

一步一步学写测试

吴大瑞 2010.01.13

Jtester 的项目配置

- 1、martini 下项目配置
- 2、使用 ant 的项目配置
- 3、Maven 项目的配置
- 5、jtester.properties 文件配置

TestNG 测试

一个简单到不能再简单的测试

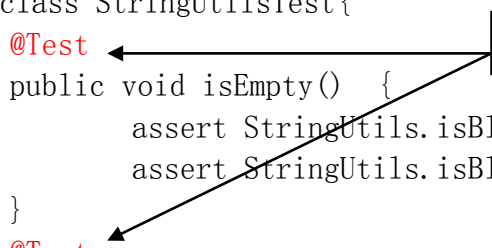
您只要用标注 `@Test` 通知框架这个类的方法是测试。

`@Test` 这个标注可以写在测试类 `class` 前，也可以写在测试方法 `method` 前。

写在测试类前，表明这个类中签名为 `public void` 的方法都是一个测试方法。

清单 1 演示了实用类 `StringUtils` 的一个最简单的测试。它测试 `StringUtils` 的两个方法：`isEmpty()` 方法检测 `String` 是否为空；`trim()` 方法从 `String` 两端删除控制字符。请注意，其中使用了 Java 指令 `assert` 来检测错误情况。

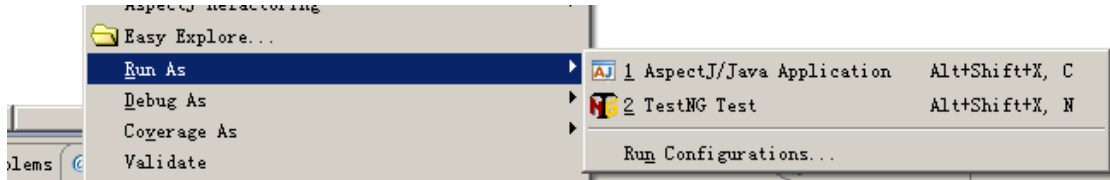
```
import com.beust.testng.annotations.*;
import org.apache.commons.lang.StringUtils;
public class StringUtilsTest{
    @Test
    public void isEmpty() {
        assert StringUtils.isBlank(null);
        assert StringUtils.isBlank("");
    }
    @Test
    public void trim() {
        assert "foo".equals(StringUtils.trim(" foo "));
    }
}
```



`@Test` 表明这 2 个方法是测试方法

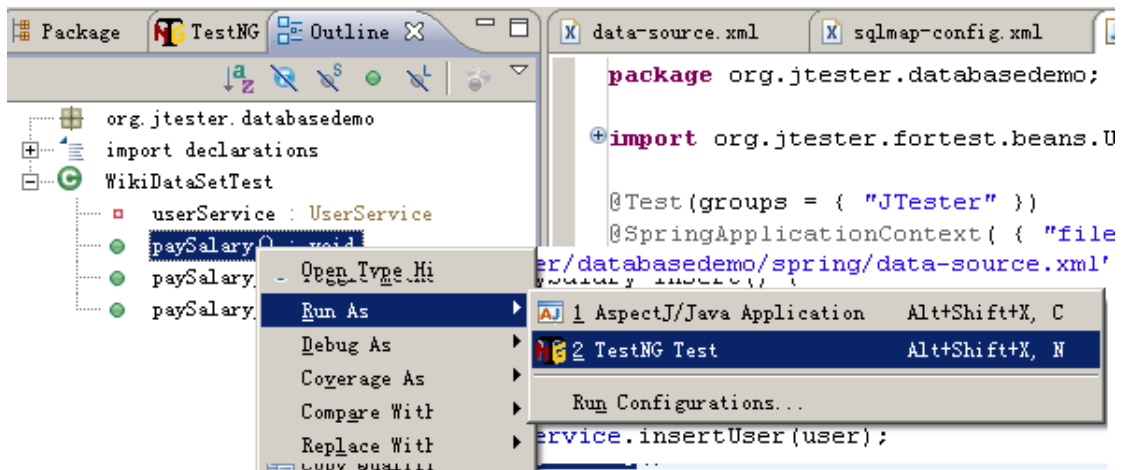
在 eclipse 中运行 testng 测试

安装eclipse插件: <http://beust.com/eclipse/> , 安装好TestNG后, 在Eclipse中单击"Window"->Show View->Other->Java->TestNG, TestNG的视图就打开了。在测试类的 java editor 中右键菜单中会出现如下选项



运行 testNG Test 就可以跑这个类中的所有测试方法了。

如果你要跑指定的测试方法, 可以打开 outline 视图



右键选定你要跑的测试, 运行 testng test 选项, 就可以跑你指定的测试方法, 而不用跑这个类了。

定义测试组

TestNG 可以将一个测试方法定义为属于一个或多个测试组, 但可以选择只运行某个测试组。要把测试加入测试组, 只要把组指定为 @Test 标注的参数, 使用的语法如下:

```
@Test(groups = {"tests.string"})
```

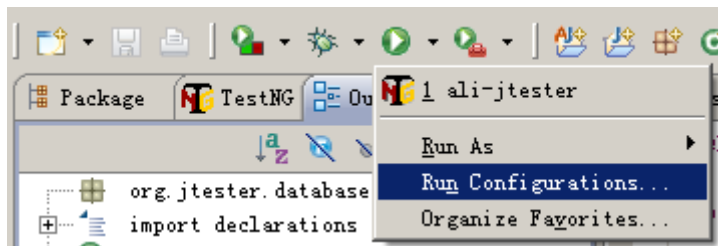
如果这个 annotation 是加在 class 前面, 表明这个类的所有测试方法都属于这个测试组, 如果这个 annotation 是加在 method 前, 只表明这个方法属于这个测试组。同时, 测试组是可以继承和叠加的。

```
@Test (groups={"mytest1"})
public class BaseTest{
    public void test1() {...}
}

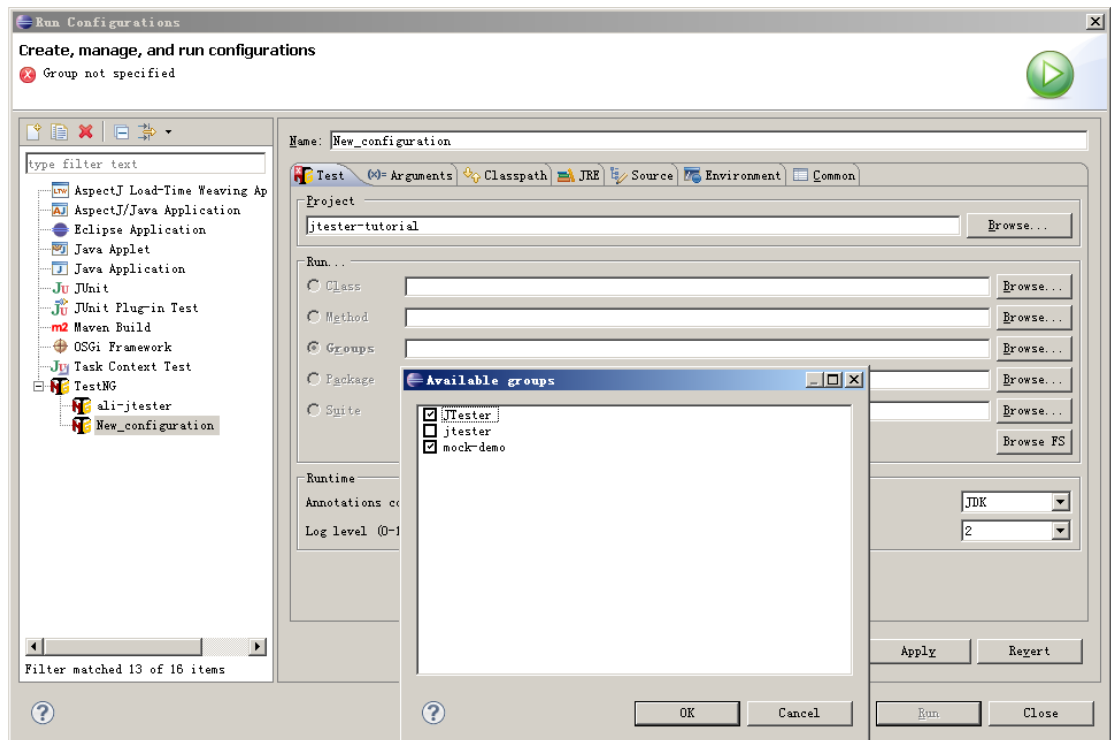
@Test (groups={"mytest2"})
public class ConcreteTest extends BaseTest{
    public void test2() {...}

    @Test (groups={"mytest3"})
    public void test3() {...}
}
```

在上面的例子中，方法 test1 属于测试组 mytest1, 方法 test2 属于测试组 mytest1 和 mytest2, 方法 test3 属于测试组 mytest1、mytest2 和 mytest3。
运行指定的测试组，在 eclipse 的 run 菜单下单击"Run Configuration"选项，如下图：

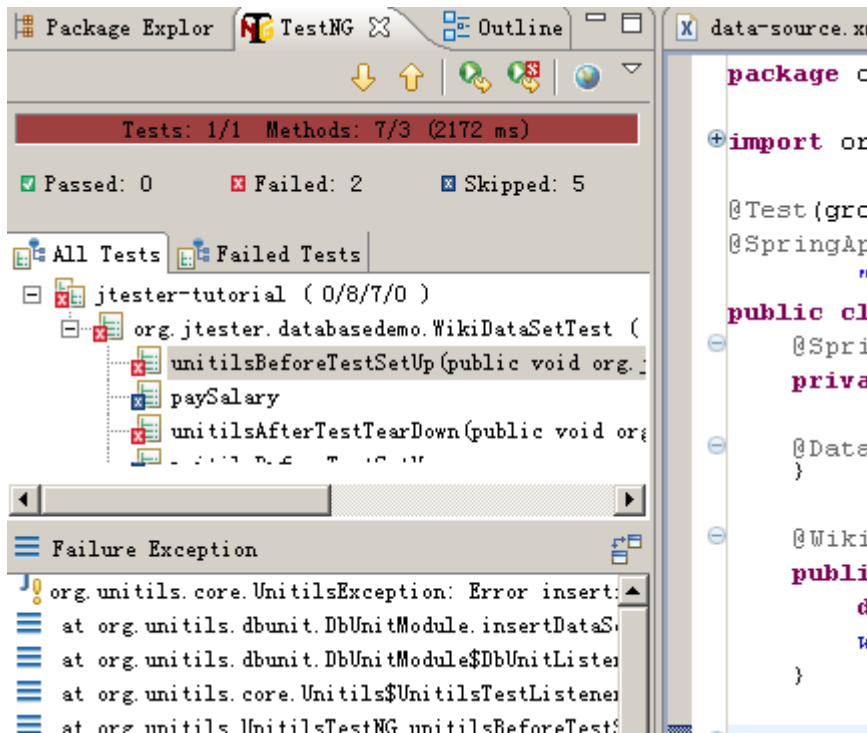


则会打开如下界面

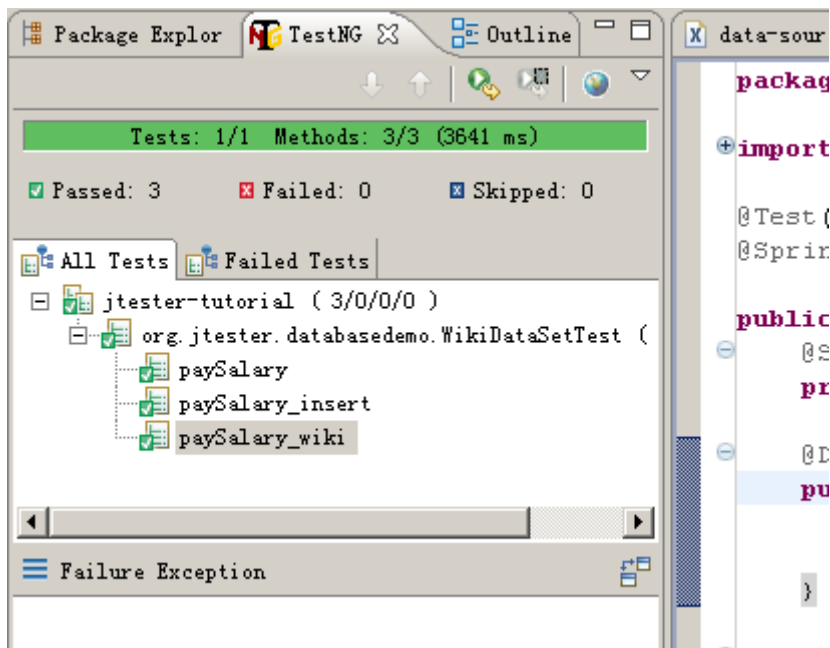


Name 这个选项是你给当前的测试一个命名，你可以随便取，project 选项是你跑的测试位于那个项目下。Class、method、groups、package 和 suite 这 5 个选项表明你可以按这 5 种方式来运行的测试。现在，我们选中 groups 选项，然后点击这个选项后对应的 Browse 按钮，会弹出一个窗口，列出所有可见的测试组，你可以选择你希望跑的测试组，然后点击 OK 和 RUN 按钮，eclipse 就会自动运行所有属于选中的测试组的测试方法。

运行结果会显示在 TestNG view 视图中，红色表示这些测试没有通过，相应的异常会显示在 Failure Exception 中。



如果测试方法运行是成功的，则 TestNG view 视图会显示绿色进度条。

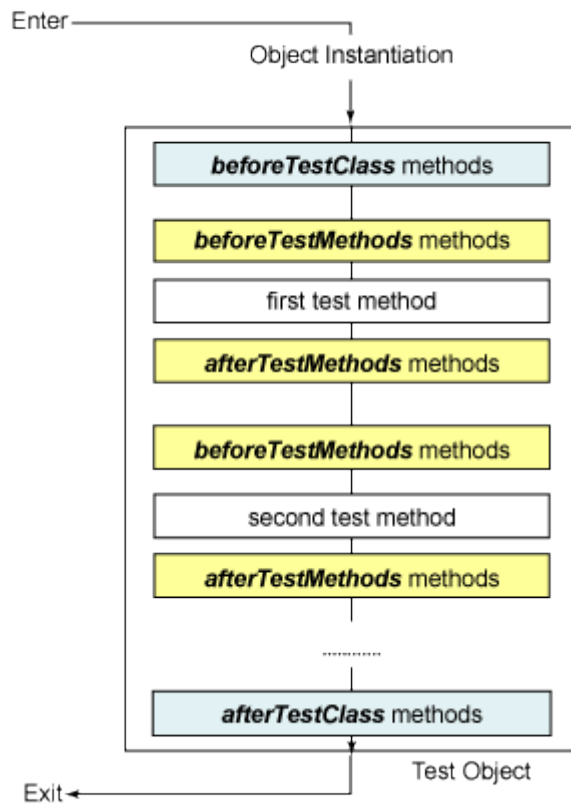


生命周期

使用 TestNG，不仅可以指定测试方法，还可以用专门的 Annotation 指定类中的其他特定生命周期：

- @BeforeSuite, 在测试套件之前运行
- @AfterSuite, 在测试套件之后运行
- @BeforeTest, 在测试方法运行之前运行
- @AfterTest, 在测试方法之后运行
- @BeforeGroups, 在测试组之前运行
- @AfterGroups, 在测试组之后运行
- @BeforeClass, 在测试类之前运行
- @AfterClass, 在测试类之后运行
- @BeforeMethod, 在测试方法之前运行
- @AfterMethod, 在测试方法之后运行

一个可能的生命周期示例图如下：



在 testng 中使用 jtester

在 testng 中使用 jtester 非常简单，只需用你的测试类继承 `org.jtester.testng.JTester` 就可以了。

断言的使用

在 jtester 中实现了强大的 fluent interface 式的断言语法。其语法格式如下：

`want.object(value).assert()...`

want 后面跟着你要断言的对象类型，目前 jtester 提供了下列几种对象的断言

- object
- string
- number (int, double, float, long)
- boolean
- collection (list, set, collection)
- date/calendar
- array (数组)
- map
- file

对象后面紧跟着具体的对这个对象属性的判断

所有对象都支持的断言

命令	说明
<code>isEqualTo(T expected)</code>	对象等于期望对象
示例 <code>want.string("test").isEqualTo("test")</code>	
<code>notEqualTo(T expected)</code>	对象不等于期望的值
示例 <code>want.string("test1").isEqualTo("test2")</code>	
<code>in(T... values)</code>	对象可以在期望值里面找到
示例 <code>want.number(2).in(1, 2, 3)</code>	
<code>notIn(T... values)</code>	对象不在指定的值里面
示例 <code>want.character('a').notIn('b', 'c');</code>	
<code>clazzIs(Class<?> clazz)</code>	对象的 class 等于期望的 class 对象实例
示例 <code>want.map(new HashMap<String, Object>()).clazzIs(String.class);</code>	
<code>is(Matcher<T> matcher)</code>	对象符合指定的断言器所指定的行为
示例 <code>want.string("1234").is(the.string().contains("23"));</code>	
<code>not(Matcher<T> matcher)</code>	对象不符合断言器指定的行为
示例 <code>want.string("1234").not(the.string().contains("13"));</code>	
<code>allOf(IAssert matcher1, IAssert matcher2, IAssert... matchers)</code>	对象满足参数里指定的所有断言
<code>allOf(Iterable<IAssert> matchers)</code>	同上
<code>anyOf(IAssert matcher1, IAssert matcher2, IAssert... matchers)</code>	对象符合任意一个断言就可以
<code>anyOf(Iterable<IAssert> matchers)</code>	同上
<code>same(T value)</code>	对象和被期望值是同一个对象
<code>any()</code>	对象可以是任意值
<code>isNull()</code>	对象值等于 null
<code>notNull()</code>	对象只不为 null
<code>reflectionEq(Object expected, ReflectionComparatorMode... modes)</code>	断言对象和期望对象在形式上是相等的

示例 want.object(Arrays.asList(3, 2, 1)).reflectionEq(Arrays.asList(1, 2, 3))	
lenientEq(Object expected)	断言对象和期望对象在某种形式上是相同的, 但忽略对象顺序关系
示例 want.object(Arrays.asList(3, 2, 1)).lenientEq(Arrays.asList(1, 2, 3))	
propertyEq(String property, Object expected)	对象指定的属性(property)值等于期望值
示例 want.object(employee).propertyEq("name", "my name");	
propertyMatch(String property, Matcher<?> matcher)	对象指定的属性符合 matcher 定义的行为
示例 want.object(user).propertyMatch("name", the.string().contains("john"))	

number 和 string 对象支持的断言

命令	说明
lessThan(T max)	对象小于期望值 max
示例 want.number(3).lessThan(4);	
lessEqual(T max);	对象小于等于期望值 max w
示例 ant.number(3).lessThan(3);	
greaterThan(T min)	对象大于期望值 min
示例 want.string("abc").greaterEqual("aaa");	
greaterEqual(T min)	对象大于等于期望值 min
示例 want.string("abc").greaterEqual("aaa");	
between(T min, T max)	对象在最小值和最大值之间(包括最大值和最小值)
示例 want.number(2.3).between(2d, 3d);	

array 和 collection 对象支持的断言

命令	说明
sizeIs(int size)	数组长度或 collection 中元素个数等于期望值
sizeEq(int size)	数组长度或 collection 中元素个数等于期望值
sizeGt(int size)	数组长度或 collection 中元素个数大于期望值
sizeGe(int size)	数组长度或 collection 中元素个数大于等于期

	望值
sizeLt(int size)	数组长度或 collection 中元素个数小于期望值
sizeLe(int size)	数组长度或 collection 中元素个数小于等于期望值
sizeNe(int size)	数组长度或 collection 中元素个数不等于期望值
hasItems(Collection<?> coll)	数组或集合中的元素包含期望集合中列出的元素
hasItems(Object value, Object... values)	数组或集合中的元素包含期望元素
hasItems(Object[] values)	数组或集合中的元素包含期望数组中列出的元素
hasItemMatch(String regular, String... regulars)	参数中列出的正则表达式可以被数组或集合中的元素满足
allItemMatch(String regular, String... regulars)	数组或集合中所有的元素 toString() 必须满足所有列出期望的正则表达式

string 对象支持的断言

命令	说明
contains(String expected)	字符串包含期望的字串 expected
示例 want.string("ddd").contains("d").contains("de");	
end(String expected)	字符串以 expected 子串结尾
示例 want.string("eeeed").end("ed");	
start(String expected)	字符串以 expected 子串开头
示例 want.string("eeeed").start("eee");	
regular(String regex)	字符串符合正则表达式 regex
示例	
eqIgnoreCase(String string)	字符串在忽略大小写的情况下等于期望值
示例 want.string("abC").eqIgnoreCase("aBc");	
eqIgnorBlank(String string)	字符串在忽略前后空格的情况下等于期望值
示例 want.string(" ab ").eqIgnorBlank("ab");	
notContain(String str)	字符串不包含特定的子串
notBlank()	字符串不是空白串

map 对象支持的断言

命令	说明
<code>hasKeys(Object key, Object... keys)</code>	map 包含参数中列出的 key 值
示例 <code>want.map(maps).hasKeys("one", "two");</code>	
<code>hasValues(Object value, Object... values)</code>	map 包含参数中列出的值对象
示例 <code>want.map(maps).hasValues("my first value", "my third value");</code>	

data/calendar 对象支持的断言

命令	说明
<code>yearIs(int year)</code>	日历值的年等于期望值
<code>yearIs(String year)</code>	日历值的年等于期望值
<code>monthIs(int month)</code>	日历值的月份等于期望值
<code>monthIs(String month)</code>	日历值的月份等于期望值
<code>dayIs(int day)</code>	日历值的日期等于期望值
<code>dayIs(String day)</code>	日历值的日期等于期望值
<code>hourIs(int hour)</code>	日历值的小时(24 小时制)等于期望值
<code>hourIs(String hour)</code>	日历值的小时(24 小时制)等于期望值

file 对象支持的断言

命令	说明
<code>isExists()</code>	指定的文件存在
<code>unExists()</code>	指定的文件不存在

在上文中那个简单的 `StringUtil` 测试采用的是 java 自带的 `assert` 语法，改为 `jtester` 断言语法实现如下：

```

import com.beust.testng.annotations.*;
import org.apache.commons.lang.StringUtils;
public class StringUtilsTest{
    @Test
    public void isEmpty() {
        want.bool(StringUtils.isBlank(null)).is(true);
        want.bool(StringUtils.isBlank(" ")).is(true);
    }
    @Test
    public void trim() {
        String str = StringUtils.trim(" foo ");
        want.string(str).isEqualTo("foo");
    }
}

```

FAQ

问题：如何测试异常

解决：如果你不是测试异常，那么你可以直接向 test 方法抛出异常。

```

@Test
public void testNormalBiz() throws Exception{
    checking(new Je() {
        {
            will.call.one(calledService).called(the.string
            will.throwException(new RuntimeException("tes
        }
    });
    callingService.call("i am a test message!");
}

```

如果你要测试异常可以在@Test 中指明异常的类型

```

@Test(expectedExceptions = { YourException.class })
public void throwException() throws Exception{
    checking(new Je() {
        {
            will.call.one(calledService).called(the.string
            will.throwException(new RuntimeException("test
        }
    });
    callingService.call("i am a test message!");
}

```

如果你不仅仅要判断测试的类型，而且还要判断异常里面具体的消息，那么你就需要在测试代码里面 try{...}catch(){...}进行判断了。

```

@Test
public void testBiz_ErrMessage() {
    checking(new Je() {
        {
            will.call.one(calledService).called(the.string().contains(
                will.throwException(new RuntimeException("test exception"))
            )
        }
    });
    try {
        callingService.call("i am a test message!");
    } catch (Exception e) {
        String message = e.getMessage();
        want.string(message).contains("你希望的消息");
    }
}

```

Spring 容器的加载

Jtester 框架对 spring 的使用，没有做更多的扩展，基本上就是 unitils 提供的功能。

spring 容器的启动

加载 spring 上下文的定义，可以使用 `@SpringApplicationContext` 轻易的搞定。

```

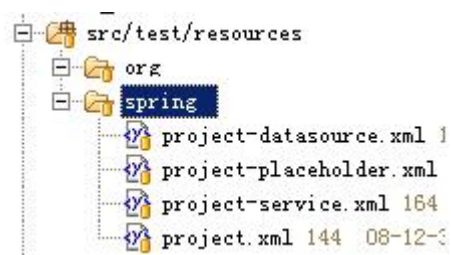
@SpringApplicationContext({"config1.xml", "config2.xml"})
public class UserServiceTest extends JTester {

}

```

就这样，jtester 框架会创建一个 *ApplicationContext*，并且会到 classpath 下面去寻找 config1.xml 和 config2.xml 这 2 个文件，并把它们加载进来。Jtester 会先扫描测试类的 superclass，如果 superclass 定义了 `@SpringApplicationContext`，那么这个子类同样也会使用 superclass 定义好的 applicationcontext，如果子类也定义了 `@SpringApplicationContext`，那将使用子类定义的。

如果 spring 文件是在 classpath 下，只需要指定 spring 的 classpath 路径就可以了。假定我们的 spring 文件的 classpath 如下图所示



我们就只需要 classpath 的路径 + spring 文件名,就可以了。当然你也可以在前面加上前

缀"classpath:"，但这个不是必须的。

```
@SpringApplicationContext({"spring/project-service.xml",
"spring/project-placeholder.xml",
"classpath:spring/project-datasource.xml"})
public class SpringLoadExampleTest extends JTester{
    //.....
}
```

你也可以使用文件路径的方式来查找 Spring 配置文件，格式如下 “file:文件路径”

```
@SpringApplicationContext({"file:./src/test/resources/spring/project-service.xml",
"file:./src/test/resources/spring/project-placeholder.xml",
"file:./src/test/resources/spring/project-datasource.xml"})
public class SpringLoadExampleTest extends JTester{
    //.....
}
```

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：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。

spring bean 的使用

Jtester 初始化好 springapplicationcontext 好后，我们就可以在测试类中使用 spring bean 了。可以使用下列 3 个 annotation 将 spring bean 注入到测试类中

@SpringBean("userService") 显式的指定要注入那个 spring bean

@SpringBeanByType 按类型方式注入

@SpringBeanByName 按名称方式注入

```
@SpringBean("userService")
private UserService userService;

@SpringBeanByName
private UserService userService;

@SpringBeanByType
private UserService userService;
```

使用 annotation 方式注入 spring bean，并不需要在测试类中声明对应的 set 方法。

FAQ

如何测试一个实现类中的 private 或 protected 方法。

场景 ,用户定义了一个接口和实现, 如下图

```
public interface MyService {
    public void mySay();
}

public class MyServiceImpl implements MyService {

    public void mySay() {
        // do business
    }

    protected void internalCall() {
        // ... do something
    }
}
```

其中, 在实现中用户还定义了一个内部方法 internalCall, 现在用户想针对 internalCall 进行测试, 但又不想把 internalCall 暴露到接口中, 怎么办?

我们可以在测试类中临时定义一个扩展接口和实现, 把 internalCall 方法暴露出来。

```
@SpringApplicationContext("org/jtester/testprotected/mybeans.xml")
public class MyServiceImplTest extends JTester {
    @SpringBeanByName
    private ExMyService myService;

    @Test
    public void internalCall() {
        // test internalCall method
        myService.internalCall();
    }

    public static interface ExMyService extends MyService {
        public void internalCall();
    }

    public static class ExMyServiceImpl extends MyServiceImpl
        implements ExMyService {

        public void internalCall() {
            super.internalCall();
        }
    }
}
```

同时 spring 的定义如下:

```
<bean id="myService"
      class="org.jtester.testprotected.MyServiceImplTest$ExMyServiceImpl" />
```

JMockit 使用

从 0.9.3 版本开始推荐使用 jmockit，原来的 jmock 功能保留（为了保证原有的测试可以通过，但不推荐使用）。

jmockit 的优点

传统 mock 方法的限制：

- 1、JDK Proxy 必须实现接口
- 2、Cglib Proxy 的 class 和方法不能是 final 限定的。
- 3、对于静态方法无能为力
- 4、对非 public 方法无能为力，或者需要花比较大的代价进行反射处理。
- 5、比较依赖于 loc 机制，对于 new 或工厂类管理的 bean 无法进行有效的测试。

Jmockit 项目基于 Java 5 SE 的 `java.lang.instrument` 机制，内部使用 ASM 库来修改 Java 的 Bytecode，是一个能帮我们解决以上问题的轻量级框架，它允许你动态的改变已有的方法，这主要基于 java 1.5 的 Instrumentation 框架，允许你重定义 private,static and final 方法，甚至是 no-arg constructors 都能够并轻易的重定义，这样便可以使得 JMockit 能够适应几乎所有的设计。

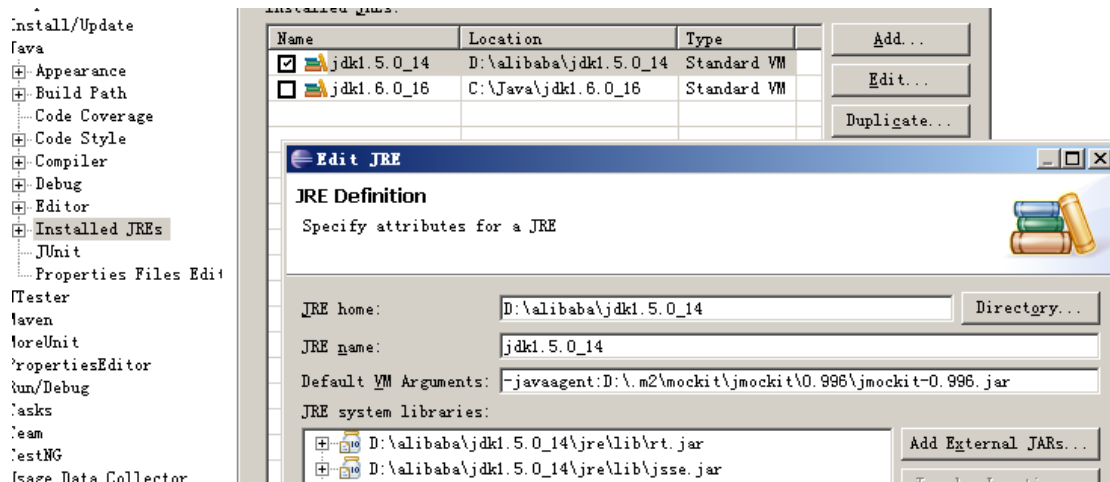
使用 mock 的场景

真实对象有着不确定的行为
真实对象很难创建
真实对象的行为很难触发
真实对象响应缓慢
真实对象是用户界面
真实对象使用了回调机制
真实对象尚未存在

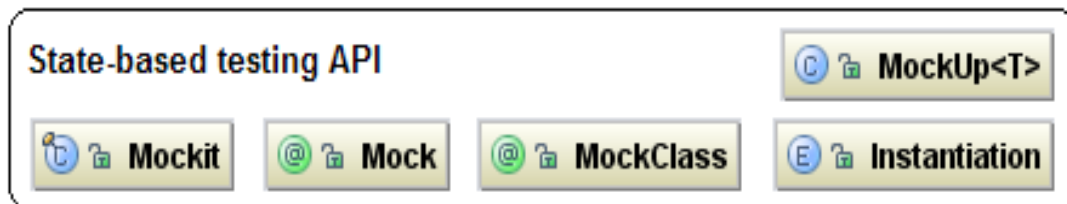
而对应的 mock 具有下面的功能
替换远程对象，如 ESB、WEB Service 对象等
替换复杂的对象
方便模块化开发
... ..

运行 jmockit 测试

对于 JDK5, 需要配置 jvm 参数 `-javaagent:jmockit.jar`, 建议大家在 eclipse 的 jre 参数里搞定。



基于状态的 jmockit 测试



一个简单的业务类


```

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 currDateTimeStr(now(), "yyyy-MM-dd");
    }

    /**
     * 返回当前日期时间的默认格式 ("yyyy-MM-dd mm:hh:ss") 字符串
     *
     * @return
     */
    public static final String currDateTimeStr() {
        return currDateTimeStr(now(), "yyyy-MM-dd HH:mm:ss");
    }

    /**
     * 返回当前时间的格式化字符串
     *
     * @param format
     * @return
     */
    public static final String currDateTimeStr(String format) {
        return currDateTimeStr(now(), format);
    }

    /**
     * 返回指定时间的格式化字符串
     *
     * @param format
     * @return
     */
    public static final String currDateTimeStr(Date date, String format) {
        SimpleDateFormat dateFormat = new SimpleDateFormat(format);
        return dateFormat.format(date);
    }

    public static final Date now() {
        return new Date();
    }
}

```

现在我们想测试日期的格式化字符串是否按照正确的格式进行格式化了, 如果使用系统的当前时间, 那么我们就没法控制格式化后输出的值。为了能够控制输入 (当前时间), 验证输出 (格式串), 我们就必须把 DateUtil 类的 now() 方法给 mock 掉。

方法一：使用@MockClass 和 @UsingMocksAndStubs

```
@UsingMocksAndStubs({ MockDateUtil.class })
public class DateUtilTest_MockClass extends JMockitBaseTest {
    @MockClass(realClass = DateUtil.class)
    public static class MockDateUtil {
        @Mock
        public static Date now() {
            Calendar cal = DateUtilTest.mockCalendar(2012, 1, 28);
            return cal.getTime();
        }
    }

    @Test
    public void testCurrDateTimeStr_format() {
        String str = DateUtil.currDateTimeStr("MM/dd/yy hh:mm:ss");
        want.string(str).isEqualTo("01/28/12 07:58:55");
    }
}
```

@UsingMocksAndStubs: 表明这个测试类用到了哪些 Mock Class。

@MockClass: 定义具体的 mock class, realClass 指明了实际业务类。

@Mock: 指定对应的方法是一个 mock 方法、mock 构造函数 (void \$init({})) 或者 mock 的静态代码块(void \$clinit({})).

对于在 Mock Class 中没有定义的方法和代码块, 测试类将调用原有的业务类对应的方法。

上面的例子, 模拟构造了一个可以预知的当前时间, 这样我们就可以根据这个时间, 断言格式化后的输出值。

在 MockClass 指定实际业务方法 (构造函数) 的一个默认实现代理, 默认实现代理方法返回该方法的一个默认值, 大部分情况下是 null。

```
@MockClass(realClass = LoginContext.class, stubs = { "(String)", "logout" })
final class MockLoginContextWithStubs {
    @Mock
    void login() {
    } // this could also have been an stub
}
```

在上面的例子中, MockLoginContextWithStubs 默认实现了所有参数为 String 的方法和 logout 方法。

方法二：使用 Mockit.setUpMock()

```
public class DateUtilTest_MockItSetUp extends JMockitBaseTest {
    public static class MockDateUtil {
        @Mock
        public static Date now() {
            Calendar cal = DateUtilTest.mockCalendar(2012, 1, 28);
            return cal.getTime();
        }
    }

    @Test
    public void testCurrDateTimeStr_format() {
        Mockit.setUpMock(DateUtil.class, MockDateUtil.class);
        String str = DateUtil.currDateTimeStr("MM/dd/yy hh:mm:ss");
        want.string(str).isEqualTo("01/28/12 07:58:55");
    }
}
```

这里@Mock 的意义的方法是一致的，不同点在于不是采用@Annotation 方式来指定要 mock 哪些业务类，而是采用编码的方式在测试方法（或@BeforeMethod，@BeforeClass）中指定 Mock 的业务类。

方法三：In-line mock classes

```
public class DateUtilTest_InlineMockClass extends JMockitBaseTest {
    @Test
    public void testCurrDateTimeStr_format() {
        Mockit.setUpMock(DateUtil.class, new Object() {
            @Mock
            public Date now() {
                Calendar cal = DateUtilTest.mockCalendar(2012, 1, 28);
                return cal.getTime();
            }
        });
        String str = DateUtil.currDateTimeStr("MM/dd/yy hh:mm:ss");
        want.string(str).isEqualTo("01/28/12 07:58:55");
    }
}
```

上例定义了一个匿名类来 mock 实际的业务类 DateUtil，使用匿名类的效果跟在外面定义一个 static 的 mock class 的效果是一样的。但在语法上有些区别，a、匿名类中不能有构造函数。所以要 mock 实际业务类的构造函数，可以使用上面提到的 public void \$init()方法。匿名类中不能定义静态方法，但只要方法的签名是一样的就可以。

一个语法上更优雅的使用匿名类实现的方式如下：

```

public class DateUtilTest_InlineMockClass extends JMockitBaseTest {
    @Test
    public void testCurrDateTimeStr_MockUp() throws Exception {
        new MockUp<DateUtil>() {
            @Mock
            public Date now() {
                Calendar cal = DateUtilTest.mockCalendar(2012, 1, 28);
                return cal.getTime();
            }
        };
        String str = DateUtil.currDateTimeStr("MM/dd/yy hh:mm:ss");
        want.string(str).isEqualTo("01/28/12 07:58:55");
    }
}

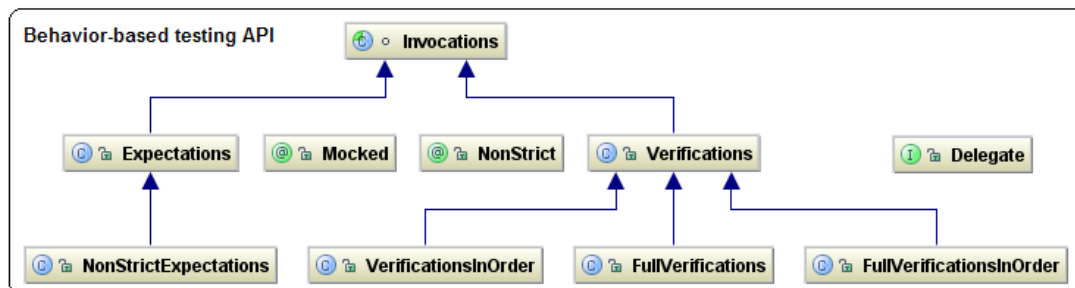
```

在这个例子中，我们同样是创建了一个匿名类，但这个匿名类是 MockUp 的子类型，并且拥有一个泛型参数指明了要 mock 那个业务类。

使用 it 来访问 mock 对象本身

正如我们可以在 class 中使用 this 来访问 class 本身的实例一样，jmockit 也对 mock class 提供了一个变量 it 来访问 mock class 的实例。

基于行为的 jmockit 测试



Mock 对象

Mock 对象是被测对象的依赖项，它可以是接口、抽象类、枚举类、实体类和带有 final 类。基本上，所有的类型都可以被 Mock，除了下列类型：primitive type，数组类型，特别需要注意的是 jre 提供的类同样也可以被 mock 的，如 java.util.* 和 java.lang.*。默认情况下，如果一个依赖项被 mock 了，那么它的所有方法和构造函数，包括它的所有父类（但不包括 java.lang.Object）中的所有方法，都将被 mock。那么在测试中，mock 对象的原始业务方法将不会被调用，而是被转向到 jMockit 来处理。下面我们会详细介绍如何显式或隐式的定义 mock 的行为。

mockit.Expectations

一个 Expectations 是给定测试方法中将会涉及到的 mock 项中预期被调用的方法或构造函数。一个 Expectations 可以包含多个预期的方法，但不必包含所有预期会被调用的方法。在 Expectations 中；除了可以指定预期的方法外，还可以指定方法的参数的精确值或约束行为（满足某个断言）；同时 Expectations 中还可以指定该方法预期的返回值（如果有）或预期抛出的异常。

```
@Mocked
DateUtil dateUtil;

@Test
public void testCurrDateStr_mockExpectations() {
    new Expectations() {
        {
            DateUtil.now(); 期望 mock 对象发生的行为
            result = mockCalendar().getTime(); 期望返回值
        }
    };
}

public static Calendar mockCalendar() {
    Calendar cal = Calendar.getInstance();
    cal.set(2010, 1, 12, 19, 58, 55);
    return cal;
}
```

上面的例子，定义了一个 mock 项 DateUtil，同时在 Expectation 中定义了预期会被调用的方法 now，以及 now 方法的返回值。

在 jmockit 中，除了提供了精确的 Expectations 外，还提供了 NonStrictExpectations。

Expectations: 在 expectation 中指定的行为一定会发生，同时发生的顺序跟定义的顺序是一致的，在 expectations 中没有定义的行为是不允许发生的。任何不期望的调用将会导致测试的失败。

NonStrictExpectation: 在 NonstrictExpectations 中指定的行为未必一定要被调用到，同时顺序也无所谓，同时如果测试方法调用到的方法在 NonstrictExpectations 没有定义的方法，那么会使用一个默认的实现来替代。

声明和使用 mock 对象

Mock 对象可以是测试类或 Expectation 的一个变量，也可以是测试方法的一个参数；但无论以在什么地方声明，被声明的 Mock 对象前面都必须有 @mockit.Mocked 或 @mockit.NonStrict 注释符。

在 mockit.Expectations 中的例子演示了如何在测试类中声明一个 mock 对象的变量。下面的 2 个例子，分别演示了如何在测试方法的参数中和 Expectations 匿名类中声明 mock 对象。

```

public void testCurrDateTimeStr_MockitExpectation_returnSequence(
    @Mocked DateUtil dateUtil) {
    new NonStrictExpectations() {
        {
            DateUtil.now();
            returns(getMockDate(), mockCalendar(2013, 2, 12).getTime());
        }
    };
}

public void testCurrDateTimeStr_MockitExpectation2() {
    new NonStrictExpectations() {
        @Mocked
        DateUtil dateUtil;
        {
            DateUtil.now();
            returns(mockCalendar(2015, 6, 25).getTime());
        }
    };
}

```

@Mocked 和 @NonStrict 的区别是一个是严格定义的 Mock 行为，一个是非严格定义的。其意义同上节的 Expectations 和 NonStrictExpectations。

Mock 对象的约束

调用次数约束

参数约束

返回值的指定

在 expectations 中指定返回值很简单，只要在指定的行为后面，紧跟后面设置 result=你预期返回值就可以了，也可以使用 returns(你预期的返回值)。这 2 种用法在多数情况下效果是一样的，唯一的区别是对 Exception 的处理。

result=Exception 实例 :将在预期行为中抛出异常。

returns(Exception 实例) :将在预期的行为中返回异常值。

使用 Delegate 来指定行为的结果和捕获参数

Delegate 是个空的接口类，用来告诉 jmockit 在运行时用来代理对应方法的一个代理类。示例如下：

```

public void testCurrDateTimeStr_Delegate() {
    new Expectations() {
        {
            DateUtil.now();
            result = new Delegate() {
                public Date now() {
                    Calendar cal = mockCalendar(2011, 1, 27);
                    return cal.getTime();
                }
            };
        }
    };
    String str = DateUtil.currDateStr();
    want.string(str).isEqualTo("2011-01-27");
}

```

如果方法有参数，那么我们也可以对方法的参数进行断言，如下面的代码片段所示：

```

new Expectations() {
    @Mocked DependencyAbc abc;
    {
        abc.intReturningMethod(anyInt, null);
        result = new Delegate() {
            // The name of this method can actually be anything.
            int intReturningMethod(int i, String s) {
                assertTrue(i > 0);
                assertEquals(i == 1 ? "one" : "other", s);
                return i + 1;
            }
        };
    }
};

```

在 `delegate` 中指定的方法，只需要参数类型跟被代理的函数一致就可以，方法的名称并不强制要求一样。但是，如果 `Delegate` 中存在 2 个方法，而且参数类型和被代理函数都一样，那么就要求方法的名称必须是一致的。

```

public void testCurrDateTimeStr_Delegate() {
    new Expectations() {
        {
            DateUtil.now();
            result = new Delegate() {
                public Date now() {
                    Calendar cal = mockCalendar(2011, 1, 27);
                    return cal.getTime();
                }

                public Date now2() {
                    Calendar cal = mockCalendar(2011, 1, 28);
                    return cal.getTime();
                }
            };
        }
    };
    String str = DateUtil.currDateStr();
    want.string(str).isEqualTo("2011-01-27");
}

```

在上面的例子中，存在 2 个方法参数类型都一样，那么名称一样的将被执行。如果不存在名称一样的将抛出找不到对应方法的异常。

java.lang.IllegalArgumentException: No compatible method found: now()

注意，方法的返回值不是方法签名的一部份，因此返回值是可以任意类型的，jmockit 会执行类型的强制转换。如果类型转换错误，将会抛出一个 ClassCastException。

Delegate 的子类可以在外部定义好，然后全局公用。

```

public void testCurrDateTimeStr_Delegate2() {
    new Expectations() {
        {
            DateUtil.now();
            result = new MyDateUtilNowDelegate();
        }
    };
    String str = DateUtil.currDateStr();
    want.string(str).isEqualTo("2311-01-27");
}

public static class MyDateUtilNowDelegate implements Delegate{
    public Date now() {
        Calendar cal = mockCalendar(2311, 1, 27);
        return cal.getTime();
    }
}

```


Fluent 语法

部分 mock 行为

当我们使用@Mocked 或@NonStrict 注解时，默认情况下，整个对象都会被 mock 的；但有时，我们可能希望只模拟这个对象中的部分方法，其它方法按照正常的业务逻辑运行。在基于状态测试章节中，我们介绍了如何使用@MockClass 和 MockUp 来模拟部分行为，那么在@Mocked 注解中又如何做到部分模拟呢？

假定我们有业务类 Collaborator

```
public class Collaborator {
    final int value;

    Collaborator() {
        value = -1;
    }

    Collaborator(int value) {
        this.value = value;
    }

    int getValue() {
        return value;
    }

    final boolean simpleOperation(int a, String b, Date c) {
        return true;
    }

    static void doSomething(boolean b, String s) {
        throw new IllegalStateException();
    }
}
```

@Mocked 有 2 个重要的属性，`methods` 和 `inverse`，我们可以显式的在 `methods` 属性中列出我们需要 mock 的方法。如下例：

```

@Test
public void staticPartialMockAClass() {
    new Expectations() {
        @Mocked(methods = "getValue")
        Collaborator collaborator;
        {
            when(new Collaborator().getValue()).thenReturn(123);
        }
    };
    // Mocked:
    Collaborator collaborator = new Collaborator();
    want.number(collaborator.getValue()).isEqualTo(123);

    // Not mocked:
    want.bool(collaborator.simpleOperation(1, "b", null)).isEqualTo(true);
    want.object(new Collaborator(45).value).isEqualTo(45);
}

```

上面 mock 变量 collaborator 是个匿名类属性，它显式指定了 getValue 会被 mock，其他方法将调用正常的业务类方法。

```

@Test
public void staticPartialMocking()
{
    new Expectations()
    {
        @Mocked({"(int)", "doInternal()", "[gs]etValue",
"complexOperation(Object)"})
        SomeBiz someBiz;
        {
            //mock 的行为
        }
    };
    //测试
}

```

上面的例子中，someBiz 方法中符合下列情形之一的都会被 mock：有一个参数且为 int 类型的方法；doInternal（）方法；getValue 和 setValue 方法以及 complexOperation(Object o) 方法。

Inverse 属性默认情况为 false，如果为 true，表示只有 methods 中显式列出的方法不被 mock，其它方法将会被 mock。

这种方法，我们需要显式的在 @Mocked 中指定需要被 mock 或不被 mock 的方法，一种方便的方法是让框架来判断哪些方法需要被 mock，而哪些方法不需要被 mock。Jmockit 提供了这么一个方便之门，你可以在 Expectations 匿名类的构造参数中指定那些需要被动态 mock 的类和变量。这样的话，在 Expectations 中指定的行为将被 mock，不在 Expectations 中指定的行为将不被 mock。

```

@Test
public void dynamicallyMockAClass() {
    new Expectations(Collaborator.class) {
        {
            when(new Collaborator().getValue()).thenReturn(123);
        }
    };
    // Mocked:
    Collaborator collaborator = new Collaborator();
    want.number(collaborator.getValue()).isEqualTo(123);

    // Not mocked:
    boolean ret = collaborator.simpleOperation(1, "b", null);
    want.bool(ret).isEqualTo(true);
    want.object(new Collaborator(45).value).isEqualTo(45);
}

```

在上面的例子中，匿名类 `Expectations` 有一个参数 `Collaborator.class`，那么 `jmockit` 就会认为在匿名类中显式声明的 `Collaborator` 行为将会被 mock，而那些没有声明的行为将调用原始的业务方法。在例子中，我们显式声明了 1 个方法 `getValue` 行为，所以只有这个方法会被 mock（也就是返回 123 这个值），其它没有声明的 `simpleOperation`，以及属性都将按原来的逻辑运算。

Jmockit 和 spring 的集成

@MockedBean

在 `jmockit` 和 `spring` 的集成和原有的 `jmock` 和 `spring` 的集成差不多，但要多写一个注解。

```

@SpringBeanByName
private SpringBeanService springBeanService1;

@MockedBean
@Mocked
protected SomeInterface dependency2;

```

`@MockedBean` 的作用是在 `spring` 容器中初始化一个名称为 `dependency2` 的 bean，它没有任何依赖项，`@mocked` 的作用同前面所说的。当真正调用到 `dependency2` bean 的方法时，框架会把它转向到 `testcase` 的属性 `dependency2` 上。

使用 `@MockedBean`，你甚至不需要在 `spring` 文件中定义具体的 bean，`@MockedBean` 会自动的帮你往容器中注入一个 Mock bean 对象。这个特性在测试中会减少你配置 `spring` 文件的工作量。假设我们有下面这个 `spring` 配置文件

```

<beans xmlns="..." default-autowire="byName">
    <bean id="userService"
class="org.jtester.fortest.service.UserServiceImpl"/>
</beans>

```

```

@Test(groups = { "jtester", "mock-demo" })
@SpringApplicationContext( { "org/jtester/fortest/spring/beans.xml"
, "org/jtester/fortest/spring/data-source.xml" })
public class MockBeanTest extends JTester {
    @SpringBeanByName
    private UserService userService;

    @MockedBean
    @Mocked
    private UserDao userDao;
    //...
}

```

userService 依赖 userDao 这个 bean，那么在 spring 容器起来后，框架会自动创建一个默认的 userDao 这个 bean，并且会自动被注入到 userService 中。

Mock 部分 springbean 行为

上节的方法对于全部 mock 来说是比较合适的，特别是有些 bean 要预先初始化很多内容（比如 ResourceManager 之类），@MockBean 会阻止 spring 去做那些很费时间的初始化工作。但如果只想部分 mock spring bean，那上面的做法就不适用了。

想 mock spring bean 的做法有下列几种办法。

1、基于状态的测试

```

@SpringBeanByName
private UserService userService;

@SpringBeanByName
private UserDao userDao;

@DbFit (when = "MockitSpringBeanTest.wiki")
public void paySalary() {
    Mockito.redefineMethods (UserDaoImpl.class, MockUserDao.class);

    //mock的行为
    double total = this.userService.paySalary("310000");
    want.number (total).isEqualTo(43000d);

    //not mock的行为
    List<User> users = userDao.findAllUser();
    want.number (users.size()).isEqualTo(2);
    Mockito.restoreAllOriginalDefinitions();
}

public static class MockUserDao {
    public List<User> findUserByPostcode(String postcode) {
        return getUserList();
    }
}

```

在测试开始前重定义要 mock 的 UserDaoImpl 方法。

2、基于行为的测试

```

@SpringBeanByName
private UserService userService;

@SpringBeanByName
private UserDao userDao;

@DbFit(when = "MockitSpringBeanTest.wiki")
public void paySalary_mockBean() {
    new Expectations() {
        @Mocked(methods = "findUserByPostcode")
        UserDaoImpl userDaoImpl;
        {
            when(userDaoImpl.findUserByPostcode(anyString)).thenReturn(getUserList());
        }
    };
    double total = this.userService.paySalary("310000");
    want.number(total).isEqualTo(4300d);

    List<User> users = userDao.findAllUser();
    want.number(users.size()).isEqualTo(2);
}

```

使用匿名类属性或参数来动态改变 UserDaoImpl 的 mock 方法。

但以上两种办法都没有办法阻止 spring 容器去调用部分 mock 的 bean 去执行初始化方法（如果有的话）。

访问私有的方法，构造函数或变量

私有属性访问：setField(tested, "someIntField", 123);

私有方法访问：invoke(tested, "intReturningMethod", 45, "");

私有构造函数访问：newInstance("some.package.AnotherDependency", true, "test");

@Inject 注解

@Inject 的作用是在测试开始前，把特定的对象（比如 mock 对象）注入到测试类的变量中，示例如下：

```

@Test(groups = "jtester")
public class MockTest_ByName extends JTester {

    private SpringBeanService springBeanService = new SpringBeanServiceImpl();

    @Inject(targets = "springBeanService", properties = "dependency1")
    @Mocked
    private SomeInterface someInterface1;

    @Inject(targets = "springBeanService", properties = "dependency2")
    @Mocked
    private SomeInterface someInterface2;
}

```

Inject 属性	作用
targets	表示这个对象要注入到测试类的哪个属性中，jtester 会在测试方法中自动查找字段名称一致的变量。比如上面的例子中 targets = "springBeanService"，Jtester 就会在 MockTest_ByName 这个测试类中查找名称为"springBeanService"的变量。targets 可以跟多个字段名称，targets= {"springBeanService1","springBeanService2"}就可把对象注入到 2 个变量中。
properties	表示 mock 对象要注入到被测对象（targets 所指定的对象）的那个属性上。在上面的例子中表示 someInterface1 这个 mocked 对象被注入到 springBeanService 的 dependency1 这个属性上。 如果 properties 这个属性为 null 或者=""，那么表示 mock 对象是按类型方式注入到被测对象中，注意：如果按类型方式注入，则被测对象中不能有多于一个 mock 类型的属性。 properties 同样也支持多个属性名称，分别对应 injectInto 中指定的被测对象。

数据库测试

jtester.properties 文件配置

jtester.properties 是 jtester 的运行配置文件，是放在 classpath 的根目录下，要使用 jtester 的数据库测试功能，必须在 jtester.properties 文件中指定下列配置项。

配置项	说明
database.type	数据库测试类型
database.driverClassName	数据库驱动的 classname
database.url	数据库连接 url
database.userName	用户名
database.password	密码
database.schemaNames	Schema 名称，如果有多个 schema 用 “;” 分割
DatabaseModule.Transactionall. value.default=commit	事务的默认管理方式，默认是提交(commit)，可选值还有回滚(rollback)，失效(disabled)
database.onlytest=true	是否只能连接本地数据库和以test开头或结尾的数据库
spring.datasource.name=dataSource	使用 jtester 中的数据库配置替换 spring 中 id=\${spring.datasource.name} 的 datasource bean 的数据库配置 jtester 默认 spring 中 dataSource bean 的配置为

	<pre><bean id="dataSource" class="..."> ... </bean></pre>
<code>dbMaintainer.disableConstraints.enabled=true</code>	是否解除数据库的外键约束和not null约束

示例

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

`database.onlytest=true`, 是否只能连接本地数据库或者以test开头或结尾的测试数据库。

推荐单独使用测试数据库, 而不要使用开发数据库作为测试库使用, 因为测试要求数据是可重现的, 这样必然要求要对数据库有完成的控制权, 特别是要经常清空数据进行重新准备数据。如果测试库和开发库合一的话, 必然要相互干扰, 要么测试的时候把开发要用的数据给清空了, 要么是开发的数据影响到了测试程序的运行准确性。

`dbMaintainer.disableConstraints.enabled=true`, 是否解除数据库的约束条件(外键和not null约束)

解除数据库的约束条件对于测试来说有很重要的作用, 外键的解除, 让我们删除和准备数据更加便捷; not null约束的解除可以让我们聚焦于我们感兴趣的数据项, 而不必关心跟本测试无关的数据项。一句话, **解除约束可以让我们测试更自由**。

另外为了使大家使用dbFit插件更方便, 可以在jtester.properties中添加一条配置用来指定数据库连接的driver jar包路径。

```
database.driverJar=D:/alibaba/repository/project/headquarters/jdbc/proxy/headquarters-jdbc-proxy-1.1.jar;D:/alibaba/antx/repository/Jakarta/commons/logging/commons-logging-1.1.jar;D:/alibaba/antx/repository/jdbc/oracle/ojdbc14.jar
```

使用@DbFit 功能来进行数据库测试

Annotation DbFit 有 2 个属性, when 和 then, 它们的值都是 wiki 文件数组, 示例如下

```
@DbFit(when = { "testcase.when.wiki" }, then = "testcase.then.wiki")
public void testcase() {
    //.....
}
```

在 testcase 执行之前, jtester 框架会先执行 when 里面指定的内容, when 通常用于准备测试所用的数据。在 testcase 执行之后, jtester 框架会执行 then 里面指定的内容, then 通常

用于验证单元测试执行后的数据状态。虽然 `when` 和 `then` 在功能上有上面的所说的区别，但在语法上都是一样的。

@DbFit 的 wiki 语法

Dbfit 是使用 wiki 来准备数据库测试数据和验证数据的，你可以在 `eclipse` 中直接进行编辑，其效果如下图所示：

```
|connect|
|clean table|tdd_user|

|insert|tdd_user| |
|id|post_code|salary|
|1|310000|1000|
|2|310000|1200|
|3|352200|1500|
|4|310000|1800|
|5|DddDdd|2300|

|commit|
```

为了友好一点，你可以使用可视化的查看工具查看，`eclipse` 插件下载地址：

<http://eclipsewiki.sourceforge.net/>

其可视化效果图如下：

connect	
clean table	tdd_user

insert	tdd_user	
id	post_code	salary
1	310000	1000
2	310000	1200
3	352200	1500
4	310000	1800
5	DddDdd?	2300

commit

正如下图所示，dbfit 必须将命令和数据放在 wiki 表格中，这样 `jtester` 才可以识别和执行相应的命令。Wiki 的表格语法是以 `|` 作为行开头，且内容是也是以 `|` 分割的。

`|cell1|cell2|cell3|`定义了表格的一行数据，其第一列数据内容为“`cell1`”，第二列是“`cell2`”，第三列是“`cell3`”。

数据库连接

Jtester 连接数据库非常的简单，只需用一条命令

```
connect
```

这样 jtester 就会默认使用 jtester.properties 中的数据库配置去连接对应的数据库。当然，如果你的测试程序中使用到多个数据库，你也可以在 connect 命令后指定要连接的数据库。

```
|Connect | SERVICE_NAME | USER_NAME | PASSWORD | DATABASE_NAME|
```

如果你使用的是非标准的属性值指定的方式进行连接，而要直接使用一条 connection url 进行连接的话，也可以这样使用。

```
|Connect | a single connection URL |
```

注意：在 dbfit 的所有 wiki 文件中，在使用数据库功能前，都要指定 connect 命令。

查看执行结果

Jtester 在执行 dbfit 模块时，会在项目的目录下生成一个 target/dbfit 文件夹，用于存放 dbfit 执行的结果。其文件名称和 when 和 then 中指定的名称加上 package 包名称。这个文件中会详细的列出 dbfit 的执行状态和异常信息）。

假如你的 wiki 文件中没有以 |connect|命令开头，那么结果文件中就会显示如下异常

```
java.lang.RuntimeException: java.lang.IllegalArgumentException: No open connection to a database is available. Make sure your database is running and that you have connected before performing any queries.
```

如果你的测试正常通过，那么你的结果文件中会有很多绿色的信息。

connect	
query	select * from tdd_user where post_code='320001'
post_code	sarary
320001	23.02

清空表数据

在单元测试中，为了保证每个 `testcase` 的独立性，`case` 的数据必须不受其他因素的干扰，保证初始环境的可重复性。这样就要求在准备数据前，要把相应表中的数据清空。`dbfit` 模块提供了一条简单的命令来清空多张表数据。

```
| clean table | table1;table2;table3 |
```

就这样，`jtester` 会将 `table1,table2,table3` 这 3 中表中的所有数据都清空。多张表名称可以用“;”或“,”进行分割。

查询数据（Query）

Query 语法格式

query	<code>select * from tdd_user where post_code='320001'</code>
post_code	salary
320001	23.02
320001	23.03

第一行第一列是关键字 `Query`
第一行第二列是一条具体的可执行的 `SQL` 语句，他的任务是查询出你想要的数据。
第二行是你期望验证的数据列名称列表，（名称列表可以少于 `select` 中指定的列，其顺序也和 `select` 中指定的无关）。
第三行及其后行是你期望 `sql` 查询出的 `ResultSet` 中要包含的具体数据，数据的顺序是无关的。
结果信息显示，如果测试通过，`wiki` 文件中的期望数据会以绿色显示。

如果 `wiki` 中指定的期望数据多于查询出的数据，结果文件中会以红色显示多出的数据，并以 `missing` 字样标注

query	<code>select * from tdd_user where post_code='320001'</code>
post_code	salary
320001	23.02
320001 missing	23.03

如果 `wiki` 中缺少了查询出的数据，结果文件中同样以红色显示少掉的数据，并以 `surplus` 字样标注。

query	select * from tdd_user where post_code='320001'
post_code	salary
320001 missing	23.04
320001 surplus	23.02

技巧: query 后面指定的 table 数据只能比较字符串是否相等, 没有比较大小等其他功能, 如果你的测试有这类的比较, 如果 count 小于某个数值等等, 可以在 select 语句中做比较, 在验证数据中做 true/flase 等判断。

有序查询 (ordered query)

Query 查询的数据比较是顺序无关的, 如果你需要严格验证数据的顺序, 可以使用 ordered query 命令。其语法结构如下:

```
|ordered query | select * from ... | |
|field1|field2|...|
|value11|value12|...|
|value21|value22|...|
|.....|
```

其结构和 query 命令式一致的, 唯一的区别是 value 的行是有顺序关系的。

参数化查询

在 dbfit 模块中是可以设置参数的, 其语法格式如下

```
| set parameter | parameter name | parameter value |
```

第一列是关键字 set parameter, 第二列是参数的名称, 第三列是参数值。
参数的引用很简单, @parameter, 放在要使用的地方就可以了。

query	select count(*) > 1 as result from tdd_user where post_code=@para
-------	---

插入数据（Insert）

Insert 命令式一般是用来准备测试数据的，其语法结构如下所示

insert	tdd_user	
id	post_code	salary
1	310000	1000
2	310000	1200

第一行第一列是关键字 Insert

第一行第二列是要插入数据的表

第二行是数据表中对应的字段名称，（列可以少于表中的字段，按需插入数据）

第三行及其后行是要插入的数据值。

执行 SQL 语句(Execute)

Execute 可以执行任何的 SQL 语句，其语法结构：

|execute |delete from table where|

或者

|execute| update table set wherer|

commit&rollback

dbfit 模块默认情况下更改的数据是回滚的，如果要使更改的数据生效，必须的 wiki 文件末尾显式的加上 commit 命令。

commit

其它命令

执行存储过程

Update 命令

NULL 值

事务管理

在 jtester 中, dbfit 和 spring 可以运行在同一个事务中, 默认情况下, jtester 的事务是 commit 方式的, 存在下列情况可以改变 jtester 中事务的运行。

- 1、在 dbfit 的 wiki 文件中有显式的 commit/rollback 命令。

Insert	tdd_user		
first_name	last_name	post_code	salary
darui.wu	wu	310012	1545.0

commit

- 2、Spring 管理的 dao/service 中存在 commit/rollback 的情况。
- 3、测试方法或测试类显式的改变了 transaction 的类型。

```
@Test
@Transactional(TransactionalMode.ROLLBACK)
@DbFit(when = "...", then = "...")
public void testDemo() {
    //.....
}
```

`@Transactional(TransactionalMode.COMMIT)` 最后commit数据

`@Transactional(TransactionalMode.ROLLBACK)` 最后rollback数据

`@Transactional(TransactionalMode.DISABLED)` jtester框架不在使用spring的事务管理, 程序员必须显式的在dbfit文件中自己commit或rollback数据。但spring bean仍然接受spring transaction的管理

- 4、Jtester.propeires 修改默认值, 可选值有(commit,rollback,disabled)
`DatabaseModule.Transactional.value.default=commit`

Martini 项目下的 ibatis 文件配置

要进行数据库测试必然涉及到 spring 文件和 ibatis 文件的配置, 这些文件管理的需要遵循以下一些原则。

- 1、按需加载原则, 测试的一个准则是速度要快, 很多 spring 文件和 ibatis 文件的加载是很耗费时间的, 不要把一些你根本用不到的文件也加载进来。
- 2、文件模块化原则, spring bean 的配置要尽可能按照功能模块的方式来划分, 不要把所有的 spring bean 不加分析的写到一个文件中。配置文件也是代码, 也需要我们进行重构, 文件太大了就要拆分。
- 3、不要重复原则, 在测试中能复用生产的配置文件时, 一定要复用。不要拷贝复制一

份，这样做会导致后期的维护成本很高。
下面我们具体解析一下如何在 martini 中配置 spring 和 ibatis 文件时做到遵循这些原则。

数据库测试 FAQ

问题：为什么 dbfit 文件中 2 条 execute 语句只执行了一条，而第二条没有被执行。

答案：DbFit 是基于表格来解析命令的，一条命令一个表格，如果你有多条 execute 语句，那么必须在 execute 语句之间加空行。

```
|execute|delete from schedule_task where id in ('1','2','3','4','5','6')|  
  
|execute|delete from job_task where id = 2|
```

问题：spring 文件中指定了多个事务时，运行测试抛出异常

`org.unitils.core.UnitilsException: Found more than one bean of type Platform
TransactionManager in the spring ApplicationContext for this class`

原因：当 spring 容器中存在多个 transaction 时，测试框架不知道应该用哪个事务来管理测试的事务，这个情况下只能在 test 前面加上 annotation

`@Transactional(TransactionalMode.DISABLED)`，取消测试方法的事务管理。

Spring 和 SQL 跟踪

升级 jtester 版本到 0.8.8 以上，在 jtester.properties 中加 2 项配置

`tracer.database=true`

`tracer.springbean=true`

这样测试在运行的过程中就会自动生成 spring 接口调用的序列图，以及记录 SQL 语句。

生成的信息记录在 target/tracer 目录对应 package 下面的 html 文件中。

test-class call schedulingBo #publishForCcbu	
paras	{"com.ali.martini.dal.dataobject.rps.EsbRpsPublish":{"id":300001819,"gmtCreate":"2009-12-23 11:23:11"}}
result	{"com.alibaba.biz.command.result.ResultSupport":{"success":false,"models":[{"entry":{"string":["_def"]}}
schedulingBo call defaultDao #getObj	
paras	{"string":"ESB_RPS_PREORDER"} {"int":300004398}
result	{"com.ali.martini.dal.dataobject.rps.EsbRpsPreorder":{"id":300004398,"gmtCreate":"2009-06-17 00:00:00 CST","creator":"zhoulili","modifier":"hzzzz","isDeleted":"n","defaultBiz":true,"prodInstanceId":104223,"gmtPublishEnd":"2009-07-17 00:00:00.0 CST","gmtPreorderBegin":"2009-06-17 00:00:00.0 CST","period":30,"periodUnit":"D","preorderType":"normal","isFrequenter":"n","preorderOrigin":"ni","adurlSetting":"n","memberId":"fenzhicheshi1","resourceType":"tp_ad","salesId":"zhoulili"}}
schedulingBo call apiHubBo #getInstanceById	
paras	{"int":104223}
result	{"com.ali.martini.dal.dataobject.rps.EsbRpsInstance":{"defaultBiz":true,"product":"bc414","keywords":null}}
apiHubBo call defaultDao #getObj	
paras	{"string":"ESB_RPS_INSTANCE"} {"int":104223}
spring-bean	SQL-Statement
defaultDao	SELECT ID , GMT_CREATE , CREATOR , GMT_MODIFIED , MODIFIER , IS_DELETED , PROD_INSTANCE_ID , CUSTOMER_ID , LEADS_ID , PAY_TYPE , STATUS , CANCEL_REASON , GMT_PUBLISH_BEGIN , GMT_PUBLISH_END , GMT_PREORDER_BEGIN , GMT_PREORDER_PERIOD , PERIOD_UNIT , PREORDER_TYPE , IS_FREQUENTER , GMT_BUFFER_DATE , PREORDER_ORIGIN , PREVIOUS_PREORDER_ID , REMARK , HOLD_FLAG , GMT_HOLD_DATE , IMAGE_URL , ADURL_SETTING , MEMBER_ID , RESOURCE_TYPE , SALES_ID FROM RPS_INSTANCE WHERE id = ? and is_deleted='n'
defaultDao	SELECT ID , GMT_CREATE , CREATOR , GMT_MODIFIED , MODIFIER , IS_DELETED , PROD_INSTANCE_ID , ATTRIBUTE1 , ATTRIBUTE2 , ATTRIBUTE3 , ATTRIBUTE4 , ATTRIBUTE5 , ATTRIBUTE6 , ATTRIBUTE7 , ATTRIBUTE8 , ATTRIBUTE9 , ATTRIBUTE10 , ATTRIBUTE11 , ATTRIBUTE12 , ATTRIBUTE13 , ATTRIBUTE14 , ATTRIBUTE15 , INSTANCE_NAME , REMARK FROM RPS_INSTANCE WHERE is_deleted='n' and is_common='n'
defaultDao	SELECT COLUMN_NAME , PROD_ATTRIBUTE , REMARK FROM RPS_COLUMN_MAPPING WHERE is_deleted='n' and PRODUCT=?
defaultDao	SELECT PRODUCT FROM RPS_INSTANCE WHERE id = ? and is_deleted='n'
defaultDao	SELECT PRODUCT FROM RPS_INSTANCE WHERE id = ? and is_deleted='n'
defaultDao	SELECT ID , GMT_CREATE , CREATOR , GMT_MODIFIED , MODIFIER , IS_DELETED , PROD_INSTANCE_ID , ORDER_ID , ORDER_ITEM_ID , ORDER_NUMBER , ORDER_ITEM_NUM , STATUS FROM RPS_ORDER WHERE is_deleted='n' and PREORDER_ID=?

@Tracer

当个别测试方法有自己的跟踪策略时，可以在测试方法前面加@Tracer，@Tracer 有 3 个属性。@Tracer(spring = true, jdbc = true, info = TOJSON)

当不想拦截 spring 调用时可以设置 spring=false

不想记录 SQL 语句时，设置 jdbc = false

当参数对象转换为 json 对象有错误时，可以使用 `info=INFO.TOSTRING` 代替，这样参数记录就使用 `object.toString()` 函数了。

FAQ

为什么 spring 初始化时报下面类型无法转换的错误

```
Caused by: org.springframework.beans.TypeMismatchException: Failed to
convert property value of type [$Proxy13] to required type
[com.ali.b2b.crm.base.i18n.I18nImpl] for property 'i18n'; nested
exception is java.lang.IllegalArgumentException: Cannot convert value
of type [$Proxy13] to required type
[com.ali.b2b.crm.base.i18n.I18nImpl] for property 'i18n': no matching
editors or conversion strategy found
    at
    org.springframework.beans.BeanWrapperImpl.convertForProperty(BeanWrapperImpl.java:391)
    ... 63 more
```

这是因为你的 spring bean 不是基于接口编程

```
/**
 * @param i18n the i18n to set
 */
public void setI18n(I18nImpl i18n) {
    this.i18n = i18n;
}

public void setI18n(I18n i18n) {
    this.i18n = i18n;
}
```

你注入的是一个实现而不是一个接口，你应该把左边的写法改成右边的写法。

这个问题涉及的 spring 实现 AOP 代理的策略，你在使用 AOP 时指定使用的是 JDK Proxy 方式还是 CGLIB 代理方式，具体知识可以参考 [spring aop](#) 方面的内容。

JTester 插件的使用

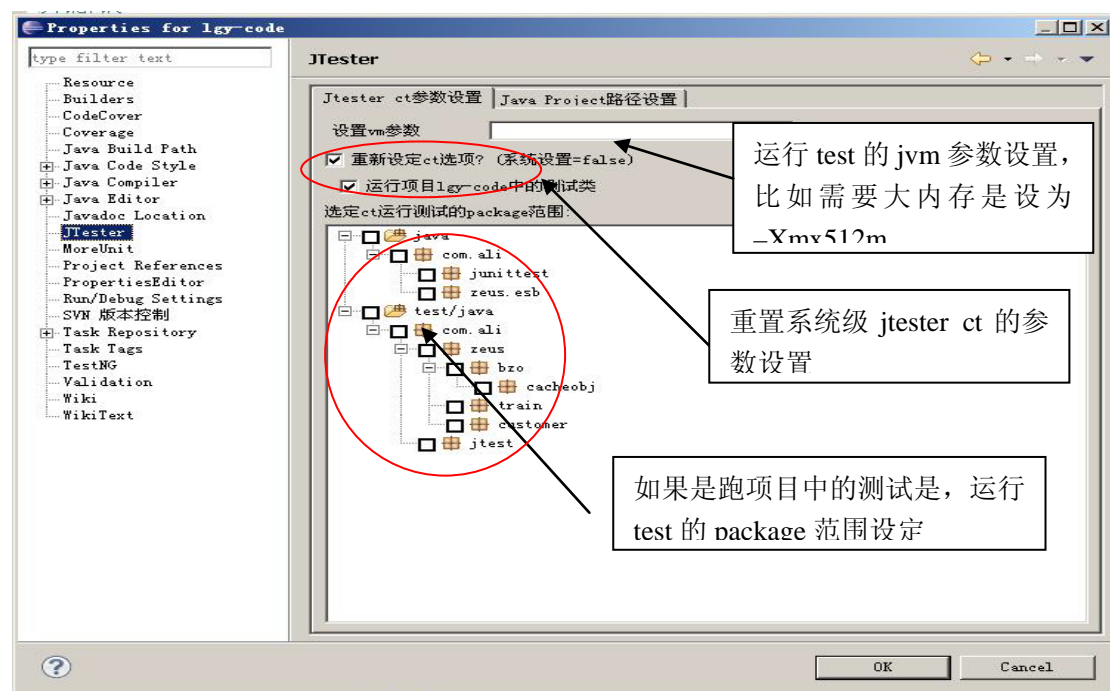
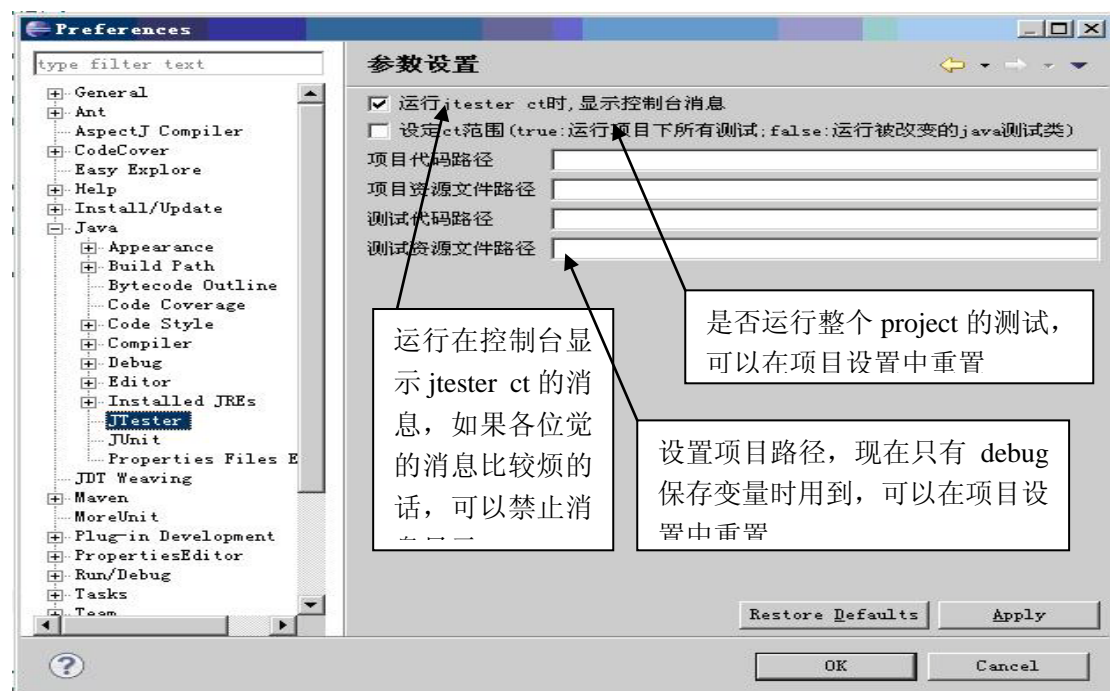
插件功能

- A、debug 模式下保存变量的值为 xml 文件。
- B、实现 IDE 模式下的持续测试功能，现在分为运行整个项目的测试和运行改变代码的测试。
- C、实现测试错误的在 eclipse ide 下以 marker 的方式标注出来。
- D、数据库测试，自动提取 dbfit 格式的数据

插件安装

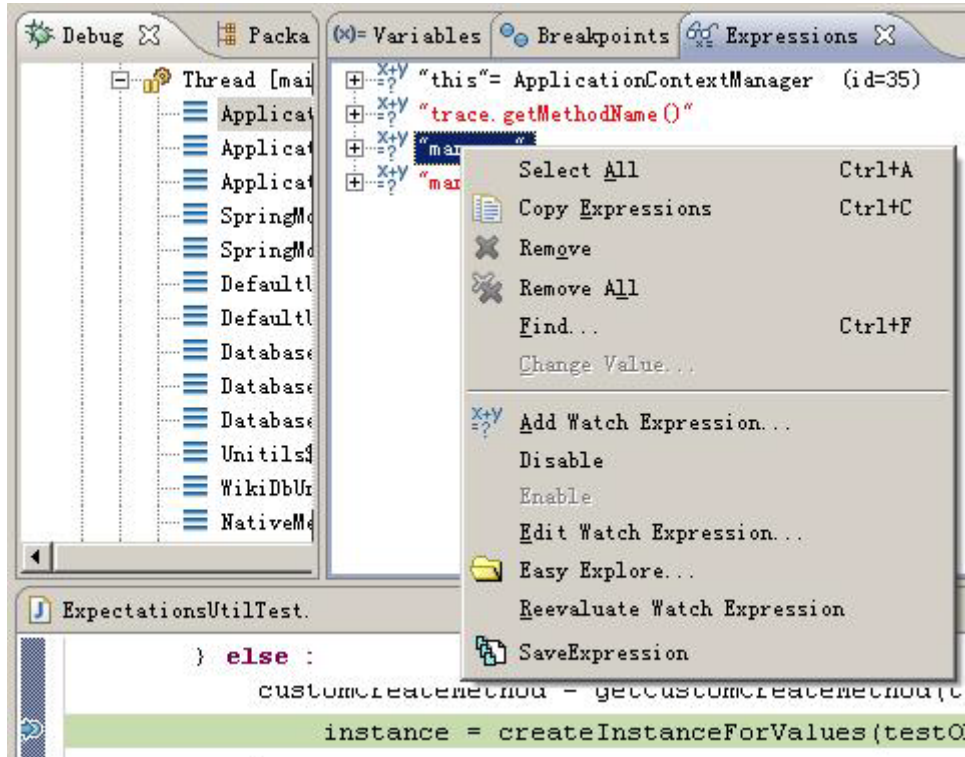
eclipse 插件的 update url: http://core-sys-dept.alibaba-inc.com:9999/eclipse_plugin/jtester 安

装完毕，在eclipse系统preference中有如下选项可以设置



录制变量的功能

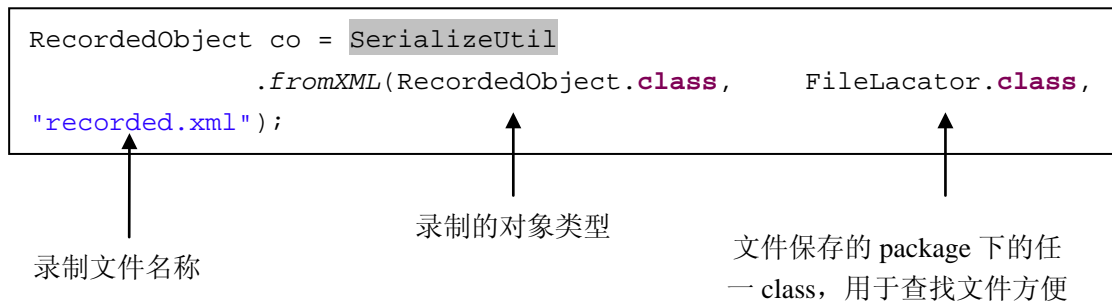
保存变量，在 debug 模式下，watch expression 或 variables 视图下，右键菜单多了个 Save Expression，如下图，单击就可以把变量转为 xml 保存到项目对应的 test resources 下相应的 package 路径下。



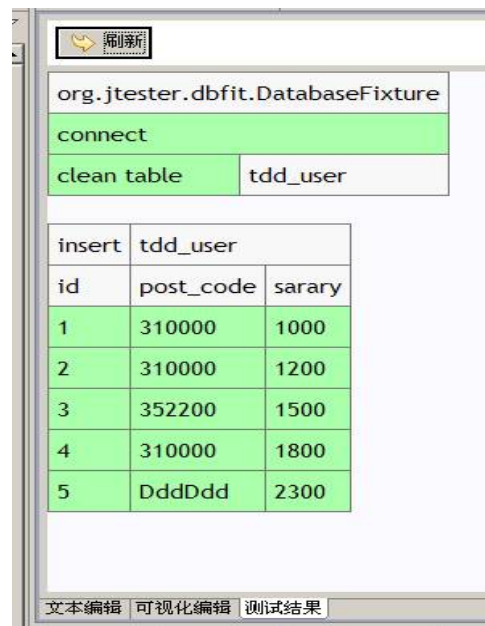
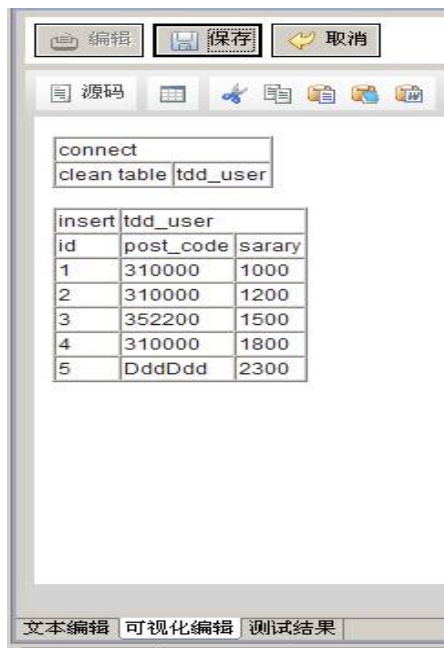
点击"Save Expression"就可以将变量的中保存到项目的 java.test 相应的 package 下，文件的名称是就是变量的名称，如果同名的文件已经存在，那么插件会自动给新保存的文件名称加上一个时间。

录制对象的使用，对象反序列化回来后，和普通变量的使用没有区别。

- 1、使用 api 方式反序列化录制对象
- 2、直接在 mock 对象中使用

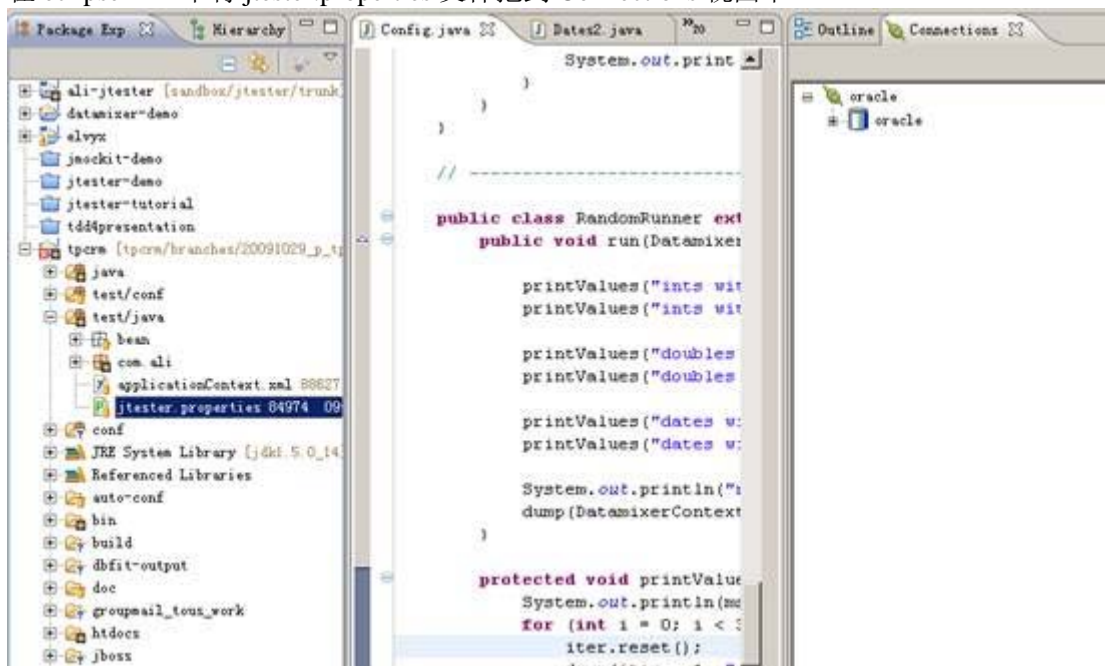


dbFit 插件编辑功能

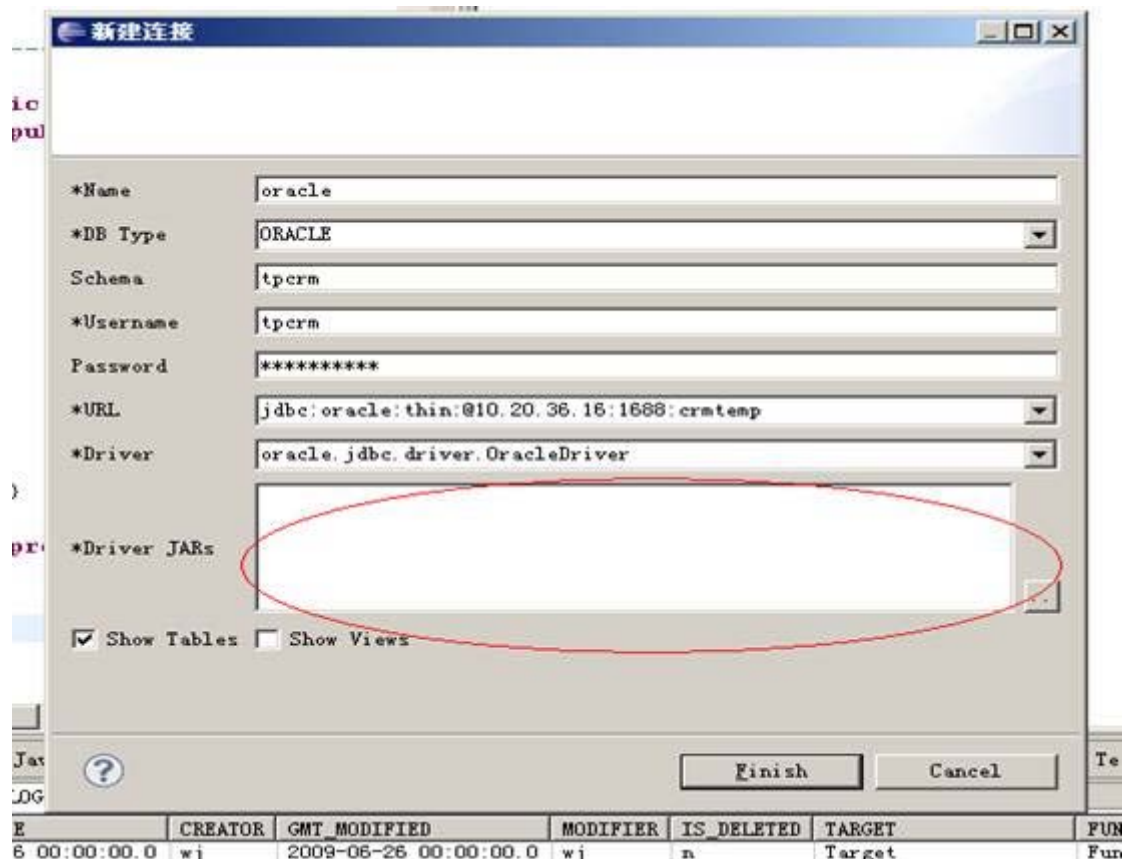


从数据库中直接拖取数据

在 eclipse IDE 中将 jtester.properties 文件拖到 Connections 视图中



插件会打开一个数据库连接框（当然也可以手动打开数据库连接框）

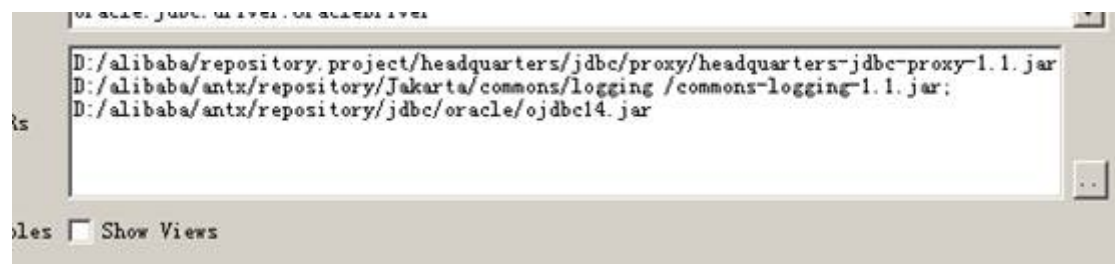


在 driver jars 输入所需要的 jar 包路径，对于我们的 oracle 连接需要以下 3 个包

D:/alibaba/repository.project/headquarters/jdbc/proxy/headquarters-jdbc-proxy-1.1.jar

D:/alibaba/antx/repository/Jakarta/commons/logging/commons-logging-1.1.jar;

D:/alibaba/antx/repository/jdbc/oracle/ojdbc14.jar

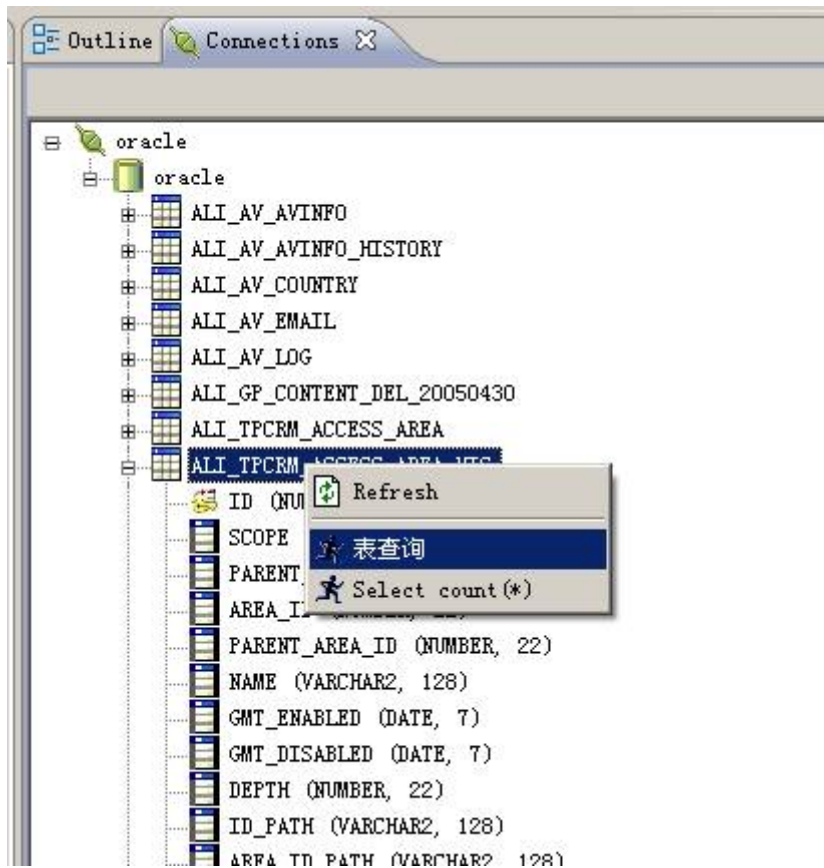


为了方便，大家也可以在 JTest.properties 中增加一条配置

```
database.driverJar=D:/alibaba/repository.project/headquarters/jdbc/proxy/headquarters-jdbc-proxy-1.1.jar;D:/alibaba/antx/repository/Jakarta/commons/logging/commons-logging-1.1.jar;D:/alibaba/antx/repository/jdbc/oracle/ojdbc14.jar
```

这样，连接框会自动解析 properties 文件列出 jar 的路径。

点击 finish 创建连接，展开表结构，可以在表或字段上右键菜单“表查询”



表数据有显示到 resultView 视图中，可以复制全表或选择数据行复制为 dbfit 格式的数据

ID	GMT_CREATE	CREATOR	GMT_MODIFIED	MODIFIER	IS_DELETED	TARGET	FUNCNAME	PARAMS
1620	2009-06-26 00:00:00.0	wj	2009-06-26 00:00:00.0	wj	n	Target	Funcname	Params
2106	2009-08-13 10:42:56.0	rs1	2009-08-13 10:42:56.0	rs1	n	productService	query	{null};queryProductDetail:(product_id:10
2107	2009-08-13 10:43:06.0	rs1	2009-08-13 10:43:07.0	rs1	n	productService	query	{null};queryProducts:(display=null, modi
2108	2009-08-13 10:43:09.0	rs1	2009-08-13 10:43:09.0	rs1	n	productService	query	{null};queryProductDetail:(product_id:10
2109	2009-08-13 10:43:25.0	rs1	2009-08-13 10:43:25.0	rs1	n	productService	query	{null};queryProducts:(display=null, modi
2110	2009-08-13 10:43:29.0	rs1	2009-08-13 10:43:29.0	rs1	n	productService	query	{null};queryProductDetail:(product_id:10
2111	2009-08-13 10:46:31.0	rs1	2009-08-13 10:46:31.0	rs1	n	productService	query	{null};queryProducts:(display=null, modi
2112	2009-08-13 10:46:45.0	rs1	2009-08-13 10:46:45.0	rs1	n	productService	query	{null};queryProductDetail:(product_id:10
2113	2009-08-13 10:49:31.0	rs1	2009-08-13 10:49:31.0	rs1	n	productService	query	{null};queryProducts:(display=null, modi
2114	2009-08-13 10:49:35.0	rs1	2009-08-13 10:49:35.0	rs1	n	productService	query	{null};queryProductDetail:(product_id:10
2115	2009-08-13 10:50:34.0	rs1	2009-08-13 10:50:34.0	rs1	n	productService	query	{null};queryProducts:(display=null, modi
2116	2009-08-13 10:50:40.0	rs1	2009-08-13 10:50:40.0	rs1	n	productService	query	{null};queryProducts:(display=null, modi
2117	2009-08-13 10:50:50.0	rs1	2009-08-13 10:50:50.0	rs1	n	productService	query	{null};queryProducts:(display=null, modi

然后就可以黏贴到 wiki 文件中了

```

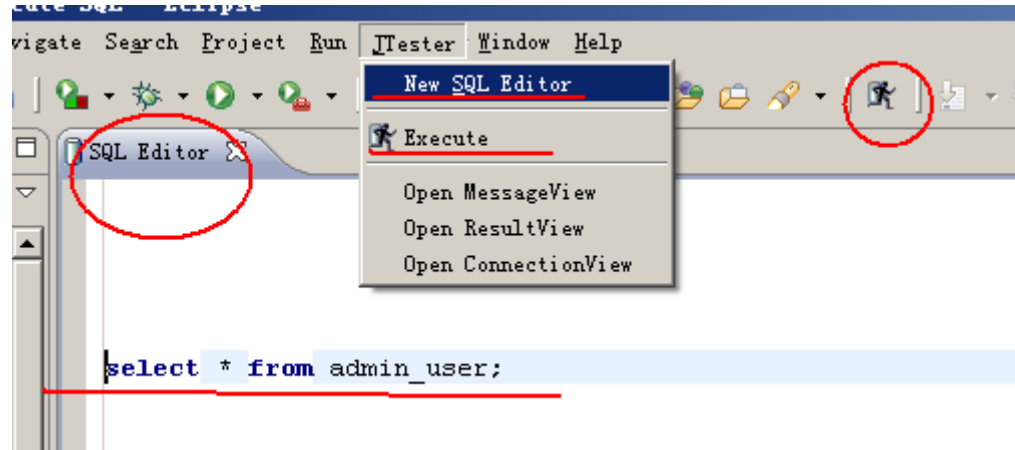
|connect|

|insert|sssl|
|ID|GMT_CREATE|CREATOR|GMT_MODIFIED|MODIFIER|IS_DELETED|TARGET|FUNC
|2106|2009-08-13 10:42:56.0|rs1|2009-08-13 10:42:56.0|rs1|n|product
organic matter:45-55%
Total nitrogen: 1.0% min, K2O: 18% min, P2O5: 6, null, null, level10
|2107|2009-08-13 10:43:06.0|rs1|2009-08-13 10:43:07.0|rs1|n|product
|2108|2009-08-13 10:43:09.0|rs1|2009-08-13 10:43:09.0|rs1|n|product
organic matter:45-55%
Total nitrogen: 1.0% min, K2O: 18% min, P2O5: 6, null, null, Model
|2109|2009-08-13 10:43:25.0|rs1|2009-08-13 10:43:25.0|rs1|n|product
|2110|2009-08-13 10:43:29.0|rs1|2009-08-13 10:43:29.0|rs1|n|product
organic matter:45-55%
Total nitrogen: 1.0% min, K2O: 18% min, P2O5: 6, null, null, Place
|2111|2009-08-13 10:46:31.0|rs1|2009-08-13 10:46:31.0|rs1|n|product
|2112|2009-08-13 10:46:45.0|rs1|2009-08-13 10:46:45.0|rs1|n|product
|2113|2009-08-13 10:49:31.0|rs1|2009-08-13 10:49:31.0|rs1|n|product
|2114|2009-08-13 10:49:35.0|rs1|2009-08-13 10:49:35.0|rs1|n|product
organic matter:45-55%
Total nitrogen: 1.0% min, K2O: 18% min, P2O5: 6, null, null, Model
|2115|2009-08-13 10:50:34.0|rs1|2009-08-13 10:50:34.0|rs1|n|product
|2116|2009-08-13 10:50:40.0|rs1|2009-08-13 10:50:40.0|rs1|n|product
|2117|2009-08-13 10:50:50.0|rs1|2009-08-13 10:50:50.0|rs1|n|product
|2119|2009-08-13 10:50:52.0|rs1|2009-08-13 10:50:53.0|rs1|n|product
|2120|2009-08-13 10:50:56.0|rs1|2009-08-13 10:50:57.0|rs1|n|product

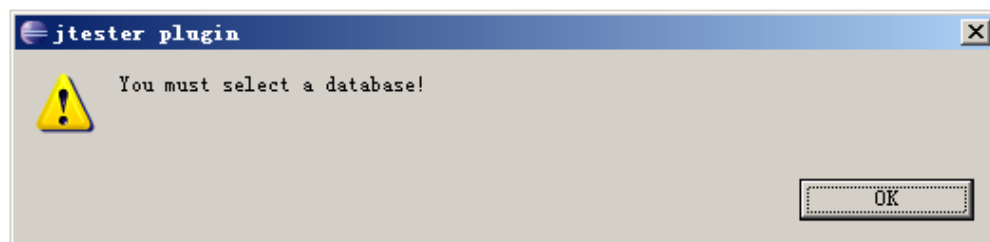
```

使用 SQL Editor 编辑器

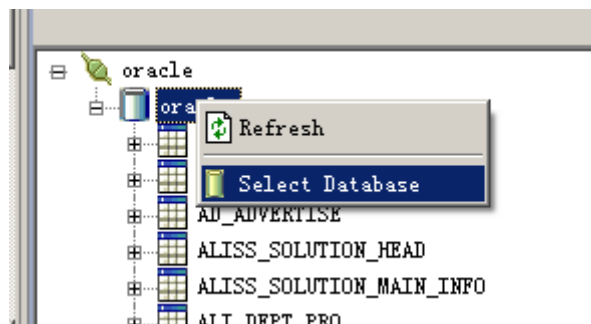
当使用默认 select * from 选取的数据没法满足你的需求时，你可以使用 SQL Editor 来查询并准备数据。



当执行对应的 sql 语句，结果同样会在 result view 中列出。
问题：为什么我执行 sql 语句时，弹出下列对话框



解决：因为你没有在 connection 视图中指定你 sql 语句执行的 schema，你可以在 connection view 的 schema 节点上右键菜单，选中一个默认的 schema。



覆盖率统计

将 jtester 升级到 0.9.0 以上版本，在项目的跟路径下建一个 instrument.xml，在文件中指定你需要覆盖的文件和不需要覆盖的文件。

instrument.xml，格式如下：

```
<instrument>
  <includes>
    <include>com/ali/martini/biz/av/remoteAv/personAv/impl/*.class</include>
    <include>com/ali/martini/biz/av/ao/impl/AvRemoteServiceAoImpl.class</include>
    <include>com/ali/martini/biz/av/util/HandleTimer.class</include>
  </includes>
  <excludes>
    <exclude>**/*Test.class</exclude>
  </excludes>
</instrument>
```

Instrument xml 文件的根元素 instrument, 它包含 2 个子节点, includes 和 excludes。Includes 指定你要覆盖到的文件, exclude 指定你要显式排除掉的文件。

不管 include 还是 exclude 的文件的后缀都是.class, 而不是.java

Martini 项目是采用 antx 来运行测试的, 所以只要增加这个文件上传到 svn, 集成服务器中会自动只覆盖到指定的文件。对于采用 ant 的项目, 需要在 ant 的 build 文件中修改 cobertura-instrument 任务, 其配置应该修改如下:

```
<cobertura-instrument datafile="${basedir.target}/cobertura.ser"
  classpathref="instrument.path" todir="${basedir.target}/instrumented-classes" >
  <instrument dir="${basedir}" includes="instrument.xml"/>
  <fileset dir="${basedir.target}/classes">
    <exclude name="**/*Test.class" />
  </fileset>
</cobertura-instrument>
```