MyBatis

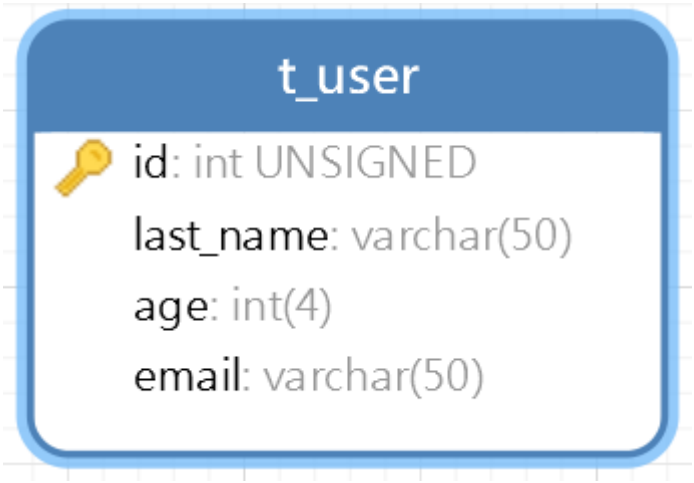
介绍

> MyBatis是一个半自动化的持久化框架，支持定制化SQL、存储过程以及高级映射；
>
> MyBatis避免了几乎所有的JDBC代码和手动设置参数以及获取结果集；
>
> 可以使用XML或注解的形式来配置和映射原生类型、接口和POJO为数据库中的记录。
>
> 更多介绍，参考MyBatis官方文档：http://www.mybatis.org/mybatis-3/index.html
>
> 相关链接：
>
> - MyBatis in Github：https://github.com/mybatis/mybatis-3
> - MyBatis-Spring：https://github.com/mybatis/spring

前期准备

数据库（以MySQL5.7为例）：

为了简单演示，先创建一张表



一个Maven项目，名字就叫demo吧；

包括数据库连接驱动，日志，以及Junit测试依赖：

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.45</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
```

```
    <version>2.11.0</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-api -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.11.0</version>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.5</version>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
</dependency>

<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

开始

首先打开MyBatis的官方文档的入门章节：http://www.mybatis.org/mybatis-3/zh/getting-started.html

参照官方文档，我们可以了解到使用MyBatis需要接下来几步：

1. 添加依赖；
2. 构建SqlSessionFactory：

> 每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为核心的。
>
> SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。
>
> 而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先定制的 Configuration 的实例构建出 SqlSessionFactory 的实例。

3. 获取SqlSession：

> 既然有了 SqlSessionFactory，顾名思义，我们就可以从中获得 SqlSession 的实例了。
>
> SqlSession 完全包含了面向数据库执行 SQL 命令所需的所有方法。
>
> 你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。

4. 获取Mapper接口实例；

下面来开始详细的说明，每一步的具体实现：

Step1、添加依赖

第一步，我们需要引入MyBatis的依赖，而使用Maven的话，我们可以轻松地导入MyBatis的依赖jar包；

进入mvn的官方仓库：https://mvnrepository.com/，搜索`mybatis`，点击相应版本，以`3.4.6`为例，将下列的依赖代码添加到pom.xml文件的即可；

Step2、从XML中构建SqlSessionFactory

由官方文档我们知道，每一个基于MyBatis的应用都是以一个`SqlSessionFactory`实例为核心的，而这个实例是通过`SqlSessionFactoryBuilder#Builder()`去加载全局配置文件(这里我们就称`mybatis-config.xml`)来构建的；

那么首先，我们来简单配置下这个全局配置文件：

mybatis-config.xml

在`src/main/resources`目录里新建一个mybatis-config.xml文件，并将官方文档上的实例的内容粘贴进来；

这里我们可以看到除了文件头有三个标签，分别是`<configuration>`，`<environments>`，`<mappers>`；

其中`configuration`是顶层标签，表示配置，所有的配置都要在这个标签内部去设置；

而`environments`是环境配置标签，而它内部可以包含多个`environment`标签配置，以此来去快速适应多种环境的切换（通过设置default属性的值），

而每一个`environment`里是对当前环境的事务管理器（transactionManager），数据源（dataSource）进行设置；

最后`mappers`是映射器路径标签，它指定了MyBatis要去哪里查找已经映射的SQL语句；

```
<?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>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username" value="${username}"/>
        <property name="password" value="${password}"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="org/mybatis/example/BlogMapper.xml"/>
  </mappers>
</configuration>
```

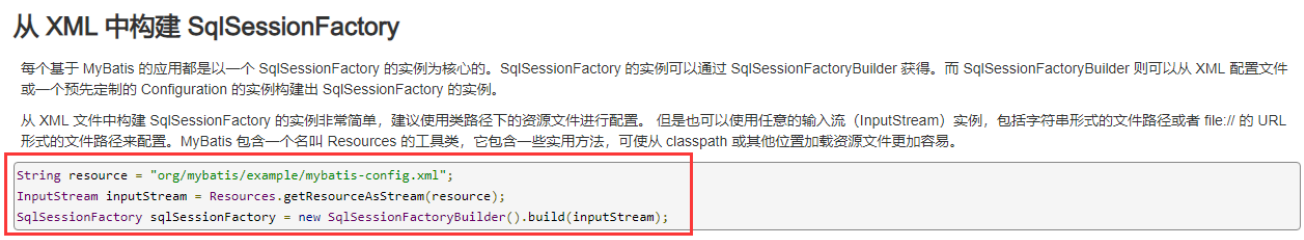
介绍完了之后，我们需要做的是，将数据源修改成我们自己的数据库（这里我的数据库名为`mybatis`）：

```
<dataSource type="POOLED">
    <property name="driver" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mybatis?useSSL=false"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</dataSource>
```

这一步暂时只对数据源进行设置，后续的配置可以参照官方文档，本文在也会慢慢补充说明；

构建SqlSessionFactory

首先我们先创建一个测试类，并创建一个测试方法，然后复制官方文档的下面一段




```
public class UserTest {

    @Test
```

```
    public void firstTest() throws IOException{
        //全局配置文件
        String resource = "mybatis-config.xml";

        //注意导包import org.apache.ibatis.io.Resources;
        //这里使用的是mybatis提供的Resources工具类，可以很方便的加载classpath下的资源文件，并转换成输入流的形式
        InputStream inputStream = Resources.getResourceAsStream(resource);

        //SqlSessionFactoryBuilder对象通过build方法加载全局配置文件，来构造出SqlSessionFactory实例
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

    }
}
```

配置好之后，我们已经获得到SqlSessionFactory了；

Step3、从SqlSessionFactory中获取 SqlSession

有了SqlSessionFactory之后，我们就可以从中获取SqlSession了；

xxxMapper.xml

在此之前，我们需要先创建一个XML映射文件，就像官方文档所说：

> MyBatis 的真正强大在于它的映射语句，这是它的魔力所在。由于它的异常强大，映射器的 XML 文件就显得相对简单。
>
> 如果拿它跟具有相同功能的 JDBC 代码进行对比，你会立即发现省掉了将近 95% 的代码。

这个文件，就是用来写我们映射语句的；

针对这个`User`实体，我们首先要创建一个`UserMapper.xml`配置文件；（其他xxx实体也通常写成xxxMapper.xml）；

在`src/main/resources`目录下新建一个mappers文件夹，然后创建`UserMapper.xml`：

```
<?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">
<mapper namespace="demo.dao.UserMapper">
    <!--
        id：为此映射语句的唯一标识，一般为接口方法名
        resultType：指定返回结果类型，这里写的是User实体的全限定类名
        #{}：代表取出传入参数的值
    -->
    <select id="getUserById" resultType="demo.bean.User">
        select * from t_user where id = #{id}
    </select>
</mapper>
```

而这个文件头，在官方文档上也有示例：

探究已映射的 SQL 语句

现在你可能很想知道 SqlSession 和 Mapper 到底执行了什么操作，但 SQL 语句映射是个相当大的话题，可能会占去文档的大部分篇幅。不过为了让你能够了解个大概，这里会给出几个例子。

在上面提到的例子中，一个语句既可以通过 XML 定义，也可以通过注解定义。我们先看看 XML 定义语句的方式，事实上 MyBatis 提供的全部特性都可以利用基于 XML 的映射语言来实现，这使得 MyBatis 在过去的数年间得以流行。如果你以前用过 MyBatis，你应该对这个概念比较熟悉。不过自那以后，XML 的配置也改进了许多，我们稍后还会再提到。这里给出一个基于 XML 映射语句的示例，它应该可以满足上述示例中 SqlSession 的调用。

```
<?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">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

其中`namespace`是指定唯一命名空间，通常我们使用dao接口的全限定类名（这里这个接口还未创建，但可以先写上）；

修改全局配置文件

创建好映射文件后，我们需要在全局配置文件中添加配置，告诉MyBatis去哪里找我们写的这个映射语句；

注意：为了方便返回结果封装，我们首先开启驼峰命名规则，在全局配置文件的`environments`标签前面添加如下配置；

```
<!-- 全局设置标签 -->
<settings>
    <!--
        开启驼峰命名规则
        DB:last_name -> Java:lastName
    -->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

然后再添加我们的映射文件路径，在`mappers`添加一行：

```
<mappers>
    <mapper resource="mappers/UserMapper.xml"/>
</mappers>
```

获取Sqlseesion

前面的配置做好之后，参考官方文档

从 SqlSessionFactory 中获取 SqlSession

既然有了 SqlSessionFactory，顾名思义，我们就可以从中获得 SqlSession 的实例了。SqlSession 完全包含了面向数据库执行 SQL 命令所需的所有方法。你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。例如：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  Blog blog = (Blog) session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);
} finally {
  session.close();
}
```

在之前的代码里后面接着写：

```
/*
```

```
 * 通过openSession方法获得SqlSession
 * openSession()支持重载，我们可以通过传参来设置一些对事务的设置；
 * 比如：传入true，表示激活autoCommit属性，事务将自动提交；
 * SqlSession session = sqlSessionFactory.openSession(true);
 */
SqlSession session = sqlSessionFactory.openSession();
try {
    //通过SqlSession的方法来执行指定的SQL映射语句，
    //第一个参数为：${mapper文件的namespace}.${select标签的id}，指定映射语句
    //第二个参数为：查询条件参数
    User user = session.selectOne("mybatis.dao.UserMapper.getUserById", 2);
    System.out.println(user);
} finally {
    session.close();
}
```

执行测试，查看结果：

```
D:\DevelopIDE\Java\jdk1.8.0_181\bin\java.exe ...
log4j:WARN No appenders could be found for logger (org.apache.ibatis.logging.LogFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
User [id=2, lastName=Jerry, age=20, email=jerry@eteam.fun]

Process finished with exit code 0
```

到此，我们完成了第一个查询。

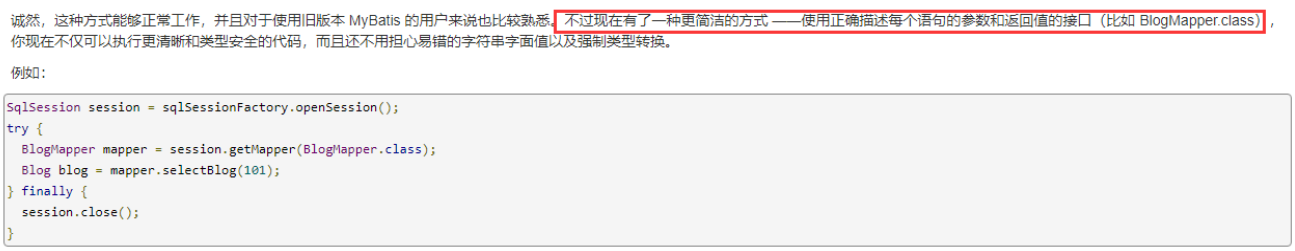
注意：上面的警告是环境下缺少日志配置文件，在`src/main/resources`目录下创建log4j的配置文件

log4j.properties

```
log4j.rootLogger=DEBUG,stdout
log4j.logger.com.ibatis=DEBUG
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%-5p %d{yyyy-MM-dd HH:mm:ss} %m%n
```

Step4、获取Mapper接口实例

我们在前一步已经完成了查询，但是官方文档又说，


诚然，这种方式能够正常工作，并且对于使用旧版本 MyBatis 的用户来说也比较熟悉。不过现在有了一种更简洁的方式 —— 使用正确描述每个语句的参数和返回值的接口（比如 BlogMapper.class），你现在不仅可以执行更清晰和类型安全的代码，而且还不用担心易错的字符串字面值以及强制类型转换。

例如：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  Blog blog = mapper.selectBlog(101);
} finally {
  session.close();
}
```

既然新的比旧的优化的好，我们就来看看如何来做；

xxxMapper.java

首先，我们需要创建一个UserMapper.java接口文件（注意到，我们通常会将接口文件和对应的映射文件同名，在这里不是必须的，但是在后期整合Spring的时候是一般需要的，所以在此先养成习惯）

UserMapper.java

```
public interface UserMapper {
    User getUserById(Integer id);
}

```

获取Mapper接口实例

我们并不需要去实现上述UserMapper接口，因为接下来将要通过MyBatis的SqlSession去获取一个接口实例；

将测试方法修改为

```
SqlSession session = sqlSessionFactory.openSession();
try {
    //1
    UserMapper mapper = session.getMapper(UserMapper.class);
    User user = mapper.getUserById(2);
    //2
    System.out.println(user);
} finally {
    session.close();
}
```

执行测试查询

```
DEBUG 2019-04-22 22:50:49 Opening JDBC Connection
DEBUG 2019-04-22 22:50:50 Created connection 1004095028.
DEBUG 2019-04-22 22:50:50 Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@3bd94634]
DEBUG 2019-04-22 22:50:50 ==>  Preparing: select * from t_user where id = ?
DEBUG 2019-04-22 22:50:50 ==> Parameters: 2(Integer)
DEBUG 2019-04-22 22:50:50 <==      Total: 1
User [id=2, lastName=Jerry, age=20, email=jerry@eteam.fun]
DEBUG 2019-04-22 22:50:50 Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@3bd94634]
DEBUG 2019-04-22 22:50:50 Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@3bd94634]
DEBUG 2019-04-22 22:50:50 Returned connection 1004095028 to pool.
```

配置完日志后，可以看到执行的语句，参数，以及返回结果等；

源码分析1：获取Mapper代理对象

在构建SqlSesionFactory时，它的Configuration类对象下其中的两个成员对象分别保存了接口对象，和已经映射了的SQL语句

```
//Mapper接口注册类
//在这个类中，有一个成员knowMappers保存了所有扫描到的Mapper接口的class对象，以及对应的代理工厂；
//成员定义：private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap<Class<?>, MapperProxyFactory<?>>();
protected final MapperRegistry mapperRegistry = new MapperRegistry(this);

//在mappedStatements中则保存了所有已经映射的SQL语句，KEY：方法的全限定名/方法名 value为MappedStatement对象（保存了所映射语句的相关设置，包括参数类型，返回值类型等等...）
protected final Map<String, MappedStatement> mappedStatements = new StrictMap<MappedStatement>("Mapped Statements collection");
```

然后通过`SqlSession#getMapper(Class<T> type)`来获得相应的映射器实例（这个实例是一个代理对象）；

其中`SqlSession#getMapper(Class<T> type)`最终是通过`MapperRegistry#getMapper(Class<T> type, SqlSession sqlSession)`方法拿到；

部分源码+注释：

```
/**
 * @author Clinton Begin
 * @author Eduardo Macarron
 * @author Lasse Voss
 */
public class MapperRegistry {

  private final Configuration config;
  /*
   * knownMappers是所有扫描到的Mapper接口，以及对应的代理工厂；
   */
  private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap<Class<?>, MapperProxyFactory<?>>();

  public MapperRegistry(Configuration config) {
    this.config = config;
  }

  @SuppressWarnings("unchecked")
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    /*
     * 根据class对象，get到对应的代理工厂
     */
    final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>) knownMappers.get(type);
    if (mapperProxyFactory == null) {
      throw new BindingException("Type " + type + " is not known to the MapperRegistry.");
    }
    try {
      /*
       * 通过mapper代理工厂返回一个映射器实例（这个实例也是一个代理对象），具体实现为：
       *   public T newInstance(SqlSession sqlSession) {
       *       final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);
```

```
         *        return newInstance(mapperProxy);
         *   }
         */
        return mapperProxyFactory.newInstance(sqlSession);
      } catch (Exception e) {
        throw new BindingException("Error getting mapper instance. Cause: " + e, e);
      }
    }
    /*
     * 省略其他方法
     */
}
```

源码分析2：通过动态代理，调用底层实现

拿到映射器实例实质是一个代理对象，当我们调用之前定义的方法，会先进入到`MapperProxy#invoke()`方法中，通过动态代理的方式，调用底层实现；

`MapperProxy.java`部分源码：

```
/**
 * @author Clinton Begin
 * @author Eduardo Macarron
 */
public class MapperProxy<T> implements InvocationHandler, Serializable {

  private static final long serialVersionUID = -6424540398559729838L;
  private final SqlSession sqlSession;
  private final Class<T> mapperInterface;
  private final Map<Method, MapperMethod> methodCache;

  public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {
    this.sqlSession = sqlSession;
    this.mapperInterface = mapperInterface;
    this.methodCache = methodCache;
  }

  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    try {
      //如果当前方法是在`Object`类中声明，那么直接放行；
      if (Object.class.equals(method.getDeclaringClass())) {
        return method.invoke(this, args);
      } else if (isDefaultMethod(method)) {
      //放行默认方法比如toString等
        return invokeDefaultMethod(proxy, method, args);
      }
    } catch (Throwable t) {
      throw ExceptionUtil.unwrapThrowable(t);
    }
    //否则，将`Method`封装成为一个`MapperMethod`；
```

```
    final MapperMethod mapperMethod = cachedMapperMethod(method);

    return mapperMethod.execute(sqlSession, args);
  }
   /*
   * 省略其他方法
   */
}
```

1）封装MapperMethod

```
private MapperMethod cachedMapperMethod(Method method) {
    //如果在methodCache中存在此方法，那么直接取出来用
    MapperMethod mapperMethod = methodCache.get(method);
    if (mapperMethod == null) {
      //如果不存在，那么就新建一个此方法，并把它放到methodCache中去
      mapperMethod = new MapperMethod(mapperInterface, method,
sqlSession.getConfiguration());
      methodCache.put(method, mapperMethod);
    }
    return mapperMethod;
  }
```

而在`MapperMethod`中，分别封装了两个内部类对象，分别是封装SQL类型信息，和参数属性信息，以及返回值和参数解析器信息；

`SqlCommand`和`MethodSignature`；

SqlCommand

```
//SqlCommand构造方法，主要封装了SQL的标签信息
public SqlCommand(Configuration configuration, Class<?> mapperInterface, Method
method) {
      final String methodName = method.getName();
      final Class<?> declaringClass = method.getDeclaringClass();
     //通过resolveMappedStatement进行解析，获得已经映射的语句对象
      MappedStatement ms = resolveMappedStatement(mapperInterface, methodName,
declaringClass,
          configuration);
      if (ms == null) {
        if (method.getAnnotation(Flush.class) != null) {
          name = null;
          type = SqlCommandType.FLUSH;
        } else {
          throw new BindingException("Invalid bound statement (not found): "
              + mapperInterface.getName() + "." + methodName);
        }
      } else {
        //name即为唯一标识（全限定名）
        name = ms.getId();
        //设置type类型（SELECT\UPDATE\...）
        type = ms.getSqlCommandType();
```

```
        if (type == SqlCommandType.UNKNOWN) {
          throw new BindingException("Unknown execution method for: " + name);
        }
      }
    }

//resolveMappedStatement解析方法
private MappedStatement resolveMappedStatement(Class<?> mapperInterface, String
methodName,
        Class<?> declaringClass, Configuration configuration) {
    //拼接全限定类名
      String statementId = mapperInterface.getName() + "." + methodName;
      //通过方法全限定名查找在Configuration中的mappedStatement
      if (configuration.hasStatement(statementId)) {
        return configuration.getMappedStatement(statementId);
      } else if (mapperInterface.equals(declaringClass)) {
        return null;
      }
      for (Class<?> superInterface : mapperInterface.getInterfaces()) {
        if (declaringClass.isAssignableFrom(superInterface)) {
          MappedStatement ms = resolveMappedStatement(superInterface, methodName,
              declaringClass, configuration);
          if (ms != null) {
            return ms;
          }
        }
      }
      return null;
    }
```

MethodSignature

```
//主要封装了该方法的标签参数信息，返回值信息以及参数解析器
public MethodSignature(Configuration configuration, Class<?> mapperInterface,
Method method) {
      Type resolvedReturnType = TypeParameterResolver.resolveReturnType(method,
mapperInterface);
      if (resolvedReturnType instanceof Class<?>) {
        this.returnType = (Class<?>) resolvedReturnType;
      } else if (resolvedReturnType instanceof ParameterizedType) {
        this.returnType = (Class<?>) ((ParameterizedType)
resolvedReturnType).getRawType();
      } else {
        this.returnType = method.getReturnType();
      }
      this.returnsVoid = void.class.equals(this.returnType);
      this.returnsMany =
configuration.getObjectFactory().isCollection(this.returnType) ||
this.returnType.isArray();
      this.returnsCursor = Cursor.class.equals(this.returnType);
      this.mapKey = getMapKey(method);
      this.returnsMap = this.mapKey != null;
      this.rowBoundsIndex = getUniqueParamIndex(method, RowBounds.class);
```

```
      this.resultHandlerIndex = getUniqueParamIndex(method, ResultHandler.class);
      this.paramNameResolver = new ParamNameResolver(configuration, method);
    }
```

2）参数处理

注意到，在通过`SqlSession`调用查询方法之前，都通过参数解析器`ParamNameResolver#getNamedParams()`对传入的参数做了处理：

```
public class ParamNameResolver {

  private static final String GENERIC_NAME_PREFIX = "param";

  /**
   * <p>
   * The key is the index and the value is the name of the parameter.<br />
   * The name is obtained from {@link Param} if specified. When {@link Param} is not specified,
   * the parameter index is used. Note that this index could be different from the actual index
   * when the method has special parameters (i.e. {@link RowBounds} or {@link ResultHandler}).
   * </p>
   * <ul>
   * <li>aMethod(@Param("M") int a, @Param("N") int b) -&gt; {{0, "M"}, {1, "N"}}</li>
   * <li>aMethod(int a, int b) -&gt; {{0, "0"}, {1, "1"}}</li>
   * <li>aMethod(int a, RowBounds rb, int b) -&gt; {{0, "0"}, {2, "1"}}</li>
   * </ul>
   */
  private final SortedMap<Integer, String> names;

  private boolean hasParamAnnotation;

  public ParamNameResolver(Configuration config, Method method) {
    final Class<?>[] paramTypes = method.getParameterTypes();
    final Annotation[][] paramAnnotations = method.getParameterAnnotations();
    final SortedMap<Integer, String> map = new TreeMap<Integer, String>();
    int paramCount = paramAnnotations.length;
    // get names from @Param annotations
    for (int paramIndex = 0; paramIndex < paramCount; paramIndex++) {
      if (isSpecialParameter(paramTypes[paramIndex])) {
        // skip special parameters
        continue;
      }
      String name = null;
      for (Annotation annotation : paramAnnotations[paramIndex]) {
        if (annotation instanceof Param) {
          hasParamAnnotation = true;
          name = ((Param) annotation).value();
          break;
        }
      }
```

```
41        if (name == null) {
42          // @Param was not specified.
43          if (config.isUseActualParamName()) {
44            name = getActualParamName(method, paramIndex);
45          }
46          if (name == null) {
47            // use the parameter index as the name ("0", "1", ...)
48            // gcode issue #71
49            name = String.valueOf(map.size());
50          }
51        }
52        map.put(paramIndex, name);
53      }
54      names = Collections.unmodifiableSortedMap(map);
55    }
56
57    private String getActualParamName(Method method, int paramIndex) {
58      if (Jdk.parameterExists) {
59        return ParamNameUtil.getParamNames(method).get(paramIndex);
60      }
61      return null;
62    }
63
64    private static boolean isSpecialParameter(Class<?> clazz) {
65      return RowBounds.class.isAssignableFrom(clazz) ||
ResultHandler.class.isAssignableFrom(clazz);
66    }
67
68    /**
69     * Returns parameter names referenced by SQL providers.
70     */
71    public String[] getNames() {
72      return names.values().toArray(new String[0]);
73    }
74
75    /**
76     * <p>
77     * A single non-special parameter is returned without a name.<br />
78     * Multiple parameters are named using the naming rule.<br />
79     * In addition to the default names, this method also adds the generic names
(param1, param2,
80     * ...).
81     * </p>
82     */
83    public Object getNamedParams(Object[] args) {
84      final int paramCount = names.size();
85      if (args == null || paramCount == 0) {
86        return null;
87      } else if (!hasParamAnnotation && paramCount == 1) {
88        return args[names.firstKey()];
89      } else {
90        final Map<String, Object> param = new ParamMap<Object>();
91        int i = 0;
```

```
      for (Map.Entry<Integer, String> entry : names.entrySet()) {
        param.put(entry.getValue(), args[entry.getKey()]);
        // add generic param names (param1, param2, ...)
        final String genericParamName = GENERIC_NAME_PREFIX + String.valueOf(i +
1);
        // ensure not to overwrite parameter named with @Param
        if (!names.containsValue(genericParamName)) {
          param.put(genericParamName, args[entry.getKey()]);
        }
        i++;
      }
      return param;
    }
  }
}
```

根据参数解析器我们可以知道，当

- 传入一个普通类型的参数时：mybatis不作任何处理，直接返回；#{任意名称}
- 传入一个带有 `@Param` 注解的参数时：参数解析器将参数封装成map，key值为注解指定的值或者为 `param1`；
- 传入多个普通类型的参数时：参数解析器将多个参数封装成map，key值为 `param1`、`param2`...`paramN`，其中也可以使用对应的索引值 `0、1...` 替换（在后期版本改成了使用arg0、arg1...）；
- 传入多个其中带有1个或多个 `@Param` 注解的参数时：参数解析器将多个参数封装成map，key值为注解指定的值或者为 `paramN`；
- 传入Map时：key值为map参数的key值；
- 传入Collecation时：可以使用 `#{collection[N]}` 来取值，其中 `List` 类型还可以使用 `#{list[N]}`，`Array` 类型还可以使用 `#{array[N]}`；

```
private Object wrapCollection(final Object object) {
    if (object instanceof Collection) {
        StrictMap<Object> map = new StrictMap<Object>();
        map.put("collection", object);
        if (object instanceof List) {
            map.put("list", object);
        }
        return map;
    } else if (object != null && object.getClass().isArray()) {
        StrictMap<Object> map = new StrictMap<Object>();
        map.put("array", object);
        return map;
    }
    return object;
}
```

3）execute调用实现

通过封装的 `mapperMethod` 调用 `execute` 方法；

```java
  /**
   * @author Clinton Begin
   * @author Eduardo Macarron
   * @author Lasse Voss
   */
  public class MapperMethod {

    private final SqlCommand command;
    private final MethodSignature method;

    public MapperMethod(Class<?> mapperInterface, Method method, Configuration
config) {
      this.command = new SqlCommand(config, mapperInterface, method);
      this.method = new MethodSignature(config, mapperInterface, method);
    }

    public Object execute(SqlSession sqlSession, Object[] args) {
      Object result;
      //根据标签类型，调用不同的接口方法
      switch (command.getType()) {
        case INSERT: {
        Object param = method.convertArgsToSqlCommandParam(args);
          result = rowCountResult(sqlSession.insert(command.getName(), param));
          break;
        }
        case UPDATE: {
          Object param = method.convertArgsToSqlCommandParam(args);
          result = rowCountResult(sqlSession.update(command.getName(), param));
          break;
        }
        case DELETE: {
          Object param = method.convertArgsToSqlCommandParam(args);
          result = rowCountResult(sqlSession.delete(command.getName(), param));
          break;
        }
        //聚焦SELECT方法
        case SELECT:
          if (method.returnsVoid() && method.hasResultHandler()) {
            executeWithResultHandler(sqlSession, args);
            result = null;
          } else if (method.returnsMany()) {
            result = executeForMany(sqlSession, args);
          } else if (method.returnsMap()) {
            result = executeForMap(sqlSession, args);
          } else if (method.returnsCursor()) {
            result = executeForCursor(sqlSession, args);
          } else {
            Object param = method.convertArgsToSqlCommandParam(args);
            //可以看到,在底层也是调用了我们之前使用的selectOne
            result = sqlSession.selectOne(command.getName(), param);
          }
          break;
        case FLUSH:
```

```
          result = sqlSession.flushStatements();
          break;
        default:
          throw new BindingException("Unknown execution method for: " +
command.getName());
      }
      if (result == null && method.getReturnType().isPrimitive() &&
!method.returnsVoid()) {
        throw new BindingException("Mapper method '" + command.getName()
            + " attempted to return null from a method with a primitive return type
(" + method.getReturnType() + ").");
      }
      return result;
    }
```

总结

通过进行DEBUG调试，对源码初步分析，可以知道MyBatis是怎么实现将接口方法，和SQL映射语句进行绑定来实现SQL操作；

关键几步，

首先在创建SqlSessionFactory时，就根据全局配置去扫描并记录了所有已经映射的SQL语句（在configuration中的mappedStatements对象中，这是一个Map，key值为接口方法名，value值为对应MappedStatement对象）；

然后，通过传入的接口，将对应的接口方法与已经映射的SQL分别进行绑定，并从mapper代理工厂中返回一个mapper代理对象（我们拿到的实例）；

最后，我们通过这个代理对象实例来调用它的方法，mybatis在进行方法的检查，重新封装，以及参数处理之后，走到了我们最开始直接调用SqlSession定义的一系列接口方法的这一步；

在探究源码的过程中，也可以看到mybatis对异常的处理语句，比如参数绑定异常，接口绑定异常，以及返回值异常，这些都可能会是将来学习初期的经常会遇到的错误；

其他介绍

一、MyBatis的系统配置还有很多常用的选项，

1. 比如在配置数据源的时候我们通常习惯于将数据库的连接信息单独写在一个properties文件中，便于修改维护，那么就可以使用`<properties>`标签；
2. 我们可以配置`<typeAliases>`标签，为我们的实体类起别名，省去冗长的全限定类名；
3. 在`<settings>`标签里，我们可以配置许多设置项，这些设置项会影响到MyBatis整个系统的运行，需要谨慎；

接下来给个示例配置：

```
<?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>

```

```xml
    <!-- 指定加载properties配置文件 -->
    <properties resource="dbConfig.properties"/>

    <!-- 全局设置标签 -->
    <settings>
        <!-- 开启驼峰命名 -->
        <setting name="mapUnderscoreToCamelCase" value="true"/>
        <!-- 将NULL值所映射的jdbcType设置为NULL，默认是OTHER -->
        <setting name="jdbcTypeForNull" value="NULL"/>
    </settings>

    <!--
        别名处理器 别名不区分大小写 也可以在bean上使用
        @Alias注解指定别名
    -->
    <typeAliases>
        <!--单独起别名-->
        <!--<typeAlias type="demo.bean.User" alias="User"/>-->

        <!--批量起别名 为当前包以及其子包下所有JavaBean起别名-->
        <package name="demo.bean"/>
    </typeAliases>

    <!--
        环境配置，可以配置多个环境下不同的数据源以及事务管理器，便于开发测试切换
        现在为develoment环境
    -->
    <environments default="development">

        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${localhost.url}"/>
                <property name="username" value="${localhost.user}"/>
                <property name="password" value="${localhost.password}"/>
            </dataSource>
        </environment>

        <environment id="test">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${aliyun.url}"/>
                <property name="username" value="${aliyun.user}"/>
                <property name="password" value="${aliyun.password}"/>
            </dataSource>
        </environment>

    </environments>

    <!--
    <databaseIdProvider type="DB_VENDOR">
```

```
        为多数据库厂商起别名
        <property name="MySQL" value="mysql"/>
    </databaseIdProvider>
    -->

    <mappers>
        <!--注册配置文件，引用类路径下的sql映射文件-->
        <mapper resource="mappers/UserMapper.xml"/>
        <mapper resource="mappers/ProductMapper.xml"/>
        <!--
            注册接口
            1、如果有sql映射文件，则必须同路径
            2、使用基于注解方式
        -->
        <mapper class="demo.dao.UserMapperAnnotation"/>
    </mappers>
</configuration>
```

更多的细节可以参照官方文档，有详细的解释；

二、映射文件的写法

XML映射文件可以说是MyBatis的灵魂。官方文档给的太详细了，这里就不多说了，地址在这：

http://www.mybatis.org/mybatis-3/zh/sqlmap-xml.html

三、逆向工程和动态SQL

了解H框架的都知道，人称的自动化框架：自动化建表，基础的CRUD都已经封装实现，不需要人为来做；

这也的确是H框架开发效率高的一个原因；

那么MyBatis也提供了一个自动化操作，逆向工程：

何为逆向工程，简单的来说就是在数据库表设计好之后，我们只需要配置好逆向工程需要的配置文件，即可通过一段代码，自动创建出与表结构对应的实体，Mapper接口，以及Mapper映射文件；

什么又是动态SQL呢？就是根据不同条件下，拼接出不同的SQL语句，可以完成不同的功能；

在实际开发中，动态SQL能大大简化我们的代码量，优化数据层的接口方法设计；而在上述的逆向工程中，就是动态SQL大量运用的最佳例子；

所以在后期，我们可以通过学习逆向工程的代码，来学习动态SQL语句的实现；在前期初学阶段，不建议上来就使用逆向工程；