Java与数据库

通过Java如何去使用数据库来帮助我们存储数据呢，这将是本章节讨论的重点。

初识JDBC

JDBC是什么？JDBC英文名为：Java Data Base Connectivity(Java数据库连接)，官方解释它是Java编程语言和广泛的数据库之间独立于数据库的连接标准的Java API，根本上说JDBC是一种规范，它提供的接口，一套完整的，允许便捷式访问底层数据库。可以用JAVA来写不同类型的可执行文件：JAVA应用程序、JAVA Applets、Java Servlet、JSP等，不同的可执行文件都能通过JDBC访问数据库，又兼备存储的优势。简单说它就是Java与数据库的连接的桥梁或者插件，用Java代码就能操作数据库的增删改查、存储过程、事务等。

我们可以发现，JDK自带了一个`java.sql`包，而这里面就定义了大量的接口，不同类型的数据库，都可以通过实现此接口，编写适用于自己数据库的实现类。而不同的数据库厂商实现的这套标准，我们称为`数据库驱动`。

准备工作

那么我们首先来进行一些准备工作，以便开始JDBC的学习：

- 将idea连接到我们的数据库，以便以后调试。
- 将mysql驱动jar依赖导入到项目中（推荐6.0版本以上，这里用到是8.0）
- 向Jetbrians申请一个学生/教师授权，用于激活idea终极版（进行JavaWeb开发需要用到，一般申请需要3-7天时间审核）不是大学生的话...emmm...懂的都懂。
- 教育授权申请地址：https://www.jetbrains.com/shop/eform/students

一个Java程序并不是一个人的战斗，我们可以在别人开发的基础上继续向上开发，其他的开发者可以将自己编写的Java代码打包为`jar`，我们只需要导入这个`jar`作为依赖，即可直接使用别人的代码，就像我们直接去使用JDK提供的类一样。

使用JDBC连接数据库

注意：6.0版本以上，不用手动加载驱动，我们直接使用即可！

```
//1. 通过DriverManager来获得数据库连接
try (Connection connection = DriverManager.getConnection("连接URL","用户名","密码");
     //2. 创建一个用于执行SQL的Statement对象
     Statement statement = connection.createStatement()){   //注意前两步都放在try()中，因为在最后需要释放资源！
    //3. 执行SQL语句，并得到结果集
    ResultSet set = statement.executeQuery("select * from 表名");
    //4. 查看结果
    while (set.next()){
        ...
    }
}catch (SQLException e){
    e.printStackTrace();
}
//5. 释放资源，try-with-resource语法会自动帮助我们close
```

其中，连接的URL如果记不住格式，我们可以打开idea的数据库连接配置，复制一份即可。（其实idea本质也是使用的JDBC，整个idea程序都是由Java编写的，实际上idea就是一个Java程序）

了解DriverManager

我们首先来了解一下DriverManager是什么东西，它其实就是管理我们的数据库驱动的：

```
public static synchronized void registerDriver(java.sql.Driver driver,
        DriverAction da)
    throws SQLException {

    /* Register the driver if it has not already been added to our list */
    if(driver != null) {
        registeredDrivers.addIfAbsent(new DriverInfo(driver, da));    //在刚启动时，mysql实现的驱动会被加载，我们可以断点调试一下。
    } else {
        // This is for compatibility with the original DriverManager
        throw new NullPointerException();
    }

    println("registerDriver: " + driver);

}
```

我们可以通过调用getConnection()来进行数据库的链接：

```
@CallerSensitive
public static Connection getConnection(String url,
    String user, String password) throws SQLException {
    java.util.Properties info = new java.util.Properties();

    if (user != null) {
        info.put("user", user);
    }
    if (password != null) {
        info.put("password", password);
    }

    return (getConnection(url, info, Reflection.getCallerClass()));   //内部有实现
}
```

我们可以手动为驱动管理器添加一个日志打印：

```
static {
    DriverManager.setLogWriter(new PrintWriter(System.out));   //这里直接设定为控制台输出
}
```

现在我们执行的数据库操作日志会在控制台实时打印。

了解Connection

Connection是数据库的连接对象，可以通过连接对象来创建一个Statement用于执行SQL语句：

```
Statement createStatement() throws SQLException;
```

我们发现除了普通的Statement，还存在PreparedStatement：

```
PreparedStatement prepareStatement(String sql)
    throws SQLException;
```

在后面我们会详细介绍PreparedStatement的使用，它能够有效地预防SQL注入式攻击。

它还支持事务的处理，也放到后面来详细进行讲解。

了解Statement

我们发现，我们之前使用了`executeQuery()`方法来执行`select`语句，此方法返回给我们一个ResultSet对象，查询得到的数据，就存放在ResultSet中！

Statement除了执行这样的DQL语句外，我们还可以使用`executeUpdate()`方法来执行一个DML或是DDL语句，它会返回一个int类型，表示执行后受影响的行数，可以通过它来判断DML语句是否执行成功。

也可以通过`excute()`来执行任意的SQL语句，它会返回一个`boolean`来表示执行结果是一个ResultSet还是一个int，我们可以通过使用`getResultSet()`或是`getUpdateCount()`来获取。

执行DML操作

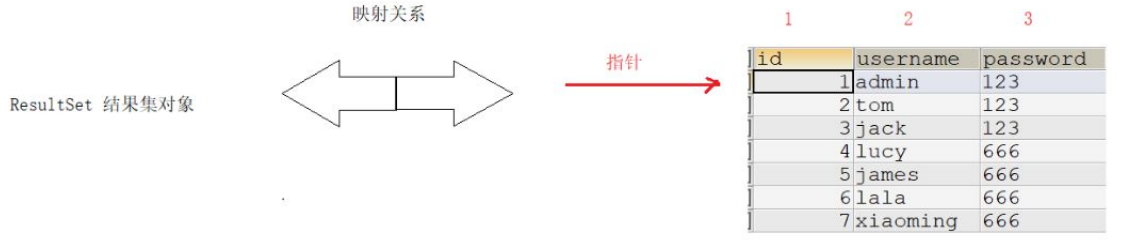
我们通过几个例子来向数据库中插入数据。

执行DQL操作

执行DQL操作会返回一个ResultSet对象，我们来看看如何从ResultSet中去获取数据：

```
//首先要明确，select返回的数据类似于一个excel表格
while (set.next()){
    //每调用一次next()就会向下移动一行，首次调用会移动到第一行
}
```

我们在移动行数后，就可以通过set中提供的方法，来获取每一列的数据。



执行批处理操作

当我们要执行很多条语句时，可以不用一次一次地提交，而是一口气全部交给数据库处理，这样会节省很多的时间。

```
public static void main(String[] args) throws ClassNotFoundException {
    try (Connection connection = DriverManager.getConnection();
         Statement statement = connection.createStatement()){

        statement.addBatch("insert into user values ('f', 1234)");
        statement.addBatch("insert into user values ('e', 1234)");   //添加每一条批处理语句
        statement.executeBatch();   //一起执行

    }catch (SQLException e){
        e.printStackTrace();
    }
}
```

将查询结果映射为对象

既然我们现在可以从数据库中获取数据了，那么现在就可以将这些数据转换为一个类来进行操作，首先定义我们的实体类：

```
public class Student {
    Integer sid;
    String name;
    String sex;

    public Student(Integer sid, String name, String sex) {
        this.sid = sid;
        this.name = name;
        this.sex = sex;
    }

    public void say(){
        System.out.println("我叫："+name+"，学号为："+sid+"，我的性别是："+sex);
    }
}
```

现在我们来进行一个转换：

```
while (set.next()){
    Student student = new Student(set.getInt(1), set.getString(2), set.getString(3));
    student.say();
}
```

注意：列的下标是从1开始的。

我们也可以利用反射机制来将查询结果映射为对象，使用反射的好处是，无论什么类型都可以通过我们的方法来进行实体类型映射：

```
private static <T> T convert(ResultSet set, Class<T> clazz){
    try {
        Constructor<T> constructor =
clazz.getConstructor(clazz.getConstructors()[0].getParameterTypes());   //默认获取
第一个构造方法
        Class<?>[] param = constructor.getParameterTypes();  //获取参数列表
        Object[] object = new Object[param.length];  //存放参数
        for (int i = 0; i < param.length; i++) {   //是从1开始的
            object[i] = set.getObject(i+1);
            if(object[i].getClass() != param[i])
                throw new SQLException("错误的类型转换："+object[i].getClass()+" ->
"+param[i]);
        }
        return constructor.newInstance(object);
    } catch (ReflectiveOperationException | SQLException e) {
        e.printStackTrace();
        return null;
    }
}
```

现在我们就可以通过我们的方法来将查询结果转换为一个对象了：

```
while (set.next()){
    Student student = convert(set, Student.class);
    if(student != null) student.say();
}
```

实际上，在后面我们会学习Mybatis框架，它对JDBC进行了深层次的封装，而它就进行类似上面反射的操作来便于我们对数据库数据与实体类的转换。

实现登陆与SQL注入攻击

在使用之前，我们先来看看如果我们想模拟登陆一个用户，我们该怎么去写：

```
try (Connection connection = DriverManager.getConnection("URL","用户名","密码");
     Statement statement = connection.createStatement();
     Scanner scanner = new Scanner(System.in)){
    ResultSet res = statement.executeQuery("select * from user where
username='"+scanner.nextLine()+"'and pwd='"+scanner.nextLine()+"';");
    while (res.next()){
        String username = res.getString(1);
        System.out.println(username+" 登陆成功！");
    }
}catch (SQLException e){
    e.printStackTrace();
}
```

用户可以通过自己输入用户名和密码来登陆，乍一看好像没啥问题，那如果我输入的是以下内容呢：

```
Test
1111' or 1=1; --
# Test 登陆成功!
```

1=1一定是true，那么我们原本的SQL语句会变为：

```
select * from user where username='Test' and pwd='1111' or 1=1; -- '
```

我们发现，如果允许这样的数据插入，那么我们原有的SQL语句结构就遭到了破坏，使得用户能够随意登陆别人的账号。因此我们可能需要限制用户的输入来防止用户输入一些SQL语句关键字，但是关键字非常多，这并不是解决问题的最好办法。

使用PreparedStatement

我们发现，如果单纯地使用Statement来执行SQL命令，会存在严重的SQL注入攻击漏洞！而这种问题，我们可以使用PreparedStatement来解决：

```
public static void main(String[] args) throws ClassNotFoundException {
    try (Connection connection = DriverManager.getConnection("URL","用户名","密
码");
         PreparedStatement statement = connection.prepareStatement("select *
from user where username= ? and pwd=?;");
         Scanner scanner = new Scanner(System.in)){

        statement.setString(1, scanner.nextLine());
        statement.setString(2, scanner.nextLine());
        System.out.println(statement);    //打印查看一下最终执行的
        ResultSet res = statement.executeQuery();
        while (res.next()){
            String username = res.getString(1);
            System.out.println(username+" 登陆成功！");
        }
    }catch (SQLException e){
        e.printStackTrace();
    }
}
```

我们发现，我们需要提前给到PreparedStatement一个SQL语句，并且使用`?`作为占位符，它会预编译一个SQL语句，通过直接将我们的内容进行替换的方式来填写数据。使用这种方式，我们之前的例子就失效了！我们来看看实际执行的SQL语句是什么：

```
com.mysql.cj.jdbc.ClientPreparedStatement: select * from user where username=
'Test' and pwd='123456'' or 1=1; -- ';
```

我们发现，我们输入的参数一旦出现`'`时，会被变为转义形式`\'`，而最外层有一个真正的`'`来将我们输入的内容进行包裹，因此它能够有效地防止SQL注入攻击！

管理事务

JDBC默认的事务处理行为是自动提交，所以前面我们执行一个SQL语句就会被直接提交（相当于没有启动事务），所以JDBC需要进行事务管理时，首先要通过Connection对象调用setAutoCommit(false) 方法, 将SQL语句的提交（commit）由驱动程序转交给应用程序负责。

```
con.setAutoCommit();   //关闭自动提交后相当于开启事务。
// SQL语句
// SQL语句
// SQL语句
con.commit();或 con.rollback();
```

一旦关闭自动提交，那么现在执行所有的操作如果在最后不进行`commit()`来提交事务的话，那么所有的操作都会丢失，只有提交之后，所有的操作才会被保存！也可以使用`rollback()`来手动回滚之前的全部操作！

```
public static void main(String[] args) throws ClassNotFoundException {
    try (Connection connection = DriverManager.getConnection("URL","用户名","密码");
         Statement statement = connection.createStatement()){

        connection.setAutoCommit(false);  //关闭自动提交，现在将变为我们手动提交
        statement.executeUpdate("insert into user values ('a', 1234)");
        statement.executeUpdate("insert into user values ('b', 1234)");
        statement.executeUpdate("insert into user values ('c', 1234)");

        connection.commit();   //如果前面任何操作出现异常，将不会执行commit()，之前的操作也就不会生效
    }catch (SQLException e){
        e.printStackTrace();
    }
}
```

我们来接着尝试一下使用回滚操作：

```
public static void main(String[] args) throws ClassNotFoundException {
    try (Connection connection = DriverManager.getConnection("URL","用户名","密码");
         Statement statement = connection.createStatement()){

        connection.setAutoCommit(false);  //关闭自动提交，现在将变为我们手动提交
        statement.executeUpdate("insert into user values ('a', 1234)");
        statement.executeUpdate("insert into user values ('b', 1234)");

        connection.rollback();   //回滚，撤销前面全部操作

        statement.executeUpdate("insert into user values ('c', 1234)");

        connection.commit();   //提交事务（注意，回滚之前的内容都没了）

    }catch (SQLException e){
        e.printStackTrace();
    }
}
```

同样的，我们也可以去创建一个回滚点来实现定点回滚：

```
public static void main(String[] args) throws ClassNotFoundException {
```

```
    try (Connection connection = DriverManager.getConnection("URL","用户名","密
码");
         Statement statement = connection.createStatement()){

        connection.setAutoCommit(false);  //关闭自动提交，现在将变为我们手动提交
        statement.executeUpdate("insert into user values ('a', 1234)");

        Savepoint savepoint = connection.setSavepoint();   //创建回滚点
        statement.executeUpdate("insert into user values ('b', 1234)");

        connection.rollback(savepoint);   //回滚到回滚点，撤销前面全部操作

        statement.executeUpdate("insert into user values ('c', 1234)");

        connection.commit();   //提交事务（注意，回滚之前的内容都没了）

    }catch (SQLException e){
        e.printStackTrace();
    }
}
```

通过开启事务，我们就可以更加谨慎地进行一些操作了，如果我们想从事务模式切换为原有的自动提交模式，我们可以直接将其设置回去：

```
public static void main(String[] args) throws ClassNotFoundException {
    try (Connection connection = DriverManager.getConnection("URL","用户名","密
码");
         Statement statement = connection.createStatement()){

        connection.setAutoCommit(false);  //关闭自动提交，现在将变为我们手动提交
        statement.executeUpdate("insert into user values ('a', 1234)");
        connection.setAutoCommit(true);   //重新开启自动提交，开启时把之前的事务模式下的
内容给提交了
        statement.executeUpdate("insert into user values ('d', 1234)");
        //没有commit也成功了！
    }catch (SQLException e){
        e.printStackTrace();
    }
```

通过学习JDBC，我们现在就可以通过Java来访问和操作我们的数据库了！为了更好地衔接，我们还会接着讲解主流持久层框架——Mybatis，加深JDBC的记忆。

使用Lombok

我们发现，在以往编写项目时，尤其是在类进行类内部成员字段封装时，需要编写大量的get/set方法，这不仅使得我们类定义中充满了get和set方法，同时如果字段名称发生改变，又要挨个进行修改，甚至当字段变得很多时，构造方法的编写会非常麻烦！

通过使用Lombok（小辣椒）就可以解决这样的问题！

我们来看看，使用原生方式和小辣椒方式编写类的区别，首先是传统方式：

```
public class Student {
    private Integer sid;
    private String name;
    private String sex;

    public Student(Integer sid, String name, String sex) {
        this.sid = sid;
        this.name = name;
        this.sex = sex;
    }

    public Integer getSid() {             //长!
        return sid;
    }

    public void setSid(Integer sid) {     //到!
        this.sid = sid;
    }

    public String getName() {             //爆!
        return name;
    }

    public void setName(String name) {    //炸!
        this.name = name;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }
}
```

而使用Lombok之后：

```
@Getter
@Setter
@AllArgsConstructor
public class Student {
    private Integer sid;
    private String name;
    private String sex;
}
```

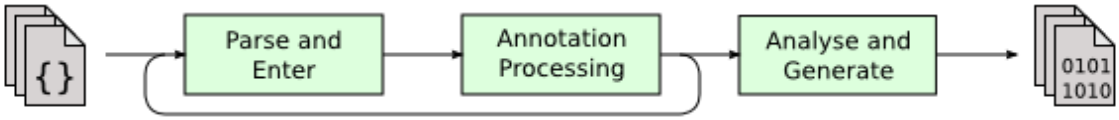
我们发现，使用Lombok之后，只需要添加几个注解，就能够解决掉我们之前长长的一串代码！

配置Lombok

- 首先我们需要导入Lombok的jar依赖，和jdbc依赖是一样的，放在项目目录下直接导入就行了。可以在这里进行下载：https://projectlombok.org/download
- 然后我们要安装一下Lombok插件，由于IDEA默认都安装了Lombok的插件，因此直接导入依赖后就可以使用了。
- 重启IDEA

Lombok是一种插件化注解API，是通过添加注解来实现的，然后在javac进行编译的时候，进行处理。

Java的编译过程可以分成三个阶段：



1. 所有源文件会被解析成语法树。
2. 调用注解处理器。如果注解处理器产生了新的源文件，新文件也要进行编译。
3. 最后，语法树会被分析并转化成类文件。

实际上在上述的第二阶段，会执行*lombok.core.AnnotationProcessor*，它所做的工作就是我们上面所说的，修改语法树。

使用Lombok

我们通过实战来演示一下Lombok的实用注解：

- 我们通过添加`@Getter`和`@Setter`来为当前类的所有字段生成get/set方法，他们可以添加到类或是字段上，注意静态字段不会生成，final字段无法生成set方法。
    - 我们还可以使用@Accessors来控制生成Getter和Setter的样式。
- 我们通过添加`@ToString`来为当前类生成预设的toString方法。
- 我们可以通过添加`@EqualsAndHashCode`来快速生成比较和哈希值方法。
- 我们可以通过添加`@AllArgsConstructor`和`@NoArgsConstructor`来快速生成全参构造和无参构造。
- 我们可以添加`@RequiredArgsConstructor`来快速生成参数只包含`final`或被标记为`@NonNull`的成员字段。
- 使用`@Data`能代表`@Setter`、`@Getter`、`@RequiredArgsConstructor`、`@ToString`、`@EqualsAndHashCode`全部注解。
    - 一旦使用`@Data`就不建议此类有继承关系，因为`equal`方法可能不符合预期结果（尤其是仅比较子类属性）。

- 使用`@Value`与`@Data`类似，但是并不会生成setter并且成员属性都是final的。
- 使用`@SneakyThrows`来自动生成try-catch代码块。
- 使用`@Cleanup`作用与局部变量，在最后自动调用其`close()`方法（可以自由更换）
- 使用`@Builder`来快速生成建造者模式。
    - 通过使用`@Builder.Default`来指定默认值。
    - 通过使用`@Builder.ObtainVia`来指定默认值的获取方式。

认识Mybatis

在前面JDBC的学习中，虽然我们能够通过JDBC来连接和操作数据库，但是哪怕只是完成一个SQL语句的执行，都需要编写大量的代码，更不用说如果我还需要进行实体类映射，将数据转换为我们可以直接操作的实体类型，JDBC很方便，但是还不够方便，我们需要一种更加简洁高效的方式来和数据库进行交互。

再次强调：学习厉害的框架或是厉害的技术，并不是为了一定要去使用它，而是它们能够使得我们在不同的开发场景下，合理地使用这些技术，以灵活地应对需要解决的问题。

MyBatis 是一款优秀的持久层框架，它支持定制化 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。MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs(Plain Ordinary Java Object,普通的 Java对象)映射成数据库中的记录。

我们依然使用传统的jar依赖方式，从最原始开始讲起，不使用Maven，有关Maven内容我们会在后面统一讲解！全程围绕官方文档讲解！

这一块内容很多很杂，再次强调要多实践！

XML语言概述

在开始介绍Mybatis之前，XML语言发明最初是用于数据的存储和传输，它可以长这样：

```
<?xml version="1.0" encoding="UTF-8" ?>
<outer>
  <name>阿伟</name>
  <desc>怎么又在玩电动啊</desc>
    <inner type="1">
    <age>10</age>
    <sex>男</sex>
  </inner>
</outer>
```

如果你学习过前端知识，你会发现它和HTML几乎长得一模一样！但是请注意，虽然它们长得差不多，但是他们的意义却不同，HTML主要用于通过编排来展示数据，而XML主要是存放数据，它更像是一个配置文件！当然，浏览器也是可以直接打开XML文件的。

一个XML文件存在以下的格式规范：

- 必须存在一个根节点，将所有的子标签全部包含。
- 可以但不必须包含一个头部声明（主要是可以设定编码格式）
- 所有的标签必须成对出现，可以嵌套但不能交叉嵌套

- 区分大小写。
- 标签中可以存在属性，比如上面的`type="1"`就是`inner`标签的一个属性，属性的值由单引号或双引号包括。

XML文件也可以使用注释：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- 注释内容 -->
```

通过IDEA我们可以使用`Ctrl`+`/`来快速添加注释文本（不仅仅适用于XML，还支持很多种类型的文件）

那如果我们的内容中出现了`<`或是`>`字符，那该怎么办呢？我们就可以使用XML的转义字符来代替：

& 或 &	&	和
< 或 <	<	小于号
> 或 >	>	大于号
&quot;	"	双引号
		空格
&copy;	©	版权符
&reg	®	注册符

如果嫌一个一个改太麻烦，也可以使用CD来快速创建不解析区域：

```
<test>
    <name><![CDATA[我看你<><><>是一点都不懂哦>>>]]></name>
</test>
```

那么，我们现在了解了XML文件的定义，现在该如何去解析一个XML文件呢？比如我们希望将定义好的XML文件读取到Java程序中，这时该怎么做呢？

JDK为我们内置了一个叫做`org.w3c`的XML解析库，我们来看看如何使用它来进行XML文件内容解析：

```
// 创建DocumentBuilderFactory对象
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
// 创建DocumentBuilder对象
try {
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document d = builder.parse("file:mappers/test.xml");
    // 每一个标签都作为一个节点
    NodeList nodeList = d.getElementsByTagName("test");  // 可能有很多个名字为test的
标签
    Node rootNode = nodeList.item(0); // 获取首个
```

```
    NodeList childNodes = rootNode.getChildNodes(); // 一个节点下可能会有很多个节点，比如根节点下就囊括了所有的节点
    //节点可以是一个带有内容的标签（它内部就还有子节点），也可以是一段文本内容

    for (int i = 0; i < childNodes.getLength(); i++) {
        Node child = childNodes.item(i);
        if(child.getNodeType() == Node.ELEMENT_NODE)  //过滤换行符之类的内容，因为它们都被认为是一个文本节点
        System.out.println(child.getNodeName() + ": "
+child.getFirstChild().getNodeValue());
        // 输出节点名称，也就是标签名称，以及标签内部的文本（内部的内容都是子节点，所以要获取内部的节点）
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

当然，学习和使用XML只是为了更好地去认识Mybatis的工作原理，以及如何使用XML来作为Mybatis的配置文件，这是在开始之前必须要掌握的内容（使用Java读取XML内容不要求掌握，但是需要知道Mybatis就是通过这种方式来读取配置文件的）

不仅仅是Mybatis，包括后面的Spring等众多框架都会用到XML来作为框架的配置文件！

初次使用Mybatis

那么我们首先来感受一下Mybatis给我们带来的便捷，就从搭建环境开始，中文文档网站：https://mybatis.org/mybatis-3/zh/configuration.html

我们需要导入Mybatis的依赖，Jar包需要在github上下载，如果卡得一匹，连不上可以在视频简介处从分享的文件中获取。同样地放入到项目的根目录下，右键作为依赖即可！（依赖变多之后，我们可以将其放到一个单独的文件夹，不然会很繁杂）

依赖导入完成后，我们就可以编写Mybatis的配置文件了（现在不是在Java代码中配置了，而是通过一个XML文件去配置，这样就使得硬编码的部分大大减少，项目后期打包成Jar运行不方便修复，但是通过配置文件，我们随时都可以去修改，就变得很方便了，同时代码量也大幅度减少，配置文件填写完成后，我们只需要关心项目的业务逻辑而不是如何去读取配置文件）我们按照官方文档给定的提示，在项目根目录下新建名为`mybatis-config.xml`的文件，并填写以下内容：

```
<?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="${驱动类（含包名）}"/>
        <property name="url" value="${数据库连接URL}"/>
        <property name="username" value="${用户名}"/>
        <property name="password" value="${密码}"/>
      </dataSource>
    </environment>
  </environments>
</configuration>
```

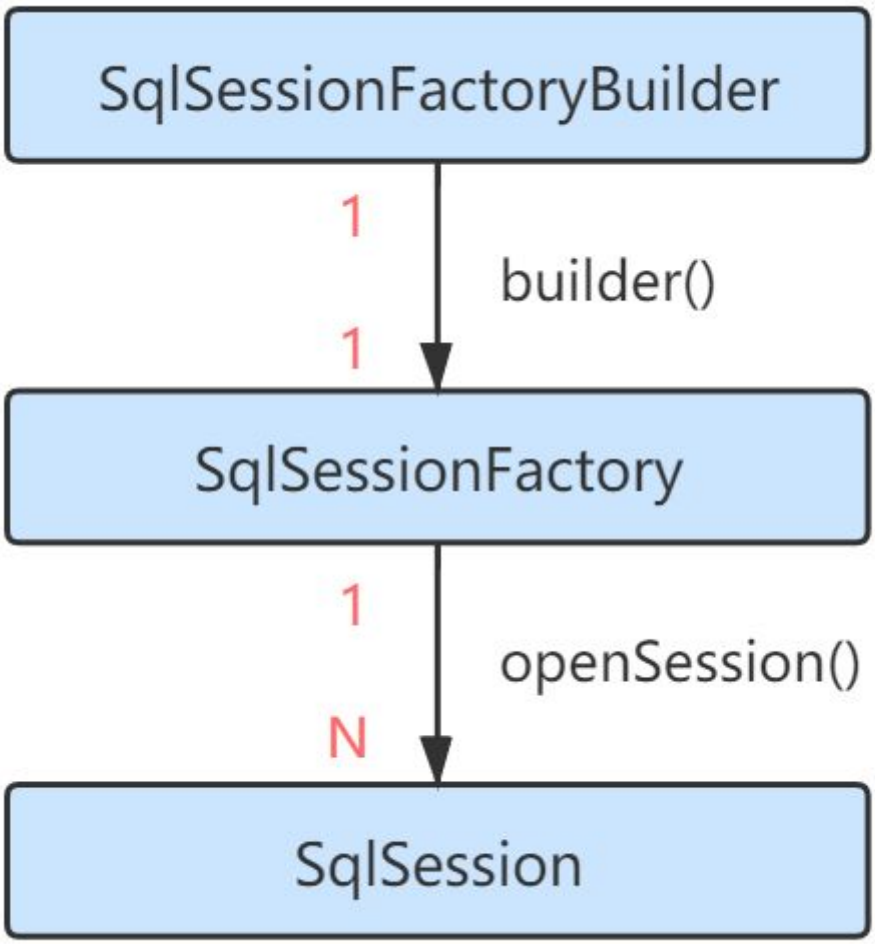
我们发现，在最上方还引入了一个叫做DTD（文档类型定义）的东西，它提前帮助我们规定了一些标签，我们就需要使用Mybatis提前帮助我们规定好的标签来进行配置（因为只有这样Mybatis才能正确识别我们配置的内容）

通过进行配置，我们就告诉了Mybatis我们链接数据库的一些信息，包括URL、用户名、密码等，这样Mybatis就知道该链接哪个数据库、使用哪个账号进行登陆了（也可以不使用配置文件，这里不做讲解，还请各位小伙伴自行阅读官方文档）

配置文件完成后，我们需要在Java程序启动时，让Mybatis对配置文件进行读取并得到一个`SqlSessionFactory`对象：

```
public static void main(String[] args) throws FileNotFoundException {
    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(new FileInputStream("mybatis-config.xml"));
    try (SqlSession sqlSession = sqlSessionFactory.openSession(true)){
            //暂时还没有业务
    }
}
```

直接运行即可，虽然没有干什么事情，但是不会出现错误，如果之前的配置文件编写错误，直接运行会产生报错！那么现在我们来看看，`SqlSessionFactory`对象是什么东西：



每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为核心的，我们可以通过`SqlSessionFactory`来创建多个新的会话，`SqlSession`对象，每个会话就相当于我不同的地方登陆一个账号去访问数据库，你也可以认为这就是之前JDBC中的`Statement`对象，会话之间相互隔离，没有任何关联。

而通过`SqlSession`就可以完成几乎所有的数据库操作，我们发现这个接口中定义了大量数据库操作的方法，因此，现在我们只需要通过一个对象就能完成数据库交互了，极大简化了之前的流程。

我们来尝试一下直接读取实体类，读取实体类肯定需要一个映射规则，比如类中的哪个字段对应数据库中的哪个字段，在查询语句返回结果后，Mybatis就会自动将对应的结果填入到对象的对应字段上。首先编写实体类，，直接使用Lombok是不是就很方便了：

```
import lombok.Data;

@Data
public class Student {
    int sid;   //名称最好和数据库字段名称保持一致，不然可能会映射失败导致查询结果丢失
    String name;
    String sex;
}
```

在根目录下重新创建一个mapper文件夹，新建名为`TestMapper.xml`的文件作为我们的映射器，并填写以下内容：

```
<?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="TestMapper">
    <select id="selectStudent" resultType="com.test.entity.Student">
        select * from student
    </select>
</mapper>
```

其中namespace就是命名空间，每个Mapper都是唯一的，因此需要用一个命名空间来区分，它还可以用来绑定一个接口。我们在里面写入了一个select标签，表示添加一个select操作，同时id作为操作的名称，resultType指定为我们刚刚定义的实体类，表示将数据库结果映射为`Student`类，然后就在标签中写入我们的查询语句即可。

编写好后，我们在配置文件中添加这个Mapper映射器：

```
<mappers>
    <mapper url="file:mappers/TestMapper.xml"/>
    <!--    这里用的是url，也可以使用其他类型，我们会在后面讲解    -->
</mappers>
```

最后在程序中使用我们定义好的Mapper即可：

```
public static void main(String[] args) throws FileNotFoundException {
    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(new FileInputStream("mybatis-config.xml"));
    try (SqlSession sqlSession = sqlSessionFactory.openSession(true)){
        List<Student> student = sqlSession.selectList("selectStudent");
        student.forEach(System.out::println);
    }
}
```

我们会发现，Mybatis非常智能，我们只需要告诉一个映射关系，就能够直接将查询结果转化为一个实体类！

配置Mybatis

在了解了Mybatis为我们带来的便捷之后，现在我们就可以正式地去学习使用Mybatis了！

由于`SqlSessionFactory`一般只需要创建一次，因此我们可以创建一个工具类来集中创建`SqlSession`，这样会更加方便一些：

```
public class MybatisUtil {

    //在类加载时就进行创建
    private static SqlSessionFactory sqlSessionFactory;
    static {
        try {
            sqlSessionFactory = new SqlSessionFactoryBuilder().build(new
FileInputStream("mybatis-config.xml"));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * 获取一个新的会话
     * @param autoCommit 是否开启自动提交（跟JDBC是一样的，如果不自动提交，则会变成事务操
作）
     * @return SqlSession对象
     */
    public static SqlSession getSession(boolean autoCommit){
        return sqlSessionFactory.openSession(autoCommit);
    }
}
```

现在我们只需要在main方法中这样写即可查询结果了：

```
public static void main(String[] args) {
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        List<Student> student = sqlSession.selectList("selectStudent");
        student.forEach(System.out::println);
    }
}
```

之前我们演示了，如何创建一个映射器来将结果快速转换为实体类，但是这样可能还是不够方便，我们每次都需要去找映射器对应操作的名称，而且还要知道对应的返回类型，再通过`SqlSession`来执行对应的方法，能不能再方便一点呢?

现在，我们可以通过`namespace`来绑定到一个接口上，利用接口的特性，我们可以直接指明方法的行为，而实际实现则是由Mybatis来完成。

```
public interface TestMapper {
    List<Student> selectStudent();
}
```

将Mapper文件的命名空间修改为我们的接口，建议同时将其放到同名包中，作为内部资源:

```
<mapper namespace="com.test.mapper.TestMapper">
    <select id="selectStudent" resultType="com.test.entity.Student">
        select * from student
    </select>
</mapper>
```

作为内部资源后，我们需要修改一下配置文件中的mapper定义，不使用url而是resource表示是Jar内部的文件:

```
<mappers>
    <mapper resource="com/test/mapper/TestMapper.xml"/>
</mappers>
```

现在我们就可以直接通过`SqlSession`获取对应的实现类，通过接口中定义的行为来直接获取结果:

```
public static void main(String[] args) {
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        TestMapper testMapper = sqlSession.getMapper(TestMapper.class);
        List<Student> student = testMapper.selectStudent();
        student.forEach(System.out::println);
    }
}
```

那么肯定有人好奇，TestMapper明明是一个我们自己定义接口啊，Mybatis也不可能提前帮我们写了实现类啊，那这接口怎么就出现了一个实现类呢?我们可以通过调用`getClass()`方法来看看实现类是个什么:

```
TestMapper testMapper = sqlSession.getMapper(TestMapper.class);
System.out.println(testMapper.getClass());
```

我们发现，实现类名称很奇怪，名称为`com.sun.proxy.$Proxy4`，它是通过动态代理生成的，相当于动态生成了一个实现类，而不是预先定义好的，有关Mybatis这一部分的原理，我们放在最后一节进行讲解。

接下来，我们再来看配置文件，之前我们并没有对配置文件进行一个详细的介绍:

```
<configuration>
    <environments default="development">
        <environment id="development">
```

```
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/study"/>
                <property name="username" value="test"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="com/test/mapper/TestMapper.xml"/>
    </mappers>
</configuration>
```

首先就从`environments`标签说起，一般情况下，我们在开发中，都需要指定一个数据库的配置信息，包含连接URL、用户、密码等信息，而`environment`就是用于进行这些配置的！实际情况下可能会不止有一个数据库连接信息，比如开发过程中我们一般会使用本地的数据库，而如果需要将项目上传到服务器或是防止其他人的电脑上运行时，我们可能就需要配置另一个数据库的信息，因此，我们可以提前定义好所有的数据库信息，该什么时候用什么即可！

在`environments`标签上有一个default属性，来指定默认的环境，当然如果我们希望使用其他环境，可以修改这个默认环境，也可以在创建工厂时选择环境：

```
sqlSessionFactory = new SqlSessionFactoryBuilder()
        .build(new FileInputStream("mybatis-config.xml"), "环境ID");
```

我们还可以给类型起一个别名，以简化Mapper的编写：

```
<!-- 需要在environments的上方 -->
<typeAliases>
    <typeAlias type="com.test.entity.Student" alias="Student"/>
</typeAliases>
```

现在Mapper就可以直接使用别名了：

```
<mapper namespace="com.test.mapper.TestMapper">
    <select id="selectStudent" resultType="Student">
        select * from student
    </select>
</mapper>
```

如果这样还是很麻烦，我们也可以直接让Mybatis去扫描一个包，并将包下的所有类自动起别名（别名为首字母小写的类名）

```
<typeAliases>
    <package name="com.test.entity"/>
</typeAliases>
```

也可以为指定实体类添加一个注解，来指定别名：

```java
@Data
@Alias("lbwnb")
public class Student {
    private int sid;
    private String name;
    private String sex;
}
```

当然，Mybatis也包含许多的基础配置，通过使用：

```xml
<settings>
    <setting name="" value=""/>
</settings>
```

所有的配置项可以在中文文档处查询，本文不会进行详细介绍，在后面我们会提出一些比较重要的配置项。

有关配置文件的介绍就暂时到这里为止，我们讨论的重心应该是Mybatis的应用，而不是配置文件，所以省略了一部分内容的讲解。

增删改查

在了解了Mybatis的一些基本配置之后，我们就可以正式来使用Mybatis来进行数据库操作了！

在前面我们演示了如何快速进行查询，我们只需要编写一个对应的映射器既可以了：

```xml
<mapper namespace="com.test.mapper.TestMapper">
    <select id="studentList" resultType="Student">
        select * from student
    </select>
</mapper>
```

当然，如果你不喜欢使用实体类，那么这些属性还可以被映射到一个Map上：

```xml
<select id="selectStudent" resultType="Map">
    select * from student
</select>
```

```java
public interface TestMapper {
    List<Map> selectStudent();
}
```

Map中就会以键值对的形式来存放这些结果了。

通过设定一个`resultType`属性，让Mybatis知道查询结果需要映射为哪个实体类，要求字段名称保持一致。那么如果我们不希望按照这样的规则来映射呢？我们可以自定义`resultMap`来设定映射规则：

```xml
<resultMap id="Test" type="Student">
    <result column="sid" property="sid"/>
    <result column="sex" property="name"/>
    <result column="name" property="sex"/>
</resultMap>
```

通过指定映射规则，我们现在名称和性别一栏就发生了交换，因为我们将其映射字段进行了交换。

如果一个类中存在多个构造方法，那么很有可能会出现这样的错误：

```
### Exception in thread "main"
org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause:
org.apache.ibatis.executor.ExecutorException: No constructor found in
com.test.entity.Student matching [java.lang.Integer, java.lang.String,
java.lang.String]
### The error may exist in com/test/mapper/TestMapper.xml
### The error may involve com.test.mapper.TestMapper.getStudentBySid
### The error occurred while handling results
### SQL: select * from student where sid = ?
### Cause: org.apache.ibatis.executor.ExecutorException: No constructor found in
com.test.entity.Student matching [java.lang.Integer, java.lang.String,
java.lang.String]
    at
org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.jav
a:30)
    ...
```

这时就需要使用`constructor`标签来指定构造方法：

```
<resultMap id="test" type="Student">
    <constructor>
        <arg column="sid" javaType="Integer"/>
        <arg column="name" javaType="String"/>
    </constructor>
</resultMap>
```

值得注意的是，指定构造方法后，若此字段被填入了构造方法作为参数，将不会通过反射给字段单独赋值，而构造方法中没有传入的字段，依然会被反射赋值，有关`resultMap`的内容，后面还会继续讲解。

如果数据库中存在一个带下划线的字段，我们可以通过设置让其映射为以驼峰命名的字段，比如`my_test`映射为`myTest`

```
<settings>
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

如果不设置，默认为不开启，也就是默认需要名称保持一致。

我们接着来看看条件查询，既然是条件查询，那么肯定需要我们传入查询条件，比如现在我们想通过sid字段来通过学号查找信息：

```
Student getStudentBySid(int sid);
```

```
<select id="getStudentBySid" parameterType="int" resultType="Student">
    select * from student where sid = #{sid}
</select>
```

我们通过使用`#{xxx}`或是`${xxx}`来填入我们给定的属性，实际上Mybatis本质也是通过`PreparedStatement`首先进行一次预编译，有效地防止SQL注入问题，但是如果使用`${xxx}`就不再是通过预编译，而是直接传值，因此我们一般都使用`#{xxx}`来进行操作。

使用`parameterType`属性来指定参数类型（非必须，可以不用，推荐不用）

接着我们来看插入、更新和删除操作，其实与查询操作差不多，不过需要使用对应的标签，比如插入操作：

```
<insert id="addStudent" parameterType="Student">
    insert into student(name, sex) values(#{name}, #{sex})
</insert>
```

```
int addStudent(Student student);
```

我们这里使用的是一个实体类，我们可以直接使用实体类里面对应属性替换到SQL语句中，只需要填写属性名称即可，和条件查询是一样的。

复杂查询

一个老师可以教授多个学生，那么能否一次性将老师的学生全部映射给此老师的对象呢，比如：

```
@Data
public class Teacher {
    int tid;
    String name;
    List<Student> studentList;
}
```

映射为Teacher对象时，同时将其教授的所有学生一并映射为List列表，显然这是一种一对多的查询，那么这时就需要进行复杂查询了。而我们之前编写的都非常简单，直接就能完成映射，因此我们现在需要使用`resultMap`来自定义映射规则：

```
<select id="getTeacherByTid" resultMap="asTeacher">
        select *, teacher.name as tname from student inner join teach on
student.sid = teach.sid
                                inner join teacher on teach.tid = teacher.tid where
teach.tid = #{tid}
</select>

<resultMap id="asTeacher" type="Teacher">
    <id column="tid" property="tid"/>
    <result column="tname" property="name"/>
    <collection property="studentList" ofType="Student">
        <id property="sid" column="sid"/>
        <result column="name" property="name"/>
        <result column="sex" property="sex"/>
    </collection>
</resultMap>
```

可以看到，我们的查询结果是一个多表联查的结果，而联查的数据就是我们需要映射的数据（比如这里是一个老师有N个学生，联查的结果也是这一个老师对应N个学生的N条记录），其中`id`标签用于在多条记录中辨别是否为同一个对象的数据，比如上面的查询语句得到的结果中，`tid`这一行始终为`1`，因此所有的记录都应该是`tid=1`的教师的数据，而不应该变为多个教师的数据，如果不加id进行约束，那么会被识别成多个教师的数据！

通过使用collection来表示将得到的所有结果合并为一个集合，比如上面的数据中每个学生都有单独的一条记录，因此tid相同的全部学生的记录就可以最后合并为一个List，得到最终的映射结果，当然，为了区分，最好也设置一个id，只不过这个例子中可以当做普通的`result`使用。

了解了一对多，那么多对一又该如何查询呢，比如每个学生都有一个对应的老师，现在Student新增了一个Teacher对象，那么现在又该如何去处理呢？

```
@Data
@Accessors(chain = true)
public class Student {
    private int sid;
    private String name;
    private String sex;
    private Teacher teacher;
}

@Data
public class Teacher {
    int tid;
    String name;
}
```

现在我们希望的是，每次查询到一个Student对象时都带上它的老师，同样的，我们也可以使用`resultMap`来实现（先修改一下老师的类定义，不然会很麻烦）：

```
<resultMap id="test2" type="Student">
    <id column="sid" property="sid"/>
    <result column="name" property="name"/>
    <result column="sex" property="sex"/>
    <association property="teacher" javaType="Teacher">
        <id column="tid" property="tid"/>
        <result column="tname" property="name"/>
    </association>
</resultMap>
<select id="selectStudent" resultMap="test2">
    select *, teacher.name as tname from student left join teach on student.sid =
teach.sid
                                                left join teacher on teach.tid =
teacher.tid
</select>
```

通过使用`association`进行关联，形成多对一的关系，实际上和一对多是同理的，都是对查询结果的一种处理方式罢了。

事务操作

我们可以在获取`SqlSession`关闭自动提交来开启事务模式，和JDBC其实都差不多：

```
public static void main(String[] args) {
    try (SqlSession sqlSession = MybatisUtil.getSession(false)){
        TestMapper testMapper = sqlSession.getMapper(TestMapper.class);

        testMapper.addStudent(new Student().setSex("男").setName("小王"));

        testMapper.selectStudent().forEach(System.out::println);
    }
}
```

我们发现，在关闭自动提交后，我们的内容是没有进入到数据库的，现在我们来试一下在最后提交事务：

```
sqlSession.commit();
```

在事务提交后，我们的内容才会被写入到数据库中。现在我们来试试看回滚操作：

```
try (SqlSession sqlSession = MybatisUtil.getSession(false)){
    TestMapper testMapper = sqlSession.getMapper(TestMapper.class);

    testMapper.addStudent(new Student().setSex("男").setName("小王"));

    testMapper.selectStudent().forEach(System.out::println);
    sqlSession.rollback();
    sqlSession.commit();
}
```

回滚操作也印证成功。

动态SQL

动态 SQL 是 MyBatis 的强大特性之一。如果你使用过 JDBC 或其它类似的框架，你应该能理解根据不同条件拼接 SQL 语句有多痛苦，例如拼接时要确保不能忘记添加必要的空格，还要注意去掉列表最后一个列名的逗号。利用动态 SQL，可以彻底摆脱这种痛苦。

我们直接使用官网的例子进行讲解。

缓存机制

MyBatis 内置了一个强大的事务性查询缓存机制，它可以非常方便地配置和定制。

其实缓存机制我们在之前学习IO流的时候已经提及过了，我们可以提前将一部分内容放入缓存，下次需要获取数据时，我们就可以直接从缓存中读取，这样的话相当于直接从内存中获取而不是再去向数据库索要数据，效率会更高。

因此Mybatis内置了一个缓存机制，我们查询时，如果缓存中存在数据，那么我们就可以直接从缓存中获取，而不是再去向数据库进行请求。

Mybatis存在一级缓存和二级缓存，我们首先来看一下一级缓存，默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存（一级缓存无法关闭，只能调整），我们来看看下面这段代码：

```
public static void main(String[] args) throws InterruptedException {
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        TestMapper testMapper = sqlSession.getMapper(TestMapper.class);
        Student student1 = testMapper.getStudentBySid(1);
        Student student2 = testMapper.getStudentBySid(1);
        System.out.println(student1 == student2);
    }
}
```

我们发现，两次得到的是同一个Student对象，也就是说我们第二次查询并没有重新去构造对象，而是直接得到之前创建好的对象。如果还不是很明显，我们可以修改一下实体类：

```
@Data
@Accessors(chain = true)
public class Student {

    public Student(){
        System.out.println("我被构造了");
    }

    private int sid;
    private String name;
    private String sex;
}
```

我们通过前面的学习得知Mybatis在映射为对象时，在只有一个构造方法的情况下，无论你构造方法写成什么样子，都会去调用一次构造方法，如果存在多个构造方法，那么就会去找匹配的构造方法。我们可以通过查看构造方法来验证对象被创建了几次。

结果显而易见，只创建了一次，也就是说当第二次进行同样的查询时，会直接使用第一次的结果，因为第一次的结果已经被缓存了。

那么如果我修改了数据库中的内容，缓存还会生效吗：

```
public static void main(String[] args) throws InterruptedException {
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        TestMapper testMapper = sqlSession.getMapper(TestMapper.class);
        Student student1 = testMapper.getStudentBySid(1);
        testMapper.addStudent(new Student().setName("小李").setSex("男"));
        Student student2 = testMapper.getStudentBySid(1);
        System.out.println(student1 == student2);
    }
}
```

我们发现，当我们进行了插入操作后，缓存就没有生效了，我们再次进行查询得到的是一个新创建的对象。

也就是说，一级缓存，在进行DML操作后，会使得缓存失效，也就是说Mybatis知道我们对数据库里面的数据进行了修改，所以之前缓存的内容可能就不是当前数据库里面最新的内容了。还有一种情况就是，当前会话结束后，也会清理全部的缓存，因为已经不会再用到了。但是一定注意，一级缓存只针对于单个会话，多个会话之间不相通。

```
public static void main(String[] args) {
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        TestMapper testMapper = sqlSession.getMapper(TestMapper.class);

        Student student2;
        try(SqlSession sqlSession2 = MybatisUtil.getSession(true)){
            TestMapper testMapper2 = sqlSession2.getMapper(TestMapper.class);
            student2 = testMapper2.getStudentBySid(1);
        }

        Student student1 = testMapper.getStudentBySid(1);
        System.out.println(student1 == student2);
    }
}
```

注意：一个会话DML操作只会重置当前会话的缓存，不会重置其他会话的缓存，也就是说，其他会话缓存是不会更新的！

一级缓存给我们提供了很高速的访问效率，但是它的作用范围实在是有限，如果一个会话结束，那么之前的缓存就全部失效了，但是我们希望缓存能够扩展到所有会话都能使用，因此我们可以通过二级缓存来实现，二级缓存默认是关闭状态，要开启二级缓存，我们需要在映射器XML文件中添加：

```
<cache/>
```

可见二级缓存是Mapper级别的，也就是说，当一个会话失效时，它的缓存依然会存在于二级缓存中，因此如果我们再次创建一个新的会话会直接使用之前的缓存，我们首先根据官方文档进行一些配置：

```
<cache
  eviction="FIFO"
  flushInterval="60000"
  size="512"
  readOnly="true"/>
```

我们来编写一个代码：

```
public static void main(String[] args) {
    Student student;
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        TestMapper testMapper = sqlSession.getMapper(TestMapper.class);
        student = testMapper.getStudentBySid(1);
    }

    try (SqlSession sqlSession2 = MybatisUtil.getSession(true)){
        TestMapper testMapper2 = sqlSession2.getMapper(TestMapper.class);
        Student student2 = testMapper2.getStudentBySid(1);
        System.out.println(student2 == student);
    }
}
```

我们可以看到，上面的代码中首先是第一个会话在进行读操作，完成后会结束会话，而第二个操作重新创建了一个新的会话，再次执行了同样的查询，我们发现得到的依然是缓存的结果。

那么如果我不希望某个方法开启缓存呢？我们可以添加useCache属性来关闭缓存：

```
<select id="getStudentBySid" resultType="Student" useCache="false">
    select * from student where sid = #{sid}
</select>
```

我们也可以使用flushCache="false"在每次执行后都清空缓存，通过这这个我们还可以控制DML操作完成之后不清空缓存。

```
<select id="getStudentBySid" resultType="Student" flushCache="true">
    select * from student where sid = #{sid}
</select>
```

添加了二级缓存之后，会先从二级缓存中查找数据，当二级缓存中没有时，才会从一级缓存中获取，当一级缓存中都还没有数据时，才会请求数据库，因此我们再来执行上面的代码：

```
public static void main(String[] args) {
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        TestMapper testMapper = sqlSession.getMapper(TestMapper.class);

        Student student2;
        try(SqlSession sqlSession2 = MybatisUtil.getSession(true)){
            TestMapper testMapper2 = sqlSession2.getMapper(TestMapper.class);
            student2 = testMapper2.getStudentBySid(1);
        }

        Student student1 = testMapper.getStudentBySid(1);
        System.out.println(student1 == student2);
    }
}
```

得到的结果就会是同一个对象了，因为现在是优先从二级缓存中获取。

读取顺序：二级缓存 => 一级缓存 => 数据库

虽然缓存机制给我们提供了很大的性能提升，但是缓存存在一个问题，我们之前在`计算机组成原理`中可能学习过缓存一致性问题，也就是说当多个CPU在操作自己的缓存时，可能会出现各自的缓存内容不同步的问题，而Mybatis也会这样，我们来看看这个例子：

```
public static void main(String[] args) throws InterruptedException {
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        TestMapper testMapper = sqlSession.getMapper(TestMapper.class);
        while (true){
            Thread.sleep(3000);
            System.out.println(testMapper.getStudentBySid(1));
        }
    }
}
```

我们现在循环地每三秒读取一次，而在这个过程中，我们使用IDEA手动修改数据库中的数据，将1号同学的学号改成100，那么理想情况下，下一次读取将无法获取到小明，因为小明的学号已经发生变化了。

但是结果却是依然能够读取，并且sid并没有发生改变，这也证明了Mybatis的缓存在生效，因为我们是从外部进行修改，Mybatis不知道我们修改了数据，所以依然在使用缓存中的数据，但是这样很明显是不正确的，因此，如果存在多台服务器或者是多个程序都在使用Mybatis操作同一个数据库，并且都开启了缓存，需要解决这个问题，要么就得关闭Mybatis的缓存来保证一致性：

```
<settings>
    <setting name="cacheEnabled" value="false"/>
</settings>
```

```
<select id="getStudentBySid" resultType="Student" useCache="false"
flushCache="true">
    select * from student where sid = #{sid}
</select>
```

要么就需要实现缓存共用，也就是让所有的Mybatis都使用同一个缓存进行数据存取，在后面，我们会继续学习Redis、Ehcache、Memcache等缓存框架，通过使用这些工具，就能够很好地解决缓存一致性问题。

使用注解开发

在之前的开发中，我们已经体验到Mybatis为我们带来的便捷了，我们只需要编写对应的映射器，并将其绑定到一个接口上，即可直接通过该接口执行我们的SQL语句，极大的简化了我们之前JDBC那样的代码编写模式。那么，能否实现无需xml映射器配置，而是直接使用注解在接口上进行配置呢？答案是可以的，也是现在推荐的一种方式（也不是说XML就不要去用了，由于Java 注解的表达能力和灵活性十分有限，可能相对于XML配置某些功能实现起来会不太好办，但是在大部分场景下，直接使用注解开发已经绰绰有余了）

首先我们来看一下，使用XML进行映射器编写时，我们需要现在XML中定义映射规则和SQL语句，然后再将其绑定到一个接口的方法定义上，然后再使用接口来执行：

```
<insert id="addStudent">
    insert into student(name, sex) values(#{name}, #{sex})
</insert>
```

```
int addStudent(Student student);
```

而现在，我们可以直接使用注解来实现，每个操作都有一个对应的注解：

```
@Insert("insert into student(name, sex) values(#{name}, #{sex})")
int addStudent(Student student);
```

当然，我们还需要修改一下配置文件中的映射器注册：

```
<mappers>
    <mapper class="com.test.mapper.MyMapper"/>
    <!--  也可以直接注册整个包下的 <package name="com.test.mapper"/>  -->
</mappers>
```

通过直接指定Class，来让Mybatis知道我们这里有一个通过注解实现的映射器。

我们接着来看一下，如何使用注解进行自定义映射规则：

```
@Results({
        @Result(id = true, column = "sid", property = "sid"),
        @Result(column = "sex", property = "name"),
        @Result(column = "name", property = "sex")
})
@Select("select * from student")
List<Student> getAllStudent();
```

直接通过`@Results`注解，就可以直接进行配置了，此注解的value是一个`@Result`注解数组，每个`@Result`注解都都一个单独的字段配置，其实就是我们之前在XML映射器中写的：

```
<resultMap id="test" type="Student">
    <id property="sid" column="sid"/>
    <result column="name" property="sex"/>
    <result column="sex" property="name"/>
</resultMap>
```

现在我们就可以通过注解来自定义映射规则了。那么如何使用注解来完成复杂查询呢？我们还是使用一个老师多个学生的例子：

```
@Results({
        @Result(id = true, column = "tid", property = "tid"),
        @Result(column = "name", property = "name"),
        @Result(column = "tid", property = "studentList", many =
            @Many(select = "getStudentByTid")
        )
})
@Select("select * from teacher where tid = #{tid}")
Teacher getTeacherBySid(int tid);

@Select("select * from student inner join teach on student.sid = teach.sid where
tid = #{tid}")
List<Student> getStudentByTid(int tid);
```

我们发现，多出了一个子查询，而这个子查询是单独查询该老师所属学生的信息，而子查询结果作为`@Result`注解的一个many结果，代表子查询的所有结果都归入此集合中（也就是之前的collection标签）

```
<resultMap id="asTeacher" type="Teacher">
    <id column="tid" property="tid"/>
    <result column="tname" property="name"/>
    <collection property="studentList" ofType="Student">
        <id property="sid" column="sid"/>
        <result column="name" property="name"/>
        <result column="sex" property="sex"/>
    </collection>
</resultMap>
```

同理，`@Result`也提供了`@One`子注解来实现一对一的关系表示，类似于之前的`assocation`标签：

```
@Results({
        @Result(id = true, column = "sid", property = "sid"),
        @Result(column = "sex", property = "name"),
        @Result(column = "name", property = "sex"),
        @Result(column = "sid", property = "teacher", one =
            @One(select = "getTeacherBySid")
        )
})
@Select("select * from student")
List<Student> getAllStudent();
```

如果现在我希望直接使用注解编写SQL语句但是我希望映射规则依然使用XML来实现，这时该怎么办呢?

```
@ResultMap("test")
@Select("select * from student")
List<Student> getAllStudent();
```

提供了`@ResultMap`注解，直接指定ID即可，这样我们就可以使用XML中编写的映射规则了，这里就不再演示了。

那么如果出现之前的两个构造方法的情况，且没有任何一个构造方法匹配的话，该怎么处理呢?

```
@Data
@Accessors(chain = true)
public class Student {

    public Student(int sid){
        System.out.println("我是一号构造方法"+sid);
    }

    public Student(int sid, String name){
        System.out.println("我是二号构造方法"+sid+name);
    }

    private int sid;
    private String name;
    private String sex;
}
```

我们可以通过`@ConstructorArgs`注解来指定构造方法:

```
@ConstructorArgs({
        @Arg(column = "sid", javaType = int.class),
        @Arg(column = "name", javaType = String.class)
})
@Select("select * from student where sid = #{sid} and sex = #{sex}")
Student getStudentBySidAndSex(@Param("sid") int sid, @Param("sex") String sex);
```

得到的结果和使用`constructor`标签效果一致，这里就不多做讲解了。

我们发现，当参数列表中出现两个以上的参数时，会出现错误:

```
@Select("select * from student where sid = #{sid} and sex = #{sex}")
Student getStudentBySidAndSex(int sid, String sex);
```

```
Exception in thread "main" org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause: org.apache.ibatis.binding.BindingException:
Parameter 'sid' not found. Available parameters are [arg1, arg0, param1, param2]
### Cause: org.apache.ibatis.binding.BindingException: Parameter 'sid' not
found. Available parameters are [arg1, arg0, param1, param2]
    at
org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.jav
a:30)
    at
org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSessio
n.java:153)
    at
org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSessio
n.java:145)
    at
org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSessio
n.java:140)
    at
org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession
.java:76)
    at org.apache.ibatis.binding.MapperMethod.execute(MapperMethod.java:87)
    at
org.apache.ibatis.binding.MapperProxy$PlainMethodInvoker.invoke(MapperProxy.java
:145)
    at org.apache.ibatis.binding.MapperProxy.invoke(MapperProxy.java:86)
    at com.sun.proxy.$Proxy6.getStudentBySidAndSex(Unknown Source)
    at com.test.Main.main(Main.java:16)
```

原因是Mybatis不明确到底哪个参数是什么，因此我们可以添加`@Param`来指定参数名称：

```
@Select("select * from student where sid = #{sid} and sex = #{sex}")
Student getStudentBySidAndSex(@Param("sid") int sid, @Param("sex") String sex);
```

探究：要是我两个参数一个是基本类型一个是对象类型呢？

```
System.out.println(testMapper.addStudent(100, new Student().setName("小
陆").setSex("男")));
```

```
@Insert("insert into student(sid, name, sex) values(#{sid}, #{name}, #{sex})")
int addStudent(@Param("sid") int sid, @Param("student")  Student student);
```

那么这个时候，就出现问题了，Mybatis就不能明确这些属性是从哪里来的：

```
### SQL: insert into student(sid, name, sex) values(?, ?, ?)
### Cause: org.apache.ibatis.binding.BindingException: Parameter 'name' not
found. Available parameters are [student, param1, sid, param2]
    at
org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.jav
a:30)
    at
org.apache.ibatis.session.defaults.DefaultSqlSession.update(DefaultSqlSession.ja
va:196)
    at
org.apache.ibatis.session.defaults.DefaultSqlSession.insert(DefaultSqlSession.ja
va:181)
    at org.apache.ibatis.binding.MapperMethod.execute(MapperMethod.java:62)
    at
org.apache.ibatis.binding.MapperProxy$PlainMethodInvoker.invoke(MapperProxy.java
:145)
    at org.apache.ibatis.binding.MapperProxy.invoke(MapperProxy.java:86)
    at com.sun.proxy.$Proxy6.addStudent(Unknown Source)
    at com.test.Main.main(Main.java:16)
```

那么我们就通过参数名称.属性的方式去让Mybatis知道我们要用的是哪个属性：

```
@Insert("insert into student(sid, name, sex) values(#{sid}, #{student.name}, #
{student.sex})")
int addStudent(@Param("sid") int sid, @Param("student")  Student student);
```

那么如何通过注解控制缓存机制呢？

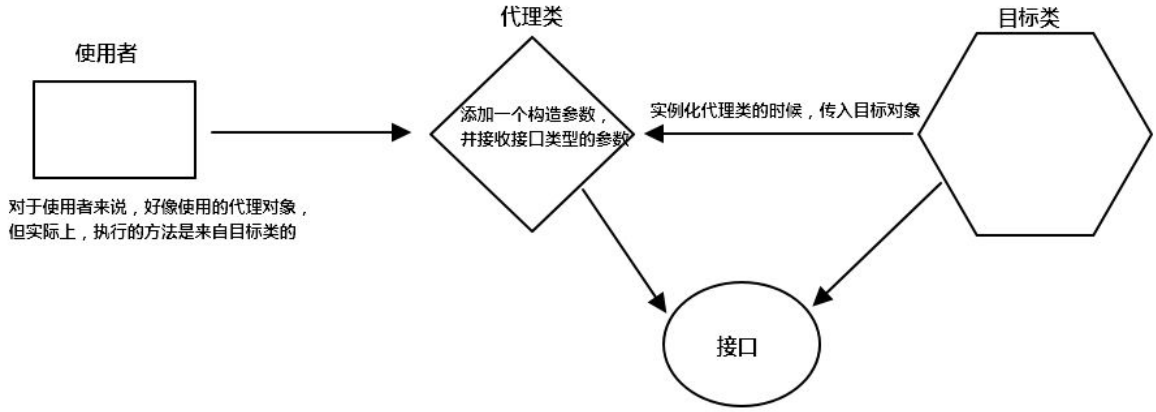
```
@CacheNamespace(readWrite = false)
public interface MyMapper {

    @Select("select * from student")
    @Options(useCache = false)
    List<Student> getAllStudent();
```

使用`@CacheNamespace`注解直接定义在接口上即可，然后我们可以通过使用`@Options`来控制单个操作的缓存启用。

探究Mybatis的动态代理机制

在探究动态代理机制之前，我们要先聊聊什么是代理：其实顾名思义，就好比我开了个大棚，里面栽种的西瓜，那么西瓜成熟了是不是得去卖掉赚钱，而我们的西瓜非常多，一个人肯定卖不过来，肯定就要去多找几个开水果摊的帮我们卖，这就是一种代理。实际上是由水果摊老板在帮我们卖瓜，我们只告诉老板卖多少钱，而至于怎么卖的是由水果摊老板决定的。



那么现在我们来尝试实现一下这样的类结构，首先定义一个接口用于规范行为：

```
public interface Shopper {

    //卖瓜行为
    void saleWatermelon(String customer);
}
```

然后需要实现一下卖瓜行为，也就是我们要告诉老板卖多少钱，这里就直接写成成功出售：

```
public class ShopperImpl implements Shopper{

    //卖瓜行为的实现
    @Override
    public void saleWatermelon(String customer) {
        System.out.println("成功出售西瓜给 ===> "+customer);
    }
}
```

最后老板代理后肯定要用自己的方式去出售这些西瓜，成交之后再按照我们告诉老板的价格进行出售：

```
public class ShopperProxy implements Shopper{

    private final Shopper impl;

    public ShopperProxy(Shopper impl){
        this.impl = impl;
    }

    //代理卖瓜行为
    @Override
    public void saleWatermelon(String customer) {
        //首先进行 代理商讨价还价行为
        System.out.println(customer + "：哥们，这瓜多少钱一斤啊？");
        System.out.println("老板：两块钱一斤。");
        System.out.println(customer + "：你这瓜皮子是金子做的，还是瓜粒子是金子做的？");
        System.out.println("老板：你瞅瞅现在哪有瓜啊，这都是大棚的瓜，你嫌贵我还嫌贵呢。");
        System.out.println(customer + "：给我挑一个。");

        impl.saleWatermelon(customer);    //讨价还价成功，进行我们告诉代理商的卖瓜行为
    }
```

```
}
```

现在我们来试试看：

```
public class Main {
    public static void main(String[] args) {
        Shopper shopper = new ShopperProxy(new ShopperImpl());
        shopper.saleWatermelon("小强");
    }
}
```

这样的操作称为静态代理，也就是说我们需要提前知道接口的定义并进行实现才可以完成代理，而Mybatis这样的是无法预知代理接口的，我们就需要用到动态代理。

JDK提供的反射框架就为我们很好地解决了动态代理的问题，在这里相当于对JavaSE阶段反射的内容进行一个补充。

```
public class ShopperProxy implements InvocationHandler {

    Object target;
    public ShopperProxy(Object target){
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        String customer = (String) args[0];
        System.out.println(customer + "：哥们，这瓜多少钱一斤啊？");
        System.out.println("老板：两块钱一斤。");
        System.out.println(customer + "：你这瓜皮子是金子做的，还是瓜粒子是金子做的？");
        System.out.println("老板：你瞅瞅现在哪有瓜啊，这都是大棚的瓜，你嫌贵我还嫌贵呢。");
        System.out.println(customer + "：行，给我挑一个。");
        return method.invoke(target, args);
    }
}
```

通过实现InvocationHandler来成为一个动态代理，我们发现它提供了一个invoke方法，用于调用被代理对象的方法并完成我们的代理工作。现在就可以通过`Proxy.newProxyInstance`来生成一个动态代理类：

```
public static void main(String[] args) {
    Shopper impl = new ShopperImpl();
    Shopper shopper = (Shopper)
Proxy.newProxyInstance(impl.getClass().getClassLoader(),
            impl.getClass().getInterfaces(), new ShopperProxy(impl));
    shopper.saleWatermelon("小强");
    System.out.println(shopper.getClass());
}
```

通过打印类型我们发现，就是我们之前看到的那种奇怪的类：`class com.sun.proxy.$Proxy0`，因此Mybatis其实也是这样的来实现的（肯定有人问了：Mybatis是直接代理接口啊，你这个不还是要把接口实现了吗？）那我们来改改，现在我们不代理任何类了，直接做接口实现：

```
public class ShopperProxy implements InvocationHandler {

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        String customer = (String) args[0];
        System.out.println(customer + "：哥们，这瓜多少钱一斤啊？");
        System.out.println("老板：两块钱一斤。");
        System.out.println(customer + "：你这瓜皮子是金子做的，还是瓜粒子是金子做的？");
        System.out.println("老板：你瞅瞅现在哪有瓜啊，这都是大棚的瓜，你嫌贵我还嫌贵呢。");
        System.out.println(customer + "：行，给我挑一个。");
        return null;
    }
}
```

```
public static void main(String[] args) {
    Shopper shopper = (Shopper)
Proxy.newProxyInstance(Shopper.class.getClassLoader(),
            new Class[]{ Shopper.class },   //因为本身就是接口，所以直接用就行
            new ShopperProxy());
    shopper.saleWatermelon("小强");
    System.out.println(shopper.getClass());
}
```

我们可以去看看Mybatis的源码。

Mybatis的学习差不多就到这里为止了，不过，同样类型的框架还有很多，Mybatis属于半自动框架，SQL语句依然需要我们自己编写，虽然存在一定的麻烦，但是会更加灵活，而后面我们还会学习JPA，它是全自动的框架，你几乎见不到SQL的影子！

使用JUnit进行单元测试

首先一问：我们为什么需要单元测试？

随着我们的项目逐渐变大，比如我们之前编写的图书管理系统，我们都是边在写边在测试，而我们当时使用的测试方法，就是直接在主方法中运行测试，但是，在很多情况下，我们的项目可能会很庞大，不可能每次都去完整地启动一个项目来测试某一个功能，这样显然会降低我们的开发效率，因此，我们需要使用单元测试来帮助我们针对于某个功能或是某个模块单独运行代码进行测试，而不是启动整个项目。

同时，在我们项目的维护过程中，难免会涉及到一些原有代码的修改，很有可能出现改了代码导致之前的功能出现问题（牵一发而动全身），而我们又不一定能立即察觉到，因此，我们可以提前保存一些测试用例，每次完成代码后都可以跑一遍测试用例，来确保之前的功能没有因为后续的修改而出现问题。

我们还可以利用单元测试来评估某个模块或是功能的耗时和性能，快速排查导致程序运行缓慢的问题，这些都可以通过单元测试来完成，可见单元测试对于开发的重要性。

尝试JUnit

首先需要导入JUnit依赖，我们在这里使用Junit4进行介绍，最新的Junit5放到Maven板块一起讲解，Jar包已经放在视频下方简介中，直接去下载即可。同时IDEA需要安装JUnit插件（默认是已经捆绑安装的，因此无需多余配置）

现在我们创建一个新的类，来编写我们的单元测试用例：

```
public class TestMain {
    @Test
    public void method(){
        System.out.println("我是测试用例1");
    }

    @Test
    public void method2(){
        System.out.println("我是测试用例2");
    }
}
```

我们可以点击类前面的测试按钮，或是单个方法前的测试按钮，如果点击类前面的测试按钮，会执行所有的测试用例。

运行测试后，我们发现控制台得到了一个测试结果，显示为绿色表示测试通过。

只需要通过打上`@Test`注解，即可将一个方法标记为测试案例，我们可以直接运行此测试案例，但是我们编写的测试方法有以下要求：

- 方法必须是public的
- 不能是静态方法
- 返回值必须是void
- 必须是没有任何参数的方法

对于一个测试案例来说，我们肯定希望测试的结果是我们所期望的一个值，因此，如果测试的结果并不是我们所期望的结果，那么这个测试就应该没有成功通过！

我们可以通过断言工具类来进行判定：

```
public class TestMain {
    @Test
    public void method(){
        System.out.println("我是测试案例！");
        Assert.assertEquals(1, 2);    //参数1是期盼值，参数2是实际测试结果值
    }
}
```

通过运行代码后，我们发现测试过程中抛出了一个错误，并且IDEA给我们显示了期盼结果和测试结果，那么现在我们来测试一个案例，比如我们想查看冒泡排序的编写是否正确：

```
@Test
public void method(){
    int[] arr = {0, 4, 5, 2, 6, 9, 3, 1, 7, 8};

    //错误的冒泡排序
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - 1 - i; j++) {
            if(arr[j] > arr[j + 1]){
                int tmp = arr[j];
                arr[j] = arr[j+1];
                // arr[j+1] = tmp;
            }
```

```
        }
    }

    Assert.assertArrayEquals(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, arr);
}
```

通过测试，我们发现得到的结果并不是我们想要的结果，因此现在我们需要去修改为正确的冒泡排序，修改后，测试就能正确通过了。我们还可以再通过一个案例来更加深入地了解测试，现在我们想测试从数据库中取数据是否为我们预期的数据：

```
@Test
public void method(){
    try (SqlSession sqlSession = MybatisUtil.getSession(true)){
        TestMapper mapper = sqlSession.getMapper(TestMapper.class);
        Student student = mapper.getStudentBySidAndSex(1, "男");

        Assert.assertEquals(new Student().setName("小明").setSex("男").setSid(1),
student);
    }
}
```

那么如果我们在进行所有的测试之前需要做一些前置操作该怎么办呢，一种办法是在所有的测试用例前面都加上前置操作，但是这样显然是很冗余的，因为一旦发生修改就需要挨个进行修改，因此我们需要更加智能的方法，我们可以通过`@Before`注解来添加测试用例开始之前的前置操作：

```
public class TestMain {

    private SqlSessionFactory sqlSessionFactory;
    @Before
    public void before(){
        System.out.println("测试前置正在初始化...");
        try {
            sqlSessionFactory = new SqlSessionFactoryBuilder()
                    .build(new FileInputStream("mybatis-config.xml"));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        System.out.println("测试初始化完成，正在开始测试案例...");
    }

    @Test
    public void method1(){
        try (SqlSession sqlSession = sqlSessionFactory.openSession(true)){
            TestMapper mapper = sqlSession.getMapper(TestMapper.class);
            Student student = mapper.getStudentBySidAndSex(1, "男");

            Assert.assertEquals(new Student().setName("小
明").setSex("男").setSid(1), student);
            System.out.println("测试用例1通过！");
        }
    }

    @Test
    public void method2(){
```

```
        try (SqlSession sqlSession = sqlSessionFactory.openSession(true)){
            TestMapper mapper = sqlSession.getMapper(TestMapper.class);
            Student student = mapper.getStudentBySidAndSex(2, "女");

            Assert.assertEquals(new Student().setName("小
红").setSex("女").setSid(2), student);
            System.out.println("测试用例2通过！");
        }
    }
}
```

同理，在所有的测试完成之后，我们还想添加一个收尾的动作，那么只需要使用`@After`注解即可添加结束动作：

```
@After
public void after(){
    System.out.println("测试结束，收尾工作正在进行...");
}
```

有关JUnit的使用我们就暂时只介绍这么多。

JUL日志系统

首先一问：我们为什么需要日志系统？

我们之前一直都在使用`System.out.println`来打印信息，但是，如果项目中存在大量的控制台输出语句，会显得很凌乱，而且日志的粒度是不够细的，假如我们现在希望，项目只在debug的情况下打印某些日志，而在实际运行时不打印日志，采用直接输出的方式就很难实现了，因此我们需要使用日志框架来规范化日志输出。

而JDK为我们提供了一个自带的日志框架，位于`java.util.logging`包下，我们可以使用此框架来实现日志的规范化打印，使用起来非常简单：

```
public class Main {
    public static void main(String[] args) {
        // 首先获取日志打印器
        Logger logger = Logger.getLogger(Main.class.getName());
        // 调用info来输出一个普通的信息，直接填写字符串即可
        logger.info("我是普通的日志");
    }
}
```

我们可以在主类中使用日志打印，得到日志的打印结果：

```
十一月 15, 2021 12:55:37 下午 com.test.Main main
信息: 我是普通的日志
```

我们发现，通过日志输出的结果会更加规范。

JUL日志讲解

日志分为7个级别，详细信息我们可以在Level类中查看：

- SEVERE（最高值）- 一般用于代表严重错误
- WARNING - 一般用于表示某些警告，但是不足以判断为错误
- INFO （默认级别） - 常规消息
- CONFIG
- FINE
- FINER
- FINEST（最低值）

我们之前通过`info`方法直接输出的结果就是使用的默认级别的日志，我们可以通过`log`方法来设定该条日志的输出级别：

```
public static void main(String[] args) {
    Logger logger = Logger.getLogger(Main.class.getName());
    logger.log(Level.SEVERE, "严重的错误", new IOException("我就是错误"));
    logger.log(Level.WARNING, "警告的内容");
    logger.log(Level.INFO, "普通的信息");
    logger.log(Level.CONFIG, "级别低于普通信息");
}
```

我们发现，级别低于默认级别的日志信息，无法输出到控制台，我们可以通过设置来修改日志的打印级别：

```
public static void main(String[] args) {
    Logger logger = Logger.getLogger(Main.class.getName());

    //修改日志级别
    logger.setLevel(Level.CONFIG);
    //不使用父日志处理器
    logger.setUseParentHandlers(false);
    //使用自定义日志处理器
    ConsoleHandler handler = new ConsoleHandler();
    handler.setLevel(Level.CONFIG);
    logger.addHandler(handler);

    logger.log(Level.SEVERE, "严重的错误", new IOException("我就是错误"));
    logger.log(Level.WARNING, "警告的内容");
    logger.log(Level.INFO, "普通的信息");
    logger.log(Level.CONFIG, "级别低于普通信息");
}
```

每个`Logger`都有一个父日志打印器，我们可以通过`getParent()`来获取：

```
public static void main(String[] args) throws IOException {
    Logger logger = Logger.getLogger(Main.class.getName());
    System.out.println(logger.getParent().getClass());
}
```

我们发现，得到的是`java.util.logging.LogManager$RootLogger`这个类，它默认使用的是ConsoleHandler，且日志级别为INFO，由于每一个日志打印器都会直接使用父类的处理器，因此我们之前需要关闭父类然后使用我们自己的处理器。

我们通过使用自己日志处理器来自定义级别的信息打印到控制台，当然，日志处理器不仅仅只有控制台打印，我们也可以使用文件处理器来处理日志信息，我们继续添加一个处理器：

```
//添加输出到本地文件
FileHandler fileHandler = new FileHandler("test.log");
fileHandler.setLevel(Level.WARNING);
logger.addHandler(fileHandler);
```

注意，这个时候就有两个日志处理器了，因此控制台和文件的都会生效。如果日志的打印格式我们不喜欢，我们还可以自定义打印格式，比如我们控制台处理器就默认使用的是`SimpleFormatter`，而文件处理器则是使用的`XMLFormatter`，我们可以自定义：

```
//使用自定义日志处理器(控制台)
ConsoleHandler handler = new ConsoleHandler();
handler.setLevel(Level.CONFIG);
handler.setFormatter(new XMLFormatter());
logger.addHandler(handler);
```

我们可以直接配置为想要的打印格式，如果这些格式还不能满足你，那么我们也可以自行实现：

```
public static void main(String[] args) throws IOException {
    Logger logger = Logger.getLogger(Main.class.getName());
    logger.setUseParentHandlers(false);

    //为了让颜色变回普通的颜色，通过代码块在初始化时将输出流设定为System.out
    ConsoleHandler handler = new ConsoleHandler(){{
        setOutputStream(System.out);
    }};
    //创建匿名内部类实现自定义的格式
    handler.setFormatter(new Formatter() {
        @Override
        public String format(LogRecord record) {
            SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss.SSS");
            String time = format.format(new Date(record.getMillis()));  //格式化日
志时间
            String level = record.getLevel().getName();  // 获取日志级别名称
            // String level = record.getLevel().getLocalizedName();   // 获取本地
化名称（语言跟随系统）
            String thread = String.format("%10s",
Thread.currentThread().getName());  //线程名称（做了格式化处理，留出10格空间）
            long threadID = record.getThreadID();   //线程ID
            String className = String.format("%-20s",
record.getSourceClassName());  //发送日志的类名
            String msg = record.getMessage();   //日志消息

          //\033[33m作为颜色代码，30~37都有对应的颜色，38是没有颜色，IDEA能显示，但是某些地
方可能不支持
            return "\033[38m" + time + "  \033[33m" + level + " \033[35m" +
threadID
```

```
                    + "\033[38m --- [" + thread + "] \033[36m" + className +
"\033[38m : " + msg + "\n";
        }
    });
    logger.addHandler(handler);

    logger.info("我是测试消息1...");
    logger.log(Level.INFO, "我是测试消息2...");
    logger.log(Level.WARNING, "我是测试消息3...");
}
```

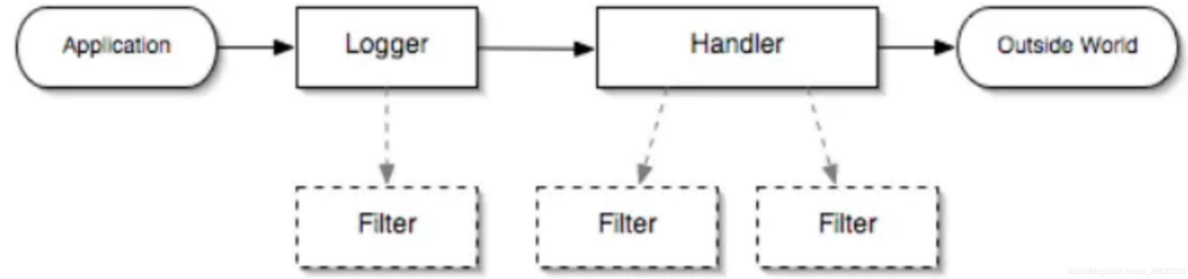
日志可以设置过滤器，如果我们不希望某些日志信息被输出，我们可以配置过滤规则：

```
public static void main(String[] args) throws IOException {
    Logger logger = Logger.getLogger(Main.class.getName());

    //自定义过滤规则
    logger.setFilter(record -> !record.getMessage().contains("普通"));

    logger.log(Level.SEVERE, "严重的错误", new IOException("我就是错误"));
    logger.log(Level.WARNING, "警告的内容");
    logger.log(Level.INFO, "普通的信息");
}
```

实际上，整个日志的输出流程如下：



Properties配置文件

Properties文件是Java的一种配置文件，我们之前学习了XML，但是我们发现XML配置文件读取实在是太麻烦，那么能否有一种简单一点的配置文件呢？我们可以使用Properties文件：

```
name=Test
desc=Description
```

该文件配置很简单，格式为`配置项=配置值`，我们可以直接通过`Properties`类来将其读取为一个类似于Map一样的对象：

```
public static void main(String[] args) throws IOException {
    Properties properties = new Properties();
    properties.load(new FileInputStream("test.properties"));
    System.out.println(properties);
}
```

我们发现，`Properties`类是继承自`Hashtable`，而`Hashtable`是实现的Map接口，也就是说，`Properties`本质上就是一个Map一样的结构，它会把所有的配置项映射为一个Map，这样我们就可以快速地读取对应配置的值了。

我们也可以将已经存在的Properties对象放入输出流进行保存，我们这里就不保存文件了，而是直接打印到控制台，我们只需要提供输出流即可：

```
public static void main(String[] args) throws IOException {
    Properties properties = new Properties();
    // properties.setProperty("test", "lbwnb");  //和put效果一样
    properties.put("test", "lbwnb");
    properties.store(System.out, "????");
    //properties.storeToXML(System.out, "????");  保存为XML格式
}
```

我们可以通过`System.getProperties()`获取系统的参数，我们来看看：

```
public static void main(String[] args) throws IOException {
    System.getProperties().store(System.out, "系统信息：");
}
```

编写日志配置文件

我们可以通过进行配置文件来规定日志打印器的一些默认值：

```
# RootLogger 的默认处理器为
handlers= java.util.logging.ConsoleHandler
# RootLogger 的默认的日志级别
.level= CONFIG
```

我们来尝试使用配置文件来进行配置：

```
public static void main(String[] args) throws IOException {
    //获取日志管理器
    LogManager manager = LogManager.getLogManager();
    //读取我们自己的配置文件
    manager.readConfiguration(new FileInputStream("logging.properties"));
    //再获取日志打印器
    Logger logger = Logger.getLogger(Main.class.getName());
    logger.log(Level.CONFIG, "我是一条日志信息");   //通过自定义配置文件，我们发现默认级别不再是INFO了
}
```

我们也可以去修改`ConsoleHandler`的默认配置：

```
# 指定默认日志级别
java.util.logging.ConsoleHandler.level = ALL
# 指定默认日志消息格式
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
# 指定默认的字符集
java.util.logging.ConsoleHandler.encoding = UTF-8
```

其实，我们阅读`ConsoleHandler`的源码就会发现，它就是通过读取配置文件来进行某些参数设置：

```
// Private method to configure a ConsoleHandler from LogManager
// properties and/or default values as specified in the class
// javadoc.
private void configure() {
    LogManager manager = LogManager.getLogManager();
    String cname = getClass().getName();

    setLevel(manager.getLevelProperty(cname +".level", Level.INFO));
    setFilter(manager.getFilterProperty(cname +".filter", null));
    setFormatter(manager.getFormatterProperty(cname +".formatter", new
SimpleFormatter()));
    try {
        setEncoding(manager.getStringProperty(cname +".encoding", null));
    } catch (Exception ex) {
        try {
            setEncoding(null);
        } catch (Exception ex2) {
            // doing a setEncoding with null should always work.
            // assert false;
        }
    }
}
```

使用Lombok快速开启日志

我们发现，如果我们现在需要全面使用日志系统，而不是传统的直接打印，那么就需要在每个类都去编写获取Logger的代码，这样显然是很冗余的，能否简化一下这个流程呢?

前面我们学习了Lombok，我们也体会到Lombok给我们带来的便捷，我们可以通过一个注解快速生成构造方法、Getter和Setter，同样的，Logger也是可以使用Lombok快速生成的。

```
@Log
public class Main {
    public static void main(String[] args) {
        System.out.println("自动生成的Logger名称："+log.getName());
        log.info("我是日志信息");
    }
}
```

只需要添加一个`@Log`注解即可，添加后，我们可以直接使用一个静态变量log，而它就是自动生成的Logger。我们也可以手动指定名称：

```
@Log(topic = "打工是不可能打工的")
public class Main {
    public static void main(String[] args) {
        System.out.println("自动生成的Logger名称："+log.getName());
        log.info("我是日志信息");
    }
}
```

Mybatis日志系统

Mybatis也有日志系统，它详细记录了所有的数据库操作等，但是我们在前面的学习中没有开启它，现在我们学习了日志之后，我们就可以尝试开启Mybatis的日志系统，来监控所有的数据库操作，要开启日志系统，我们需要进行配置：

```
<setting name="logImpl" value="STDOUT_LOGGING" />
```

`logImpl`包括很多种配置项，包括 SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING，而默认情况下是未配置，也就是说不打印。我们这里将其设定为STDOUT_LOGGING表示直接使用标准输出将日志信息打印到控制台，我们编写一个测试案例来看看效果：

```
public class TestMain {

    private SqlSessionFactory sqlSessionFactory;
    @Before
    public void before(){
        try {
            sqlSessionFactory = new SqlSessionFactoryBuilder()
                    .build(new FileInputStream("mybatis-config.xml"));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    @Test
    public void test(){
        try(SqlSession sqlSession = sqlSessionFactory.openSession(true)){
            TestMapper mapper = sqlSession.getMapper(TestMapper.class);
            System.out.println(mapper.getStudentBySidAndSex(1, "男"));
            System.out.println(mapper.getStudentBySidAndSex(1, "男"));
        }
    }
}
```

我们发现，两次获取学生信息，只有第一次打开了数据库连接，而第二次并没有。

现在我们学习了日志系统，那么我们来尝试使用日志系统输出Mybatis的日志信息：

```
<setting name="logImpl" value="JDK_LOGGING" />
```

将其配置为JDK_LOGGING表示使用JUL进行日志打印，因为Mybatis的日志级别都比较低，因此我们需要设置一下`logging.properties`默认的日志级别：

```
handlers= java.util.logging.ConsoleHandler
.level= ALL
java.util.logging.ConsoleHandler.level = ALL
```

代码编写如下：

```
@Log
```

```
public class TestMain {

    private SqlSessionFactory sqlSessionFactory;
    @Before
    public void before(){
        try {
            sqlSessionFactory = new SqlSessionFactoryBuilder()
                    .build(new FileInputStream("mybatis-config.xml"));
            LogManager manager = LogManager.getLogManager();
            manager.readConfiguration(new
FileInputStream("logging.properties"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Test
    public void test(){
        try(SqlSession sqlSession = sqlSessionFactory.openSession(true)){
            TestMapper mapper = sqlSession.getMapper(TestMapper.class);
            log.info(mapper.getStudentBySidAndSex(1, "男").toString());
            log.info(mapper.getStudentBySidAndSex(1, "男").toString());
        }
    }
}
```

但是我们发现，这样的日志信息根本没法看，因此我们需要修改一下日志的打印格式，我们自己创建一个格式化类：

```
public class TestFormatter extends Formatter {
    @Override
    public String format(LogRecord record) {
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss.SSS");
        String time = format.format(new Date(record.getMillis()));  //格式化日志时
间
        return time + " : " + record.getMessage() + "\n";
    }
}
```

现在再来修改一下默认的格式化实现：

```
handlers= java.util.logging.ConsoleHandler
.level= ALL
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = com.test.TestFormatter
```

现在就好看多了，当然，我们还可以继续为Mybatis添加文件日志，这里就不做演示了。

使用Maven管理项目

注意：开始之前，看看你C盘空间够不够，最好预留2GB空间以上！

吐槽：很多电脑预装系统C盘都给得巨少，就算不装软件，一些软件的缓存文件也能给你塞满，建议有时间重装一下系统重新分配一下磁盘空间。

Maven 翻译为"专家"、"内行"，是 Apache 下的一个纯 Java 开发的开源项目。基于项目对象模型（缩写：POM）概念，Maven利用一个中央信息片断能管理一个项目的构建、报告和文档等步骤。Maven 是一个项目管理工具，可以对 Java 项目进行构建、依赖管理。Maven 也可被用于构建和管理各种项目，例如 C#，Ruby，Scala 和其他语言编写的项目。Maven 曾是 Jakarta 项目的子项目，现为由 Apache 软件基金会主持的独立 Apache 项目。

通过Maven，可以帮助我们做：

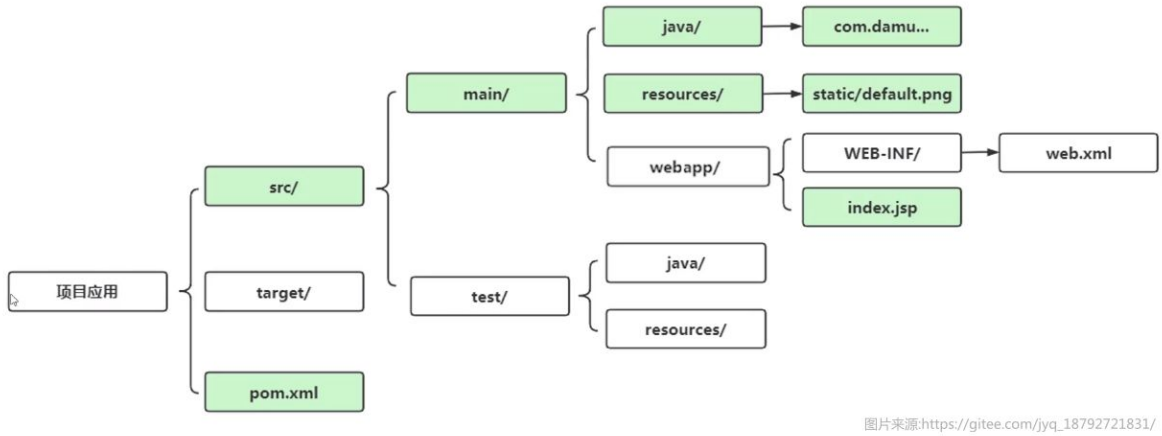
- 项目的自动构建，包括代码的编译、测试、打包、安装、部署等操作。
- 依赖管理，项目使用到哪些依赖，可以快速完成导入。

我们之前并没有讲解如何将我们的项目打包为Jar文件运行，同时，我们导入依赖的时候，每次都要去下载对应的Jar包，这样其实是很麻烦的，并且还有可能一个Jar包依赖于另一个Jar包，就像之前使用JUnit一样，因此我们需要一个更加方便的包管理机制。

Maven也需要安装环境，但是IDEA已经自带了Maven环境，因此我们不需要再去进行额外的环境安装（无IDEA也能使用Maven，但是配置过程很麻烦，并且我们现在使用的都是IDEA的集成开发环境，所以这里就不讲解Maven命令行操作了）我们直接创建一个新的Maven项目即可。

Maven项目结构

我们可以来看一下，一个Maven项目和我们普通的项目有什么区别：



图片来源:https://gitee.com/jyq_18792721831/

那么首先，我们需要了解一下POM文件，它相当于是我们整个Maven项目的配置文件，它也是使用XML编写的：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>MavenTest</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
```

```
    </properties>

</project>
```

我们可以看到，Maven的配置文件是以`project`为根节点，而`modelVersion`定义了当前模型的版本，一般是4.0.0，我们不用去修改。

`groupId`、`artifactId`、`version`这三个元素合在一起，用于唯一区别每个项目，别人如果需要将我们编写的代码作为依赖，那么就必须通过这三个元素来定位我们的项目，我们称为一个项目的基本坐标，所有的项目一般都有自己的Maven坐标，因此我们通过Maven导入其他的依赖只需要填写这三个基本元素就可以了，无需再下载Jar文件，而是Maven自动帮助我们下载依赖并导入。

- `groupId` 一般用于指定组名称，命名规则一般和包名一致，比如我们这里使用的是`org.example`，一个组下面可以有很多个项目。
- `artifactId` 一般用于指定项目在当前组中的唯一名称，也就是说在组中用于区分于其他项目的标记。
- `version` 代表项目版本，随着我们项目的开发和改进，版本号也会不断更新，就像LOL一样，每次赛季更新都会有一个大版本更新，我们的Maven项目也是这样，我们可以手动指定当前项目的版本号，其他人使用我们的项目作为依赖时，也可以根本版本号进行选择（这里的SNAPSHOT代表快照，一般表示这是一个处于开发中的项目，正式发布项目一般只带版本号）

`properties`中一般都是一些变量和选项的配置，我们这里指定了JDK的源代码和编译版本为1.8，无需进行修改。

Maven依赖导入

现在我们尝试使用Maven来帮助我们快速导入依赖，我们需要导入之前的JDBC驱动依赖、JUnit依赖、Mybatis依赖、Lombok依赖，那么如何使用Maven来管理依赖呢？

我们可以创建一个`dependencies`节点：

```
<dependencies>
    //里面填写的就是所有的依赖
</dependencies>
```

那么现在就可以向节点中填写依赖了，那么我们如何知道每个依赖的坐标呢？我们可以在：https://mvnrepository.com/ 进行查询（可能打不开，建议用流量，或是直接百度某个项目的Maven依赖），我们直接搜索lombok即可，打开后可以看到已经给我们写出了依赖的坐标：

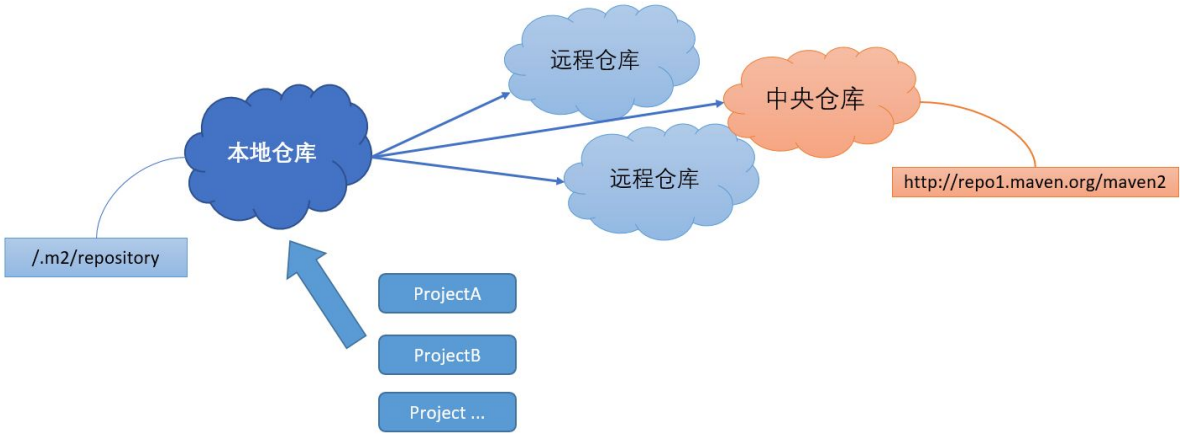
```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.22</version>
    <scope>provided</scope>
</dependency>
```

我们直接将其添加到`dependencies`节点中即可，现在我们来编写一个测试用例看看依赖导入成功了没有：

```
public class Main {
    public static void main(String[] args) {
        Student student = new Student("小明", 18);
        System.out.println(student);
    }
}
```

```
@Data
@AllArgsConstructor
public class Student {
    String name;
    int age;
}
```

项目运行成功，表示成功导入了依赖。那么，Maven是如何进行依赖管理呢，以致于如此便捷的导入依赖，我们来看看Maven项目的依赖管理流程：



通过流程图我们得知，一个项目依赖一般是存储在中央仓库中，也有可能存储在一些其他的远程仓库（私服），几乎所有的依赖都被放到了中央仓库中，因此，Maven可以直接从中央仓库中下载大部分的依赖（Maven第一次导入依赖是需要联网的），远程仓库中下载之后，会暂时存储在本地仓库，我们会发现我们本地存在一个`.m2`文件夹，这就是Maven本地仓库文件夹，默认建立在C盘，如果你C盘空间不足，会出现问题！

在下次导入依赖时，如果Maven发现本地仓库中就已经存在某个依赖，那么就不会再去远程仓库下载了。

可能在导入依赖时，小小伙伴们会出现卡顿的问题，我们建议配置一下IDEA自带的Maven插件远程仓库地址，我们打开IDEA的安装目录，找到`安装根目录/plugins/maven/lib/maven3/conf`文件夹，找到`settings.xml`文件，打开编辑：

找到mirros标签，添加以下内容：

```
<mirror>
      <id>nexus-aliyun</id>
      <mirrorOf>*</mirrorOf>
      <name>Nexus aliyun</name>
      <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

这样，我们就将默认的远程仓库地址（国外），配置为国内的阿里云仓库地址了（依赖的下载速度就会快起来了）

Maven依赖作用域

除了三个基本的属性用于定位坐标外，依赖还可以添加以下属性：

- **type**：依赖的类型，对于项目坐标定义的packaging。大部分情况下，该元素不必声明，其默认值为jar
- **scope**：依赖的范围（作用域，着重讲解）
- **optional**：标记依赖是否可选
- **exclusions**：用来排除传递性依赖（一个项目有可能依赖于其他项目，就像我们的项目，如果别人要用我们的项目作为依赖，那么就需要一起下载我们项目的依赖，如Lombok）

我们着重来讲解一下`scope`属性，它决定了依赖的作用域范围：

- **compile** ： 为默认的依赖有效范围。如果在定义依赖关系的时候，没有明确指定依赖有效范围的话，则默认采用该依赖有效范围。此种依赖，在编译、运行、测试时均有效。
- **provided** ： 在编译、测试时有效，但是在运行时无效，也就是说，项目在运行时，不需要此依赖，比如我们上面的Lombok，我们只需要在编译阶段使用它，编译完成后，实际上已经转换为对应的代码了，因此Lombok不需要在项目运行时也存在。
- **runtime** ： 在运行、测试时有效，但是在编译代码时无效。比如我们如果需要自己写一个JDBC实现，那么肯定要用到JDK为我们指定的接口，但是实际上在运行时是不用自带JDK的依赖，因此只保留我们自己写的内容即可。
- **test** ： 只在测试时有效，例如：JUnit，我们一般只会在测试阶段使用JUnit，而实际项目运行时，我们就用不到测试了，那么我们来看看，导入JUnit的依赖：

同样的，我们可以在网站上搜索Junit的依赖，我们这里导入最新的JUnit5作为依赖：

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
```

我们所有的测试用例全部编写到Maven项目给我们划分的test目录下，位于此目录下的内容不会在最后被打包到项目中，只用作开发阶段测试使用：

```
public class MainTest {

    @Test
    public void test(){
        System.out.println("测试");
        //Assert在JUnit5时名称发生了变化Assertions
        Assertions.assertArrayEquals(new int[]{1, 2, 3}, new int[]{1, 2});
    }
}
```

因此，一般仅用作测试的依赖如Unit只保留在测试中即可，那么现在我们再来添加JDBC和Mybatis的依赖：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.27</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.7</version>
</dependency>
```

我们发现，Maven还给我们提供了一个`resource`文件夹，我们可以将一些静态资源，比如配置文件，放入到这个文件夹中，项目在打包时会将资源文件夹中文件一起打包的jar中，比如我们在这里编写一个Mybatis的配置文件：

```
<?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>
    <settings>
        <setting name="mapUnderscoreToCamelCase" value="true"/>
        <setting name="cacheEnabled" value="true"/>
        <setting name="logImpl" value="JDK_LOGGING" />
    </settings>
    <!-- 需要在environments的上方 -->
    <typeAliases>
        <package name="com.test.entity"/>
    </typeAliases>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/study"/>
                <property name="username" value="test"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper class="com.test.mapper.TestMapper"/>
    </mappers>
</configuration>
```

现在我们创建一下测试用例，顺便带大家了解一下Junit5的一些比较方便的地方：

```
public class MainTest {

    //因为配置文件位于内部，我们需要使用Resources类的getResourceAsStream来获取内部的资源文件
    private static SqlSessionFactory factory;

    //在JUnit5中@Before被废弃，它被细分了：
```

```
    @BeforeAll // 一次性开启所有测试案例只会执行一次 (方法必须是static)
    // @BeforeEach 一次性开启所有测试案例每个案例开始之前都会执行一次
    @SneakyThrows
    public static void before(){
        factory = new SqlSessionFactoryBuilder()
                .build(Resources.getResourceAsStream("mybatis.xml"));
    }


    @DisplayName("Mybatis数据库测试")  //自定义测试名称
    @RepeatedTest(3)  //自动执行多次测试
    public void test(){
        try (SqlSession sqlSession = factory.openSession(true)){
            TestMapper testMapper = sqlSession.getMapper(TestMapper.class);
            System.out.println(testMapper.getStudentBySid(1));
        }
    }
}
```

那么就有人提问了，如果我需要的依赖没有上传的远程仓库，而是只有一个Jar怎么办呢？我们可以使用第四种作用域：

- **system**：作用域和provided是一样的，但是它不是从远程仓库获取，而是直接导入本地Jar包：

```
<dependency>
     <groupId>javax.jntm</groupId>
     <artifactId>lbwnb</artifactId>
     <version>2.0</version>
     <scope>system</scope>
     <systemPath>C://学习资料/4K高清无码/test.jar</systemPath>
</dependency>
```

比如上面的例子，如果scope为system，那么我们需要添加一个systemPath来指定jar文件的位置，这里就不再演示了。

Maven可选依赖

当项目中的某些依赖不希望被使用此项目作为依赖的项目使用时，我们可以给依赖添加`optional`标签表示此依赖是可选的，默认在导入依赖时，不会导入可选的依赖：

```
<optional>true</optional>
```

比如Mybatis的POM文件中，就存在大量的可选依赖：

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.30</version>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.30</version>
```

```
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
  <optional>true</optional>
</dependency>
 ...
```

由于Mybatis要支持多种类型的日志，需要用到很多种不同的日志框架，因此需要导入这些依赖来做兼容，但是我们项目中并不一定会使用这些日志框架作为Mybatis的日志打印器，因此这些日志框架仅Mybatis内部做兼容需要导入使用，而我们可以选择不使用这些框架或是选择其中一个即可，也就是说我们导入Mybatis之后想用什么日志框架再自己加就可以了。

Maven排除依赖

我们了解了可选依赖，现在我们可以让使用此项目作为依赖的项目默认不使用可选依赖，但是如果存在那种不是可选依赖，但是我们导入此项目有不希望使用此依赖该怎么办呢，这个时候我们就可以通过排除依赖来防止添加不必要的依赖：

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

我们这里演示了排除JUnit的一些依赖，我们可以在外部库中观察排除依赖之后和之前的效果。

Maven继承关系

一个Maven项目可以继承自另一个Maven项目，比如多个子项目都需要父项目的依赖，我们就可以使用继承关系来快速配置。

我们右键左侧栏，新建一个模块，来创建一个子项目：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>MavenTest</artifactId>
        <groupId>org.example</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
```

```
    <artifactId>ChildModel</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

</project>
```

我们可以看到，IDEA默认给我们添加了一个parent节点，表示此Maven项目是父Maven项目的子项目，子项目直接继承父项目的`groupId`，子项目会直接继承父项目的所有依赖，除非依赖添加了optional标签，我们来编写一个测试用例尝试一下:

```
import lombok.extern.java.Log;

@Log
public class Main {
    public static void main(String[] args) {
        log.info("我是日志信息");
    }
}
```

可以看到，子项目也成功继承了Lombok依赖。

我们还可以让父Maven项目统一管理所有的依赖，包括版本号等，子项目可以选取需要的作为依赖，而版本全由父项目管理，我们可以将`dependencies`全部放入`dependencyManagement`节点，这样父项目就完全作为依赖统一管理。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.27</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.7</version>
        </dependency>
    </dependencies>
```

```
</dependencyManagement>
```

我们发现，子项目的依赖失效了，因为现在父项目没有依赖，而是将所有的依赖进行集中管理，子项目需要什么再拿什么即可，同时子项目无需指定版本，所有的版本全部由父项目决定，子项目只需要使用即可：

```
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

当然，父项目如果还存在dependencies节点的话，里面的内依赖依然是直接继承：

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.8.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
      ...
```

Maven常用命令

我们可以看到在IDEA右上角Maven板块中，每个Maven项目都有一个生命周期，实际上这些是Maven的一些插件，每个插件都有各自的功能，比如：

- `clean`命令，执行后会清理整个`target`文件夹，在之后编写Springboot项目时可以解决一些缓存没更新的问题。
- `validate`命令可以验证项目的可用性。
- `compile`命令可以将项目编译为.class文件。
- `install`命令可以将当前项目安装到本地仓库，以供其他项目导入作为依赖使用
- `verify`命令可以按顺序执行每个默认生命周期阶段（`validate`，`compile`，`package`等）

Maven测试项目

通过使用`test`命令，可以一键测试所有位于test目录下的测试案例，请注意有以下要求：

- 测试类的名称必须是以`Test`结尾，比如`MainTest`
- 测试方法上必须标注`@Test`注解，实测`@RepeatedTest`无效

这是由于JUnit5比较新，我们需要重新配置插件升级到高版本，才能完美的兼容Junit5：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <!-- JUnit 5 requires Surefire version 2.22.0 or higher -->
            <version>2.22.0</version>
        </plugin>
    </plugins>
</build>
```

现在`@RepeatedTest`、`@BeforeAll`也能使用了。

Maven打包项目

我们的项目在编写完成之后，要么作为Jar依赖，供其他模型使用，要么就作为一个可以执行的程序，在控制台运行，我们只需要直接执行`package`命令就可以直接对项目的代码进行打包，生成jar文件。

当然，以上方式仅适用于作为Jar依赖的情况，如果我们需要打包一个可执行文件，那么我不仅需要将自己编写的类打包到Jar中，同时还需要将依赖也一并打包到Jar中，因为我们使用了别人为我们通过的框架，自然也需要运行别人的代码，我们需要使用另一个插件来实现一起打包：

```
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.1.0</version>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>com.test.Main</mainClass>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

在打包之前也会执行一次test命令，来保证项目能够正常运行，当测试出现问题时，打包将无法完成，我们也可以手动跳过，选择`执行Maven目标`来手动执行Maven命令，输入`mvn package -Dmaven.test.skip=true`来以跳过测试的方式进行打包。

最后得到我们的Jar文件，在同级目录下输入`java -jar xxxx.jar`来运行我们打包好的Jar可执行程序（xxx代表文件名称）

- `deploy`命令用于发布项目到本地仓库和远程仓库，一般情况下用不到，这里就不做讲解了。
- `site`命令用于生成当前项目的发布站点，暂时不需要了解。

我们之前还讲解了多模块项目，那么多模块下父项目存在一个`packing`打包类型标签，所有的父级项目的packing都为pom，packing默认是jar类型，如果不作配置，maven会将该项目打成jar包。作为父级项目，还有一个重要的属性，那就是modules，通过modules标签将项目的所有子项目引用进来，在build父级项目时，会根据子模块的相互依赖关系整理一个build顺序，然后依次build。

实战：基于Mybatis+JUL+Lombok+Maven的图书管理系统（带单元测试）

项目需求：

- 在线录入学生信息和书籍信息
- 查询书籍信息列表
- 查询学生信息列表
- 查询借阅信息列表
- 完整的日志系统