# Mockito简单教程

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

API文档：http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html

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

## Mockito引入

首先添加maven依赖

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>1.9.5</version>

<scope>test</scope>

</dependency>

当然mockito需要junit配合使用

<dependency>

  <groupId>junit</groupId>

  <artifactId>junit</artifactId>

  <version>4.11</version>

  <scope>test</scope>

</dependency>

然后为了使代码更简洁，最好在测试类中导入静态资源

import static org.mockito.Mockito.\*;

import static org.junit.Assert.\*;

下面我们开始使用mockito来做测试 。

## 1、验证行为

@Test

public void verify\_behaviour(){

    //模拟创建一个List对象

    List mock = mock(List.class);

    //使用mock的对象

    mock.add(1);

    mock.clear();

    //验证add(1)和clear()行为是否发生

    verify(mock).add(1);

    verify(mock).clear();

}

## 2、模拟我们所期望的结果

@Test

public void when\_thenReturn(){

    //mock一个Iterator类

    Iterator iterator = mock(Iterator.class);

    //预设当iterator调用next()时第一次返回hello，第n次都返回world

    when(iterator.next()).thenReturn("hello").thenReturn("world");

    //使用mock的对象

    String result = iterator.next() + " " + iterator.next() + " " + iterator.next();

    //验证结果

    assertEquals("hello world world",result);

}

@Test(expected = IOException.class)

public void when\_thenThrow() throws IOException {

    OutputStream outputStream = mock(OutputStream.class);

    OutputStreamWriter writer = new OutputStreamWriter(outputStream);

    //预设当流关闭时抛出异常

    doThrow(new IOException()).when(outputStream).close();

    outputStream.close();

}

## 3、参数匹配

@Test

public void with\_arguments(){

    Comparable comparable = mock(Comparable.class);

    //预设根据不同的参数返回不同的结果

    when(comparable.compareTo("Test")).thenReturn(1);

    when(comparable.compareTo("Omg")).thenReturn(2);

    assertEquals(1, comparable.compareTo("Test"));

    assertEquals(2, comparable.compareTo("Omg"));

    //对于没有预设的情况会返回默认值

    assertEquals(0, comparable.compareTo("Not stub"));

}

除了匹配制定参数外，还可以匹配自己想要的任意参数

@Test

public void with\_unspecified\_arguments(){

    List list = mock(List.class);

    //匹配任意参数

    when(list.get(anyInt())).thenReturn(1);

    when(list.contains(argThat(new IsValid()))).thenReturn(true);

    assertEquals(1, list.get(1));

    assertEquals(1, list.get(999));

    assertTrue(list.contains(1));

    assertTrue(!list.contains(3));

}

private class IsValid extends ArgumentMatcher<List>{

    @Override

    public boolean matches(Object o) {

        return o == 1 || o == 2;

    }

}

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

@Test

public void all\_arguments\_provided\_by\_matchers(){

    Comparator comparator = mock(Comparator.class);

    comparator.compare("nihao","hello");

    //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

    verify(comparator).compare(anyString(),eq("hello"));

    //下面的为无效的参数匹配使用

    //verify(comparator).compare(anyString(),"hello");

}

## 4、验证确切的调用次数

@Test

public void verifying\_number\_of\_invocations(){

    List list = mock(List.class);

    list.add(1);

    list.add(2);

    list.add(2);

    list.add(3);

    list.add(3);

    list.add(3);

    //验证是否被调用一次，等效于下面的times(1)

    verify(list).add(1);

    verify(list,times(1)).add(1);

    //验证是否被调用2次

    verify(list,times(2)).add(2);

    //验证是否被调用3次

    verify(list,times(3)).add(3);

    //验证是否从未被调用过

    verify(list,never()).add(4);

    //验证至少调用一次

    verify(list,atLeastOnce()).add(1);

    //验证至少调用2次

    verify(list,atLeast(2)).add(2);

    //验证至多调用3次

    verify(list,atMost(3)).add(3);

}

## 5、模拟方法体抛出异常

@Test(expected = RuntimeException.class)

public void doThrow\_when(){

    List list = mock(List.class);

    doThrow(new RuntimeException()).when(list).add(1);

    list.add(1);

}

## 6、验证执行顺序

@Test

public void verification\_in\_order(){

    List list = mock(List.class);

    List list2 = mock(List.class);

    list.add(1);

    list2.add("hello");

    list.add(2);

    list2.add("world");

    //将需要排序的mock对象放入InOrder

    InOrder inOrder = inOrder(list,list2);

    //下面的代码不能颠倒顺序，验证执行顺序

    inOrder.verify(list).add(1);

    inOrder.verify(list2).add("hello");

    inOrder.verify(list).add(2);

    inOrder.verify(list2).add("world");

}

## 7、确保模拟对象上无互动发生 [在CODE上查看代码片](https://code.csdn.net/snippets/457368)

@Test

public void verify\_interaction(){

    List list = mock(List.class);

    List list2 = mock(List.class);

    List list3 = mock(List.class);

    list.add(1);

    verify(list).add(1);

    verify(list,never()).add(2);

    //验证零互动行为

    verifyZeroInteractions(list2,list3);

}

## 8、找出冗余的互动(即未被验证到的)

@Test(expected = NoInteractionsWanted.class)

public void find\_redundant\_interaction(){

    List list = mock(List.class);

    list.add(1);

    list.add(2);

    verify(list,times(2)).add(anyInt());

    //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

    verifyNoMoreInteractions(list);

    List list2 = mock(List.class);

    list2.add(1);

    list2.add(2);

    verify(list2).add(1);

    //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

    verifyNoMoreInteractions(list2);

}

## 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：

@Mock

private List mockList;

OK，我们再用注解的mock对象试试

@Test

public void shorthand(){

    mockList.add(1);

    verify(mockList).add(1);

}

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

public class MockitoExample2 {

    @Mock

    private List mockList;

    public  MockitoExample2(){

        MockitoAnnotations.initMocks(this);

    }

    @Test

    public void shorthand(){

        mockList.add(1);

        verify(mockList).add(1);

    }

}

或者使用built-in runner：MockitoJUnitRunner

@RunWith(MockitoJUnitRunner.class)

public class MockitoExample2 {

    @Mock

    private List mockList;

    @Test

    public void shorthand(){

        mockList.add(1);

        verify(mockList).add(1);

    }

}

更多的注解还有@Captor,@Spy,@InjectMocks

## 10、连续调用

@Test(expected = RuntimeException.class)

public void consecutive\_calls(){

    //模拟连续调用返回期望值，如果分开，则只有最后一个有效

    when(mockList.get(0)).thenReturn(0);

    when(mockList.get(0)).thenReturn(1);

    when(mockList.get(0)).thenReturn(2);

    when(mockList.get(1)).thenReturn(0).thenReturn(1).thenThrow(new RuntimeException());

    assertEquals(2,mockList.get(0));

    assertEquals(2,mockList.get(0));

    assertEquals(0,mockList.get(1));

    assertEquals(1,mockList.get(1));

    //第三次或更多调用都会抛出异常

    mockList.get(1);

}

## 11、使用回调生成期望值

@Test

public void answer\_with\_callback(){

    //使用Answer来生成我们我们期望的返回

    when(mockList.get(anyInt())).thenAnswer(new Answer<Object>() {

        @Override

        public Object answer(InvocationOnMock invocation) throws Throwable {

            Object[] args = invocation.getArguments();

            return "hello world:"+args[0];

        }

    });

    assertEquals("hello world:0",mockList.get(0));

    assertEquals("hello world:999",mockList.get(999));

}

## 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

@Test(expected = IndexOutOfBoundsException.class)

public void spy\_on\_real\_objects(){

    List list = new LinkedList();

    List spy = spy(list);

    //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

    //when(spy.get(0)).thenReturn(3);

    //使用doReturn-when可以避免when-thenReturn调用真实对象api

    doReturn(999).when(spy).get(999);

    //预设size()期望值

    when(spy.size()).thenReturn(100);

    //调用真实对象的api

    spy.add(1);

    spy.add(2);

    assertEquals(100,spy.size());

    assertEquals(1,spy.get(0));

    assertEquals(2,spy.get(1));

    verify(spy).add(1);

    verify(spy).add(2);

    assertEquals(999,spy.get(999));

    spy.get(2);

}

## 13、修改对未预设的调用返回默认期望值

@Test

public void unstubbed\_invocations(){

    //mock对象使用Answer来对未预设的调用返回默认期望值

    List mock = mock(List.class,new Answer() {

        @Override

        public Object answer(InvocationOnMock invocation) throws Throwable {

            return 999;

        }

    });

    //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

    assertEquals(999, mock.get(1));

    //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

    assertEquals(999,mock.size());

}

## 14、捕获参数来进一步断言

@Test

public void capturing\_args(){

    PersonDao personDao = mock(PersonDao.class);

    PersonService personService = new PersonService(personDao);

    ArgumentCaptor<Person> argument = ArgumentCaptor.forClass(Person.class);

    personService.update(1,"jack");

    verify(personDao).update(argument.capture());

    assertEquals(1,argument.getValue().getId());

    assertEquals("jack",argument.getValue().getName());

}

 class Person{

    private int id;

    private String name;

    Person(int id, String name) {

        this.id = id;

        this.name = name;

    }

    public int getId() {

        return id;

    }

    public String getName() {

        return name;

    }

}

interface PersonDao{

    public void update(Person person);

}

class PersonService{

    private PersonDao personDao;

    PersonService(PersonDao personDao) {

        this.personDao = personDao;

    }

    public void update(int id,String name){

        personDao.update(new Person(id,name));

    }

}

## 15、真实的部分mock

@Test

public void real\_partial\_mock(){

    //通过spy来调用真实的api

    List list = spy(new ArrayList());

    assertEquals(0,list.size());

    A a  = mock(A.class);

    //通过thenCallRealMethod来调用真实的api

    when(a.doSomething(anyInt())).thenCallRealMethod();

    assertEquals(999,a.doSomething(999));

}

class A{

    public int doSomething(int i){

        return i;

    }

}

## 16、重置mock

@Test

public void reset\_mock(){

    List list = mock(List.class);

    when(list.size()).thenReturn(10);

    list.add(1);

    assertEquals(10,list.size());

    //重置mock，清除所有的互动和预设

    reset(list);

    assertEquals(0,list.size());

}