

# jTester使用指南

吴大瑞

---

# jTester使用指南

吴大瑞

出版日期 2011-11-29

---

# 目录

1. 安装jtester .....	1
maven用户安装 .....	1
ant用户安装 .....	1
安装TestNG插件 .....	1
编写一个简单的测试用例 .....	2
maven方式运行 .....	4
2. TestNG语法简介 .....	6
TestNg注解介绍 .....	6
TestNg 参数化测试 .....	8
异常测试 .....	10
TestNg分组测试和suite文件编写 .....	12
3. jTester断言介绍 .....	14
什么是测试断言? .....	14
jTester断言基本介绍 .....	16
jTester断言语法详解 .....	18
断言String对象 .....	19
Java基本类型的断言 .....	24
对普通PoJo对象进行断言 .....	25
断言Map对象 .....	33
断言Collection对象或Array对象 .....	35
4. 使用DataMap准备和验证数据 .....	39
为什么要使用DataMap? .....	39
配置jtester.properties文件 .....	40
DataMap语法详解 .....	42
对指定表进行数据插入操作 .....	43
对指定表进行数据验证操作 .....	50
其他数据库操作 .....	52
多数据库测试 .....	53
使用eclipse插件 .....	55
5. 在测试中集成Spring .....	61
加载spring容器 .....	61
@AutoBeanInject让框架自动查找和注册需要的bean .....	63
@AutoBeanInject规则详解 .....	65
Spring Bean依赖项查找规则 .....	66
特殊实现类的bean注册 .....	66
自动注册的bean如何实现spring的init-method方法 .....	67
声明bean的简单属性 .....	71
使用@SpringBeanFrom DIY你需要的bean .....	74
Spring模块注解 .....	76
6. 反射调用私有方法或JDK代理的方法 .....	78
调用私有方法 .....	80
访问私有变量 .....	82
使用反射方式构造对象实例 .....	83
7. 在测试代码中使用Mock .....	85
静态mock, new MockUp的使用 .....	85
mock构造函数和静态代码块 .....	87
new MockUp和spring的集成 .....	88
针对静态mock做断言 .....	90
动态mock, new Expectations的使用 .....	94

---

## 表格清单

1. 文档编订历史 .....	
-----------------	--

---

## 范例清单

1.1. jtester maven配置 .....	1
1.2. 一个简单测试用例写法 .....	2
1.3. maven-surefire-plugin配置示例 .....	4
2.1. TestNG的简单测试 .....	6
2.2. TestNG生命周期Annotation使用 .....	7
2.3. 参数化测试示例 .....	10
2.4. 异常测试代码一 .....	11
2.5. 异常测试代码二 .....	11
2.6. 非异常测试应该将异常直接抛到测试方法上 .....	12
2.7. 对测试进行分组 .....	12
2.8. 指定若干组的suite文件 .....	13
3.1. 判断字符串等于期望值 .....	19
3.2. 忽略大小写时字符串比较 .....	20
3.3. 忽略空格时字符串比较 .....	20
3.4. 忽略单双引号时字符串比较 .....	20
3.5. 所有空格等价时字符串比较 .....	21
3.6. 单双引号等价时字符串比较 .....	21
3.7. 单双引号等价时字符串比较 .....	21
3.8. 使用多个模糊判断模式进行字符串复合断言 .....	21
3.9. 判断字符串中包含指定的子字符串 .....	22
3.10. 判断字符串中依次包含若干子字符串 .....	22
3.11. 判断字符串不包含子字符串 .....	22
3.12. 判断字符串以指定的子字符串开头 .....	23
3.13. 判断字符串以指定的子字符串结尾 .....	23
3.14. 判断一个字符串是否符合正则表达式 .....	23
3.15. 判断一个字符串是否是若干个候选字符串中的一个 .....	23
3.16. 判断一个字符串是空字符串, 非空字符串, 非空白字符串等情况 .....	24
3.17. 字符串是任意值均可通过的断言 .....	24
3.18. 对布尔值进行断言 .....	24
3.19. 对Character(char)进行断言 .....	25
3.20. 对数值类型进行断言 .....	25
3.21. 判断2个PoJo对象是否相等 .....	25
3.22. 反射比较2个PoJo对象 .....	27
3.23. 反射比较2个PoJo对象, 忽略特定字段比较示例 .....	27
3.24. 反射比较2个PoJo对象, 忽略数组(集合)属性中元素顺序 .....	29
3.25. 忽略默认值时比较2个对象 .....	29
3.26. 忽略元素顺序时比较2个对象 .....	29
3.27. 忽略默认值, 日期, 元素顺序时比较2个对象 .....	29
3.28. 判断2个对象是否是同一个对象 .....	30
3.29. 普通对象的单属性比较 .....	30
3.30. 对PoJo对象String属性的宽松断言 .....	30
3.31. 使用延迟生效断言判断PoJo对象的属性 .....	31
3.32. 使用EqMode判断对象属性 .....	31
3.33. propertyEqMap (老版本reflectEqMap) 比较多个属性值 .....	31
3.34. propertyEqMap (老版本reflectEqMap) 使用EqMode模式比较多个属性值 .....	32
3.35. 级联属性的比较 .....	33
3.36. 通过反射比较断言Map对象 .....	33
3.37. 通过指定属性断言Map .....	34
3.38. 断言map对象元素个数 .....	35
3.39. 断言map对象key值或value值 .....	35
3.40. 集合大小或数组长度的断言 .....	36
3.41. 断言集合包含特定元素 .....	36
3.42. 断言集合包含一系列元素 .....	36

3.43. 断言集合包含指定元素之一 .....	37
3.44. 断言列表中所有元素符合所有的断言要求 .....	37
3.45. 断言列表中所有元素必须都必须满足至少一个断言器 .....	37
3.46. 列表中任意元素符合所有的断言 .....	38
3.47. 列表中任意元素符合断言 .....	38
3.48. 对列表中元素进行属性断言 .....	38
4.1. 往数据库中插入一条数据 .....	39
4.2. 验证数据库中已存在的数据（单条数据） .....	40
4.3. mysql配置示例 .....	41
4.4. oracle配置示例 .....	41
4.5. 往数据库中插入多条数据(示例一) .....	46
4.6. 数据库中插入多条数据(示例二) .....	46
4.7. 数据自定义生成示例 .....	46
4.8. 插入数据的综合示例 .....	47
4.9. 以json的方式插入数据 .....	47
4.10. 验证多条数据的示例(有序比较) .....	50
4.11. 验证多条数据的示例(无序比较) .....	51
4.12. 带条件的数据查询验证 .....	51
4.13. 执行SQL语句示例一(单条sql语句) .....	52
4.14. 执行SQL语句示例二（多条sql语句） .....	52
4.15. 使用SqlSet结构批量执行sql语句 .....	52
4.16. 执行SQL文件 .....	52
4.17. 多数据库时的jtester.properties文件配置 .....	53
4.18. 多数据库的切换示例一 .....	54
4.19. 多数据库的切换示例二 .....	54
5.1. 加载Spring容器示例(使用classpath路径) .....	61
5.2. 加载Spring容器示例(使用文件路径) .....	61
5.3. 使用@SpringBeanByName把容器中的bean注入到代码中 .....	62
5.4. @AutoBeanInject方式加载spring示例 .....	64
5.5. 自动注册bean时，显式的指定特殊实现类示例 .....	67
5.6. 显式指明bean的init-method属性 .....	68
5.7. 显式指明bean的init-method属性(使用@SpringBeanByName init属性) .....	69
5.8. 显式指明bean的init-method属性(使用@SpringInitMethod注解) .....	69
5.9. 显式指明bean的init-method属性(使用@SpringInitMethod注解) .....	70
5.10. 在spring容器初始化前mock示例 .....	71
5.11. 通过@SpringBeanByName初始化实现类属性 .....	72
5.12. 通过@SpringBeanByName实现引用其它bean对象的功能 .....	73
5.13. 通过@SpringBeanByName实现引用其它bean对象的功能 .....	73
5.14. 通过@SpringBeanFrom初始化实现类中复杂属性的值 .....	74
5.15. @SpringBeanFrom使用示例（new实例） .....	75
5.16. @SpringBeanFrom使用示例（mock对象） .....	75
5.17. @SpringBeanFrom使用示例（动态赋值） .....	76
6.1. 私有方法测试示例 .....	80
6.2. 反射调用时，显式指定方法的声明类 .....	81
6.3. 反射调用静态方法示例 .....	81
6.4. 反射方式给变量赋值和取值 .....	82
6.5. 反射方式给变量赋值和取值(显示声明变量的声明类) .....	83
6.6. 反射方式给静态变量赋值和取值 .....	83
6.7. 使用反射方式构造实例 .....	84
6.8. 从json串中构造实例 .....	84
7.1. 使用new MockUp静态mock的一个简单例子 .....	86
7.2. 使用new MockUp mock父类中定义的方法 .....	87
7.3. 如何mock构造函数 .....	88
7.4. 如何mock静态代码块 .....	88
7.5. new Mock和spring的集成-实现类 .....	89
7.6. new Mock和spring的集成-接口类 .....	89

7.7. new Mock和spring的集成-@SpringInitMethod使用 .....	90
7.8. 在new MockUp的mock方法中断言传入的参数值 .....	92
7.9. 限制new MockUp中mock方法被调用的次数 .....	94

---

# 第 1 章 安装jtester

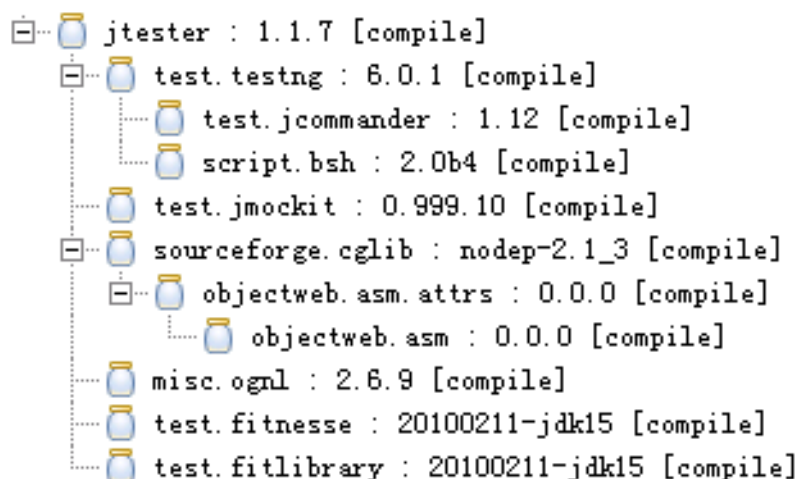
## maven用户安装

如果你是maven构建的项目，使用jtester非常简单，配置下列maven依赖项就可以了。

### 例 1.1. jtester maven配置

```
<dependencies>
<dependency>
<groupId>com.alibaba.crm.shared</groupId>
<artifactId>jtester</artifactId>
<version>1.1.8</version>
<scope>test</scope>
</dependency>
</dependencies>
```

jTester显式依赖了下列jar，只要你声明了上面的依赖定义，这些jar自动会被依赖进来。



其他spring, ibatis, mysql驱动(oracle驱动)等jar不是jTester必须的包，由具体的应用自己加载。

用maven声明jTester依赖时，请将scope设置成test，因为这些包只有测试周期才会用到，设置test模式，可以避免把这些对应用无关的jar打入到war中，或传递依赖。

## ant用户安装

如果你是ant用户，就必须自己一个个去下载对应的jar包，自己管理。

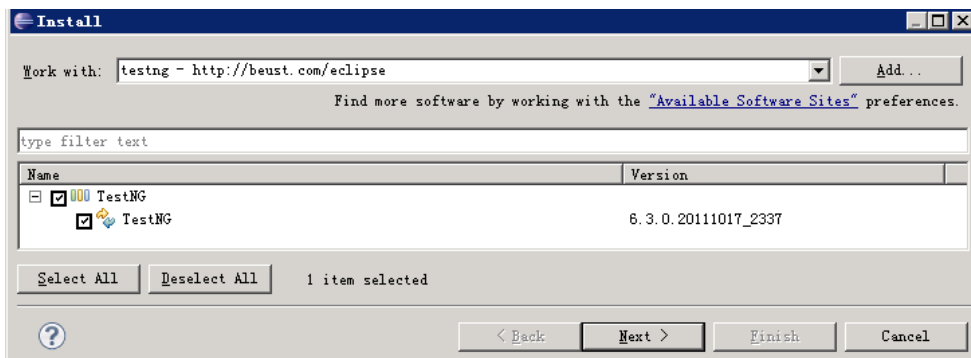
## 安装TestNG插件

因为jTester默认是基于TestNG开发的，在配置好jTester jar后，要想运行jTester的测试，你还需要安装TestNG的eclipse插件

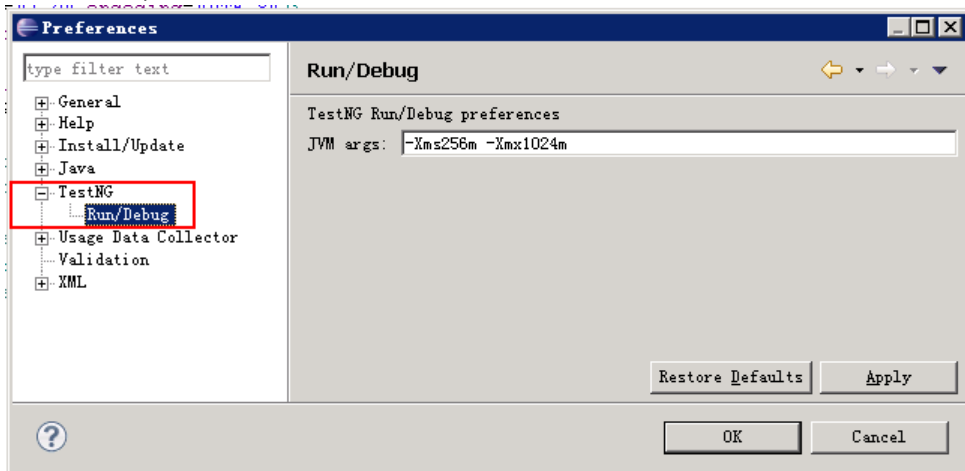
TestNG plugin update site: <http://beust.com/eclipse>

大家在eclipse的Help->Install New Software菜单中进行更新就可以：





安装完毕，在eclipse的Window->Preferences菜单中会出现TestNG的选项：



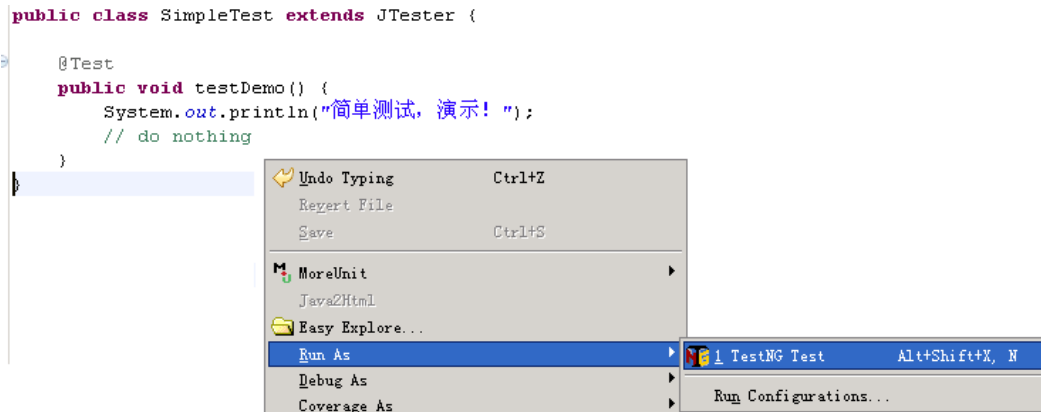
## 编写一个简单的测试用例

现在我们可以开始写一个简单的测试用例。

### 例 1.2. 一个简单测试用例写法

```
public class SimpleTest extends JTester {  
  
    @Test  
    public void testDemo() {  
        System.out.println("简单测试, 演示!");  
        // do nothing  
    }  
}
```

测试类SimpleTest要继承基类JTester，测试方法是一个public void 类型的无参方法。并且在该方法上需要标注@Test，告诉TestNG这是一个测试方法。编写完毕，我们在测试类的java视图中右键菜单：



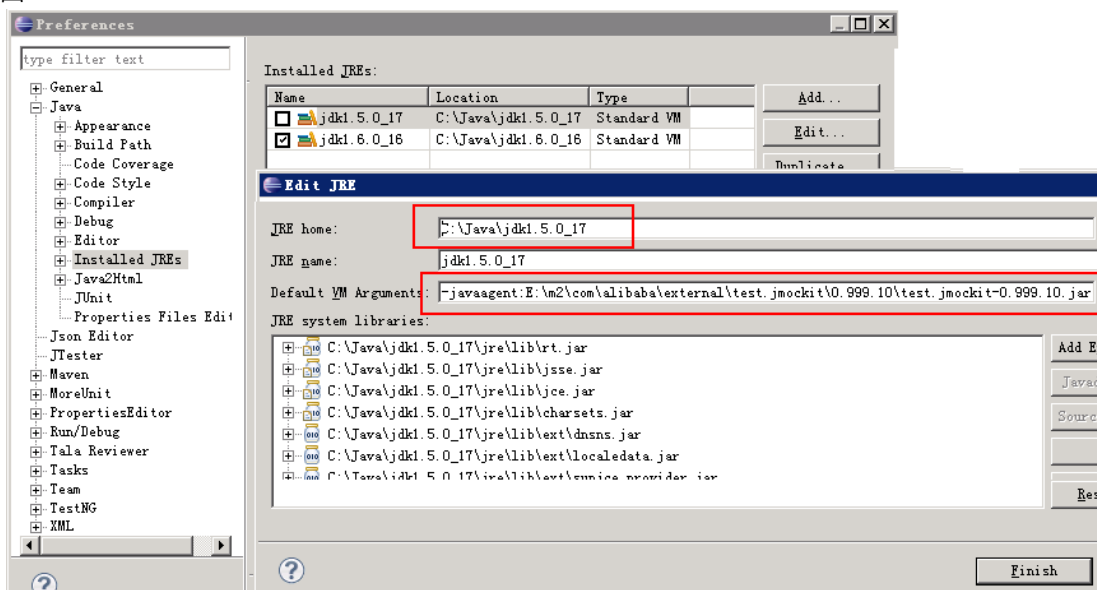
let's go!

控制台中打印出一堆错误:

```
java.lang.IllegalStateException: JMockit has not been initialized.
Check that your Java 5 VM has been started with the
-javaagent:E:\m2\com\alibaba\external\test.jmockit\0.999.10\test.jmockit-0.999.10.jar
command line option.
at mockit.internal.startup.AgentInitialization.initializeAccordingToJDKVersion
(AgentInitialization.java:24)
at mockit.internal.startup.Startup.initializeIfNeeded(Startup.java:208)
at org.jtester.core.Startup.initializeIfNeeded(Startup.java:27)
at org.jtester.module.core.JMockitModule.init(JMockitModule.java:15)
at org.jtester.module.core.loader.ModulesLoader.loading(ModulesLoader.java:86)
at org.jtester.module.core.CoreModule.<init>(CoreModule.java:71)
at org.jtester.module.core.CoreModule.initSingletonInstance(CoreModule.java:61)
```

我们注意到其中一句话: Check that your Java 5 VM has been started with the -javaagent:E:\m2\com\alibaba\external\test.jmockit\0.999.10\test.jmockit-0.999.10.jar

这是因为jTester依赖了jMockit jar包, 要想让jmockit生效, 必须在jvm参数中设置变量: -javaagent:E:\m2\com\alibaba\external\test.jmockit\0.999.10\test.jmockit-0.999.10.jar. 根据你本地jar包的地址不同, 这个值会被智能的打印出来, 大家只要把它拷贝下来粘贴到jvm参数中就可以的, 如下图:

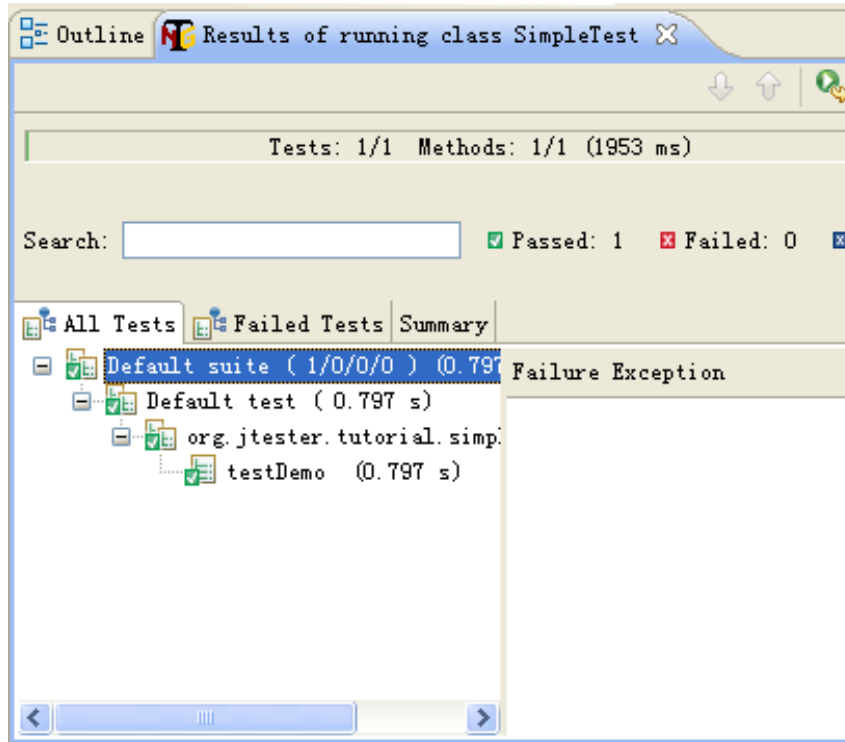


## 注意

我们要注意一下红色框框住的位置，JRE Home应该设置生成JDK安装的路径，而不是JRE路径，否则运行时可能有问题。

现在，我们再次运行，这时测试方法应该会被成功运行。控制台中会打印出下面的语句：简单测试，演示！

同时，打开TestNG Result视图，我们可以看到结果如下。



现在，我们已经顺利的运行了一个简单的测试，下一章开始我们将介绍具体的TestNG语法。

## maven方式运行

以上演示了eclipse中如何运行一个jTester测试。如果你要用maven来运行测试，那么还需要配置maven-surefire-plugin插件。

### 例 1.3. maven-surefire-plugin配置示例

```
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId>
<version>2.6</version>
<configuration>
<testFailureIgnore>>false</testFailureIgnore>
<argLine>-javaagent:"${settings.localRepository}/mockit/jmockit/${jmockit.version}/jmockit-${jmockit.version}.jar"</argLine>
<suiteXmlFiles>
<suiteXmlFile>src/main/resources/testng.xml</suiteXmlFile>
</suiteXmlFiles>
</configuration>
</plugin>
</plugins>
</build>
```

其中变量`${settings.localRepository}`是maven的系统变量，指向本地maven仓库地址，`${jmockit.version}`是jmockit的变量，你需要在maven pom文件的properties中定义，目前版本是0.999.10。

```
<properties>
  <jmockit.version>0.999.10</jmockit.version>
</properties>
```

testng.xml文件是testng的suite文件，具体语法参加testng语法介绍章节。

---

## 第 2 章 TestNG语法简介

在本章，将向大家介绍TestNG框架的基本用法。TestNG 的灵感来自 JUnit，同时尽量保持后者的简单性；但是，TestNG在消除JUnit的大多数限制的同时 增加了很多对开发很有用的特性，它使开发人员可以编写更加灵活、更加强大的测试。TestNG 大量使用 Java Annotation（JDK 5.0 引入）来定义测试。

TestNG 测试类本身是个普通的Java PoJo对象，它无需扩展任何特殊类，也不需要测试方法的任何命名约定，只许标注 @Test 通知TestNG框架：我是一个测试方法。但如果要使用JUniter框架的特性，那么测试类就必须继承一个名为JUniter的基类。

### TestNg注解介绍

@Test，用于标注该方法是个测试方法，或该类是个测试类，这个类中所有的public方法默认情况下都是测试方法。

现在，我们先写一个简单的例子来测试一下Jakarta Common Lang 库中的StringUtils这个类的2个方法：isEmpty() 方法检测 String 是否为空；trim() 方法从 String两端删除控制字符。

#### 例 2.1. TestNG的简单测试

```
import org.apache.commons.lang.StringUtils;
import org.testng.annotations.Test;

public class StringUtilsTest {
    @Test
    public void isEmpty() {
        assert StringUtils.isBlank(null);
        assert StringUtils.isBlank("");
    }

    @Test
    public void trim() {
        String result = StringUtils.trim("  foo  ");
        assert "foo".equals(result);
    }
}
```

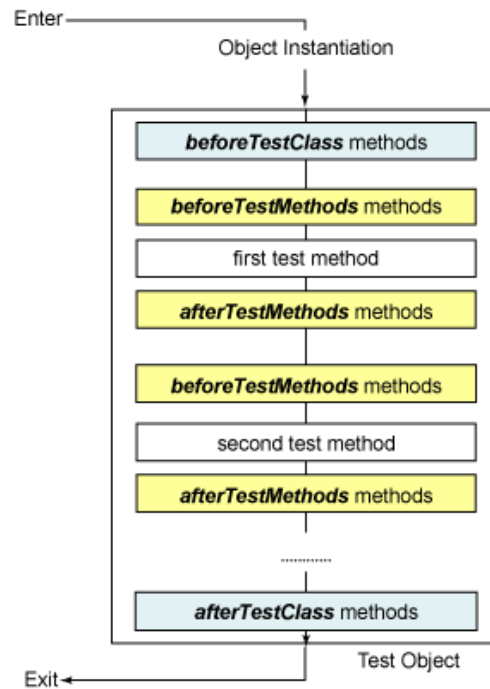
#### 注意

在没有介绍断言前，我们暂时用Java的assert语法作断言。

写一个TestNG的测试很简单吧！在TestNG框架中，不仅可以用@Test标注来指定测试方法，还有其它专用的标注指定类中的其他特定方法，这些方法叫做 配置方法。TestNG中常用的配置方法有四种类型：

- @BeforeMethod  
方法在类中的每个测试方法执行之前都会重新执行。
- @AfterMethod  
方法在类中的每个测试方法执行之后都会重新执行。
- @BeforeClass  
方法在类实例化之后，但是在测试方法(包括@BeforeMethod方法)运行之前执行。
- @AfterClass  
方法在类中的所有测试方法(包括@AfterMethod方法)执行之后执行。

完整的TestNG运行生命周期如下图所示：



现在，我们写一段简单的代码来演示一下TestNG的生命周期。

## 例 2.2. TestNG生命周期Annotation使用

```

public class TestNgDemo {
    @BeforeClass
    public void method1() {
        System.out.println("before class");
    }

    @AfterClass
    public void method2() {
        System.out.println("after class");
    }

    @BeforeMethod
    public void method3() {
        System.out.println("\tbefore method");
    }

    @AfterMethod
    public void method4() {
        System.out.println("\tafter method");
    }

    @Test
    public void test1() {
        System.out.println("\t\ttest1 method");
    }

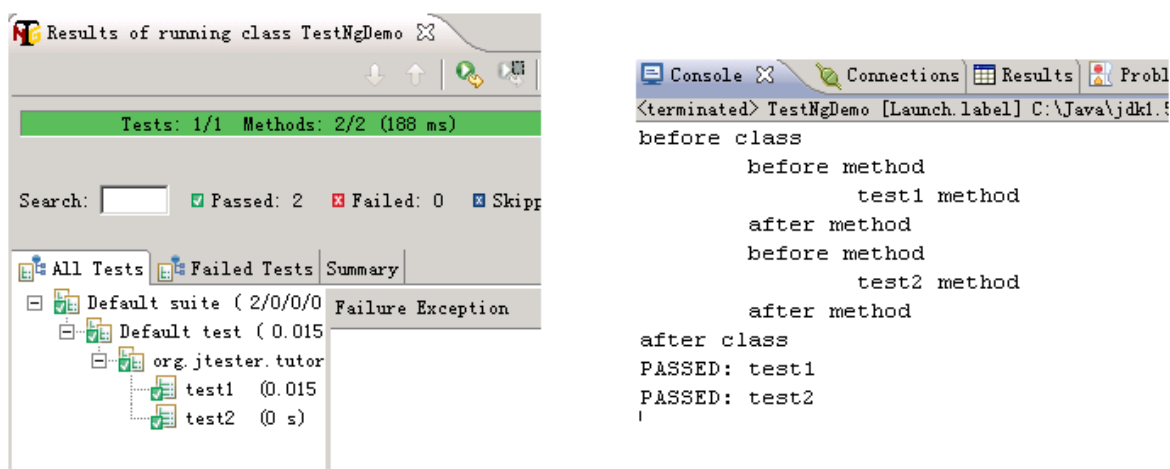
    @Test
    public void test2() {
        System.out.println("\t\ttest2 method");
    }
}

```

上例，定义了下列方法和行为：

- 2个@Test方法test1、test2：它们分别被执行一次。
- 一个@BeforeMethod方法：它会被重复执行2次，分别在test1,test2执行之前被执行。
- 一个@AfterMethod方法：它会被重复执行2次，分别在test1,test2执行之后被执行。
- 一个@BeforeClass方法：无论定义了多少个@Test方法，它都只被执行一次，在所有方法之前被执行。
- 一个@AfterClass方法：无论定义了多少个@Test方法，它都只被执行一次，在所有方法之后被执行。

运行上述例子，结果如下图所示：



## 注意

在同一个类中，同样也可以定义多个@BeforeMethod,@BeforeClass,@AfterMethod,@AfterClass方法，但同一标注的多个方法执行顺序是没有不定的。

## TestNg 参数化测试

TestNG 中另一个有趣的特性是参数化测试，很多情况下我们对同一场景的测试，需要构造不同的边界数据进行测试。在这种情况下，除了个别参数值不一样外，大部分代码都是一样的。如果我们为每组数据写一个方法，会有很多重复的工作和重复的代码。在这种情况下，参数化测试就可以发挥很大的作用。

假定我们有下面这么一段代码，它的功能很简单，用来判断输入值input是在[min, max]区间内，也就是input必须大于等于min，小于等于max。

```

public boolean between(int input, int min, int max) {
    return input >= min && input <= max;
}
  
```

如果要对这么一段代码进行测试，我们必须构造多个场景，我们先假定min和max是正常情况（min = 1, max = 8）：

- input值小于下限的情况，比如: input = 0, 这时between方法返回值应该是false。
- input值等于下限的情况，比如: input = 1, 这时between方法返回值应该是true。
- input值在区间中的情况，比如: input = 5, 这时between方法返回值应该是true。

- input值等于上限的情况，比如: input = 8, 这时between方法返回值应该是true。
- input值大于上限的情况，比如: input = 9, 这时between方法返回值应该是false。

我们对第一种情况写个测试：

```
/**
 * 假定区间是 [1,8]<br>
 * 我们输入值为0<br>
 * between返回值应该是false
 */
@Test
public void testBetween_NumberLessThanMin_ShouldReturnFalse() {
    boolean result = between(0, 1, 8);
    assert result == false;
}
```

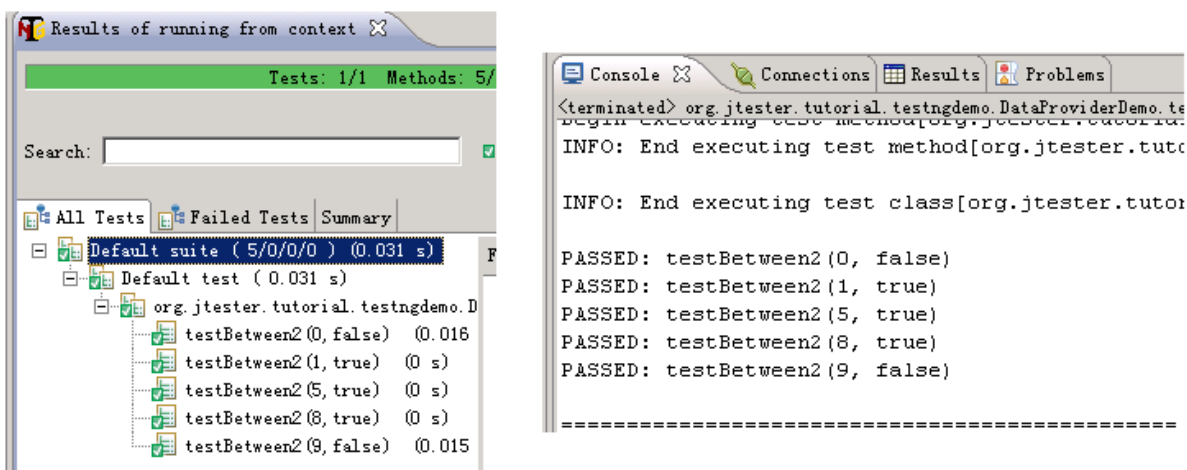
类似的，我们可以分别为input= 1,5,8,9这4组写个测试，但大家可能会觉的繁琐，但从测试完整性的角度，这些边界值是必须测试的。这时候，TestNG的注解@DataProvider开始发挥作用了，我们可以为上面的5组边界值写出下面的测试代码：

```
/**
 * 假定区间是 [1,8]<br>
 * 我们输入不同的边界值<br>
 * between返回值应该要符合我们的期望值expected
 */
@Test(dataProvider = "dataForBetween")
public void testBetween(int input, boolean expected) {
    boolean result = between(input, 1, 8);
    assert result == expected;
}

@DataProvider
public Object[][] dataForBetween() {
    return new Object[][] { // <br>
        /** 当输入值是0时，返回值应该是false */
        new Object[] { 0, false },
        /** 当输入值是1时，返回值应该是true */
        new Object[] { 1, true },
        /** 当输入值是5时，返回值应该是true */
        new Object[] { 5, true },
        /** 当输入值是8时，返回值应该是true */
        new Object[] { 8, true },
        /** 当输入值是9时，返回值应该是false */
        new Object[] { 9, false } };
}
```

执行上面的测试，我们发现，虽然我们只写了2个方法，但TestNG却执行了5组测试：





@DataProvider式的测试是很有效的公用测试方法的机制，但上面构造二维的形式有点不够直观，jTester对TestNG作了一点封装，可以使用DataIterator来返回不同的边界值。

### 例 2.3. 参数化测试示例

```
/**
 * 假定区间是 [1,8]<br>
 * 我们输入不同的边界值<br>
 * between返回值应该要符合我们的期望值expected
 */
@Test(dataProvider = "dataForBetween")
public void testBetween(int input, boolean expected) {
    boolean result = between(input, 1, 8);
    assert result == expected;
}

@DataProvider
public Iterator dataForBetween() {
    return new DataIterator() {
        {
            /** 当输入值是0时，返回值应该是false */
            data(0, false);
            /** 当输入值是1时，返回值应该是true */
            data(1, true);
            /** 当输入值是5时，返回值应该是true */
            data(5, true);
            /** 当输入值是8时，返回值应该是true */
            data(8, true);
            /** 当输入值是9时，返回值应该是false */
            data(9, false);
        }
    };
}
```

在DataIterator中的每个data都表示一组测试数据，它的参数个数必须和测试方法testBetween的参数个数一致。这些值也是被一一对应传给测试方法。

## 异常测试

在上一个章节中，我们用@DataProvider测试了5组边界值测试。 这是我们是假定区间值本身是正确的，即 $\max \geq \min$ 。那如果区间本身是错误的情况下，会发生什么呢？

在这种情况下，可能不同的业务有不同的处理方式：

- 处理方式一：如果传入的区间 $\max < \min$ ,要求将max和min对调。

- 处理方式二：如果传入的区间 $\max < \min$ ,要求抛出一个运行时异常。

这里，我们采用方式二，如果区间错误，会抛出一个异常。比如我们现在有一组值:  $\text{input} = 5, \min = 8, \max = 1$ ，当我们传入这组参数时，要求方法会抛出一个异常：

### 例 2.4. 异常测试代码一

```
@Test(expectedExceptions = RuntimeException.class)
public void testBetween_RangeIsError() {
    between(5, 8, 1);
}
```

在TestNG中要期望一个api调用会抛出一个异常，只需要在@Test的属性expectedExceptions中声明你期望的异常类型就可以了。expectedExceptions的值是个Class[]类型，这就允许你声明多个异常，比如写成：@Test(expectedExceptions = {ExceptionOne.class, ExceptionTwo.class})

我们运行上面的测试，结果在控制台打印出下列错误消息：

```
Expected exception java.lang.RuntimeException but got org.testng.TestException:
Method org.jtester.tutorial.testngdemo.DataProviderDemo.testBetween_RangeIsError()
should have thrown an exception of class java.lang.RuntimeException
at org.testng.internal.Invoker.handleInvocationResults(Invoker.java:1416)
at org.testng.internal.Invoker.invokeTestMethods(Invoker.java:1184)
```

错误消息翻译成中文就是：测试期望得到一个运行时异常，但结果却是一个TestNG的测试异常。

这是因为我们的方法，没有返回我们要的异常，testng测试错误。也就是between方法没有对异常数据进行处理，我们修改between方法如下：

```
public boolean between(int input, int min, int max) {
    if (min > max) {
        throw new RuntimeException("the number min must less than the number max.");
    }
    return input >= min && input <= max;
}
```

这时，我们再运行上面的异常测试，结果就显示为绿色，测试通过！

上面的例子中，我们只是验证了期望要抛出的异常类型，如果要验证异常的消息，上面Annotation的方式就无法做到了，需要进行try catch：

### 例 2.5. 异常测试代码二

```
@Test
public void testBetween_RangeIsError() {
    try {
        between(5, 8, 1);
        assert false;
    } catch (Exception e) {
        String message = e.getMessage();
        assert message == "the number min must less than the number max.";
    }
}
```

我们使用try catch方式将异常消息捕获，然后对异常进行断言。

### 注意

在try{}代码块的最后一行 assert false; 这个语句是必须，如果没有这条语句，between方法没有抛出异常，那么测试方法也会成功通过，那样就达不到我们测试异常的目的。

如果你不是想要对异常进行测试，在测试代码中就不需要有try catch代码块，即使api会显式的抛出异常，你也无需捕获，只需要将它们抛到测试方法上就可以的。例如这样：

### 例 2.6. 非异常测试应该将异常直接抛到测试方法上

```
@Test
public void testBetween_Normal() throws Exception {
    // 这是一个正常测试
}
```

## TestNg分组测试和suite文件编写

当我们的测试越来越多的时候，我们可能需要对测试分类，比如工具类测试，数据库测试，接口测试等等。甚至开发很多的时候，我们也需要给每个开发的测试进行分类；或者根据模块进行分类。总之，我们会有很多分类的理由，在某个特定的时刻，我们可能只想跑一组或多组特定类别的测试。

在TestNG框架中，对测试分类特别简单，只需要在@Test(groups="name")的groups属性中标明这个属性属于哪个组就可以了。

### 例 2.7. 对测试进行分组

```
@Test(groups = { "web" })
public class TestNgGroupsDemo {
    @Test(groups = { "darui.wu", "checkin" })
    public void test1() {
        //测试
    }

    @Test(groups = { "broken" })
    public void test2() {
        //测试
    }

    @Test(groups = { "debug" })
    public void test3() {
        //测试
    }

    @Test(groups = { "esb", "debug" })
    public void test4() {
        //测试
    }
}
```

在测试类上面标注@Test(groups)表明该类中所有的测试都属于这个组，例子中表示test1,test2,test3,test4都属于web组。

在测试方法上面标注@Test(groups)只表明该方法属于这个组，它和类上面的groups是个并集关系，对于上面的例子，各个测试所属组别分别是：

- test1: 属于 web, darui.wu, checkin 这3个组。
- test2: 属于web, broken这2个组。
- test3: 属于web, debug这2个组。
- test4: 属于web, esb, debug这3个组。

分组完毕，那么我们怎么根据不同的策略跑不同组别的测试呢？我们需要写一个testNG的suite文件。在testng suite文件中，我们可以很灵活根据不同的需要定义需要跑的测试方法或类。下面是一个例子：

### 例 2.8. 指定若干组的suite文件

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="your_test_suite_name">
  <test name="your_test_name">
    <groups>
      <run>
        <include name="checkin" />
        <include name="web" />
        <exclude name="broken" />
      </run>
    </groups>
    <packages>
      <package name="org.jtester.*" />
    </packages>
  </test>
</suite>
```

上面的例子指定了跑org.jtester目录（包括子目录）下，所有属于checkin或web组的测试，但排除属于broken组的测试。

更详细的信息你可以参考官网：<http://testng.org/doc/documentation-main.html#testng-xml>

## 第 3 章 JTester断言介绍

任何测试都应该包含3个阶段：

- Arrange

准备测试环境，包括数据，mock外部接口，启动容器等等，甚至包括机器的时间。这部分的内容我们在数据库测试和mock章节会做详细的介绍。

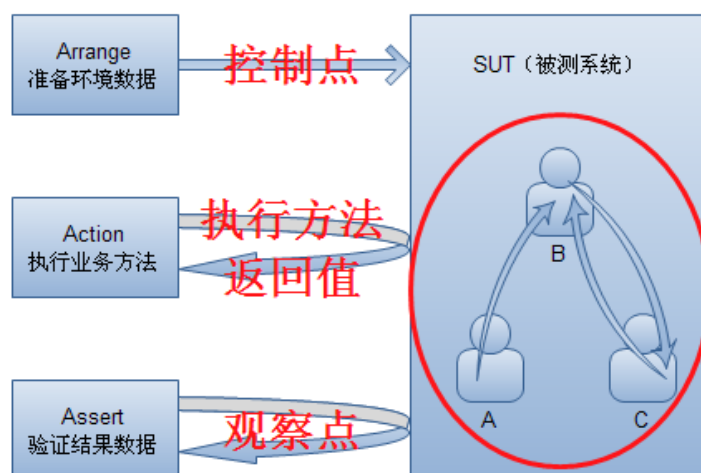
- Action

执行要测试的目标对象，一般来说就是执行java api，传入预设好的参数。

- Assert

判断执行结果是否符合预期，结果可能是API直接返回值，数据库状态，机器环境，或者API内部传给外部接口的参数。

用交互图来表示，应该就是下面的样子：



这也就是测试中著名的AAA论断，我们这章要讲的就是Assert，可以这么说，如果你的测试方法中没有Assert，那么这个方法就不能称之为测试。

在具体介绍JTester断言前，我们先介绍一下什么是断言？

### 什么是测试断言？

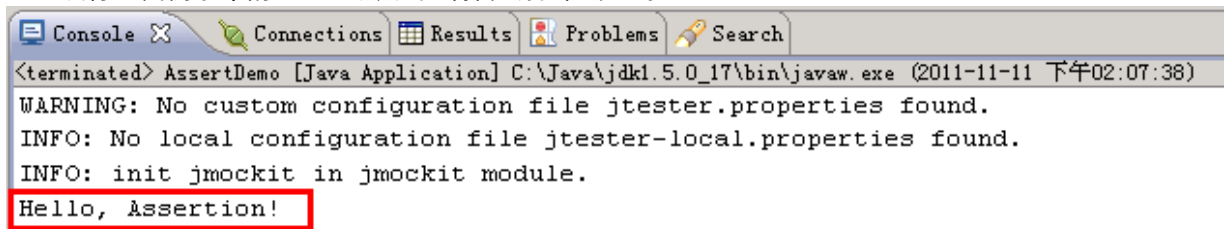
在从JDK1.4开始，java引入了assert关键字，但Java默认在执行时是不启动断言检查的，这时候java代码中的所有assert语句都是不生效的，哈哈，就是和注释一样，无声无息！如果要开启断言检查，则需要用开关-enableassertions或-ea来开启。

但大家在使用junit和testng测试框架时，是无需关心这个参数的，框架在启动测试时，本身会把这个开关打开的。

```
public class AssertDemo extends JTester {
    public static void main(String[] args) {
        assert false;
        System.out.println("Hello, Assertion!");
    }

    @Test
    public void test() {
        assert false;
        System.out.println("Hello, Assertion!");
    }
}
```

执行上面例子中的main函数，控制台会打印出语句：Hello, Assertion!



```
<terminated> AssertDemo [Java Application] C:\Java\jdk1.5.0_17\bin\javaw.exe (2011-11-11 下午02:07:38)
WARNING: No custom configuration file jtester.properties found.
INFO: No local configuration file jtester-local.properties found.
INFO: init jmockit in jmockit module.
Hello, Assertion!
```

这个说明，语句assert false;默认没有生效！但运行test方法，测试就会抛出错误：



```
<terminated> org.jtester.module.assertion.AssertDemo.test [Launch.label] C:\Java\jdk1.5.0_17\bin\javaw.exe (2011-11-11 下午02:14)
INFO: No local configuration file jtester-local.properties found.
INFO: init jmockit in jmockit module.
INFO:
FAILED: test
java.lang.AssertionError
    at org.jtester.module.assertion.AssertDemo.test (AssertDemo.java:15)
    at sun.reflect.NativeMethodAccessorImpl.invoke0 (Native Method)
    at java.lang.reflect.Method.invoke (Method.java:585)
    at sun.reflect.NativeMethodAccessorImpl.invoke0 (Native Method)
    at java.lang.reflect.Method.invoke (Method.java:585)
    at org.jtester.bytecode.reflector.MethodAccessor.invoke (MethodAccessor.java:45)
    at org.jtester.bytecode.reflector.MethodAccessor.invokeUnThrow (MethodAccessor.java:68)
    at org.jtester.core.testng.JMockitHookable.run (JMockitHookable.java:52)
    at org.jtester.core.testng.JTesterHookable.run (JTesterHookable.java:60)
    at sun.reflect.NativeMethodAccessorImpl.invoke0 (Native Method)
    at java.lang.reflect.Method.invoke (Method.java:585)

=====
Default test
Tests run: 1, Failures: 1, Skips: 0
=====
```

## 注意

记住，assert抛出的是错误（Error），不是异常（Exception）！

```
 * @version 1.7, 12/19/03
 * @since   JDK1.4
 */
public class AssertionError extends Error {
    /**
     * Constructs an AssertionError with no detail message.
     */
    public AssertionError() {
    }
}
```

assert关键字语法很简单，有两种用法：

- assert <boolean表达式>  
如果<boolean表达式>结果为true，则程序继续执行。  
如果结果为false，则抛出AssertionError，程序终止执行。
- assert <boolean表达式> : <错误信息表达式>  
如果<boolean表达式>结果为true，则程序继续执行。  
如果结果为false，则程序抛出java.lang.AssertionError，并输出（控制台打印出）<错误信息表达式>。

列举一些assert例子：

```
assert    0 < value;
assert    0 < value : "value="+value;
assert    obj != null : "object can't be null.";
```

```
assert    isExisted() : "something doesn't exist!";
```

在业务类中使用assert，有很多陷阱需要避免，但在测试中，这是一个必须的法宝，如果测试方法中没有assert语句，那么这个测试就不应该存在。

当然，我们在程序中不会直接使用assert语法，这个功能太弱了，无法满足我们的需求。JUnit和TestNG都提供了自己的assert断言语法，它们大概定义了下列assert API：

assertTrue(condition) assertTrue(String message, boolean condition)	断言条件为真，若不满足，则抛出带消息（如果有）的AssertionError错误。
assertFalse(condition) assertFalse(String message, boolean condition)	断言条件为假，若不满足，则抛出带消息（如果有）的AssertionError错误。
assertEquals(Object actual, Object expected) isEqualTo(Object expected, Object actual)	断言2个对象相等，如果不满足，抛出AssertionError错误。
assertNull(String message, Object object)	断言对象为空，如果不满足，抛出AssertionError错误。
assertNotNull(String message, Object object)	断言对象不为空，如果不满足，抛出AssertionError错误。
assertSame(Object expected, Object actual)	断言2个引用指向同一个对象，如果不满足，抛出AssertionError错误。
assertNotSame(Object unexpected, Object actual)	断言2个引用不是指向同一个目标，如果是，抛出AssertionError错误。
fail(String message)	一个永远错的断言，抛出需要的信息。（这种断言在异常测试中需要用到，详见TestNG异常测试章节）

还有这些断言的变种形式，但这些断言也是太基本了，根本无法满足我们千变万化的需求，从下节开始，我们将介绍jTester方便但强大的断言语法。

## jTester断言基本介绍

java原生的和TestNG (junit) 的断言语法实在是太弱了，根本无法接受稍微复杂一点点的断言。比如说要判断2个object对象中某2个属性是否相等？

为了解决这个问题，我们需要写一个返回值是布尔表达式的函数，来判断2个对象的2个属性是否相等，比如下面的例子：

```
@Test
public void testUserNameAndAgeEquals() {
    User expected = new User("jobs.he", "xxxx", "xxxx", 30);
    // 执行某个业务方法，返回一个User对象
    User actual = someBusinessApi();
    Assert.assertTrue(isUserNameAndAgeEquals(actual, expected));
}

/**
 * 判断2个User对象名字和年龄是否相等
 */
private boolean isUserNameAndAgeEquals(User actual, User expected) {
    if (actual == null || expected == null) {
        return false;
    }
    String expectedName = expected.getName();
    if (expectedName == null) {
        if (actual.getName() != null) {
            return false;
        }
    } else if (expectedName.equals(actual.getName()) == false) {
        return false;
    }

    if (expected.getAge() == actual.getAge()) {
        return true;
    }
}
```



```

} else {
    return false;
}
}

```

这种方式太土了，为了测试，我写了一个判断方法，因为if语句太多了，我也没有把握保证这个方法是否完全正确。什么？为这个方法再写个测试？那子子孙孙无穷尽了。

## 注意

单元测试的代码是否需要测试？

答案是显然的：不需要！如果你觉的要，那就是你的测试代码太复杂了，测试代码应该简单易懂，一目了然！任何复杂和易错的测试代码都是不可取的，如果你觉的业务代码难测，那应该首先重构业务代码，而不是写出一个复杂测试。

对于上面那种情况，我们一定有很好的解决办法。有一个开源的工具：[hamcrest](http://code.google.com/p/hamcrest/wiki/Tutorial), <http://code.google.com/p/hamcrest/wiki/Tutorial>。Hamcrest提供了一系列通用的匹配器(Matcher)，这些匹配器可以灵活使用已定义的规则，程序员更加精确的表达自己的测试思想，指定所想设定的测试条件。Hamcrest 提供了一个全新的断言语法——assertThat。使用assertThat断言语句，结合Hamcrest提供（或者自定义的）匹配器，就可以测试所要的期望结果。

assertThat 的基本语法如下：assertThat( [value], [matcher statement] );

- value  
想要测试的对象，实际目标值。
- matcher statement  
Hamcrest 匹配器实例，如果 value 值与 matcher statement 所表达的期望值相符，则测试成功，否则测试失败。

具体的Hamcrest语法，大家可以参考官方站点，这不是本文档的重点，这里只给出一个例子，给大家有个感性的认识：

```

@Test
public void hamcrestDemo() {
    String str = "hello world!";
    MatcherAssert.assertThat(str, anyOf(contains("developer"), contains("Works")));
}

```

上面的例子，Hamcrest断言str字符串中包含子串"developer"或者包含子串"Works"。但实际上str并不包含这2个字符串，运行后会有上面结果呢？控制台中会打印出非常详细的信息：

```

FAILED: hamcrestDemo
java.lang.AssertionError:
Expected: (expected string containing developer
, but actual string is:hello world!.
or expected string containing Works
, but actual string is:hello world!.
)
but: was "hello world!"
at ext.jtester.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:21)

```

相比于原生的assert和TestNG (junit) 的断言，hamcrest语法有下面的优点：

- 风格统一，Hamcrest只有一条断言API：assertThat，配合丰富的匹配器使用。
- 具有很强的易读性，而且使用起来更加灵活
- 丰富的匹配器可以组合使用，以达到复杂的目的。
- 错误信息更加易懂、可读且具有描述性（descriptive）。
- 开发可以自定义自己的匹配器。



啰哩啰唆写了这么多有关Hamcrest的东西，到底和jTester断言有上面关系呢？是的，jTester断言是对Hamcrest匹配器的一种包装，虽然hamcrest有很多优点，但也有下面2个大的缺点：

- 有太多的匹配器，开发根本不知道他要断言的情况是否有一种断言器可以使用？
- 即使知道有，开发未必记得住断言器的名称！

jTester断言器希望能够继承Hamcrest的优点，克服它的缺点，达到下面的目标：

- 易用，基本无需记忆。  
采用了Fluent API的方式，能够充分利用IDE的智能提示功能。
- 强大，大部分断言场景都能覆盖到。  
扩展了很多Hamcrest的匹配器，特别是属性断言，可以解决开发的很多难点。
- 错误信息明确。  
这是Hamcrest本身的优点，毫无疑问，jTester更应该发扬光大。

在jTester的概念中，断言被分成2种类型，有不同的语法：

- 即时生效的断言  
want.java类型(实际值).期望结果 ()....

```
@Test
public void assertString() {
    String alibabaAddress = "杭州滨江网商路699号";

    want.string(alibabaAddress).start("杭州").end("699号").contains("网商路").regular(".*699.*");
}
```

这条语句是立刻判断alibabaAddress字符串是否以“杭州”开头，以“699号”结尾，中间包含字符串“网商路”，并且整个字符串符合正则表达式“.\*699.\*”。任何一个个性的不符合都将抛出AssertionError，终止测试的运行。

- 延后生效的断言  
the.java类型().期望结果 ()....

```
@Test
public void assertString2() {
    String alibabaAddress = "杭州滨江网商路699号";

    IStringAssert matcher = the.string().start("杭州").end("699号").contains("网商路").regular(".*699.*");
    want.string(alibabaAddress).all(matcher);
}
```

这个例子the.string()只是预先定义了一个Matcher，意思是将来如果有个字符串串给我，那它必须符合哪些哪些规则。随后以后want中的all方法接受了这个Matcher，表示alibabaAddress这个字符串必须符合预定义的Matcher行为。

我们大概描述了jTester断言的基本语法，从下节开始我们将详细介绍各种java类型断言的详细用法。

## jTester断言语法详解

jTester断言分为即时生效的和延后生效的，这2种断言，立即生效的需要以want开头，需要立即传入实际值进行判断。延后生效的以the开头，不需要立即传入参数实际值，其它判断形式都是一样的。所以我们将重点介绍即时生效的断言，jTester针对具体的java类型 又分出下列形式：

断言string字符串	want.string(str).....
-------------	-----------------------

断言布尔对象	want.bool(bl).....
断言数值	want.number(number).....
断言普通PoJo对象	want.object(obj).....
断言map对象	want.map(map).....
断言collection对象或array对象	want.list(coll/array).....
具体内容	具体内容

## 断言String对象

### 判断字符串是否等于期望值

相等判断，下面2个用法都是等效的,eq()是isEqualTo()的简写形式:

- want.string(实际值).isEqualTo(期望值)
- want.string(实际值).eq(期望值)

示例如下:

#### 例 3.1. 判断字符串等于期望值

```
@Test
public void testStringEqual() {
    String actual = "I am a string.";
    // 断言实际只actual 等于期望值: I am a string.
    want.string(actual).isEqualTo("I am a string.");
    // eq是isEqualTo的简写
    want.string(actual).eq("I am a string.");
}
```

### 注意

经常有同学把相等判断写成equals(...), equals是java object的基类方法, 在jTester断言中是无效的, 使用这个方法会抛出一个异常:

```
org.jtester.exception.JTesterException:
    the method can't be used, please use isEqualTo() instead
at org.jtester.hamcrest.iassert.common.impl.Assert.equals(Assert.java:109)
at org.jtester.module.assertion.StringAssertion.testEquals(StringAssertion.java:21)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

## 字符串的模糊判断

我们上面简单判断了2个字符串是否相同, 但很多情况下, 我们也许不需要这么严格的判断, 比如上面的实际值是"I am a string.", 我们也认为和"I am a string."相等。我们并不关心里面的空格到底有多少个, 只要把意思表达对了就可以了。

### 注意

为什么需要字符串的模糊判断呢? 因为太严格的判断, 会使测试变的很脆弱, 我们需要让测试对一些情况有一定的容忍度, 特别是对异常消息, 错误描述等等这些情况, 多一个会车符, 少一个空格符并没有对 实际的信息有影响, 在业务上也并不认为这是一个bug。这时候, 我们就需要用到StringMode这么一个字符串判断模式的选项, StringMode有6个选项:

- IgnoreCase  
将实际值和期望值都转成小写字符串, 然后2者再进行比较。我们也可以直接使用等效的快捷API: eqIgnoreCase。

### 例 3.2. 忽略大小写时字符串比较

```
@Test
public void testIgnoreCase() {
    String actual = "I AM A STRING.";
    want.string(actual).eq("i am a STRING.", StringMode.IgnoreCase);
}
/**
 * 快捷api: eqIgnoreCase
 */
@Test
public void testIgnoreCaseEqual() {
    String actual = "I AM A STRING.";
    want.string(actual).eqIgnoreCase("i am a STRING.");
}
```

- IgnoreSpace

将实际值和期望值之间的空格, Tab符, 回车, 换行, 换页等字符都过滤掉, 然后2者再进行比较。也可以直接使用等效的快捷API: eqIgnoreSpace。

### 例 3.3. 忽略空格时字符串比较

```
@Test
public void testIgnoreSpace() {
    String actual = "I    am\t\n a \f string.";
    want.string(actual).eq(" Iam a string. ", StringMode.IgnoreSpace);
}
/**
 * 快捷api: eqIgnoreSpace
 */
@Test
public void testIgnoreSpaceEqual() {
    String actual = "I    am\t\n a \f string.";
    want.string(actual).eqIgnoreSpace(" Iam a string. ");
}
```

- IgnoreQuato

将实际值和期望值中的双引号"和单引号'都过滤掉, 然后2者再进行比较。

### 例 3.4. 忽略单双引号时字符串比较

```
@Test
public void testIgnoreQuato() {
    String actual = "{name:\"darui.wu\", address:\"杭州市\"}";
    want.string(actual).eq("{'name': 'darui.wu', 'address': '杭州市'}",
    StringMode.IgnoreQuato);
}
```

- SameAsSpace

将实际值或期望值中的任何空格, Tab符, 回车, 换行, 换页等字符都替换成一个空格", 然后2者再进行比较。即所有空格被认为是等价的。也可以使用快捷API: eqWithStripSpace。

### 例 3.5. 所有空格等价时字符串比较

```
@Test
public void testSameAsSpace() {
    String actual = "I    am\t\n a \f string.";
    want.string(actual).eq("I am a string.", StringMode.SameAsSpace);
}
/**
 * 快捷api: eqWithStripSpace
 */
@Test
public void testEqWithStripSpace() {
    String actual = "I    am\t\n a \f string.";
    want.string(actual).eqWithStripSpace("I am a string.");
}
```

- SameAsQuato

将实际值和期望值中的双引号"和单引号'都替换成单引号', 然后2者在进行比较。即单双引号被认为是等价的。

### 例 3.6. 单双引号等价时字符串比较

```
@Test
public void testSameAsQuato() {
    String actual = "{name: \"darui.wu\", address: \"杭州市\"}";
    want.string(actual).eq("{name: 'darui.wu', address: '杭州市'}",
    StringMode.SameAsQuato);
}
```

- SameAsSlash

将实际值和期望值中的斜杠'\'和单反斜杠'/'都替换成斜杠, 然后2者在进行比较。即斜杠和单反斜杠被认为是等价的。

### 例 3.7. 单双引号等价时字符串比较

```
@Test
public void testSameAsSlash() {
    String actual = "d:/abc\\e/1.txt";
    want.string(actual).eq("d:/abc/e/1.txt", StringMode.SameAsSlash);
}
```

我们上面解释了各种模糊比较字符串的情形, 如果我又要忽略空格, 又要单双引号等价的情况下怎么办? 这时候, 我们可以使用多个StringMode进行复合判断。

### 例 3.8. 使用多个模糊判断模式进行字符串复合断言

```
@Test
public void testStringEqCompound() {
    String actual = "{name: \"darui.wu\", address: \"杭州市\"}";
    want.string(actual).eq("{name: 'darui.wu', address: '杭州市'}", StringMode.SameAsQuato,
    StringMode.IgnoreSpace);
}
```

同时, jTester断言是Fluent API形式的语法, 你可以一个断言跟着一个断言进行。

## 断言子字符串

前面2小节, 演示了如何断言字符串等于期望值, 或者在某些模糊状态下和期望值相等。但还有很多情况, 我们可能不用判断整个字符串的情况, 只需要实际字符串中包含某些子字符串就可以了。这时候, 我们就要用到contains, containsInOrder, notContain, start, end等断言API。

## 例 3.9. 判断字符串中包含指定的子字符串

```

@Test
public void testContains() {
    String actual = "{name: 'darui.wu', address: '杭州市'}";
    want.string(actual).contains("darui.wu");
}
/**
 * 断言字符串在忽略空格的情况下, 包含2个子字符串"darui.wu", "zhejianghangzhou"
 */
@Test
public void testContains_MultipleStr() {
    String actual = "{name: 'darui.wu', address: 'zhengjiang hangzhou'}";
    want.string(actual).contains(new String[]{"darui.wu", "zhejianghangzhou"},
    StringMode.IgnoreSpace);
}

```

## 例 3.10. 判断字符串中依次包含若干子字符串

```

@Test
public void testContainsInOrder() {
    String actual = "abc efg";
    want.string(actual).containsInOrder("abc", "efg");
}
/**
 * 在不忽略大小写的情况下, 希望字符串"Abc Efg"包含子串"abc","efg"。
 * ,但实际上不含子串"abc", 断言抛出错误。
 */
@Test(expectedExceptions = AssertionError.class)
public void testContainsInOrder_NoModes() {
    String actual = "Abc Efg";
    want.string(actual).containsInOrder(new String[] { "abc", "efg" });
}
/**
 * 在忽略大小写的情况下, 字符串"Abc Efg"包含子串"abc","efg"
 */
@Test
public void testContainsInOrder_HasModes() {
    String actual = "Abc Efg";
    want.string(actual).containsInOrder(new String[] { "abc", "efg" },
    StringMode.IgnoreCase);
}

```

## 例 3.11. 判断字符串不包含子字符串

```

/**
 * 希望字符串不包含子串"abc"和"efg", 但实际上包含了"abc", 断言会抛出错误
 */
@Test(expectedExceptions = AssertionError.class)
public void testNotContains_Failure() {
    want.string("abc cba").notContain(new String[] { "abc", "efg" });
}
/**
 * 希望字符串不包含子串"acb"和"efg", 实际上也不包含, 断言通过
 */
public void testNotContains() {
    want.string("abc cba").notContain(new String[] { "acb", "efg" });
}

```

### 例 3.12. 判断字符串以指定的子字符串开头

```
/**
 * 在忽略空格和大小写的情况下，实际字符串以abc开头
 */
@Test
public void testStart() {
    want.string("a B C====").start("abc", StringMode.IgnoreCase, StringMode.IgnoreSpace);
}
```

### 例 3.13. 判断字符串以指定的子字符串结尾

```
/**
 * 在忽略空格和大小写的情况下，实际字符串以abc结尾
 */
@Test
public void testEnd() {
    want.string("====a b C").end("abc", StringMode.IgnoreCase, StringMode.IgnoreSpace);
}
```

## 其它字符串断言

前面几个小节详细介绍了String字符串断言常用的语法：在各种模式下的相等，包含等判断。这一节，我们把剩下的一些断言给介绍完毕。

判断一个字符是否符合正则表达式，常用于邮箱判断，手机判断，身份证判断等等。

### 例 3.14. 判断一个字符串是否符合正则表达式

```
/**
 * 正则表达式演示，期望实际值是个163邮箱
 */
@Test
public void testRegular_StringDoesIsAEMail() {
    String actual = "darui.wu@163.com";
    want.string(actual).regular("[\\w\\d\\.\\_]+@163\\.com");
}
```

### 例 3.15. 判断一个字符串是否是若干个候选字符串中的一个

```
/**
 * 期望用户邮箱是163邮箱或者gmail邮箱，且不是sohu邮箱和sina邮箱
 */
@Test
public void testInCadidateStrings() {
    String actual = "darui.wu@163.com";
    want.string(actual).in("darui.wu@163.com", "darui.wu@gmail.com")
        .notIn("darui.wu@sohu.com", "darui.wu@sina.cn");
}
```

### 例 3.16. 判断一个字符串是空字符串，非空字符串，非空白字符串等情况

```
/**
 * 期望字符串是null
 */
public void testNull() {
    String actual = null;
    want.string(actual).isNull();
}

/**
 * 期望字符串非空且不是空串
 */
public void testNotNull() {
    String actual = "adfasd";
    want.string(actual).notNull().notBlank();
}
```

还有一种比较特殊的情况，我们希望字符串是任何值都可以通过断言，比如在Mock一个API时，我们不关心参数的值，只关心返回的模拟值（具体详见Mock章节）。

### 例 3.17. 字符串是任意值均可通过的断言

```
public void testAny() {
    want.string("adfasd").any();
    want.string(null).any();
    want.string("").any();
}
```

好，到现在我们把最常用的断言类型，字符串断言给全部介绍完毕，下节我们将介绍如何对一个普通对象进行断言。

## Java基本类型的断言

在java语言体系中，除了String这个基本类型外，还有7个primitive类型及其对应的java对象类型：

- 布尔对象  
boolean, Boolean
- 数值对象  
int(Integer), long(Long), short(Short), double(Double), float(Float), BigInteger, BigDecimal, 包括Byte(byte)
- 字符对象  
char(Character)
- byte对象  
byte(Byte)

对于这些基本的对象类型，jTester都提供了简单的断言语法。  
布尔对象的断言很简单，就是直接判断一个布尔值是true还是false。

### 例 3.18. 对布尔值进行断言

```
@Test
public void testBoolean() {
    want.bool(true).is(true);
    want.bool(new Boolean(false)).is(false);
}
```



Character(char)的判断同样很简单，直接判断这个值是否等于(或者不等于)另外一个值。

### 例 3.19. 对Character(char)进行断言

```
@Test
public void testCharacter() {
    want.character('a').is('a');
    want.character('a').notEqualTo('b');
}
```

这些基本对象中，稍微复杂一点的就数值类型，可以有等于，大于，小于，大于等于，小于等于，不等于，在什么范围内等等。

want.number(number).eq(expected)	同isEqualTo，断言数值等于期望值
want.number(number).isLt(expected)	同isLessThan，断言数值小于期望值
want.number(number).isLe(expected)	同isLessEqual，断言数值小于等于期望值
want.number(number).isGt(expected)	同isGreaterThan，断言数值大于期望值
want.number(number).isGe(expected)	同isGreaterEqual，断言数值大于等于期望值
want.number(number).isBetween(min, max)	断言数值小于等于最大值且大于等于最小值
want.number(number).notEqualTo(expected)	断言数值不等于期望值

### 例 3.20. 对数值类型进行断言

```
@Test
public void testNumberAssert() {
    want.number(new BigInteger("11100")).isEqualTo(11100);
    want.number(21.00D).isLt(100D);
    want.number(1.1F).isGt(1.0F);

    want.number(1).isBetween(0, 2);
    want.number(4).isGe(4);
    want.number(5).isLe(5);
}
```

## 对普通PoJo对象进行断言

在Java世界，存在大量的普通PoJo对象，它没有一个预先设置好的类型，框架自然也不可能针对特定的PoJo对象写一个通用的断言API。虽然针对特定的PoJo对象是不可能的，但PoJo对象还是有共通的地方：它们都是由属性值组成，一般都拥有对应的get和set方法。既然如此，我们就可以无视PoJo类型，直接对PoJo对象的属性进行断言。

在本章的第二节，我们提到一个问题：如何判断2个object对象中某2个属性是否相等？本节将解答这个问题。

对普通java对象的断言，以 want.object(obj)... 形式进行，下面我们首先从判断2个java对象是否相等开始。

## 判断2个PoJo对象是否相等

### 例 3.21. 判断2个PoJo对象是否相等

```
@Test
public void testEqual() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    want.object(user).eq(new User("jobs.he", "浙江省杭州市", "310010", 30));
}
```

上面的断言是判断2个User对象通过java equal方法进行比较，如果返回值是true，则认为2个对象是相等的；如果是false，则2个对象是不等的。这就要求PoJo对象要重写 (@Override) equals和hashmap这2个方法。如果没有实现这2个方法，运行上面的例子，框架会抛出下面的错误：



```

FAILED: testEqual
java.lang.AssertionError:
Expected: <org.jtester.module.assertion.beans.User@138c63>
but: was <org.jtester.module.assertion.beans.User@165f738>
at ext.jtester.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:21)

```

但即使PoJo对象实现了equals方法，也未必能符合我们断言的要求。比如User对象实现了下面的equals方法：

```

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((address == null) ? 0 : address.hashCode());
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;

    User other = (User) obj;

    if (name == null) {
        if (other.name != null) return false;
    } else if (!name.equals(other.name)) return false;

    if (address == null) {
        if (other.address != null) return false;
    } else if (!address.equals(other.address)) return false;

    return true;
}

```

如图中的红框所示，User对象认为只要name和address是一样的，那么就认为这2个对象是相等的。这在某些业务场景下是合理的，但如果我们需要关注User对象的属性postcode的时候，那么下面的测试即使postcode不等，测试也是会通过的。

```

@Test
public void testEqual_PostCodeNotEqual() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    want.object(user).eq(new User("jobs.he", "浙江省杭州市", "310012", 45));
}

```

这显然和我们的测试初衷背道而驰，我们需要对各个属性都进行判断。针对这种情况，jTester提供了反射判断相等的API：eqByReflect

```

@Test
public void testEqual_eqByReflect() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    want.object(user).eqByReflect(new User("jobs.he", "浙江省杭州市", "310012", 45));
}

```

这样，我们执行testEqual\_eqByReflect测试方法，控制台上就会打印出2个对象的差异点：

```

FAILED: testEqual_ReflectionEq
java.lang.AssertionError:
Expected:
Expected: User<name="jobs.he", address="浙江省杭州市", postCode="310012", age=45>
Actual: User<name="jobs.he", address="浙江省杭州市", postCode="310010", age=30>

--- Found following differences ---
age: expected: 45, actual: 30
postCode: expected: "310012", actual: "310010"

--- Difference detail tree ---
expected: User<name="jobs.he", address="浙江省杭州市", postCode="310012", age=45>
actual: User<name="jobs.he", address="浙江省杭州市", postCode="310010", age=30>

age expected: 45
age actual: 30

postCode expected: "310012"
postCode actual: "310010"

but: was <org.jtester.module.assertion.beans.User@c087cb80>
at ext.jtester.hamcrest.MatcherAssert.assertThat (MatcherAssert.java:21)

```

如图中所示，jTester框架详细列出了2个User对象属性的差异值。反之，如果2个User对象的所有属性值都是相同的，那么测试就会通过。

### 例 3.22. 反射比较2个PoJo对象

```

@Test
public void testEqual_eqByReflect() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    want.object(user).eqByReflect(new User("jobs.he", "浙江省杭州市", "310010", 30));
}

```

上面演示了全属性比较的情况，jTester断言框架还允许你设置比较模式，忽略某些字段的比较，比如我们对User对象进行比较时，我们不在乎address是否相同，那么我们就可以使用EqMode.IGNORE\_DEFAULTS模式，并把期望对象的address值设为null。这样jTester在比较实际User值时，将不会对address对象进行断言。具体示例如下：

### 例 3.23. 反射比较2个PoJo对象，忽略特定字段比较示例

```

@Test
public void testEqual_eqByReflect_IgnoreDefault() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    want.object(user).eqByReflect(new User("jobs.he", null, "310010", 30),
        EqMode.IGNORE_DEFAULTS);
}

```

模式EqMode.IGNORE\_DEFAULTS表示，当期望值属性设置为默认值（对象为null,primitive数值为0,boolean为false）时，忽略实际对象的对应属性的比较。EqMode总共有3种模式：

- IGNORE\_DEFAULTS  
当期望值的属性设置为默认值时，忽略对实际值对应属性的比较。
- IGNORE\_DATES  
忽略实际值日期属性的比较。
- IGNORE\_ORDER  
当属性值是个数组或集合类型是，比较这个属性时，忽略2个数组（集合）中对象的顺序因素。

IGNORE\_DEFAULTS的例子我们上面已经解释了，IGNORE\_DATES的情况比较简单，剩下一个IGNORE\_ORDER我们给个具体例子演示一下。假定我们的User对象中新增了一个属性phones（电话），每个用户可能有多个电话，所有用String[]来表示：

```

public class User {
    private String name;

    private String address;

    private String postCode;

    private int age;

    private String[] phones;

    public void setPhones(String[] phones) {
        this.phones = phones;
    }
}

```

首先我们不加IGNORE\_ORDER模式运行下面的例子：

```

@Test
public void testEqual_eqByReflect_IgnoreOrder() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    user.setPhones(new String[] { "15990038123", "13906471234" });

    User expected = new User("jobs.he", null, "310010", 30);
    expected.setPhones(new String[] { "13906471234", "15990038123" });

    want.object(user).eqByReflect(expected, EqMode.IGNORE_DEFAULTS);
}

```

运行上面例子，控制台报出数组phones的差异：

```

FAILED: testEqual_ReflectionEq_IgnoreOrder
java.lang.AssertionError:
Expected:
Expected: User<name="jobs.he", address=null, postCode="310010", age=30, phones=["13906471234", "15990038123"]
Actual: User<name="jobs.he", address="浙江省杭州市", postCode="310010", age=30, phones=["15990038123", "13906471234"]

--- Found following differences ---
phones[1]: expected: "15990038123", actual: "13906471234"
phones[0]: expected: "13906471234", actual: "15990038123"

--- Difference detail tree ---
expected: User<name="jobs.he", address=null, postCode="310010", age=30, phones=["13906471234", "15990038123"]
actual: User<name="jobs.he", address="浙江省杭州市", postCode="310010", age=30, phones=["15990038123", "13906471234"]

phones expected: ["13906471234", "15990038123"]
phones actual: ["15990038123", "13906471234"]

phones[1] expected: "15990038123"
phones[1] actual: "13906471234"

phones[0] expected: "13906471234"
phones[0] actual: "15990038123"

```

从上图红框的地方，我们看出比较的不同。但实际上，我们不关心这2个电话谁前谁后，只要有他们就可以了，这样，我们可以给比较加上模式IGNORE\_ORDER，忽略数组和集合中元素的顺序。

## 例 3.24. 反射比较2个PoJo对象，忽略数组（集合）属性中元素顺序

```

@Test
public void testEqual_eqByReflect_IgnoreOrder() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    user.setPhones(new String[] { "15990038123", "13906471234" });

    User expected = new User("jobs.he", null, "310010", 30);
    expected.setPhones(new String[] { "13906471234", "15990038123" });

    want.object(user).eqByReflect(expected, EqMode.IGNORE_ORDER, EqMode.IGNORE_DEFAULTS);
}

```

这时运行上面的例子，测试就可以通过。

针对上面提到的比较模式，jTester也提供了3个快捷比较API：

- eqIgnoreDefault: 忽略默认值时的比较，等价于设置了EqMode.IGNORE\_DEFAULTS的eqByReflect比较。

## 例 3.25. 忽略默认值时比较2个对象

```

@Test
public void testEqIgnoreDefault() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    want.object(user).eqIgnoreDefault(new User("jobs.he", null, "310010", 30));
}

```

- eqIgnoreOrder: 忽略元素顺序时的比较，等价于设置了EqMode.IGNORE\_ORDER的eqByReflect比较。

## 例 3.26. 忽略元素顺序时比较2个对象

```

@Test
public void testEqIgnoreOrder() {
    String[] phones = new String[] { "15990038123", "13906471234" };
    String[] expecteds = new String[] { "13906471234", "15990038123" };
    want.object(phones).eqIgnoreOrder(expecteds);
}

```

- eqIgnoreAll: 相当于设置了IGNORE\_DEFAULTS, IGNORE\_DATES, IGNORE\_ORDER三者都设置的eqByReflect比较。

## 例 3.27. 忽略默认值，日期，元素顺序时比较2个对象

```

@Test
public void testEqIgnoreAll() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    user.setPhones(new String[] { "15990038123", "13906471234" });

    User expected = new User("jobs.he", null, "310010", 30);
    expected.setPhones(new String[] { "13906471234", "15990038123" });

    want.object(user).eqIgnoreAll(expected);
}

```

上面我们介绍了如何判断2个对象是否相等，或者在某些情况下是否相等。还有一种情况，我们需要判断2个对象引用是否是同一个对象，这时候光判断对象相等是不够的，必须判断对象的引用地址是否相同，我们可以使用API: `the.object(obj).same(other)` 来进行判断。

### 例 3.28. 判断2个对象是否是同一个对象

```
@Test
public void testSame() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);

    User expected = user;
    expected.setName("wawa");
    // 断言user对象就是expected对象，并且用户名称已经被修改为"wawa"了
    want.object(user).same(expected).propertyEq("name", "wawa");
}
```

到现在为止，我们介绍了如何判断2个对象的是否相同，或者排除某些属性后2个对象是否相同。这个是显式排除属性的比较，如果我们要显式的指定一个或多个属性比较时，应该要怎么做呢？下节我们将介绍对象的属性比较。

## 对2个PoJo对象指定的属性进行比较

前面的2个对象直接比较，反射比较，过滤属性比较等技术已经可以解决很多场景下对普通PoJo对象的断言的问题。但假如一个PoJo对象有二三十个属性，但我们真正关心的属性就其中的一两个，或者三四个。这时候如果你按照前面过滤的方式你比较对象，我们就必须显式的设定null的属性。这个会大大的增加我们的测试工作量，而且代码很难看，不直观。jTester提供了对指定的属性直接进行断言的语法。

`propertyEq(String property, Object expected)`，用于验证单个属性的值等于（通过反射比较）等于期望值expected。

### 例 3.29. 普通对象的单属性比较

```
@Test
public void testPropertyEq() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    want.object(user).propertyEq("name", "jobs.he");
}
```

上面的例子我们简单验证了一个String属性（user对象的name），我们在String断言的章节介绍了很多对字符串做宽松断言的语法，这这里我们同样也可以使用。下面的例子，我们忽略大小写和空格，对user name做了断言：

### 例 3.30. 对PoJo对象String属性的宽松断言

```
@Test
public void testPropertyEq_StringMode() {
    User user = new User("Jobs . He", "浙江省杭州市", "310010", 30);
    // 忽略name属性中的大小写和空格
    want.object(user).propertyEq("name", "jobs.he", StringMode.IgnoreSpace,
    StringMode.IgnoreCase);
}
```

但如果我要对string属性做更复杂的断言，比如我们希望user对象的address地址中包含了“杭州”这个关键字，并且以“浙江”开头，以“699号”结尾。这么复杂的表达方式，我们使用上面简单的API可能就无法无能为力了。这时候，我们要么直接把address的属性通过get方式，或者通过反射的方式把值给取出来。作`the.string(address)...`形式的断言。或者，我们要用到了延后生效的断言：这个延后生效的断言希望传进来的值必须包含了“杭州”这个关键字，并且以“浙江”开头，以“699号”结尾。

### 例 3.31. 使用延后生效断言判断PoJo对象的属性

```
@Test
public void testPropertyEq_Matcher() {
    User user = new User("jobs.he", "浙江省杭州市滨江区网商路699号", "310010", 30);
    want.object(user).propertyMatch("address", the.string().contains("杭州").start("浙江").end("699号"));
}
```

我们已经可以通过各种方式，对一个简单的属性对象表达我们想要的结果。如果属性是个复杂对象，我们同样也可以使用上面的API进行比较。比如user的地址是个数组，如果直接比较同样可以用上面的propertyEq和propertyMatch方式，但如果我们期望忽略数组中的元素顺序，那么我们就必须添加EqMode模式。

### 例 3.32. 使用EqMode判断对象属性

```
@Test
public void testPropertyEq_EqMode() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    user.setPhones(new String[] { "15990038123", "13906471234" });

    want.object(user).propertyEq("phones", new String[] { "13906471234", "15990038123" },
    EqMode.IGNORE_ORDER);
}
```

上面针对单个属性我们作了断言，如果我们要同时断言2个或2个以上的属性，我们就可以用propertyEqMap(DataMap map, EqMode...modes)这个API（老版本reflectEqMap）。它的参数是个DataMap对象，其实就是一个LinkedHashMap<String, Object>对象。它的key值表示要比较的属性名称，它的value值是对应属性的期望值。比如我们同时要比User对象的名字和地址这2个属性，我们可以这样写：

### 例 3.33. propertyEqMap（老版本reflectEqMap）比较多个属性值

```
@Test
public void testPropertyEqMap() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    want.object(user).propertyEqMap(new DataMap() {
        {
            this.put("name", "jobs.he");
            this.put("address", "浙江省杭州市");
        }
    });
}
```

同样的，我们也可以给propertyEqMap（老版本reflectEqMap）增加比较模式EqMode的，但这种模式的添加是针对比较对象全局的，无法针对单一属性设置。比如下面的比较：

```
@Test
public void testPropertyEqMap_EqMode() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    user.setPhones(new String[] { "15990038123", "13906471234" });

    want.object(user).propertyEqMap(new DataMap() {
        {
            this.put("name", "jobs.he");
            this.put("address", "浙江省杭州市");
            this.put("phones", new String[] { "13906471234", "15990038123" });
        }
    });
}
```

}

如果我们直接这样比较，那jTester会抛出下面的错误：

```

FAILED: testPropertyEqMap_EqMode
java.lang.AssertionError:
Expected: [{ "name"="jobs.he", "address"="浙江省杭州市", "phones"=["13906471234", "15990038123"]}]
Actual: [{ "address"="浙江省杭州市", "name"="jobs.he", "phones"=["15990038123", "13906471234"]}]

--- Found following differences ---
[0].phones[1]: expected: "15990038123", actual: "13906471234"
[0].phones[0]: expected: "13906471234", actual: "15990038123"

--- Difference detail tree ---
expected: [{ "name"="jobs.he", "address"="浙江省杭州市", "phones"=["13906471234", "15990038123"]}]
actual: [{ "address"="浙江省杭州市", "name"="jobs.he", "phones"=["15990038123", "13906471234"]}]

[0].phones expected: ["13906471234", "15990038123"]
[0].phones actual: ["15990038123", "13906471234"]

[0].phones[1] expected: "15990038123"
[0].phones[1] actual: "13906471234"

[0].phones[0] expected: "13906471234"
[0].phones[0] actual: "15990038123"

but: was <org.jtester.module.assertion.beans.User@c087cb80>

```

这个错误信息我们前面已经看到过，就是属性phones的2个属性值顺序和期望的字符串数组不一致，但我们不关心它们的顺序，所以我们同样可以给比较增加个EqMode.IgnoreOrder模式。

**例 3.34. propertyEqMap（老版本reflectEqMap）使用EqMode模式比较多个属性值**

```

@Test
public void testPropertyEqMap_EqMode() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    user.setPhones(new String[] { "15990038123", "13906471234" });

    want.object(user).propertyEqMap(new DataMap() {
        {
            this.put("name", "jobs.he");
            this.put("address", "浙江省杭州市");
            this.put("phones", new String[] { "13906471234", "15990038123" });
        }
    }, EqMode.IGNORE_ORDER);
}

```

到目前为止，我们都只是对普通对象的直接属性进行比较。如果对象的属性是个复合对象，我们希望对属性的属性的属性的属性值进行断言，jTester是否支持呢？jTester的属性是支持ognl格式的，即以"."级联各个属性的值。比如，我再给User对象增加一个属性 assistor：

```

public class User {
    private String name;

    private String[] phones;

    private User assistor;

    public void setAssistor(User assistor) {
        this.assistor = assistor;
    }
}

```

这样，我们对assistor属性的属性就可以这样表示：assistor.name, assistor.phones。相应的测试可以写成这样：



## 例 3.35. 级联属性的比较

```

@Test
public void testCascadeProperty() {
    User user = new User("jobs.he", "浙江省杭州市", "310010", 30);
    user.setPhones(new String[] { "15990038123", "13906471234" });
    user.setAssistor(new User() {
        {
            this.setName("david");
            this.setAddress("天涯海角");
            this.setPhones(new String[] { "086-571-88888888", "13305718888" });
        }
    });
    want.object(user).propertyEq("assistor.name", "david").propertyEqMap(new DataMap() {
        {
            this.put("name", "jobs.he");
            this.put("phones", new String[] { "13906471234", "15990038123" });
            this.put("assistor.phones", new String[] { "13305718888", "086-571-88888888" });
        }
    }, EqMode.IGNORE_ORDER);
}

```

到目前为止，我们已经把PoJo的直接比较，反射比较，属性比较的内容已经讲解完毕，从下章开始我们介绍复杂类型的断言。

## 断言Map对象

在某种形式上，我们可以把Map对象看做是PoJo对象，把String类型的key看做对象的属性，value看做属性值。Map<String,?>对象。

特别是Map<String,?>类型，我们可以所以针对PoJo对象的断言语法同样适用于

## 例 3.36. 通过反射比较断言Map对象

```

@Test
public void demoMapAssert() {
    Map<String, Object> actual = new HashMap() {
        {
            this.put("my_string", "i am a string.");
            this.put("my_int", 100);
            this.put("my_array", new int[] { 100, 999 });
        }
    };
    want.map(actual).eqByReflect(new HashMap() {
        {
            this.put("my_string", "i am a string.");
            this.put("my_int", 0);
            this.put("my_array", new int[] { 999, 100 });
        }
    }, EqMode.IGNORE_ORDER, EqMode.IGNORE_DEFAULTS);
}

```

我们也可以指定key值来断言Map对象。



## 例 3.37. 通过指定属性断言Map

```

@Test
public void demoMapPropertyEq() {
    Map<String, Object> actual = new HashMap() {
        {
            this.put("my_string", "i am a string.");
            this.put("my_int", 100);
            this.put("my_array", new int[] { 100, 999 });
        }
    };
    want.map(actual).propertyEq("my_string", "i am a string.").propertyEqMap(new DataMap() {
        {
            this.put("my_int", 0);
            this.put("my_array", new int[] { 999, 100 });
        }
    }, EqMode.IGNORE_ORDER, EqMode.IGNORE_DEFAULTS);
}

```

## 注意

对应Map对象来说，eqByReflect和propertyEq这2个方法是非常相似的，但reflectionEq是进行全属性比较的，propertyEq是进行指定属性比较的。

例如对上面比较2个属性的propertyEqMap，如果改为eqByReflect来比较：

```

@Test
public void demoMapEqByReflect() {
    Map<String, Object> actual = new HashMap() {
        {
            this.put("my_string", "i am a string.");
            this.put("my_int", 100);
            this.put("my_array", new int[] { 100, 999 });
        }
    };
    want.map(actual).eqByReflect(new DataMap() {
        {
            this.put("my_int", 0);
            this.put("my_array", new int[] { 999, 100 });
        }
    }, EqMode.IGNORE_ORDER, EqMode.IGNORE_DEFAULTS);
}

```

运行上面的测试，测试会跑出下面的错误：

```

FAILED: demoMapReflectionEq
java.lang.AssertionError:
Expected:
  Expected: {"my_int"=0, "my_array"=[999, 100]}
  Actual: {"my_string"="i am a string.", "my_array"=[100, 999], "my_int"=100}

--- Found following differences ---
"my_string": expected: "i am a string.", actual: ""

--- Difference detail tree ---
expected: {"my_int"=0, "my_array"=[999, 100]}
actual: {"my_string"="i am a string.", "my_array"=[100, 999], "my_int"=100}

"my_string" expected: "i am a string."
"my_string" actual: ""

```

从错误中我们可以看出，即使我们在期望值中没有指定"my\_string"这个key，jTester框架也是会去比较的，因为eqByReflect这个方法是对整个对象进行比较的。

上面讲了Map和普通PoJo相同的断言方式，同时Map有自己的特色，所以也应该有针对Map特色的断言。

Map是个容器，jTester可以对容器类的对象断言容器中元素的数量进行断言。

want.map(map).sizeEq(size) 或 sizeEq(size)	断言实际对象map中的元素个数刚好等于size。
want.map(map).sizeNe(size)	断言实际对象map中的元素个数不等于size。
want.map(map).sizeGt(size)	断言实际对象map中的元素个数大于size。
want.map(map).sizeGe(size)	断言实际对象map中的元素个数大于等于size。
want.map(map).sizeLt(size)	断言实际对象map中的元素个数小于size。
want.map(map).sizeLe(size)	断言实际对象map中的元素个数小于等于size。
want.map(map).sizeBetween(min, max)	断言实际对象map中的元素个数大于等于min，且小于等于max。

### 例 3.38. 断言map对象元素个数

```
@Test
public void testMapSize() {
    Map<String, String> actual = new HashMap<String, String>() {
        {
            this.put("key1", "value1");
            this.put("key2", "value2");
            this.put("key3", "value3");
        }
    };
    want.map(actual).sizeEq(3).sizeGe(2).sizeLt(4).sizeBetween(2, 4);
}
```

上面例子断言actual map中元素等于3，大于等于2，小于4，且在2和4之间。哈哈，怎么这么多废话啊，演示而已！除了对元素个数进行判断外，Map是key-value结构，我们也可以直接对key或value进行断言。

hasKeys(key1, key2, ..., keyN)	断言Map对象中存在参数中指定的key值。
hasValues(value1, value2, ..., valueN)	断言Map对象中存在参数中指定的value值。
hasEntry(key1, value1, key2, value2, ..., keyN, valueN)	断言Map对象中存在参数中指定值对。
hasEntry(Entry1, Entry2, ..., EntryN)	断言Map对象中存在参数中指定值对。

### 例 3.39. 断言map对象key值或value值

```
@Test
public void testMapEntry() {
    Map<Integer, String> actual = new HashMap<Integer, String>() {
        {
            this.put(new Integer(1), "value1");
            this.put(new Integer(2), "value2");
            this.put(new Integer(3), "value3");
        }
    };
    want.map(actual).hasKeys(1, 3).hasValues("value1", "value2").hasEntry(1, "value1", 3, "value3");
}
```

## 断言Collection对象或Array对象

在java世界中，除了单个对象（简单对象，复合对象）外，还存在一类集合类型。主要包括以java.util.Collection为基类的List, Set, Collection，和数组对象。在jTester断言中，这2类对象是被当做同一类型来处理的，都是List类型。

对于List类型，我们可能需要判断里面的元素，或者元素的属性，元素数量的多少等等。容器元素数量的判断和Map的断言一样，使用sizeXX(size)的语法。

### 例 3.40. 集合大小或数组长度的断言

```
@Test
public void testListSize() {
    String[] arr = new String[] { "a", "b", "c" };
    want.list(arr).sizeEq(3);

    List list = Arrays.asList(1, 3, 4, 5, 6);
    want.list(list).sizeEq(5);
}
```

要判断集合中或数组中是否包含特定的元素，可以使用hasItems(...)系列断言。

- hasItems(Object item)

断言集合（数组）中包含有参数value。下面的例子，断言字符串数组中包含字符串"b"，Integer集合中包含数字5：

### 例 3.41. 断言集合包含特定元素

```
@Test
public void testListHasItem() {
    String[] arr = new String[] { "a", "b", "c" };
    want.list(arr).hasItems("b");

    List list = Arrays.asList(1, 3, 4, 5, 6);
    want.list(list).hasItems(5);
}
```

- hasAllItems(Object value, Object... values)

断言集合（数组）中包含有参数中指定的所有元素，和多个hasItems(item)的效果一样。下面的例子，断言字符串数组中包含字符串"b"和"c"，Integer集合中包含数字5, 1 和 4：

### 例 3.42. 断言集合包含一系列元素

```
@Test
public void testListHasAllItem() {
    String[] arr = new String[] { "a", "b", "c" };
    want.list(arr).hasAllItems("b", "c");

    List list = Arrays.asList(1, 3, 4, 5, 6);
    want.list(list).hasAllItems(5, 1, 4);
}
```

- hasAnyItems(Object value, Object... values)

断言集合（数组）中包含有参数中指定的元素之一。即参数中的元素有一个在集合中，断言就通过；没有都不在集合中，断言失败。下面的例子中，数组虽然没有包含"f"，但包含了"b"，所以断言通过。集合虽然没有包含9和10，但包含了5，所以断言通过。

## 例 3.43. 断言集合包含指定元素之一

```
@Test
public void testListHasAnyItem() {
    String[] arr = new String[] { "a", "b", "c" };
    want.list(arr).hasAnyItems("b", "f");

    List list = Arrays.asList(1, 3, 4, 5, 6);
    want.list(list).hasAnyItems(5, 9, 10);
}
```

hasItems系列断言，是直接比较集合中的对象和期望的对象。如果对于复杂的对象，不需要那么严格的相等，上面的方法是无能为力的。这时候，我们就需要用到延后断言：期望元素具有上面性质。我们可以使用matchXXX系列的断言。

- allItemsMatchAll(Matcher matcher, Matcher... matchers)

期望集合中所有的元素都和参数中的断言器匹配。即只要有一个元素不符合任意一个断言器的要求，断言就失败。下面的例子要求数组中3个字符串都满足参数的2个延后断言（Matcher）的要求，第一个Matcher要求每个字符都是以"test"开头，第二个Matcher要求每个字符串都满足正则表达式"\w{4}\d"。

## 例 3.44. 断言列表中所有元素符合所有的断言要求

```
@Test
public void testAllItemsMatchAll() {
    String[] list = new String[] { "test1", "test2", "test3" };
    want.list(list).allItemsMatchAll(the.string().start("test"),
    the.string().regular("\\w{4}\\d"));
}
```

- allItemsMatchAny(Matcher matcher, Matcher... matchers)

期望集合中所有的元素都和参数中的任意一个断言器匹配。即只要有一个元素不符合都不符合所有断言器的要求，断言就失败。下面的例子要求数组中3个字符串必须都满足参数中2个延后断言器中的至少一个。实际情况是，第一个Matcher，3个字符串都是以"test"开头，所以都满足；第二个Matcher，3个字符串只有一个等于"test1"，所以不满足。但2个断言器中，有一个被完全满足，所有整个断言器是通过的。

## 例 3.45. 断言列表中所有元素必须都必须满足至少一个断言器

```
@Test
public void testAllItemsMatchAny() {
    String[] list = new String[] { "test1", "test2", "test3" };
    want.list(list).allItemsMatchAny(the.string().start("test"),
    the.string().eq("test1"));
}
```

- anyItemsMatchAll

期望列表中必须有一个元素符合所有的断言器要求，对下面的例子列表中test1符合：a、等于"test1"，b、以test开头，c、符合正则表达式"\w{4}\d"。虽然其它2个字符串"test2","test3"不能完全满足参数中的3个断言器，但这个断言器还是通过的。

## 例 3.46. 列表中任意元素符合所有的断言

```
@Test
public void testMatch_AnyItemMatchAll() {
    String[] list = new String[] { "test1", "test2", "test3" };
    want.list(list).anyItemsMatchAll(the.string().eq("test1"),
    the.string().start("test"),
    the.string().regular("\\w{4}\\d"));
}
```

- anyItemsMatchAny

这是最宽松的断言，期望列表中至少有一个元素符合一个断言就可以了。

## 例 3.47. 列表中任意元素符合断言

```
@Test
public void testAnyItemsMatchAny() {
    String[] list = new String[] { "test1", "test2", "test3" };
    want.list(list).anyItemsMatchAny(the.string().eq("test1"),
    the.string().eq("test2"), the.string().eq("test4"));
}
```

- match(ItemsMode itemsMode, MatchMode matchMode, Matcher matcher, Matcher... matchers)

上面4个API其实是match函数中 ItemsMode和MatchMode这2个参数的组合：

- allItemsMatchAll:ItemsMode.AllItems, MatchItems.MatchAll
- allItemsMatchAny:ItemsMode.AllItems, MatchItems.MatchAny
- anyItemsMatchAll:ItemsMode.AnyItems, MatchItems.MatchAll
- anyItemsMatchAny:ItemsMode.AnyItems, MatchItems.MatchAny

以上的断言都是针对集合中的元素，做直接或间接的断言。我们在PoJo章节介绍了属性的断言，同样的，我们也可以直接对集合中的元素对象做属性断言。这个功能是jTester中最强大也是用的最多的功能点之一。我们在PoJo和Map那一章介绍过propertyEqMap的用法（老版本reflectEqMap），但那个是针对单个对象的，在列表中也适用这个API，不过需要加个参数：propertyEqMap(int size, DataMap map)。这个函数期望列表中的元素是size个，其要比较的属性是DataMap的key值，DataMap的value是一个数组，分别一一对应列表中size个元素的属性值。

## 例 3.48. 对列表中元素进行属性断言

```
@Test
public void testPropertyEq() {
    User[] users = new User[] { new User("jobs", "杭州", "310012", 30),
    new User("mary", "上海", "111222", 28),
    new User("wade", "北京", "333444", 40) };
    want.list(users).propertyEqMap(3, new DataMap() {
        {
            this.put("name", "jobs", "mary", "wade");
            this.put("address", "杭州", "上海", "北京");
        }
    });
}
```

这个例子断言数组users里面有3个元素，并且判断了2个属性"name"和"address"，对应于3个元素，它们分别是("jobs","杭州"),("mary","上海")和("wade","北京")。

因为这个断言在数据库数据验证方面用的特别多，所以在数据库验证章节有特别的介绍。更详细的语法大家可以参考那个章节的内容。

## 第 4 章 使用DataMap准备和验证数据

### 为什么要使用DataMap?

早先的jTester中提供了dbFit方式来准备和验证数据库数据，应该来说，这个工具解决了很多问题。实际使用过程中，开发同学反映编辑和准备数据比较麻烦。数据操作错误需要在单独的html文件中才能查看到，也比较麻烦。jTester从1.1.6开始推出了一种新的数据库数据准备和验证的方法 -- DataMap方式。DataMap对比DbFit有以下几个特性：

- 准备数据和验证数据是在java代码中，无需额外的文件。
- 因为只有java代码，数据编辑会更方便一些。
- 验证数据库数据和jTester中其它断言方式一致，错误信息直接显示在测试方法上。
- 只需要关注自己感兴趣的字段，无关的字段框架会自动会帮忙填充。
- 构造数据灵活，可以根据自己需要构造特定规则的数据。

我们下面通过示例来简单的演示jTester是如何通过DataMap往数据库中插入数据和验证数据中已经存在的数据。

#### 例 4.1. 往数据库中插入一条数据

```
db.table("tdd_user").clean().insert(new DataMap() {  
    {  
        this.put("id", "1");  
        this.put("first_name", "darui.wu");  
        this.put("my_date", new Date());  
    }  
}).commit();
```

DataMap(它是 LinkedHashMap<String, Object> 子类)是jTester中用于准备和验证数据的关键对象, 上面那段代码完成了4件事：

- db.table("tdd\_user")  
指定了插入数据的操作对象(表tdd\_user)，这条语句接下的操作都是针对表tdd\_user的。
- clean()  
清空表tdd\_user中所有数据。
- insert( ... )  
往表tdd\_user中插入一条数据，其中id字段值为1，first\_name字段值为darui.wu，my\_date字段值为客户端的当前日期。
- commit()  
提交前面删除数据操作和插入数据操作。

运行上面那段简单的代码后，数据库中数据状态应该如下：

id	first_name	last_name	post_code	salary	address_id	my_date	big_int
1	darui.wu					2011-11-02 10:59:53.0	

## 例 4.2. 验证数据库中已存在的数据（单条数据）

```
db.table("tdd_user").query().propertyEqMap(new DataMap() {
{
    this.put("id", 1);
    this.put("first_name", "wu");
    this.put("last_name", "json");
}
});
```

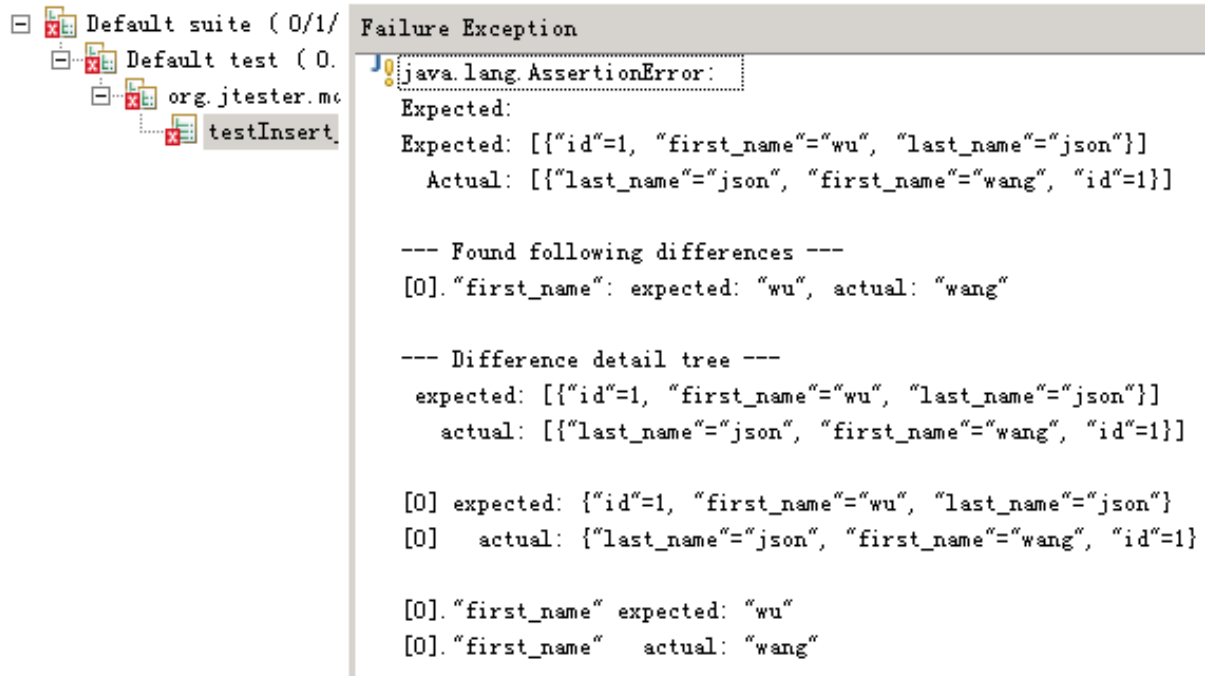
这段代码片段完成了3件事：

- `db.table("tdd_user")`  
同插入数据操作，这条语句指定了验证数据的操作对象(表tdd\_user)，这条语句接下的操作都是针对表tdd\_user的。
- `query()`  
执行 `select * from tdd_user` 这条SQL语句，返回数据结果集的断言器。
- `propertyEqMap( ... )`  
数据结果集和`propertyEqMap`（老版本`reflectEqMap`）方法中的参数（`DataMap`数据）进行比较。

## 注意

jTester只比较在`DataMap`中指定的字段，没有指定的字段不做比较。

如果比较正确，就会显示测试通过。如果数据不匹配，比如“first\_name”的值不为“wu”，测试就会显示如下错误：



The screenshot shows a test failure in jTester. The left pane displays the project structure with 'testInsert' selected. The right pane shows the 'Failure Exception' details for a 'java.lang.AssertionError'.

```
Failure Exception
java.lang.AssertionError:
Expected:
Expected: [{"id"=1, "first_name"="wu", "last_name"="json"}]
Actual: [{"last_name"="json", "first_name"="wang", "id"=1}]

--- Found following differences ---
[0]. "first_name": expected: "wu", actual: "wang"

--- Difference detail tree ---
expected: [{"id"=1, "first_name"="wu", "last_name"="json"}]
actual: [{"last_name"="json", "first_name"="wang", "id"=1}]

[0] expected: {"id"=1, "first_name"="wu", "last_name"="json"}
[0] actual: {"last_name"="json", "first_name"="wang", "id"=1}

[0]. "first_name" expected: "wu"
[0]. "first_name" actual: "wang"
```

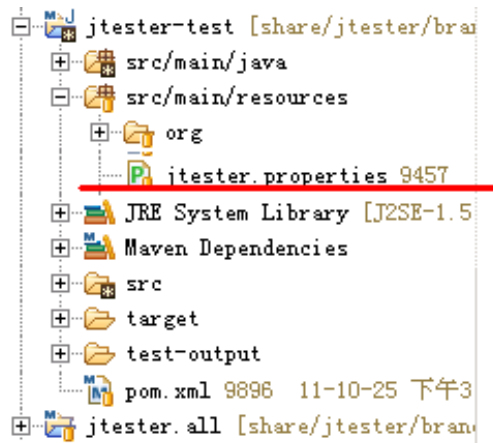
从图中可以看出，jTester对不匹配的数据有着详细的说明，让开发一眼就能看出错误在哪里？

jTester的`DataMap`有着很强大，同时又很容易使用的功能，但在继续深入功能讲解之前，我们必须把要操作的数据库连接给配置好。

## 配置jtester.properties文件

要在jTester框架下进行数据库相关的测试，必须在测试代码的classpath根路径下放置一个jtester.properties文件。并在该文件中配置相应的数据源。





在classpath根路径下建好jtester.properties文件后，需要配置下面选项的值。

- database.type  
数据库类型，目前支持mysql,oracle两种
- database.url  
数据库连接url，比如jdbc:mysql://localhost/presentationtd?characterEncoding=UTF8
- database.userName  
连接数据库的用户名
- database.password  
用户密码
- database.schemaNames  
数据库的具体schema
- database.driverClassName  
数据库连接驱动class的全称

为了方便大家理解，下面分别给一个mysql配置示例和一个oracle配置示例

#### 例 4.3. mysql配置示例

```
database.type=mysql
database.url=jdbc:mysql://localhost/presentationtd?characterEncoding=UTF8
database.userName=root
database.password=password
database.schemaNames=presentationtd
database.driverClassName=com.mysql.jdbc.Driver
```

#### 例 4.4. oracle配置示例

```
database.type=oracle
database.url=jdbc:oracle:thin:@10.20.14.45:1521:crmp?
args[applicationEncoding=UTF-8,databaseEncoding=UTF-8]
database.userName=eve_test
database.schemaNames=eve_test
database.password=xxxx
database.driverClassName=com.alibaba.china.jdbc.SimpleDriver
```



好，文件配置好以后，我们就可以在测试代码中使用DataMap方式进行数据库测试了。

## DataMap语法详解

在本章的第一节，我们通过2个简单例子介绍了如何往数据库中插入数据和验证数据库中已存在的数据。但例子只演示了一条数据的操作，如何对多条数据进行操作呢？本节我们将详细介绍DataMap的各种使用方式。

在JTester框架中内置了一个变量 `db`，只要测试类继承了JTester这个基类，就可以直接使用这个变量。对数据库的所有操作都可以通过这个变量所关联的api完成。

```
public interface IJTester {

    final IDBOperator db = new DBOperator();

    //其它定义
}
```

从定义中，我们知道db变量是一个IDBOperator对象实例，那么IDBOperator提供了哪些操作呢？

```
public interface IDBOperator {
    /**
     * 清空指定的若干张表的数据
     *
     * @param table
     * @param mores
     * @return
     */
    public IDBOperator cleanTable(String table, String... mores);

    /**
     * 针对表进行操作
     *
     * @param table
     * @return
     */
    public ITableOp table(String table);

    /**
     * 提交数据
     *
     * @return
     */
    public IDBOperator commit();

    /**
     * 回滚未提交的数据
     *
     * @return
     */
    public IDBOperator rollback();

    /**
     * 查询数据列表，并进行断言
     *
     * @param sql
     * @return
     */
    public ICollectionAssert query(String sql);

    /**
     * 执行sql语句集合
     *
     */
}
```

```

* @param sqlSet
* @return
*/
public IDBOperator execute(SqlSet sqlSet);

/**
 * 执行单条语句，执行多条请使用 execute(SqlSet)
 *
 * @param sql
 * @return
 */
public IDBOperator execute(String sql);

/**
 * 执行文件中的sql语句
 *
 * @param sqlFile
 * @return
 */
public IDBOperator execute(File sqlFile);

/**
 * 使用数据源来执行下列的数据库操作<br>
 * dataSource的名称在jtester.properties文件中定义
 *
 * @param dataSource
 * @return
 */
public IDBOperator useDB(String dataSource);

/**
 * 使用jtester.properties中配置的默认数据源
 *
 * @return
 */
public IDBOperator useDefaultDB();
}

```

根据api定义，我们大概归纳一下IDBOperator的操作：

- 数据的提交(commit)和回滚(rollback)
- 清空多张表的数据 cleanTable("table1","table2")
- 对指定表进行操作 table("table")
- 直接执行查询语句query("select ...")，返回一个数据集(Map List结构)，并进行断言。
- 直接执行SQL语句。
- 多个数据库切换操作。

只要我们已经配置好jtester.properties文件，就可以执行数据库测试的相关操作，下面我们将一一详细介绍各个功能的使用。。

## 对指定表进行数据插入操作

对指定表进行操作：db.table("table")，返回的是一个ITableOp对象。ITableOp是jTester DataMap中最重要的操作对象。针对单表的准备数据和验证数据都可以通过这个对象来完成，我们首先来看看ITableOp的API设计：

```

public interface ITableOp {
    /**
     * 清空数据表
     */
    ITableOp clean();

    /**
     * 根据json插入数据
     */
    ITableOp insert(String json, String... more);

    /**
     * 批量插入count条数据
     * @param count 需要插入的数据的数量
     * @param data 需要插入的数据 Map, key:字段, value: n条数据的数组集合和策略
     */
    ITableOp insert(int count, DataMap data);

    /**
     * 根据Map的key（表字段）插入数据
     */
    ITableOp insert(DataMap data, DataMap... datas);

    /**
     * 插入数据集
     */
    ITableOp insert(AbstractDataSet dataset);

    /**
     * 查询表数据，做数据断言
     */
    ICollectionAssert query();

    /**
     * 根据条件查询数据，并返回数据断言器
     */
    ICollectionAssert queryWhere(String where);

    /**
     * 根据条件查询数据，并返回数据断言器
     */
    ICollectionAssert queryWhere(DataMap where);

    /**
     * 查询表count(*)的值，并且返回断言器
     */
    INumberAssert count();
}

```

总的来说，对象ITableOp可以完成下面几个操作：

- 清空单张表中的数据。

```
db.table("YOUR_TABLE").clean().commit();
```

- 以各种方式插入数据。

```
db.table("YOUR_TABLE").insert(...);
```

- 验证数据库中已存在的数据。

```
db.table("YOUR_TABLE").query()....;
```

## 用DataMap插入多条数据

我们在本章第一节介绍了如何往数据库中插入一条数据。但现实中准备数据往往需要多条数据，那么如何简便的往数据库中插入多条数据呢？恩，我们可以使用API: `db.table("YOUR_TABLE").insert(n, DataMap)`; 往数据库中批量插入n条数据。现在先给一个简单的代码示例，然后再详细讲解语法。

```
db.table("tdd_user").clean().insert(2, new DataMap() {
    {
        this.put("id", new Integer[] { 1, 2 });
        this.put("first_name", new Object[] { "darui.wu", "data.iterator" });
    }
});
```

示例往数据库中插入了2条记录，操作完毕数据库中的结果集如下：

id	first_name	last_name	post_code	salary	address_id	my_date	big_int
1	darui.wu						
2	data.iterator						

在插入多条数据时，DataMap的key值还是代表数据库中相应字段名称，但value值不再是单一数据对象，而是数据对象数组。这样，`this.put("id", new Integer[] { 1, 2 })`，就表示插入2个数据时，第一条数据的ID字段值为1，第二条为2；同样的，`this.put("first_name", new Object[] { "darui.wu", "data.iterator" })`表示插入2条数据时，第一条数据的FIRST\_NAME字段的值为"darui.wu"，第二条数据的FIRST\_NAME字段的值为"data.iterator"。

事实上，jTester框架在内部是将这个DataMap分解成2个DataMap来处理，上面的代码片段等效于下面的写法。

```
db.table("tdd_user").clean().insert(new DataMap() {
    {
        this.put("id", 1);
        this.put("first_name", "darui.wu");
    }
});

db.table("tdd_user").insert(new DataMap() {
    {
        this.put("id", 2);
        this.put("first_name", "data.iterator");
    }
});
```

或下面的写法：

```
db.table("tdd_user").clean().insert(new DataMap() {
    {
        this.put("id", 1);
        this.put("first_name", "darui.wu");
    }
}, new DataMap() {
    {
        this.put("id", 2);
        this.put("first_name", "data.iterator");
    }
});
```

## 注意

如果要插入n条数据，可以为每个字段构造一个n维的数组，但如果数组维数没有达到n，则实际插入时后面几条数据以数组的最后一个数据为准。如果多于n，则舍弃多余的数据。相对于要对每个字段显式的构造数组，我们可以用更快捷的书写方式：

#### 例 4.5. 往数据库中插入多条数据(示例一)

```
db.table("tdd_user").clean().insert(2, new DataMap() {
{
    this.put("id", 1, 2);
    this.put("first_name", "darui.wu", "data.iterator");
}
});
```

这种写法的效果和前面的数组方式是等效的。这时候，put(...) API的第一个参数是字段名称，后面的参数分别对应是要插入数据的第 n 个数据。最后再讲解一个例子结束本节内容。

#### 例 4.6. 数据库中插入多条数据(示例二)

```
db.table("tdd_user").clean().insert(3, new DataMap() {
{
    this.put("id", 1, 2, 3);
    this.put("first_name", "wu");
    this.put("last_name", "darui", "davey");
}
});
```

上面的示例代码往数据库表 tdd\_user 中插入3条记录，这3条记录分别是 (1, "wu", "darui"), (2, "wu", "davey"), (3, "wu", "davey")

到现在为止，我们详细介绍了使用DataMap往数据库插入一条或多条数据的用法。那数据结构DataMap还有其他功能没？下一节，将介绍DataMap的数据提供器的功能。

### DataMap的数据提供器功能

上节介绍的插入数据用法在绝大部分情况下已经足够使用了，但在个别的情况下需要制造有关联意义的数据，比如说某个表中有2个字段user\_name 和 email，我们要求构造的数据email是以user\_name加 @163.com 组成。我们可以写成下面这种方式：

#### 例 4.7. 数据自定义生成示例

```
db.table("YOUR_TABLE").insert(3, new DataMap() {
{
    this.put("user_name", "darui.wu", "jobs.he", "Daniel.Hu");
    this.put("email", new DataGenerator() {
        @Override
        public Object generate(int index) {
            return value("user_name") + "@163.com";
        }
    });
}
});
```

上面的示例代码往数据库中插入3条数据： ("darui.wu", "darui.wu@163.com"), ("jobs.he", "jobs.he@163.com"), ("Daniel.Hu", "Daniel.Hu@163.com")。这时候DataMap的key仍然是字段名称，但value变成了数据生成器（DataGenerator）：它按指定规则生成字段email的值。

DataGenerator的定义及说明

```
public abstract class DataGenerator {
/**
 * 生成第n个数据<br>
 * index计数从0开始
 */
public abstract Object generate(int index);

/**
 * 获取已经设置好字段的对应值
```

```

*/
public Object value(String fieldName) {
    // ...
    return this.dataMap.get(fieldName);
}
}

```

DataGenerator是个抽象类，它主要有2个方法：generator(int index) 和 value( fieldName)。其中value(fieldName)是获取第 n 条数据已经设置好的fieldName字段的值，可以在generator中直接使用。

generator(int)是一个抽象方法，参数index是要生成的数据的计数器（index从0开始计数）。比如总共要插入3条数据，那么index就分别是：0,1,2。

在JTester框架中，内置三种常用的数据生成器：

- 自增数据生成器：IncreaseDataGenerator  
使用方式：DataGenerator.increase(start, step),其中from和step是数值类型，start是起始值，step是自增步长。
- 随机数据生成器：RandomDataGenerator  
使用方式：DataGenerator.random(Class),生成对应类型（数值Number，字符串String）的随机值。
- 循环数据生成器：RepeatDataGenerator  
使用方式：DataGenerator.repeat(value1, value2),以value1, value2, ....., value1, value2这样的方式循环返回对应的值。

为了方便解释DataMap中value值的各种不同情况，下面给出一个综合使用各种情况的示例

#### 例 4.8. 插入数据的综合示例

```

db.table("tdd_user").clean().insert(5, new DataMap() {
{
    this.put("id", DataGenerator.increase(100, 1));
    this.put("first_name", "wu");
    this.put("last_name", DataGenerator.random(String.class));
    this.put("post_code", DataGenerator.repeat("310012", "310000"));
    this.put("my_date", new Object[] { new Date(), "2011-09-06" });
}
}).commit();

```

示例往数据库中插入5条数据，其中id字段是自增方式生成的，first\_name字段是固定值"wu"，last\_name是个随机字符串，post\_code是"310012"和"310000"的循环值，my\_date字段的第一个值是客户端时间当前值，后面4个值都是"2011-09-06"

这样，我们往数据库中插入的5条数据分别是(假定当前时间是2011-11-03 16:56:32)：(100, "wu", "随机值1", "310012", "2011-11-03 16:56:32"), (101, "wu", "随机值2", "310000", "2011-09-06"), (102, "wu", "随机值3", "310012", "2011-09-06"), (103, "wu", "随机值4", "310000", "2011-09-06"), (104, "wu", "随机值5", "310012", "2011-09-06")

## 准备数据的一些补充

上面几节已经完整的介绍了如何插入一条或多条数据到数据库中，并且介绍了如何生成自定义的数据。在JTester框架中还提供了其它方式的数据插入，比如已json方式(在框架内部，同样会将json转为DataMap来处理)来插入数据等，在这一节将作一个简单是介绍，供可能需要的同学参考。

#### 例 4.9. 以json的方式插入数据

```

db.table("tdd_user").clean().insert("{\"id\":1, 'first_name': 'wang', 'last_name': 'json'}");

```

上面的示例插入一条数据，共3个字段。

在很多情况下，我们有很多测试方法需要准备的测试数据都是一样的，那么应该怎么处理呢？

- 当前测试类中所有的测试方法都要公用数据准备过程。

可以在测试类中定义一个@BeforeMethod (TestNG)或 @Before (JUnit)的方法，在beforemethod方法中准备数据。

```
public class InsertMethodBeforeMethod extends JTester {
    @BeforeMethod
    public void initTddUserData() {
        db.table("数据库表").clean().insert(new DataMap() {
            {
                this.put("字段一", "字段一的值");
                this.put("字段二", "字段二的值");
                this.put("字段三", "字段三的值");
            }
        });
    }

    @Test
    public void testBussinesse1() {
        // 业务测试一
    }

    @Test
    public void testBussinesse2() {
        // 业务测试二
    }
}
```

- 在当前测试类中很多测试方法要公用数据准备过程，但不是所有的方法都需要。

可以在类中定义一个私有的帮助方法，在帮助方法中准备数据，其它测试方法调用这个帮助方法。

```
public class UseHelperInitDataMethod extends JTester {
    /**
     * 私有的构造数据库中数据的帮助方法
     */
    private void initTddUserData() {
        db.table("数据库表").clean().insert(new DataMap() {
            {
                this.put("字段一", "字段一的值");
                this.put("字段二", "字段二的值");
                this.put("字段三", "字段三的值");
            }
        });
    }

    /**
     * 这个测试方法需要用到数据准备，调用帮助方法
     */
    @Test
    public void testBussinesse1() {
        initTddUserData();
        // 业务测试一
    }

    /**
     * 这个测试方法不需要用到数据准备，不调用帮助方法
     */
    @Test
    public void testBussinesse2() {
        // 业务测试二
    }
}
```

```

* 这个测试方法需要用到数据准备，调用帮助方法
*/
@Test
public void testBussinesse3() {
    initTddUserData();
    // 业务测试二
}
}

```

- 也可以定义一个公用的数据集DataSet子类，在需要的地方调用数据集插入。

```

public class UseCommonDataSet extends JTester {
    /**
     * 这个测试方法需要用到数据准备
     */
    @Test
    public void testBussinesse1() {
        db.table("数据表").insert(new TddUserTableDataSet());
        // 业务测试一
    }

    /**
     * 这个测试方法不需要用到数据准备
     */
    @Test
    public void testBussinesse2() {
        // 业务测试二
    }

    /**
     * 这个测试方法需要用到数据准备
     */
    @Test
    public void testBussinesse3() {
        db.table("数据表").insert(new TddUserTableDataSet());
        // 业务测试二
    }

    static class TddUserTableDataSet extends DataSet {
        public TddUserTableDataSet() {
            data(new DataMap() {
                {
                    this.put("字段1", "字段1的值");
                    this.put("字段2", "字段2的值");
                    this.put("字段3", "字段3的值");
                }
            });
            data(2, new DataMap() {
                {
                    this.put("字段1", "字段1的值", "字段1的值");
                    this.put("字段2", "字段2的值");
                    this.put("字段3", "字段3的值");
                }
            });
            data("{\"字段1':'字段1的值','字段2':'字段2的值','字段3':'字段3的值'}");
        }
    }
}

```

在方式三中，DataSet中准备了4条数据，其中3条以DataMap方式准备的，一条是json形式的。到现在为止，我们已经完整的介绍了如何往数据库中插入数据，从下节开始，我们将介绍如何验证数据库中已经存在的数据。在结束本部分内容前，强调一下准备数据需要关注的要点：



## 注意

- 在任何情况，针对单元测试准备的数据都要做到少而精。如果2条数据可以达到测试的目的，就千万不要准备3条数据。
- 在大部分情况下，针对单元测试准备数据都只需要关注自己感兴趣的字段，无关的字段由框架自己生成插入。
- 单元测试中，每个测试都只准备自己需要的数据，不要把好几个测试方法需要的数据杂糅在一起，这样做即没法很好的反映你的测试意图，也导致后续的测试维护变难。
- 如果有专门的单元测试数据库，就需要在准备数据开始前，全表清空其中所有的数据，然后再准备自己的数据，以避免数据干扰。

## 对指定表进行数据验证操作

前面我们比较完整的介绍了如何往数据库中插入需要的数据以供后面的业务方法使用。在业务方法执行完以后，业务方法有可能往数据库中新增或修改了数据，这时候我们就需要验证数据库中数据状态是否符合我们的期望。

在本章内容的第一节，我们就给了一个简单的例子用于验证数据库中的一条数据，并给出了如果验证错误，框架会给出什么样的提升信息。要验证数据，无非就2种情况，全表验证和带条件查询的验证。在大部分情况下，单元测试都需要一个专门的单元测试数据库，在开始测试之前需要清空表中的数据全新准备自己需要的数据。在这种情况下，因为表中的数据量不多，就可以简单的使用全表验证的方式；反之，如果表中的数据量比较大，就需要使用带条件的查询进行比较。

- 全表查询的比较: `db.table("表名").query()`  
`query()`返回的对象是一个map list对象的断言器，所有针对List的断言都可以在这里使用。
- 带条件查询的比较: `db.table("表名").query("条件")`  
`queryWhere("条件")`返回的对象是一个map list对象的断言器，用法同上。这里的“条件”是SQL语句中where关键字后面的内容。
- 验证表中数据量: `db.table("表名").count()`  
`count()`方法返回的是个数值断言器。

当数据库中有多条数据时的全表验证的示例

### 例 4.10. 验证多条数据的示例(有序比较)

```
db.table("tdd_user").query().propertyEqMap(2, new DataMap() {
{
    this.put("id", 1, 2);
    this.put("first_name", "darui.wu", "data.iterator");
    this.put("post_code", "310012");
    this.put("address_id", 0);
    this.put("sarary", 0.0);
}
});
```

上面的例子中验证数据和前面介绍的插入数据是非常类似的，jTester框架把数据从数据库中查询出来转换为Map List对象，map中的key是字段名称，value值是对应的数据对象。例子中期望数据库中总共有2条数据，并且第一条数据是(1, "darui.wu", "310012", 0, 0.0)，第二条数据是(2, "data.iterator", "310012", 0, 0.0)。一目了然，需要强调一点的是：

## 注意

在数据验证中，只比较显式在DataMap中指定的数据，没有指定的数据不会被比较。

在上面的比较中，是数据的有序比较，也就是第一条数据和第二条数据顺序不能对调，但在大部分情况下，只需要比较数据库中有这个数据就可以了，不大会关心数据的顺序关系。这时就需要用到List比较模式EqMode。

```
public enum EqMode {
```

```

/**
 * 当期望值是个默认值（对象：null，数字：0）时忽略比较
 */
IGNORE_DEFAULTS,

/**
 * 忽略日期类型的比较
 */
IGNORE_DATES,

/**
 * 比较时忽略数值的顺序
 */
IGNORE_ORDER
}

```

这里我们要忽略数据的顺序关系，就需要用到EqMode.IGNORE\_ORDER，完整的比较片段代码如下：

#### 例 4.11. 验证多条数据的示例(无序比较)

```

db.table("tdd_user").query().propertyEqMap(2, new DataMap() {
{
    this.put("id", 1, 2);
    this.put("first_name", "darui.wu", "data.iterator");
    this.put("post_code", "310012");
    this.put("address_id", 0);
    this.put("sarary", 0.0);
}
}, EqMode.IGNORE_ORDER);

```

如果需要带上查询条件，就可以写成下面这种方式。

#### 例 4.12. 带条件的数据查询验证

```

db.table("tdd_user").queryWhere("post_code=310012").propertyEqMap(2, new DataMap() {
{
    this.put("id", 1, 2);
    this.put("first_name", "darui.wu", "data.iterator");
    this.put("address_id", 0);
    this.put("sarary", 0.0);
}
}, EqMode.IGNORE_ORDER);

```

在queryWhere中带上条件 post\_code=310012，其它的和全表比较方式是一摸一样的。

在个别的情况下，我们甚至不需要验证具体的数据内容，只需要验证数据库中的数据量，那么就可以简单的写成下面代码：

```
db.table("tdd_user").count().eq(2);
```

好了，比较数据就这么简单，但有2个注意点需要强调一下：

### 注意

- DataMap中的key是字段名称，mysql的是小写的，orcle的是大写的，所以验证数据时需要注意一下key值的大小写问题。  
插入数据时无需关注大小写问题。
- 比较value值时是比较字段对应的java对象，所以DataMap中的value值必须是完整的java对象，而不能用String来表示。  
插入数据时可以用string来表示任何数据对象。
- Date类型的比较是个例外，它允许是Date类型或对应的String字符串： yyyy-MM-dd HH:mm:ss, yyyy-MM-dd或HH:mm:ss。比如下面这样：

```
db.table("tdd_user").query().propertyEqMap(2, new DataMap() {
    {
        this.put("my_date", "2011-09-10", "2011-09-12");
    }
}, EqMode.IGNORE_ORDER);
```

## 其他数据库操作

在前面的章节中介绍了如何利用DataMap结构往数据库中插入数据和验证数据。还可以辅助性的使用json格式或DataSet方式。但除了这些经过jTester封装过的数据结构外，我们还可以直接使用sql语句的形式来进行数据库的操作。下面，我就以简单代码的形式演示jTester框架中剩余的数据库操作。

下面给出各种执行sql语句的实例：

### 例 4.13. 执行SQL语句示例一(单条sql语句)

```
@Test
public void testExecute() {
    db.execute("insert tdd_user(id, first_name) values(1, \"darui.wu\")").commit();
}
```

### 例 4.14. 执行SQL语句示例二（多条sql语句）

```
@Test
public void testExecute() {
    db.execute("insert tdd_user(id, first_name) values(1, \"darui.wu\")",
        "insert tdd_user(id, first_name) values(2, \"cody.zhang\")",
        "insert tdd_user(id, first_name) values(3, \"jobs.he\")")
        .commit();
}
```

### 例 4.15. 使用SqlSet结构批量执行sql语句

```
@Test
public void testExecute_UseSqlSet() {
    db.execute(new SqlSet() {
        {
            sql("insert tdd_user(id, first_name) values(1, \"darui.wu\")");
            sql("insert tdd_user(id, first_name) values(2, \"jobs.he\")");
        }
    }).commit();
}
```

### 例 4.16. 执行SQL文件

```
@Test
public void testExecute_FromFile() {
    final String file = System.getProperty("user.dir")
        + "/src/main/resources/org/jtester/module/database/dbop/sql-demo.sql";
    db.cleanTable("tdd_user").execute(new File(file));
}
```

sql-demo.sql文件内容是个标准的SQL文件

```
delete from tdd_user;

insert into tdd_user(id,first_name,last_name)
values(100,"wu","darui");

insert into tdd_user(id,first_name,last_name)
```

```
values(101,"he","jobs");
commit;
```

执行复杂的多表查询，返回数据集断言

```
@Test
public void tetsComplexQuery() {
    db.query("select A.*,B.* from A,B where A.id=B.custome_id").propertyEqMap(2, new
    DataMap() {
        {
            this.put("A.id", 1, 2);
            this.put("A.name", "darui.wu", "jobs.he");
            this.put("B.address", "杭州", "广州");
        }
    }, EqMode.IGNORE_ORDER);
}
```

## 注意

多表的复杂查询，必须在IDBOperator下直接执行，即db.query(...)，不能使用db.table("xxx")...这种指定表的形式来实现。后面的断言形式就一样了,至于为什么后面DataMap中的key值是A.name等形式，是因为你select语句中展现出来的字段名称是这种形式的。如果你select的时候指定了别名，那么后面的key就要使用别名。

```
@Test
public void tetsComplexQuery() {
    db.query("select A.id as id, A.name as name,B.address as address from A,B where
    A.id=B.custome_id").propertyEqMap(2, new DataMap() {
        {
            this.put("id", 1, 2);
            this.put("name", "darui.wu", "jobs.he");
            this.put("address", "杭州", "广州");
        }
    }, EqMode.IGNORE_ORDER);
}
```

## 多数据库测试

多数情况下，应用程序中只需要一个数据源就可以了。但也有不少时候，我们会涉及到多个数据库的操作。我们前面所有操作都是基于在jtester.properties文件中配置默认数据源的,默认数据源配置示例如下：

```
database.type=mysql
database.url=jdbc:mysql://localhost/presentationtd?characterEncoding=UTF8
database.userName=root
database.password=password
database.schemaNames=presentationtd
database.driverClassName=com.mysql.jdbc.Driver
```

如果需要第二个数据源，就必须追加配置。比如我们还有一个数据源eve，那么我们就需要以eve为前缀，按照上面的配置全新配置一个数据源。我们必须数据库配置项前面都要加"eve."作为前缀识别，配置示例格式如下：

### 例 4.17. 多数据库时的jtester.properties文件配置

```
eve.database.type=oracle
eve.database.url=jdbc:oracle:thin:@10.20.14.45:1521:crmp?
args[applicationEncoding=UTF-8,databaseEncoding=UTF-8]
eve.database.userName=eve_test
eve.database.password=xxxx
eve.database.driverClassName=com.alibaba.china.jdbc.SimpleDriver
```

配置完毕，我们就可以在程序调用 `db.useDB("eve")` 来切换数据源，简单示例：

#### 例 4.18. 多数据库的切换示例一

```
@Test
public void testMultipleDB() {
    db.useDB("eve").table("YOURTABLE").insert(5, new DataMap() {
        {
            // 要插入的数据
        }
    });
    // 执行业务方法
    db.table("YOURTABLE").query().propertyEqMap(3, new DataMap() {
        {
            // 你要验证的数据
        }
    });
}
```

在任何测试方法中，jTester框架都会将当前可以被使用的数据源置为默认数据源，如果你在测试中要使用第二个数据源，必须显式的使用 `db.useDB("数据源前缀")` 来切换数据源，在切换完毕，当前可用数据源就变成了你指定的数据源。所以后面的任何`db....` 的操作默认都切换到了指定的数据源下。

### 注意

即使你在前一个测试中将数据源切换到指定的数据源，在这个测试方法结束后，数据源又回到了默认的状态。因此如果你下一个测试方法如果要用指定的数据源，必须使用`db.useDB("数据源前缀")`重新切换一次。

如果你在一个测试中既用到了指定的数据源，又用到了默认数据源。那么你必须使用下面2个api来回切换数据源：`db.useDB("指定数据源")`和 `db.useDefaultDB()`。

下面给一个稍微复杂一点的示例：

#### 例 4.19. 多数据库的切换示例二

```
@Test
public void testMultipleDB2() {
    db.useDB("eve").table("YOURTABLE").insert(5, new DataMap() {
        {
            // 在指定数据源要插入的数据
        }
    });
    db.useDefaultDB().table("YOURTABLE2").insert(2, new DataMap() {
        {
            // 在默认数据源要插入的数据
        }
    });
    // 执行业务方法
    db.useDB("eve").table("YOURTABLE").query().propertyEqMap(3, new DataMap() {
        {
            // 在指定数据源你要验证的数据
        }
    });
    db.useDefaultDB().table("YOURTABLE2").query().propertyEqMap(1, new DataMap() {
        {
            // 在默认数据源你要验证的数据
        }
    });
}
```

好了，2个数据源的测试也是很简单的，如果你有更多的数据源，那你只需要定义相应的数据源前缀，如法炮制就可以了。

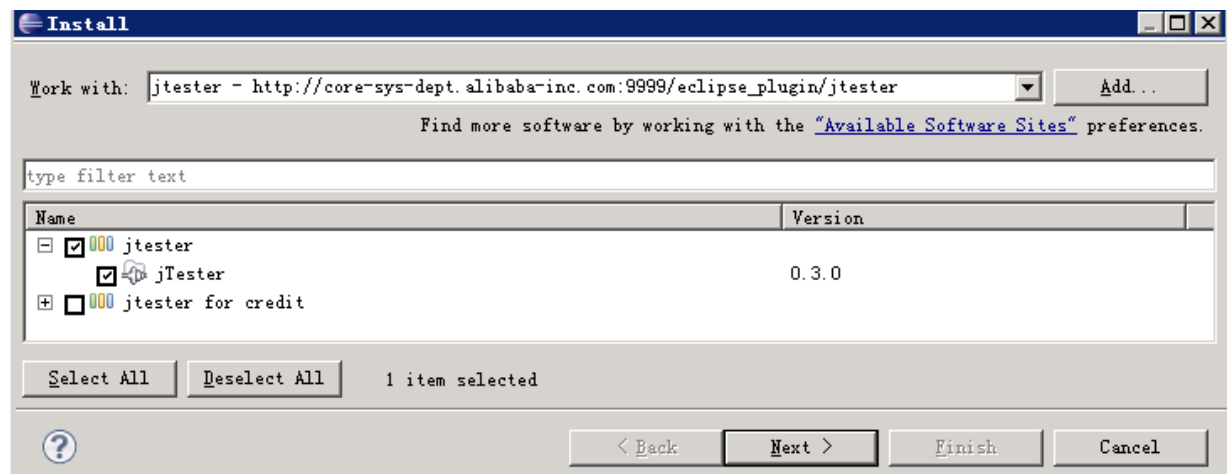
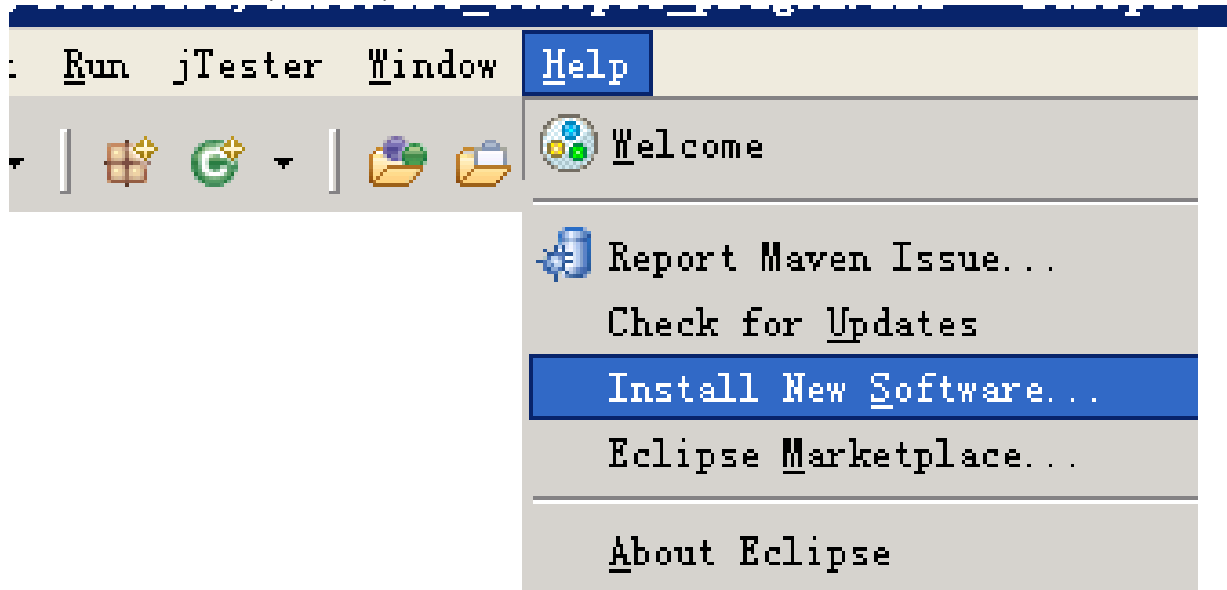
## 注意

在存在多个数据源的状态下，只有默认数据源才有测试方法级别的事务管理，其它的数据源的在测试方法级别都是无事务的（业务方法中的事务由业务代码自己管理，jTester框架不干涉），所以对其它数据源的所有操作都是即时提交的，在测试结束后是没有办法回滚。

## 使用eclipse插件

eclipse（只支持3.5及以上版本） update site: [http://core-sys-dept.alibaba-inc.com:9999/eclipse\\_plugin/jtester](http://core-sys-dept.alibaba-inc.com:9999/eclipse_plugin/jtester)

更新方式，在eclipse的Help -> Install New Software 菜单中



新增一个update site，并且在列出的版本中选择最新的jtester plugin版本(目前是0.3版本)，列表中jtester是标准版本，for credit是中文站诚信发展部定制的版本。

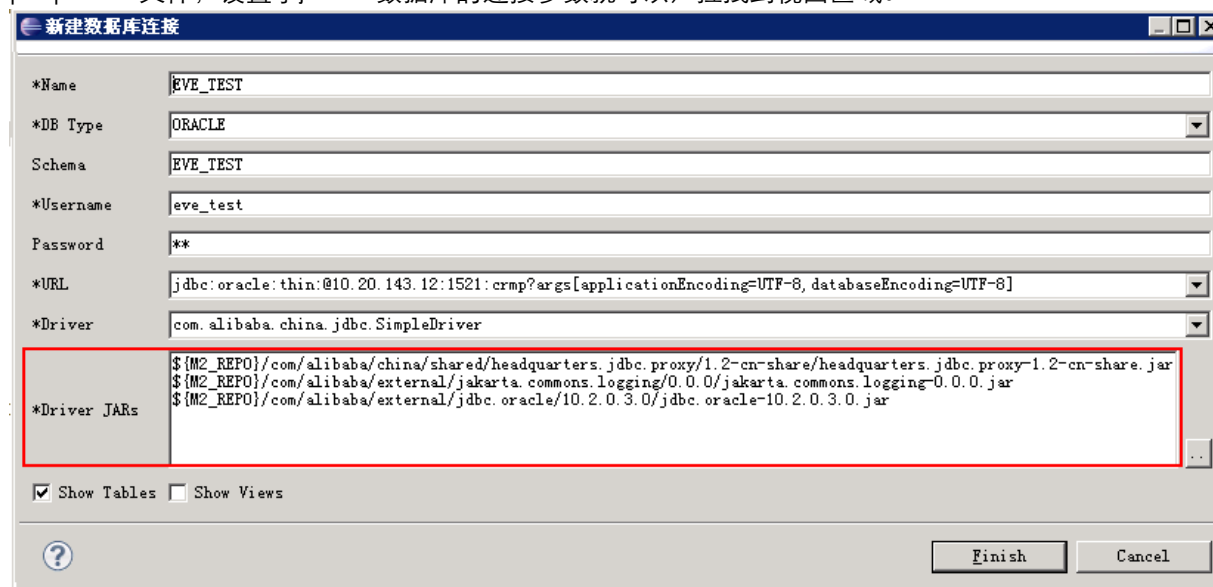
插件安装好以后，在Window->Preference菜单中会有一个jTester菜单：



这些子菜单的作用下面我们会一一讲解，首先我们先打开数据库连接视图 Open Database View



数据库连接视图的右键菜单“新建数据库连接”可以打开一个数据库连接对象对话框，但jTester插件提供了一个更方便的建立连接的方式。你可以把jtester.properties（不限于这个文件名，只要是properties文件，设置了jtester数据库的连接参数就可以）拉拽到视图区域。



连接视图中的大部分选项我们在jtester.properties配置的章节中已经作了说明，除了Name和Driver Jars这两项。

对话框中的Name只是给显示的连接起个名字，可以随便乱起。



Driver Jars是插件建立数据库连接需要用到的jar包，它可能包括对应的数据库驱动jar包，一些logging的jar包（如果驱动用到的话）。为了避免每次建立连接的时候都要指定一下这些jar的路径，你也可以在jtester.properties文件中事先声明一下，比如我们用到的Oracle驱动：

```
database.type=oracle
database.schemaNames=eve_test
database.url=jdbc:oracle:thin:@10.20.143.12:1521:crmp?
args[applicationEncoding=UTF-8,databaseEncoding=UTF-8]
database.userName=eve_test
database.password=ca
database.dialect=oracle
database.driverClassName=com.alibaba.china.jdbc.SimpleDriver

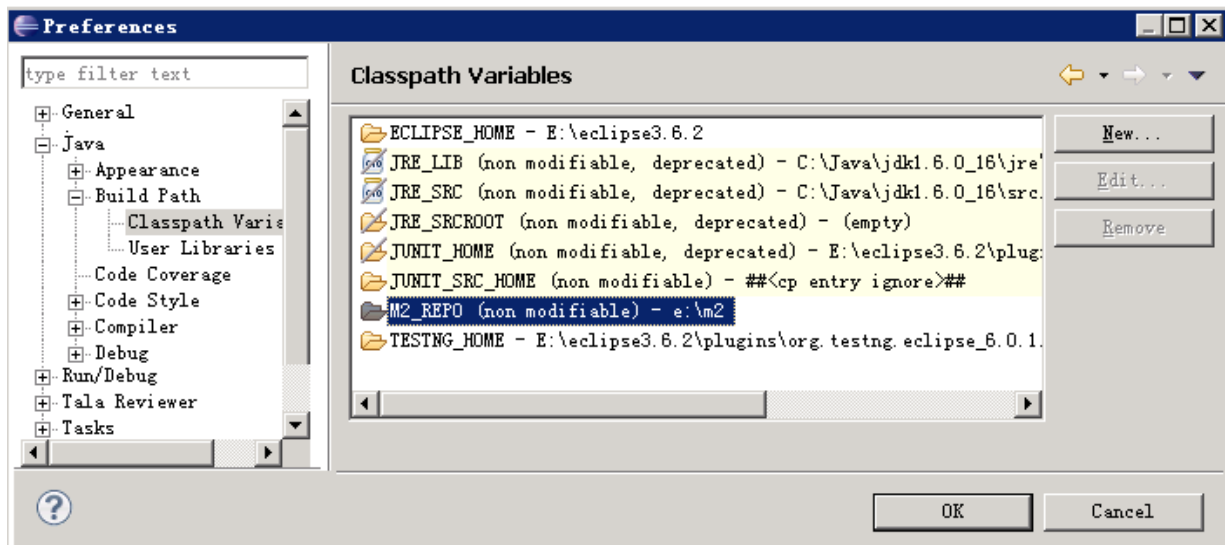
database.driverJar=${M2_REPO}/com/alibaba/china/shared/headquarters.jdbc.proxy/1.2-cn-share/headquarters.jdbc.proxy-1.2-cn-share.jar;\
${M2_REPO}/com/alibaba/external/jakarta.commons.logging/0.0.0/\
jakarta.commons.logging-0.0.0.jar;\
${M2_REPO}/com/alibaba/external/jdbc.oracle/10.2.0.3.0/jdbc.oracle-10.2.0.3.0.jar
```

database.driverJar配置项中的变量\${M2\_REPO}是你本地的maven仓库地址，它的值一般会出现在eclipse的classpath variable配置项中。

将properties文件拖拽到连接视图区域，弹出对话框时，开发还可以根据自己要连接的数据库修改对应的属性：比如properties文件中配置的是单元测试数据库的属性，可以更改为开发库的连接属性。那样就可以从开发库拷贝你需要的数据作为测试数据使用。

## 注意

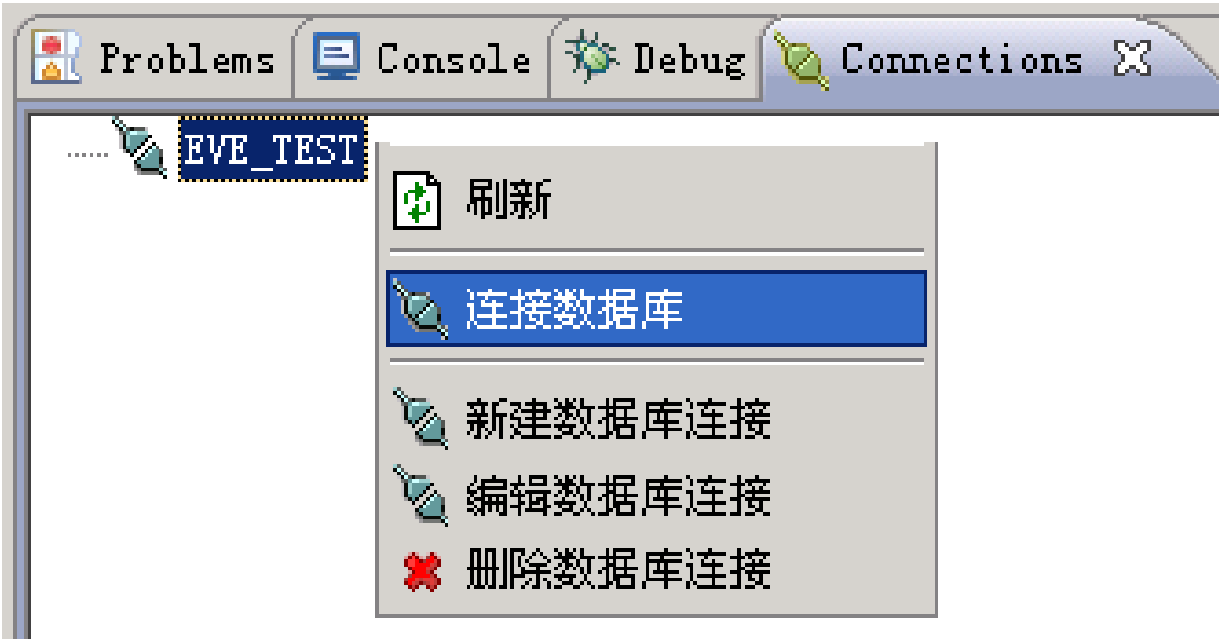
从开发数据库拷贝数据的时候，你有选择的拷贝数据，不要眉毛胡子一起抓，拷贝大量无关紧要的数据，这样做反映不出你数据的真实意图，也导致了后续测试维护的难度。



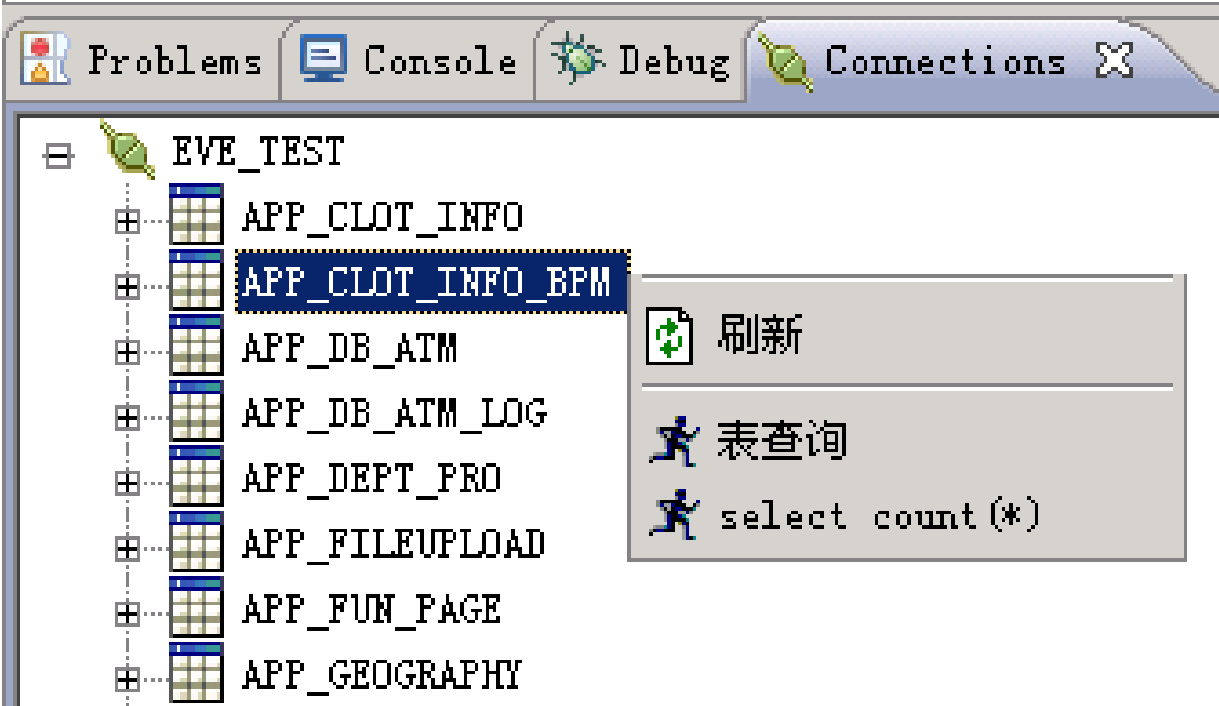
如果没有出现这个选项（或者选项中的值不是你设定的本地maven仓库路径），那就需要你手工添加一下。

在数据库连接对话框中配置好参数以后，点击确定，我们就建立好一条数据库连接了。





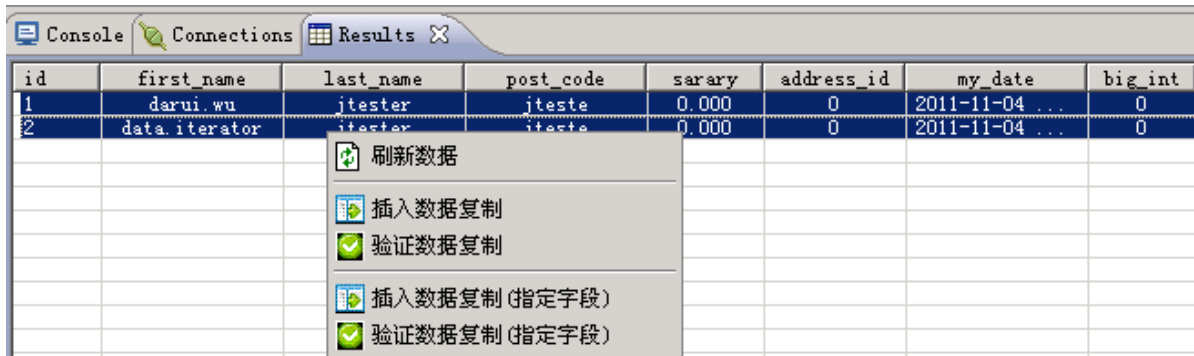
其右键菜单主要有列4项：“新建数据库连接”，“编辑数据库连接”，“删除数据库连接” 这3个选项就不细说了， 反正看菜单名字就明白是什么回事。菜单“连接数据库” 是根据刚才配置的连接信息，建立一个真正的数据库连接。展开连接节点，其列出的子节点是数据库对应的表名称：



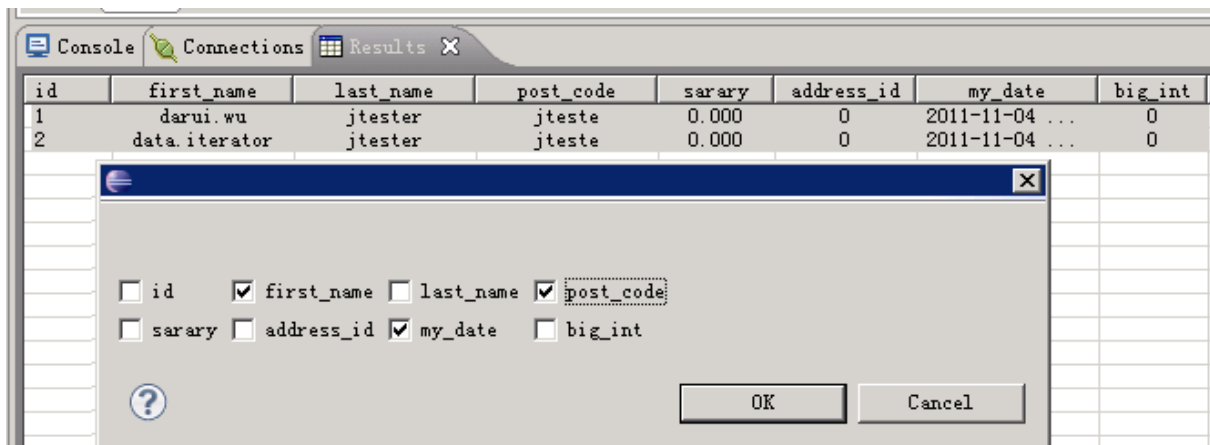
选定一张表，我们执行“表查询”右键菜单命令，插件会将对应的表中的数据查询出来，显示到Data Results的视图中：

Results							
id	first_name	last_name	post_code	salary	address_id	my_date	big_int
1	darui.wu	jtester	jteste	0.000	0	2011-11-04 ...	0
2	data.iterator	jtester	jteste	0.000	0	2011-11-04 ...	0

选定特定的行数据，右键菜单有如下选择：

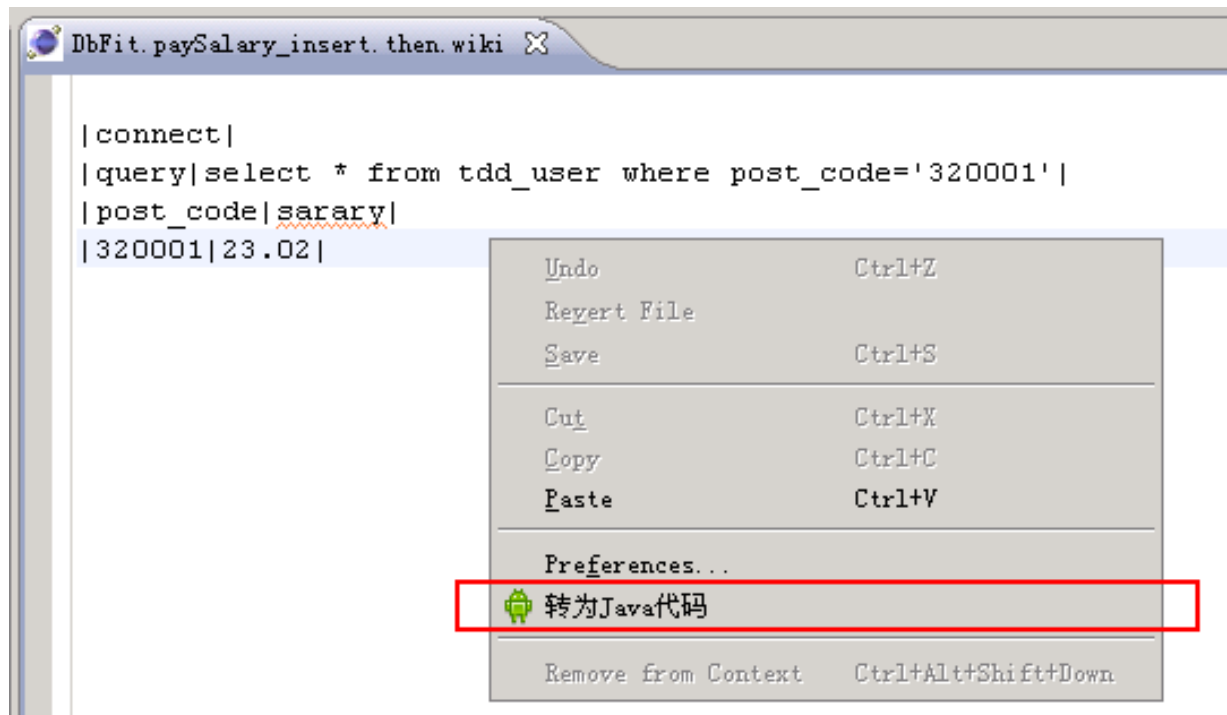


- 插入数据复制  
将指定行的数据复制为 `db.table("表名").clean().insert(...)` 形式的java代码。
- 验证数据复制  
将指定行的数据复制为 `db.table("表名").query().reflectEqMap(...)` 形式的java代码。
- 插入数据复制（指定字段）  
同“插入数据复制”，但是会弹出一个对话框让你选择需要插入哪些字段的值。
- 验证数据复制（指定字段）  
同“验证数据复制”，但是会弹出一个对话框让你选择需要验证哪些字段的值。



数据复制好以后，我们就可以粘贴到在对应的测试代码中。

jTester插件除了提供直接复制拷贝数据外，也提供将原先的dbfit wiki数据转换为DataMap形式的java代码功能。打开wiki文件，右键菜单“”。



转换后，对应的java代码就会保存到系统的剪贴板中，开发只要把代码从剪贴板中粘贴出来即可。

## 第 5 章 在测试中集成Spring

jTester框架对spring做了很多封装，方便开发使用，其特性有：

- 在测试类上标注@SpringApplicationContext加载spring文件,初始化Spring容器。
- 通过在测试类上标注@AutoBeanInject，框架自动查找和注册对应的测试类会用到的实现类的bean定义。减少spring配置的难度，增加单元测试的可维护性。
- 通过在测试中定义@SpringBeanByName（或@SpringBeanByType)注解的字段，将spring容器中的bean实例注入到测试类中。
- 通过@SpringBeanFrom注解，可以精确指定spring bean的值（这个bean可以是直接new出来的，或@Mocked的字段，或运行时指定的值）。

### 加载spring容器

要在单元测试中集成spring，只需要在测试类前面加@SpringApplicationContext注解，其属性值中指定要加载的spring文件，jTester框架在执行改测试类前，会到classpath下查找并加载配置文件，然后初始化好spring容器以供后续使用，示例代码如下：

#### 例 5.1. 加载Spring容器示例(使用classpath路径)

```
@SpringApplicationContext({"spring/data-source.xml", "spring/biz-service.xml" })
public class SpringUsageDemo extends JTester {
    //注入spring bean

    public void test(){
        //具体测试
    }
}
```

如上定义，jtester框架会给测试类SpringUsageDemo创建一个ApplicationContext，并且会到classpath下面去寻找"spring/data-source.xml" 和"spring/biz-service.xml" 这2个文件，并把它们加载进来。

#### 注意

@SpringApplicationContext注解是支持测试类的继承关系的，如果当前测试类没有定义@SpringApplication，Jtester就会查看测试类的superclass是否定义了，一直查到JAVA基类为止。如果superclass和测试子类都定义了@SpringApplicationContext，那么jTester框架使用测试子类中定义的配置文件，而不会使用父类中定义的文件，换言之：  
@SpringApplicationContext中定义的配置文件在父子测试类中是覆盖关系，不是merge关系。

@SpringApplicationContext默认是查找工程的classpath中定义的文件，如果你要使用文件地址来查找配置文件，那么你可以用加"file:相对文件路径"的方式定义，示例如下：

#### 例 5.2. 加载Spring容器示例(使用文件路径)

```
@SpringApplicationContext({"file:./src/main/resources/spring/data-source.xml", "file:./src/main/resources/spring/biz-service.xml" })
public class SpringUsageDemo_FilePath extends JTester {
    //注入spring bean

    @Test
    public void test(){
        //具体测试
    }
}
```

如果你的spring文件是定义在兄弟工程中，就需要使用 "file:../兄弟工程目录/spring文件相对于项目跟目录的路径/spring文件名称.xml" 这样指定。

## 注意

- 如果可以，请使用classpath路径方式定义，尽量不要使用文件路径方式定义。
- 这里的兄弟工程目录指的是disk上的路径，不是eclipse中的project名称
- 使用文件路径的时候，请使用相对路径的方式，不要使用绝对路径，因为测试程序不是你一个人会执行，其他人会checkout下来执行，服务器端也可能会定时执行。

定义好spring容器后，我们就可以在测试代码中使用了，但如何把容器中的bean暴露给测试代码呢？这就需要用到@SpringBeanByName和@SpringBeanByType这两位兄弟。

### 例 5.3. 使用@SpringBeanByName把容器中的bean注入到代码中

```
@SpringApplicationContext({"spring/data-source.xml", "spring/biz-service.xml" })
public class SpringUsageDemo extends JTester {
    /**
     * 没有显式指定bean的名称，按照字段的名称寻找spring bean注入
     */
    @SpringBeanByName
    CustomerService customerService;

    /**
     * 显式的指定bean的名称
     */
    @SpringBeanByName("customerService")
    CustomerService customerServiceByName;

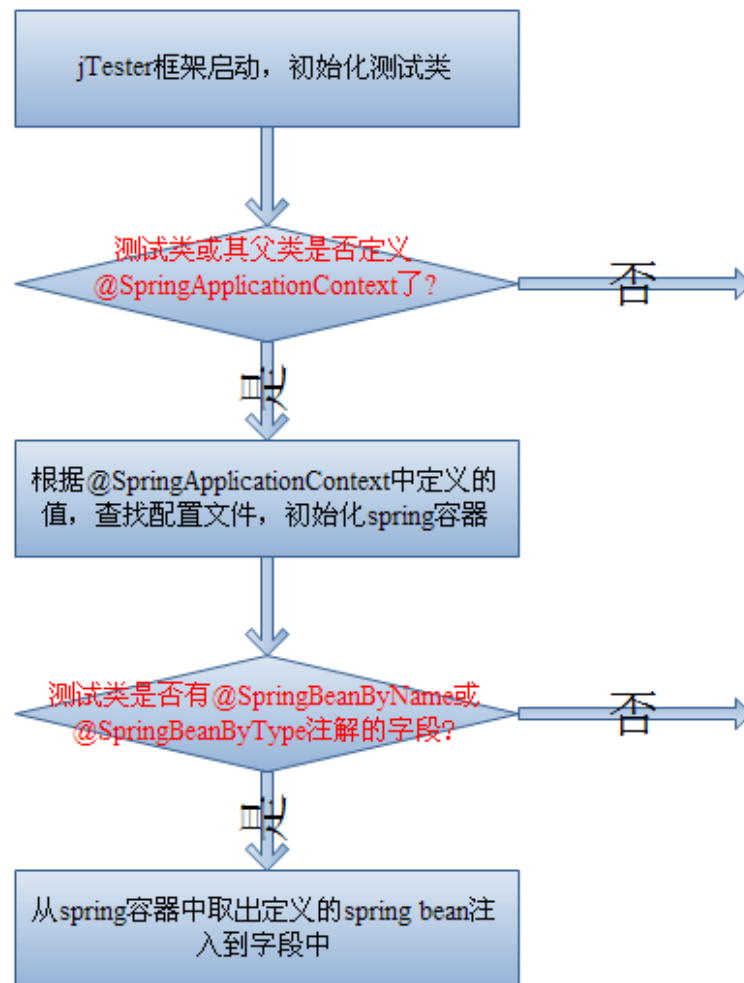
    /**
     * 按类型方式注入
     */
    @SpringBeanByType
    CustomerService customerServiceByType;

    @Test
    public void test(){
        //具体测试代码
    }
}
```

顾名思义，@SpringBeanByName是按照spring bean的name (id) 从已经初始化好的SpringApplicationContext中查找到spring bean，然后把这个bean的值赋给customerService。如果@SpringBeanByName没有设置value值，那么jtester默认会按照改字段的名称来查找spring bean，如果显式定义了value值，那么就按照value值从spring容器中查找name为value的bean。上例中的@SpringBeanByName("customerService")就会到spring容器中查找"customerService"这个bean，所以对于上面的例子而言，字段CustomerService customerServiceByName和CustomerService customerService的值是同一个bean实例。

@SpringBeanByType就是按照类型方式从spring容器中查找bean实例，如果该类型的bean有且只有一个，那么这个bean就会被赋值给定义的字段；如果没有或者多于一个，jtester就会抛出异常。

上面的几个简单例子演示了怎样加载spring容器和使用spring bean，现在让我们简单介绍一下jtester spring模块的基本工作原理，以加强大家的理解。



### 注意

在测试类中注入spring bean时，只需用在对应的字段前面写`@SpringBeanByName`或`@SpringBeanByType`，无需set和get方法。

TestNG框架还有可能将set (get) 方法当作测试方法处理，导致测试类初始化出错。

## @AutoBeanInject让框架自动查找和注册需要的bean

上节讲了如何初始化spring容器和注入spring bean到测试实例中，这些功能已经足够完成一个spring相关的测试。但如果有上百上千个测试类，如果维护spring配置文件呢，如果众多的测试类共享一份spring配置文件，将会使spring配置文件变的很庞大，一个人对配置文件的修改会影响到众多的测试，同时测试也会加载很多它根本就不会用到的spring bean，会使测试变得很慢。这么说来，共享spring配置文件并不是一个好主意，那么我为每个测试类单独准备一份配置文件怎么样呢？这样似乎可以解决很多问题，我对配置文件的修改不会影响到其他测试，可以针对该测试只加载自己需要的bean。但相反的问题来了，每个测试类准备一份配置文件，将会使的配置文件有成千上百份，这些文件维护起来就一点也不简单了。

这是个两难的选择，该怎么办了？jTester提供了根据测试类自动测试spring bean的功能可以很好的解决这个问题。首先，我们先给出一份示例，然后再说明jTester如何完成自动注册spring bean的功能。

## 例 5.4. @AutoBeanInject方式加载spring示例

```

@Test(description = "动态注册spring bean演示")
@SpringApplicationContext({ "spring/data-source.xml" })
@AutoBeanInject(maps = { @BeanMap(intf = "**.*", impl = "**.*Impl"),
    @BeanMap(intf = "**.*", impl = "**.impl.*Impl") })
public class BeanDanymicRegisterDemo extends JTester {
    /**
     * 此时的@SpringBeanByName还扮演了另外一重功能，自动注册的bean入口
     */
    @SpringBeanByName
    CustomerService customerService;

    @Test(description = "演示动态注册spring bean功能")
    public void testDanymicRegister() {
        // 具体测试
    }
}

```

上面的例子，@SpringApplicationContext 只加载了一个文件"data-source.xml"，我们先来看看data-source.xml文件的定义内容。

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans .....>
    <context:annotation-config />
    <bean id="dataSource" destroy-
method="close" class="org.apache.commons.dbcp.BasicDataSource" >
        <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url" value="jdbc:mysql://localhost/udb" />
        <property name="username" value="root" />
        <property name="password" value="password" />
    </bean>
    <bean
id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource">
            <ref local="dataSource" />
        </property>
    </bean>
    <tx:advice id="defaultTxAdvice">
        <tx:attributes>
            <tx:method name="*" rollback-for="Exception" />
        </tx:attributes>
    </tx:advice>
    <aop:config>
        <aop:pointcut id="ao_bo"
            expression="execution ....." />
        <aop:advisor pointcut-ref="ao_bo" advice-ref="defaultTxAdvice" />
    </aop:config>
    <bean
id="sqlMapClient" class="org.jtester.module.spring.ibatis.SqlMapClientFactoryBeanEx">
        <property name="dataSource" ref="dataSource" />
        <property name="configLocation" value="spring/sqlmap-config.xml" />
    </bean>
</beans>

```

在data-source.xml文件中，只定义了数据源，以及spring事务管理策略，并没有定义CustomerService这个bean，那么测试类中定义了@SpringBeanByName CustomerService customerService; 这个变量，它的值又是从哪里来的呢？另外我们注意到，测试类前面除了有个@SpringApplicationContext这个注解，还多了@AutoBeanInject注解。这个注解告诉jtester框架：“你在启动spring容器前，帮我注册我可能会用到的spring bean”。

但JTester框架怎么会知道你后面的测试会用到上面bean呢，测试类中带@SpringBeanByName（或@SpringBeanByType）字段肯定是测试要用到的。JTester框架就是从这里入口，根据定义好的规则自动查找和注册@SpringBeanByName字段所对应的bean定义，然后再递归查找和定义这些bean的依赖项。

现在我们明白了，customerService是JTester框架在启动spring容器时自动注册的，那么这个bean的id和class是什么呢？对于bean id，我们的规则跟前面还是一样，如果@SpringBeanByName没有指定value值，那么id的值就是字段的名称，反之则是value值。

Spring bean id确定了，class呢？我们先分为2种情况讲解。

- 字段本身是个可实例化的class，那么class直接取字段field type。
- 字段的type是接口或者抽象类，那JTester就需要根据@AutoBeanInject中定义的@BeanMap规则来查找具体的实现类。

对上面的例子，CustomerService的全称是org.jtester.tutorial.biz.service.CustomerService。我们先根据第一条规则查找实现类org.jtester.tutorial.biz.service.CustomerServiceImpl是否存在，如果存在则终止查找，spring bean的class就是org.jtester.tutorial.biz.service.CustomerServiceImpl；如果不存在，我们根据第二条规则查找org.jtester.tutorial.biz.service.impl.CustomerServiceImpl。如果第二条规则找到了实现，那么就结束查找。如果遍历完所有的规则，都没有发现实现类的class，那JTester就会抛出一个异常，告诉你没有查找到实现类。

我们演示了@AutoBeanInject的示例代码，下面来详细分析@AutoBeanInject的具体查找规则。

## @AutoBeanInject规则详解

上节我们给自己注册bean定义了2组规则：

- @BeanMap(intf = "\*\*.\*", impl = "\*\*.\*Impl")
- @BeanMap(intf = "\*\*.\*", impl = "\*\*.impl.\*Impl")

在这里，@BeanMap表示一组规则，intf值是接口类全称表达式，impl是对应的实现类全称表达式。表达式中的\*\*是表示类的package，单个\*表示类的名称。

intf = "\*\*.\*", impl = "\*\*.\*Impl", 表示实现类的package和接口类的package路径是一样的，但实现类的名称是接口类的名称加Impl。

intf = "\*\*.\*", impl = "\*\*.impl.\*Impl", 表示实现类的package是在接口类的package下的子目录impl中，实现类的名称是接口类的名称加Impl。

intf和impl中的\*\*可以有多个，但必须对应。单个\*只能出现在类名称中，也可以有多个，但2者也必须对应。

再举一个复杂的例子，假设我们有个接口类全称是com.alibaba.service.biz.order.IQueryServiceUser，我们定义了规则@BeanMap(intf = "\*\*.service.\*\*.\*Service\*", impl = "\*\*.concrete.\*\*.\*Impl\*")。这个规则表示，如果接口类的package中包含service这个目录，那么这个package先被service分成3段："com.alibaba"，"service"和"biz.order"。对应的实现package用"concrete"代替"service"再把这3段拼接起来就是实现类的package："com.alibaba.concrete.biz.order"。

类的名称I\*Service\*，表示接口类的名称以I开头，并且包含Service这个单词，同时根据Service这个单词把类名分成4段："I"，"Query"，"Service"和"User"。对应的实现类名称用空字符("")代替"I"，用"Impl"代替"Service"，再把这4段拼起来"QueryImplUser"就是实现类名称。这样我们根据这条规则就得出实现类的全称是："com.alibaba.concrete.biz.order.QueryImplUser"。

上面我们对规则进行了分解，讲解了JTester查找实现类的过程，但如果一个接口类不在intf定义规则集中，那么对应的实现类也就查找不到。如果接口类符合intf定义，并且拼装出来了实现类的全称，还要看classpath中是否真的存在这个实现类：如果存在，则查找成功；如果不存在，则根据下条规则继续查找。

### 注意

上面这个复杂的规则仅仅是YY出来的，在平时的编程中，接口和实现的位置应该规范化：以简单，规则少为佳。



像下面这个例子中（业务系统中真实的例子），接口和实现的位置就五花八门，各显神通，显然不是一个好现象。

```
@AutoBeanInject(maps = { @BeanMap(intf = "**.*", impl = "**.*Impl"),
    @BeanMap(intf = "**.*Bo", impl = "**.*Impl"),
    @BeanMap(intf = "**.*", impl = "**.impl.*Impl"),
    @BeanMap(intf = "**.service.*", impl = "**.impl.*Impl"),
    @BeanMap(intf = "**.service.*Bo", impl = "**.service.impl.*Impl"),
    @BeanMap(intf = "**.service.*Bo", impl = "**.impl.*Impl"),
    @BeanMap(intf = "**.service.*Bo", impl = "**.*Impl") })
```

## Spring Bean依赖项查找规则

上节讲了jTester框架如何根据@AutoBeanInject中定义的规则集，自动的查找和注册测试类显式声明的bean（@SpringBeanByName和@SpringBeanByType字段）。同时jTester框架会进一步查找这些bean所依赖的bean，并且把它们也注册掉，那么jTester是如何查找这些bean的依赖呢，有哪些规则可以遵循？

- 第一步：jTester框架会记录所有显式声明的spring bean（@SpringBeanByName和@SpringBeanByType注解的字段）。
- 第二步：jTester会递归查找已注册bean的所有可能依赖，并且根据类似规则查找这些依赖项的实现，如果找到，则注册依赖项的bean。

jTester查找的依赖项以符合spring规范为准，主要有以下几种方式：

- 已注册bean实现类中声明了依赖项的set方法（只有一个参数的set方法）。
- 已注册bean实现类中声明了@Resource注解字段。
- 已注册bean实现类中声明了@AutoWired注解的字段。

对于第一种情形，jtester框架自动会以byName的方式进行依赖注入，对于第二，第三种情形，必须在spring配置文件中有所声明下列定义，spring容器才会完成依赖注入的功能。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" .....>
  <context:annotation-config />

  <bean id="基础bean" class="....." />
</beans>
```

### 注意

如果根据@AutoBeanInject中列出的规则，没有查找到一个依赖项的实现类的情况下。如果@AutoBeanInject(ignoreNotFound=true)（这个是默认值），则jTester仅仅是打印一条警告，该依赖项的值为null；如果@AutoBeanInject(ignoreNotFound=false)，则jTester会抛出异常：依赖实现没有找到。

## 特殊实现类的bean注册

我们已经详细的介绍了如何使用@SpringApplicationContext来加载基本spring配置文件，通过@AutoBeanInject定义主动注册bean规则让框架帮我们自动管理bean，减少我们的配置工作和维护工作，使的测试代码更加稳健。

定义规则查找对于能规范化的编程来说，已经可以解决大部分实现类查找的问题。但凡事都有例外。对于例外的情况，我们可以追加规则，但那样会列出又臭又长的规则集。我们必须明白，规则太多了，就等于没有规则。

现实编程中，还存在一种情况：一个接口对应着多个实现，我们想指定一种具体的实现，但通过规则先找到的是另外一种实现或者找不到实现，那怎么办呢？我们可以在基础spring配置文件中手工定义这个bean。但如果这个bean并不是通用的（绝大多数测试类都会用到，比如dataSource），那么我们是不推荐这种作法的。对于特殊的实现类，我们推荐使用@SpringBeanByName(clazz=具体实现类class)中显式指定实现类class的方式。

## 例 5.5. 自动注册bean时，显式的指定特殊实现类示例

```

import org.jtester.tutorial.biz.service.irregular.CustomerServiceIrregularImpl;
import org.jtester.tutorial.biz.service.irregular.InvoiceDaoIrregularImpl;

@Test(description = "动态注册spring bean演示,显式指定使用哪个实现类")
@SpringApplicationContext({ "spring/data-source.xml" })
@AutoBeanInject(maps = { @BeanMap(intf = "**.*", impl = "**.*Impl"),
    @BeanMap(intf = "**.*", impl = "**.impl.*Impl") })
public class BeanDanymicRegisterDemo_ExplicitClass extends JTester {
    // 这这里，通过clazz显式指明了customerService的具体实现类是哪个？
    @SpringBeanByName(clazz = CustomerServiceIrregularImpl.class)
    CustomerService customerService;

    @SpringBeanByName(clazz = InvoiceDaoIrregularImpl.class)
    InvoiceDao invoiceDao;

    @Test
    public void test(){
        // 具体测试
    }
}

```

如例子所示，@SpringBeanByName显式指定了customerService和invoiceDao的实现类，那么jTester框架就会根据指定好的class来注册spring bean，而不会去根据规则查找实现。

@SpringBeanByName和@SpringBeanByType中都还有2个属性init和properties的作用不明确

## 自动注册的bean如何实现spring的init-method方法

通过在@SpringBeanByName显式的指定实现类的class，我们已经解决了一个接口多实现类的问题，现在让我们再来考虑一个问题：我们代码中需要用到一个bean，这个bean会在实例化时初始化一些资源。在正常业务代码使用过程中，我们有2种定义方式：

- 通过配置文件，在bean元素上面声明init方法。

```

<bean id="resourceManager" init-method="init"
    scope="singleton" class="com.ali.b2b.crm.base.resource.impl.ResourceManagerImpl" >
    <property name="initQueryPagesize" value="10000"/>
</bean>

```

- 另一种办法是在实现类上声明org.springframework.beans.factory.InitializingBean接口，并实现对应的public void afterPropertiesSet() throws Exception 方法。

```

public class ResourceManagerImpl implements ResourceManager, InitializingBean {
    /**
     * InitializingBean接口类方法，只要实现类实现了这个接口，无需在配置文件中声明
     * Spring容器在实例化ResourceManagerImpl时，会自动调用afterPropertiesSet方法
     */
    public void afterPropertiesSet() throws Exception {
        // 初始化资源代码
    }
}

```

对于自动注册的bean，如果init方法是第二种方式实现的，spring容器在实例化这个bean的时候，会自动执行afterPropertiesSet的方法。如果是第一种方式，我们需要用到@SpringBeanByName和@SpringBeanByType中的一个名为init的属性。在前面的各个章节中我们已经多次提到了这2个Annotation的作用，现在让我们看看这2个Annotation的完整定义。

@SpringBeanByName定义。

```

/**
 * 用在测试类中需要spring bean注入或注册的字段上
 */

```

```

@Target({ FIELD, METHOD })
@Retention(RUNTIME)
@SuppressWarnings("rawtypes")
public @interface SpringBeanByName {
    /**
     * 显式的指定要注入或注册的spring bean name<br>
     * 如果没有指定值，则name为对应字段的名称
     */
    String value() default "";

    /**
     * 显式的指定spring bean主动注册时要实现的实现类class<br>
     * 如果指定了这个值，即使基础spring配置文件中已经有同名的bean定义<br>
     * 也会以指定实现类的定义覆盖配置文件中的定义。
     */
    Class clazz() default SpringBeanByName.class;

    /**
     * spring bean配置中的init-method方法<br>
     */
    String init() default "";

    /**
     * 定义bean的简单属性值，和别名引用bean的情形
     */
    Property[] properties() default {};
}

```

@SpringBeanByType定义

```

public @interface SpringBeanByType {
    /**
     * 显式的指定spring bean主动注册时要实现的实现类class
     */
    Class value() default SpringBeanByType.class;

    /**
     * spring bean配置中的init-method方法<br>
     */
    String init() default "";

    /**
     * 定义bean的简单属性值，和别名引用bean的情形
     * @return
     */
    Property[] properties() default {};
}

```

在@SpringBeanByName和@SpringBeanByType的定义中都有一个init的属性，通过这个属性我们可以显式的指定spring的init方法，等效于在spring文件的bean定义中的init-method属性。

### 例 5.6. 显式指明bean的init-method属性

```

public class BeanDanymicRegisterDemo extends JTester {
    // 这这里，通过clazz显式指明了customerService的具体实现类是哪个？
    @SpringBeanByName(init="initQueryPagesize")
    ResourceManager resourceManager;

    @Test
    public void test(){
        // 具体测试
    }
}

```

对于这种类型bean，如果我们想在测试中个性化的控制资源初始化，以期达到望目标，有没有办法呢？我们可以在测试类中继承要个性化初始化的bean实现类，结合@SpringBeanByName显式指定实现类是我们测试中的扩张类的办法来达到目标。

### 例 5.7. 显式指明bean的init-method属性(使用@SpringBeanByName init属性)

```
public class BizTrendsSendContentTest extends JTester {
    @SpringBeanByName(claz = ResourceManagerImplEx.class, init="initResource")
    private ResourceManager resourceManager;

    @Test
    public void test(){
        //具体测试
    }

    public static class ResourceManagerImplEx extends ResourceManagerImpl {
        public void initResource() throws Exception {
            Map<String, String> properties = new HashMap<String, String>();
            properties.put("file.resource.loader.path", System.getProperty("user.dir") + "/deploy/
templates/vm");
            properties.put("input.encoding", "UTF-8");
            properties.put("output.encoding", "UTF-8");
            this.setProperties(properties);
        }
    }
}
```

### 例 5.8. 显式指明bean的init-method属性(使用@SpringInitMethod注解)

```
public class BizTrendsSendContentTest extends JTester {
    @SpringBeanByName(claz = ResourceManagerImplEx.class)
    private ResourceManager resourceManager;

    @Test
    public void test(){
        //具体测试
    }

    public static class ResourceManagerImplEx extends ResourceManagerImpl {
        @SpringInitMethod
        public void initResource() throws Exception {
            Map<String, String> properties = new HashMap<String, String>();
            properties.put("file.resource.loader.path", System.getProperty("user.dir") + "/deploy/
templates/vm");
            properties.put("input.encoding", "UTF-8");
            properties.put("output.encoding", "UTF-8");
            this.setProperties(properties);
        }
    }
}
```

在上面的测试类中，ResourceManagerImplEx继承了ResourceManagerImpl，并且在初始化根据自己的实际需要初始化了资源。在这里，我们引入了一个新的注解@SpringInitMethod，这个注解告诉JTester框架，这个方法是spring bean的init-method方法，JTester在注册spring bean时就会设置init-method的值。这样spring框架在实例完bean之后会调用initmethod方法进行bean的状态初始化。

还有一种方式，我们也可以直接实现org.springframework.beans.factory.InitializingBean接口，这样spring容器也会自动的做bean状态的初始化工作。

## 例 5.9. 显式指明bean的init-method属性(使用@SpringInitMethod注解)

```

public class BizTrendsSendContentTest extends JTester {
    @SpringBeanByName(claz = ResourceManagerImplEx.class)
    private ResourceManager resourceManager;

    @Test
    public void test(){
        //具体测试
    }

    public static class ResourceManagerImplEx extends ResourceManagerImpl implements
    InitializingBean{
        private void initResource() throws Exception {
            Map<String, String> properties = new HashMap<String, String>();
            properties.put("file.resource.loader.path", System.getProperty("user.dir") + "/deploy/
templates/vm");
            properties.put("input.encoding", "UTF-8");
            properties.put("output.encoding", "UTF-8");
            this.setProperties(properties);
        }

        /**
         * InitializingBean接口类方法，只要实现类实现了这个接口，无需在配置文件中声明
         * Spring容器在实例化ResourceManagerImpl时，会自动调用afterPropertiesSet方法
         */
        @Override
        public void afterPropertiesSet() throws Exception {
            initResource();
        }
    }
}

```

在上面的例子中，我们通过afterPropertiesSet方法个性化的设置了ResourceManagerImplEx这个类的资源文件，以达到控制测试环境的目的。但还存在一种情况，我们需要在spring容器启动前初始化资源。打个比方，我们有个实现类，它在初始化函数中读取或初始化了一下资源，我们希望把这个实现类的构造函数mock掉，加载测试期望的资源。对于这个案例，如果容器已经加载好了，再去定义初始化函数的mock，显然是无法达到目的。这样我们必须想办法在JTester框架初始化spring容器前把对应的实现类构造函数mock掉。在测试类中定义一个@SpringInitMethod注解的方法可以达到这个目的。

## 例 5.10. 在spring容器初始化前mock示例

```

public class BizTrendsSendContentTest extends JTester {
    @SpringBeanByName(clazz = ResourceManagerImpl.class)
    private ResourceManager resourceManager;

    @Test
    public void test(){
        //具体测试
    }
    /**
     *这个方法会在jTester框架初始化Spring容器前被调用
     */
    @SpringInitMethod
    protected void mockResourceManagerConstroctor() {
        new Mock<ResourceManagerImpl>(){
            /**
             * $init 在jmockit中是构造函数名称
             */
            @Mock
            public void $init() {
                Map<String, String> properties = new HashMap<String, String>();
                properties.put("file.resource.loader.path", System.getProperty("user.dir") + "/deploy/
templates/vm");
                properties.put("input.encoding", "UTF-8");
                properties.put("output.encoding", "UTF-8");
                this.setProperties(properties);
            }
        };
    }
}

```

对应上面的代码例子，jTester在启动spring容器前，会先执行mockResourceManagerConstroctor方法。这个方法定义了ResourceManagerImpl实现类的构造函数的mock实现方式（具体的mock语法请参照mock章节的讲解）。

## 注意

为了避免TestNg把mockResourceManagerConstroctor当作一个测试方法，mockResourceManagerConstroctor必须被定义成非public的类型。

## 声明bean的简单属性

在一些spring配置文件中，除了指定了id，class等属性外，可能还有properties属性，里面可以是简单的值，也可以是List、Map等对象，还可能是另外一个spring bean。

比如下面这样的spring配置：

```

<bean id="webFileManageBo" class="com.ali.martini.biz.WebFileManageBoImpl">
    <property name="webUrlBase" value="/images/htmlEditor" />
    <property name="baseDir" value="${project.dir}/deploy/htdocs/upload/htmlEdit/htmlEditor" />
</bean>

```

上面的bean定义声明了bean的id，实现类class，同时还声明了实现类中两个属性的初始值。如果通过前面的@SpringBeanByName可以实现声明id和实现类（隐式或显式）的功能，但不能达到初始化属性值的目的。在本章上节中的@SpringBeanByName（@SpringBeanByType）定义中，还有一个属性properties的作用没有介绍。在这个属性中就可以达到初始化实现类中简单属性的目的。

```

/**
 * 用在测试类中需要spring bean注入或注册的字段上
 */
@Target({ FIELD, METHOD })
@Retention(RUNTIME)
public @interface SpringBeanByName {

```



```
// 其它定义

/**
 * 定义bean的简单属性值，和别名引用bean的情形
 */
Property[] properties() default {};
}
```

```
@Retention(RUNTIME)
public @interface Property {
    /**
     * 属性名称
     */
    String name();

    /**
     * 属性值
     */
    String value() default "";

    /**
     * ref bean名称
     */
    String ref() default "";

    /**
     * ref bean的class实现类
     */
    Class clazz() default Property.class;
}
```

Property中的name用来指定实现类中要初始化的属性名称，value值用来给属性赋值。上面spring文件的bean定义，转换为@SpringBeanByName定义就如下例子：

### 例 5.11. 通过@SpringBeanByName初始化实现类属性

```
public class SpringBeanByNameProperty extends JTester {

    @SpringBeanByName(properties = { @Property(name = "webUrlBase", value = "/images/
htmlEditor"),
        @Property(name = "baseDir", value = "${project.dir}/deploy/htdocs/upload/htmlEdit/
htmlEditor") })
    WebFileManageBo webFileManageBo;

    @Test
    public void test() {
        // 具体测试
    }
}
```

实现类的属性值是个简单类型的string，int或其他primitive类型，通过上面的方法定义就可以了。如果属性值其他bean对象，比如下面的spring配置：

```
<bean id="webFileManageBo" class="org.jtester.module.spring.beans.WebFileManageBoImpl">
    <property name="webUrlBase" value="/images/htmlEditor" />
    <property name="baseDir" value="${project.dir}/deploy/htdocs/upload/htmlEdit/htmlEditor" />
    <property name="myResourceManager" ref="resourceManager" />
</bean>
```

属性webUrlBase和baseDir的定义我们上面的例子已经介绍了，属性myResourceManager是个ref对象，它引用的是另外一个bean实例。这样，我们就需要用到Property对象中的ref属性，对应上面的spring配置，我们用@SpringBeanByName自动注册的例子如下：

## 例 5.12. 通过@SpringBeanByName实现引用其它bean对象的功能

```
public class SpringBeanByNameProperty extends JTester {

    @SpringBeanByName(properties = { @Property(name = "webUrlBase", value = "/images/
htmlEditor"),
        @Property(name = "baseDir", value = "${project.dir}/deploy/htdocs/upload/htmlEdit/
htmlEditor"),
        @Property(name = "myResourceManager", ref = "resourceManager") })
    WebFileManageBo webFileManageBo;

    @Test
    public void test() {
        // 具体测试
    }
}
```

这样，webFileManageBo bean对象中的属性myResourceManager值就是一个已定义的bean resourceManager的引用值。如果resourceManager是个未定义的bean对象，我们也可以在这类順便定义掉。

## 例 5.13. 通过@SpringBeanByName实现引用其它bean对象的功能

```
public class SpringBeanByNameProperty extends JTester {

    @SpringBeanByName(properties = { @Property(name = "webUrlBase", value = "/images/
htmlEditor"),
        @Property(name = "baseDir", value = "${project.dir}/deploy/htdocs/upload/htmlEdit/
htmlEditor"),
        @Property(name = "myResourceManager", ref = "resourceManager",
        clazz=MyResourceManagerImpl.class) })
    WebFileManageBo webFileManageBo;

    @Test
    public void test() {
        // 具体测试
    }
}
```

实现类中的属性除了简单值，bean引用，还有可能是复杂对象，比如List和Map：

```
<bean id="webFileManageBo3" class="org.jtester.module.spring.beans.WebFileManageBoImpl">
  <property name="webUrlBase" value="/images/htmlEditor" />
  <property name="baseDir" value="${project.dir}/deploy/htdocs/upload/htmlEdit/htmlEditor" />
  <property name="myResourceManager" ref="resourceManager" />
  <property name="htmlRootDir">
    <map>
      <entry key="plan_htmlRootDir">
        <value>${project.dir}/deploy/htdocs/plan</value>
      </entry>
      <entry key="planTemplate_htmlRootDir">
        <value>${project.dir}/deploy/htdocs/planTemplate</value>
      </entry>
      <entry key="knowledge_htmlRootDir">
        <value>${project.dir}/deploy/htdocs/knowledge</value>
      </entry>
      <entry key="material_htmlRootDir">
        <value>${project.dir}/deploy/htdocs/material</value>
      </entry>
    </map>
  </property>
</bean>
```



```

</map>
</property>
</bean>

```

这时，通过上面的手法去定义，会比较麻烦，我们可以利用@SpringBeanFrom（具体语法下节介绍）定义一个bean，名称为htmlRootDir，让spring容器自动注入。

#### 例 5.14. 通过@SpringBeanFrom初始化实现类中复杂属性的值

```

public class SpringBeanByNameProperty extends JTester {

    @SpringBeanByName(properties = { @Property(name = "webUrlBase", value = "/images/
htmlEditor"),
        @Property(name = "baseDir", value = "${project.dir}/deploy/htdocs/upload/htmlEdit/
htmlEditor"),
        @Property(name = "myResourceManager", ref = "resourceManager",
        clazz=MyResourceManagerImpl.class) })
    WebFileManageBo webFileManageBo;

    @SpringBeanFrom
    Map<String,String> htmlRootDir = new HashMap<String,String>(){
        {
            this.put("plan_htmlRootDir", "${project.dir}/deploy/htdocs/plan");
            this.put("planTemplate_htmlRootDir", "${project.dir}/deploy/htdocs/planTemplate");
            this.put("knowledge_htmlRootDir", "${project.dir}/deploy/htdocs/knowledge");
            this.put("material_htmlRootDir", "${project.dir}/deploy/htdocs/material");
        }
    };

    @Test
    public void test() {
        // 具体测试
    }
}

```

有关spring bean自定义的内容基本讲解完毕，下一节我们将详细介绍一下@SpringBeanFrom的各种用法。

## 使用@SpringBeanFrom DIY你需要的bean

下面介绍的@SpringBeanFrom功能，将让你有自己定制spring bean的感觉。@SpringBeanFrom，顾名思义，表示spring容器中bean的值是来自于@SpringBeanFrom标注字段的值。既然是来自字段的值，那这个值就可以是运行时变化的，它可以是mock变量，也可以是new实例，如果该字段未赋值，就是null。既然是运行时变化的，如果该字段在测试代码当中改变了值，那容器后面取到的值就是改变后的值，而不是原始值。下面先用简单例子演示：

## 例 5.15. @SpringBeanFrom使用示例 (new实例)

```

@SpringApplicationContext({ "spring/data-source.xml" })
@AutoBeanInject(maps = { @BeanMap(intf = "**.*", impl = "**.*Impl"),
@BeanMap(intf = "**.*", impl = "**.impl.*Impl") })
public class SpringBeanFromDemo extends JTester {
    @SpringBeanByName
    ResourceManager resourceManager;

    @SpringBeanByName(claz = CustomerServiceImpl.class)
    CustomerService customerService;

    @SpringBeanFrom
    CustomerDao customerDao = new CustomerDaoImpl() {
        @Override
        public String doNothing() {
            return "manually new dao.";
        }
    };

    public void test(){
        // 实际测试代码
    }
}

```

在上面的例子中，我们做了以下事情：

- 使用了@SpringApplicationContext加载了最基本的spring配置文件。
- 使用@AutoBeanInject定义自动注册时查找实现类的规则集。
- 使用@SpringBeanByName隐式定义bean，id="resourceManager"，实现类有规则集查找。
- 使用@SpringBeanByName显示定义bean，id="customerService"，实现类是CustomerServiceImpl。
- 使用@SpringBeanFrom定义bean，id="customerDao"，它的目前值是new CustomerDaoImpl() {...}。

通过@SpringBeanFrom我们可以随心所欲的创建自己需要的bean，上面例子中的bean是new实例化的，还可以是个mock对象。

## 例 5.16. @SpringBeanFrom使用示例 (mock对象)

```

@SpringApplicationContext({ "spring/data-source.xml" })
@AutoBeanInject(maps = { @BeanMap(intf = "**.*", impl = "**.*Impl"),
@BeanMap(intf = "**.*", impl = "**.impl.*Impl") })
public class SpringBeanFromDemo extends JTester {
    @SpringBeanFrom
    @NonStrict
    @Mocked
    CustomerDao customerDao;

    public void test(){
        new Expectations(){
            {
                //mock行为
            }
        };
        // 实际测试代码
    }
}

```

也可以是个动态赋值对象。

例 5.17. @SpringBeanFrom使用示例（动态赋值）

```
@SpringApplicationContext({ "spring/data-source.xml" })
@AutoBeanInject(maps = { @BeanMap(intf = "**.*", impl = "**.*Impl"),
@BeanMap(intf = "**.*", impl = "**.impl.*Impl") })
public class SpringBeanFromDemo extends JTester {
    @SpringBeanFrom
    ResourceManager resourceManager;

    @BeforeMethod
    public void initMockResourceManager(){
        private Map<String, List<Option> values = new HashMap<String,List<Option>>(){
            {
                this.put("AV_CN_PROVINCE_ABORAD", new ArrayList(){
                    {
                        this.add(new OptionDo("4858","4858"));
                    }
                });
            }
        };
        this.resourceManager = new MockUp<ResourceManager>(){
            @Mock
            public List<Option> getOptionList(String resName){
                return values.get(resName);
            }
        }.getMockInstance();
    }

    public void test(){
        // 实际测试代码
    }
}
```

在上面的例子中，resouceManager的原始值是个null对象，在测试方法中resourceManager被赋值一个接口类的MockUp实例。 这样resourceManager就变为了一个静态mock的对象。也就是说由@SpringBeanFrom定义的bean对象的实际值是可以运行时动态改变的。

Spring模块注解

到现在为止，我们已经完整的介绍了JTester框架中spring功能的使用， 我们回顾一下这章中出现的注解，以及其对应的功能。

注解	具体功能
@SpringApplicationContext	放在测试类前面，用来加载测试中用到的spring配置文件。 可以和@AutoBeanInject配合使用， 只配置最少公约的spring文件， 其余的由JTester框架主动注册加载。
@AutoBeanInject	放在测试类前面，用来告诉JTester框架进行spring bean的自动注册。 可以在@AutoBeanInject中定义若干个规则@BeanMap
@BeanMap	JTester框架自动注册spring bean的规则。
@SpringBeanByName	按名称往spring容器注册当前字段的spring bean， 或从spring容器注入spring bean到当前字段。
@SpringBeanByType	按类型往spring容器注册当前字段的spring bean， 或从spring容器注入spring bean到当前字段。
@SpringInitMethod	有2个作用： <ul style="list-style-type: none"><li>• 用在扩展的bean实现类方法上，是Spring Bean的初始化方法，和@SpringBeanByName/@SpringBeanByType配合使用</li></ul>

## 在测试中集成Spring

---

	<ul style="list-style-type: none"><li>• 用在测试类的方法上，该方法会在spring容器启动前被调用。</li></ul>
@SpringBeanFrom	把测试类中指定字段的值注入到spring容器中。

## 第 6 章 反射调用私有方法或JDK代理的方法

在测试中，我们经常会碰到开发询问，私有方法是否需要测试？怎么测试？诸如此类的问题，私有方法要不要测试是个仁者见仁，智者见智的问题。个人觉得如果你的私有类中有重要的业务逻辑，通过public方法无法测试完整，或过于复杂时，那么私有方法就必须测试。

如何测试？只要你在测试方法中就可以调用到私有方法就可以测试，我们可以通过反射的方式进行访问，例如下面这个示例：

```
@SpringBeanByName
private WebFileManageBo webFileManageBo;

@Test
public void accessPrivateMethod() throws Exception {
    Method method = WebFileManageBoImpl.class.getMethod("sayHello", String.class);
    boolean isAccessible = method.isAccessible();
    method.setAccessible(true);
    try {
        String result = (String) method.invoke(webFileManageBo, "darui.wu");
        // 具体断言
    } finally {
        method.setAccessible(isAccessible);
    }
}
```

在上面的例子中，通过原生的反射方法调用显得异常麻烦，调用一个sayHello方法，使用了5条语句（还不包括try finally语句）。而且还存在一个问题，如果实例webFileManageBo是个Spring JDK代理（Spring AOP处理过，比如事务管理等），那么上面的反射调用将会出现方法无法找到的错误。

在Jtester框架中，对反射访问做了封装，它可以很方便的访问私有方法，也可以访问经过Spring AOP代理过的类的私有方法。在Jtester基类中内置定义了一个变量reflector,通过这个变量可以进行反射调用。

```
public interface IJTester {
    final JTesterReflector reflector = new JTesterReflector();

    //其它内置变量
}
```

我们先看看JTesterReflector类的具体定义：

```
public class JTesterReflector {
    /**
     * 调用目标target中的私有方法。
     */
    public <T> T invoke(Object target, String method, Object... paras);
    /**
     * 调用目标target中的私有方法（在clazz类中定义的）。
     */
    public <T> T invoke(Class clazz, Object target, String method, Object... paras);
    /**
     * 调用类target中的静态方法method
     */
    public <T> T invokeStatic(Class target, String method, Object... paras);
    /**
     * 设置目标对象target变量field的值
     */
    public void setField(Object target, String field, Object value);
    /**
     * 设置目标对象target的变量field（定义在clazz中）的值
     */
    public void setField(Class clazz, Object target, String field, Object value);
    /**

```

```

* 返回target对象中名称为field的字段
*
* @param target 字段所有者
* @param field 字段名称
* @return 字段值
*/
public <T> T getField(Object target, String field);

/**
* 返回target对象中名称为field的字段
*
* @param clazz 定义字段的类（可能是target对象的父类或target自身）
* @param target 字段所有者实例
* @param field 字段值
*/
public <T> T getField(Class clazz, Object target, String field);

/**
* get class's static value<br>
* 获得clazz的静态变量值
*/
public <T> T getStaticField(Class clazz, String field);

/**
* set class's static value<br>
* 设置clazz的静态变量值
*/
public void setStaticField(Class clazz, String field, Object value);

/**
* 返回spring代理的目标对象
*/
public <T> T getSpringAdvisedTarget(Object source);

/**
* 创建target对象field字段的代理实例<br>
* 用于运行时转移代理操作到字段对象上
*/
public <T> T newProxy(Class target, String fieldname);

/**
* 创建clazz对象实例<br>
* 不是通过 new Construction()形式
*/
public <T> T newInstance(Class<T> clazz);

/**
* 从json字符串创建对象
*/
public <T> T newInstance(String json);

/**
* 从json字符串创建对象
*/
public <T> T newInstance(String json, Class<T> clazz);
}

```

从定义中，JTesterReflector的操作可以分为三类：

- 方法调用类型的，包括静态方法和非静态方法。
- 变量设置和取值类型的，包括静态和非静态的变量。

- 构造类的实例。

下面，我们将分章节介绍这些内容。

## 调用私有方法

反射调用一个对象target中的私有方法，在jTester框架中是非常简单的，基本形式就如下例：

### 例 6.1. 私有方法测试示例

```
public class ReflectorDemo extends JTester {
    @SpringBeanByName
    private WebFileManageBo webFileManageBo;

    @Test
    public void reflectorAccessor() throws Exception {
        String result = reflector.invoke(webFileManageBo, "sayHello", "darui.wu");
        // 具体断言
    }
}
```

例子中只需要一行话就等效于前面的5行话，和普通的方法调用形式基本类似。在本例中，反射调用是通过API: `reflector.invoke`来达到的。其中，方法的所有者是`webFileManageBo`，要调用是有一个String类型的参数，名称为`sayHello`的方法，具体的参数值是`"darui.wu"`。

上面的反射调用，不但对原生的目标对象有效，而且对经过Spring代理过的目标对象同样有效。因为jTester内部会先对目标对象做一个判断，如果是原生对象就直接反射调用，如果是个Advised对象（Spring代理对象），jTester就会先把真正的目标对象先得到，然后才对这个目标对象做反射调用。因此不会存在方法找不到的错误。

```
public static Object getAdvisedObject(Object target) {
    if (target instanceof org.springframework.aop.framework.Advised) {
        try {
            return ((org.springframework.aop.framework.Advised)
                target).getTargetSource().getTarget();
        } catch (Exception e) {
            throw new JTesterException(e);
        }
    } else {
        return target;
    }
}
```

但上面的反射调用，对类中中只有一个签名形式一样的私有类是适用，但我们假设一种情况，B类继承于A类，A类和B类中都有一个sayHello方法。

```
public class A {
    private String sayHello(String name) {
        return "private method of Class A";
    }
}

public class B extends A {
    private String sayHello(String name) {
        return "private method of Class B";
    }
}
```

同时，我们有下面的测试方法：

```
B target = new B();

@Test
public void testInvoke() {
```

```
String result = reflector.invoke(target, "sayHello", "nothing");
System.out.println(result);
}
```

这时候，sayHello方法应该是距离target实现类最近的方法，也即是B类中的sayHello方法，调用的返回值是“private method of Class B”。假如我们的真实意图是想调用A类中的sayHello方法，那应该怎么办？如果想这样做，你在调用时必须显式的指定声明这个方法的类，示例如下：

### 例 6.2. 反射调用时，显式指定方法的声明类

```
B target = new B();

@Test
public void testInvoke() {
    String result = reflector.invoke(A.class, target, "sayHello", "nothing");
    System.out.println(result);
}
```

这时候，上面的例子就会访问A类中的sayHello方法，返回值就是“private method of Class A”。

如果你要访问的是静态方法，可以使用API: reflector.invokeStatic。这是就无需一个实例化的目标对象，只要指定要访问的类就可以了。同样的，假定现在我们的A,B类定义如下：

```
public class A {
    private static String myStaticMethod() {
        return "static of A";
    }

    private String sayHello(String name) {
        return "private method of Class A";
    }
}

public class B extends A {
    private static String myStaticMethod() {
        return "static of B";
    }

    private String sayHello(String name) {
        return "private method of Class B";
    }
}
```

要访问A类和B类的静态方法，可以如下例使用：

### 例 6.3. 反射调用静态方法示例

```
B target = new B();

@Test
public void testInvokeStaticOfB() {
    String result = reflector.invokeStatic(B.class, "myStaticMethod");
    want.string(result).isEqualTo("static of B");
}

@Test
public void testInvokeStaticOfA() {
    String result = reflector.invokeStatic(A.class, "myStaticMethod");
    want.string(result).isEqualTo("static of A");
}
```



## 访问私有变量

有开发同学经常问，我想mock一个变量怎么办？这个提问是有问题的，我们可以mock方法，但不能mock变量。因为变量是可以直接取值和赋值的。跟反射调用一样，我们一样可以对私有变量进行反射访问。

现在我们假定类A的定义如下：

```
public class A {
    private String name;

    //其它变量或方法定义
}
```

要给A类中的变量name赋值和取值，可以分别使用API：reflector.setField和reflector.getField。示例如下：

### 例 6.4. 反射方式给变量赋值和取值

```
A targetA = new A();

@Test
public void invokeField() {
    reflector.setField(targetA, "name", "darui.wu");
    String result = reflector.getField(targetA, "name");
    want.string(result).isEqualTo("darui.wu");
}
```

在例子中，reflector.setField操作是A类的实例对象targetA中的名为name的变量，并给这个变量赋值"darui.wu"。而reflector.getField是将值给取出。和反射调用方法类似，变量是距离targetA实现类最近的变量声明，也即是A类中的name字段。

现在，我们假定A，B类的定义如下：

```
public class A {
    private static Integer age;

    private String name;

    //其它变量或方法定义
}

public class B extends A {
    private static Integer age;

    private String name;

    //其它变量或方法定义
}
```

有测试方法如下：

```
B targetB = new B();

@Test
public void invokeField() {
    reflector.setField(targetB, "name", "darui.wu");
    String result = reflector.getField(targetB, "name");
    want.string(result).isEqualTo("darui.wu");
}
```

上述例子的操作都是针对B类中的变量name进行的，如果我们要操作A类中变量，就必须显示的声明。

## 例 6.5. 反射方式给变量赋值和取值(显示声明变量的声明类)

```

B target = new B();

@Test
public void invokeField2() {
    reflector.setField(B.class, target, "name", "darui.wu");
    reflector.setField(A.class, target, "name", "jobs.he");

    String result = reflector.getField(B.class, target, "name");
    want.string(result).isEqualTo("darui.wu");

    result = reflector.getField(A.class, target, "name");
    want.string(result).isEqualTo("jobs.he");
}

```

静态变量的访问和静态方法访问一样，无需指定一个实例化对象，只要声明定义类就可以了。使用 API: `getStaticField` 和 `setStaticField`，示例如下：

## 例 6.6. 反射方式给静态变量赋值和取值

```

public void invokeStatic() {
    reflector.setStaticField(A.class, "age", 30);
    Integer age = reflector.getStaticField(A.class, "age");
    want.number(age).eq(30);
}

```

## 使用反射方式构造对象实例

在jTester反射模块中，除了方法调用，变量取值赋值外，还存在一类 API构造实例。为什么需要有构造实例的API呢，难道我自己不能够new出一个实例吗？有下列几种情形你可能无法new一个实例：

- 类中没有无参构造函数，只有有参的构造函数。但在测试时，可能你仅仅需要一个实例，根本不关心实例是如何构造出来的，如果你随便传参数给有参构造函数，有可能会造成构造函数错误。
- 类中的构造函数都是私有的，没有可供外部使用的构造函数。

jTester可以克服上面2个问题，不管你的类的构造函数是否有参或无参，是否是public或者private的，它都可以帮助你实例出一个对象。

假定我们有下面的类ErrorConstructor:

```

public class ErrorConstructor {
    private String name;

    private Integer age;

    private Integer grade;

    private boolean isFemale;

    public ErrorConstructor(String arg1, String arg2) {
        throw new RuntimeException("error");
    }

    public String sayHello(String name) {
        return "hello, " + name;
    }
}

```

这个类的构造函数是变态的，实际上当然没有这样简单粗暴的抛出异常的构造函数，这里仅仅是为了举例。如果我们要测试ErrorConstructor中的sayHello方法，也许大家会无可奈何，但jTester允许你调用它的api直接实例化ErrorConstructor对象。

### 例 6.7. 使用反射方式构造实例

```
@Test
public void reflectorConstructor() {
    ErrorConstructor instance = reflector.newInstance(ErrorConstructor.class);
    String result = instance.sayHello("darui.wu");
    want.string(result).eq("hello, darui.wu");
}
```

好像很神奇哦，这种构造方式利用了jdk很底层的方法，类似于C/C++中分配内存的方式，先给对象ErrorConstructor分配一块内存，然后再把对象强制转换为ErrorConstructor对象。既然这样，有参构造函数中发生的任何事情都跟它无关。但如果一个对象有无参构造函数，jTester默认还是先调用无参的构造函数的。

上述方式构造出来的实例，其里面的变量都是默认状态的，如果我们要给它们赋一个默认值，怎么办？我们当然可以调用reflector.setField方式，给实例中变量一一赋值，如下例所示：

```
@Test
public void reflectorConstructor_SetField() {
    ErrorConstructor instance = reflector.newInstance(ErrorConstructor.class);
    reflector.setField(instance, "name", "jobs.he");
    String name = instance.getName();
    want.string(name).eq("jobs.he");
}
```

这种方式在设置一个两个变量时，是挺方便的，但如果要设置的变量很多，就会变的比较低效。这时候，我还可以从json中反序列对象。

### 例 6.8. 从json串中构造实例

```
@Test
public void newInstanceFromJSON() {
    ErrorConstructor instance = reflector.newInstance("{name:'jobs.he', age:30, grade: 5, isFemal:false}",
        ErrorConstructor.class);

    want.object(instance).propertyEqMap(new DataMap() {
        {
            this.put("name", "jobs.he");
            this.put("age", 30);
            this.put("grade", 5);
            this.put("isFemal", false);
        }
    });
}
```

构造对象就这么简单,一条命令搞定。

## 第 7 章 在测试代码中使用Mock

单元测试的职责是验证当前代码的正确性，但实际上，代码可能和其它模块或系统发生着千丝万缕的联系。这时候，我们必须把当前的代码和外围的系统或模块隔离开。这时候我们就必须用到mock，stub这些手段。

相对于stub,fack object等手段，Mock可以让我们比较自由的模拟依赖项的行为，很多在现实环境下无法重现的场景，我们都可以进行演练。但mock也有它的缺点，就是模拟的行为必须要基于实际api可能行为，如果实际api的契约发生了变化，mock没有做对应的修改，那么可能就失去意义。

mock的框架有很多，jtester使用的是jmockit，jmockit的官方网站是：<http://code.google.com/p/jmockit/>， 其也提供了详尽的文档说明<http://jmockit.googlecode.com/svn/trunk/www/tutorial.html>。在这里，我只做一些我们常用的mock方法和使用的介绍。

首先，jmockit的mock从行为状态上可以分为静态mock和动态mock。静态mock其实是一种变形的stub，它修改对应实现类的方法字节码（如果是接口就生成字节码）， 并通过java.instrument（参加：<http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html>）技术使字节码生效， 所以静态mock不存在录制，回放，验证这些mock技术常用过程。反之动态mock就存在上述说的3个阶段，我们在动态mock章节会做详细的介绍。

因为静态mock简单有效，但可以解决我们80%场景下的模拟，所以我们下面先介绍静态mock的使用。

### 静态mock，new MockUp的使用

静态mock的使用非常的简单，其语法是如下：

```
new MockUp<定义具体方法的实现类>(){
    @Mock
    public 返回值类型 具体方法名称(具体方法参数列表...){
        //mock的行为
    }
};
```

为了更好的说明new MockUp的语法，我们先介绍一个具体的例子。比如有下面DateUtil工具类。

```
public class DateUtil {
    /**
     * 返回当前日期的默认格式("yyyy-MM-dd")字符串
     *
     * @return
     */
    public static final String currDateStr() {
        return toDateTimeStr(now(), "yyyy-MM-dd HH:mm:ss");
    }

    /**
     * 返回当前机器环境时间
     */
    private static final Date now() {
        return new Date();
    }
}
```

DateUtil工具类有一个public静态方法，一个private静态方法：private方法就是简单的返回一个当前的时间；public方法是返回当前日期的格式化字符串。现在，我们想测试格式化的正确性（这个格式化是很有可能发生错误，比如MM和mm写错了，HH和hh写错了等等）。

如果我们直接调用DateUtil.currDateStr()方法，返回的是当前时间的格式字符串，我们没有办法作断言，及时我们可以用正则表达式判断字符串 "\d{4}-\d{2}-\d{2}"是否格式正确，也没有办法判断前面提到的MM mm HH hh写错的情况。这时候，我们就可以用上mock的这个工具，我们可以模拟now()方法，返回一个特定的时间。

## 例 7.1. 使用new MockUp静态mock的一个简单例子

```

public class DateUtilTest extends JTester {
    @Test
    public void testCurrDateStr() {
        new MockUp<DateUtil>() {
            @Mock
            public Date now() {
                return getMockDate(2010, 1, 1);
            }
        };
        String currDate = DateUtil.currDateStr();
        want.string(currDate).isEqualTo("2010-01-01");
    }

    /**
     * 构造一个指定的时间
     *
     * @param year
     * @param month
     * @param day
     * @return
     */
    private static Date getMockDate(int year, int month, int day) {
        Calendar cal = Calendar.getInstance();
        cal.set(year, month - 1, day, 0, 0, 0);
        return cal.getTime();
    }
}

```

上面的例子中，我们使用mock技术替换了DateUtil的now()方法，返回了一个我们自己构造出来的时间"2010年1月1号"，这样我们就可以对格式化字符串进行断言了。这里MockUp中的泛型参数是实现now方法的实现类DateUtil，mock的方法前面要加一个注解@Mock，并且限定符不再是public static，而是public。

## 注意

被mock的方法，不管在原有的实现类中限定符是什么：

static,private,package,protected等等限定条件，其对应的mock方法的限定符通通是public且不能加static。

静态mock的使用要点：

- new MockUp<T>(){}匿名类，其泛型表示要mock的对象的类型。
- 对象非接口类型，mock泛型必须是定义该方法的实现类，如果实现类是父类，那么泛型必须是父类，而不可以是子类。
- @Mock public 方法签名：在mock方法前要加注解@Mock,所有的方法，无论原来的限定符是上面，在mock方法中都是public。

比如说，我们有下面这段代码：

```

public abstract class BaseResourceManager{
    protected boolean reloadClot(ClotInfoDo clotInfoDo, int queryPageSize) {
        // 从他数据库加载特定的资源
        return true;
    }

    // 其它方法
}

```

```
public class ResourceManager extends BaseResourceManager{
    public String getOptionValue(String resName, String key, String language) {
        String option = "test";
        // 返回特定的资源项
        return option;
    }
}
```

现在，我们由于测试需要，我们从不从数据库加载资源，而要加载我们自己准备的资源。这样我们必须mock ResourceManager的reloadClot方法。但下列的写法是错误的，因为reloadClot不是定义在ResourceManager类中，而是定义在其父类BaseResourceManager中。

```
@Test
public void test(){
    // 错误的mock方式,ResourceManager类中没有reLoadClot字节码可供修改
    new MockUp<ResourceManager>(){
        @Mock
        public boolean reloadClot(ClotInfoDo clotInfoDo, int queryPageSize) {
            // mock行为
            return true;
        }
    };
    // 具体测试
}
```

正确的写法应该如下：

### 例 7.2. 使用new MockUp mock父类中定义的方法

```
@Test
public void test(){
    // 正确的mock方式，其泛型是定义reLoadClot方法的基类BaseResourceManager
    new MockUp<BaseResourceManager>(){
        @Mock
        public boolean reloadClot(ClotInfoDo clotInfoDo, int queryPageSize) {
            // mock行为
            return true;
        }
    };
    // 具体测试
}
```

## mock构造函数和静态代码块

常规的方法限定符public,private,package,protected,static,final，在mock方法中，这些限定符通通使用public来表示。除了这些常规的函数外，还有2个特殊的函数：构造函数和静态代码块。这个2个特殊函数也是使用public限定符来表示的：

- public void \$init(){ ... }  
表示构造函数，如果要表示有参构造函数，可以加上参数：public void \$init(String arg1,Integer arg2){}
- public void \$clinit(){ ... }  
表示静态代码块，或者是静态变量的初始化。

比如，我们有下面的业务代码：

```
public class MyService {
    public static String DEFAULT_NAME;
    static{
        DEFAULT_NAME = "default service name";
    }

    public String name;
    public MyService(){
        this.name = DEFAULT_NAME;
    }

    public MyService(String name){
        this.name = name;
    }
}
```

假如我们想mock构造函数，我们可以使用\$init来指代构造函数：

### 例 7.3. 如何mock构造函数

```
public class MyServiceTest extends JTester {
    @Test
    public void testMyService() {
        new MockUp<MyService>() {
            MyService it;

            @Mock
            public void $init() {
                it.name = "mock name";
            }
        };
        // 具体的测试业务方法
    }
}
```

在这里，出现了一个变量it，它是jmockit内置的变量，用来表示当前mock对象的实际示例对象。在这里，就表示一个具体的MyService对象。

如果我们想mock静态代码块，比如上面的static{ DEFAULT\_NAME = "default service name"; }，可以使用函数 public void \$clinit(){ ... }来表示。

### 例 7.4. 如何mock静态代码块

```
public class MyServiceTest extends JTester {
    @Test
    public void testMyService() {
        new MockUp<MyService>() {
            @Mock
            public void $clinit() {
                MyService.DEFAULT_NAME = "mock name";
            }
        };
        // 具体的测试业务方法
    }
}
```

## new MockUp和spring的集成

new MockUp方式是采用替换原有实现类的字节码来达到mock的目的，不管实例是来自哪里，这些被改变的字节码都是可以生效的。因此对应实现类的mock，和spring集成并不需要作额外的工作。



## 例 7.5. new Mock和spring的集成-实现类

```

@SpringApplicationContext({"要加载的spring文件"})
public class YourTest extends JTester {

    @Test
    public void test() {
        new MockUp<YourServiceImpl>() {
            @Mock
            public 返回值    具体要mock的方法() {
                // mock方法体
                return mock值;
            }
        };
        // 具体测试
    }
}

```

在这里，spring容器中有个YourServiceImpl实现类的bean，我们在具体的测试方法中定义对应的MockUp方法就可以达到mock YourServiceImpl方法的目的。

对应实现类，我们采用上述办法就可以；但如果是接口呢？因为接口没有对应字节码，直接采用上述的方式，是无法达到我们的目的。在jmockit中，对接口的new MockUp<接口> 是给生成一个代理类的字节码，我们必须把代理类的实例注入到spring容器中，这样我们就必须用到@SpringBeanFrom这个注解。比如下面代码：

```

public interface IEsbUserService{
    public List<User> getUserByNameFilter(String filter);
}

```

这里IEsbUserService是个Esb接口，它是个远程接口类，在本系统中只有接口，没有实现。如果我们想在spring容器中mock这个接口，可以声明如下：

## 例 7.6. new Mock和spring的集成-接口类

```

@SpringApplicationContext({"要加载的spring文件"})
public class YourTest extends JTester {
    @SpringBeanFrom
    IEsbUserService userService = new MockUp<IEsbUserService>(){
        @Mock
        public List<User> getUserByNameFilter(String filter){
            return new ArrayList<User>();//你mock的值
        }
    }.getMockInstance();

    @Test
    public void test() {
        new MockUp<YourServiceImpl>() {
            @Mock
            public 返回值    具体要mock的方法() {
                // mock方法体
                return mock值;
            }
        };
        // 具体测试
    }
}

```

这样，测试类中变量userService是一个mock实例，@SpringBeanFrom的作用 (@SpringBeanFrom的详细用法参加spring章节)是往spring容器中定义一个bean，userService，具体值就是变量的实际值:在这里就是一个mock实例。

其id为



## 注意

new MockUp的getMockInstance只针对接口类有效，对实现类是无效的。对实现类，getMockInstance返回值永远是null。

如果对实现类的new MockUp使用 @SpringBeanFrom和getMockInstance，那么对应的spring bean就是一个null，所以具体的调用过程中会抛出NullPointerException。

上面的mock都是针对普通的方法，如果我们想mock一个实现类的构造函数（静态代码块），或者spring bean的init方法中要调用到的方法。这时候我们在测试类或测试方法中定义的新MockUp并不会在spring容器初始化过程中起作用。因为spring容器的初始化是在@BeforeClass的生命周期实现的。为了使mock的方法在spring容器启动时就能生效，我们引进@SpringInitMethod这个注解。

### 例 7.7. new Mock和spring的集成-@SpringInitMethod使用

```
@SpringApplicationContext({"要加载的spring文件"})
public class YourTest extends JTester {
    @SpringInitMethod
    protected void mockUserServiceImpl(){
        new MockUp<UserServiceImpl>(){
            @Mock
            public void $init(){
                //具体mock行为
            }
        };
    }

    @Test
    public void test() {
        // 具体测试
    }
}
```

上面的mockUserServiceImpl方法会在spring容器初始化之前被调用，这样具体的mock行为在容器初始化前就生效了。比如上面的例子，spring容器在实例化UserServiceImpl的时候，不是调用UserServiceImpl的原始构造函数，而是调用mock的构造函数\$init。

## 针对静态mock做断言

我们知道一个方法总是有输入和输出的，输入包括直接输入和间接输入。直接输入就是方法的参数，间接输入是函数体中读取的外围参数，比如环境的变量，时间，数据库等等，或者从外部接口获取到的参量；直接输出就是方法的返回值，间接输出就是方法中被改变的环境变量，时间，数据库状态等，或者是向外部接口post计算好的参量。如果我们要测试这个方法，那么mock对象就是那些间接输入和输出，而我们的验证对象是直接输出和间接输出。

那对应mock的方法，我们怎样做验证呢？下面，我们会从2个方面对mock的方法做验证：

- 对mock方法的参数进行断言
- 对mock方法的调用次数进行断言

假定我们有下面的业务代码，我们要把方法传入的参数，进行一系列的运算后，然后调用一个外部接口，将这个运算结果作为外部接口的输入。

```
public class CustomerService {
    EsbService esbService;

    /**
     * 根据客户id，单价，商品数量计算客户应付金额
     *
     * @param customerId
     *      客户id
     * @param unitPrice
     *      商品单价
     * @param count
```

```
*          商品数量
* @return
*/
public double shouldPayFor(String customerId, double unitPrice, int count) {
    double totalAmount = unitPrice * count;
    double discount = 1.0;
    if (totalAmount >= 1000.0) {
        discount = 0.8;
    } else if (totalAmount >= 500.0) {
        discount = 0.9;
    }
    double amount = totalAmount * discount;
    // 外部接口, 根据会员ID返回会员等级
    int level = esbService.getCustomerLevel(customerId);

    // getDiscountByAmountAndLevel 外部接口, 根据总金额和会员等级再获取一定的折扣
    double discountAgain = esbService.getDiscountByAmountAndLevel(amount, level);

    return amount * discountAgain;
}

public void setEsbService(EsbService esbService) {
    this.esbService = esbService;
}
}
```

这是一个非常简化的模型, 假如我们要测试shouldPayFor这个方法, 我们就必须mock掉2个外部接口getCustomerLevel和getDiscountByAmountAndLevel。这时, 我们可以在getDiscountByAmountAndLevel这个方法中验证商品的经过第一次折扣后的总价。

## 例 7.8. 在new MockUp的mock方法中断言传入的参数值

```

public class CustomerServiceTest extends JTester {
    private CustomerService customerService = new CustomerService();

    /**
     * 验证总价经过2次打折后的应付金额
     *
     * @param unitPrice
     *         商品单价
     * @param count
     *         商品数量
     * @param amountFirst
     *         经过第一次打折后的商品总价的期望值
     */
    @Test(dataProvider = "dataForShouldPayFor")
    public void testShouldPayFor(double unitPrice, int count, final double amountFirst) {
        customerService.setEsbService(new MockUp<EsbService>() {
            @Mock
            public double getDiscountByAmountAndLevel(double amount, int level) {
                // 验证经过第一次打折后的总价
                want.number(amount).isEqualTo(amountFirst);
                // 不管用户总价和用户等级, 返回的折扣总为9折
                return 0.9d;
            }

            @Mock
            public int getCustomerLevel(String customerId) {
                return 1;
            }
        }).getMockInstance();

        double amount = customerService.shouldPayFor("1001", unitPrice, count);
        want.number(amount).isEqualTo(amountFirst * 0.9);
    }

    @DataProvider
    public Iterator dataForShouldPayFor() {
        return new DataIterator() {
            {
                data(1.0d, 1000, 800.0d);
                data(2.0d, 300, 540.0d);
                data(2.0d, 200, 400.0d);
            }
        };
    }
}

```

在上面的测试样例中，我们在mock方法getDiscountByAmountAndLevel中对第一次打折的商品总价这个中间值做了断言：want.number(amount).isEqualTo(amountFirst)。new MockUp是静态改变实现类的字节码，但如果该方法没有被调用到，那么mock方法中对参数的断言也就不会被运行到，但测试不会感知到这一点的。

```

public class CustomerService {
    EsbService esbService;

    /**
     * 根据客户id, 单价, 商品数量计算客户应付金额
     */
    public double shouldPayFor(String customerId, double unitPrice, int count) {
        double totalAmount = unitPrice * count;
        double discount = 1.0;
        if (totalAmount >= 1000.0) {

```

```

    discount = 0.8;
} else if (totalAmount >= 500.0) {
    discount = 0.9;
}
double amount = totalAmount * discount;
// 外部接口, 根据会员ID返回会员等级
int level = esbService.getCustomerLevel(customerId);

// getDiscountByAmountAndLevel 外部接口, 根据总金额和会员等级再获取一定的折扣
double discountAgain = 0.9; //esbService.getDiscountByAmountAndLevel(amount, level);

return amount * discountAgain;
}

public void setEsbService(EsbService esbService) {
    this.esbService = esbService;
}
}

```

比如上面的代码片段, 我们并没有调用外部接口, 而是直接给discountAgain赋了个0.9的值(开发中, 可能经常这样干)。但如果运行上面的测试代码, 测试照样通过, 这样就可能留下一个隐患(开发忘了改回来了)。我们可以看一下@Mock这个注解定义的变量:

```

public @interface Mock
{
    /**
     * mock方法期望被调用的次数, 0表示不会被调用到, 负值表示不限制调用次数 (0或任意次都可以)
     * 正值表示刚好被调用若干次 (即表示minInvocations = maxInvocations)
     */
    int invocations() default -1;

    /**
     * 期望最少被调用的次数, 默认值是0
     */
    int minInvocations() default 0;

    /**
     * 期望最多被调用的次数, 默认值是-1, 表示无限制
     */
    int maxInvocations() default -1;
}

```

这样, 我们就可以给mock方法getDiscountByAmountAndLevel加上调用次数的限制:

## 例 7.9. 限制new MockUp中mock方法被调用的次数

```

public class CustomerServiceTest extends JTester {
    private CustomerService customerService = new CustomerService();

    /**
     * 验证总价经过2次打折后的应付金额
     *
     * @param unitPrice 商品单价
     * @param count 商品数量
     * @param amountFirst 经过第一次打折后的商品总价的期望值
     */
    @Test(dataProvider = "dataForShouldPayFor")
    public void testShouldPayFor(double unitPrice, int count, final double amountFirst) {
        customerService.setEsbService(new MockUp<EsbService>() {
            @Mock(invocations = 1)
            public double getDiscountByAmountAndLevel(double amount, int level) {
                // 验证经过第一次打折后的总价
                want.number(amount).isEqualTo(amountFirst);
                // 不管用户总价和用户等级, 返回的折扣总为9折
                return 0.9d;
            }

            @Mock
            public int getCustomerLevel(String customerId) {
                return 1;
            }
        }).getMockInstance();

        double amount = customerService.shouldPayFor("1001", unitPrice, count);
        want.number(amount).isEqualTo(amountFirst * 0.9);
    }

    @DataProvider
    public Iterator dataForShouldPayFor() {
        return new DataIterator() {
            {
                data(1.0d, 1000, 800.0d);
                data(2.0d, 300, 540.0d);
                data(2.0d, 200, 400.0d);
            }
        };
    }
}

```

这样，我们限制了getDiscountByAmountAndLevel方法必须被调用一次@Mock(invocations = 1)。对于上面直接给discountAgain赋值0.9，而不调用外部接口的情况，运行测试就会抛出异常，从而可以检测到具体的错误。

## 动态mock，new Expectations的使用