BL-06 データベースアクセス機能

■ 概要

◆ 機能概要

- TERASOLUNA Batch Framework for Java ver 3.6.x(以下、TERASOLUNA Batch)では、MyBatis3 の Mapper インタフェースを使用するデータベースアクセス機能を提供する。MyBatis3 と Spring Framework の連携には、MyBatis-Spring を使用している。MyBatis3、MyBatis-Spring の汎用的な説明は「TERASOLUNA Server Framework for Java (5.x) Development Guideline」(以下、ガイドライン)を参照することとし、本書は TERASOLUNA Batch 向けの機能説明に特化して説明することとする。

- MyBatis3、Mapper インタフェース、MyBatis-Spring の詳細な説明は、ガイドラインの以下の項目を参照すること。

  - MyBatis3

「5.2.1.1. MyBatis3 について」
(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#dataaccessmybatis3overviewaboutmybatis3)

  - Mapper インタフェース

「5.2.4.1. Mapper インタフェースの仕組みについて」
(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#dataaccessmybatis3appendixaboutmappermechanism)

  - MyBatis-Spring

「5.2.1.2. Mybatis3 と Spring の連携について」
(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#mybatis3spring)

- ガイドラインを参照する際の注意事項

  ガイドラインは TERASOLUNA Server Framework for Java 5.x 向けの記述となっているため、記述内容の読み替えが必要になる箇所がある。

  ➢ 業務処理を提供するクラスの読み替え

  TERASOLUNA Server Framework for Java では、データベースアクセスを伴う業務処理を Service クラスに実装しているが、TERASOLUNA Batch ではアーキテクチャの違いから、BLogic クラスに実装する。そのため、ガイドラインの Service は BLogic に読み替えること。

  ➢ データベースアクセスに関係するクラスの読み替え

  TERASOLUNA Server Framework for Java では Entity と Repository を規定しているが、TERASOLUNA Batch では考え方の違い(※5)から、DTO と DAO を規定している。そのため、ガイドラインの Entity は DTO、Repository は DAO に読み替えること。

  (※5)Entity/Repository と DTO/DAO の違い

  Entity はデータベースのあるテーブルの 1 レコードを表現するクラスであり、Repository は Entity の問い合わせや、作成、更新、削除のような CRUD 処理を担うクラスである。一方、DTO はデータベースのあるテーブルの 1 レコードを表現するものに限らず、処理に必要なデータをまとめて表現するクラスであり、DAO はデータベースアクセスを担うクラスである。

  バッチ処理では複数のテーブルを結合し、処理に必要な項目のみを抜き出すことが多いため、Entity/Repository の考え方にはそぐわないことが多い。バッチ処理で無理に Entity/Repository の考え方を採用すると、処理に必要なデータを取得するための SQL 発行回数が多くなってしまいがちなので、使用する際は注意すること。

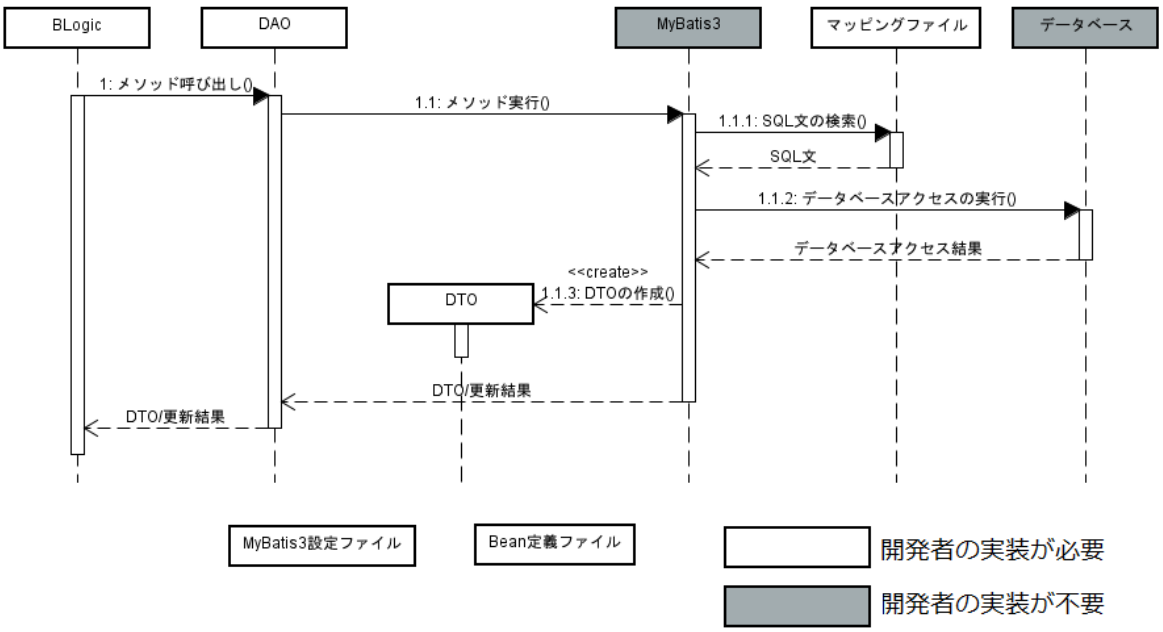
- MyBatis3 のドキュメントについて

  本書が参照する URL から"/ja"を取り除くと、英語のドキュメントを参照することができる。

  日本語のドキュメント：http://mybatis.github.io/mybatis-3/ja/index.html

  英語のドキュメント：http://mybatis.github.io/mybatis-3/index.html

◆ 概念図



◆ 解説

処理の流れは概念図の通り。概念図を構成する各要素の説明を以下に示す。

	構成要素名	説明
1	BLogic	業務処理を行うクラス。データベースを行う際は DAO のメソッドを呼び出す。
2	DAO	データベースアクセスを行うクラス。実際の処理は MyBatis3 に委譲する。インタフェースを定義するだけで良く、実装クラスを作成する必要がない。
3	DTO	データベースアクセスの結果を保持するクラス。String 型などを直接指定することも可能である。
4	MyBatis3	呼び出された DAO のメソッドに対応する SQL を実行し、結果を DTO として返却する。更新系 SQL の実行時は、更新件数を返却する。
5	マッピングファイル	データベースアクセスを行う SQL を定義する MyBatis3 の設定ファイル。
6	MyBatis3 設定ファイル	MyBatis3 の動作をカスタマイズする設定ファイル。
7	Bean 定義ファイル	データベースアクセスに関する定義を行う Spring Framework の Bean 定義ファイル。
8	データベース	アクセス対象のデータベース。

■ 使用方法

◆ コーディングポイント

【コーディングポイントの構成】

- データソースの Bean 定義
- SqlSessionFactory の Bean 定義
- DAO の作成
  - 参照系
  - 更新系
- マッピングファイルの作成
  - マッピングファイルの配置
  - Namespace の設定
  - SQL 定義の追加
    - 参照系
    - 更新系
    - 動的 SQL について
- DAO の Bean 定義
- ビジネスロジックの実装
  - @Inject アノテーションを使用した依存性の注入
  - データベースアクセスの実行

- データソースの Bean 定義

DAO がアクセスするデータソースの Bean 定義を追加する。TERASOLUNA Batch がアクセスするデータソースを設定する Bean 定義ファイル(ブランクプロジェクトでは、beansAdminDef ディレクトリ配下の AdminDataSource.xml)と、各ジョブがアクセスするデータソースを設定する Bean 定義ファイル(ブランクプロジェクトでは beansDef ディレクトリ配下の dataSource.xml)の両方に設定が必要となる。

✧ データソースの Bean 定義例(beansDef/dataSource.xml)

```
<!-- DBCP のデータソースを設定する。    -->
<context:property-placeholder location="mybatis/jdbc.properties" />
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp2.BasicDataSource">
      <property name="driverClassName" value="${jdbc.driver}" />
      <property name="url" value="${jdbc.url}" />
      <property name="username" value="${jdbc.username}" />
      <property name="password" value="${jdbc.password}" />
      <property name="maxTotal" value="10" />
      <property name="maxIdle" value="1" />
      <property name="maxWaitMillis" value="5000" />
</bean>
```

設定値はプロパティファイルに切り離し、プレースホルダを使用して設定する。

✧ プロパティファイルの作成例(**mybatis/jdbc.properties**)

```
jdbc.driver=org.postgresql.Driver
jdbc.url=jdbc:postgresql://127.0.0.1:5432/postgres
jdbc.username=postgres
jdbc.password=postgres
```

- SqlSessionFactory の Bean 定義

DAO の Bean 定義に必要な SqlSessionFactory の Bean 定義を追加する。DAO が使用するデータソースと MyBatis3 設定ファイルの格納先を設定する。

✓ データソース
"dataSource"プロパティに、データソースの Bean 定義 ID を指定する。

✓ MyBatis3 設定ファイル
"configLocation"プロパティに、MyBatis3 設定ファイルの格納先を指定する。MyBatis3 の設定をカスタマイズしない場合は、指定しなくても良い。

✧ SqlSessionFactory の Bean 定義例(beansDef/dataSource.xml)

```
<!-- SqlSessionFactory 定義 -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="configLocation" value="mybatis/mybatis-config.xml" />
    <property name="dataSource" ref="dataSource" />
</bean>
```

- DAO の作成

ビジネスロジックからデータベースアクセスを実行するための DAO を作成する。データベースへのアクセスパターンごとにメソッドを宣言する。

➢ 参照系

データを取得する場合、メソッドは次のように宣言する。

✓ 戻り値
データベースアクセスの結果をマッピングする DTO を指定する。結果が複数件になる場合は、DTO の配列またはコレクションで指定する。

✓ 引数
SQL パラメータ引数を格納する DTO を指定する。全件取得の場合など、パラメータ引数がない場合は、引数を指定しなくて良い。

✧ DAO のメソッドの宣言例(参照系)

```
public interface SampleDao {

    String findPersonNameByReservationNo(FindReservationInputDto dto)

    ReservationDto findOneReservationDataByReserveDate(FindReservationInputDto dto);

    List<ReservationDto> findAllReservationData();

}
```

➢ 更新系

データを挿入・更新・削除する場合、メソッドは次のように宣言する。

✓ 戻り値
int を指定することで、挿入・更新・削除件数を取得することができる。

✓ 引数
SQL パラメータ引数を格納する DTO を指定する。パラメータ引数がない場合は、引数を指定しなくて良い。

✧ DAO のメソッドの宣言例(更新系)

```
public interface SampleDao {

    int insertReservationData(InsertReservationInputDto dto);

    int insertInitialData();

    int updateReservationData(UpdateReservationInputDto dto);

    int updateBussinessDate();

    int deleteReservationData(DeleteReservationInputDto dto);

    int deleteAllOldReservationData();

}
```

- マッピングファイルの作成
  - マッピングファイルの配置

    マッピングファイルは以下の構成で作成する。

    - ✓ 配置先
      DAO インタフェースが格納されているパッケージ階層と同一階層
    - ✓ ファイル名
      DAO インタフェース名 + ".xml"

  - namespace の設定

    マッピングファイル内のルート要素 mapper タグの namespace 属性には、DAO のフルパス(本書では FQCN の意味で使用している)を設定すること。

    ✧ マッピングファイルの作成例(com/example/sample/SampleDao.xml)

```
<mapper namespace="com.example.sample.SampleDao">
…(SQL の設定)…
</mapper>
```

  - SQL 定義の追加

    mapper タグの子要素として、DAO に宣言したメソッドごとに select タグ等を用いて SQL 定義を追加する。SQL 定義方法の詳細は、MyBatis3 のドキュメント「Mapper XML ファイル」(http://mybatis.github.io/mybatis-3/ja/sqlmap-xml.html)を参照すること。

    - 参照系

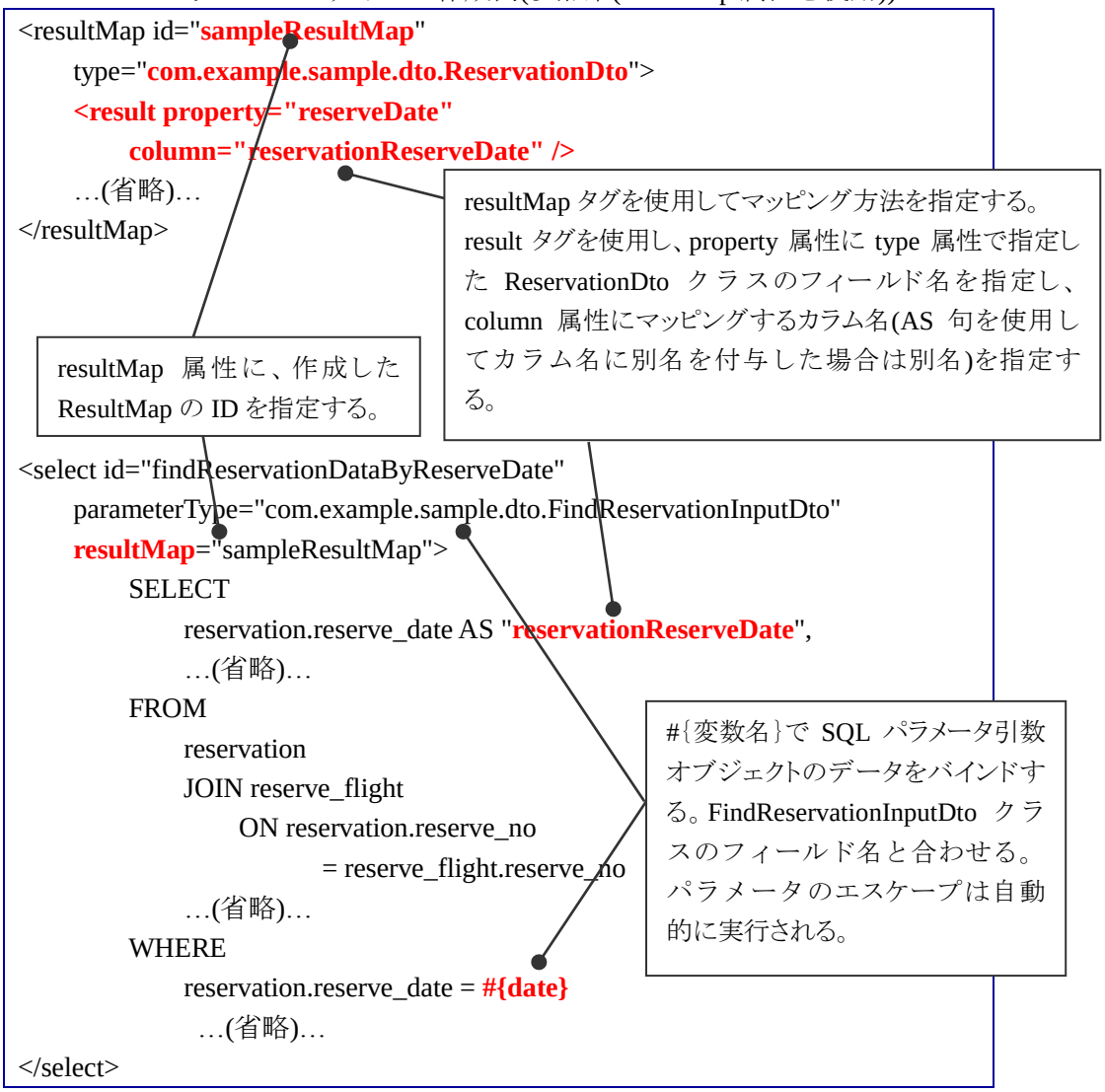
      select タグを使用して SQL を定義する。

      - ✓ SQL_ID の設定
        select タグの id 属性に、DAO に宣言したメソッド名と同じ文字列を指定する。
      - ✓ ResultMap の作成と設定
        MyBatis3 がデータベースアクセス結果のマッピングに使用する ResultMap を、resultMap タグを使用して作成する。select タグの resultMap 属性に、作成した ResultMap の ID を指定する。
        取得したカラム名と DTO のプロパティ名が一致する場合は、resultType 属性に DTO を指定することで、ResultMap の作成と設定を省略できる。(SQL の AS 句を使用してカラム名に別名を付与し、DTO のプロパティ名と一致させてもよい)

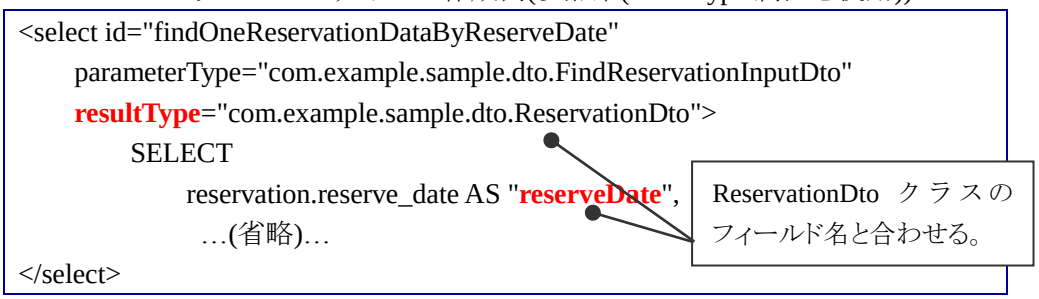
✓ SQL パラメータ引数の設定

parameterType 属性に、SQL パラメータ引数を格納する DTO を指定する。この属性は省略できる。

✧ マッピングファイルの作成例(参照系(resultMap 属性を使用))



✧ マッピングファイルの作成例(参照系(resultType 属性を使用))



- 更新系

insert、update、delete タグを使用して SQL を定義する。

✓ SQL_ID の設定

insert、update、delete タグの id 属性に、DAO に宣言したメソッド名と同じ文字列を指定する。

✓ 更新結果の取得

件数の取得をマッピングファイルで考慮する必要はないため、resultType 属性は指定しない。

✓ SQL パラメータ引数の設定

parameterType 属性に、SQL パラメータ引数を格納する DTO を指定する。この属性は省略できる。

✧ マッピングファイルの作成例(更新系)

```
<insert id="insertReservationData"
      parameterType="com.example.sample.dto.InsertReservationInputDto">
      INSERT INTO RESERVATION (
            RESERVE_ID
            …(省略)…
      ) VALUES (
            #{id}
            …(省略)…
      )
</insert>
<update id="updateReservationData"
      parameterType="com.example.sample.dto.UpdateReservationInputDto">
      UPDATE RESERVATION SET
            RESERVE_DATE = #{newReserveDate}
            …(省略)…
      WHERE
            RESERVATION_ID = #{id}
</update>
<delete id="deleteReservationData"
      parameterType="com.example.sample.dto.DeleteReservationInputDto">
      DELETE FROM RESERVATION
      WHERE
            RESERVATION_ID = #{id}
</delete>
```

- 動的 SQL について

MyBatis3 では、SQL を動的に組み立てるための XML 要素と、OGNL ベースの式(Expression 言語)を使用することで、動的 SQL を組み立てる仕組みを提供している。詳細は、「動的 SQL」 (http://mybatis.github.io/mybatis-3/ja/dynamic-sql.html)を参照すること。

- DAO の Bean 定義

DAO をビジネスロジックで使用するために Bean 定義を追加する。DAO のフルパスと、使用する SqlSessionFactory を設定する。

✓ DAO のフルパス
"mapperInterface"プロパティに DAO のフルパスを設定する。

✓ 使用する SqlSessionFactory の設定
"sqlSessionFactory"プロパティに SqlSessionFactory の Bean 定義 ID を設定する。DAO がアクセスするデータソースは、ここで設定した SqlSessionFactory によって決定する。複数のデータソースを使用する場合は、対象の DAO がどのデータソースを使用するのかを確認し、適切な SqlSessionFactory の Bean 定義 ID を設定すること。

✧ DAO の Bean 定義例(ジョブ Bean 定義ファイル(単一のデータソース))

```
<!-- SampleDao 設定 -->
<bean id="sampleDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface" value="com.example.sample.SampleDao" />
    <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

DAO のフルパスを設定する。

✧ DAO の Bean 定義例(ジョブ Bean 定義ファイル(複数のデータソース))

```
<!-- SampleDao_1 設定 -->
<bean id="sampleDao_1" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface" value="com.example.sample.SampleDao_1" />
    <property name="sqlSessionFactory" ref="sqlSessionFactory_1" />
</bean>

<!-- SampleDao_2 設定 -->
<bean id="sampleDao_2" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface" value="com.example.sample.SampleDao_2" />
    <property name="sqlSessionFactory" ref="sqlSessionFactory_2" />
</bean>
```

SampleDao_1 は sqlSessionFactory_1 に対応するデータソースにアクセスする。

SampleDao_2 は sqlSessionFactory_2 に対応するデータソースにアクセスする。

✓ Bean 定義ファイルの設定箇所について

DAO の Bean 定義は、各ジョブがアクセスするデータソースを設定する Bean 定義ファイル(ブランクプロジェクトでは beansDef/dataSource.xml) に設定せず、ジョブ Bean 定義ファイル (beansDef/(ジョブ ID).xml)に設定することを推奨する。dataSource.xml に設定すると、ジョブが使用しない DAO を DI コンテナで管理することになるため、ジョブの起動に時間がかかる、消費するメモリ量が増えるといったデメリットがあるためである。

- ビジネスロジックの実装
  - @Inject アノテーションを使用した依存性の注入

    ビジネスロジックのフィールドに DAO を宣言し、@Inject アノテーションを使用して依存性を注入する。
  - データベースアクセスの実行

    DAO のフィールドに定義されたメソッドを実行する。SQL パラメータ引数がある場合は、メソッドを実行する前にパラメータ引数オブジェクトを作成し、メソッド実行時にメソッドの引数に与える。
    - 参照系

      データベースアクセスの結果をメソッドの戻り値として取得できるため、メソッドの戻り値を取得してデータベースアクセスの結果を処理する。

      ✧ ビジネスロジック実装例(参照系)

```
@Inject
SampleDao sampleDao;

public int execute(BLogicParam arg0) {
    ...(省略)...
    FindReservationInputDto dto
        = new FindReservationInputDto();
    // 検索条件の設定
    dto.setId("xxx");
    ...(省略)...
    ReservationDto resavationDto
        = sampleDao.findOneReservationDataByReserveDate(dto);
    ...(省略)...
}
```

sampleDao に DAO 実装クラスのインスタンスが設定される。データベースアクセスには sampleDao の各メソッドを使用する。

SQL パラメータ引数オブジェクトを生成し、DAO のメソッドの引数に渡す。

- 更新系

更新件数をメソッドの戻り値として取得できるため、メソッドの戻り値を取得して更新の妥当性チェックを行うことができる。

✧ ビジネスロジック実装例(更新系)

```
@Inject
SampleDao sampleDao;

public int execute(BLogicParam arg0) {
    ...(省略)...
    UpdateReservationInputDto dto
        = new UpdateReservationInputDto();
    // 更新情報を設定
    dto.setId("xxx");
    ... (省略)...
    int updateCount
        = sampleDao.updateReservationData(dto);
    if(updateCount == 0) {
        // エラー処理
    }
    ...(省略)...
}
```

SQL パラメータ引数オブジェクトを生成し、DAO のメソッドの引数に渡す。

戻り値を使用した更新結果の判定も可能。

■ リファレンス

◆ 構成クラス

TERASOLUNA が独自に提供するクラスはない。

◆ 拡張ポイント

- MyBatis3 の設定

「SqlSessionFactory の Bean 定義」で指定する MyBatis3 設定ファイル(ブランクプロジェクトでは mybatis/mybatis-config.xml、または、mybatisAdmin/mybatis-config.xml)内で設定可能な設定項目のうち、よく使う設定について、以下に示す。その他の設定項目については、MyBatis3 のドキュメント「設定」(http://mybatis.github.io/mybatis-3/ja/configuration.html)を参照すること。

- 実行モードの設定

MyBatis3 では、DAO が SQL を実行する際の挙動を「実行モード」として、SIMPLE、REUSE、BATCH の 3 つから選択する。各実行モードの挙動の詳細は、ガイドラインの「5.2.2.3.2. SQL 実行モードの設定」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#sql)を参照すること。

MyBatis3 標準は「SIMPLE」だが、設定ファイルの「defaultExecutorType」項目を明示的に指定することで、実行モードを指定することができる。[1]
「defaultExecutorType」項目の指定についての詳細は、ガイドラインの「5.2.3.4.1. PreparedStatement 再利用モードの利用」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#preparedstatement)を参照すること。

- TypeHandler の設定

MyBatis3 標準でサポートされていない Java クラスとのマッピングが必要な場合や、MyBatis3 標準の振舞いを変更する必要がある場合は、独自の TypeHandler を作成してマッピングを行う必要がある。
詳細は、ガイドラインの「5.2.2.3.5. TypeHandler の設定」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#typehandler)を参照すること。

---

[1] 「defaultExecutorType」項目はすべての DAO の実行モードに影響する。DAO ごとの指定方法は「DAO ごとの実行モードの変更」を参照すること。

- TypeAlias の設定

TypeAlias を設定すると、後述する SQL の設定で指定する SQL パラメータ引数オブジェクトのクラスや結果をマッピングするクラスに対して、エイリアス名(短縮名)を割り当てることができる。
詳細は、ガイドラインの「5.2.2.3.3. TypeAlias の設定」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#typealias)を参照すること。

- DAO ごとの実行モードの変更

バッチ処理ではオンライン処理と異なり、大量のデータを挿入・更新・削除するケースが多々ある。そのような場合、使用する DAO の実行モードとして「BATCH」を選択すると、性能の向上が期待できる。個々の DAO ごとに実行モードを切り替える場合は、DAO の Bean 定義の際に、使用する SqlSessionFactory を指定する代わりに、SqlSessionTemplate を指定する。

➢ SqlSessionTemplate の Bean 定義

SqlSessionTemplate 自身のコンストラクタを使用して SqlSessionTemplate を生成する。生成には、SqlSessionFactory と実行モードを指定する文字列を設定する。

✓ SqlSessionFactory
コンストラクタの第 1 引数(constructor-arg index="0")に、SqlSessionTemplate の生成に使用する SqlSessionFactory の Bean 定義 ID を指定する。

✓ 実行モードの設定
コンストラクタの第 2 引数(construnctor-arg index="1")に、「SIMPLE」「REUSE」「BATCH」から選択した実行モードを指定する。

✧ SqlSessionTemplate の Bean 定義例(beansDef/dataSource.xml)

```
<!-- SqlSessionTemplate 定義 -->
<bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
    <constructor-arg index="1" value="SIMPLE" />
</bean>
<bean
    id="batchSqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
    <constructor-arg index="1" value="BATCH" />
</bean>
```

➢ DAO の Bean 定義

"sqlSessionFactory"プロパティの代わりに、"sqlSessionTemplate"プロパティを使用する。

✧ DAO ごとの実行モードの変更例(beansDef/B000001.xml)

```
<!-- SMP000Dao 設定 -->
<bean class="org.mybatis.spring.mapper.MapperFactoryBean">
<property name="mapperInterface"
        value="jp.terasoluna.batch.tutorial.sample000.SMP000Dao" />
<property name="sqlSessionTemplate" ref="sqlSessionTemplate" />
</bean>
```

選択した実行モードで生成された SqlSessionTemplate を設定する。

■ 関連機能

- 『BL-03 トランザクション管理機能』

■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)
- チュートリアル(terasoluna-batch-tutorial)

■ 注意事項

◆ 大量データの取得について

- ResultHandler の使用

大量のデータを返すようなクエリを記述する場合には、引数に ResultHandler を追加することで、ResultHandler を使用しない場合と比べ、メモリの消費量を抑制することができる。

TERASOLUNA Batch では、入力データを取得する際は『AL-041 入力データ取得機能(コレクタ)』を使用することを推奨しているため、詳細は『AL-041 入力データ取得機能(コレクタ)』を参照すること。

- fetchSize 属性の指定

大量のデータを返すようなクエリを記述する場合には、JDBC ドライバに対して適切な fetchSize を設定する必要がある。fetchSize 属性は、JDBC ドライバとデータベース間の通信において、一度の通信で取得するデータの件数を設定するパラメータである。

fetchSize 属性を省略した場合は、JDBC ドライバのデフォルト値が使用されるため、デフォルト値が全件取得する JDBC ドライバの場合、メモリの枯渇の原因になる可能性があるので、 注意が必要となる。

TERASOLUNA Batch3.6.x でサポートする MyBatis3.3.0 以降では、fetchSize の設定方法は 2 通りある。

- select タグの fetchSize 属性に SQL 単位の fetchSize を設定する
- MyBatis3 設定ファイルの defaultFetchSize 項目にデフォルトの fetchSize を設定する

設定についての詳細は、ガイドラインの「5.2.2.3.1. fetchSize の設定」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#fetchsize)を参照すること。

◆ N+1 問題への対応について

N+1 問題とは、一覧テーブルと明細テーブルのような 1:N の関係にあるデータを取得する際に発生しがちな問題である。一覧テーブルからデータを取得した後に取得したデータ 1 件ごとに明細テーブルにアクセスするなど、レコード数に比例して SQL の実行回数が増えてしまうことにより、データベースへの負荷およびターンアラウンドタイムの劣化を引き起こす。

N+1 問題の詳細は、ガイドラインの「5.1.4.1. N+1 問題の対策方法」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessCommon.html#n-1)を参照すること。

N+1 問題を回避するためには、関連するテーブルを結合し、1 回の SQL で必要なデータを取得する必要がある。この取得結果を DTO にマッピングする方法として、MyBatis3 では ResultMap に collection タグを使用する方法を提供している。collection タグについては、MyBatis3 のドキュメント「Mapper XML ファイル」(http://mybatis.github.io/mybatis-3/ja/sqlmap-xml.html)の「collection」節を参照すること。

◆ DAO の実行モードに「BATCH」を選択した場合について

- 適度な間隔でのバッチ更新の実行

バッチ更新を行う場合、トランザクション内での DAO の更新系メソッドの呼び出し回数に応じてメモリ上に SQL パラメータ引数の情報が蓄積されていく。そのため、適度な間隔でバッチ更新を実行し、メモリ上に蓄積された SQL パラメータ引数の情報をデータベースに送信しないとメモリが枯渇してしまう。トランザクション内でバッチ更新を実行するためには、SqlSession インタフェースの flushStatements メソッドを使用すること。なお、flushStatements メソッドを使用してもコミットは実行されない。

TERASOLUNA Batch3.6.x でサポートする MyBatis3.3.0 以降では、@Flush アノテーションを使用することで、flushStatements メソッドを使用する際の SqlSession インタフェースのインジェクションが不要となった。ここでは、@Flush アノテーションを使用する例を示す。

✧ flushStatements メソッドの使用例(DAO)

```
public interface SampleDao {

    int deleteReservationData(DeleteReservationInputDto dto);

    @Flush
    List<BatchResult> flushStatements();

}
```

flushStatements というメソッドを定義し、@Flush アノテーションを付与する。戻り値は、SqlSession インタフェースの flushStatements メソッドと同じ、List<BatchResult>となる。

✧ flushStatements メソッドの使用例(ビジネスロジック)

```
    @Inject
    SampleDao sampleDao;

    public int doMain(BLogicParam arg0) {
        ... (省略)...
        int updateCount = 0;
        for (UpdateReservationInputDto dto : updateReservationInputDtoList) {
            sampleDao.updateReservationData(dto);
            updateCount++;
            if (updateCount % 1000 == 0) {
                // 1000 件単位にバッチ更新を実行する。
                sampleDao.flushStatements();
            }
        }
        ...(省略)...
    }
```

例では、DAO のメソッドを呼び出した回数をカウントしておき、1000 件呼び出すごとにバッチ更新を実行している。コミットは行わない。

- 戻り値の件数の考慮

バッチ更新を行う場合、DAO のメソッドからの戻り値は常に-2147482646 件となる。戻り値の件数をチェックする必要がある場合は、SqlSession インタフェースの flushStatements メソッドを使用してコミット前にバッチ更新を実行し、戻り値の DTO に含まれる更新件数を取得すること。ただし、正しい更新件数を取得できるかどうかは、JDBC ドライバの実装に依存することに注意すること。詳細は、ガイドラインの「5.2.3.4.3.1. 更新結果の判定」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#dataaccessmybatis3howtoextendexecutortypebatchnotesupdateresult)を参照すること。

- DAO のメソッドを呼び出す順番の考慮

バッチ更新を行う場合は、DAO のメソッドを呼び出す順番に注意する必要がある。具体的には、以下の 2 点に注意すること。

- ✓ バッチ更新時に参照系の処理を呼び出すと、内部でバッチ更新が実行される。意図しないタイミングでバッチ更新が実行されないように注意すること。

- ✓ バッチ更新時は DAO の更新系のメソッドで呼び出される SQL が変化するたびに新しい PreparedStetement を生成する。そのため、BLogic 内で実行する SQL が複数ある場合は、同一の SQL を連続して実行するよう調整し、無駄な PreparedStatement が生成されないように注意すること。なお、動的 SQL を用いた場合も、生成される SQL が異なれば新しい PreparedStatement が生成されることに注意すること。

詳細は、「5.2.3.4.3.3. Repository のメソッドの呼び出し順番」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#dataaccessmybatis3howtoextendexecutortypebatchnotesmethodcallorder)を参照すること。

- 複数のデータソースを使用したトランザクション制御について

バッチ更新時は SQL の実行タイミングがコミット時または SqlSession インタフェースの flushStatements メソッドを使用したバッチ更新の実行時となるため、SQL 実行時エラーも同じタイミングでしか検出できない。そのため、あるデータソースのトランザクションをコミットした後に別のデータソースのトランザクションをコミットしようとして SQL 実行エラーが発生した場合など、意図したロールバックが実行できずデータ不整合が発生することがある。

コミット前に SqlSession インタフェースの flushStatements メソッドを使用したバッチ更新を実行し、必要なエラー処理を行うことで、データ不整合が発生しないように注意すること。

■ 備考

◆ SQL 実行ログの出力

MyBatis3 は SLF4J にログの出力を委譲している。ロガー名は「DAO のフルパス + "." + メソッド名」となる。このロガー名に対するログレベルを変更することで、SQL 実行ログを出力することができる。

- ✓ 取得したレコード内容の出力が不要な場合
  ログレベルを DEBUG に設定することで、実行される SQL、バインドされる SQL パラメータ引数、実行結果の件数が出力される。

```
[DEBUG] ==>  Preparing: (実行される SQL)
[DEBUG] ==> Parameters: (バインドされる SQL パラメータ引数)
[DEBUG] <==      Total: (実行結果の件数)
```

✓ 取得したレコード内容の出力が必要な場合

ログレベルを TRACE に設定することで、DEBUG ログの内容に加えて、以下の通り、取得したレコードの内容が出力される。

```
[DEBUG] ==>  Preparing: (実行されるSQL)
[DEBUG] ==> Parameters: (バインドされるSQLパラメータ引数)
[TRACE] <==      Columns: (カラム名)
[TRACE] <==          Row: (取得したレコードの内容)
[DEBUG] <==        Total: (実行結果の件数)
```

以下に設定例を示す。

✧ SLF4J+Logback での設定例(logback.xml)

```
<configuration>
…(省略)…
    <logger name="com.example.sample" level="INFO " />
    <logger name="com.example.sample.SampleDao"
        level="DEBUG" />
    <logger name="com.example.sample.Sample2Dao" level="DEBUG" />
    <logger
        name="com.example.sample.SampleDao.findOneReservationDataByReserveDate"
        level="TRACE" />
…(省略)…
</configuration>
```

SampleDao、Sample2Dao インタフェースの各メソッド実行時に実行される SQL、バインドされる SQL パラメータ引数、実行結果の件数ログを出力する。

SampleDao インタフェースの findOneReservationDataByReserveDate メソッド実行時のみ、SQL 実行時に取得したレコードの内容を追加で出力する。

✓ MyBatis3 の logPrefix 属性の使用について

MyBatis3 の logPrefix 属性を使用すると、DAO のロガー名にプレフィックスを付与できる。DAO とその他のクラスでロガー名を区別できるようになるため、DAO ごとの設定が不要になる。設定例を以下に示す。

✧ SLF4J+Logback での設定例(logback.xml)

```
<configuration>
…(省略)…
    <logger name="MyBatis" level="TRACE" />
    <logger name="com.example.sample" level="INFO " />
…(省略)…
</configuration>
```

DAO のロガー名に、MyBatis3 設定ファイルで設定した接頭辞を付与する。

✧ logPrefix 属性の設定例(mybatis-config.xml)

```
<configuration>
    <settings>
        <setting name="logPrefix" value="MyBatis." />
    </settings>
</configuration>
```

◆ SQL パラメータ引数の与え方について

MyBatis3 では、SQL パラメータ引数を複数与えることができる。その際の SQL パラメータ引数は、DAO のメソッドの宣言時に引数に付与する@Param アノテーションで指定した名称で取得する。

✧ SQL パラメータ引数を複数与える例(ビジネスロジック)

```
@Inject
SampleDao sampleDao;

public int execute(BLogicParam arg0) {
    ...(省略)...
    // 更新情報を設定
    int id = 123;
    String reserveDate = new Date();
    ... (省略)...
    int updateCount
        = sampleDao.updateReservationData(id, reserveDate, …(省略)… );
    if(updateCount == 0) {
        // エラー処理
    }
    ...(省略)...
}
```

✧ SQL パラメータ引数を複数与える例(DAO)

```
public interface SampleDao {

    ReservationDto findOneReservationDataByReserveDate(
        @Param("id") int id, @Param("reserveDate") Date reserveDate,
        …(省略)…);

}
```

✧ SQL パラメータ引数を複数与える例(マッピングファイル)

```
<update id="updateReservationData"
    parameterType="com.example.sample.dto.UpdateReservationInputDto">
    UPDATE RESERVATION SET
        RESERVE_DATE = #{reserveDate}
        …(省略)…
    WHERE
        RESERVATION_ID = #{id}
</update>
```

◆ SqlSessionTemplate のクローズについて

MyBatis3 の SqlSessionTemplate を Bean 定義した場合、Logback のログ出力設定によっては、ジョブの実行終了後(アプリケーションコンテキストの破棄時)に以下のような WARN レベルのログが出力される。

✧ SqlSessionTemplate クローズ時の WARN ログ

```
[yyyy/MM/dd hh:mm:ss] [main] [o.s.b.f.s.DisposableBeanAdapter] [WARN ] Invocation of destroy method 'close' failed on bean with name 'batchSqlSessionTemplate': java.lang.UnsupportedOperationException: Manual close is not allowed over a Spring managed SqlSession
java.lang.UnsupportedOperationException: Manual close is not allowed over a Spring managed SqlSession
    at org.mybatis.spring.SqlSessionTemplate.close(SqlSessionTemplate.java:310) ~[mybatis-spring-1.2.2.jar:1.2.2]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.7.0_79]
    // 省略
```

このログが出力されても、リソースのクローズ自体は Spring Framework によって行われており、ジョブやフレームワークの動作に影響はない。

この事象は将来の MyBatis-Spring バージョンアップで修正される予定となっているが、それまでの間、システム運用上の理由などでこのログを出力させたくない場合は、SqlSessionTemplate の Bean 定義に destroy-method 属性を追加し、クローズ時に他への副作用が発生せず、ログを出力しないメソッド(たとえば、getExecutorType メソッド)を指定すること。

✧ SqlSessionTemplate の設定例(destroy-method 属性)

```
<bean id="sqlSessionTemplate"
        class="org.mybatis.spring.SqlSessionTemplate"
        destroy-method="getExecutorType">
    <constructor-arg index="0" ref="sqlSessionFactory"/>
    <constructor-arg index="1" value="REUSE"/>
</bean>
<bean id="batchSqlSessionTemplate"
        class="org.mybatis.spring.SqlSessionTemplate"
        destroy-method="getExecutorType">
    <constructor-arg index="0" ref="sqlSessionFactory"/>
    <constructor-arg index="1" value="BATCH"/>
</bean>
```