1. 单元测试框架的基本介绍

1.1 什么是单元测试

单元测试(Unit Testing)是指对软件中的程序单元进行检查和验证。单元测试的特点如下：

- 程序单元是最小可测试部件，通常采用基于类的方法进行测试。
- 程序单元和其他单元是相互独立的。
- 单元测试的执行速度很快。
- 单元测试发现的问题，相对容易定位。
- 单元测试通常由开发人员来完成。
- 通过了解代码的实现逻辑进行的测试，通常称为白盒测试。

1.2 JUnit 单元测试框架

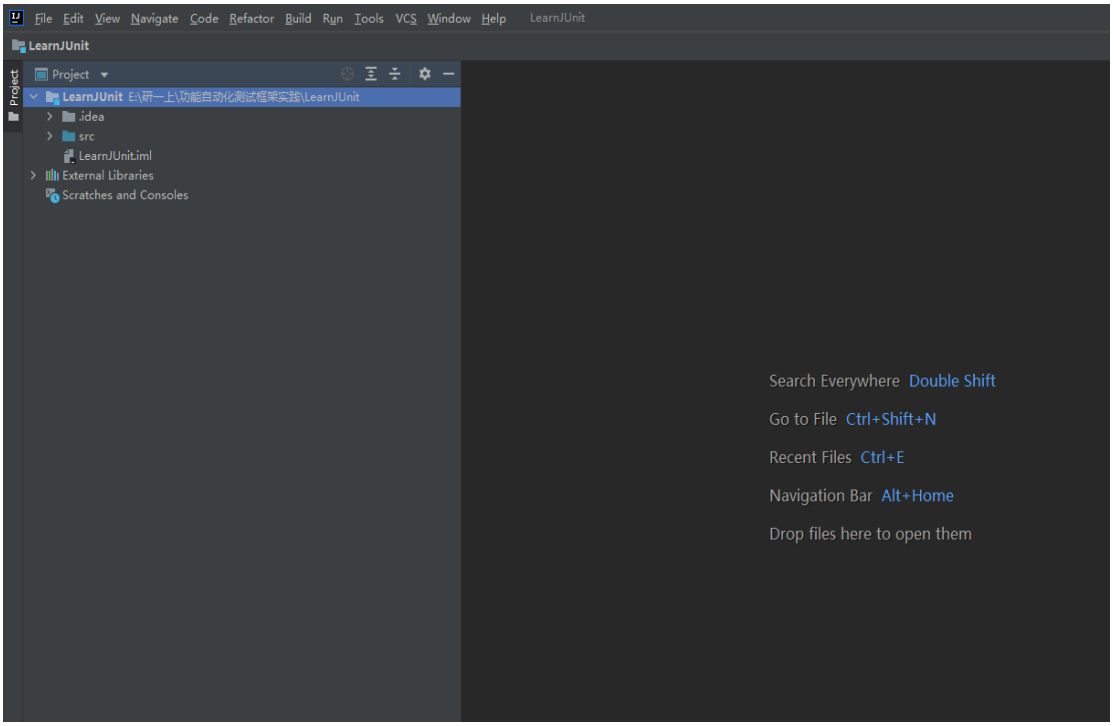
JUnit单元测试框架是基于Java语言的主流单元测试框架，IntelliJ IDEA 已经集成了 JUnit 单元测试框架。JUnit 是一个回归测试框架（Regression Testing Framework），主要用于 Java 语言程序的单元测试，目前使用的主流版本是 JUnit4 及以上版本。

1.3 TestNG 单元测试框架

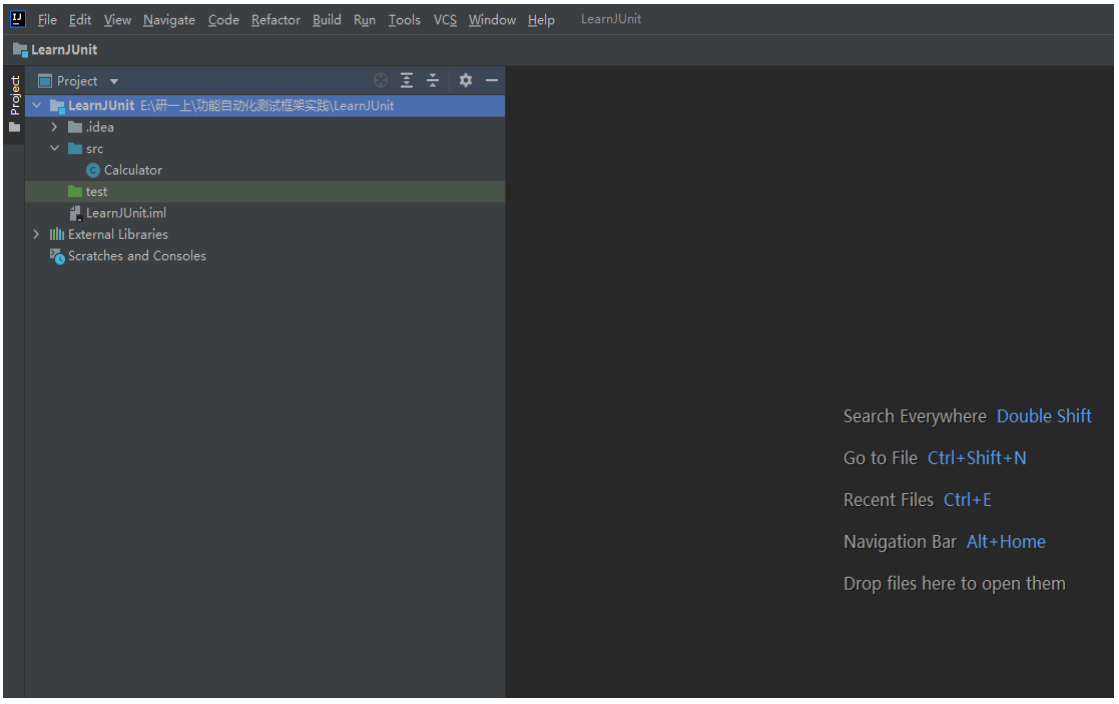
TestNG 单元测试框架比 JUnit 单元测试框架更强大，它提供了更多的扩展功能，消除了一些老式框架的限制，让程序员通过注解、分组、序列化和参数化等各种方式组织和执行自动化测试脚本。具有可提供全面的 HTML 格式测试报告、支持并发测试、参数化测试更简单、支持输出日志、支持更多功能的注解的优点。

2. JUnit 的安装与使用

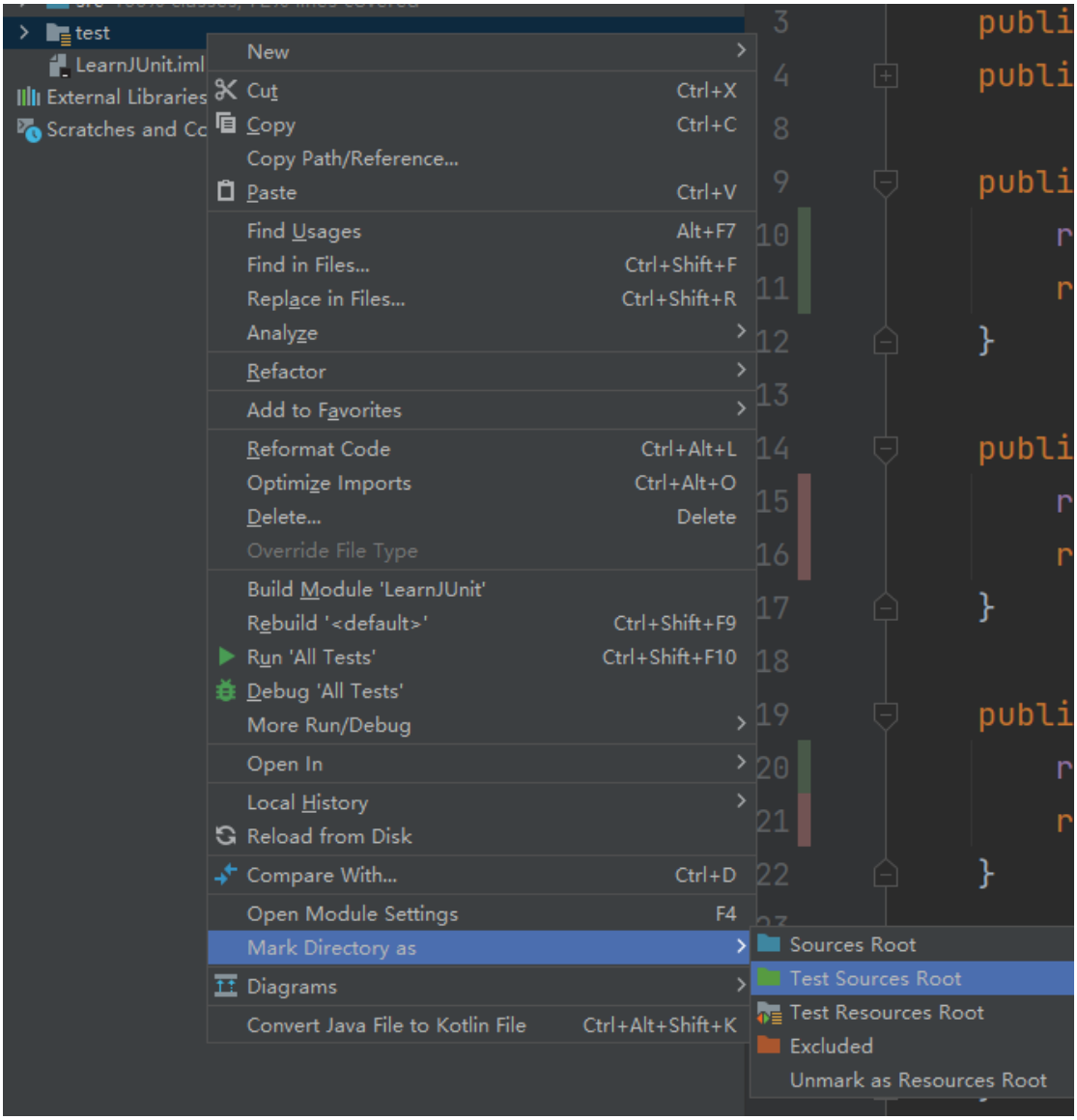
1） 打开 IDEA 创建一个新项目 LearnJUnit



2） 在 src 文件夹下新建 Java 类 Calculator。在 LearnJUnit 文件夹下创建 test 文件夹，并标记 test 文件夹为 Test 类（绿色）。



鼠标右键 test 文件夹，Mark Directory as Test Sources Root，方便后面的测试类直接生成至该目录下。

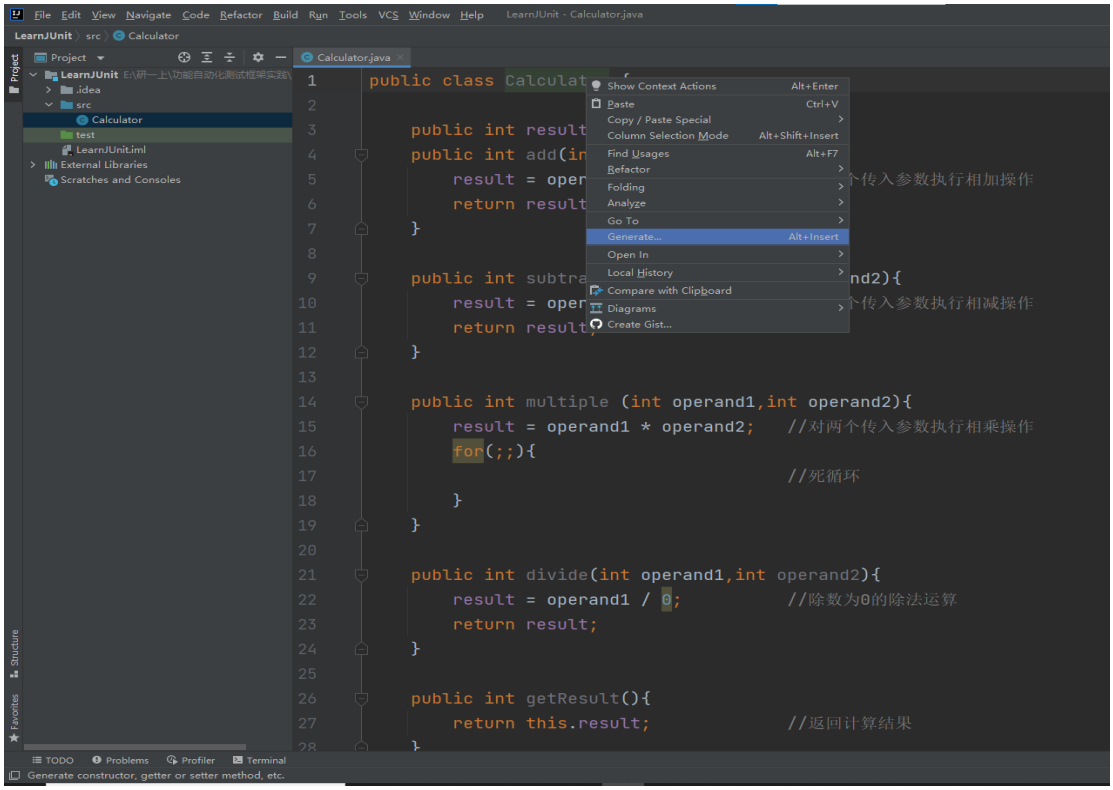


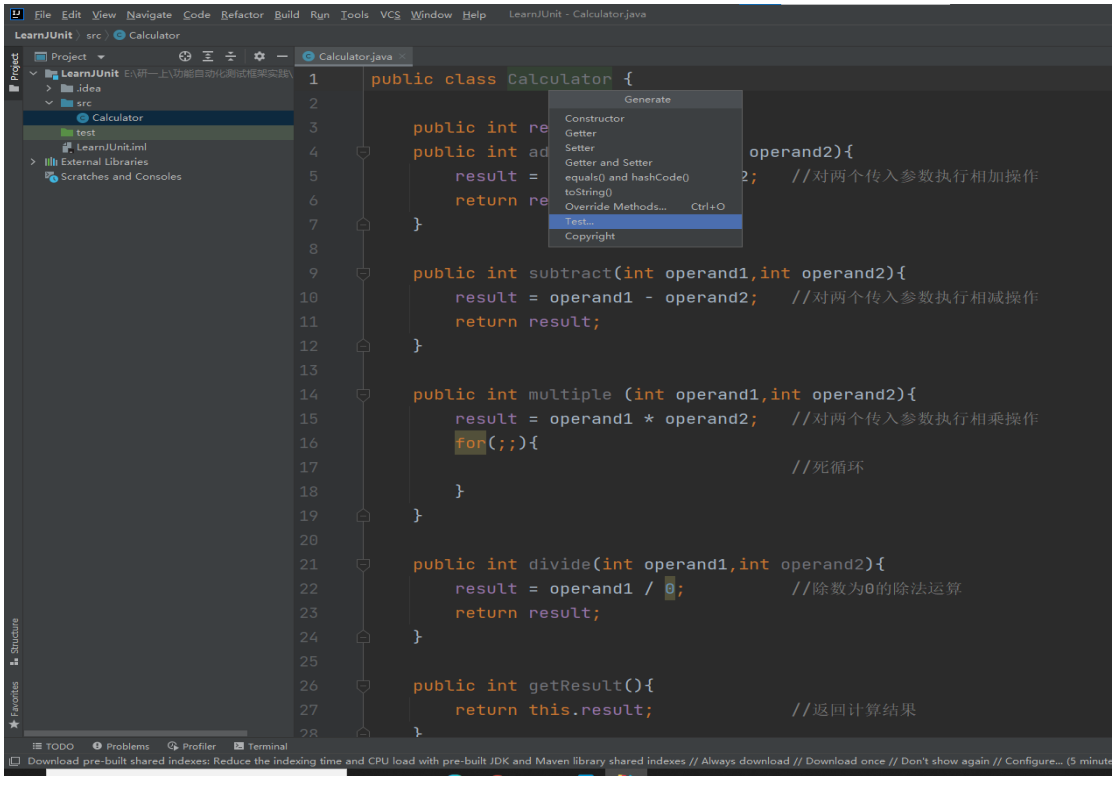
3） 被测试类 Calculator 代码如下

```
public class Calculator {

    public int result = 0;

    public int add(int operand1,int operand2){
        result = operand1 + operand2;        //对两个传入参数执行相加操作
        return result;
    }

    public int subtract(int operand1,int operand2){
        result = operand1 - operand2;        //对两个传入参数执行相减操作
        return result;
    }

    public int multiple (int operand1,int operand2){
        result = operand1 * operand2;        //对两个传入参数执行相乘操作
        return result;
    }

    public int divide(int operand1,int operand2){
        result = operand1 / operand2;        //对两个传入参数执行相除操作
        return result;
    }

    public int getResult(){
        return this.result;                  //返回计算结果
    }

}
```

4） 创建 JUnit4 的测试代码

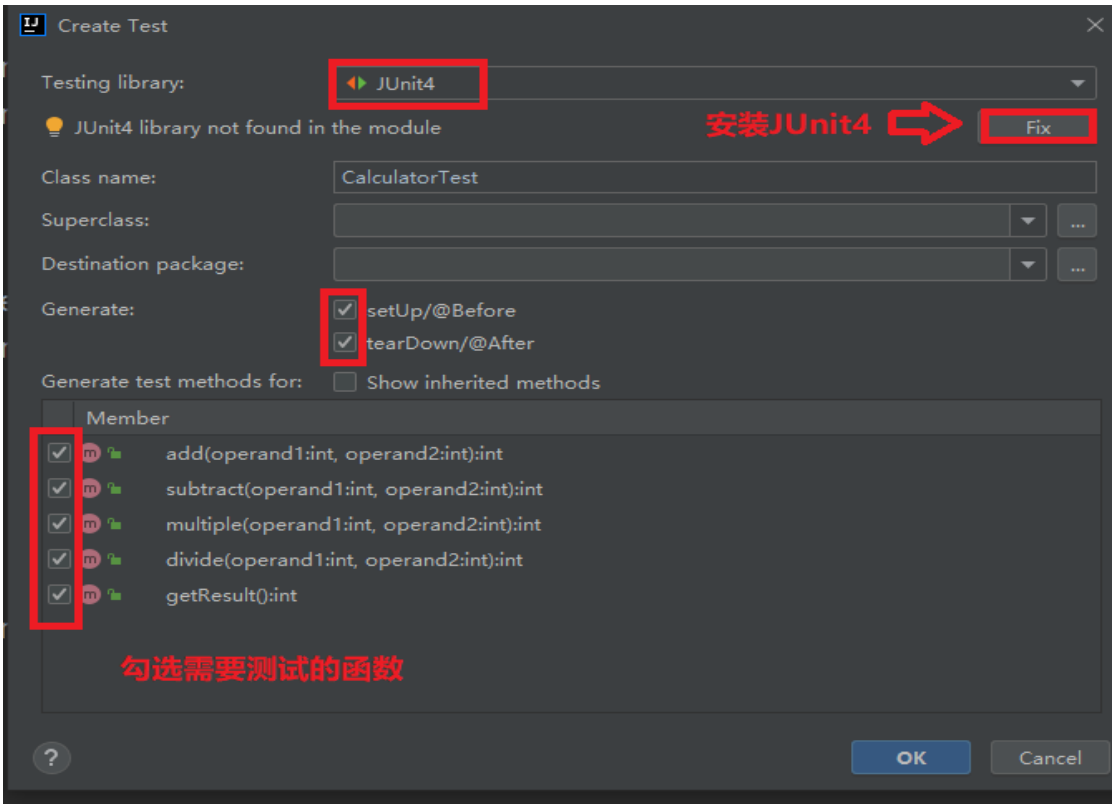
在 Calculator 类下，单击鼠标右键，在弹出的快捷菜单中选择 Generate... -> Test…



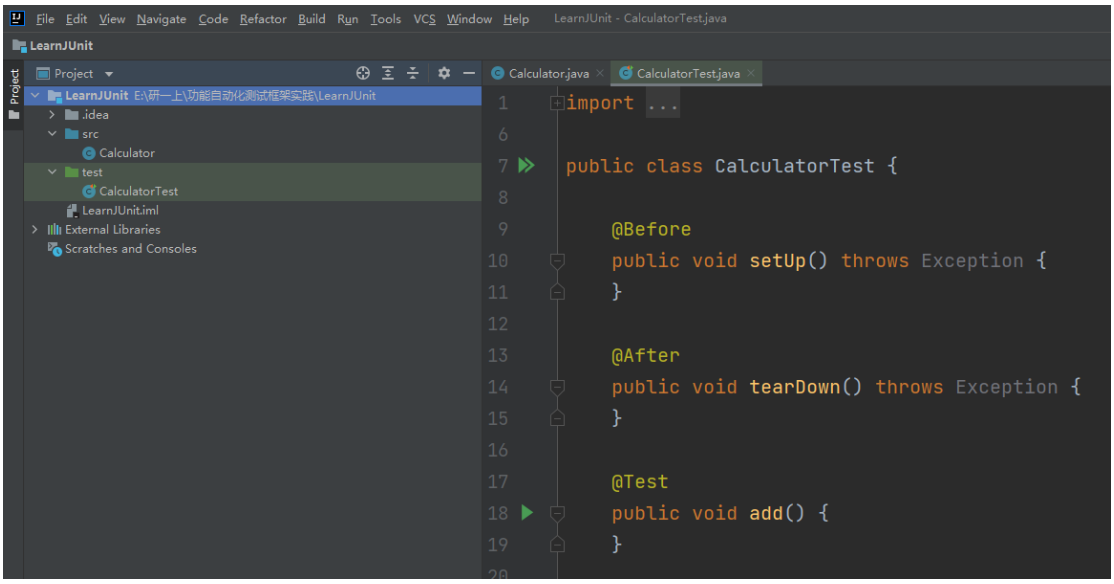


在弹出的 Create Test 框中，选 Testing library 为 JUnit4，若未导入过 JUnit4 库会出现提示语句，点击 Fix 点击 Ok 即可自动导入 JUnit4 库。可勾选上 Generate 后的 setUp/@Before 和 tearDown/@After，再

在下方勾选上需要测试哪些函数，如下图所示。



点击 OK 之后在 test 文件夹下自动生成了 CalculatorTest 类模板代码



5） 上述完成了 JUnit4 测试用例模板的创建工作，针对 Calculator 类的内部实现逻辑，在此模板的基础上编写 JUnit4 单元测试代码。

说明：

测试类中每个方法上方均含由一个@字符的关键字描述，此关键字为 JUnit4 新增的注解（Annotation）功能，以下为常见的注解及其含义。

	注解含义
@BeforeClass	表示使用此注解的方法在测试类被调用之前执行，在一个测试类只能声明此注解一次，此注解对应的方法只能执行一次。
@AfterClass	表示使用此注解的方法在测试类被调用结束退出之前执行，在一个测试类只能声明此注解一次，此注解对应的方法只能执行一次。
@Before	表示使用此注解的方法在每个@Test 调用之前执行，即一个类中有多少个@Test 注解方法，@Before 注解方法就会被调用多少次。
@After	表示使用此注解的方法在每个@Test 调用之后执行，即一个类中有多少个@Test 注解方法，@After 注解方法就会被调用多少次。
@Test	表示使用此注解的方法为一个单元测试用例，在一个测试类中可以多次声明此注解，每个注解为@Test 的方法只执行一次。
@Ignore	表示使用此注解的方法为暂时不执行的测试用例方法，会被 JUnit4 忽略执行。

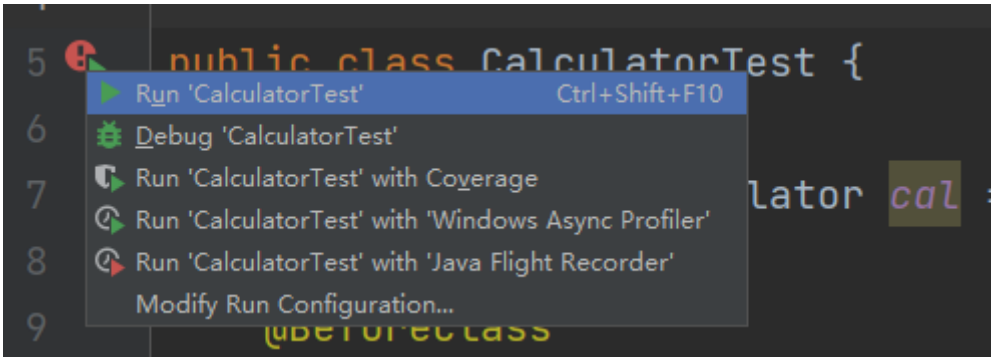
CalculatorTest.java

```java
import org.junit.*;

import static org.junit.Assert.*;

public class CalculatorTest {

    private static Calculator cal = new Calculator();

    @BeforeClass
    public static void setupBeforClass() throws Exception{
        System.out.println("@BeforeClass");
    }

    @AfterClass
    public static void teardownAfterClass() throws Exception{
        System.out.println("@AfterClass");
    }

    @Before
    public void setUp() throws Exception {
        System.out.println("测试开始");
    }

    @After
    public void tearDown() throws Exception {
        System.out.println("测试结束");
    }

```

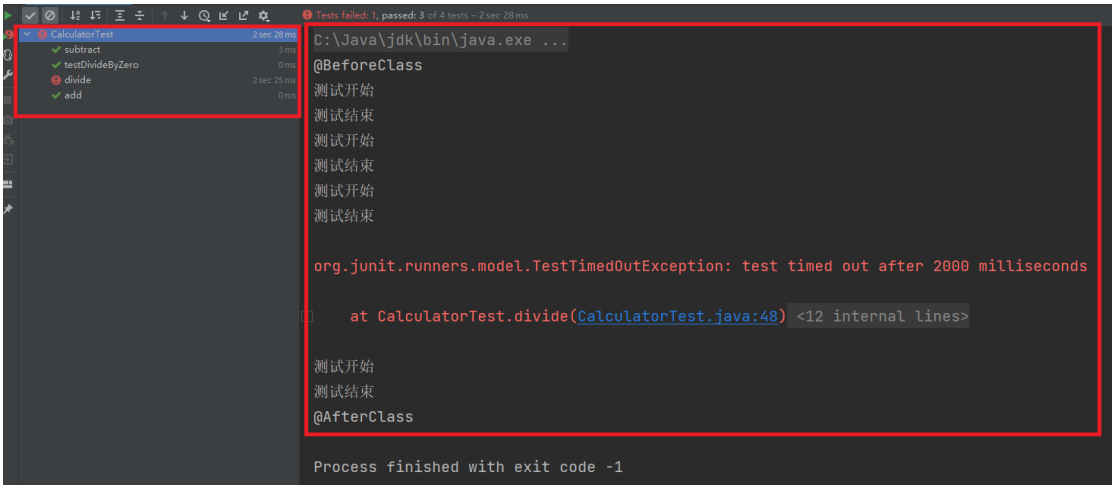
CalculatorTest.java 完整代码如下：

```
import org.junit.*;

import static org.junit.Assert.*;

public class CalculatorTest {

    private static Calculator cal = new Calculator();

    @BeforeClass
    public static void setupBeforClass() throws Exception{
        System.out.println("@BeforeClass");
    }

    @AfterClass
    public static void teardownAfterClass() throws Exception{
        System.out.println("@AfterClass");
    }

    @Before
    public void setUp() throws Exception {
        System.out.println("测试开始");
    }

    @After
    public void tearDown() throws Exception {
        System.out.println("测试结束");
    }

    @Test
    public void add() {
        cal.add(2,2);
        assertEquals(4,cal.getResult());
    }

    @Test
    public void subtract() {
        cal.subtract(4,2);
        assertEquals(2,cal.getResult());
    }

    @Ignore
    public void multiple() {
        fail("Not yet implemented");
    }

    @Test(timeout = 2000)
    public void divide() {
        for(;;);
    }

    @Test(expected = ArithmeticException.class)
    public void testDivideByZero(){
        cal.divide(4,0);
    }
}
```

此时运行测试代码



运行结果如下：



在 Console 界面的输出结果中，可看出 setUpBeforeClass()和 tearDownAfterClass()方法在整个测试类的运行过程中只执行了一次，SetUp()和 tearDown()方法在每次@Test 方法执行之前均被调用，执行了多次。

```
@Test(timeout = 2000)
public void divide() {
    for(;;);
}
```

上例代码中的表达式“timeout=2000”表示此测试用例的执行时间不能超过 2000 毫秒。由于方法体中的实现代码为死循环，所以此方法的执行时间肯定超过了 2 秒，导致此测试用例执行失败。

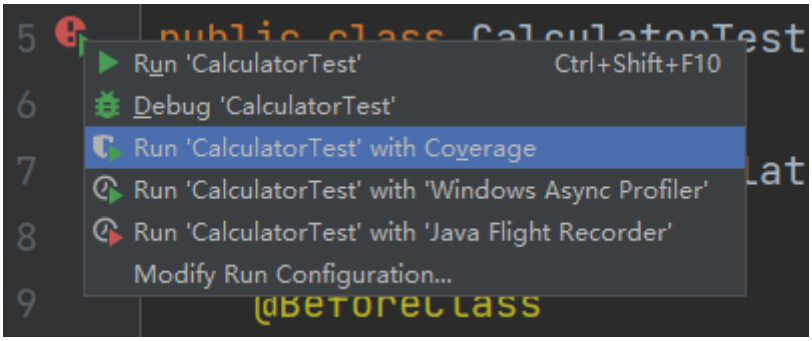
```
@Test(expected = ArithmeticException.class)
public void testDivideByZero(){
    cal.divide(4,0);
}
```

上例代码中的表达式“expected＝ArithmeticException.class”表示此方法执行后，必须抛出 ArithmeticException 异常错误才能认为测试执行成功。此方法的实现代码包含“4/0”的非法计算逻辑，因为除数不能为 0，因此测试程序执行后会抛出 ArithmeticException 异常。此测试方法接收抛出的异常信息，判断是否为 ArithmeticException 异常。如果是，则设定此测试用例为执行成功状态。此方法主要用于验证某种异常是否被正确抛出。

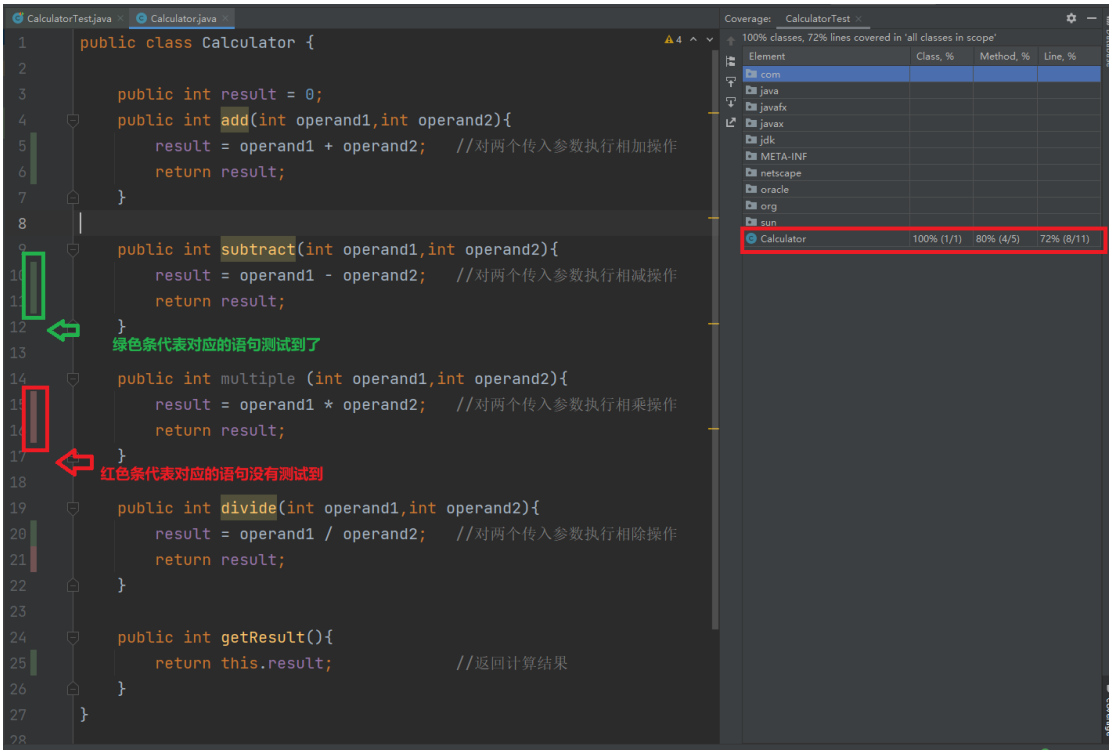
```
@Test
public void subtract() {
    cal.subtract(4,2);
    assertEquals(2,cal.getResult());
}
```

包含 assert 关键字的方法通常称为断言方法，上述代码用于判断期望结果是否和代码的实际执行结果一致，若一致就继续执行后续代码；若不一致则设定此测试用例为执行失败状态，且不继续执行后续代码。此测试方法调用 Calculator 实例的 subtract()方法，分别传入 4 和 2 参数，调用 assertEquals()方法断言实际计算结果是否等于 2。实际计算结果为 2，所以测试程序断言成功，设定测试用例的执行为成功状态。

6) 再次运行 CalculatorTest 测试类，选择 Run with Coverage



运行后得到当前被测试代码 Calculator 类的覆盖率



左侧 Calculator 类中，绿色条代表对应的语句测试到了，红色条代表对应的语句没有测试到。右侧框中显示了 CalculatorTest 测试代码覆盖了 Calculator 类中 80%的方法和 72%的语句。

7) 使用 JUnit 编写的 WebDriver 脚本

可打开前两次的 IDEA 项目，修改代码为：

```
public WebDriver driver;
String url = "http://www.baidu.com";

@Before
public void setUp() throws Exception{
    System.out.println("测试开始");
    System.setProperty("webdriver.chrome.driver",
   "src/main/resources/chromedriver.exe");// chromedriver 服务地址
    driver = new ChromeDriver(); // 新建一个 WebDriver 的对象，但是 new 的是谷歌的驱动
}

@After
public void tearDown() throws Exception{
    System.out.println("测试结束");
    Thread.currentThread().sleep(2000); //等待两秒后再执行
    driver.quit();     // 关闭打开的浏览器
}

@Test
public void test(){
    driver.get(url+"/");   //打开百度首页
    driver.findElement(By.id("kw")).sendKeys("JUnit"); //在搜索框中输入"JUnit"
    driver.findElement(By.id("su")).click();                  //点击"百度一下"按钮
}
```

说明：

在 setUp()函数中进行测试前的准备工作，打开 Chrome 浏览器。

tearDown()函数主要负责测试用例执行后的环境清理和还原工作，此处清理工作是等待 2 秒后关闭已打开的 Chrome 浏览器。

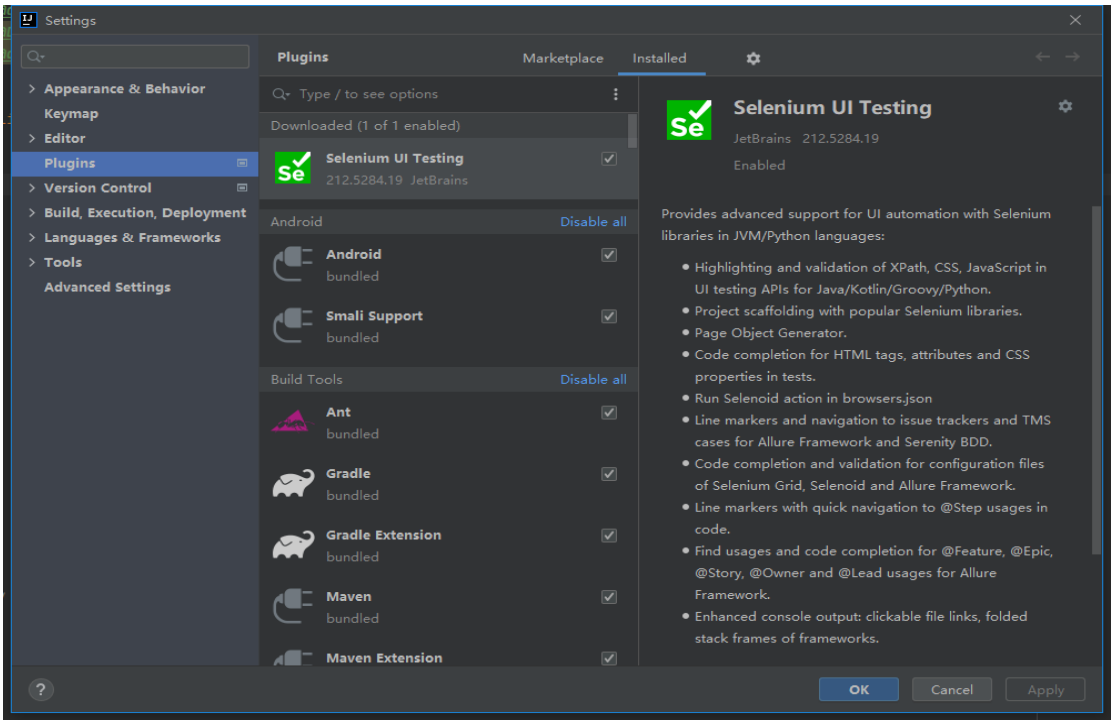
在 test()函数中执行测试用例代码，搜索 JUnit。

运行结果为：

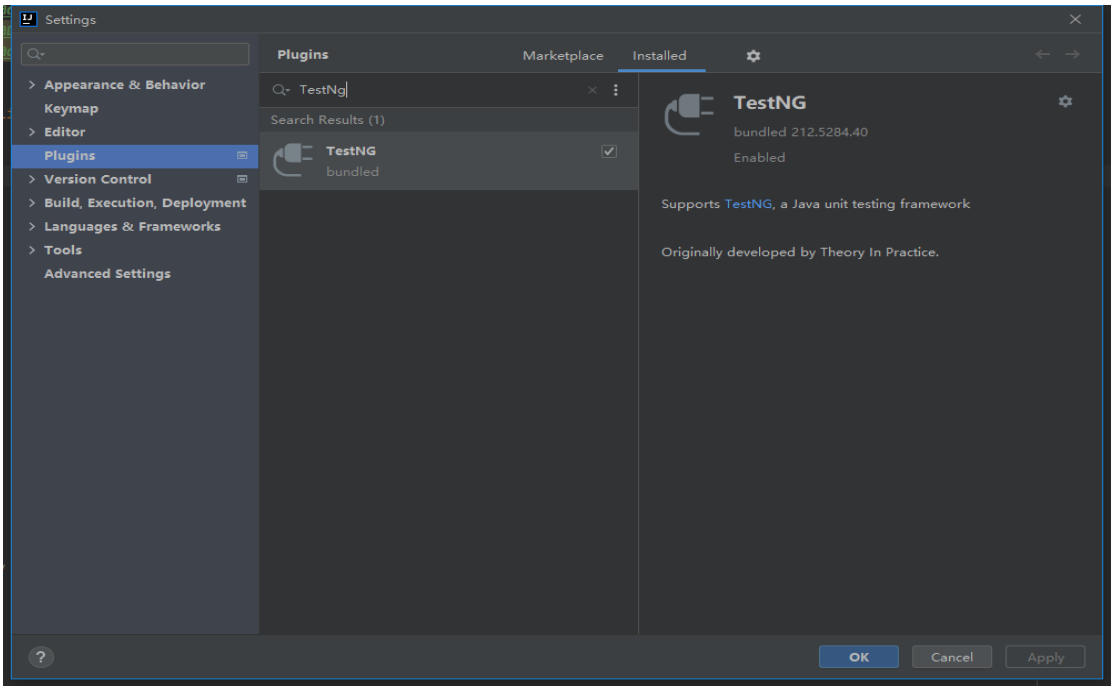
3. TestNG 的安装与使用

一、 安装

1. 打开 IDEA 的 Settings 页面

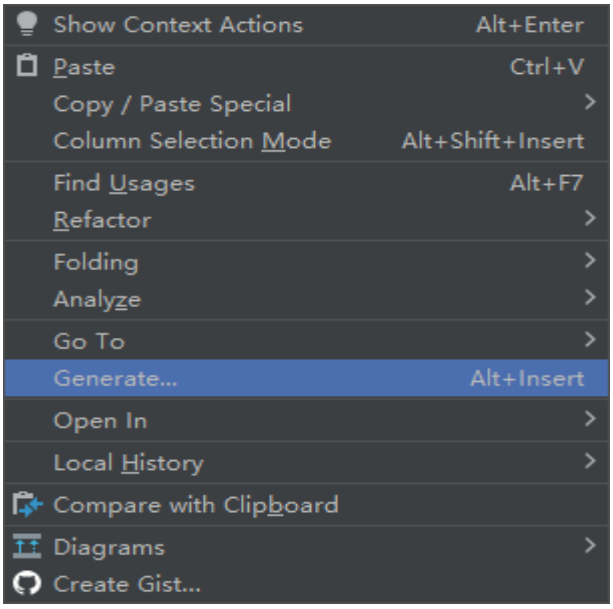


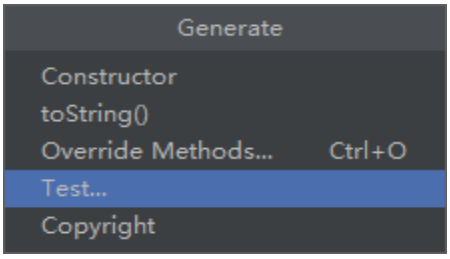
2. 在 Installed 下搜索 TestNG 检查 TestNG 是否被安装，若不存在则在 Marketplace 界面下搜索并下载



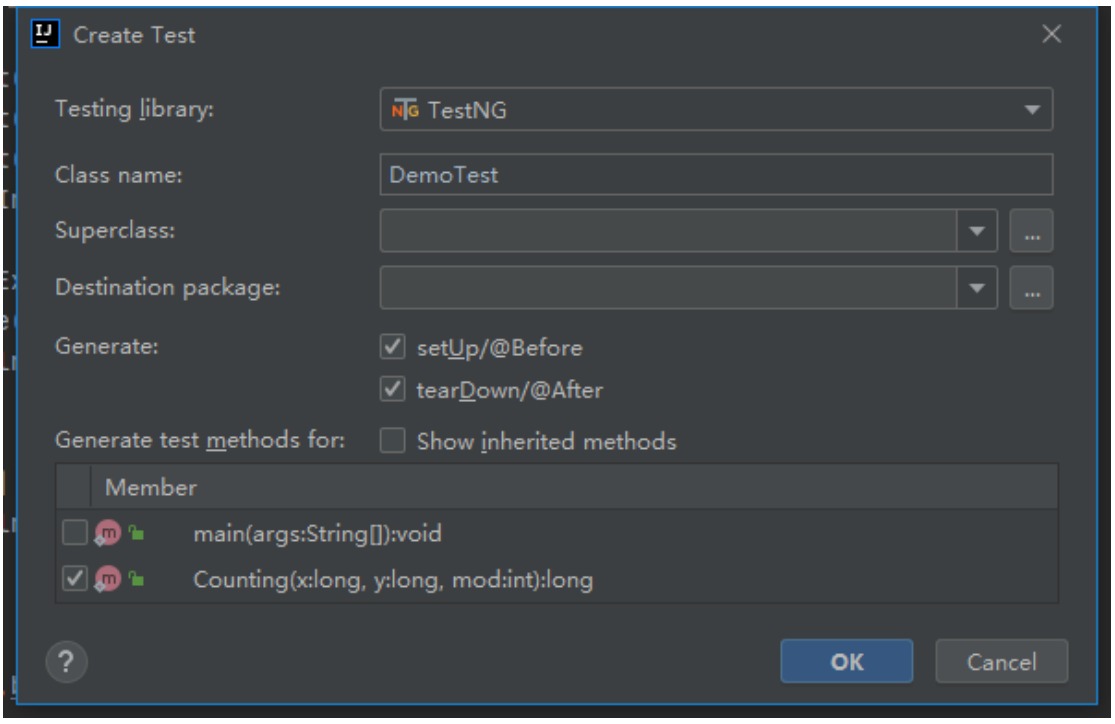
二、 使用

1. 生成一个测试类





点击 Generate 选择 Test，出现如下界面，



在 pom.xml 添加依赖

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>RELEASE</version>
    <scope>test</scope>
</dependency>
```

添加断言 assertEquals ，判断函数的输出是否符合预期，若实际得到的值和期望不同抛出异常如图

2. 以下介绍各个注解

注解	描述
@BeforeSuite	注解的方法将只运行一次，运行所有测试前此套件中。
@AfterSuite	注解的方法将只运行一次此套件中的所有测试都运行之后。
@BeforeClass	注解的方法将只运行一次先行先试在当前类中的方法调用。
@AfterClass	注解的方法将只运行一次后已经运行在当前类中的所有测试方法。
@BeforeTest	注解的方法将被运行之前的任何测试方法属于内部类的<test>标签的运行。
@AfterTest	注解的方法将被运行后，所有的测试方法，属于内部类的<test>标签的运行。
@BeforeGroups	组的列表，这种配置方法将之前运行。此方法是保证在运行属于任何这些组第一个测试方法，该方法被调用。
@AfterGroups	组的名单，这种配置方法后，将运行。此方法是保证运行后不久，最后的测试方法，该方法属于任何这些组被调用。
@BeforeMethod	注解的方法将每个测试方法之前运行。
@AfterMethod	被注释的方法将被运行后，每个测试方法。
@DataProvider	标志着一个方法，提供数据的一个测试方法。注解的方法必须返回一个 Object[] []，其中每个对象[]的测试方法的参数列表中可以分配。 该@Test 方法，希望从这个 DataProvider 的接收数据，需

	要使用一个 dataProvider 名称等于这个注解的名字。
@Factory	作为一个工厂，返回 TestNG 的测试类的对象将被用于标记的方法。该方法必须返回 Object[]。
@Listeners	定义一个测试类的监听器。
@Parameters	介绍如何将参数传递给@Test 方法。
@Test	标记一个类或方法作为测试的一部分。

3. TestNG 使用 xml 运行

TestNG 除了在 IDE 中使用还可以通过编写 XML 文件来运行 TestNG。<suite>为 Testing.xml 文档中最上层的元素一个文件只能有<suite>，是 xml 文件的根级<suite>由<test>和<parameters>组成以下是一个简单 xml 直接点击运行就可以运行指定的测试类。

```
<suite name="TestNGSuite">
   <parameter name = "a" value = "Hello"/>
   <test name = "test1">
      <groups>
         <run>
            <include name = "分组一"/>
         </run>
      </groups>
      <classes>
         <class name = "DemoTest"></class>
      </classes>
   </test>
</suite>
```

4. 分组运行测试类

通过 groups 可以进行分组运行，指定测试类中的部分 test 方法运行，MedoTest 类如下

```
import org.testng.annotations.*;
import org.testng.annotations.Test;

import static org.testng.Assert.*;

public class DemoTest {

    @BeforeMethod
    public void setUp() {
        System.out.println("在每个测试方法开始运行前执行");
    }
```

```java
@Test(groups = "分组一")
public void testCase(){
    System.out.println("执行测试 2-2");

}

@Test(groups = "分组二")
public void testCase2_1(){
    System.out.println("执行测试 2_1");
}

@Test(groups = "分组二")
@Parameters({"a"})
public void testCase2_2(String a){
    System.out.println("执行测试 2_2" + a);
}

@AfterClass
public void afterClass(){
    System.out.println("在当前测试类的最后一个测试方法结束运行后执行");
}

@BeforeClass
public void beforeClass(){
    System.out.println("在当前测试类的第一个测试方法开始调用前执行");
}

@BeforeTest
public void beforeTest(){
    System.out.println("在测试类中的 Test 开始运行前执行");
}

@AfterTest
public void afterTest(){
    System.out.println("在测试类中的 Test 结束运行后执行");
}

@BeforeSuite
public void beforeSuite(){
    System.out.println("在当前测试集合中的所有测试程序开始运行前执行");
}
```

```
    @AfterMethod
    public void tearDown() {
        System.out.println("在所有测试方法运行结束后执行");
    }

    @Test(groups = "分组一")
    public void testCounting() {
        Demo demo = new Demo();
        assertEquals(demo.Counting(2,3,8),0);
        System.out.println("Hello Test");
    }
    @AfterSuite
    public void afterSuite(){
        System.out.println("在当前测试集合中的所有测试程序结束运行后开始执
行");
    }

}
```

5. 添加参数在测试方法中

测试类中的@Parameters()注释可以为被注释的方法添加参数将参数和测试的方法进行分离在 xml 文件中实现传入参数如<parameter name = "a" value = "Hello"/>即可传入 Hello 字符串。

6. @test 注释中的 priority 规定了测试方法的运行优先级，从小到大。

```java
import org.testng.annotations.Test;

import static org.testng.Assert.assertEquals;

public class DemoTest_1 {
    @Test(priority = 2)
    public void testCase_2() {
        System.out.println("我是第 2 个");
    }

    @Test(priority = 1)
    public void testCase_1() {
        System.out.println("我是第 1 个");
    }

    @Test(priority = 0)
    public void testCase_0() {
        System.out.println("我是第 0 个");
    }

    @Test(priority = 3)
    public void testCase_3() {
        System.out.println("我是第 3 个");
    }
}
```

7. 依赖测试：

在 test 注释中添加 dependsOnMethods 参数等待某个方法运行完毕后才可运行

```java
import org.testng.annotations.Test;

public class DemoTest_2 {

    @Test(dependsOnMethods = {"testCase_2"})
    public void testCase_1() {
        System.out.println("testCase_2 运行完毕");
    }

    @Test()
    public void testCase_2() {
        System.out.println("testCase_2 正在运行");
    }

}
```

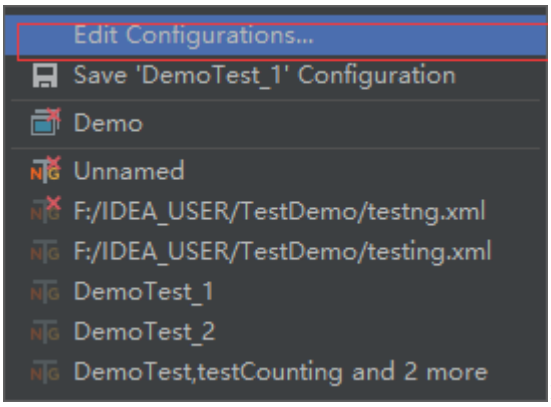
8. 跳过某个测试：

在 test 注释中添加 enabled = false 属性跳过某个测试方法

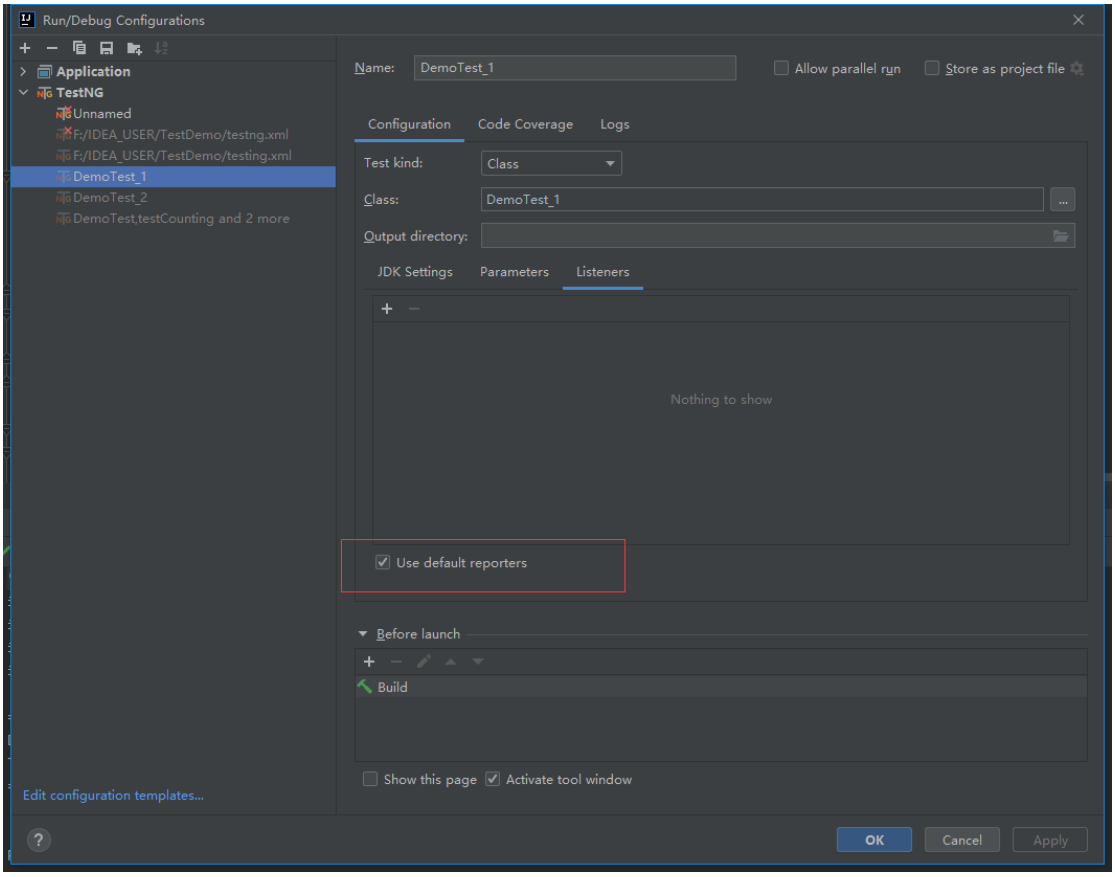
9. 自定义报告：

使用 Reporter.log();自定义报告

10. 生成报告：



选择测试类



选择 Use default reporters 即可

生成报告如图：