# 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

### 常规注解

* @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，切没有入参。
* @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
* @Before / @After：在每个测试方法执行之前 / 后执行，被注解的方法必须是public，无返回值，无参数
* @BeforeClass / @AfterClass：在所有测试方法执行之前 / 后执行，被注解的方法必须是public，static，无返回值，无参数；
* @Suite.SuiteClasses：一次执行多个测试类的所有方法

### Rule测试

具体使用看后面的

* @Rule：声明一个Rule，会在每个单元测试方法的时候使用，要求该变量使用public修饰
* @ClassRule：声明一个Rule，会在类的所有方法运行完或者之后执行，要求使用public static修饰

### Theory测试注解

具体使用看后面的

* @DataPoint：用于声明这是一个用于组合理论测试方法入参的参数，要求是用public static修饰
* @DataPoints：用于声明这是一组用于理论测试方法入参的参数，要是使用public static修饰
* @Theory：这是一个理论测试，类似于@Test
* @TestedOn：用于对参数进行限制，指明只有哪些参数被使用。

## 核心类、方法

### Assert（断言）

断言assert用于断定一个结果，如果实际的结果不对，那么这个测试用例就会失败，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，已经废弃，应该使用MatcherAssert的相应方法。

### Assume(假设)

假定某个条件是否成立，如果不成立，则跳过测试，而不是报错！

* assumeTrue / assumeFalse：假设某个条件成立或者不成立
* assumeNotNull：判断非空
* assumeNoException：判断没有异常
* assumeThat：坚定某个条件成立

### MatcherAssert(匹配器断言)

这个类替代了Assert中的assertThat方法，

* assertThat：断言某个条件成立

### CoreMatchers（匹配器）

使用MatcherAssert.assertThat和Assume.assumeThat的时候，我们需要传入一个Matcher，由于junit集成了hamcrest，所有我们可以使用CoreMatchers来使用一些预置的匹配器，这主要如下：

1. 用于对象

* is(T) / equalTo(T)：判断内容是否一样，调用了equals
* sameInstance / theInstance(T)：是某个对象，==判断
* nullValue / notNullValue：空/非空
* not(T)：不等于，使用的是equals
* isA(Class<T>) / instanceOf(Class<T>) / any(Class<T>)：是什么类型的

1. 用于字符串

* startsWith / endsWith(String)：某个字符串是什么开头/结尾的
* containsString(String)：包含某个字符串

1. 用于集合

* hasItems(T...)：集合中是否有指定的所有元素
* hasItems(Matcher<? super T>...)：是否有元素能匹配所有的匹配器
* hasItem(T): 是否有某个元素
* hasItem(Matcher<? super T>)：是否有匹配某个匹配器的元素
* everyItem(Matcher<? super T>):是否所有元素都满足要求

1. 组合匹配器（可以组合多个匹配器一起工作）

* either ：是否能满组其中一个匹配器，只能使用or连接
* both：用来连接多个匹配器，之间可以使用and和or来连接，如果是and，要求都匹配，如果是or，要求某个匹配
* anyOf(Matcher<? super T>...)：是否满足其中一个匹配器
* allOf(Matcher<? super T>...)：是否满足所有匹配器

## 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

### 超时时间

可以要求某个测试方法的执行时间不能超过多少，这有两种：

1. 通过@Test的timeout属性来要求某个方法不能超过多久
2. 通过@Rule来要求任何一个方法都不能超过多久

|  |
| --- |
| package *com.firewolf.testcase.juint4*;  import *org.junit.*Rule; import *org.junit.*Test; import *org.junit.rules.Timeout*;   /\*\*  \* Description: 超时测试  \* Author: liuxing  \* Email: 1483345163@qq.com  \* Time: 2020/6/1 10:54 下午  \*/ public class *TimeOutTest* {   /\*\*  \* 要求任何一个测试方法的执行时间不能超过1秒  \*/  @Rule  public *Timeout* timeout = *Timeout*.seconds(1);  /\*\*  \* 这个执行时间不能超过1秒  \* @throws *Exception* \*/  @Test(timeout = 1000)  public void test() throws *Exception*{  *Thread*.sleep(500);  }   @Test  public void test2() throws *Exception*{  *Thread*.sleep(2600);  } } |

### 一次执行多个测试类

|  |
| --- |
| package *com.firewolf.testcase.juint4*;  import *org.junit.runner.*RunWith; import *org.junit.runners.Suite*;  /\*\*  \* Description: 一次性测试多个类的测试方法  \* Author: liuxing  \* Email: 1483345163@qq.com  \* Time: 2020/6/1 11:04 下午  \*/ @RunWith(*Suite*.class) @Suite.SuiteClasses(value = {  *AssertTest*.class,  *ParameterizedTest*.class }) public class *SuitTest* { } |

### 参数化测试

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，参数话测试要求必须使用Parameterized 这个Runner

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

### Assume跳过某些条件的测试

如果我们有多个数据进行测试，而某些测试在某些数据出现的时候，会产生错误，那么，我们可以通过Assume来跳过某些测试，而不是报错

|  |
| --- |
| package *com.firewolf.testcase.juint4*;  import static *org.junit.Assume*.\*;  import *org.hamcrest.CoreMatchers*; import *org.junit.*Test;  /\*\*  \* Description:   \* Author: liuxing  \* Email: 1483345163@qq.com  \* Time: 2020/6/1 10:15 下午  \*/ public class *AssumptionsTest* {  @Test  public void test() {  int a = 1 + 2; // 调用某个方法计算出来结果  assumeTrue(a != 0);  int b = 100 / a;  try {  *String* s = new String("aaa"); //计算出来的二级果  assumeNotNull(s);  } catch (*Exception e*) {  assumeNoException(*e*);  }  assumeThat(a, *CoreMatchers*.is(3));  } } |

## assertThat/assumeThat语法与Matcher

在上面的的一些常用场景之后，我们来讲讲比较复杂的assertThat和assumeThat，这两个方法的定义里面都需要传入一个实现了Matcher接口的实例对象，这里以assertThat为例来讲解

assertThat定义如下：

assertThat([reason, ]T actual, Matcher<? super T> matcher)，

其中，reason为断言失败时的输出信息，actual为断言的值或对象，matcher为断言的匹配器，里面的逻辑决定了给定的actual对象满不满足断言。

在org.hamcrest.CoreMatchers类中组织了所有JUnit内置的Matcher，调用其任意一个方法都会创建一个与方法名字相关的Matcher。

### CoreMatchers匹配器使用

CoreMatchers中内置的匹配器在核心API中已经介绍了，这里直接给出代码示例

|  |
| --- |
| package *com.firewolf.testcase.juint4*;   import *org.hamcrest.Description*; import *org.hamcrest.Matcher*; import *org.junit.*Test;  import *java.util.Arrays*; import *java.util.List*;  import static *org.hamcrest.CoreMatchers*.\*; import static *org.hamcrest.MatcherAssert*.assertThat;  /\*\*  \* Description: Matcher  \* Author: liuxing  \* Email: 1483345163@qq.com  \* Time: 2020/6/1 11:54 下午  \*/ public class *MatcherTest* {   @Test  public void test1() {  *String* s = new String("Hello,World");   // 不需要匹配器的用法  assertThat("字符串长度小于5", s.length() > 5);  // 用于对象的  // is(T) / equalTo(T)：判断内容是否一样，调用了equals  assertThat(s, is(new String("Hello,World")));  assertThat(s, equalTo("Hello,World"));   // sameInstance / theInstance(T) ：是某个对象，==判断  assertThat(s, theInstance(s));  assertThat(s, sameInstance(s));   // nullValue：空值  *Object* a = null;  assertThat(null, nullValue());  assertThat(null, nullValue(*String*.class));   // notNullValue: 非空值  assertThat(s, notNullValue(*String*.class));  assertThat(s, notNullValue());   // not(T) : 不等于，equals  assertThat(s, not("hello"));   // isA(Class<T>) / instanceOf(Class<T>) / any(Class<T>)：是什么类型的  assertThat(s, isA(*String*.class));  assertThat(s, instanceOf(*String*.class));    // 字符串相关  // startsWith / endsWith(String)：某个字符串是什么开头/结尾的  assertThat(s, endsWith("ld"));  // containsString(String)：包含某个字符串  assertThat(s, containsString(","));    // 集合  *List*<*String*> sList = *Arrays*.asList("aa", "bb", "ccc");  // hasItems(T...)：集合中是否有指定的所有元素  assertThat(sList, hasItems("aa", "bb"));  // 要求找到能匹配上所有的匹配器的元素  // hasItems(Matcher<? super T>...)：是否有元素能匹配所有的匹配器  assertThat(sList, hasItems(isA(*String*.class), new StringLengthMatcher(2)));  // hasItem(T): 是否有某个元素  // hasItem(Matcher<? super T>)：是否有匹配某个匹配器的元素  //everyItem(Matcher<? super T>):是否所有元素都满足要求  assertThat(sList, everyItem(new StringLengthMatcher(1)));   // 多个匹配器组合  // either ：是否能满组其中一个匹配器，只能使用or连接  assertThat(s, either(containsString("Hello")).or(containsString("111")));   // both：用来连接多个匹配器，之间可以使用and和or来连接，如果是and，要求都匹配，如果是or，要求某个匹配  assertThat(s, both(containsString("Hello")).and(containsString("World")).and(containsString(",")).or(containsString("aaa")));   // anyOf(Matcher<? super T>...)：是否满足其中一个匹配器  assertThat(s,anyOf(containsString("Hello"),containsString("BBB")));   // allOf(Matcher<? super T>...)：是否满足所有匹配器  assertThat(s,allOf(containsString("Hello"),containsString("World")));   }  } |

### 自定义匹配器

除了可以使用CoreMatchers中的匹配器之外，我们还可以实现自己的匹配器。我们可以通过实现org.hamcrest.Matcher接口或者是继承org.hamcrest.BaseMatcher来自定义自己的匹配器，重要的是去实现matches方法。

|  |
| --- |
| /\*\*  \* 自定义Matcher  \*/ class *StringLengthMatcher* implements *Matcher*<*String*> {   private int length;   public StringLengthMatcher(int *length*) {  this.length = *length*;  }   @Override  public boolean matches(*Object item*) {  return ((*String*) *item*).length() > length;  }   @Override  public void describeMismatch(*Object item*, *Description mismatchDescription*) {  }   @Override  public void \_dont\_implement\_Matcher\_\_\_instead\_extend\_BaseMatcher\_() {  }   @Override  public void describeTo(*Description description*) {  } } |
| 继承BaseMatcher  class *StringLengthMatcher2* extends *BaseMatcher*<*String*> {  private int length;   public StringLengthMatcher2(int *length*) {  this.length = *length*;  }   @Override  public boolean matches(*Object item*) {  return ((*String*) *item*).length() > length;  }   @Override  public void describeTo(*Description description*) {   } } |

## Theory（理论）

JUnit4引入了Theory（理论）机制，这个概念类似于参数化测试。在参数化测试中，我们需要给定所有具体的测试数据组，而在理论测试中，用户只需给定了一些数据，JUnit自动利用这些数据组合出各种各种可能的组合来执行测试。

官网地址：<https://github.com/junit-team/junit4/wiki/Theories>

理论测试要求必须使用org.junit.experimental.theories.Theories这个Runner，依赖@DataPoint、@DataPoints、@TestedOn等注解来完成，示例如下：

### 逐个声明参数@DataPoint

|  |
| --- |
| package *com.firewolf.testcase.juint4.s11\_theory*;  import *org.hamcrest.CoreMatchers*; import *org.junit.Assert*; import *org.junit.Assume*; import *org.junit.*ClassRule; import *org.junit.experimental.theories.*DataPoint; import *org.junit.experimental.theories.Theories*; import *org.junit.experimental.theories.*Theory; import *org.junit.rules.Verifier*; import *org.junit.runner.*RunWith;  /\*\*  \* Description:  \* Author: liuxing  \* Email: 1483345163@qq.com  \* Time: 2020/6/6 10:15 下午  \*/ @RunWith(*Theories*.class) public class *TheoryTest* {   /\*\*  \* 验证结果，  \*/  @ClassRule  public static *Verifier* verifier = new *Verifier*() {  @Override  protected void verify() throws *Throwable* {  *Assert*.assertEquals(totalCount, 9);  *Assert*.assertEquals(passedCount, 6);  }  };   private static int totalCount = 0;  private static int passedCount = 0;   /\*\*  \* @DataPoint用于组合参数的数据，要求必须使用public static修饰  \*/  @DataPoint  public static int a = 0;   @DataPoint  public static int b = 1;   @DataPoint  public static int c = 2;   /\*\*  \* @param first  \* @param second  \* @Theory用于说明这是一个理论测试  \*/  @Theory  public void testDevide(int *first*, int *second*) {  totalCount += 1;  // 要求跳过second=0的数据  *Assume*.assumeThat(*second*, *CoreMatchers*.not(0));  *Assert*.assertTrue((*first* / *second*) >= 0);  passedCount += 1;  }  } |

### 声明一组参数@DataPoints

|  |
| --- |
| @RunWith(*Theories*.class) public class *TheoryTest2* {   /\*\*  \* 使用@DataPoints声明一组参数  \*/  @DataPoints  public static int[] getTestData() {  return new int[]{0, 1, 2};  }   /\*\*  \* @param first  \* @param second  \* @Theory用于说明这是一个理论测试  \*/  @Theory  public void testDevide(int *first*, int *second*) {  // 要求跳过second=0的数据  *Assume*.assumeTrue(*second* != 0);  *Assert*.assertTrue((*first* / *second*) >= 0);  }  } |

### 限定参数

我们可以看到，在上面的测试例子中，由于除数不能是0，所以我们使用Assume来跳过了second为0的情况，而实际上，Theory提供了@TestedOn来限定参数，如：

|  |
| --- |
| @Theory public void testDevide(int *first*, @TestedOn(ints = {1, 2}) int *second*) {  // 要求跳过second=0的数据  *Assert*.assertTrue((*first* / *second*) >= 0); } |

然而，@TestedOn只能用于限定整数的，如果是其他类型的，我们可以参看@TestedOn的实现来完成，具体可以参看@TestedOn注解以及官网地址<https://github.com/junit-team/junit4/wiki/Theories>，这里不做赘述。

## Rule（Junit扩展）

### Rule简介

Rule是JUnit4中的新特性，它让我们可以扩展JUnit的功能，灵活地改变测试方法的行为。JUnit中用@Rule和@ClassRule两个注解来实现Rule扩展，这两个注解需要放在实现了TestRule接口的成员变量（@Rule）或者静态变量（@ClassRule）上。@Rule和@ClassRule的不同点是，@Rule是方法级别的，每个测试方法执行时都会调用被注解的Rule，而@ClassRule是类级别的，在执行一个测试类的时候只会调用一次被注解的Rule。

### JUnit内置Rule

JUnit4中默认实现了一些常用的Rule：

1. **ExternalResource**

用于在测试用例开始之前创建外部资源，如：文件、Socket连接、服务器、数据库连接等等，如：

|  |
| --- |
| public class *ExternalResourceRuleTest* {   private *File* file;   @Rule  public *ExternalResource* resource = new *ExternalResource*() {  @Override  protected void before() throws *Throwable* {  file = *File*.createTempFile("test", "txt");  }   @Override  protected void after() {  file.delete();  }  };    @Test  public void testFile(){  *System*.***out***.println(file.getAbsolutePath());  } } |

1. **TemporaryFolder**

TemporaryFolder继承自ExternalResource会在测试方法执行之前创建一个临时目录

|  |
| --- |
| public class *TemporaryFolderTest* {   @Rule  public *TemporaryFolder* temporaryFolder = new TemporaryFolder();   @Test  public void testTempararyFolder() throws *Exception* {  *File* file = temporaryFolder.newFile("aaa.txt");  *System*.***out***.println(temporaryFolder.getRoot());  *System*.***out***.println(file.getAbsolutePath());  } } |

1. **Verifier**

Verifier 用来在测试完成之后做一些校验，以验证结果是否正确：

|  |
| --- |
| public class *VerifierTest* {   private *String* result;   @Rule  public *Verifier* verifier = new *Verifier*() {  @Override  protected void verify() throws *Throwable* {  if (!"success".equals(result)) {  throw new RuntimeException("result is failed!");  }  }  };   @Test  public void test() {  result = "failed";  } } |

1. **ErrorCollector**

ErrorCollector继承自Verifier，用于收集错误，然后在最后一次性报告出来

|  |
| --- |
| public class *ErrorCollectorTest* {   @Rule  public *ErrorCollector* collector = new ErrorCollector();   @Test  public void test() {   try {  int i = 1 / 0;  *System*.***out***.println("计算结果");  } catch (*Exception e*) {  collector.addError(*e*);  }  try {  *String* a = null;  *System*.***out***.println(a.length());  } catch (*Exception e*) {  collector.addError(*e*);  }  *System*.***out***.println("xxx");  } } |

1. **TestWatcher**

TestWatcher 定义了五个触发点，分别是测试成功，测试失败，测试开始，测试完成，测试跳过，能让我们在每个触发点执行自定义的逻辑。有点类似于AOP

|  |
| --- |
| public class *TestWatcherTest* {   @Rule  public *TestWatcher* testWatcher = new *TestWatcher*() {  @Override  protected void succeeded(*Description description*) {  *System*.***out***.println(*description*.getDisplayName() + "方法执行成功了");  }   @Override  protected void failed(*Throwable e*, *Description description*) {  *System*.***out***.println(*description*.getDisplayName() + "方法执行失败了");  }   @Override  protected void skipped(*AssumptionViolatedException e*, *Description description*) {  *System*.***out***.println(*description*.getDisplayName() + "方法被跳过了");  }   @Override  protected void starting(*Description description*) {  *System*.***out***.println(*description*.getDisplayName() + "方法开始执行了....");  }   @Override  protected void finished(*Description description*) {  *System*.***out***.println(*description*.getDisplayName() + "方法执行结束了");  }  };   @Test  public void test(){  *System*.***out***.println("xxxx");  } } |

1. **TestName**

TestName继承自TestWatcher，用于获取当前执行的测试用例的名字

|  |
| --- |
| public class *TestNameTest* {   @Rule  public *TestName* testName = new TestName();   @Test  public void test() {  *System*.***out***.println(testName.getMethodName());  } } |

1. **TimeOut**

用于验证方法执行不能超过多久

|  |
| --- |
| \*/ public class *TimeOutTest* {   /\*\*  \* 要求任何一个测试方法的执行时间不能超过1秒  \*/  @Rule  public *Timeout* timeout = *Timeout*.seconds(1);  /\*\*  \* 这个执行时间不能超过1秒  \* @throws *Exception* \*/  @Test(timeout = 1000)  public void test() throws *Exception*{  *Thread*.sleep(500);  }   @Test  public void test2() throws *Exception*{  *Thread*.sleep(600);  } } |

1. **ExpectedException**

异常断言

|  |
| --- |
| public class *ExpectedExceptionTest* {   @Rule  public *ExpectedException* expectedException = *ExpectedException*.none();   @Test  public void test() {  expectedException.expect(*NullPointerException*.class);  throw new NullPointerException();  } } |

### Rule实现原理

在Junit运行的时候，通常都会通过@RunWith(....)的方式进行执行，我们注意看的话， 会发现所有的Runner最终都继承了BlockJUnit4ClassRunner，同样，如果不指定@RunWith，默认也是使用BlockJUnit4ClassRunner中，有一个methodBlock方法，这个方法会在测试方法执行前被调用：

|  |
| --- |
| protected *Statement* methodBlock(final *FrameworkMethod method*) {  *Object* test;  try {  test = new *ReflectiveCallable*() {  @Override  protected *Object* runReflectiveCall() throws *Throwable* {  return createTest(*method*);  }  }.run();  } catch (*Throwable e*) {  return new Fail(*e*);  }   *Statement* statement = methodInvoker(*method*, test);  statement = possiblyExpectingExceptions(*method*, test, statement);  statement = withPotentialTimeout(*method*, test, statement);  statement = withBefores(*method*, test, statement);  statement = withAfters(*method*, test, statement);  statement = withRules(*method*, test, statement);  statement = withInterruptIsolation(statement);  return statement; } |

在JUnit执行每个测试方法之前，methodBlock方法都会被调用，用于把该测试包装成一个Statement。Statement代表一个具体的动作，例如测试方法的执行，Before方法的执行或者Rule的调用，类似于J2EE中的Filter，Statement也使用了责任链模式，将Statement层层包裹，就能形成一个完整的测试，JUnit最后会执行这个Statement。从上面代码可以看到，有以下内容被包装进Statement中：

1. 测试方法的执行；
2. 异常测试，对应于@Test(expected=XXX.class)；
3. 超时测试，对应与@Test(timeout=XXX)；
4. Before方法，对应于@Before注解的方法；
5. After方法，对应于@After注解的方法；
6. Rule的执行。
7. 中断当前线程。

在Statement中，可以用evaluate方法控制Statement执行的先后顺序，这个可以参看RunBefores，同样，我们可以通过查看各个部分，进行相应的扩展

### 自定义Rule

我们可以通过实现TestRule接口然后返回一个自己的Statement来实现自己的Rule，如完成一个自己的Rule来计算方法耗时：

|  |
| --- |
| class *SelfTimeRule* implements *TestRule* {  @Override  public *Statement* apply(*Statement base*, *Description description*) {  return new *Statement*() {  @Override  public void evaluate() throws *Throwable* {  try {  long start = *System*.currentTimeMillis();  *base*.evaluate();  long end = *System*.currentTimeMillis();  *System*.***out***.println("总共耗时：" + (end - start));  } catch (*Throwable throwable*) {  *throwable*.printStackTrace();  }  }  };  } } |

使用：

|  |
| --- |
| public class *SelfTimeRuleTest* {  @Rule  public *SelfTimeRule* timeRule = new SelfTimeRule();   @Test  public void test() throws *Exception* {  *Thread*.sleep(1000);  } } |

# Junit5

# 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-core</artifactId>  <version>3.3.3</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