# 次世代RDB剱“Tsurugi” JDBC


## Tsurugi JDBCを作った話

剱 TSURUGI

NAUTILUS

2025年11月15日

# 目次

# 自己紹介

- 株式会社ノーチラス・テクノロジーズ
  - 菱田 真人

- Tsurugi開発での担当
  - Iceaxe（Javaライブラリー（JDBCではない））
  - Tsurugi SQLコンソール（tgsql）
  - Tsubakuro/Rust（およびTsurugi ODBCドライバー）
  - Tsurugi MCPサーバー
  - 原価計算ベンチマーク

- 個人的に、Javaに関するウェブページを制作


# Tsurugiの概要

# Tsurugi（劔）とは

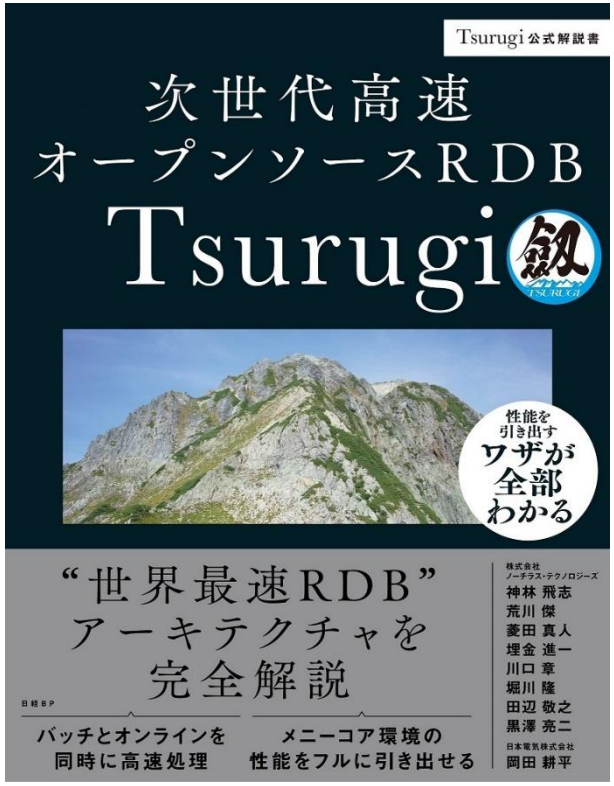
- OSSのRDBMS
  - https://www.tsurugidb.com/
  - https://github.com/project-tsurugi/tsurugidb
  - NEDOの支援を受けて、各大学研究機関・NEC・ノーチラス各社が協力して開発
  - 2023/10/5 1.0.0-BETA1公開。現時点の最新は1.7.0

- インメモリーDB
  - 現代的なメニーコア・大容量メモリーのハードウェアが対象
  - 現時点では単ノード。将来的にはレプリケーションも

- トランザクション分離レベルはSERIALIZABLE
  - Tsurugiを使う際に従来のRDBMSとは異なる考慮が必要

# Tsurugiの書籍

- 『次世代高速オープンソースRDB Tsurugi』
  - 2023/10/5のTsurugi 1.0.0-BETA1公開と同時に、Tsurugiを解説した書籍が発売




p.175に「JDBCを提供する予定はありません」と書かれているが、提供することになった

# TsurugiはインメモリーDB

- インメモリーDB
  - データは全てメモリー上に置く
    - 例外的に、BLOBはファイルとしてディスクに置いている
    - スピルアウト機能は（まだ）無い
  - 永続化としてディスクに書く
    - Tsurugiサーバー再起動時に永続化されたデータをディスクから読むが、それ以外に稼働中にディスクから読むことは無い

- ハードウェア環境
  - メニーコア（30コア以上、できれば100コア以上）
    - 少ないコア数（10コア以下）では、PostgreSQLの方が速そう
      - ただしPostgreSQLはバッチ処理とオンライン処理の混在は無理
  - 大容量メモリー（500GB以上）
  - 少量データでちょっと試すだけなら、普通のサーバーでも動く
    - Dockerイメージも提供されている

# TsurugiはRDBMS

- SQL
  - 基本的にANSI準拠・PostgreSQL似
    - 現時点ではまだ未実装の機能（構文・関数）は多い
      - https://github.com/project-tsurugi/tsurugidb/blob/master/docs/sql-features.md
- ACIDトランザクション
  - トランザクション分離レベルはSERIALIZABLE
  - Tsurugi独自のトランザクション理論（epoch・MVCCベース）
    - 書籍の第3部
- 内部はトランザクション機能付きKVS
  - 実体はMasstree（キー順に並べるツリー構造）
    - JavaのTreeMap<Pk, Record>のようなイメージ
  - 他のRDBMSだと、プライマリキーに対して暗黙にユニークインデックスが作られたりするが、Tsurugiでは自動的にプライマリキーで並ぶ

# トランザクション分離レベル

- https://ja.wikipedia.org/wiki/トランザクション分離レベル によると、「どれほどの一貫性、正確性で実行するかを4段階で定義したもの」

1. READ UNCOMMITTED
2. READ COMMITTED
   - PostgreSQL, Oracle, SQL Serverのデフォルト
3. REPEATABLE READ
   - MySQLのデフォルト
4. SERIALIZABLE
   - Tsurugi

# SERIALIZABLEとは

- SERIALIZABLE
  - 2つのトランザクションT1とT2を同時に（並列に）実行した時、順番に（T1→T2あるいはT2→T1の順で）実行したのと同じ結果になる
  - TsurugiはSERIALIZABLE（のみ）

- 例えばREAD COMMITTEDは
  - コミットされたデータを読む
  - トランザクション実行中に他トランザクションがデータを更新してコミットしたら、そのデータが読めてしまう

# SERIALIZABLE用のプログラミング

- 同時に実行されている複数のトランザクションがSERIALIZABLEの条件を満たせない場合、（SQL実行時や）コミット時にシリアライゼーションエラーが発生する（アボートする）

- シリアライゼーションエラーが発生したら、トランザクション（トランザクション内で実行されたSQL）を再実行（リトライ）する
  - リトライは（DBMSが行うのではなく、）アプリケーション側で行う必要がある
  - いわば楽観ロック

# Tsurugiのトランザクション種別（1）

- TsurugiでSERIALIZABLEを実現する為の内部プロトコルはOCC・LTXの2種類

- Tsurugiでトランザクション開始時にユーザーが指定するトランザクション種別はOCC・LTX・RTXの3種類
  - トランザクション毎に指定する（異なる種類を指定可能）
    - DB全体の設定で一括して切り替えるようなものではない

# Tsurugiのトランザクション種別（2）

- OCC（short transaction・optimistic concurrency control）
  - 実行時間が短いトランザクション（数十ミリ秒・いわゆるオンライン処理向け）
  - SQL実行時点の最新データを読む（READ COMMITTEDと同様）
  - 他トランザクションとの競合時にシリアライゼーションエラーになりやすい。OCC同士では先にコミットした方が優先される
- LTX（long transaction）
  - 実行時間が長いトランザクション（いわゆるバッチ処理向け）
  - トランザクション開始時点のデータを読む
  - LTX同士では先に始まった方が優先度が高い。一番最初に実行を開始したLTXはシリアライゼーションエラーにならない
- RTX（read only transaction）
  - 読み取り専用トランザクション（内部プロトコルではLTXの一種）
  - トランザクション開始以前のデータを読む
    - 他トランザクション終了直後にそのデータが読めるとは限らない（タイムラグがある）
  - 他トランザクションと競合しない


Tsurugiにアクセスする

# クライアントライブラリーやツール

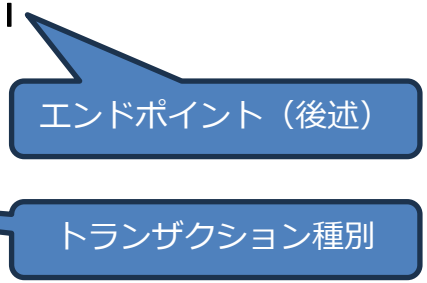
# Tsurugiにアクセスするクライアントライブラリーやツール

1. Tsurugi SQLコンソール（対話型ツール）
2. Tsubakuro/Java（Java基礎ライブラリー）
3. Iceaxe（Java高機能ライブラリー）
4. Tsubakuro/Rust（Rustライブラリー）
5. Tsurugi ODBCドライバー
6. Tsurugi MCPサーバー（自然言語でDBアクセス）
7. Embulkプラグイン（ETLツール）
8. PostgreSQL FDW（JDBCドライバー）
9. Tsurugi JDBC（JDBCドライバー）

# Tsurugi SQLコンソール（tgsql）（書籍の第6章）

- TsurugiのDBに対してSQLを実行するCLIツール
  - Tsubakuro/Java（後述）を使っている
    - JavaなのでLinuxでもWindowsでも使用可能

```
$ $TSURUGI_HOME/bin/tgsql -c ipc:tsurugi
tgsql> begin;
transaction started. option=[
  type: OCC
]
Time: 48.87 ms
tgsql> select * from tb;
[v: INT]
[10]
(1 row)
Time: 24.594 ms
tgsql> commit;
transaction commit(DEFAULT) finished.
Time: 7.054 ms
```



# Tsubakuro/Java(書籍の第11章)

書籍では『Tsubakuro』

- JavaでTsurugiにアクセスする為の通信ライブラリー
  - 基礎的な通信ライブラリー（Java11）
    - 様々な機能を持っている
      - SQL実行（JDBCではない）
      - KVSアクセス（まだ非公開）
      - データストア（テーブル単位のバックアップ・リストア）
      - …
    - 非同期API（使いやすさより実行効率優先）

他のJavaライブラリーやツールはTsubakuro/Javaを使っている

Tsurugiの全ての機能を使えるようにすることが目的なので、JDBCではない

  - ソースコード
    - https://github.com/project-tsurugi/tsubakuro
  - jarファイル（Mavenセントラルリポジトリー）
    - https://central.sonatype.com/search?q=tsubakuro

# Tsubakuro/Java（Java通信ライブラリー）の例

```
try (var session = SessionBuilder.connect("tcp://localhost:12345").create();
        var sqlClient = SqlClient.attach(session)) {
    var option = TransactionOption.newBuilder()
            .setType(TransactionType.SHORT).build();
    try (var transaction = sqlClient.createTransaction(option).await()) {
            transaction.executeStatement("update tb set v = 20").await();
            transaction.commit().await();
    }
}
```

1. Sessionを生成する
2. SqlClient等のクライアントを生成する
3. createTransactionやexecuteStatementやcommitはFutureResponseを返す

# Tsubakuro/Javaのinsertの例

```
var sql = "insert into customer values(:id, :name, :age)";
var placeholder = List.of(
    Placeholders.of("id", AtomType.INT8),
    Placeholders.of("name", AtomType.CHARACTER),
    Placeholders.of("age", AtomType.INT4));

try (var ps = sqlClient.prepare(sql, placeholder).await()) {
    var parameter = List.of(
        Parameters.of("id", 1L),
        Parameters.of("name", "tsurugi"),
        Parameters.of("age", 0));
    transaction.executeStatement(ps, parameter).await();
}

transaction.commit().await();
```

# Tsubakuro/Javaのselectの例

```
var sql = "select * from customer";
try (var rs = transaction.executeQuery(sql).await()) {
    var columnList = rs.getMetadata().getColumns();
    while (rs.nextRow()) {
        for (int i = 0; rs.nextColumn(); i++) {
            var column = columnList.get(i);
            String name = column.getName();
            var value = switch (column.getAtomType()) {
                case INT4 -> rs.fetchInt4Value();
                case INT8 -> rs.fetchInt8Value();
                case CHARACTER -> rs.fetchCharacterValue();
                default -> throw new AssertionError(column);
            };
            System.out.printf("%s=%s%n", name, value);
        }
    }
}
transaction.commit().await();
```

# Iceaxe（Javaライブラリー）（書籍の第8章）

- TsurugiでSQLを実行するJavaライブラリー
  - Tsubakuro/Javaをラップして、便利な機能を提供している
    - 同期API
    - SQL実行機能のみ（将来的にはKVSアクセスにも対応する予定）
      - IceaxeもJDBCではない
    - シリアライゼーションエラー発生時にリトライする機能を持つ
  - ソースコード
    - https://github.com/project-tsurugi/iceaxe
  - jarファイル（Mavenセントラルリポジトリー）
    - https://central.sonatype.com/artifact/com.tsurugidb.iceaxe/iceaxe-core

```
implementation 'com.tsurugidb.iceaxe:iceaxe-core:1.12.0'
```

# IceaxeとJDBCのクラスの対比

| Iceaxe | Tsubakuro/Java | JDBC |
|---|---|---|
| TsurugiConnector | （Connector） | （Driver） |
| TsurugiSession | Session | Connection |
| TsurugiTransaction | Transaction | Connection |
| TsurugiSqlQuery<br>TsurugiSqlStatement | | Statement |
| TsurugiSqlPreparedQuery<br>TsurugiSqlPreparedStatement | PreparedStatement | PreparedStatement |
| TsurugiQueryResult | ResultSet | ResultSet |
| TsurugiResultEntity | | |

- JDBCのConnectionはSessionとTransactionを兼ねている
  - Connectionは1接続1トランザクションだが、Tsurugiでは1セッションで複数トランザクションが可能

# Iceaxeの例

```
var connector = TsurugiConnector.of("tcp://localhost:12345");
try (var session = connector.createSession()) {
    try (var ps = session.createStatement("update tb set v = 20")) {
        var setting = TgTmSetting.ofAlways(TgTxOption.ofOCC());
        var tm = session.createTransactionManager(setting);
        tm.execute(transaction -> {
            transaction.executeAndGetCount(ps);
        });
    }
}
```

- TransactionMangerを生成する
  - その中でTransactionインスタンスが生成される
  - tm.execute()に渡すラムダ式に、トランザクション1回分の処理を記述する
    - シリアライゼーションエラーが発生したときに再度実行される

# Iceaxeのinsertの例

```
var sql = "insert into customer values(:id, :name, :age)";
var mapping = TgParameterMapping.of(CustomerEntity.class)
    .addLong("id", CustomerEntity::getId)
    .addString("name", CustomerEntity::getName)
    .addInt("age", CustomerEntity::getAge);

try (var ps = session.createStatement(sql, mapping)) {
    tm.execute(transaction -> {
        var entity = new CustomerEntity();
        entity.setId(1);
        entity.setName("tsurugi");
        entity.setAge(0);

        transaction.executeAndGetCount(ps, entity);
    });
}
```

# Iceaxeのselectの例

```
var sql = "select c_id,c_name,c_age from customer";
var mapping = TgResultMapping.of(CustomerEntity::new)
    .addLong("c_id", CustomerEntity::setId)
    .addString("c_name", CustomerEntity::setName)
    .addInt("c_age", CustomerEntity::setAge);

try (var ps = session.createQuery(sql, mapping)) {
    List<CustomerEntity> entityList = tm.execute(transaction -> {
        return transaction.executeAndGetList(ps);
    });

    for (var entity : entityList) {
        System.out.println(entity);
    }
}
```

# Tsubakuro/Rust

Tsubakuro/Rustが作られたことにより、『Tsubakuro』はクライアントライブラリーの総称という位置付けに変わり、従来のTsubakuroは『Tsubakuro/Java』と呼ぶことになった。

- Rust版のクライアントライブラリー
  - https://github.com/project-tsurugi/tsubakuro-rust
  - Tsubakuro/Javaからの移植だが、TCP接続のみ・SQL関連機能のみ
- C ABI形式の関数群を提供

以前「C++ライブラリーを提供予定」と言っていたものが、これ。

  - これにより、C言語のライブラリーが呼べる言語から使用できる
    - PHPのPDO TSURUGI
      - https://github.com/SakiTakamachi/pdo_tsurugi
    - Pythonから呼ぶライブラリーは無い（と思う）が、呼び出せる
- ODBCドライバー（C言語の関数群）を提供

# Tsurugi ODBCドライバー

- ODBC3
  - ODBC2には未対応
- ソースコード
  - https://github.com/project-tsurugi/tsubakuro-rust/tree/master/tsubakuro-rust-odbc
- Windows用インストーラー（ダウンロード元）
  - https://github.com/project-tsurugi/tsubakuro-rust/releases
    - インストーラーはInno Setupで作成

# Tsurugi MCPサーバー

- TsurugiのテーブルがMCPサーバー
  - LLM（いわゆるAI）から呼ばれる
- ソースコード
  - https://github.com/project-tsurugi/tsurugi-mcp-server
    - MCPのJava SDKで作成
    - TsurugiへのアクセスにはIceaxeを使用
- 日本語で指示してDBを操作（SQLを生成・実行）
  - 「Tsurugiのテーブル一覧を表示して」→listTables
  - 「XXXテーブルの定義を教えて」→getTableMetadata
  - 「XXXテーブルの中を見せて」→select文
  - 「XXXをYYYで更新して」→update文（扱いには注意！）

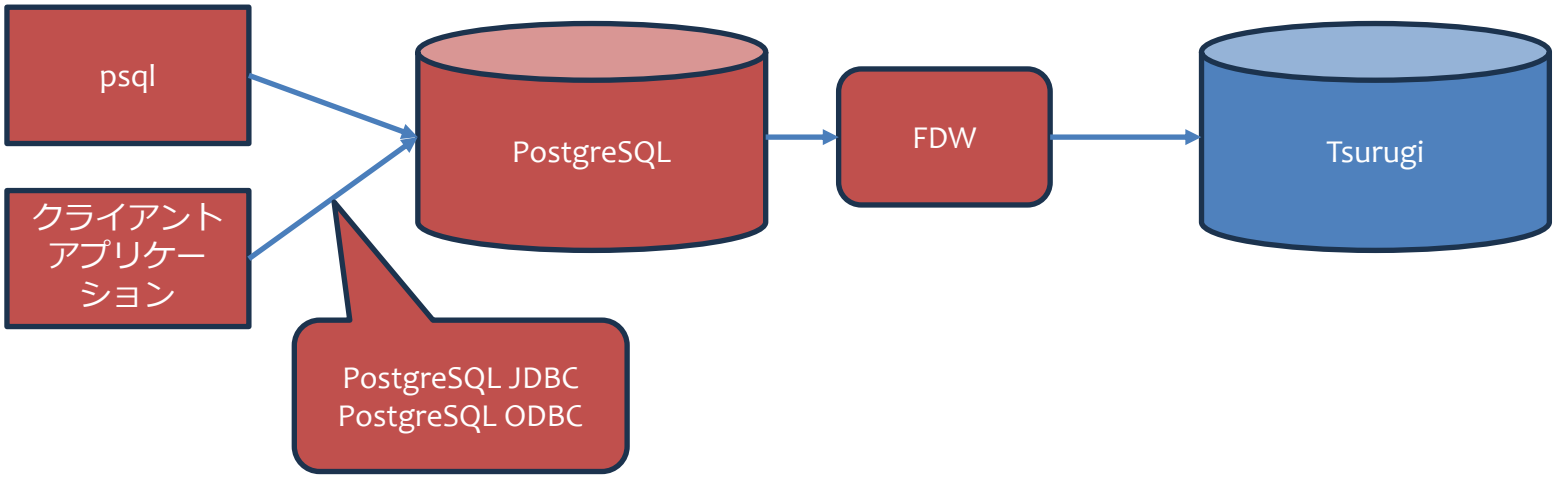
# Embulkプラグイン

- Embulkは、OSSのETLツール
  - プラグインで各種RDBMS等に対応している

- embulk-input-tsurugidb
- embulk-output-tsurugidb
  - Tsurugiの公式ツールではない
    - https://github.com/hishidama/embulk-input-tsurugidb
    - https://github.com/hishidama/embulk-output-tsurugidb
  - Embulk0.10以降
    - Embulkの公式な対応JavaバージョンはJava8だが、Java11が必要
  - configの書き方はPostgreSQLのEmbulkプラグインとほぼ互換

# PostgreSQL FDW（書籍の第7章）

- PostgreSQL経由でTsurugiにアクセスする
  - NECが開発
  - PostgreSQLのForeign Data Wrapper機能を利用
    - PostgreSQLのテーブルを操作すると、裏でTsurugiにアクセスする
  - psqlやPostgreSQLのJDBC・ODBCが使用可能
    - 大量データを扱うのは強く非推奨



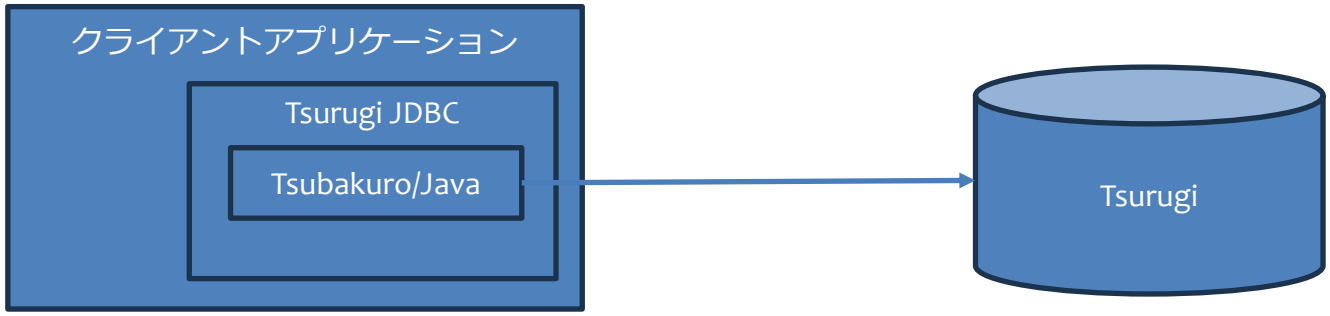
# PostgreSQL FDW経由のJDBCの例

- https://github.com/project-tsurugi/tsurugi_fdw/blob/master/sample/jdbc-sample/module/src/main/java/com/tsurugidb/fdw/jdbc/sample/JdbcSample.java

```
String url = "jdbc:postgresql://localhost:5432/postgres";
Connection connection = DriverManager.getConnection(url);

connection.setAutoCommit(false); // 以前は「SELECT tg_start_transaction()」

String sql = "INSERT INTO fdw_sample (col, tm) VALUES (?, ?)";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setInt(1, 1);
ps.setTime(2, Time.valueOf(LocalTime.now()));
ps.executeUpdate();

connection.commit(); // 以前は「SELECT tg_commit()」
```

# Tsurugi JDBC

- Tsurugiに直接アクセスするJDBCライブラリー
  - ノーチラス・テクノロジーズがサポート
  - Tsurugi 1.7.0（2025/10/30）で公開
  - 内部ではTsubakuro/Javaを呼び出し、Tsurugiに直接接続する




  - まだ開発初期段階のため、今後挙動が変わる可能性がある

# JDBCライブラリーの比較

| 観点 | PostgreSQL FDW | Tsurugi JDBC |
|---|---|---|
| ユーザーから見えるDB | PostgreSQL | Tsurugi |
| JDBCドライバー | PostgreSQL JDBC | Tsurugi JDBC |
| 操作方法 | PostgreSQLのテーブルを操作すると、FDW経由でTsurugiのテーブルにアクセスされる | Tsurugiのテーブルを（直接）操作する |
| 事前準備 | PostgreSQLのテーブルとTsurugiのテーブルを紐づけておく | 不要 |
| 開発・商用サポート | NEC<br><br>（Tsurugiに接続するFDWを開発） | ノーチラス・テクノロジーズ<br>（JDBCドライバーを開発） |

# TsurugiにアクセスするJavaライブラリーの位置付け

- Tsubakuro/Java
  - Tsurugiに接続する基礎となるライブラリー・高速
  - 主な用途：ミドルウェア開発
- Iceaxe
  - 高機能
  - Tsubakuro/Javaを使用
  - 主な用途： Tsurugiの特性を生かしたアプリケーション開発
- Tsurugi JDBC
  - JDBC準拠（Java標準）
  - Tsubakuro/Javaを使用
  - 主な用途： Javaエコシステムを利用した開発


Tsurugi特有の機能の使用方法

# Tsurugi JDBCの使用方法

# Tsurugi JDBC

- Java11（JDBC 4.3）
  - 現時点ではBLOB,CLOB未対応
- ソースコード
  - https://github.com/project-tsurugi/tsurugi-jdbc
- 日本語ドキュメント
  - https://github.com/project-tsurugi/tsurugi-jdbc/blob/master/docs/usage_ja.md
- jarファイル（Mavenセントラルリポジトリー）
  - https://central.sonatype.com/artifact/com.tsurugidb.jdbc/tsurugi-jdbc

```
implementation 'com.tsurugidb.jdbc:tsurugi-jdbc:0.1.0'
```

# JDBC URL

- 接頭辞は「jdbc:tsurugi:」
  - その直後にエンドポイントを指定する
- エンドポイント
  - TsurugiのDBサーバーの接続先・接続方法
  - 接続方法は、現時点ではIPC接続とTCP接続の2種類
    - IPC接続
      - Linuxの共有メモリーを介してデータを送受信する方式
      - TCP接続より高速だが、Tsurugiサーバーと同一のマシン上でしか使用できない
      - JDBC URLの例）jdbc:tsurugi:ipc:tsurugi
    - TCP接続
      - TCP/IPのソケット通信を使ってデータを送受信する方式
      - IPC接続より低速だが、どのマシンからでも接続可能
      - JDBC URLの例）jdbc:tsurugi:tcp://localhost:12345

# ユーザー認証

- Tsurugi 1.6.0から利用可能
- 現時点のTsurugiでは、デフォルトでは認証不要
- 認証を有効にした場合、認証方法は以下の3種類
  - ユーザー・パスワード
    - user, password
  - 認証トークン
    - authToken
  - 認証ファイル
    - credentials
      - 認証ファイルのパスを指定する

# Tsurugi JDBCでユーザー認証を指定する例

- クエリー文字列で指定する

```
var url =
"jdbc:tsurugi:tcp://localhost:12345?user=tsurugi&password=password";
var connection = DriverManager.getConnection(url);
```

- プロパティーで指定する

```
var url = "jdbc:tsurugi:tcp://localhost:12345";
var info = new Properties();
info.setProperty("user", "tsurugi");
info.setProperty("password", "password");
var connection = DriverManager.getConnection(url, info);
```

- メソッドの引数で指定する

```
var url = "jdbc:tsurugi:tcp://localhost:12345";
var connection = DriverManager.getConnection(url, "tsurugi",
"password");
```

# トランザクションオプション

- Tsurugiでは、トランザクション開始時にトランザクションオプションを指定する
  - 特にトランザクション種別は必須
    - OCC – 実行時間が短いトランザクション
    - LTX – 実行時間が長いトランザクション
      - 更新対象テーブルのテーブル名をwrite preserveで指定する
    - RTX – 読み取り専用トランザクション
      - スキャン分割数（並列数）をscan parallelで指定可能

- Tsurugi JDBCでは、事前にトランザクションオプションを登録しておき、トランザクション開始時に使用する
  - デフォルトのトランザクション種別はOCC

# LTXを初期設定する例

- クエリー文字列で指定する

```
var url = "jdbc:tsurugi:tcp://localhost:12345?
transactionType=LTX&writePreserve=tb1,tb2";
var connection = DriverManager.getConnection(url);
```

- プロパティーで指定する

```
var url = "jdbc:tsurugi:tcp://localhost:12345";
var info = new Properties();
info.setProperty("transactionType", "LTX");
info.setProperty("writePreserve", "tb1,tb2");
var connection = DriverManager.getConnection(url, info);
```

  - プロパティーの値はひとつの文字列なので、write preserve はテーブル名をカンマ区切りで指定する

# LTXをConnectionで設定する例

- プロパティーで指定する

```
connection.setClientInfo("transactionType", "LTX");
connection.setClientInfo("writePreserve", "tb1,tb2");
```

- ダウンキャストしてセッターメソッドで指定する

```
var tsurugiConnection = connection.unwrap(TsurugiJdbcConnection.class);
tsurugiConnection.setTransactionType(TsurugiJdbcTransactionType.LTX);
tsurugiConnection.setWritePreserve(List.of("tb1", "tb2"));
```

  - トランザクション実行中に変更した場合は、次のトランザクション開始時から新しい設定が使われる

# RTXを初期設定する例

- クエリー文字列で指定する

```
var url = "jdbc:tsurugi:tcp://localhost:12345?
transactionType=RTX&scanParallel=8";
var connection = DriverManager.getConnection(url);
```

- プロパティーで指定する

```
var url = "jdbc:tsurugi:tcp://localhost:12345";
var info = new Properties();
info.setProperty("transactionType", "RTX");
info.setProperty("scanParallel", "8");
var connection = DriverManager.getConnection(url, info);
```

# RTXをConnectionで設定する例

- プロパティーで指定する

```
connection.setClientInfo("transactionType", "RTX");
connection.setClientInfo("scanParallel", "8");
```

- ダウンキャストしてセッターメソッドで指定する

```
var tsurugiConnection = connection.unwrap(TsurugiJdbcConnection.class);
tsurugiConnection.setTransactionType(TsurugiJdbcTransactionType.RTX);
tsurugiConnection.setTransactionScanParallel(8);
```

- setReadOnly(true) – RTXにする
- setReadOnly(false) – OCCにする

# その他のオプション

- Tsurugi JDBCにはトランザクションオプションの他にもオプションがあるが、設定方法はトランザクションオプションと同様
  - コミットオプション
    - コミット種別のデフォルトはDEFAULT（サーバー側の設定に従う）
  - シャットダウンオプション
    - シャットダウン種別のデフォルトはGRACEFUL（Connectionクローズ時、実行中のリクエストがあったら、それらが終了するのを待つ）

- 設定できるオプションはTsurugiConfig.javaで定義されている

# Statement（SQLの実行方法）

- Statementの作り方やSQLの実行方法は、通常のJDBCと同じ
  - Tsurugiはテーブル名・カラム名の大文字小文字を区別する
    - SQL文の語や関数名は区別しない
  - JDBCのSQLエスケープ構文には未対応
    - {d '2025-11-15'} とか {t '11:00:00'} みたいな
- PreparedStatementについては、基本的には通常のJDBCと同様に使用できるが、Tsurugi JDBC特有の制限がある
- CallableStatement関連は使用できない
  - 現在のTsurugiにはプロシージャーが無いため

# PreparedStatement

- PreparedStatementにセットするパラメーターの型とTsurugi内で処理されるプレースホルダーのデータ型は完全に一致している必要がある
  - 例えばBIGINTのカラム（Javaではlong）に対し
    - setLong(2, 123) …OK
    - setObject(2, 123L) …OK
    - setDouble(2, 123) …NG
    - setInt(2, 123) …本当は想定と違うんだけど通る
    - setBigDecimal(2, BigDecimal.valueOf(123)) …同上
    - setBigDecimal(2, BigDecimal.valueOf(123.4)) …NG

Tsubakuro/JavaとTsurugi間の通信データとしては、整数は一種類なため？

# PreparedStatementのデータ型

| PreparedStatementのメソッド | 対象となるTsurugiの型 |
|---|---|
| setInt() | INT |
| setLong() | BIGINT |
| setFloat() | REAL |
| setDouble() | DOUBLE |
| setBigDecimal() | DECIMAL |
| setString() | CHAR, VARCHAR |
| setBytes() | BINARY, VARBINARY |
| setDate() | DATE |
| setTime() | TIME |
| setTime(Calendarあり) | TIME WITH TIME ZONE |
| setTimestamp() | TIMESTAMP |
| setTimestamp(Calendarあり) | TIMESTAMP WITH TIME ZONE |

日時を扱う場合は、java.timeのクラスをsetObject()で入れる方が確実

# ParameterMetaData

- PreparedStatement#getParameterMetaData()でParameterMetaDataが取得できる、が

```
var sql = "insert into test values(?, ?)";
try (var ps = connection.prepareStatement(sql)) {
    var metaData = ps.getParameterMetaData();
    int count = metaData.getParameterCount();
    System.out.printf("count=%d%n", count); // count=0
    for (int i = 1; 1 <= count; i++) {
        int sqlType = metaData.getParameterType(i);
        System.out.printf("%d=%d%n", i, sqlType);
    }
}
```

- Tsurugi JDBCの場合、prepareStatement()の直後のメタデータはパラメーター数が0

# PreparedStatementの内部実装(1)

- TsubakuroのJavaのPreparedStatementを使用
  - 単純名が全く同じなので紛らわしい
    - ScalaやRustのようにimport時に別名を付けたい…
  - 以下のように使う

```
var sql = "insert into test values(?, ?)";
var placeholderList = List.of(
        Placeholders.of("1", AtomType.INT4),  // INT
        Placeholders.of("2", AtomType.INT8)); // BIGINT
try (var lowPs = sqlClient.prepare(sql, placeholderList).await()) {
    var parameterList = List.of(
            Parameters.of("1", 100), Parameters.of("2", 1234L));
    lowTransaction.executeStatement(lowPs, parameterList).await();
}
```

  - PS生成時にプレースホルダー（データ型）一覧が必要になる
  - しかしJDBCのprepareStatement()時点では分からない

# PreparedStatementの内部実装(2)

- SQLが実行されるまで、Tsubakuro/JavaのPreparedStatementの生成を遅延させる

```
var sql = "insert into test values(?, ?)";
try (var ps = connection.prepareStatement(sql) {
    ps.setInt(1, 100);
    // placeholderList.add(Placeholders.of("1", AtomType.INT4);
    // parameterList.add(Parameters.of("1", 100));
    ps.setLong(2, 1234L);
    // placeholderList.add(Placeholders.of("2", AtomType.INT8);
    // parameterList.add(Parameters.of("2", 1234L));
    ps.executeUpdate();
    // lowPs = sqlClient.prepare(sql, placeholderList).await();
    // lowTransaction.executeStatement(lowPs, parameterList).await();
}
```

  - したがって、preparedStatement()直後はパラメーターの情報が無い
  - また、型情報もAtomTypeの分しか取れない

# シリアライゼーションエラー

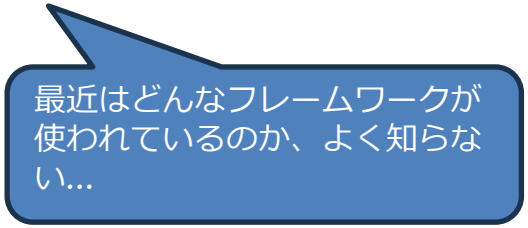
- Tsurugiでは（トランザクション分離レベルがSerializableのRDBMSでは）、コミット時やSQL実行時にシリアライゼーションエラーが発生することがある
  - selectのみのトランザクションでも、コミットしてシリアライゼーションエラーが発生しないことを確認する必要がある
- Tsurugi JDBCではシリアライゼーションエラー発生時SQLTransactionRollbackExceptionをスローする
  - SQLStateは40001（serialization failure）
- シリアライゼーションエラーが発生したらアプリケーション側（フレームワーク）でリトライする
  - JDBC側ではリトライできない

# DatabaseMetaData

- キーワード（予約語）の取得や使用できる関数一覧の取得などは未対応
  - ODBCにも同様の関数があるが、同じく未対応
  - これらのデータはTsurugiのバージョンによって増減があるのでDBサーバー側から返してほしいが、その機能は無い

# JDBCを使用するフレームワーク

- プログラマーが直接JDBC APIを使用することは少ない
- JDBCを使用するフレームワークに対して簡易的なテストを実施
  - ひとまずそのままで使えそう
    - MyBatis
    - Spring JDBC
    - Spring Data JPA（ただし、裏でHibernateを使用）
    - Spring Data JDBC（設定はPostgreSQLのものを使用）
  - Tsurugi専用の実装が必要そう
    - Hibernate



# Hibernate Dialect

- HibernateのTsurugi用Dialect（最小限の機能）
- ソースコード
  - https://github.com/project-tsurugi/tsurugi-jdbc/tree/master/modules/tsurugi-hibernate
- jarファイル（Mavenセントラルリポジトリー）
  - https://central.sonatype.com/artifact/com.tsurugidb.jdbc/tsurugi-hibernate

```
implementation 'com.tsurugidb.jdbc:
tsurugi-hibernate:0.1.0'
```

  - Dialectのクラス名は com.tsurugidb.hibernate.TsurugiDialect

NAUTILUS


JDBCを実装した感想

# Tsurugi JDBCの内部実装

# トランザクション分離レベルの変更

- Connection#setTransactionIsolation()でトランザクション分離レベルを設定できる
  - JDBC4.3仕様 10.2.1章 によると、より高い（より制約の厳しい）分離レベルに置き換えることが許されている
- Tsurugiのトランザクション分離レベルは一番厳しいシリアライザブル（のみ）である

```
@Override
public void setTransactionIsolation(int level) throws SQLException {
    // Tsurugi is TRANSACTION_SERIALIZABLE only
}

@Override
public int getTransactionIsolation() throws SQLException {
    return TRANSACTION_SERIALIZABLE;
}
```

# instanceof

- ResultSetやPreparedStatementでは、値を別の型に変換する必要があるので、instanceofを多用

```
if (value instanceof Number) {
    return ((Number)value).intValue();
}
```

instanceofとキャストに同じ型を書く必要がある
（コピペ修正漏れが発生しても、コンパイルエラーにならない）

```
// Java16
if (value instanceof Number number) {
    return number.intValue();
}
```

```
// Java21
return switch (value) {
    case Number number -> number.intValue();
};
```

  - 残念ながら、Tsurugi JDBCはJava11

# DatabaseMetaData#getColumns()

- テーブルのカラム定義をResultSetで返すメソッド
  - TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME, DATA_TYPE, TYPE_NAME, COLUMN_SIZE, BUFFER_LENGTH, DECIMAL_DIGITS, NUM_PREC_RADIX, …

1. Javadocに「BUFFER_LENGTH is not used」
2. なぜ使われないカラムが用意されているのか？
3. でもカラム名には見覚えがある…
4. ODBCのSQLColumns関数と同じ！

- ODBCでは、値を格納するバッファーを用意するためにバイト数を返すBUFFER_LENGTHがあるが、JDBCには不要

# DatabaseMetaData#xxxCaseIdentifiers()

- Tsurugiでは、テーブル名やカラム名は大文字小文字を区別して処理し、そのまま格納する
  - supportsMixedCaseIdentifiers() - true
    - 大文字小文字を区別して処理し、大文字小文字混在で格納するか
  - storesUpperCaseIdentifiers() - false
    - 大文字小文字を区別しないで処理し、大文字で格納するか
  - storesLowerCaseIdentifiers() - false
    - 大文字小文字を区別しないで処理し、小文字で格納するか
  - storesMixedCaseIdentifiers() - false
    - 大文字小文字を区別しないで処理し、大文字小文字混在で格納するか
- Hibernateのデフォルトは下3つだけで判定し、全部falseだと大文字に変換したSQLを生成する

# java.sql.DateとLocalDateとの変換

- JDBCはjava.sql.Dateを使うが、Tsubakuro/JavaはLocalDateなので、変換が必要
- java.sql.Date.valueOf(localDate)がある、が…

```
jshell> java.sql.Date.valueOf(LocalDate.of(2025, 11, 15))
$1 ==> 2025-11-15

jshell> $1.toLocalDate()
$2 ==> 2025-11-15

jshell> java.sql.Date.valueOf(LocalDate.of(-1, 1, 1))
$3 ==> 0002-01-01

jshell> $3.toLocalDate()
$4 ==> 0002-01-01
```

- java.sql.Dateは内部ではエポックで保持しているので、エポックを使って変換する事にした

# java.sql.TimeとLocalTimeとの変換

- JDBCはjava.sql.Timeを使うが、Tsubakuro/JavaはLocalTimeなので変換が必要
- だが、エポックを使って変換すると…

```
var zone = ZoneId.systemDefault();
var time = LocalTime.of(12, 30, 59);
var t1 = new java.sql.Time(TimeUnit.SECONDS.toMillis(
  ZonedDateTime.of(LocalDate.EPOCH, time, zone).toEpochSecond()));
var t2 = new java.sql.Time(TimeUnit.SECONDS.toMillis(
  ZonedDateTime.of(LocalDate.now(), time, zone).toEpochSecond()));
assertEquals(t1, t2);
↓
AssertionFailedError:
expected: java.sql.Time@298a5e20<12:30:59>
 but was: java.sql.Time@2a7f1f10<12:30:59>
```

時分秒は同じなのに、不一致としてFAIL

- 素直にjava.sql.Time.valueOf(localTime)を使う

# java.sql.TimestampとLocalDateTimeとの変換

- TIMESTAMPのカラムに対して JDBCはjava.sql.Timestampを使うが、 Tsubakuro/JavaはLocalDateTimeなので、変換が必要
- java.sql.Timestamp.valueOf(localDateTime)相当の処理を、java.sql.Dateと同様にエポックで変換

# java.sql.TimestampとOffsetDateTimeとの変換

- TIMESTAMP WITH TIME ZONEのカラムに対してJDBCはjava.sql.Timestamp+Calendarを使うが、Tsubakuro/JavaはOffsetDateTimeなので、変換が必要
- 変換方法は2通り考えられる
  1. ```
     var t =
     Timestamp.valueOf(odt.toLocalDateTime());
     var ldt = t.toLocalDateTime();
     var odt2 = OffsetDateTime.of(ldt, offset);
     ```
  2. ```
     var t = Timestamp.from(odt.toInstant());
     var i = t.toInstant();
     var odt2 = OffsetDateTime.ofInstant(i, zone);
     ```

# java.sql.Timestamp → OffsetDateTime

- setTimestamp(timestamp, calendar)に対し
  1. ZonedDateTimeはLocalDateTime+ZoneIdである
  2. java.sql.TimestampはLocalDateTime相当、CalendarはZoneId相当
  3. ZonedDateTime.of(timestamp.toLocalDateTime(), calendar.getTimeZone().toZoneId()).toOffsetDateTime()
  - しかし、これだとPostgreSQLの実行結果と合わない…
- setTimestamp(timestamp, calendar)に対し
  1. OffsetDateTime.ofInstant(timestamp.toInstant(), calendar.getTimeZone().toZoneId())
  - こちらの方式とした


終わり

# まとめ

# まとめ

- Tsurugiはメニーコア・大容量メモリーを前提としたインメモリーのRDBMS

- Tsurugiのトランザクション分離レベルはSERIALIZABLE
  - トランザクション開始時にトランザクション種別（OCC・LTX・RTX）を指定する。
  - SQL実行時やコミット時にシリアライゼーションエラーが発生する可能性がある。（selectのみのトランザクションでもコミットが必要）
    - select for updateでの排他制御は行わない（行えない）
  - シリアライゼーションエラーが発生したら、アプリケーション側でトランザクションを再実行する。

# まとめ（JDBC）

- TsurugiのJava用クライアントライブラリー
  - Tsubakuro/Java（JDBCではない）
    - Tsurugiの全ての機能を使えるようにする目的
  - Iceaxe （JDBCではない）
    - Tsubakuro/JavaのSQL機能を便利に使えるようにする目的
  - PostgreSQL FDW（PostgreSQLのJDBC利用可）
  - Tsurugi JDBC（←New！）

- Tsurugi JDBCを使ってみて、不具合が見つかったらやさしく教えてください
  - MyBatis, Hibernate, Spring, …（最近では何が使われている？）
  - Tsurugi Advent Calendar 2025