# Junit4

## 基本介绍

Junit4是Java用户写单元测试用到最多的一种技术，通过一些注解让我们的多个测试用例跑起来，从而检测代码的正确性。

* 用途：Junit一般用来验证独立功能的业务逻辑，比如工具方法等
* 官网地址：<https://junit.org/junit4/>

## 核心注解

* @Test：用在方法上面，表示这个方法是一个测试用例方法，要求这个方法必须是public void 来修饰的
* @FixMethodOrder：用于方法，让测试方法按照指定的顺序执行，可选值都在MethodSorters中，有三个：JVM（按照虚拟机加载顺序，其实随机）、DEFAULT（默认顺序，随机）、NAME\_ASCENDING（按照名字的ASCII顺序）
* @Ignore：忽略某个测试类或者方法，可以用于方法和类上
* @Parameters：传递参数的测试，声明参数；
* @Parameter：获取传递的参数

## 核心类、方法

### Assert（断言）

断言assert用于断定一个结果，提供了大量的方法，通常建议使用静态导入的方式来引入，这样使用起来更加简单；

* assertTrue / assertFalse: 断言某个条件表达式正确 / 错误；
* assertEquals / assertNotEquals：断言某两个对象内容一样 / 不一样
* assertArrayEquals: 断言两个数组的元素内容一样
* assertNull / assertNotNull: 断言某个元素是空 / 非空
* assertSame / assertNotSame: 断言两个元素一样 / 不一样，比较实用的是==
* assertThat: 断言某个操作会产生异常

## 常用场景

### 各种断言

|  |
| --- |
| package *com.firewolf.testcase.juint4*;  import *org.junit.*Test;  import *java.util.ArrayList*; import *java.util.List*;  import static *org.junit.Assert*.\*;  /\*\*  \* 描述：断言，用于断定各种结果  \* Author：liuxing  \* Date：2020/6/1  \*/ public class *AssertTest* {   /\*\*  \* assertTrue / assertFalse  \*/  @Test  public void testAssertTrue() {  // 验证某个条件表达式  assertTrue(1 == 1);   assertFalse(1 == 2);   // 当结果不正确的时候，抛出指定信息的异常  assertTrue("1不等于2", 1 == 2);  }    /\*\*  \* assertEquals / assertNotEquals  \*/  @Test  public void testAssertEquals() {  assertEquals(100, 100);  assertNotEquals("aaa", "bbb");   assertEquals(new String("aaa"), new String("aaa"));  }   /\*\*  \* assertArrayEquals  \*/  @Test  public void testAssertArrayEquals() {  int[] a = {1, 2, 3};  int[] b = {1, 2, 3};  assertArrayEquals(a, b);  }   /\*\*  \* assertNull / assertNotNull  \*/  @Test  public void testAssertNull() {  *String* a = null;  assertNull(a);   assertNotNull("111");  }   /\*\*  \* assertSame / assertNotSame  \*/  @Test  public void testAssertSame() {  assertSame("aaa", "aaa");  assertNotSame(new String("aaa"), new String("aaa"));  }   /\*\*  \* 断言某个操作会产生某种异常  \*/  @Test  public void testAssertThrows() {   assertThrows(*ArithmeticException*.class, () -> {  int a = 1 / 0;  });   *List* s = new ArrayList();  assertThrows(*IndexOutOfBoundsException*.class, () -> *s*.get(1));  } } |

### 固定方法执行顺序

有时候，我们需要方法按照指定的顺序执行，如：A方法做了插入，B方法做了查询，我们需要在B方法查到数据，那么需要A方法执行了才可以，

|  |
| --- |
| package *com.firewolf.testcase.juint4*;  import *org.junit.*FixMethodOrder; import *org.junit.*Test; import *org.junit.runners.MethodSorters*;  /\*\*  \* 让方法按照固定的顺序执行  \*/ @FixMethodOrder(*MethodSorters*.***NAME\_ASCENDING***) public class *MethodOrderTest* {   @Test  public void testA() {  *System*.***out***.println("first");  }   @Test  public void testB() {  *System*.***out***.println("second");  }   @Test  public void testC() {  *System*.***out***.println("third");  } } |

### 异常处理

异常的处理有三种方法：

|  |
| --- |
| package *com.firewolf.testcase.juint4*;  import *org.junit.*Test;  import static *org.junit.Assert*.\*;  /\*\*  \* 描述：处理异常  \* Author：liuxing  \* Date：2020/6/1  \*/ public class *ExceptionTest* {   /\*\*  \* 通过assertThrows断言异常的出现  \*/  @Test  public void testException1() {  *ArithmeticException* exception = assertThrows(*ArithmeticException*.class, () -> {  int a = 1 / 0;  });  *System*.***out***.println(exception.getMessage());   assertThrows("/ by zero", *ArithmeticException*.class, () -> {  int a = 1 / 0;  });  }   /\*\*  \* 捕获异常之后再做断言  \*/  @Test  public void testException2() {  try {  int a = 1 / 0;  } catch (*Exception e*) {  assertEquals(*e*.getClass(), *ArithmeticException*.class);  assertEquals(*e*.getMessage(), "/ by zero");  }  }   /\*\*  \* 在方法上面指定可能出现的异常  \*/  @Test(expected = *ArithmeticException*.class)  public void testException3() {  int a = 1 / 0;  } } |

### 忽略测试

我们可以使用@Ignore注解来忽略某个类或者某个方法的测试

### 为测试传递参数

通常，我们都是直接在测试用例里面写参数，然而，如果有多组参数需要进行测试，我们可以给测试方法传递参数，有如下几种方式：

1. 通过构造器传入多个参数

|  |
| --- |
| package *com.firewolf.testcase.juint4*;  import *org.junit.Assert*; import *org.junit.*Test; import *org.junit.runner.*RunWith; import *org.junit.runners.Parameterized*;  import *java.util.Arrays*; import *java.util.List*;  import *org.junit.runners.Parameterized*.Parameters;  /\*\*  \* 描述：给测试传递参数，多个参数  \* Author：liuxing  \* Date：2020/6/1  \*/ @RunWith(*Parameterized*.class) public class *ParameterizedTest* {   /\*\*  \* 注意，这个方法要求是public static修饰的  \*  \* @return  \*/  @Parameters  public static *List*<*String*[]> datas() {  return *Arrays*.asList(  new String[][]{  {"hello", "Hello"},  {"hello WORLD", "Hello World"}  });  }   private *String* original, expected;   /\*\*\*  \* 使用构造器来接受参数，注意顺序和传进来的参数是对应的  \* @param original  \* @param expected  \*/  public ParameterizedTest(*String original*, *String expected*) {  this.original = *original*;  this.expected = *expected*;  }   @Test  public void test1() {  *Assert*.assertEquals(expected, *StringUtils*.firstChar2UpCase(original));  } } |

通过构造器来获取参数，需要注意参数的位置进行对应，

1. 通过@Paramter接受参数

|  |
| --- |
| @RunWith(*Parameterized*.class) public class *ParameterizedTest2* {   /\*\*  \* 注意，这个方法要求是public static修饰的  \* 可以通过name属性给测试方法取名，里面可以使用如下占位符：  \* {idnex}：第几组测试  \* {0}: 第一个参数  \* {1}: 第二个参数  \* ....  \*  \* @return  \*/  @Parameters(name = "{index} : StringUtils.firstChar2UpCase({0})={1}")  public static *List*<*String*[]> datas() {  return *Arrays*.asList(  new String[][]{  {"hello", "Hello"},  {"hello WORLD", "Hello World"}  });  }   // 使用注解接受参数，要求参数是public修饰的  // 接受第一个参数，默认为0，不用指定  @Parameter  public *String* original;  // 接受第二个参数，指定为1  @Parameter(1)  public *String* expected;    @Test  public void test1() {  *Assert*.assertEquals(expected, *StringUtils*.firstChar2UpCase(original));  }  } |

我们通过@Paramter来获取参数，通过指定下标来表明获取的是第几个参数，默认不传入，表明获取的是第一个，

1. 单个参数

如果我们的方法只需要一个参数，那么我们可以如下处理：

|  |
| --- |
| package *com.firewolf.testcase.juint4*;  import *org.junit.Assert*; import *org.junit.*Test; import *org.junit.runner.*RunWith; import *org.junit.runners.Parameterized*; import *org.junit.runners.Parameterized*.Parameter; import *org.junit.runners.Parameterized*.Parameters;  import *java.util.Arrays*; import *java.util.List*;  /\*\*  \* 描述：给测试传递参数，单个参数  \* Author：liuxing  \* Date：2020/6/1  \*/ @RunWith(*Parameterized*.class) public class *ParameterizedTest3* {   /\*\*  \* 注意，这个方法要求是public static修饰的  \*  \* @return  \*/  @Parameters  public static *List*<*String*> datas() {  return *Arrays*.asList(  "hello",  "world"  );  }   // 使用注解接受参数，要求参数是public修饰的  // 接受第一个参数，默认为0，不用指定  @Parameter  public *String* original;    @Test  public void test1() {  *Assert*.assertEquals(5, original.length());  } } |

1. 为测试取名

正如第一个示例代码，我们可以给每一次测试取一个自己期望的名字，这可以通过给@Parameters指定name属性来做到，在name的值中，可以使用过如下占位符：

* {index}：第几组测试
* {0}： 第一个参数
* {1}：第二个参数说

超时时间

# Mockito

## Mock介绍

### 什么是mock（what）

mock就是在测试过程中，对于一些不容易构造/获取的对象，创建一个mock对象来模拟对象的行为；

### 为什么要使用mock（why）

在对代码进行单元测试过程中，经常会有以下的情况发生：

|  |
| --- |
| class A 依赖 class B  class B 依赖 class C和class D  class C 依赖 ...  class D 依赖 ...  1.被测对象依赖的对象构造复杂  我们想对class A进行单元测试，需要构造大量的class B、C、D等依赖造步骤多、耗时较长)  ，对象，他们的构造这时我们可以利用mock去构造过程复杂(体现在构虚拟的class B、C、D  对象用于class A的测试，因为我们只是想测试class A的行为是否符合预期，我们并不需要  测试依赖对象。  2.被测单元依赖的模块尚未开发完成，而被测对象需要依赖模块的返回值进行测试：  ----- 比如service层的代码中，包含对dao层的调用，但dao层代码尚未实现  ----- 比如web的前端依赖后端接口获取数据进行联调测试，但后端接口并未开发完成 |

### mock使用场景(when & where)

1. 单元测试/接口测试中测试对象依赖其他对象，这些对象的构造复杂、耗时或者根本无法构造(未交付)

2. 我们只测试对象内部逻辑的质量，不关心依赖对象的逻辑正确性和稳定性

### mock原则

* 不需要对所有的依赖对象进行mock，只对那些构造复杂、构造耗时较长的依赖进行mock
* 如果做分层自动化，高层的测试设计可以基于以下假设：低层的测试已保证低层对象的质量，高层对低层的依赖可以mock，无需关心所依赖的低层对象的内部逻辑质量

### 常用mock技术

* powermock
* easymock
* mockito

## **Mockito介绍**

mockito是众多mock技术中的佼佼者，功能强大，api简洁。

相关文档地址：

* 官方地址：<https://site.mockito.org/>
* 文档地址：<https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>
* 中文文档：<https://github.com/hehonghui/mockito-doc-zh#0>

## **核心API**

这里列举一些Mockito中常用的一些API，主要如下：

### ArgumentMatchers

ArgumentMatchers是用来做参数匹配的，主要有如下几个方法：

* anyInt：任何int类型的参数，类似的还有anyLong/anyByte等等。
* any(Class<T>)：某种类型的任意实例
* eq：等于某个值的时候，如果是对象类型的，则看toString方法
* isA：匹配某种类型
* matches：使用正则表达式进行匹配

### Mockito

mockito是最核心的类，继承自ArgumentMatchers，具有我们需要的大部分方法，如下：

1. 创建对象

* mock：构建一个我们需要的对象；可以mock具体的对象，也可以mock接口。
* spy：构建监控对象

1. 验证行为

* verify：验证某种行为

1. Mock行为

* when：当执行什么操作的时候，一般配合doXXX或者OngoingStubbing的thenXXX一起使用。表示执行了一个操作之后产生什么效果。
* doNothing：不做任何处理
* doReturn：返回一个结果
* doThrow：抛出一个指定异常
* doAnswer：指定一个操作，传入Answer
* doCallRealMethod：返回真实业务执行的结果，只能用于监控对象

1. 验证次数

* times：某个操作执行了多少次
* atLeastOnce：某个操作至少执行一次
* atLeast：某个操作至少执行指定次数
* atMost：某个操作至多执行指定次数
* atMostOnce：某个操作至多执行一次

### OngoingStubbing

OngoingStubbing用于mock操作的结果：

* thenReturn：指定一个返回的值
* thenThrow：抛出一个指定异常
* then：指定一个操作，需要传入自定义Answer；
* thenCallRealMethod：返回真实业务执行的结果，只能用于监控对象

## **常用场景**

### 引入mockito

使用mockito我们可以引入下面的依赖：

|  |
| --- |
| <dependency>  <groupId>org.mockito</groupId>  <artifactId>mockito-all</artifactId>  <version>2.0.2-beta</version>  <scope>test</scope>  </dependency> |

而在springboot中，只要一来了spring-boot-test，就自动了引入了Mockito相关依赖，不需要我们做额外工作。

### mock对象或行为

spy对象的使用方式，要先执行do等方法，mock对象也可以这样使用，而mock对象可先使用doxxx方法，也可以使用thenXXX方法