## 一段常见的代码

单元测试是项目的重要组成部分。尤其是对持续发展的产品，单元测试在后期的维护，回归有重要等方面有重要作用。

  这样代码在项目中随处可见，看看我们应该如何测试

**Java代码  [IMG_256](http://jilen.iteye.com/blog/javascript:void())**

1. **public** **class** NotifyService {
2. **private** UserCenter uc;
3. **private** MessageCenter mc;
5. **public** **void** sendMessage(**long** userId, String message) {
6. String email = uc.getUser(userId).getEmail();
7. mc.sendEmail(email, message);
8. }
10. **public** **void** setUc(UserCenter uc) {
11. **this**.uc = uc;
12. }
14. **public** **void** setMc(MessageCenter mc) {
15. **this**.mc = mc;
16. }
17. }

UserCenter和MessageCenter都是接口，User是一个简单的JavaBean

由于uc和mc乃外部依赖，此类不需也不应保证uc和mc的正确性，此类只需保证：

  假设uc和mc是正确的，那么我也是正确的。

所以需要隔离依赖--使用**mock**

## 使用EasyMock

如果使用EasyMock，此类之单元测试或许如下[需要static import org.easymock.EasyMock类的相关方法]

**Java代码  [IMG_257](http://jilen.iteye.com/blog/javascript:void())**

1. **public** **class** EasyMockNotifyServiceTest {
2. **private** NotifyService notifyService;
3. **private** UserCenter uc;
4. **private** MessageCenter mc;
6. @Before
7. **public** **void** setUp() {
8. notifyService = **new** NotifyService();
9. uc = createMock(UserCenter.**class**);
10. mc = createMock(MessageCenter.**class**);
11. notifyService.setUc(uc);
12. notifyService.setMc(mc);
13. }
15. @Test
16. **public** **void** testSendMessage() {
17. Long id = 1L;
18. String email = "foo@bar";
19. String message = "hello";
20. expect(uc.getUser(id)).andReturn(createUserWithEmail(email));
21. mc.sendEmail(eq(email), eq(message));
22. replay(uc);
23. replay(mc);
24. notifyService.sendMessage(id, message);
25. verify(mc);//verify a mocked behavior
26. }
28. **private** User createUserWithEmail(String email) {
29. User user = **new** User();
30. user.setEmail(email);
31. **return** user;
32. }
33. }

   当然，此测试并不充分，easymock需要mock每个依赖，对mock的所有方法调用作expect，然后验证需要验证的行为。

**Java代码  [IMG_258](http://jilen.iteye.com/blog/javascript:void())**

1. mc.sendEmail(eq(email), eq(message));

    此处代码理解起来有些怪异，实际上，此处的语义应该是 expect mc.sendEmail... called，即方法执行完毕后sendMail必以正确的参数调用，奈何java 泛型中并未覆盖void型，所以通常会在mock行为调用之后

加上此行以明确语义

**Java代码  [IMG_259](http://jilen.iteye.com/blog/javascript:void())**

1. expectLastCall().times(1);

    另外一个值得注意的地方

**Java代码  [IMG_260](http://jilen.iteye.com/blog/javascript:void())**

1. verify(mc);//verify a mocked behavior

    并未验证uc，我的想法是对于uc我们需要它提供数据(桩)，而不需要验证其行为。

## 使用Mockito

    假使使用Mockito，单元测试也许是这个样子的[需要static import org.mockito.Mockito类的相关方法]

**Java代码  [IMG_261](http://jilen.iteye.com/blog/javascript:void())**

1. **public** **class** NotifyServiceTest {
2. **private** NotifyService notifyService;
3. **private** UserCenter uc;
4. **private** MessageCenter mc;
6. @Before
7. **public** **void** setUp() {
8. notifyService = **new** NotifyService();
9. uc = mock(UserCenter.**class**);
10. mc = mock(MessageCenter.**class**);
11. notifyService.setUc(uc);
12. notifyService.setMc(mc);
13. }
15. @Test
16. **public** **void** testSendMessage() {
17. **long** userId = 1L;
18. String email = "foo@bar";
19. when(uc.getUser(userId)).thenReturn(createUserWithEmail(email));
20. notifyService.sendMessage(userId, "hello");
21. verify(mc).sendEmail(eq(email), eq("hello"));
22. }
24. **private** User createUserWithEmail(String email) {
25. User user = **new** User();
26. user.setEmail(email);
27. **return** user;
28. }
30. }

看到testSendMessage方法

**Java代码  [IMG_262](http://jilen.iteye.com/blog/javascript:void())**

1. @Test
2. **public** **void** testSendMessage() {
3. **long** userId = 1L;
4. String email = "foo@bar";
5. when(uc.getUser(userId)).thenReturn(createUserWithEmail(email));
6. notifyService.sendMessage(userId, "hello");
7. verify(mc).sendEmail(eq(email), eq("hello"));
8. }

语义不言自明

测试前，从uc获得email

测试后，必须调用mc.sendEmail，所以验证之

## 1自动生成Mock类

在需要Mock的属性上标记@Mock注解，然后@RunWith(MockitoJUnitRunner.class)或者在setUp()方法中显示调用MockitoAnnotations.initMocks(this);生成Mock类即可。

## 2自动注入Mock类到被测试类

只要在被测试类上标记@InjectMocks，Mockito就会自动将标记@Mock、@Spy等注解的属性值注入到被测试类中。

**[java]** [view plain](http://blog.csdn.net/dc_726/article/details/8568537) [copy](http://blog.csdn.net/dc_726/article/details/8568537)

1. **import** **static** org.mockito.Mockito.when;
3. **import** <a href="http://lib.csdn.net/base/17" **class**="replace\_word" title="Java EE知识库" target="\_blank" style="color:#df3434; font-weight:bold;">java</a>.util.Collections;
4. **import** java.util.List;
6. **import** javax.annotation.Resource;
8. **import** org.junit.Assert;
9. **import** org.junit.Before;
10. **import** org.junit.Test;
11. **import** org.junit.runner.RunWith;
12. **import** org.mockito.InjectMocks;
13. **import** org.mockito.Mock;
14. **import** org.mockito.MockitoAnnotations;
15. importorg.springframework.test.context.ContextConfiguration;
16. **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
18. **import** com.cdai.ssi.user.dao.UserDao;
19. **import** com.cdai.ssi.user.domain.UserDomain;
20. **import** com.cdai.ssi.user.dto.UserDto;
21. importcom.cdai.ssi.user.service.UserService;
23. @RunWith(SpringJUnit4ClassRunner.**class**)
24. @ContextConfiguration({"classpath:spring/spring-config.xml"})
25. **public** **class** UserServiceTest {
27. @InjectMocks
28. @Resource(name= "userService")
29. privateUserService userService;
31. @Mock
32. privateUserDao userDao;
34. @Before
35. publicvoid setUp() {
36. MockitoAnnotations.initMocks(**this**);
37. }
39. @Test
40. publicvoid testQueryAll() {
41. when(userDao.selectAll()).
42. thenReturn(Collections.<UserDomain>emptyList());
44. List<UserDto>dtoList = userService.queryAll();
46. Assert.assertTrue(dtoList.isEmpty());
47. }
49. }

@InjectMocks的问题是：如果被测试类是代理类，那么注入会失效。比如上面的UserService如果是事务或者其他AOP代理类，那么进入@Test方法时UserService中的DAO属性不会被Mock类替换。

## 3 Mock方法定制再也不用录制、播放了

Mockito的Mock方法定制可读性很强，而且也不需要像EasyMock那样录制播放，定制后就可以使用。

例如：

when(userDao.selectAll()).

                   thenReturn(Collections.<UserDomain>emptyList());

## 4有些方法想Mock定制，有些想调用真实方法

因为@Mock针对接口生成Mock类，所以我们是没法调用到真实的实现类的方法。可以使用@Spy注解标注属性，并且标注@Resource注解让Spring注入真实实现类，那么Mockito就会自动生成Spy类。

例如：

@InjectMocks

@Resource(name ="userService")

privateUserService userService;

@Spy

@Resource

privateUserDao userDao;

Spy类就可以满足我们的要求。如果一个方法定制了返回值或者异常，那么就会按照定制的方式被调用执行；如果一个方法没被定制，那么调用的就是真实类的方法。

如果我们定制了一个方法A后，再下一个测试方法中又想调用真实方法，那么只需在方法A被调用前，调用Mockito.reset(spyObject);就行了。

**[java]** [view plain](http://blog.csdn.net/dc_726/article/details/8568537) [copy](http://blog.csdn.net/dc_726/article/details/8568537)

1. **import** **static** org.mockito.Mockito.when;
3. **import** org.mockito.Mockito;
5. **public** **class** TestMockObject implementsITestMock {
7. publicstatic **void** main(String[] args) {
9. ITestMockmock = Mockito.mock(TestMockObject.**class**);
10. System.out.println(mock.test1());
11. System.out.println(mock.test2());
13. ITestMockspy = Mockito.spy(**new** TestMockObject());
14. System.out.println(spy.test1());
15. System.out.println(spy.test2());
17. when(spy.test1()).thenReturn(100);
18. System.out.println(spy.test1());
20. Mockito.reset(spy);
21. System.out.println(spy.test1());
22. System.out.println(spy.test2());
24. when(spy.test1()).thenReturn(104);
25. System.out.println(spy.test1());
26. }
28. @Override
29. publicint test1() {
30. System.out.print("RealTest1()!!! - ");
31. return1;
32. }
34. @Override
35. publicint test2() {
36. System.out.print("RealTest2()!!! - ");
37. return2;
38. }
40. }

输出为：

0

0

Real Test1()!!! - 1

Real Test2()!!! - 2

Real Test1()!!! - 100

Real Test1()!!! - 1

Real Test2()!!! - 2

Real Test1()!!! - 104

要注意的是，对Spy对象的方法定制有时需要用另一种方法：

===============================================================================

**Importantgotcha on spying real objects!**

Sometimes it's impossible to usewhen(Object) for stubbing spies. Example:

  List list = new LinkedList();

  List spy = spy(list);

  //Impossible: real method is called so spy.get(0) throwsIndexOutOfBoundsException (the list is yet empty)

  when(spy.get(0)).thenReturn("foo");

  //You have to use doReturn() for stubbing

  doReturn("foo").when(spy).get(0);

===============================================================================

因为用when(spy.f1())会导致f1()方法被真正执行，所以就需要另一种写法。

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. }