Java API

既然你已经知道如何配置 MyBatis 和创建映射文件，你就已经准备好来提升技能了。MyBatis 的 Java API 就是你收获你所做的努力的地方。正如你即将看到的，和 JDBC 相比，MyBatis 很大程度简化了你的代码并保持代码简洁，容易理解并维护。MyBatis 3 已经引入了很多重要的改进来使得 SQL 映射更加优秀。

应用目录结构

在我们深入 Java API 之前，理解关于目录结构的最佳实践是很重要的。MyBatis 非常灵活，你可以用你自己的文件来做几乎所有的事情。但是对于任一框架，都有一些最佳的方式。

让我们看一下典型的应用目录结构：

```
/my_application
  /bin
  /devlib
  /lib                <-- MyBatis *.jar 文件在这里。
  /src
    /org/myapp/
      /action
      /data           <-- MyBatis 配置文件在这里，包括映射器类，XML 配置，XML 映射文件。
        /mybatis-config.xml
        /BlogMapper.java
        /BlogMapper.xml
      /model
      /service
      /view
    /properties       <-- 在你 XML 中配置的属性文件在这里。
  /test
    /org/myapp/
      /action
      /data
      /model
      /service
      /view
    /properties
  /web
    /WEB-INF
      /web.xml
```

当然这是推荐的目录结构，并非强制要求，但是使用一个通用的目录结构将更利于大家沟通。

这部分内容剩余的示例将假设你使用了这种目录结构。

SqlSession

使用 MyBatis 的主要 Java 接口就是 SqlSession。你可以通过这个接口来执行命令，获取映射器和管理事务。我们会概括讨论一下 SqlSession 本身，但是首先我们还是要了解如何获取一个 SqlSession 实例。SqlSession 是由 SqlSessionFactory 实例创建的。SqlSessionFactory 对象包含创建 SqlSession 实例的所有方法。而 SqlSessionFactory 本身是由 SqlSessionFactoryBuilder 创建的，它可以从 XML、注解或手动配置 Java 代码来创建 SqlSessionFactory。

注意 当 Mybatis 与一些依赖注入框架（如 Spring 或者 Guice）同时使用时，SqlSession 将被依赖注入框架所创建，所以你不需要使用 SqlSessionFactoryBuilder 或者 SqlSessionFactory，可以直接看 SqlSession 这一节。请参考 Mybatis-Spring 或者 Mybatis-Guice 手册了解更多信息。

SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 有五个 build() 方法，每一种都允许你从不同的资源中创建一个 SqlSessionFactory 实例。

```
SqlSessionFactory build(InputStream inputStream)
SqlSessionFactory build(InputStream inputStream, String environment)
SqlSessionFactory build(InputStream inputStream, Properties properties)
SqlSessionFactory build(InputStream inputStream, String env, Properties props)
SqlSessionFactory build(Configuration config)
```

第一种方法是最常用的，它使用了一个参照了 XML 文档或上面讨论过的更特定的 mybatis-config.xml 文件的 Reader 实例。可选的参数是 environment 和 properties。environment 决定加载哪种环境，包括数据源和事务管理器。比如：

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
        ...
    <dataSource type="POOLED">
        ...
  </environment>
  <environment id="production">
    <transactionManager type="MANAGED">
        ...
    <dataSource type="JNDI">
        ...
  </environment>
</environments>
```

如果你调用了参数有 environment 的 build 方法，那么 MyBatis 将会使用 configuration 对象来配置这个 environment。当然，如果你指定了一个不合法的 environment，你就会得到错误提示。如果你调用了不带 environment 参数的 build 方法，那么就使用默认的 environment（在上面的示例中指定为 default="development" 的代码）。

如果你调用了参数有 properties 实例的方法，那么 MyBatis 就会加载那些 properties（属性配置文件），并在配置中可用。那些属性可以用${propName} 语法形式多次用在配置文件中。

回想一下，属性可以从 mybatis-config.xml 中被引用，或者直接指定它。因此理解优先级是很重要的。我们在文档前面已经提及它了，但是这里要再次重申：

如果一个属性存在于这些位置，那么 MyBatis 将会按照下面的顺序来加载它们：

- 首先读取在 properties 元素体中指定的属性；
- 其次，读取从 properties 元素的类路径 resource 或 url 指定的属性，且会覆盖已经指定了的重复属性；
- 最后，读取作为方法参数传递的属性，且会覆盖已经从 properties 元素体和 resource 或 url 属性中加载了的重复属性。

因此，通过方法参数传递的属性的优先级最高，resource 或 url 指定的属性优先级中等，在 properties 元素体中指定的属性优先级最低。

总结一下，前四个方法很大程度上是相同的，但是由于覆盖机制，便允许你可选地指定 environment 和/或 properties。以下给出一个从 mybatis-config.xml 文件创建 SqlSessionFactory 的示例：

```
String resource = "org/mybatis/builder/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
```

注意到这里我们使用了 Resources 工具类，这个类在 org.apache.ibatis.io 包中。Resources 类正如其名，会帮助你从类路径下、文件系统或一个 web URL 中加载资源文件。看一下这个类的源代码或者通过你的 IDE 来查看，就会看到一整套相当实用的方法。这里给出一个简表：

```
URL getResourceURL(String resource)
URL getResourceURL(ClassLoader loader, String resource)
InputStream getResourceAsStream(String resource)
InputStream getResourceAsStream(ClassLoader loader, String resource)
Properties getResourceAsProperties(String resource)
Properties getResourceAsProperties(ClassLoader loader, String resource)
Reader getResourceAsReader(String resource)
Reader getResourceAsReader(ClassLoader loader, String resource)
File getResourceAsFile(String resource)
File getResourceAsFile(ClassLoader loader, String resource)
InputStream getUrlAsStream(String urlString)
Reader getUrlAsReader(String urlString)
Properties getUrlAsProperties(String urlString)
Class classForName(String className)
```

最后一个 build 方法的参数为 Configuration 实例。configuration 类包含你可能需要了解 SqlSessionFactory 实例的所有内容。Configuration 类对于配置的自查很有用，它包含查找和操作 SQL 映射（当应用接收请求时便不推荐使用）。作为一个 Java API 的 configuration 类具有所有配置的开关，这些你已经了解了。这里有一个简单的示例，教你如何手动配置 configuration 实例，然后将它传递给 build() 方法来创建 SqlSessionFactory。

```
DataSource dataSource = BaseDataTest.createBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();

Environment environment = new Environment("development", transactionFactory, dataSource);

Configuration configuration = new Configuration(environment);
configuration.setLazyLoadingEnabled(true);
configuration.setEnhancementEnabled(true);
configuration.getTypeAliasRegistry().registerAlias(Blog.class);
configuration.getTypeAliasRegistry().registerAlias(Post.class);
configuration.getTypeAliasRegistry().registerAlias(Author.class);
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);

SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(configuration);
```

现在你就获得一个可以用来创建 SqlSession 实例的 SqlSessionFactory 了！

SqlSessionFactory

SqlSessionFactory 有六个方法创建 SqlSession 实例。通常来说，当你选择这些方法时你需要考虑以下几点：

- **事务处理**：我需要在 session 使用事务或者使用自动提交功能（auto-commit）吗？（通常意味着很多数据库和/或 JDBC 驱动没有事务）
- **连接**：我需要依赖 MyBatis 获得来自数据源的配置吗？还是使用自己提供的配置？
- **执行语句**：我需要 MyBatis 复用预处理语句和/或批量更新语句（包括插入和删除）吗？

基于以上需求，有下列已重载的多个 openSession() 方法供使用。

```
SqlSession openSession()
SqlSession openSession(boolean autoCommit)
SqlSession openSession(Connection connection)
SqlSession openSession(TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCommit)
SqlSession openSession(ExecutorType execType, Connection connection)
Configuration getConfiguration();
```

默认的 openSession()方法没有参数，它会创建有如下特性的 SqlSession：

- 会开启一个事务（也就是不自动提交）。
- 将从由当前环境配置的 DataSource 实例中获取 Connection 对象。
- 事务隔离级别将会使用驱动或数据源的默认设置。
- 预处理语句不会被复用，也不会批量处理更新。

这些方法大都是可读性强的。向 `autoCommit` 可选参数传递 `true` 值即可开启自动提交功能。若要使用自己的 `Connection` 实例，传递一个 `Connection` 实例给 `connection` 参数即可。注意并未覆写同时设置 `Connection` 和 `autoCommit` 两者的方法，因为 MyBatis 会使用正在使用中的、设置了 Connection 的环境。MyBatis 为事务隔离级别调用使用了一个 Java 枚举包装器，称为 `TransactionIsolationLevel` ，若不使用它，将使用 JDBC 所支持五个隔离级（ `NONE` 、 `READ_UNCOMMITTED` 、 `READ_COMMITTED` 、 `REPEATABLE_READ` 和 `SERIALIZABLE` ），并按它们预期的方式来工作。

还有一个可能对你来说是新见到的参数，就是 `ExecutorType` 。这个枚举类型定义了三个值:

- `ExecutorType.SIMPLE` ：这个执行器类型不做特殊的事情。它为每个语句的执行创建一个新的预处理语句。
- `ExecutorType.REUSE` ：这个执行器类型会复用预处理语句。
- `ExecutorType.BATCH` ：这个执行器会批量执行所有更新语句，如果 SELECT 在它们中间执行，必要时请把它们区分开来以保证行为的易读性。

注意 在 SqlSessionFactory 中还有一个方法我们没有提及，就是 getConfiguration()。这 个方法会返回一个 Configuration 实例，在运行时你可以使用它来自检 MyBatis 的配置。

注意 如果你使用的是 MyBatis 之前的版本，你要重新调用 openSession，因为旧版本的 session、事务和批量操作是分离开来的。如果使用的是新版本，那么就不必这么做了，因为它们现在都包含在 session 的作用域内了。你不必再单独处理事务或批量操作就能得到想要的全部效果。

SqlSession

正如上面所提到的，SqlSession 实例在 MyBatis 中是非常强大的一个类。在这里你会看到所有执行语句、提交或回滚事务和获取映射器实例的方法。

在 SqlSession 类中有超过 20 个方法，所以将它们组合成易于理解的分组。

执行语句方法

这些方法被用来执行定义在 SQL 映射的 XML 文件中的 SELECT、INSERT、UPDATE 和 DELETE 语句。它们都会自行解释，每一句都使用语句的 ID 属性和参数对象，参数可以是原生类型（自动装箱或包装类）、JavaBean、POJO 或 Map。

```
<T> T selectOne(String statement, Object parameter)
<E> List<E> selectList(String statement, Object parameter)
<T> Cursor<T> selectCursor(String statement, Object parameter)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

selectOne 和 selectList 的不同仅仅是 selectOne 必须返回一个对象或 null 值。如果返回值多于一个，那么就会抛出异常。如果你不知道返回对象的数量，请使用 selectList。如果需要查看返回对象是否存在，可行的方案是返回一个值即可（0 或 1）。selectMap 稍微特殊一点，因为它会将返回的对象的其中一个属性作为 key 值，将对象作为 value 值，从而将多结果集转为 Map 类型值。因为并不是所有语句都需要参数，所以这些方法都重载成不需要参数的形式。

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

```
<T> T selectOne(String statement)
<E> List<E> selectList(String statement)
<T> Cursor<T> selectCursor(String statement)
<K,V> Map<K,V> selectMap(String statement, String mapKey)
int insert(String statement)
int update(String statement)
int delete(String statement)
```

A Cursor offers the same results as a List, except it fetches data lazily using an Iterator.

```
try (Cursor<MyEntity> entities = session.selectCursor(statement, param)) {
   for (MyEntity entity:entities) {
      // process one entity
   }
}
```

最后，还有 select 方法的三个高级版本，它们允许你限制返回行数的范围，或者提供自定义结果控制逻辑，这通常在数据集合庞大的情形下使用。

```
<E> List<E> selectList (String statement, Object parameter, RowBounds rowBounds)
<T> Cursor<T> selectCursor(String statement, Object parameter, RowBounds rowBounds)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey, RowBounds rowbounds)
void select (String statement, Object parameter, ResultHandler<T> handler)
void select (String statement, Object parameter, RowBounds rowBounds, ResultHandler<T> handler)
```

RowBounds 参数会告诉 MyBatis 略过指定数量的记录，还有限制返回结果的数量。RowBounds 类有一个构造方法来接收 offset 和 limit，另外，它们是不可二次赋值的。

```
int offset = 100;
int limit = 25;
RowBounds rowBounds = new RowBounds(offset, limit);
```

所以在这方面，不同的驱动能够取得不同级别的高效率。为了取得最佳的表现，请使用结果集的 SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE 的类型(换句话说：不用 FORWARD_ONLY)。

ResultHandler 参数允许你按你喜欢的方式处理每一行。你可以将它添加到 List 中、创建 Map 和 Set，或者丢弃每个返回值都可以，它取代了仅保留执行语句过后的总结果列表的死板结果。你可以使用 ResultHandler 做很多事，并且这是 MyBatis 自身内部会使用的方法，以创建结果集列表。

Since 3.4.6, ResultHandler passed to a CALLABLE statement is used on every REFCURSOR output parameter of the stored procedure if there is any.

它的接口很简单。

```
package org.apache.ibatis.session;
public interface ResultHandler<T> {
  void handleResult(ResultContext<? extends T> context);
}
```

ResultContext 参数允许你访问结果对象本身、被创建的对象数目、以及返回值为 Boolean 的 stop 方法，你可以使用此 stop 方法来停止 MyBatis 加载更多的结果。

使用 ResultHandler 的时候需要注意以下两种限制：

- 从被 ResultHandler 调用的方法返回的数据不会被缓存。
- 当使用结果映射集（resultMap）时，MyBatis 大多数情况下需要数行结果来构造外键对象。如果你正在使用 ResultHandler，你可以给出外键（association）或者集合（collection）尚未赋值的对象。

批量立即更新方法

有一个方法可以刷新（执行）存储在 JDBC 驱动类中的批量更新语句。当你将 `ExecutorType.BATCH` 作为 `ExecutorType` 使用时可以采用此方法。

```
List<BatchResult> flushStatements()
```

事务控制方法

控制事务作用域有四个方法。当然，如果你已经设置了自动提交或你正在使用外部事务管理器，这就没有任何效果了。然而，如果你正在使用 JDBC 事务管理器，由Connection 实例来控制，那么这四个方法就会派上用场：

```
void commit()
void commit(boolean force)
void rollback()
void rollback(boolean force)
```

默认情况下 MyBatis 不会自动提交事务，除非它侦测到有插入、更新或删除操作改变了数据库。如果你已经做出了一些改变而没有使用这些方法，那么你可以传递 true 值到 commit 和 rollback 方法来保证事务被正常处理（注意，在自动提交模式或者使用了外部事务管理器的情况下设置 force 值对 session 无效）。很多时候你不用调用 rollback()，因为 MyBatis 会在你没有调用 commit 时替你完成回滚操作。然而，如果你需要在支持多提交和回滚的 session 中获得更多细粒度控制，你可以使用回滚操作来达到目的。

注意 MyBatis-Spring 和 MyBatis-Guice 提供了声明事务处理，所以如果你在使用 Mybatis 的同时使用了Spring 或者 Guice，那么请参考它们的手册以获取更多的内容。

本地缓存

Mybatis 使用到了两种缓存：本地缓存（local cache）和二级缓存（second level cache）。

每当一个新 session 被创建，MyBatis 就会创建一个与之相关联的本地缓存。任何在 session 执行过的查询语句本身都会被保存在本地缓存中，那么，相同的查询语句和相同的参数所产生的更改就不会二度影响数据库了。本地缓存会被增删改、提交事务、关闭事务以及关闭 session 所清空。

默认情况下，本地缓存数据可在整个 session 的周期内使用，这一缓存需要被用来解决循环引用错误和加快重复嵌套查询的速度，所以它可以不被禁用掉，但是你可以设置 localCacheScope=STATEMENT 表示缓存仅在语句执行时有效。

注意，如果 localCacheScope 被设置为 SESSION，那么 MyBatis 所返回的引用将传递给保存在本地缓存里的相同对象。对返回的对象（例如 list）做出任何更新将会影响本地缓存的内容，进而影响存活在 session 生命周期中的缓存所返回的值。因此，不要对 MyBatis 所返回的对象作出更改，以防后患。

你可以随时调用以下方法来清空本地缓存：

```
void clearCache()
```

确保 SqlSession 被关闭

```
void close()
```

你必须保证的最重要的事情是你要关闭所打开的任何 session。保证做到这点的最佳方式是下面的工作模式:

```
try (SqlSession session = sqlSessionFactory.openSession()) {
    // following 3 lines pseudocod for "doing some work"
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
}
```

注意 就像 SqlSessionFactory，你可以通过调用当前使用中的 SqlSession 的 getConfiguration 方法来获得 Configuration 实例。

```
Configuration getConfiguration()
```

使用映射器

```
<T> T getMapper(Class<T> type)
```

上述的各个 insert、update、delete 和 select 方法都很强大，但也有些繁琐，可能会产生类型安全问题并且对于你的 IDE 和单元测试也没有实质性的帮助。在上面的入门章节中我们已经看到了一个使用映射器的示例。

因此，一个更通用的方式来执行映射语句是使用映射器类。一个映射器类就是一个仅需声明与 SqlSession 方法相匹配的方法的接口类。下面的示例展示了一些方法签名以及它们是如何映射到 SqlSession 上的。

```
public interface AuthorMapper {
  // (Author) selectOne("selectAuthor",5);
  Author selectAuthor(int id);
  // (List<Author>) selectList(“selectAuthors”)
  List<Author> selectAuthors();
  // (Map<Integer,Author>) selectMap("selectAuthors", "id")
  @MapKey("id")
  Map<Integer, Author> selectAuthors();
  // insert("insertAuthor", author)
  int insertAuthor(Author author);
  // updateAuthor("updateAuthor", author)
  int updateAuthor(Author author);
  // delete("deleteAuthor",5)
  int deleteAuthor(int id);
}
```

总之，每个映射器方法签名应该匹配相关联的 SqlSession 方法，而字符串参数 ID 无需匹配。相反，方法名必须匹配映射语句的 ID。

此外，返回类型必须匹配期望的结果类型，单返回值时为所指定类的值，多返回值时为数组或集合。所有常用的类型都是支持的，包括：原生类 型、Map、POJO 和 JavaBean。

注意 映射器接口不需要去实现任何接口或继承自任何类。只要方法可以被唯一标识对应的映射语句就可以了。

注意 映射器接口可以继承自其他接口。当使用 XML 来构建映射器接口时要保证语句被包含在合适的命名空间中。而且，唯一的限制就是你不能在两个继承关系的接口中拥有相同的方法签名（潜在的危险做法不可取）。

你可以传递多个参数给一个映射器方法。如果你这样做了，默认情况下它们将会以 "param" 字符串紧跟着它们在参数列表中的位置来命名，比如：#{param1}、#{param2}等。如果你想改变参数的名称（只在多参数情况下），那么你可以在参数上使用 @Param("paramName") 注解。

你也可以给方法传递一个 RowBounds 实例来限制查询结果。

映射器注解

因为最初设计时，MyBatis 是一个 XML 驱动的框架。配置信息是基于 XML 的，而且映射语句也是定义在 XML 中的。而到了 MyBatis 3，就有新选择了。MyBatis 3 构建在全面且强大的基于 Java 语言的配置 API 之上。这个配置 API 是基于 XML 的 MyBatis 配置的基础，也是新的基于注解配置的基础。注解提供了一种简单的方式来实现简单映射语句，而不会引入大量的开销。

注意 不幸的是，Java 注解的的表达力和灵活性十分有限。尽管很多时间都花在调查、设计和试验上，最强大的 MyBatis 映射并不能用注解来构建——并不是在开玩笑，的确是这样。比方说，C#属性就没有这些限制，因此 MyBatis.NET 将会比 XML 有更丰富的选择。也就是说，基于 Java 注解的配置离不开它的特性。

注解如下表所示：

注解	使用对象	相对应的 XML	描述
`@CacheNamespace`	类	`<cache>`	为给定的命名空间（比如类）配置缓存。属性有：`implemetation`，`eviction`，`flushInterval`，`size`，`readWrite`，`blocking` 和`properties`。
`@Property`	N/A	`<property>`	指定参数值或占位值（placeholder）（能被 `mybatis-config.xml` 内的配置属性覆盖）。属性有：`name`，`value`。（仅在MyBatis 3.4.2以上版本生效）
`@CacheNamespaceRef`	类	`<cacheRef>`	参照另外一个命名空间的缓存来使用。属性有：`value`，`name`。如果你使用了这个注解，你应设置 `value` 或者 `name` 属性的其中一个。`value` 属性用于指定 Java 类型而指定命名空间（命名空间名就是指定的 Java 类型的全限定名），`name` 属性（这个属性仅在MyBatis 3.4.2以上版本生效）直接指定了命名空间的名字。
`@ConstructorArgs`	方法	`<constructor>`	收集一组结果传递给一个结果对象的构造方法。属性有：`value`，它是形式参数数组。
`@Arg`	N/A	• `<arg>` • `<idArg>`	单参数构造方法，是 ConstructorArgs 集合的一部分。属性有：`id`，`column`，`javaType`，`jdbcType`，`typeHandler`，`select` 和 `resultMap`。id 属性是布尔值，来标识用于比较的属性，和 `<idArg>` XML 元素相似。
`@TypeDiscriminator`	方法	`<discriminator>`	一组实例值被用来决定结果映射的表现。属性有：`column`，`javaType`，`jdbcType`，`typeHandler` 和 `cases`。cases 属性是实例数组。

注解	使用对象	相对应的 XML	描述
`@Case`	N/A	`<case>`	单独实例的值和它对应的映射。属性有：`value`，`type`，`results`。results 属性是结果数组，因此这个注解和实际的 `ResultMap` 很相似，由下面的 `Results` 注解指定。
`@Results`	方法	`<resultMap>`	结果映射的列表，包含了一个特别结果列如何被映射到属性或字段的详情。属性有：`value`，`id`。value 属性是 `Result` 注解的数组。这个 id 的属性是结果映射的名称。
`@Result`	N/A	• `<result>` • `<id>`	在列和属性或字段之间的单独结果映射。属性有：`id`，`column`，`javaType`，`jdbcType`，`typeHandler`，`one`，`many`。id 属性是一个布尔值，来标识应该被用于比较（和在 XML 映射中的 `<id>` 相似）的属性。one 属性是单独的联系，和 `<association>` 相似，而 many 属性是对集合而言的，和 `<collection>` 相似。它们这样命名是为了避免名称冲突。
`@One`	N/A	`<association>`	复杂类型的单独属性值映射。属性有：`select`，已映射语句（也就是映射器方法）的全限定名，它可以加载合适类型的实例。`fetchType` 会覆盖全局的配置参数 `lazyLoadingEnabled`。 **注意** 联合映射在注解 API中是不支持的。这是因为 Java 注解的限制,不允许循环引用。
`@Many`	N/A	`<collection>`	映射到复杂类型的集合属性。属性有：`select`，已映射语句（也就是映射器方法）的全限定名，它可以加载合适类型的实例的集合，`fetchType` 会覆盖全局的配置参数 `lazyLoadingEnabled`。 **注意** 联合映射在注解 API中是不支持的。这是因为 Java 注解的限制，不允许循环引用
`@MapKey`	方法		这是一个用在返回值为 Map 的方法上的注解。它能够将存放对象的 List 转化为 key 值为对象的某一属性的 Map。属性有：`value`，填入的是对象的属性名，作为 Map 的 key 值。

注解	使用对象	相对应的 XML	描述
`@Options`	方法	映射语句的属性	这个注解提供访问大范围的交换和配置选项的入口，它们通常在映射语句上作为属性出现。 `Options` 注解提供了通俗易懂的方式来访问它们，而不是让每条语句注解变复杂。属性有： `useCache=true`， `flushCache=FlushCachePolicy.DEFAULT`， `resultSetType=DEFAULT`， `statementType=PREPARED`， `fetchSize=-1`， `timeout=-1`， `useGeneratedKeys=false`， `keyProperty=""`， `keyColumn=""`， `resultSets=""`。值得一提的是， Java 注解无法指定 `null` 值。因此，一旦你使用了 `Options` 注解，你的语句就会被上述属性的默认值所影响。要注意避免默认值带来的预期以外的行为。 注意： `keyColumn` 属性只在某些数据库中有效（如 Oracle、PostgreSQL等）。请在插入语句一节查看更多关于 `keyColumn` 和 `keyProperty` 两者的有效值详情。
• `@Insert` • `@Update` • `@Delete` • `@Select`	方法	• `<insert>` • `<update>` • `<delete>` • `<select>`	这四个注解分别代表将会被执行的 SQL 语句。它们用字符串数组（或单个字符串）作为参数。如果传递的是字符串数组，字符串之间先会被填充一个空格再连接成单个完整的字符串。这有效避免了以 Java 代码构建 SQL 语句时的“丢失空格”的问题。然而，你也可以提前手动连接好字符串。属性有： `value` ，填入的值是用来组成单个 SQL 语句的字符串数组。
• `@InsertProvider` • `@UpdateProvider` • `@DeleteProvider` • `@SelectProvider`	方法	• `<insert>` • `<update>` • `<delete>` • `<select>`	允许构建动态 SQL。这些备选的 SQL 注解允许你指定类名和返回在运行时执行的 SQL 语句的方法。（自从MyBatis 3.4.6开始，你可以用 `CharSequence` 代替 `String` 来返回类型返回值了。）当执行映射语句的时候，MyBatis 会实例化类并执行方法，类和方法就是填入了注解的值。你可以把已经传递给映射方法了的对象作为参数，"Mapper interface type" 和 "Mapper method" and "Database ID" 会经过 `ProviderContext` （仅在MyBatis 3.4.5及以上支持）作为参数值。（MyBatis 3.4及以上的版本，支持多参数传入）属性有： `value`，`type`，`method`。 `value` and `type` 属性需填入类(The `type` attribute is alias for `value`, you must be specify either one)。 `method` 需填入该类定义了的方法名 (Since 3.5.1, you can omit `method` attribute, the MyBatis will resolve a target method via the `ProviderMethodResolver` interface. If not resolve by it, the MyBatis use the reserved fallback method that named `provideSql`)。 **注意** 接下来的小节将会讨论类，能帮助你更轻松地构建动态 SQL。

注解	使用对象	相对应的 XML	描述
`@Param`	参数	N/A	如果你的映射方法的形参有多个，这个注解使用在映射方法的参数上就能为它们取自定义名字。若不给出自定义名字，多参数（不包括 `RowBounds` 参数）则先以 "param" 作前缀，再加上它们的参数位置作为参数别名。例如 `#{param1}`，`#{param2}` ，这个是默认值。如果注解是 `@Param("person")` ，那么参数就会被命名为 `#{person}`。
`@SelectKey`	方法	`<selectKey>`	这个注解的功能与 `<selectKey>` 标签完全一致，用在已经被 `@Insert` 或 `@InsertProvider` 或 `@Update` 或 `@UpdateProvider` 注解了的方法上。若在未被上述四个注解的方法上作 `@SelectKey` 注解则视为无效。如果你指定了 `@SelectKey` 注解，那么 MyBatis 就会忽略掉由 `@Options` 注解所设置的生成主键或设置（configuration）属性。属性有： `statement` 填入将会被执行的 SQL 字符串数组， `keyProperty` 填入将会被更新的参数对象的属性的值， `before` 填入 `true` 或 `false` 以指明 SQL 语句应被在插入语句的之前还是之后执行。 `resultType` 填入 `keyProperty` 的 Java 类型和用 `Statement`、`PreparedStatement` 和 `CallableStatement` 中的 `STATEMENT`、 `PREPARED` 或 `CALLABLE` 中任一值填入 `statementType`。默认值是 `PREPARED`。
`@ResultMap`	方法	N/A	这个注解给 `@Select` 或者 `@SelectProvider` 提供在 XML 映射中的 `<resultMap>` 的id。这使得注解的 select 可以复用那些定义在 XML 中的 ResultMap。如果同一 select 注解中还存在 `@Results` 或者 `@ConstructorArgs` ，那么这两个注解将被此注解覆盖。
`@ResultType`	方法	N/A	此注解在使用了结果处理器的情况下使用。在这种情况下，返回类型为 void，所以 Mybatis 必须有一种方式决定对象的类型，用于构造每行数据。如果有 XML 的结果映射，请使用 `@ResultMap` 注解。如果结果类型在 XML 的 `<select>` 节点中指定了，就不需要其他的注解了。其他情况下则使用此注解。比如，如果 @Select 注解在一个将使用结果处理器的方法上，那么返回类型必须是 void 并且这个注解（或者@ResultMap）必选。这个注解仅在方法返回类型是 void 的情况下生效。
`@Flush`	方法	N/A	如果使用了这个注解，定义在 Mapper 接口中的方法能够调用 `SqlSession#flushStatements()` 方法。（Mybatis 3.3及以上）

映射申明样例

这个例子展示了如何使用 @SelectKey 注解来在插入前读取数据库序列的值：

```
@Insert("insert into table3 (id, name) values(#{nameId}, #{name})")
@SelectKey(statement="call next value for TestSequence", keyProperty="nameId", before=true, resultT
ype=int.class)
int insertTable3(Name name);
```

这个例子展示了如何使用 @SelectKey 注解来在插入后读取数据库识别列的值：

```
@Insert("insert into table2 (name) values(#{name})")
@SelectKey(statement="call identity()", keyProperty="nameId", before=false, resultType=int.class)
int insertTable2(Name name);
```

这个例子展示了如何使用 `@Flush` 注解去调用 `SqlSession#flushStatements()` ：

```
@Flush
List<BatchResult> flush();
```

这些例子展示了如何通过指定 @Result 的 id 属性来命名结果集：

```
@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(column = "cid", javaType = Integer.class, id = true),
  @Arg(column = "name", javaType = String.class)
})
@Select("select * from company where id = #{id}")
Company getCompanyById(Integer id);
```

这个例子展示了单一参数使用 @SqlProvider 注解：

```
@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName")
List<User> getUsersByName(String name);

class UserSqlBuilder {
  public static String buildGetUsersByName(final String name) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      if (name != null) {
        WHERE("name like #{value} || '%'");
      }
      ORDER_BY("id");
    }}.toString();
  }
}
```

这个例子展示了多参数使用 @SqlProvider 注解：

```
@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 static 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 static String buildGetUsersByName(@Param("orderByColumn") final String orderByColumn) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      WHERE("name like #{name} || '%'");
      ORDER_BY(orderByColumn);
    }}.toString();
  }
}
```

This example shows usage the default implementation of `ProviderMethodResolver` (available since MyBatis 3.5.1 or later):

```
@SelectProvider(UserSqlProvider.class)
List<User> getUsersByName(String name);

// Implements the ProviderMethodResolver on your provider class
class UserSqlProvider implements ProviderMethodResolver {
  // In default implementation, it will resolve a method that method name is matched with mapper me
thod
  public static String getUsersByName(final String name) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      if (name != null) {
        WHERE("name like #{value} || '%'");
      }
      ORDER_BY("id");
    }}.toString();
  }
}
```