

Maven&MyBatis

目标

- 能够使用Maven进行项目的管理
- 能够完成Mybatis代理方式查询数据
- 能够理解Mybatis核心配置文件的配置

1. Maven

Maven是专门用于管理和构建Java项目的工具，它的主要功能有：

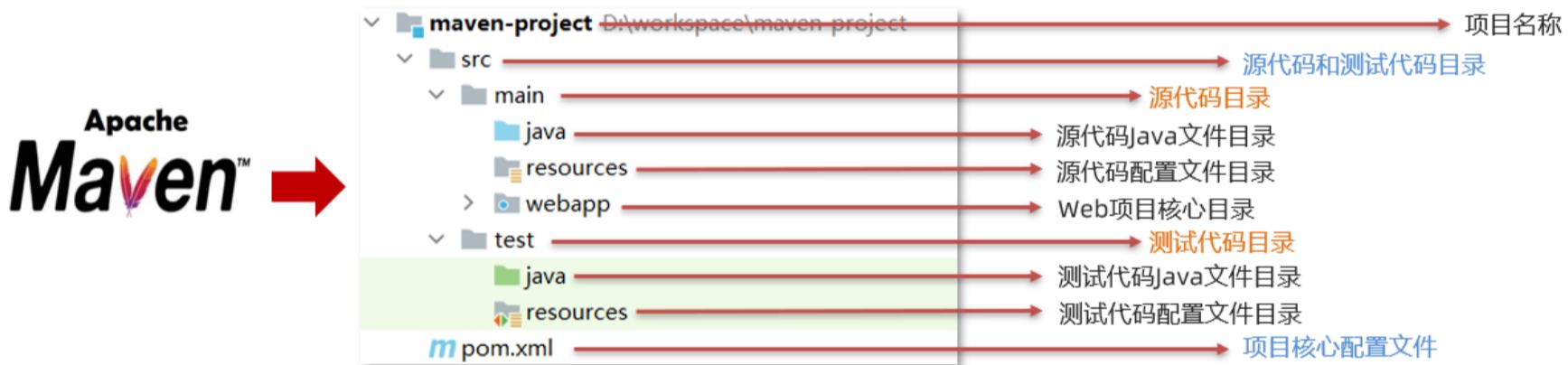
- 提供了一套标准化的项目结构
- 提供了一套标准化的构建流程（编译，测试，打包，发布……）
- 提供了一套依赖管理机制

标准化的项目结构：

项目结构我们都知道，每一个开发工具（IDE）都有自己不同的项目结构，它们互相之间不通用。我再eclipse中创建的目录，无法在idea中进行使用，这就造成了很大的不方便，如下图：前两个是以后开发经常使用的开发工具



而Maven提供了一套标准化的项目结构，所有的IDE使用Maven构建的项目完全一样，所以IDE创建的Maven项目可以通用。如下图右边就是Maven构建的项目结构。



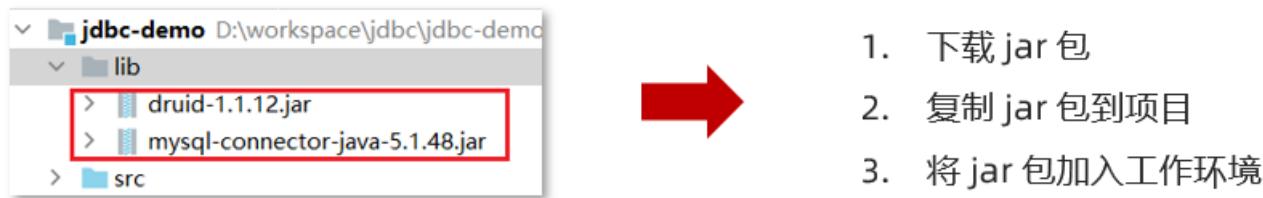
标准化的构建流程：



如上图所示我们开发了一套系统，代码需要进行编译、测试、打包、发布，这些操作如果需要反复进行就显得特别麻烦，而Maven提供了一套简单的命令来完成项目构建。

依赖管理：

依赖管理其实就是管理你项目所依赖的第三方资源（jar包、插件）。如之前我们项目中需要使用JDBC和Druid的话，就需要去网上下载对应的依赖包（当前之前是老师已经下载好提供给大家了），复制到项目中，还要将jar包加入工作环境这一系列的操作。如下图所示

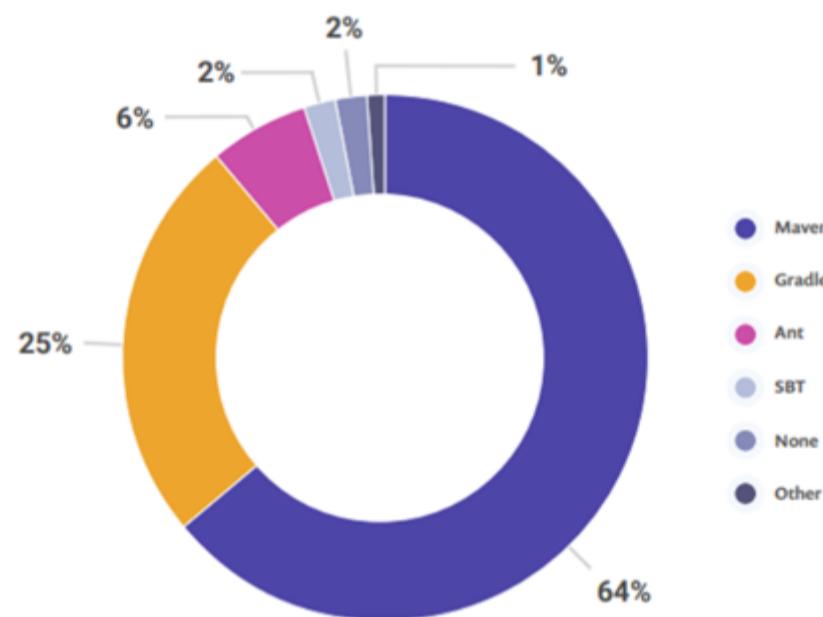


而Maven使用标准的 **坐标** 配置来管理各种依赖，只需要简单的配置就可以完成依赖管理。



如上图右边所示就是mysql驱动包的坐标，在项目中只需要写这段配置，其他都不需要我们担心，Maven都帮我们进行操作了。

市面上有很多构建工具，而Maven依旧还是主流构建工具，如下图是常用构建工具的使用占比



1.1 Maven简介

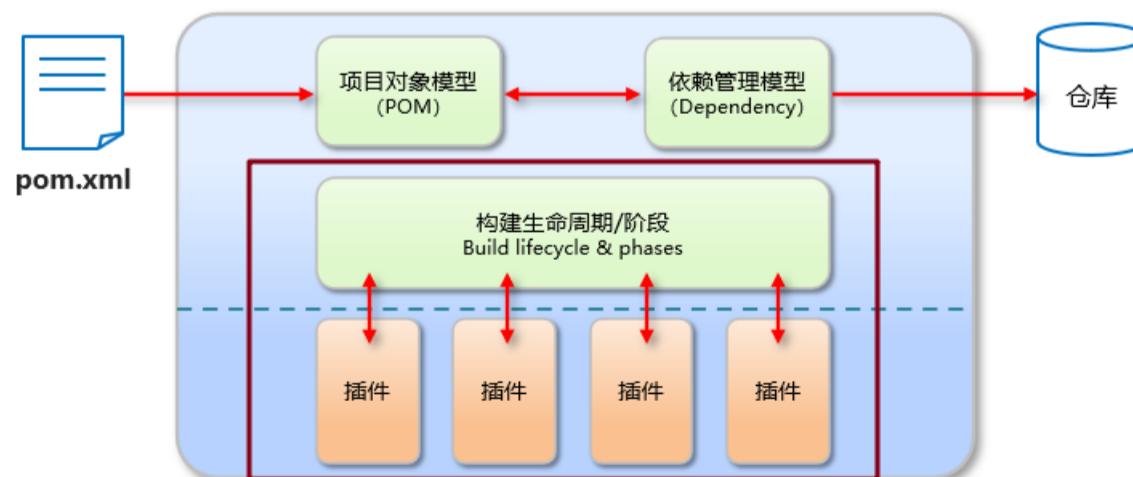
Apache Maven 是一个项目管理和构建**工具**，它基于项目对象模型(POM)的概念，通过一小段描述信息来管理项目的构建、报告和文档。

官网：<http://maven.apache.org/>

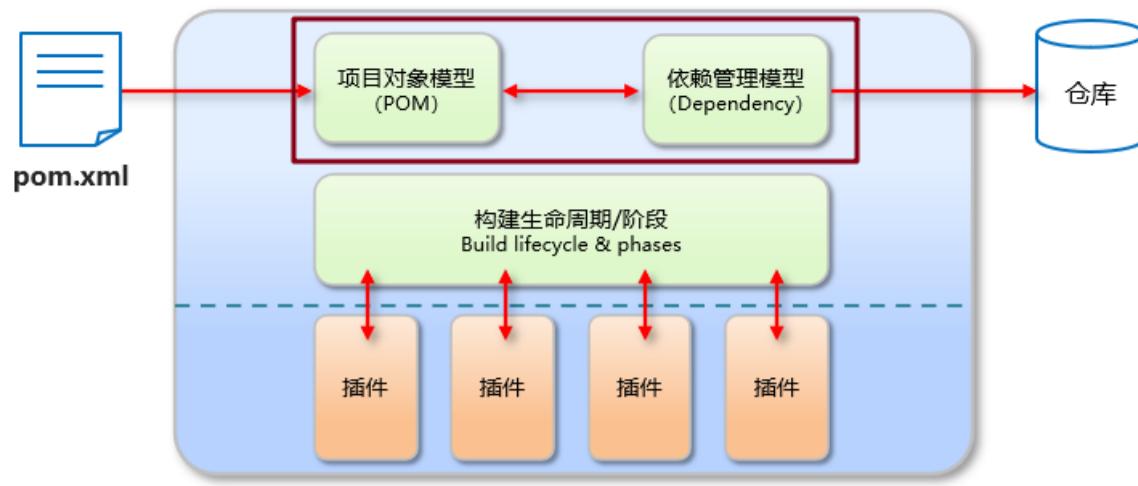
通过上面的描述大家只需要知道Maven是一个工具即可。Apache 是一个开源组织，将来我们会学习很多Apache提供的项目。

1.1.1 Maven模型

- 项目对象模型 (Project Object Model)
- 依赖管理模型(Dependency)
- 插件(Plugin)



如上图所示就是Maven的模型，而我们先看紫色框起来的部分，他就是用来完成 **标准化构建流程** 。如我们需要编译，Maven提供了一个编译插件供我们使用，我们需要打包，Maven就提供了一个打包插件提供我们使用等。



上图中紫色框起来的部分，项目对象模型就是将我们自己抽象成一个对象模型，有自己专属的坐标，如下图所示是一个 Maven项目：

A screenshot of the Eclipse IDE interface. On the left is the 'Project' view showing a project named 'jdbc-demo' with a sub-project 'maven-project'. The 'src' folder and 'pom.xml' file are visible. The main editor window shows the XML content of 'pom.xml'. A red box highlights the 'maven-project' section. In the code, the 'groupId', 'artifactId', and 'version' tags are highlighted with a red box and labeled '当前项目的坐标' (Current project's coordinate).

```
<?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>com.itheima</groupId>
    <artifactId>maven-project</artifactId>
    <version>1.0-SNAPSHOT</version>
```

依赖管理模型则是使用坐标来描述当前项目依赖哪儿些第三方jar包，如下图所示

A screenshot of the Eclipse IDE interface, similar to the previous one. The 'Project' view shows the same 'jdbc-demo' and 'maven-project' structure. The main editor window shows the 'pom.xml' file. A red box highlights the 'maven-project' section. In the code, the 'dependencies' section is highlighted with a red box and labeled '项目依赖mysql驱动包' (Project dependency on MySQL driver package).

```
<groupId>com.itheima</groupId>
<artifactId>maven-project</artifactId>
<version>1.0-SNAPSHOT</version>

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

<!-- 引入 mysql 驱动jar包-->
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.32</version>
    </dependency>
</dependencies>
```

上述Maven模型图中还有一部分是仓库。如何理解仓库呢？

1.1.2 仓库

大家想想这样的场景，我们创建Maven项目，在项目中使用坐标来指定项目的依赖，那么依赖的jar包到底存储在什么地方呢？其实依赖jar包是存储在我们的本地仓库中。而项目运行时从本地仓库中拿需要的依赖jar包。

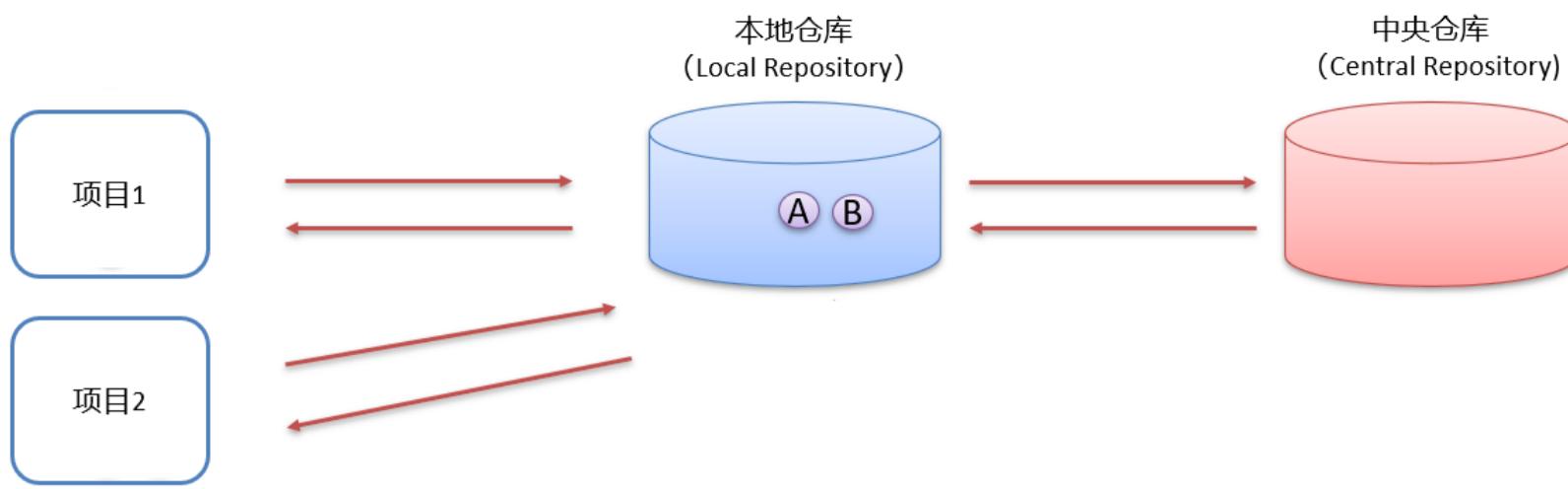
仓库分类：

- 本地仓库：自己计算机上的一个目录
- 中央仓库：由Maven团队维护的全球唯一的仓库
 - 地址：<https://repo1.maven.org/maven2/>
- 远程仓库(私服)：一般由公司团队搭建的私有仓库

今天我们只学习远程仓库的使用，并不会搭建。

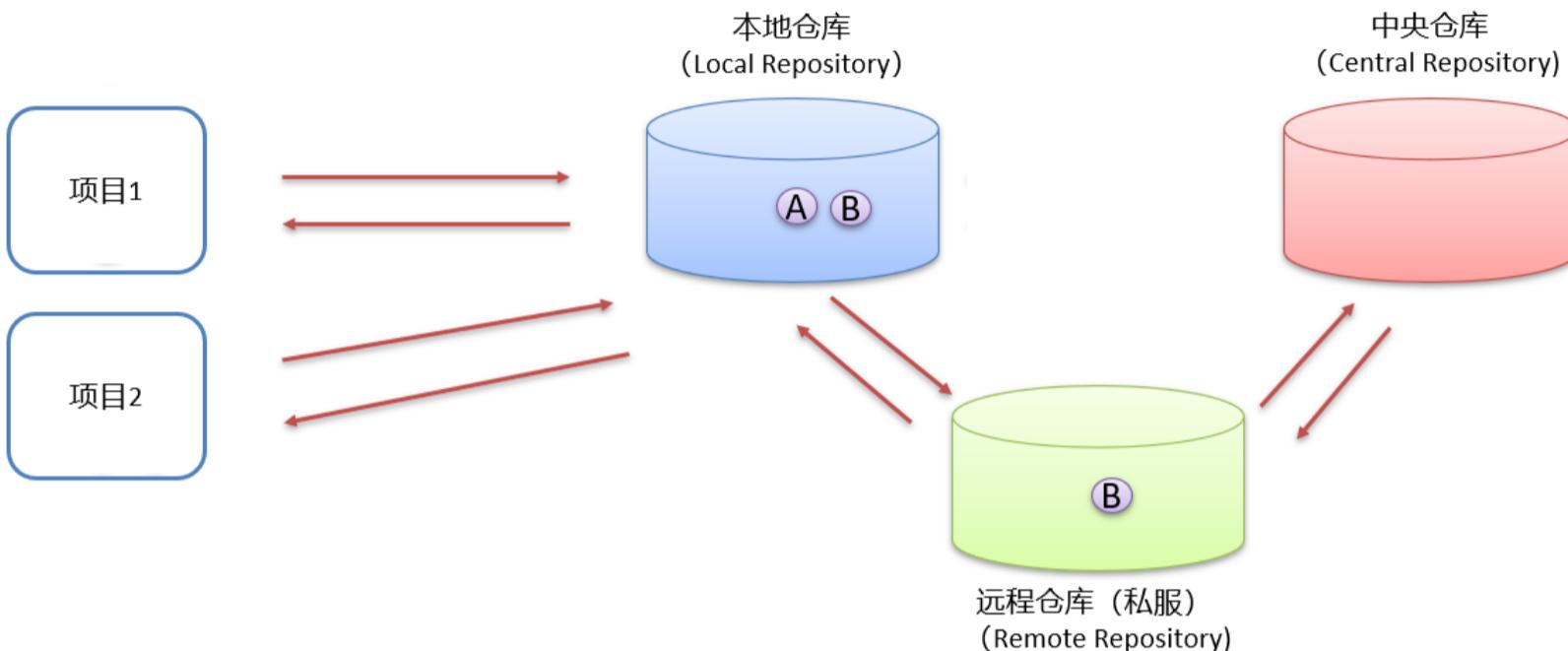
当项目中使用坐标引入对应依赖jar包后，首先会查找本地仓库中是否有对应的jar包：

- 如果有，则在项目直接引用；
- 如果没有，则去中央仓库中下载对应的jar包到本地仓库。



如果还可以搭建远程仓库，将来jar包的查找顺序则变为：

本地仓库 --> 远程仓库--> 中央仓库



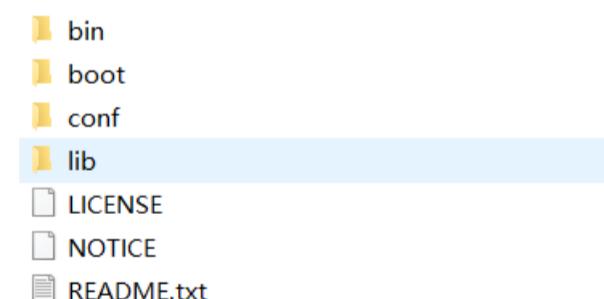
1.2 Maven安装配置

- 解压 apache-maven-3.6.1.rar 既安装完成



建议解压缩到没有中文、特殊字符的路径下。如课程中解压缩到 D:\software 下。

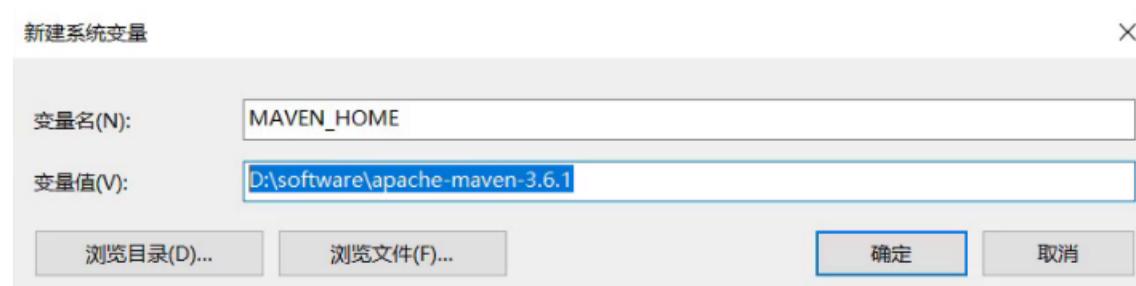
解压缩后的目录结构如下：



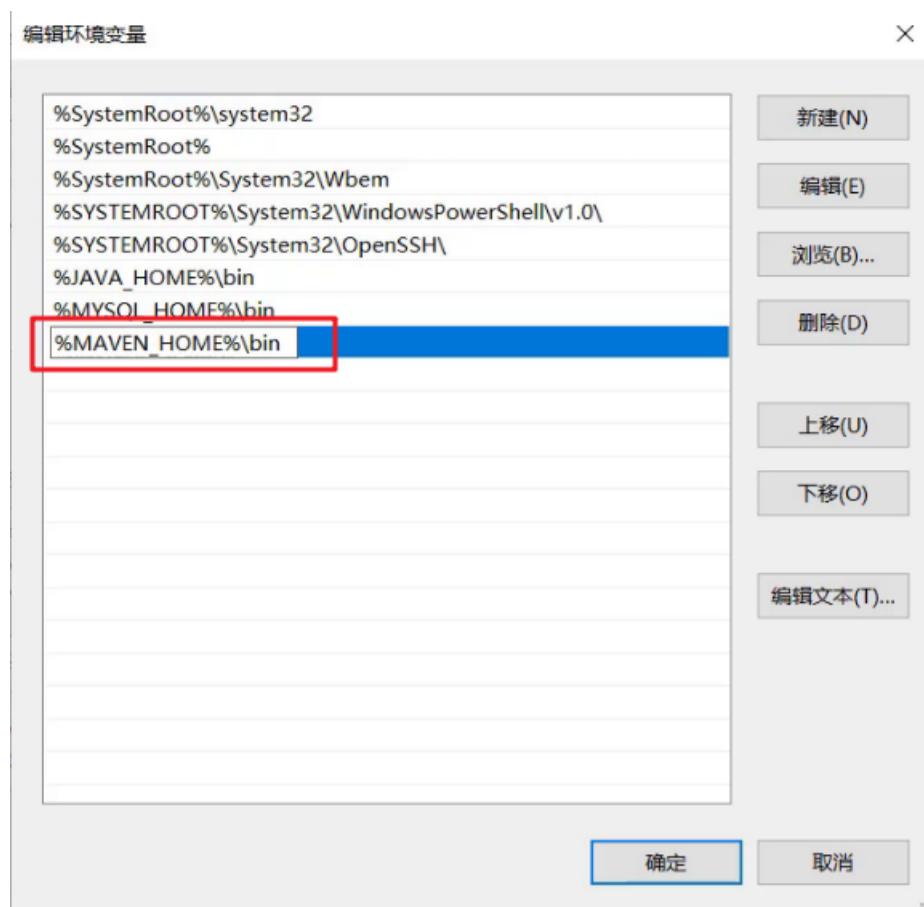
- bin 目录：存放的是可执行命令。mvn 命令重点关注。
- conf 目录：存放Maven的配置文件。settings.xml 配置文件后期需要修改。
- lib 目录：存放Maven依赖的jar包。Maven也是使用java开发的，所以它也依赖其他的jar包。
- 配置环境变量 MAVEN_HOME 为安装路径的bin目录

此电脑 右键 --> 高级系统设置 --> 高级 --> 环境变量

在系统变量处新建一个变量 MAVEN_HOME



在 Path 中进行配置



打开命令提示符进行验证，出现如图所示表示安装成功

```
C:\Users\super>mvn -version
Apache Maven 3.6.1 (d66c9c0b3152b2e69ee9bac180bb8fcc8e6af555; 2019-04-05T03:00:29+08:00)
Maven home: D:\software\apache-maven-3.6.1\bin\..
Java version: 1.8.0_172, vendor: Oracle Corporation, runtime: D:\software\jdk8_72\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

- 配置本地仓库

修改 conf/settings.xml 中的 `<localRepository>` 为一个指定目录作为本地仓库，用来存储jar包。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd
  >
  <!-- localRepository
    | The path to the local repository maven will use to store artifacts.
    |
    | Default: ${user.home}/.m2/repository
  <localRepository>/path/to/local/repo</localRepository>
  -->
  <localRepository>D:\software\apache-maven-3.6.1\mvn_resp</localRepository>
  <!-- interactiveMode
    | If true, maven will ask for confirmation before overwriting existing files.
    |
    | Default: false
  <interactiveMode>false</interactiveMode>

```

- 配置阿里云私服

中央仓库在国外，所以下载jar包速度可能比较慢，而阿里公司提供了一个远程仓库，里面基本也都有开源项目的jar包。

修改 conf/settings.xml 中的 `<mirror>` 标签，为其添加如下子标签：

```
1 <mirror>
2   <id>alimaven</id>
3   <name>aliyun maven</name>
4   <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
5   <mirrorOf>central</mirrorOf>
6 </mirror>
```

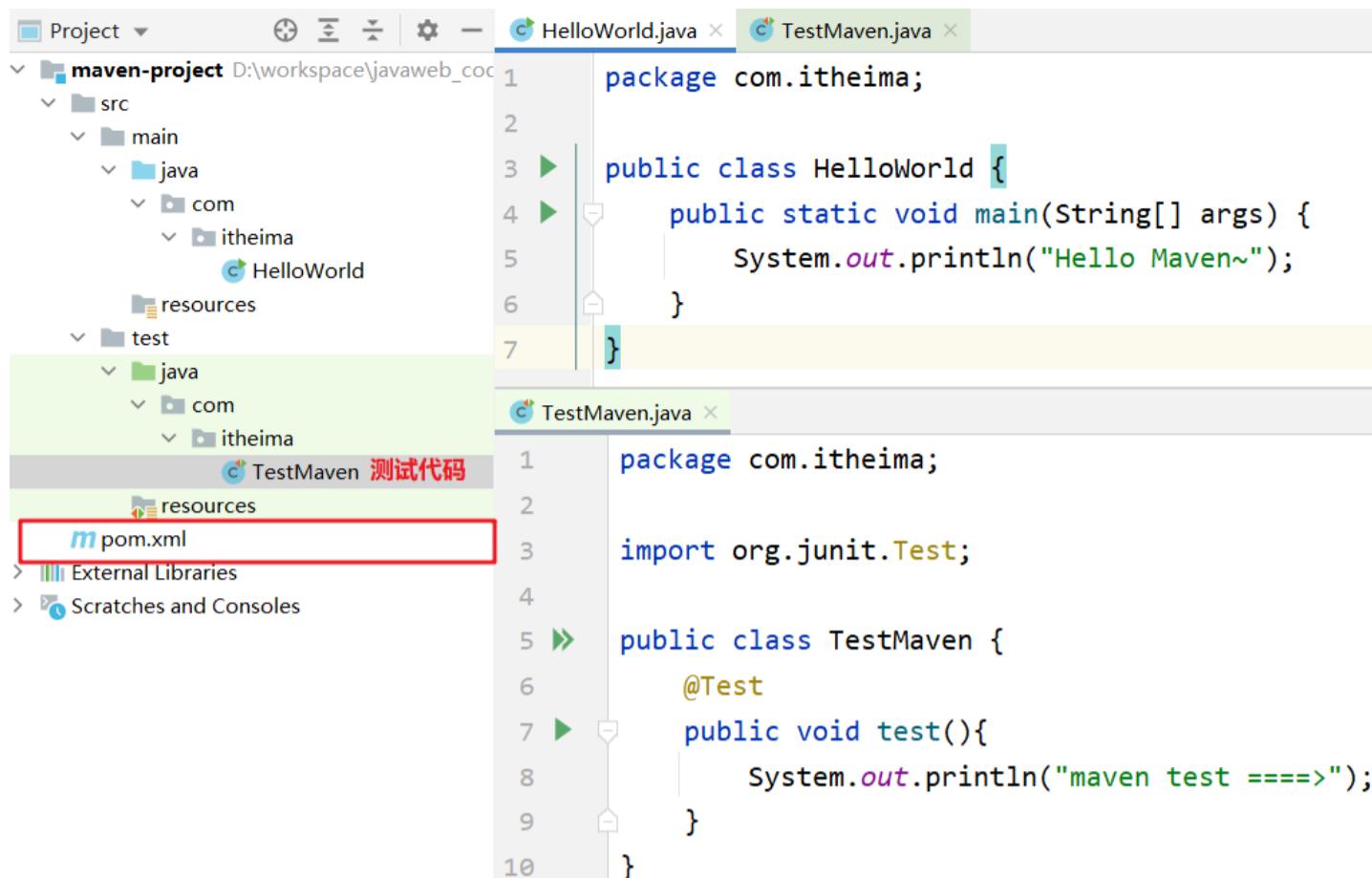
1.3 Maven基本使用

1.3.1 Maven 常用命令

- compile：编译
- clean：清理
- test：测试
- package：打包
- install：安装

命令演示：

在 `资料\代码\maven-project` 提供了一个使用Maven构建的项目，项目结构如下：



而我们使用上面命令需要在磁盘上进入到项目的 `pom.xml` 目录下，打开命令提示符

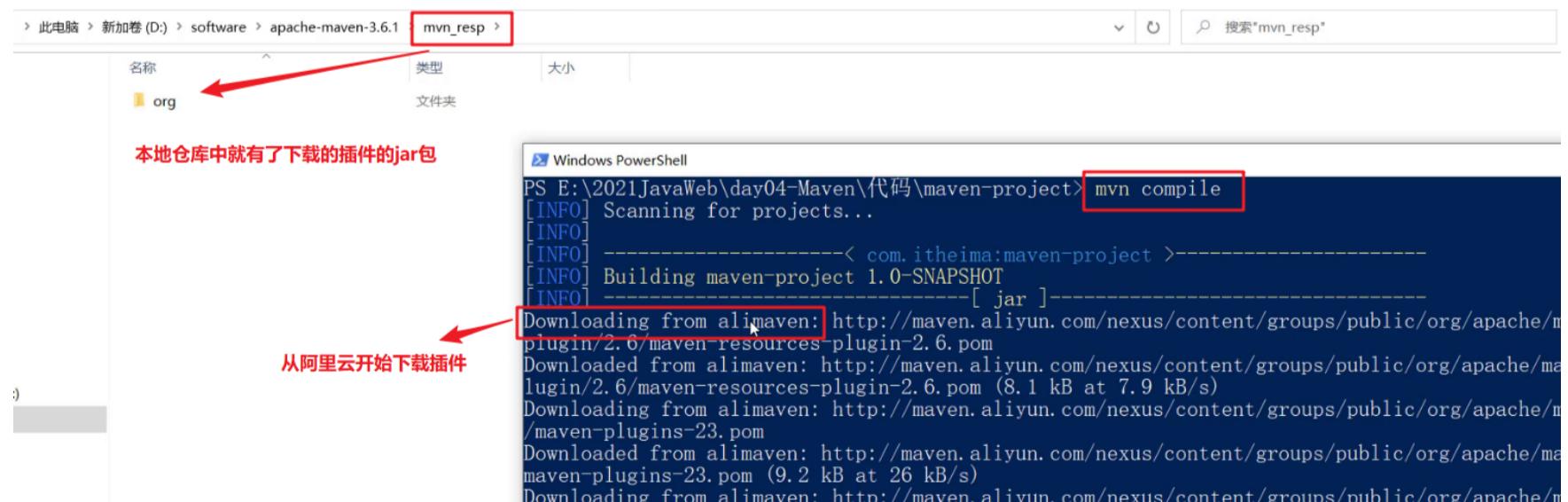


编译命令演示：

1 | `compile` : 编译

执行上述命令可以看到：

- 从阿里云下载编译需要的插件的jar包，在本地仓库也能看到下载好的插件
- 在项目下会生成一个 `target` 目录



同时在项目下会出现一个 `target` 目录，编译后的字节码文件就放在该目录下



清理命令演示：

```
1 | mvn clean
```

执行上述命令可以看到

- 从阿里云下载清理需要的插件jar包
- 删除项目下的 target 目录

```
PS E:\2021JavaWeb\day04-Maven\代码\maven-project> mvn clean
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.itheima:maven-project >-----
[INFO] Building maven-project 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
Downloading from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/2.5/maven-clean-plugin-2.5.pom
Downloaded from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/2.5/maven-clean-plugin-2.5.pom (3.9 kB at 4.1 kB/s)
Downloading from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/maven-plugins-22.pom
Downloaded from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/maven-plugins-22.pom (13 kB at 40 kB/s)
Downloading from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/2.5/maven-clean-plugin-2.5.jar
Downloaded from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/2.5/maven-clean-plugin-2.5.jar (25 kB at 77 kB/s)
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ maven-project ---
Downloading from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/codehaus/us-utils-3.0.pom
Progress (1) · 4 1 kB
```

打包命令演示：

```
1 | mvn package
```

执行上述命令可以看到：

- 从阿里云下载打包需要的插件jar包
- 在项目的 target 目录下有一个jar包（将当前项目打成的jar包）

```
PS E:\2021JavaWeb\day04-Maven\代码\maven-project> mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.itheima:maven-project >-----
[INFO] Building maven-project 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
Downloading from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/plugin/2.12.4/maven-surefire-plugin-2.12.4.pom
Downloaded from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/plugin/2.12.4/maven-surefire-plugin-2.12.4.pom (10 kB at 12 kB/s)
Downloading from alimaven: http://maven.aliyun.com/nexus/content/groups/public/org/apache/maven/
```

测试命令演示：

```
1 | mvn test
```

该命令会执行所有的测试代码。执行上述命令效果如下

```
-----  
T E S T S  
-----  
Running com.itheima.TestMaven  
maven test ===>  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.066 sec  
  
Results :  
  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0  
  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 1.955 s  
[INFO] Finished at: 2021-05-06T21:04:49+08:00  
[INFO] -----
```

安装命令演示：

```
1 | mvn install
```

该命令会将当前项目打成jar包，并安装到本地仓库。执行完上述命令后到本地仓库查看结果如下：

i > 新加卷 (D:) > software > apache-maven-3.6.1 > mvn_resp > com > itheima > maven-project > 1.0-SNAPSHOT		
名称	类型	大小
_remote.repositories	REPOSITORIES ...	1 KB
maven-metadata-local.xml	XML 文件	1 KB
maven-project-1.0-SNAPSHOT.jar	Executable Jar File	3 KB
maven-project-1.0-SNAPSHOT.pom	POM 文件	2 KB

1.3.2 Maven 生命周期

Maven 构建项目生命周期描述的是一次构建过程经历经历了多少个事件

Maven 对项目构建的生命周期划分为3套：

- clean：清理工作。
- default：核心工作，例如编译，测试，打包，安装等。
- site：产生报告，发布站点等。这套声明周期一般不会使用。

同一套生命周期内，执行后边的命令，前面的所有命令会自动执行。例如默认（default）生命周期如下：



当我们执行 `install`（安装）命令时，它会先执行 `compile` 命令，再执行 `test` 命令，再执行 `package` 命令，最后执行 `install` 命令。

当我们执行 `package`（打包）命令时，它会先执行 `compile` 命令，再执行 `test` 命令，最后执行 `package` 命令。

默认的生命周期也有对应的很多命令，其他的一般都不会使用，我们只关注常用的：

validate (校验)	校验项目是否正确并且所有必要的信息可以完成项目的构建过程。
initialize (初始化)	初始化构建状态，比如设置属性值。
generate-sources (生成源代码)	生成包含在编译阶段中的任何源代码。
process-sources (处理源代码)	处理源代码，比如说，过滤任意值。
generate-resources (生成资源文件)	生成将会包含在项目包中的资源文件。
process-resources (处理资源文件)	复制和处理资源到目标目录，为打包阶段做好准备。
compile (编译)	编译项目的源代码。
process-classes (处理类文件)	处理编译生成的文件，比如说对Java class文件做字节码改善优化。
generate-test-sources (生成测试源代码)	生成包含在编译阶段中的任何测试源代码。
process-test-sources (处理测试源代码)	处理测试源代码，比如说，过滤任意值。
generate-test-resources (生成测试资源文件)	为测试创建资源文件。
process-test-resources (处理测试资源文件)	复制和处理测试资源到目标目录。
test-compile (编译测试源码)	编译测试源代码到测试目标目录。
process-test-classes (处理测试类文件)	处理测试源码编译生成的文件。
test (测试)	使用合适的单元测试框架运行测试（ <u>Junit</u> 是其中之一）。
prepare-package (准备打包)	在实际打包之前，执行任何的必要的操作为打包做准备。
package (打包)	将编译后的代码打包成可分发格式的文件，比如JAR、WAR或者EAR文件。
pre-integration-test (集成测试前)	在执行集成测试前进行必要的动作。比如说，搭建需要的环境。
integration-test (集成测试)	处理和部署项目到可以运行集成测试环境中。
post-integration-test (集成测试后)	在执行集成测试完成后进行必要的动作。比如说，清理集成测试环境。
verify (验证)	运行任意的检查来验证项目包有效且达到质量标准。
install (安装)	安装项目包到本地仓库，这样项目包可以用作其他本地项目的依赖。
deploy (部署)	将最终的项目包复制到远程仓库中与其他开发者和项目共享。

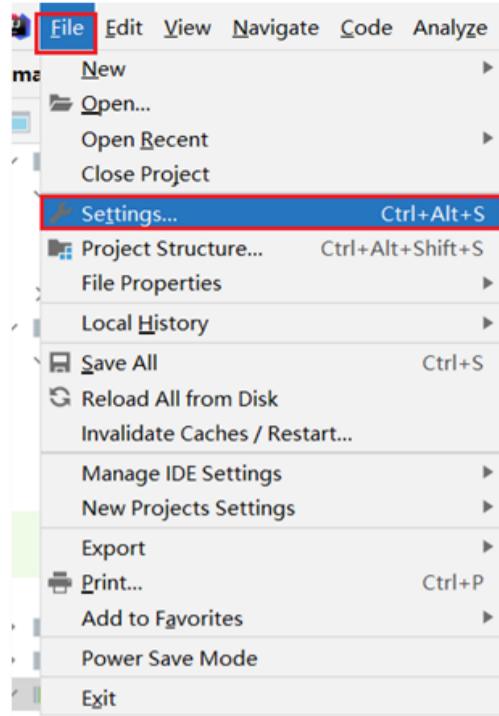
1.4 IDEA使用Maven

以后开发中我们肯定会在高级开发工具中使用Maven管理项目，而我们常用的高级开发工具是IDEA，所以接下来我们会讲解Maven在IDEA中的使用。

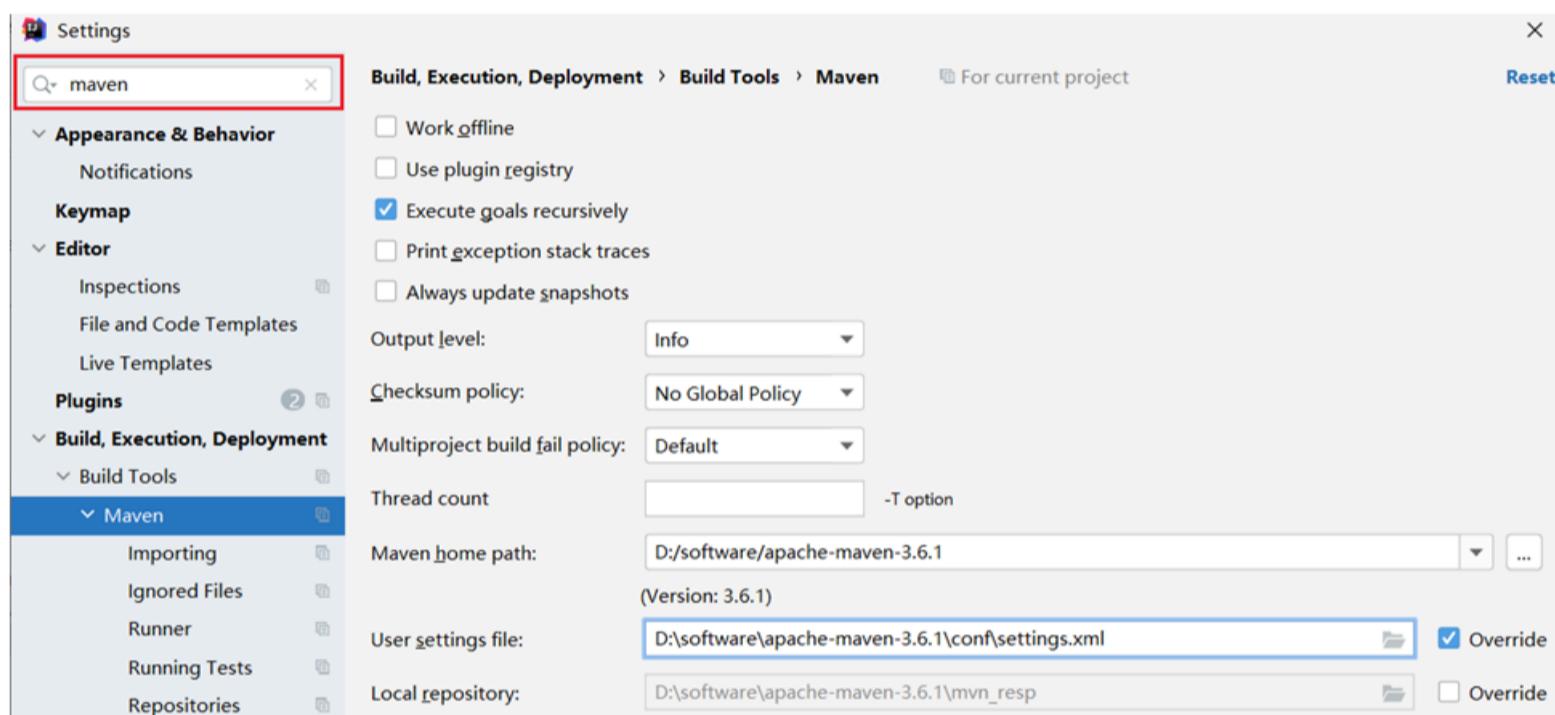
1.4.1 IDEA配置Maven环境

我们需要先在IDEA中配置Maven环境：

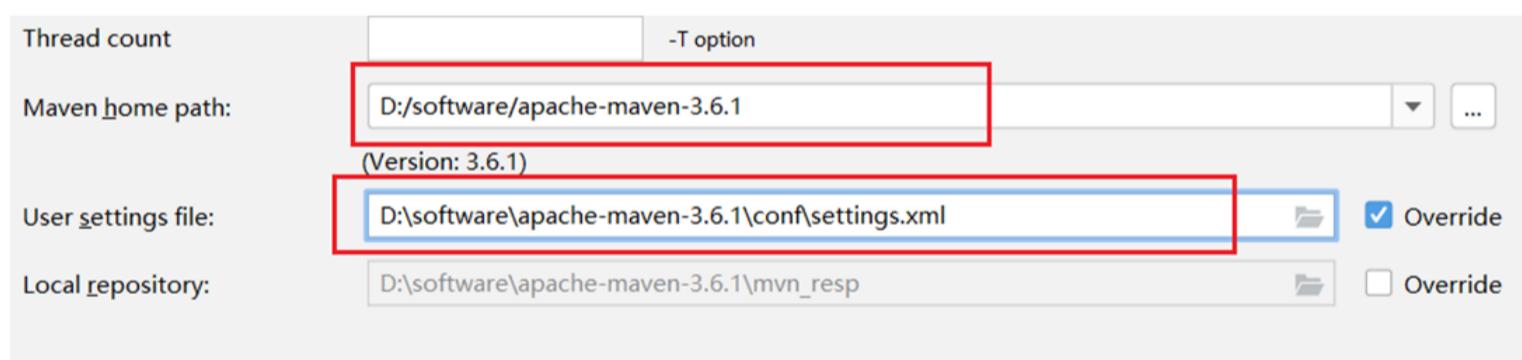
- 选择 IDEA 中 File --> Settings



- 搜索 maven



- 设置 IDEA 使用本地安装的 Maven，并修改配置文件路径



1.4.2 Maven 坐标详解

什么是坐标？

- Maven 中的坐标是**资源的唯一标识**
- 使用坐标来定义项目或引入项目中需要的依赖

Maven 坐标主要组成

- groupId: 定义当前Maven项目隶属组织名称（通常是域名反写，例如：com.itheima）
- artifactId: 定义当前Maven项目名称（通常是模块名称，例如 order-service、goods-service）
- version: 定义当前项目版本号

如下图就是使用坐标表示一个项目：

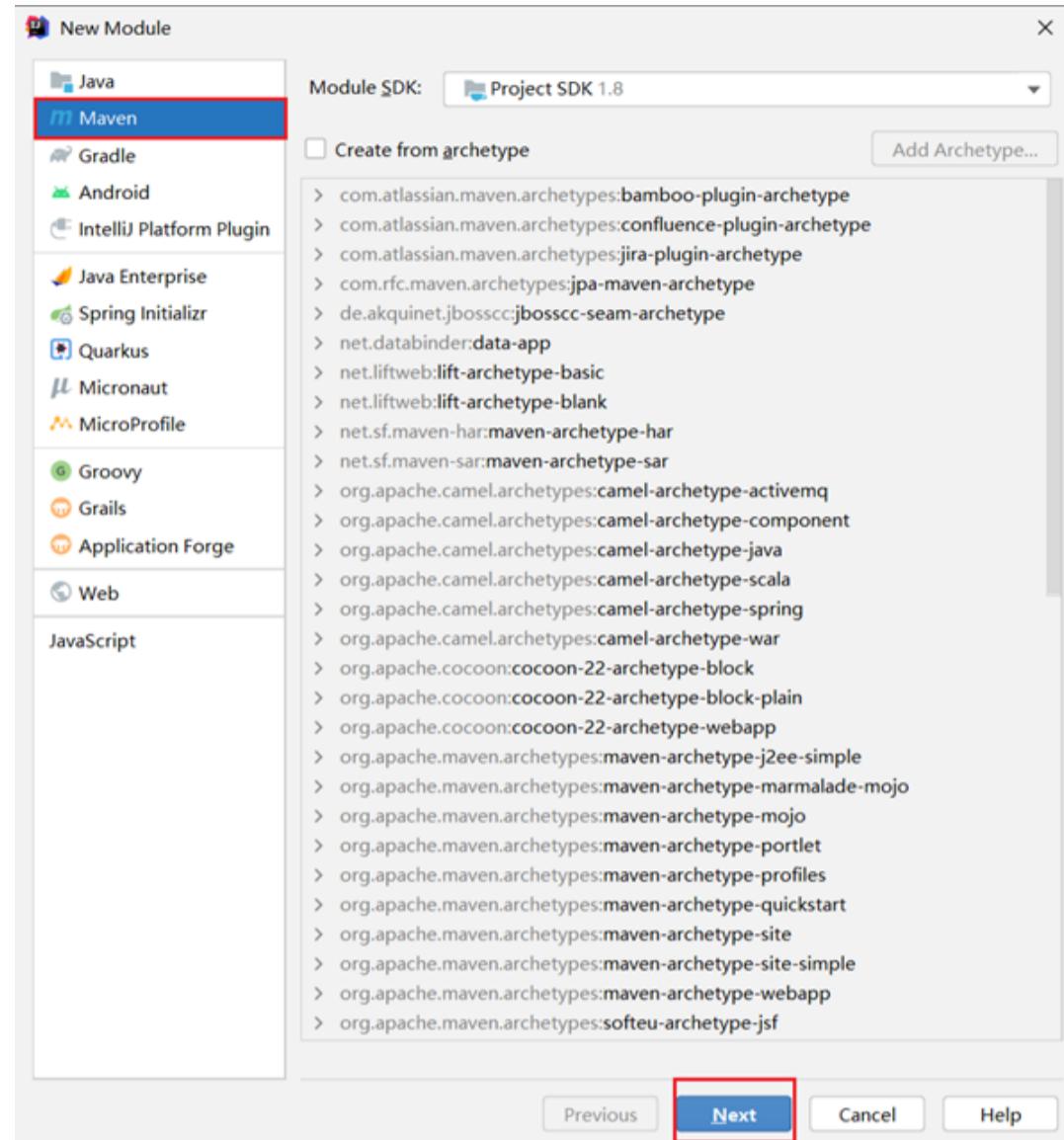
```
<groupId>com.itheima</groupId>
<artifactId>maven-demo</artifactId>
<version>1.0-SNAPSHOT</version>
```

注意：

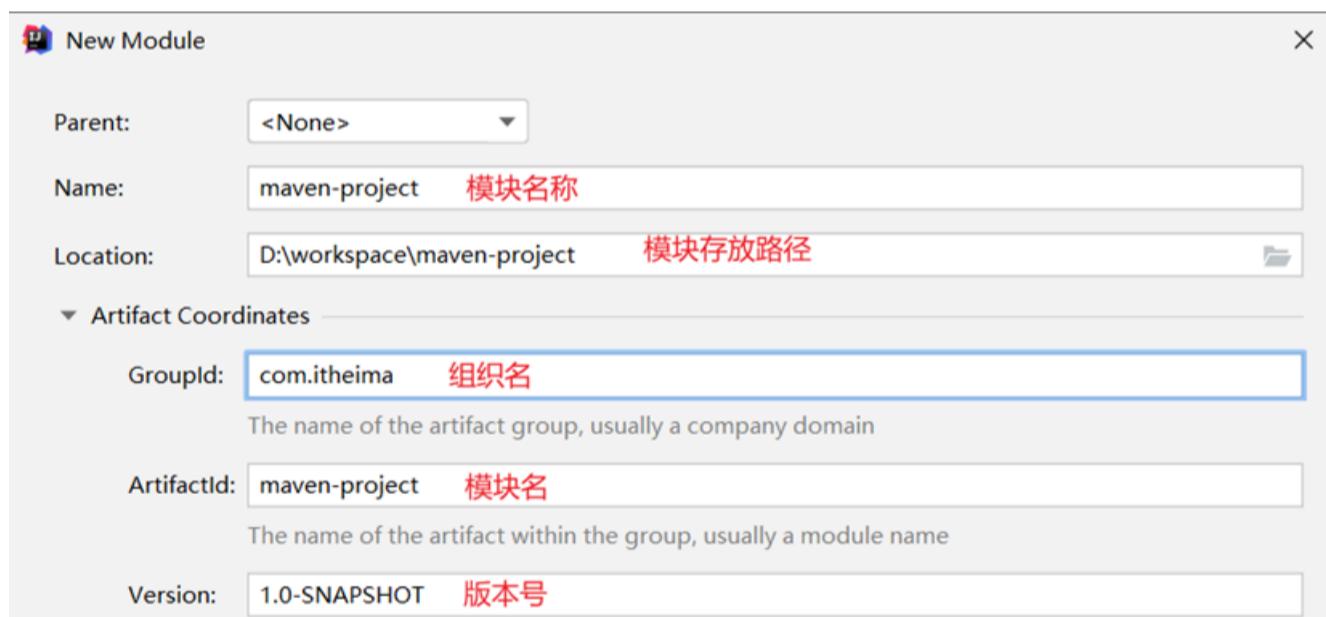
- 上面所说的资源可以是插件、依赖、当前项目。
- 我们的项目如果被其他的项目依赖时，也是需要坐标来引入的。

1.4.3 IDEA 创建 Maven项目

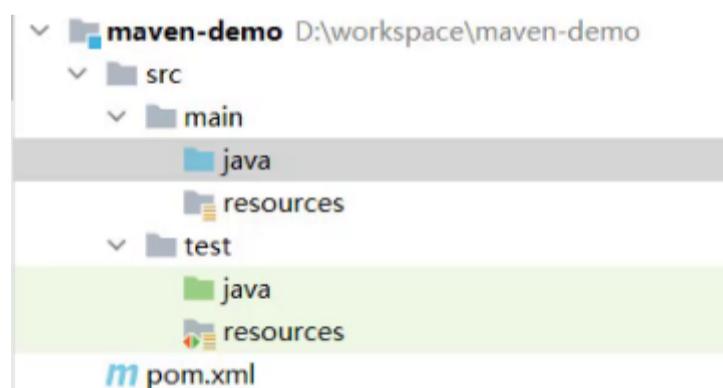
- 创建模块，选择Maven，点击Next



- 填写模块名称，坐标信息，点击finish，创建完成



创建好的项目目录结构如下：

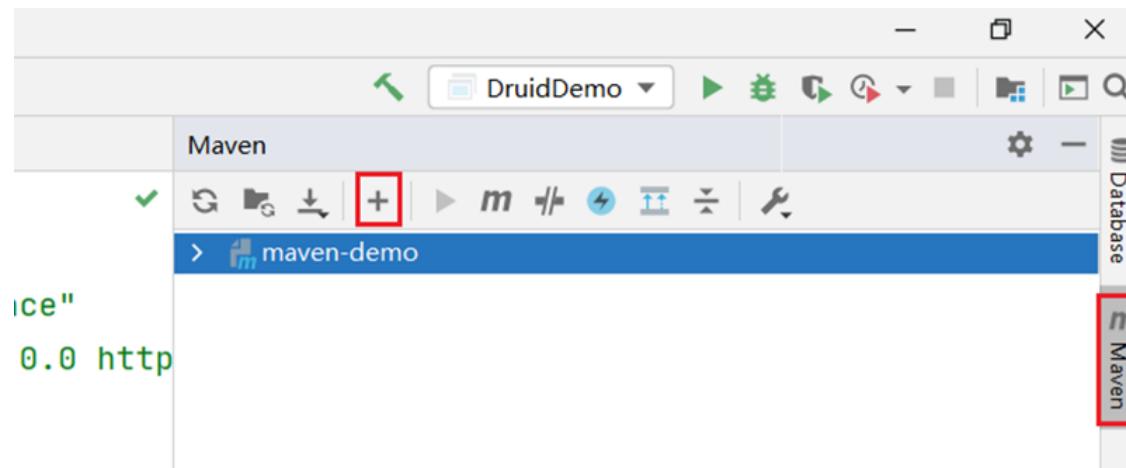


- 编写 HelloWorld，并运行

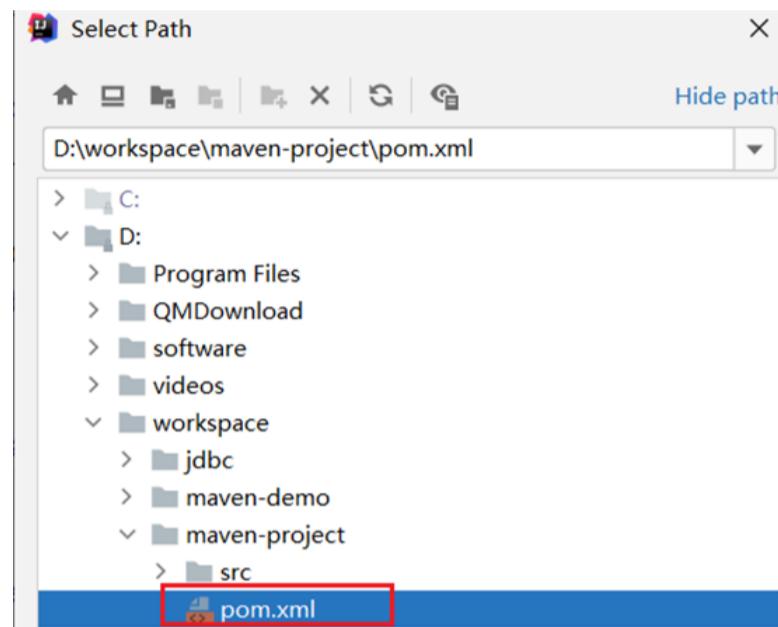
1.4.4 IDEA 导入 Maven项目

大家在学习时可能需要看老师的代码，当然也就需要将老师的代码导入到自己的IDEA中。我们可以通过以下步骤进行项目的导入：

- 选择右侧Maven面板，点击 + 号

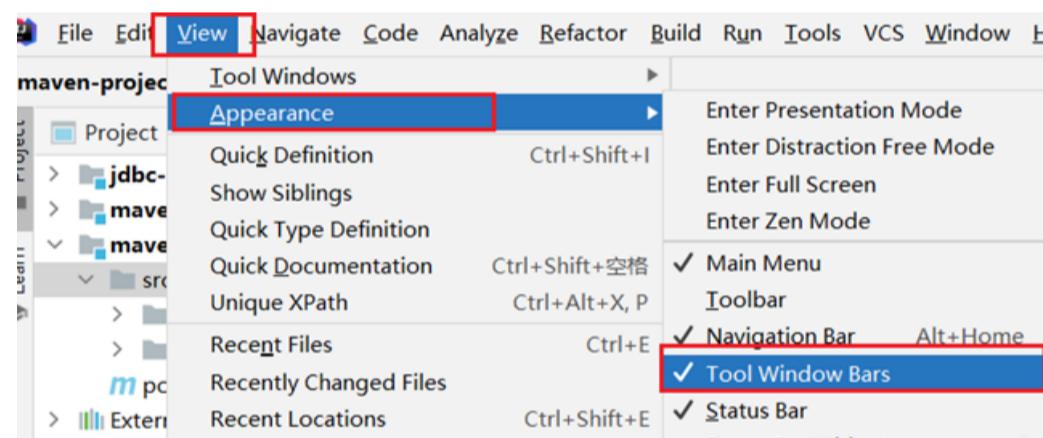


- 选中对应项目的pom.xml文件，双击即可

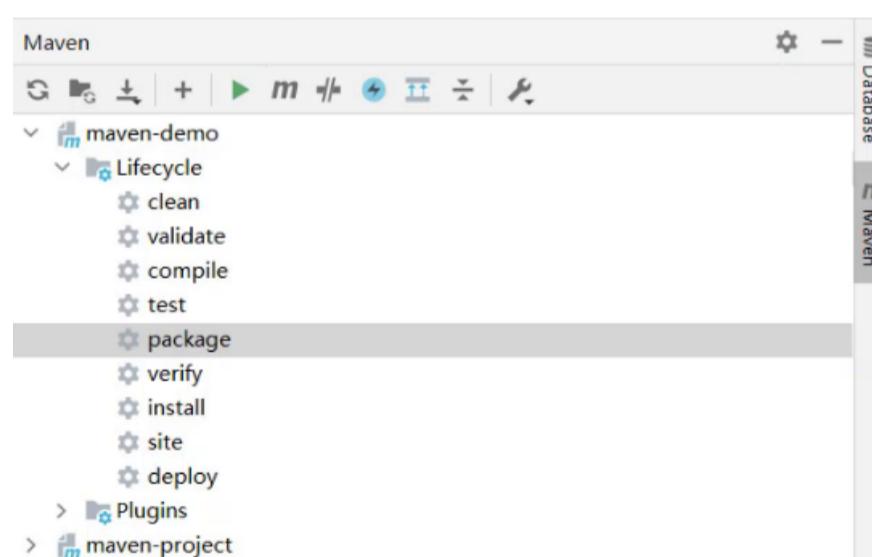


- 如果没有Maven面板，选择

View --> Appearance --> Tool Window Bars

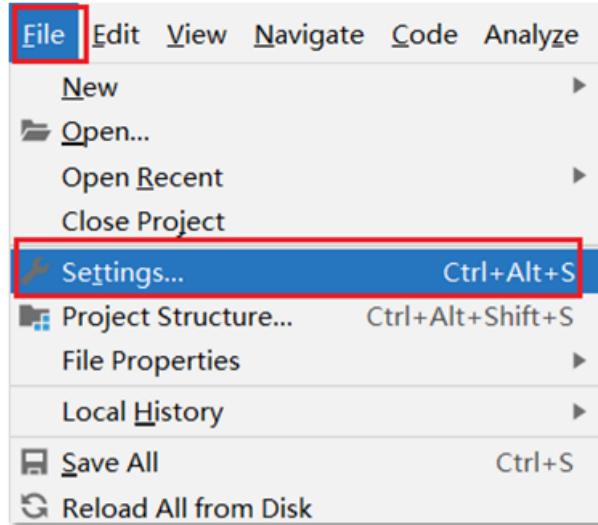


可以通过下图所示进行命令的操作：

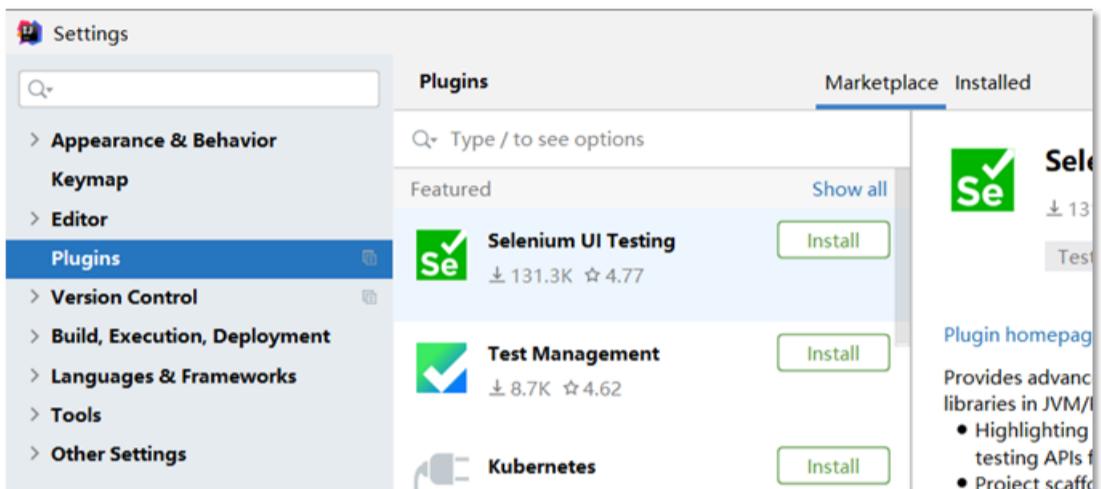


配置 Maven-Helper 插件

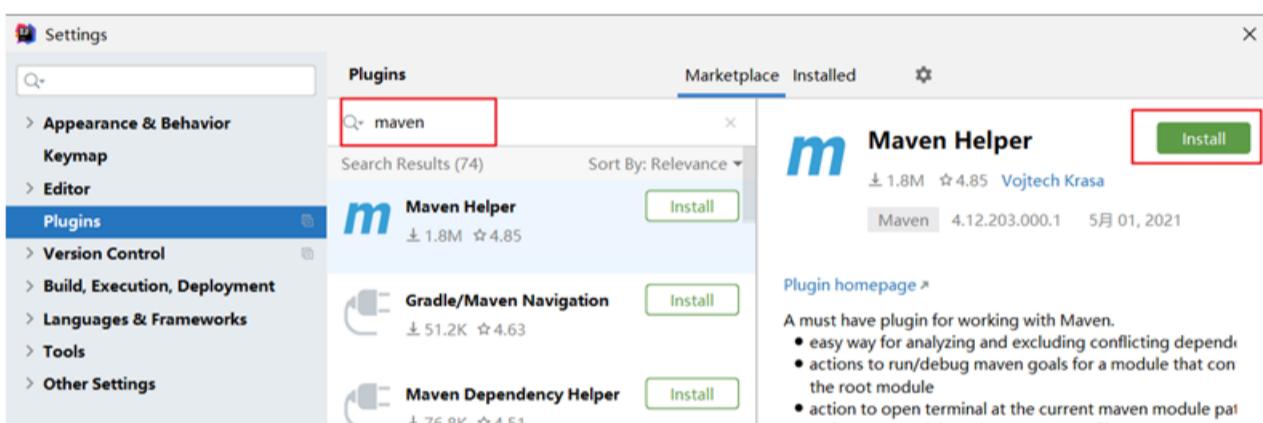
- 选择 IDEA 中 File --> Settings



- 选择 Plugins

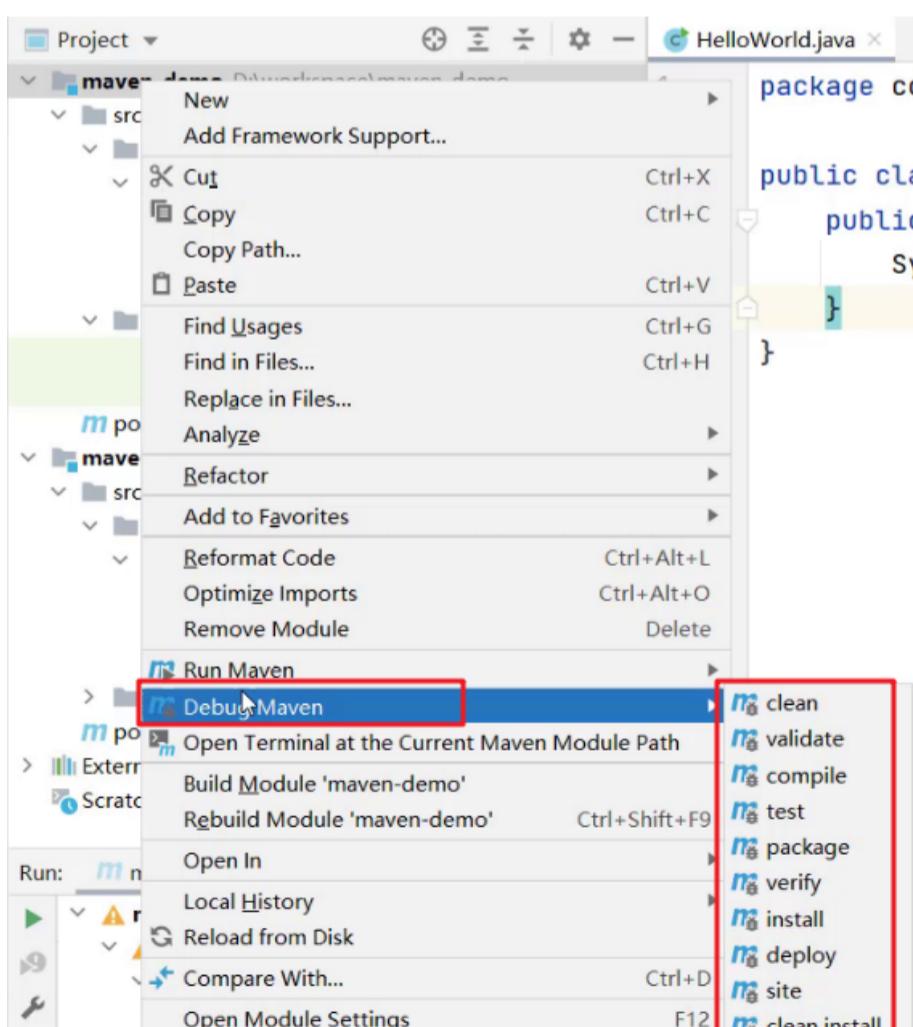


- 搜索 Maven，选择第一个 Maven Helper，点击Install安装，弹出面板中点击Accept



- 重启 IDEA

安装完该插件后可以通过 选中项目右键进行相关命令操作，如下图所示：



1.5 依赖管理

1.5.1 使用坐标引入jar包

使用坐标引入jar包的步骤：

- 在项目的 pom.xml 中编写 标签
- 在 标签中 使用 引入坐标
- 定义坐标的 groupId, artifactId, version

```
<dependencies>
    <!-- mysql 坐标 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.32</version>
    </dependency>
</dependencies>
```

- 点击刷新按钮，使坐标生效



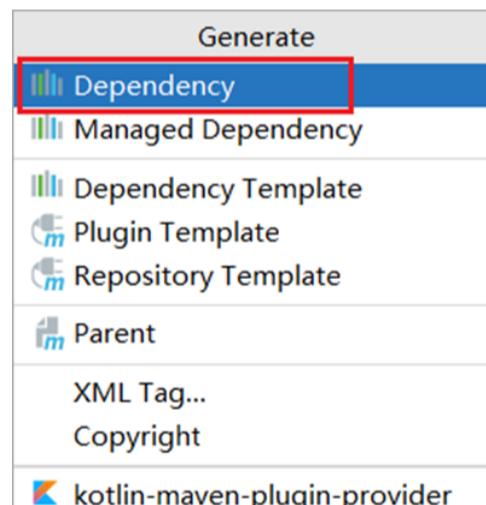
注意：

- 具体的坐标我们可以到如下网站进行搜索
- <https://mvnrepository.com/>

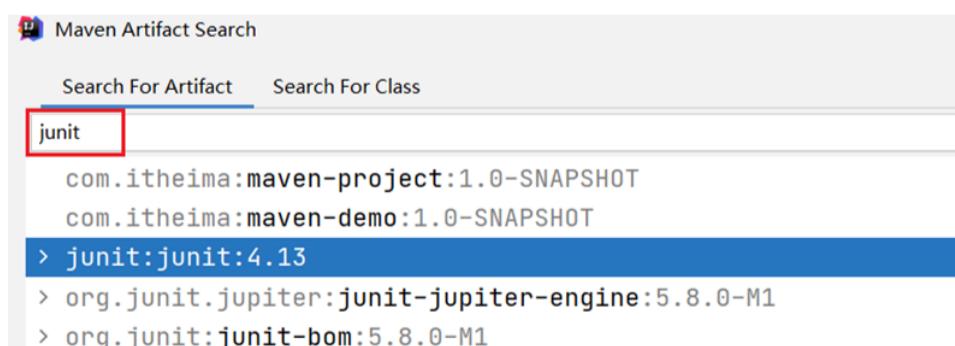
快捷方式导入jar包的坐标：

每次需要引入jar包，都去对应的网站进行搜索是比较麻烦的，接下来给大家介绍一种快捷引入坐标的方式

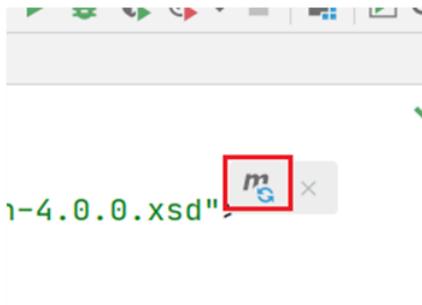
- 在 pom.xml 中按 alt + insert，选择 Dependency



- 在弹出的面板中搜索对应坐标，然后双击选中对应坐标



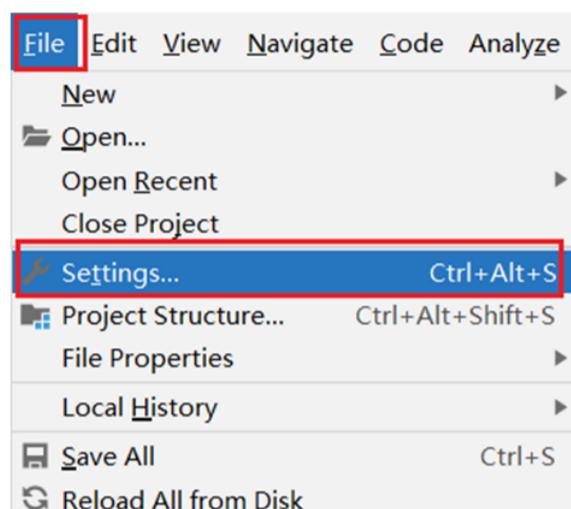
- 点击刷新按钮，使坐标生效



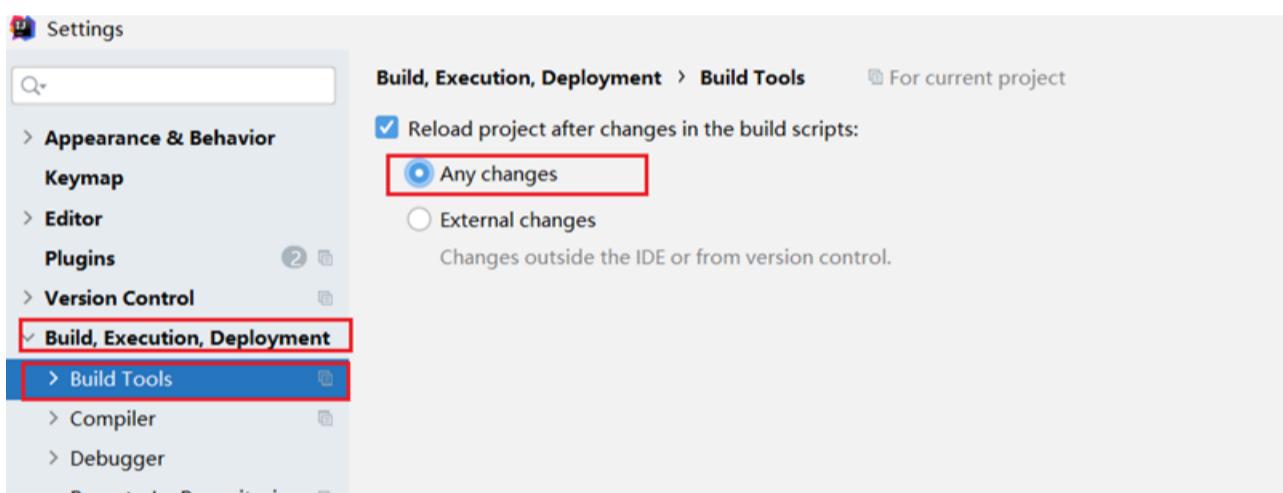
自动导入设置：

上面每次操作都需要点击刷新按钮，让引入的坐标生效。当然我们也可以通过设置让其自动完成

- 选择 IDEA 中 File --> Settings



- 在弹出的面板中找到 Build Tools



- 选择 Any changes，点击 ok 即可生效

1.5.2 依赖范围

通过设置坐标的依赖范围(scope)，可以设置 对应jar包的作用范围：编译环境、测试环境、运行环境。

如下图所示给 `junit` 依赖通过 `scope` 标签指定依赖的作用范围。那么这个依赖就只能作用在测试环境，其他环境下不能使用。

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>test</scope>
</dependency>
```

那么 `scope` 都可以有哪些取值呢？

依赖范围	编译classpath	测试classpath	运行classpath	例子
compile	Y	Y	Y	logback
test	-	Y	-	Junit
provided	Y	Y	-	servlet-api
runtime	-	Y	Y	jdbc驱动
system	Y	Y	-	存储在本地的jar包

- compile：作用于编译环境、测试环境、运行环境。

- test：作用于测试环境。典型的就是Junit坐标，以后使用Junit时，都会将scope指定为该值
- provided：作用于编译环境、测试环境。我们后面会学习 `servlet-api`，在使用它时，必须将 `scope` 设置为该值，不然运行时就会报错
- runtime：作用于测试环境、运行环境。jdbc驱动一般将 `scope` 设置为该值，当然不设置也没有任何问题

注意：

- 如果引入坐标不指定 `scope` 标签时，默认就是 `compile` 值。以后大部分jar包都是使用默认值。

2, Mybatis

2.1 Mybatis概述

2.1.1 Mybatis概念

- MyBatis 是一款优秀的持久层框架，用于简化 JDBC 开发
- MyBatis 本是 Apache 的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github
- 官网：<https://mybatis.org/mybatis-3/zh/index.html>

持久层：

- 负责将数据到保存到数据库的那一层代码。

以后开发我们会将操作数据库的Java代码作为持久层。而Mybatis就是对jdbc代码进行了封装。

- JavaEE三层架构：表现层、业务层、持久层

三层架构在后期会给大家进行讲解，今天先简单的了解下即可。

框架：

- 框架就是一个半成品软件，是一套可重用的、通用的、软件基础代码模型
- 在框架的基础之上构建软件编写更加高效、规范、通用、可扩展

举例给大家简单的解释一下什么是半成品软件。大家小时候应该在公园见过给石膏娃娃涂鸦



如下图所示有一个石膏娃娃，这个就是一个半成品。你可以在这个半成品的基础上进行不同颜色的涂鸦



了解了什么是Mybatis后，接下来说说以前 `JDBC代码` 的缺点以及Mybatis又是如何解决的。

2.1.2 JDBC 缺点

下面是 JDBC 代码，我们通过该代码分析都存在什么缺点：

```
//1. 注册驱动  
Class.forName("com.mysql.jdbc.Driver");  
//2. 获取Connection连接  
String url = "jdbc:mysql://db1?useSSL=false"; ①  
String uname = "root";  
String pwd = "1234";  
Connection conn = DriverManager.getConnection(url, uname, pwd);  
//接收输入的查询条件  
String gender = "男";  
// 定义sql  
String sql = "select *from tb_user where gender = ?"; ②  
//获取PreparedStatement对象  
PreparedStatement pstmt = conn.prepareStatement(sql);  
// 设置? 的值  
pstmt.setString(1,gender); ③  
//执行sql  
ResultSet rs = pstmt.executeQuery();  
//遍历Result, 获取数据  
User user = null;  
ArrayList<User> users = new ArrayList<>();  
while (rs.next()) {  
    //获取数据  
    int id = rs.getInt("id");  
    String username = rs.getString("username");  
    String password = rs.getString("password"); ④  
    //创建对象, 设置属性值  
    user = new User();  
    user.setId(id);  
    user.setUsername(username);  
    user.setPassword(password);  
    user.setGender(gender);  
    //装入集合  
    users.add(user);  
}
```

- 硬编码

- 注册驱动、获取连接

上图标1的代码有很多字符串，而这些是连接数据库的四个基本信息，以后如果要将Mysql数据库换成其他的关系型数据库的话，这四个地方都需要修改，如果放在此处就意味着要修改我们的源代码。

- SQL语句

上图标2的代码。如果表结构发生变化，SQL语句就要进行更改。这也不方便后期的维护。

- 操作繁琐

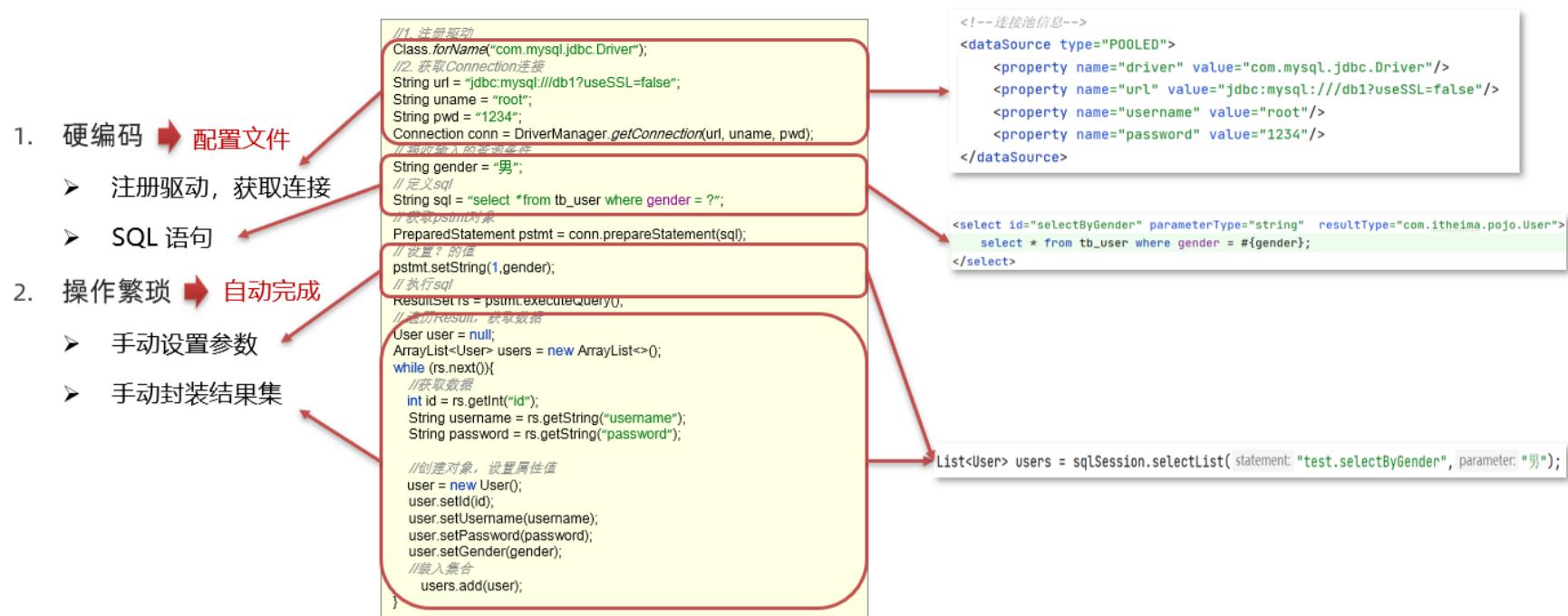
- 手动设置参数
 - 手动封装结果集

上图标4的代码是对查询到的数据进行封装，而这部分代码是没有什么技术含量，而且特别耗费时间的。

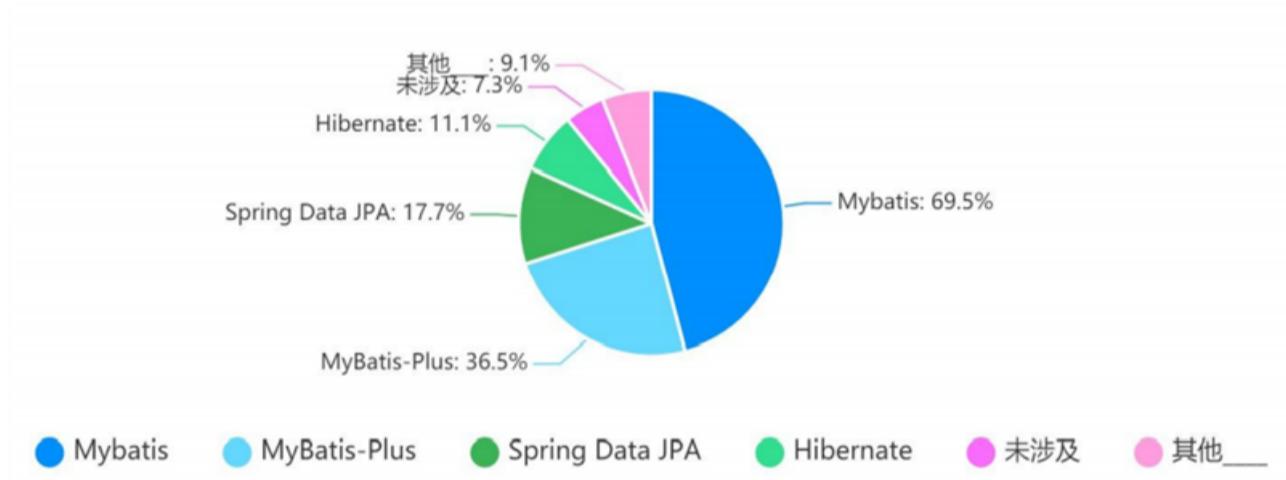
2.1.3 Mybatis 优化

- 硬编码可以配置到**配置文件**
- 操作繁琐的地方mybatis都**自动完成**

如图所示



下图是持久层框架的使用占比。



2.2 Mybatis快速入门

需求：查询user表中所有的数据

- 创建user表，添加数据

```

1 create database mybatis;
2 use mybatis;
3
4 drop table if exists tb_user;
5
6 create table tb_user(
7     id int primary key auto_increment,
8     username varchar(20),
9     password varchar(20),
10    gender char(1),
11    addr varchar(30)
12 );
13
14 INSERT INTO tb_user VALUES (1, 'zhangsan', '123', '男', '北京');
15 INSERT INTO tb_user VALUES (2, '李四', '234', '女', '天津');
16 INSERT INTO tb_user VALUES (3, '王五', '11', '男', '西安');

```

- 创建模块，导入坐标

在创建好的模块中的 pom.xml 配置文件中添加依赖的坐标

```

1 <dependencies>
2     <!--mybatis 依赖-->
3     <dependency>
4         <groupId>org.mybatis</groupId>
5         <artifactId>mybatis</artifactId>
6         <version>3.5.5</version>
7     </dependency>
8
9     <!--mysql 驱动-->
10    <dependency>

```

```

11      <groupId>mysql</groupId>
12      <artifactId>mysql-connector-java</artifactId>
13      <version>5.1.46</version>
14  </dependency>
15
16  <!--junit 单元测试-->
17  <dependency>
18      <groupId>junit</groupId>
19      <artifactId>junit</artifactId>
20      <version>4.13</version>
21      <scope>test</scope>
22  </dependency>
23
24  <!-- 添加slf4j日志api -->
25  <dependency>
26      <groupId>org.slf4j</groupId>
27      <artifactId>slf4j-api</artifactId>
28      <version>1.7.20</version>
29  </dependency>
30  <!-- 添加logback-classic依赖 -->
31  <dependency>
32      <groupId>ch.qos.logback</groupId>
33      <artifactId>logback-classic</artifactId>
34      <version>1.2.3</version>
35  </dependency>
36  <!-- 添加logback-core依赖 -->
37  <dependency>
38      <groupId>ch.qos.logback</groupId>
39      <artifactId>logback-core</artifactId>
40      <version>1.2.3</version>
41  </dependency>
42 </dependencies>

```

注意：需要在项目的 resources 目录下创建logback的配置文件

- 编写 MyBatis 核心配置文件 --> 替换连接信息 解决硬编码问题

在模块下的 resources 目录下创建mybatis的配置文件 `mybatis-config.xml`，内容如下：

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6
7     <typeAliases>
8         <package name="com.itheima.pojo"/>
9     </typeAliases>
10
11    <!--
12        environments: 配置数据库连接环境信息。可以配置多个environment，通过default属性切换不同的
13        environment
14        -->
15        <environments default="development">
16            <environment id="development">
17                <transactionManager type="JDBC"/>
18                <dataSource type="POOLED">
19                    <!--数据库连接信息-->
20                    <property name="driver" value="com.mysql.jdbc.Driver"/>
21                    <property name="url" value="jdbc:mysql:///mybatis?useSSL=false"/>
22                    <property name="username" value="root"/>
23                    <property name="password" value="1234"/>
24                </dataSource>
25            </environment>
26            <environment id="test">

```

```

27         <transactionManager type="JDBC"/>
28     <dataSource type="POOLED">
29         <!--数据库连接信息-->
30         <property name="driver" value="com.mysql.jdbc.Driver"/>
31         <property name="url" value="jdbc:mysql:///mybatis?useSSL=false"/>
32         <property name="username" value="root"/>
33         <property name="password" value="1234"/>
34     </dataSource>
35   </environment>
36 </environments>
37 <mappers>
38   <!--加载sql映射文件-->
39   <mapper resource="UserMapper.xml"/>
40 </mappers>
41 </configuration>

```

- 编写 SQL 映射文件 -> 统一管理sql语句，解决硬编码问题

在模块的 `resources` 目录下创建映射配置文件 `UserMapper.xml`，内容如下：

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3 <mapper namespace="test">
4   <select id="selectAll" resultType="com.itheima.pojo.User">
5     select * from tb_user;
6   </select>
7 </mapper>

```

- 编码

- 在 `com.itheima.pojo` 包下创建 User类

```

1 public class User {
2     private int id;
3     private String username;
4     private String password;
5     private String gender;
6     private String addr;
7
8     //省略了 setter 和 getter
9 }

```

- 在 `com.itheima` 包下编写 MybatisDemo 测试类

```

1 public class MyBatisDemo {
2
3     public static void main(String[] args) throws IOException {
4         //1. 加载mybatis的核心配置文件，获取 SqlSessionFactory
5         String resource = "mybatis-config.xml";
6         InputStream inputStream = Resources.getResourceAsStream(resource);
7         SqlSessionFactory sqlSessionFactory = new
8         SqlSessionFactoryBuilder().build(inputStream);
9
10        //2. 获取SqlSession对象，用它来执行sql
11        SqlSession sqlSession = sqlSessionFactory.openSession();
12        //3. 执行sql
13        List<User> users = sqlSession.selectList("test.selectAll"); //参数是一个字符串，该
14        //字符串必须是映射配置文件的namespace.id
15        System.out.println(users);
16        //4. 释放资源
17        sqlSession.close();
18    }
19 }

```

解决SQL映射文件的警告提示：

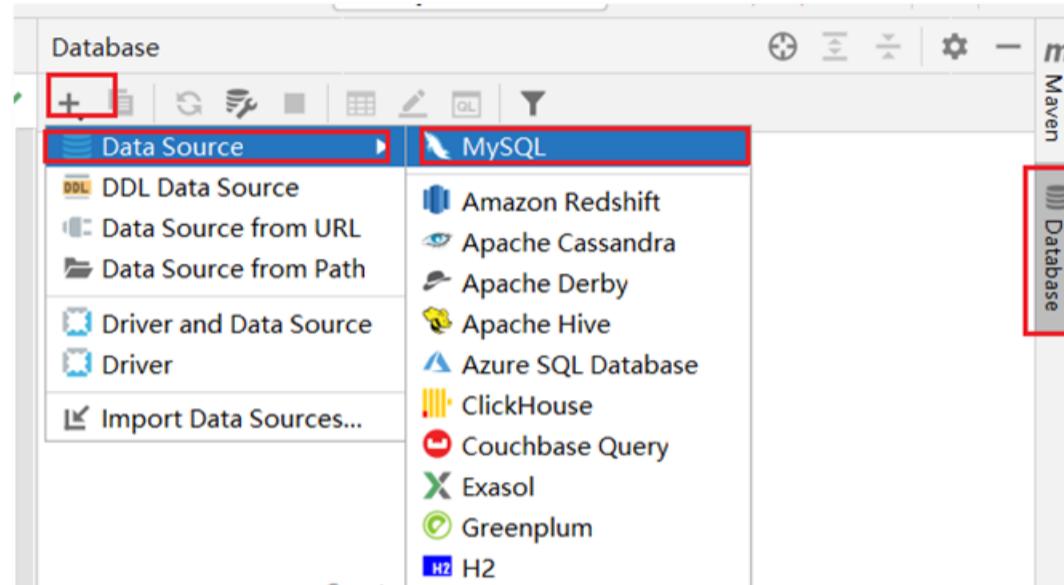
在入门案例映射配置文件中存在报红的情况。问题如下：

```
<mapper namespace="test">
    <select id="selectAll" resultType="com.itheima.pojo.User">
        select * from tb_user;
    </select>
</mapper>
```

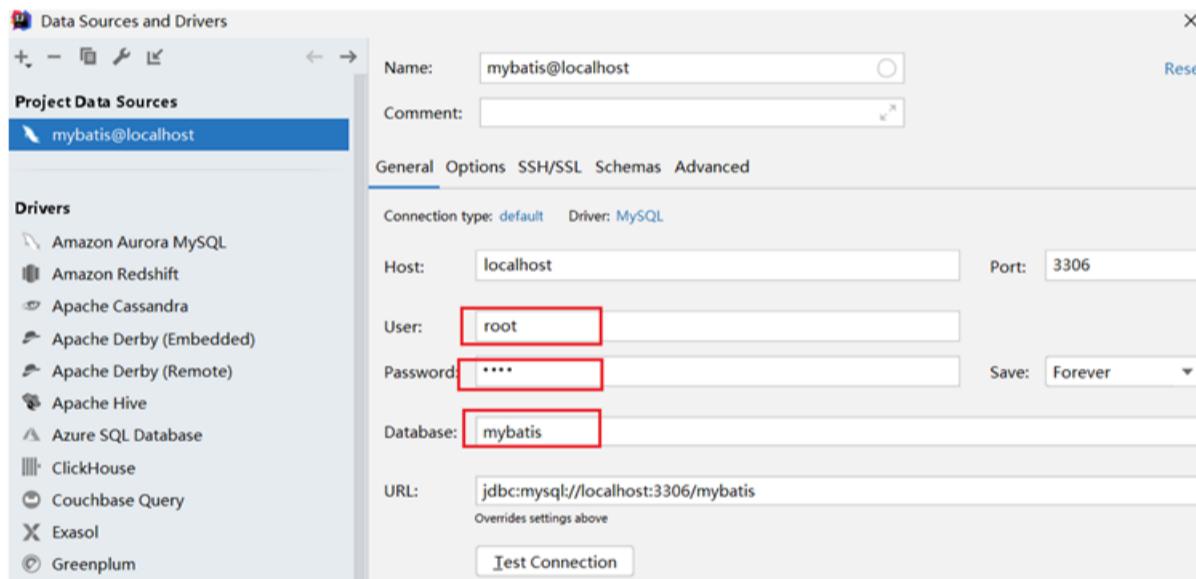
- 产生的原因：Idea和数据库没有建立连接，不识别表信息。但是大家一定要记住，它并不影响程序的执行。
- 解决方式：在Idea中配置MySQL数据库连接。

IDEA中配置MySQL数据库连接

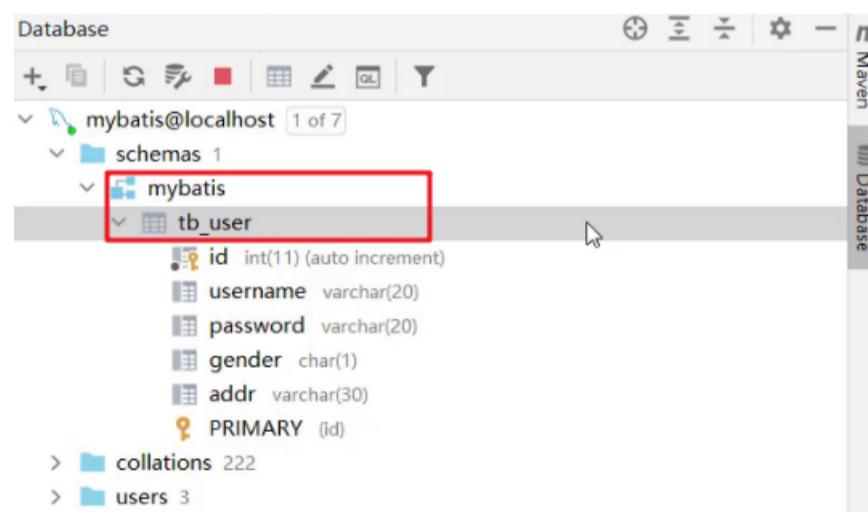
- 点击IDEA右边框的 **Database**，在展开的界面点击 **+** 选择 **Data Source**，再选择 **MySQL**



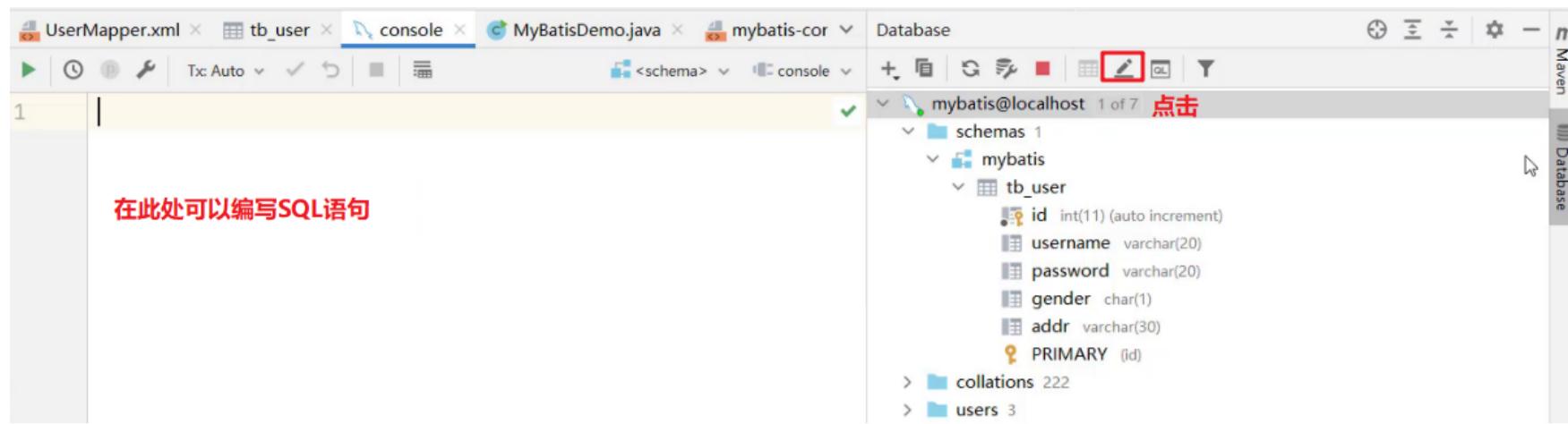
- 在弹出的界面进行基本信息的填写



- 点击完成后就能看到如下界面



而此界面就和 navicat 工具一样可以进行数据库的操作。也可以编写SQL语句



2.3 Mapper代理开发

2.3.1 Mapper代理开发概述

之前我们写的代码是基本使用方式，它也存在硬编码的问题，如下：

```
//3. 执行sql
List<User> users = sqlSession.selectList(statement: "test.selectAll");
System.out.println(users);
```

这里调用 `selectList()` 方法传递的参数是映射配置文件中的 `namespace.id` 值。这样写也不便于后期的维护。如果使用 Mapper 代理方式（如下图）则不存在硬编码问题。

```
//3. 获取接口代理对象
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
//4. 执行方法，其实就是执行sql语句
List<User> users = userMapper.selectAll();
```

通过上面的描述可以看出 Mapper 代理方式的目的：

- 解决原生方式中的硬编码
- 简化后期执行SQL

Mybatis 官网也是推荐使用 Mapper 代理的方式。下图是截止官网的图片

为了这个简单的例子，我们似乎写了不少配置，但其实并不多。在一个 XML 映射文件中，可以定义无数个映射语句，这样一来，XML 头部和文档类型声明部分就显得微不足道了。文档的其它部分很直白，容易理解。它在命名空间 “org.mybatis.example.BlogMapper” 中定义了一个名为 “selectBlog” 的映射语句，这样你就可以用全限定名 “org.mybatis.example.BlogMapper.selectBlog” 来调用映射语句了，就像上面例子中那样：

```
Blog blog = (Blog) session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);
```

你可能会注意到，这种方式和用全限定名调用 Java 对象的方法类似。这样，该命名就可以直接映射到在命名空间中同名的映射器类，并将已映射的 select 语句匹配到对应名称、参数和返回类型的方法。因此你就可以像上面那样，不费吹灰之力地在对应的映射器接口调用方法，就像下面这样：

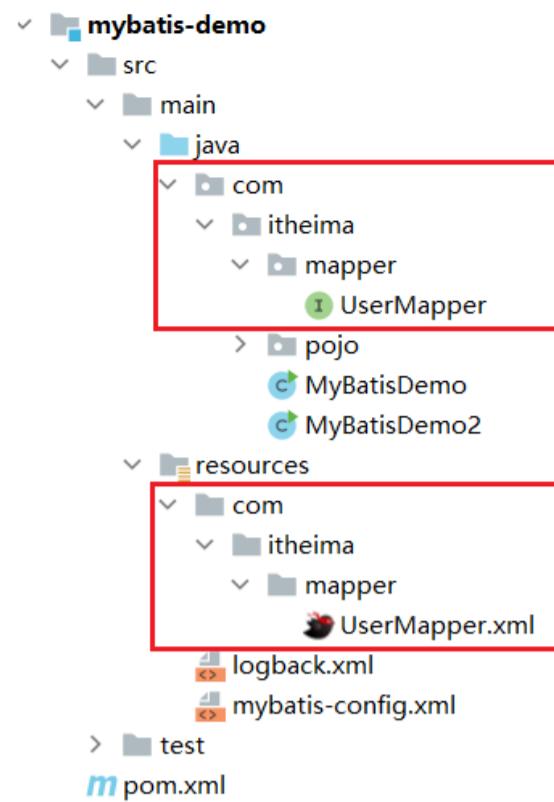
```
BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

第二种方法有很多优势，首先它不依赖于字符串字面值，会更安全一点；其次，如果你的 IDE 有代码补全功能，那么代码补全可以帮你快速选择到映射好的 SQL 语句。

2.3.2 使用Mapper代理要求

使用Mapper代理方式，必须满足以下要求：

- 定义与SQL映射文件同名的Mapper接口，并且将Mapper接口和SQL映射文件放置在同一目录下。如下图：



- 设置SQL映射文件的namespace属性为Mapper接口全限定名

```

<!--
  namespace: 名称空间。必须是对应接口的全限定名
-->
<mapper namespace="com.itheima.mapper.UserMapper">

```

- 在Mapper接口中定义方法，方法名就是SQL映射文件中sql语句的id，并保持参数类型和返回值类型一致

```

UserMapper.xml
<!--
  namespace: 名称空间。必须是对应接口的全限定名
-->
<mapper namespace="com.itheima.mapper.UserMapper">
  <select id="selectAll" resultType="com.itheima.pojo.User">
    select *
    from tb_user;
  </select>
</mapper>

UserMapper.java
package com.itheima.mapper;

import ...

public interface UserMapper {
  List<User> selectAll();
}

```

2.3.3 案例代码实现

- 在 `com.itheima.mapper` 包下创建 `UserMapper` 接口，代码如下：

```

1 public interface UserMapper {
2     List<User> selectAll();
3     User selectById(int id);
4 }

```

- 在 `resources` 下创建 `com/itheima/mapper` 目录，并在该目录下创建 `UserMapper.xml` 映射配置文件

```

1 <!--
2   namespace: 名称空间。必须是对应接口的全限定名
3 -->
4 <mapper namespace="com.itheima.mapper.UserMapper">
5   <select id="selectAll" resultType="com.itheima.pojo.User">
6     select *
7     from tb_user;
8   </select>
9 </mapper>

```

- 在 `com.itheima` 包下创建 `MybatisDemo2` 测试类，代码如下：

```

1  /**
2  * Mybatis 代理开发
3  */
4  public class MyBatisDemo2 {
5
6      public static void main(String[] args) throws IOException {
7
8          //1. 加载mybatis的核心配置文件, 获取 SqlSessionFactory
9          String resource = "mybatis-config.xml";
10         InputStream inputStream = Resources.getResourceAsStream(resource);
11         SqlSessionFactory sqlSessionFactory = new
12         SqlSessionFactoryBuilder().build(inputStream);
13
14         //2. 获取SqlSession对象, 用它来执行sql
15         SqlSession sqlSession = sqlSessionFactory.openSession();
16         //3. 执行sql
17         //3.1 获取UserMapper接口的代理对象
18         UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
19         List<User> users = userMapper.selectAll();
20
21         System.out.println(users);
22         //4. 释放资源
23         sqlSession.close();
24     }
25 }
```

注意：

如果Mapper接口名称和SQL映射文件名称相同，并在同一目录下，则可以使用包扫描的方式简化SQL映射文件的加载。也就是将核心配置文件的加载映射配置文件的配置修改为

```

1 <mappers>
2     <!--加载sql映射文件-->
3     <!-- <mapper resource="com/itheima/mapper/UserMapper.xml"/>-->
4     <!--Mapper代理方式-->
5     <package name="com.itheima.mapper"/>
6 </mappers>
```

2.4 核心配置文件

核心配置文件中现有的配置之前已经给大家进行了解释，而核心配置文件中还可以配置很多内容。我们可以通过查询官网看可以配置的内容



配置

MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息。配置文档的顶层结构如下：

- configuration (配置)
 - properties (属性)
 - settings (设置)
 - typeAliases (类型别名)
 - typeHandlers (类型处理器)
 - objectFactory (对象工厂)
 - plugins (插件)
 - environments (环境配置)
 - environment (环境变量)
 - transactionManager (事务管理器)
 - dataSource (数据源)
 - databaseIdProvider (数据库厂商标识)
 - mappers (映射器)

接下来我们先对里面的一些配置进行讲解。

2.4.1 多环境配置

在核心配置文件的 `environments` 标签中其实是可以配置多个 `environment`，使用 `id` 给每段环境起名，在 `environments` 中使用 `default='环境id'` 来指定使用哪儿段配置。我们一般就配置一个 `environment` 即可。

```
1 <environments default="development">
2   <environment id="development">
3     <transactionManager type="JDBC"/>
4     <dataSource type="POOLED">
5       <!--数据库连接信息-->
6       <property name="driver" value="com.mysql.jdbc.Driver"/>
7       <property name="url" value="jdbc:mysql://mybatis?useSSL=false"/>
8       <property name="username" value="root"/>
9       <property name="password" value="1234"/>
10    </dataSource>
11  </environment>
12
13  <environment id="test">
14    <transactionManager type="JDBC"/>
15    <dataSource type="POOLED">
16      <!--数据库连接信息-->
17      <property name="driver" value="com.mysql.jdbc.Driver"/>
18      <property name="url" value="jdbc:mysql://mybatis?useSSL=false"/>
19      <property name="username" value="root"/>
20      <property name="password" value="1234"/>
21    </dataSource>
22  </environment>
23 </environments>=
```

2.4.2 类型别名

在映射配置文件中的 `resultType` 属性需要配置数据封装的类型（类的全限定名）。而每次这样写是特别麻烦的，Mybatis 提供了 `类型别名 (typeAliases)` 可以简化这部分的书写。

首先需要在核心配置文件中配置类型别名，也就意味着给pojo包下所有的类起了别名（别名就是类名），不区分大小写。内容如下：

```
1 <typeAliases>
2   <!--name属性的值是实体类所在包-->
3   <package name="com.itheima.pojo"/>
4 </typeAliases>
```

通过上述的配置，我们就可以简化映射配置文件中 `resultType` 属性值的编写

```
1 <mapper namespace="com.itheima.mapper.UserMapper">
2   <select id="selectAll" resultType="user">
3     select * from tb_user;
4   </select>
5 </mapper>
```