jTester 快速上手

版本	时间	作者	变更说明
1.0	2013-11-23	colddew	基于 2011-11-29 版 jTester 使用指南、jTester 项目托管在 googcode 上的历史版本、jTester 基于的多个开源测试框架的官方文档等资料整理
1.1	2015-01-28	colddew	jTester 由 1.1.8 升级到 1.1.9 版本，修复了部分 bug 并增强了对 sqlserver 的支持，对文档中过时的内容进行了更新

Preface

熟悉 java 单元测试的同学应该能体会到,对 java 程序如果只是单纯的使用 junit 或 testng 这样的基础单元测试框架往往很难应对各种复杂的单元测试情况,通常需要借助很多第三方的框架和技术（easymock、jmock、dbunit 等)。而这些框架和技术的学习又会增加学习成本和难度，所以有人在这些 java 基础单元测试工具的基础上开发出一些测试框架(如 unitils)将多种 java 单元测试技术整合在一起，以提高开发效率。

而 jTester 也是一个基于 java 的单元测试框架,他集成了众多优秀的开源框架,如:junit、testng、dbunit、unitils、jmockit 等，并在这些框架的基础上进行了扩展，使得单元测试更加方便和强大。它不但能帮助我们提高开发效率，节省了大量学习成本，也避免了我们陷入需要甄别众多测试框架的泥潭。

项目的原作者是阿里的一名架构师，jtester-1.1.8 版本之前的源码托管在 googlecode 平台上，目前该项目已经更名并迁移到其他平台。本文基于 jtester-1.1.9 版本，该版本修复了 jtester-1.1.8 版本部分 bug 并增强了对 sqlserver 的支持。该版本基本可以满足目前我们对单元测试框架的需求，帮助我们应对断言、数据库测试、spring 集成、mock 等应用场景。

本文旨在帮助开发人员快速掌握 jTester 的基本操作，讲解一些 jTester 使用指南的关键内容、没有提到的但是比较重要的内容、过时的需要注意更新的内容，更多详细用法还需要参阅 jTester 使用指南或网上搜索相关资源。

另开源软件存在问题在所难免，如果遇到问题或者发现 bug，欢迎大家提出我们共同完善该框架，同样如果本文中有说错的地方也欢迎指正。目前本人维护的 jtester 分支源码托管在 github 上(https://github.com/colddew/jtester),工程依赖的中央仓库中不存在的第三方 jar 包放在 lib 目录下，感兴趣的同学可以 fork 下源码。本文涉及的 demo 同样也托管在 github 上（https://github.com/colddew/jtester-demo)，demo 中有更加丰富的例子以及文档，可以作为学习 jTester 的参考。

Hello World

这里只介绍用 maven 构建的过程，非 maven 的项目只要将下面描述的依赖 jar 包加入 build path 中即可。如果 jtester 已经上传到私服，开发人员只需要在 pom 中引用下面这段依赖即可，其它的依赖项会自动加载。

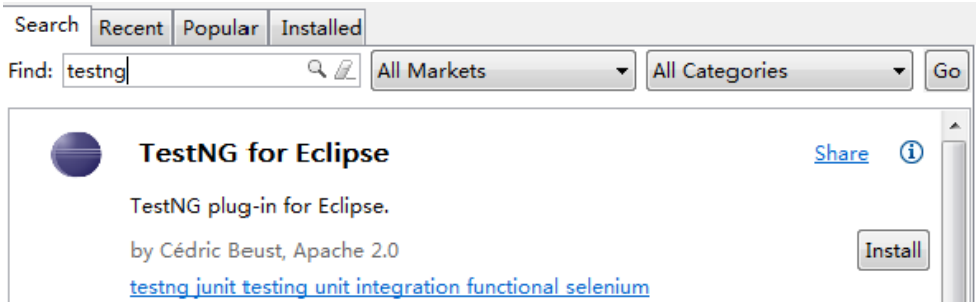
```
<dependency>
    <groupId>org.jtester</groupId>
    <artifactId>jtester</artifactId>
    <version>1.1.9</version>
    <scope>test</scope>
</dependency>
```

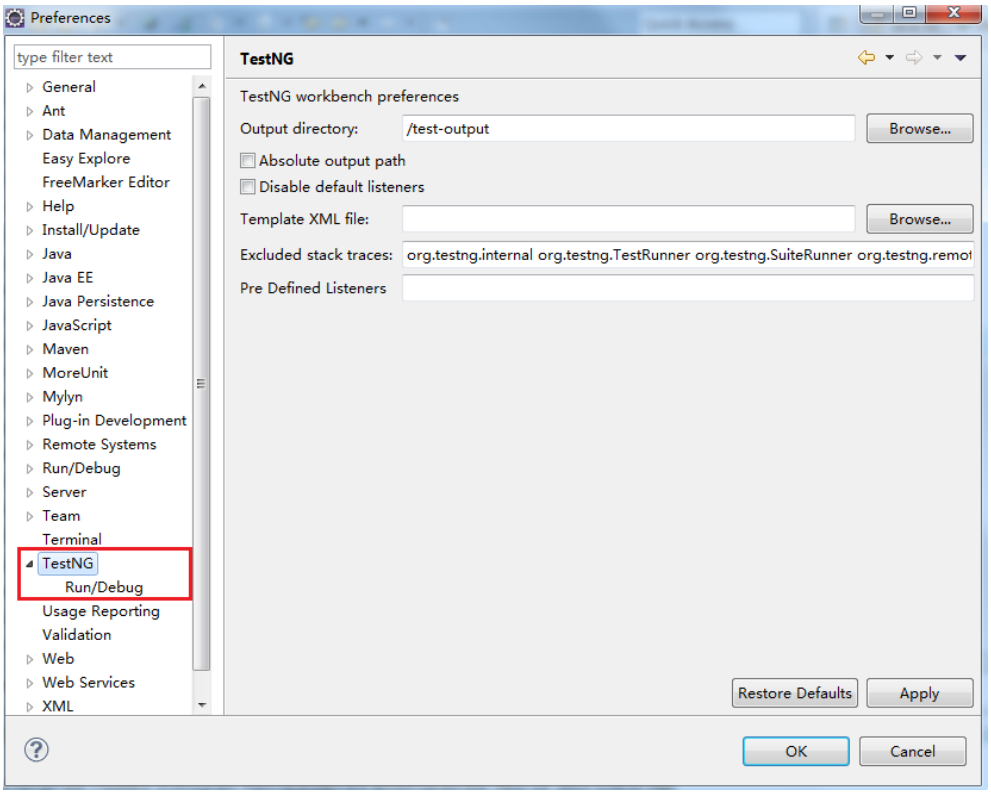
当然没有私服也没关系，按照下面的步骤可以达到同样的效果，首先需要将工程依赖的中央仓库中不存在的第三方 jar 包发布到我们的本地仓库。在命令行只要执行类似下面这段命令即可，使用时需要将标红处替换成自己的依赖坐标和地址，坐标可以随意写，只要 pom 里引入的依赖和这里定义的一致就可以了。

```
mvn install:install-file -DgroupId=com.oracle -DartifactId=jdbc.oracle -Dversion=10.2.0.3.0
-Dpackaging=jar -Dfile=H:/测试/jtester-jar/jdbc.oracle-10.2.0.3.0.jar
```

这样就可以将第三方 jar 包（fitnesse-20100211.jar、jdbc.oracle-10.2.0.3.0.jar）导入本地仓库。当然还有其它的方式，比如手工在本地仓库里建依赖的目录，然后将 jar 包 copy 进去，不过有时这样想法简单的方式也能快速地帮我们达到目的，比如添加源文件时如果本地仓库里目录结构存在了，直接把源码的 jar 包 copy 进去就可以了。最后我们需要在 clone 的 jtester 工程的根目录执行 mvn install 将 jtester-1.1.9 发布到本地仓库中，这样我们就可以像本节开始那样引入 jTester 了。

因为我们使用了 testng 进行单元测试，需要在 eclipse 里安装 testng 插件，只需要在 eclipse marketplace 里选择安装即可，安装成功后 preferences 就可以看见 testng 了。

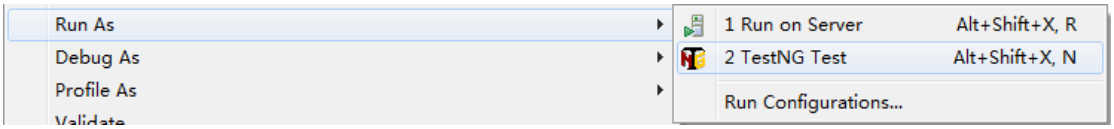




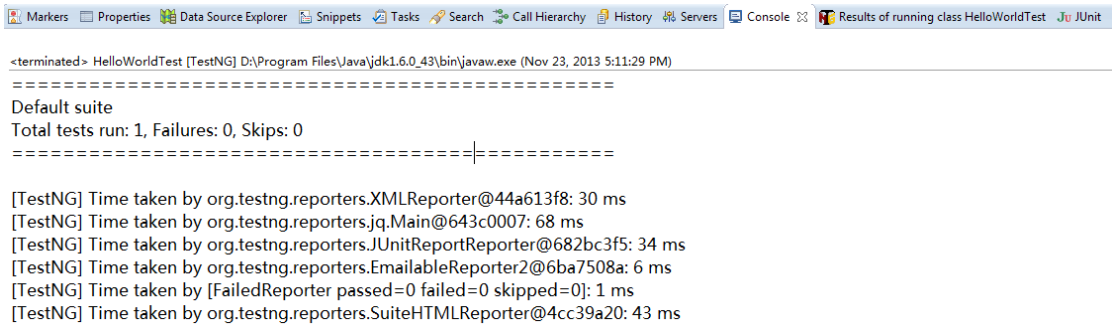
准备工作做好了，先来看下 jTester 的 hello world 代码，只需要继承自 JTester，下面的 annotation 是不是很眼熟，不过这个是 testng 的注解，但这里也可以看出从 junit 切换到 testng 是很容易的事，不需要花很多学习成本。

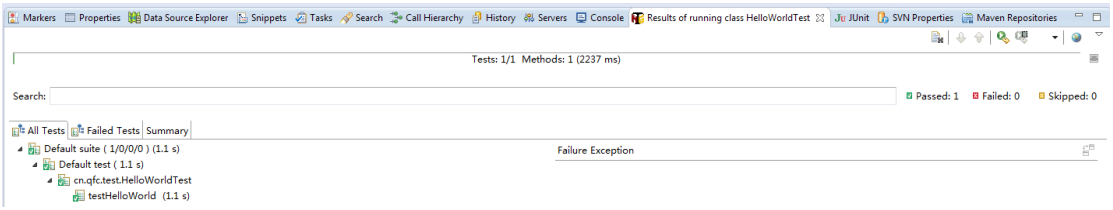
```
public class HelloWorldTest extends JTester {
    @Test
    public void testHelloWorld() {
        System.out.println("hello world");
    }
}
```

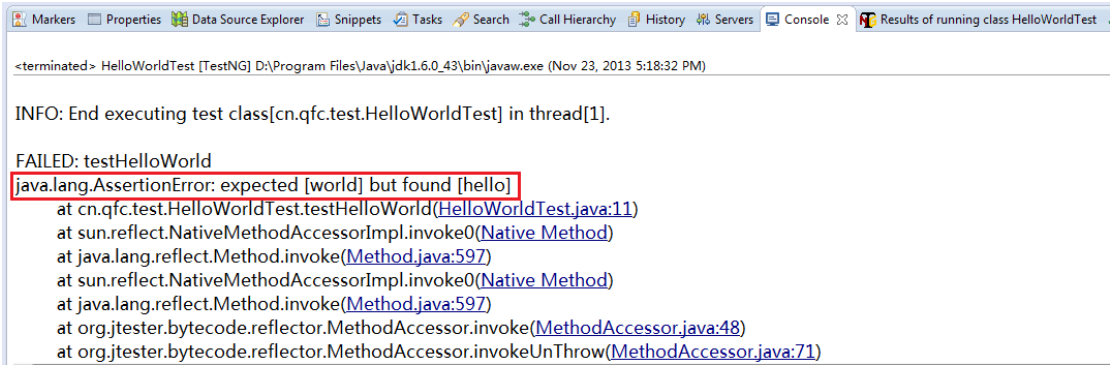
执行测试也很 easy，只要在类的任何一个地方右键，在 run as 里选择 testng test 即可。

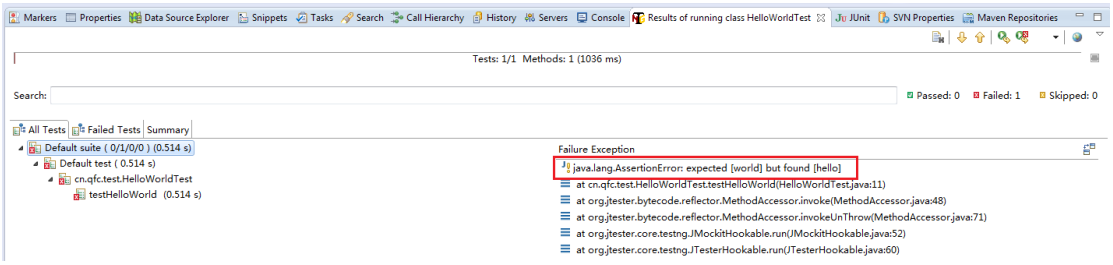


下面分别显示了执行结果是成功和失败的情况，在控制台和 testng 视图显示的情况，testng 视图中的显示和 junit 视图是类似的，这里值得一提的是 console 里显示的内容，如果断言是错误的，不仅会打印错误的堆栈信息，还会给出详细的错误提示帮助我们定位问题，而 junit 断言错误时 console 是没有信息的，我觉得这是 testng 做的比 junit 好的地方。










```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <testFailureIgnore>false</testFailureIgnore>
        <suiteXmlFiles>
            <suiteXmlFile>testng.xml</suiteXmlFile>
        </suiteXmlFiles>
    </configuration>
</plugin>
```

如果需要用 maven 命令来执行 testng 测试，还需要在 pom 中配置 maven-surefire-plugin 插件，上面标红的为测试配置文件，testng.xml 里指定了需要测试的类和方法，suiteXmlFile 文件路径从项目根目录开始算起。这里还有一点需要注意的是，由于我们目前的项目基本都是使用了 jdk1.6 以上的版本,所以 jTester 使用指南中关于 jmockit 虚拟机参数的设置就不需要了。

Why testng

提到 ng 估计很多人第一反应会想到是拍电影的名词，但此 ng 非彼 ng，其英文全拼是 next generation，为啥取这个名字从官网了解到的情况是因为作者在使用 junit 的时候各种不爽，最终导致了他决定重新造轮子。个人体会 testng 确实有一些好用的地方是 junit 所没有的，而且一般一个开源项目支持 junit 往往也会同时支持 testng，可见其影响力还是有一些的。

testng 也是采用 annotation 分别标识测试的方法、测试方法前后执行的方法、测试类前后执行的方法。测试异常也很简单，通过注解不但可以验证异常类型，而且可以很简单的验证异常内容，想想 junit 是不是费劲了点。

```
public class ExceptionTest extends JTester {
    @Test(expectedExceptions = RuntimeException.class,
            expectedExceptionsMessageRegExp = "this is a exception")
    public void testException() {
        throw new RuntimeException("this is a exception");
    }
}
```

```
public class SuiteTest extends JTester {
    @Test(groups={"g1"})
    public void testDemo1() {
        System.out.println("################ jtester demo1 ################");
    }
    @Test(groups={"g2"})
    public void testDemo2() {
        System.out.println("################ jtester demo2 ################");
    }
}
```

```
<suite name="suite-testng">
    <test name="suite-testng">
        <groups>
            <run>
                <include name="g1" />
                <exclude name="g2" />
            </run>
        </groups>
        <packages>
            <package name="cn.qfc.*" />
        </packages>
    </test>
</suite>
```

另一个好用的特性是分组测试，junit 中没有这个概念，testng 可以针对不同的测试目的

进行分组，先给测试方法加@Test(groups={"g1"})注解，使其添加到一个组里，然后将需要进行测试的类和方法加入我们之前提到的 testng.xml 文件中，其他没有分组的方法或者被排除的分组将不会被测试到，一个使用场景比如我们借助单元测试来调试远程方法调用的方法最终是需要被排除的，由于这些方法都是没有断言的，不能作为一个完整的单元测试存在的。分组测试还有一个好处，就是为日后引入持续集成做好了准备工作，因为需要测试的类和方法我们都已经在配置文件中定义好了。

Assert

junit、testng 和 jdk 都提供了断言的功能，但这些功能都是相对较弱。jtester 基于 hamcrest 提供了丰富的断言功能，使得我们可以轻松的针对字符串、基本数据类型、对象、Map、集合类进行测试。

```
@Test
public void testString() {
    String actual = "qfc-test-cfq";
    want.string(actual).eq("qfc-test-cfq");
    want.string(actual).any();
    want.string(actual).notNull().notBlank();
    want.string(actual).in("qfc-test-cfq", "qfc-test2-cfq");
    want.string(actual).regular("^qfc.*cfq$");
}
```

```
@Test
public void testBasicType() {
    Integer i = new Integer(1);
    Long l = new Long(2);
    Boolean b = Boolean.TRUE;
    Date date2 = new Date(1);
    BigDecimal bd = new BigDecimal(10);

    want.number(i).eq(1).isGt(0).isLt(2).isBetween(0, 2);
    want.number(l).eq(2L).isGt(1L).isLt(3L).isBetween(1L, 3L);
    want.bool(b).is(true);
    want.date(date2).eq(new Date(1));
    want.number(bd).eq(new BigDecimal(10)).isGt(new BigDecimal(9))
        .isLt(new BigDecimal(11)).isBetween(new BigDecimal(9), new BigDecimal(11));
}
```

上面的代码是对字符串和基本数据类型进行的一些断言操作，标红的就是所谓的流式断言，这是作者介绍该框架时强调的地方，好处就是操作起来比较连贯一致。但是也有人评论说这种方式不好的，大概的意思可能是语句太长引起混乱之类的，这也是见仁见智的吧。具体的断言方法比较多，使用 eclipse 的代码提示就能够让我们猜出个大概，详细的例子请参阅 jTester 使用指南。另外一点需要注意的是，jtester 对日期类型的断言可能存在一些问题，有的时候精确到微秒可能会提示你断言失败，有的时候又会采取宽松的断言，只要比较的日期都不为空就认为是正确的，这个使用时要多加小心，或者避免对日期进行断言。

DB Test

如果单元测试依赖数据库中的数据，一旦数据库改变或者清空，单元测试就可能不通过了，这就是传说的一锤子买卖。针对数据库的单元测试，通常应该使用独立的数据库，测试前会清空待测试的表以避免数据干扰（清空数据不是绝对的，比如不太变化的基础数据就不需要清空）。准备测试数据可以使用多种方式，目前主要的开源数据库测试框架，通常都是使用独立的文件准备测试数据的，比如：xml、excel、wiki 等，频繁地编辑数据和查看数据让我们觉得很麻烦。jtester 使用 DataMap 来准备和验证数据，使我们可以用一致方式来准备数据和测试，避免反复在各种编辑器之间来回的切换，这也是我觉得 jtester 最好用的地方。jtester 支持 mysql、oracle、sqlserver 等数据库，编写测试前我们需要先准备好配置文件 jtester.properties 放在 classpath 路径下，通常放在 src/test/resources 目录下即可。

下面为 sqlserver 的配置，这在 jTester 使用指南中没有相关的说明，其他两种数据库的配置都可以找得到。这里有一个属性 database.only.testdb.allowing 的配置要特别小心，如果连接数据库的地址是远程的就会提示你配置 database.only.testdb.allowing=false，这主要是为了确保我们连接的是用于单元测试的数据库，避免误删数据库中的重要数据库。

```
##sqlserver remote
#database.type=sqlserver
#database.url=jdbc:jtds:sqlserver://192.168.200.20:1433/qfc_pay_test
#database.userName=qfc_pay
#database.password=H6d8J9(&52u(^DJ
#database.schemaNames=qfc_pay_test
#database.driverClassName=net.sourceforge.jtds.jdbc.Driver
#database.only.testdb.allowing=false
```

```
    db.table("user2").clean().insert(new DataMap() {
        {
            this.put("username", "jtester");
            this.put("password", "jtester");
            this.put("datee", "2011-11-11 11:11:11");
            this.put("bd", new BigDecimal("99.99"));
        }
    }).commit();
    db.table("user2").insert("{'username':'jtester2','password':'jtester2','datee':'2013-11-05
11:11:11','bd':'8.88'}").commit();
    db.table("user2").count().eq(2);
    db.table("user2").query().propertyEqMap(2, new DataMap() {
        {
            this.put("bd", new BigDecimal("99.99"), new BigDecimal("8.88"));
            this.put("datee", "2011-11-11 11:11:11", "2013-11-05 11:11:11");
            this.put("password", "jtester", "jtester2");
            this.put("username", "jtester", "jtester2");
        }
    });
```

准备好配置文件后，我们来看一个例子基本就可以理解数据库测试是如何进行的了。上面的测试首先清空了待测试的表，然后分别采用两种方式向表中插入了两条数据，最后对数据库中的数据进行了断言。上面的 clean 操作是为了保证当前测试不受历史数据的影响，就是因为这个操作才使得我们在配置数据库连接时要格外小心。

上面的插入和验证操作中的测试字段都是可选的，这使得我们可以轻松的只准备和验证我们关心的数据，其他数据由框架帮我们自动生成。还有一点要注意的是如果使用 oracle 数据库，断言时的字段必须使用大写的。jTester 使用指南中还提供多种更加复杂的使用方式，如果实在看不下去了没必要机械地都一次看完，其中 20%的基本方法就可以帮我们解决 80%的问题了，如果遇到不能解决的问题再去查文档就可以了。

Spring Integrated

项目中使用 spring 的场景很多，jtester 提供了一系列的 annotation 对 spring 进行封装方便我们的使用。下面的例子中@SpringApplicationContext 是用于加载 spring 配置文件的，通常在 src/test/resources 包下放置有和 src/main/resources/applicationContext.xml 类似的配置文件，但这些配置文件和我们开发环境的配置文件应该是独立开的。依赖 bean 的注入使用@SpringBeanByName，如果同一个接口有多个实现类，还可以在这个注解里通过 claz 属性指定具体的实现类，@SpringBeanByType 是和这个类似的。如果测试里需要有大量需要注入的 bean，可以使用@AutoBeanInject 利用框架帮助我们自动注入，但需要我们配置注入规则，jTester 使用指南中有详细的讲解，从中我们也可以看到开发时接口和实现类的摆放位置，或者说包结构定义规范和合理的重要性，否则注入规则的配置会一团乱。

由此引申出一点需要注意的是，由于我们的项目是基于 maven 构建的，业务和测试的源码、资源文件都有对应的目录结构，因此业务类和测试类的路径最好保持一致，即 src/main/java 和 src/test/java 后面的文件路径一致，比如业务类的完整路径为 A.B.C.***Service，对应的测试类路径就应为 A.B.C.***ServiceTest。这样不仅看起来很清晰，方便我们查找定位，而且有些测试工具能帮我们自动检测出哪些类和方法做了测试，哪些没有做，这些是题外话了。

```
@SpringApplicationContext({"applicationContext.xml"})
public class SpringTest extends JTester {
    @SpringBeanByName
    private Phone phone;
    @Test
    public void testSpringBean() {
        want.object(phone).propertyEq("home", "jtester");
        want.object(phone).propertyEq("office", "jtester");
    }
}
```

如果需要注入的 bean 对测试环境有较强依赖比较难加载时，@SpringBeanFrom 可以给我们提供一些帮助。我们可以直接 new 出一个实例来，或者通过 mock 造出一个我们期望的实例来，下面的例子就演示了 mock 掉一个接口的方式，mock 是啥东东后面会有介绍。

```
    @SpringBeanFrom
    ResourceManager resourceManager;
    @BeforeMethod
    public void initMockResourceManager(){
        resourceManager = new MockUp<ResourceManager>(){
            @Mock
            public List<Option> getOptionList(String resName){
                return values.get(resName);
            }
        }.getMockInstance();
    }
```

Mock

单元测试主要是验证当前模块内部逻辑的正确性，或者说我们应该保持单元测试的独立性，不依赖于外部接口和方法，降低对外部的耦合，这样的话如果测试失败了，肯定是由于当前模块的问题造成的。

当然单元测试的粒度是很难拿捏的，这与开发人员的素质和水平是有关的，甚至还会受开发人员写单元测试时主观意识的影响,但有些简单的原则我们是可以遵守的。比如远程方法调用、action 层都是不需要纳入单元测试范畴的，远程方法调用属于集成测试的范畴，应该由测试人员来写自动化测试脚本，如果纳入单元测试则粒度太粗；action 层如果写的规范的话是不应该包含任何业务逻辑的，因此也没有必要写单元测试了。

言归正传，使用 mock 技术可以帮助我们将当前模块和其他模块或系统隔离开，还可以现实许多现实环境无法重现的场景。所谓 mock 就是对所调用的代码模拟其行为返回一个值，例如下面的场景：我们测试的方法中需要调用某个 api 方法，但由于环境问题 api 调不通或者该 api 干脆还未实现，这时我们又不想影响我们主要业务逻辑的测试，我们就需要使用 mock 技术。

jtester 是基于 jmockit 开源项目来实现 mock 的，jmockit 被奉为用来做 mock 的神器，从其官方提供的资料可以看出其貌似是无所不能的，无论是测 abstract、final、private、static 的类，亦或构造方法，亦或接口，亦或异常，只要你想不到没有测不了。

Feature	EasyMock	jMock	Mockito	Unitils Mock	PowerMock: EasyMock API	PowerMock: Mockito API	JMockit
Invocation count constraints	√	√	√		√	√	√
Recording strict expectations	√	√			√		√
Explicit verification			√	√		√	√
Partial mocking	√		√	√	√	√	√
No method call to switch from record to replay			√	√		√	√
No extra code for implicit verification			*N/A*	*N/A*		*N/A*	√
No extra "prepare for test" code	√	√	√	√			√
No need to use `@RunWith` annotation or base test class	√	√	√				√
Consistent syntax between `void` and non-`void` methods		√		√			√
Argument matchers for some parameters only, not all				√			√
Easier argument matching based on properties of value objects	√		√	√	√	√	√
Cascading mocks			√	√		√	√
Support for mocking multiple interfaces			√			√	√
Support for mocking annotation types		√	√	√		√	√
Partially ordered expectations		√					√
Mocking of constructors and *final*/*static*/*native*/*private* methods					√	√	√
Declarative application of mocks/stubs to whole test classes					√	√	√
Auto-injection of mocks			√	√		√	√
Mocking of "new-ed" objects					√	√	√
Support for mocking enum types					√	√	√
Declarative mocks for the test class (mock fields)			√	√	√	√	√
Declarative mocks for test methods (parameters, local fields)							√
Special fields for "any" argument matching							√
Use of an special field to specify invocation results							√
Use of special fields to specify invocation count constraints							√
Expectations with custom error messages							√
On-demand mocking of unspecified implementation classes							√
Capture of instances created by code under test							√
Recording & verification of expectations in loops							√
Support for covariant return types							√
"Duck typing" mocks for state-based tests							√
Single jar file in the classpath is sufficient to use mocking API			√		*N/A*	*N/A*	√
Total	6/32	7/32	13/31	11/31	9/31	14/30	32/32
Total when ignoring JMockit-only features	6/22	7/22	13/21	11/21	9/21	14/20	22/22

jmockit 分为基于行为和基于状态的测试，基于行为的 mock 是站在目标测试代码外部的角度，通常主要模拟行为，jtester 中使用 Expectations 和 Verifications 实现。基于状态的 mock

是站在目标测试代码内部的角度，可以对传入的参数进行检查、匹配，返回某些结果，jtester中使用MockUp<GenericType>实现。对于两种测试方式，我们一般都是使用内部类来实现。

下面的几段代码分别演示了对接口、调用次数、方法输入参数、异常等进行测试的使用方法，采用基于状态的测试，基本上都是一个套路，mock、调用方法、断言，只是先后顺序有调整而已，代码可读性很好，无需过多解释，这里有一点需要注意的是.getMockInstance()只在mock接口时需要使用。

```
@Test
public void testInterface() {
    ITestService testService = new MockUp<ITestService>() {
        @Mock
        public String service() {
            return "interface";
        }
    }.getMockInstance();
    String service = testService.service();
    want.string(service).eq("interface");
}
```

```
@Test
public void testParameter() {
    new MockUp<TestServiceImpl>() {
        @Mock(invocations = 1)
        public void service3(Phone phone) {
            want.object(phone).propertyEq("home", "parameter")
                .propertyEq("office", "parameter");
        }
    };
    Phone phone = new Phone();
    phone.setHome("parameter");
    phone.setOffice("parameter");
    new TestServiceImpl().service3(phone);
}
```

```
@Test(expectedExceptions = RuntimeException.class,
        expectedExceptionsMessageRegExp = "exception")
public void testException() {
    new Expectations() {
        {
            testService2.service();
            result = new RuntimeException("exception");
        }
    };
    new TestServiceImpl().service4(testService2);
}
```

下图展示的是基于行为的测试执行模型，Record 和 Verify 分别对应 jtester 中的 Expectations 和 Verifications 的 api，Expectations 中定义了 mock 方法的调用次数和返回值，可以定义多个 Expectations，执行测试的时候就必须严格按照上面定义的顺序和次数执行测试，Verifications 对于严格的期望 Expectations 是默认执行的，无需显示调用，如果需要显示的验证 Verifications 通常使用非严格的期望 NonStrictExpectations 录制期望。

下面演示了基于行为的测试过程，完全吻合上面的模型，不再赘述。值得一提的是@Mocked 是对需要 mock 对象的注解，写在这里是限定其作用域只在方法内部。当然也可以对整个测试类的成员变量添加@Mocked 注解，只是其作用范围更大，如果多个方法都引用该成员变量可能带来不必要的麻烦，使用的时候多注意。详细的 mock 使用方法请参阅 jTester 使用指南以及 jmockit 的官方文档，目前 jmockit 项目已经从 googlecode 托管平台迁移到 github 上了。

```
@Test
public void testSimpleMock(@Mocked final TestServiceImpl testServiceImpl) {
    // 录制期望
    new Expectations(testServiceImpl) {
        {
            testServiceImpl.service();
            times = 2;
            result = "simple";

            testServiceImpl.service2();
            result = "simple";
        }
    };
    // 执行测试（严格按照录制的顺序和次数进行）
    String service = testServiceImpl.service();
    testServiceImpl.service();
    String service2 = testServiceImpl.service2();
    // 验证结果
    want.string(service).eq("simple");
    want.string(service2).eq("simple");
}
```