# 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

### 常用注解

* @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测试注解

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的时候执行后续步骤，否则直接跳过后面的步骤
* assumeNotNull：假设不为空
* assumeNoException：假设没有异常
* assumeThat：假定某个条件成立

### 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

## 引入juit4

如果要使用juint4，只需要引入下面的依赖：

|  |
| --- |
| <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.13</version>  <scope>test</scope> </dependency> |

## 常用场景

### 简单使用

如果我们需要编写测试用例，只需要如下使用即可：

|  |
| --- |
| package *com.firewolf.testcase.juint4.s00\_helloworld*;  import *org.junit.*Test;  /\*\*  \* 描述：  \* Author：liuxing  \* Date：2020/6/19  \*/ public class *HelloWorld* {   @Test  public void test() {  *System*.***out***.println("这是一个测试用例");  } } |

需要注意的是，我们大多数测试方法都需要使用 public void来进行修饰，而且，不能有参数（参数化测试例外）

### 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

### 假设

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.s08\_assume*;  import static *org.junit.Assume*.\*;  import *org.hamcrest.CoreMatchers*; import *org.junit.*Test;  /\*\*  \* Description: 假设  \* Author: liuxing  \* Email: 1483345163@qq.com  \* Time: 2020/6/1 10:15 下午  \*/ public class *AssumeTest* {  @Test  public void test() {  int a = 1 + 2; // 调用某个方法计算出来结果  assumeTrue(a != 0); // 因为条件成立，所有走了后面的步骤，如果不成立，后面的就不再执行  int b = 100 / a;  try {  *System*.***out***.println("xxxx");  *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));  } } |

### 生命周期方法

我们可以在某个测试类/方法之前/之后做一些我们需要做的操作，如：打开资源、关闭资源等；这主要借助于@Before / @After @BeforeClass / @AfterClass等四个注解。

|  |
| --- |
| package *com.firewolf.testcase.juint4.s12\_lifecycle*;  import *org.junit.*\*;  /\*\*  \* 描述： 生命周期方法  \* Author：liuxing  \* Date：2020/6/19  \*/ public class *LifeCycleTest* {   @BeforeClass  public static void allBefore() {  *System*.***out***.println("所有测试方法开始之前");  }   @AfterClass  public static void allAfter() {  *System*.***out***.println("所有测试方法结束之后");  }   @Before  public void eachBefore() {  *System*.***out***.println("某个测试方法要开始了");  }   @After  public void eachAfter() {  *System*.***out***.println("某个测试方法要结束了");  }   @Test  public void test1() {  *System*.***out***.println("test1");  }   @Test  public void test2() {  *System*.***out***.println("test2");  }  } |

结果：

所有测试方法开始之前

某个测试方法要开始了

test1

某个测试方法要结束了

某个测试方法要开始了

test2

某个测试方法要结束了

所有测试方法结束之后

## 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);  } } |

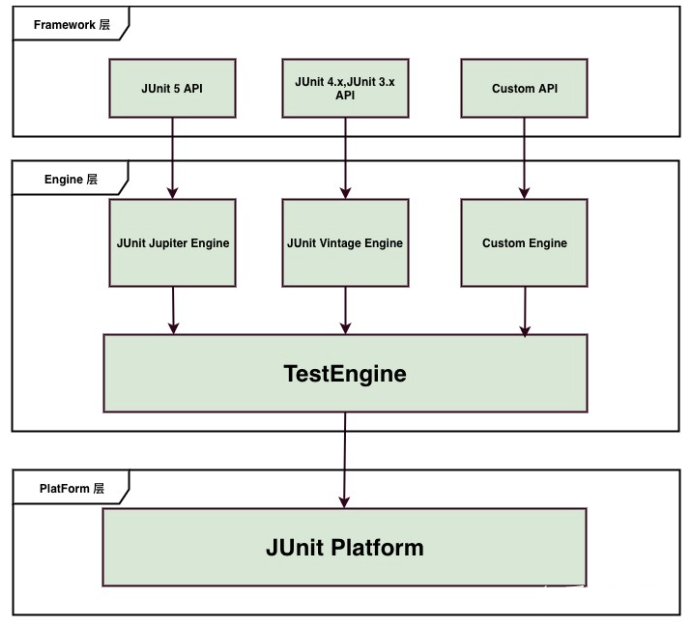
# Junit5

## 基本介绍

和Junit4一样，Junit5也是用来写测试用例的，相对junit4，提供了更多更方便的使用方法；

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

* JUnit Platform: Junit Platform是在JVM上启动测试框架的基础，不仅支持Junit自制的测试引擎，其他测试引擎也都可以接入。
* JUnit Jupiter: JUnit Jupiter提供了JUnit5的新的编程模型，是JUnit5新特性的核心。内部 包含了一个测试引擎，用于在Junit Platform上运行。
* JUnit Vintage: 由于JUint已经发展多年，为了照顾老的项目，JUnit Vintage提供了兼容JUnit4.x,Junit3.x的测试引擎。



官方地址：<https://junit.org/junit5/>

官方文档：<https://junit.org/junit5/docs/current/user-guide/>

## 核心注解

Junit5相对于Junit4提供了一套新的注解，都在org.junit.jupiter.api包下面，核心注解如下：

### 创建测试方法

* @Test：用来标志这个方法是一个测试方法，和Junit4不同的是，这个注解不再提供任何属性；
* @ParameterizedTest：标志方法是一个参数化测试方法；
* @RepeatedTest：标志方法是一个可重复测试方法
* @TestFactory：测试工厂，用来动态生成测试的

### 生命周期注解

* @BeforeEach / @AfterEach：每个测试方法执行之前 / 之后
* @BeforeAll / @AfterAll：所有测试方法执行之前 / 之后

### 辅助类注解

* @Tag / @Tags：给测试类/方法打标签，用于分组和过滤
* @DisplayName：测试类或者测试方法的名字
* @DisplayNameGeneration：为测试指定一个生成测试名字的类
* @TestMethodOrder / @Order：@TestMethodOrder用在类上面让测试方法按照指定的顺序执行， @Order用于指定当前这个方法的顺序编号；

### 条件测试注解

* @Disabled：忽略掉某个测试类或者测试方法
* @EnabledOnOs / @DisabledOnOs：在指定的操作系统下启动测试 / 不进行测试
* @EnabledOnJre / @DisabledOnJre：在指定jdk版本下
* @EnabledIfSystemProperty / @DisabledIfSystemProperty：在某个系统属性满足某种条件的时候
* @EnabledIfEnvironmentVariable / @DisabledIfEnvironmentVariable：在某个环境变量满足某种条件的时候

### 参数化测试注解

* @NullSource：为测试方法传入null
* @EmptySource：传入空数据，如：java.lang.String, java.util.List, java.util.Set, java.util.Map, primitive arrays (e.g., int[], char[][], etc.), object arrays (e.g.,String[], Integer[][], etc. )
* @NullAndEmptySource：@NullSource和@EmptySource这两个注解的组合
* @ValueSource：可以为方法传入八种基本数据类型、java.lang.String类型、java.lang.Class类型。  
  方式为：ints={},strings={} 等等；
* @EnumSource：为测试传入枚举
* @MethodSource：通过方法来传入参数
* @CsvSource：通过Csv格式传入一组参数
* @CsvFileSource：通过csv文件注入参数

### 其他注解

* @Nested：内嵌测试，用于把同一类测试放在一个内部类中。
* @Timeout：表名测试方法执行超时时间
* @ExtendWith：
* @RegisterExtension：
* @TempDir：

## 核心类、方法

### Assertions（断言）

Assertions提供了一些断言方法，这些方法内部调用org.junit.jupiter.api.AssertXXX 等类（如：AssertTrue、AssertNull等）里面的一些方法，方法如下：

* assertTrue / assertFalse：断言某个表达式的结果是正确/错误的
* assertNull / assertNotNull：断言某个对象是null / 非null
* assertEquals / assertNotEquals：断言某两个数据是相同/不同的，对象调用equals方法
* assertArrayEquals：断言两个数组是的原始是相同的，对象元素对比使用的是equlas方法
* assertIterableEquals：断言两个集合相同，调用每个元素的equals
* assertLinesMatch：判断指定的字符串集合是否能满足匹配对应的正则表达式，一一对应的。
* assertSame / assertNotSame：断言两个对象一样，对象比较的是地址
* assertAll： 用来归类一组断言，不管前面的断言是否成功，都会继续后面的
* assertThrows / assertDoesNotThrow：断言会抛出指定异常 / 断言不会抛出异常

### Assumptions（假设）

Assumptions类似于Junit4的Assume，用于在某些条件不成立的时候忽略掉测试，从而避免一些问题，有如下几个方法：

* assumeTrue：假设条件成立的时候执行后续
* assumeFalse：假设条件不成立的时候执行后续操作
* assumingThat：假设条件成立的时候执行给定操作

## 引入Junit5

## 常用场景

### 简单使用

我们只需要在某个方法上添加@Test，这个方法就变成了单元测试方法。除此之外，我们还可以通过在方法上面添加@RepeatedTest, @ParameterizedTest, @TestFactory, or @TestTemplate.来把这个方法变成单元

|  |
| --- |
| /\*\*  \* 描述：  \* Author：liuxing  \* Date：2020/6/19  \*/ public class *HelloWorld* {   @Test  public void helloworld() {  *System*.***out***.println("这是juint5");  } } |

### 断言

|  |
| --- |
| package *com.firewolf.testcase.junit5.s01\_assertions*;  import *org.junit.jupiter.api.*Test;  import *java.util.Arrays*; import *java.util.List*;  import static *org.junit.jupiter.api.Assertions*.\*;  /\*\*  \* 描述：  \* Author：liuxing  \* Date：2020/6/19  \*/ public class *AssertionsTest* {    /\*\*  \* assertTrue / assertFalse：断言某个表达式的结果是正确/错误的  \*/  @Test  public void testAssertTrue\_False() {  boolean flag = 1 > 3;  assertTrue(!flag);  assertFalse(flag);  }    /\*\*  \* assertNull / assertNotNull ： 断言某个对象是null / 非null  \*/  @Test  public void testAssertNull\_NotNull() {  *Object* o = null;  *Object* o2 = new Object();  assertNull(o);  assertNotNull(o2);  }   /\*\*  \* assertEquals / assertNotEquals：断言某两个数据是相同/不同的，对象调用equals方法  \*/  @Test  public void testAssertEquals\_NotEquals() {  assertEquals("aaa", "aaa");  assertNotEquals("aaa", "bbb");  }    /\*\*  \* assertArrayEquals：断言两个数组是的原始是相同的，对象元素对比使用的是equlas方法  \*/  @Test  public void testAssertArrayEquals() {  assertArrayEquals(new int[]{1, 2, 3}, new int[]{1, 2, 3});  }    /\*\*  \* assertIterableEquals：断言两个集合相同，调用每个元素的equals  \*/  @Test  public void testAssertIterableEquals() {  assertIterableEquals(*Arrays*.asList(1, 2, 3), *Arrays*.asList(1, 2, 3));  }   /\*\*  \* 使用正则匹配，第一个参数是正则表达式列表，第二个是要用来被匹配的字符串列表，只有在正则表达式数量和字符串数量一致且对应位置的正则能匹配上字符串的时候，才会返回true  \*/  @Test  public void testAssertLinesMatch() {  *List*<*String*> regexs = *Arrays*.asList("^\\d.\*\_$", "^\\d+$");  *List*<*String*> values = *Arrays*.asList("23ss\_", "1234");  assertLinesMatch(regexs, values);  }    /\*\*  \* assertSame/ assertNotSame：判断两个数据是否相同，==判断  \*/  @Test  public void testSame\_NotSame() {  assertSame("123", "123");  assertNotSame("123", new String("123"));   }    /\*\*  \* assertAll： 用来归类一组断言，不管前面的断言是否成功，都会继续后面的  \*/  @Test  public void testAssertAll() {  assertAll(  () -> assertEquals(1, 2),  () -> assertNull(null)  );  }   /\*\*  \* assertThrows / assertDoesNotThrow：断言会抛出指定异常 / 断言不会抛出异常  \*/  public void testAssertThrows\_NotThrows() {  assertThrows(*RuntimeException*.class, () -> {  int a = 1 / 0;  });   assertDoesNotThrow(() -> {  int a = 1 + 2;  });  } } |

### 生命周期方法

可以通过@BeforeAll, @AfterAll, @BeforeEach, or @AfterEach等注解来声明一些生命周期方法

|  |
| --- |
| package *com.firewolf.testcase.junit5.s02\_lifecycle*;  import *org.junit.jupiter.api.*\*;  /\*\*  \* 描述：生命周期方法，需要依赖@BeforeAll, @AfterAll, @BeforeEach, or @AfterEach等注解  \* Author：liuxing  \* Date：2020/6/19  \*/ public class *LifeCycleTest* {   /\*\*  \* 在所有测试方法执行之前  \*/  @BeforeAll  public static void beforeAll() {  *System*.***out***.println("before All");  }   /\*\*  \* 在所有测试方法执行之后  \*/  @AfterAll  public static void afterAll() {  *System*.***out***.println("after all");  }   /\*\*  \* 在每个测试方法执行前执行  \*/  @BeforeEach  public void beforeEach() {  *System*.***out***.println("before each");  }   /\*\*  \* 在每个测试方法执行之后执行  \*/  @AfterEach  public void afterEach() {  *System*.***out***.println("after each");  }   @Test  public void test1() {  *System*.***out***.println("test1");  }   @Test  public void test2() {  *System*.***out***.println("test2");  } } |

### 给测试取名

我们可以给测试类或者测试方法、内部测试类、内部测试类的测试方法取名。

1. 使用@DisplayName注解

@DisplayName可以用于类上或者是方法上，分别是给测试类取名或者是给测试方法取名

|  |
| --- |
| @DisplayName("这是一个特殊测试") public class *DisplayNameTest* {   @DisplayName("测试1")  @Test  public void test1() {  *System*.***out***.println("test1");  }  } |

1. 使用@DisplayNameGeneration注解

可以通过给类上面添加@DisplayNameGeneration注解来指定一个测试名字生成器，方法如下：

|  |
| --- |
| package *com.firewolf.testcase.junit5.s03\_displayname*;  import *org.junit.jupiter.api.*DisplayNameGeneration; import *org.junit.jupiter.api.DisplayNameGenerator*; import *org.junit.jupiter.api.*Nested; import *org.junit.jupiter.api.*Test;  import *java.lang.reflect.Method*;  /\*\*  \* 描述：  \* Author：liuxing  \* Date：2020/6/19  \*/ @DisplayNameGeneration(*MyDisplayNameGenerator*.class) public class *DisplayNameGeneratorTest* {   @Test  public void testAAA() {  *System*.***out***.println("AAA");  }   @Nested  class *A* {  @Test  public void testA\_123() {  *System*.***out***.println("testA\_123");  }   @Test  public void testB\_123() {  *System*.***out***.println("testB\_123");  }  } }  class *MyDisplayNameGenerator* implements *DisplayNameGenerator* {   /\*\*  \* 类名  \*  \* @param testClass  \* @return  \*/  @Override  public *String* generateDisplayNameForClass(*Class*<?> *testClass*) {  return "这是一个测试类：" + *testClass*.getSimpleName();  }   /\*\*  \* 内部测试类的名字  \*  \* @param nestedClass  \* @return  \*/  @Override  public *String* generateDisplayNameForNestedClass(*Class*<?> *nestedClass*) {  return "这是内部测试类：" + *nestedClass*.getSimpleName();  }   /\*\*  \* 各个方法的名字  \*  \* @param testClass  \* @param testMethod  \* @return  \*/  @Override  public *String* generateDisplayNameForMethod(*Class*<?> *testClass*, *Method testMethod*) {  return "这是" + *testClass*.getSimpleName() + "类的" + *testMethod*.getName() + "测试方法";  } } |

### 假设

|  |
| --- |
| package *com.firewolf.testcase.junit5.s04\_assumptions*;  import *org.junit.jupiter.api.*Test;  import static *org.junit.jupiter.api.Assumptions*.\*;  /\*\*  \* 描述：  \* Author：liuxing  \* Date：2020/6/20  \*/ public class *AssumptionsTest* {    @Test  public void test() {  int a = 1;   // 表达式为true的时候执行后续  assumeTrue(a == 1);  *System*.***out***.println("计算结果正确");   // 表达式为false的时候执行后续  assumeFalse(a % 2 == 0);  *System*.***out***.println("结果是奇数");    // 当前面的条件成立的时候执行后面的  assumingThat(a < 10, () -> {  *System*.***out***.println("数字小于10");  });  }  } |

### 忽略测试

有时候，我们可能由于没有bug没有修复导致测试执行不过，而会影响整个测试，整个时候，我们可以选择性的不执行某个测试类或者测试方法，我们通过@Disabled 可以实现这个功能。

|  |
| --- |
| package *com.firewolf.testcase.junit5.s05\_disable*;  import *org.junit.jupiter.api.*Disabled; import *org.junit.jupiter.api.*Test;  /\*\*  \* 描述：  \* Author：liuxing  \* Date：2020/6/20  \*/ //@Disabled public class *DisableTest* {   @Test  @Disabled("有Bug还没有修复，先忽略")  public void test1() {  *System*.***out***.println("test1");  }   @Test  public void test2() {  *System*.***out***.println("test2");  } } |

### 条件测试

可以在满足某种条件的时候进行测试或者在满足什么条件的时候不进行测试；有如下几种条件测试：

* 在指定的操作系统下
* 在指定jdk版本下
* 在某个系统属性满足某种条件的时候
* 在某个环境变量满足某种条件的时候
* 自定义条件测试

需要注意的是，目前来说，这些注解虽然可以用在类上面，但是取没有任何效果

|  |
| --- |
| public class *ConditionTest* {    @Test  @EnabledOnOs(*OS*.***WINDOWS***) // 在windows环境下执行  @EnabledOnJre(*JRE*.***JAVA\_8***)  @EnabledIfSystemProperty(named = "os.name", matches = ".\*10.\*") // 要求win10  public void test1() {  *System*.***out***.println("这是windows10下的java8环境");  }   /\*\*  \* 如果环境变量中有special=true的时候，就禁用测试  \*/  @Test  @DisabledIfEnvironmentVariable(named = "special", matches = "true")  public void test2() {  *System*.***out***.println("这是特殊测试");  }   /\*\*  \* 自定义注解  \*/  @Test  @EnableOnSaturday  public void testSelf() {  *System*.***out***.println("xxxx");  } }  /\*\*  \* 自定义Condition注解，参考EnabledOnJre进行实现  \*/ @Target({*ElementType*.***TYPE***, *ElementType*.***METHOD***}) @Retention(*RetentionPolicy*.***RUNTIME***) @Documented @ExtendWith(*EnableOnSelfCondition*.class) @interface EnableOnSaturday { }  class *EnableOnSelfCondition* implements *ExecutionCondition* {  private static final *ConditionEvaluationResult* ***ENABLED\_BY\_DEFAULT*** = enabled("@EnableOnSaturday is not present");   static final *ConditionEvaluationResult* ***ENABLED\_ON\_SATURDAY*** = enabled("Enabled on Saturday ");   static final *ConditionEvaluationResult* ***DISABLED\_ON\_SATURDAY*** = disabled("Disabled on Saturday ");   @Override  public *ConditionEvaluationResult* evaluateExecutionCondition(*ExtensionContext context*) {  *Optional*<EnableOnSaturday> optional = findAnnotation(*context*.getElement(), EnableOnSaturday.class);  if (optional.isPresent()) {  *LocalDateTime* now = *LocalDateTime*.now();  *DayOfWeek* dayOfWeek = now.getDayOfWeek();  if (dayOfWeek == *DayOfWeek*.***SATURDAY***) {  return ***ENABLED\_ON\_SATURDAY***;  } else {  return ***DISABLED\_ON\_SATURDAY***;  }  }  return ***ENABLED\_BY\_DEFAULT***;  } } |

### 标签与过滤

1. 对测试分组

我们可以通过给测试用例打标签的方式来对测试进行分组，之后，在运行项目的所有测试的时候，可以指定哪些组的需要运行，哪些不需要。

|  |
| --- |
| public class *TagTest* {    @Tag("test")  @Test  void test1() {  *System*.***out***.println("test1");  }   @Tags({  @Tag("dev"),  @Tag("test")  })  @Test  void test2() {  *System*.***out***.println("test2");  }    @Dev  void test3() {  *System*.***out***.println("test3");  } } |

1. 自定义Tag

可以自定义Tag如下：

|  |
| --- |
| // 自定义Tag标签，可以通过集成@Tag和@Test，让我们的测试方法或者类上面去掉不需要的@Test @Target({*ElementType*.***TYPE***, *ElementType*.***METHOD***}) @Retention(*RetentionPolicy*.***RUNTIME***) @Tag("dev") @Test @interface Dev {  } |

之后，直接在测试方法上面添加@Dev即可。

需要注意的是，如果类上面添加了@Tag或者@Tags，那么方法上面的这些注解就会失效。

1. 分组运行

如果我们需要进行分组运行，需要给项目引入插件：

|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>3.0.0-M3</version> </plugin> |

之后有两种方式来指定要运行的测试组以及要跳过的测试组：

1. 配置插件

|  |
| --- |
| <configuration>  <!-- include tags -->  <groups>dev</groups>  <!-- exclude tags -->  <excludedGroups>test</excludedGroups>  </configuration> |

1. 命令行参数

Mvn clean test -Dgroups="dev" -DexcludedGroups="test"

### 测试顺序

我们可以执行测试方法的执行顺序，通常用于互相之间有依赖的，这依赖于@TestMethodOrder注解。

|  |
| --- |
| @TestMethodOrder(*MethodOrderer*.*OrderAnnotation*.class) public class *OrderTest* {  @Test  @Order(1)  public void test1(){  *System*.***out***.println("test1");  }  @Test  @Order(4)  public void test2(){  *System*.***out***.println("test2");  }  @Test  @Order(3)  public void test3(){  *System*.***out***.println("test3");  } } |

@Order中数字越大，则越后执行。

MethodOrderer有三个实现：

* OrderAnnotation：根据@Order注解中的顺序来执行
* Alphanumeric：按照方法名的顺序，通过compareTo对比出来的
* Random：随机执行

### 内嵌测试类

我们通过在测试类的内部再创建一个测试类的方式，来把同一类的测试放在一起，然后通过@Nested注解让内容测试类的方法也都被测试

|  |
| --- |
| public class *NestedTest* {   @Test  void test1() {  *System*.***out***.println("test1");  }   @Nested  class *NestedTests*{  @Test  void test1(){  *System*.***out***.println("nested test1");  }  }  } |

### 超时时间

可以通过在测试类或者测试方法上面添加@TimeOut注解来告诉junit这个测试的时间不能超过多久，用于类上面，表示这个类下面的所有方法都不能超过指定时间。

|  |
| --- |
| public class *TimeOutTest* {  @Test  @Timeout(value = 1, unit = *TimeUnit*.***SECONDS***)  void test() throws *InterruptedException* {  *Thread*.sleep(2000);  } } |

* 全局禁用超时

可以全局禁用超时时间的控制，需要设置启动参数：junit.jupiter.execution.timeout.mode=disabled，可以设置的值有：disabled/disabled/disabled\_on\_debug可以通过maven插件或者控制台的方式指定，

Maven插件配置：

|  |
| --- |
| <configuration>  <properties>  <configurationParameters>  junit.jupiter.execution.timeout.mode=disabled  </configurationParameters>  </properties> </configuration> |

### 给测试注入内置参数

我们可以测试类的构造器以及测试方法传入一些内置的参数，这主要有：

* TestInfo：测试信息
* TestReporter：测试汇报
* RepetitionInfo：重复测试信息，只能用于@RepeatedTest

|  |
| --- |
| public class *InjectParams* {   InjectParams(*TestInfo testInfo*) {  *System*.***out***.println("构造器："+*testInfo*.getDisplayName());  }   /\*\*  \* 测试信息TestInfo：  \* @param testInfo  \*/  @Test  @DisplayName("TEST 1")  @Tag("my-tag")  void test1(*TestInfo testInfo*) {  assertEquals("TEST 1", *testInfo*.getDisplayName());  assertTrue(*testInfo*.getTags().contains("my-tag"));  }   /\*\*  \* 注入测试报告对象，允许我们对测试报告进行更改  \* @param testReporter  \*/  @Test  void test2(*TestReporter testReporter*) {  *testReporter*.publishEntry("a key", "a value");  }   /\*\*  \* 重复测试参数  \* @param repetitionInfo  \*/  @RepeatedTest(5)  void test3(*RepetitionInfo repetitionInfo*) {  assertEquals(5, *repetitionInfo*.getTotalRepetitions());  } } |

### 重复执行

有时候我们需要一个方法多执行几次才能达到我们测试的目的，比如定时任务等。  
我们可以使用@RepeatedTest来完成这个功能，可以通过如下属性来进行设置：

* value：重复次数，必填
* name：这次执行的名字，里面可以用到三个定义好的占位符：
* {displayName}：测试的展示名，如果方法上面使用了@DisplayName注解的话，就会是这个注解里面的值，否则显示为方法名
* {currentRepetition} ：当前执行的次数
* {totalRepetitions}：总次数

|  |
| --- |
| public class *RepeatTestDriver* {    @DisplayName("repeat test")  @RepeatedTest(value = 4, name = "执行测试： {displayName}, 第 {currentRepetition} / {totalRepetitions} 次！ ")  void testRepeat(*TestInfo testInfo*) {  assert !*testInfo*.getDisplayName().contains("3");  } } |

### 参数化测试

有时候我们希望给测试用例传入我们需要的参数，这个时候，我们可以使用下面的一系列注解来完成这个事情注解来完成这个需求。

#### **开启参数化测试**

开启参数化测试需要两个步骤：

1. 引入依赖

|  |
| --- |
| <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-params</artifactId>  <version>5.6.2</version>  <scope>test</scope> </dependency> |

1. 如果我们希望给方法传入参数，可以通过标注@ParameterizedTest注解

作用：标注这是一个带参数的单元测试；

参数：

* name：每一个参数对应的测试方法名字，可以使用如下占位符：{index}（第Index个参数）、{arguments}（当前这组参数），{0}/{1}/…（这组参数的第几个参数）、{displayName}：方法名

#### 简单参数

1. 传入空值

* 可以通过@NullSource传入null；
* 通过@EmptySource传入空值，java.lang.String, java.util.List, java.util.Set, java.util.Map, primitive arrays (e.g., int[], char[][], etc.), object arrays (e.g.,String[], Integer[][], etc.)。
* @NullAndEmptySource相当于上面两个注解的组合

1. 数组传入数据（@ValueSource）

参数：

* ints：传入一组整数

…：这些参数和ints类似，可以传入八种基本数据类型、java.lang.String类型、java.lang.Class类型。  
方式为：ints={},strings={} 等等；

1. 传入枚举（@EnumSource）

参数：

* value：枚举类型
* names：关心的枚举集合。
* mode：对names的处理方式，默认是INCLUDE，也就是传入names指定的枚举，也可以使用EXCLUDE来排除指定的枚举；

|  |
| --- |
| @ParameterizedTest // 传入三个字符串 @ValueSource(strings = {"haha", "hehe", "heihei"}) // 传入null和"" @NullAndEmptySource void testValueSourceStringParams(*String str*) {  *System*.***out***.println("|" + *str* + "|"); }  /\*\*  \* 传入枚举  \*  \* @param unit  \*/ @ParameterizedTest @EnumSource(value = *ChronoUnit*.class, names = {"SECONDS", "DAYS"}, mode = EnumSource.*Mode*.***INCLUDE***) void testEumSource(*ChronoUnit unit*) {  assertNotNull(*unit*); } |

#### **方法生成参数**

可以通过@MethodSource注解来指定一个方法来参数

参数：

* value：提供参数方法名，如果方法是当前类的，就直接给方法名，如果是其他类的，需要使用类的包名#方法名形式传入，如：com.firewolf.busi.example.ParamTestDriver#provider

要求：方法必须是静态类型，且方法返回的必须是有个Stream类型，如：IntStream、Stream等等。

|  |
| --- |
| @ParameterizedTest @MethodSource("com.firewolf.testcase.junit5.s13\_parameterized.ParameterizedTestDriver#provider") void testMethodParams(*ChronoUnit chronoUnit*) {  *System*.***out***.println(*chronoUnit*); }  static *Stream*<*ChronoUnit*> provider() {  return *Stream*.of(*ChronoUnit*.***HALF\_DAYS***, *ChronoUnit*.***DAYS***,*ChronoUnit*.***FOREVER***); } |

#### **csv格式传入参数**

通过@CsvSource注解可以把csv格式的文本来为方法传入一组参数，会被自动解析，

参数：

* value：参数
* delimiter：各个值之间的分隔符，默认为’,’

注意点：

* 多个单词的参数：如果某项数据里面包含了分隔符，那么需要使用’'括起来；
* 传入null：需要想传入null，那么这一项不写即可；
* 如果数据项多于参数，那么后面的会被丢弃

|  |
| --- |
| @ParameterizedTest @CsvSource(value = {  "apple ; 2; heihei",  " ; 1; heihei",  "'lemon; lime'; 2; haha",  "nal; 0xF1; hehe" }, delimiter = ';') void testWithCsvSource(*String fruit*, int *rank*) {  *System*.***out***.println("fruit=" + *fruit* + ",rank=" + *rank*);  assertNotEquals(0, *rank*); } |

#### **csv文件传入参数**

可以通过@CsvFileSource注解把一个csv文件作为参数给测试方法传入一组参数

参数：

* value：参数
* delimiter：各个值之间的分隔符，默认为’,’
* resources：文件，这个文件需要放在src/test/resources/下面，然后文件路径以/文件名的形式
* lineSeparato：换行符，默认为\n
* encoding：文件编码，默认为utf-8

注意事项：如果文件中的某项数据包含了分隔符，那么需要使用""来引用起来

|  |
| --- |
| @ParameterizedTest @CsvFileSource(resources = "/test.csv", delimiter = ';') void testWithCsvFileSource(*String fruit*, int *rank*) {  *System*.***out***.println("fruit=" + *fruit* + ",rank=" + *rank*);  assertNotEquals(0, *rank*);  assertNotEquals(0, *rank*); } |

#### **ArgumentsAccessor（参数转换器）**

我们可以把csv传入的参数转成我们需要的类型，然后作为测试用例的参数传入方法。主要有两种方式：

1. ArgumentsAccessor  
   给测试用例传入一个ArgumentsAccessor类型的参数，然后在方法里面进行封装：

|  |
| --- |
| @ParameterizedTest @CsvSource({  "Jane, Doe, F, 1990-05-20",  "John, Doe, M, 1990-10-22" }) void testWithArgumentsAccessor(*ArgumentsAccessor arguments*) {  *Person* person = new Person(*arguments*.getString(0),  *arguments*.getString(1),  *arguments*.get(2, *String*.class),  *arguments*.get(3, *LocalDate*.class));   if (person.getFirstName().equals("Jane")) {  assertEquals("F", person.getGender());  } else {  assertEquals("M", person.getGender());  }  assertEquals("Doe", person.getLastName());  assertEquals(1990, person.getDateOfBirth().getYear()); } |

1. 自定义 ArgumentsAccessor

我们可以实现自己的ArgumentsAccessor，这个类需要实现接口ArgumentsAggregator ；然后使用@AggregateWith注解来指定我们自己定义的转换器。

|  |
| --- |
| @ParameterizedTest @CsvSource({  "Jane, Doe, F, 1990-05-20",  "John, Doe, M, 1990-10-22" }) void testWithArgumentsAccessor2(@AggregateWith(*PersonArgumentsAccessor*.class) *Person person*) {  if (*person*.getFirstName().equals("Jane")) {  assertEquals("F", *person*.getGender());  } else {  assertEquals("M", *person*.getGender());  }  assertEquals("Doe", *person*.getLastName());  assertEquals(1990, *person*.getDateOfBirth().getYear()); } |
| class *PersonArgumentsAccessor* implements *ArgumentsAggregator* {  @Override  public *Person* aggregateArguments(*ArgumentsAccessor arguments*, *ParameterContext parameterContext*) throws *ArgumentsAggregationException* {  *Person* person = new Person(*arguments*.getString(0),  *arguments*.getString(1),  *arguments*.get(2, *String*.class),  *arguments*.get(3, *LocalDate*.class));   return person;  } } |

### 动态测试

我们可以通过@TestFactory来动态的生成一些测试，要求方法返回：单个DynamicNode 或者是DynamicNode的Stream, Collection, Iterable, Iterator，DynamicNode有一个常用的实现DynamicTest，我们可以通过DynamicContainer的一些方法来生成DynamicContainer.

|  |
| --- |
| package *com.firewolf.testcase.junit5.s14\_dynamic*;  import *org.junit.jupiter.api.DynamicNode*; import *org.junit.jupiter.api.DynamicTest*; import *org.junit.jupiter.api.*TestFactory;  import *java.util.Arrays*; import *java.util.Collection*; import *java.util.List*; import *java.util.stream.Stream*;  import static *org.junit.jupiter.api.DynamicContainer*.dynamicContainer; import static *org.junit.jupiter.api.DynamicTest*.\*; import static *org.junit.jupiter.api.Assertions*.\*;  /\*\*  \* 描述：动态测试，通过@TestFactory来生成测试，要求方法返回：单个DynamicNode 或者是DynamicNode的Stream, Collection, Iterable, Iterator  \* Author：liuxing  \* Date：2020/6/22  \*/ public class *DynamicTestDriver* {   /\*\*  \* 返回的不是DynamicNode，会抛出异常  \*  \* @return  \*/  @TestFactory  *List*<*String*> dynamicTestsWithInvalidReturnType() {  return *Arrays*.asList("Hello");  }    /\*\*  \* 创建单个测试  \*  \* @return  \*/  @TestFactory  *DynamicTest* dynamicNodeSingleTest() {  int a = 1;  return dynamicTest("dynamicNodeSingleTest", () -> assertTrue(*a* == 1));  }   /\*\*  \* 一组测试，可以生成Collection、Iterator、Iterable、数组、Stream等类型的都可以  \*  \* @return  \*/  @TestFactory  *Collection*<*DynamicNode*> dynamicTestsFromCollection() {  return *Arrays*.asList(  dynamicTest("1st dynamic test", () -> *System*.***out***.println("dynamicTestsFromCollection")),  dynamicTest("2nd dynamic test", () -> *System*.***out***.println("dynamicTestsFromCollection"))  );  }   /\*\*  \* 只有单个测试的测试容器，里面返回一组测试  \* @return  \*/  @TestFactory  *DynamicNode* dynamicNodeSingleContainer() {  return dynamicContainer("palindromes",  *Stream*.of("racecar", "radar", "mom", "dad")  .map(*text* -> dynamicTest(*text*, () -> assertTrue(*text*.length() > 2))  ));  }   /\*\*  \* 测试容器，可以理解为二级目录  \*  \* @return  \*/  @TestFactory  *Stream*<*DynamicNode*> dynamicTestsWithContainers() {  return *Stream*.of("A", "B", "C")  .map(*input* -> dynamicContainer("Container " + *input*, *Stream*.of(  dynamicTest("not null", () -> assertNotNull(*input*)),  dynamicContainer("properties", *Stream*.of(  dynamicTest("length > 0", () -> assertTrue(*input*.length() > 0)),  dynamicTest("not empty", () -> assertFalse(*input*.isEmpty()))  ))  )));  }   } |

### 支持Juint4

Junit5对junit4也提供了一些支持，比如：Junit4中的Rule以及@Ignore都不见了，那么我们可以通过下面的方式来使用。

#### **引入Junit集成依赖**

|  |
| --- |
| <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-migrationsupport</artifactId>  <version>5.6.2</version>  <scope>test</scope> </dependency> |

#### **使用Rule**

**一旦引入了上面的依赖，我们就可以正常的使用Junit4中的Rule功能了， 官方说需要在类上面添加注解@EnableRuleMigrationSupport，经测试，不加也可以使用**

# 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

# 规范建议

### 测试类所在位置

建议测试类的包路径和被测试类的包路径保持一致；

### 测试方法名

和原始方法名保持一致，如果是重载方法，建议按顺序加上编号即可。