二

14 探究 MyBatis 结果集映射机制背后的秘密（上）

在前面介绍 MyBatis 解析 Mapper.xml 映射文件的过程中，我们看到 `<resultMap>` 标签会被解析成 ResultMap 对象，其中定义了 ResultSet 与 Java 对象的映射规则，简单来说，也就是**一行数据记录如何映射成一个 Java 对象**，这种映射机制是 MyBatis 作为 ORM 框架的核心功能之一。

ResultMap 只是定义了一个静态的映射规则，那在运行时，MyBatis 是如何根据映射规则将 ResultSet 映射成 Java 对象的呢？当 MyBatis 执行完一条 select 语句，**拿到 ResultSet 结果集之后，会将其交给关联的 ResultSetHandler 进行后续的映射处理。**

ResultSetHandler 是一个接口，其中定义了三个方法，分别用来处理不同的查询返回值：

```
public interface ResultSetHandler {

    // 将ResultSet映射成Java对象

    <E> List<E> handleResultSets(Statement stmt) throws SQLException;

    // 将ResultSet映射成游标对象

    <E> Cursor<E> handleCursorResultSets(Statement stmt) throws SQLException;

    // 处理存储过程的输出参数

    void handleOutputParameters(CallableStatement cs) throws SQLException;

}
```

在 MyBatis 中只提供了一个 ResultSetHandler 接口实现，即 DefaultResultSetHandler。下面我们就以 DefaultResultSetHandler 为中心，介绍 MyBatis 中 ResultSet 映射的核心流程。

结果集处理入口

你如果有 JDBC 编程经验的话，应该知道在数据库中执行一条 Select 语句通常只能拿到一个 ResultSet，但这只是我们最常用的一种查询数据库的方式，其实数据库还支持同时返回

多个 ResultSet 的场景，例如在存储过程中执行多条 Select 语句。MyBatis 作为一个通用的持久化框架，不仅要支持常用的基础功能，还要对其他使用场景进行全面的支持。

DefaultResultSetHandler 实现的 handleResultSets() 方法支持多个 ResultSet 的处理（单 ResultSet 的处理只是其中的特例），相关的代码片段如下：

```
public List<Object> handleResultSets(Statement stmt) throws SQLException {
    // 用于记录每个ResultSet映射出来的Java对象
    final List<Object> multipleResults = new ArrayList<>();
    int resultSetCount = 0;
    // 从Statement中获取第一个ResultSet，其中对不同的数据库有兼容处理逻辑，
    // 这里拿到的ResultSet会被封装成ResultSetWrapper对象返回
    ResultSetWrapper rsw = getFirstResultSet(stmt);
    // 获取这条SQL语句关联的全部ResultMap规则。如果一条SQL语句能够产生多个ResultSet，
    // 那么在编写Mapper.xml映射文件的时候，我们可以在SQL标签的resultMap属性中配置多个
    // <resultMap>标签的id，它们之间通过","分隔，实现对多个结果集的映射
    List<ResultMap> resultMaps = mappedStatement.getResultMaps();
    int resultMapCount = resultMaps.size();
    validateResultMapsCount(rsw, resultMapCount);
    while (rsw != null && resultMapCount > resultSetCount) { // 遍历ResultMap集合
        ResultMap resultMap = resultMaps.get(resultSetCount);
        // 根据ResultMap中定义的映射规则处理ResultSet，并将映射得到的Java对象添加到
        // multipleResults集合中保存
        handleResultSet(rsw, resultMap, multipleResults, null);
        // 获取下一个ResultSet
        rsw = getNextResultSet(stmt);
        // 清理nestedResultObjects集合，这个集合是用来存储中间数据的
        cleanUpAfterHandlingResultSet();
        resultSetCount++; // 递增ResultSet编号
    }
```

```
        // 下面这段逻辑是根据ResultSet的名称处理嵌套映射，你可以暂时不关注这段代码，
        // 嵌套映射会在后面详细介绍
        ...
        // 返回全部映射得到的Java对象
        return collapseSingleResultList(multipleResults);
    }
```

这里我们先来看一下遍历多结果集时使用到的 getFirstResultSet() 方法和 getNextResultSet() 方法，这两个方法底层都是依赖 java.sql.Statement 的 getMoreResults() 方法和 getUpdateCount() 方法检测是否存在后续的 ResultSet 对象，检测成功之后，会通过 getResultSet() 方法获取下一个 ResultSet 对象。

这里获取到的 ResultSet 对象，会被包装成 ResultSetWrapper 对象返回。

ResultSetWrapper 主要用于封装 ResultSet 的一些元数据，其中记录了 ResultSet 中每列的名称、对应的 Java 类型、JdbcType 类型以及每列对应的 TypeHandler。

另外，ResultSetWrapper 可以将底层 ResultSet 的列与一个 ResultMap 映射的列进行交集，得到参与映射的列和未被映射的列，分别记录到 mappedColumnNamesMap 集合和 unMappedColumnNamesMap 集合中。这两个集合都是 Map `<String, List<String>>` 类型，其中最外层的 Key 是 ResultMap 的 id，Value 分别是参与映射的列名集合和未被映射的列名集合。

除了记录上述元数据以外，ResultSetWrapper 还封装了一套查询上述元数据的方法，例如，我们可以通过 getMappedColumnNames() 方法查询一个 ResultMap 映射了当前 ResultSet 的哪些列，还可以通过 getJdbcType()、getTypeHandler() 等方法查询指定列对应的 JdbcType、TypeHandler 等。

简单映射

了解了处理 ResultSet 的入口逻辑之后，下面我们继续来深入了解一下 DefaultResultSetHandler 是如何处理单个结果集的，这部分逻辑的入口是 handleResultSet() 方法，其中会根据第四个参数，也就是 parentMapping，判断当前要处理的 ResultSet 是嵌套映射，还是外层映射。

无论是处理外层映射还是嵌套映射，**都会依赖 handleRowValues() 方法完成结果集的处理**（通过方法名也可以看出，handleRowValues() 方法是处理多行记录的，也就是一个结果

集）。

至于 handleRowValues() 方法，其中会通过 handleRowValuesForNestedResultMap() 方法处理包含嵌套映射的 ResultMap，通过 handleRowValuesForSimpleResultMap() 方法处理不包含嵌套映射的简单 ResultMap，如下所示：

```
public void handleRowValues(ResultSetWrapper rsw, ResultMap resultMap, ResultHandle
    if (resultMap.hasNestedResultMaps()) { // 包含嵌套映射的处理流程
        ensureNoRowBounds();
        checkResultHandler();
        handleRowValuesForNestedResultMap(rsw, resultMap, resultHandler, rowBounds,
    } else { // 简单映射的处理
        handleRowValuesForSimpleResultMap(rsw, resultMap, resultHandler, rowBounds,
    }
}
```

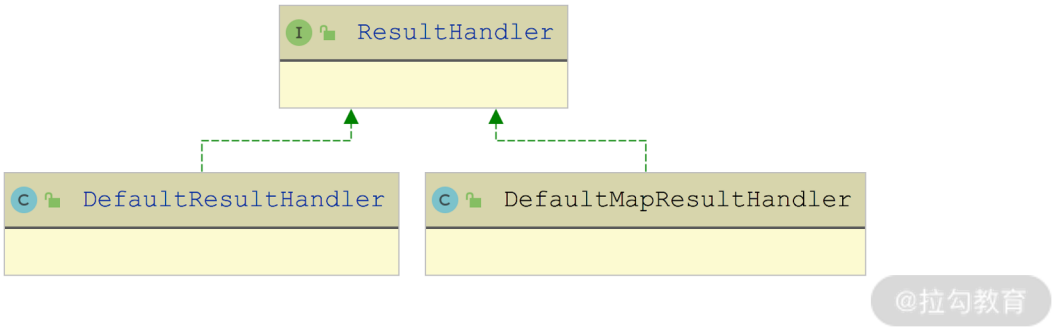
这里我们重点来看 handleRowValuesForSimpleResultMap() 方法如何映射一个 ResultSet 的，该方法的核心步骤可总结为如下。

1. 执行 skipRows() 方法跳过多余的记录，定位到指定的行。
2. 通过 shouldProcessMoreRows() 方法，检测是否还有需要映射的数据记录。
3. 如果存在需要映射的记录，则先通过 resolveDiscriminatedResultMap() 方法处理映射中用到的 Discriminator，决定此次映射实际使用的 ResultMap。
4. 通过 getRowValue() 方法对 ResultSet 中的一行记录进行映射，映射规则使用的就是步骤 3 中确定的 ResultMap。
5. 执行 storeObject() 方法记录步骤 4 中返回的、映射好的 Java 对象。

在开始详细介绍上述映射流程中的每一步之前，我们先来看一下贯穿整个映射过程的两个辅助对象——**DefaultResultHandler 和 DefaultResultContext**。

在 DefaultResultSetHandler 中维护了一个 resultHandler 字段（ResultHandler 接口类型）指向一个 DefaultResultHandler 对象，其核心作用是存储多个结果集映射得到的 Java 对象。

ResultHandler 接口有两个默认实现，如下图所示：



ResultHandler 接口继承图

DefaultResultHandler 实现的底层使用 ArrayList <Object> 存储映射得到的 Java 对象，DefaultMapResultHandler 实现的底层使用 Map <K, V> 存储映射得到的 Java 对象，其中 Key 是从结果对象中获取的指定属性的值，Value 就是映射得到的 Java 对象。

至于 DefaultResultContext 对象，它的生命周期与一个 ResultSet 相同，每从 ResultSet 映射得到一个 Java 对象都会暂存到 DefaultResultContext 中的 resultObject 字段，等待后续使用，同时 DefaultResultContext 还可以计算从一个 ResultSet 映射出来的对象个数（依靠 resultCount 字段统计）。

了解了 handleRowValuesForSimpleResultMap() 方法的核心步骤以及全部贯穿整个映射流程的辅助对象之后，下面我们开始深入每个步骤进行详细分析。

1. ResultSet 的预处理

有 MyBatis 使用经验的同学可能知道，我们可以通过 RowBounds 指定 offset、limit 参数实现分页的效果。**这里的 skipRows() 方法就会根据 RowBounds 移动 ResultSet 的指针到指定的数据行，这样后续的映射操作就可以从这一行开始。**

skipRows() 方法会检查 ResultSet 的属性，如果是 TYPE_FORWARD_ONLY 类型，则只能通过循环 + ResultSet.next() 方法（指针的逐行前移）定位到指定的数据行；反之，可以通过 ResultSet.absolute() 方法直接移动指针。

处理 RowBounds 的另一个方法是 shouldProcessMoreRows() 方法，其中会检查当前已经映射的行是否达到了 RowBounds.limit 字段指定的行数上限，如果达到，则返回 false，停止后续操作。当然，控制是否进行后续映射操作的条件还有 ResultSet.next() 方法（即结果集中是否还有数据）。

通过上述分析我们可以看出，通过 RowBounds 实现的分页功能实际上还是会将全部数据加载到 ResultSet 中，而不是只加载指定范围的数据，所以我们可以认为 RowBounds 实现的是一种“假分页”。这种“假分页”在数据量大的时候，性能就会很差，在处理大数据量分页

时，建议通过 SQL 语句 where 条件 + limit 的方式实现分页。

2. 确定 ResultMap

在完成 ResultSet 的预处理之后，接下来会**通过 resolveDiscriminatedResultMap() 方法处理 标签，确定此次映射操作最终使用的 ResultMap 对象。**

为了更加方便和完整地描述 resolveDiscriminatedResultMap() 方法的核心流程，这里我们结合一个简单示例进行分析，比如，现在有一个 ResultSet 包含 id、name、classify、subClassify 四列，并且由 animalMap 来映射该 ResultSet，具体如下图所示：

< discriminator>处理示例图

通过 resolveDiscriminatedResultMap() 方法确定 ResultMap 的流程大致是这样的：

- 首先按照 animalMap 这个 ResultMap 映射这行记录，该行记录中的 classify 列值为 mammalia，根据其中定义的 `<discriminator>` 标签的配置，会选择使用 mammaliaMap 这个 ResultMap 对当前这条记录进行映射；
- 接下来看 mammaliaMap 这个 ResultMap，其中的 `<discriminator>` 标签检查的是 subClassify 的列值，当前记录的 subClassify 列值为 human，所以会选择 humanMap 这个 ResultMap 映射当前这条记录，得到一个 Human 对象。

了解了上述基本流程之后，下面我们来看 resolveDiscriminatedResultMap() 方法的具体实

现：

```
public ResultMap resolveDiscriminatedResultMap(ResultSet rs, ResultMap resultMap, S
    // 用于维护处理过的ResultMap唯一标识
    Set<String> pastDiscriminators = new HashSet<>();
    // 获取ResultMap中的Discriminator对象，这是通过<resultMap>标签中的<discriminator>
    Discriminator discriminator = resultMap.getDiscriminator();
    while (discriminator != null) {
        // 获取当前待映射的记录中Discriminator要检测的列的值
        final Object value = getDiscriminatorValue(rs, discriminator, columnPrefix)
        // 根据上述列值确定要使用的ResultMap的唯一标识
        final String discriminatedMapId = discriminator.getMapIdFor(String.valueOf(
        if (configuration.hasResultMap(discriminatedMapId)) {
            // 从全局配置对象Configuration中获取ResultMap对象
            resultMap = configuration.getResultMap(discriminatedMapId);
            // 记录当前Discriminator对象
            Discriminator lastDiscriminator = discriminator;
            // 获取ResultMap对象中的Discriminator
            discriminator = resultMap.getDiscriminator();
            // 检测Discriminator是否出现了环形引用
            if (discriminator == lastDiscriminator || !pastDiscriminators.add(discr
                break;
            }
        } else {
            break;
        }
    }
    // 返回最终要使用的ResultMap
    return resultMap;
```

```
}
```

3. 创建映射结果对象

经过 resolveDiscriminatedResultMap() 方法解析，我们最终确定了当前记录使用哪个 ResultMap 进行映射。

接下来要做的就是**按照 ResultMap 规则进行各个列的映射，得到最终的 Java 对象**，这部分逻辑是在下面要介绍的 getRowValue() 方法完成的，其核心步骤如下：

- 首先根据 ResultMap 的 type 属性值创建映射的结果对象；
- 然后根据 ResultMap 的配置以及全局信息，决定是否自动映射 ResultMap 中未明确映射的列；
- 接着根据 ResultMap 映射规则，将 ResultSet 中的列值与结果对象中的属性值进行映射；
- 最后返回映射的结果对象，如果没有映射任何属性，则需要根据全局配置决定如何返回这个结果值，这里不同场景和配置，可能返回完整的结果对象、空结果对象或是 null。

下面是 getRowValue() 方法的核心实现：

```
private Object getRowValue(ResultSetWrapper rsw, ResultMap resultMap, String column
    final ResultLoaderMap lazyLoader = new ResultLoaderMap();
    // 根据ResultMap的type属性值创建映射的结果对象
    Object rowValue = createResultObject(rsw, resultMap, lazyLoader, columnPrefix);
    if (rowValue != null && !hasTypeHandlerForResultObject(rsw, resultMap.getType()
        final MetaObject metaObject = configuration.newMetaObject(rowValue);
        boolean foundValues = this.useConstructorMappings;
        // 根据ResultMap的配置以及全局信息，决定是否自动映射ResultMap中未明确映射的列
        if (shouldApplyAutomaticMappings(resultMap, false)) {
            foundValues = applyAutomaticMappings(rsw, resultMap, metaObject, column
        }
        // 根据ResultMap映射规则，将ResultSet中的列值与结果对象中的属性值进行映射
        foundValues = applyPropertyMappings(rsw, resultMap, metaObject, lazyLoader,
```

```
            // 如果没有映射任何属性，需要根据全局配置决定如何返回这个结果值，
            // 这里不同场景和配置，可能返回完整的结果对象、空结果对象或是null
            foundValues = lazyLoader.size() > 0 || foundValues;
            rowValue = foundValues || configuration.isReturnInstanceForEmptyRow() ? row
        }
        return rowValue;
    }
```

可以看到这里的第一步，也就是创建映射的结果对象，这部分逻辑位于 createResultObject() 方法中。这个方法中有两个关键步骤：一个是调用另一个 createResultObject() 重载方法来创建结果对象，另一个是通过 ProxyFactory 创建代理对象来处理延迟加载的属性。

由于我们重点分析的是简单 ResultSet 的映射流程，所以接下来我们重点看 createResultObject() 重载方法是如何创建映射结果对象的。

首先进行一些准备工作：获取 ResultMap 中 type 属性指定的结果对象的类型，并创建该类型对应的 MetaClass 对象；获取 ResultMap 中配置的 `<constructor>` 标签信息（也就是对应的 ResultMapping 对象集合），如果该信息不为空，则可以确定结果类型中的唯一构造函数。

然后再根据四种不同的场景，使用不同的方式创建结果对象，下面就是这四种场景的核心逻辑。

- 场景一，ResultSet 中只有一列，并且能够找到一个 TypeHandler 完成该列到目标结果类型的映射，此时可以直接读取 ResultSet 中的列值并通过 TypeHandler 转换得到结果对象。这部分逻辑是在 createPrimitiveResultObject() 方法中实现的，该场景多用于 Java 原始类型的处理。
- 场景二，如果 ResultMap 中配置了 `<constructor>` 标签，就会先解析 `<constructor>` 标签中指定的构造方法参数的类型，并从待映射的数据行中获取对应的实参值，然后通过反射方式调用对应的构造方法来创建结果对象。这部分逻辑在 createParameterizedResultObject() 方法中实现。
- 场景三，如果不满足上述两个场景，则尝试查找默认构造方法来创建结果对象，这里使用前面介绍的 ObjectFactory.create() 方法实现，底层原理还是 Java 的反射机制。
- 场景四，最后会检测是否已经开启了自动映射功能，如果开启了，会尝试查找合适的构造方法创建结果对象。这里首先会查找 @AutomapConstructor 注解标注的构造方法，

查找失败之后，则会尝试查找每个参数都有 TypeHandler 能与 ResultSet 列进行映射的构造方法，确定要使用的构造方法之后，也是通过 ObjectFactory 完成对象创建的。这部分逻辑在 createByConstructorSignature() 方法中实现。

4. 自动映射

创建完结果对象之后，下面就可以开始映射各个字段了。

在简单映射流程中，会先通过 shouldApplyAutomaticMappings() 方法**检测是否开启了自动映射**，主要检测以下两个地方。

- 检测当前使用的 ResultMap 是否配置了 autoMapping 属性，如果是，则直接根据该 autoMapping 属性的值决定是否开启自动映射功能。
- 检测 mybatis-config.xml 的 `<settings>` 标签中配置的 autoMappingBehavior 值，决定是否开启自动映射功能。autoMappingBehavior 指定 MyBatis 框架如何进行自动映射，该属性有三个可选值：①NONE，表示完全关闭自动映射功能；②PARTIAL，表示只会自动映射没有定义嵌套映射的 ResultMap；③FULL，表示完全打开自动映射功能，这里会自动映射所有 ResultMap。autoMappingBehavior 的默认值是 PARTIAL。

当确定当前 ResultMap 需要进行自动映射的时候，会通过 applyAutomaticMappings() 方法进行自动映射，其中的核心逻辑大致可描述为如下。

- 首先，从 ResultSetWrapper 中获取所有未映射的列名，然后逐个处理每个列名。通过列名获取对应的属性名称，这里会将列名转换为小写并截掉指定的前缀，得到相应的属性名称。
- 然后，检测结果对象中是否有上面得到的属性。如果属性不存在，则通过全局配置的 AutoMappingUnknownColumnBehavior 进行处理。如果属性存在，则检测该属性是否有合适的 TypeHandler；如果不存在合适的 TypeHandler，依旧是通过全局配置的 AutoMappingUnknownColumnBehavior 进行处理。
- 经过上述检测之后，就可以创建 UnMappedColumnAutoMapping 对象将该列与对应的属性进行关联。在 UnMappedColumnAutoMapping 中记录了列名、属性名以及相关的 TypeHandler。
- 最后，遍历上面得到 UnMappedColumnAutoMapping 集合，通过其中的 TypeHandler 读取列值并转换成相应的 Java 类型，再通过 MetaObject 设置到相应属性中。

这样就完成了自动映射的功能。

5. 正常映射

完成自动映射之后，MyBatis 会**执行 applyPropertyMappings() 方法处理 ResultMap 中明确要映射的列**，applyPropertyMappings() 方法的核心流程如下所示。

- 首先从 ResultSetWrapper 中明确需要映射的列名集合，以及 ResultMap 中定义的 ResultMapping 对象集合。
- 遍历全部 ResultMapping 集合，针对每个 ResultMapping 对象为 column 属性值添加指定的前缀，得到最终的列名，然后执行 getPropertyMappingValue() 方法完成映射，得到对应的属性值。
- 如果成功获取到了属性值，则通过结果对象关联的 MetaObject 对象设置到对应属性中。

在 getPropertyMappingValue() 方法中，主要处理了三种场景的映射：

- 第一种是基本类型的映射，这种场景直接可以通过 TypeHandler 从 ResultSet 中读取列值，并在转化之后返回；
- 第二种和第三种场景分别是嵌套映射和多结果集的映射，这两个逻辑相对复杂，在下一讲我们再详细介绍。

6. 存储对象

通过上述 5 个步骤，我们已经完成简单映射的处理，得到了一个完整的结果对象。接下来，我们就要**通过 storeObject() 方法把这个结果对象保存到合适的位置**。

这里处理的简单映射，如果是一个嵌套映射中的子映射，那么我们就需要将结果对象保存到外层对象的属性中；如果是一个普通映射或是外层映射的结果对象，那么我们就需要将结果对象保存到 ResultHandler 中。

明确了结果对象的存储位置之后，我们来看 storeObject() 方法的具体实现：

```
private void storeObject(...) throws SQLException {
    if (parentMapping != null) {
        // 嵌套查询或嵌套映射的场景，此时需要将结果对象保存到外层对象对应的属性中
        linkToParents(rs, parentMapping, rowValue);
    } else {
        // 普通映射(没有嵌套映射)或是嵌套映射中的外层映射的场景，此时需要将结果对象保存到R
        callResultHandler(resultHandler, resultContext, rowValue);
```

```
    }
}
```

总结

这一讲我们重点介绍了结果集映射，这是 MyBatis 的核心实现之一。

首先我们介绍了 ResultSetHandler 接口以及 DefaultResultSetHandler 这个默认实现，并讲解了单个结果集映射的入口：handleResultSet() 方法。

接下来，我们继续深入，详细分析了 handleRowValuesForSimpleResultMap() 方法实现简单映射的核心步骤，其中涉及预处理 ResultSet、查找并确定 ResultMap、创建并填充映射结果对象、自动映射、正常映射、存储映射结果对象这六大核心步骤。

上一页 下一页