软件工程环境

学习目标

>

第三方库

背景

IT行业流行一句话，叫做“不要重复造轮子”，以Java语言为例：

- JDK已提供的功能，可以通过相应的 API 直接使用，不用自己重新实现。
- JDK没有提供的功能，在富有开源精神的 IT行业，大部分通用功能也都能在开源社区找到。

概念

某个编程语言在安装好开发环境以后，默认会提供基础API。除此之外，还可以由第三方提供代码库（我们称为第三方库，一般为后缀为jar的文件），我们可以通过第三方库提供的API来使用某些已有的功能，而不用重复造轮子。

> 第三方库jar文件为一个压缩文件，可以使用解压软件打开，里边包含的都是class文件，即编译好的可在JVM上运行的字节码。

一个Java程序基于某个第三方库来实现某些功能时，该第三方库也称为该Java程序的**依赖包。**

第三方库和API的关系

API，即Application Programming Interface，应用程序接口。

为软件库开放出来的功能调用接口。这里的接口指调用方，被调用方约定的一组调用规范定义。大家在JDK API文档中看到的都是API，如：

> java.util
>
> ## Class Arrays
>
> | 返回类型 | 方法名 | 方法描述 |
> | --- | --- | --- |
> | `static <T> List<T>` | `asList(T... a)` | Returns a fixed-size list backed by the specified array. |

> 以上为Arrays工具类提供的静态方法 `asList`，包括方法名，方法参数（可变的多个泛型对象），返回类型（实现List接口的对象），整个方法定义构成了Arrays工具类提供的API：将某类型的多个对象转换为填充了这些对象的List。

第三方库和API的关系，类似于餐馆和点餐，对于餐馆（第三方库，被调用方）提供的点餐服务（API）来说，消费者（调用方）不用关心餐馆内部如何实现。

如何使用第三方库

在Java中，使用第三方库，主要体现在开发java代码时引入依赖，编译时添加编译依赖，及运行时添加运行依赖。

示例：`jansi`是一个提供控制台输出彩色字符的第三方库，引入该依赖，并输出彩色的内容。

步骤一：开发java程序

使用普通文本编辑器（如记事本，notepad++，Visual Studio Code等）编写java代码后保存在本地

先使用`import`引入依赖，再打印彩色内容：

```
import org.fusesource.jansi.AnsiConsole;

import static org.fusesource.jansi.Ansi.Color.*;
import static org.fusesource.jansi.Ansi.ansi;

public class Main {

    public static void main(String[] args) {
        AnsiConsole.systemInstall();
        int lineNumber = 0;
        System.out.print( ansi().fg(MAGENTA).a(++lineNumber));
        System.out.println( ansi().fg(CYAN).a("面试官：").fg(RED).a("你都会什么编程语言？") );
        System.out.print( ansi().fg(MAGENTA).a(++lineNumber));
        System.out.println( ansi().fg(BLUE).a("我：     ").fg(GREEN).a("精通C、C++、Java、HTML、JavaScript等语言，的单词拼写") );
        System.out.print( ansi().fg(MAGENTA).a(++lineNumber));
        System.out.println( ansi().fg(CYAN).a("面试官：").fg(RED).a("就这？会什么系统嘛？") );
        System.out.print( ansi().fg(MAGENTA).a(++lineNumber));
        System.out.println( ansi().fg(BLUE).a("我：     ").fg(GREEN).a("精通Windows、Mac、Linux系统，的关机") );
        AnsiConsole.systemUninstall();
    }
}
```

步骤二：javac编译

参考：javac编译命令说明

先下载`jansi`依赖包，下载好保存在本地任意路径：jansi下载链接

再使用`javac`命令来编译（注意依赖包路径要改成自己的）：

```
javac -cp "E:/test/lib/jansi-2.3.3.jar" -encoding UTF-8 Main.java
```

执行成功，cmd不会出现异常报错信息，并且会在当前路径下生成Main.class文件。

步骤三：java运行

参考：java运行命令说明

使用`java`命令来运行（注意依赖包路径要改成自己的）：

```
java -cp ".;E:/test/lib/jansi-2.3.3.jar" Main
```

输出结果如下：

```
E:\test>java -cp ".;E:/test/lib/jansi-2.3.3.jar" Main
1面试官：你都会什么编程语言？
2我：    精通C、C++、Java、HTML、JavaScript等语言，的单词拼写
3面试官：就这？会什么系统嘛？
4我：    精通Windows、Mac、Linux系统，的关机
```

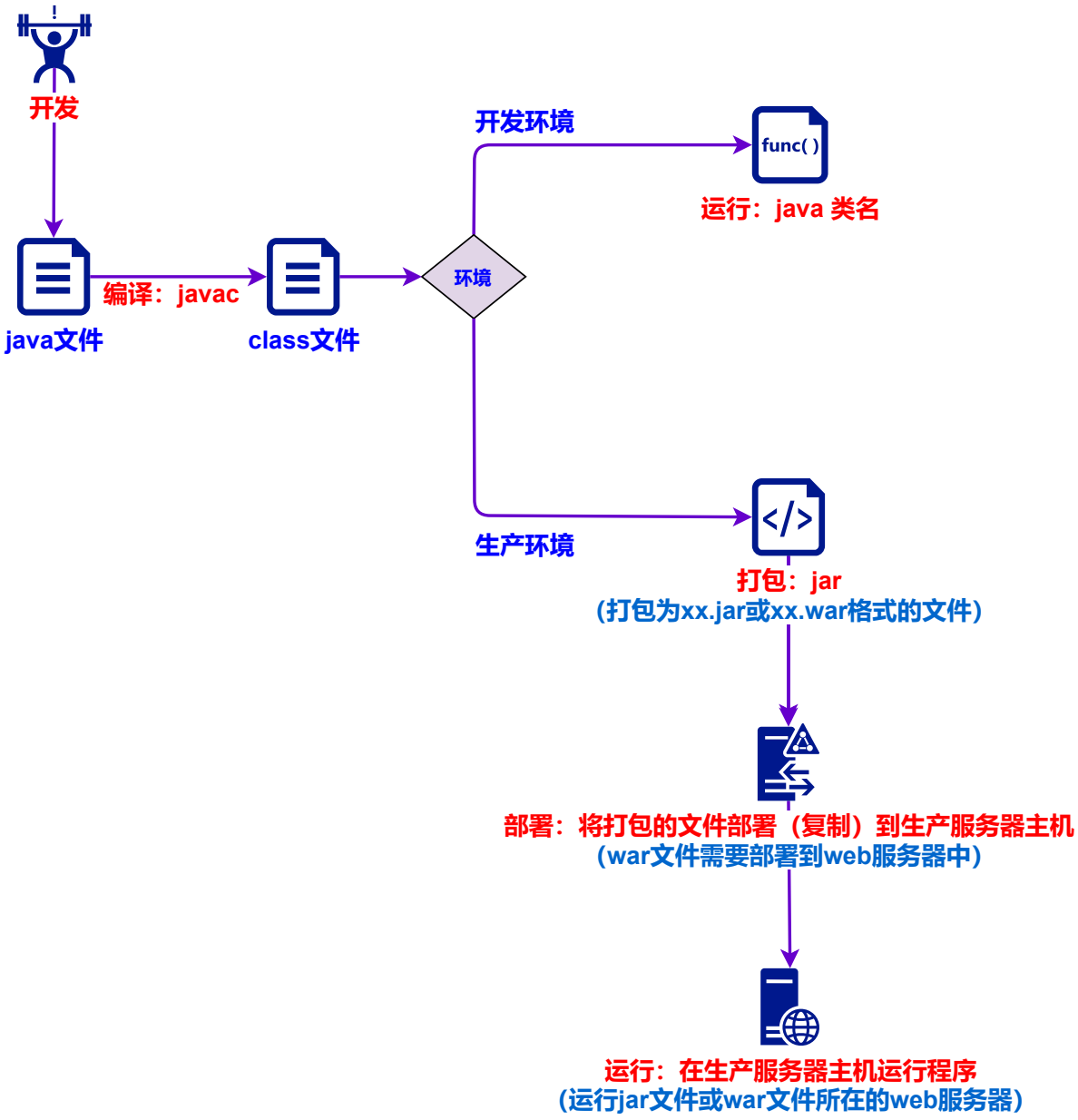
Java程序标准流程

流程

对于真实的java程序流程来说，要分为开发环境和测试环境，生产环境等等，不同环境的流程都不一样：

- 开发环境中，一般开发完java代码后，编译并运行即可。
- 生产环境中，不是在开发人员电脑中运行程序，而是在生产的服务器主机中运行

详细流程如下：



编译：javac

关于 `javac` 命令的参数说明

我们主要关注以下几个常用参数：

参数语法	语法说明	示例
-cp *path* or -classpath *path*	指定编译时依赖包的路径。 多个依赖包时： （1）Windows中以 `;` 间隔 （2）Mac/Linux中以 `:` 间隔 最好整个路径前后加上双引号	javac -cp "E:/test/lib/jansi-2.3.3.jar;E:/test/lib/junit-4.13.1.jar" 类名.java
-Djava.ext.dirs=*directories*	指定编译时依赖包所在目录	javac -Djava.ext.dirs="E:/test/lib" 类名.java
-d *directory*	指定编译生成的class文件存放路径 指定的路径需要存在 如果不设置，默认保存在当前路径	javac -d "E:/test/classes" 类名.java
-encoding *encoding*	指定编译时的编码格式，一般使用UTF-8	javac -encoding UTF-8 类名.java

运行：java

关于 `java` 命令的参数说明

我们主要关注以下常用参数：

参数语法	语法说明	示例
-cp *path* or -classpath *path*	指定运行时依赖包的路径。 和javac不同： 使用 `.` 代表原来的类加载路径，追加第三方依赖包	java -cp ".;E:/test/lib/jansi-2.3.3.jar;E:/test/lib/junit-4.13.1.jar" 类名
-Dfile.encoding=编码	指定运行时的编码格式	java -Dfile.encoding=UTF-8 类名

> 执行java程序会：
>
> 1. 开启java虚拟机
> 2. 加载class字节码到内存
> 3. java虚拟机执行字节码指令（翻译为机器码）
> 4. cpu执行机器码

打包：jar

在项目代码很多时，常常需要把整个项目代码，包括依赖包，都打包为一个压缩文件：

- 如果是web项目（后边学习，这里简单了解即可），会打包为war文件。
- 一般的项目，打包为jar文件，对于jar文件来说，一般是以下作用：
  - 作为依赖包提供给其他项目使用
  - 作为应用程序，包含一个入口类（main主函数的类），可以直接使用 `java -jar 文件名` 执行。

> jar文件中需要包含 `META-INF/MANIFEST.MF` 文件，作为描述信息，格式为键: 值，我们主要关注两个：
>
> ```
> Main-Class: 入口类的全限定名（包名+类名）
> Class-Path: 指定jar文件运行时依赖包的路径。多个依赖包时：（1）Windows中以 ; 间隔
> （2）Mac/Linux中以 : 间隔
> ```

对于以上的打包操作来说，jdk提供了jar命令进行打包操作：关于 `jar` 命令的参数说明

我们主要关注以下几个常用参数：

参数语法	语法说明
c	创建一个jar文件
f	指定生成的jar文件名
v	输出命令执行结果
e	指定jar文件中的入口类（主函数所在类）

jar命令使用格式为：

- 作为依赖包提供给其他项目使用：

  ```
  jar cvf 生成的文件名 包含的文件或目录[多个之间Windows以 ; 间隔，Mac或Linux以 : 间隔]
  ```

- 作为应用程序：

  ```
  jar cvfe 生成的文件名 入口类的全限定名 包含的文件或目录
  ```

  以上命令可以生成 `META-INF/MANIFEST.MF` 及 `Main-Class` ，但需要手动配置 `Class-Path`

示例

将以上使用 `jansi` 第三方库的 class文件打包为jar文件并执行：

```
jar cvfe main.jar Main Main.class
```

以下为执行结果：

```
E:\test>jar cvfe main.jar Main Main.class
已添加清单
正在添加: Main.class(输入 = 1455) (输出 = 818)(压缩了 43%)
```

以上命令执行成功会在当前路径下，生成一个main.jar的文件，用解压工具（如WinRAR）打开，再双击使用编辑器打开其中的 `META-INF/MANIFEST.MF` 文件，添加 `Class-Path`：

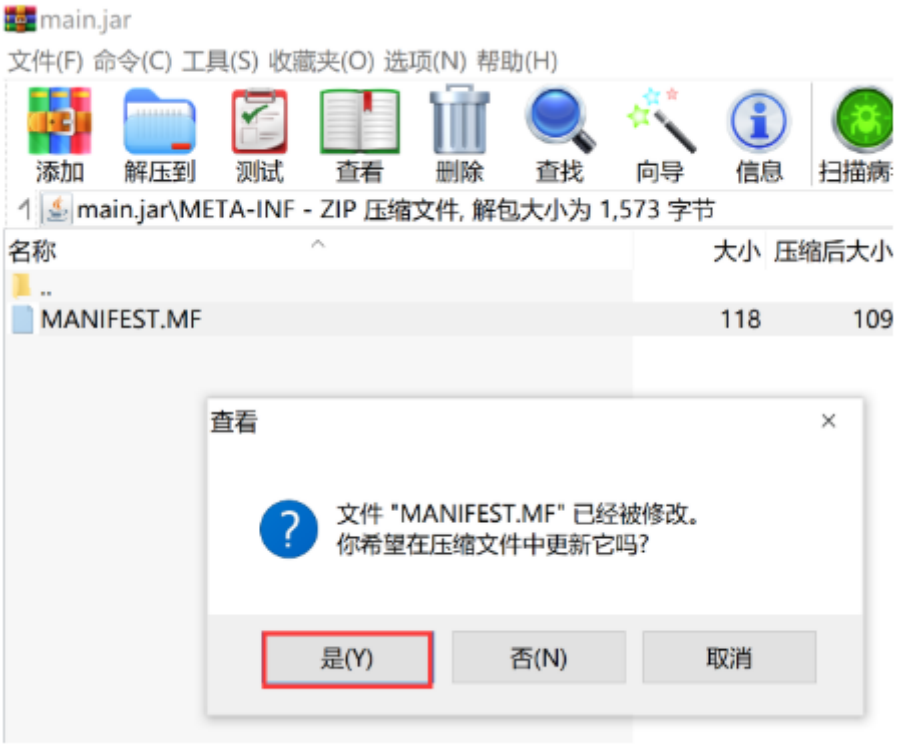
以下为文件内容：

```
Manifest-Version: 1.0
Created-By: 1.8.0_281 (Oracle Corporation)
Main-Class: Main
```

在第4行添加 `Class-Path`，设置为 `jansi` 依赖包的路径（建议设置为生成的main.jar包到 jansi 依赖包的相对路径），修改后为：

```
Manifest-Version: 1.0
Created-By: 1.8.0_281 (Oracle Corporation)
Main-Class: Main
Class-Path: lib/jansi-2.3.3.jar
```

注意修改完成后，在解压工具中更新：



修改完成后，即可以直接运行：

```
java -jar main.jar
```

构建工具

背景（了解）

通过以上Java程序的流程可以看出，在不同的阶段，需要使用原生的JDK命令来进行相关处理，且使用时并不太方便：

- 依赖包需要手动下载：如果依赖包还有自身的依赖，需要递归下载。
- 编译（javac）、打包（jar）、运行（java）需要指定依赖包路径参数，即便使用IDEA等工具，还需要配置后才能操作，比较麻烦。
- 了解：打包后的结构作为生产环境运行结构，需要和开发环境运行结构一致：除了依赖包，可能还需要在运行时加载配置文件、资源文件等。此时手动操作一般是复制到编译路径，将整个编译路径中的文件打包；或使用IDEA等工具，进行配置后再操作。
- 上传到代码库中的项目代码、依赖包、配置文件和资源文件，其他开发人员下载以后，还需要重新配置，使用非常不方便。

由于以上操作比较繁琐，使用不便，就有了项目构建工具来专门做这种项目构建的工作。

常见构建工具（了解）

对于Java项目来说，使用的比较广泛的构建工具主要有三种：

- Ant：比较老的一款构建工具，构建时需要自行指定编译路径，资源路径等，依赖包还是需要自行下载。目前主流的项目都不再使用。

- Maven：采取“约定优于配置”的思想，约定好编译路径，资源路径等（也可以配置修改），依赖包可以从网络自行下载，Maven主要基于XML文件进行配置。目前Java服务端项目大多使用Maven来构建。
- Gradle：和Maven功能非常类似，只是采取基于Groovy脚本语言的DSL语法进行配置。安卓开发几乎都是基于Gradle。

以上构建工具，我们主要学习Maven。

关于Maven

Maven是一个项目构建工具，创建的项目只要遵循Maven规范（称为Maven项目），即可使用Maven来进行管理：编译，打包等。

Maven安装

下载解压

先下载Maven，为了后续开发环境一致，我们使用统一版本：Maven 3.6.3下载链接

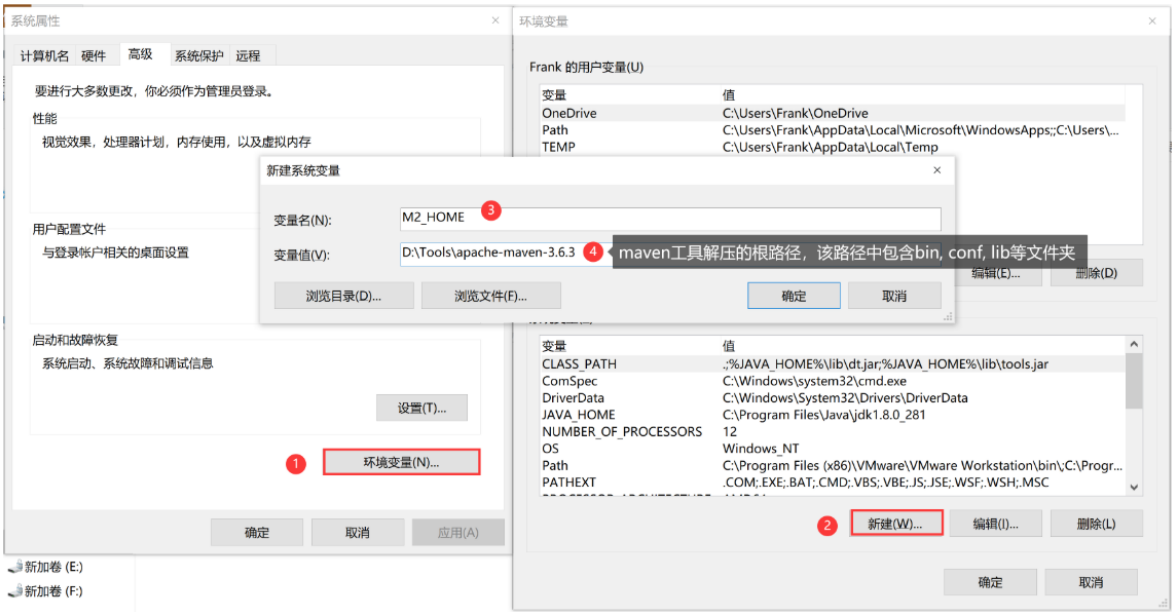
将下载的文件用解压工具解压到本地目录中，解压后的结构如下：

要在cmd中全局使用Maven命令，需要配置环境变量。

Maven环境变量配置

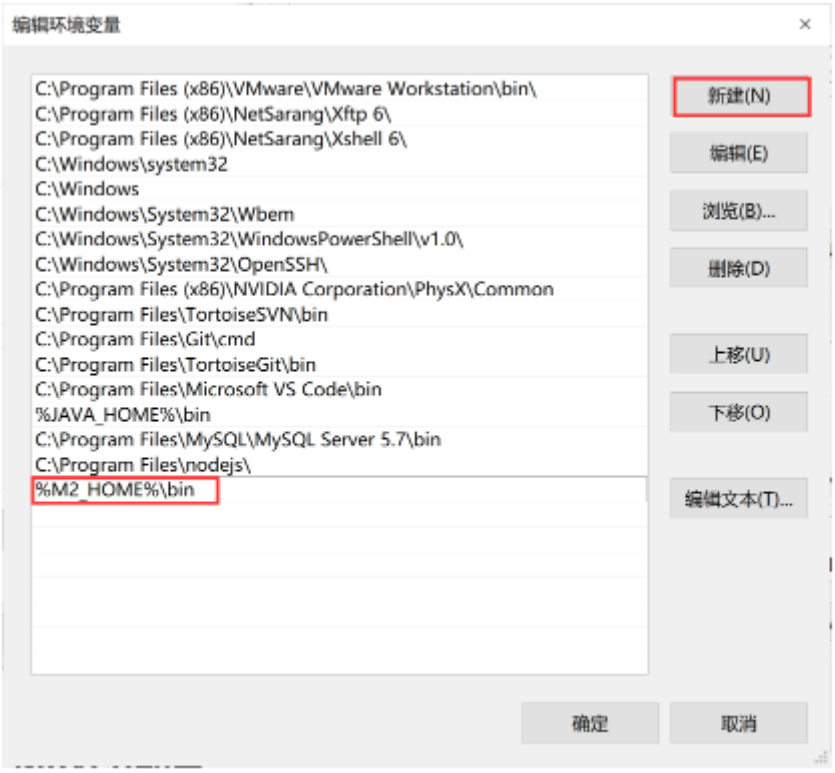
右键”此电脑“，点击”属性“，并打开”高级系统设置“中的”环境变量“，添加Maven环境变量：

先定义Maven根路径变量 `M2_HOME`：



再在 `Path` 环境变量中，添加Maven工具bin目录为全局命令路径：

```
%M2_HOME%\bin
```



检测安装配置是否成功

在cmd中运行Maven命令

```
mvn -version
```

以下为安装配置成功的输出内容：

```
C:\Users\Frank>mvn -version
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: D:\Tools\apache-maven-3.6.3\bin\..
Java version: 1.8.0_281, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk1.8.0_281\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

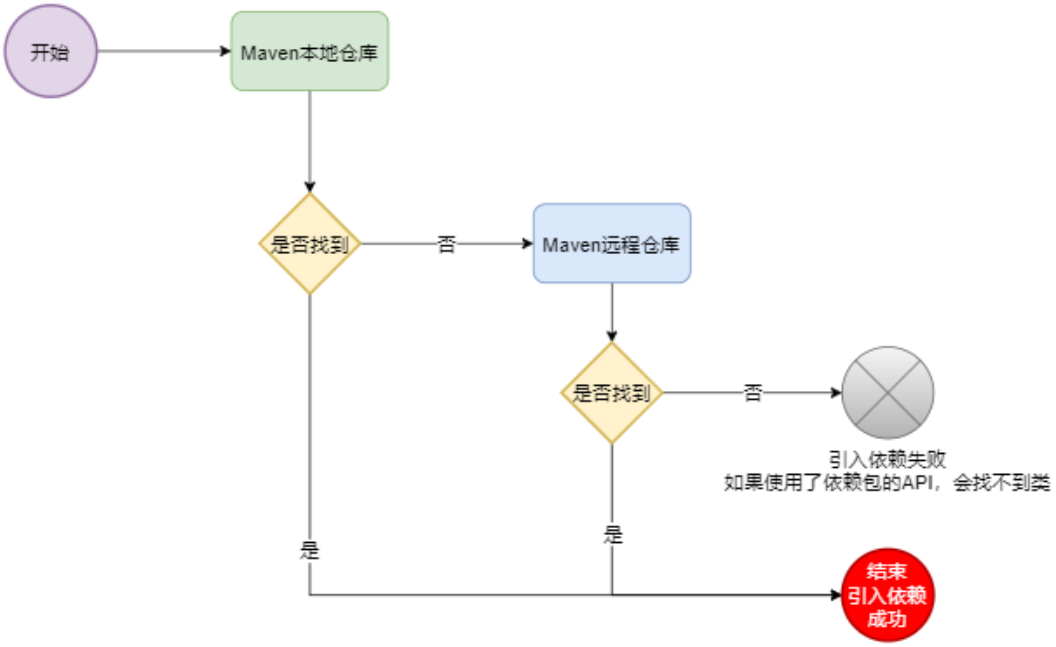
Maven使用及作用

Maven构建工具对Maven项目的管理，我们主要关注以下几点：

全局依赖管理

Maven项目中可以引入依赖包，引入后，加载依赖包的方式为在Maven仓库中搜索。

> Maven仓库可以理解为存放依赖包的仓库，分为本地仓库和远程仓库两种。



在Maven中可以配置以上仓库：

Maven根目录下的 conf 目录中，`settings.xml` 文件即为Maven的全局配置文件，以上的Maven仓库都可以在其中进行配置。我们主要配置本地仓库，远程仓库暂不配置（此时会使用Maven默认的官方远程仓库）。

以下为 `settings.xml` 中，本地仓库配置的地方：

```
46  <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
47            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
48            xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://mave
49    <!-- localRepository
50     | The path to the local repository maven will use to store artifacts.
51     |
52     | Default: ${user.home}/.m2/repository
53    <localRepository>/path/to/local/repo</localRepository>
54    -->
```

在XML中 `<!--` 为注释开始，`-->` 为注释结束，以上 `<localRepository>` 标签即为本地仓库的配置，将该标签所有内容全部复制到注释外，并指定为本地某个目录。以下为修改后的内容：

```
<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
  <!-- localRepository
   | The path to the local repository maven will use to store arti
   |
   | Default: ${user.home}/.m2/repository
  <localRepository>/path/to/local/repo</localRepository>
  -->
  <localRepository>D:/Repository/Maven</localRepository>
```

Maven初次使用从网络中下载大量的依赖包，可以使用我们准备好的本地仓库内容，解压到以上路径即可。注意解压后，在以上路径下的内容为：

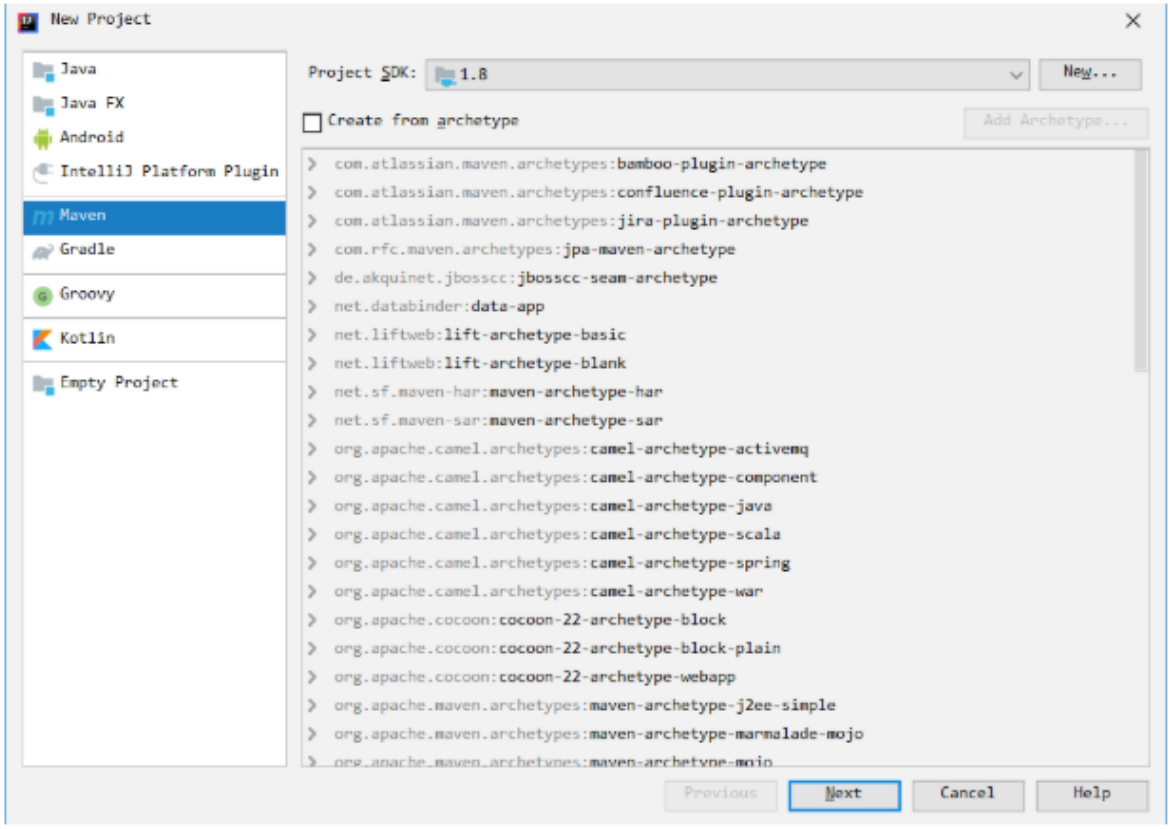
创建Maven项目

我们把符合Maven约定规范的项目称为Maven项目。

在IDEA中可以方便的创建Maven项目，建议先配置IDEA的Maven本地仓库。

参考：附录 - IDEA配置Maven本地仓库

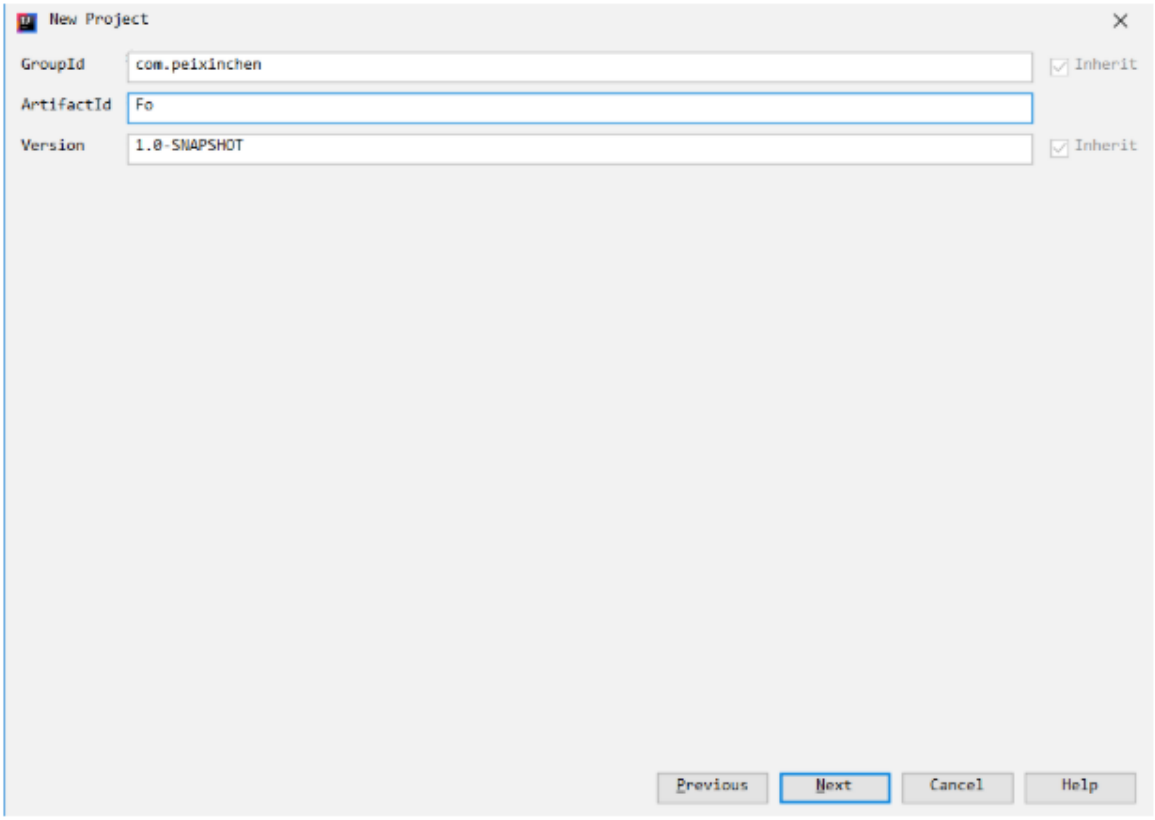
完成以上配置后，在IDEA中新建 maven 类型的项目



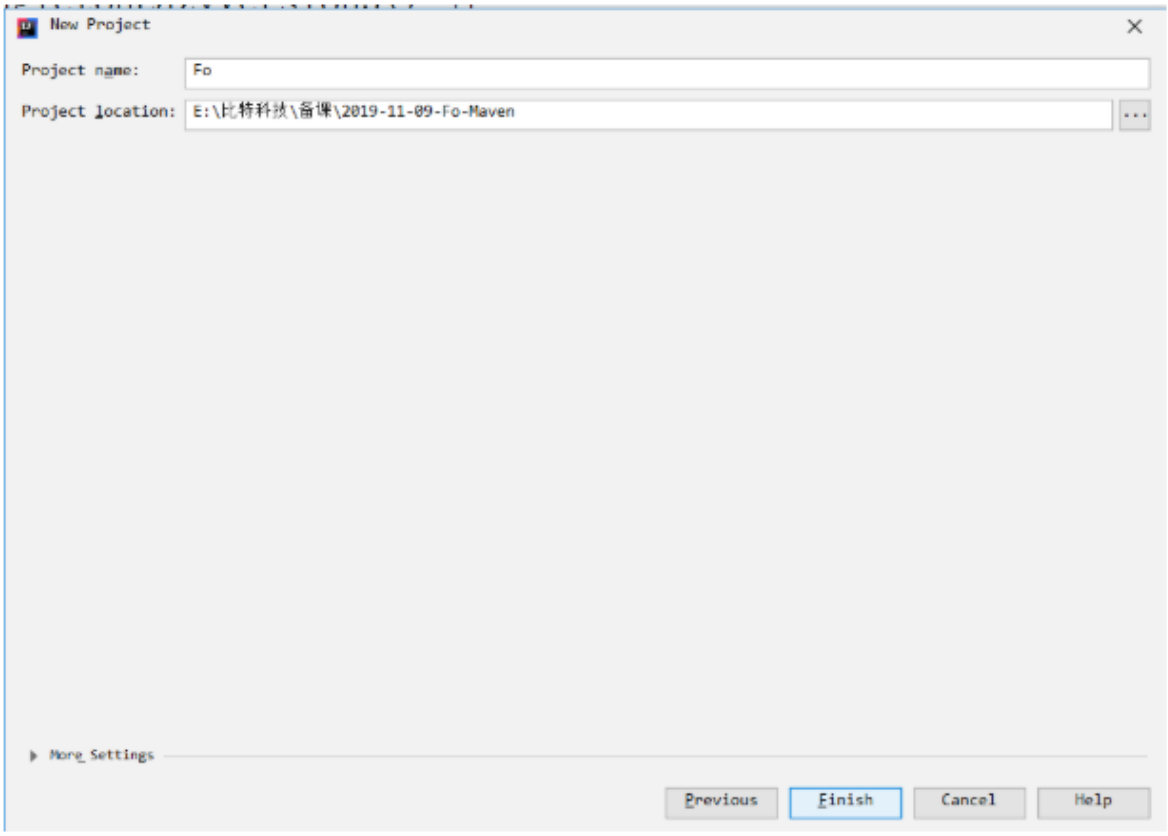
点击 Next 后，需要输入以下信息：

- GroupId：组织id、组织名，一般使用公司的域名，如com.alibaba。同时按规范，自己写的 java 都在这样的包下。个人使用随便取自己英文名或拼音都行。
- ArtifactId：产品id、产品名，项目即产品，所以一般使用本项目名，多个英文之间 `-` 间隔。
- Version：产品的版本号，本项目可能有多个版本提供给别人使用。

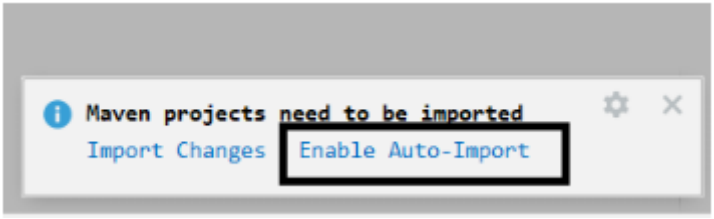
以上三个字段代表了唯一的一个产品，我们创建 Maven 项目，引入其他产品（依赖包）都需要使用这些字段标识唯一的一个产品或项目。



选择项目路径



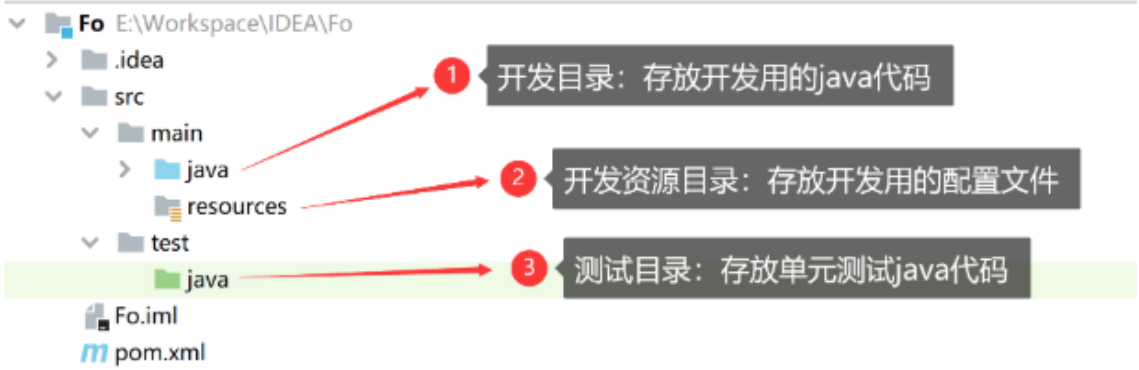
创建完成后，如果右下角出现以下弹窗，需要开启自动导入功能：



至此项目新建完成。

Maven项目结构的默认约定

观察开发项目结构：



> 了解：IDEA创建时，会省略一个 `src/test/resources` 的目录，代表测试资源目录，用于存放单元测试用的配置文件，可以自行创建。

观察编译后的运行时结构：

- 在 `src/main/resources` 目录下，创建一个配置文件（任意文件都行，以下为创建的 `test.xml` 文件）
- 在 `src/main/java` 目录下，随便创建一个类，编写 `main` 方法运行后，会生成编译结构：

Maven项目配置

Maven配置文件：

以上创建的Maven项目中，项目根路径下，会生成一个 `pom.xml` 的配置文件，即为Maven项目的配置文件，Maven项目构建工具就是基于该配置文件来对Maven项目进行管理。

> 该XML文件的格式采取一种叫"项目对象模型"的概念（Project Object Model，简称POM），因此Maven给这个文件取名为 pom.xml
>
> xml中可以使用 `<!-- 这里是注释 -->` 这样的方式注释，IDEA中和 java注释快捷键一样，Ctrl+/

在该配置文件中，我们暂时只关注两点：

- 指定JDK版本：默认情况下，maven 会使用 1.5 版本进行代码检查，需要修改为 1.8。在属性标签 `<properties>` 中配置即可。

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

- 引入依赖包：当前项目需要引入第三方库依赖包时使用。在 `<dependencies>` 标签中配置 `<dependency>` 标签，每个 `<dependency>` 标签为一个引入的依赖包，使用组织id，产品id，版本号标识要引入的依赖包。以下引入 `jansi` 和 `mysql` 两个依赖包：

```
<dependencies>
    <dependency>
        <groupId>org.fusesource.jansi</groupId>
        <artifactId>jansi</artifactId>
        <version>2.3.3</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.49</version>
    </dependency>
</dependencies>
```

以下为全部 `pom.xml` 内容：

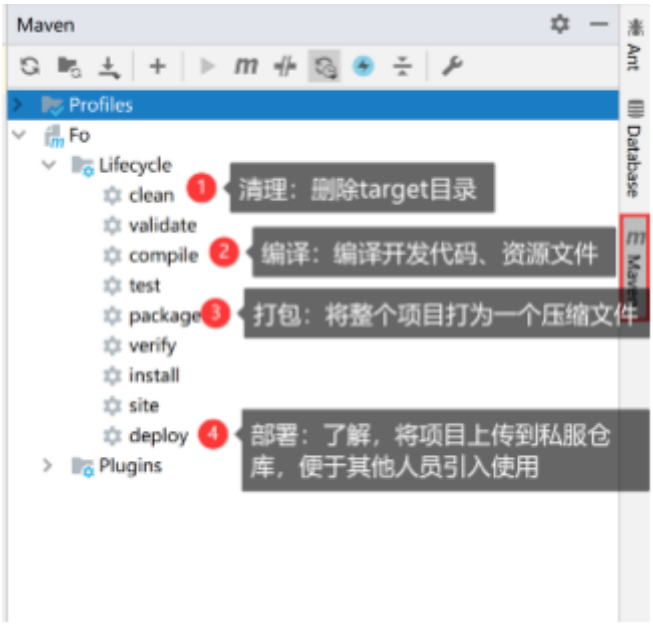
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Fo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.fusesource.jansi</groupId>
            <artifactId>jansi</artifactId>
            <version>2.3.3</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.49</version>
        </dependency>
    </dependencies>

</project>
```

注意

在IDEA中修改了 `pom.xml` 后，记得在Maven面板中刷新才能生效：

Maven的生命周期命令

Maven构建工具主要目的就是对Maven项目进行管理，这些都是通过Maven的生命周期命令来管理的。在IDEA创建的Maven项目中，打开Maven面板，展开项目的Lifecycle即为生命周期命令：



对于以上命令，大家可以双击运行并观察执行结果。

注意：一定要检查控制台输出日志，出现 `BUILD SUCCESS` 才算执行成果：

```
[INFO] Building Fo 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ Fo ---
[INFO] Deleting E:\Workspace\IDEA\Fo\target
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  0.192 s
[INFO] Finished at: 2021-07-19T18:28:33+08:00
[INFO] ------------------------------------------------------------------
```

软件运行时的依赖资源

软件在运行时，需要以下资源（注意区分软件的开发目录和运行目录）：

运行时环境

对于Java程序来说，JRE就是必须的运行时环境。

> PS：JDK是Java程序开发需要的环境，JDK包含了JRE。

程序代码

对于Java程序来说，class文件就是运行时需要的程序代码。

外部资源

在软件外部的任何资源，如果程序代码中会使用，就是运行时必须的外部资源。常见的外部资源如：

- 数据库资源：程序运行时，需要连接某个数据库服务器，并调用服务器提供的服务资源。

  > 数据库包括：
  >
  > 1. 关系型数据库（RDBMS数据库）如MySQL，Oracle，SQL Server，DB2等。
  > 2. 非关系型数据库（NoSQL数据库）如Redis，MongoDB，HBase等。
  >
  > 以上数据库都可以提供服务，让其他程序调用。

- 文件资源：程序代码中可能会使用外部的本地文件/文件夹。如服务端在代码中读取某个本地文件内容并返回给客户端。
- 其他：还有很多类型外部资源，如程序中依赖其他系统提供的URL资源，依赖MQ消息中间件等等。

内部资源

对于Java程序来说，在程序内部（运行时目录），除class文件外的其他文件，如果程序运行时会使用，都属于内部资源。常见的内部资源如：

- 配置文件：专门用于配置某些信息，以便于程序运行时加载使用的文件，常见的配置文件格式有xml，properties等。

  > 有些信息虽然可以直接在程序代码中保存，但会存在灵活性，维护性等等问题。在不同的环境需要灵活调整的信息，在设计上，一般会考虑保存在配置文件中，程序从配置文件读取：
  >
  > - 例如数据库URL，账号，密码等信息，不同开发人员可能需要连接本机数据库，或开发环境数据库；有时候可能需要连接测试数据库调试问题；项目部署到生产环境时，又要调整为连接生产数据库。
  > - 例如引用外部本地文件资源的路径，不同开发人员在本机运行程序时，需要调整为本地某个路径；程序在开发环境，测试环境，生产环境运行时，又需要调整为当前环境主机的本地路径。
  > - 如果以上信息保存在程序中，那就得重新编译，步骤繁琐，且容易出错。
  >
  > PS（了解）：更进一步，配置信息也不一定必须保存在程序内的配置文件中，还可以保存在其他地方，如数据库，其他系统URL资源。

- 其他文件：除配置文件之外的其他文件，如图片，视频，文本文件，Web课程会学习的HTML，JS文件等。

日志

生产环境中，如果没有日志文件，那几乎很难定位到程序出现的问题。

> Java中一般使用日志框架，可以输出到控制台，也可以输出到日志文件。此时不再使用`System.out.print`打印。
>
> 日志框架一般提供调试信息，普通信息，警告信息，错误信息等，便于不同模式下打印不同的信息。
>
> 还可以使用不同的日志文件，按多种维度来记录：如按日期，按业务，按分层（不同的Java包）等等。以便于更快速的查找到日志信息。

示例

待定

IDEA中的软件环境

idea编辑器集成各个插件，本质还是执行相关指令

javac指令的集成

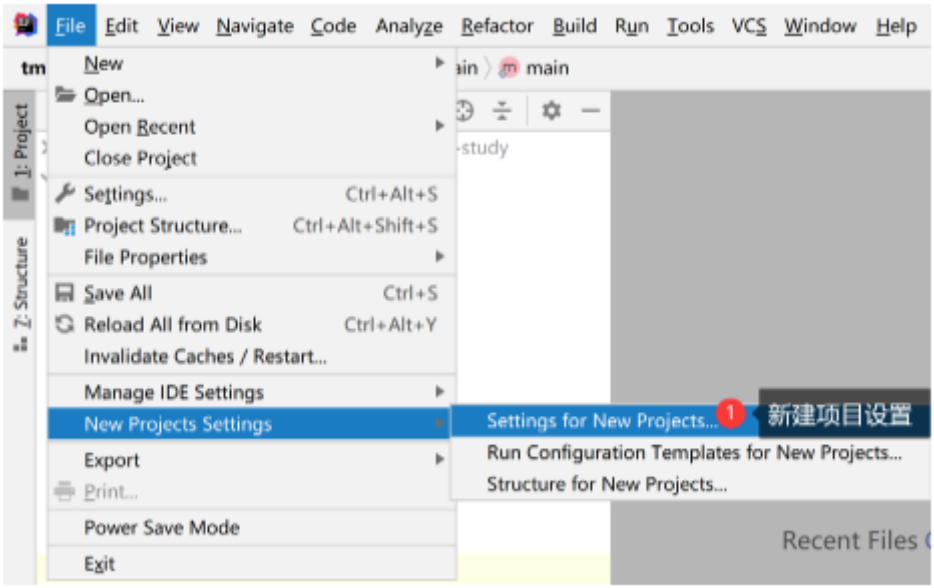
java指令的集成

maven集成

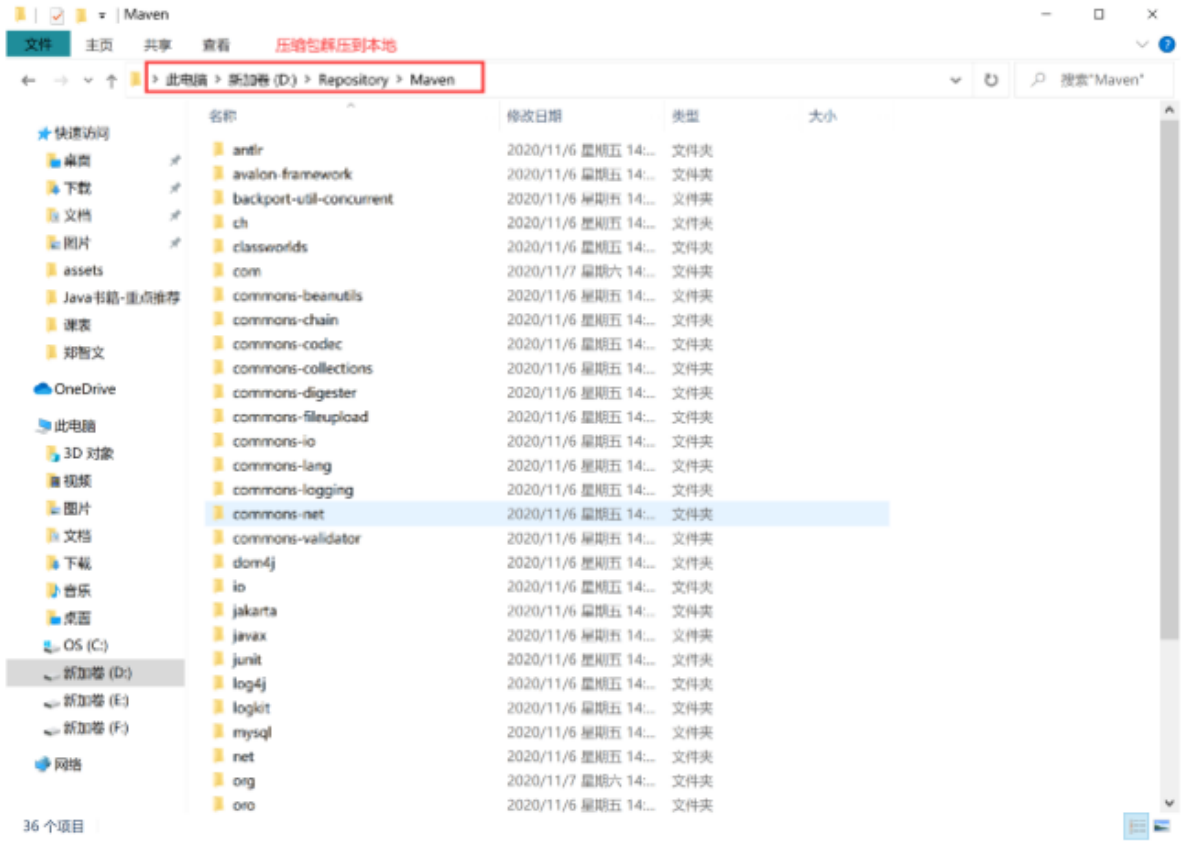
附录

IDEA配置Maven本地仓库

首先，随便创建一个普通的 Java 项目，然后打开新建项目设置（该设置对当前项目不生效，对新建的项目生效，配置好以后，新建的Maven项目就会沿用该设置）：



下载Repository.zip，确保存放在本地某个路径中，之后别移动了。并解压压缩包，解压后是这样的：



IDEA配置Maven