param-name>
                <param-value>myDataSource</param-value>
        </context-param>
```

ucp_config.xml

```
<ucp-properties>
      <connection-pool
      connection-factory-class-name="oracle.jdbc.replay.OracleDataSourceImpl"
      connection-pool-name="pool1"
      initial-pool-size="10"
      max-connections-per-service="15"
      max-pool-size="30"
      min-pool-size="2"
      password="******"
      url="jdbc:oracle:thin:@myhost:5521/myservice"
      user="scott">
      <connection-property name="autoCommit" value="false"></connection-property>
      <connection-property name="oracle.net.OUTBOUND_CONNECT_TIMEOUT" value="2000">
      </connection-property>
      <data-source data-source-name="myDataSource" description="pdb1" service="ac">
      </data-source>
      </connection-pool>
</ucp-properties>
```

CDI経由でデータソースを取得するサーブレットのサンプル

```
        @Inject
        @UCPResource
        private DataSource ds;
```

JNDI経由でデータソースを取得するサーブレットのサンプル

```
        @WebServlet("/OracleUcp")
        public class OracleUcp extends HttpServlet {
           private DataSource ds = null;
```

```
    // JNDIを使用してデータソースの参照を取得する
    @Override
    public void init() throws ServletException {
        Context initContext;
        try {
            initContext = new InitialContext();
            ds = (DataSource)
initContext.lookup("java:/datasources/mypool_usingwl");
```

データソースを使用するサーブレットのサンプル

```
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        // プールから接続を取得する
        try (Connection conn = ds.getConnection();
             Statement st = conn.createStatement()) {
            ResultSet rs = null;
            rs = st.executeQuery("select empno, ename, job from emp");
```

既知の制約事項：コンテナから提供される管理ツールおよび監視ツールをプールに適用することはできません

クラウド・ネイティブのアプリケーションをサポートするための機能拡張

マイクロサービス・フレームワークでのJDBCとUCPの構成

- **SpringBoot**：SpringBootプロジェクトでのOracle JDBCドライバの構成方法については https://bit.ly/3m4zQuHを、SpringでのUCPの構成方法についてはhttps://bit.ly/3nDr6Mrを参照してください。
- **Helidon**：Helidon MPとSEの両方がOracle UCPデータソースをサポートしており、提供されるすべての機能を継承しています。Helidonは、UCPデータソースのマイクロサービスへの自動注入、Kubernetes環境およびATP環境におけるOracle DB機能（マイクロサービス・アーキテクチャに直接関連する機能を含む）の使用の簡素化など、さまざまな統合機能と便利な機能を提供しています。HelidonプロジェクトでのOracle JDBCドライバの構成方法については、https://bit.ly/2IP18a5を参照してください。
- **Micronaut**：MicronautプロジェクトでのOracle JDBCドライバの構成方法については、https://bit.ly/2KEw4KXを参照してください。
- **Quarkus**：QuarkusプロジェクトでのOracle JDBCドライバの構成方法については、https://bit.ly/2J6dYR9を参照してください。

非ファイル・ベース・システムからのウォレットのロード

このリリースでは、OracleDataSource上でSSLContextを設定するための新規APIによって、メモリや非ファイル・ベース・システムからウォレットをロードする方法を開発者が完全に制御できるようになりました。

以下のコードは、このAPIの使用方法を示しています。

```
static SSLContext createSSLContext()
        throws GeneralSecurityException, IOException {
        TrustManagerFactory trustManagerFactory =
            TrustManagerFactory.getInstance("PKIX");
        KeyManagerFactory keyManagerFactory =
            KeyManagerFactory.getInstance("PKIX");

        trustManagerFactory.init(loadKeyStore());
        keyManagerFactory.init(loadKeyStore(), null);

        SSLContext sslContext = SSLContext.getInstance("SSL");
        sslContext.init(
            keyManagerFactory.getKeyManagers(),
            trustManagerFactory.getTrustManagers(),
            null);
        return sslContext;
    }

    static KeyStore loadKeyStore()
        throws IOException, GeneralSecurityException {

        // このサンプルでは標準のファイル・ベースのソースを使用していますが、
        // 非ファイル・ベースのソースからInputStreamを使用してキーストアをロードする際にも再利用できます。
        try (InputStream keyStoreStream =
                  Files.newInputStream(Paths.get("cwallet.sso"))) {
            KeyStore keyStore = KeyStore.getInstance("SSO", new OraclePKIProvider());
            keyStore.load(keyStoreStream, null);
            // ここでは、keyStore.load(KeyStore.LoadStoreParameter)も使用できます。

            return keyStore;
        }
    }
```

前リリースでのセキュリティの機能拡張

前リリースでは以下のセキュリティの機能拡張がありました。

- HTTPSプロキシ構成のサポート
- 自動プロバイダ解決（OraclePKIProvider）
- URL内でのサーバーのドメイン名（*oracle.net.ssl_server_cert_dn*）の設定
- URL内での新規ウォレット・プロパティ（my_wallet_directory）のサポート
- Key Store Service（KSS）のサポート

JDBCでのネイティブJSONデータ型のサポート

Oracle Database 21cリリースでは、ネイティブJSONデータ型と自律型JSONクラウド・サービスが提供されます。oracle.sql.jsonパッケージのJava APIは、ネイティブJSONタイプ値とそのバイナリ形式の記憶域にアクセスする、JSONタイプ値の作成、変更、問合せを実行する、JSONタイプ値をデータベースで使用されるものと同じJSONバイナリ形式にエンコーディングまたはデコーディングする、JSONタイプ値とJSONテキストを相互に変換する、JSON-Pインタフェース（javax.json.*など）を使用してJSONタイプ値をバインディングしアクセスするといった目的で使用できます。

以下のJDBCサンプル・コードは、ネイティブJSONデータ型の使用方法を示しています。

```
import java.sql.PreparedStatement; import
java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
import oracle.sql.json.OracleJsonFactory;
import oracle.sql.json.OracleJsonObject;

public class JsonExample {

    public static void main(String[] args) throws SQLException {
        OracleDataSource ds = new OracleDataSource();
        ds.setURL("jdbc:oracle:thin:@myhost:1521:orcl");
        ds.setUser(<user>);
        ds.setPassword(<password>);
        OracleConnection con = (OracleConnection) ds.getConnection();

        // JSON列を含む表を作成し、値を1つ挿入する
        Statement stmt = con.createStatement();
        stmt.executeUpdate("CREATE TABLE fruit (data JSON)");
        stmt.executeUpdate("INSERT INTO fruit VALUES ('{"name":"pear","count":10}')");

        // 別のJSONオブジェクトを作成する
        OracleJsonFactory factory = new OracleJsonFactory();
        OracleJsonObject orange = factory.createObject();
        orange.put("name", "orange");
        orange.put("count", 12);

        // orangeオブジェクトを挿入する
        PreparedStatement pstmt = con.prepareStatement("INSERT INTO fruit VALUES (:1)");
        pstmt.setObject(1, orange, OracleType.JSON);
        pstmt.executeUpdate();
        pstmt.close();

        // pearオブジェクトを取得する
        ResultSet rs = stmt.executeQuery("SELECT data FROM fruit f WHERE f.data.name =
'pear'");
        rs.next();
        OracleJsonObject pear = rs.getObject(1, OracleJsonObject.class);
        int count = pear.getInt("count");
```

```
        // pearオブジェクトの編集可能なコピーを作成する
        pear = factory.createObject(pear);
        pear.put("count", count + 1); pear.put("color", "green");

        // pearオブジェクトを更新する
        pstmt = con.prepareStatement("UPDATE fruit f SET data = :1 WHERE f.data.name =
'pear');
        pstmt.setObject(1, pear, OracleType.JSON);
        pstmt.executeUpdate();
        pstmt.close();

        rs.close();
        stmt.close();
        con.close();
    }
}
```

診断とトレースに関する機能拡張

このリリースでは、診断に関する新たな機能拡張として接続識別子が導入されました。

診断の精度向上のための接続識別子

RDBMSで個々の接続または接続グループに対して接続IDを割り当てることができます。

1. JDBC接続文字列内で名前と値のペアとして指定します（CONNECTION_ID =<value>）

```
(description =(address =(protocol=)( port =)(host=))
        (connect_data=(service_name=)(..)(CONNECTION_ID=<value>))
 )
```

2. 接続識別子（CONNECTION_ID）がログに記録されるようになります

```
“Exception in thread "main" java.sql.SQLRecoverableException:ORA-
12506, TNS:listener rejected connection based on service ACL filtering
(CONNECTION_ID=ovDT/bCGfJngU602xAph8g==)”
```

DMSメトリックとクライアント情報

classpath内に含まれるDynamic Monitoring System（DMS）関連のjar（ojdbc8dms.jar、ojdbc11dms.jar）に診断メトリックとjava.util.loggingの限定的サポートが追加されました。

JDBCでは、setClientInfo()メソッドとgetClientInfo()メソッドで、Actions、ClientId、ExecutionContextId、Module、Stateを含むエンド・ツー・エンド・メトリックによるアプリケーションのタグ付けが可能になりました。

ミッション・クリティカルなデプロイメントに対応したパフォーマンスとスケーラビリティ

新機能のJDBCリアクティブ拡張、ドライバでの仮想スレッドのサポート、Reactive Streams Ingestionライブラリ、GraalVMネイティブ・イメージ用のドライバ構成、シャーディング・データソースによって、ミッション・クリティカルなJavaアプリケーションのパフォーマンスとスケーラビリティが大幅に向上します。

ドライバでの仮想スレッドのサポート

Project Loomは、コストの低い"仮想スレッド"、限定継続、および末尾呼出しの除去を使用して"高スループットの同時実行アプリケーションの作成と維持における複雑さを軽減する"ことを目指しています。仮想スレッドは、同期呼出しが非同期呼出しと同様のパフォーマンスを発揮できるようにするものです。

Oracle Database 21cのJDBCでは、低コストのスレッドによる同期データベース・アクセス、つまり仮想スレッドを使用した標準JDBC呼出しがサポートされるようになりました（ネイティブ・メソッド呼出し、synchronizedによるイントリンシック・ロック）。言い換えれば、単純なブロッキングJDBC呼出しを複雑なリアクティブ・ストリーム呼出しにリファクタリングする必要がありません。

以下は、JDBCでの仮想スレッドの使用方法を示したサンプル・コードです。

```
// 仮想スレッドを使用する、またはカーネル・スレッドを使用するようにタスク・エグゼキュータを設定する
    final Executor taskExecutor =
        useVirtualThreads
            ? newVirtualThreadExecutor(kernelThreadExecutor)
            : kernelThreadExecutor;
    …

    // このエグゼキュータを使用して複数のJDBC呼出しを実行する
    for (int i = 0; i < taskCount; i++) {
        taskExecutor.execute(() -> {
            executeSql("SELECT * FROM emp");
            completionLatch.countDown();
        });
    }
    …

    // すべてのタスクが完了するまで、定期的にメッセージを出力する
    awaitCompletion(useVirtualThreads, taskCount, completionLatch);
```

Reactive Streams Ingestionライブラリ

Oracle Database 21cのJDBCドライバには、多数の同時実行ソースから得られる大量のデータ（センサー、通話詳細記録、ログ）をOracle Databaseに高速に取り込むための新規Javaライブラリが導入されました。

このライブラリは、着信データ・ストリームをグループに分けてバッファリングし、その後コントロールをクライアント・スレッドに戻します。一方で、別のスレッド・プールを使用して、データベース・ブロック内へのダイレクト・パス挿入によって（つまり、SQLレイヤーを迂回して）データベースI/Oを実行します。

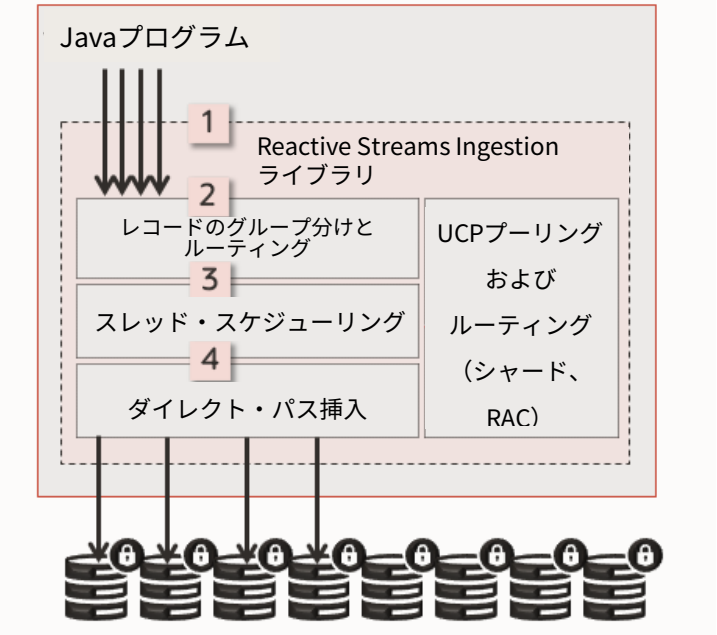

図3 Reactive Streams Ingestion

このライブラリ（rsi.jar）は次のシンプルなAPIを提供します。

- Push Publisher：もっとも単純な使用方法、バックエンドへのプレッシャーなし。
- Flow Publisher：Java Flow API（パブリッシャ、サブスクライバ、サブスクリプション）を実装。
- オブジェクト・リレーショナル・マッピング用のRecord API

以下のショート・ビデオでReactive Streams Ingestionの実例をご覧いただけます。

詳細とサンプル・コードについてはOracle JDBCのドキュメントを参照してください。

JDBCリアクティブ拡張

Oracle Database 21cのJDBCドライバは、ノンブロッキング・ネットワークI/Oによる非同期データベース・アクセスをサポートするように拡張されました。この機能拡張では、Java Flow標準のサブスクライバ型とパブリッシャ型が公開されます。また、R2DBC API[1]、および演算子（map、reduce、filters）、同時実行性モデリング、監視、トレースが提供されているリアクティブ・ストリーム・ライブラリ（Reactor、RxJava、Akka、Vert.x、Spring、jOOQ、Querydsl、Kotysaなど）との相互運用が可能なAPIが公開されます。

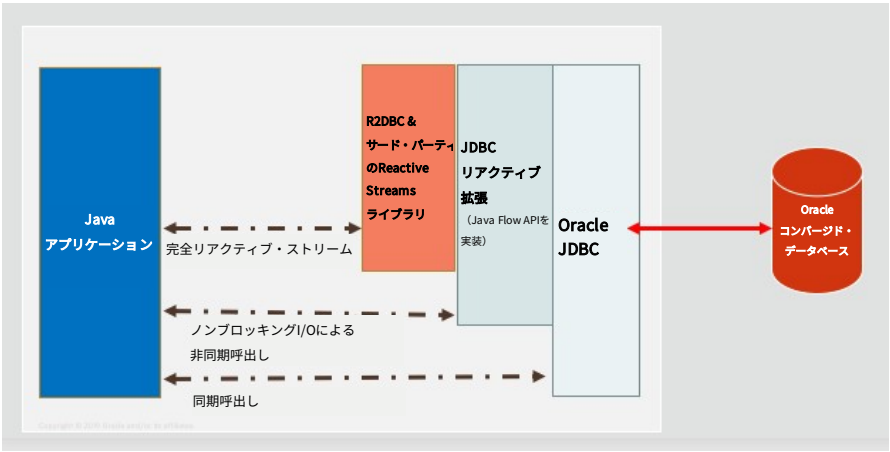

図4 Oracle JDBC 21cによるデータベース・アクセス

1 Oracle R2DBCドライバは近日Githubで入手可能になる予定です。

以下のサンプル・コードは、接続を非同期式でオープンする方法を示しています。

```
/**
 * 新しい接続を非同期式でオープンする
 * @param dataSource URL、ユーザー、パスワードを構成したデータソース
   @return A 1つの接続を発生させるパブリッシャ
 * @throws SQLException 接続をオープンする前にデータベース・
 * アクセス・エラーが発生した場合
 */
Flow.Publisher<OracleConnection> openConnection(DataSource dataSource)
    throws SQLException {
    return dataSource.unwrap(OracleDataSource.class)
        .createConnectionBuilder()
        .buildConnectionPublisherOracle();
}
```

詳しくは、Oracle JDBCのドキュメントを参照してください。

GraalVMネイティブ・イメージ向けのドライバ構成

GraalVMはミッション・クリティカルなアプリケーションの実行を高速化します（起動速度が向上する、実行時のメモリ使用量が少ないなど）。Oracle Database 21cのJDBCドライバでは、META- INF/native-image内でGraalVMを構成できるようになりました。ドライバに追加されたコードは、orai18nとxmlparserv2の関連jarに関するものであり、前者はNLSまたは国際化のサポート、後者はXML解析に使用されます。

次の手順によって、プレーンなJDBCコードのDataSourceSample.javaをネイティブ・イメージとして実行できます。

1) GraalVM Updaterツールを使用してネイティブ・イメージをインストールします。
   `gu install native-image`
2) 40行目のDB_URLをデータベースを指定するように変更し、適切なクラスパスを指定してこのJavaファイルをコンパイルします。
   `javac -cp .:./ojdbc11.jar DataSourceSample.java`
3) ネイティブ・イメージ・ビルダーを実行します。
   `native-image -cp .:./ojdbc11.jar DataSourceSample`
4) イメージを実行します（この段階でJVMは不要になります）。
   `./datasourcesample`

ドライバの機能拡張に加えて、JDBC開発チームはこのドライバと関連jarをGraalVMネイティブ・イメージに統合するために、Helidon、Micronaut、Quarkusのチームと積極的に連携してきました。

シャーディング・データソース

Java SE 9 JDBC 4.3では、ShardingKeyを作成するためのShardingKeyインタフェースとShardingKeyBuilderインタフェースが導入されましたが、これらの新規APIを使用するためには既存のアプリケーションを変更する必要がありました。
この点は多くのJava開発者にとって問題でした。
Oracle Databases 21cのJDBCドライバでは、シャード・データベースへのJavaでの接続を容易にするための新規JDBCデータソースが導入されました。
これにより、接続を要求する前にシャーディング・キーを明示的に設定する必要がなくなりました。
単一のシャード向けのSQL文では、WHERE句にシャーディング・キーを含める必要があります。
例：select id, name from customer where id = ?

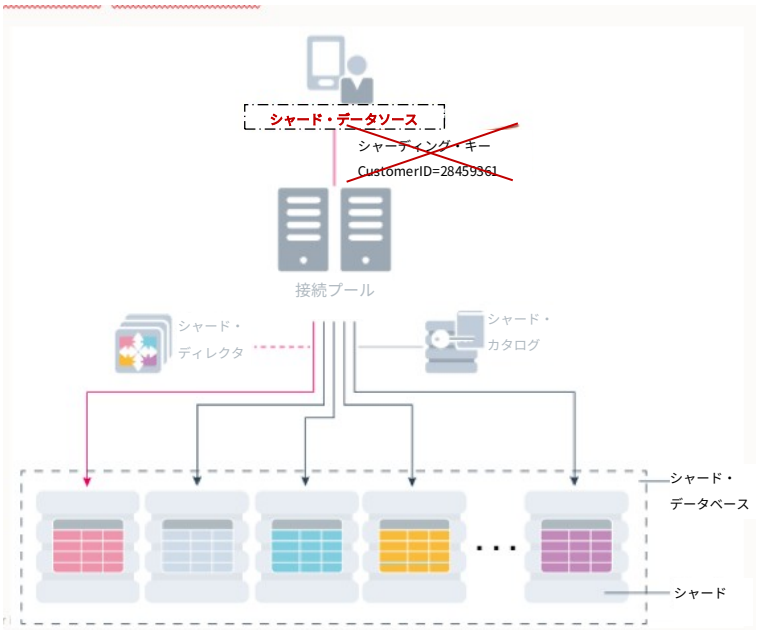

図5 シャーディング・データソース

シャーディング・データソースを使用するための要件とベスト・プラクティスを次に示します。

1) Java開発者はoracle.jdbc.useShardingDriverConnection
システム・プロパティをtrueに設定する必要があります（コード内またはJDKレベルで）。
Properties prop = new Properties(); prop.setProperty("oracle.jdbc.useShardingDriverConnection", "true");

2) Java開発者はシャード・ディレクタ（別名GSM）接続文字列を使用する必要があります。これによって接続が適切なシャードにルーティングされます。
final static String gsmURL = "jdbc:oracle:thin:@(DESCRIPTION = (ADDRESS = (HOST = ...)(PORT = ...)(PROTOCOL = tcp))(CONNECT_DATA = (SERVICE_NAME = ...)))";

3) シャーディング・データソースでは、自動コミットがオフの場合に、単一のシャードへのローカル・トランザクションがサポートされます。ただし、Java開発者はローカル・トランザクションが常に単一のシャード上で開始し終了することを指示するために、データソース・プロパティallowSingleShardTxnをTRUEに設定する必要があります。

Oracle Database 21c時点の既知の制限事項を次に示します。

- シャーディング・データソースではJDBC TYPE 4シン・ドライバのみがサポートされます。JDBC TYPE 2 OCIドライバまたはRDBMSサーバーサイドTYPE 2ドライバ（別名KPRBドライバ）はサポートされていません。
- 現時点で、シャーディング・データソースでは複数のシャードにまたがるトランザクションはサポートされていません。
- シャーディング・データソースでは、ダイレクト・パス・ロード、JDBC Dynamic Monitoring Service（DMS）メトリックなどの一部のOracle JDBC機能拡張はサポートされていません。

次のビデオで、シャーディング・データソースの実例をご覧いただけます。

停止時間ゼロのミッション・クリティカルなデプロイメント

停止時間ゼロは、ミッション・クリティカルなアプリケーションでは不可欠となる *サービス品質* です。アプリケーション・コンティニュイティ（AC）、透過的AC（TAC）、およびトランザクション・ガードが、Javaクライアント・アプリケーションのためのおもな停止時間ゼロ・メカニズムとなります。データベース内で実行されるJavaコードも、一時停止ゼロのパッチ適用とセキュリティ機能拡張の恩恵を受けられます。

アプリケーション・コンティニュイティ（AC）と透過的AC

アプリケーション・コンティニュイティ（AC）は、ドライバ・メモリ内で、アプリケーションの処理単位（通常はトランザクション）の実行中に発生したすべてのデータベース呼出しを記録し、インスタンス、ホスト、またはネットワークの障害が発生したときに、例外を隠した後、同じデータベースの別のインスタンスに対して、それらの呼出しを再実行することです。再実行時、結果（すなわち、結果セットのチェックサム）が同一の場合は、ACが正常に機能しており、問題が起きていないかのように、コントロールがアプリケーションに戻されて続行されます。同一でない場合は、例外が再キャストされ、Javaコードが表示されます。Java開発者の観点によるACおよびTACについては、こちらのブログ記事の停止時間ゼロに関するセクションで説明しています。

このリリースでは、データベース・サービスにRESET_STATEという新しい属性が追加されました。この属性は、処理単位（つまりトランザクション）の完了時にアプリケーションによって設定されたか使用された状態をクリアするものであり、Javaコードに対しては透過的です。具体的には、カーソルがキャンセルされ、PL/SQLグローバルがクリアされ、セッション一時表が切り捨てられ、セッション一時LOBがクリアされます。

データベース内のJVM - 高可用性とセキュリティの機能拡張

データベース内のJVM（別名OJVM）は、インプレース・データ処理のため、Webサービス、Hadoopサーバー、サード・パーティのデータベースおよびレガシー・システムを呼び出すため、サード・パーティJavaライブラリを実行するため、およびJavaベース言語（Jython、Groovy Kotlin、Clojure、Scala、JRuby）を実行するために使用されます。また、OJVMは、AQ – JMS、XDB、Spatial、Scheduler、Java XA、OLAPなどのデータベース・コンポーネントに使用されます。

このリリースでは、長らく待ち望まれたクラスタ化データベース（RAC）環境での一時停止ゼロとローリング・パッチ機能、およびその他のセキュリティ機能拡張が提供されます。

OJVMのRACローリング・パッチ

このリリースでは、一時停止ゼロとOJVMのRACローリング・パッチ機能が提供されます。OJVMのRACローリング・パッチは以前のリリース（18cおよび19c）でも可能でしたが、数秒の一時停止期間があり、その間はクラスタ全体でJavaが利用不可になりました。

セキュリティの機能拡張

*ロックダウン・プロファイル*は、プラガブル・データベース（PDB）とコンテナ・データベース（CDB）での特定の操作や機能を制限するメカニズムです。以前のリリースでは、次のロックダウン・プロファイルが実装されました。JavaへのOSファイル・アクセスおよびネットワーキングを無効化するもの（DB 19c）、java.lang.RuntimePermissionおよびjava.io.FilePermissionの付与を無効化するものです。

このリリースでは、次のロックダウン・プロファイルが実装されています。OSファイル・アクセスをPATH_PREFIXのパス内に限定するもの、およびOSプロセスをフォークする際のOSユーザー識別子を指定するものです。

結論

この技術概要では、Java開発者およびアーキテクトのオンボーディングやエクスペリエンスを容易にする、Oracle Database 21cの最新の機能拡張について説明しました。これらの機能拡張により、ミッション・クリティカルでクラウド・ネイティブなJavaアプリケーションの開発とデプロイも容易になります。

以下の「参考資料」セクションに、開発者向けガイドとJavadocへのリンクを記載しています。
最新のリソースについては、ランディング・ページのhttps://www.oracle.com/jdbc/をご覧ください。

参考資料

JDBC開発者ガイド

Java開発者ガイド

Universal Connection Pool開発者ガイド

JDBC Java APIリファレンス

Universal Connection Pool APIリファレンス

RAC FAN Events Java APIリファレンス