

# 实验一：白盒测试工具Junit

---

71119103 许润

## 实验一：白盒测试工具Junit

- 一、安装maven和junit
  - 1、安装maven
  - 2、新建maven项目
  - 3、导入Junit的jar包
- 二、完成课件基本实验内容
  - 1、基本使用
  - 2、Junit套件测试
  - 3、Junit时间测试
  - 4、Junit异常测试
  - 5、Junit参数化测试
- 三、使用Junit的6个注解

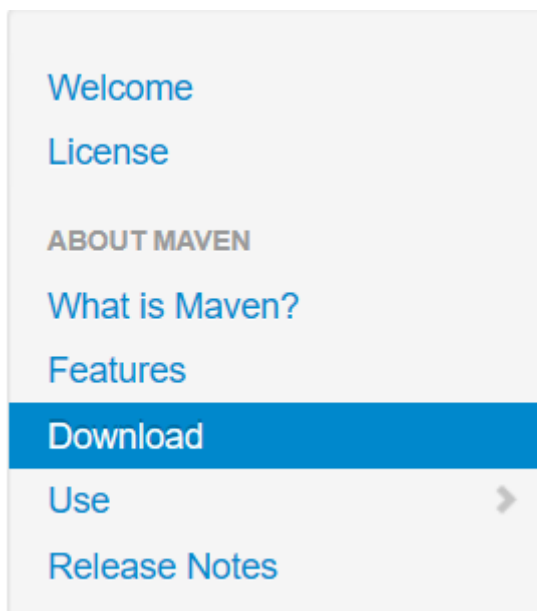
## 一、安装maven和junit

---

### 1、安装maven

①点此进入[maven官网](#)下载

②选择左侧的 Download



③点击链接进行下载

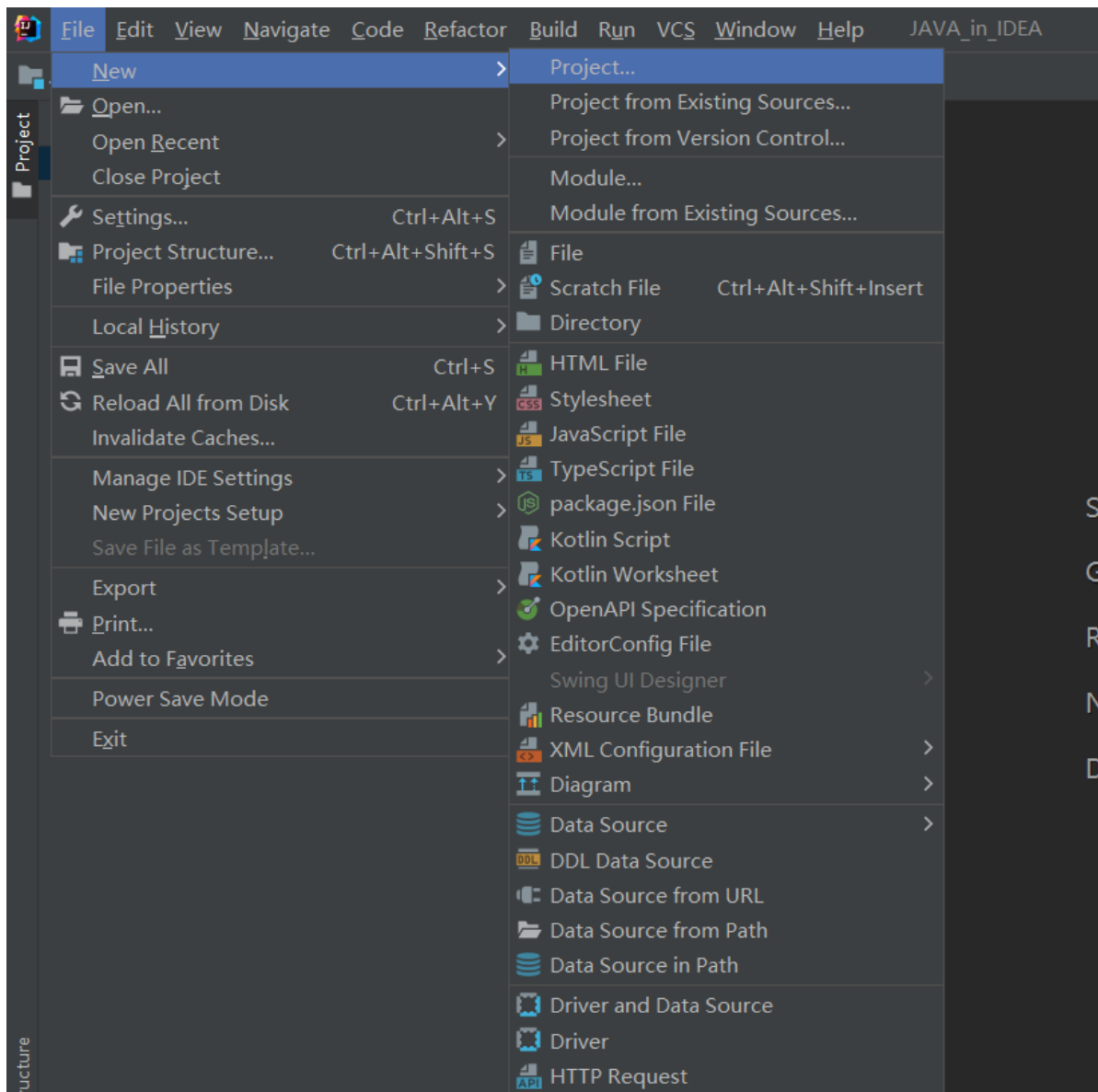
	Link	Checksums	Signature
Binary tar.gz archive	<a href="#">apache-maven-3.8.5-bin.tar.gz</a>	<a href="#">apache-maven-3.8.5-bin.tar.gz.sha512</a>	<a href="#">apache-maven-3.8.5-bin.tar.gz.asc</a>
Binary zip archive	<a href="#">apache-maven-3.8.5-bin.zip</a>	<a href="#">apache-maven-3.8.5-bin.zip.sha512</a>	<a href="#">apache-maven-3.8.5-bin.zip.asc</a>
Source tar.gz archive	<a href="#">apache-maven-3.8.5-src.tar.gz</a>	<a href="#">apache-maven-3.8.5-src.tar.gz.sha512</a>	<a href="#">apache-maven-3.8.5-src.tar.gz.asc</a>
Source zip archive	<a href="#">apache-maven-3.8.5-src.zip</a>	<a href="#">apache-maven-3.8.5-src.zip.sha512</a>	<a href="#">apache-maven-3.8.5-src.zip.asc</a>

④打开cmd，输入 `mvn-version` 验证maven是否安装成功

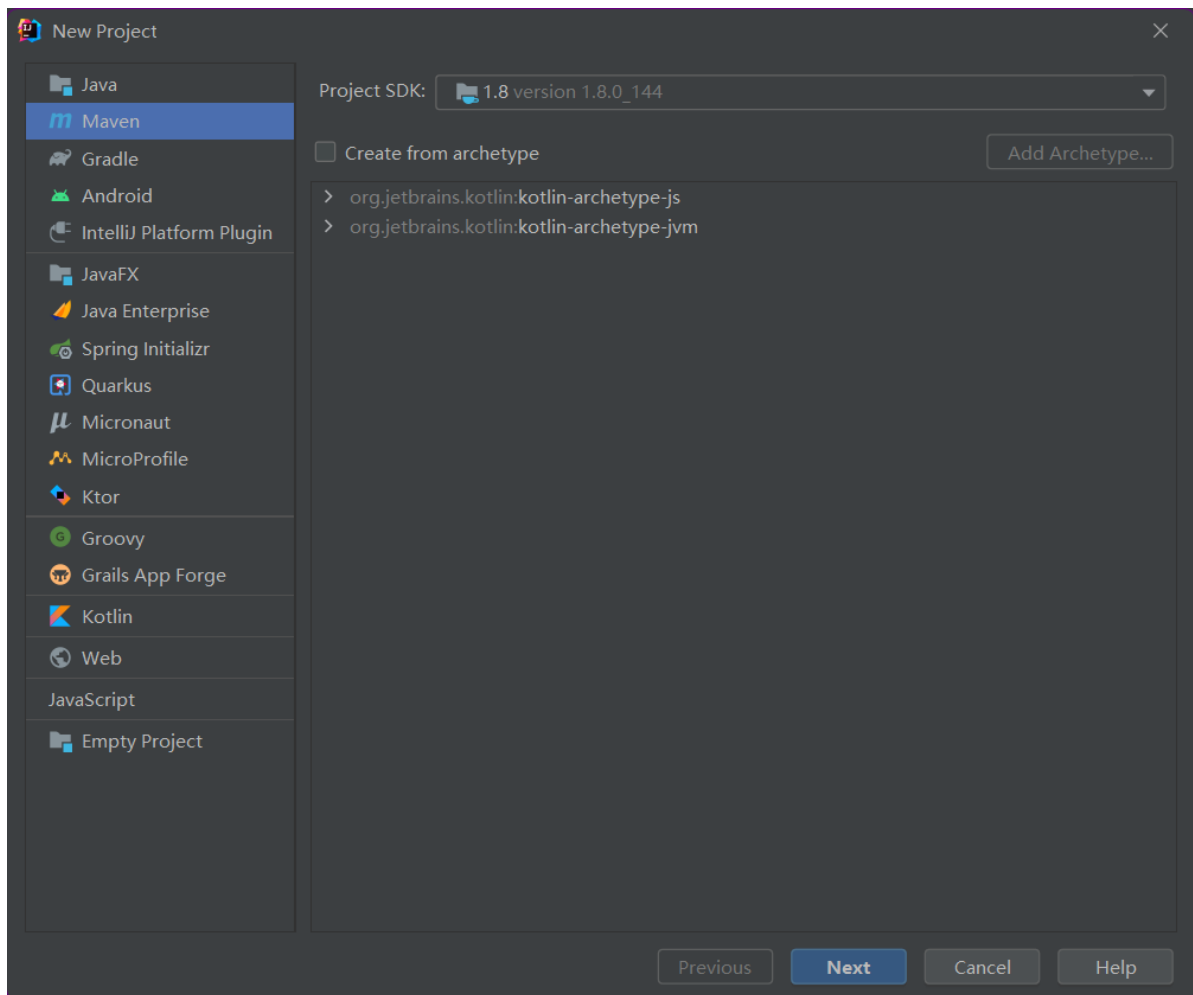
```
C:\Users\asus>mvn -version
Apache Maven 3.6.1 (d66c9c0b3152b2e69ee9bac180bb8fcc8e6af555; 2019-04-05T03:00:29+08:00)
Maven home: E:\Installed software\Maven\maven\apache-maven-3.6.1-bin\apache-maven-3.6.1\bin\..
Java version: 11.0.4, vendor: Oracle Corporation, runtime: E:\Installed software\jdk11
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

## 2、新建maven项目

①点击 `File` —> `New` —> `Project`



②选择 maven，创建名为 junit 的maven项目

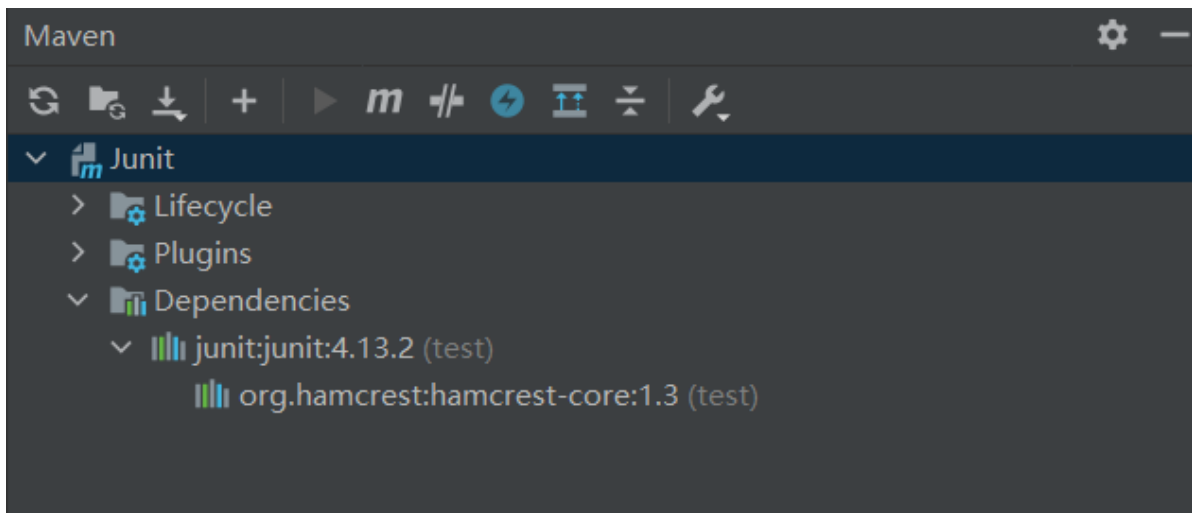


### 3、导入Junit的jar包

- ①、进入[Maven仓库](#)
- ②、选择最新的Junit，复制到 `pom.xml` 中

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/junit/junit -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

- ③刷新，自动将包下载下来



至此，已经成功导入了JUnit!

## 二、完成课件基本实验内容

观察项目目录可以发现，src下有两个文件夹，分别为main和test,其中main->java目录下开发项目，test->java目录下编写测试文件。

### 1、基本使用

我们在main->java目录下创建TestJunit类作为测试用例，test->java目录下编写测试文件TestDemo来对测试用例进行测试，并且用TestRunner来打印输出测试结果文件。

#### TestJunit.class

```
package demo;

public class TestJunit {
    private String message;

    public TestJunit(String message){
        this.message = message;
    }

    public String print(){
        System.out.println(this.message);
        return message;
    }
}
```

#### TestDemo.class

```
import demo.TestJunit;
import org.junit.Test;

import static org.junit.Assert.*;

public class TestDemo {
    private String message = "Hello word!";
    private TestJunit t1 = new TestJunit(message);
}
```

```

@Test
public void test(){
    assertEquals(message,t1.print());
}
}

```

### TestRunner.class

```

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestDemo.class);
        for(Failure failure:result.getFailures()){
            System.out.println(failure.toString());
        }
        System.out.println("测试结果: " + result.wasSuccessful());
    }
}

```

内容相等和修改字符串内容后，使其不相等，分别运行TestRunner.class,两种结果如下：

The screenshot shows the Run console of an IDE. The title bar says 'Run: TestRunner x'. The output text is as follows:

```

"E:\Installed software\JDK\bin\java.exe" ...
Hello Word!
测试结果: true
Process finished with exit code 0

```

The screenshot shows the Run console of an IDE. The title bar says 'Run: TestRunner x'. The output text is as follows:

```

"E:\Installed software\JDK\bin\java.exe" ...
Hello Word!
test(TestDemo): expected:<Hello [Run]!> but was:<Hello [Word]!>
测试结果: false
Process finished with exit code 0

```

## 2、Junit套件测试

测试套件就是捆绑几个单元测试用例并且一起执行它们。在Junit中，@RunWith和@Suite注解用来运行套件测试。

我们在在test->java的目录中创建TestDemo2测试文件，以及TestSuite类来捆绑TestDemo和TestDemo2这两个单元测试用例并且一起执行它们。

### TestDemo2.class

```
import demo.TestJUnit;
import org.junit.Test;
import static org.junit.Assert.*;

public class TestDemo2 {
    private String message = "Hello word!";
    private TestJUnit t1 = new TestJUnit(message);

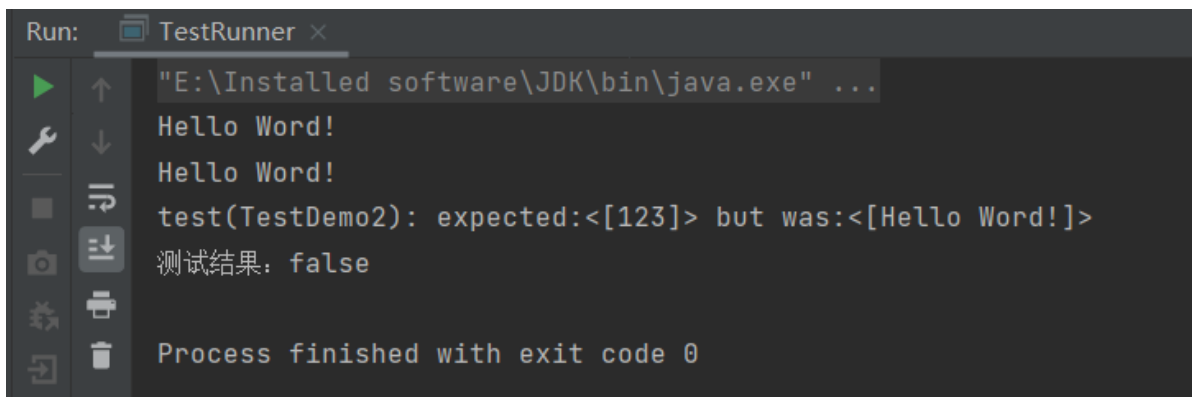
    @Test
    public void test(){
        message = "123";
        assertEquals(message,t1.print());
    }
}
```

### TestSuite.class

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestDemo.class,
    TestDemo2.class,
})
public class TestSuite {
}
```

结果输出如下：可以发现TestDemo和TestDemo两个测试用例皆执行



```
Run: TestRunner x
"E:\Installed software\JDK\bin\java.exe" ...
Hello Word!
Hello Word!
test(TestDemo2): expected:<[123]> but was:<[Hello Word!]>
测试结果: false
Process finished with exit code 0
```

## 3、Junit时间测试

如果一个测试用例比起指定的毫秒数花费了更多的时间，那么Junit将自动将它标记为失败。  
timeout 参数和 @Test 注释一起使用。

本例中我们修改TestDemo类，为其设置一个timeout为2000的@Test注释，并在test()方法中添加一个时间休眠。

### TestDemo.class

```
import demo.TestJUnit;
import org.junit.Test;

import static org.junit.Assert.*;
```

```

public class TestDemo {
    private String message = "Hello word!";
    private TestJunit t1 = new TestJunit(message);

    @Test(timeout = 2000)
    public void test(){
        try{
            Thread.sleep(5000);
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        assertEquals(message,t1.print());
    }
}

```

输出结果如下：

```

Run: TestRunner x
"E:\Installed software\JDK\bin\java.exe" ...
java.lang.InterruptedException Create breakpoint : sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at TestDemo.test(TestDemo.java:13) <12 internal lines>
Hello Word!
Hello Word!
test(TestDemo): test timed out after 2000 milliseconds
test(TestDemo2): expected:<[123]> but was:<[Hello Word!]>
测试结果: false
Process finished with exit code 0

```

可以发现TestDemo报错：test timed out after 2000 milliseconds；在上面的例子中，test() 方法将在5000milliseconds后返回，超出了预定的时间，因此JUnit引擎将其标记为超时。

## 4、Junit异常测试

在写单元测试的时候，经常会遇到需要断言方法需要抛出一个异常这种场景，这时，就会用到JUnit的异常测试功能。此时可以使用@Test注解自带的 expected 属性来断言需要抛出一个异常。在运行测试的时候，此方法必须抛出异常，这个测试才算通过，反之则反。

本例中我们修改TestDemo类，为其设置一个名为ArithmeticException的异常。如果i小于0，那么就抛出一个ArithmeticException。

### TestDemo.java

```

import demo.TestJunit;
import org.junit.Test;

import static org.junit.Assert.*;

public class TestDemo {
    private String message = "Hello word!";
    private TestJunit t1 = new TestJunit(message);

    @Test(expected = ArithmeticException.class)
    public void test(){
        Student s1 = new Student();
    }
}

```



```

        s1.test(-1);
    }
}

class Student{
    public boolean test(int i) {
        if(i<0){
            throw new IllegalArgumentException("我是异常!");
        }else{
            return true;
        }
    }
}

```

输出结果如下：

```

"E:\Installed software\JDK\bin\java.exe" ...
java.lang.Exception: Unexpected exception, expected<java.lang.ArithmeticException> but was<java.lang.IllegalArgumentException>
<18 internal lines>
Caused by: java.lang.IllegalArgumentException Create breakpoint : 我是异常!
    at Student.test(TestDemo.java:20)
    at TestDemo.test(TestDemo.java:13) <9 internal lines>
    ... 17 more

```

可以发现用于测试代码抛出一个ArithmeticException异常。

## 5、Junit参数化测试

参数化测试允许开发人员使用不同的值反复运行同一测试。有以下5个步骤：

- 1、用RunWith(Parameterized.class)来注释test类。
- 2、创建一个由@Parameter注释的公共静态方法，它返回一个对象的集合（数组）来作为测试数据集集合。
- 3、创建一个公共的构造函数，它接受和一行测试数据相等同的东西。
- 4、为每一列测试数据创建一个实例变量。
- 5、用实例变量作为测试数据的来源来创建你的测试用例。

本例中我们创建PrimeNumberChecker类，作为被测试的对象，判读输入的数是否为一个质数。创建测试类TestPrimeNumberChecker，使用@Parameterized.Parameters指定测试参数，并使用@RunWith(Parameterized.class)将测试参数绑定到测试方法中。

### PrimeNumberChecker.class

```

package demo;

public class PrimeNumberChecker {
    public boolean validate(final int num){
        for(int i = 2;i<(num/2);i++){
            if(num%i==0){
                return false;
            }
        }
        return true;
    }
}

```

## TestPrimeNumberChecker.class

```
import demo.PrimeNumberChecker;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import static org.junit.Assert.*;
import java.util.Arrays;
import java.util.Collection;

@RunWith(Parameterized.class)
public class TestPrimeNumberChecker {
    private int inputNum;
    private boolean expectedRes;
    private PrimeNumberChecker primeNumberChecker;

    @Before
    public void initialize(){
        primeNumberChecker = new PrimeNumberChecker();
    }

    public TestPrimeNumberChecker(int inputNum,boolean expectedRes){
        this.inputNum = inputNum;
        this.expectedRes = expectedRes;
    }

    @Parameterized.Parameters
    public static Collection primeNumber(){
        return Arrays.asList(new Object[][]{
            {2,true},
            {6,false},
            {19,true},
            {22,false}
        });
    }

    @Test
    public void testPrimeNumberChecker(){
        System.out.println("Parameterized Number is: " + inputNum);
        assertEquals(expectedRes,primeNumberChecker.validate(inputNum));
    }
}
```

输出结果如下:

```
Run: TestRunner x
"E:\Installed software\JDK\bin\java.exe" ...
Parameterized Number is: 2
Parameterized Number is: 6
Parameterized Number is: 19
Parameterized Number is: 22
测试结果: true
Process finished with exit code 0
```

### 三、使用Junit的6个注解

- `@Test` 指定一个测试案例
- `@Before` 指定该方法需要在@Test方法前运行，该方法会针对每一个测试用例执行，且在测试方法前执行。**注意：**必须是public void，不能为static。不止运行一次，根据用例数而定。
- `@After` 指定该方法需要在@Test方法后运行，该方法会针对每一个测试用例执行，且在测试方法后执行
- `@BeforeClass` 指定该方法需要在类中所有方法前运行，对应的方法会首先执行，且执行一次。当我们运行几个有关联的用例时，可能会在数据准备或其它前期准备中执行一些相同的命令，这个时候为了让代码更清晰，更少冗余，可以将公用的部分提取出来，放在一个方法里，例如创建数据库连接、读取文件等。**注意：**方法名可以任意，但必须是public static void，即公开、静态、无返回。这个方法只会运行一次。
- `@AfterClass` 指定该方法需要在类中所有方法后运行，对应的方法会最后执行，且执行一次。**注意：**同样必须是public static void，即公开、静态、无返回。这个方法只会运行一次。
- `@Ignore` 指定不需要执行测试的方法。有时候我们想暂时不运行某些测试方法\测试类，可以在方法前加上这个注解。在运行结果中，junit会统计忽略的用例数，来提醒你。但是不建议经常这么做，因为这样的坏处时，容易忘记去更新这些测试方法，导致代码不够干净，用例遗漏。

我们来试验一下，我新建一个测试类AnnotationTest，然后每个注解都用了，其中有两个用@Test标记的方法分别是test1和test2，还有一个用@Ignore标记的方法test3。

#### AnnotationTest.class

```
import org.junit.*;
import static org.junit.Assert.*;

/**
 * Created by xurun on 2022/3/26.
 */
public class AnnotationTest {

    @BeforeClass
    public static void setUpBeforeClass() {
        System.out.println("BeforeClass");
    }

    @AfterClass
```

```

    public static void tearDownAfterClass() {
        System.out.println("AfterClass");
    }

    @Before
    public void setUp() {
        System.out.println("Before");
    }

    @After
    public void tearDown() {
        System.out.println("After");
    }

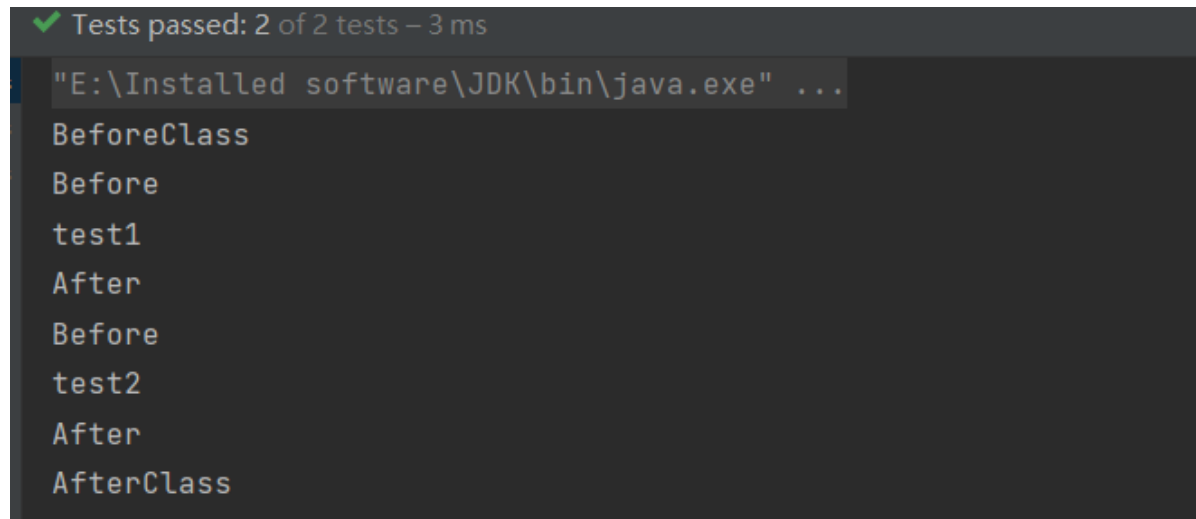
    @Test
    public void test1() {
        System.out.println("test1");
    }

    @Test
    public void test2() {
        System.out.println("test2");
    }

    @Ignore
    public void test3() {
        System.out.println("test3");
    }
}

```

运行结果如下:



```

✓ Tests passed: 2 of 2 tests – 3 ms
"E:\Installed software\JDK\bin\java.exe" ...
BeforeClass
Before
test1
After
Before
test2
After
AfterClass

```

**解释一下:** `@BeforeClass` 和 `@AfterClass` 在类被实例化前就被调用了, 而且只执行一次, 通常用来初始化和关闭资源。 `@Before` 和 `@After` 和在每个 `@Test` 执行前后都会被执行一次, 所以在 `test1` 执行前后各执行一次, 在 `test2` 执行前后各执行一次。被 `@Ignore` 标记的 `test3` 方法没有被执行。

进一步做实验, 添加一个成员变量:

```
int i = 0;
```

然后把test1改为：

```
i++;  
System.out.println("test1的i为" + i);
```

然后把test2改为：

```
i++;  
System.out.println("test2的i为" + i);
```

运行结果如下：

```
✓ Tests passed: 2 of 2 tests – 3 ms  
"E:\Installed software\JDK\bin\java.exe" ...  
BeforeClass  
Before  
test1的i为1  
After  
Before  
test2的i为1  
After  
AfterClass  
  
Process finished with exit code 0
```

可以看到 test1 和 test2 的 i 都只自增了一次，所以 test1 的执行不会影响 test2，因为执行 test2 时又把测试类重新实例化了一遍。如果希望 test2 的执行受 test1 的影响怎么办呢？可以把 int i 改为 static。

最后关于这些注解还有一个要说明的就是，你可以把多个方法标记为@BeforeClass、@AfterClass、@Before、@After。他们都会在相应阶段被执行。