## 介绍

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（隔离测试一个类）到集成测试（测试由有多个类多个包甚至多个外部框架组成的整个系统，例如运用服务器）。测试的过程的三个典型步骤，注意和junit(4.0)相比，多了一个将测试信息添加到testng.xml文件或者build.xml。测试信息尤其是测试数据不再写死在测试代码中，好处就是修改测试数据时不需要修改代码/编译了，从而有助于将测试人员引入单元测试/集成测试。

编写一个测试的过程有三个典型步骤：

* 编写测试的 业务逻辑并在代码中插入TestNG annotation。
* 将测试信息添加到testng.xml文件或者build.xml中。
* 运行TestNG

### 基本概念：

* suite由xml文件描述。它包含一个或多个测试并被定义为<suite>标签
* test由<test>描述并包含一个或者多个TestNG类
* TestNG类是包含至少一个TestNG annotation的Java类，由<class>标签描述并包含一个或多个测试方法
* 测试方法是源文件中带有@Testd注释的java方法
* 基本概念，相比junit的TestCase/TestSuite，TestNG有suite/test/test method三个级别，即将test/test method明确区分开了。junit中的TestCase将test/test method混合，比较容易让人概念不清晰，尤其是新手。

TestNG测试可以被@BeforeXXX 和 @AfterXXX annotations配置，容许在特定点的前后执行一些java逻辑，这些点上面已经列出。

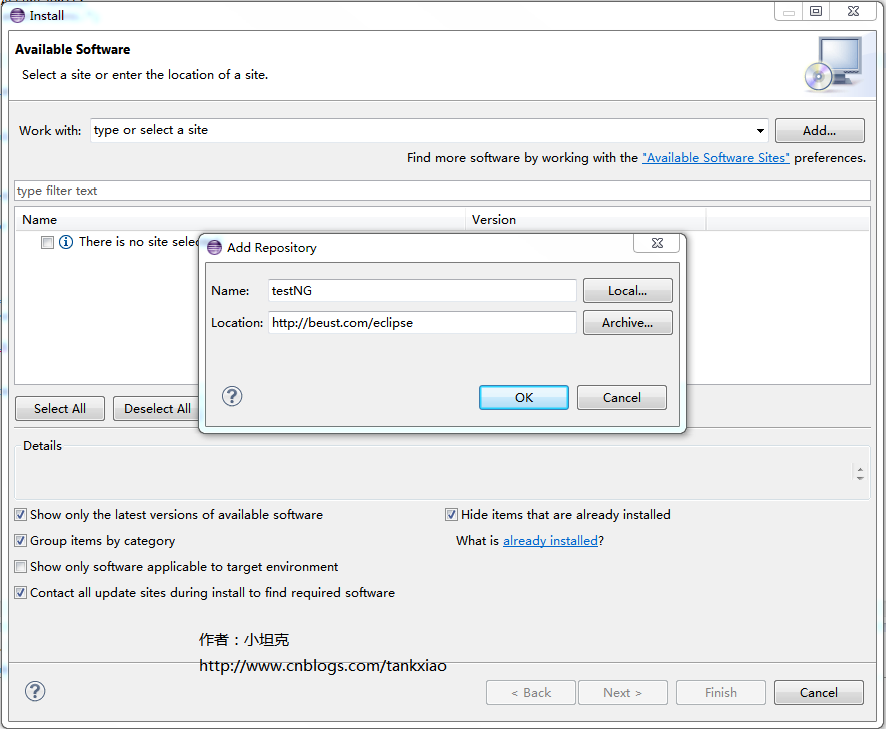
这份手册的剩余部分将讲述以下内容：

* 所有的annotation列表并带有简短说明，为TestNG的多种功能性提供参考, 你可能需要参考为每个annotation提供的代码片段来学习细节。
* testng.xml文件描述，它的语法和如果指定它。
* 多个特性的详细列表和怎样结合annotation和testng.xml来使用它们

### 安装TestNG

#### 在线安装

安装testNG插件打开Eclipse Help ->Install New Software , 然后Add： <http://beust.com/eclipse>



#### 离线安装

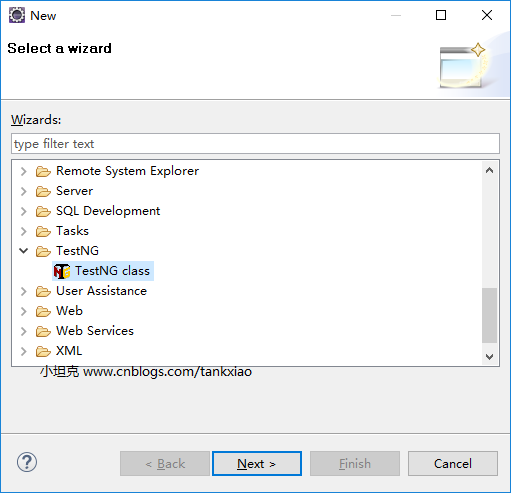
首先下载testng离线包:　 <http://pan.baidu.com/s/1hrEaX6k>

1. 将解压后的文件..\eclipse-testng离线包\features\org.testng.eclipse\_6.9.9.201510270734　 文件夹　放到　　　　　　eclipse--》features目录下

2. 将解压后的文件..\eclipse-testng离线包\plugins\org.testng.eclipse\_6.9.8.201510130443　 文件夹　 放到 eclipse--》plugins目录下

3. 重启eclipse

4. 验证是否安装成功，file-->new-->other-->TestNg



#### Jar包下载地址

下载地址：<http://jcenter.bintray.com/org/testng>

TestNG的官方文档原文请见 <http://testng.org/doc/documentation-main.html>

## Annotation注释

|  |  |  |
| --- | --- | --- |
| **@BeforeSuite @AfterSuite @BeforeTest @AfterTest @BeforeGroups @AfterGroups @BeforeClass @AfterClass @BeforeMethod @AfterMethod** | | **Configuration information for a TestNG class:**   **@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，此方法在<test> 标记将要运行前。 **@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。 |
|  | alwaysRun | 对于每个bufore方法(beforeSuite, beforeTest, beforeTestClass 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。 |
|  | dependsOnGroups | 这个方法依赖的组列表 |
|  | dependsOnMethods | 这个方法依赖的方法列表 |
|  | enabled | 这个类的方法是否激活 |
|  | groups | 这个类或方法所属的分组列表 |
|  | inheritGroups | 如果设置为true，这个方法被属于在类级别被@Test annotation指定的组 |
|  | | |
| **@DataProvider** | | **标记一个方法用于为测试方法提供数据。**  **被注释的方法必须返回Object[][], 其中每个Object[]可以指派为这个测试方法的参数列表。**  **从这个DataProvider接收数据@Test方法需要使用一个和当前注释相同名称的dataProvider名称**  **name 这个DataProvider的名称** |
|  | name | 这个DataProvider的名称。如果未定义，dataprovider将自动命名为方法名。 |
|  | parallel | 如果设置为true，测试中的data provider会同时运行。默认值为false。 |
|  | | |
| **@Factory** | | **标记方法作为一个返回对象的工厂，这些对象将被TestNG用于作为测试类。这个方法必须返回Object[]。** |
|  | | |
| **@Listeners** | | **在一个测试类中定义监听器。** |
|  | value | 一串数组类扩展org.testng.ITestNGListener. |
|  | | |
| **@Parameters** | | **描述如何传递参数给@Test方法。** |
|  | value | 用于填充这个方法的参数的变量列表 |
|  | | |
| **@Test** | | **标记一个类或方法作为测试的一部分** |
|  | alwaysRun | 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。 |
|  | dataProvider | 这个测试方法的data provider的名称 |
|  | dataProviderClass | 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。 |
|  | dependsOnGroups | 当前方法依赖的组列表 |
|  | dependsOnMethods | 当前方法依赖的方法列表 |
|  | description | 当前方法的描述 |
|  | enabled | 当前类的方法/方法是否被激活 |
|  | expectedExceptions | 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. |
|  | groups | 当前类/方法所属的组列表 |
|  | invocationCount | 当前方法被调用的次数 |
|  | invocationTimeOut | 该测试所被调用次数的最大耗时，单位毫秒。如果调用次数未定义忽略本属性。 |
|  | priority | 测试方法优先级，低优先级的优先预订。 |
|  | successPercentage | 当前方法期望的成功率。 |
|  | singleThreaded | 如果设置为true，即使测试集现在运行方法设置为parallel="methods"，测试类中的所有方法运行在同一个线程中。本属性只可以在类一级使用，忽略在方法一级的使用。  注：依次调用本属性（已过时） |
|  | timeOut | 当前方法容许花费的最大时间，单位毫秒。 |
|  | threadPoolSize | 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：

上面是TestNG中用到的annotation列表，从中我们可以看到TestNG提供的一些特性

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

# testng.xml

调用TestNG由几种不同方法：

\* 使用testng.xml文件

\* 从命令行

这节描述testng.xml的格式。

当前testng.xml的DTD文件可以从官方找到：http://testng.org/testng-1.0.dtd.php。

下面是testng.xml文件的一个例子：

|  |
| --- |
| <!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >  <suite name="Suite1" verbose="1" >  <test name="Nopackage" >  <classes>  <class name="NoPackageTest" />  </classes>  </test>  <test name="Regression1" >  <classes>  <class name="test.sample.ParameterSample" />  <class name="test.sample.ParameterTest" />  </classes>  </test>  </suite> |

你可以指定包名替代类名：

|  |
| --- |
| <!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >  <suite name="Suite1" verbose="1" >  <test name="Regression1" >  <packages>  <package name="test.sample" />  </packages>  </test>  </suite> |

在这个例子中，TestNG将在包test.sample中查找所有的类，并只保留带有TestNG annotation的类。

你同样可以指定包含或不包含的组和方法:

|  |
| --- |
| <test name="Regression1">  <groups>  <run>  <exclude name="brokenTests" />  <include name="checkinTests" />  </run>  </groups>    <classes>  <class name="test.IndividualMethodsTest">  <methods>  <include name="testMethod" />  </methods>  </class>  </classes>  </test> |

你同样可以在testng.xml中定义新的组，指定属性的额外详细情况，比如是否并行运行测试，使用多少线程，是否运行junit测试，等等...

请查看DTD文件了解完整的特性列表。

# 运行TestNG

TestNG可以以不同的方式调用：

\* Command line

\* Eclipse

\* IntelliJ's IDEA

1) 命令行

假设你已经将TestNG加入到class path，调用TestNG最简单的方法事下面的：

java org.testng.TestNG testng1.xml [testng2.xml testng3.xml ...]

必须指定最少一个描述你试图测试的TestNG suite的xml文件。另外，下面的命令行参数可以使用：

| 命令行参数列表 | | |
| --- | --- | --- |
| **操作** | **参数** | **说明** |
| -configfailurepolicy | skip|continue | TestNG是否继续执行suite中剩下的测试集，或者跳过@Before\*失败方法。模式习惯跳过。 |
| -d | A directory | The directory where the reports will be generated (defaults to test-output). |
| -dataproviderthreadcount | The default number of threads to use for data providers when running tests in parallel. | This sets the default maximum number of threads to use for data providers when running tests in parallel. It will only take effect if the parallel mode has been selected (for example, with the -parallel option). This can be overridden in the suite definition. |
| -excludegroups | A comma-separated list of groups. | The list of groups you want to be excluded from this run. |
| -groups | A comma-separated list of groups. | The list of groups you want to run (e.g. "windows,linux,regression"). |
| -listener | A comma-separated list of Java classes that can be found on your classpath. | Lets you specify your own test listeners. The classes need to implement [org.testng.ITestListener](http://testng.org/javadocs/org/testng/ITestListener.html) |
| -methods | A comma separated list of fully qualified class name and method. For examplecom.example.Foo.f1,com.example.Bar.f2. | Lets you specify individual methods to run. |
| -methodselectors | A comma-separated list of Java classes and method priorities that define method selectors. | Lets you specify method selectors on the command line. For example: com.example.Selector1:3,com.example.Selector2:2 |
| -parallel | methods|tests|classes | If specified, sets the default mechanism used to determine how to use parallel threads when running tests. If not set, default mechanism is not to use parallel threads at all. This can be overridden in the suite definition. |
| -reporter | The extended configuration for a custom report listener. | Similar to the -listener option, except that it allows the configuration of JavaBeans-style properties on the reporter instance.  Example: -reporter com.test.MyReporter:methodFilter=\*insert\*,enableFiltering=true  You can have as many occurences of this option, one for each reporter that needs to be added. |
| -sourcedir | A semi-colon separated list of directories. | The directories where your javadoc annotated test sources are. This option is only necessary if you are using javadoc type annotations. (e.g. "src/test" or "src/test/org/testng/eclipse-plugin;src/test/org/testng/testng"). |
| -suitename | The default name to use for a test suite. | This specifies the suite name for a test suite defined on the command line. This option is ignored if the suite.xml file or the source code specifies a different suite name. It is possible to create a suite name with spaces in it if you surround it with double-quotes "like this". |
| -testclass | A comma-separated list of classes that can be found in your classpath. | A list of class files separated by commas (e.g. "org.foo.Test1,org.foo.test2"). |
| -testjar | A jar file. | Specifies a jar file that contains test classes. If a testng.xml file is found at the root of that jar file, it will be used, otherwise, all the test classes found in this jar file will be considered test classes. |
| -testname | The default name to use for a test. | This specifies the name for a test defined on the command line. This option is ignored if the suite.xml file or the source code specifies a different test name. It is possible to create a test name with spaces in it if you surround it with double-quotes "like this". |
| -testnames | A comma separated list of test names. | Only tests defined in a <test> tag matching one of these names will be run. |
| -testrunfactory | A Java classes that can be found on your classpath. | Lets you specify your own test runners. The class needs to implement [org.testng.ITestRunnerFactory](http://testng.org/javadocs/org/testng/ITestRunnerFactory.html). |
| -threadcount | The default number of threads to use when running tests in parallel. | This sets the default maximum number of threads to use for running tests in parallel. It will only take effect if the parallel mode has been selected (for example, with the -parallel option). This can be overridden in the suite definition. |
| -xmlpathinjar | The path of the XML file inside the jar file. | This attribute should contain the path to a valid XML file inside the test jar (e.g. "resources/testng.xml"). The default is "testng.xml", which means a file called "testng.xml" at the root of the jar file. This option will be ignored unless -testjar is specified. |

# Test methods, Test classes and Test groups

## Test groups

TestNG容许执行复杂的测试方法分组。不仅可以申明方法属于组，而且可以指定分组包含其他分组。

然后TestNG可以被调用，并被要求包含某些分组和排除其他的分组。

这将提供怎样划分测试的最大弹性，并且如果想运行两个不同的测试装置不需要重新编译。

例如，非常普遍的需要至少两个种类的测试

\* Check-in tests. 这些测试将在提交新代码之前运行. 它们典型的被要求快速而且仅仅确认没有基础功能被破坏。

\* Functional tests. 这些测试将覆盖所有的软件功能，并且必须运行至少1天，尽管理想的是连续运行.

代表性的，check-in测试是功能性测试的子集。TestNG容许用非常直接的方式说明这个。

例如： 可以这样构造测试，申明完整的测试类属于"functest"组，另外两个方法属于组"checkintest":

[java] view plain copy print?

public class Test1 {

@Test(groups = { "functest", "checkintest" })

public void testMethod1() {

}

@Test(groups = {"functest", "checkintest"} )

public void testMethod2() {

}

@Test(groups = { "functest" })

public void testMethod3() {

}

}

调用TestNG，使用

[xhtml] view plain copy print?

<test name="Test1">

<groups>

<run>

<include name="functest"/>

</run>

</groups>

<classes>

<class name="example1.Test1"/>

</classes>

</test>

将运行在类中的所有测试方法，如果使用checkintest调用则将只运行testMethod1()和testMethod2().

这里由其他例子，这次使用正则表达式。假设某些测试方法可能无法在Linux上运行，测试将是类似如此：

[java] view plain copy print?

@Test

public class Test1 {

@Test(groups = { "windows.checkintest" })

public void testWindowsOnly() {

}

@Test(groups = {"linux.checkintest"} )

public void testLinuxOnly() {

}

@Test(groups = { "windows.functest" )

public void testWindowsToo() {

}

}

你可以使用下面的testng.xml文件只启动Windows方法：

[xhtml] view plain copy print?

<test name="Test1">

<groups>

<run>

<include name="windows.\*"/>

</run>

</groups>

<classes>

<class name="example1.Test1"/>

</classes>

</test>

注意：TestNG使用正则表达，而不是wildmats。注意这个差别。

Method groups同样可以包含或排除个别方法：

[xhtml] view plain copy print?

<test name="Test1">

<classes>

<class name="example1.Test1">

<methods>

<include name=".\*enabledTestMethod.\*"/>

<exclude name=".\*brokenTestMethod.\*"/>

</methods>

</class>

</classes>

</test>

这在需要使莫个单独的方法失效而不想重新编译时非常方便，但是不建议太多的使用这个机制，因为这将可能破坏你的测试框架 如果你开始重构你的java代码（标签中使用的正则表达式可能不再匹配你的方法）

## Groups of groups

"functest" itself will contain the groups "windows" and "linux" while "checkintest will only contain "windows". Here is how you would define this in your property file:

组可以包含其他组。这些组被称为"MetaGroups"。例如，你可能想定义一个"all"组，包括"checkintest"和"functest"。"functest"自身将包含组 "windows" 和 "linux"，而"checkintest"将包含"windows".

|  |
| --- |
| [xhtml] view plain copy print?  <test name="Regression1">  <groups>  <define name="functest">  <include name="windows"/>  <include name="linux"/>  </define>    <define name="all">  <include name="functest"/>  <include name="checkintest"/>  </define>    <run>  <include name="all"/>  </run>  </groups>    <classes>  <class name="test.sample.Test1"/>  </classes>  </test> |

## Exclusion groups

TestNG 容许包含组也容许排除组.

例如，当由因为最近的修改而临时破坏的测试而又没有时间去修复它们时非常有用。无论如何，你想要干净的运行功能性测试，因此你想要是这些测试失效，但是记住它们重新被激活。

一个简单的解决这个问题的方法是创建一个称为"broken"的组并让这些测试方法归属它。例如，在上面的例子中，我知道testMethod2() 现在被破坏了，所有我想关闭它：

|  |
| --- |
| [java] view plain copy print?  @Test(groups = {"checkintest", "broken"} )  public void testMethod2() {  } |

现在我所想要做的只是在运行中排除这个组：

|  |
| --- |
| [xhtml] view plain copy print?  <test name="Simple example">  <groups>  <run>  <include name="checkintest"/>  <exclude name="broken"/>  </run>  </groups>  <classes>  <class name="example1.Test1"/>  </classes>  </test> |

用这种方法，我将得到一个干净的测试运行，同时记录了那些被破坏并想要后续修复的测试。

注意：你也可以通过使用在@Test and @Before/After annotations上的"enabled"属性在个体的层面上关闭测试，

## Partial groups

你可以在类的级别上定义组，然后在方法的层次上添加组：

|  |
| --- |
| [java] view plain copy print?  @Test(groups = { "checkin-test" })  public class All {  @Test(groups = { "func-test" )  public void method1() { ... }  public void method2() { ... }  } |

在这个类中，method2() 属于组"checkin-test"，在类的级别定义。而method1() 同时属于 "checkin-test" 和 "func-test".

## 参数化

测试方法不要求是无参数的。你可以在每个测试方法上使用任意数量的参数，并指示testNG传递正确的参数。

有两种方式用于设置参数：使用testng.xml或者编程式。

### 参数化testng.xml

如果你要为你的参数使用简单值，你可以在你的testng.xml中明确指定：

|  |
| --- |
| [java] view plain copy print?  @Parameters({ "first-name" })  @Test  public void testSingleString(String firstName) {  System.out.println("Invoked testString " + firstName);  assert "Cedric".equals(firstName);  } |

在这个代码中，我们明确指定java方法的参数“firstName”应该接收名为“first-name”xml参数的值。这个xml参数在testng.xml中定义：

|  |
| --- |
| <suite name="My suite">  <parameter name="first-name" value="Cedric"/>  <test name="Simple example">  <!-- --> |

同样的方法可以用于注解@Before/After和@Factory:

|  |
| --- |
| [java] view plain copy print?  @Parameters({ "datasource", "jdbcDriver" })  @BeforeMethod  public void beforeTest(String ds, String driver) {  m\_dataSource = ds; // look up the value of datasource  m\_jdbcDriver = driver;  } |

这次，两个java参数ds和driver将分别接收被设置给属性datasource和jdbc-driver的值。

参数可以通过可选注解来声明为可选：

@Parameters("db")

@Test

public void testNonExistentParameter(@Optional("MySQL") String db) { }

如果在testng.xml文件中没有找到名为"db"的参数，测试方法将接受在@Optional注解中指定的默认值："mysql"

@Parameters 注解可以在下面位置使用：

\* 在任何有@Test, @Before/After或者@Factory注解的方法上

\* 在测试类的最多一个构造函数上。这种情况下，当TestNG需要实例化测试类时，他将调用这个特别的带有初始化为testng.xml中指定的值的参数的构造函数。这个特性可以被用于初始化类内部的值域为将用于测试方法的值。

注意:

\* xml参数被以在注解中出现的相同顺序映射到java参数，如果参数数量不匹配testNG将发生错误。

\* 参数是有范围的。在testng.xml中，你可以在<suite>标签或者<test>标签下声明参数。如果两个参数同名，在<test>标签下定义的参数优先。非常适用于这样的场合：需要指定一个应用于所有测试的参数，但是又希望在特定测试用覆盖它的值。

### 参数化 DataProviders

在testng.xml中指定参数，对于以下情况是不够的：

\* 不使用testng.xml

\* 需要传递复杂参数，或者参数需要从java中创建（复杂对象，从属性文件或者数据库中读取的对象）在这种情况下，你可以使用Data Provider来提供你测试需要的数值。Data Provider是类中的一个返回对象数组的数组的方法。这个方法带有@DataProvider注解：

|  |
| --- |
| //这个方法将提供数据给任何声明它的Data Provider名为"test1"的测试方法  [java] view plain copy print?  @DataProvider(name = "test1")  public Object[][] createData1() {  return new Object[][] {  { "Cedric", new Integer(36) },  { "Anne", new Integer(37)},  };  } |

//这个方法声明它的数据将由名为"test1"的Data Provider提供

|  |
| --- |
| [java] view plain copy print?  @Test(dataProvider = "test1")  public void verifyData1(String n1, Integer n2) {  System.out.println(n1 + " " + n2);  } |

将打印

Cedric 36

Anne 37

@Test方法用dataProvider属性来指定它的Data Provider。这个名字必须符合同一个类中用@DataProvider(name="...")注解的方法，它们要使用同一个匹配的名字。

默认，将在当前类或者它的基类中查找data provider。如果你想将data provider放置到另一个类中，需要将这个data provider方法设置为静态方法，并在dataProviderClass属性中指定在哪个类中可以找到这个方法。

|  |
| --- |
| [java] view plain copy print?  public static class StaticProvider {  @DataProvider(name = "create")  public static Object[][] createData() {  return new Object[][] {  new Object[] { new Integer(42) }  }  }  }  public class MyTest {  @Test(dataProvider = "create", dataProviderClass = StaticProvider.class)  public void test(Integer n) {  //  }  } |

Data Provider方法将返回下面两个类型中的一种：

The Data Provider method can return one of the following two types:

\* 对象数组的数组(Object[][]) ，外围数据的大小是测试方法将被调用的次数，而内层数组的大小和类型必须和测试方法的参数列表匹配。如同上面举例说明的。

\* <Object[]>的Iterator,和Object[][]的唯一差别在于Iterator容许延迟创建测试数据。testNG将一个接一个的调用iterator，再用iterator返回的参数调用测试方法。如果有很多参数集合需要传递给方法而又不想一开始就创建所有参数，会非常有用。

如果将测试方法的第一个参数申明为java.lang.reflect.Method，TestNG将使用这个第一个参数来传递当前测试方法。当多个测试方法使用同一个@DataProvider而需要依当前申请数据的方法而定来返回不同值时特别有用。

举例说明，下面的代码在@DataProvider中打印测试方法的名字:

[java] view plain copy print?

@DataProvider(name = "dp")

public Object[][] createData(Method m) {

System.out.println(m.getName()); // print test method name

return new Object[][] { new Object[] { "Cedric" }};

}

@Test(dataProvider = "dp")

public void test1(String s) {

}

@Test(dataProvider = "dp")

public void test2(String s) {

}

将会显示:

test1

test2

### Dependent methods

有些时候，你需要你的测试方法按照一个特定的顺序被调用。这非常有用，比如：

\* 在运行更多测试方法前确认特定数量的测试方法调用完成并且成功

\* 初始化测试并希望这个初始化方法也作为测试方法（被标记为@Before/After的方法将不作为最终报告的一部分）

为了做到这点，需要使用@Test注解的dependsOnMethods属性或者dependsOnGroups属性。

有两种依赖：

\* 强依赖。在运行你的测试方法前所有依赖方法必须运行并且成功。哪怕有一个依赖方法失败，测试方法都不会被调用，在报告中将被标记为SKIP。

\* 软依赖。测试方法在依赖方法运行后总是会被运行，即使某些依赖方法失败。对于只想确认测试方法是按照特定顺序运行，而测试方法并不真正依赖其他方法是否成功的情况，非常有用。软依赖通过在@Test注解中增加"alwaysRun=true"来实现。

这里有一个强依赖的例子：

@Test

public void serverStartedOk() {}

@Test(dependsOnMethods = { "serverStartedOk" })

public void method1() {}

在这个例子中，method1()被申明依赖于方法serverStartedOk()，这保证serverStartedOk() 方法将总是首先被调用。

也可以让方法依赖于完整的测试组：

@Test(groups = { "init" })

public void serverStartedOk() {}

@Test(groups = { "init" })

public void initEnvironment() {}

@Test(dependsOnGroups = { "init.\* })

public void method1() {}

在这里例子中，method1()被申明依赖于任何匹配正则表达式"init.\*"的组，这保证了方法serverStartedOk()和initEnvironment()总是在method1()前被调用。

注意：前面说明说，在测试运行期间，属于同一个组的方法的调用顺序并不保证相同。如果一个方法的依赖失败了，而且是强依赖(默认alwaysRun=false)，这个方法将不被标记为FAIL而是SKIP。被跳过的方法在最终的报告中报告（在HTML中用红和绿之外的其他颜色），这很重要，因为被跳过的方法并不一定是失败。

dependsOnGroups和dependsOnMethods都接受正则表达式作为参数。对于dependsOnMethods, 如果你依赖的方法巧合有多个重载的版本，所有装载的方法都将被调用。如果你只想调用重载的方法中的一个，请使用dependsOnGroups。

有关方法依赖的更高级的例子，请参考本文档，将使用继承来提供一个优雅的解决方案来处理多重依赖的问题。

### Factories

工厂类容许你动态创建测试案例。例如，想象你需要创建一个测试方法，访问一个web站点上的页面很多次，而你希望用不同的值来调用它：

[java] view plain copy print?

public class TestWebServer {

@Test(parameters = { "number-of-times" })

public void accessPage(int numberOfTimes) {

while (numberOfTimes-- > 0) {

// access the web page

}

}

}

testng.xml:

[xhtml] view plain copy print?

<test name="T1">

<parameter name="number-of-times" value="10"/>

<class name= "TestWebServer" />

</test>

<test name="T2">

<parameter name="number-of-times" value="20"/>

<class name= "TestWebServer"/>

</test>

<test name="T3">

<parameter name="number-of-times" value="30"/>

<class name= "TestWebServer"/>

</test>

这种方式很快就会变的难于管理，所以作为替换品，你可以使用factory：

[java] view plain copy print?

public class WebTestFactory {

@Factory

public Object[] createInstances() {

Object[] result = new Object[10];

for (int i = 0; i < 10; i++) {

result[i] = new WebTest(i \* 10);

return result;

}

}

而新的测试类是这样：

[java] view plain copy print?

public class WebTest {

private int m\_numberOfTimes;

public WebTest(int numberOfTimes) {

m\_numberOfTimes = numberOfTimes;

}

@Test

public void testServer() {

for (int i = 0; i < m\_numberOfTimes; i++) {

// access the web page

}

}

}

testng.xml只需要引用简单引用这个包含factory方法的类，因为测试实例将在运行时被创建。

<class name="WebTestFactory" />

工厂类将像@Test和@Before/After一样接收参数，必须返回Object[]。返回的对象可以是任何类（不一定要求是和factory类一样），并且他们甚至都不需要包含TestNG的注解（这种情况下他们将被testNG忽略）。

### Class level annotations

@Test注解可以放置在类上：

[java] view plain copy print?

@Test

public class Test1 {

public void test1() {

}

public void test2() {

}

}

类级别注解的效果是将这个类的所有的public方法都变成测试方法，即使他们没有被注解。还可以在需要增加属性的方法上重复@Test注解。

例如:

[java] view plain copy print?

@Test

public class Test1 {

public void test1() {

}

@Test(groups = "g1")

public void test2() {

}

}

将方法test1()和test2()都变成测试方法，但是在此之上，test2()现在属于组"g1".

### Parallel running and time-outs

可以通过使用parallel属性要求TestNG在单独的线程中运行测试。这个属性可以在两个值中取其一：

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

<suite name="My suite" parallel="tests" thread-count="5">

\* parallel="methods": TestNG将在单独的线程中运行测试方法，除了那些依赖其他测试方法的，这些将在同一个线程中运行，以保证他们的执行顺序。

\* parallel="tests": TestNG将在一个线程中运行所有在同一个<test>标签中的测试方法，但是每个<test>标签将在单独的线程中运行。这种方式容许把所有不是线程安全的类分组到相同的<test>标签中，保证他们将在相同的线程中运行，有利于TestNG使用尽可能多的线程来运行测试。

此外，thread-count属性容许指定运行时将分配多少线程。

注意：@Test的属性timeOut在并发和非并发模型下都可以工作。

也可以指定@Test方法在不同的线程中被调用。可以使用threadPoolSize属性来实现这样的结果：

@Test(threadPoolSize = 3, invocationCount = 10, timeOut = 10000)

public void testServer() {

}

在这个例子中，方法testServer将被3个不同线程调用10次。此外，10秒种的time-out属性保证任何线程都不会长时间阻塞。

### Rerunning failed tests

套件中的测试失败时，每次testNG都会在输出目录中创建一个名为testng-failed.xml的文件。这个xml文件包含只重新运行这些失败的测试方法的必要信息，容许只运行这些失败的测试而不必运行全部测试。因此，一种典型的情况将是这样：

java -classpath testng.jar;%CLASSPATH% org.testng.TestNG -d test-outputs testng.xml

java -classpath testng.jar;%CLASSPATH% org.testng.TestNG -d test-outputs test-outputs/testng-failed.xml

注意testng-failed.xml将包含所有必要的依赖方法，所以可以保证运行失败的方法而不运行任何被跳过的(失败)方法。

### JUnit tests

TestNG可以运行junit测试。所需要的只是在testng.classNames属性中指定junit测试类，并设置testng.junit属性为true。

<test name="Test1" junit="true">

<classes>

<!-- -->

这种情况下TestNG的行为类似jnit：

\* 类中所有以test\*开头的方法将被运行。

\* 如果测试类中有方法setUp(), 将在每次测试方法调用前被执行。

\* 如果测试类中有方法tearDown(),将在每次测试方法调用后被执行。

### Running TestNG programmatically

在自己的程序中调用testNG也很简单:

TestListenerAdapter tla = new TestListenerAdapter();

TestNG testng = new TestNG();

testng.setTestClasses(new Class[] { Run2.class });

testng.addListener(tla);

testng.run();

这个范例创建了一个TestNG对象并运行测试类Run2。还增加了一个TestListener。你可以使用适配器类org.testng.TestListenerAdapter或自己实现org.testng.ITestListener。这个接口包含多个回调方法，使得可以追踪测试的开始，成功，失败等等。

类似的，可以使用testng.xml文件调用TestNG或者自己创建一个虚拟的testng.xml文件。为了做到这点，需要使用org.testng.xml包的类：XmlClass, XmlTest, 等等。每个类对应他们xml标签。

例如，假设你想创建下面的虚拟文件：

<suite name="TmpSuite" >

<test name="TmpTest" >

<classes>

<class name="test.failures.Child" />

<classes>

</test>

</suite>

你将使用下面的代码：

XmlSuite suite = new XmlSuite();

suite.setName("TmpSuite");

XmlTest test = new XmlTest(suite);

test.setName("TmpTest");

List<XmlClass> classes = new ArrayList<XmlClass>();

classes.add(new XmlClass("test.failures.Child"));

test.setXmlClasses(classes) ;

然后你可以将XmlSuite传递给TestNG：

List<XmlSuite> suites = new ArrayList<XmlSuite>();

suites.add(suite);

TestNG tng = new TestNG();

tng.setXmlSuites(suites);

tng.run();

完整的API请参考javadoc。

### BeanShell and advanced group selection

如果testng.xml中的<include>和<exclude>标签还不足够满足你的需要，你可以使用BeanShell表达式来决定是否需要将一个特定的测试方法包含在测试操作中。只需要在<test>标签下指定这个表达式：

[xhtml] view plain copy print?

<test name="BeanShell test">

<method-selectors>

<method-selector>

<mce:script language="beanshell"><!--

groups.containsKey("test1")

--></mce:script>

</method-selector>

</method-selectors>

<!-- -->

当发现testng.xml中有<script>标签，TestNG将忽略当前<test>标签中的以后的组和方法的<include>和<exclude>标签：BeanShell表达式将是决定一个测试方法是否包含的唯一方法。

这里有一些BeanShell脚本的额外信息：

\* 必须返回boolean值。除了这个约束，任何有效的BeanShell代码都被容许.(例如，你可能想在工作日返回true而在周末返回false，这将容许你更加日期不同差异性的运行测试。

\* TestNG为了便利定义了以下变量：

java.lang.reflect.Method method: 当前测试方法

org.testng.ITestNGMethod testngMethod: 当前测试方法的描述

java.util.Map<String, String> groups: 当前测试方法所属组的Map

\* 你可能需要在你的表达式前后增加CDATA声明（如上面所示）以避免讨厌的xml转义字符

# Junit与TestNG对比之Assert类

比较的版本是Junit 4.11 vs TestNG 6.8.5

## 断言方法中的message参数位置不同

Junit位于assert方法的第一个位置；

TestNG的位于assert方法的最后一个，功能上没有差异，只是会影响使用习惯。

## assertThat方法（[Harmcrest](https://github.com/hamcrest/JavaHamcrest)）

Junit支持；

TestNG并不支持此方法。

## 对于数组的断言

Junit通过assertArrayEquals专门针对各种类型数字进行断言，支持多种类型的数组参数；

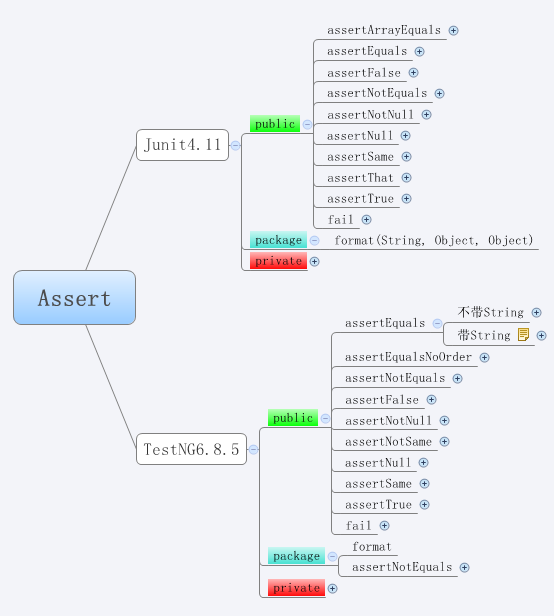
TestNG的assertArrayEquals方法，仅接受一种参数是Object。

## 对集合类型的断言

Junit没有包含对集合类型的断言方法；

TestNG在assertEquals方法中支持Map、Collection、Iterator。

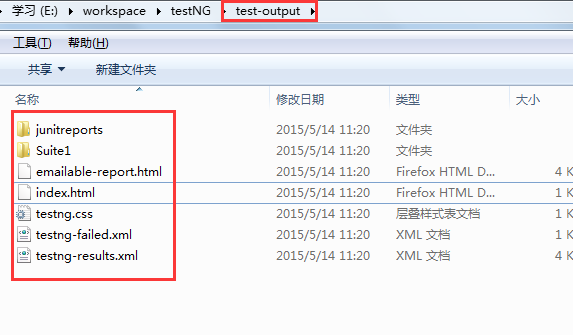
附图：Junit与TestNG对比



# [testNG之测试报告](http://www.cnblogs.com/yuan-yuan/p/4503524.html)

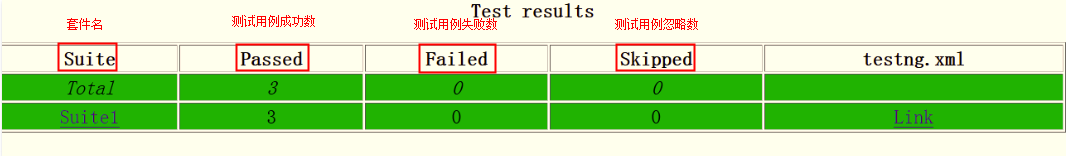
**测试报告**

　　执行完测试用例之后，会在项目的test-output（默认目录）下生成测试报告

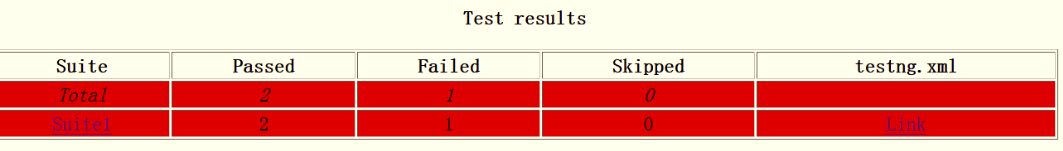


打开index.html文件，测试结果摘要，包括：套件名、测试用例成功数、测试用例失败数、测试用例忽略数和testng.xml文件

测试用例都成功的话，测试结果以绿底标志：



测试用例有失败的话，测试结果以红底标志：



点击"Link"链接，可以查看testng.xml文件的内容：

[复制代码](javascript:void(0);)

<?xml version="1.0" encoding="UTF-8"?>

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

<suite verbose="0" name="Suite1" parallel="false">

<test name="testNG1" preserve-order="false">

<classes>

<class name="testNG1"/>

<class name="testNG2"/>

<class name="testNG3"/>

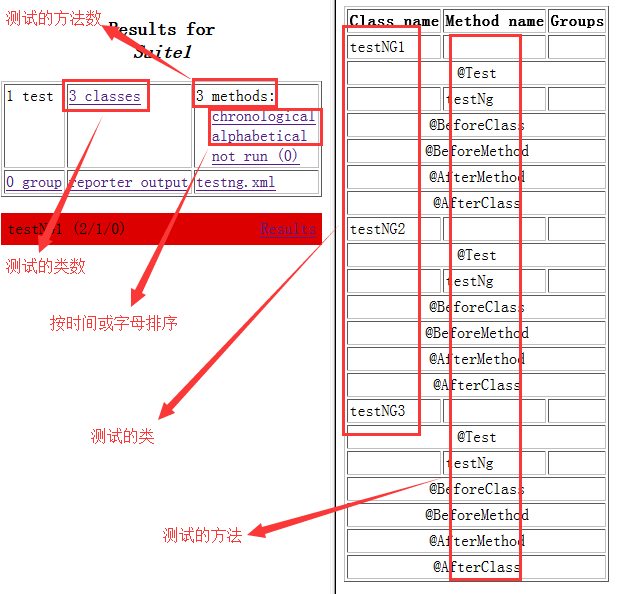
</classes>

</test>

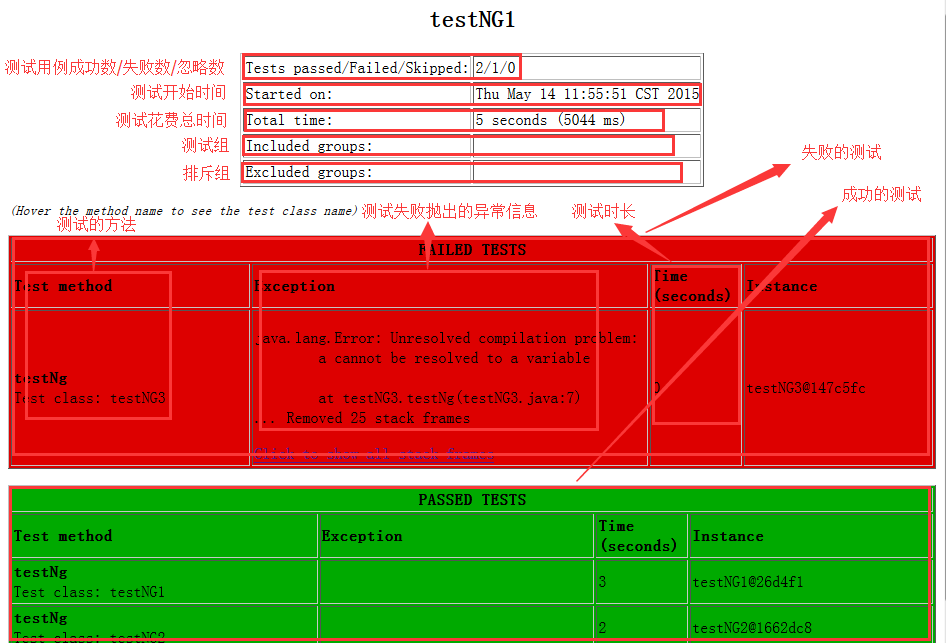
</suite>

[复制代码](javascript:void(0);)

点击"Suite1"链接，可以查看更详细的信息：



点击"Results"链接，可以查看详细的测试结果：



**testng-xslt**

testNG自带生成的测试报告不太美观，可以使用testng-xslt进行美化：

1、下载testng-xslt包，<http://testng-xslt.googlecode.com/files/testng-xslt-1.1.zip>

　  现在google可能访问不了，可到我的网盘里下载http://pan.baidu.com/s/1bn4hR9H

2、复制testng-results.xsl（testng-xslt-1.1\src\main\resources）文件到test-output目录下

3、复制saxon-8.7.jar（testng-xslt-1.1\lib）文件到project的lib目录下（没有的话，自己新建）

4、安装ant，http://mirror.bit.edu.cn/apache/ant/binaries/apache-ant-1.9.4-bin.zip

5、配置好环境变量PATH

6、在project目录下，新建build.xml文件，内容如下：

[复制代码](javascript:void(0);)

<project name="testNG" basedir="." >

<property name="lib.dir" value="lib" />

<path id="test.classpath" >

<!-- adding the saxon jar to your classpath -->

<fileset dir="${lib.dir}" includes="\*.jar" />

</path>

<target name="transform" >

<!-- 需要根据自己的环境进行配置（将E:/workspace/testNG/替换成你自己project的目录） -->

<xslt in="E:/workspace/testNG/test-output/testng-results.xml" style="E:/workspace/testNG/test-output/testng-results.xsl"

out="E:/workspace/testNG/test-output/index1.html" >

<!-- 需要根据自己的环境进行配置（将E:/workspace/testNG/替换成你自己project的目录） -->

<param name="testNgXslt.outputDir" expression="E:/workspace/testNG/test-output/" />

<classpath refid="test.classpath" />

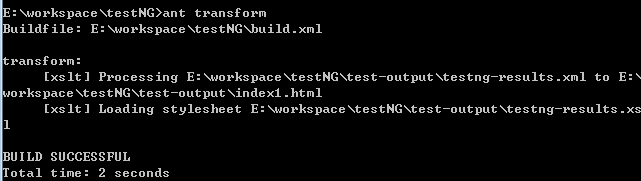
</xslt>

</target>

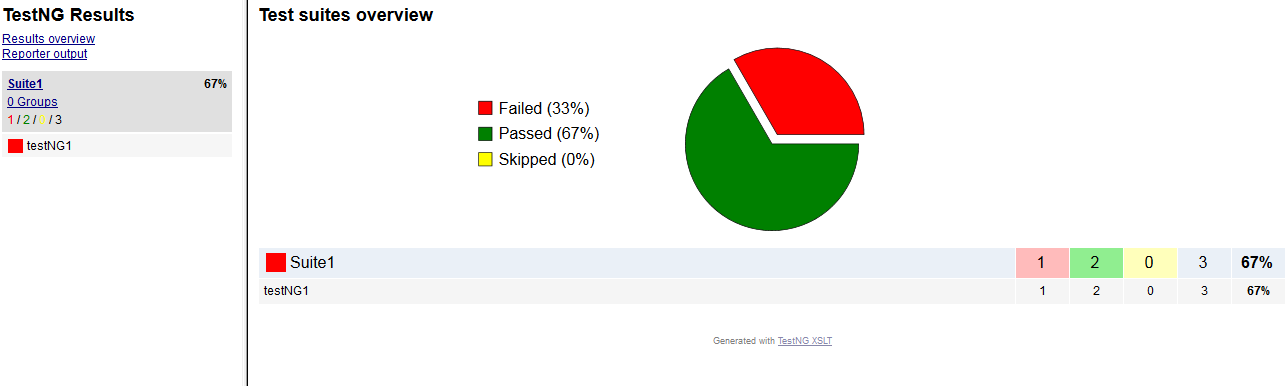
</project>

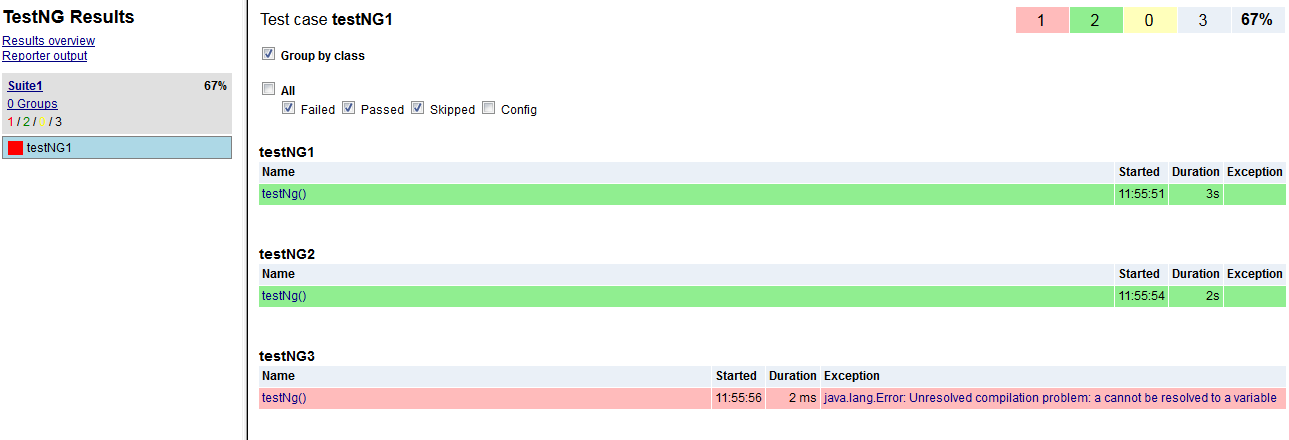
[复制代码](javascript:void(0);)

7、在cmd里，切换到project的目录，执行ant transform：



8、到配置的路径下，打开生成的文件index1.html，以图形化的界面展示测试结果：





信息都差不多，只是页面优化，更加美观了

# 问题：

## Cannot find class in classpath

org.testng.TestNGException:   
Cannot find class in classpath: TestNGExample  
 at org.testng.xml.XmlClass.loadClass(XmlClass.[Java](http://lib.csdn.net/base/17):81)  
 at org.testng.xml.XmlClass.init(XmlClass.java:73)  
 at org.testng.xml.XmlClass.<init>(XmlClass.java:59)  
 at org.testng.xml.TestNGContentHandler.startElement(TestNGContentHandler.java:543)  
 at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.startElement(AbstractSAXParser.java:506)  
 at com.sun.org.apache.xerces.internal.parsers.AbstractXMLDocumentParser.emptyElement(AbstractXMLDocumentParser.java:182)  
 at com.sun.org.apache.xerces.internal.impl.dtd.XMLDTDValidator.emptyElement(XMLDTDValidator.java:766)  
 at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanStartElement(XMLDocumentFragmentScannerImpl.java:1347)  
 at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl$FragmentContentDriver.next(XMLDocumentFragmentScannerImpl.java:2768)  
 at com.sun.org.apache.xerces.internal.impl.XMLDocumentScannerImpl.next(XMLDocumentScannerImpl.java:606)  
 at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument(XMLDocumentFragmentScannerImpl.java:511)  
 at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(XML11Configuration.java:846)  
 at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(XML11Configuration.java:775)  
 at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(XMLParser.java:123)  
 at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse(AbstractSAXParser.java:1210)  
 at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.parse(SAXParserImpl.java:628)  
 at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl.parse(SAXParserImpl.java:332)  
 at javax.xml.parsers.SAXParser.parse(SAXParser.java:195)  
 at org.testng.xml.XMLParser.parse(XMLParser.java:39)  
 at org.testng.xml.SuiteXmlParser.parse(SuiteXmlParser.java:17)  
 at org.testng.xml.SuiteXmlParser.parse(SuiteXmlParser.java:10)  
 at org.testng.xml.Parser.parse(Parser.java:168)  
 at org.testng.TestNG.initializeSuitesAndJarFile(TestNG.java:311)  
 at org.testng.remote.RemoteTestNG.run(RemoteTestNG.java:88)  
 at org.testng.remote.RemoteTestNG.initAndRun(RemoteTestNG.java:204)  
 at org.testng.remote.RemoteTestNG.main(RemoteTestNG.java:175)

  此问题原因：没有配置环境变量

  此问题解决方法：

### 设置TestNG的环境

设置TESTNG\_HOME环境变量指向TestNG的jar 存放在您的机器上的基本目录位置。假设，我们已经储存了testng-6.8.jar， TestNG各种操作系统上的文件夹如下：

|  |  |
| --- | --- |
| **OS** | **输出** |
| Windows | Set the environment variable TESTNG\_HOME to C:\TESTNG |
| Linux | export TESTNG\_HOME=/usr/local/TESTNG |
| Mac | export TESTNG\_HOME=/Library/TESTNG |

### 设置CLASSPATH变量

设置CLASSPATH环境变量指向TestNG的jar文件位置。假设，我们已经储存了testng-6.8.jar, TestNG在各种操作系统上的文件夹如下：

|  |  |
| --- | --- |
| **OS** | **输出** |
| Windows | 设置环境变量 CLASSPATH 为 %CLASSPATH%;%TESTNG\_HOME%\testng-6.8.jar; |
| Linux | export CLASSPATH=$CLASSPATH:$TESTNG\_HOME/testng-6.8.jar: |
| Mac | export CLASSPATH=$CLASSPATH:$TESTNG\_HOME/testng-6.8.jar: |

# 附件

## testng-1.0.dtd

原文地址：<http://testng.org/testng-1.0.dtd.php>

|  |
| --- |
| <!—  本内容是DTD的主要部分的快速预览。更多信息  <a href="http://testng.org">main web site</a>.    A <b>suite</b> is made of <b>tests</b> and <b>parameters</b>.    A <b>test</b> is made of three parts:    <ul>  <li> <b>parameters</b>, which override the suite parameters  <li> <b>groups</b>, made of two parts  <li> <b>classes</b>, defining which classes are going to be part  of this test run  </ul>    In turn, <b>groups</b> are made of two parts:  <ul>  <li> Definitions, which allow you to group groups into  bigger groups  <li> Runs, which defines the groups that the methods  must belong to in order to be run during this test  </ul>    Cedric Beust & Alexandru Popescu  @title DTD for TestNG  @root suite    -->      <!-- A suite is the top-level element of a testng.xml file -->  <!ELEMENT suite (groups?,(listeners|packages|test|parameter|method-selectors|suite-files)\*) >    <!-- Attributes: -->  <!--  @attr name The name of this suite (as it will appear in the reports)  @attr junit Whether to run in JUnit mode.  @attr verbose How verbose the output on the console will be.  This setting has no impact on the HTML reports.  @attr parallel Whether TestNG should use different threads  to run your tests (might speed up the process)  @attr parent-module A module used to create the parent injector of all guice injectors used  in tests of the suite  @attr guice-stage The stage with which the parent injector is created  @attr configfailurepolicy Whether to continue attempting Before/After  Class/Methods after they've failed once or just skip remaining.  @attr thread-count An integer giving the size of the thread pool to use  if you set parallel.  @attr annotations If "javadoc", TestNG will look for  JavaDoc annotations in your sources, otherwise it will  use JDK5 annotations.  @attr time-out The time to wait in milliseconds before aborting the  method (if parallel="methods") or the test (parallel="tests")  @attr skipfailedinvocationcounts Whether to skip failed invocations.  @attr data-provider-thread-count An integer giving the size of the thread pool to use  for parallel data providers.  @attr object-factory A class that implements IObjectFactory that will be used to  instantiate the test objects.  @attr allow-return-values If true, tests that return a value will be run as well  -->  <!ATTLIST suite  name CDATA #REQUIRED  junit (true | false) "false"  verbose CDATA #IMPLIED  parallel (false | methods | tests | classes | instances) "false"  parent-module CDATA #IMPLIED  guice-stage (DEVELOPMENT | PRODUCTION | TOOL) "DEVELOPMENT"  configfailurepolicy (skip | continue) "skip"  thread-count CDATA "5"  annotations CDATA #IMPLIED  time-out CDATA #IMPLIED  skipfailedinvocationcounts (true | false) "false"  data-provider-thread-count CDATA "10"  object-factory CDATA #IMPLIED  group-by-instances (true | false) "false"  preserve-order (true | false) "true"  allow-return-values (true | false) "false"  >    <!-- A list of XML files that contain more suite descriptions -->  <!ELEMENT suite-files (suite-file)\* >    <!ELEMENT suite-file ANY >  <!ATTLIST suite-file  path CDATA #REQUIRED  >    <!--  Parameters can be defined at the <suite> or at the <test> level.  Parameters defined at the <test> level override parameters of the same name in <suite>  Parameters are used to link Java method parameters to their actual value, defined here.  -->  <!ELEMENT parameter ANY>  <!ATTLIST parameter  name CDATA #REQUIRED  value CDATA #REQUIRED >    <!--  Method selectors define user classes used to select which methods to run.  They need to implement <tt>org.testng.IMethodSelector</tt>  -->  <!ELEMENT method-selectors (method-selector\*) >  <!ELEMENT method-selector ((selector-class)\*|script) >  <!ELEMENT selector-class ANY>  <!ATTLIST selector-class  name CDATA #REQUIRED  priority CDATA #IMPLIED  >  <!ELEMENT script ANY>  <!ATTLIST script  language CDATA #REQUIRED  >    <!--  A test contains parameters and classes. Additionally, you can define additional groups ("groups of groups")  -->    <!ELEMENT test (method-selectors?,parameter\*,groups?,packages?,classes?) >    <!--  @attr name The name of this test (as it will appear in the reports)  @attr junit Whether to run in JUnit mode.  @attr verbose How verbose the output on the console will be.  This setting has no impact on the HTML reports.  Default value: suite level verbose.  @attr parallel Whether TestNG should use different threads  to run your tests (might speed up the process)  @attr thread-count An integer giving the size of the thread pool to be used if  parallel mode is used. Overrides the suite level value.  @attr annotations If "javadoc", TestNG will look for  JavaDoc annotations in your sources, otherwise it will  use JDK5 annotations.  @attr time-out the time to wait in milliseconds before aborting  the method (if parallel="methods") or the test (if parallel="tests")  @attr enabled flag to enable/disable current test. Default value: true  @attr skipfailedinvocationcounts Whether to skip failed invocations.  @attr preserve-order If true, the classes in this tag will be run in the same order as  found in the XML file.  @attr allow-return-values If true, tests that return a value will be run as well  -->  <!ATTLIST test  name CDATA #REQUIRED  junit (true | false) "false"  verbose CDATA #IMPLIED  parallel CDATA #IMPLIED  thread-count CDATA #IMPLIED  annotations CDATA #IMPLIED  time-out CDATA #IMPLIED  enabled CDATA #IMPLIED  skipfailedinvocationcounts (true | false) "false"  preserve-order (true | false) "true"  group-by-instances (true | false) "false"  allow-return-values (true | false) "false"  >    <!--  Defines additional groups ("groups of groups") and also which groups to include in this test run  -->  <!ELEMENT groups (define\*,run?,dependencies?) >    <!ELEMENT define (include\*)>  <!ATTLIST define  name CDATA #REQUIRED>    <!-- Defines which groups to include in the current group of groups -->  <!ELEMENT include ANY>  <!ATTLIST include  name CDATA #REQUIRED  description CDATA #IMPLIED  invocation-numbers CDATA #IMPLIED>    <!-- Defines which groups to exclude from the current group of groups -->  <!ELEMENT exclude ANY>  <!ATTLIST exclude  name CDATA #REQUIRED>    <!-- The subtag of groups used to define which groups should be run -->  <!ELEMENT run (include?,exclude?)\* >    <!ELEMENT dependencies (group\*)>    <!ELEMENT group ANY>  <!ATTLIST group  name CDATA #REQUIRED  depends-on CDATA #REQUIRED>    <!-- The list of classes to include in this test -->  <!ELEMENT classes (class\*,parameter\*) >  <!ELEMENT class (methods\*) >  <!ATTLIST class  name CDATA #REQUIRED >    <!-- The list of packages to include in this test -->  <!ELEMENT packages (package\*) >  <!-- The package description.  If the package name ends with .\* then subpackages are included too.  -->  <!ELEMENT package (include?,exclude?)\*>  <!ATTLIST package  name CDATA #REQUIRED >    <!-- The list of methods to include/exclude from this test -->  <!ELEMENT methods (include?,exclude?,parameter?)\* >    <!-- The list of listeners that will be passed to TestNG -->  <!ELEMENT listeners (listener\*) >    <!ELEMENT listener ANY>  <!ATTLIST listener  class-name CDATA #REQUIRED > |