[mockito简单教程](http://blog.csdn.net/sdyy321/article/details/38757135)

2014-08-24 13:50 10378人阅读 [评论](http://blog.csdn.net/sdyy321/article/details/38757135/#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sdyy321/article/details/38757135/#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

java util（2） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

官网： [http://mockito.org](http://mockito.org/)

API文档：http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html

项目源码：https://github.com/mockito/mockito

首先添加maven依赖

**[html]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. **<dependency>**
2. **<groupId>**org.mockito**</groupId>**
3. **<artifactId>**mockito-all**</artifactId>**
4. **<version>**1.9.5**</version>**
5. **<scope>**test**</scope>**
6. **</dependency>**

当然mockito需要junit配合使用

**[html]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. **<dependency>**
2. **<groupId>**junit**</groupId>**
3. **<artifactId>**junit**</artifactId>**
4. **<version>**4.11**</version>**
5. **<scope>**test**</scope>**
6. **</dependency>**

然后为了使代码更简洁，最好在测试类中导入静态资源

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. **import** **static** org.mockito.Mockito.\*;
2. **import** **static** org.junit.Assert.\*;

下面我们开始使用mockito来做测试

1、验证行为

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** verify\_behaviour(){
3. //模拟创建一个List对象
4. List mock = mock(List.**class**);
5. //使用mock的对象
6. mock.add(1);
7. mock.clear();
8. //验证add(1)和clear()行为是否发生
9. verify(mock).add(1);
10. verify(mock).clear();
11. }

2、模拟我们所期望的结果

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** when\_thenReturn(){
3. //mock一个Iterator类
4. Iterator iterator = mock(Iterator.**class**);
5. //预设当iterator调用next()时第一次返回hello，第n次都返回world
6. when(iterator.next()).thenReturn("hello").thenReturn("world");
7. //使用mock的对象
8. String result = iterator.next() + " " + iterator.next() + " " + iterator.next();
9. //验证结果
10. assertEquals("hello world world",result);
11. }

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test(expected = IOException.**class**)
2. **public** **void** when\_thenThrow() **throws** IOException {
3. OutputStream outputStream = mock(OutputStream.**class**);
4. OutputStreamWriter writer = **new** OutputStreamWriter(outputStream);
5. //预设当流关闭时抛出异常
6. doThrow(**new** IOException()).when(outputStream).close();
7. outputStream.close();
8. }

3、参数匹配

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** with\_arguments(){
3. Comparable comparable = mock(Comparable.**class**);
4. //预设根据不同的参数返回不同的结果
5. when(comparable.compareTo("Test")).thenReturn(1);
6. when(comparable.compareTo("Omg")).thenReturn(2);
7. assertEquals(1, comparable.compareTo("Test"));
8. assertEquals(2, comparable.compareTo("Omg"));
9. //对于没有预设的情况会返回默认值
10. assertEquals(0, comparable.compareTo("Not stub"));
11. }

除了匹配制定参数外，还可以匹配自己想要的任意参数

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** with\_unspecified\_arguments(){
3. List list = mock(List.**class**);
4. //匹配任意参数
5. when(list.get(anyInt())).thenReturn(1);
6. when(list.contains(argThat(**new** IsValid()))).thenReturn(**true**);
7. assertEquals(1, list.get(1));
8. assertEquals(1, list.get(999));
9. assertTrue(list.contains(1));
10. assertTrue(!list.contains(3));
11. }
13. **private** **class** IsValid **extends** ArgumentMatcher<List>{
14. @Override
15. **public** **boolean** matches(Object o) {
16. **return** o == 1 || o == 2;
17. }
18. }

需要注意的是如果你使用了参数匹配，那么所有的参数都必须通过matchers来匹配

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** all\_arguments\_provided\_by\_matchers(){
3. Comparator comparator = mock(Comparator.**class**);
4. comparator.compare("nihao","hello");
5. //如果你使用了参数匹配，那么所有的参数都必须通过matchers来匹配
6. verify(comparator).compare(anyString(),eq("hello"));
7. //下面的为无效的参数匹配使用
8. //verify(comparator).compare(anyString(),"hello");
9. }

4、验证确切的调用次数

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** verifying\_number\_of\_invocations(){
3. List list = mock(List.**class**);
4. list.add(1);
5. list.add(2);
6. list.add(2);
7. list.add(3);
8. list.add(3);
9. list.add(3);
10. //验证是否被调用一次，等效于下面的times(1)
11. verify(list).add(1);
12. verify(list,times(1)).add(1);
13. //验证是否被调用2次
14. verify(list,times(2)).add(2);
15. //验证是否被调用3次
16. verify(list,times(3)).add(3);
17. //验证是否从未被调用过
18. verify(list,never()).add(4);
19. //验证至少调用一次
20. verify(list,atLeastOnce()).add(1);
21. //验证至少调用2次
22. verify(list,atLeast(2)).add(2);
23. //验证至多调用3次
24. verify(list,atMost(3)).add(3);
25. </span>}

5、模拟方法体抛出异常

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test(expected = RuntimeException.**class**)
2. **public** **void** doThrow\_when(){
3. List list = mock(List.**class**);
4. doThrow(**new** RuntimeException()).when(list).add(1);
5. list.add(1);
6. }

6、验证执行顺序

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** verification\_in\_order(){
3. List list = mock(List.**class**);
4. List list2 = mock(List.**class**);
5. list.add(1);
6. list2.add("hello");
7. list.add(2);
8. list2.add("world");
9. //将需要排序的mock对象放入InOrder
10. InOrder inOrder = inOrder(list,list2);
11. //下面的代码不能颠倒顺序，验证执行顺序
12. inOrder.verify(list).add(1);
13. inOrder.verify(list2).add("hello");
14. inOrder.verify(list).add(2);
15. inOrder.verify(list2).add("world");
16. }

7、确保模拟对象上无互动发生

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** verify\_interaction(){
3. List list = mock(List.**class**);
4. List list2 = mock(List.**class**);
5. List list3 = mock(List.**class**);
6. list.add(1);
7. verify(list).add(1);
8. verify(list,never()).add(2);
9. //验证零互动行为
10. verifyZeroInteractions(list2,list3);
11. }

8、找出冗余的互动(即未被验证到的)

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test(expected = NoInteractionsWanted.**class**)
2. **public** **void** find\_redundant\_interaction(){
3. List list = mock(List.**class**);
4. list.add(1);
5. list.add(2);
6. verify(list,times(2)).add(anyInt());
7. //检查是否有未被验证的互动行为，因为add(1)和add(2)都会被上面的anyInt()验证到，所以下面的代码会通过
8. verifyNoMoreInteractions(list);
10. List list2 = mock(List.**class**);
11. list2.add(1);
12. list2.add(2);
13. verify(list2).add(1);
14. //检查是否有未被验证的互动行为，因为add(2)没有被验证，所以下面的代码会失败抛出异常
15. verifyNoMoreInteractions(list2);
16. }

9、使用注解来快速模拟

在上面的测试中我们在每个测试方法里都mock了一个List对象，为了避免重复的mock，是测试类更具有可读性，我们可以使用下面的注解方式来快速模拟对象：

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Mock
2. **private** List mockList;

OK，我们再用注解的mock对象试试

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** shorthand(){
3. mockList.add(1);
4. verify(mockList).add(1);
5. }

运行这个测试类你会发现报错了，mock的对象为NULL，为此我们必须在基类中添加初始化mock的代码

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. **public** **class** MockitoExample2 {
2. @Mock
3. **private** List mockList;
5. **public** MockitoExample2(){
6. MockitoAnnotations.initMocks(**this**);
7. }
9. @Test
10. **public** **void** shorthand(){
11. mockList.add(1);
12. verify(mockList).add(1);
13. }
14. }

或者使用built-in runner：MockitoJUnitRunner

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @RunWith(MockitoJUnitRunner.**class**)
2. **public** **class** MockitoExample2 {
3. @Mock
4. **private** List mockList;
6. @Test
7. **public** **void** shorthand(){
8. mockList.add(1);
9. verify(mockList).add(1);
10. }
11. }

更多的注解还有@Captor,@Spy,@InjectMocks

10、连续调用

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test(expected = RuntimeException.**class**)
2. **public** **void** consecutive\_calls(){
3. //模拟连续调用返回期望值，如果分开，则只有最后一个有效
4. when(mockList.get(0)).thenReturn(0);
5. when(mockList.get(0)).thenReturn(1);
6. when(mockList.get(0)).thenReturn(2);
7. when(mockList.get(1)).thenReturn(0).thenReturn(1).thenThrow(**new** RuntimeException());
8. assertEquals(2,mockList.get(0));
9. assertEquals(2,mockList.get(0));
10. assertEquals(0,mockList.get(1));
11. assertEquals(1,mockList.get(1));
12. //第三次或更多调用都会抛出异常
13. mockList.get(1);
14. }

11、使用回调生成期望值

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** answer\_with\_callback(){
3. //使用Answer来生成我们我们期望的返回
4. when(mockList.get(anyInt())).thenAnswer(**new** Answer<Object>() {
5. @Override
6. **public** Object answer(InvocationOnMock invocation) **throws** Throwable {
7. Object[] args = invocation.getArguments();
8. **return** "hello world:"+args[0];
9. }
10. });
11. assertEquals("hello world:0",mockList.get(0));
12. assertEquals("hello world:999",mockList.get(999));
13. }

12、监控真实对象

使用spy来监控真实的对象，需要注意的是此时我们需要谨慎的使用when-then语句，而改用do-when语句

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test(expected = IndexOutOfBoundsException.**class**)
2. **public** **void** spy\_on\_real\_objects(){
3. List list = **new** LinkedList();
4. List spy = spy(list);
5. //下面预设的spy.get(0)会报错，因为会调用真实对象的get(0)，所以会抛出越界异常
6. //when(spy.get(0)).thenReturn(3);
8. //使用doReturn-when可以避免when-thenReturn调用真实对象api
9. doReturn(999).when(spy).get(999);
10. //预设size()期望值
11. when(spy.size()).thenReturn(100);
12. //调用真实对象的api
13. spy.add(1);
14. spy.add(2);
15. assertEquals(100,spy.size());
16. assertEquals(1,spy.get(0));
17. assertEquals(2,spy.get(1));
18. verify(spy).add(1);
19. verify(spy).add(2);
20. assertEquals(999,spy.get(999));
21. spy.get(2);
22. }

13、修改对未预设的调用返回默认期望值

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** unstubbed\_invocations(){
3. //mock对象使用Answer来对未预设的调用返回默认期望值
4. List mock = mock(List.**class**,**new** Answer() {
5. @Override
6. **public** Object answer(InvocationOnMock invocation) **throws** Throwable {
7. **return** 999;
8. }
9. });
10. //下面的get(1)没有预设，通常情况下会返回NULL，但是使用了Answer改变了默认期望值
11. assertEquals(999, mock.get(1));
12. //下面的size()没有预设，通常情况下会返回0，但是使用了Answer改变了默认期望值
13. assertEquals(999,mock.size());
14. }

14、捕获参数来进一步断言

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** capturing\_args(){
3. PersonDao personDao = mock(PersonDao.**class**);
4. PersonService personService = **new** PersonService(personDao);
6. ArgumentCaptor<Person> argument = ArgumentCaptor.forClass(Person.**class**);
7. personService.update(1,"jack");
8. verify(personDao).update(argument.capture());
9. assertEquals(1,argument.getValue().getId());
10. assertEquals("jack",argument.getValue().getName());
11. }
13. **class** Person{
14. **private** **int** id;
15. **private** String name;
17. Person(**int** id, String name) {
18. **this**.id = id;
19. **this**.name = name;
20. }
22. **public** **int** getId() {
23. **return** id;
24. }
26. **public** String getName() {
27. **return** name;
28. }
29. }
31. **interface** PersonDao{
32. **public** **void** update(Person person);
33. }
35. **class** PersonService{
36. **private** PersonDao personDao;
38. PersonService(PersonDao personDao) {
39. **this**.personDao = personDao;
40. }
42. **public** **void** update(**int** id,String name){
43. personDao.update(**new** Person(id,name));
44. }
45. }

15、真实的部分mock

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** real\_partial\_mock(){
3. //通过spy来调用真实的api
4. List list = spy(**new** ArrayList());
5. assertEquals(0,list.size());
6. A a  = mock(A.**class**);
7. //通过thenCallRealMethod来调用真实的api
8. when(a.doSomething(anyInt())).thenCallRealMethod();
9. assertEquals(999,a.doSomething(999));
10. }

13. **class** A{
14. **public** **int** doSomething(**int** i){
15. **return** i;
16. }
17. }

16、重置mock

**[java]** [view plain](http://blog.csdn.net/sdyy321/article/details/38757135/) [copy](http://blog.csdn.net/sdyy321/article/details/38757135/)

[在CODE上查看代码片](https://code.csdn.net/snippets/457368)

1. @Test
2. **public** **void** reset\_mock(){
3. List list = mock(List.**class**);
4. when(list.size()).thenReturn(10);
5. list.add(1);
6. assertEquals(10,list.size());
7. //重置mock，清除所有的互动和预设
8. reset(list);
9. assertEquals(0,list.size());
10. }

### [单元测试系列之3：测试整合之王Unitils](http://stamen.iteye.com/blog/1480316)

**博客分类：**

* [04\_SSH](http://stamen.iteye.com/category/5483)
* [10\_Spring 3.x企业应用开发实战](http://stamen.iteye.com/category/209694)
* [02\_企业应用](http://stamen.iteye.com/category/6530)

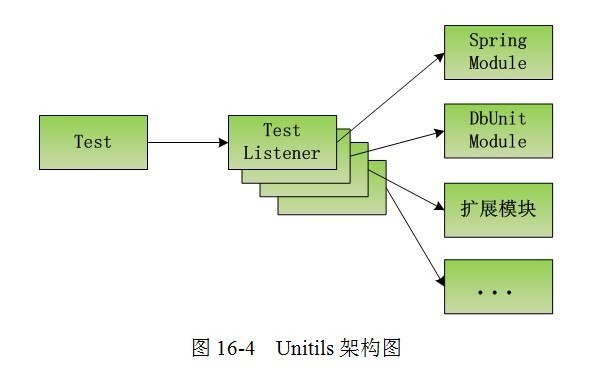
[spring](http://www.iteye.com/blogs/tag/spring)[企业应用](http://www.iteye.com/blogs/tag/%E4%BC%81%E4%B8%9A%E5%BA%94%E7%94%A8)[测试框架](http://www.iteye.com/blogs/tag/%E6%B5%8B%E8%AF%95%E6%A1%86%E6%9E%B6)

**引述**：程序测试对保障应用程序正确性而言，其重要性怎么样强调都不为过。JUnit是必须事先掌握的测试框架，大多数测试框架和测试工具都在此基础上扩展而来，Spring对测试所提供的帮助类也是在JUnit的基础上进行演化的。直接使用JUnit测试基于Spring的应用存在诸多不便，不可避免地需要将大量的精力用于应付测试夹具准备、测试现场恢复、访问测试数据操作结果等边缘性的工作中。Mockito、Unitils、Dbunit等框架的出现，这些问题有了很好的解决方案，特别是Unitils结合Dbunit对测试DAO层提供了强大的支持，大大提高了编写测试用例的效率和质量。   
  
   **Unitils海纳百川（Junit,dbunit,mockito spring hibernate and so on..），以打造一个在实际应用开发中真正实战的测试框架，是致力于应用实战的开发者不得不学习的开源框架。**   
  
**Unitils概述**   
  
   Unitils测试框架目的是让单元测试变得更加容易和可维护。Unitils构建在DbUnit与EasyMock项目之上并与JUnit和TestNG相结合。支持数据库测试，支持利用Mock对象进行测试并提供与Spring和Hibernate相集成。Unitils设计成以一种高度可配置和松散耦合的方式来添加这些服务到单元测试中，目前其最新版本是3.1。   
  
**Unitils功能特点**

* 自动维护和强制关闭单元测试数据库（支持Oracle、Hsqldb、MySQL、DB2）。
* 简化单元测试数据库连接的设置。
* 简化利用DbUnit测试数据的插入。
* 简化Hibernate session管理。
* 自动测试与数据库相映射的Hibernate映射对象。
* 易于把Spring管理的Bean注入到单元测试中，支持在单元测试中使用Spring容器中的Hibernate SessionFactory。
* 简化EasyMock Mock对象创建。
* 简化Mock对象注入，利用反射等式匹配EasyMock参数。

**Unitils模块组件**   
  
   Unitils通过模块化的方式来组织各个功能模块，采用类似于Spring的模块划分方式，如unitils-core、unitils-database、unitils-mock等。比以前整合在一个工程里面显得更加清晰，目前所有模块如下所示：

* unitils-core：核心内核包。
* unitils-database：维护测试数据库及连接池。
* unitils-DbUnit：使用DbUnit来管理测试数据。
* unitils-easymock：支持创建Mock和宽松的反射参数匹配。
* unitils-inject：支持在一个对象中注入另一个对象。
* unitils-mock：整合各种Mock，在Mock的使用语法上进行了简化。
* unitils-orm：支持Hibernate、JPA的配置和自动数据库映射检查。
* unitils-spring：支持加载Spring的上下文配置，并检索和Spring Bean注入。

   Unitils的核心架构中包含Moudule和TestListener两个概念，类似Spring中黏连其他开源软件中的FactoryBean概念。可以看成第三方测试工具的一个黏合剂。整体框架如图16-4所示：   
  
   
  
   通过TestListener可以在测试运行的不同阶段注入某些功能。同时某一个TestListener又被一个对应的Module所持有。Unitils也可以看成一个插件体系结构，TestListener在整个Unitils中又充当了插件中扩展点的角色，从TestListener这个接口中我们可以看到，它可以在crateTestObject、before(after)Class、before(after)TestMethod、beforeSetup、afterTeardown的不同切入点添加不同的动作。   
  
**Unitils配置文件**

* unitils-defaults.properties：默认配置文件，开启所有功能。
* unitils.properties：项目级配置文件，用于项目通用属性配置。
* unitils-local.properties：用户级配置文件，用于个人特殊属性配置。

    Unitils的配置定义了一般配置文件的名字unitils.properties和用户自定义配置文件unitils-local.properties，并给出了默认的模块及模块对应的className，便于Unitils加载对应的模块module。但是如果用户分别在unitils.properties文件及unitils -local.properties文件中对相同属性配置不同值时，将会以unitils-local.properties 的配置内容为主。   
  
**Unitils断言**   
  
   典型的单元测试一般都包含一个重要的组成部分：对比实际产生的结果和希望的结果是否一致的方法，即断言方法（assertEquals）。Unitils 为我们提供了一个非常实用的断言方法，我们以第2章中编写的用户领域对象User为蓝本，比较两个User对象的实例来开始认识Unitils的断言之旅。   
  
**assertReflectionEquals：反射断言**   
  
    在Java世界中，要比较现有两个对象实例是否相等，如果类没有重写equals()方法，用两个对象的引用是否一致作为判断依据。有时候，我们并不需要关注两个对象是否引用同一个对象，只要两个对象的属性值一样就可以了。在JUnit单元测试中，有两种测试方式进行这样的场景测试：一是在比较实体类中重写equals()方法，然后进行对象比较；二是把对象实例的属性一个一个进行比较。不管采用哪种方法，都比较烦锁，Unitils为我们提供了一种非常简单的方法，即使用ReflectionAssert.assertReflectionEquals方法， 如代码清单16-11所示：

**Java代码  [收藏代码](javascript:void())**

1. **package** com.baobaotao.test;
2. **import** java.util.\*;
3. **import** org.junit.Test;
4. **import** **static** org.unitils.reflectionassert.ReflectionAssert.\*;
5. **import** **static** org.unitils.reflectionassert.ReflectionComparatorMode.\*;
6. **import** com.baobaotao.domain.User;
8. **public** **class** AssertReflectionEqualsTest {
9. @Test
10. **public** **void** testReflection(){
11. User user1 = **new** User("tom","1234");
12. User user2 = **new** User("tom","1234");
13. ReflectionAssert.assertReflectionEquals(user1, user2);
14. }
16. }

    ReflectionAssert. AssertReflectionEquals（期望值，实际值，比较级别）方法为我们提供了各种级别的比较断言。下面我们依次介绍这些级别的比较断言。   
    ReflectionComparatorMode.LENIENT\_ORDER：忽略要断言集合collection 或者array 中元素的顺序。   
    ReflectionComparatorMode.IGNORE\_DEFAULTS：忽略Java类型默认值，如引用类型为null，整型类型为0，或者布尔类型为false时，那么断言忽略这些值的比较。   
    ReflectionComparatorMode.LENIENT\_DATES：比较两个实例的Date是不是都被设置了值或者都为null，而忽略Date的值是否相等。   
  
**assertLenientEquals：断言**   
  
   ReflectionAssert 类为我们提供了两种比较断言：既忽略顺序又忽略默认值的断言assertLenientEquals，使用这种断言就可以进行简单比较。下面通过实例学习其具体的用法，如代码清单16-12所示。

**Java代码  [收藏代码](javascript:void())**

1. **package** com.baobaotao.test;
2. **import** java.util.\*;
3. …
4. **public** **class** AssertReflectionEqualsTest {
5. Integer orderList1[] = **new** Integer[]{1,2,3};
6. Integer orderList2[] = **new** Integer[]{3,2,1};

9. //① 测试两个数组的值是否相等，忽略顺序
10. //assertReflectionEquals(orderList1, orderList2,LENIENT\_ORDER);
11. assertLenientEquals(orderList1, orderList2);

14. //② 测试两个对象的值是否相等，忽略时间值是否相等
15. User user1 = **new** User("tom","1234");
16. Calendar cal1 = Calendar.getInstance();
17. user1.setLastVisit(cal1.getTime());
18. User user2 = **new** User("tom","1234");
19. Calendar cal2 = Calendar.getInstance();
20. cal2.set(Calendar.DATE, 15);
21. user2.setLastVisit(cal2.getTime());
22. //assertReflectionEquals(user1, user2,LENIENT\_DATES);
23. assertLenientEquals(user1, user2);
24. }

**assertPropertyXxxEquals：属性断言**   
  
   assertLenientEquals 和assertReflectionEquals 这两个方法是把对象作为整体进行比较，ReflectionAssert 类还给我们提供了只比较对象特定属性的方法：assertPropertyReflection   Equals()和assertPropertyLenientEquals()。下面通过实例学习其具体的用法，如代码清单16-13所示。

**Java代码  [收藏代码](javascript:void())**

1. **package** com.baobaotao.test;
2. **import** java.util.\*;
3. …
4. **public** **class** AssertReflectionEqualsTest {
5. User user = **new** User("tom","1234");
6. assertPropertyReflectionEquals("userName", "tom", user);
7. assertPropertyLenientEquals("lastVisit", **null**, user);
8. }

   assertPropertyReflectionEquals()断言是默认严格比较模式但是可以手动设置比较级别的断言，assertPropertyLenientEquals()断言是具有忽略顺序和忽略默认值的断言。   
  
  
**集成Spring**   
  
   Unitils 提供了一些在Spring 框架下进行单元测试的特性。Spring 的一个基本特性就是，类要设计成为没有Spring 容器或者在其他容器下仍然易于进行单元测试。但是很多时候在Spring 容器下进行测试还是非常有用的。   
   Unitils 提供了以下支持 Spring 的特性：

* ApplicationContext 配置的管理；
* 在单元测试代码中注入Spring 的Beans；
* 使用定义在Spring 配置文件里的Hibernate SessionFactory；
* 引用在Spring 配置中Unitils 数据源。

**ApplicationContext 配置**   
   可以简单地在一个类、方法或者属性上加上@SpringApplicationContext 注解，并用Spring的配置文件作为参数，来加载Spring应用程序上下文。下面我们通过实例来介绍一下如何创建ApplicationContext。

**Java代码  [收藏代码](javascript:void())**

1. **import** org.junit.Test;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.unitils.UnitilsJUnit4;
4. **import** org.unitils.spring.annotation.SpringApplicationContext;
5. **import** org.unitils.spring.annotation.SpringBean;
6. **import** com.baobaotao.service.UserService;
7. **import** **static** org.junit.Assert.\*;
8. //①用户服务测试
9. **public** **class** UserServiceTest **extends** UnitilsJUnit4 {
11. //①-1 加载Spring配置文件
12. @SpringApplicationContext({"baobaotao-service.xml", "baobaotao-dao.xml"})
13. **private** ApplicationContext applicationContext;
15. //①-1 加载Spring容器中的Bean
16. @SpringBean("userService")
17. **private** UserService userService;
19. //①-3 测试Spring容器中的用户服务Bean
20. @Test
21. **public** **void** testUserService (){
22. assertNotNull(applicationContext);
23. assertNotNull(userService.findUserByUserName("tom"));
24. }
25. }
26. …

   在①-1处，通过@SpringApplicationContext 注解加载baobaotao-service.xml和baobaotao- dao.xml两个配置文件，生成一个Spring应用上下文，我们就可以在注解的范围内引用applicationContext这个上下文。在①-2处，通过@SpringBean注解注入当前Spring容器中相应的Bean，如实例中加载ID为“userService”的Bean到当前测试范围。在①-3处，通过JUnit断言验证是否成功加载applicationContext和userService。Unitils加载Spring上下文的过程是：首先扫描父类的@SpringApplicationContext注解，如果找到了就在加载子类的配置文件之前加载父类的配置文件，这样就可以让子类重写配置文件和加载特定配置文件。   
   细心的读者可能会发现，采用这种方式加载Spring应用上下文，每次执行测试时，都会重复加载Spring应用上下文。Unitils为我们提供在类上加载Spring应用上下文的能力，以避免重复加载的问题。

**Java代码  [收藏代码](javascript:void())**

1. …
2. @SpringApplicationContext({"baobaotao-service.xml", "baobaotao-dao.xml"})
3. **public** **class** BaseServiceTest **extends** UnitilsJUnit4 {
5. //加载Spring上下文
6. @SpringApplicationContext
7. **public** ApplicationContext applicationContext;
9. }

     
   在父类BaseServiceTest里指定了Spring配置文件，Spring应用上下文只会创建一次，然后在子类SimpleUserServiceTest 里会重用这个应用程序上下文。加载Spring应用上下文是一个非常繁重的操作，如果重用这个Spring应用上下文就会大大提升测试的性能。

**Java代码  [收藏代码](javascript:void())**

1. …
2. **public** **class** SimpleUserServiceTest **extends** BaseServiceTest {
4. //① Spring容器中加载Id为"userService"的Bean
5. @SpringBean("userService")
6. **private** UserService userService1;
8. //② 从Spring容器中加载与UserService相同类型的Bean
9. @SpringBeanByType
10. **private** UserService userService2;
12. //③ 从Spring容器中加载与userService相同名称的Bean
13. @SpringBeanByName
14. **private**  UserService userService;
16. //④ 使用父类的Spring上下文
17. @Test
18. **public** **void** testApplicationContext(){
19. assertNotNull(applicationContext);
20. }
22. @Test
23. **public** **void** testUserService(){
24. assertNotNull(userService.findUserByUserName("tom"));
25. assertNotNull(userService1.findUserByUserName("tom"));
26. assertNotNull(userService2.findUserByUserName("tom"));
27. }
28. }
29. …

   在①处，使用@SpringBean 注解从Spring容器中加载一个ID为userService的Bean。在②处，使用@ SpringBeanByType注解从Spring容器中加载一个与UserService相同类型的Bean，如果找不到相同类型的Bean，就会抛出异常。在③处，使用@SpringBeanByName 注解从Spring容器中加载一个与当前属性名称相同的Bean。   
  
**集成Hibernate**   
  
   Hibernate是一个优秀的O / R开源框架，它极大地简化了应用程序的数据访问层开发。虽然我们在使用一个优秀的O/R框架，但并不意味我们无须对数据访问层进行单元测试。单元测试仍然非常重要。它不仅可以确保Hibernate映射类的映射正确性，也可以很便捷地测试HQL查询等语句。Unitils为方便测试 Hibernate，提供了许多实用的工具类，如HibernateUnitils就是其中一个，使用assertMappingWithDatabaseConsistent()方法，就可以方便测试映射文件的正确性。   
  
**SessionFactory 配置**   
  
   可以简单地在一个类、方法或者属性上加上@ HibernateSessionFactory 注解，并用Hibernate的配置文件作为参数，来加载Hibernate上下文。下面我们通过实例来介绍一下如何创建SessionFactory。

**Java代码  [收藏代码](javascript:void())**

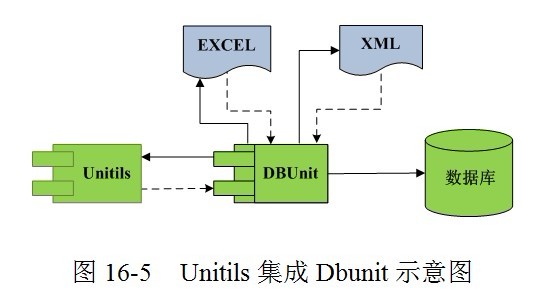
1. …
2. @HibernateSessionFactory("hibernate.cfg.xml")
3. **public** **class** BaseDaoTest **extends** UnitilsJUnit4 {
4. @HibernateSessionFactory
5. **public** SessionFactory sessionFactory;

8. @Test
9. **public** **void** testSessionFactory(){
10. assertNotNull(sessionFactory);
11. }
12. }

    在父类BaseDaoTest里指定了Hibernate配置文件，Hibernate应用上下文只会创建一次，然后在子类SimpleUserDaoTest里会重用这个应用程序上下文。加载Hibernate应用上下文是一个非常繁重的操作，如果重用这个Hibernate应用上下文就会大大提升测试的性能。

**Java代码  [收藏代码](javascript:void())**

1. …
2. **public** **class** SimpleUserDaoTest **extends** BaseDaoTest {
3. **private** UserDao userDao;
5. //① 初始化UserDao
6. @Before
7. **public** **void** init(){
8. userDao = **new** WithoutSpringUserDaoImpl();
9. userDao.setSessionFactory(sessionFactory); //使用父类的SessionFactory
10. }
12. //② Hibernate映射测试
13. @Test
14. **public** **void** testMappingToDatabase() {
15. HibernateUnitils.assertMappingWithDatabaseConsistent();
16. }
18. //③ 测试UserDao
19. @Test
20. **public** **void** testUserDao(){
21. assertNotNull(userDao);
22. assertNotNull(userDao.findUserByUserName("tom"));
23. assertEquals("tom", userDao.findUserByUserName("tom").getUserName());
24. }
25. }
26. …

   为了更好演示如何应用Unitils测试基于Hibernate数据访问层，在这个实例中不使用Spring框架。所以在执行测试时，需要先创建相应的数据访问层实例，如实例中的userDao。其创建过程如①处所示，先手工实例化一个UserDao，然后获取父类中创建的SessionFactory，并设置到UserDao中。在②处，使用Unitils提供的工具类HibernateUnitils中的方法测试我们的Hibernate映射文件。在③处，通过JUnit的断言验证 UserDao相关方法，看是否与我们预期的结果一致。   
  
**集成Dbunit**   
      
   Dbunit是一个基于JUnit扩展的数据库测试框架。它提供了大量的类，对数据库相关的操作进行了抽象和封装。Dbunit通过使用用户自定义的数据集以及相关操作使数据库处于一种可知的状态，从而使得测试自动化、可重复和相对独立。虽然不用Dbunit也可以达到这种目的，但是我们必须为此付出代价（编写大量代码、测试及维护）。既然有了这么优秀的开源框架，我们又何必再造轮子。目前其最新的版本是2.4.8。   
    随着Unitils的出现，将Spring、Hibernate、DbUnit等整合在一起，使得DAO层的单元测试变得非常容易。Unitils采用模块化方式来整合第三方框架，通过实现扩展模块接口org.unitils.core.Module来实现扩展功能。在Unitils中已经实现一个DbUnitModule，很好整合了DbUnit。通过这个扩展模块，就可以在Unitils中使用Dbunit强大的数据集功能，如用于准备数据的@DataSet注解、用于验证数据的@ExpectedDataSet注解。Unitils集成DbUnit流程图如图16-5所示。   
  
   
  
**16.4.5  自定义扩展模块**   
  
   Unitils通过模块化的方式来组织各个功能模块，对外提供一个统一的扩展模块接口org.unitils.core.Module来实现与第三方框架的集成及自定义扩展。在Unitils中已经实现目前一些主流框架的模块扩展，如Spring、Hibernate、DbUnit、Testng等。如果这些内置的扩展模块无法满足需求，我们可以实现自己的一些扩展模块。扩展Unitils模块很简单，如代码清单16-19所示。

**Java代码  [收藏代码](javascript:void())**

1. **package** sample.unitils.module;
2. **import** java.lang.reflect.Method;
3. **import** org.unitils.core.TestListener;
4. **import** org.unitils.core. Module;
5. //① 实现Module接口
6. **public** **class** CustomExtModule **implements** Module {
7. //② 实现获取测试监听的方法
8. **public** TestListener getTestListener() {
9. **return** **new** CustomExtListener();
10. }
12. //② 新建监听模块
13. **protected** **class** CustomExtListener **extends** TestListener {
14. //③ 重写 TestListener里的相关方法，完成相关扩展的功能
15. @Override
16. **public** **void** afterTestMethod(Object testObject, Method testMethod,
17. Throwable testThrowable) {
18. …
19. }
21. @Override
22. **public** **void** beforeTestMethod(Object testObject, Method testMethod) {
23. …
24. }
25. }
26. …
27. }

在①处新建自定义扩展模块CustomExtModule，实现Module接口。在②处新建自定义监听模块，继承TestListener。在③处重写（@Override）TestListener里的相关方法，完成相关扩展的功能。实现自定义扩展模块之后，剩下的工作就是在Unitils配置文件unitils.properties中注册这个自定义扩展的模块：

**引用**

unitils.modules=…,custom   
unitils.module. custom.className= sample.unitils.module.CustomExtModule