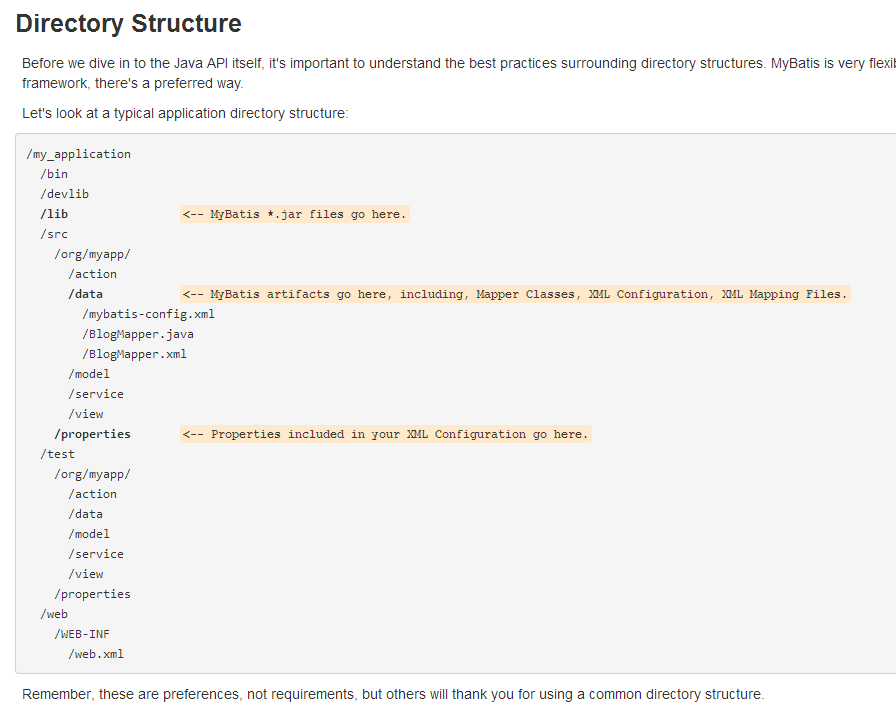
MyBatis-API

# mybatis-API

[http://www.mybatis.org/mybatis-3/java-api.html#](http://www.mybatis.org/mybatis-3/java-api.html)

# 建议的目录结构Directory Structure



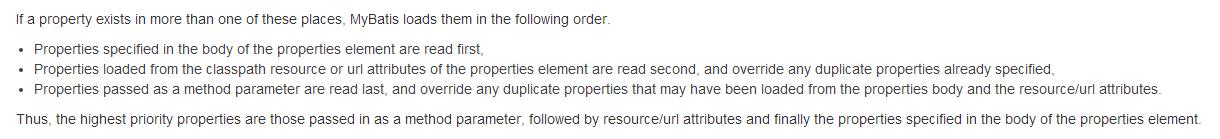


# SqlSessions：关键接口

**The primary Java interface for working with MyBatis is the SqlSession.**

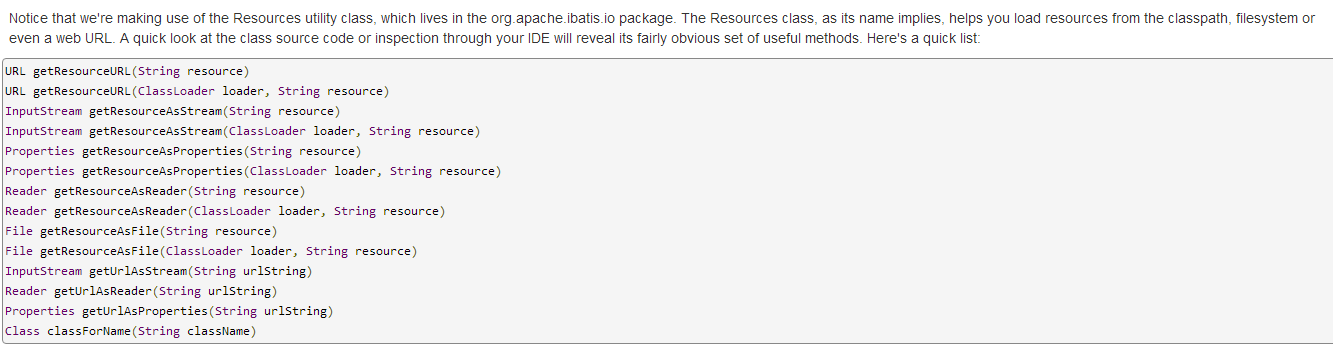
Through this interface you can execute commands, get mappers and manage transactions. We'll talk more about **SqlSession** itself shortly, but first we have to learn how to acquire an instance of **SqlSession**. SqlSessions are created by a **SqlSessionFactory** instance. The SqlSessionFactory contains methods for creating instances of SqlSessions all different ways. The SqlSessionFactory itself is created by the **SqlSessionFactoryBuilder** that can create the SqlSessonFactory from XML, Annotations or hand coded Java configuration.

**properties 属性加载的顺序**：

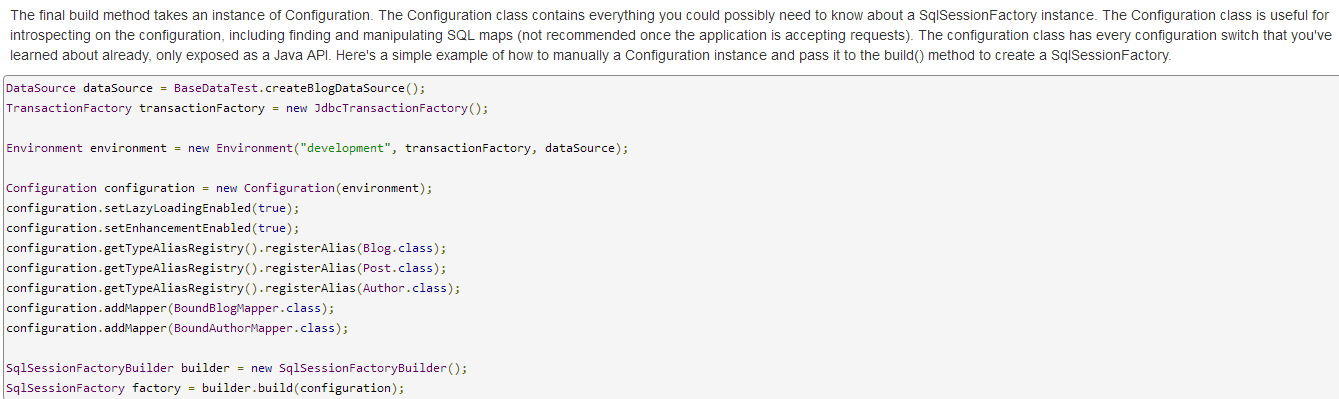


首先加载**全局配置文件中property标签指定**的，然后**加载properties中指定的properties文件**，最后是**利用代码手动配置**的。利用Configuration对象的addXxx方法添加。

# Resources工具类介绍



# 通过代码创建SqlsessionFactory



# SqlSessionFactory

对于一个数据库的连接，需要创建一个**SqlSessionFactory**，当然声明多个environments，可以创建多个**SqlSessionFactory对象**。注意一个SqlSessionFactory只可以连接一个数据库。

SqlSessionFactory的两大用途：

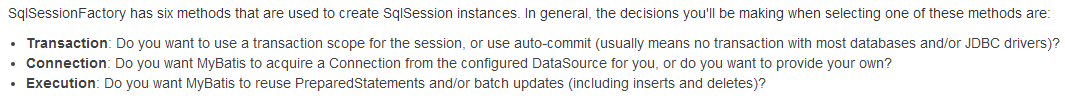
## 获取SqlSession （重中之重）

见下面介绍。

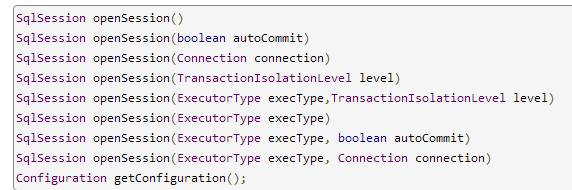
## 获取Configuration对象

除了介绍从**SqlSessionFactory**中获取SqlSession对象外，还有一个很重要的方法，就是getCounfigurantion()方法，获取**Configuration**对象，可以获取自己的配置信息。

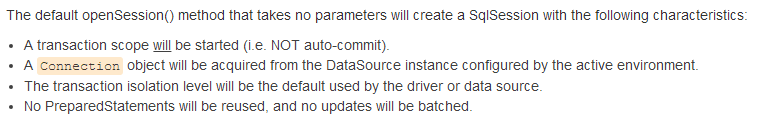
# 如何从SqlSessionFactory中获取SqlSession对象



* **Transaction**: Do you want to use a transaction scope for the session, or use auto-commit (usually means no transaction with most databases and/or JDBC drivers)?
* **Connection**: Do you want MyBatis to acquire a Connection from the configured DataSource for you, or do you want to provide your own?
* **Execution**: Do you want MyBatis to reuse PreparedStatements and/or batch updates (including inserts and deletes)?



## 无参数的openSession()



对于有参数的**openSession**方法，可以对事物、自动提交等功能进行限制。

## ExecutorType介绍

The one parameter that might be new to you is ExecutorType. This enumeration defines 3 values:

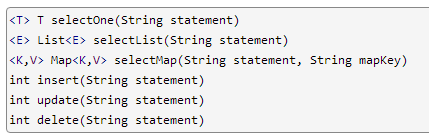
* ExecutorType.SIMPLE: This type of executor does nothing special. It creates a new PreparedStatement for each execution of a statement.
* ExecutorType.REUSE: This type of executor will reuse PreparedStatements.
* ExecutorType.BATCH: This executor will batch all update statements and demarcate them as necessary if SELECTs are executed between them, to ensure an easy-to-understand behavior.

# SqlSession的API（多余20个方法）分类介绍

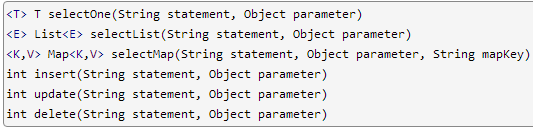
## Statement Execution Methods

**SELECT, INSERT, UPDATE and DELETE**

### 无参数的

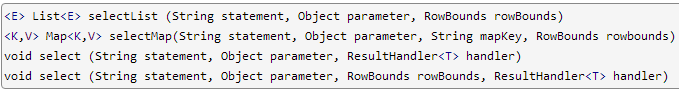


### 需要传入参数的



### 高级查询方法select（限定查询的行的范围）

Finally, there are three advanced versions of the select methods that allow you to restrict the range of rows to return, or provide custom result handling logic, usually for very large data sets



**RowBounds 介绍 见后面。**

The RowBounds parameter causes MyBatis to skip the number of records specified, as well as limit the number of results returned to some number. The RowBounds class has a constructor to take both the offset and limit, and is otherwise immutable.

注意：

### selectOne、selectList、selectMap的区别

只有明确知道只有一条记录或没有时，使用selectOne，否则要使用selectList。

在不知道具体记录个数时，尽量使用selectList，或者先查询count个数，然后选择使用。如果是多条记录，用selectOne查询，会报出异常。

The **selectMap** is a special case in that it is designed to convert a list of results into a Map based on one of the properties in the resulting objects.

### insert、update、delete返回的是影响的行数rows。

The value returned by the insert, update and delete methods indicate the number of rows affected by the statement.

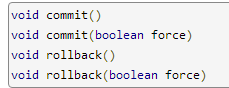
## Batch update statement Flush Method（批处理更新刷新方法）

There is method for flushing(executing) batch update statements that stored in a JDBC driver class at any timing. This method can be used when you use the ExecutorType.BATCH as ExecutorType.



## Transaction Control Methods（事物控制方法）

There are four methods for controlling the scope of a transaction. Of course, these have no effect if you've chosen to use auto-commit or if you're using an external transaction manager. However, if you're using the JDBC transaction manager, managed by the Connection instance, then the four methods that will come in handy are:



**By default MyBatis does not actually commit** unless it detects that the database has been changed by a call to insert, update or delete. If you've somehow made changes without calling these methods, then you can pass true into the commit and rollback methods to guarantee that it will be committed (note, you still can't force a session in auto-commit mode, or one that is using an external transaction manager). Most of the time you won't have to call rollback(), as MyBatis will do that for you if you don't call commit. However, if you need more fine grained control over a session where multiple commits and rollbacks are possible, you have the rollback option there to make that possible.

默认，MyBatis是不会自动提交的。在提交前，自动会回滚到提交之前。

## Local Cache（本地缓存）

MyBatis uses two caches: a local cache and a second level cache.

本地缓存和二级缓存。

Each time a new session is created MyBatis creates a local cache and attaches it to the session. Any query executed within the session will be stored in the local cache so further executions of the same query with the same input parameters will not hit the database. The local cache is cleared upon update, commit, rollback and close.

By default local cache data is used for the whole session duration. This cache is needed to resolve circular references and to speed up repeated nested queries, so it can never be completely disabled but you can configure the local cache to be used just for the duration of an statement execution by setting localCacheScope=STATEMENT.

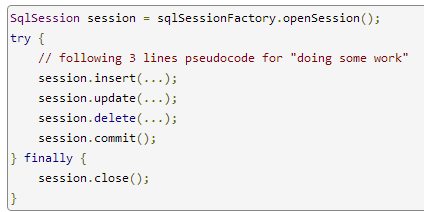
Note that when the localCacheScope is set to SESSION, MyBatis returns references to the same objects which are stored in the local cache. Any modification of returned object (lists etc.) influences the local cache contents and subsequently the values which are returned from the cache in the lifetime of the session. Therefore, as best practice, do not to modify the objects returned by MyBatis.



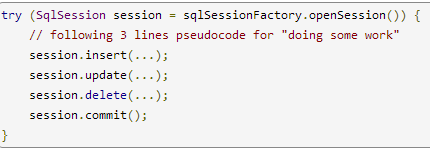
## close()方法 : Ensuring that SqlSession is Closed



The most important thing you must ensure is that you close any sessions that you open. The best way to ensure this is to use the following unit of work pattern:



Or, If you are using jdk 1.7+ and MyBatis 3.2+, you can use the try-with-resources statement:



## getConfiguration方法

 Just like SqlSessionFactory, you can get the instance of Configuration that the SqlSession is using by calling the getConfiguration() method.



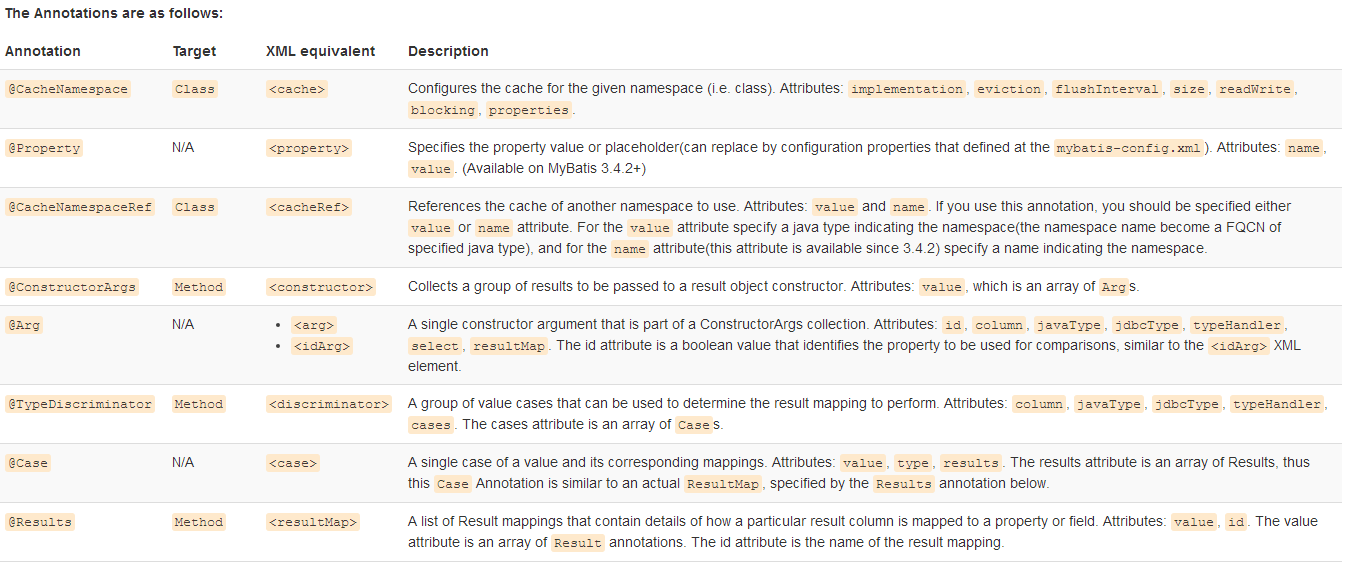
## Using Mappers ：getMapper方法

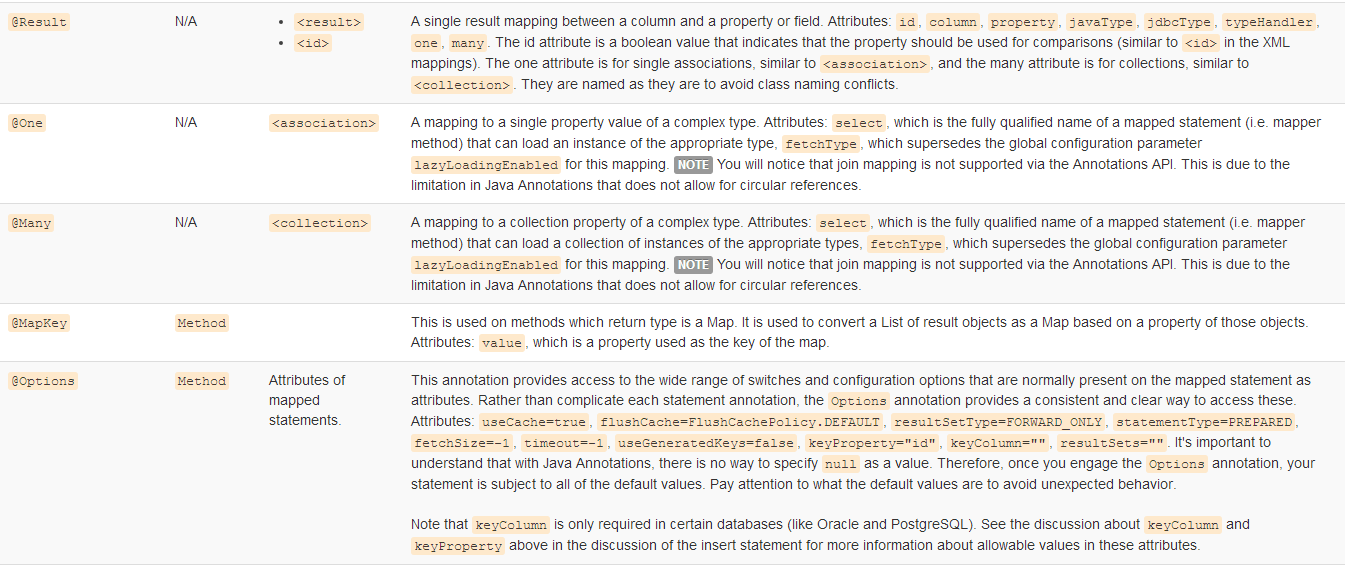
用于Mapper代理开发方式。

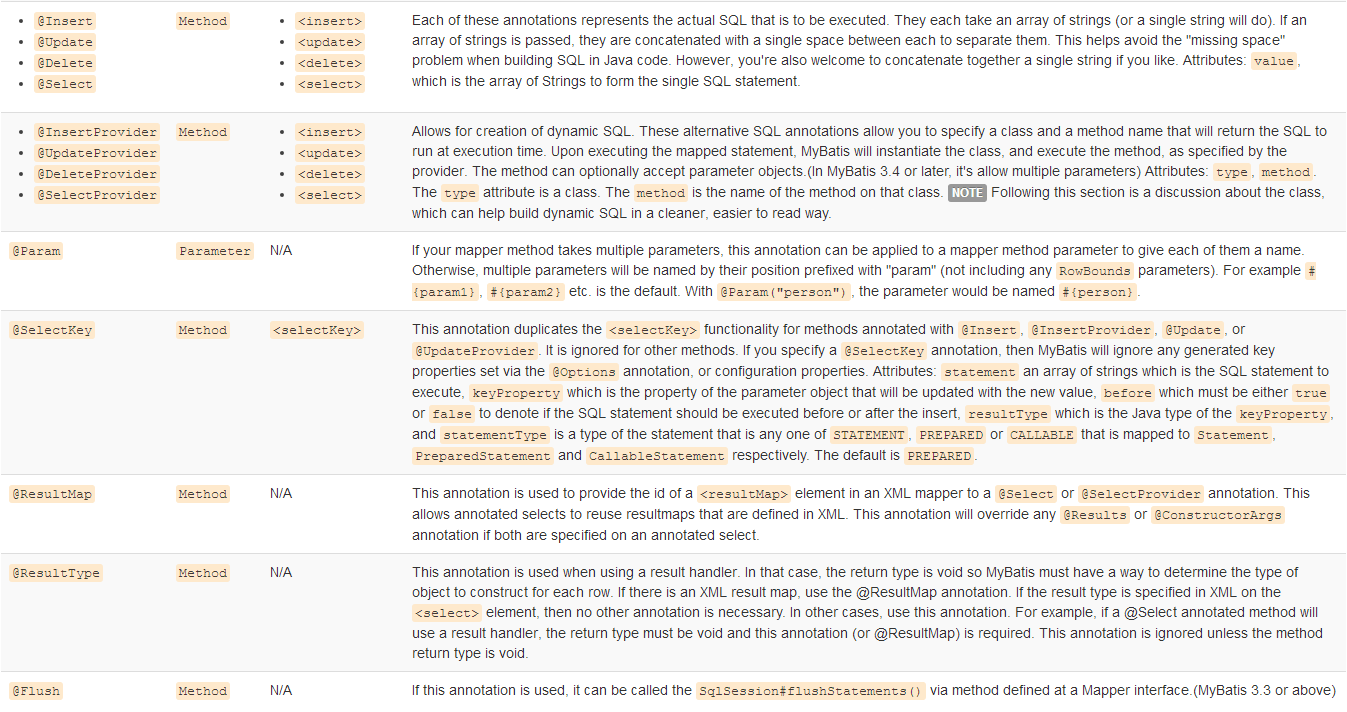
## Mapper Annotations Mapper注解

所有的注解详细参见，Mapper Annotations。

<http://www.mybatis.org/mybatis-3/java-api.html#sqlSessions>

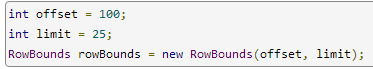






# RowBounds

The RowBounds parameter causes MyBatis to skip the number of records specified, as well as limit the number of results returned to some number. The RowBounds class has a constructor to take both the offset and limit, and is otherwise immutable.

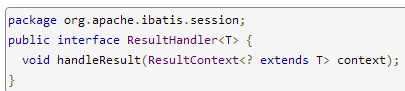


# ResultHandler 与 ResultContext

**Different drivers** are able to achieve different levels of efficiency in this regard. For the best performance, use result set types of **SCROLL\_SENSITIVE** or **SCROLL\_INSENSITIVE** (in other words: not FORWARD\_ONLY).

The **ResultHandler** parameter allows you to handle each row however you like. You can add it to a List, create a Map, Set, or throw each result away and instead keep only rolled up totals of calculations. You can do pretty much anything with the **ResultHandler**, and it's what MyBatis uses internally itself to build result set lists.

The interface is very simple.



The **ResultContext** parameter gives you access to the result object itself, a count of the number of result objects created, and a Boolean stop() method that you can use to stop MyBatis from loading any more results.

Using a ResultHandler has two limitations that you should be aware of:

* Data got from an method called with a ResultHandler will not be cached.
* When using advanced resultmaps MyBatis will probably require several rows to build an object. If a ResultHandler is used you may be given an object whose associations or collections are not yet filled.

# Mapper Annotation Examples:Mapper注解示例

This example shows using the **@SelectKey annotation** to **retrieve(获取) a value** from a sequence before an insert:

@Insert("insert into table3 (id, name) values(#{nameId}, #{name})")

@SelectKey(statement="call next value for TestSequence", keyProperty="nameId", before=**true**, resultType=**int.class**)

**int** insertTable3(Name name);

This example shows using the @**SelectKey** annotation to retrieve an identity value after an insert:

@Insert("insert into table2 (name) values(#{name})")

@SelectKey(statement="call identity()", keyProperty="nameId", before=**false**, resultType=**int.class**)

**int** insertTable2(Name name);

This example shows using the @Flush annotation to call the SqlSession#flushStatements():

@Flush

List<BatchResult> flush();

These examples show how to name a ResultMap by specifying id attribute of @Results annotation.

@Results(id = "userResult", value = {

@Result(property = "id", column = "uid", id = **true**),

@Result(property = "firstName", column = "first\_name"),

@Result(property = "lastName", column = "last\_name")

})

@Select("select \* from users where id = #{id}")

User getUserById(Integer id);

@Results(id = "companyResults")

@ConstructorArgs({

@Arg(property = "id", column = "cid", id = **true**),

@Arg(property = "name", column = "name")

})

@Select("select \* from company where id = #{id}")

Company getCompanyById(Integer id);

This example shows solo parameter using the SelectProvider annotation:

@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName")

List<User> getUsersByName(String name);

class UserSqlBuilder {

public String buildGetUsersByName(final String name) {

return new SQL(){{

SELECT("\*");

FROM("users");

if (name != null) {

WHERE("name like #{value} || '%'");

}

ORDER\_BY("id");

}}.toString();

}

}

This example shows multiple parameters using the Sql Provider annotation:

@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName")

List<User> getUsersByName(

@Param("name") String name, @Param("orderByColumn") String orderByColumn);

class UserSqlBuilder {

// If not use @Param, you should be define same arguments with mapper method

public String buildGetUsersByName(

final String name, final String orderByColumn) {

return new SQL(){{

SELECT("\*");

FROM("users");

WHERE("name like #{name} || '%'");

ORDER\_BY(orderByColumn);

}}.toString();

}

// If use @Param, you can define only arguments to be used

public String buildGetUsersByName(@Param("orderByColumn") final String orderByColumn) {

return new SQL(){{

SELECT("\*");

FROM("users");

WHERE("name like #{name} || '%'");

ORDER\_BY(orderByColumn);

}}.toString();

}

}