演示Maven命令：

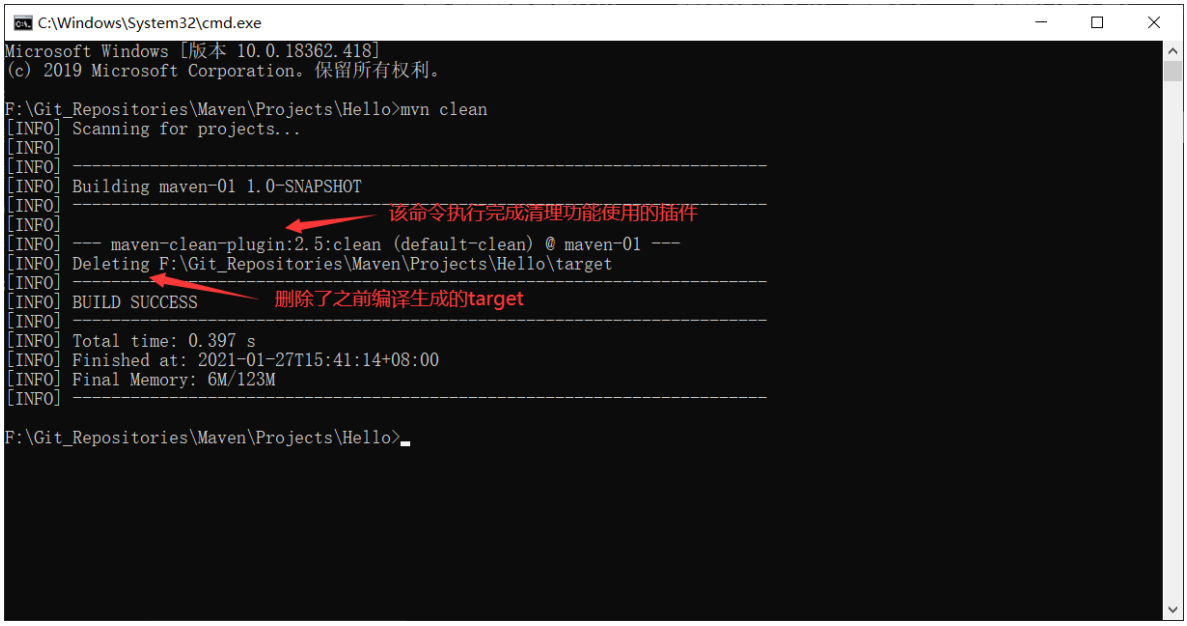
```
mvn clean：清理，删除原来编译和测试的目录，target目录。但是已经install到仓库中的包不会被清除
mvn compile：编译主程序（只编译src/main/java文件下的java文件），在当前目录下（项目目录）生成一个target目录，里面存放编译生成的字节码文件
mvn test-compile：编译测试程序（只编译src/test/java文件下的java文件），结果和上面一样
mvn test：测试，生成一个surefire-reports目录，保存测试结果
mvn package：打包主程序，编译、编译测试、测试、按照pom文件中的配置把主程序打包生成jar文件或war包
mvn install：安装主程序，把本工程打包，按照工程的坐标存放保存到本地仓库中
mvn deploy：部署主程序
```

在src/test/java中添加测试代码，测试的是主程序中的HelloMaven，所以编写的测试程序所在包要与HelloMaven中的一致。创建一个com.studymyself.TestHelloMaven类，具体测试代码如下

```
package com.studymyself;


public class TestHelloMaven{

    @test
    public void testAdd(){

        System.out.println("Maven == junit ==执行testAdd()");

        //下面编写的是测试add方法是否正确的代码
        HelloMaven hello = new HelloMaven();

        int res = hello.add(10,20);

        //验证10+20是不是等于30，需要使用junit提供的方法来对比结果
        //junit包中的Assert中的一个方法
        //Assert.assertEquals(期望值，得到的实际值),该方法中如果两个值相等证明正确，不等则抛出异常
        Assert.assertEquals(30,res);

    }

}
```

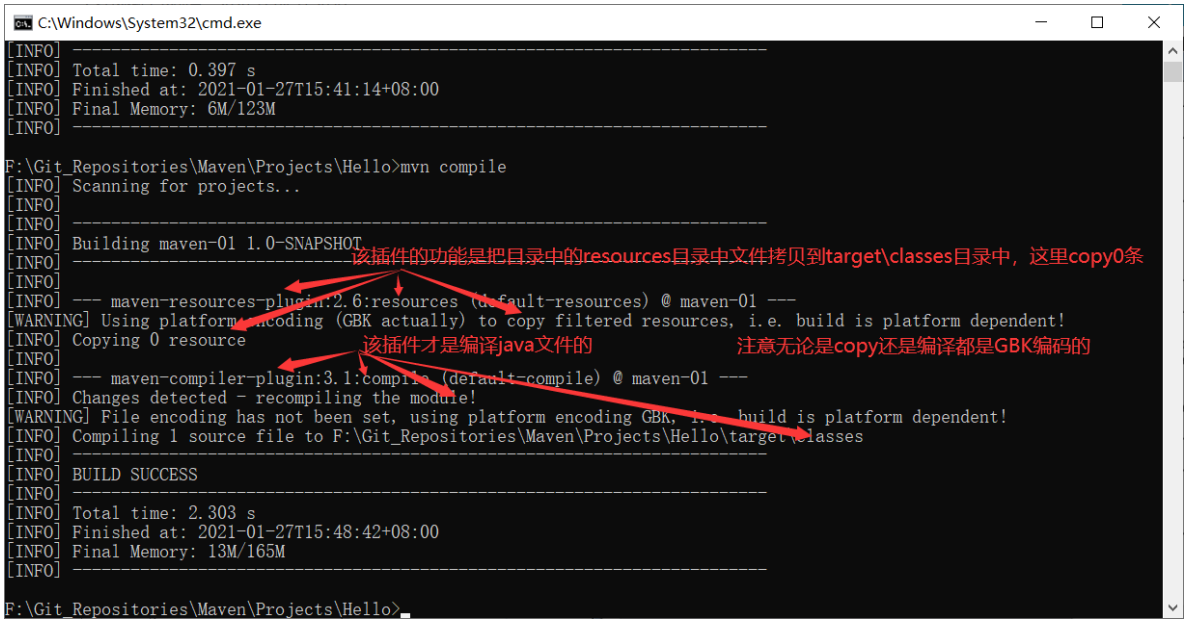
需要注意的是单单使用maven的命令构建工程，需要在有pom文件的目录下执行maven命令。

第一步：mvn clean



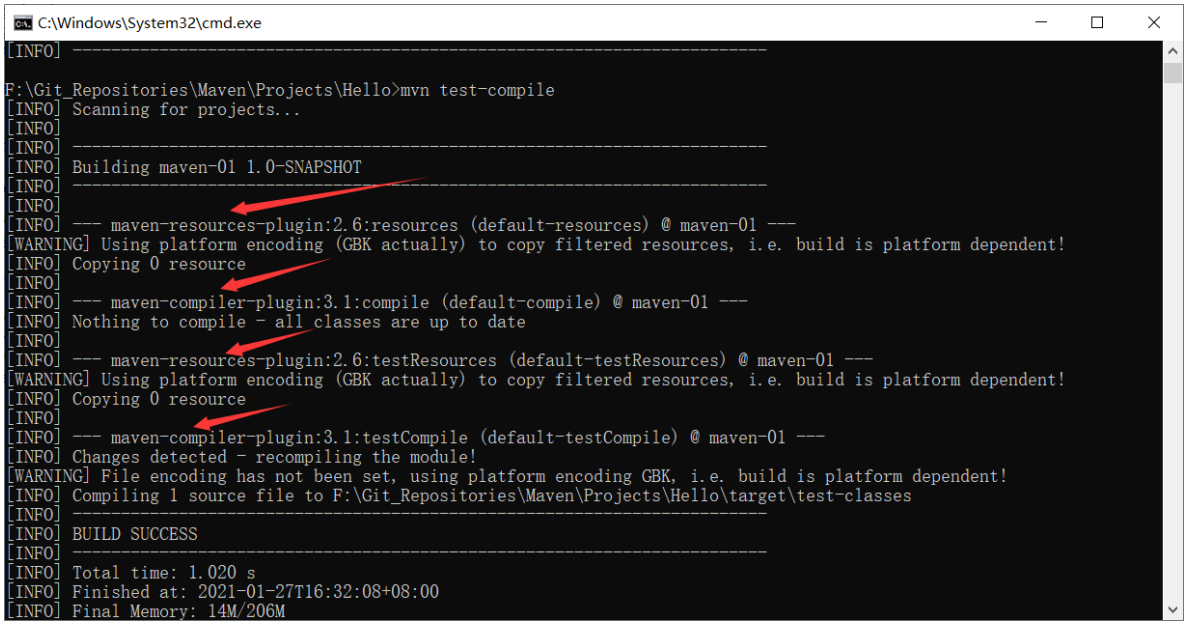
可以看到上图中maven使用清理功能插件，删除了先前编译生成的target

第二步、mvn compile（编译主程序Java文件）



从上图可以知道该命令需要用到两个插件完成功能，一个是编译生成target目录存放class文件的，另一个是将程序所需资源resources拷贝到target/classes目录下

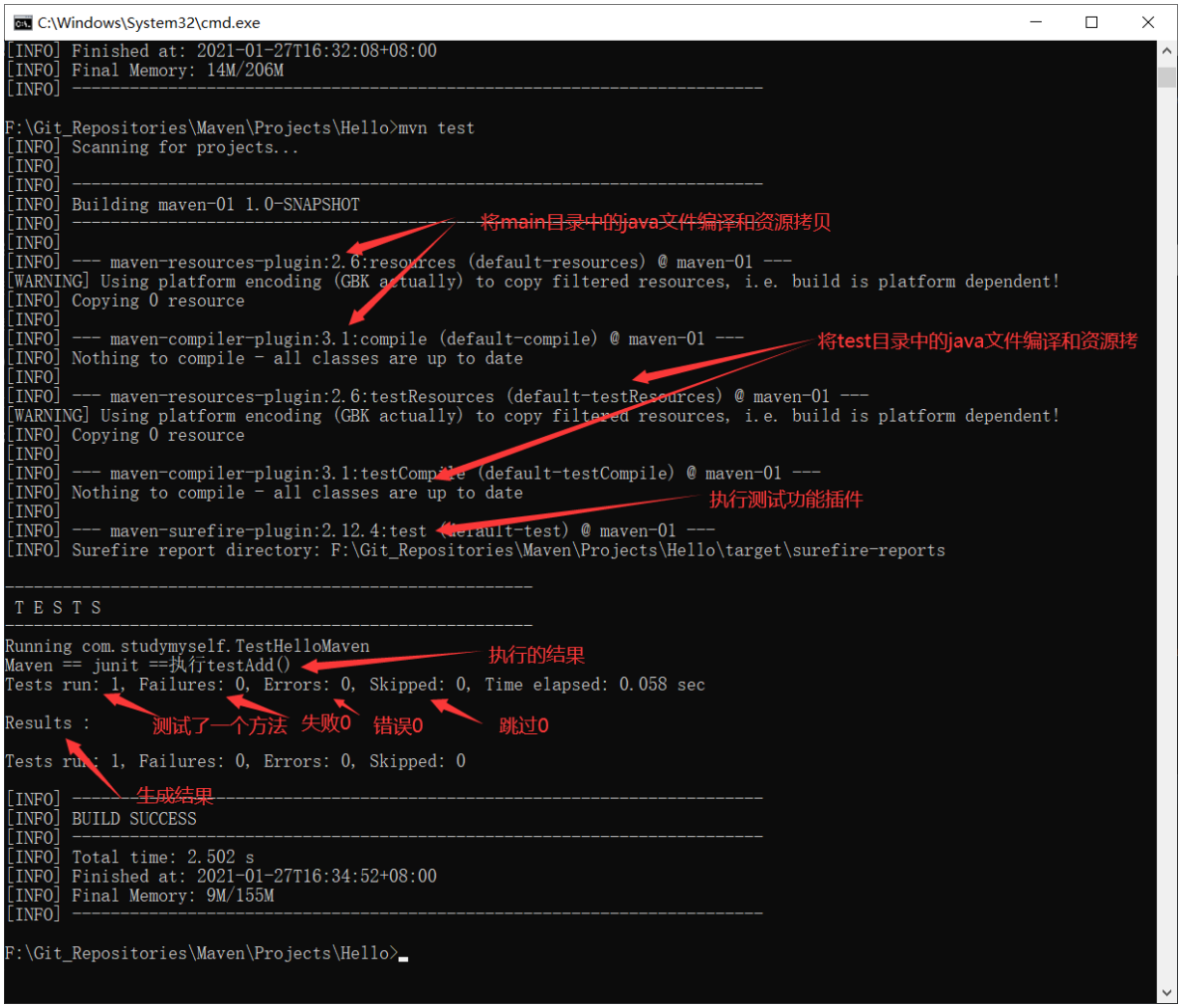
第三步、mvn test-compile（编译测试程序java文件）



可以看到里面有四个插件，由于maven的生命周期在执行某个阶段时会将前面的阶段都执行一次，即前面两个插件是

mvn compile所做的那些事

第四步、mvn test（测试）



从上面我们可以知道，执行mvn test命令，maven把前面所有的构建阶段都执行了一遍。所以我们在测试程序再添加一个测试方法时，不再需要执行前面清理编译等命令了。

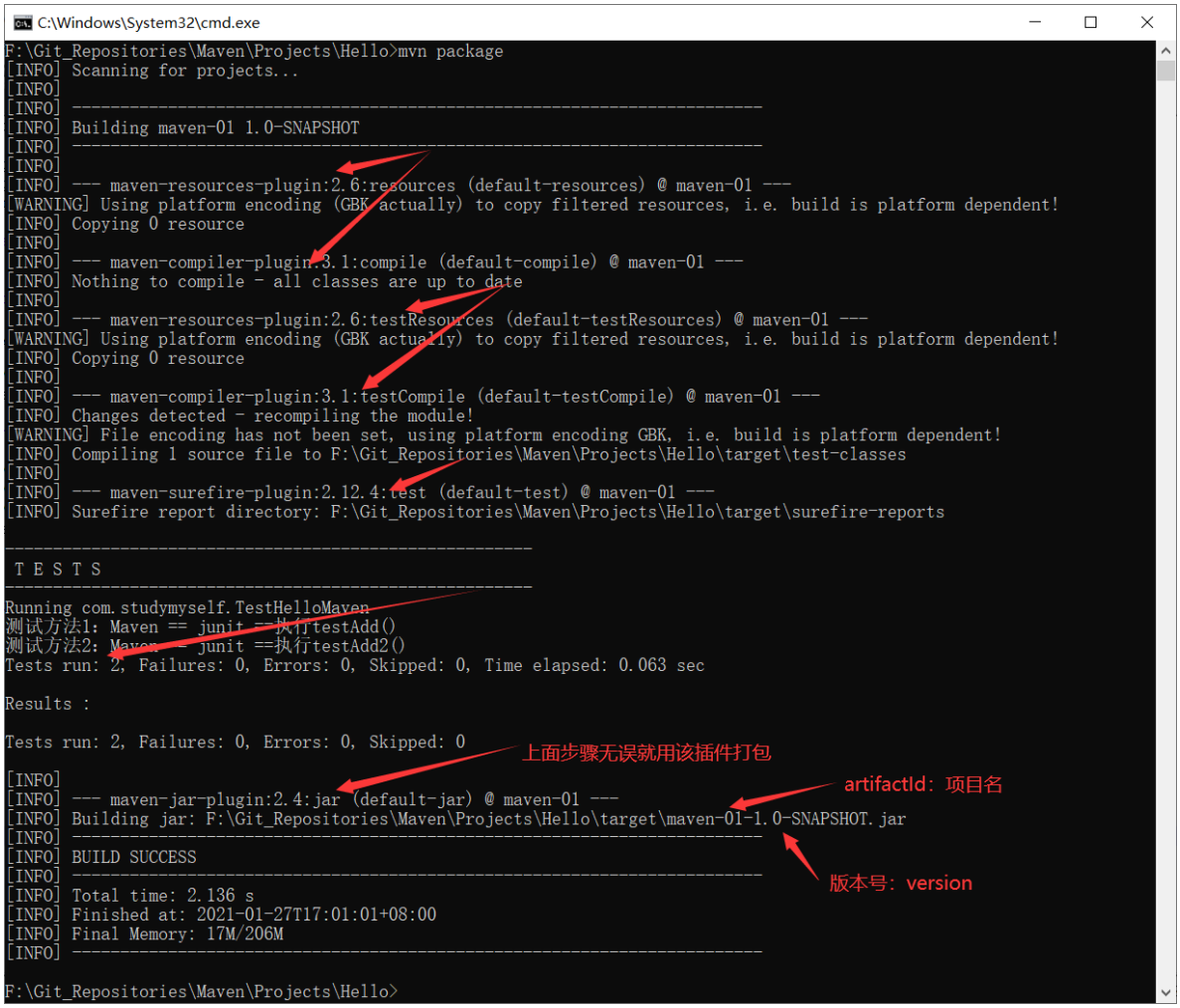
```
    package com.studymyself;
```

```
import org.junit.Assert;
import org.junit.Test;
import com.studymyself.HelloMaven;

public class TestHelloMaven{

    @Test
    public void testAdd(){

        System.out.println("测试方法1：Maven == junit ==执行testAdd()");

        //下面编写的是测试add方法是否正确的代码
        HelloMaven hello = new HelloMaven();

        int res = hello.add(10,20);

        //验证10+20是不是等于30，需要使用junit提供的方法来对比结果
        //junit包中的Assert中的一个方法
        //Assert.assertEquals(期望值，得到的实际值),该方法中如果两个值相等证明正确，不等则
抛出异常
        Assert.assertEquals(30,res);

    }
    @Test
        public void testAdd2(){

            System.out.println("测试方法2：Maven == junit ==执行testAdd2()");

            //下面编写的是测试add方法是否正确的代码
            HelloMaven hello = new HelloMaven();

            int res = hello.add(10,20);

            //验证10+20是不是等于30，需要使用junit提供的方法来对比结果
            //junit包中的Assert中的一个方法
            //Assert.assertEquals(期望值，得到的实际值),该方法中如果两个值相等证明正确，
不等则抛出异常
            Assert.assertEquals(50,res);

        }

}
```

```
再次执行测试命令，出现有测试失败的方法，并报错抛出异常。存放结果位置是
[ERROR] Please refer to
F:\Git_Repositories\Maven\Projects\Hello\target\surefire-reports for the
individual test results.
```

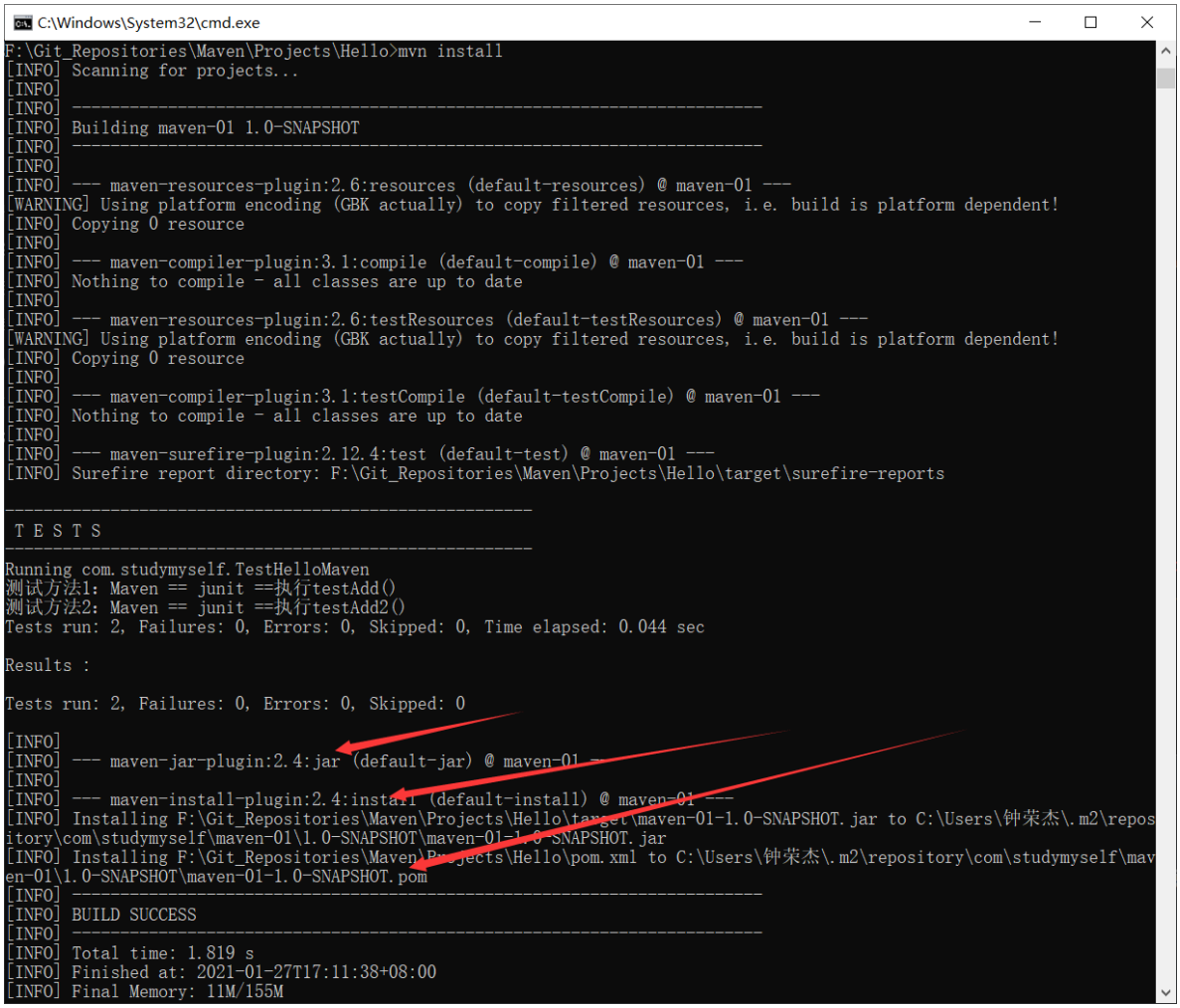
第五步、mvn package（打包）

注意：当我们的测试程序中有测试方法失败时，是无法打包的，执行该命令会把之前的构建过程都执行一次，测试阶段有失败的话，这个项目肯定没有完成，所以打包会失败



打包文件只包含src/main目录下的class文件和资源文件以及其他文件

第六步、mvn install（将jar包安装到本地仓库，其他项目需要用就可以直接添加其依赖就行了）



执行完前面的阶段，可以看到将打包好的jar包存放到了本地仓库repository的com\studymyself\maven-01\1.0-SNAPSHOT目录中。怎么存，就是由pom文件中的项目坐标确定的。

有时候我们需要在pom文件中配置编译插件，用build属性标签，如下:

以后想配置什么插件就直接到网上搜索，就能得到，直接复制粘贴不需要自己写

```
<?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>

    <!--上面的东西都是固定的-->

    <!--下面三个称之为坐标，三个值来指定项目的名称的唯一标识-->

    <!-- 公司或者组织的唯一标志，并且配置时生成的路径也是由此生成，这里先简写-->
    <groupId>com.studymyself</groupId>
    <!--项目名字-->
    <artifactId>maven-01</artifactId>
    <!--项目的版本号-->
    <version>1.0-SNAPSHOT</version>
```

```
    <dependencies>

    <!--测试单元-->
    <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <!--依赖作用范围-->
    <scope>test</scope>
    </dependency>
    </dependencies>

    <build>
        <!--配置插件标签-->
        <plugins>
            <!--具体插件配置标签-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <!--插件名称-->
                <artifactId>maven-compiler-plugin</artifactId>
                <!--插件版本-->
                <version>3.8.1</version>
                <!--配置插件的信息-->
                <configuration>
                    <!--告诉maven，项目代码要在jdk1.8上编译-->
                    <source>1.8</source>
                    <!--告诉maven，项目代码要在jdk1.8上运行-->
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>


</project>
```