[**TestNG官方文档中文版(1)-介绍**](http://www.blogjava.net/aoxj/archive/2008/03/18/187105.html)

最近决定开始使用TestNG来替代junit，作为未来的单元测试基础组件。
    google了一下没有找到好的详细的文档，最后发现官网上的文档其实挺全的，决定翻译一下。
    TestNG的官方文档请见： http://testng.org/doc/documentation-main.html

1. 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）。

    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

在欢迎页面上可以找到快速入门示例。

下面是这篇文档使用的概念：

    \* 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

    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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

注：上面的内容很简短，但是请注意其中的一些细节。

1. 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。
    可以用于集成测试，这个特性是我选择TestNG的最重要的原因。

2. 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。

3. 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官方文档中文版(2)-annotation**](http://www.blogjava.net/aoxj/archive/2008/03/19/187380.html)

    TestNG的官方文档的中文翻译版第二章，原文请见 http://testng.org/doc/documentation-main.html

2 - Annotation

这里是TestNG中用到的annotation的快速预览，还有它们的属性。

@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。
属性：
    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

@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[]

@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
    successPercentage     当前方法期望的成功率
    sequential     如果设置为true，当前测试类上的所有方法保证按照顺序运行。甚至测试们在parallel="true"的情况下.
            这个属性只能用于类级别，如果用于方法级别将被忽略。
    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官方文档中文版(3)-testng.xml**](http://www.blogjava.net/aoxj/archive/2008/03/19/187382.html)

    TestNG的官方文档的中文翻译版第3章，原文请见 http://testng.org/doc/documentation-main.html

    3 - testng.xml

    调用TestNG由几种不同方法：

    \* 使用testng.xml文件
    \* 使用ant
    \* 从命令行

    这节描述testng.xml的格式（文档的后面会讲到ant和命令行）。

    当前testng.xml的DTD文件可以从官方找到：http://testng.org/testng-1.0.dtd。（为了方便使用，你可能更喜欢浏览HTML版本）。
    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。

[**TestNG官方文档中文版(4)-运行TestNG**](http://www.blogjava.net/aoxj/archive/2008/03/22/187938.html)

4 - 运行TestNG

TestNG可以以不同的方式调用：

    \* Command line
    \* ant
    \* 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

选项        参数                文档说明
-d        一个目录            生成报告的目录（ test-output）
-sourcedir    分号隔开的目录列表        带有javadoc注释的测试源文件目录. 这个选项只在使用javadoc类型的annotation时才有效.
                        (例如 "src/test" or "src/test/org/testng/eclipse-plugin;src/test/org/testng/testng").
-testclass    可以在classpath路径中找到的逗号分隔的类列表。逗号分隔的类文件列表(例如 "org.foo.Test1,org.foo.test2").
-groups        逗号分隔的组列表        要运行的组列表(例如 "windows,linux,regression").
-excludegroups    逗号分隔的组列表        不想包含在这次运行中的组列表
-testrunfactory    可以在classpath中找到的java类    指定测试的runner.这个类需要实现接口org.testng.ITestRunnerFactory .
-listener    可以在classpath路径中找到的逗号分隔的类列表。    指定测试的listener. 这个类需要实现接口org.testng.ITestListener
-parallel    methods|tests            如果指定, 设置运行测试时如何使用并发线程的默认机制.如果不设置，默认机制是完全不使用并发线程。这个设置可以被suite定义覆盖.
-threadcount    并发测试运行时默认使用的线程数    用于设置并发测试时默认的线程数. 只在并发模式被选择时才生效 (例如, 打开 -parallel 选项). 这个设置可以被suite定义覆盖.
-suitename    测试套件使用的默认名称.        指定在命令行上定义的测试套件的名称。如果suite.xml文件或源代码指定了另外一个不同的套件名称，这个选项将被忽略。可以创建带空格的套件名称，如果在名称前后加双引号如"like this".
-testname    测试使用的默认名称.        指定在命令行上定义的测试的名称。如果suite.xml文件或源代码指定了另外一个不同的测试名称，这个选项将被忽略。可以创建带空格的测试名称，如果在名称前后加双引号如"like this".
-reporter    扩展配置用于自定义报告listenner.    类似 -listener 选项, 除了容许reporter示例上由javabean形式的配置.
    例如: -reporter com.test.MyReporter:methodFilter=\*insert\*,enableFiltering=true

可以通过不带任何参数直接调用TestNFG来获得这个文档。

可以将命令行开关写到txt文件中，例如c:\command.txt, 然后告诉TestNG使用这个文件类找到参数:

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

另外，可以通过jvm的命令行来传递参数给TestNG，例如

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

TestNG能够理解的参数
属性             类型                     文档
testng.test.classpath     分号分隔的包含测试类的一系列目录     如果这个属性被设置，TestNG将使用它替代从class path来查找测试类. 如果你正在使用在xml文件里面的包标签并且在classpath路径中由很多类而大部分都不是测试类的时候比较方便

举例:

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

注意 ant 任务和testng.xml容许用更多的参数来启动TestNG（包含的方法，指定的参数，等等），因此可以认为命令行适用于学习TestNG并且想快速入门。

2) Ant

可以这样定义TestNG的ant任务：

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

这个任务运行TestNG测试，并且通常是在单独的jvm中。接受下面的属性：

属性名            描述                是否必须
annotations         字符串"JDK"或者"Javadoc". 定义测试适用的注释类型.如果使用"Javadoc", 则需要同时指定"sourcedir".     不是必须. 如果适用jkd5则默认为"JDK"，如果适用jdk1.4则默认为"Javadoc"
classfilesetref     要运行的测试类的FileSet结构的引用.
classpath         要运行的测试的PATH-like 结构.
classpathref         要运行的测试的PATH-like 结构的引用.
dumpCommand         打印TestNG启动命令.     不是必须,默认false
enableAssert         开启JDK 1.4的断言.     不是必须,默认true
failureProperty     失败发生时要设置的属性的名称. 只有haltonfailure没有设置时才有效.     不是必须.
haltonfailure         如果测试运行期间发生失败，停止构造过程.     不是必须,默认false
haltonskipped         如果发生至少一次测试跳过，停止构造过程.        不是必须,默认false
groups             要运行的组列表，空格或逗号分隔
excludedgroups         排除在外的组列表，空格或逗号分隔
jvm             使用的jvm，将被Runtime.exec()运行     java
listeners         逗号或空格分隔的全路径类列表，需要实现org.testng.ITestListener或org.testng.IReporter     不是必须
outputdir         报告输出目录             不是必须,默认输出到test-output.
skippedProperty     当发生测试被跳过时设置的property的名称.只有当haltonskipped没有设置时才使用     不是必须
sourcedir         用于jdk1.4测试的PATH-like结构(使用JavaDoc形式的annotations)
sourcedirref         用于jdk1.4测试的PATH-like结构的引用(使用JavaDoc形式的annotations)
suiteRunnerClass     TestNG启动器的全路径名称    不是必须.  默认使用org.testng.TestNG
parallel         运行测试时使用的并行模式 - methods或者tests     不是必须 - 如果没有指定，并行模式不被选择
threadCount         运行时使用的线程数量。如果并行模式被同时指定否则忽略。     默认1
testJar         包含测试和套件定义的jar包路径
timeOut         所有测试必须运行完成的最大超时时间，单位毫秒
useDefaultListeners     是否使用默认监听器和报告器.     默认true.
workingDir         运行TestNG前ant任务应该转移到的目录。
xmlfilesetref        用于要测试的套件定义的FileSet结构的引用
suitename         设置测试套件的默认名称如果在suite的xml文件或者源代码中都没有被定义。    不是必须，默认设置为"Ant suite"
testname        设置测试的默认名称如果在suite的xml文件或者源代码中都没有被定义。    不是必须，默认设置为"Ant test"

属性classpath, classpathref或者内嵌的<classpath>必须设置一个，用于提供测试的classpath

属性xmlfilesetref, classfilesetref 或者内嵌的 <xmlfileset>, 分别的<classfileset>必须使用用来提供测试

注意：如果使用jdk1.4，属性attributes sourcedir, sourcedirref 或者内嵌的 <sourcedir> 必须提供.

注意：使用<classfileset> 并不自动按添加测试类到classpath: 需要报告这些在classpath中的任务要工作的类

内嵌元素
classpath
 <testng> 任务支持一个内嵌的<classpath> 元素来提供PATH-like的结构.

bootclasspath
bootstrap类文件的位置可以用这个PATH形式的结构指定-如果fork没有设置则被忽略

xmlfileset
套餐定义（testng.xml）可以通过一个FiltSet结构传递给任务

classfileset
TestNG可以直接在类上运行，同样支持FiltSet结构

sourcedir
PATH形式的结构，用于jdk1.4的测试，使用javadoc annotation

jvmarg
通过内嵌的<jvmarg>元素将额外的参数传递给新的虚拟机，例如：

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

sysproperty
使用内嵌的<sysproperty>元素来指定类需要的系统属性。在测试的执行期间虚拟机可以获取这些属性。 这个元素的属性和环境变量相同。

<testng>
   <sysproperty key="basedir" value="${basedir}"/>
   <!-- ... -->
</testng>
将运行测试并且使得测试可以访问basedir属性

reporter

内部的<reporter>元素是一个可选的方式，用于注入自定义的报告监听器，容许用户为调整运行时的报告期行为而
这个元素强制要求设置classname属性，指示自定义监听器的类。为了设置报告期属性，<reporter>元素可以包含多个内嵌的<property>元素来提供name和value属性，如下所示：

<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) {...}
  ...
}
请注意这里仅仅支持有限的属性类型：String, int, boolean, byte, char, double, float, long, short.

env
可以通过内嵌的 <env>元素给TestNG的单独的虚拟机传递指定的环境变量。
要查阅<env> 元素属性的详细描述，请查看ant的exec任务的描述。

举例：
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>

[**TestNG官方文档中文版(5)-测试方法/类和组**](http://www.blogjava.net/aoxj/archive/2008/03/23/188013.html)

5 - Test methods, Test classes and Test groups
5.1 - 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":

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

<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：
@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：

<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：

<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）
5.2 - 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".

<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>

5.3 - 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：

@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：

<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，

5.4 - 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：

@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".

[**TestNG官方文档中文版(6)-参数**](http://www.blogjava.net/aoxj/archive/2008/09/02/226434.html)

5.5 - Parameters

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。

  5.5.1 - Parameters from 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：

@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">
  <-- ![E:\Documents and Settings\fred.fanj\Local Settings\Temp\CyberArticle\af8d9545cde1241506a39d4e8fd58edf_1032_files\dot[1].gif]() -->

同样的方法可以用于注解@Before/After和@Factory:

@Parameters({ "datasource", "jdbcDriver" })
@BeforeMethod
public void beforeTest(String ds, String driver) {
  m\_dataSource = ![E:\Documents and Settings\fred.fanj\Local Settings\Temp\CyberArticle\af8d9545cde1241506a39d4e8fd58edf_1032_files\dot[1].gif]();                              // 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) { ![E:\Documents and Settings\fred.fanj\Local Settings\Temp\CyberArticle\af8d9545cde1241506a39d4e8fd58edf_1032_files\dot[1].gif]() }

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。

  5.5.2 - Parameters with 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
@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
@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。

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) {
    // ![E:\Documents and Settings\fred.fanj\Local Settings\Temp\CyberArticle\af8d9545cde1241506a39d4e8fd58edf_1032_files\dot[1].gif]()
  }
}

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。

Here is an example of this feature for both JDK 1.4 and JDK5 (note that the JDK 1.4 example does not use Generics):
这里有一个同时适用于JDK 1.4和JDK5的例子（注意JDK 1.4的例子不使用注解）：

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

@DataProvider(name = "test1")
public Iterator createData() {
  return new MyIterator(DATA);
}

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:

@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

  5.5.3 - Parameters in reports

被用于调用测试方法的参数将在TestNG生成的HTML报告中显示。实例如下：
![E:\Documents and Settings\fred.fanj\Local Settings\Temp\CyberArticle\af8d9545cde1241506a39d4e8fd58edf_1032_files\parameters[1].png]()

[**TestNG官方文档中文版(7)-方法依赖和工厂**](http://www.blogjava.net/aoxj/archive/2008/09/02/226462.html)

5.6 - 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。

5.7 - 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：

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>

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：

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：

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）。

[**TestNG官方文档中文版(8)-类级别注解和并发**](http://www.blogjava.net/aoxj/archive/2008/09/05/227103.html)

5.8 - Class level annotations

@Test注解可以放置在类上：

@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:

@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".

5.9 - 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。

[**TestNG官方文档中文版(9)-重复失败测试,junit测试,jdk1.4支持，编程式调用和Beanshell**](http://www.blogjava.net/aoxj/archive/2008/09/05/227223.html)

5.10 - 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。

5.11 - 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>
    <!-- ![E:\Documents and Settings\fred.fanj\Local Settings\Temp\CyberArticle\af8d9545cde1241506a39d4e8fd58edf_1035_files\dot[1].gif]() -->

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。

5.12 - JDK 1.4

TestNG也可以在JDK1.4下工作。在这种情况下，需要使用发布的jdk1.4的jar文件（名为testng-...-jdk14.jar）。唯一的差别是在于注解，jdk1.4下使用流行的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语法的规则非常简洁，和jdk1.5注解的唯一差别是数组串数组需要特别写成单独的，逗号或空格分隔的字符串。虽然值周围的双引号是可选的，但还是建议在任何情况下都使用双引号，以保证将来迁移到jdk1.5时可以比较容易。

同样需要在<testng>的ant任务中指明sourcedir属性(或者在命令行中使用-sourcedir），以便testNG可以找到你的测试文件的源代码来解析javadoc注解。

这里是jdk1.4和jdk5注解的语法对照表：

    （表格在blog中不好排版，不在这里发了，详细内容请参考官方文档的原文：http://testng.org/doc/documentation-main.html#jdk-14。）

更多jdk1.4的支持范例，请参考发行包中的test-14文件夹，这里包含全部的JDK 1.5测试对应的使用javadoc注解的内容。

5.13 - 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。

5.14 - 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：

  <test name="BeanShell test">
   <method-selectors>
     <method-selector>
       <script language="beanshell"><![CDATA[
         groups.containsKey("test1")
       ]]></script>
     </method-selector>
   </method-selectors>
  <!-- ![E:\Documents and Settings\fred.fanj\Local Settings\Temp\CyberArticle\af8d9545cde1241506a39d4e8fd58edf_1035_files\dot[1].gif]() -->

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。