mybatis
(./)

Java API

既然你已经知道如何配置 MyBatis 以及如何创建映射，是时候来尝点甜头了。MyBatis 的 Java API 就是这个甜头。稍后你将看到，和 JDBC 相比，MyBatis 大幅简化你的代码并力图保持其简洁、容易理解和维护。为了使得 SQL 映射更加优秀，MyBatis 3 引入了许多重要的改进。

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，你可以随意安排你的文件。但和其它框架一样，目录结构有一种最佳实践。

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 XML 配置中出现的属性值在这里。
  /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，但使用一个通用的目录结构将更有利于大家沟通。

本章接下来的示例将假定你遵循这种目录结构。

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，获取映射器示例和管理事务。在介绍 SqlSession 接口之前，我们先来了解如何获取一个 SqlSession 实例。SqlSessions 是由 SqlSessionFactory 实例创建的。SqlSessionFactory 对象包含创建 SqlSession 实例的各种方法。而 SqlSessionFactory 本身是由 SqlSessionFactoryBuilder 创建的，它可以从 XML、注解或 Java 配置代码来创建 SqlSessionFactory。

提示 当 Mybatis 与一些依赖注入框架（如 Spring 或者 Guice）搭配使用时，SqlSession 将被依赖注入框架创建并注入，所以你不需要使用 SqlSessionFactoryBuilder 或者 SqlSessionFactory，可以直接阅读 SqlSession 这一节。请参考 Mybatis-Spring 或者 Mybatis-Guice 手册以了解更多信息。

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，它接受一个指向 XML 文件（也就是之前讨论的 mybatis-config.xml 文件）的 InputStream 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>
```

如果你调用了带 environment 参数的 build 方法，那么 MyBatis 将使用该环境对应的配置。当然，如果你指定了一个无效的环境，会收到错误。如果你调用了不带 environment 参数的 build 方法，那么就会使用默认的环境配置（在上面的示例中，通过 default="development" 指定了默认环境）。

如果你调用了接受 properties 实例的方法，那么 MyBatis 就会加载这些属性，并在配置中提供使用。绝大多数场合下，可以用 ${propName} 形式引用这些配置值。

回想一下，在 mybatis-config.xml 中，可以引用属性值，也可以直接指定属性值。因此，理解属性的优先级是很重要的。在之前的文档中，我们已经介绍过了相关内容，但为了方便查阅，这里再重新介绍一下：

如果一个属性存在于下面的多个位置，那么 MyBatis 将按照以下顺序来加载它们：

- 首先，读取在 properties 元素体中指定的属性；
- 其次，读取在 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，但提供了不同的覆盖选项，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，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。在略读该类的源代码或用 IDE 查看该类信息后，你会发现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 方法接受一个 Configuration 实例。Configuration 类包含了对一个 SqlSessionFactory 实例你可能关心的所有内容。在检查配置时，Configuration 类很有用，它允许你查找和操纵 SQL 映射（但当应用开始接收请求时不推荐使用）。你之前学习过的所有配置开关都存在于 Configuration 类，只不过它们是以 Java API 形式暴露的。以下是一个简单的示例，演示如何手动配置 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，当你选择其中一个方法时，你需要考虑以下几点：

- **事务处理**：你希望在 session 作用域中使用事务作用域，还是使用自动提交（auto-commit）？（对很多数据库和/或 JDBC 驱动来说，等同于关闭事务支持）
- **数据库连接**：你希望 MyBatis 帮你从已配置的数据源获取连接，还是使用自己提供的连接？
- **语句执行**：你希望 MyBatis 复用 PreparedStatement 和/或批量更新语句（包括插入语句和删除语句）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，它会创建具备如下特性的 SqlSession：

- 事务作用域将会开启（也就是不自动提交）。
- 将由当前环境配置的 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。

相信你已经能从方法签名中知道这些方法的区别。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，我们没有提供同时设置 Connection 和 autoCommit 的方法，这是因为 MyBatis 会依据传入的 Connection 来决定是否启用 autoCommit。对于事务隔离级别，MyBatis 使用了一个 Java 枚举包装器来表示，称为 TransactionIsolationLevel，事务隔离级别支持 JDBC 的五个隔离级别（NONE、READ_UNCOMMITTED、READ_COMMITTED、REPEATABLE_READ 和 SERIALIZABLE），并且与预期的行为一致。

你可能对 ExecutorType 参数感到陌生。这个枚举类型定义了三个值:

- ExecutorType.SIMPLE： 该类型的执行器没有特别的行为。它为每个语句的执行创建一个新的预处理语句。
- ExecutorType.REUSE： 该类型的执行器会复用预处理语句。
- ExecutorType.BATCH： 该类型的执行器会批量执行所有更新语句，如果 SELECT 在多个更新中间执行，将在必要时将多条更新语句分隔开来，以方便理解。

提示 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，你可以在运行时使用它来检查 MyBatis 的配置。

提示 如果你使用过 MyBatis 的旧版本，可能还记得 session、事务和批量操作是相互独立的。在新版本中则不是这样。上述三者都包含在 session 作用域内。你不必分别处理事务或批量操作就能得到想要的全部效果。

SqlSession

正如之前所提到的，SqlSession 在 MyBatis 中是非常强大的一个类。它包含了所有执行语句、提交或回滚事务以及获取映射器实例的方法。

SqlSession 类的方法超过了 20 个，为了方便理解，我们将它们分成几种组别。

语句执行方法

这些方法被用来执行定义在 SQL 映射 XML 文件中的 SELECT、INSERT、UPDATE 和 DELETE 语句。你可以通过名字快速了解它们的作用，每一方法都接受语句的 ID 以及参数对象，参数可以是原始类型（支持自动装箱或包装类）、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，就会抛出异常。如果你不知道返回对象会有多少，请使用 selectList。如果需要查看某个对象是否存在，最好的办法是查询一个 count 值（0 或 1）。selectMap 稍微特殊一点，它会将返回对象的其中一个属性作为 key 值，将对

象作为 value 值，从而将多个结果集转为 Map 类型值。由于并不是所有语句都需要参数，所以这些方法都具有一个不需要参数的重载形式。

游标（Cursor）与列表（List）返回的结果相同，不同的是，游标借助迭代器实现了数据的惰性加载。

```
try (Cursor<MyEntity> entities = session.selectCursor(statement, param)) {
   for (MyEntity entity:entities) {
      // 处理单个实体
   }
}
```

insert、update 以及 delete 方法返回的值表示受该语句影响的行数。

```
<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)
```

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，或是提供自定义结果处理逻辑，通常在数据集非常庞大的情形下使用。

```
<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 rowbound
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，并限制返回结果的数量。RowBounds 类的 offset 和 limit 值只有在构造函数时才能传入，其它时候是不能修改的。

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

数据库驱动决定了略过记录时的查询效率。为了获得最佳的性能，建议将 ResultSet 类型设置为 SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE（换句话说：不要使用 FORWARD_ONLY）。

ResultHandler 参数允许自定义每行结果的处理过程。你可以将它添加到 List 中、创建 Map 和 Set，甚至丢弃每个返回值，只保留计算后的统计结果。你可以使用 ResultHandler 做很多事，这其实就是 MyBatis 构建结果列表的内部实现办法。

从版本 3.4.6 开始， `ResultHandler` 会在存储过程的 REFCURSOR 输出参数中传递使用的 `CALLABLE` 语句。

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和当前已被创建的对象数目，另外还提供了一个返回值为 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个限制：

- 使用带 ResultHandler 参数的方法时，收到的数据不会被缓存。
- 当使用高级的结果映射集（resultMap）时，MyBatis 很可能需要数行结果来构造一个对象。如果你使用了 ResultHandler，你可能会接收到关联（association）或者集合（collection）中尚未被完整填充的对象。

立即批量更新方法

当你将 `ExecutorType` 设置为 `ExecutorType.BATCH` 时，可以使用这个方法清除（执行）缓存在 JDBC 驱动类中的批量更新语句。

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

事务控制方法

有四个方法用来控制事务作用域。当然，如果你已经设置了自动提交或你使用了外部事务管理器，这些方法就没什么作用了。然而，如果你正在使用由 Connection 实例控制的 JDBC 事务管理器，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，除非它侦测到调用了插入、更新或删除方法改变了数据库。如果你没有使用这些方法提交修改，那么你可以在 commit 和 rollback 方法参数中传入 true 值，来保证事务被正常提交（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）。大部分情况下你无需调用 rollback()，因为 MyBatis 会在你没有调用 commit 时替你完成回滚操作。不过，当你要在一个可能多次提交或回滚的 session 中详细控制事务，回滚操作就派上用场了。

提示 MyBatis-Spring 和 MyBatis-Guice 提供了声明式事务处理，所以如果你在使用 Mybatis 的同时使用了 Spring 或者 Guice，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 执行过的查询结果都会被保存在本地缓存中，所以，当再次执行参数相同的相同查询时，就不需要实际查询数据库了。本地缓存将会在做出修改、事务提交或回滚，以及关闭 session 时清空。

默认情况下，本地缓存数据的生命周期等同于整个 session 的周期。由于缓存会被用来解决循环引用问题和加快重复嵌套查询的速度，所以无法将其完全禁用。但是你可以通过设置 localCacheScope=STATEMENT 来只在语句执行时使用缓存。

注意，如果 localCacheScope 被设置为 SESSION，对于某个对象，MyBatis 将返回在本地缓存中唯一对象的引用。对返回的对象（例如 list）做出的任何修改将会影响本地缓存的内容，进而将会影响到在本次 session 中从缓存返回的值。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()
```

对于你打开的任何 session，你都要保证它们被妥善关闭，这很重要。保证妥善关闭的最佳代码模式是这样的：

```
SqlSession session = sqlSessionFactory.openSession();
try (SqlSession session = sqlSessionFactory.openSession()) {
    // 假设下面三行代码是你的业务逻辑
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
}
```

提示 和 SqlSessionFactory 一样，你可以调用当前使用的 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，它们并不符合类型安全，对你的 IDE 和单元测试也不是那么友好。因此，使用映射器类来执行映射语句是更常见的做法。

我们已经在之前的入门章节中见到过一个使用映射器的示例。一个映射器类就是一个仅需声明与 SqlSession 方法相匹配方法的接口。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，字符串参数 ID 无需匹配。而是由方法名匹配映射语句的 ID。

此外，返回类型必须匹配期望的结果类型，返回单个值时，返回类型应该是返回值的类，返回多个值时，则为数组或集合类，另外也可以是游标（Cursor）。所有常用的类型都是支持的，包括：原始类型、Map、POJO 和 JavaBean。

提示 映射器接口不需要去实现任何接口或继承自任何类。只要方法签名可以被用来唯一识别对应的映射语句就可以了。

提示 映射器接口可以继承自其他接口。在使用 XML 来绑定映射器接口时，保证语句处于合适的命名空间中即可。唯一的限制是，不能在两个具有继承关系的接口中拥有相同的方法签名（这是潜在的危险做法，不可取）。

你可以传递多个参数给一个映射器方法。在多个参数的情况下，默认它们将会以 param 加上它们在参数列表中的位置来命名，比如：#{param1}、#{param2}等。如果你想（在有多个参数时）自定义参数的名称，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

设计初期的 MyBatis 是一个 XML 驱动的框架。配置信息是基于 XML 的，映射语句也是定义在 XML 中的。而在 MyBatis 3 中，我们提供了其它的配置方式。MyBatis 3 构建在全面且强大的基于 Java 语言的配置 API 之上。它是 XML 和注解配置的基础。注解提供了一种简单且低成本的方式来实现简单的映射语句。

提示 不幸的是，Java 注解的表达能力和灵活性十分有限。尽管我们花了很多时间在调查、设计和试验上，但最强大的 MyBatis 映射并不能用注解来构建——我们真没开玩笑。而 C# 属性就没有这些限制，因此 MyBatis.NET 的配置会比 XML 有更大的选择余地。虽说如此，基于 Java 注解的配置还是有它的好处的。

注解如下表所示：

注解	使用对象	XML 等价形式	描述
@CacheNamespace	类	<cache>	为给定的命名空间（比如类）配置缓存。属性：implemetation、eviction、flushInterval、size、readWrite、blocking、properties。
@Property	N/A	<property>	指定参数值或占位符（placeholder）（该占位符能被 mybatis-config.xml 内的配置属性替换）。属性：name、value。（仅在 MyBatis 3.4.2 以上可用）
@CacheNamespaceRef	类	<cacheRef>	引用另外一个命名空间的缓存以供使用。注意，即使共享相同的全限定类名，在 XML 映射文件中声明的缓存仍被识别为一个独立的命名空间。属性：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 属性用于指定能够表示该命名空间的 Java 类型（命名空间名就是该 Java 类型的全限定类名），name 属性（这个属性仅在 MyBatis 3.4.2 以上可用）则直接指定了命名空间的名字。
@ConstructorArgs	方法	<constructor>	收集一组结果以传递给一个结果对象的构造方法。属性：value，它是一个 Arg 数组。
@Arg	N/A	• <arg> • <idArg>	ConstructorArgs 集合的一部分，代表一个构造方法参数。属性：id、column、javaType、jdbcType、typeHandler、select、resultMap。id 属性和 XML 元素 <idArg> 相似，它是一个布尔值，表示该属性是否用于唯一标识和比较对象。从版本 3.5.4 开始，该注解变为可重复注解。

注解	使用对象	XML 等价形式	描述
@TypeDiscriminator	方法	<discriminator>	决定使用何种结果映射的一组取值（case）。属性：column、javaType、jdbcType、typeHandler、cases。cases 属性是一个 Case 的数组。
@Case	N/A	<case>	表示某个值的一个取值以及该取值对应的映射。属性：value、type、results。results 属性是一个 Results 的数组，因此这个注解实际上和 ResultMap 很相似，由下面的 Results 注解指定。
@Results	方法	<resultMap>	一组结果映射，指定了对某个特定结果列，映射到某个属性或字段的方式。属性：value、id。value 属性是一个 Result 注解的数组。而 id 属性则是结果映射的名称。从版本 3.5.4 开始，该注解变为可重复注解。
@Result	N/A	• <result> • <id>	在列和属性或字段之间的单个结果映射。属性：id、column、javaType、jdbcType、typeHandler、one、many。id 属性和 XML 元素 <id> 相似，它是一个布尔值，表示该属性是否用于唯一标识和比较对象。one 属性是一个关联，和 <association> 类似，而 many 属性则是集合关联，和 <collection> 类似。这样命名是为了避免产生名称冲突。
@One	N/A	<association>	复杂类型的单个属性映射。属性：select，指定可加载合适类型实例的映射语句（也就是映射器方法）全限定名；fetchType，指定在该映射中覆盖全局配置参数 lazyLoadingEnabled。**提示** 注解 API 不支持联合映射。这是由于 Java 注解不允许产生循环引用。
@Many	N/A	<collection>	复杂类型的集合属性映射。属性：select，指定可加载合适类型实例集合的映射语句（也就是映射器方法）全限定名；fetchType，指定在该映射中覆盖全局配置参数 lazyLoadingEnabled。**提示** 注解 API 不支持联合映射。这是由于 Java 注解不允许产生循环引用。

注解	使用对象	XML 等价形式	描述
@MapKey	方法		供返回值为 Map 的方法使用的注解。它使用对象的某个属性作为 key，将对象 List 转化为 Map。属性：value，指定作为 Map 的 key 值的对象属性名。
@Options	方法	映射语句的属性	该注解允许你指定大部分开关和配置选项，它们通常在映射语句上作为属性出现。与在注解上提供大量的属性相比，Options 注解提供了一致、清晰的方式来指定选项。属性：useCache=true、flushCache=FlushCachePolicy.DEFAULT、resultSetType=DEFAULT、statementType=PREPARED、fetchSize=-1、timeout=-1、useGeneratedKeys=false、keyProperty=""、keyColumn=""、resultSets=""。注意，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非预期行为。 注意：keyColumn 属性只在某些数据库中有效（如 Oracle、PostgreSQL 等）。要了解更多关于 keyColumn 和 keyProperty 可选值信息，请查看“insert, update 和 delete”一节。
• @Insert • @Update • @Delete • @Select	方法	• <insert> • <update> • <delete> • <select>	每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，字符串数组会被连接成单个完整的字符串，每个字符串之间加入一个空格。这有效地避免了用 Java 代码构建 SQL 语句时产生的“丢失空格”问题。当然，你也可以提前手动连接好字符串。属性：value，指定用来组成单个 SQL 语句的字符串数组。

注解	使用对象	XML 等价形式	描述
• @InsertProvider • @UpdateProvider • @DeleteProvider • @SelectProvider	方法	• <insert> • <update> • <delete> • <select>	允许构建动态 SQL。这些备选的 SQL 注解允许你指定返回 SQL 语句的类和方法，以供运行时执行。（从 MyBatis 3.4.6 开始，可以使用 CharSequence 代替 String 来作为返回类型）。当执行映射语句时，MyBatis 会实例化注解指定的类，并调用注解指定的方法。你可以通过 ProviderContext 传递映射方法接收到的参数、"Mapper interface type" 和 "Mapper method"（仅在 MyBatis 3.4.5 以上支持）作为参数。（MyBatis 3.4 以上支持传入多个参数）属性：type、method。type 属性用于指定类名。method 用于指定该类的方法名（从版本 3.5.1 开始，可以省略 method 属性，MyBatis 将会使用 ProviderMethodResolver 接口解析方法的具体实现。如果解析失败，MyBatis 将会使用名为 provideSql 的降级实现）。**提示** 接下来的“SQL 语句构建器”一章将会讨论该话题，以帮助你以更清晰、更便于阅读的方式构建动态 SQL。
@Param	参数	N/A	如果你的映射方法接受多个参数，就可以使用这个注解自定义每个参数的名字。否则在默认情况下，除 RowBounds 以外的参数会以 "param" 加参数位置被命名。例如 #{param1}，#{param2}。如果使用了 @Param("person")，参数就会被命名为 #{person}。

注解	使用对象	XML 等价形式	描述
@SelectKey	方法	<selectKey>	这个注解的功能与 <selectKey> 标签完全一致。该注解只能在 @Insert 或 @InsertProvider 或 @Update 或 @UpdateProvider 标注的方法上使用，否则将会被忽略。如果标注了 @SelectKey 注解，MyBatis 将会忽略掉由 @Options 注解所设置的生成主键或设置（configuration）属性。属性：statement 以字符串数组形式指定将会被执行的 SQL 语句，keyProperty 指定作为参数传入的对象对应属性的名称，该属性将会更新成新的值，before 可以指定为 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 则指定 keyProperty 的 Java 类型。statementType 则用于选择语句类型，可以选择 STATEMENT、PREPARED 或 CALLABLE 之一，它们分别对应于 Statement、PreparedStatement 和 CallableStatement。默认值是 PREPARED。
@ResultMap	方法	N/A	这个注解为 @Select 或者 @SelectProvider 注解指定 XML 映射中 <resultMap> 元素的 id。这使得注解的 select 可以复用已在 XML 中定义的 ResultMap。如果标注的 select 注解中存在 @Results 或者 @ConstructorArgs 注解，这两个注解将被此注解覆盖。

注解	使用对象	XML 等价形式	描述
@ResultType	方法	N/A	在使用了结果处理器的情况下，需要使用此注解。由于此时的返回类型为 void，所以 Mybatis 需要有一种方法来判断每一行返回的对象类型。如果在 XML 有对应的结果映射，请使用 @ResultMap 注解。如果结果类型在 XML 的 <select> 元素中指定了，就不需要使用其它注解了。否则就需要使用此注解。比如，如果一个标注了 @Select 的方法想要使用结果处理器，那么它的返回类型必须是 void，并且必须使用这个注解（或者 @ResultMap）。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 接口中的方法就能够调用 SqlSession#flushStatements() 方法。（Mybatis 3.3 以上可用）

映射注解示例

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, re
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自增列的值：

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

这个例子展示了如何使用 @Flush 注解来调用 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);
```

这个例子展示了如何使用单个参数的 @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();
  }
}
```

这个例子展示了如何使用多个参数的 @SqlProvider 注解：

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

class UserSqlBuilder {

  // 如果不使用 @Param，就应该定义与 mapper 方法相同的参数
  public static String buildGetUsersByName(
      final String name, final String orderByColumn) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      WHERE("name like #{name} || '%'");
      ORDER_BY(orderByColumn);
    }}.toString();
  }

  // 如果使用 @Param，就可以只定义需要使用的参数
  public static String buildGetUsersByName(@Param("orderByColumn") final String orderByColumn)
    return new SQL(){{
      SELECT("*");
      FROM("users");
      WHERE("name like #{name} || '%'");
      ORDER_BY(orderByColumn);
    }}.toString();
  }
}
```

以下例子展示了 ProviderMethodResolver （3.5.1 后可用）的默认实现使用方法：

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

// 在你的 provider 类中实现 ProviderMethodResolver 接口
class UserSqlProvider implements ProviderMethodResolver {
  // 默认实现中，会将映射器方法的调用解析到实现的同名方法上
  public static String getUsersByName(final String name) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      if (name != null) {
        WHERE("name like #{value} || '%'");
      }
      ORDER_BY("id");
    }}.toString();
  }
}
```