TestNG

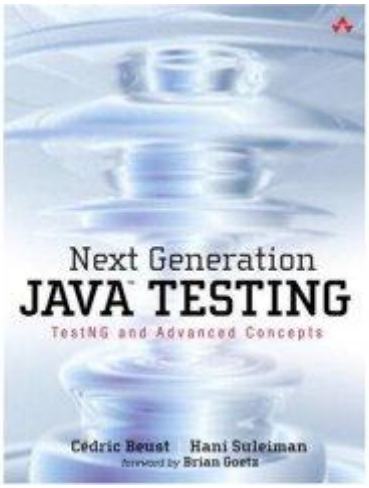
帮助文档（译）

由 Coyote 收集整理

目录

序

TestNG



点此查看更多信息

Cédric Beust (cedric at beust.com)
Current version: 5.10
Created: April 27th, 2004
Last Modified: July 22nd, 2009

TestNG 是一个开源测试框架。它源自于 JUnit 和 NUnit 的启发，并且引入了若干新的特性，使得它更易用，例如：

- JDK 5 Annotations (JDK 1.4 则可以通过 JavaDoc annotations 获得支持).
- 灵活的测试配置。
- 支持数据驱动测试 (使用 @DataProvider)。
- 支持参数。
- 允许在隶属服务器上的分布式测试。
- 强大的执行模型 (再也不需要 TestSuite)。
- 被多数工具和插件支持(Eclipse, IDEA, Maven, etc...).
- 内嵌 BeanShell 进一步提供扩展性
- 默认提供 JDK 运行时和记录日志的函数（没有依赖性）
- 为应用程序服务器测试提供依赖方法。

TestNG 本身就被设计用来覆盖所有类型的测试：单元测试（Unit Testing），功能测试（functional），端到端（end-to-end），集成测试（integration），etc...

我开始接触 TestNG 是源于 JUnit 某些不足而带来的挫折感，我把它们记录在我的 blog 中，在 here 和 here 。阅读这些文档可能会让你更清楚我要用 TestNG 做什么。 你可以看 主要特征一览 和另一篇 文章，其中详细的描绘了一个使用若干 TestNG 特征结合的例子，它是针对一个自然和可维护的测试设计的。

这里是个简单的测试例子：

```
package example1;

import org.testng.annotations.*;

public class SimpleTest {

  @BeforeClass
  public void setUp() {
     // code that will be invoked when this test is instantiated
  }

  @Test(groups = { "fast" })
  public void aFastTest() {
     System.out.println("Fast test");
  }

  @Test(groups = { "slow" })
  public void aSlowTest() {
      System.out.println("Slow test");
  }

}
```

这个 setUp() 方法会测试类被构建之后和所有测试方法执行之前被调用。在这个例子中，我们会运行 fast 组，所以， aFastTest() 会被调用而 aSlowTest() 则会被跳过。

要注意的事情：

- 不需要扩展任何类或接口
- 尽管上面的例子使用了 JUnit 的习惯，但是你可以给方法起任何你喜欢的名字，实际上是 annotation 告诉了 TestNG 他们到底是不是测试方法
- 一个测试方法可以从属与一个或者多个组

一旦当你的测试类被编译到 build 目录，你可以通过命令行下，使用 ant 命令来调用它，一个 ant 任务或者 XML 文件：

```
<project default="test">
```

```
  <path id="cp">
     <pathelement location="lib/testng-testng-4.4-jdk15.jar"/>
     <pathelement location="build"/>
  </path>

  <taskdef name="testng" classpathref="cp"
               classname="org.testng.TestNGAntTask" />

  <target name="test">
     <testng classpathref="cp" groups="fast">
        <classfileset dir="build" includes="example1/*.class"/>
     </testng>
  </target>

</project>
```

使用 ant 调用它：

```
c:> ant
Buildfile: build.xml

test:
[testng] Fast test
[testng] ===============================================
[testng] Suite for Command line test
[testng] Total tests run: 1, Failures: 0, Skips: 0
[testng] ===============================================


BUILD SUCCESSFUL
Total time: 4 seconds
```

之后你就可以浏览你的测试结果了

```
start test-output\index.html (on Windows)
```

需求

TestNG 运行在 JDK 1.4 和 5.　上例是以 JDK 5 为前提的，所以使用其中的 annotation，但是可以通过使用 JavaDoc-type 的 annotations 轻松转换到 JDK 1.4 上。可以参阅 JDK 1.4 部分获得更多细节内容。

Mailing-lists

The users mailing-list can be found in two different places:

- As an email mailing-list on Google Groups.
- As a Web forum, kindly hosted by Open Symphony.

The Web forum and the mailing-list are connected to each other, so you only need to subscribe to one.

项目位置

如果你有意帮助 TestNG 或者 IDE 的插件，可以你可以查询如下地址：

- TestNG
- Eclipse plug-in
- IDEA IntelliJ plug-in
- NetBeans plug-in

1 - 简介

TestNG 是一个测试框架，它被设计为用来简化广泛的设计需求，从单元测试（单独测试一个类）到集成测试（测试由多个类、报甚至外部框架，例如应用程序服务器，所组成的系统）.

编写一个测试通常分为三步：

- 编写测试业务逻辑，并且在你的代码中插入 TestNG annotations 。
- 在 testng.xml 或 build.xml 添加你的测试信息。例如类名，希望运行的组等等
- 运行 TestNG.

你可以在欢迎 页面上找到一个快速入门的例子。

文档中会使用到如下的概念：

- 一套测试（suite）由一个 XML 文件所表示。它能够包含一个或者多个测试，〈suite〉标记来定义。
- 用〈test〉标记来表示一个测试，并且可以包含一个或者多个 TestNG 类。
- 所谓 TestNG 类，是一个 Java 类，它至少包含一个 TestNG annotation。它由<class〉标记表示，并且可以包含一个或着多个测试方法。
- 测试方法，就是一个普通的 Java 方法，在由@Test 标记。

TestNG 测试可以通过@BeforeXXX 和@AfterXXX annotations 来标记，允许在特定的生命周期点上执行一些 Java 逻辑，这些点可以是上述列表中任意的内容。

本手册的其余部分将会解释如下内容：

- 所有 annotation 的列表并且给出扼要的解释。它会告诉你很多 TestNG 所提供的功能，但是你需要详细阅读其余部分来获得关于这些注解更详细的信息。
- 关于 testng.xml 文件的说明。它的语法以及你能在里面写什么。
- 上述内容的详细说明，同时也有结合了 annotation 和 testing.xml 文件的使用说明。

2 - Annotations

如下就是在TestNG中可以使用的annotation的速查预览，并且其中给出了属性：

	TestNG 类的配置信息：
@BeforeSuite @AfterSuite @BeforeTest @AfterTest @BeforeGroups @AfterGroups @BeforeClass @AfterClass @BeforeMethod @AfterMethod	@BeforeSuite: 被注解的方法，会在当前 suite 中所有测试方法之 前 被调用。 @AfterSuite: 被注解的方法，会在当前 suite 中所有测试方法之 后 被调用。 @BeforeTest: 被注解的方法，会在测试（原文就是测试，不是测试方法）运行 前 被调用 @AfterTest: 被注解的方法，会在测试（原文就是测试，不是测试方法）运行 后 被调用 @BeforeGroups: 被注解的方法会在组列表中之前被调用。这个方法会在每个组中第一个测试方法被调用之前被调用。 @AfterGroups: 被注解的方法会在组列表中之后被调用。这个方法会在每个组中最后一个测试方法被调用之后被调用。 @BeforeClass: 被注解的方法，会在当前类第一个测试方法运行前被调用 @AfterClass: 被注解的方法，会在当前类所有测试方法运行后被调用 @BeforeMethod: 被注解的方法，会在运行每个测试方法之前调用 @AfterMethod: 被注解的方法，会在每个测试方法运行之后被调用
alwaysRun	对于在方法之前的调用(beforeSuite, beforeTest, beforeTestClass 和 beforeTestMethod, 除了 beforeGroups): 若为 true，这个配置方法无视其所属的组而运行 对于在方法之后的调用(afterSuite, afterClass, ...): 若为 true， 这个配置方法会运行，即使其之前一个或者多个被调用的方法失败或者被跳过。
dependsOnGroups	方法所依赖的一组 group 列表
dependsOnMethods	方法所依赖的一组 method 列表
enabled	在当前 class/method 中被此 annotation 标记的方法是否参与测试（不启用则不在测试中运行）
groups	一组 group 列表，指明了这个 class/method 的所属。
inheritGroups	如果是 true，则此方法会从属于在类级由@Test 注解中所指定的组
@DataProvider	把此方法标记为为测试方法提供数据的方法。被此注释标记的方法必须返回 Object[][]，其中的每个 Object[]可以被分配给测试方法列表中的方法当做参数。那些需要从 DataProvider 接受数据的@Test 方法，需要使用一个 dataprovider 名字，此名称必须与这个注解中的名字相同。
name	DataProvider 的名字
@Factory	把一个方法标记为工厂方法，并且必须返回被 TestNG 测试类所使用的对象们。 此方法必须返回

	Object[]。
@Parameters	说明如何给一个 @Test 方法传参。
value	方法参数变量的列表
@Test	把一个类或者方法标记为测试的一部分。
alwaysRun	如果为 true，则这个测试方法即使在其所以来的方法为失败时也总会被运行。
dataProvider	这个测试方法的 dataProvider
dataProviderClass	指明去那里找 data provider 类。如果不指定，那么就当前测试方法所在的类或者它个一个基类中去找。如果指定了，那么 data provider 方法必须是指定的类中的静态方法。
dependsOnGroups	方法所依赖的一组 group 列表
dependsOnMethods	方法所以来的一组 method 列表
description	方法的说明
enabled	在当前 class/method 中被此 annotation 标记的方法是否参与测试（不启用则不在测试中运行）
expectedExceptions	此方法会抛出的异常列表。如果没有异常或者一个不在列表中的异常被抛出，则测试被标记为失败。
groups	一组 group 列表，指明了这个 class/method 的所属。
invocationCount	方法被调用的次数。
invocationTimeOut	当前测试中所有调用累计时间的最大毫秒数。如果 invocationCount 属性没有指定，那么此属性会被忽略。
successPercentage	当前方法执行所期望的 success 的百分比
sequential	如果是 true，那么测试类中所有的方法都是按照其定义顺序执行，即使是当前的测试使用 parallel="methods"。此属性只能用在类级别，如果用在方法级，就会被忽略。
timeOut	当前测试所能运行的最大毫秒数
threadPoolSize	此方法线程池的大小。 此方法会根据制定的 invocationCount 值，以多个线程进行调用。 注意：如果没有指定 invocationCount 属性，那么此属性就会被忽略

3 - testng.xml

调用 TestNG 有多种方式：

- 使用 testng.xml 文件
- 使用 ant
- 通过命令行

本节对 testng.xml 的格式进行说明(你会在下文找到关于 ant 和命令行的相关文档)。

目前给 testng.xml 所使用的 DTD 可以在主页：http://testng.org/testng-1.0.dtd 上找到（考虑到您能更方便，可以浏览 HTML 版）。

下面是个 testng.xml 文件的例子：

```
<!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>
```

你也可以指定包名来替代类名：

```
<!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 会查看所有在 test.sample 的类，并且只保留含有 TestNG annotations 的类。

你也可以指定要包含和排除掉的组和方法：

```
<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>
```

你也可以在 testng.xml 定义新的 group，并且在属性中指明额外的详细信息，例如是否并行运行，使用多少个线程，并且是否正在运行 JUnit 测试等等……请参阅 DTD 获取完整的功能列表，或者继续阅读本文。

4 - 运行

TestNG 可以使用多种方式调用：

- 命令行
- ant
- Eclipse
- IntelliJ's IDEA

4.1 命令行

本节将只介绍如何从命令行运行 TestNG。如果您对其他方式感兴趣，那么就点击上面的链接查看更多信息。

假设 TestNG 已经在你的类路径中，最简单的调用方式如下：

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

你至少要指定一个 XML 文件，它描述了你要运行的 TestNG suite。此外，还有如下命令行参数：

命令行参数		
选项	**参数**	**说明**
-d	目录	报告会生成的目录 (默认是 test-output).

命令行参数		
选项	**参数**	**说明**
-excludegroups	逗号分隔的组列表	要在当前运行中被排除的组列表
-groups	逗号分隔的组列表	想要运行的组(e.g. "windows,linux,regression").
-listener	逗号分隔的 Java 类列表，它们都可以在你的类路径中找到	让你指定你自己的监听器。这些类需要实现 org.testng.ITestListener
-parallel	方法 | 测试	如果指定了，那么在运行测试的时候，所使用的默认的机制就会决定如何去使用并行 | 线程。反之则不会。 这是可以在 suite 定义中被覆盖的
-reporter	自定义报告监听器的扩展配置	类似于 -listener 选项，允许在报告器实例中配置 JavaBean 的样式属性。例如： -reporter com.test.MyReporter:methodFilter=*insert*,enableFiltering=true 这个选项不限次数，根据需要一样一个。
-sourcedir	分号间隔的目录列表	使用了 JavaDoc 类型的 annotation 的源码所在的目录。这个选项只有你在使用 JavaDoc 类型的注解时才会有用。(e.g. "src/test" or "src/test/org/testng/eclipse-plugin;src/test/org/testng/testng").
-suitename	test suite 默认的名字	指明了在命令行中定义的 test suite 的名字。这个选项在 suite.xml 或源码指定了不同的名字时会被忽略。如果使用双引号括起来，就可在名字中使用空格。例如："like this"。
-testclass	逗号分隔的类列表，它们必须能在类路径中被找到	逗号分隔的测试类的列表 (e.g. "org.foo.Test1,org.foo.test2").
-testjar	一个 jar 文件	指定了一个包含了测试类的 Jar 文件。如果 testng.xml 在 jar 文件的根目录被找到，就使用之，反之，jar 文件中所有的类都会被当成测试类。
-testname	测试所使用的默认名字	它为在命令行中定义的测试指定了名字。这个选项在 suite.xml 或源码指定了不同的名字时会被忽略。如果使用双引号括起来，就可在名字中使用空格。例如："like this"。
-testrunfactory	可以在类路径中找到的 Java 类	让你指定你自己的测试运行器，相关的类必须实现 org.testng.ITestRunnerFactory.
-threadcount	在并行测试的时候默认使用的线程数	并行运行中所使用的最大线程数。只在使用并行模式中有效（例如，使用 -parallel 选项）。它可以在 suite 定义中被覆盖。

上面的参数说明可以通过不带任何参数运行 TestNG 来获得。

你也可以把命令行开关放到文件中，例如说 c:\command.txt，之后告诉 TestNG 使用这个文件来解析其参数：

```
C:> more c:\command.txt
  -d test-output testng.xml
  C:> java org.testng.TestNG @c:\command.txt
```

此外 TestNG 也可以在命令行下向其传递 JVM 参数。例如：

```
java -Dtestng.test.classpath="c:/build;c:/java/classes;"
org.testng.TestNG testng.xml
```

如下是 TestNG 所能理解的属性：

系统属性		
属性	**类型**	**说明**
testng.test.classpath	分号分的一系列目录，其中包含了你的测试类	如果指定了这个属性，TestNG 就会查找你的测试类而不是类路径。这在你的类路径中有很多类，而大多数又不是测试类，或者在 xml 文件中使用 package 标记的时候会很方便。

例子：

java org.testng.TestNG -groups windows,linux -testclass org.test.MyTest

注意 ant task 和 testng.xml 允许你使用更多的参数来运行 TestNG（要包含的方法、指定的参数等等），所以你在学习 TestNG 的时候考虑使用命令行，因为这样能让你快速进步。

4.2 ANT

TestNG 与 Ant Task

使用如下方法在 ant 任务中定义 TestNG：

```
<taskdef resource="testngtasks" classpath="testng.jar"/>
```

下面的任务运行了 TestNG 测试，并且总是运行在分支式 JVM 中。它接受如下属性：

属性	说明	必需
annotations	字符串 "JDK" 或者 "Javadoc"，定义了测试中使用哪种注解。如果使用了 Javadoc，那么也需要制定"sourcedir".	不是。默认是 "JDK"，如果你使用的是 JDK 5 的包，反之就是 "Javadoc"，如果你使用了 JDK 1.4 的包
classfilesetref	要运行的测试类所引用	

	的 FileSet 到的结构。	
classpath	路径，PATH 类似于要运行的测试。	
classpathref	到某个路径下要运行的测试的引用	
dataProviderThreadCount	给 data provider 提供的用来运行测试的线程数量。除非启动并行模式，否则默认为忽略	1
delegateCommandSystemProperties	把命令行属性传递为系统属性	非必需，默认为 false
dumpCommand	打印出 TestNG 运行的命令	非必需，默认为 false
enableAssert	启用 JDK 1.4 的断言	非必需，默认为 true
failureProperty	失败事件中设置的属性名称，只有在 haltonfailure 没有设置的时候使用	非必需
haltonfailure	在测试运行中，如果发生失败则停止构建过程	非必需，默认为 false
haltonskipped	只要至少发生一处 skipped 测试，就停止构建过程。	非必需，默认为 false
groups	要运行的组的名字，通过逗号分隔	
excludedgroups	要排除的组的名字，通过逗号分隔	
jvm	要使用的 JVM 通过使用 Runtime.exec()来调用	java
listeners	逗号或者空格分隔的全程类名列表，其中的类用作 TestNG 的监听器（例如 org.testng.ITestListener 或 org.testng.IReporter）	非必需
outputdir	测试报表输出目录	非必需，默认为 test-output.

skippedProperty	发生 skipped 事件时候属性的名字，只有在 haltonskipped 没有设置的时候生效	非必需
sourcedir	给 JDK 1.4 使用的一个像路径一样的测试结构（说白了就是个目录）使用（JavaDoc 的那种注解）	
sourcedirref	给 JDK 1.4 使用的到一个像路径已将结构源码包的引用（使用 JavaDoc 那样的注解）	
suiteRunnerClass	完整 TestNG 启动器的类名	非必需，默认为 org.testng.TestNG
parallel	使用并行模式运行测试——可以是 methods 或 tests	如果没出现就是非必需的。此时不使用并行模式
threadCount	并行模式下线程的数量。如果没有使用并行模式，则被忽略	1
testJar	志向一个包含有 tests 和 suite 定义的 jar 文件的路径	
timeOut	以毫秒记的，所有的测试最多可以总计运行的时间	
useDefaultListeners	是否使用默认的监听器和报表生成器	默认为 true
workingDir	ant 运行 TestNG 的工作目录	
xmlfilesetref	在要运行的 suite 定义中指向 FileSet 结构的引用。	
suitename	如果 suite 的 xml 文件或者源码中没有指定，那么可以在这里指定 test suite 的名字	非必需，默认为 "Ant suite"
testname	如果 suite 的 xml 文件或者源码中没有指定，那么	非必需，默认为 "Ant suite"

	可以在这里指定 test 的名字	

属性 classpath, classpathref 或者内嵌的 <classpath> 必需提供运行测试的类路径。

属性 xmlfilesetref, classfilesetref 或内嵌的 <xmlfileset>, <classfileset> 必需分别得提供自己使用的类测试。

注意： 如果使用 JDK 1.4 那么必需使用属性 sourcedir, sourcedirref 或内嵌的<sourcedir>。

注意： 使用 <classfileset> 不会自动添加测试类到你的类路径中中，你可能需要重复处理这些类一边骑能够在任务中被正确的使用。

内嵌的元素

classpath

<testng> 任务支持内嵌的 <classpath> 元素，它表示一个路径一样的结构（*PATH*-like structure）。

bootclasspath

引导类程序的位置，可以通过使用这个类似路径的结构来指定。如果 fork 没有被设置就被忽略。

xmlfileset

suite 的定义 (testng.xml) 可以通过使用 FileSet 结构传递给 task

classfileset

TestNG 也可以直接运行类，通过使用 FileSet 结构。

sourcedir

路径似的结构，给那些使用 JDK 1.4 测试中使用 Javadoc 风格的注解的类使用的

jvmarg

通过内嵌的 <jvmarg> 元素传递给 JVM 的其他参数，例如：

```
<testng>
      <jvmarg value="-Djava.compiler=NONE" />
      <!-- ... -->
</testng>
```

sysproperty

使用 <sysproperty> 元素来为类指定特殊的系统属性值。这些属性在 VM 执行测试的时候可以被调用。次元素的属性与*环境变量* 一样：

```
<testng>
      <sysproperty key="basedir" value="${basedir}"/>
      <!-- ... -->
</testng>
```

will run the test and make the basedir property available to the test.

reporter

内部的 <reporter> 元素可以用来注入自定义的报表监听器，允许用户设置自定义的属性，以便在运行时调整报表器的行为。
这个元素有个 classname 属性，它是必须的，其中指明了自定义监听器的类。为了设置报表器的属性， <reporter> 隐患苏可以包含多个内嵌的 <property> 元素，并且会提供 name 值 value 属性，见下面的例子：

```
<testng ...>
      ...
      <reporter classname="com.test.MyReporter">
            <property name="methodFilter" value="*insert*"/>
            <property name="enableFiltering" value="true"/>
      </reporter>
      ...
</testng>

public class MyReporter {

    public String getMethodFilter() {...}
    public void setMethodFilter(String methodFilter) {...}
    public boolean isEnableFiltering() {...}
```

```
    public void setEnableFiltering(boolean enableFiltering) {...}
    ...
}
```

你要考虑好，在目前的情况下只支持有限的属性类型：String, int, boolean, byte, char, double, float, long, short.

env

可以通过使用 <env> 元素来为在 VM 中被调用的 TestNG 传递环境变量。关于 <env> 元素属性的说明，请参看 *exec* 任务的说明。

例子

Suite xml

```
<testng classpathref="run.cp"
                outputDir="${testng.report.dir}"
                sourcedir="${test.src.dir}"
                haltOnfailure="true">

      <xmlfileset dir="${test14.dir}" includes="testng.xml"/>
</testng>
```

Class FileSet

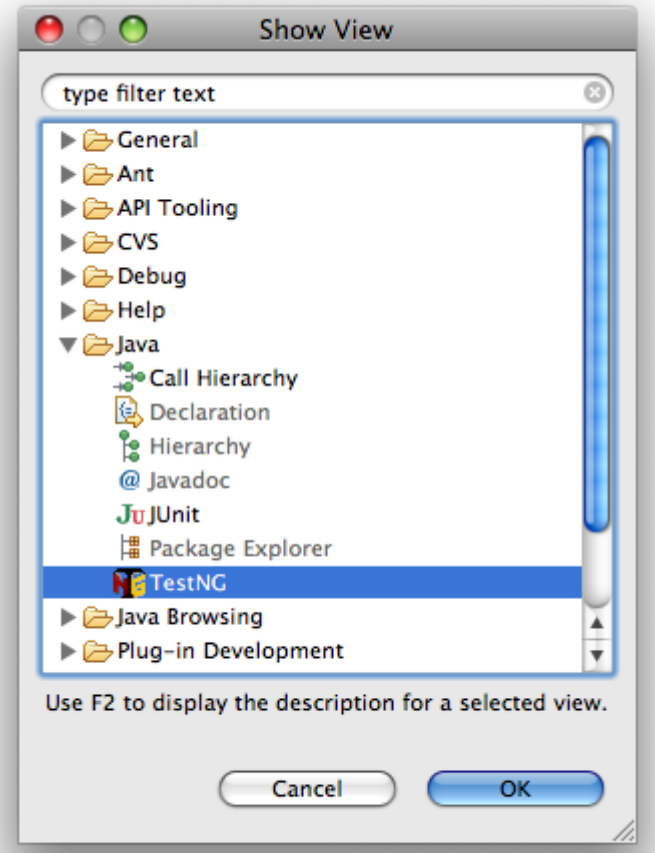
```
<testng classpathref="run.cp"
                outputDir="${testng.report.dir}"
                haltOnFailure="true"M verbose="2">
        <classfileset dir="${test.build.dir}" includes="**/*.class" />
</testng>
```

4.3 TestNG 的 Eclipse 插件

TestNG 的 Eclipse 插件允许你轻松的在 Eclipse 下运行 TestNG 测试，并且方便的监视其执行和输出。它由自己单独的项目，参看在 code.google.com 上的项目 叫做 testng-eclipse。

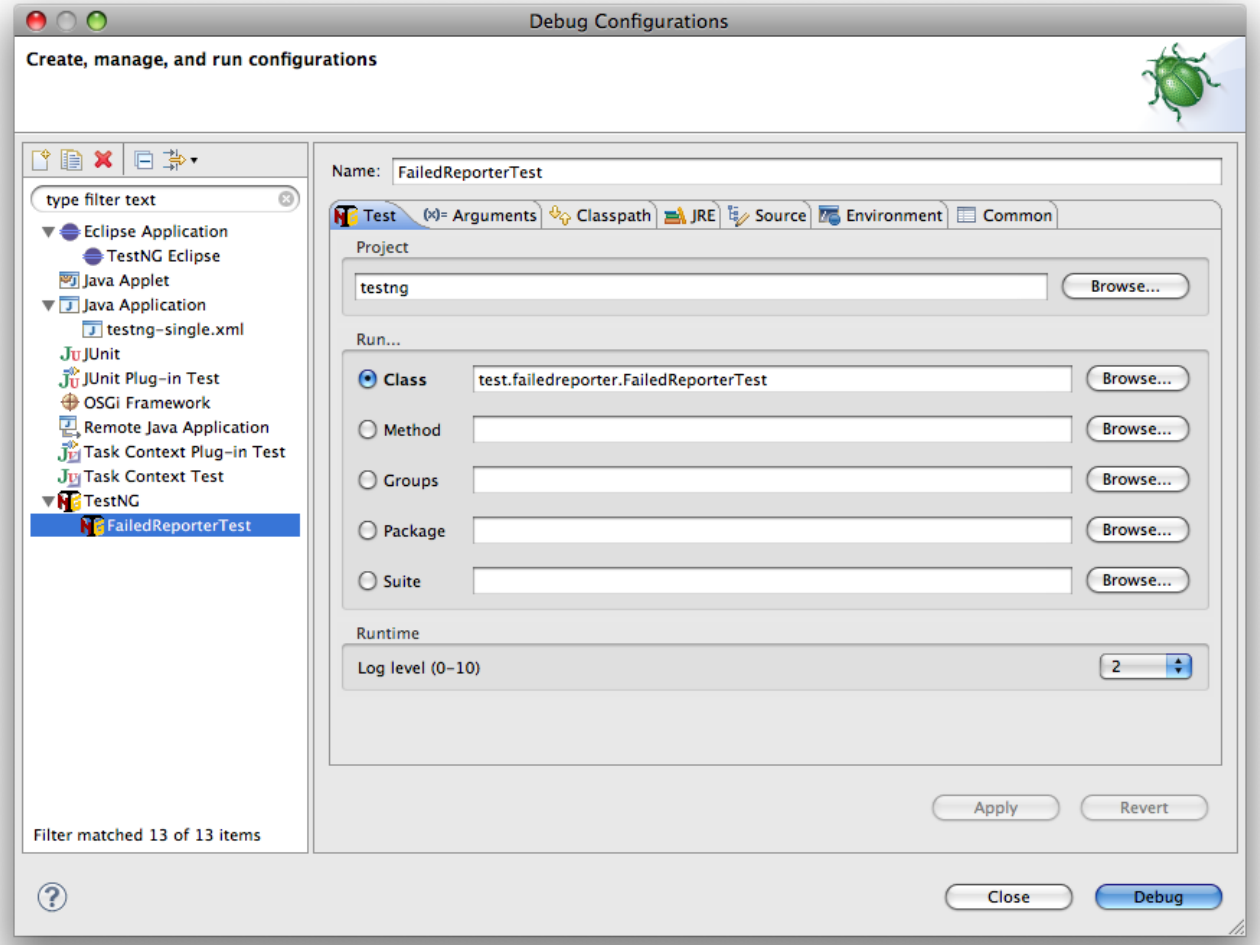
4.3.1 - 安装

一旦已经安装了 插件，重启动 Eclipse 并且选择菜单中的 Window / Show View / Other... 之后你能看到 TestNG 视图被列在 Java 类别里。



4.3.2 - 创建 TestNG 运行配置

一旦你已经完成穿件带有 TestNG 注解的类，和/或一个或多个 testng.xml 文件，你就可以创建一个 TestNG 运行配置了。选择 Run / Run... (or Run / Debug...) 菜单，并且创建一个新的 TestNG 配置：

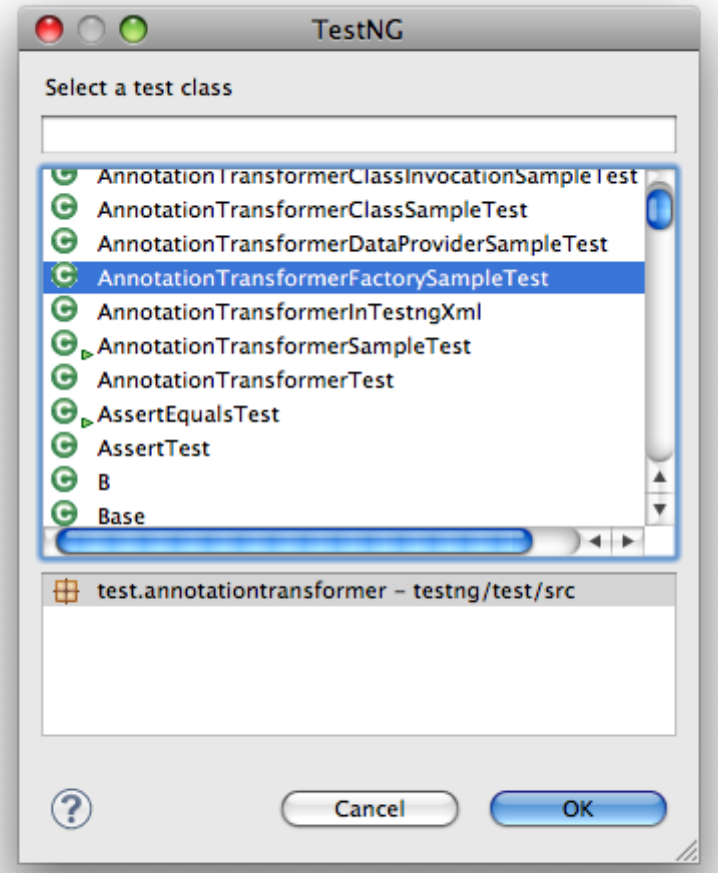


你要改改配置的名气，并且选一个项目，可以通过单击 Browse... 按钮搞定。

之后你可以选择用如下的方式运行 TestNG 测试：

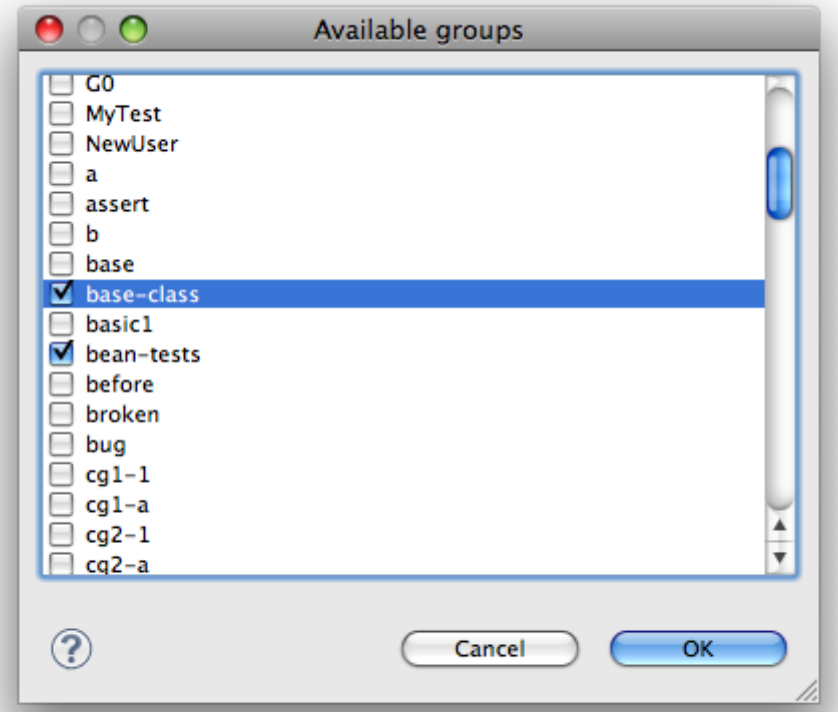
4.3.2.1 - 按照类文件

注意在 Class 附近的复选框被选中的，之后从项目中选择你的类。你可以单击 Browse... 按钮，然后直接从列表里面选。这个列表仅仅包含含有 TestNG 注解的类：



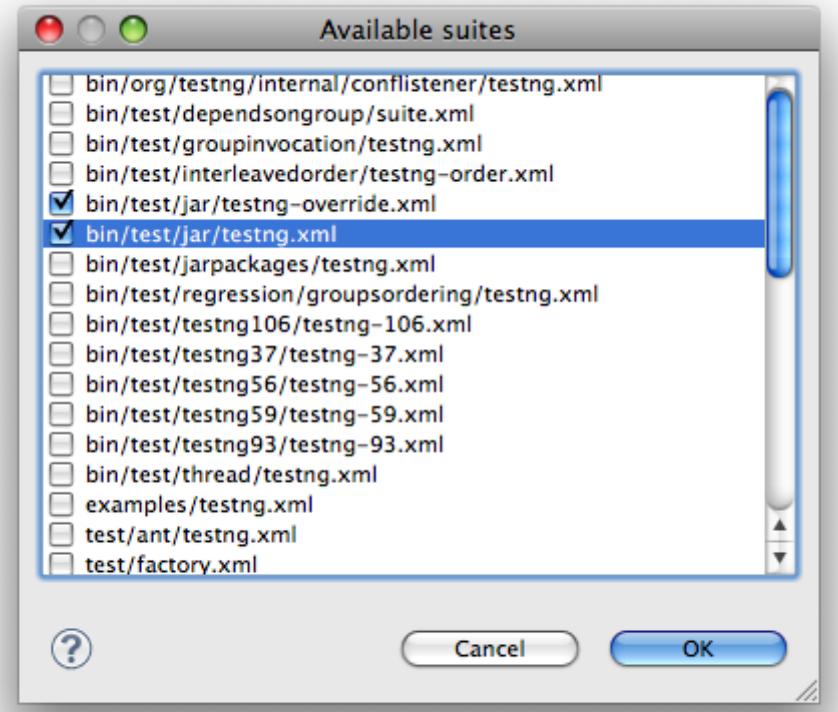
4.3.2.2 - 按照组

如果你想要运行一个或者多个组，可以在文本框中输入，或者按 Browse... 按钮从里面选：



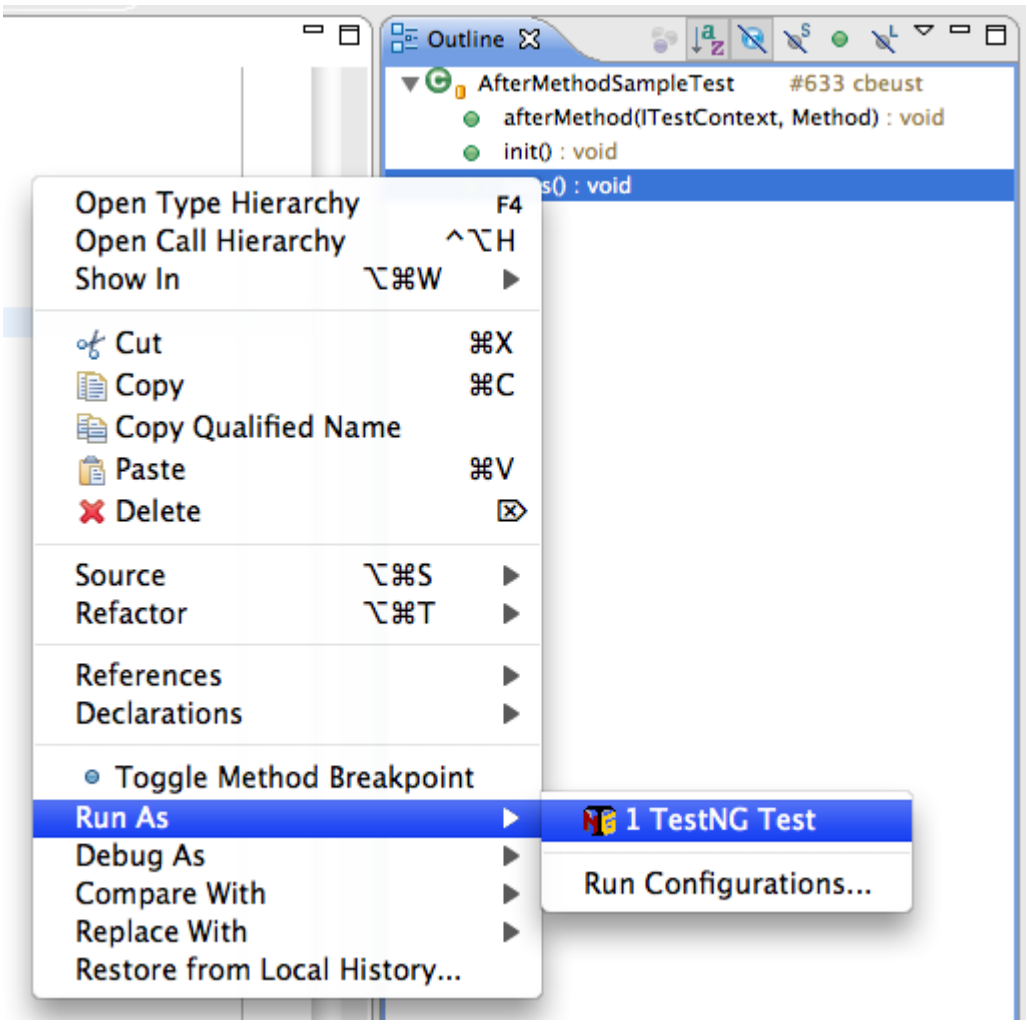
4.3.2.3 - 按照定义文件

最后，你还可以为项目选择一套测试定义文件。这个文件不必非得明明为 testng.xml，插件会自动识别在你项目中所有的 TestNG XML 文件：



4.3.2.4 - 按照方法

这种情况不能直接从运行对话框中完成，但是可以直接从 Outline 视图中完成：

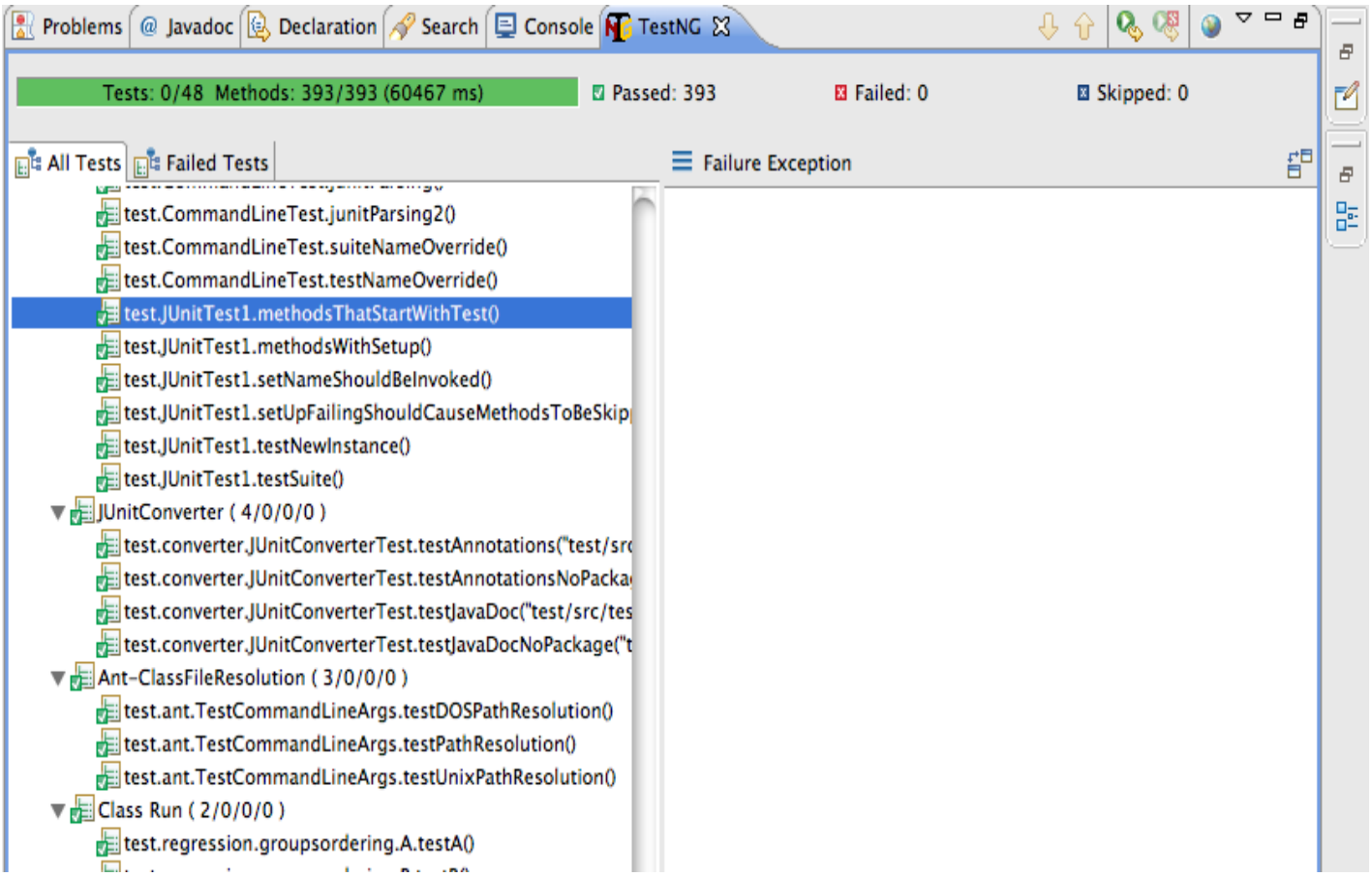


你可以右键单击任何一个测试方法，然后选择 Run as... / TestNG test ，并且只有被选择的方法会被运行（这个没有在上图中表示，因为我没有找到给弹出菜单截图的办法）。

按照方法运行测试也是可以从包浏览器中调用的，当然 Java Browser 视图中也行。

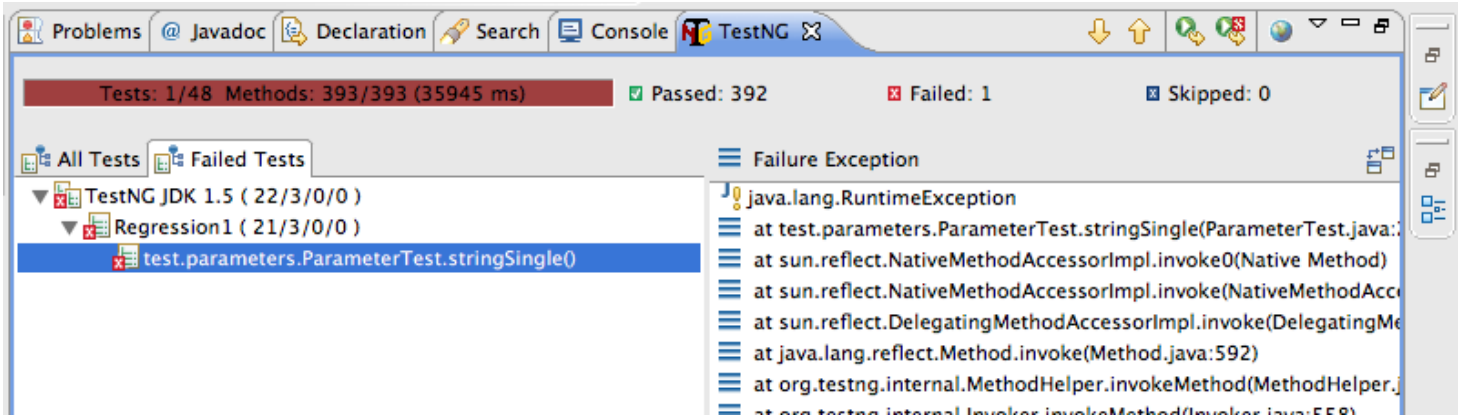
一旦你选择上述方法之一，你也可以选择日志级别，并且让 TestNG 是否运行在 1.4 或者 1.5 的模式下。之后你可以通过按下 Debug （或 Run）按钮来运行。此时，你会被切换到 Debug 视图，并且会打开 TestNG 主视图。

4.3.3 - 查看测试结果



上图显示一组成功运行的测试：进度条是绿的，并且没有失败报告。All tests 标签显示给你所有运行过的方法和类的列表。

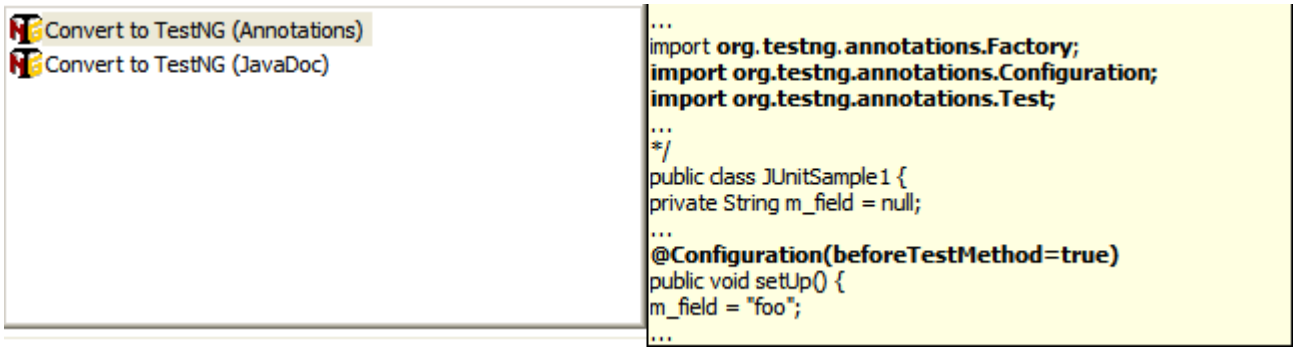
如果你的运行包含失败，视图看起来会像这样：



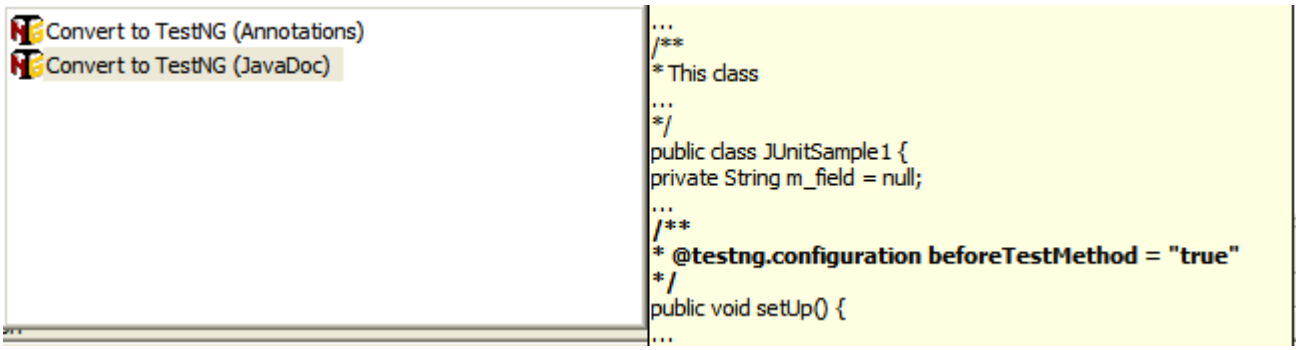
你可以使用 Failed tests 标签，让其只显示失败的测试，这样方便你选择其中之一，并在右侧查看弹栈信息。你还可以直接在那些弹栈信息中的某一条上（原文：offending line）直接双击，这样会直接跳转到你的代码中。

4.3.4 - 转换 JUnit 测试

当你编辑 JUnit 测试类的时候，按下 Ctrl-1（快速修复），并且插件会给你转换到 TestNG 的选项。你即可以选择使用 JDK5 注解：



也可以选择使用 JavaDoc 注解：



4.4 Maven

Maven2 本身就支持 TestNG 而无需下载任何额外的插件（除了 TestNG 自己）。

当前版本

当前最新的且比较好的 Surefire 插件是 2.4。 你可以参考这里 Surefire 网站（而这里是 TestNG 特别指南）。

配置

目标

目标	说明
test	编译和运行你的测试
site	创建你自己的 Maven 生成站点，其中包括你的 TestNG 报告输出

属性

属性	可选？	说明
includes	No	逗号分隔的正则表达式样式，其中包含源码例如：. Ie **/Test*.java
groups	Yes	逗号分割的用来被测试的组名。如果是空的，默认是所有的组都被运行。
excludeGroups	Yes	逗号分隔的不需要被测试的组。
suiteXmlFiles	Yes	逗号分隔的志向 testng.xml 的文件路径列表。(src/test-data/testng-core.xml,src/test-data/testng-functional.xml) *警告：当定义 suiteXmlFiles 其他大多数的参数都会被忽略。*
threadCount	Yes	用来运行测试的线程数。
parallel	Yes	当使用多线程的时候，是否要并行运行测试。使用 *tests* 来让每个测试运行在自己的线程中，或者使用 *methods* 使得每个被调用的方法运行在不同的线程里。

Java 1.4

为了能够使用 javadoc 风格的注解，你当前必须使用 1.4 版的 JVM 来运行 Maven 以便能够看到效果。如果是使用 1.5 的 JVM 来处理这些 javadoc 风格的注解，那么什么都不会发生。这个问题以后会消灭的。

报告样例

使用 TestNG 的 surefire 报告可以看 这里 。

Maven TestNG *原型 (Martin Gilday)*

Martin Gilday 已经为 Maven2 用户添加了新的原型，使其更容易使用 TestNG。你可以在他的 blog here 中找到更多内容。 但是基本的配置我已经在下面列出来了。

要创建一个使用原型的项目，你只要简单的制定我的代码库和原型 id。

```
   mvn archetype:create -DgroupId=org.martingilday -DartifactId=test1
-DarchetypeGroupId=org.martingilday
-DarchetypeArtifactId=testng-archetype
      -DarchetypeVersion=1.0-SNAPSHOT
-DremoteRepositories=http://www.martingilday.org/repository/
```

当然了，你可以替换为自己的 groudId 和 artifactId。

别忘了经常来 Martin's blog 看看有没有更多的更新。

Maven 1 (by Andrew Glover)

TestNG Maven 插件非常简单，并且由两个目标和一系列可选的属性组成。

当前 1.1 版的插件，有 TestNG 官方发行版绑定。要使用这个插件，就把 maven-testng-plugin-.jar 拷贝到 $MAVEN_HOME/plugins 目录即可。

关于最新版的插件 (1.2 于 12/12/05)，更新你的 maven.repo.remote ，它在 http://www.vanwardtechnologies.com/repository/ 并且使用如下命令：maven plugin:download. Maven 会产生如下的内容：

artifactId:	maven-testng-plugin
groupId:	testng
version:	1.2

目标

目标	说明
testng	运行 TestNG
testng:junit-report	创建 JUnit 风格报表

属性

属性	可选?	说明
maven.testng.suitexml.name	Yes	XML 文件名- 默认是 testng.xml
maven.testng.suitexml.dir	Yes	XML 文件所在目录。默认是 ${basedir}/test/conf
maven.testng.output.dir	Yes	默认的报表目录。默认是 ${maven.build.dir}/testng-output
maven.testng.source.dir	Yes	对于 1.4 的源码位置，默认是 ${basedir}/test/java/
maven.testng.report.dir	Yes	JUnit 报表的目录，默认是 ${maven.build.dir}/testngJunitReport

通过使用 TestNG 的 maven 插件生成的 JUnit 风格的测试报告可以看这里。

5 - 测试方法、测试类和测试组

5.1 - 测试组

TestNG 允许你将测试方法归类为不同的组。不仅仅是可以声明某个方法属于某个组，而且还可以让组包含其他的组。这样 TestNG 可以调用或者请求包含一组特定的组（或者正则表达式）而排除其他不需要组的集合。这样，如果你打算将测试分成两份的时候，就无需重新编译。这个特点，会给你在划分组的时候带来很大的灵活性。

例如，通常将测试划分为两种类别是再常见不过的了：

- 检查性测试（Check-in test）：这些测试在你提交新代码之前就会运行。它们一般都是很快进行的，并且保证没有哪个基本的功能是不好使的。
- 功能性测试（Functional test）：这些测试涵盖你的软件中所有的功能，并且至少每天运行一次，不过你也可能希望他们持续的运行。

典型的来说，检测性测试通常是功能性测试的一个子集。TestNG 允许你根据个人感觉来进行组划分。例如，你可能希望把你所有的测试类都划归为"functest"组，并且额外的有几个方法输入"checkintest"组。

```
public class Test1 {
   @Test(groups = { "functest", "checkintest" })
   public void testMethod1() {
   }
```

```
   @Test(groups = {"functest", "checkintest"} )
   public void testMethod2() {
   }

   @Test(groups = { "functest" })
   public void testMethod3() {
   }

}
```

通过下面的内容调用 TestNG

```
<test name="Test1">
   <groups>
      <run>
         <include name="functest"/>
      </run>
   </groups>
   <classes>
      <class name="example1.Test1"/>
   </classes>
</test>
```

以上会运行上面那个类中所有的测试，当药使用 checkintest 进行调用的时候，就仅仅运行 testMethod1()和 testMethod2()。

下面是另外一个例子。这次使用正则表达式。假定有些测试方法不应该运行在 Linux 环境下，你的测试会看起来像：

```
@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() {
   }
}
```

然后你就可以使用下面这个 testng.xml 来只运行在 Windows 下的方法：

```
<test name="Test1">
   <groups>
      <run>
         <include name="windows.*"/>
      </run>
   </groups>

   <classes>
      <class name="example1.Test1"/>
   </classes>
</test>
```

注意：TestNG 使用的是 正则表达式，而不是 通配符。注意这二者的区别（例如，"anything" 是匹配于 "." -- 点和星号 -- 而不是星号 "*").*

5.2 方法组

也可以单独排除和包含若干方法：

```
<test name="Test1">
   <classes>
      <class name="example1.Test1">
         <methods>
            <include name=".*enabledTestMethod.*"/>
            <exclude name=".*brokenTestMethod.*"/>
         </methods>
        </class>
   </classes>
</test>
```

这样就可以不用编译而处理任何不需要的方法了，但是我不推荐过分的使用这个技巧，因为如果你要重构你的代码，那么这有可能让你的测试框架出问题（在标签中使用的方法可能再也不会匹配你的方法名了）。

5.3 - 组中组

测试组也可以包含其他组。这样的组叫做“元组”（MetaGroups）。例如，你可能要定义一个组 all 来包含其他的组，chekcintest 和 functest。"functest" 本身只包含了组 windows 和 linux，而"checkintest"仅仅包含 windows。你就可以在属性文件中这样定义：

```
<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>
```

5.4 - 排除组

TestNG 允许你包含组，当然也可以排除之。

譬如说，因为最近的改动，导致当前的测试中断并且，你还没有时间修复这些问题都是司空见惯的。但是，你还需要自己的功能测试可以正确运行，所以，制药简单的让这些不需要的测试失效就可以了。但是别忘记在以后需要的时候，要重新让其生效。

一个简单的办法来解决这个问题就是创建一个叫做"broken"组， 然后使得这些测试方法从属于那个组。例如上面的例子，假设我知道 testMethod2()会中断，所以我希望使其失效：

```
@Test(groups = {"checkintest", "broken"} )
public void testMethod2() {
}
```

而我所需要做的一切就是从运行中排除这个组：

```
<test name="Simple example">
   <groups>
      <run>
         <include name="checkintest"/>
         <exclude name="broken"/>
      </run>
   </groups>
```

```
   <classes>
      <class name="example1.Test1"/>
   </classes>
</test>
```

通过这种办法，我们既可以得到整洁的测试运行，同时也能够跟踪那些需要稍后修正的中断的测试。

注意：你可以可以通过使用"enabled"属性来完成，这个属性适用于@Test 和 @Before/After annotation。

5.5 - 局部组

可以在类级别定义组，之后还可以在方法级定义组：

```
@Test(groups = { "checkin-test" })
public class All {

   @Test(groups = { "func-test" )
   public void method1() { ... }

   public void method2() { ... }
}
```

在这个类中，method2() 类级组"checkin-test"的一部分，而 method1()即属于"checkin-test"也属于"func-test"组。

5.6 - 参数

测试方法是可以带有参数的。每个测试方法都可以带有任意数量的参数，并且可以通过使用 TestNG 的@Parameters 向方法传递正确的参数。

设置方式有两种方法：使用 testng.xml 或者程序编码。

5.6.1 - 使用 testng.xml 设置参数

如果只使用相对简单的参数，你可以在你的 testng.xml 文件中指定：

```
@Parameters({ "first-name" })
@Test
public void testSingleString(String firstName) {
   System.out.println("Invoked testString " + firstName);
```

```
   assert "Cedric".equals(firstName);
}
```
在这段代码中，我们让 firstName 参数能够接到 XML 文件中叫做 first-name 参数的值。这个 XML 参数被定义在 testng.xml：
```
<suite name="My suite">
   <parameter name="first-name"    value="Cedric"/>
   <test name="Simple example">
   <-- ... -->
```
类似的，它也可以用在 @Before/After 和 @Factory 注解上：
```
@Parameters({ "datasource", "jdbcDriver" })
@BeforeMethod
public void beforeTest(String ds, String driver) {
   m_dataSource = ...;                                                    //
```
查询数据源的值
```
   m_jdbcDriver = driver;
}
```
这次有两个 Java 参数 *ds* 和 *driver* 会分别接收到来自属性 datasource 和 jdbc-driver 所指定的值。

参数也可以通过 Optional 注释来声明：

```
@Parameters("db")
@Test
public void testNonExistentParameter(@Optional("mysql") String db)
{ ... }
```
如果在你的 testng.xml 文件中没有找到"db"，你的测试方法就会使用 @Optional 中的值： "mysql"。

@Parameters 可以被放置到如下位置：

- 在任何已经被 @Test, @Before/After 或 @Factory 注解过的地方。
- 在测试类中至多被放到一个构造函数签。这样，TestNG 才能在需要的时候使用 testng.xml 中特定的参数来实例化这个类。这个特性可以被用作初始化某些类中的值，以便稍后会被类中其他的方法所使用。

注意：

- *XML 中的参数会按照 Java 参数在注解中出现的顺序被映射过去，并且如果数量不匹配，TestNG 会报错。*
- *参数是有作用范围的。在 testng.xml 中，你即可以在<suite> 标签下声明，也可以在 <test>下声明。如果两个参数都有相同的名字，那么，定义在 <test> 中的有优先权。这在你需要覆盖某些测试中特定参数的值时，会非常方便。*

5.6.2 — 使用 DataProviders 提供参数

在 testng.xml 中指定参数可能会有如下的不足：

- 如果你压根不用 testng.xml.
- 你需要传递复杂的灿说，或者从 Java 中创建参数（复杂对象，对象从属性文件或者数据库中读取的 etc...）

这样的话，你就可以使用 Data Provider 来给需要的测试提供参数。所谓数据提供者，就是一个能返回对象数组的数组的方法，并且这个方法被@DataProvider 注解标注：

```
//这个方法会服务于任何把它（测试方法）的数据提供者的名字为"test1"方法
@DataProvider(name = "test1")
public Object[][] createData1() {
  return new Object[][] {
     { "Cedric", new Integer(36) },
     { "Anne", new Integer(37)},
  };
}

//这个测试方法，声明其数据提供者的名字为“test1”
@Test(dataProvider = "test1")
public void verifyData1(String n1, Integer n2) {
  System.out.println(n1 + " " + n2);
}
```

结果会打印

```
Cedric 36
Anne 37
```

被@Test 标注的方法通过 dataProvider 属性指明其数据提供商。这个名字必须与@DataProvider(name="...") 中的名字相一致。

默认的情况下，数据提供者会查找当前的测试类或者测试类的基类。如果你希望它能够被其他的类所使用，那么就要将其指定为 static，并且通过 dataProviderClass 属性指定要使用的类：

```
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 方法可以返回如下两种类型中的一种：

- 含有多个对象的数组 (Object[][])，其中第一个下标指明了测试方法要调用的次数，第二个下标则完全与测试方法中的参数类型和个数相匹配。上面的例子已经说明。
- 另外一个是迭代器 Iterator<Object[]>。二者的区别是迭代器允许你延迟创建自己的测试数据。TestNG 会调用迭代器，之后测试方法会一个接一个的调用由迭代器返回的值。在你需要传递很多参数组给测试组的时候，这样你无须提前创建一堆值。

下面是使用 JDK 1.4 和 JDK5 的例子 (注意 JDK 1.4 的例子不适用泛型)：

```
/**
  * @testng.data-provider name="test1"
  */
public Iterator createData() {
   return new MyIterator(DATA);
)
@DataProvider(name = "test1")
public Iterator
```

如果你声明的 @DataProvider 使用 java.lang.reflect.Method 作为第一个参数，TestNG 会把当前的测试方法当成参数传给第一个参数。这一点在你的多个测试方法使用相同的@DataProvider 的时候，并且你想要依据具体的测试方法返回不同的值时，特别有用。

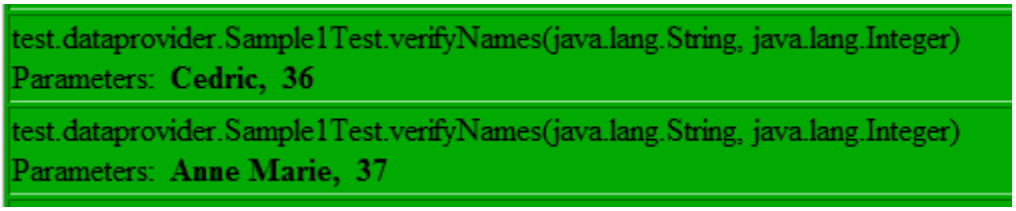
例如，下面的代码它内部的 @DataProvider 中的测试方法的名字：

```
@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) {
}
```
所以会显示：
test1
test2
Data provider 可以通过属性 parallel 实现并行运行：
```
@DataProvider(parallel = true)
// ...
```
使用 XML 文件运行的 data provider 享有相同的线程池，默认的大小是 10.你可以通过修改该在 <suite> 标签中的值来更改：
```
<suite name="Suite1" data-provider-thread-count="20" >
...
```
如果你需要让指定的几个 data provider 运行在不同的线程中，那么就必须通过不同的 xml 文件来运行。

5.6.3 - 在报告中的参数

在测试方法调用中所使用的参数，会在由 TestNG 中生成 HTML 报告里显示出来。下面是几个例子：



5.7 - 依赖方法

有些时候，需要按照特定顺序调用测试方法。对于下面的情况，这非常有用：

- 确保在进行更多的方法测试之前，有一定数量的测试方法已经成功完成。
- 在初始化测试的时候，同时希望这个初始化方法也是一个测试方法（ @Before/After 不会出现在最后生成的报告中）。

为此，你可以使用 @Test 中的 dependsOnMethods 或 dependsOnGroups 属性。

这两种依赖：

- Hard dependencies（**硬依赖**）。所有的被依赖方法必须成功运行。只要有一个出问题，测试就不会被调用，并且在报告中被标记为 SKIP。
- Soft dependencies（**软依赖**）。 即便是有些依赖方法失败了，也一样运行。如果你只是需要保证你的测试方法按照顺序执行，而不关心他们的依

赖方法是否成功。那么这种机制就非常有用。可以通过添加 "alwaysRun=true" 到 @Test 来实现软依赖。

硬依赖的例子：

```
@Test
public void serverStartedOk() {}

@Test(dependsOnMethods = { "serverStartedOk" })
public void method1() {}
```

此例中，method1() 依赖于方法 serverStartedOk()，从而保证 serverStartedOk() 总是先运行。

也可以让若干方法依赖于组：

```
@Test(groups = { "init" })
public void serverStartedOk() {}

@Test(groups = { "init" })
public void initEnvironment() {}

@Test(dependsOnGroups = { "init.* })
public void method1() {}
```

本例中,method1()依赖于匹配正则表达式"init.*"的组，由此保证了 serverStartedOk()和 initEnvironment() 总是先于 method1()被调用。

注意：正如前面所说的那样，在相同组中的调用可是在夸测试中不保证顺序的。

如果你使用硬依赖,并且被依赖方法失败(alwaysRun=false,即默认是硬依赖)，依赖方法则**不是**被标记为 FAIL 而是 SKIP。被跳过的方法会被在最后的报告中标记出来（HTML 既不用红色也不是绿色所表示），主要是被跳过的方法不是必然失败，所以被标出来做以区别。

无论 dependsOnGroups 还是 dependsOnMethods 都可以接受正则表达式作为参数。对于 dependsOnMethods，如果被依赖的方法有多个重载，那么所有的重载方法都会被调用。如果你只希望使用这些重载中的一个，那么就应该使用 dependsOnGroups。

更多关于依赖方法的例子，请参考这篇文章，其中也包含了对多重依赖使用继承方式来提供一种优雅的解决方案。

5.8 - 工厂

工厂允许你动态的创建测试。例如，假设你需要创建一个测试方法，并用它来多次访问一个 web 页面，而且每次都带有不同的参数：

```
public class TestWebServer {
   @Test(parameters = { "number-of-times" })
   public void accessPage(int numberOfTimes) {
      while (numberOfTimes-- > 0) {
        // access the web page
      }
   }
}
```

testng.xml:

```
<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>
```

参数一旦多起来，就难以管理了，所以应该使用工厂来做：

```
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;
   }
}
```

新的测试类如下：

```
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
      }
   }
}
```

你的 testng.xml 只需要引用包含工厂方法的类，而测试实例自己会在运行时创建：

```
<class name="WebTestFactory" />
```

工厂方法可以接受诸如 @Test 和 @Before/After 所标记的参数，并且会返回 Object[]。这些返回的对象可以是任何类（不一定是跟工厂方法相同的类），并且他们甚至都不需要 TestNG 注解（在例子中会被 TestNG 忽略掉）

5.9 - 类级注解

通常 @Test 也可以用来标注类，而不仅仅是方法：

```
@Test
public class Test1 {
   public void test1() {
   }

   public void test2() {
   }
}
```

处于类级的 @Test 会使得类中所有的 public 方法成为测试方法，而不管他们是否已经被标注。当然，你仍然可以用 @Test 注解重复标注测试方法，特别是要为其添加一些特别的属性时。

例如：

```
@Test
public class Test1 {
   public void test1() {
   }

   @Test(groups = "g1")
   public void test2() {
   }
}
```

上例中 test1() 和 test2() 都被处理，不过在此之上 test2() 现在还属于组

"g1"。

5.10 - 并行运行于超时

你可以通过在 suite 标签中使用 *parallel* 属性来让测试方法运行在不同的线程中。这个属性可以带有如下这样的值：

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

- **parallel="methods"**: TestNG 会在不同的线程中运行测试方法，除非那些互相依赖的方法。那些相互依赖的方法会运行在同一个线程中，并且遵照其执行顺序。
- **parallel="tests"**: TestNG 会在相同的线程中运行相同的<test>标记下的所有方法，但是每个<test>标签中的所有方法会运行在不同的线程中。这样就允许你把所有非线程安全的类分组到同一个<test>标签下，并且使其可以利用 TestNG 多线程的特性的同时，让这些类运行在相同的线程中。
- **parallel="classes"**: TestNG 会在相同线程中相同类中的运行所有的方法，但是每个类都会用不同的线程运行。

此外，属性 *thread-count* 允许你为当前的执行指定可以运行的线程数量。

注意：@Test 中的属性 timeOut 可以工作在并行和非并行两种模式下。

你也可以指定 @Test 方法在不同的线程中被调用。你可以使用属性 threadPoolSize 来实现：

```
@Test(threadPoolSize = 3, invocationCount = 10,   timeOut = 10000)
public void testServer() {
```

上例中，方法 testServer 会在 3 个线程中调用 10 次。此外，10 秒钟的超时设定也保证了这三个线程中的任何一个都永远不会阻塞当前被调用的线程。

5.11 - 再次运行失败的测试

每次测试 suite 出现失败的测试，TestNG 就会在输出目录中创建一个叫做 testng-failed.xml 的文件。这个 XML 文件包含了重新运行那些失败测试的必要信息，使得你可以无需运行整个测试就可以快速重新运行失败的测试。所以，一个典型的会话看起来像：

```
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
```

要注意的是，testng-failed.xml 已经包含了所有失败方法运行时需要的依赖，所以完全可以保证上次失败的方法不会出现任何 SKIP。

5.12 - JUnit 测试

TestNG 能够运行 JUnit 测试。所有要做的工作就是在 testng.classNames 属性中设定要运行的 JUnit 测试类，并且把 testng.junit 属性设置为 true：

```
<test name="Test1"    junit="true">
   <classes>
      <!-- ... -->
```

TestNG 在这种情况下所表现的行为与 JUnit 相似：

- 所有类中要运行的测试方法由 test* 开头
- 如果类中有 setUp() 方法，则其会在每个测试方法执行前被调用
- 如果类中有 tearDown() 方法，则其会在每个测试方法之后被调用
- 如果测试类包含 suite() 方法，则所有的被这个方法返回的测试类都会被调用

5.13 - JDK 1.4

TestNG 也可以工作在 JDK 1.4 下。你需要使用 JDK 1.4 jar ，在发行版中可以找到（ 名为 *testng-...-jdk14.jar*）。唯一的不同就是注解的方式，它使用的流行的 XDoclet JavaDoc 注解语法：

```
public class SimpleTest {
  /**
   * @testng.before-class = "true"
   */
   public void setUp() {
      // code that will be invoked when this test is instantiated
  }
  /**
   * @testng.test groups = "functest" dependsOnGroups = "group1,group2"
   */
   public void testItWorks() {
      // your test code
  }
}
```

JavaDoc 语法的规则是相当的直接，并且与 JDK 1.5 注解唯一的不同就在于字符串数组被表示为单个逗号或者空格分隔的形式。然而，用双引号标注值时可选的，不过我极力推荐你使用双引号，以便以后 迁移到 JDK 1.5 会更容易些。

你也需要在使用 ant 任务中在 <testng> 标记中指定 sourcedir 属性 （或 -sourcedir 在命令行下），这样 TestNG 就能够按顺序找到测试文件，并且解析 JavaDoc 注解。

下标就是比较 JDK 1.4 和 JDK 5 注解的异同。

JDK 5	JDK 1.4
@Test(groups = { "a", "b" }, dependsOnMethods = { "m1" })	/** * @testng.test groups = "a b" * dependsOnMethods = "m1" */
@AfterMethod(alwaysRun = true)	/** * @testng.before-method alwaysRun = "true" */
@Parameters({ "first-name", "last-name" })	/** * @testng.parameters value = "first-name last-name" */
@Factory @Parameters({ "first-name", "last-name"})	/** * @testng.factory * @testng.parameters value = "first-name last-name" */
@DataProvider(name = "test1")	/** * @testng.data-provider name="test1" */

关于 TestNG's JDK 1.4 的例子和支持，请参看发行版中的 test-14/ 目录，其中包含了完整的 JDK 1.5 被映射到 JavaDoc 注解的说明。

5.14 - 程序化运行 TestNG

你可以在程序中非常轻松的调用 TestNG 的测试：

```
TestListenerAdapter tla = new TestListenerAdapter();
TestNG testng = new TestNG();
testng.setTestClasses(new Class[] { Run2.class });
testng.addListener(tla);
testng.run();
```

本例中创建了一个 TestNG 对象，并且运行测试类 Run2。它添加了一个

TestListener（这是个监听器）。你既可以使用适配器类 org.testng.TestListenerAdapter 来做，也可以实现 org.testng.ITestListener 接口。这个接口包含了各种各样的回调方法，能够让你跟踪测试什么时候开始、成功、失败等等。

类似的你可以用 testng.xml 文件调用或者创建一个虚拟的 testng.xml 文件来调用。为此，你可以使用这个包 org.testng.xml 中的类： XmlClass、XmlTest 等等。每个类都对应了其在 xml 中对等的标签。

例如，假设你要创建下面这样的虚拟文件：

```
<suite name="TmpSuite" >
   <test name="TmpTest" >
      <classes>
         <class name="test.failures.Child"   />
      <classes>
      </test>
</suite>
```

你需要使用如下代码：

```
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) ;
```

之后你可以传递这个 XmlSuite 给 TestNG：

```
List<XmlSuite> suites = new ArrayList<XmlSuite>();
suites.add(suite);
TestNG tng = new TestNG();
tng.setXmlSuites(suites);
tng.run();
```

请参阅 JavaDocs 来获取完整的 API。

5.15 - BeanShell 于高级组选择

如果 <include> 和 <exclude> 不够用，那就是用 BeanShell 表达式来决定是否一个特定的测试方法应该被包含进来。只要在 <test> 标签下使用这个表达式就好了：

```
<test name="BeanShell test">
     <method-selectors>
```

```
        <method-selector>
           <script language="beanshell"><![CDATA[
              groups.containsKey("test1")
           ]]></script>
        </method-selector>
     </method-selectors>
  <!-- ... -->
```

当在 testng.xml 文件中找到 <script> 标签后，TestNG 就会忽略在当前<test>标签内组和方法的的 <include> 和 <exclude> 标签： BeanShell 就会成为测试方法是否被包含的唯一决定因素。

下面是关于 BeanShell 脚本的额外说明：

- 必须返回一个 boolean 值。除了这个约束以外，任何有效的 BeanShell 代码都是允许的（例如，可能需要在工作日返回 true 而在周末返回 false，这样就允许你可以依照不同的日期进行测试）。
- TestNG 定义了如下的变量供你调用：
  java.lang.reflect.Method method: 当前的测试方法
  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映射
- 你也可能需要使用 CDATA 声明来括起 Beanshell 表达式（就像上例）来避免对 XML 保留字冗长的引用。

5.16 - 注解转换器

TestNG 允许你在运行时修改所有注解的内容。在源码中注解大多数时候都能正常工作的时时非常有用的（这句原文就是这意思，感觉不太对劲），但是有几个情况你可能会改变其中的值。

为此，你会用到注解转换器（ Annotation Transformer ）。

所谓注解转换器，就是实现了下面接口的类：

```
public interface IAnnotationTransformer {

   /**
     * This method will be invoked by TestNG to give you a chance
     * to modify a TestNG annotation read from your test classes.
     * You can change the values you need by calling any of the
     * setters on the ITest interface.
     *
     * Note that only one of the three parameters testClass,
     * testConstructor and testMethod will be non-null.
```

```
     *
     * @param annotation The annotation that was read from your
     * test class.
     * @param testClass If the annotation was found on a class, this
     * parameter represents this class (null otherwise).
     * @param testConstructor If the annotation was found on a constructor,
     * this parameter represents this constructor (null otherwise).
     * @param testMethod If the annotation was found on a method,
     * this parameter represents this method (null otherwise).
     */
   public void transform(ITest annotation, Class testClass,
         Constructor testConstructor, Method testMethod);
}
```

像其他的 TestNG 监听器，你可以指定在命令行或者通过 ant 来指定这个类：

```
   java org.testng.TestNG -listener MyTransformer testng.xml
```

或者在程序中：

```
   TestNG tng = new TestNG();
   tng.setAnnotationTransformer(new MyTransformer());
   // ...
```

当调用 transform() 的时候，你可以调用任何在 ITest test 参数中的设置方法来在进一步处理之前改变它的值。

例如，这里是你如何覆盖属性 invocationCount 的值，但是只有测试类中的 invoke() 方法受影响：

```
   public class MyTransformer implements IAnnotationTransformer {
      public void transform(ITest annotation, Class testClass,
            Constructor testConstructor, Method testMethod)
      {
         if ("invoke".equals(testMethod.getName())) {
            annotation.setInvocationCount(5);
         }
      }
   }
```

IAnnotationTransformer 只允许你修改一个 @Test 注解。如果你需要修改其他的（假设说配置注解 @Factory 或 @DataProvider），使用 IAnnotationTransformer2。

5.17 - 方法拦截器

一旦 TestNG 计算好了测试方法会以怎样的顺序调用，那么这些方法就会分为两组：

- *按照顺序运行的方法*。这里所有的方法都有相关的依赖，并且所有这些方法按照特定顺序运行。
- *不定顺序运行的方法*。这里的方法不属于第一个类别。方法的运行顺序是随机的，下一个说不准是什么（尽管如此，默认情况下 TestNG 会尝试通过类来组织方法）。

为了能够让你更好的控制第二种类别，TestNG 定义如下接口：

```
public interface IMethodInterceptor {

    List<IMethodInstance>  intercept(List<IMethodInstance>  methods,
ITestContext context);

}
```

方法中叫做 methods 的那个列表参数包含了所有以不定序运行的方法。你的 intercept 方法也要返回一个 IMethodInstance 列表,它可能是下面情况之一：

- 内容与参数中接收的一致，但是顺序不同
- 一组 IMethodInstance 对象
- 更大的一组 IMethodInstance 对象

一旦你定义了拦截器，就把它传递个 TestNG，用下面的方式：

```
java   -classpath   "testng-jdk15.jar:test/build"   org.testng.TestNG
-listener test.methodinterceptors.NullMethodInterceptor \
   -testclass test.methodinterceptors.FooTest
```

关于 ant 中对应的语法，参见 listeners 属性 ant 文档 中的说明。

例如，下面是个方法拦截器会重新给方法排序，一遍“fast”组中的方法总是先执行：

```
public List<IMethodInstance> intercept(List<IMethodInstance> methods,
ITestContext context) {
   List<IMethodInstance> result = new ArrayList<IMethodInstance>();
   for (IMethodInstance m : methods) {
      Test test = m.getMethod().getMethod().getAnnotation(Test.class);
      Set<String> groups = new HashSet<String>();
      for (String group : test.groups()) {
         groups.add(group);
      }
      if (groups.contains("fast")) {
         result.add(0, m);
      }
      else {
         result.add(m);
      }
   }
```

```
    return result;
}
```

5.18 - 从 JavaDoc 注解迁移到 JDK 注解

如果项目伊始使用了 javadoc annotations，但是之后要转换到 JDK annotations，这样需要把所有这些都进行转换。

TestNG 提供了一个工具来帮你做这些。

java org.testng.AnnotationConverter -srcdir *directory* [-overwrite|-d *destdir*] [-quiet]
上述命令会把源文件从一个格式转换到另外一个。如果不带参数运行，就会显示所有参数的用法。

注意，转换后的文件，可能不是非常漂亮，所以要检查一下其正确性。所以最好用版本控制工具，如果不喜欢可以撤销操作!转换器本身也假设你使用了上面 所推荐的语法规约。

5.19 - TestNG 监听器

有很多接口可以用来修改 TestNG 的行为。这些接口都被统称为 "TestNG 监听器"。下面是目前支持的监听器的列表：

- IAnnotationTransformer (doc, javadoc)
- IReporter (doc, javadoc)
- ITestListener (doc, javadoc)
- IMethodInterceptor (doc, javadoc)
- IInvokedMethodListener (doc, javadoc)

当你实现了这些接口，你可以让 TestNG 知道这些接口，有如下方式：

- 在命令行下使用 -listener
- ant 中使用 <listeners>
- 在 testng.xml 中使用 <listeners> 标签

下面是第三种方式的例子：

```
<suite>

<listeners>
   <listener class-name="com.example.MyListener" />
```

```
    <listener class-name="com.example.MyMethodInterceptor" />
</listeners>

...
```

5.20 - 依赖注入

TestNG 允许你在自己的方法中声明额外的参数。这时，TestNG 会自动使用正确的值填充这些参数。依赖注入就使用在这种地方：

- 任何 @Before 或 @Test 方法可以声明一个类型为 ITestContext 的参数。
- 任何 @After 都可以声明一个类型为 ITestResult 的单数，它代表了刚刚运行的测试方法。
- 任何 @Before 和 @After 方法都能够声明类型为 XmlTest 的参数，它包含了当前的 <test> 参数。
- 任何 @BeforeMethod 可以声明一个类型为 java.lang.reflect.Method 的参数。这个参数会接收 @BeforeMethod 完成调用的时候马上就被调用的那个测试方法当做它的值。
- 任何 @BeforeMethod 可以声明一个类型为 Object[] 的参数。这个参数会包含要被填充到下一个测试方法中的参数的列表，它既可以又 TestNG 注入，例如 java.lang.reflect.Method 或者来自 @DataProvider。
- 任何 @DataProvider 可以声明一个类型为 ITestContext 或 java.lang.reflect.Method 的参数。后一种类型的参数，会收到即将调用的方法作为它的值。

5.21 - 监听方法调用

无论何时 TestNG 即将调用一个测试（被@Test 注解的）或者配置（任何使用 @Before or @After 注解标注的方法），监听器 IInvokedMethodListener 都可以让你得到通知。你所要做的就是实现如下接口：

```
public interface IInvokedMethodListener extends ITestNGListener {
    void beforeInvocation(IInvokedMethod method, ITestResult testResult);
    void afterInvocation(IInvokedMethod method, ITestResult testResult);
}
```

并且就像在 关于 TestNG 监听器一节 中所讨论的那样，将其声明为一个监听器。

6 - 测试结果

6.1 - 成功、失败和断言

如果一个测试没有抛出任何异常就完成运行或者说抛出了期望的异常（参见 @Test 注解的 expectedExceptions 属性文档），就说，这个测试时成功的。

测试方法的组成常常包括抛出多个异常，或者包含各种各样的断言（使用 Java "assert" 关键字）。一个 "assert" 失败会触发一个 AssertionErrorException，结果就是测试方法被标记为失败（记住，如果你看不到断言错误，要在加上 -ea 这个 JVM 参数）。

下面是个例子：

```
@Test
public void verifyLastName() {
   assert "Beust".equals(m_lastName) : "Expected name Beust, for" +
m_lastName;
}
```

TestNG 也包括 JUnit 的 Assert 类，允许你对复杂的对象执行断言：

```
import static org.testng.AssertJUnit.*;
//...
@Test
public void verify() {
   assertEquals("Beust", m_lastName);
}
```

注意，上述代码使用了静态导入，以便能够使用 assertEquals 方法，而无需加上它的类前缀。

6.2 - 日志与结果

当运行 SuiteRunner 的时候会指定一个目录，之后测试的结果都会保存在一个在那个目录中叫做 index.html 的文件中。这个 index 文件指向其他多个 HTML 和文本文件，被指向的文件包含了整个测试的结果。你可以再 这里 看到一个例子。

通过使用监听器和报表器，可以很轻松的生成自己的 TestNG 报表：

- **监听器** 实现接口 org.testng.ITestListener ，并且会在测试开始、通过、失败等时刻实时通知

- **报告器** 实现接口 org.testng.IReporter ，并且当整个测试运行完毕之后才会通知。IReporter 接受一个对象列表，这些对象描述整个测试运行的情况

例如，如果你想要生成一个 PDF 报告，那么就不需要实时通知，所以用 IReporter。如果需要写一个实时报告，例如用在 GUI 上，还要在每次测试时（下面会有例子和解释）有进度条或者文本报告显示点（"."），那么 ITestListener 是你最好的选择。

6.2.1 一 日志监听器

这里是对每个传递进来的测试显示"."的监听器，如果测试失败则显示 "F" ，跳过则是"S"：

```
public class DotTestListener extends TestListenerAdapter {
   private int m_count = 0;

   @Override
   public void onTestFailure(ITestResult tr) {
      log("F");
   }

   @Override
   public void onTestSkipped(ITestResult tr) {
      log("S");
   }

   @Override
   public void onTestSuccess(ITestResult tr) {
      log(".");
   }

   private void log(String string) {
      System.out.print(string);
      if (m_count++ % 40 == 0) {
         System.out.println("");
      }
   }
}
```

上例中，我们选择扩展 TestListenerAdapter ，它使用空方法实现了 ITestListener 。所以我不需要去重写那些我不需要的方法。如果喜欢可以直接实现接口。

这里是我使用这个新监听器调用 TestNG 的例子：

```
java -classpath testng.jar;%CLASSPATH% org.testng.TestNG -listener
org.testng.reporters.DotTestListener test\testng.xml
```
输出是：
```
.....................................
.....................................
.....................................
.....................................
.....................................
.....................
===============================================
TestNG JDK 1.5
Total tests run: 226, Failures: 0, Skips: 0
===============================================
```
注意，当你使用 -listener 的时候，TestNG 会自动的检测你所使用的监听器类型。

6.2.2 — 日志报表

org.testng.IReporter 接口只有一个方法：

public void generateReport(List<ISuite> suites, String outputDirectory)
这个方法在 TestNG 中的所有测试都运行完毕之后被调用，这样你可以方法这个方法的参数，并且通过它们获得刚刚完成的测试的所有信息。

6.2.3 — JUnit 报表

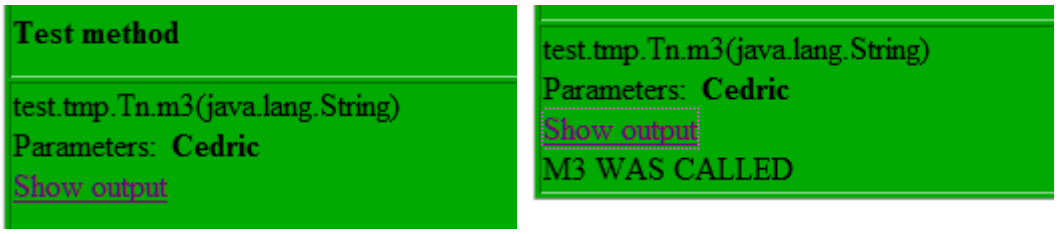
TestNG 包含了一个可以让 TestNG 的结果和输出的 XML 能够被 JUnitReport 所使用的监听器。这里 有个例子，并且 ant 任务创建了这个报告：

```
<target name="reports">
   <junitreport todir="test-report">
      <fileset dir="test-output">
         <include name="*/*.xml"/>
      </fileset>

      <report format="noframes"   todir="test-report"/>
   </junitreport>
</target>
```
注意： 由于 JDK 1.5 和 JUnitReports 不兼容性，导致了 frame 版本不能够正常工作，所以你需要指定 "noframes" 使其能够正常工作。

6.2.4 — 报表 API

如果你要在 HTML 报告中显示日志信息，那么就要用到类 org.testng.Reporter：

```
Reporter.log("M3 WAS CALLED");
```



6.2.5 — XML 报表

TestNG 提供一种 XML 报表器，使得能够捕捉到只适用于 TestNG 而不适用与 JUnit 报表的那些特定的信息。这在用户的测试环境必须要是用 TestNG 特定信息的 XML，而 JUnit 又不能够提供这些信息的时候非常有用。下面就是这种报表器生成 XML 的一个例子：

```
<testng-results>
      <suite name="Suite1">
            <groups>
                  <group name="group1">
                        <method signature="com.test.TestOne.test2()"
name="test2" class="com.test.TestOne"/>
                        <method signature="com.test.TestOne.test1()"
name="test1" class="com.test.TestOne"/>
                  </group>
                  <group name="group2">
                        <method signature="com.test.TestOne.test2()"
name="test2" class="com.test.TestOne"/>
                  </group>
            </groups>
            <test name="test1">
                  <class name="com.test.TestOne">
                        <test-method status="FAIL" signature="test1()"
name="test1" duration-ms="0"
                                                started-at="2007-05-28T12
:14:37Z" description="someDescription2"
                                                finished-at="2007-05-28T1
2:14:37Z">
                                    <exception
class="java.lang.AssertionError">
                                          <short-stacktrace>java.lang.Asser
tionError
```

```
                                        ... Removed 22 stack frames
                                    </short-stacktrace>
                                </exception>
                            </test-method>
                            <test-method status="PASS" signature="test2()"
name="test2" duration-ms="0"
                                         started-at="2007-05-28T12
:14:37Z" description="someDescription1"
                                         finished-at="2007-05-28T1
2:14:37Z">
                            </test-method>
                            <test-method status="PASS" signature="setUp()"
name="setUp" is-config="true" duration-ms="15"
                                         started-at="2007-05-28T12
:14:37Z" finished-at="2007-05-28T12:14:37Z">
                            </test-method>
                        </class>
                    </test>
            </suite>
</testng-results>
```

这个报表器是随着默认监听器一起诸如的，所以你默认的情况下就可以得到这种类型的输出。这个监听器提供了一些属性，可以修改报表来满足你的需要。下表包含了这些属性，并做简要说明：

属性	说明	Default value
outputDirectory	一个 String 指明了 XML 文件要存放的目录	TestNG 输出目录
timestampFormat	指定报表器生成日期字段的格式	yyyy-MM-dd'T'HH:mm:ss'Z'
fileFragmentationLevel	值为 1，2 或 3 的整数，指定了 XML 文件的生成方式： 1 - 在一个文件里面生成所有的结果 2 - 每套测试都单独生成一个 XML 文件，这些文件被链接到一个主文件 3 - 同 2，不过添加了引用那些套测试的测试用例的 XML 文件	1
splitClassAndPackageNames	boolean 值，指明了对 <class> 元素的生成方式。例如，你在 false 的时候得到 <class class="com.test.MyTest"> ，在 true 的时候 <class class="MyTest" package="com.test"> 。	false
generateGroupsAttribute	boolean 值指定了对<test-method> 元素是否生成 groups 属性。这个功能的目的是，对那些无需遍历整个<group> 元素，且只含有一个测试方法的组来说，可以提供一种更直接的解析组的方式。	false
stackTraceOutputMethod	指定在发生异常的时候生成异常，追踪弹栈信息的类型，有如下可选值： 0 - 无弹栈信息 (只有异常类和消息).	2

	1 - 尖端的弹栈信息，从上到下只有几行 2 - 带有所有内部异常的完整的弹栈方式 3 - 短长两种弹栈方式	
generateDependsOnMethods	对于<test-method>元素，使用这个属性来开启/关闭 depends-on-methods 属性。	true
generateDependsOnGroups	对于<test-method>属性开启 depends-on-groups 属性。	true

为了配置报表器，你可以在命令行下使用 -reporter 选项，或者在 Ant 任务中嵌入<reporter> 元素。对于每个情况，你都必须指明类 org.testng.reporters.XMLReporter。但是要注意你不能够配置内建的报表器，因为这个只使用默认配置。如果你的确需要 XML 报表，并且使用自定义配置，那么你就不得不手工完成。可以通过自己添加一两个方法，并且禁用默认监听器达到目的。

从 JUnit 迁移

使用 JUnitConverter

通过使用 org.testng.JUnitConverter，可以轻松的从 JUnit 迁移过来。调用方式如下：

- -annotation | -javadoc (必须的)
  若有 -annotation，JUnitConverter 会插入 JDK5 的注解，如果使用了 -javadoc，工具会生成 JavaDoc 注解。
- -srcdir | -source <fileName> (必须的)
  -srcdir 所指定的目录会被迭代遍历，并且任何以.java 结束的文件都会被处理。如果使用 -source，只有指定的 Java 文件会被处理。
- -d | -overwrite (强制的)
  默认情况下，JUnitConverter 会修改它找到的文件。如果用这个选项来指定某个目录，那么被修改的文件会创建在这个指定的目录中。如果使用 -overwrite，则直接修改源码。
- -quiet
  JUnitConverter 就不会在控制台上打印出任何信息。

下面是把在 src/ 目录下所有 JUnit 的测试类转化为使用 TestNG 的例子：

```
java org.testng.JUnitConverter -overwrite -annotation -srcdir src
```

注意：

- *JUnitConverter 使用了来自 tools.jar 的类，它位于 $JAVA_HOME/lib/tools.jar，所以要保证这个 jar 文件在你的类路径下。*
- *JUnitConverter 只能运行于 JDK5，所以如果有错误（例如 "unknown -quiet parameter"），要保证你的类路径仅仅包含 JDK5 的 jar 文件。*

JUnitConverter 也包含 ant 任务，可以这样调用：

```
<project name="test" default="init">
   <target name="init">
      <taskdef resource="testngtasks" />
   </target>

   <target name="junitconvert" depends="init">
      <junit-converter sourcedir="C:\dev\projects\test\java"
outputdir="C:\dev\projects\temp" annotations="false" />
   </target>
</project>
```

断言

注意类 org.testng.Assert 使用不同的方式来排序，而不是 JUnit 所使用的那样。如果你的代码使用了 JUnit 的断言，就需要在类中使用静态导入：

```
import static org.testng.AssertJUnit.*;
```