TestNG使用手册



**中科软科技股份有限公司**

**寿险事业本部**

**二〇一八年一月**

中科软科技文档专有信息声明

本文档包含的所有信息属Sinosoft（中科软科技股份有限公司）专有。接收公司仅供使用。所有内容均为机密信息，若未事先征得Sinosoft的书面同意，接收公司确认将不会泄露给其它公司或与此项目无关的任何个人。接收公司需确保遵守与此有关的法律、法规和条款，来谨慎地使用这些信息。

尽管Sinosoft已经尽力使本文档的内容全面准确，但仍可能有技术方面不够准确的地方或印刷错误。此处的信息将被不定期修改；这些信息将包含在本文档的未来版本中。Sinosoft可以在任何时间对本文档中说明的产品或程序进行修改，而不必通知您。

Sinosoft是中科软科技股份有限公司的注册商标。本文档提及的其他公司、产品和服务的名称，可能是其他公司的商标或服务的标志。

Copyright © 2018 Sinosoft Co., Ltd.

目录

[一 TestNG简介 4](#_Toc24886)

[二 Eclipse安装TestNG插件 6](#_Toc16951)

[三 TestNG注解 13](#_Toc7674)

[四 TestNG注解属性和值 15](#_Toc7180)

[五 TestNG常规测试及案例 17](#_Toc1210)

[1. TestNG的测试方法 17](#_Toc7474)

[2. TestNG 需要的jar包 17](#_Toc10935)

[3. TestNG简单应用 18](#_Toc21007)

[六 TestNG高级应用及案例 19](#_Toc11838)

[1. 参数化测试 19](#_Toc7480)

[（1） 利用testng.xml传递参数 19](#_Toc5929)

[（2） 利用@DataProvider传递参数 21](#_Toc8296)

[2. 依赖测试 28](#_Toc31845)

[3. 组测试 30](#_Toc10869)

[4. 异常测试 33](#_Toc7841)

[5. 并行测试（使用线程实现） 34](#_Toc14774)

[6. 延迟数据提供（数据延迟加载） 38](#_Toc27036)

[7. 异步测试（使用线程实现） 40](#_Toc24235)

[七 TestNG结果输出和查看以及失败案例重新测试 42](#_Toc6309)

[八 附件 45](#_Toc26867)

**关于本文档**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **主 题** | | TestNG使用手册 | | | | |
| **说　　明** | | 本文档介绍了TestNG基本注解的作用以及TestNG的高级应用和案例 | | | | |
| **适用对象** | | 项目经理、开发人员、配置人员、测试人员 | | | | |
| **修 订 历 史** | | | | | | |
| **版 本** | **章 节** | | **类 型** | **日 期** | **作 者** | **说 明** |
| 1.0 |  | | C | 2018-6-6 | 张东亮 | 创建文档 |

说明：类型－创建（C）、修改（U）、删除（D）、增加（A）；

# 一 TestNG简介

1、TestNG是Java中的一个测试框架，类似于JUnit 和NUnit, 功能都差不多，只是功能更加强大，使用也更方便。

2、Java中已经有一个JUnit的测试框架了。TestNG比JUnit功能强大的多。 测试人员一般用TestNG来写自动化测试。开发人员一般用JUnit写单元测试。

3、TestNG是一个设计用来简化广泛的测试需求的测试框架，从单元测试到集成测试。这个是TestNG设计的出发点，不仅仅是单元测试，而且可以用于集成测试。设计目标的不同，对比junit的只适合用于单元测试，TestNG无疑走的更远。

4、TestNG和JUnit的比较

**相同点：** 1.都可以做忽略测试，可以忽略某个测试方法(在方法上面注释)，也可以忽略某个测试类(在类的上面注释)；

2.都支持数据驱动测试，只是用法不一样；

3.超时测试，就是在规定时间内如果没有测试完成，就认定测试失败；

4.异常测试，就是在运行这个单元测试的时候应该要捕获到指定的异常，才算测试成功。

**不同点：**

1.TestNg允许分组测试；

2.TestNg允许只运行失败的例子；

3.TestNg允许依赖测试（类似于ant的依赖）；

4.TestNg支持并行测试（支持测试方法(methods)，测试类(classes)，小的测试套件（tests），可以大大提高测试效率；

5.TestNg标签比junit丰富；

6.testng被@BeforeClass 和@AfterClass注释的方法可以不写成static方法；

7.TestNg被@Test标记的方法可以有输入参数，而在junit中是不行的；

8.testNg可以通过标注的方式来顺序执行；

9.testng中子类不会运行父类中的@BeforeClass和@AfterClass；而在junit中会先运行父类的@BeforeClass，再运行自己的@BeforeClass；而@AfterClass是先运行自己的，再运行父类的。

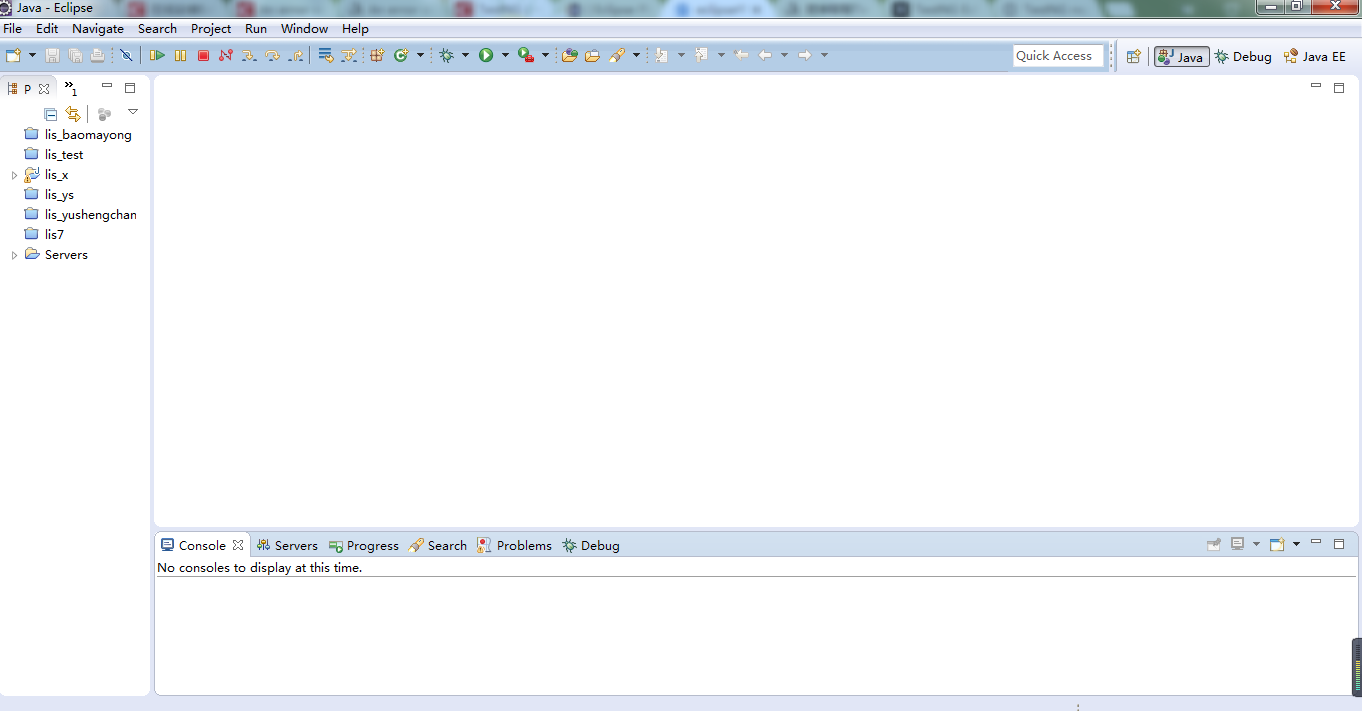
**下面是注解类的比较：**

|  |  |  |
| --- | --- | --- |
| 功能 | JUnit | TestNG |
| 标注为类/方法为测试类和方法 | @Test | @Test |
| 标注为在suite中所有测试之前运行 | 无 | @BeforeSuite |
| 标注为在suite中所有测试之后运行 | 无 | @AfterSuite |
| 标注为在测试之前运行 | 无 | @BeforeTest |
| 标注为在测试之后运行 | 无 | @AfterTest |
| 标注为在测试Group中第一个测试方法之前运行 | 无 | @BeforeGroups |
| 标注为在测试Group中最后一个测试方法之后运行 | 无 | @AfterGroups |
| 标注为当前测试类中第一个测试方法之前运行 | @BeforeClass | @BeforeClass |
| 标注为当前测试类中最后一个测试方法之后运行 | @AfterClass | @AfterClass |
| 标注为在每次测试方法之前运行 | @Before | @BeforeMethod |
| 标注为在每次测试方法之后运行 | @After | @AfterMethod |
| 忽略某测试，让其不执行 | @Ignore | @Test(enable=false) |
| 期待测试抛出什么异常 | @Test(expected=XXXException.class) | @Test(exceptedExceptions=XXXException.class) |
| 测试超时，如果测试的执行时间超过了毫秒为单位设置的时间，那么就停止测试并且标记为失败 | @Test(timeout=1000) | @Test(timeout=1000) |

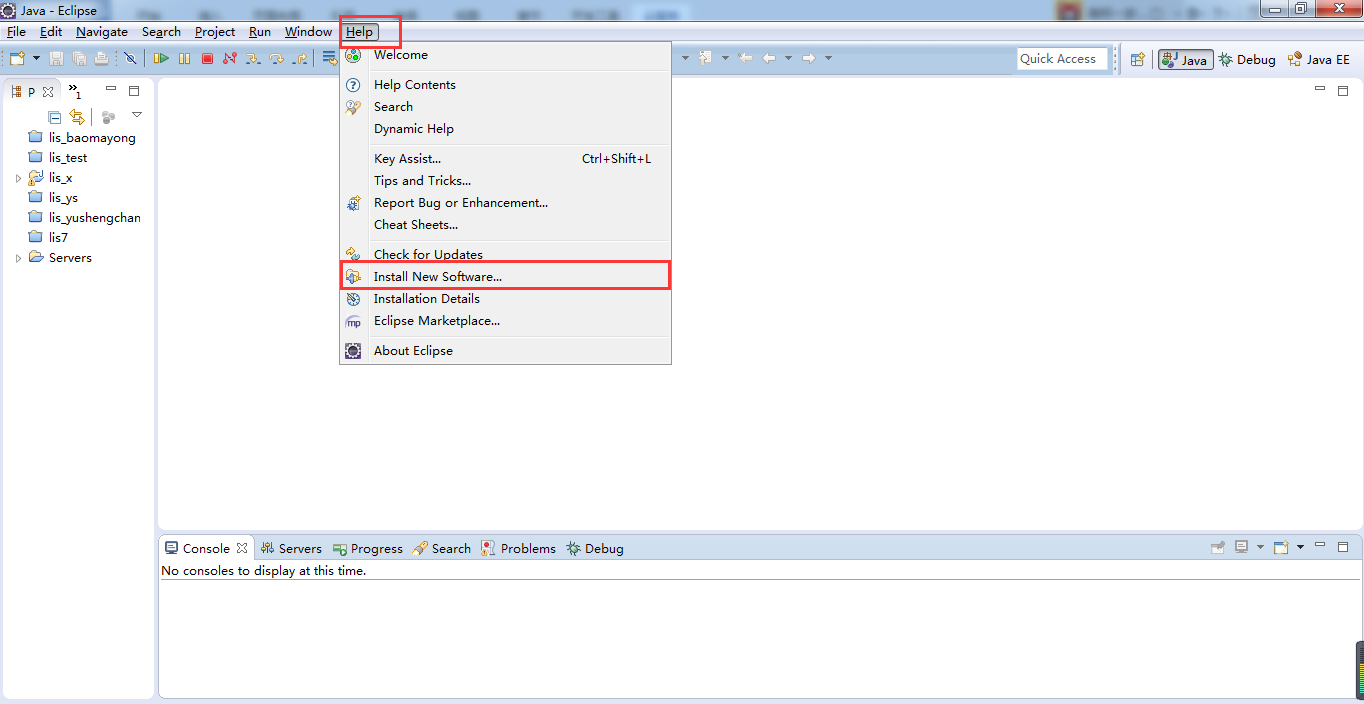
# 二 Eclipse安装TestNG插件

步骤：

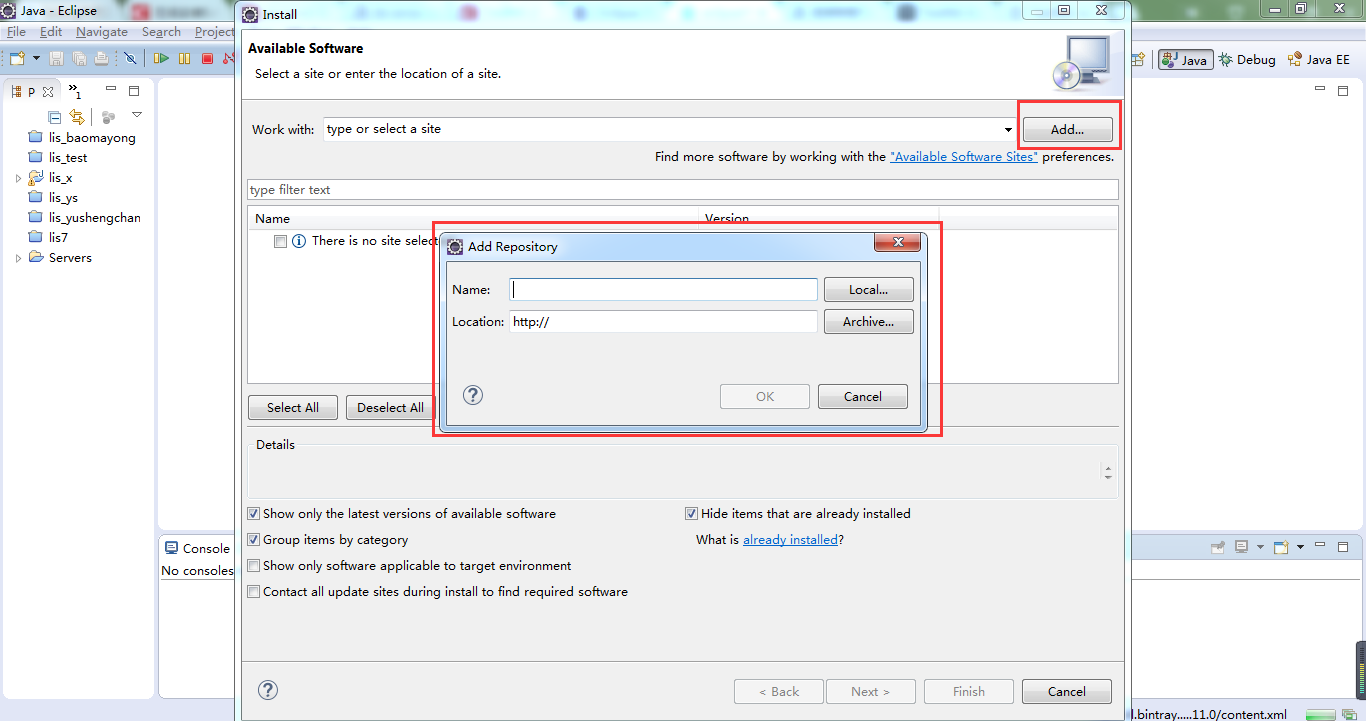
1. 启动eclipse程序，进入主界面



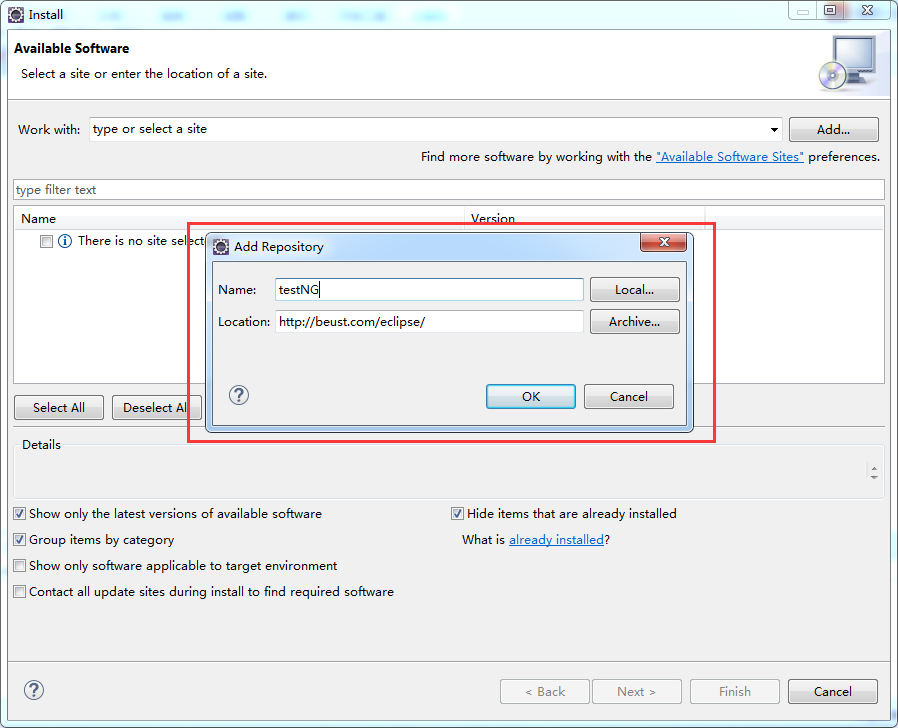
1. 主界面 菜单栏找到【help】-【Install New Software，进入安装界面



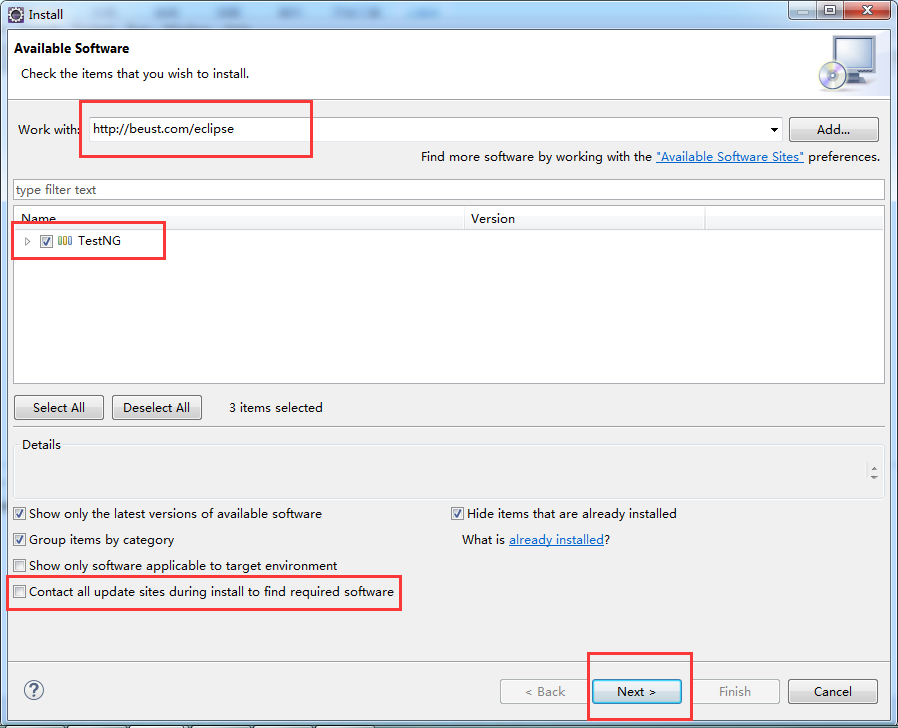
1. 安装界面，点击【Add】按钮，如下图，弹出输入弹框

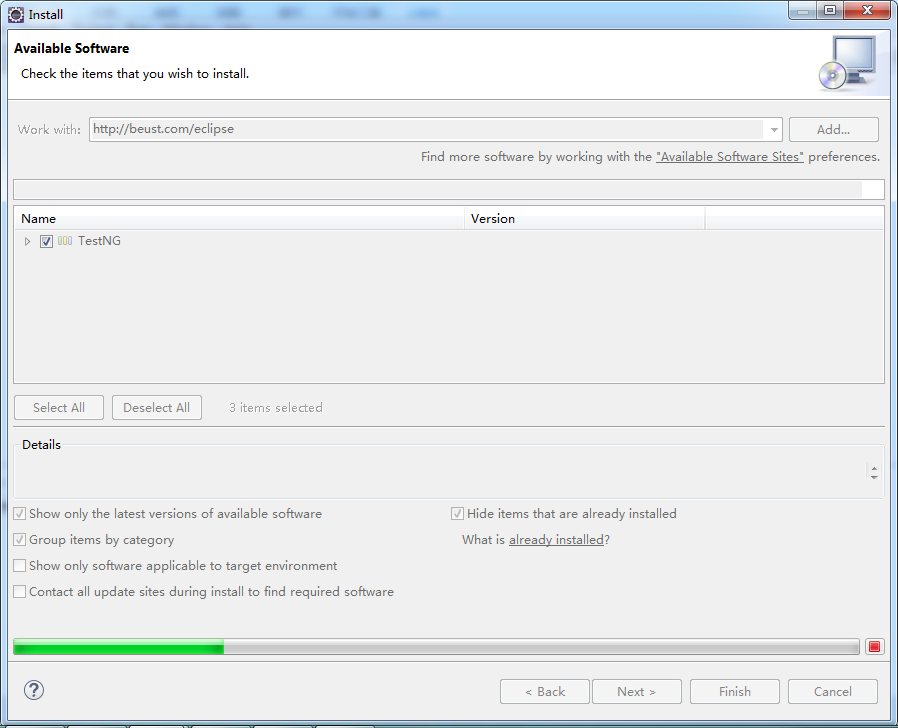


1. 在输入弹框中，输入name，输入在线安装地址：[http://beust.com/eclipse/ ， 如下图，并点击【确认】按钮](http://beust.com/eclipse/，如下图，并点击【确认】按钮)

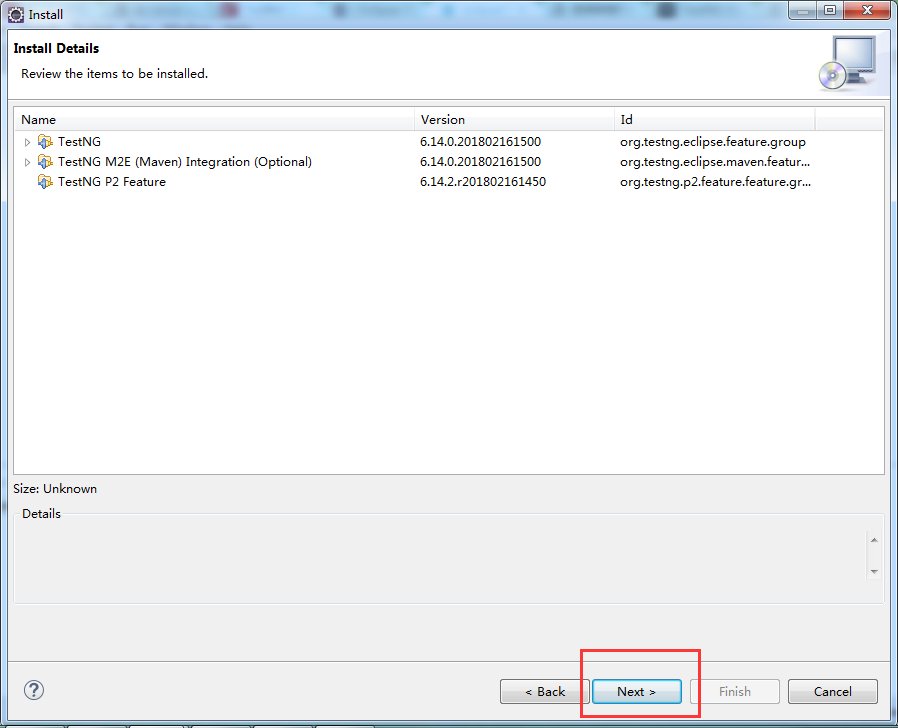


1. 点击【确认】按钮后 ，选择可安装的地址、勾选要安装的软件名称，点击【next】进入下载

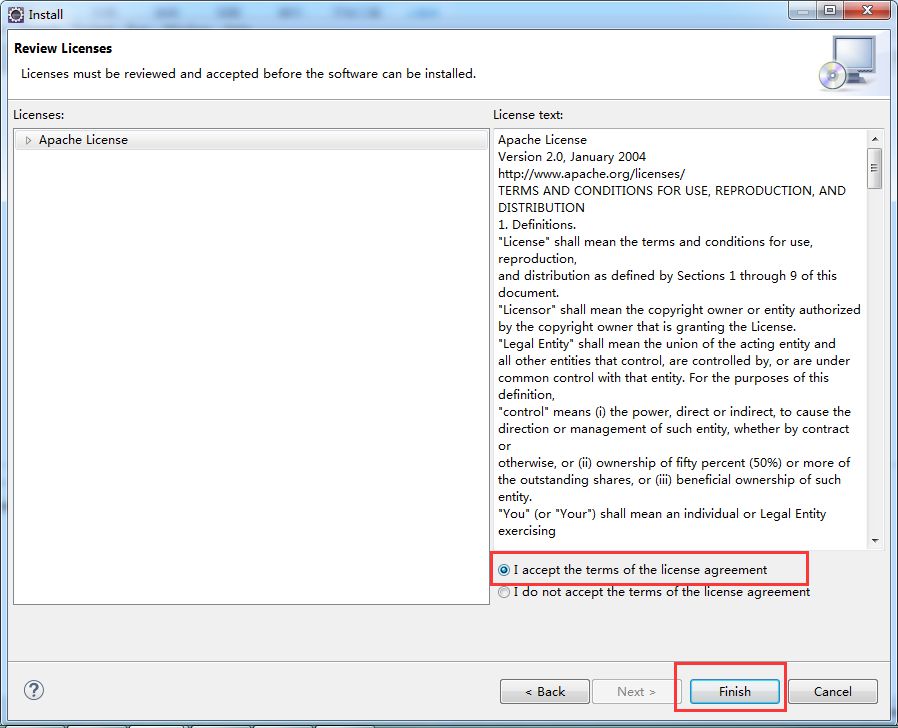


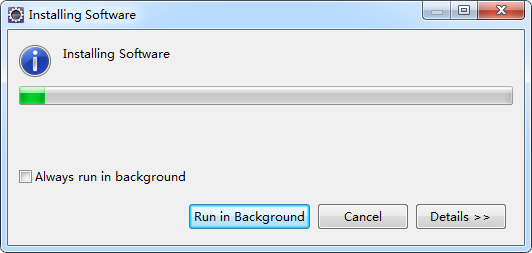


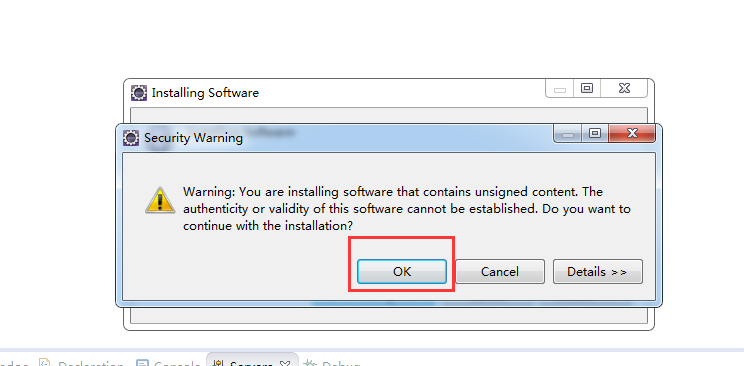
1. 下载完成后 确认需要安装的信息，如下图



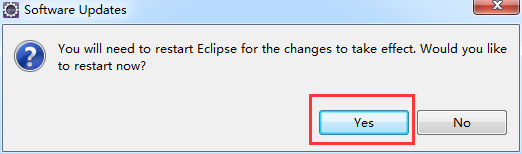
1. 确认后 进入安装界面，如下，点击 同意【I accept.....】



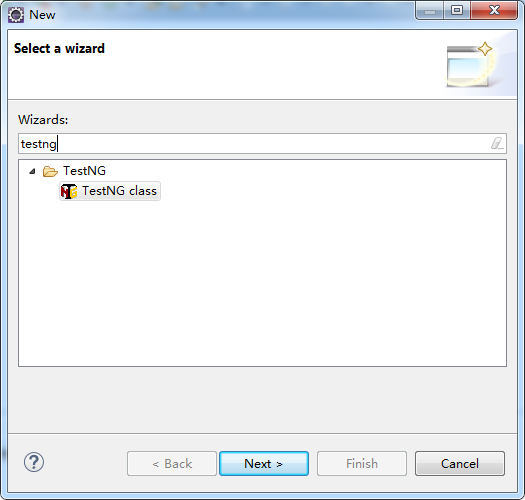




1. 安装完毕 需要重启，如下图 ，点击【yes】重启程序



1. 重启程序后  可新建testNG测试类



注意：在线安装时 可能会报错，若报错了 可重新再次安装一遍，安装时把“Contact all update sites during install to find required software”这个勾去掉

# 三 TestNG注解

|  |  |
| --- | --- |
| **注解** | **描述及作用** |
| @Test | 标记一个类或方法作为测试的一部分 |
| @BeforeSuite | 被注释的方法将在所有测试运行前运行 |
| @AfterSuite | 被注释的方法将在所有测试运行后运行 |
| @BeforeTest | 被注释的方法将在测试运行前运行 |
| @AfterTest | 被注释的方法将在测试运行后运行 |
| @BeforeGroups | 被配置的方法将在列表中的gourp前运行。这个方法保证在第一个属于这些组的测试方法调用前立即执行。 |
| @AfterGroups | 被配置的方法将在列表中的gourp后运行。这个方法保证在最后一个属于这些组的测试方法调用后立即执行。 |
| @BeforeClass | 被注释的方法将在当前类的第一个测试方法调用前运行。 |
| @AfterClass | 被注释的方法将在当前类的所有测试方法调用后运行。 |
| @BeforeMethod | 被注释的方法将在每一个测试方法调用前运行。 |
| @AfterMethod | 被注释的方法将在每一个测试方法调用后运行。 |
| @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。 |
| @Parameters | 介绍如何将参数传递给@Test方法。 |
| @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。 |

**注：**

作用域的不一样划分出了各种不同的标签：

一个套件（suite）包含多个测试(test)

一个测试（test）包含多个类（class）

一个测试（test）包含多个组（group）

一个类（class）包含多个方法（method）

# 四 TestNG注解属性和值

|  |  |  |
| --- | --- | --- |
| **注解属性** | **适用的注解** | **描述及作用** |
| alwaysRun | Before\*/After\*/Test | 对于每个bufore方法(beforeSuite, beforeTest, beforeClass 和 beforeTestMethod, 但是不包括 beforeGroups)： 如果设置为true，被配置的方法将总是运行而不管它属于哪个组。  对于after方法(afterSuite, afterClass, ...)： 如果设置为true，被配置的方法甚至在一个或多个先调用的方法失败或被忽略时也将运行。  对于after方法(Test)：如果设置为true，这个测试方法将总是运行，甚至当它依赖的方法失败时。 |
| enabled | Before\*/After\*/Test | 当前类的方法/方法是否被激活（是否有效） |
| groups | Before\*/After\*/Test | 当前类/方法所属的组列表（分组）。其他注释都可使用该属性。  注意：一旦@test使用了属性groups，其他要运行在该测试中的方法必须加groups属性，不然不会执行。 |
| dependsOnGroups | Before\*/After\*/Test | 这个方法依赖的组列表 |
| dependsOnMethods | Before\*/After\*/Test | 这个方法依赖的方法列表 |
| name | DataProvider | 这个DataProvider的名称 |
| value | Parameters | 用于填充这个方法的参数的变量列表 |
| dataProvider | Test | 这个测试方法的data provider的名称 |
| timeOut | Test | 当前方法容许花费的最大时间，单位毫秒。 |
| dataProviderClass | Test | 用于查找data provider的类。 如果不指定，将在当前测试方法所在的类或者它的基类上查找data provider。 如果这个属性被指定, 则data provider方法需要是指定类的static方法。 |
| description | Test | 当前方法的描述 |
| expectedExceptions | Test | 测试方法期望抛出的异常列表。如果没有异常或者抛出的不是列表中的任何一个，当前方法都将标记为失败 |
| invocationCount | Test | 当前方法被调用的次数 |
| successPercentage | Test | 当前方法期望的成功率 |
| threadPoolSize | Test | 当前方法的线程池大小。方法将被多线程调用，次数由invocationCount参数指定。 注意：如果invocationCount没有指定则这个属性将被忽略 |

**注：**

1. before方法和after方法带来了足够丰富的测试生命周期控制；  
   2. dependsOnGroups/dependsOnMethods 提供了依赖检查机制，并可以严格控制执行顺序；  
   3. DataProvider 使得对同一个方法的测试覆盖变的非常轻松，非常适合进行边界测试，只要给出多种测试数据就可以针对一个测试方法进行覆盖；  
   4. expectedExceptions 使得异常测试变的非常轻松；  
   5. invocationCount/threadPoolSize 终于可以简单的直接进行多线程测试了，这个绝对是junit的超级弱项，回想junit中那个万恶的System.exist(0)；  
   6. timeOut 终于不用死等然后手工强行关闭测试，TestNG想的太周到了。

# 五 TestNG常规测试及案例

## TestNG的测试方法

* 右键要执行的方法，点Run As ->TestNG Test。
* 通过testng.xml文件来执行。把要执行的case, 放入testng.xml文件中。 右键点击testng.xml,  点Run As ->TestNG Test。

## TestNG 需要的jar包

* 非maven项目需要导入jar包：jcommander-1.72.jar、testng-6.14.2.jar。jar包可以在附件中的bin目录下找到。
* maven项目在pom.xml文件中添加以下代码：

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>6.14.2</version>

</dependency>

解析Excel需要的jar包

<dependency>

<groupId>net.sourceforge.jexcelapi</groupId>

<artifactId>jxl</artifactId>

<version>2.6.12</version>

</dependency>

* 解析Excel需要的jar包：jxl-2.6.12.jar，可以在附件中的bin目录下找到。

## TestNG简单应用

简单的测试主要是对@Test的应用。在测试方法之前做的事要在@BeforeTest标记的方法中实现；在测试方法之后做的事要在@AfterTest标记的方法中实现。

场景：适用于一个单独的测试。案列如下：

* Testng.java测试类代码

**package** com.sinosoft.demo;

**import** org.testng.annotations.AfterTest;

**import** org.testng.annotations.BeforeTest;

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

**public** **class** Testng {

@BeforeTest

**public** **void** beforeTest() {

System.***out***.println("beforeTest");

}

@Test

**public** **void** test() {

System.***out***.println("这是最简单的testng测试！");

}

@AfterTest

**public** **void** afterTest() {

System.***out***.println("afterTest");

}

}

* Testng.xml代码

*<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >*

<suite name="Suite" parallel="false">

<test name="Test">

<classes>

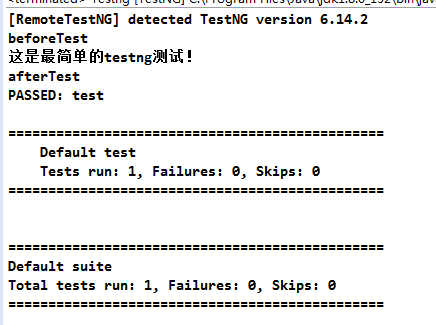
<class name="com.sinosoft.demo.Testng"/>

</classes>

</test>

</suite>

* 运行结果如下：



# 六 TestNG高级应用及案例

## 参数化测试

### 利用testng.xml传递参数

利用testng.xml传递参数可以在testng.xml文件中明确显示参数值，随时更改参数值以来测试不同条件的案例；但有一定的局限性，参数值必须是java的基本数据类型，并且参数值明确知道什么值。使用这种方式传参需要注意：1、参数类型必须能够转化，否则会报错；2、在测试类中引用了一个xml中没有定义的参数名，这时也会报错。需要注意！

场景：适用于参数是地址、用户名和密码、数据库的信息配置等场景。

主要是对 @Parameters 的应用。在 \*.xml文件中设置参数值，然后在测试类中用@Parameter使用。案例如下：

（案例说明：在XmlParamTest.xml中用标签<parameter>定义参数名和值，在XmlParamTest.java中用@Parameters使用。）

* XmlParamTest.java 测试类代码

**package** com.sinosoft.demo;

**import** org.testng.annotations.Parameters;

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

**public** **class** XmlParamTest {

@Parameters("filePath")

@Test

**public** **void** test2(String filePath) {

System.***out***.println("数据保存的路径为："+filePath);

}

@Parameters({"username", "password"})

@Test

**public** **void** loginTest(String username, String password) {

System.***out***.println("用户登录信息：用户名："+username+"，密码："+password);

}

}

* XmlParamTest.xml文件代码

*<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >*

<suite name="Suite" parallel="false">

<parameter name="filePath" value="C:\\Users\\Administrator\\Desktop\\data.xls"></parameter>

<test name="Test">

<parameter name="username" value="001"></parameter>

<parameter name="password" value="001"></parameter>

<classes>

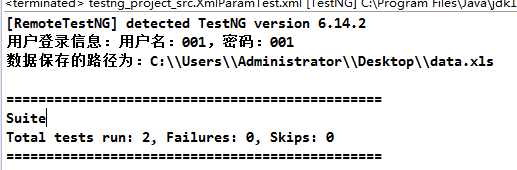
<class name="com.sinosoft.demo.XmlParamTest"/>

</classes>

</test>

</suite>

* 测试结果为：



### 利用@DataProvider传递参数

如果向测试方法传递的参数不是java基本数据类型，或者需要的值只能在运行时创建，或者参数值太多且案例条件太多，这时应该用@DataProvider传递参数。

主要是对 @DataProvider 的应用。定义一个方法，这个方法必须返回Object[][]或Iterator，用@DataProvider修饰，可不加name属性，默认为方法名。然后在@Test里用dataProvider=”参数名”使用。

场景：适用于一个测试方法用多条不同数据测试不同结果的场景。例如保全核保测试，使用不同的测试数据测试核保通过、核保规则校验拦住核保不通过、核保失败等场景。案例如下（数据来源于Excel）：

（案例说明：使用Excel提供数据；在ExcelReadDataUtil.java中解析Excel数据并封装为List<Map<String, String>>结构的数据，然后在DataProviderTest.java中用@DataProvider标记的方法获取参数，在@Test中用dataProvider=”参数名”使用。）

* DataProviderTest.java 测试类代码

**package** com.sinosoft.demo;

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

**import** java.util.Map;

**import** org.testng.annotations.DataProvider;

**public** **class** DataProviderTest {

@Test(dataProvider = "param")

**public** **void** UWTest(Map<String, String> paramMap) {

String edorappno = paramMap.get("edorappno");

String contNo = paramMap.get("contNo");

String riskCode = paramMap.get("riskCode");

**double** prem = Double.*valueOf*(paramMap.get("prem"));

**double** amnt = Double.*valueOf*(paramMap.get("amnt"));

**if**("A0015E".equals(riskCode)){

**if**((prem>0 && prem<100) && (amnt>10000 && amnt<500000)){

System.***out***.println("核保成功！");

} **else** {

System.***out***.println("保全受理号："+edorappno+"，保单号："+contNo+"，产品："+riskCode+"，核保失败！");

}

} **else** **if**("B0018A".equals(riskCode)){

**if**((prem>10 && prem<1000) && (amnt>50000 && amnt<100000)){

System.***out***.println("核保成功！");

} **else** {

System.***out***.println("保全受理号："+edorappno+"，保单号："+contNo+"，产品："+riskCode+"，核保失败！");

}

} **else** {

System.***out***.println("核保失败！没有该产品！");

}

}

@DataProvider(name="param")

**public** Object[][] getData() {

ExcelReadDataUtil u = **new** ExcelReadDataUtil("C:\\Users\\Administrator\\Desktop\\data.xls","Sheet1");

u.readExcel();

**return** u.getData();

}

}

* ExcelReadDataUtil.java 代码

**package** com.sinosoft.demo;

**import** java.io.File;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Set;

**import** jxl.Sheet;

**import** jxl.Workbook;

**import** jxl.read.biff.BiffException;

**public** **class** ExcelReadDataUtil **implements** Iterator{

**private** String excelFilePath;

**private** String sheetName;

**private** **int** rowsCount = 2;

**private** **int** currentRowNum = 1;

**private** List<Map<String, String>> dataList = **new** ArrayList<Map<String, String>>();

**private** Set<Map<String, String>> dataSet = **new** HashSet<Map<String, String>>();

**public** Set<Map<String, String>> getDataSet() {

**return** dataSet;

}

**public** List<Map<String, String>> getDataList() {

**return** dataList;

}

ExcelReadDataUtil(String excelPath, String sheetName){

**this**.excelFilePath = excelPath;

**this**.sheetName = sheetName;

}

/\*\*

\* 获取Excle数据

\* **@return** Object[][]形式的数据

\*/

**public** Object[][] getData(){

Object[][] obj = **null**;

**if**(dataList.size() > 0){

obj = **new** Object[dataList.size()][];

**for**(**int** i=0; i<dataList.size(); i++){

obj[i] = **new** Object[]{dataList.get(i)};

}

}

**return** obj;

}

/\*\*

\* 读取Excel数据放到List中

\*/

**public** **void** readExcel(){

**int** rows = 0;

**int** columns = 0;

List<String> arrkey = **new** ArrayList<String>();

File f = **new** File(excelFilePath);

**try** {

Workbook workbook = Workbook.*getWorkbook*(f);

Sheet sheet = workbook.getSheet(sheetName);

rows = sheet.getRows();

columns = sheet.getColumns();

Map<String, String> arrmap = **null**;

// 获得首行的列名，作为hashmap的key值

**for** (**int** c = 0; c < columns; c++) {

String cellvalue = sheet.getCell(c, 0).getContents();

arrkey.add(cellvalue);

}

// 把数据添加到List中

**for** (**int** r = 1; r < rows; r++) {

arrmap = **new** HashMap<String, String>();

**for** (**int** c = 0; c < columns; c++) {

String cellvalue = sheet.getCell(c, r).getContents();

arrmap.put(arrkey.get(c), cellvalue);

}

dataList.add(arrmap);

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (BiffException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

/\*\*

\* 读取Excel数据放到Set中

\*/

**public** **void** readExcelToSet(){

**int** rows = 0;

**int** columns = 0;

List<String> arrkey = **new** ArrayList<String>();

File f = **new** File(excelFilePath);

**try** {

Workbook workbook = Workbook.*getWorkbook*(f);

Sheet sheet = workbook.getSheet(sheetName);

rows = sheet.getRows();

columns = sheet.getColumns();

Map<String, String> arrmap = **null**;

// 获得首行的列名，作为hashmap的key值

**for** (**int** c = 0; c < columns; c++) {

String cellvalue = sheet.getCell(c, 0).getContents();

arrkey.add(cellvalue);

}

// 把数据添加到List中

**for** (**int** r = 1; r < rows; r++) {

arrmap = **new** HashMap<String, String>();

**for** (**int** c = 0; c < columns; c++) {

String cellvalue = sheet.getCell(c, r).getContents();

arrmap.put(arrkey.get(c), cellvalue);

}

dataSet.add(arrmap);

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (BiffException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

/\*\*

\* 读取Excel数据中第n条数据 n>0

\*/

**public** Map<String, String> readExcelForN(**int** n){

**if**(n<=0){

**return** **null**;

}

**int** rows = 0;

**int** columns = 0;

List<String> arrkey = **new** ArrayList<String>();

File f = **new** File(excelFilePath);

Map<String, String> arrmap = **null**;

**try** {

Workbook workbook = Workbook.*getWorkbook*(f);

Sheet sheet = workbook.getSheet(sheetName);

rows = sheet.getRows();

columns = sheet.getColumns();

**if**(n>=rows){

**return** **null**;

}

**this**.rowsCount = rows;

// 获得首行的列名，作为hashmap的key值

**for** (**int** c = 0; c < columns; c++) {

String cellvalue = sheet.getCell(c, 0).getContents();

arrkey.add(cellvalue);

}

// 把数据添加到Map中

arrmap = **new** HashMap<String, String>();

**for** (**int** c = 0; c < columns; c++) {

String cellvalue = sheet.getCell(c, n).getContents();

arrmap.put(arrkey.get(c), cellvalue);

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (BiffException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** arrmap;

}

@Override

**public** **boolean** hasNext() {

// **TODO** Auto-generated method stub

**return** currentRowNum < rowsCount;

}

@Override

**public** Object next() {

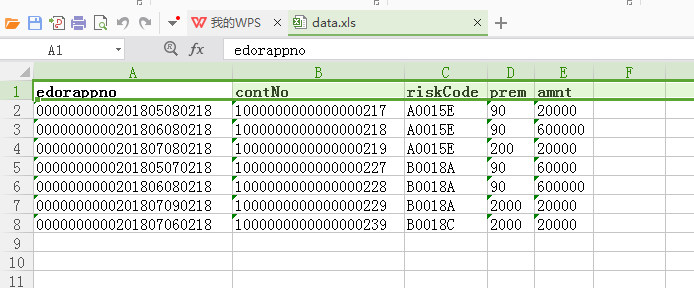
// **TODO** Auto-generated method stub

**return** **new** Object[]{readExcelForN(currentRowNum++)};

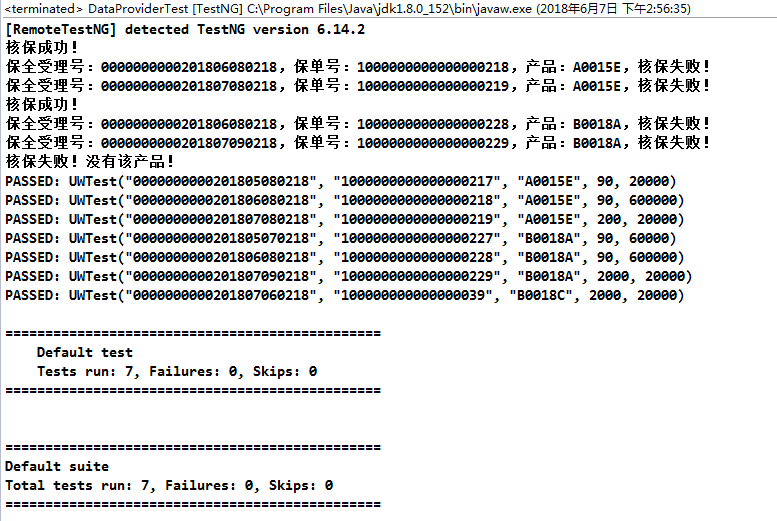
}

}

* Excel数据：



* 测试结果为：



## 依赖测试

TestNG依赖测试是 一个测试方法必须在另一个测试方法执行之后才能执行，就是一个测试方法依赖于另一个测试方法；当被依赖的测试方法失败时，依赖的测试方法则被跳过。

主要是对 @Test的属性 dependsOnMethods和dependsOnGroups的应用。

场景：适用于两个测试在业务上有逻辑关系的两个测试。例如：契约流程中的签单和交费，必须先交费，成功后才能签单；案列如下：

（案例说明：dependsOnGroups是依赖于组，dependsMethods是依赖于方法。在测试类RelyTest.java中，方法sign1和pay1属于组XQYGroup，方法edorAppTest依赖于组XQYGroup，所以只有当方法sign1和pay1都执行完后，edorAppTest方法才执行；看方法sign1依赖于方法pay1，所以只有当pay1方法执行完后，sign1方法才执行；在看方法sign2依赖于方法pay2，而pay2是失败的，交费失败是不能签单的，程序会自动跳过签单方法sign2。）

* RelyTest.java 测试类代码

**package** com.sinosoft.demo;

**import** org.testng.Assert;

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

**public** **class** RelyTest {

@Test(dependsOnGroups="XQYGroup")

**public** **void** edorAppTest() {

System.***out***.println("签单成功后才可申请保全项目！");

}

@Test(dependsOnMethods="pay1", groups="XQYGroup")

**public** **void** sign1() {

System.***out***.println("sign1签单成功！");

}

@Test(groups="XQYGroup")

**public** **void** pay1() {

System.***out***.println("pay1交费成功！");

}

@Test(dependsOnMethods="pay2")

**public** **void** sign2() {

System.***out***.println("sign2签单成功！");

}

@Test

**public** **void** pay2() {

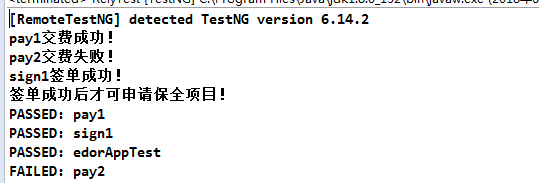
System.***out***.println("pay2交费失败！");

Assert.*fail*();

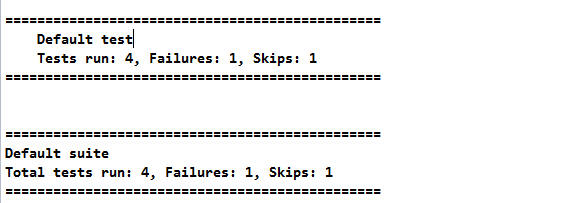
}

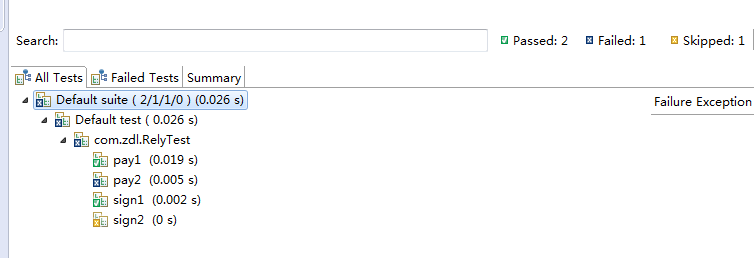
}

* 运行结果为：









## 组测试

在实际业务场景中，我们经常遇到只想执行个别或者某一部分/某一类型的测试用例，这时我们可以使用TestNG的分组测试方法。

主要是对 属性groups的应用。注解类@Before\*/@After\*/@Test都可用该属性来区分属于哪个组套件。

场景：适用于可分模块测试的测试。例如：可按照 新契约、保全、理赔 三个模块分组测试；案列如下：

（案例说明：该测试类分了两组---新契约组（XQY）和保全组（BQ），GroupTest.java中定义了两组5个测试方法，在GroupTest.xml中配置需要运行的组，GroupTest.xml(1)中配置的是只运行组XQY，所以结果运行了方法signTest、appCOntTest、pay；GroupTest.xml(2)中配置的是运行组BQ，不运行组XQY，所以结果运行了方法edorConfirmTest、appEdorTest。）

* GroupTest.java 测试类代码

**package** com.sinosoft.demo;

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

**public** **class** GroupTest {

@Test(groups="XQY")

**public** **void** signTest() {

System.***out***.println("签单测试属于新契约！");

}

@Test(groups="BQ")

**public** **void** edorConfirmTest() {

System.***out***.println("保全确认测试属于保全！");

}

@Test(groups="XQY")

**public** **void** appContTest() {

System.***out***.println("投保测试属于新契约！");

}

@Test(groups="BQ")

**public** **void** appEdorTest() {

System.***out***.println("申请保全测试属于保全！");

}

@Test(groups={"XQY","BQ"})

**public** **void** pay() {

System.***out***.println("收费测试既有新契约收费也有保全收费");

}

}

* GroupTest.xml（1） 文件代码是（运行时包含组XQY）：

*<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >*

<suite name="Suite" parallel="false">

<test name="Test">

<groups>

<run>

<include name="XQY"/>

</run>

</groups>

<classes>

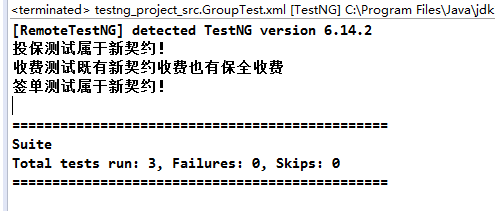
<class name="com.sinosoft.demo.GroupTest"/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

运行结果为：



* GroupTest.xml（2） 文件代码是（运行时包含组BQ，不包含XQY）：

*<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >*

<suite name="Suite" parallel="false">

<test name="Test">

<groups>

<run>

<include name="BQ"/>

<exclude name="XQY"/>

</run>

</groups>

<classes>

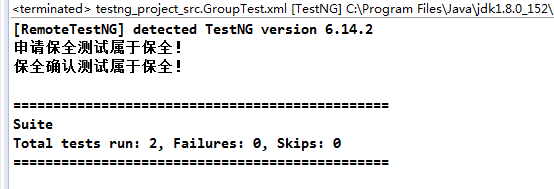
<class name="com.sinosoft.demo.GroupTest"/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

运行结果为：



## 异常测试

场景：测试中，有时候我们期望某些代码抛出异常。

TestNG通过@Test(expectedExceptions)  来判断期待的异常， 并且判断Error Message。案列如下：

（案例说明：测试方法test1是期望抛出IllegalArgumentException异常，并且异常信息为NullPoint；其他测试方法都和期望的不符，所以不通过。）

* ExceptionTest.java 测试类代码

**package** com.sinosoft.demo;

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

**public** **class** ExceptionTest {

@Test(expectedExceptions=IllegalArgumentException.**class**, expectedExceptionsMessageRegExp="NullPoint")

**public** **void** test1() {

System.***out***.println("这个方法抛出异常IllegalArgumentException，异常信息为NullPoint");

**throw** **new** IllegalArgumentException("NullPoint");

}

@Test(expectedExceptions=IllegalArgumentException.**class**, expectedExceptionsMessageRegExp="NullPoint")

**public** **void** test2() {

System.***out***.println("这个方法抛出异常IllegalArgumentException，异常信息为Null");

**throw** **new** IllegalArgumentException("Null");

}

@Test(expectedExceptions=IllegalArgumentException.**class**, expectedExceptionsMessageRegExp="NullPoint")

**public** **void** test3() {

System.***out***.println("这个方法抛出异常NullPointerException，异常信息为Null");

**throw** **new** NullPointerException("NullPoint");

}

@Test(expectedExceptions=IllegalArgumentException.**class**, expectedExceptionsMessageRegExp="NullPoint")

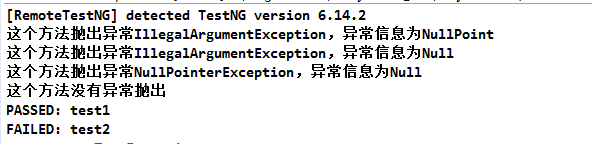
**public** **void** test4() {

System.***out***.println("这个方法没有异常抛出");

}

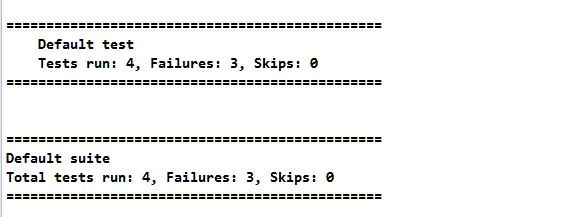
}

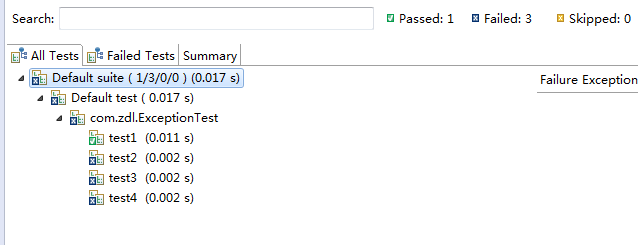
* 运行结果为：











## 并行测试（使用线程实现）

主要对@Test的属性 invocationCount 和 threadPoolSize的应用。invocationCount设置运行次数，threadPoolSize设置线程池的大小，由几个线程运行。

场景：在实际业务场景中，为了测试代码的安全性和性能，通常要在短时间内进行大量的访问测试，比如压测等。测试有多线程的代码时也要用到并行测试。案例如下：

（案例说明：invocationCount=10是测试方法要重复执行10次；threadPoolSize=5是线程池大小为5，就是用5个线程执行；PairingTest.xml的配置也是同样效果。）

* PairingTest.java 测试类代码

**package** com.sinosoft.demo;

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

**public** **class** PairingTest {

@Test(invocationCount=10, threadPoolSize=5)

**public** **void** test() {

System.***err***.println("The current thread id is :"+Thread.*currentThread*().getId());

**int** i=0;

**while**(i<10){

i++;

}

}

}

* PairingTest.xml 代码

*<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >*

<suite name="Suite" parallel="methods" thread-count="5">

<test name="Test">

<classes>

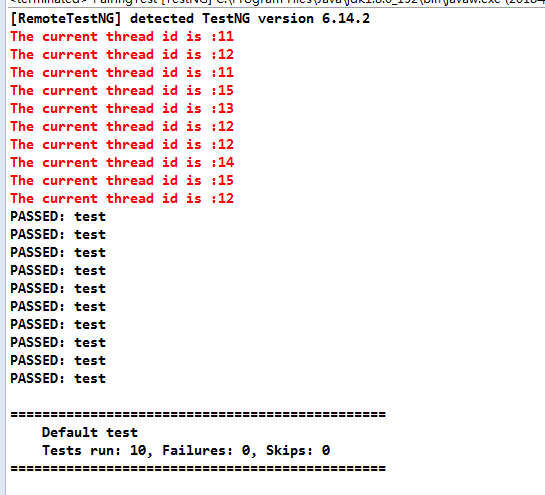
<class name="com.sinosoft.demo.PairingTest"/>

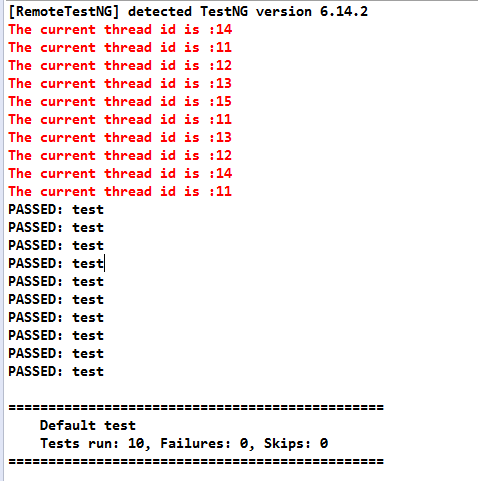
</classes>

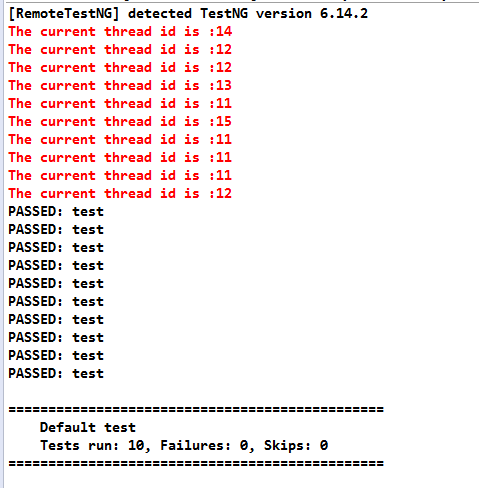
</test> <!-- Test -->

</suite> <!-- Suite -->

* 测试结果为：（多个）







## 延迟数据提供（数据延迟加载）

场景：数据延迟加载是为了防止测试的数据过大，造成内存溢出的情况。

原理是当要使用测试数据中的某条是才加载这条数据，用完后废弃掉，而不是提前创建所有数据。主要是对@DataProvider返回迭代器（Iterator）类型数据的应用。案例如下：

（案例说明：例子还是参数化测试中的核保例子，只是获取测试参数的方式不同，这个是用迭代器Iterator来实现的；就是先去Excel中取一条测试数据，然后测试；完成后再去取第二条数据测试，依次执行知道最后一条数据执行完成。）

* LazyDataTest.java 测试类代码

**package** com.sinosoft.demo;

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

**import** java.util.Iterator;

**import** java.util.Map;

**import** org.testng.annotations.DataProvider;

**public** **class** LazyDataTest {

@Test(dataProvider = "param")

**public** **void** UWTest(Map<String, String> paramMap) {

String edorappno = paramMap.get("edorappno");

String contNo = paramMap.get("contNo");

String riskCode = paramMap.get("riskCode");

**double** prem = Double.*valueOf*(paramMap.get("prem"));

**double** amnt = Double.*valueOf*(paramMap.get("amnt"));

**if**("A0015E".equals(riskCode)){

**if**((prem>0 && prem<100) && (amnt>10000 && amnt<500000)){

System.***out***.println("核保成功！");

} **else** {

System.***out***.println("保全受理号："+edorappno+"，保单号："+contNo+"，产品："+riskCode+"，核保失败！");

}

} **else** **if**("B0018A".equals(riskCode)){

**if**((prem>10 && prem<1000) && (amnt>50000 && amnt<100000)){

System.***out***.println("核保成功！");

} **else** {

System.***out***.println("保全受理号："+edorappno+"，保单号："+contNo+"，产品："+riskCode+"，核保失败！");

}

} **else** {

System.***out***.println("核保失败！没有该产品！");

}

}

@DataProvider(name="param")

**public** Iterator getData() {

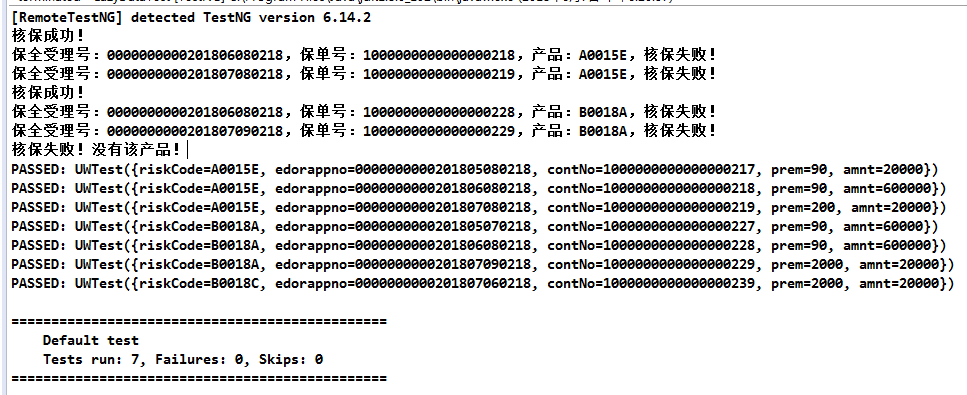
**return** **new** ExcelReadDataUtil("C:\\Users\\Administrator\\Desktop\\data.xls","Sheet1");

}

}

注：ExcelReadDataUtil.java代码和 参数化测试中的 ExcelReadDataUtil.java一样。

* 测试结果为（运行结果看不出来体现在哪，只能体会代码了）：



## 异步测试（使用线程实现）

这里用到了属性timeOut和dependsOnMethods，以及用 .xml 设置线程启动。

场景：适用于 两个测试，在逻辑上有关系，且一个测试依赖于另一个测试的返回结果。例如：发送消息和处理消息以及处理返回的消息；案例如下（这里需要运行AsynchronizationTest.xml）：

（案例说明：AsynchronizationTest.java中方法sendMessage是发送消息；waitForAnswer是等待消息回复，每1s查看下回复结果，最多等待10s，然后完成整个测试；answerMessage是等待3s（实际业务中是处理耗费的时间）回复结果。）

* AsynchronizationTest.java 测试类代码

**package** com.sinosoft.demo;

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

**public** **class** AsynchronizationTest {

**public** **boolean** isAnswer = **false**;

@Test

**public** **void** sendMessage() {

System.***err***.println("The current thread id is :"+Thread.*currentThread*().getId()+"--消息已发送！");

}

@Test(timeOut=10000, dependsOnMethods="sendMessage")

**public** **void** waitForAnswer() {

**while**(!isAnswer){

**try** {

System.***err***.println("The current thread id is :"+Thread.*currentThread*().getId()+"--正在等待消息回复。。。");

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

@Test(dependsOnMethods="sendMessage")

**public** **void** answerMessage() {

**try** {

Thread.*sleep*(3000);//这里模拟实际逻辑处理所消耗的时间

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

isAnswer = **true**;

System.***err***.println("The current thread id is :"+Thread.*currentThread*().getId()+"--消息已回复！");

}

}

* AsynchronizationTest.xml 代码

*<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >*

<suite name="Suite" parallel="methods" thread-count="3">

<test name="Test">

<classes>

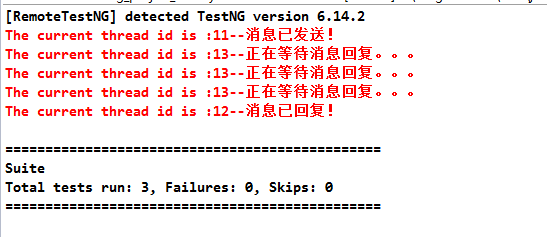
<class name="com.sinosoft.demo.AsynchronizationTest"/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

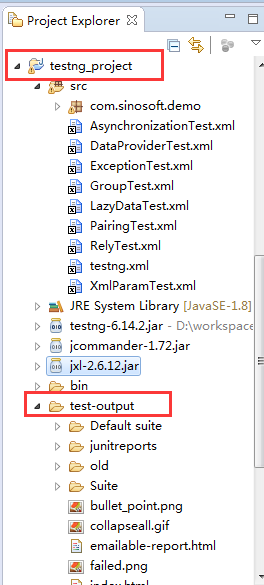
* 测试结果为：



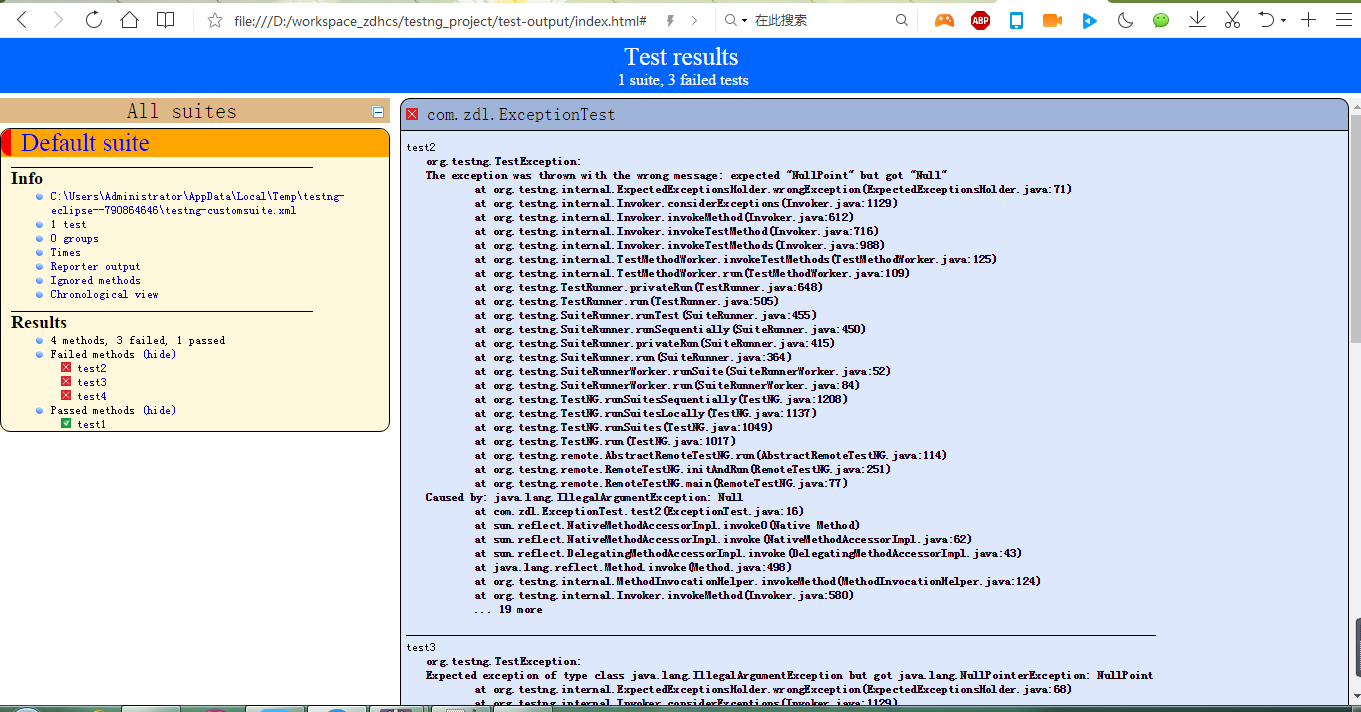
# 七 TestNG结果输出和查看以及失败案例重新测试

（这里的测试用例是上面的异常测试案例）

TestNG会把每次执行的测试结果输出到项目下test-output文件夹下，如图：



查看测试结果可以在浏览器中打开 test-output下的 index.html 文件，效果如下：



对于失败的或者跳过的测试案例，TestNG会在test-output下的testng-failed.xml文件中生成相对应的测试案例，修改后可以重新执行testng-failed.xml，成功的案例不必再执行。testng-failed.xml文件如下：

*<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">*

<suite guice-stage="DEVELOPMENT" name="Failed suite [Default suite]">

<test thread-count="5" name="Default test(failed)">

<classes>

<class name="com.sinosoft.demo.ExceptionTest">

<methods>

<include name="test2"/>

<include name="test3"/>

<include name="test4"/>

</methods>

</class> <!-- com.sinosoft.demo.ExceptionTest -->

</classes>

</test> <!-- Default test(failed) -->

</suite> <!-- Failed suite [Default suite] -->

# 八 附件

data.xls是参数化测试用到的excel数据文件。

导入附件项目：解压文件testng\_project.rar，例如解压到D:/workspace/testng\_project。打开eclipse，选择 File -> Import -> General 下的 Existing Projects into Workspace -> 点击【Next】 -> 在打开的窗口中选择 Select root directory -> 点击【Browse...】选择解压的文件夹，这里是 D:/workspace/testng\_project -> 点击【Finish】等待导入成功。

**注**：该附件是TestNG的demo，在eclipse中导入该demo时要注意，导入完成后要引入bin目录下的jar包。