**一、MyBatis样例**

**1、MyBatis配置文件：mybatis-config.xml**

其中包括数据库连接信息、类型别名等等.

<? xml version=*"1.0"* encoding=*"utf-8"* ?>

<!DOCTYPE configuration PUBLIC

"-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<typeAliases> <!-- 类型别名 -->

<typeAlias type=*"com.mybatis3.domain.Student"* alias=*"Student"* />

</typeAliases>

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"* /> <!-- 事务管理器 -->

<dataSource type=*"POOLED"*> <!-- 数据源 -->

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/why"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"why1993"*/>

</dataSource>

</environment>

</environments>

<mappers resource=*"com/mybatis3/mappers/StudentMapper.xml"* />

</configuration>

**2、映射文件StudentsMapper.xml**

创建一个包含了映射的SQL语句的StudentMapper.xml文件（映射文件）。

<? xml version=*"1.0"* encoding=*"utf-8"* ?>

<!DOCTYPE mapper PUBLIC

"-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- SQL映射器映射文件 -->

<mapper namespace=*"com.mybatis3.mappers.StudentMapper"*>

<resultMap type=*"Student"* id=*"StudentResult"*>

<id property=*"stuId"* column=*"stud\_id"* />

<result property=*"name"* column=*"name"* />

<result property=*"email"* column=*"email"* />

<result property=*"dob"* column=*"dob"* />

</resultMap>

<select id=*"findAllStudents"* resultMap=*"StudentResult"*>

select \* from students

</select>

<select id=*"findStudentById"* parameterType=*"int"* resultType=*"Student"*>

select stud\_id as stuid, name, email, dob from students where stud\_id=#{id}

</select>

<insert id=*"insertStudent"* parameterType=*"Student"*>

insert into students(name.email,dob) values(#{name},#{email},#{dob})

</insert>

</mapper>

**3、辅助类MyBatisSqlSessionFactory**

创建了一个SqlSessionFactory对象，将使用它来获得SqlSession对象和执行映射的SQL语句。

**package** com.mybatis.util;

**import** java.io.\*;

**import** org.apache.ibatis.io.Resources;

**import** org.apache.ibatis.session.\*;

**public** **class** MyBatisSqlSessionFactory {

**private** **static** SqlSessionFactory *sqlSessionFactory*;

**public** **static** SqlSessionFactory getSqlSessionFactory(){

**if**(*sqlSessionFactory* == **null**){

InputStream inputStream;

**try**{

inputStream = Resources.*getResourceAsStream*("mybatis-config.xml");

*sqlSessionFactory* = **new** SqlSessionFactoryBuilder().build(inputStream);

}**catch**(IOException e){

**throw** **new** RuntimeException(e.getCause());

}

}

**return** *sqlSessionFactory*;

}

**public** **static** SqlSession openSession(){ //获得SqlSession对象

**return** *getSqlSessionFactory*().openSession();

}

}

**4、新建StudentMapper 接口和StudentService 类**

创建一个StudentMapper接口，其定义的方法名和在Mapper XML配置文件定义的SQL映射语句名称相同；再创建一个StudentService.java 类，包含了一些业务操作的实现。

**1）首先, 创建JavaBean Student.java**

**package** com.mybatis.domain;

**import** java.util.Date;

//JavaBean

**public** **class** Student {

**private** Integer stuid;

**private** String name;

**private** String email;

**private** Date dob;

// setters and getters

}

2）创建映射器Mapper接口StudentMapper.java 其方法签名和StudentMapper.xml 中定义的SQL映射定义名相同（即映射器Mapper接口中的方法名跟StudentMapper.xml中的id的值相同）

**package** com.mybatis3.mappers;

**import** java.util.List;

**import** com.mybatis.domain.Student;

**public** **interface** StudentMapper {

List<Student> findAllStudent();

Student findStudentById(Integer id);

**void** insertStudent(Student student);

}

注意：StudentMapper.xml 中namespace的值被设置成com.mybatis3.mappers.StudentMapper，是StudentMapper接口的完全限定名。这使我们可以使用接口来调用映射的SQL语句。

**二、引导MyBatis**

**1、使用XML配置**MyBatis

配置文件中的各个组成部分：

**1）**environment

MyBatis支持配置多个dataSource 环境，可以将应用部署到不同的环境上，如DEV(开发环境)、TEST（测试换将）、QA（质量评估环境）、UAT(用户验收环境)、PRODUCTION（生产环境），可以通过将默认environment值设置成想要的environment id 值。

在上述的配置中，默认的环境environment 被设置成development。当需要将程序部署到生产服务器上时，你不需要修改什么配置，只需要将默认环境environment值设置成生产环境的environment id 属性即可。

有时候，我们可能需要在相同的应用下使用多个数据库。比如我们可能有SHOPPING-CART数据库来存储所有的订单明细；使用REPORTS数据库存储订单明细的合计，用作报告。如果你的应用需要连接多个数据库，你需要将每个数据库配置成独立的环境，并且为每一个数据库创建一个SqlSessionFactory。

可以如下为每个环境创建一个SqlSessionFactory:

inputStream = Resources.getResourceAsStream("mybatis-config.xml");

defaultSqlSessionFactory = new SqlSessionFactoryBuilder(). build(inputStream); cartSqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStre am, "shoppingcart");

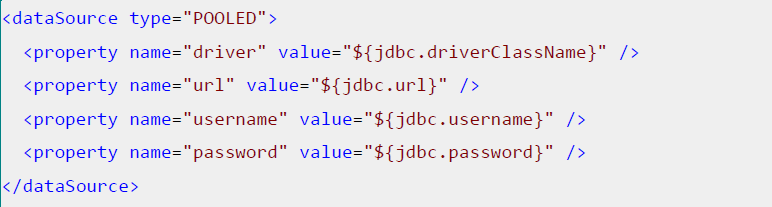
reportSqlSessionFactory = new SqlSessionFactoryBuilder(). build(inputStream, "reports");

创建SqlSessionFactory时，如果没有明确指定环境environment id，则会使用默认的环境environment来创建。在上述的源码中，默认的SqlSessionFactory便是使用shoppingcart 环境设置创建的。对于每个环境environment,我们需要配置dataSource 和transactionManager 元素。



注意：**你可以配置多种环境,但你只能为每个 SqlSessionFactory 实例选择一个。**

**2）数据源**dataSource**：被用来配置数据库连接属性**



dataSource的类型可以配置成其内置类型之一，如UNPOOLED、POOLED、JNDI

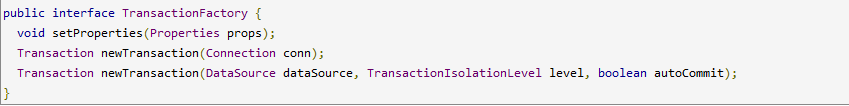
* 如果将类型设置成UNPOOLED，MyBatis会为每一个数据库操作创建一个新的连接，并关闭它。该方式适用于只有小规模数量并发用户的简单应用程序上。
* 如果将属性设置成POOLED，MyBatis会创建一个数据库连接池，连接池中的一个连接将会被用作数据库操作。一旦数据库操作完成，MyBatis会将此连接返回给连接池。在开发或测试环境中，经常使用此种方式。
* 如果将类型设置成JNDI，MyBatis从在应用服务器向配置好的JNDI数据源dataSource获取数据库连接。在生产环境中，优先考虑这种方式。

**3）事务管理器TransactionManager**

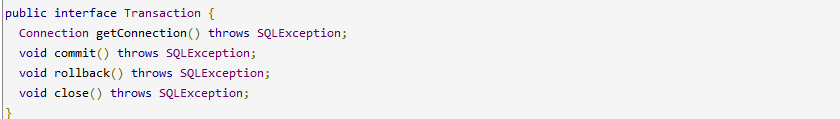
MyBatis支持两种类型的事务管理器：JDBC and MANAGED.

* JDBC事务管理器被用作当应用程序负责管理数据库连接的生命周期（提交、回退等等）的时候。当你将TransactionManager属性设置成JDBC，MyBatis内部将使用JdbcTransactionFactory类创建TransactionManager。例如，部署到Apache Tomcat的应用程序，需要应用程序自己管理事务（即简单使用JDBC的提交和回滚设置，依赖于从数据源得到的连接管理事务）。
* MANAGED 事务管理器是当由应用服务器负责管理数据库连接生命周期的时候使用。当你将TransactionManager属性设置成MANAGED时，MyBatis内部使用ManagedTransactionFactory 类创建事务管理器TransactionManager。例如，当一个JavaEE的应用程序部署在类似JBoss，WebLogic，GlassFish应用服务器上时，它们会使用EJB进行应用服务器的事务管理能力。在这些管理环境中，你可以使用MANAGED事务管理器。（注：Managed是托管的意思，即是应用本身不去管理事务，而是把事务管理交给应用所在的服务器进行管理。）

这两种事务管理器都不需要任何属性。然而它们都是类型别名,要替换使用它们,你需 要放置将你自己的类的完全限定名或类型别名,它们引用了你对 TransactionFactory 接口的实现类。



任何在XML中配置的属性在实例化之后将会被传递给setProperties()方法。你的实现类需要创建一个事务接口的实现,这个接口也很简单:

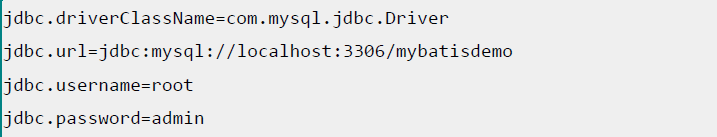


使用这两个接口,你可以完全自定义 MyBatis 对事务的处理。

**4）属性properties（文件值优于默认值）**

属性配置元素可以将配置值具体化到一个属性文件中，并且使用属性文件的key名作为占位符。在上述的配置中，我们将数据库连接属性具体化到了application.properties文件中，并且为driver，URL等属性使用了占位符。

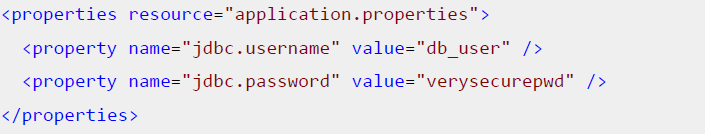
（1）在applications.properties文件中配置数据库连接参数，如下所示：



（2）在mybatis-config.xml文件中，为属性使用application.properties 文件中定义的占位符：



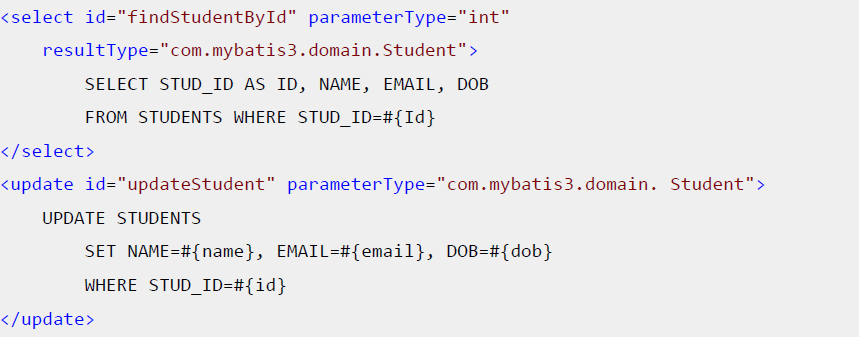
并且，你可以在<properties>元素中配置默认参数的值，如果<properties>中定义的元素和属性文件定义元素的key值相同，它们会被属性文件中定义的值覆盖。



这里，如果application.properties文件包含值jdbc.username和jdbc.password，则上述定义的username和password的值db\_user 和verysecurepwd将会被application.properties中定义的对应的jdbc.username和jdbc.password值覆盖。

**5）类型别名typeAliases**

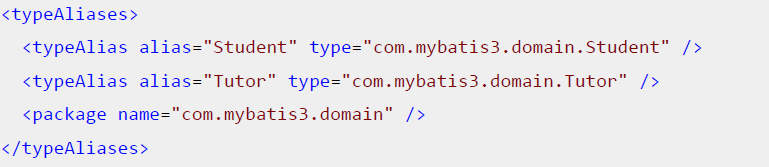
类型别名是为 Java 类型命名一个短的名字。它只和 XML 配置有关，只用来减少类完全限定名的多余部分。在SQLMapper配置文件中，对于resultType和parameterType属性值，我们需要使用JavaBean的完全限定名。如下：



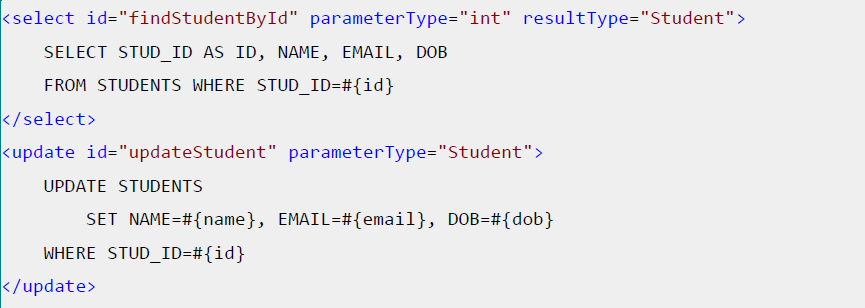
这里我们为resultType 和parameterType属性值设置为Student类型的完全限定名：

**com.mybatis3.domain.Student**

我们可以为完全限定名取一个别名（alias），然后其需要使用完全限定名的地方使用别名，而不是到处使用完全限定名。如下例子所示，为完全限定名起一个别名：



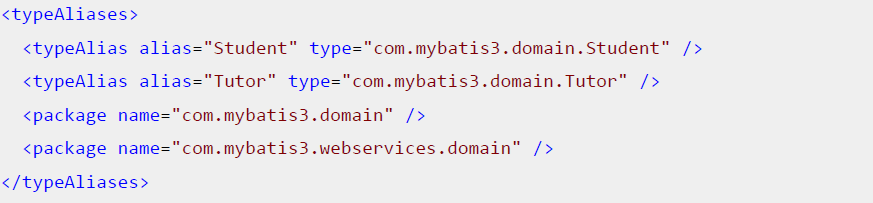
然后在SQL Mapper 映射文件中, 如下使用Student的别名：



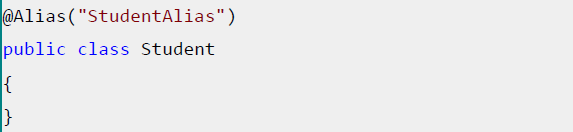
你可以不用为每一个JavaBean单独定义别名, 你可以为提供需要取别名的JavaBean所在的包(package)，MyBatis会自动扫描包内定义的JavaBeans，然后分别为JavaBean注册一个小写字母开头的非完全限定的类名形式的别名。如下所示，提供一个需要为JavaBeans起别名的包名：



如果Student.java 和Tutor.java Bean定义在com.mybatis3.domain包中，则com.mybatis3.domain.Student的别名会被注册为student。而com.mybatis3.domain.Tutor别名将会被注册为tutor。示例如下：



还有另外一种方式为JavaBeans起别名，使用注解@Alias:



@Alias注解将会覆盖配置文件中的<typeAliases>定义。

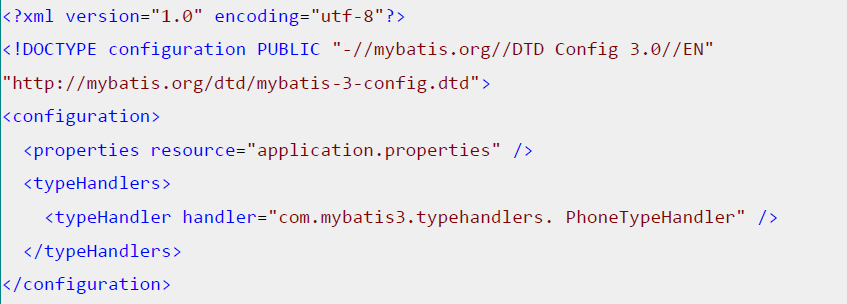
**6）类型处理器typeHandlers**

MyBatis对于以下的类型使用内建的类型处理器：所有的基本数据类型、基本类型的包裹类型、byte[]、java.util.Date、java.sql.Date、java,sql.Time、java.sql.Timestamp、java枚举类型等。所以当MyBatis发现属性的类型属于上述类型，他会使用对应的类型处理器将值设置到PreparedStatement中，同样地，当从SQL结果集构建JavaBean时，也有类似的过程。即无论是 MyBatis 在预处理语句中设置一个参数, 还是从结果集中取出一个值时, 类型处理器被用来将获取的值以合适的方式转换成 Java 类型。

那如果我们给了一个自定义的对象类型，来存储存储到数据库呢？PhoneNumber，我们可以创建一个自定义的类型处理器：

（1）MyBatis 提供了抽象类BaseTypeHandler<T> ，我们可以继承此类创建自定义类型处理器（其中，T为自定义数据类型）

（2）一旦我们实现了自定义的类型处理器，我们需要在mybatis-config.xml中注册它：

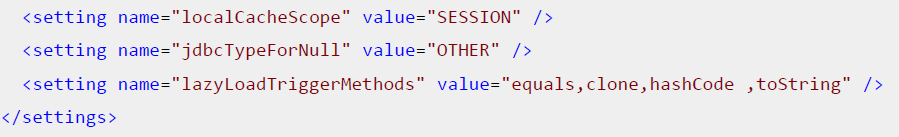


注册PhoneTypeHandler后, MyBatis 就能够将Phone类型的对象值存储到VARCHAR类型的列上。

**7）全局参数设置setting**

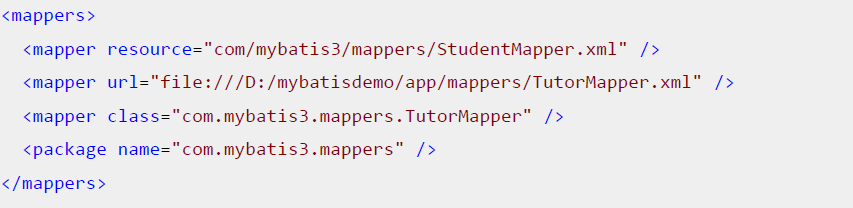
为满足应用特定的需求，MyBatis默认的全局参数设置可以被覆盖(overridden)掉，如下所示：（默认设置）





**8）SQL映射定义Mappers**

Mapper XML文件中包含的SQL映射语句将会被应用通过使用其statement id来执行。我们需要在mybatis-config.xml 文件中配置SQL Mapper文件的位置。



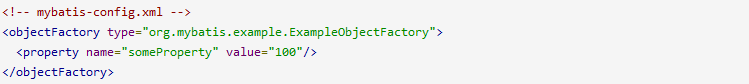
以上每一个<mapper> 标签的属性有助于从不同类型的资源中加载映射mapper：

* resource属性用来指定在classpath中的mapper文件。
* url属性用来通过完全文件系统路径或者web URL地址来指向mapper文件
* class属性用来指向一个mapper接口
* package属性用来指向可以找到Mapper接口的包名

**9）objectFactory**

MyBatis 每次创建结果对象新的实例时, 它使用一个 ObjectFactory 实例来完成。 如果参数映射存在，默认的 ObjectFactory 不比使用默认构造方法或带参数的构造方法实例化目标类做的工作多。如果你想重写默认的 ObjectFactory,你可以创建你自己的。比如:





ObjectFactory 接口很简单。它包含两个创建用的方法,一个是处理默认构造方法的,另外一个是处理带参数构造方法的。最终，setProperties方法可以被用来配置ObjectFactory。 在初始化你的ObjectFactory实例后，objectFactory元素体中定义的属性会被传递给setProperties方法。

**2、使用Java API配置MyBatis**

**1）创建SqlSessionFactory**

**public** **static** SqlSessionFactory getSqlSessionFactory() {

SqlSessionFactory sqlSessionFactory = **null**;

**try** {

DataSource dataSource = DataSourceFactory.getDataSource(); //获取数据源

TransactionFactory transactionFactory = **new** JdbcTransactionFactory(); //创建事务工厂对象

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

Configuration configuration = **new** Configuration(environment); //创建一个配置对象

configuration.getTypeAliasRegistry().registerAlias("student", Student.**class**); //设置类型别名

configuration.getTypeHandlerRegistry().register(PhoneNumber. **class**, PhoneTypeHandler.**class**);

configuration.addMapper(StudentMapper.**class**);

sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(configuration);

} **catch** (Exception e) {

**throw** **new** RuntimeException(e);

}

**return** sqlSessionFactory;

}

**2）环境配置Environment**

我们需要为想使用MaBatis连接的每一个数据库创建一个Enviroment对象。为了使用每一个环境，我们需要为每一个环境environment创建一个SqlSessionFactory对象。而创建Environment对象，我们需要java.sql.DataSource和TransactionFactory实例。下面让我们看看如何创建DataSource 和TransactionFactory对象。

**3）数据源DataSource**

MyBatis支持三种内建的DataSource类型: UNPOOLED、POOLED 和JNDI.

* UNPOOLED类型的数据源dataSource为每一个用户请求创建一个数据库连接。在多用户并发应用中，不建议使用。
* POOLED类型的数据源dataSource创建了一个数据库连接池，对用户的每一个请求，会使用缓冲池中的一个可用的Connection对象，这样可以提高应用的性能。MyBatis提供了org.apache.ibatis.datasource.pooled.PooledDataSource 实现javax.sql.DataSource来创建连接池。
* JNDI类型的数据源dataSource使用了应用服务器的数据库连接池，并且使用JNDI查找来获取数据库连接。

让我们看一下怎样通过MyBatis的PooledDataSource获得DataSource对象，如下：

**public** **class** DataSourceFactory {

**public** **static** DataSource getDataSource() {

String driver = "com.mysql.jdbc.Driver";

String url = "jdbc:mysql://localhost:3306/mybatisdemo";

String username = "root"; String password = "admin";

PooledDataSource dataSource = **new** PooledDataSource(driver, url, username, password);

**return** dataSource;

} }

一般在生产环境中，DataSource会被应用服务器配置，并通过JNDI获取DataSource对象，如下所示：

**public** **class** DataSourceFactory {

**public** **static** DataSource getDataSource() {

String jndiName = "java:comp/env/jdbc/MyBatisDemoDS";

**try** {

InitialContext ctx = **new** InitialContext();

DataSource dataSource = (DataSource) ctx.lookup(jndiName);

**return** dataSource;

} **catch** (NamingException e) {

**throw** **new** RuntimeException(e);

} } }

当前有一些流行的第三方类库，如commons-dbcp和c3p0实现了java.sql.DataSource,你可以使用它们来创建dataSource。

**4）事务工厂TransactionFactory**

MyBatis支持一下两种TransactionFactory实现：

* JdbcTransactionFactory
* ManagedTransactionFactory

如果你的应用程序运行在未托管（non-managed）的环境中，你应该使用JdbcTransactionFactory：

DataSource *dataSource* = DataSourceFactory.getDataSource();

TransactionFactory *txnFactory* = **new** JdbcTransactionFactory();

Environment environment = **new** Environment("development", txnFactory, dataSource);

如果你的应用程序运行在未托管（non-managed）的环境中,并且使用容器支持的事务管理服务，你应该使用ManagedTransactionFactory：

DataSource *dataSource* = DataSourceFactory.getDataSource();

TransactionFactory *txnFactory* = **new** ManagedTransactionFactory();

Environment environment = **new** Environment("development", txnFactory, dataSource);

**5）类型别名typeAliases**

MyBatis提供以下几种通过Configuration对象注册类型别名的方法：

1. 根据默认的别名规则，使用一个类的首字母小写、非完全限定的类名作为别名注册，可使用以下代码：configuration.getTypeAliasRegistry().registerAlias(Student.**class**);
2. 指定别名注册，可使用以下代码：configuration.getTypeAliasRegistry().registerAlias("Student",Student.**class**);
3. 通过类的完全限定名注册相应类别名，可使用一下代码：configuration.getTypeAliasRegistry().registerAlias("Student", "com.mybatis3.domain.Student");
4. 为某一个包中的所有类注册别名，可使用以下代码：configuration.getTypeAliasRegistry().registerAliases("com. mybatis3.domain");
5. 为在com.mybatis3.domainpackage包中所有的继承自Identifiable类型的类注册别名，可使用以下代码：

configuration.getTypeAliasRegistry().registerAliases("com. mybatis3.domain", Identifiable.**class**);

**6）类型处理器typeHandlers**

MyBatis提供了一系列使用Configuration对象注册类型处理器（type handler）的方法。我们可以通过以下方式注册自定义的类处理器：

1. 为某个特定的类注册类处理器:

configuration.getTypeHandlerRegistry().register(PhoneNumber. **class**, PhoneTypeHandler.**class**);

1. 注册一个类处理器:

configuration.getTypeHandlerRegistry().register(PhoneTypeHandler. **class**);

1. 注册com.mybatis3.typehandlers包中的所有类型处理器：

configuration.getTypeHandlerRegistry().register("com.mybatis3.typehandlers");

**7）全局参数设置Settings**

MyBatis提供了一组默认的，能够很好地适用大部分的应用的全局参数设置。然而，你可以稍微调整这些参数，让它更好地满足你应用的需要。你可以使用下列方法将全局参数设置成想要的值。

configuration.setCacheEnabled(**true**);

configuration.setLazyLoadingEnabled(**false**);

configuration.setMultipleResultSetsEnabled(**true**);

configuration.setUseColumnLabel(**true**);

configuration.setUseGeneratedKeys(**false**);

configuration.setAutoMappingBehavior(AutoMappingBehavior.PARTIAL);

configuration.setDefaultExecutorType(ExecutorType.SIMPLE);

configuration.setDefaultStatementTimeout(25);

configuration.setSafeRowBoundsEnabled(**false**);

configuration.setMapUnderscoreToCamelCase(**false**);

configuration.setLocalCacheScope(LocalCacheScope.SESSION);

configuration.setAggressiveLazyLoading(**true**);

configuration.setJdbcTypeForNull(JdbcType.OTHER);

Set<String> lazyLoadTriggerMethods = **new** HashSet<String>();

lazyLoadTriggerMethods.add("equals");

lazyLoadTriggerMethods.add("clone");

lazyLoadTriggerMethods.add("hashCode");

lazyLoadTriggerMethods.add("toString");

configuration.setLazyLoadTriggerMethods(lazyLoadTriggerMethods );

**8）Mappers**

MyBatis提供了一些使用Configuration对象注册Mapper XML文件和Mappe接口的方法。

1. 添加一个Mapper接口，可使用以下代码:

configuration.addMapper(StudentMapper.**class**);

1. 添加com.mybatis3.mappers包中的所有Mapper XML文件或者Mapper接口，可使用以下代码：

configuration.addMappers("com.mybatis3.mappers");

1. 添加所有com.mybatis3.mappers 包中的拓展了特定Mapper接口的Maper接口，如aseMapper,可使用如下代码：

configuration.addMappers("com.mybatis3.mappers", BaseMapper.**class**);

**3、自定义MyBatis日志**

MyBatis使用其内部LoggerFactory作为真正的日志类库使用的门面。其内部的LaggerFactory会将日志记录任务委托给如下的所示某一个日志实现，日志记录优先级由上到下顺序递减：SLF4J、ApacheCommons Logging、Log4j2、Log4j、JDKlogging

如果MyBatis未发现上述日志记录实现，则MyBatis的日志记录功能无效。

如果你的运行环境中，在classpath中有多个可用的日志类库，并且你希望MyBaits使用某个特定的日志实现，你可以通过调用以下其中一个方法：

* org.apache.ibatis.logging.LogFactory.useSlf4jLogging();
* org.apache.ibatis.logging.LogFactory.useLog4JLogging();
* org.apache.ibatis.logging.LogFactory.useLog4J2Logging();
* org.apache.ibatis.logging.LogFactory.useJdkLogging();
* org.apache.ibatis.logging.LogFactory.useCommonsLogging();
* org.apache.ibatis.logging.LogFactory.useStdOutLogging();

**4、范围和生命周期**

**1）SqlSessionFactoryBuilder**

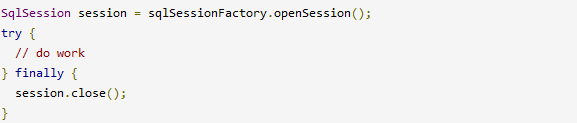
这个类可以被实例化、使用和丢弃。一旦你创建了 SqlSessionFactory 后,这个类就不需要存在了。 因此，SqlSessionFactoryBuilder 实例的最佳范围是**方法范围** (也就是本地方法变量)。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例, 但是最好的方式是不需要保持它一直存在来保证所有 XML 解析资源,因为还有更重要的事情要做。

**2）SqlSessionFactory**

一旦被创建，SqlSessionFactory 应该在你的应用执行期间都存在。没有理由来处理或重新创建它。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次。 这样的操作将被视为是非常糟糕的。 因此 SqlSessionFactory 的最佳范围是**应用范围**。 有很多方法可以做到, 最简单的就是使用**单例模式或者静态单例模式**。

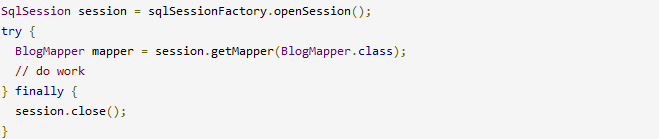
**3）SqlSession**

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例**不能被共享**,也是线程不安全的。因此最佳的范围是**请求或方法范围**。绝对不能将 SqlSession 实例的引用放在一个类的静态字段甚至是实例字段中。 也绝不能将 SqlSession 实例的引用放在任何类型的管理范围中, 比如Serlvet架构中的 HttpSession。 如果你现在正用任意的 Web 框架, 要考虑SqlSession放在一个和 HTTP请求对象相似的范围内。换句话说，基于收到的HTTP请求，你可以打开了一个SqlSession，然后返回响应，就可以关闭它了。关闭 Session 很重要,你应该确保使 用 finally 块来关闭它。下面的示例就是一个确保 SqlSession 关闭的基本模式:



**4）Mapper实例**

映射器是你创建绑定映射语句的接口。映射器接口的实例可以从 SqlSession 中获得。那么从技术上来说,当被请求时,任意映射器实例的最宽范围和 SqlSession 是相同的。然而, 映射器实例的最佳范围是**方法范围**。也就是说,它们应该在使用它们的方法中被请求,然后 就抛弃掉。它们不需要明确地关闭,那么在请求对象中保留它们也就不是什么问题了,这和 SqlSession 相似。你也许会发现,在这个水平上管理太多的资源的话会失控。保持简单,将 映射器放在方法范围内。下面的示例就展示了这个实践:



**5、Mapper XML文件**

**1）映射器配置文件和映射器接口**

MyBatis通过使用映射器Mapper接口提供了更好的调用映射语句的方法。一旦我们通过映射器配置文件配置了映射语句，我们可以创建一个完全对应的一个**映射器接口**，接口名跟配置文件名相同，接口所在包名也跟配置文件所在包名完全一样(如StudentMapper.xml所在的包名是com.mybatis3.mappers，对应的接口名就是com.mybatis3.mappers.StudentMapper

.java ）。映射器接口中的方法签名也跟映射器配置文件中完全对应：方法名为配置文件中id值；方法参数类型为parameterType对应值；方法返回值类型为returnType对应值。

**注：**即使映射器Mapper接口可以以类型安全的方式调用映射语句，但是我们负责书写正确的，匹配**方法名**、**参数类型**、和**返回值**的**映射器Mapper接口**。如果映射器Mapper接口中的方法和XML中的映射语句不能匹配，会在运行期抛出一个异常。实际上，指定parameterType是可选的；MyBatis 可以使用反射机制来决定parameterType。但是，从配置可读性的角度来看，最好指定parameterType属性。如果parameterType没有被提及，开发者必须查看Mapper XML配置和Java代码了解传递给语句的输入参数的数据类型。

**2）映射语句**

MyBatis提供了多种元素来配置不同类型的语句，如SELECT、INSERT、UPDATE、DELETE，接下来让我们看看如何具体配置映射语句

（1）insert语句

一个INSERT SQL语句可以在<insert>元素在映射器XML配置文件中配置，如下所示：

<insert id=*"insertStudent"* parameterType=*"Student"*>

INSERT INTO STUDENTS(STUD\_ID,NAME,EMAIL, PHONE)

VALUES(#{studId},#{name},#{email},#{phone})

</insert>

这里我们使用一个ID insertStudent，可以在命名空间com.mybatis3.mappers.

StudentMapper.insertStudent中唯一标识。parameterType属性应该是一个完全限定类名或者是一个类型别名（alias）。

也可以如下调用这个语句：

**int** count =

sqlSession.insert("com.mybatis3.mappers.StudentMapper.insertStudent", student);

sqlSession.insert() 方法返回执行INSERT语句后所影响的行数。

如果不使用名空间（namespace）和语句id来调用映射语句，你可以通过创建一个映射器Mapper接口，并以类型安全的方式调用方法，如下所示：

**package** com.mybatis3.mappers;

**public** **interface** StudentMapper{ **int** insertStudent(Student student); }

可以如下调用insertStudent 映射语句：

StudentMapper *mapper* = sqlSession.getMapper(StudentMapper.**class**);

**int** count = mapper.insertStudent(student);

【自动生成主键】

在上述的INSERT语句中，我们为可以自动生成（auto-generated）主键的列STUD\_ID插入值。我们可以使用useGeneratedKeys和keyProperty属性让数据库生成auto\_increment列的值，并将生成的值设置到其中一个输入对象属性内，如下所示：

<insert id=*"insertStudent"* parameterType=*"Student"*

useGeneratedKeys=*"true"* keyProperty=*"studId"*>

INSERT INTO STUDENTS(NAME, EMAIL, PHONE) VALUES(#{name},#{email},#{phone})

</insert>

这里STUD\_ID列值将会被MySQL 数据库自动生成，并且生成的值会被设置到student对象的studId属性上。

StudentMapper mapper = sqlSession.getMapper(StudentMapper.**class**);

mapper.insertStudent(student);

现在可以如下获取插入的STUDENT记录的STUD\_ID 的值：

**int** studentId = student.getStudId();

有些数据库如Oracle并不支持AUTO\_INCREMENT 列，其使用序列（SEQUENCE）来生成主键值。假设我们有一个名为STUD\_ID\_SEQ 的序列来生成SUTD\_ID主键值。使用如下代码来生成主键：

<insert id=*"insertStudent"* parameterType=*"Student"*>

<selectKey keyProperty=*"studId"* resultType=*"int"* order=*"BEFORE"*>

SELECT ELEARNING.STUD\_ID\_SEQ.NEXTVAL FROM DUAL

</selectKey>

INSERT INTO STUDENTS(STUD\_ID,NAME,EMAIL, PHONE) VALUES(#{studId},#{name},#{email},#{phone})

</insert>

这里我们使用了<selectKey>子元素来生成主键值，并将值保存到Student对象的studId属性上。属性order=“before”表示MyBatis将取得序列的下一个值作为主键值，并且在执行INSERT SQL语句之前将值设置到studId属性上。

我们也可以在获取序列的下一个值时，使用触发器（trigger）来设置主键值，并且在执行INSERT SQL语句之前将值设置到主键列上。如果你采取这样的方式，则对应的INSERT映射语句如下所示：

<insert id=*"insertStudent"* parameterType=*"Student"*>

INSERT INTO STUDENTS(NAME,EMAIL, PHONE) VALUES(#{name},#{email},#{phone})

<selectKey keyProperty=*"studId"* resultType=*"int"* order=*"AFTER"*>

SELECT ELEARNING.STUD\_ID\_SEQ.CURRVAL FROM DUAL

</selectKey>

</insert>

（2）update语句

一个UPDATE SQL语句可以在<update>元素在映射器XML配置文件中配置，如下所示：

<update id=*"updateStudent"* parameterType=*"Student"*>

UPDATE STUDENTS SET NAME=#{name}, EMAIL=#{email}, PHONE=#{phone} WHERE STUD\_ID=#{studId}

</update>

我们可以如下调用此语句：

**int** noOfRowsUpdated =

sqlSession.update("com.mybatis3.mappers.StudentMapper.updateStudent", student);

sqlSession.update() 方法返回执行UPDATE语句之后影响的行数。

如果不使用名空间（namespace）和语句id来调用映射语句，你可以通过创建一个映射器Mapper接口，并以类型安全的方式调用方法，如下所示：

**package** com.mybatis3.mappers;

**public** **interface** StudentMapper { **int** updateStudent(Student student); }

你可以使用映射器Mapper接口来调用updateStudent语句，如下所示：

StudentMapper *mapper* = sqlSession.getMapper(StudentMapper.**class**);

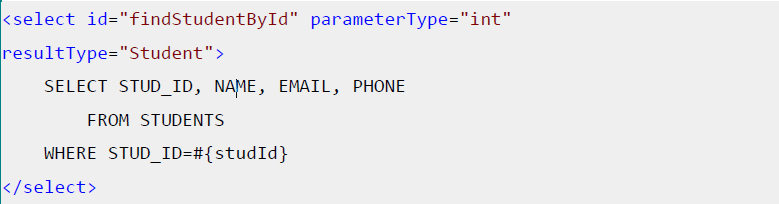
**int** *noOfRowsUpdated* = *mapper*.updateStudent(student);

（3）delete语句

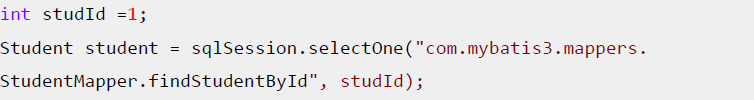
sqlSession.delete() 方法返回delete语句执行后影响的行数，其它类似update、insert。

（4）select语句

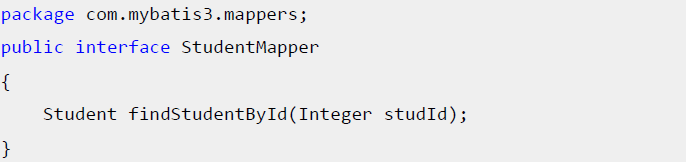
MyBatis真正强大的功能，在于映射SELECT查询结果到JavaBeans 方面的极大灵活性。让我们看看一个简单的select查询是如何（在MyBatis中）配置的，如下所示：



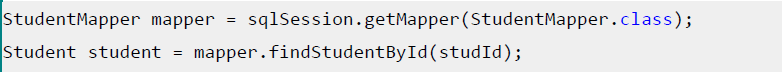
可以如下调用此语句：



如果不使用名空间（namespace）和语句id来调用映射语句，你可以通过创建一个映射器Mapper接口，并以类型安全的方式调用方法，如下所示：

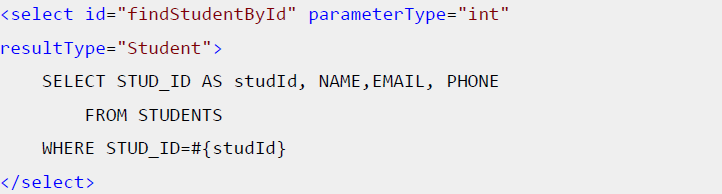


你可以使用映射器Mapper接口来调用updateStudent语句，如下所示：



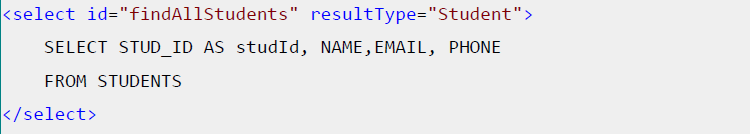
如果你检查Student对象的属性值，你会发现studId属性值并没有被stud\_id列值填充。这是因为MyBatis自动对JavaBean中和列名匹配的属性进行填充。这就是为什么name ,email,和phone属性被填充，而studId属性没有被填充。

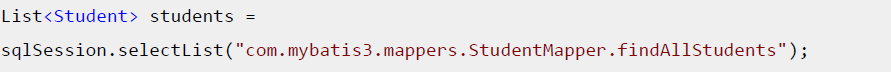
为了解决这一问题，我们可以为列名起一个可以与JavaBean中属性名匹配的别名，如下所示：



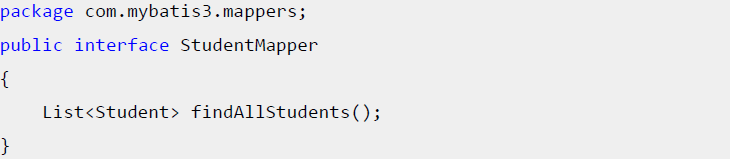
现在，Student 这个Bean对象中的值将会恰当地被stud\_id,name,email,phone列填充了。

现在，让我们看一下如何执行返回多条结果的SELECT语句查询，如下所示：





映射器Mapper接口StudentMapper可以如下定义：



使用上述代码，我们可以如下调用findAllStudents语句：



如果你注意到上述的SELECT映射定义，你可以看到，我们为所有的映射语句中的stud\_id起了别名。我们可以使用ResultMaps，来避免上述的到处重复别名。我们稍后会继续讨论。

除了java.util.List，你也可以是由其他类型的集合类，如Set,Map，以及（SortedSet）。MyBatis根据集合的类型，会采用适当的集合实现，如下所示：

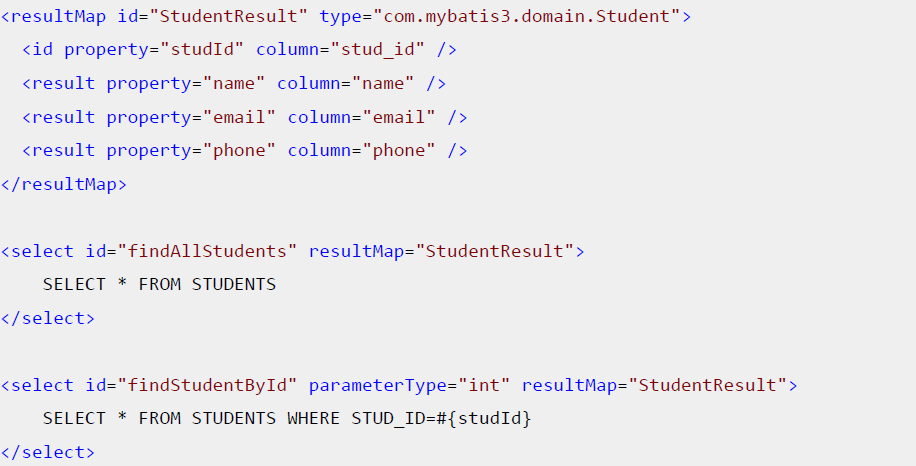
* 对于List，Collection，Iterable类型，MyBatis将返回java.util.ArrayList
* 对于Map类型，MyBatis将返回java.util.HashMap
* 对于Set类型，MyBatis将返回java.util.HashSet
* 对于SortedSet类型，MyBatis将返回java.util.TreeSet

**3）结果集映射ResultMaps**

ResultMaps被用来将SQL SELECT语句的结果集映射到JavaBeans的属性中。我们可以定义结果集映射ResultMaps并且在一些SELECT语句上引用resultMap。MyBatis的结果集映射ResultMaps特性非常强大，你可以使用它将简单的SELECT语句映射到复杂的一对一和一对多关系的SELECT语句上。（即描述如何从数据库结果集中来加载对象）

（1）简单ResultMap

一个映射了查询结果和Student JavaBean的简单的resultMap定义如下：



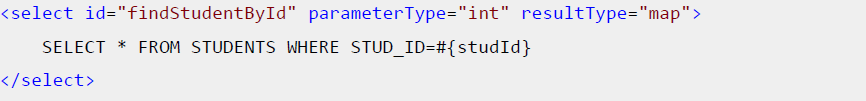
表示resultMap的StudentResult id值应该在此名空间内是唯一的。并且type属性应该是完全限定类名或者是返回类型的别名。

<result>子元素被用来将一个resultset列映射到JavaBean的一个属性中。

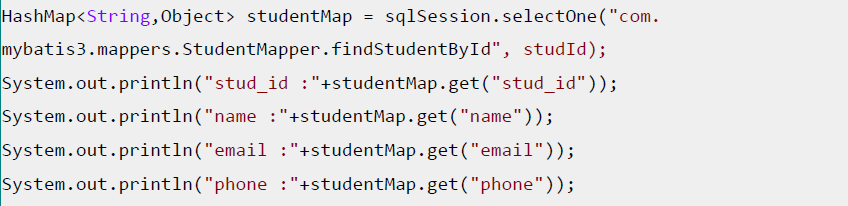
<id>元素和<result>元素功能相同，不过它**被用来映射到唯一标识属性**，用来区分和比较对象（一般和主键列相对应）。

在<select>语句中，我们使用了resultMap属性，而不是resultType来引用StudentResult映射。当<select>语句中配置了resutlMap属性，MyBatis会使用此数据库列名与对象属性映射关系来填充JavaBean中的属性。（注：resultType和resultMap二者只能用其一，不能同时使用）

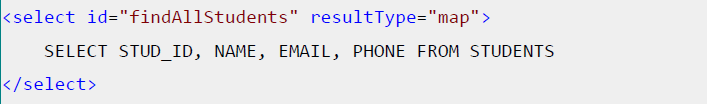
让我们来看另外一个<select>映射语句定义的例子，怎样将查询结果填充到HashMap中。如下所示：



在上述的<select>语句中，我们将resultType配置成map，即java.util.HashMap的别名。在这种情况下，结果集的列名将会作为Map中的key值，而列值将作为Map的value值。



让我们再看一个使用resultType=”map”,返回多行结果的例子：



由于resultType=”map”和语句返回多行，则最终返回的数据类型应该是List<HashMap<String,Object>>，如下所示：

List<HashMap<String, Object>> studentMapList = sqlSession.selectList("com.mybatis3.mappers.StudentMapper.findAllStudents");

**for**(HashMap<String, Object> *studentMap* : studentMapList) {

System.*out*.println("studId :" + *studentMap*.get("stud\_id"));

System.*out*.println("name :" + *studentMap*.get("name"));

System.*out*.println("email :" + *studentMap*.get("email"));

System.*out*.println("phone :" + *studentMap*.get("phone"));

}

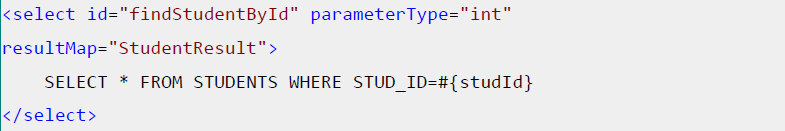
（2）扩展ResultMap

我们可以从从另外一个<resultMap>，拓展出一个新的<resultMap>，这样，原先的属性映射可以继承过来，以实现。

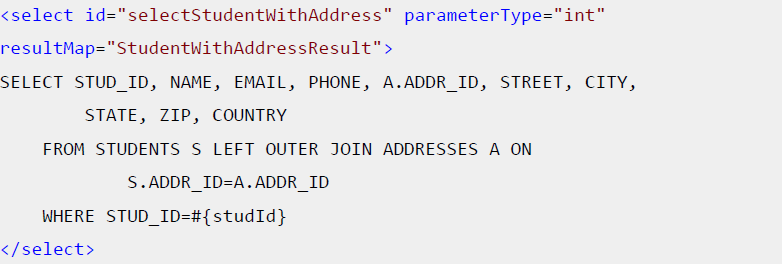


id为StudentWithAddressResult的resultMap拓展了id为StudentResult的resultMap。

如果你**只想映射Student数据**，你可以使用id为StudentResult 的resultMap,如下所示：



如果你想将映射Student数据和Address数据，你可以使用id为StudentWithAddressResult的resultMap：



**4）一对一映射**



下面让我们看一下怎样取Student明细和其Address明细：

Student和Address 的JavaBean以及映射器Mapper XML文件定义如下所示：

**public** **class** Student {

**private** Integer studId;

**private** String name;

**private** String email;

**private** PhoneNumber phone;

**private** Address address;

//setters & getters

}

**public** **class** Address {

**private** Integer addrId;

**private** String street;

**private** String city;

**private** String state;

**private** String zip;

**private** String country;

// setters & getters }}



我们可以使用**圆点记法**为**内嵌的对象的属性赋值**。在上述的resultMap中，Student的address属性使用了圆点记法被赋上了address对应列的值。同样地，我们可以访问任意深度的内嵌对象的属性。我们可以如下访问内嵌对象属性：

//接口定义

**public** **interface** StudentMapper {

Student selectStudentWithAddress(**int** studId); }

//使用

**int** *studId* = 1;

StudentMapper *studentMapper* = sqlSession.getMapper(StudentMapper.**class**);

Student *student* = *studentMapper*.selectStudentWithAddress(*studId*);

System.out.println("Student :" + student);

System.out.println("Address :" + student.getAddress());

上述样例展示了一对一关联映射的一种方法。然而，使用这种方式映射，如果address结果需要在其他的SELECT映射语句中映射成Address对象，我们需要为每一个语句重复这种映射关系。MyBatis提供了更好地实现一对一关联映射的方法：嵌套结果ResultMap和嵌套select查询语句。接下来，我们将讨论这两种方式。

（1）使用嵌套结果ResultMap实现一对一关系映射

我们可以使用一个嵌套结果ResultMap方式来获取Student及其Address信息，代码如下：

<resultMap type=*"Address"* id=*"AddressResult"*>

<id property=*"addrId"* column=*"addr\_id"* />

<result property=*"street"* column=*"street"* />

<result property=*"city"* column=*"city"* />

<result property=*"state"* column=*"state"* />

<result property=*"zip"* column=*"zip"* />

<result property=*"country"* column=*"country"* />

</resultMap>

<resultMap type=*"Student"* id=*"StudentWithAddressResult"*>

<id property=*"studId"* column=*"stud\_id"* />

<result property=*"name"* column=*"name"* />

<result property=*"email"* column=*"email"* />

<association property=*"address"* resultMap=*"AddressResult"* />

</resultMap>

<select id=*"findStudentWithAddress"* parameterType=*"int"*

resultMap=*"StudentWithAddressResult"*>

SELECT STUD\_ID, NAME, EMAIL, A.ADDR\_ID, STREET, CITY, STATE, ZIP, COUNTRY

FROM STUDENTS S LEFT OUTER JOIN ADDRESSES A ON S.ADDR\_ID=A.ADDR\_ID

WHERE STUD\_ID=#{studId}

</select>

元素<association>被用来导入“有一个”(has-one)类型的关联。在上述的例子中，我们使用了<association>元素引用了另外的在同一个XML文件中定义的<resultMap>。

我们也可以使用<association>定义**内联的resultMap**，代码如下所示：

<resultMap type=*"Student"* id=*"StudentWithAddressResult"*>

<id property=*"studId"* column=*"stud\_id"* />

<result property=*"name"* column=*"name"* />

<result property=*"email"* column=*"email"* />

<association property=*"address"* javaType=*"Address"*>

<id property=*"addrId"* column=*"addr\_id"* />

<result property=*"street"* column=*"street"* />

<result property=*"city"* column=*"city"* />

<result property=*"state"* column=*"state"* />

<result property=*"zip"* column=*"zip"* />

<result property=*"country"* column=*"country"* />

</association>

</resultMap>

使用嵌套结果ResultMap方式，关联的数据可以通过简单的查询语句（如果需要的话，需要与joins 连接操作配合）进行加载。

（2）使用嵌套查询实现一对一关系映射

我们可以通过使用嵌套select查询来获取Student及其Address信息，代码如下：

<resultMap type=*"Address"* id=*"AddressResult"*>

<id property=*"addrId"* column=*"addr\_id"* />

<result property=*"street"* column=*"street"* />

<result property=*"city"* column=*"city"* />

<result property=*"state"* column=*"state"* />

<result property=*"zip"* column=*"zip"* />

<result property=*"country"* column=*"country"* />

</resultMap>

<select id=*"findAddressById"* parameterType=*"int"* resultMap=*"AddressResult"*>

SELECT \* FROM ADDRESSES WHERE ADDR\_ID=#{id}

</select>

<resultMap type=*"Student"* id=*"StudentWithAddressResult"*>

<id property=*"studId"* column=*"stud\_id"* />

<result property=*"name"* column=*"name"* />

<result property=*"email"* column=*"email"* />

<association property=*"address"* column=*"addr\_id"* select=*"findAddressById"* />

</resultMap>

<select id=*"findStudentWithAddress"* parameterType=*"int"*

resultMap=*"StudentWithAddressResult"*>

SELECT \* FROM STUDENTS WHERE STUD\_ID=#{Id}

</select>

在此方式中，<association>元素的select属性被设置成了id为findAddressById的语句。这里，两个分开的SQL语句将会在数据库中执行，第一个调用findStudentById加载student信息，而第二个调用findAddressById来加载address信息。

Addr\_id列的值将会被作为**输入参数**传递给selectAddressById语句。

我们可以如下调用findStudentWithAddress映射语句：

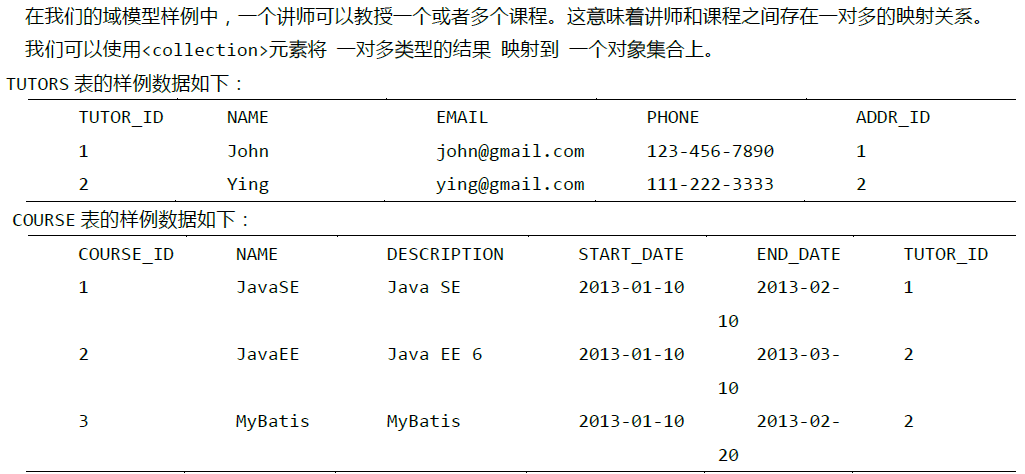
StudentMapper mapper = sqlSession.getMapper(StudentMapper.**class**);

Student student = mapper.selectStudentWithAddress(studId);

System.out.println(student);

System.out.println(student.getAddress());

**5）一对多映射**



在上述的表数据中，John讲师教授一个课程，而Ying讲师教授两个课程。

Course和Tutor的JavaBean定义如下：

**public** **class** Tutor {

**private** Integer tutorId;

**private** String name;

**private** String email;

**private** Address address;

**private** List<Course> courses;

// setters & getters

}

**public** **class** Course {

**private** Integer courseId;

**private** String name;

**private** String description;

**private** Date startDate;

**private** Date endDate;

**private** Integer tutorId;

//setters & getters }

现在让我们看看如何获取讲师信息以及其所教授的课程列表信息。

**<collection>元素**被用来将多行课程结果映射成一个课程Course对象的一个集合。和一对一映射一样，我们可以使用**嵌套结果ResultMap**和**嵌套Select语句**两种方式映射实现一对多映射。

（1）使用内嵌结果ResultMap实现一对多映射

我们可以使用嵌套结果resultMap方式获得讲师及其课程信息，代码如下：

<resultMap type=*"Course"* id=*"CourseResult"*>

<id column=*"course\_id"* property=*"courseId"* />

<result column=*"name"* property=*"name"* />

<result column=*"description"* property=*"description"* />

<result column=*"start\_date"* property=*"startDate"* />

<result column=*"end\_date"* property=*"endDate"* />

</resultMap>

<resultMap type=*"Tutor"* id=*"TutorResult"*>

<id column=*"tutor\_id"* property=*"tutorId"* />

<result column=*"tutor\_name"* property=*"name"* />

<result column=*"email"* property=*"email"* />

<collection property=*"courses"* resultMap=*"CourseResult"* />

</resultMap>

<select id=*"findTutorById"* parameterType=*"int"* resultMap=*"TutorResult"*>

SELECT T.TUTOR\_ID, T.NAME AS TUTOR\_NAME, EMAIL, C.COURSE\_ID, C.NAME, DESCRIPTION,

START\_DATE, END\_DATE FROM TUTORS T **LEFT OUTER JOIN** ADDRESSES A ON T.ADDR\_ID=A.ADDR\_ID

**LEFT OUTER JOIN** COURSES C ON T.TUTOR\_ID=C.TUTOR\_ID WHERE T.TUTOR\_ID=#{tutorId}

</select>

这里我们使用了一个简单的使用了JOINS连接的Select语句获取讲师及其所教课程信息。**<collection>元素**的resultMap属性设置成了CourseResult，CourseResult包含了Course对象属性与表列名之间的映射。

（2）使用嵌套select语句实现一对多映射

我们可以使用嵌套Select语句方式获得讲师及其课程信息，代码如下：

<resultMap type=*"Course"* id=*"CourseResult"*>

<id column=*"course\_id"* property=*"courseId"* />

<result column=*"name"* property=*"name"* />

<result column=*"description"* property=*"description"* />

<result column=*"start\_date"* property=*"startDate"* />

<result column=*"end\_date"* property=*"endDate"* />

</resultMap>

<resultMap type=*"Tutor"* id=*"TutorResult"*>

<id column=*"tutor\_id"* property=*"tutorId"* />

<result column=*"tutor\_name"* property=*"name"* />

<result column=*"email"* property=*"email"* />

<association property=*"address"* resultMap=*"AddressResult"* />

<collection property=*"courses"* column=*"tutor\_id"* select=*"findCoursesByTutor"* />

</resultMap>

<select id=*"findTutorById"* parameterType=*"int"* resultMap=*"TutorResult"*>

SELECT T.TUTOR\_ID, T.NAME AS TUTOR\_NAME, EMAIL FROM TUTORS T WHERE T.TUTOR\_ID=#{tutorId}

</select>

<select id=*"findCoursesByTutor"* parameterType=*"int"* resultMap=*"CourseResult"*>

SELECT \* FROM COURSES WHERE TUTOR\_ID=#{tutorId}

</select>

在这种方式中，<aossication>元素的select属性被设置为id 为findCourseByTutor的语句，用来触发单独的SQL查询加载课程信息。**tutor\_id这一列值将会作为输入参数**传递给findCouresByTutor 语句。

**public** **interface** TutorMapper {

Tutor findTutorById(**int** tutorId);

}

TutorMapper mapper = sqlSession.getMapper(TutorMapper.**class**);

Tutor tutor = mapper.findTutorById(tutorId);

System.out.println(tutor);

List<Course> courses = tutor.getCourses();

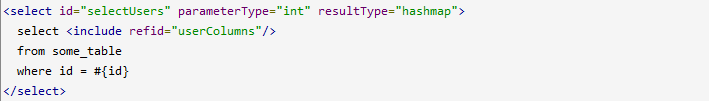
**for** (Course course : courses) { System.*out*.println(course); }

**6）SQL**

这个元素可以被用来定义可重用的 SQL 代码段,可以包含在其他语句中。比如:

**<sql id=*"userColumns"*> id,username,password </sql>**

这个 SQL 片段可以被包含在其他语句中,例如:



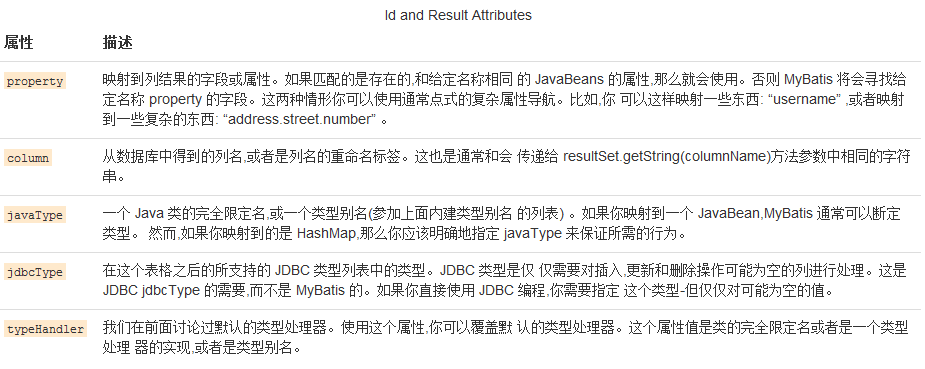
**7）resultMap**



（1）id&result

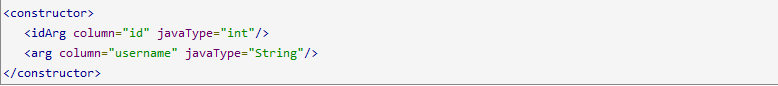


这些是结果映射最基本内容，id 和 result 都映射一个单独列的值到简单数据类型(字符串、整型、双精度浮点数、日期等)的单独属性或字段。这两者之间的唯一不同是id表示的结果将是当比较对象实例时用到的标识属性。这帮助来改进整体表现，特别是缓存和嵌入结果映射(也就是联合映射)。下列是其具有的属性：





（2）构造方法constructor

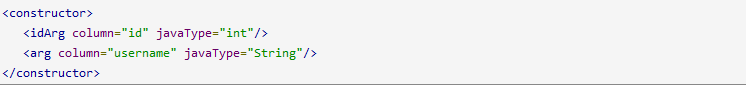


对于大多数数据传输对象(Data Transfer Object,DTO)类型,属性可以起作用,而且像 你绝大多数的领域模型, 指令也许是你想使用一成不变的类的地方。通常包含引用或查询数 据的表很少或基本不变的话对一成不变的类来说是合适的。**构造方法注入**允许你在初始化时 为类设置属性的值,而不用暴露出公有方法。MyBatis也支持私有属性和私有JavaBeans属 性来达到这个目的,但是一些人更青睐构造方法注入，构造方法元素支持这个。

看看下面这个构造方法:

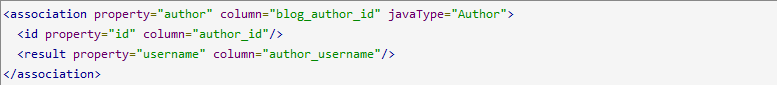


为了向这个构造方法中注入结果，MyBatis需要通过它的**参数的类型**来**标识构造方法**。 Java 没有自查(反射)参数名的方法。所以当创建一个构造方法元素时,保证参数是按顺序 排列的,而且数据类型也是确定的。



剩余的属性和规则和固定的 id 和 result 元素是相同的。

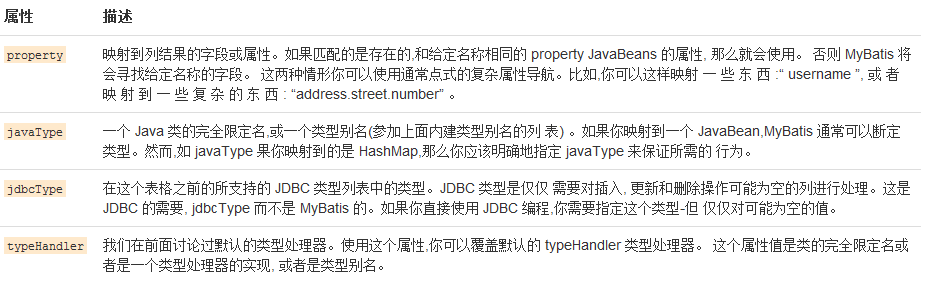
（3）关联association



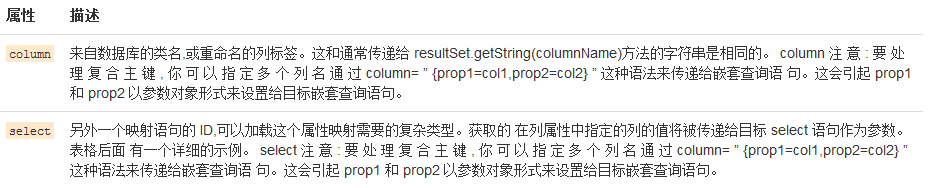
**关联元素**处理“有一个”类型的关系。比如,在我们的示例中,一个博客有一个用户。 关联映射就工作于这种结果之上。你指定了目标属性,来获取值的列,属性的 java 类型(很多情况下MyBatis可以自己算出来)，如果需要的话还有 jdbc 类型,如果你想覆盖或获取的 结果值还需要类型控制器。

关联中不同的是你需要告诉 MyBatis **如何加载关联**。MyBatis 在这方面会有两种不同的 方式:

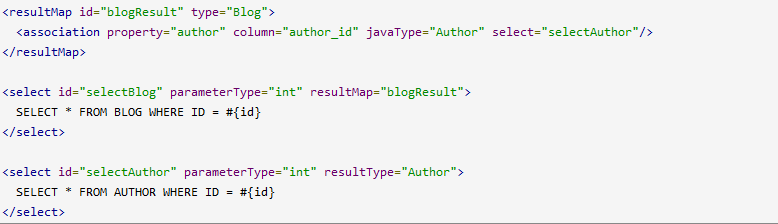
* 嵌套查询：通过执行另外一个 SQL 映射语句来返回预期的复杂类型。
* 嵌套结果：使用嵌套结果映射来处理重复的联合结果的子集。首先,让我们来查看这个元素的属性。所有的你都会看到,它和普通的只由 select 和 resultMap 属性的结果映射不同。



* 关联的嵌套查询：



示例：



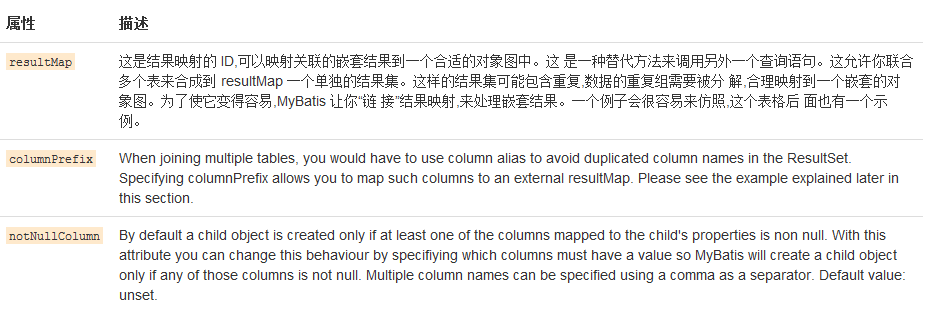
我们有两个查询语句：一个来加载博客,另外一个来加载作者,而且博客的结果映射描 述了“selectAuthor”语句应该被用来加载它的 author 属性。其他所有的属性将会被自动加载,假设它们的**列和属性名**相匹配。

这种方式很简单，但是对于**大型数据集合和列表**将不会表现很好。问题就是我们熟知的 “N+1 查询问题”。概括地讲，N+1 查询问题可以是这样引起的:

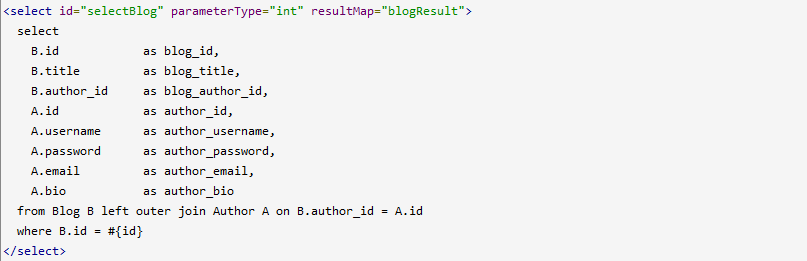
* 你执行了一个单独的 SQL 语句来获取结果列表(就是“+1”)。
* 对返回的每条记录,你执行了一个查询语句来为每个加载细节(就是“N”)。

这个问题会导致成百上千的 SQL 语句被执行。这通常不是期望的。MyBatis 能延迟加载这样的查询就是一个好处，因此你可以分散这些语句同时运行的消耗。然而，如果你加载一个列表，之后迅速迭代来访问嵌套的数据,你会调用所有的延迟加载，这样的行为可能是很糟糕的。

* 关联的嵌套结果：



在上面你已经看到了一个非常复杂的嵌套关联的示例。下面这个是一个非常简单的示例 来说明它如何工作。代替执行一个分离的语句,我们联合博客表和作者表在一起,就像:



注意这个联合查询, 以及采取保护来确保所有结果被唯一而且清晰的名字来重命名。 这使得映射非常简单。现在我们可以映射这个结果:



在上面的示例中你可以看到博客的作者关联代表着“authorResult”结果映射来加载作 者实例。

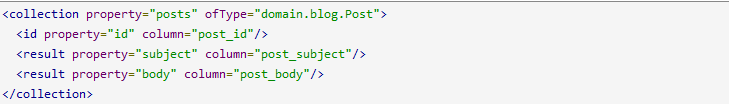
**非常重要:** 在嵌套结果映射中 id 元素扮演了非常重要的角色。应该通常指定一个或多个属性,它们可以用来唯一标识结果。实际上就是如果你离开她了,但是有一个严重的性能问题时 MyBatis 仍然可以工作。选择的属性越少越好,它们可以**唯一地标识结果**。主键就是一个显而易见的选择(尽管是联合主键)。

现在,上面的示例用了外部的结果映射元素来映射关联。这使得 Author 结果映射可以 重用。然而,如果你不需要重用它的话,或者你仅仅引用你所有的结果映射合到一个单独描 述的结果映射中。你可以嵌套结果映射。这里给出使用这种方式的相同示例:



（4）集合collection

用于处理多个类型关联的数据……



**集合元素**的作用几乎和关联是相同的。实际上,它们也很相似,文档的异同是多余的。所以我们更多关注于它们的不同。我们来继续上面的示例,一个博客只有一个作者。但是博客有很多文章。在博客类中, 这可以由下面这样的写法来表示:



要**映射嵌套结果集合到 List**中,我们使用集合元素。就像关联元素一样,我们可以从 连接中使用嵌套查询,或者嵌套结果。

* 集合的嵌套查询：

首先,让我们看看使用嵌套查询来为博客加载文章：



这里你应该注意很多东西,但大部分代码和上面的关联元素是非常相似的。首先,你应 该注意我们使用的是**集合元素**。然后要注意那个新的“ofType”属性。这个属性用来区分 **JavaBean(或字段)属性类型和集合包含的类型**来说是很重要的。所以你可以读出下面这个 映射:



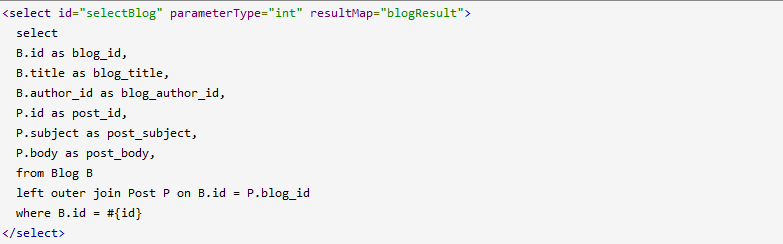
读作: “在 Post 类型的 ArrayList 中的 posts 的集合。”

javaType 属性是不需要的,因为 MyBatis 在很多情况下会为你算出来。所以你可以缩短 写法:

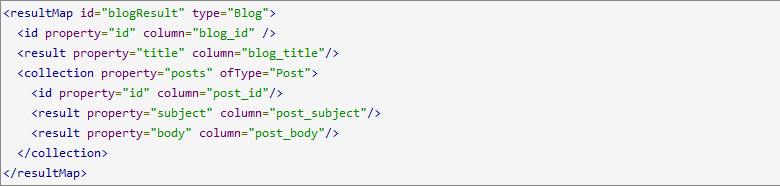


* 集合的嵌套结果：

至此,你可以猜测集合的嵌套结果是如何来工作的,因为它和关联完全相同,除了它应 用了一个“ofType”属性 。



我们又一次联合了博客表和文章表,而且关注于保证特性,结果列标签的简单映射。现 在用文章映射集合映射博客,可以简单写为:



同样,要记得 id 元素的重要性,如果你不记得了,请阅读上面的关联部分。

同样, 如果你引用更长的形式允许你的结果映射的更多重用, 你可以使用下面这个替代的映射:



这个对你所映射的内容没有深度,广度或关联和集合相联合的限制。当映射它们时你应该在大脑中保留它们的表现。你的应用在找到最佳方法前要一直进行的**单元测试和性能测试**。好在 myBatis 让你后来可以改变想法,而不对你的代码造成很小(或任何)影响。

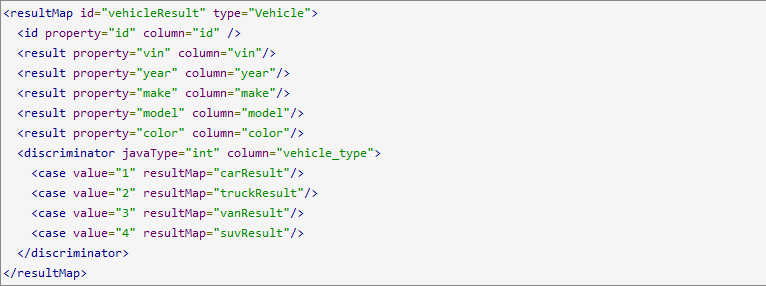
**高级关联和集合映射**是一个深度的主题。文档只能给你介绍到这了。加上一点联系,你 会很快清楚它们的用法。

（5）鉴别器discriminator

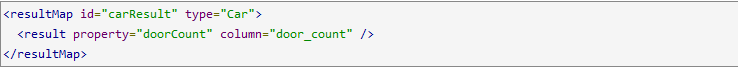


有时一个单独的数据库查询也许返回很多不同 (但是希望有些关联) 数据类型的结果集。鉴别器元素就是被设计来处理这个情况的，还有包括类的继承层次结构。鉴别器非常容易理解，因为它的表现很像 Java 语言中的 switch 语句。

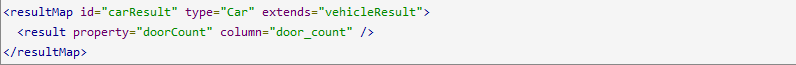
定义鉴别器指定了column 和 javaType 属性。列是 MyBatis 查找比较值的地方。JavaType 是需要**被用来保证等价测试的合适类型**(尽管字符串在很多情形下都会有用)。比如:



在这个示例中, MyBatis 会从结果集中得到每条记录, 然后比较它的 vehicle 类型的值。如果它匹配任何一个鉴别器的实例,那么就使用这个实例指定的结果映射。换句话说,这样做完全是剩余的结果映射被忽略(除非它被扩展,这在第二个示例中讨论) 。如果没有任何一个实例相匹配,那么 MyBatis 仅仅使用鉴别器**块外定义的结果映射**。所以,如果 carResult 按如下声明:



那么只有 doorCount 属性会被加载。这步完成后完整地允许鉴别器实例的独立组,尽管 和父结果映射可能没有什么关系。这种情况下,我们当然知道 cars 和 vehicles 之间有关系, 如 Car 是一个 Vehicle 实例。因此,我们想要剩余的属性也被加载。我们设置的结果映射的简单改变如下。



现在 vehicleResult 和 carResult 的属性都会被加载了。

尽管曾经有些人会发现这个外部映射定义会多少有一些令人厌烦之处。 因此还有另外一 种语法来做简洁的映射风格。比如:



**注意：**这些都是结果映射，如果你不指定任何结果，那么MyBatis将会为你自动匹配列和属性。所以这些例子中的大部分是很冗长的,而其实是不需要的。也就是说,很多数据库是很复杂的,我们不太可能对所有示例都能依靠它。

（6）缓存cache

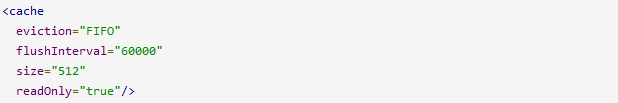
MyBatis 包含一个非常强大的**查询缓存特性**,它可以非常方便地配置和定制。MyBatis 3 中的缓存实现的很多改进都已经实现了,使得它更加强大而且易于配置。

**默认情况下是没有开启缓存的**,除了局部的 **session 缓存**,可以增强变现而且处理循环依赖也是必须的。要开启二级缓存,你需要在你的 SQL 映射文件中添加一行: **<cache/>**

字面上看就是这样。这个简单语句的效果如下:

* + 映射语句文件中的所有 select 语句将会**被缓存**；
  + 映射语句文件中的所有 insert,update 和 delete 语句会**刷新缓存**；
  + 缓存会使用 Least Recently Used(LRU,最近最少使用的)算法来收回；
  + 根据时间表(比如 no Flush Interval,没有刷新间隔), 缓存不会以任何时间顺序 来刷新；
  + 缓存会存储列表集合或对象(无论查询方法返回什么)的 1024 个引用；
  + 缓存会被视为是 read/write(可读/可写)的缓存,意味着对象检索不是共享的,而 且可以安全地被调用者修改,而不干扰其他调用者或线程所做的潜在修改。

所有的这些属性都可以通过缓存元素的属性来修改。比如:



这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会 导致冲突。

可用的收回策略有（默认的是 LRU）:

* LRU – 最近最少使用的：移除最长时间不被使用的对象；
* FIFO – 先进先出：按对象进入缓存的顺序来移除它们；
* SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象；
* WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。

**flushInterval(刷新间隔)**可以被设置为任意的正整数,而且它们代表一个合理的毫秒 形式的时间段。默认情况是不设置,也就是没有刷新间隔,缓存仅仅调用语句时刷新。

**size(引用数目)**可以被设置为任意正整数,要记住你缓存的**对象数目**和你运行环境的 可用**内存资源数目**。默认值是 1024。

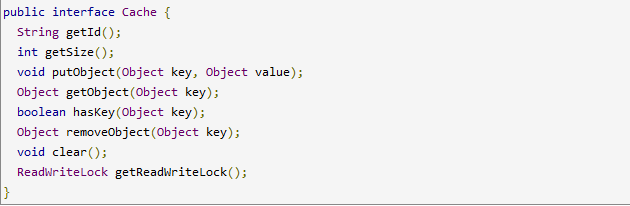
**readOnly(只读)**属性可以被设置为 true 或 false。只读的缓存会给所有调用者**返回缓 存对象的相同实例**，因此这些对象不能被修改。这提供了很重要的性能优势，可读写的缓存会返回缓存对象的拷贝(通过序列化) 。这会慢一些,但是安全,因此默认是 false。

* 使用自定义缓存

除了这些自定义缓存的方式, 你也可以通过**实现你自己的缓存**或为其他第三方缓存方案创建适配器来完全覆盖缓存行为。



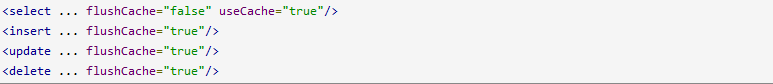
这个示例展示了如何使用一个自定义 的缓存实现。type属性指定的类必须实现 org.mybatis.cache.Cache 接口。这个接口是MyBatis框架中很多复杂的接口之一，但是简单给定它做什么就行。



要配置你的缓存, 简单和公有的 JavaBeans 属性来配置你的缓存实现, 而且是通过 cache 元素来传递属性, 比如, 下面代码会在你的缓存实现中调用一个称为 “setCacheFile(String file)” 的方法:



你可以使用所有简单类型作为 JavaBeans 的属性，MyBatis 会进行转换。记得**缓存配置和缓存实例**是绑定在 SQL 映射文件的**命名空间**是很重要的。因此,所有在相同命名空间的语句正如绑定的缓存一样，语句可以修改和缓存交互的方式，或在语句的基础上使用两种简单的属性来完全排除它们。默认情况下,语句可以这样来配置:



因为那些是默认的,你明显不能明确地以这种方式来配置一条语句。相反,如果你想改 变默认的行为,只能设置 flushCache 和 useCache 属性。比如,在一些情况下你也许想排除 从缓存中查询特定语句结果,或者你也许想要一个查询语句来刷新缓存。相似地,你也许有 一些更新语句依靠执行而不需要刷新缓存。

* 参照缓存

回想一下上一节内容, 这个**特殊命名空间**的**唯一缓存**会被使用或者刷新相同命名空间内的语句。也许将来的某个时候，你会想在命名空间中共享相同的缓存配置和实例。在这样的情况下你可以使用 cache-ref 元素来引用另外一个缓存。



**6、动态SQL**

有时候，静态的SQL语句并不能满足应用程序的需求。我们可以根据一些条件，来动态地构建SQL语句。例如，在Web应用程序中，有可能有一些搜索界面，需要输入一个或多个选项，然后根据这些已选择的条件去执行检索操作。在实现这种类型的搜索功能，我们可能需要根据这些条件来**构建动态的SQL语句**。如果用户提供了任何输入条件，我们需要将那个条件添加到SQL语句的**WHERE子句**中。

MyBatis通过使用<if>,<choose>,<where>,<foreach>,<trim>元素提供了对构造动态SQL语句的高级别支持。

**1）If条件**

<if>元素**被用来有条件地嵌入SQL片段**，如果测试条件被赋值为true，则相应地SQL片段将会被添加到SQL语句中。

假定我们有一个课程搜索界面，设置了**讲师（Tutor）**下拉列表框，**课程名称（CourseName）**文本输入框，**开始时间（StartDate）**输入框，**结束时间（EndDate）**输入框，作为搜索条件。假定课讲师下拉列表是必须选的，其他的都是可选的。

当用户点击搜索按钮时，我们需要显示符合以下条件的列表：

* 特定讲师的课程
* 课程名包含输入的课程名称关键字的课程；如果课程名称输入为空，则取所有课程
* 在开始时间和结束时间段内的课程

我们可以对应的映射语句，如下所示：(XML code)

<resultMap type=*"Course"* id=*"CourseResult"*>

<id column=*"course\_id"* property=*"courseId"* />

<result column=*"name"* property=*"name"* />

<result column=*"description"* property=*"description"* />

<result column=*"start\_date"* property=*"startDate"* />

<result column=*"end\_date"* property=*"endDate"* />

</resultMap>

<select id=*"searchCourses"* parameterType=*"hashmap"* resultMap=*"CourseResult"*>

SELECT \* FROM COURSES WHERE TUTOR\_ID= #{tutorId}

<if test=*"courseName != null"*>

AND NAME LIKE #{courseName}

</if>

<if test=*"startDate != null"*>

AND START\_DATE >= #{startDate}

</if>

<if test=*"endDate != null"*>

AND END\_DATE <= #{endDate}

</if>

</select>

(Java code)

**public** **interface** CourseMapper {

List<Course> searchCourses(Map<String, Object> map);

}

**public** **void** searchCourses() {

Map<String, Object> map = **new** HashMap<String, Object>();

map.put("tutorId", 1);

map.put("courseName", "%java%");

map.put("startDate", **new** Date());

CourseMapper mapper = sqlSession.getMapper(CourseMapper.**class**);

List<Course> courses = mapper.searchCourses(map);

}

**for** (Course course : courses) {

System.*out*.println(course); }

此处将生成查询语句SELECT \* FROM COURSES WHERE TUTOR\_ID= ? AND NAME like ? AND START\_DATE >= ?。准备根据给定条件的动态SQL查询将会派上用场。

**2）choose、when和otherwhile条件**

有时候，查询功能是以**查询类别**为基础的。首先，用户需要选择是否希望通过选择讲师，课程名称，开始时间，或结束时间作为**查询条件类别**来进行查询，然后**根据选择的查询类别，输入相应的参数**。在这样的情景中，我们需要只使用其中一种查询类别。

MyBatis 提供了<choose>元素支持此类型的SQL预处理。现在让我们书写一个适用此情景的SQL映射语句。如果没有选择查询类别，则查询开始时间在今天之后的课程，代码如下：

<select id=*"searchCourses"* parameterType=*"hashmap"* resultMap=*"CourseResult"*>

SELECT \* FROM COURSES

<choose>

<when test=*"searchBy == 'Tutor'"*>

WHERE TUTOR\_ID= #{tutorId}

</when>

<when test=*"searchBy == 'CourseName'"*>

WHERE name like #{courseName}

</when>

<otherwise>

WHERE TUTOR start\_date >= now()

</otherwise>

</choose>

</select>

MyBatis计算<choose>测试条件的值，且使用第一个值为TRUE的子句。如果没有条件为true，则使用<otherwise>内的子句。

**3）where条件**

有时候，所有的查询条件（criteria）应该是**可选的**。在需要使用至少一种查询条件的情况下，我们应该使用WHERE子句。并且，如果有多个条件，我们需要在条件中添加AND或OR。MyBatis提供了<where>元素支持这种类型的动态SQL语句。

在我们查询课程界面，我们假设**所有的查询条件是可选**的。进而，当需要提供一个或多个查询条件时，应该改使用WHERE子句。

<select id=*"searchCourses"* parameterType=*"hashmap"* resultMap=*"CourseResult"*>

SELECT \* FROM COURSES

<where>

<if test=*" tutorId != null "*>

TUTOR\_ID= #{tutorId}

</if>

<if test=*"courseName != null"*>

AND name like #{courseName}

</if>

<if test=*"startDate != null"*>

AND start\_date >= #{startDate}

</if>

<if test=*"endDate != null"*>

AND end\_date <= #{endDate}

</if>

</where>

</select>

<where>元素只有**在其内部标签有返回内容时才会在动态语句上插入WHERE条件语句**。并且，如果WHERE子句以AND或者OR打头，则打头的AND或OR将会被移除。

如果tutor\_id参数值为null，并且courseName参数值不为null，则<where>标签会将AND name like #{courseName}中的AND移除掉，生成的SQL WHERE子句为：where name like #{courseName}。

**4）<trim>条件**

<trim>元素和<where>元素类似，但是<trim>提供了在**添加前缀(**prefixOverrides**)/后缀(**suffixOverrides**)**或者**移除前缀/后缀**方面提供更大的灵活性。

<select id=*"searchCourses"* parameterType=*"hashmap"* resultMap=*"CourseResult"*>

SELECT \* FROM COURSES

<trim prefix=*"WHERE"* prefixOverrides=*"AND | OR"*>

<if test=*" tutorId != null "*>

TUTOR\_ID= #{tutorId}

</if>

<if test=*"courseName != null"*>

AND name like #{courseName}

</if>

</trim>

</select>

这里如果任意一个<if>条件为true，<trim>元素会插入WHERE,并且移除紧跟WHERE后面的AND或OR。

**5）foreach循环**

另外一个强大的动态SQL语句构造标签即是<foreach>。它可以**迭代遍历**一个数组或者列表，**构造AND/OR条件**或**一个IN子句**。

假设我们想找到tutor\_id为1，3，6的讲师所教授的课程，我们可以传递一个tutor\_id组成的**列表**给映射语句，然后通过<foreach>遍历此列表构造动态SQL。

<select id=*"searchCoursesByTutors"* parameterType=*"map"* resultMap=*"CourseResult"*>

SELECT \* FROM COURSES

<if test=*"tutorIds != null"*>

<where>

<foreach item=*"tutorId"* collection=*"tutorIds"*>

OR tutor\_id=#{tutorId}

</foreach>

</where>

</if>

</select>

**public** **interface** CourseMapper {

List<Course> searchCoursesByTutors(Map<String, Object> map);

}

**public** **void** searchCoursesByTutors() {

Map<String, Object> map = **new** HashMap<String, Object>();

List<Integer> tutorIds = **new** ArrayList<Integer>();

tutorIds.add(1);

tutorIds.add(3);

tutorIds.add(6);

map.put("tutorIds", tutorIds);

CourseMapper mapper = sqlSession.getMapper(CourseMapper.**class**);

List<Course> courses = mapper.searchCoursesByTutors(map);

**for** (Course course : courses) {

System.*out*.println(course); }

}

现在让我们来看一下怎样使用<foreach>生成IN子句：



**6）set条件**

<set>元素和<where>元素类似，如果**其内部条件判断有任何内容返回**时，他会插入SET SQL片段。

<update id=*"updateStudent"* parameterType=*"Student"*>

update students

<set>

<if test=*"name != null"*>

name=#{name},

</if>

<if test=*"email != null"*>

email=#{email},

</if>

<if test=*"phone != null"*>

phone=#{phone},

</if>

</set>

where stud\_id=#{id}

</update>

这里，如果<if>条件返回了**任何文本内容**，<set>将会插入set关键字和其文本内容，并且会剔除将末尾的“，”。在上述的例子中，如果phone!=null，<set>将会让会移除phone=#{phone}后的逗号“,”，生成set phone=#{phone}。

**7、MyBatis食谱**

除了简化数据库编程外，MyBatis还提供了各种功能，这些对实现一些常用任务非常有用，比如**按页加载表数据**、**存取CLOB/BLOB类型的数据**、**处理枚举类型值**等等。让我们来看看其中一些特性吧。

**1）处理枚举类型**

MyBatis支持开箱方式持久化enum 类型属性。假设STUDENTS表中有一列gender（性别）类型为varchar，存储”MALE”或者“FEMALE”两种值。并且，Student对象有一个enum类型的gender属性，如下所示：

**public** **enum** Gender { *FEMALE*, *MALE* }

默认情况下，MyBatis使用EnumTypeHandler来处理enum类型的Java属性，并且将其存储为enum值的名称。你不需要为此做任何额外的配置。你可以可以向使用基本数据类型属性一样使用enum类型属性，代码如下：

**public** **class** Student {

**private** Integer id;

**private** String name;

**private** String email;

**private** PhoneNumber phone;

**private** Address address;

**private** Gender gender;

//setters and getters

}

<insert id=*"insertStudent"* parameterType=*"Student"* useGeneratedKeys=*"true"*

keyProperty=*"id"*>

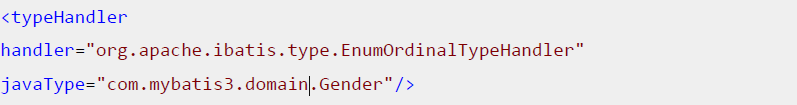
insert into students(name,email,addr\_id, phone,gender)

values(#{name},#{email},#{address.addrId},#{phone},#{gender})

</insert>

当你执行insertStudent语句的时候，MyBatis会取Gender枚举（FEMALE/MALE）的名称，然后将其存储到GENDER列中。

如果你希望存储原enum的顺序位置，而不是enum名，你需要明确地配置它。如果你想存储FEMALE为0，MALE为1到gender列中，你需要在mybatis-config.xml文件中配置EnumOrdinalTypeHandler:



**2）传入多个输入参数**

MyBatis中的映射语句有一个**parameterType**属性来**制定输入参数的类型**。如果我们想给映射语句**传入多个参数**的话，我们可以将所有的输入参数放到**HashMap**中，将HashMap传递给映射语句。

MyBatis 还提供了另外一种传递多个输入参数给映射语句的方法。假设我们想通过给定的name和email信息查找学生信息，定义查询接口如下：

Public **interface** StudentMapper {

List<Student> findAllStudentsByNameEmail(String name, String email); }

MyBatis 支持将多个输入参数传递给映射语句，并以**#{param}**的语法形式**引用**它们：

<select id=*"findAllStudentsByNameEmail"* resultMap=*"StudentResult"*>

select stud\_id, name,email, phone from Students

where name=#{param1} and email=#{param2}

</select>

这里#{param1}引用第一个参数name，而#{param2}引用了第二个参数email：

StudentMapper studentMapper = sqlSession.getMapper(StudentMapper. **class**);

studentMapper.findAllStudentsByNameEmail(name, email);

**3）多行结果集映射成Map**

如果你有一个映射语句返回多行记录，并且你想以HashMap的形式存储记录的值，使用记录列名作为key值，而记录对应值或为value值。我们可以使用sqlSession.selectMap(),如下所示：

<select id=*" findAllStudents"* resultMap=*"StudentResult"*>

select \* from Students

</select>

Map<Integer, Student> studentMap = sqlSession.

selectMap("com.mybatis3.mappers.StudentMapper.findAllStudents", "studId");

这里studentMap将会将studId作为key值，而Student对象作为value值。

**4）使用RowBounds对结果集进行分页**

有时候，我们会需要跟海量的数据打交道，比如一个有数百万条数据级别的表。由于计算机内存的现实我们不可能一次性加载这么多数据，我们可以获取到数据的一部分。特别是在Web应用程序中，分页机制被用来以一页一页的形式**展示海量的数据**。

MyBatis可以使用**RowBounds逐页加载表数据**。RowBounds对象可以使用**offset**和**limit**参数来构建。参数offset表示**开始位置**，而limit表示要取的**记录的数目**。假设如果你想每页加载并显示25条学生的记录，你可以使用如下的代码：

<select id=*"findAllStudents"* resultMap=*"StudentResult"*>

select \* from Students

</select>

然后，你可以加载如下加载第一页数据（前25条）：

**int** offset =0 , limit =25;

RowBounds rowBounds = **new** RowBounds(offset, limit);

List<Student> = studentMapper.getStudents(rowBounds);

若要展示第二页，使用offset=25,limit=25;第三页，则为offset=50，limit=25。

**三、使用注解配置SQL映射器**

**1、在映射器Mapper接口上使用注解**

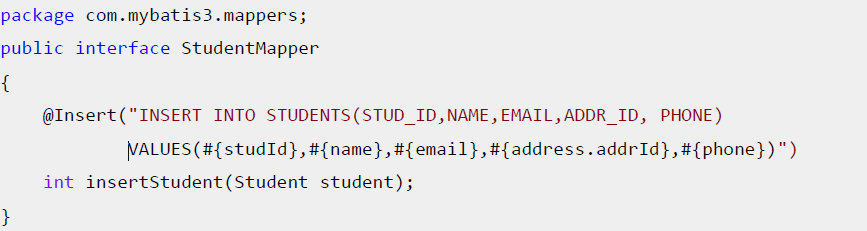
MyBatis对于大部分的基于XML的映射器元素（包括<select>,<update>）提供了对应的基于注解的配置项。然而在某些情况下，基于注解配置还不能支持基于XML的一些元素。

**2、映射语句**

MyBatis提供了多种注解来支持不同类型的语句(statement)如SELECT、INSERT、UPDATE、DELETE。让我们看一下具体怎样配置映射语句。

**1）@insert**

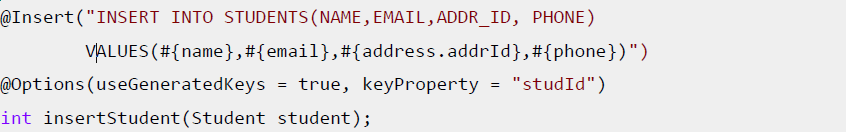
我们可以使用@Insert注解来定义一个INSERT映射语句



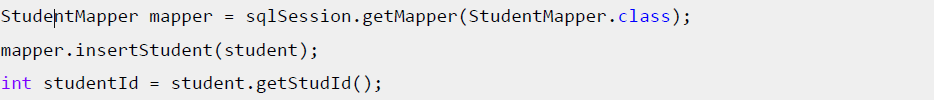
使用了@Insert注解的insertMethod()方法将返回insert语句执行后影响的行数。

【自动生成主键】

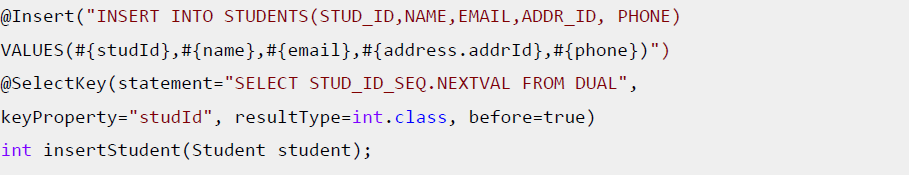
在上一章中我们讨论过**主键列值可以自动生成**。我们可以使用**@Options注解**的**userGeneratedKeys** 和**keyProperty**属性让数据库产生auto\_increment（自增长）列的值，然后将生成的值设置到输入参数对象的属性中。



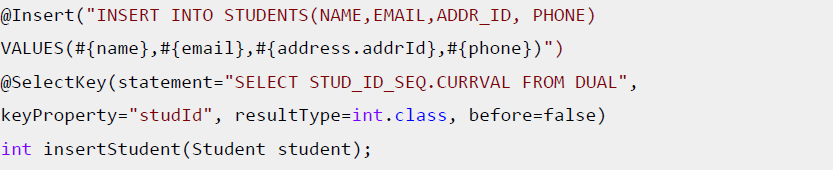
这里STUD\_ID列值将会通过MySQL数据库自动生成。并且生成的值将会被设置到student对象的studId属性中。



有一些数据库如Oracle，并不支持AUTO\_INCREMENT列属性，它使用序列（SEQUENCE）来产生主键的值。我们可以使用**@SelectKey注解来为任意SQL语句来指定主键值**，作为主键列的值。假设我们有一个名为STUD\_ID\_SEQ的序列来生成STUD\_ID主键值：

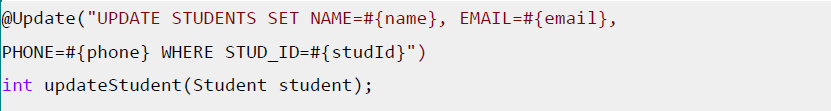


这里我们使用了**@SelectKey**来生成主键值，并且存储到了student对象的studId属性上。由于我们设置了before=true,该语句将会在**执行INSERT语句之前执行**。如果你使用序列作为触发器来设置主键值，我们可以在INSERT语句执行后，从sequence\_name.currval获取数据库产生的主键值。

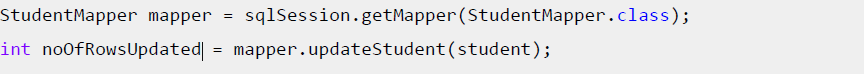


**2）@update**

我们可以使用@Update注解来定义一个UPDATE映射语句，如下所示：

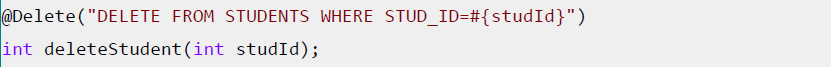


使用了@Update的updateStudent()方法将会返回执行了update语句后影响的行数。



**3）@delete**

我们可以使用@Delete注解来定义一个DELETE映射语句，如下所示：

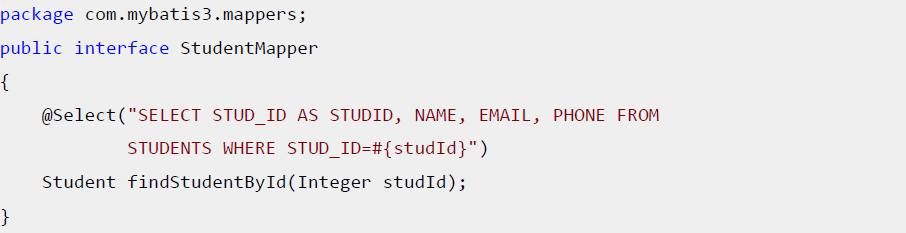


使用了@Delete的deleteStudent()方法将会返回执行了update语句后影响的行数

**4）@select**

我们可以使用@Select注解来定义一个SELECT映射语句。

让我们看一下怎样使用注解配置一个简单的select查询。



为了将列名和Student bean**属性名匹配**，我们为stud\_id起了一个studId的别名。如果**返回了多行结果**，将抛出TooManyResultsException异常。

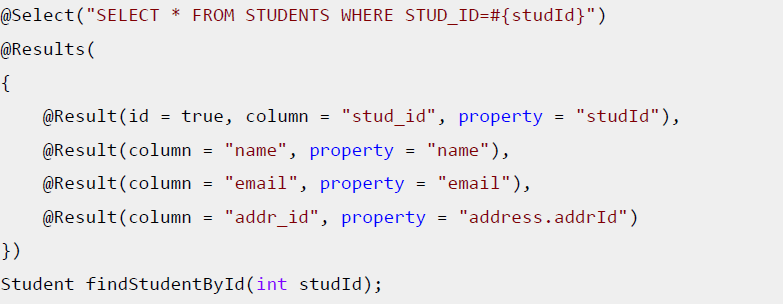
**3、结果映射**

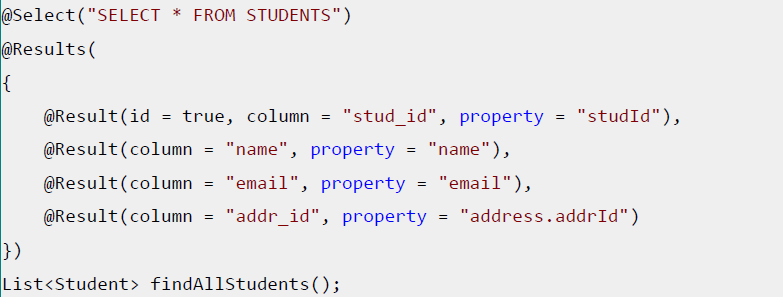
我们可以将查询结果通过别名或者是**@Results注解与JavaBean属性**映射起来。现在让我们看看怎样使用**@Results注解**将指定列于指定JavaBean属性映射器来，执行SELECT查询的：



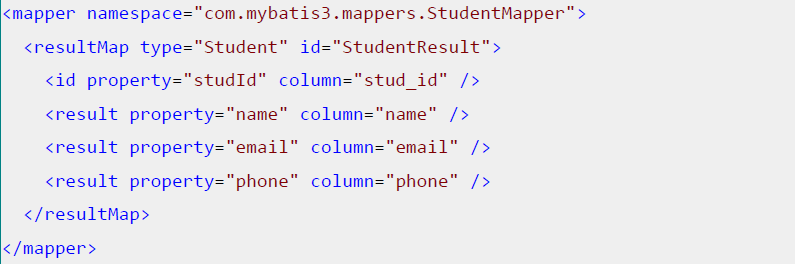
@Results注解和映射器XML配置文件元素<resultMap>想对应。然而，MyBatis3.2.2不能为@Results注解赋予一个ID。所以，不像<resultMap>元素，我们**不应在不同的映射语句中重用@Results声明**。这意味着即使@Results注解完全相同，我们也需要(在不同的映射接口中)重复@Results声明。

例如，看下面的findStudentById()和findAllStudents()方法：

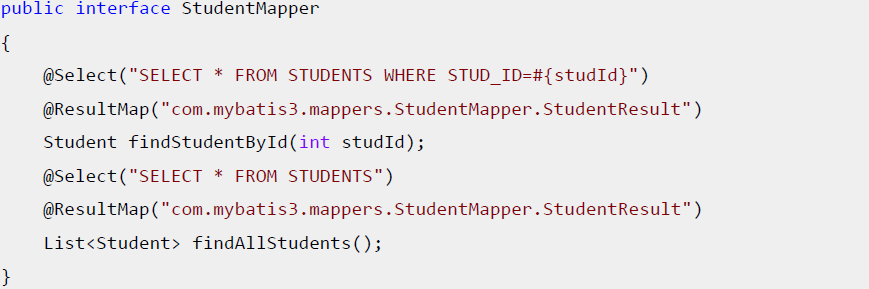




这里两个语句的@Results配置完全相同，但是我必须得重复它。这里有一个解决方法。我们可以创建一个**映射器Mapper配置文件**，然后配置<resultMap>元素，然后使用@ResultMap注解引用此<resultMap>。

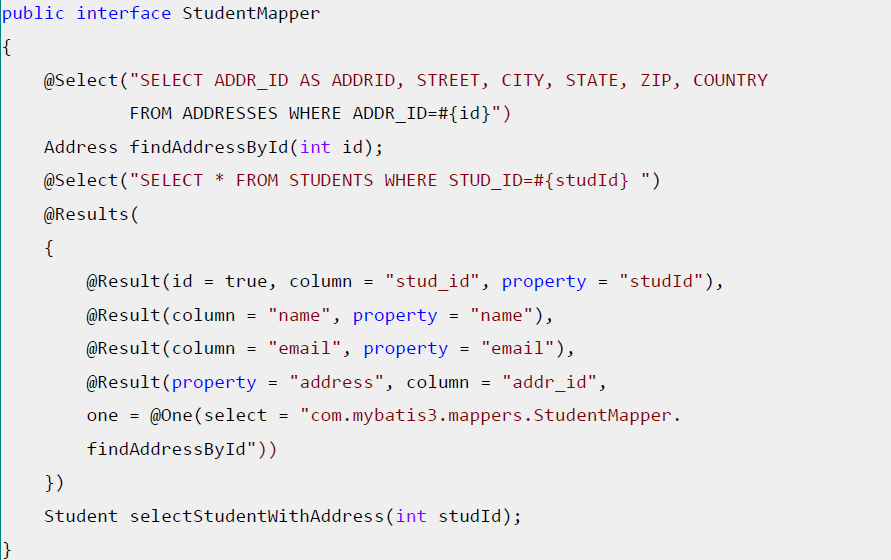


在StudentMapper.java中，使用@ResultMap引用名为StudentResult的resultMap。

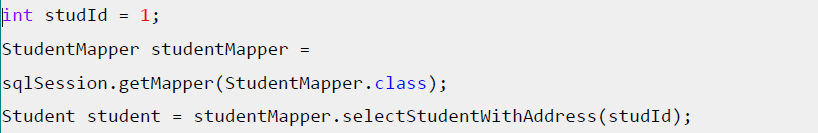


**1）一对一映射**

MyBatis提供了**@One注解来使用嵌套select语句**（Nested-Select）加载一对一关联查询数据。让我们看看怎样使用@One注解获取学生及其地址信息。



这里我们使用了**@One注解的select属性**来指定一个使用了**完全限定名的方法**上，该方法会返回一个Address对象。使用column=”addr\_id”,则**STUEDNTS表中列addr\_id的值将会作为输入参数传递给findAddressById()方法**。如果@One SELECT查询**返回了多行结果**，则会抛出TooManyResultsException异常。

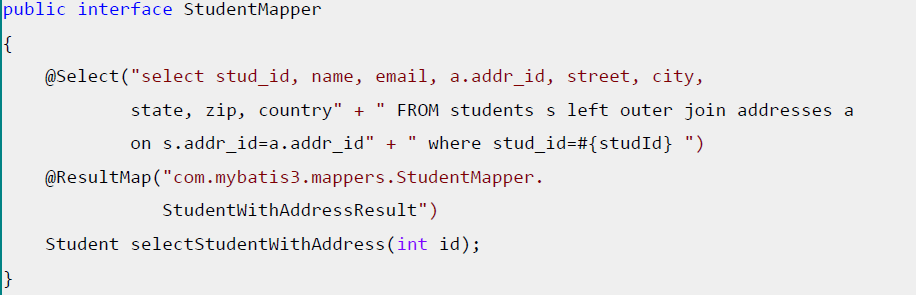




使用XML配置SQL映射器中我们讨论过，我们可以通过基于XML的映射器配置，使用**嵌套**结果**ResultMap来加载一对一关联的查询**。而MyBatis3.2.2版本，并没有对应的注解支持。但是我们可以在映射器Mapper配置文件中配置<resultMap>并且使用@ResultMap注解来引用它。

在StudentMapper.xml中配置<resultMap>，如下所示：

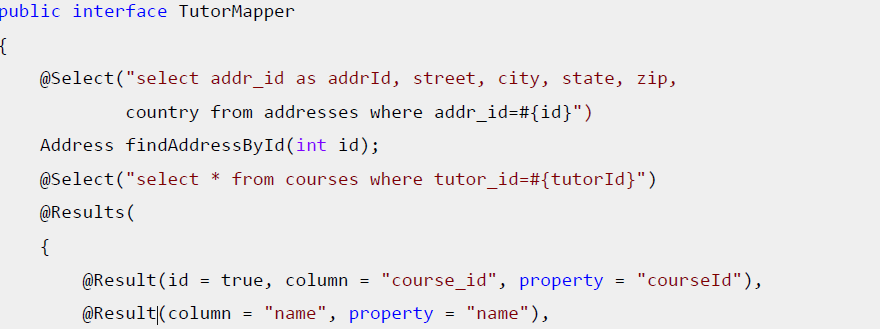




**2）一对多映射**

MyBatis提供了@Many注解，用来使用嵌套Select语句加载一对多关联查询。

现在让我们看一下如何使用@Many注解获取一个讲师及其教授课程列表信息：



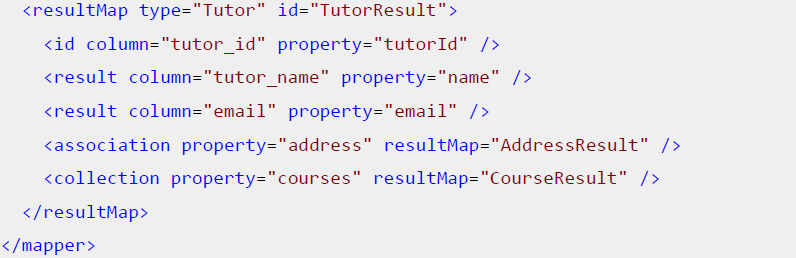


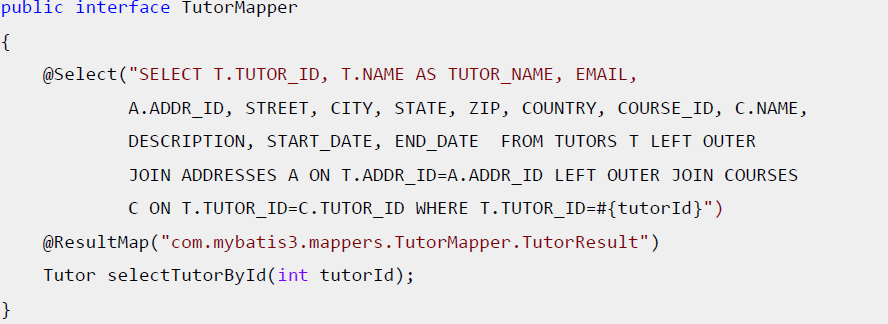
这里我们使用了@Many注解的select属性来指向一个完全限定名称的方法，该方法将返回一个List<Course>对象。使用column=”tutor\_id”，TUTORS表中的tutor\_id列值将会作为输入参数传递给findCoursesByTutorId()方法。

在第三章，使用XML配置SQL映射器中我们讨论过，我们可以通过基于XML的映射器配置，使用嵌套结果**ResultMap来加载一对多关联的查询**。而MyBatis3.2.2版本，并没有对应的注解支持。但是我们可以在映射器Mapper配置文件中配置<resultMap>并且使用@ResultMap注解来引用它。

在TutorMapper.xml中配置<resultMap>,如下所示：





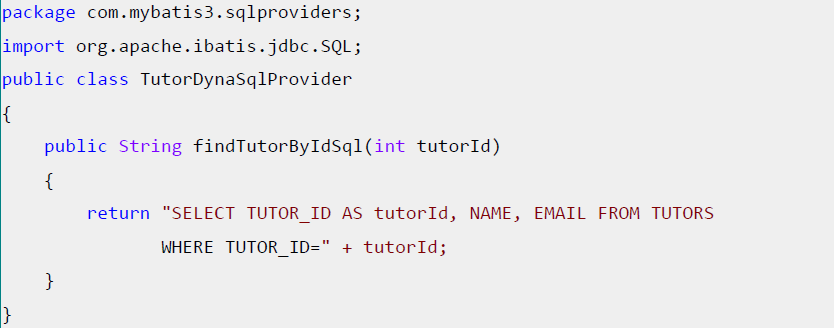


**4、动态SQL**

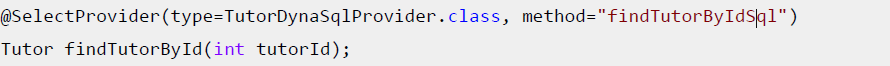
有时候我们需要根据输入条件动态地构建SQL语句。MyBatis提供了各种注解如@InsertProvider、@UpdateProvider、@DeleteProvider和@SelectProvider，来帮助构建动态SQL语句，然后让MyBatis执行这些SQL语句。

**1）@SelectProvider**

现在让我们来看一个使用@SelectProvider注解来创建一个简单的SELECT映射语句的例子。创建一个TutorDynaSqlProvider.java类，以及findTutorByIdSql()方法，如下所示：

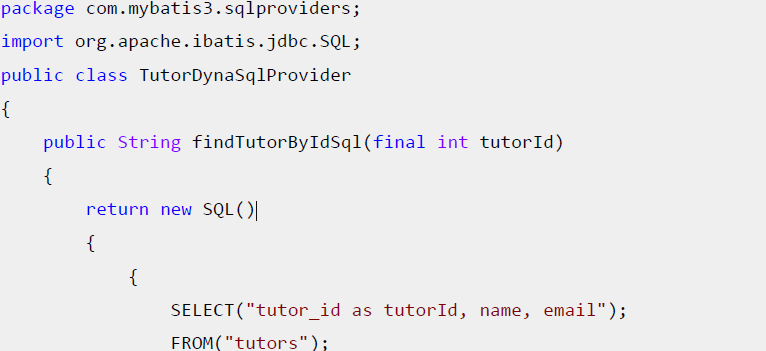


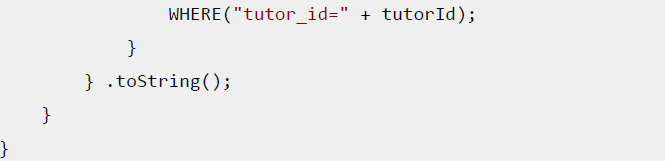
在TutorMapper.java接口中创建一个映射语句，如下：



这里我们使用了@SelectProvider来**指定了一个类**，及**其内部的方法**，用来提供需要执行的SQL语句。但是使用字符串拼接的方法构建SQL语句是非常困难的，并且容易出错。所以MyBaits提供了一个SQL工具类不使用字符串拼接的方式，简化构造动态SQL语句。

现在，让我们看看如何使用org.apache.ibatis.jdbc.SQL工具类来准备相同的SQL语句。





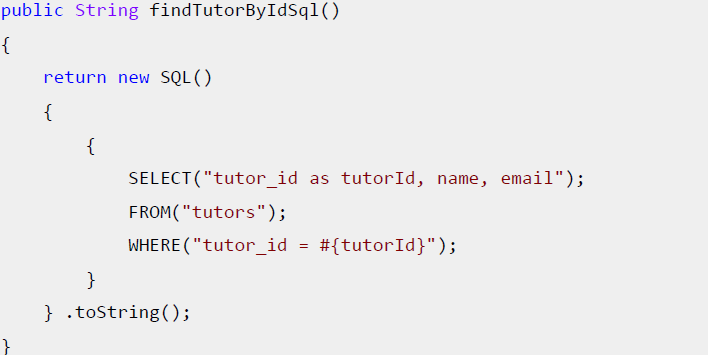
SQL工具类会处理以合适的空格前缀和后缀来构造SQL语句。

动态**SQL provider**方法可以接收以下其中一种参数：

* 无参数
* 和映射器Mapper接口的方法同类型的参数
* java.util.Map

如果SQL语句的准备不取决于输入参数，你可以使用不带参数的SQL Provider方法。

例如：



这里我们没有使用输入参数构造SQL语句，所以它可以是一个无参方法。

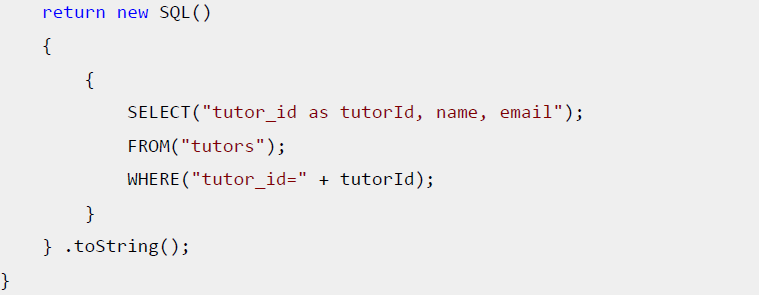
如果映射器Mapper接口方法只有一个参数，那么可以定义SQL Provider方法，它接受一个与Mapper接口方法相同类型的参数。

例如映射器Mapper接口有如下定义：

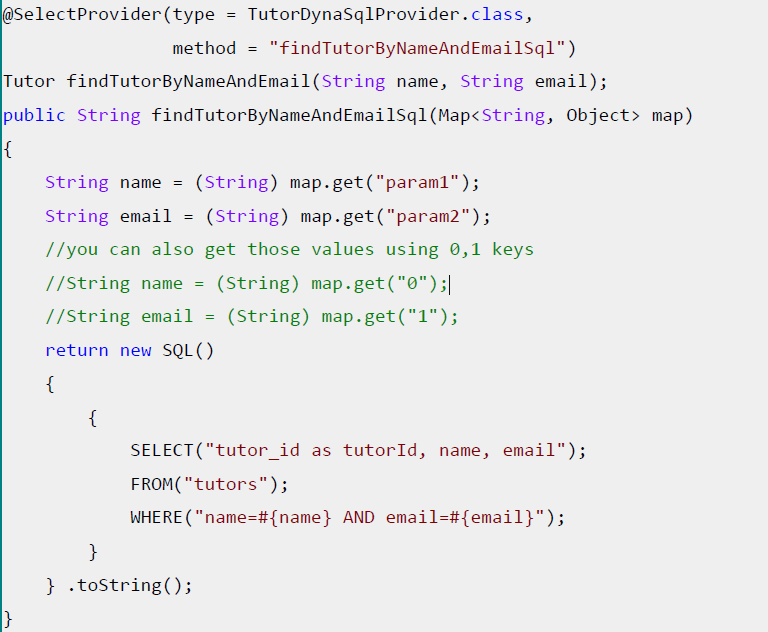


这里findTutorById(int)方法只有一个int类型的参数。我们可以定义findTutorByIdSql(int)方法作为SQL provider方法。

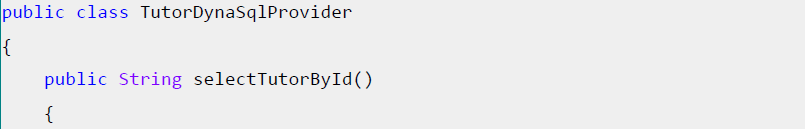


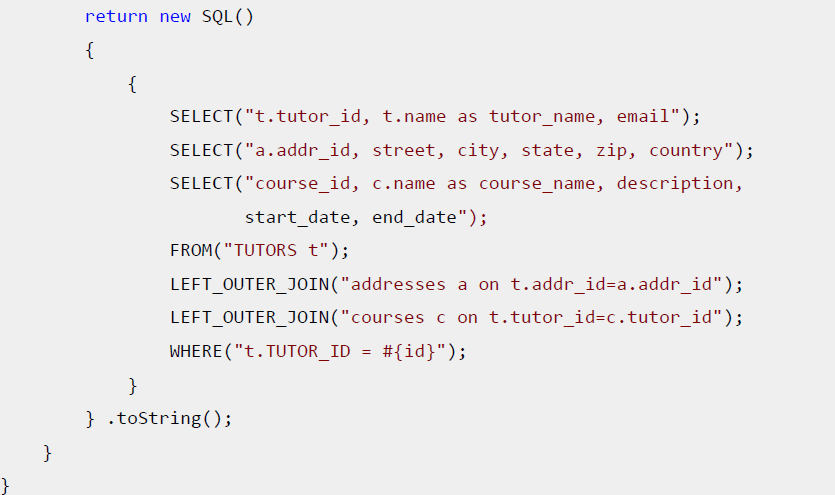


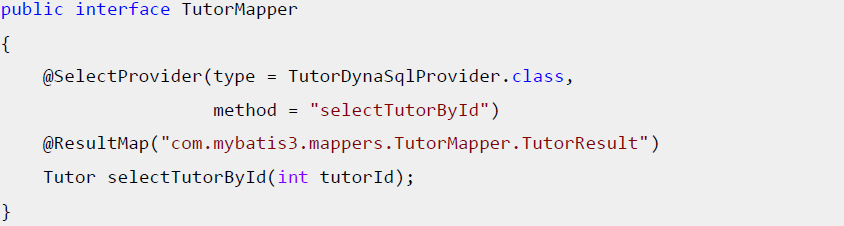
如果**映射器Mapper接口**有多个输入参数，我们可以使用参数类型为java.util.Map的方法作为SQL provider方法。然后映射器Mapper接口方法**所有的输入参数**将会被放到map中，以param1,param2等等作为key，将输入参数按序作为value。你也可以使用0，1，2等作为key值来取的输入参数。



SQL工具类也提供了其他的方法来表示JOINS，ORDER\_BY，GROUP\_BY等等。让我们看一个使用LEFT\_OUTER\_JOIN的例子：







由于没有支持使用内嵌结果ResultMap的一对多关联映射的注解支持，我们可以使用基于XML的<resultMap>配置，然后与@ResultMap映射。

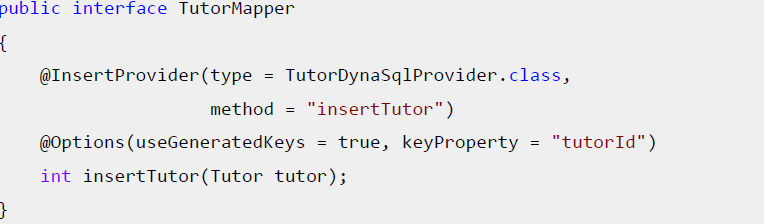




使用了动态的SQL provider，我们可以取得讲师及其地址和课程明细。

**2）@InsertProvider**

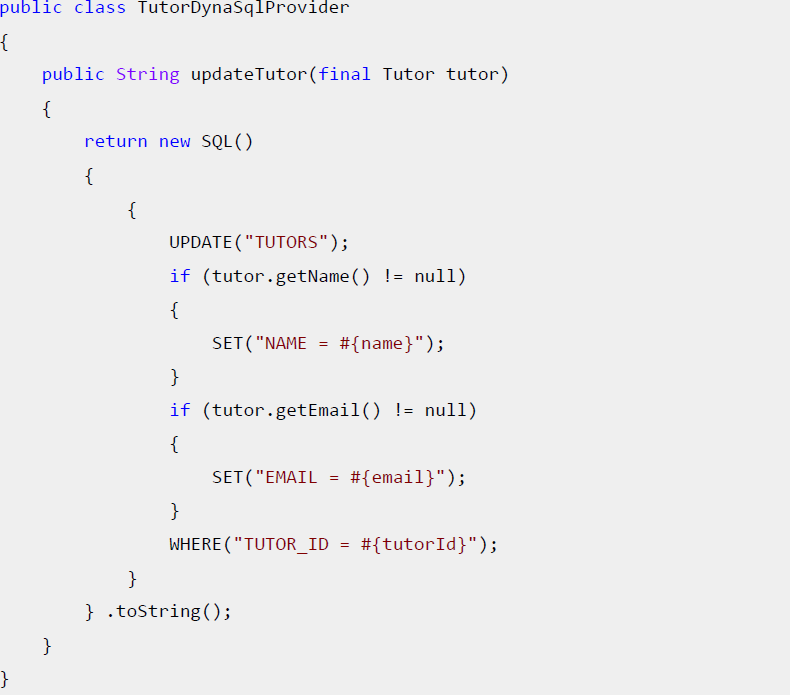
我们可以使用@InsertProvider注解创建动态的INSERT语句，如下所示

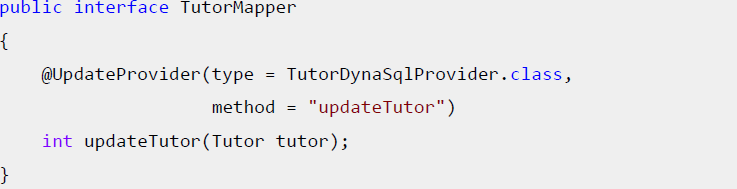




**3）@UpdateProvider**

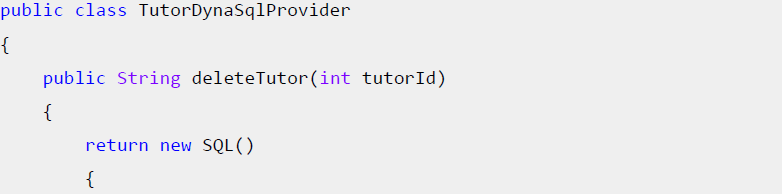
我们可以通过@UpdateProvider注解创建UPDATE语句，如下所示：

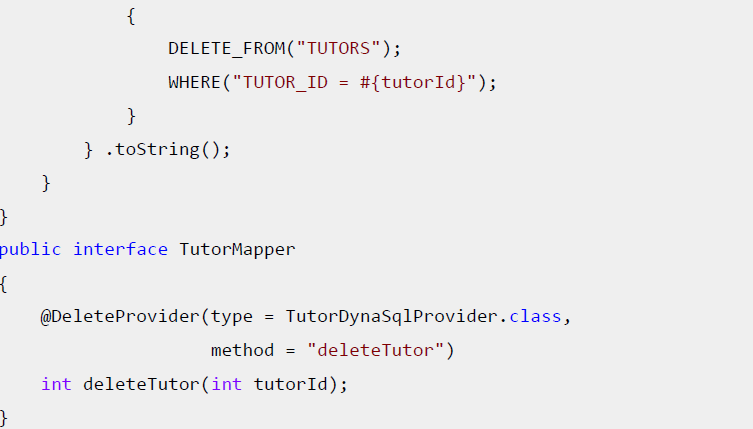




**4）@DeleteProvider**

我们可以使用@DeleteProvider注解创建动态地DELETE语句,如下所示：





**附录：**

**1、select元素属性**

<select id=*"selectPerson"* parameterType=*"int"* parameterMap=*"deprecated"*

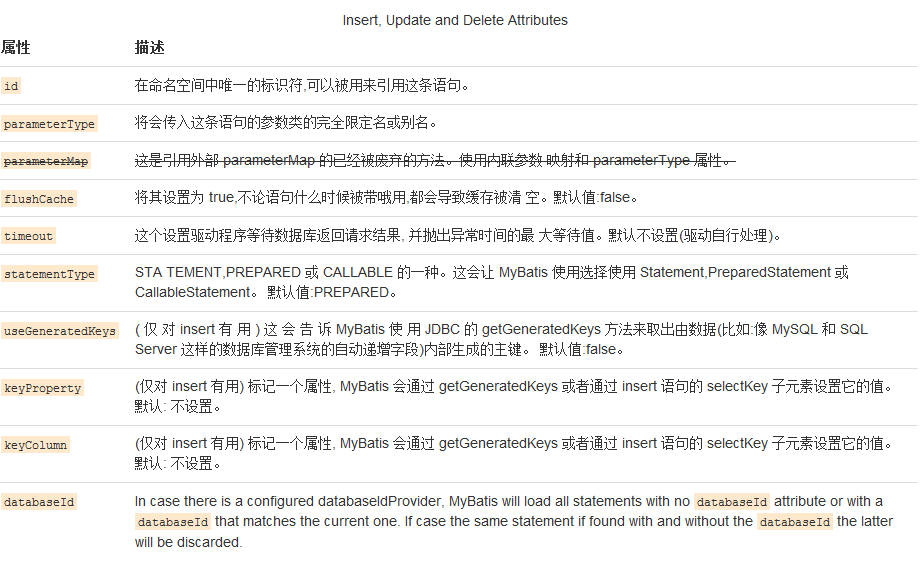
resultType=*"hashmap"* resultMap=*"personResultMap"* flushCache=*"false"* useCache=*"true"* timeout=*"10000"* fetchSize=*"256"* statementType=*"PREPARED"* resultSetType=*"FORWARD\_ONLY"*>



**2、insert、update和delete属性**

数据变更语句 insert,update 和 delete 在它们的实现中非常相似:





**3、selectKey元素**



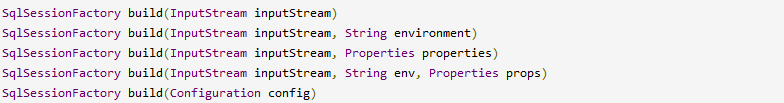


**4、Java API（补充）**

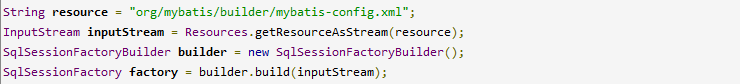
使用 MyBatis 的主要 Java 接口就是 SqlSession。尽管你可以使用这个接口执行命令,获取映射器和管理事务。我们会讨论 SqlSession 本身更多,但是首先我们还是要了解如果获取一个 SqlSession 实例。SqlSessions 是由 SqlSessionFactory 实例创建的。SqlSessionFactory 对象包含创建SqlSession 实例的所有方法。而 SqlSessionFactory 本身是由 SqlSessionFactoryBuilder 创建的,它可以从XML 配置,注解或手动配置 Java 来创建 SqlSessionFactory。

**1）SqlSessionFactoryBuilder**

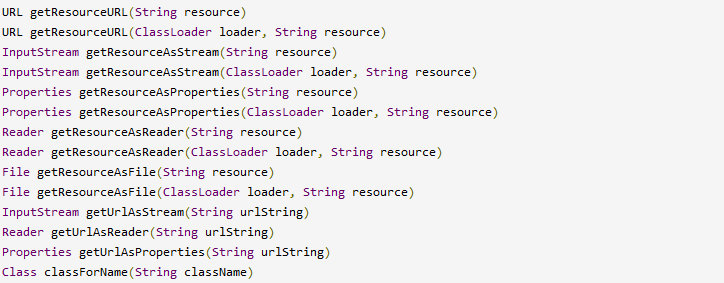
SqlSessionFactoryBuilder 有五个 build()方法,每一种都允许你从不同的资源中创建一个SqlSessionFactory实例。



这里给出一个从 mybatis-config.xml 文件创建 SqlSessionFactory 的示例:



注意这里我们使用了 Resources 工具类,这个类在 org.mybatis.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 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,就会看到一整套有用的方法。这里给出一个简表:



* 最后一个 build 方法使用了一个 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,包含查找和 操作 SQL 映射(不推荐使用,因为应用正接收请求) 。configuration 类有所有配置的开关, 这些你已经了解了,只在 Java API 中露出来。这里有一个简单的示例,如何手动配置 configuration 实例,然后将它传递给 build()方法来创建 SqlSessionFactory。

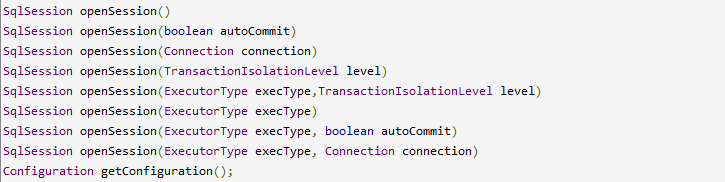


**2）SqlSessionFactory**

SqlSessionFactory 有六个方法可以用来创建 SqlSession 实例。通常来说,如何决定是你选择下面这些方法时:

* **Transaction (事务)**: 你想为 session 使用事务或者使用自动提交(通常意味着很多 数据库和/或 JDBC 驱动没有事务)?
* **Connection (连接)**: 你想 MyBatis 获得来自配置的数据源的连接还是提供你自己
* **Execution (执行)**: 你想 MyBatis 复用预处理语句和/或批量更新语句(包括插入和 删除)?

重载的 openSession()方法签名设置允许你选择这些可选中的任何一个组合：



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)；
* 连接对象会从由活动环境配置的数据源实例中得到；
* 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。

这些方法大都可以自我解释的。开启自动提交, “true” 传递给可选的 autoCommit 参数。提供自定义的连接,传递一个 Connection 实例给 connection 参数。注意没有覆盖同时设置 Connection 和 autoCommit 两者的方法,因为 MyBatis 会使用当前 connection 对象提供的设 置。 MyBatis 为事务隔离级别调用使用一个 Java 枚举包装器, 称为 TransactionIsolationLevel, 否 则 它 们 按 预 期 的 方 式 来 工 作 , 并 有 JDBC 支 持 的 5 级 ( NONE、READ\_UNCOMMITTED、READ\_COMMITTED、REPEA TABLE\_READ、SERIALIZA BLE)

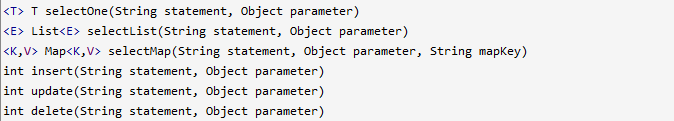
还有一个可能对你来说是新见到的参数,就是 ExecutorType。这个枚举类型定义了 3 个 值:

* 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 在它们中间执行还会标定它们是 必须的,来保证一个简单并易于理解的行为。

**3）SqlSession**

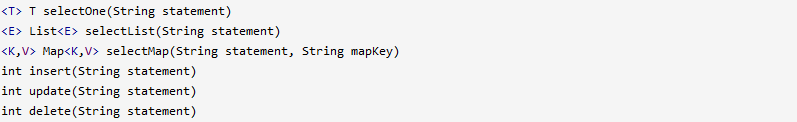
**（1）语句执行方法**

这些方法被用来执行定义在 SQL 映射的XM文件中的SELECT、INSERT、UPDAET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。



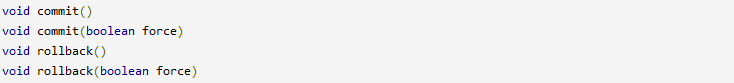
selectOne 和 selectList 的不同仅仅是 selectOne 必须返回一个对象。 如果多余一个, 或者 没有返回 (或返回了 null) 那么就会抛出异常。 , 如果你不知道需要多少对象, 使用 selectList。

如果你想检查一个对象是否存在,那么最好返回统计数(0 或 1) 。因为并不是所有语句都需要参数,这些方法都是有不同重载版本的,它们可以不需要参数对象。



**（2）事务控制方法**

控制事务范围有四个方法。当然，如果你已经选择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 事务管理员,由 Connection 实 例来控制,那么这四个方法就会派上用场:



**（3）使用映射器**

