目錄

Java单元测试学习笔记

介绍

TDD和BDD是日常开发中比不可少的一个重要环节。

这份Java测试学习笔记, 记录在Java日常开发中做单元测试的基本知识和常用技巧.

导航

如果看到的是github的源代码, 请点击下面的链接:

Java测试学习笔记@gitbook

这里是一个tdd,bdd集成的 demo

JUnit

JUnit是一个Java语言的单元测试框架。它由Kent Beck和Erich Gamma建立，逐渐成为源于Kent Beck的sUnit的xUnit家族中最为成功的一个。JUnit有它自己的JUnit扩展生态圈。多数Java的开发环境都已经集成了JUnit作为单元测试的工具。

资料收集

- 官网
- 代码仓库

Maven依赖

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

Gradle

```
apply plugin: 'java'

dependencies {
  testCompile 'junit:junit:4.12'
}
```

用suites做套件测试

使用Suite作为runner,你可以手动创建一个包含多个类的测试套件.相当于JUint 3.8.x的 *static test suite()*方法.使用*@RunWith(Suite.class)*和*@SuiteClasses(TestClass1.class, ...)*注解,当你运行该类的时候,它可以运行所有的SutieClasses指定的类

例子

如下所示,该类是suite注解的一个演示,它不需要实现其他内容.注意@RunWith注解,它指定了JUnit 4运行是使用*org.junit.runners.Suite*来执行这个特定的测试类,它结合@Suite注解,告诉Suite runner该套件包含哪些测试类及顺序。

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  TestFeatureLogin.class,
  TestFeatureLogout.class,
  TestFeatureNavigate.class,
  TestFeatureUpdate.class
})

public class FeatureTestSuite {
  // the class remains empty,
  // used only as a holder for the above annotations
}
```

测试执行顺序

在设计上Junit是没有指定测试方法的执行顺序的。到目前为止,方法的调用也只是根据反射API返回的顺序执行.然而,使用JVM的顺序是不明智的，因为JAVA不提供任何平台的执行顺序,而且JDK 7返回的是不确定的随机执行顺序.所以,你所写的测试代码通常也是没有执行顺序的.但是有的时候有预测的失败，比在某些平台上随机失败要好很多

从4.11版本,JUnit 将默认使用一些指定的顺序,但是不能确定它使用哪种顺序（MethodSorters.DEFAULT）.要改变执行顺序,只需要简单的在你的测试类中使用*@FixMethodOrder*注解并且指定可用的*MethodSorters*,如下:

- @FixMethodOrder(MethodSorters.JVM):由JVM返回的顺序执行方法,该方法可能每次的执行顺序都不同
- @FixMethodOrder(MethodSorters.NAME_ASCENDING):按照测试方法名在字典中的排序执行

例子

```
package com.junit.learning.order;

import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runners.MethodSorters;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class TestMethodOrder {
    @Test
    public void testA() {
        System.out.println("first");
    }
    @Test
    public void testB() {
        System.out.println("second");
    }
    @Test
    public void testC() {
        System.out.println("third");
    }
}
```

上面的例子将按照方法名在字典中的排序按照升序执行

测试异常

预期异常

你如何校验你的代码按照预期抛出异常?完成代码通常是比较简单的,但要确保代码的行为在特殊情况下与预期相同是至关重要的。举个栗子:

```
new ArrayList<Object>().get(0);
```

这段代码会抛出一个IndexOutOfBoundsException。*@Test*注解有一个可选参数"expected",它的值是*Throwable*的子类,如果我们想要验证*ArrayList*抛出一个正确的异常,我们只需要写下面这段代码:

```
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
     new ArrayList<Object>().get(0);
}
```

expected参数应该需要小心使用,上述代码只要有任何地方抛出IndexOutOfBoundsException就会通过测试,

try/cach 语法

在JUnit 3.X版本中只能使用try/cach来测试异常

```
@Test
public void testExceptionMessage() {
    try {
        new ArrayList<Object>().get(0);
        fail("Expected an IndexOutOfBoundsException to be thrown"
);
    } catch (IndexOutOfBoundsException anIndexOutOfBoundsExcepti
on) {
        assertThat(anIndexOutOfBoundsException.getMessage(), is(
"Index: 0, Size: 0"));
    }
}
```

ExpectedException规则

另外,你也可以使用ExpectedException规则,这个规则是表示你不仅关心有异常抛出,而且也关心异常信息：

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void shouldTestExceptionMessage() throws IndexOutOfBounds
Exception {
    List<Object> list = new ArrayList<Object>();

    thrown.expect(IndexOutOfBoundsException.class);
    thrown.expectMessage("Index: 0, Size: 0");
    list.get(0); // execution will never get past this line
}
```

expectMessage中还可以使用匹配器,使得在测试代码中多了一些灵活性

```
thrown.expectMessage(JUnitMatchers.containsString("Size: 0"));
```

此外，你可以使用匹配器来检查异常，如果有用的话它嵌入了要验证的状态。例如

```
import static org.hamcrest.Matchers.hasProperty;
import static org.hamcrest.Matchers.is;
import static org.hamcrest.Matchers.startsWith;

import javax.ws.rs.NotFoundException;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class TestExy {
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void shouldThrow() {
        TestThing testThing = new TestThing();
        thrown.expect(NotFoundException.class);
        thrown.expectMessage(startsWith("some Message"));
        thrown.expect(hasProperty("response", hasProperty("statu
s", is(404))));
        testThing.chuck();
    }

    private class TestThing {
        public void chuck() {
            Response response = Response.status(Status.NOT_FOUND
).entity("Resource not found").build();
            throw new NotFoundException("some Message", response
);
        }
    }
}
```

对于规则的ExpectedException的扩展讨论，请参阅此博客

忽略测试

如果由于某种原因，你不想一个测试失败,你只是希望JUnit忽略它,暂时禁用该测试方法.

想要忽略Junit忽略一个测试方法,你可以修改该方法的实现,也可以删除@Test注解.但是这样做了JUnit不会再报告有这个测试.或者你可以在@Test前后添加*Ignore*注解.JUnit不会再跑加了@Ignore注解的方法,但是报告中会有统计忽略的方法数量

> 以上都是官网的翻译,亲自实验后发现它眞的好用,在实际开发中不仅可以做到以上所述，它跟@Test最大的区别在于:maven跑测试的适合添加了@ignore的测试不会跑,但是又不影响你手动跑该测试.删除@Test的方式，如果你想要跑这个测试还需要将@Test加回来

@Ignore可以记录一个测试被忽略的原因（可选参数）

```
@Ignore("Test is ignored as a demonstration")
@Test
public void testSame() {
    assertThat(1, is(1));
}
```

测试超时

如果你希望一个测试运行时间过长的情况下自动变成测试失败,有以下两种方式来实现：

@Test中的timeout参数(适用于测试方法)

您可以指定以毫秒为单位超时时间。如果超过了时间限制,它会由异常触发一个失败：

```
@Test(timeout=1000)
public void testWithTimeout() {
   ...
}
```

它通过单独起一条线程来实现的,如果测试运行超过了规定的超时时间,测试将会失败并且JUnit会中断线程运行测试.如果测试超时，执行的是一个可中断的操作，运行测试线程可以退出(如果该测试是一个无限死循环,运行这个测试的线程将永远运行,但是其他的测试也可以继续执行)

Timeout Rule(适用于测试类中的所有测试用例)

Timout Rule适用于一个测试类中所有的测试方法(共用一个超时时间).并会运行除了由timeout参数在单个测试上指定的其他任何超时时间（详情请看这里）

```
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

public class HasGlobalTimeout {
    public static String log;
    private final CountDownLatch latch = new CountDownLatch(1);

    @Rule
    public Timeout globalTimeout = Timeout.seconds(10); // 10 se
conds max per method tested

    @Test
    public void testSleepForTooLong() throws Exception {
        log += "ran1";
        TimeUnit.SECONDS.sleep(100); // sleep for 100 seconds
    }

    @Test
    public void testBlockForever() throws Exception {
        log += "ran2";
        latch.await(); // will block
    }
}
```

Timeout Rule指定的超时适用于整个测试类,包括@Before或者@After方法,如果测试方法是一个无限循环(或者其他原因不能响应中断),那么@After方法将不会被调用

另外需要注意

```
    package com.junit.learning.timeout;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.rules.Timeout;

    import java.util.concurrent.TimeUnit;

    public class HasGlobalTimeout2 {
        public static String log;
```

```
        @Rule
        public Timeout globalTimeout = Timeout.seconds(10); // 1
0 seconds max per method tested

        /**
         * 这个方法的执行到失败的时间是5秒,也就是说timeout参数比rule的时
间要小的话覆盖rule的时间
         */
        @Test(timeout = 1000 * 5)
        public void testSleepForTooLong() throws Exception {
            log += "ran1";
            for (int i = 0; i >= 0; i++) {
                System.out.println(i);
                TimeUnit.SECONDS.sleep(1);
            }
            // sleep for 100 seconds
        }

        /**
         * 这个方法的执行到失败的时间是10秒,也就是说timeout参数比rule的
时间要大的话没法覆盖rule
         */
        @Test(timeout = 1000 * 20)
        public void testSleepForTooLong2() throws Exception {
            log += "ran1";
            for (int i = 0; i >= 0; i++) {
                System.out.println(i);
                TimeUnit.SECONDS.sleep(1);
            }
            // sleep for 100 seconds
        }
    }
```

Assume

> 直译为假设

它实际是对方法的参数进行合法性校验的，如果校验不合格则直接抛异常，而不执行测试，默认的BlockJUnit4ClassRunner及其子类会捕获这个异常并跳过当前测试，如果使用自定义的Runner则无法保证行为，视Runner的实现而定。如果在@Before或@After中定义Assume,将作为类的所有@Test方法都设置了Assume

如果有的时候必须规定具备某个条件才允许测试，但又不判断为fail,则可以使用：

```
package com.junit.learning.assume;

import org.junit.Test;

import java.io.File;

import static org.assertj.core.api.Assertions.assertThat;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assume.assumeThat;

public class AssumeTest {
    @Test
    public void filename(){
        assumeThat(File.separatorChar, is('/'));
        assertThat(true).isTrue();
    }


    @Test
    public void filenameFail(){
        assumeThat(File.separatorChar, is('\\'));
        assertThat(true).isTrue();
    }
}
```

参数测试

Parameterized Runner实现了参数测试,当跑一个参数测试类的时候,它的实例是由测试方法和测试数据的元素交叉创建的

举例,如下**Fibonacci**方法**:**

```
    package com.junit.learning.parameterized;

    import org.junit.Test;
    import org.junit.runner.RunWith;
    import org.junit.runners.Parameterized;

    import java.util.Arrays;
    import java.util.Collection;

    import static org.assertj.core.api.Assertions.assertThat;

    @RunWith(Parameterized.class)
    public class FibonacciTest {

        @Parameterized.Parameters
        public static Collection<Object[]> data() {
            return Arrays.asList(new Object[][]{
                    {0, 0}, {1, 1}, {2, 1}, {3, 2}, {4, 3}, {5, 5
}, {6, 8}
            });
        }

        private int fInput;

        private int fExpected;

        public FibonacciTest(int input, int expected) {
            fInput = input;
            fExpected = expected;
        }

        @Test
        public void test() {
            assertThat(fExpected).isEqualTo(Fibonacci.compute(fI
nput));
        }

    }
```

FibonacciTest的每个参数都由两个参数的构造方法和标记了@Parameters注解的方法中的值来创建. FibonacciTest运行测试类时的执行顺序如下:

1. 首先会初始化@Parameters注解标记的数据.
2. 然后循环这些数据
3. 每一个循环中的值都会传递（注入）到构造函数FibonacciTest(int input, int expected)中
4. test()中的断言

使用@Parameter注入属性（不再用构造函数）

也可以使用@Parameter注解将值注入到字段中,而不需要构造方法.如下

```
package com.junit.learning.parameterized;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

import java.util.Arrays;
import java.util.Collection;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(Parameterized.class)
public class FibonacciTest2 {

    @Parameterized.Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][]{
                {0, 0}, {1, 1}, {2, 1}, {3, 2}, {4, 3}, {5, 5}, {
6, 8}
        });
    }

    @Parameterized.Parameter // first data value (0) is default
    public /* NOT private */ int fInput;

    @Parameterized.Parameter(value = 1)
    public /* NOT private */ int fExpected;
    @Test
    public void test() {
        assertThat(fExpected).isEqualTo(Fibonacci.compute(fInput
));
    }

}
```

目前这种方式只适用于public字段(看这里)

单个参数测试

参数化测试

如果你只需要一个参数,你不需要用数组来包装它,相反,你可以提供一个可迭代数组或对象数组

```
@Parameters
public static Iterable<? extends Object> data() {
    return Arrays.asList("first test", "second test");
}
```

或者

```
@Parameters
public static Object[] data() {
    return new Object[] { "first test", "second test" };
}
```

识别**test case**个体

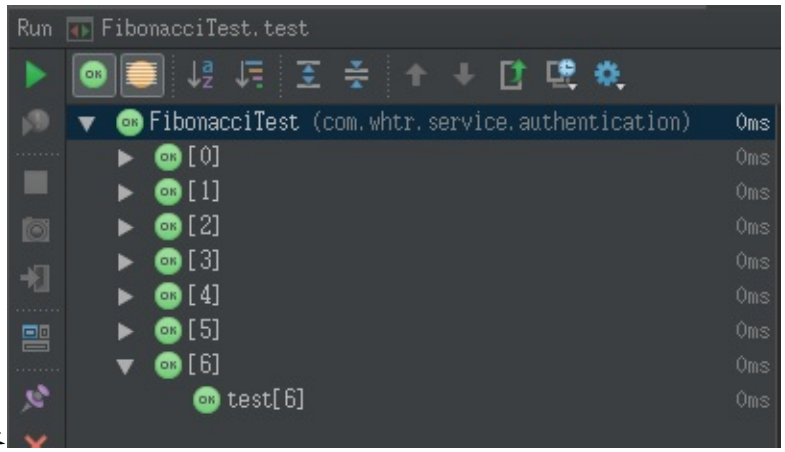
为了容易识别各个test case中的测试参数，你可以在@Parameters注解中提供一个名称,此名称允许包含在运行时替换的占位符

- {index}: 当前参数下标
- {0}, {1}, ...:第1,2...参数值,注意：单引号'应该转义为双引号''

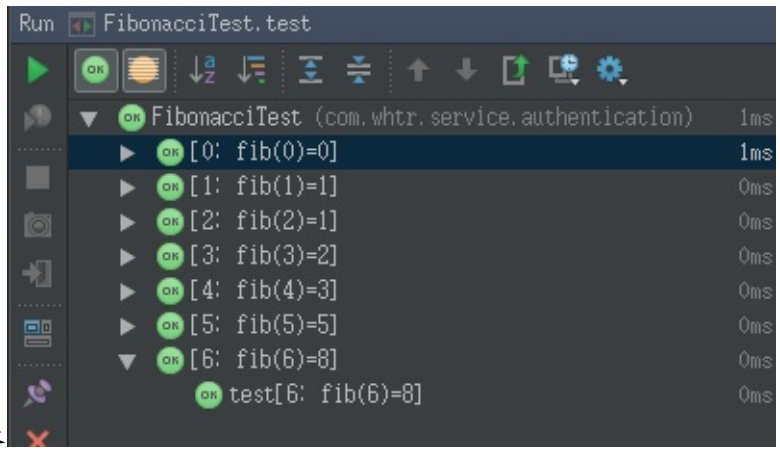
例子

```
package com.junit.learning.parameterized;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

import java.util.Arrays;
import java.util.Collection;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(Parameterized.class)
public class FibonacciTest3 {

    @Parameterized.Parameters(name = "{index}: fib({0})={1}")
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][] {
                { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }
, { 5, 5 }, { 6, 8 }
        });
    }

    @Parameterized.Parameter // first data value (0) is default
    public /* NOT private */ int fInput;

    @Parameterized.Parameter(value = 1)
    public /* NOT private */ int fExpected;
    @Test
    public void test() {
        assertThat(fExpected).isEqualTo(Fibonacci.compute(fInput
));
    }

}
```

加名称之前在idea中运行的效果



加名称之后在idea中的运行效果



在上面给出的例子中，参数化转轮创建像名称[1：FIB（3）= 2]。如果没有指定名称，当前的参数指标将被默认使用。

参考

参数测试,你还可以看看这里

JUnitParams

关于

JUnitParams添加了一个新的JUunit的runner，提供了写起来更容易,可读性更高的参数测试,要求JUnit的版本>=4.6。 git仓库

跟JUnit标准的Parametrised runner的主要区别如下:

> 看这里之前请先阅读一遍JUint的参数测试

- 更明确 - 参数不再是类的属性,而是方法级别的参数
- 更少的代码量 - 你不再需要写一个构造方法来设置参数
- 也可以在Parametrised类中的非Parametrised方法混合使用
- 参数可以通过CSV字符串或者parameters供应类来传递
- parameters供应类提供了足够多的参数让你能够区分不同的测试场景
- 你可以写一个测试方法来提供参数(没有外部类或者静态属性的话)
- 你可以在你的IDE中看到明确的参数(在JUnit的Parametrised中只可以看到参数的序号)

maven依赖

```
<dependency>
  <groupId>pl.pragmatists</groupId>
  <artifactId>JUnitParams</artifactId>
  <version>1.0.5</version>
  <scope>test</scope>
</dependency>
```

Quickstart

请查看下一篇章,快速开始

Quickstart

如果你只是想看JUnitParams的一些简单使用和主要用法,你可以查看这里(建议动手开发之前阅读一遍)

1. 如何使用？比如说有一个Person类:

```
public class Person {
    private int age;

    public Person(int age) {
        this.age = age;
    }

    public boolean isAdult() {
        return age >= 18;
    }

    @Override
    public String toString() {
        return "Person of age: " + age;
    }
}
```

然后你要去测试它,于是你需要创建一个测试类并且定义一个JUnitParamsRunner，比如:

```
@RunWith(JUnitParamsRunner.class)
public class PersonTest {
...
```

现在假设你想有一个简单的测试,检查某特定年龄的人是成年人。您可以用@Parameters注解定义测试参数值：

```
 @Test
 @Parameters({ "17, false", "22, true" })
 public void personIsAdult(int age, boolean valid) throws Ex
ception {
     assertThat(new Person(age).isAdult(), is(valid));
 }
```

上面代码中有几个值是确定的,如果你想要有更多的值,如果你想要更多的值,你可以在类中写一个方法来提供你需要的值:

```
 @Test
 @Parameters(method = "adultValues")
 public void personIsAdult(int age, boolean valid) throws Ex
ception {
     assertEquals(valid, new Person(age).isAdult());
 }
 private Object[] adultValues() {
     return new Object[]{
                 new Object[]{13, false},
                 new Object[]{17, false},
                 new Object[]{18, true},
                 new Object[]{22, true}
             };
 }
```

如果你的测试方法名不长,你可以忽略@Parameters注解中的method参数,只需要将提供值的方法的名称跟你的测试方法名类似（以parametersFor开头+测试方法名）,如下:

```
 @Test
 @Parameters
 public void personIsAdult(int age, boolean valid) throws Ex
ception {
     assertEquals(valid, new Person(age).isAdult());
 }

 private Object[] parametersForPersonIsAdult() {
     return new Object[]{
                 new Object[]{13, false},
                 new Object[]{17, false},
                 new Object[]{18, true},
                 new Object[]{22, true}
         };
 }
```

是不是感觉这个测试并不符合面向对象思想？在这个测试方法中有一个非常愚蠢的构造函数调用,那么让我们传递整个Person对象:

```
 @Test
 @Parameters
 public void isAdult(Person person, boolean valid) throws Ex
ception {
     assertThat(person.isAdult(), is(valid));
 }

 private Object[] parametersForIsAdult() {
     return new Object[]{
                 new Object[]{new Person(13), false},
                 new Object[]{new Person(17), false},
                 new Object[]{new Person(18), true},
                 new Object[]{new Person(22), true}
             };
 }
```

有时候你需要一些外部依赖来提供参数(为啥呢？可能因为它的复杂度，会干扰测试类,也可能是因为你已经拥有它).JUnitParams还提供了从文件或者数据库读取参数的功能,如下:

```
@Test
@Parameters(source = PersonProvider.class)
public void personIsAdult(Person person, boolean valid) {
    assertThat(person.isAdult(), is(valid));
}
```

此时PersonProvider类必须至少有一个静态方法来提供对象数组,通常像下面这样:

```
public class PersonProvider {
    public static Object[] provideAdults() {
        return new Object[]{
                  new Object[]{new Person(25), true},
                  new Object[]{new Person(32), true}
                };
    }

    public static Object[] provideTeens() {
        return new Object[]{
                  new Object[]{new Person(12), false},
                  new Object[]{new Person(17), false}
                };
    }
}
```

Rules规则

Rules允许在测试类的每个测试方法的行为灵活的加成或者重新定义它的行为,测试人员可以重复使用或者扩展以下提供的rule，或者自己拓展一个

例子

下面这个例子中使用了TemporaryFolder和ExpectedException规则:

```
    public class DigitalAssetManagerTest {
      @Rule
      public TemporaryFolder tempFolder = new TemporaryFolder();

      @Rule
      public ExpectedException exception = ExpectedException.non
e();

      @Test
      public void countsAssets() throws IOException {
        File icon = tempFolder.newFile("icon.png");
        File assets = tempFolder.newFolder("assets");
        createAssets(assets, 3);

        DigitalAssetManager dam = new DigitalAssetManager(icon,
assets);
        assertEquals(3, dam.getAssetCount());
      }

      private void createAssets(File assets, int numberOfAssets)
throws IOException {
        for (int index = 0; index < numberOfAssets; index++) {
          File asset = new File(assets, String.format("asset-%d.
mpg", index));
          Assert.assertTrue("Asset couldn't be created.", asset.
createNewFile());
        }
      }

      @Test
      public void throwsIllegalArgumentExceptionIfIconIsNull() {
        exception.expect(IllegalArgumentException.class);
        exception.expectMessage("Icon is null, not a file, or do
esn't exist.");
        new DigitalAssetManager(null, null);
      }
    }
```

JUnit提供的一些基本rule

请看下面的内容

ErrorCollector Rule

ErrorCollector允许发现第一个问题之后继续执行测试代码.(比如：搜索表中所有有问题的行,并报告这些行)：

```
package com.junit.learning;

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ErrorCollector;

public class ErrorCollectorTwiceTest {
    @Rule
    public ErrorCollector collector = new ErrorCollector();

    @Test
    public void example() {
        collector.addError(new Throwable("first thing went wrong"
));
        collector.addError(new Throwable("second thing went wron
g"));
    }
}
```

运行结果如下:

```
java.lang.Throwable: first thing went wrong
java.lang.Throwable: second thing went wrong
```

ExternalResource Rules

ExternalResource为子类提供了两个接口，分别是进入测试之前和退出测试之后，一般它是作为对一些资源在测试前后的控制，如Socket的开启与关闭、Connection的开始与断开、临时文件的创建与删除等。如果ExternalResource用在@ClassRule注解字段中，before()方法会在所有@BeforeClass注解方法之前调用；after()方法会在所有@AfterClass注解方法之后调用，不管在执行@AfterClass注解方法时是否抛出异常。如果ExternalResource用在@Rule注解字段中，before()方法会在所有@Before注解方法之前调用；after()方法会在所有@After注解方法之后调用。

```
    package com.junit.learning;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.rules.ExternalResource;
    import org.junit.rules.TemporaryFolder;

    import static org.assertj.core.api.Assertions.assertThat;

    public class ExternalResourceTest {

        @Rule
        public TemporaryFolder folder = new TemporaryFolder();

        @Rule
        public ExternalResource resource = new ExternalResource(
) {
            @Override
            protected void before() throws Throwable {
                folder.create();
            }

            @Override
            protected void after() {
                folder.delete();
            }
        };

        @Test
        public void testFoo() {
            assertThat(folder.getRoot().isDirectory()).isTrue();
        }
    }
```

TemporaryFolder Rule

TemporaryFolder规则允许在测试方法创建文件或者文件夹，并且在方法完成时删除创建好的文件或文件夹.当资源无法被删除时默认情况下不会抛出异常:

- TemporaryFolder#newFolder(String... folderNames) 递归创建临时目录
- TemporaryFolder#newFile() 创建一个随机名称的新文件,#newFolder()创建一个随机名称的新文件夹
- 从4.13版本开始TemporaryFolder允许严格审核删除资源文件，AssertionError方法可以断言资源文件不能被删除.该功能只能通过使用#builder（）方法来选择。默认情况下严格审核被禁用，它保持向后兼容性

```
  @Rule
  public TemporaryFolder folder = TemporaryFolder.builder().
assureDeletion().build();
```

TestName Rule

TestName是用来测试方法名的

```
public class NameRuleTest {
  @Rule
  public TestName name = new TestName();

  @Test
  public void testA() {
    assertEquals("testA", name.getMethodName());
  }

  @Test
  public void testB() {
    assertEquals("testB", name.getMethodName());
  }
}
```

TestWatchman/TestWatcher Rules

- 从4.9版本开始TestWatcher替代TestWatchman,它实现的是TestRule接口而不是MethodRule --http://junit.org/javadoc/latest/org/junit/rules/TestWatcher.html
- JUnit从4.7开始采用TestWatchman,它使用的是MethodRule接口，现在已经不建议使用
- TestWatcher(还有已经不建议使用的TestWatchman)是为记下测试行动规则的基类，并不需修改它。例如，这个类将保留每个通过和未通过测试的日志

> TestWatcher为子类提供了四个事件方法以监控测试方法在运行过程中的状态，一般它可以作为信息记录使用。如果TestWatcher作为@ClassRule注解字段，则该测试类在运行之前（调用所有的@BeforeClass注解方法之前）会调用starting()方法；当所有@AfterClass注解方法调用结束后，succeeded()方法会被调用；若@AfterClass注解方法中出现异常，则failed()方法会被调用；最后，finished()方法会被调用；所有这些方法的Description是Runner对应的Description。如果TestWatcher作为@Rule注解字段，则在每个测试方法运行前（所有的@Before注解方法运行前）会调用starting()方法；当所有@After注解方法调用结束后，succeeded()方法会被调用；若@After注解方法中跑出异常，则failed()方法会被调用；最后，finished()方法会被调用；所有Description的实例是测试方法的Description实例。

```
package com.junit.learning;
import org.junit.After;
import org.junit.AssumptionViolatedException;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TestRule;
import org.junit.rules.TestWatcher;
import org.junit.runner.Description;
import org.junit.runners.model.Statement;

import static org.assertj.core.api.Assertions.assertThat;

public class WatchmanMethodTest {

    private static String watchedLog = "";

    @Rule
```

```
        public TestRule watchman = new TestWatcher() {
            @Override
            public Statement apply(Statement base, Description d
escription) {
                return super.apply(base, description);
            }

            @Override
            protected void succeeded(Description description) {
                watchedLog += description.getDisplayName() + " "
 + "success!\n";
                System.out.println(watchedLog);
            }

            @Override
            protected void failed(Throwable e, Description descr
iption) {
                watchedLog += description.getDisplayName() + " "
 + e.getClass().getSimpleName() + "\n";
                System.out.println(watchedLog);
            }

            @Override
            protected void skipped(AssumptionViolatedException e
, Description description) {
                watchedLog += description.getDisplayName() + " "
 + e.getClass().getSimpleName() + "\n";
                System.out.println(watchedLog);
            }

            @Override
            protected void starting(Description description) {
                super.starting(description);
                System.out.println("starting");
            }

            @Override
            protected void finished(Description description) {
                super.finished(description);
                System.out.println("finished");
            }
        };
```

```
    @Before
    public void before() {
        System.out.println("before");
    }

    @After
    public void after() {
        System.out.println("after");
    }

    @Test
    public void fails() {
        System.out.println("fails");
        assertThat(1).isEqualTo(2);
    }

    @Test
    public void succeeds() {
        System.out.println("success");
    }
}
```

执行结果:

```
starting
before
success
after
succeeds(com.junit.learning.WatchmanTest) success!
finished

starting
before
fails
after
succeeds(com.junit.learning.WatchmanTest) success!
fails(com.junit.learning.WatchmanTest) AssertionError
finished
```

```
    package com.junit.learning;

    import org.assertj.core.api.exception.RuntimeIOException;
    import org.junit.AfterClass;
    import org.junit.AssumptionViolatedException;
    import org.junit.BeforeClass;
    import org.junit.ClassRule;
    import org.junit.Test;
    import org.junit.rules.TestRule;
    import org.junit.rules.TestWatcher;
    import org.junit.runner.Description;
    import org.junit.runners.model.Statement;

    import static org.assertj.core.api.Assertions.assertThat;

    public class WatchmanClassTest {

        private static String watchedLog ="";

        @ClassRule
        public static TestRule watchman = new TestWatcher() {
            @Override
            public Statement apply(Statement base, Description d
escription) {
                return super.apply(base, description);
            }

            @Override
            protected void succeeded(Description description) {
                watchedLog += description.getDisplayName() + " "
 + "success!\n";
                System.out.println(watchedLog);
            }

            @Override
            protected void failed(Throwable e, Description descr
iption) {
                watchedLog += description.getDisplayName() + " "
 + e.getClass().getSimpleName() + "\n";
                System.out.println(watchedLog);
            }
```

```
            @Override
            protected void skipped(AssumptionViolatedException e
, Description description) {
                watchedLog += description.getDisplayName() + " "
 + e.getClass().getSimpleName() + "\n";
                System.out.println(watchedLog);
            }

            @Override
            protected void starting(Description description) {
                super.starting(description);
                System.out.println("starting");
            }

            @Override
            protected void finished(Description description) {
                super.finished(description);
                System.out.println("finished");
            }
        };

        @BeforeClass
        public static void before(){
            System.out.println("before");
        }

        @AfterClass
        public static void after(){
            System.out.println("after");
            throw new RuntimeIOException("");
        }

        @Test
        public void fails() {
            System.out.println("fails");
            assertThat(1).isEqualTo(2);
        }

        @Test
        public void succeeds() {
            System.out.println("success");
        }
```

```
    }
```

运行结果如下:

```
    success
    fails
    starting
    before
    after
    com.junit.learning.WatchmanClassTest RuntimeIOException
    finished
```

Verifier Rule

Verifier是在每个测试方法已经结束的时候，再加入一些额外的逻辑，只有额外的逻辑也通过，才表示测试成功，否则，测试依旧失败，即使在之前的运行中都是成功的

```
    package com.junit.learning;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.rules.Verifier;

    import static org.assertj.core.api.Assertions.assertThat;

    public class VerifierTest {
        private static String sequence = "";
        @Rule
        public Verifier collector = new Verifier() {
            @Override
            protected void verify() {
                assertThat(sequence).isEqualTo("test verify ");
            }
        };

        @Test
        public void example() {
            sequence += "test ";
            assertThat(sequence).isEqualTo("test ");//即便这里校验正确,由于Verifier的验证错误,也会导致他的验证失败
        }

        //成功
        @Test
        public void verifierRunsAfterTest() {
            sequence = "test verify ";
        }
    }
```

Gitbook该文件修订时间： 2017-03-13 05:37:11

Timeout Rule

该规则适用于测试类中的所有测试方法

```
    package com.junit.learning.rules;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.rules.TestRule;
    import org.junit.rules.Timeout;

    import java.util.concurrent.TimeUnit;

    public class TimeoutTest {
        public static String log;

        @Rule
        public TestRule globalTimeout = new Timeout(2, TimeUnit.
SECONDS);

        @Test
        public void testInfiniteLoop1() {
            log += "ran1";
            for(;;) {}
        }

        @Test
        public void testInfiniteLoop2() {
            log += "ran2";
            for(;;) {}
        }
    }
```

ExpectedException Rules

允许测试预期的异常类型和消息

```
    package com.junit.learning.rules;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.rules.ExpectedException;

    import static org.mockito.Matchers.startsWith;

    public class ExpectedExceptionTest {
        @Rule
        public ExpectedException thrown = ExpectedException.none
();

        @Test
        public void throwsNothing() {

        }

        //失败
        @Test
        public void throwsNullPointerException() {
            thrown.expect(NullPointerException.class);
        }

        //成功
        @Test
        public void throwsNullPointerExceptionWithMessage() {
            thrown.expect(NullPointerException.class);
            thrown.expectMessage("happened?");
            thrown.expectMessage(startsWith("What"));
            throw new NullPointerException("What happened?");
        }
    }
```

ClassRule

ClassRule扩展了方法级别的规则,添加了可以影响类的运行的的静态属性。任何ParentRunner的子类,包括标准BlockJUnit4ClassRunner和Suite类,都支持ClassRule。

```
    package com.junit.learning.rules;

    import org.junit.ClassRule;
    import org.junit.Rule;
    import org.junit.rules.ExternalResource;
    import org.junit.rules.TemporaryFolder;
    import org.junit.runner.RunWith;
    import org.junit.runners.Suite;

    @RunWith(Suite.class)
    @Suite.SuiteClasses({A.class, B.class})
    public class ClassRuleTest {
        @Rule
        public static TemporaryFolder folder = new TemporaryFold
er();

        @ClassRule
        public static ExternalResource resource = new ExternalRe
source() {
            @Override
            protected void before() throws Throwable {
                folder.create();
            }

            @Override
            protected void after() {
                folder.delete();
            }
        };
    }
```

以上例子会在A，B开始之前创建一个文件,并在其结束之后删除文件

Gitbook该文件修订时间： 2017-03-13 05:37:11

RuleChain

RuleChain支持TestRule的排序

```
    package com.junit.learning.rules;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.rules.RuleChain;
    import org.junit.rules.TestRule;
    import org.junit.runner.Description;
    import org.junit.runners.model.Statement;

    import static org.assertj.core.api.Assertions.assertThat;

    public class RuleChainTest {
        @Rule
        public TestRule chain = RuleChain
                .outerRule(new LoggingRule("outer rule"))
                .around(new LoggingRule("middle rule"))
                .around(new LoggingRule("inner rule"));

        @Test
        public void example() {
            assertThat(true).isTrue();
        }

        public class LoggingRule implements TestRule{
            private String str = "";
            public LoggingRule(String str){
                this.str = str;
            }

            public Statement apply(Statement base, Description d
escription) {
                System.out.println(str+base.toString()+descripti
on);
                return base;
            }
        }
    }
```

自定义规则

大多数自定义规则都是ExternalResource规则的延伸,一般情况下你需要实现TestRule接口:

实现TestRule需要自定义一个构造方法,添加的方法用于测试,并且提供一个新的Statement。比如说有一下需求:为每个测试记录日志

```
import java.util.logging.Logger;

import org.junit.rules.TestRule;
import org.junit.runner.Description;
import org.junit.runners.model.Statement;

public class TestLogger implements TestRule {
  private Logger logger;

  public Logger getLogger() {
    return this.logger;
  }

  @Override
  public Statement apply(final Statement base, final Description
 description) {
    return new Statement() {
      @Override
      public void evaluate() throws Throwable {
        logger = Logger.getLogger(description.getTestClass().get
Name() + '.' + description.getDisplayName());
        base.evaluate();
      }
    };
  }
}
```

```
import java.util.logging.Logger;

import org.example.junit.TestLogger;
import org.junit.Rule;
import org.junit.Test;

public class MyLoggerTest {

  @Rule
  public TestLogger logger = new TestLogger();

  @Test
  public void checkOutMyLogger() {
    final Logger log = logger.getLogger();
    log.warn("Your test is showing!");
  }

}
```

Theories

> 翻译理论 你读过数学理论吗？它看起来通常像这样：对于所有的a，b>0，都可以得出：a+b>a，a+b>b。只是我们看到的定义通常难以理解。譬如可以这样描述：它囊括了一个相当大的范围内(在此是无穷大)的所有元素(或者是元素的组合) 目前看来它是Parameterized改良版

它分为两部分：一个是提供数据点集(比如待测试的数据)的方法，另一个是理论本身。这个理论看起来几乎就像一个测试，但是它有一个不同的注解(@Theory)，并且它需要参数。类通过使用数据点集的任意一种可能的组合来执行所有理论。

```
    package com.junit.learning.theories;

    import org.junit.experimental.theories.DataPoint;
    import org.junit.experimental.theories.Theories;
    import org.junit.experimental.theories.Theory;
    import org.junit.runner.RunWith;

    import static org.hamcrest.core.IsNot.not;
    import static org.hamcrest.core.StringContains.containsStrin
g;
    import static org.junit.Assume.assumeThat;

    @RunWith(Theories.class)
    public class TheoriesTest {
           @DataPoints
            public static int[] positiveIntegers() {
                return new int[]{1, 10, 1234567};
            }

            @Theory
            public void a_plus_b_is_greater_than_a_and_greater_t
han_b(Integer a, Integer b) {
                assertThat(a + b > a).isTrue();
                assertThat(a + b > b).isTrue();
            }
    }
```

行为

Theories支持在参数中内嵌一套整数:

```
    @Theory
    public void multiplyIsInverseOfDivideWithInlineDataPoints(@T
estedOn(ints = {0, 5, 10}) int amount,
            @TestedOn(ints = {0, 1, 2}) int m) {
        assumeThat(m, not(0));
        System.out.println("amount:"+amount+",m:"+m);
    }
```

允许结果如下:

```
amount:0,m:1
amount:0,m:2
amount:5,m:1
amount:5,m:2
amount:10,m:1
amount:10,m:2
```

由此可以看出这种方式是以参数的并集来运行的

你也可以自定义来拓展参数的提供方案.

```
package com.junit.learning.theories;

import org.junit.experimental.theories.ParametersSuppliedBy;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.PARAMETER;

@ParametersSuppliedBy(BetweenSupplier.class)
@Retention(RetentionPolicy.RUNTIME)
@Target(PARAMETER)
public @interface Between {
    int first();

    int last();
}
```

```
    package com.junit.learning.theories;

    import org.junit.experimental.theories.ParameterSignature;
    import org.junit.experimental.theories.ParameterSupplier;
    import org.junit.experimental.theories.PotentialAssignment;

    import java.util.ArrayList;
    import java.util.List;

    public class BetweenSupplier extends ParameterSupplier {

        @Override
        public List<PotentialAssignment> getValueSources(Paramet
erSignature sig) {
            Between annotation = sig.getAnnotation(Between.class
);

            ArrayList list = new ArrayList();
            for (int i = annotation.first(); i <= annotation.las
t(); i++) {
                list.add(PotentialAssignment.forValue("ints", i)
);
            }
            return list;
        }
    }
```

```
    @Theory
    public void multiplyIsInverseOfDivideWithInlineDataPoints2(@
Between(first = -100, last = 100) int amount,
            @Between(first = -100, last = 100) int m) {
        assumeThat(m, not(0));
        System.out.println("amount:"+amount+",m:"+m);
    }
```

Test fixtures

Test fixtures是对象的一种稳定状态.测试运行的基准.一个test fixture是为了在测试运行之前提供稳定的,公共的可重复的运行环境.例如:

- 准备输入数据和创建Mock对象
- 加载数据库或者已知的数据
- 复制特定的已经文件,创建一个test fixture用来初始化某些状态

JUnit提供可以在每个测试运行前后都运行fixture,或者在所有测试方法前后只运行一次fixture的注解 JUnit有两个类级别(@BeforeClass和@AfterClass),两个方法级别(@Afer和@Before),总共四个fixture注解. fixture更深层次的设计理念,如何使用rule执行他们可以查看这里

举例:

```
    package com.junit.learning.fixtures;

    import org.junit.After;
    import org.junit.AfterClass;
    import org.junit.Before;
    import org.junit.BeforeClass;
    import org.junit.Test;

    import java.io.Closeable;
    import java.io.IOException;

    public class TestFixturesExample {
        static class ExpensiveManagedResource implements Closeab
le {
            @Override
            public void close() throws IOException {}
        }

        static class ManagedResource implements Closeable {
            @Override
            public void close() throws IOException {}
        }

        @BeforeClass
        public static void setUpClass() {
```

```
            System.out.println("@BeforeClass setUpClass");
            myExpensiveManagedResource = new ExpensiveManagedRes
ource();
        }

        @AfterClass
        public static void tearDownClass() throws IOException {
            System.out.println("@AfterClass tearDownClass");
            myExpensiveManagedResource.close();
            myExpensiveManagedResource = null;
        }

        private ManagedResource myManagedResource;
        private static ExpensiveManagedResource myExpensiveManag
edResource;

        private void println(String string) {
            System.out.println(string);
        }

        @Before
        public void setUp() {
            this.println("@Before setUp");
            this.myManagedResource = new ManagedResource();
        }

        @After
        public void tearDown() throws IOException {
            this.println("@After tearDown");
            this.myManagedResource.close();
            this.myManagedResource = null;
        }

        @Test
        public void test1() {
            this.println("@Test test1()");
        }

        @Test
        public void test2() {
            this.println("@Test test2()");
        }
```

```
    }
```

运行结果如下:

```
    @Before setUp
    @Test test1()
    @After tearDown
    @Before setUp
    @Test test2()
    @After tearDown
    @BeforeClass setUpClass
    @AfterClass tearDownClass
```

Categories

> 直译：分类,分类执行的意思

Categories runner只运行那些有@Category注解或者@IncludeCategory注解的类,或者它的子类的方法。任何类或者接口都可以被作为categories.@IncludeCategory(SuperClass.class)和@Category({SubClass.class})标记的类将会被运行.

你可以可以用@ExcludeCategory注解排除一些类.

例子:

```
public class A{
    @Test
    public void a() {
        System.out.println("A.a");
        fail();
    }

    @Category(SlowTests.class)
    @Test
    public void b() {
        System.out.println("A.b");
    }
}
```

```
@Category({SlowTests.class, FastTests.class})
public class B {
    @Test
    public void c() {
        System.out.println("B.c");
    }
}
```

```
public interface FastTests {
}
public interface SlowTests {

}
```

```
@RunWith(Categories.class)
@Categories.IncludeCategory(SlowTests.class)
@Suite.SuiteClasses( { A.class, B.class })
public class SlowTestSuite {
}
```

运行结果如下:

```
A.b
B.c
```

```
@RunWith(Categories.class)
@Categories.IncludeCategory(SlowTests.class)
@Categories.ExcludeCategory(FastTests.class)
@Suite.SuiteClasses( { A.class, B.class })
public class SlowTestSuite2 {
}
```

运行结果如下:

```
A.b
```

在maven中使用categories

你可以使用maven-surefire-plugin或者maven-failsafe-plugin来配置categories的include或者exclude集合.如果不使用任何选项，所有测试将被默认执行

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <groups>com.junit.learning.category.FastTests</groups>
      </configuration>
    </plugin>
  </plugins>
</build>
```

如果要排除特定的列表,则使用标签

categories的典型应用

categories用于在测试中添加元数据。 categories的用途一般如下:

- 这些类型的测试都可能用到:单元测试,集成测试,冒烟测试,回归测试,性能测试
- 怎样快速的执行测试: SlowTests, QuickTests
- 自动化测试:每日部署测试
- 测试状态: 不稳定测试,进行中测试

> 根据上面文字,可以理解为categories的主要用途是用给不同测试范围准备不同的元数据的

多线程并发

JUnit推荐了一个多线程的断言方法(需要根据自己的需求复制或修改):

```
public static void assertConcurrent(final String message, final
List<? extends Runnable> runnables, final int maxTimeoutSeconds)
throws InterruptedException {
    final int numThreads = runnables.size();
    final List<Throwable> exceptions = Collections.synchronizedL
ist(new ArrayList<Throwable>());
    final ExecutorService threadPool = Executors.newFixedThreadP
ool(numThreads);
    try {
        final CountDownLatch allExecutorThreadsReady = new Count
DownLatch(numThreads);
        final CountDownLatch afterInitBlocker = new CountDownLat
ch(1);
        final CountDownLatch allDone = new CountDownLatch(numThr
eads);
        for (final Runnable submittedTestRunnable : runnables) {
            threadPool.submit(new Runnable() {
                public void run() {
                    allExecutorThreadsReady.countDown();
                    try {
                        afterInitBlocker.await();
            submittedTestRunnable.run();
                    } catch (final Throwable e) {
                        exceptions.add(e);
                    } finally {
                        allDone.countDown();
                    }
                }
            });
        }
        // wait until all threads are ready
        assertTrue("Timeout initializing threads! Perform long l
asting initializations before passing runnables to assertConcurr
ent", allExecutorThreadsReady.await(runnables.size() * 10, TimeU
nit.MILLISECONDS));
        // start all test runners
```

```
        afterInitBlocker.countDown();
        assertTrue(message +" timeout! More than" + maxTimeoutSe
conds + "seconds", allDone.await(maxTimeoutSeconds, TimeUnit.SEC
ONDS));
    } finally {
        threadPool.shutdownNow();
    }
    assertTrue(message + "failed with exception(s)" + exceptions
, exceptions.isEmpty());
}
```

Enclosed

先看例子:

```
    package com.junit.learning.enclosed;

    import org.junit.Test;
    import org.junit.experimental.runners.Enclosed;
    import org.junit.runner.RunWith;

    import static org.assertj.core.api.Assertions.assertThat;

    public class OutsideClassTest {

        public static class insideClassTest{
            @Test
            public void testInsideClass(){
                System.out.println("inside");
                OutsideClass.insideClass insideClass = new Outsi
deClass.insideClass("str");
                assertThat(insideClass.getStr()).isEqualTo("str"
);
            }
        }

        @Test
        public void testInsideClass(){
            System.out.println("outside");
            OutsideClass.insideClass insideClass = new OutsideCl
ass.insideClass("str");
            assertThat(insideClass.getStr()).isEqualTo("str");
        }
    }
```

如果在外部类中执行整个类的话,那么运行结果是:outside。但是有时候我们只希望运行内部类,那Enclode runner就派上用场了:

```
    package com.junit.learning.enclosed;

    import org.junit.Test;
    import org.junit.experimental.runners.Enclosed;
    import org.junit.runner.RunWith;

    import static org.assertj.core.api.Assertions.assertThat;

    @RunWith(Enclosed.class)
    public class OutsideClassTest {

        public static class insideClassTest{
            @Test
            public void testInsideClass(){
                System.out.println("inside");
                OutsideClass.insideClass insideClass = new Outsi
deClass.insideClass("str");
                assertThat(insideClass.getStr()).isEqualTo("str"
);
            }
        }

        @Test
        public void testInsideClass(){
            System.out.println("outside");
            OutsideClass.insideClass insideClass = new OutsideCl
ass.insideClass("str");
            assertThat(insideClass.getStr()).isEqualTo("str");
        }
    }
```

运行结果是:inside。 也就是说在外部内加了@RunWith(Enclosed.class)之后,只会执行内部内中的方法

System Rules

> 是JUnit用来测试使用java.lang.System的rule的集合

系统规则的范围分为五个部分：

1. System.out, System.err and System.in：从命令行中读取和写入命令的应用可以使用SystemErrRule，SystemOutRule和TextFromStandardInputStream提供的输入文本和验证输出；应用有时候会用到System.out或System.err.可以使用DisallowWriteToSystemOut 和DisallowWriteToSystemErr确保System.out和System.err没有调用
2. System.exit：使用ExpectedSystemExit规则来测试代码调用的System.exit(…). 校验System.exit(…).已经被调用,也可以校验代码的调用状态和断言应用终止
3. System Properties:如果你需要测试系统配置,你需要测试,并且在测试之后恢复它们,可以使用 ClearSystemProperties,ProvideSystemProperty 和RestoreSystemProperties来实现
4. Environment Variables:如果你的测试需要用的环境变量,你可以使用EnvironmentVariables
5. Security Managers:可以使用ProvideSecurityManager为测试提供一个SecurityManager，测试结束之后恢复.

maven依赖

> 需要JUnit 4.9版本

```
<dependency>
  <groupId>com.github.stefanbirkner</groupId>
  <artifactId>system-rules</artifactId>
  <version>1.16.0</version>
  <scope>test</scope>
</dependency>
```

Provide Properties

ProvideSystemProperty为测试提供一个系统属性,并且在测试结束之后恢复

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.ProvideSystemPrope
rty;

    import static org.assertj.core.api.Assertions.assertThat;

    public class ProvidePropertiesTest {
        @Rule
        public final ProvideSystemProperty myPropertyHasMyValue
                = new ProvideSystemProperty("MyProperty", "MyVal
ue");

        @Rule
        public final ProvideSystemProperty otherPropertyIsMissin
g
                = new ProvideSystemProperty("OtherProperty", null
);

        @Test
        public void overrideProperty() {
            assertThat(System.getProperty("MyProperty")).isEqual
To("MyValue");
            assertThat(System.getProperty("OtherProperty")).isNu
ll();
        }
    }
```

也可以将上述例子简化成单个实例

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.ProvideSystemPrope
rty;

    import static org.assertj.core.api.Assertions.assertThat;

    public class ProvidePropertiesTest2 {
        @Rule
        public final ProvideSystemProperty properties
                = new ProvideSystemProperty("MyProperty", "MyVal
ue").and("OtherProperty", null);

        @Test
        public void overrideProperty() {
            assertThat(System.getProperty("MyProperty")).isEqual
To("MyValue");
            assertThat(System.getProperty("OtherProperty")).isNu
ll();
        }
    }
```

可以用ProvideSystemProperty读取配置文件,主要有以下两种用法

```
@Rule
public final ProvideSystemProperty properties
 = ProvideSystemProperty.fromFile("/home/myself/example.properti
es");
```

> 从系统路径中读取文件

```
@Rule
public final ProvideSystemProperty properties
 = ProvideSystemProperty.fromResource("example.properties");
```

> 从resources目录读取文件

Gitbook该文件修订时间： 2017-03-13 05:37:11

Clear Properties

ClearSystemProperties在测试之前删除资源文件,测试结束之后恢复

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.ClearSystemPropert
ies;

    import static org.assertj.core.api.Assertions.assertThat;

    public class ClearProperties {
        @Rule
        public final ClearSystemProperties myPropertyIsCleared =
new ClearSystemProperties("MyProperty");

        @Test
        public void overrideProperty() {
            assertThat(System.getProperty("MyProperty")).isNull(
);
        }
    }
```

Restore Properties

RestoreSystemProperties规则:当测试完成后（无论是通过或失败）撤销所有系统属性的变化。

```
package com.junit.learning.systemtest;

import org.junit.AfterClass;
import org.junit.Rule;
import org.junit.Test;
import org.junit.contrib.java.lang.system.RestoreSystemPropertie
s;

public class RestorePropertiesTest {
    @Rule
    public final RestoreSystemProperties restoreSystemProperties
            = new RestoreSystemProperties();

    @AfterClass
    public static void tearDown(){
        System.out.println(System.getProperty("MyProperty"));
    }

    @Test
    public void overrideProperty() {
        //after the test the original value of "MyProperty" will
 be restored.
        System.setProperty("MyProperty", "other value");
    }
}
```

System.err and System.out

SystemErrRule和SystemOutRule可以记录写入类中的System.err或System.out.记录通过getLog()调用

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.SystemErrRule;

    import static org.assertj.core.api.Assertions.assertThat;

    public class SystemErrRuleTest {
        @Rule
        public final SystemErrRule systemErrRule = new SystemErr
Rule().enableLog();

        @Test
        public void writesTextToSystemErr() {
            System.err.print("hello world");
            assertThat(systemErrRule.getLog()).isEqualTo("hello
world");
        }
    }
```

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.SystemOutRule;

    import static org.assertj.core.api.Assertions.assertThat;

    public class SystemOutRuleTest {
        @Rule
        public final SystemOutRule systemOutRule = new SystemOut
Rule().enableLog();

        @Test
        public void writesTextToSystemOut() {
            System.out.print("hello world");
            assertThat(systemOutRule.getLog()).isEqualTo("hello
world");
        }
    }
```

如果你的验证代码中有分隔符或者反斜杠等跟操作系统相关(Linux: \n, Windows: \r\n)的验证.可以使用getLogWithNormalizedLineSeparator

```
    @Test
    public void writesTextToSystemOut2() {
        System.out.print(String.format("hello world%n"));
        assertThat(systemOutRule.getLogWithNormalizedLineSeparat
or()).isEqualTo("hello world\n");
    }
```

如果要清除已经写入到日志中的一些文本日志，可以这样：

```
    @Test
    public void writesTextToSystemErr2() {
        System.err.print("hello world");
        systemErrRule.clearLog();
        System.err.print("foo");
        assertThat(systemErrRule.getLog()).isEqualTo("foo");
    }
```

上面的例子字符串海是输出到System.err或者System.out中了,可以调用mute()来禁止输出

```
    @Rule
    public final SystemErrRule systemErrRule = new SystemErrRule
().enableLog().mute();

    @Rule
    public final SystemOutRule systemOutRule = new SystemOutRule
().enableLog().mute();
```

但是在测试失败的时候它是有用的，muteForSuccessfulTests()允许在测试失败的时候输出。

```
    @Rule
    public final SystemErrRule systemErrRule = new SystemErrRule
().muteForSuccessfulTests();

    @Rule
    public final SystemOutRule systemOutRule = new SystemOutRule
().muteForSuccessfulTests();
```

不允许写入System.out和System.err

当你写入数据到System.err或者System.out时DisallowWriteToSystemErr和DisallowWriteToSystemOut会使测试失败.

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.DisallowWriteToSys
temOut;

    public class DisallowWriteToSystemOutTest {
        @Rule
        public final DisallowWriteToSystemOut disallowWriteToSys
temOut
                = new DisallowWriteToSystemOut();

        @Test
        public void this_test_fails() {
            System.out.println("some text");
        }
    }
```

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.DisallowWriteToSys
temErr;

    public class DisallowWriteToSystemErrTest {
        @Rule
        public final DisallowWriteToSystemErr disallowWriteToSys
temErr
                = new DisallowWriteToSystemErr();

        @Test
        public void this_test_fails() {
            System.err.println("some text");
        }
    }
```

System.in

TextFromStandardInputStream可以用来测试你使用的System.in。
provideLines(String...)可以获取出System.in的两个数字,并计算这些数字的总和

```
package com.junit.learning.systemtest;

import java.util.Scanner;

public class Summarize {
    public static int sumOfNumbersFromSystemIn() {
        Scanner scanner = new Scanner(System.in);
        int firstSummand = scanner.nextInt();
        int secondSummand = scanner.nextInt();
        return firstSummand + secondSummand;
    }
}
```

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.TextFromStandardIn
putStream;

    import static org.assertj.core.api.Assertions.assertThat;
    import static org.junit.contrib.java.lang.system.TextFromSta
ndardInputStream.emptyStandardInputStream;

    public class SummarizeTest {
        @Rule
        public final TextFromStandardInputStream systemInMock =
emptyStandardInputStream();

        @Test
        public void summarizesTwoNumbers() {
            systemInMock.provideLines("1", "2");
            assertThat(Summarize.sumOfNumbersFromSystemIn()).isE
qualTo(3);
        }
    }
```

可以用TextFromStandardInputStream来模拟异常,方便测试代码是否正确处理异常

```
    systemInMock.throwExceptionOnInputEnd(new IOException());
```

System.exit()

如果你的代码调用了System.exit()，ExpectedSystemExit可以帮你测试它的调用。但是没有办法使用JUnit的断言了,你可以提供一个断言对象到ExpectedSystemExit(这里只能使用JUnit的Assertion,不能使用assertj)

```
    package com.whtr.service.authentication.resources.impl;

    import com.whtr.dolphin.contract.BusinessStatusRuntimeExcept
ion;
    import com.whtr.dolphin.contract.SystemStatusRuntimeExceptio
n;
    import com.whtr.service.authentication.resources.UserResourc
e;
    import com.whtr.service.authentication.resources.pojo.UserPo
jo;
    import com.whtr.soa.contracts.registration.registrationservi
ce.CustomerModel;
    import com.whtr.soa.contracts.registration.registrationservi
ce.IRegistrationService;
    import com.whtr.soa.contracts.registration.registrationservi
ce.ValidateCustomerReq;
    import com.whtr.soa.contracts.registration.registrationservi
ce.ValidateCustomerResp;

    import org.junit.After;
    import org.junit.Before;
    import org.junit.Test;
    import org.mockito.InjectMocks;
    import org.mockito.Mock;
    import org.mockito.MockitoAnnotations;


    import java.rmi.RemoteException;

    import static org.assertj.core.api.Assertions.assertThat;
    import static org.mockito.Mockito.reset;
    import static org.mockito.Mockito.when;

    public class UserResourceImplTest {
```

```
        @InjectMocks
        private UserResource userResource;

        @Mock
        private IRegistrationService iRegistrationService;

        @Before
        public void setUp() throws Exception {
            userResource = new UserResourceImpl();
            MockitoAnnotations.initMocks(this);
        }

        @After
        public void tearDown() throws Exception {
            reset(iRegistrationService);
        }

        @Test
        public void testValidateCustomer() throws Exception {
            String phone = "123456";
            String password = "123456";

            ValidateCustomerReq req = new ValidateCustomerReq();
            req.setPhone(phone);
            req.setPassword(password);

            ValidateCustomerResp validateCustomerResp = new Vali
dateCustomerResp();
            validateCustomerResp.setSuccess(true);

            CustomerModel customerModel = new CustomerModel();
            customerModel.setId(123);

            validateCustomerResp.setCustomer(customerModel);
            when(iRegistrationService.validateCustomer(req)).the
nReturn(validateCustomerResp);
            UserPojo validate = userResource.validateCustomer(ph
one, password);
            assertThat(validate).isNotNull();
        }

        @Test(expected = BusinessStatusRuntimeException.class)
```

```
        public void testValidateCustomer_false() throws Exceptio
n {
            String phone = "123456";
            String password = "123456";

            ValidateCustomerReq req = new ValidateCustomerReq();
            req.setPhone(phone);
            req.setPassword(password);

            ValidateCustomerResp validateCustomerResp = new Vali
dateCustomerResp();
            validateCustomerResp.setSuccess(false);
            validateCustomerResp.setErrorCode(1);

            CustomerModel customerModel = new CustomerModel();
            customerModel.setId(123);

            validateCustomerResp.setCustomer(customerModel);
            when(iRegistrationService.validateCustomer(req)).the
nReturn(validateCustomerResp);
            userResource.validateCustomer(phone, password);
        }


        @Test(expected = BusinessStatusRuntimeException.class)
        public void testValidateCustomer_false2() throws Excepti
on {
            String phone = "123456";
            String password = "123456";

            ValidateCustomerReq req = new ValidateCustomerReq();
            req.setPhone(phone);
            req.setPassword(password);

            ValidateCustomerResp validateCustomerResp = new Vali
dateCustomerResp();
            validateCustomerResp.setSuccess(false);
            validateCustomerResp.setErrorCode(2);

            CustomerModel customerModel = new CustomerModel();
            customerModel.setId(123);
```

```
            validateCustomerResp.setCustomer(customerModel);
            when(iRegistrationService.validateCustomer(req)).the
nReturn(validateCustomerResp);
            userResource.validateCustomer(phone, password);
        }


        @Test(expected = SystemStatusRuntimeException.class)
        public void testValidateCustomer_exception() throws Exce
ption {
            String phone = "123456";
            String password = "123456";

            ValidateCustomerReq req = new ValidateCustomerReq();
            req.setPhone(phone);
            req.setPassword(password);

            ValidateCustomerResp validateCustomerResp = new Vali
dateCustomerResp();
            validateCustomerResp.setSuccess(false);

            CustomerModel customerModel = new CustomerModel();
            customerModel.setId(123);

            validateCustomerResp.setCustomer(customerModel);
            when(iRegistrationService.validateCustomer(req)).the
nThrow(new RemoteException());
            userResource.validateCustomer(phone, password);
        }
    }
```

Environment Variables

EnvironmentVariables可以在测试之前设置环境变量,并在测试之后恢复

```
    package com.junit.learning.systemtest;

    import org.junit.Rule;
    import org.junit.Test;
    import org.junit.contrib.java.lang.system.EnvironmentVariabl
es;

    import static org.assertj.core.api.Assertions.assertThat;

    public class EnvironmentVariablesTest {
        @Rule
        public final EnvironmentVariables environmentVariables
                = new EnvironmentVariables();

        @Test
        public void setEnvironmentVariable() {
            environmentVariables.set("name", "value");
            assertThat(System.getenv("name")).isEqualTo("value")
;
        }
    }
```

Security Manager

ProvideSecurityManager可以在你的测试中替换系统的security manager。

```
public void MyTest {
    private final MySecurityManager securityManager
     = new MySecurityManager();

    @Rule
    public final ProvideSecurityManager provideSecurityManager
     = new ProvideSecurityManager(securityManager);

    @Test
    public void overrideProperty() {
        assertEquals(securityManager, System.getSecurityManager(
));
    }
}
```

> 这个没研究透 例子不一定能跑，我自己也理解不了 以后再研究

JUnit Toolbox

> 直译就是JUnit的工具箱，官网地址

JUnit的工具箱提供了一些用JUnit编写的自动化测试类:

- MultithreadingTester--辅助类,可以同时运行多个线程,用于做压力测试
- PollingWait--等待异步操作
- ParallelRunner--用多个工作线程同时调用@Theory方法分配参数,同时还调用所有的@Test方法
- ParallelParameterized--替换JUnit的Parameterized,多线程同时运行所有的@Test方法,并且分配参数
- WildcardPatternSuite--用来替换Suite和Categories所以用法也基本一样,但是允许你使用通配符模式来指定子类测试套件类。
- ParallelSuite--WildcardPatternSuite的扩展，同时使用多个工作线程执行其子类。虽然它延伸WildcardPatternSuite,但不强迫使用通配符模式，也可以执行使用JUnit的@SuiteClasses注解标记的子类

maven依赖

```
<dependency>
    <groupId>com.googlecode.junit-toolbox</groupId>
    <artifactId>junit-toolbox</artifactId>
    <version>2.2</version>
</dependency>
```

MultithreadingTester

1. 如果你想同时在多线程中运行一个或多个RunnableAsserts。应该这样做:new MultithreadingTester().add(...).run();例如:

```
@Test(timeout = 5000)
public void test() {
    RunnableAssert ra = new RunnableAssert("foo") {
        @Override
        public void run() {
            fail("foo");
        }
    };
    boolean success = false;
    try {
        new MultithreadingTester().add(ra).run();
        success = true;
    } catch (Throwable expected) {}
    assertFalse(success);
}
```

2. 在任何一个线程中如果抛出了异常或者错误,run()方法(还有调用了run()的方法)会到失败,在默认情况下线程数是100,每个线程执行RunnableAssert1000次.如果你想改变这些默认值,你可以这样:new MultithreadingTester().numThreads(...).numRoundsPerThread(...).add(...).run(); 例如设置两个线程数,每个线程执行断言一次:

```
@Test(timeout = 5000)
public void test_with_long_running_worker() {
    new MultithreadingTester().numThreads(2).numRoundsPerTh
read(1).add(() -> {
        Thread.sleep(1000);
        return null;
    }).run();
}
```

3. 自定义CountingRunnableAssert用来统计执行次数

```
 private class CountingRunnableAssert extends RunnableAssert
{
     protected AtomicInteger count = new AtomicInteger(0);

     protected CountingRunnableAssert() {
         super("CountingRunnableAssert");
     }

     @Override
     public void run() {
         count.incrementAndGet();
     }
 }

 @Test(timeout = 5000)
 public void test_with_one_RunnableAssert() {
     CountingRunnableAssert ra1 = new CountingRunnableAssert
();
     new MultithreadingTester().numThreads(11)
                               .numRoundsPerThread(13)
                               .add(ra1)
                               .run();
     assertThat(ra1.count.get(), is(11 * 13));
 }
```

也可以多个CountingRunnableAssertRunnableAssert一起使用,但是分配到每个CountingRunnableAssert的执行次数是根据执行总次数来不确定分配的,RunnableAssert可以add的个数最大值跟线程数相等.

```
 //不平均分配执行次数
 @Test(timeout = 5000)
 public void test_with_two_RunnableAsserts() {
     CountingRunnableAssert ra1 = new CountingRunnableAssert
();
     CountingRunnableAssert ra2 = new CountingRunnableAssert
();
     new MultithreadingTester().numThreads(3)
                               .numRoundsPerThread(1)
                               .add(ra1)
                               .add(ra2)
                               .run();
     //不平均分配执行次数
     assertThat(ra1.count.get(), is(2));
     assertThat(ra2.count.get(), is(1));
 }
```

```
 //RunnableAssert不能超过numThreads
 @Test(timeout = 5000)
 public void test_with_more_RunnableAsserts_than_threads() {
     RunnableAssert ra = new CountingRunnableAssert();
     MultithreadingTester mt = new MultithreadingTester().nu
mThreads(2)
                               .add(ra)
                               .add(ra)
                               .add(ra);
     try {
         mt.run();
         fail("IllegalStateException expected");
     } catch (IllegalStateException expected) {
         System.out.println(expected);
     }
 }
```

4. 这种情况会发生死锁

```
@Test(timeout = 5000)
 public void test_that_deadlock_is_detected() {
     try {
         final Object lock1 = new Object();
```

```
        final CountDownLatch latch1 = new CountDownLatch(1)
;
        final Object lock2 = new Object();
        final CountDownLatch latch2 = new CountDownLatch(1)
;
        new MultithreadingTester().numThreads(2).numRoundsP
erThread(1).add(
            new RunnableAssert("synchronize on lock1 and lo
ck2") {
                @Override
                public void run() throws Exception {
                    synchronized (lock1) {
                        latch2.countDown();
                        latch1.await();
                        synchronized (lock2) {
                            fail("Reached unreachable state
ment.");
                        }
                    }
                }
            },
            new RunnableAssert("synchronize on lock2 and lo
ck1") {
                @Override
                public void run() throws Exception {
                    synchronized (lock2) {
                        latch1.countDown();
                        latch2.await();
                        synchronized (lock1) {
                            fail("Reached unreachable state
ment.");
                        }
                    }
                }
            }
        ).run();
        fail("RuntimeException expected");
    } catch (RuntimeException expected) {
        assertThat(expected.getMessage(), allOf(
            containsString("Detected 2 deadlocked threads:\
n"),
            containsString("MultithreadingTesterTest.java")
```

```
,
                containsString("MultithreadingTesterTest.java")
            ));
        }
 }
```

PollingWait

等待异步操作的辅助类

> 直译:等待投票

1. pollEvery：每隔多久投票一次; unit:如果执行过程中报错了就会重复执行给定的RunnableAssert,直到配置好的超时时间后抛出一个AssertionError(换句话说,如果在指定的超时时间内执行成功一次就算这个测试成功)。每次调用sleep()方法都会重新配置pollEvery()的间隔时间并释放CPU的其他线程;timeoutAfter:设置超时时间

```
    package com.junit.learning.toolbox;


    import com.googlecode.junittoolbox.PollingWait;
    import com.googlecode.junittoolbox.RunnableAssert;

    import org.junit.Test;

    import java.util.concurrent.atomic.AtomicInteger;

    import static java.util.concurrent.TimeUnit.SECONDS;
    import static org.assertj.core.api.Assertions.assertThat;

    public class PollingWaitTest {
        private PollingWait wait = new PollingWait().timeoutAfte
r(5, SECONDS)
                .pollEvery(1, SECONDS);

        @Test
        public void test_auto_complete() throws Exception {
            final AtomicInteger atomicInteger = new AtomicIntege
r(0);
            wait.until(new RunnableAssert("'cheesecake' is displ
ayed in auto-complete <div>") {
                @Override
                public void run() throws Exception {
                    atomicInteger.incrementAndGet();

                    System.out.println(atomicInteger.get());
                    if(atomicInteger.get()==6){
                        assertThat(this.toString()).contains("'c
heesecake'");
                    }
                    assertThat(this.toString()).contains("'chees
ecake111'");
                }
            });
        }
    }
```

这里的时间都是不精确的，存在误差

ParallelRunner

用多个工作线程同时调用@Theory方法分配参数,同时还调用所有的@Test方法，继承并拓展JUnit的Theories，默认情况下测试的最大线程数将是可利用的处理器的数目。

```
package com.junit.learning.toolbox;

import com.googlecode.junittoolbox.ParallelRunner;

import org.junit.Test;
import org.junit.experimental.theories.Theory;
import org.junit.experimental.theories.suppliers.TestedOn;
import org.junit.runner.RunWith;

import static org.hamcrest.core.IsNot.not;
import static org.junit.Assume.assumeThat;

@RunWith(ParallelRunner.class)
public class ParallelRunnerTest {


    @Theory
    public void th(@TestedOn(ints = {0, 5, 10}) int amount,
            @TestedOn(ints = {0, 1, 2}) int m) {
        assumeThat(m, not(0));
        System.out.println("amount:" + amount + ",m:" + m);
    }

    @Test
    public void test1() throws InterruptedException {
        System.out.println("test1");
    }

    @Test
    public void test2() throws InterruptedException {
        System.out.println("test2");
    }

    @Test
    public void test3() throws InterruptedException {
        System.out.println("test3");
    }


}
```

mockito

Mockito介绍

mockito的介绍： Mocking framework for unit tests written in Java.

一句话：mockito是目前业界使用最广泛的Java单元测试mock框架。

升级2.0版本

目前mockito的稳定版本是1.10.19, 最新的版本是2.0.48-beta。

> 注：以上信息截至2016-2-24.

在2.0版本中mockito做了很多改进，mockito官方也推荐升级到2.0，理由如下：

> In order to continue improving Mockito and further improve the unit testing experience, we want you to upgrade to 2.0. 为了持续改进Mockito和提东改进单元测试体验，我们希望您升级到2.0

当然，2.0的API和1.0相比有些改变，也正是如此版本号才从1.0升级到2.0：

> Mockito follows semantic versioning and contains breaking changes only on major version upgrades. In the lifecycle of a library, breaking changes are necessary to roll out a set of brand new features that alter the existing behavior or even change the API. We hope that you enjoy Mockito 2.0!

不兼容的变更列表：

- Mockito和Hamcrest解藕，并修改了定制的匹配器(matchers)API
- 调整Stubbing API，避免在JDK7+平台上出现的难以回避的编译期警告(逻辑有

矛盾啊)。这仅仅影响二进制兼容性，编译兼容性不受影响.

> TBD: 第二条不理解，后面看懂了再来补充说明具体改变。

mockito资料

网站

- 官网
- 代码托管于github

文档

- Javadoc兼官方文档： 见过的最漂亮的Javadoc

教程

- Mockito refcard in dzone

书籍

- Mockito for Spring

Javadoc兼官方文档

前言

Mockito有个挺有意思的地方，它家的官方文档就是Javadoc, mockito是这么解释的：

> All documentation is kept in javadocs because it guarantees consistency between what's on the web and what's in the source code. It allows access to documentation straight from the IDE even if you work offline. It motivates Mockito developers to keep documentation up-to-date with the code that they write, every day, with every commit.
>
> 所有文档都保存在javadoc中，因为这样可以保证web内容和源代码的一致性。甚至当你在线下工作时也可以从IDE中直接访问文档。它鼓励mockito的开发人员每天每次提交都保持文档和他们编写的代码的及时更新。

这里翻译来自mockito javadoc中的部分内容。

内容

- Mockito Javadoc首页：由于内容太长，拆分成多个部分

Mockito 1-9节

> 内容翻译自 Mockito Javadoc首页 的1-9节。

1. 让我们验证某些行为

```
//静态导入mockito，这样代码看上去干净一些
import static org.mockito.Mockito.*;

//创建mock
List mockedList = mock(List.class);

//使用 mock 对象
mockedList.add("one");
mockedList.clear();

//验证
verify(mockedList).add("one");
verify(mockedList).clear();
```

Once created, a mock will remember all interactions. Then you can selectively verify whatever interactions you are interested in.

一旦创建，mock对象会记住所有的交互。然后你可以有选择性的验证你感兴趣的任何交互。

> 注：这里和easymock的默认行为(strick mock)不同

2. 再来一点stubbing?

```
//可以mock具体的类，而不仅仅是接口
LinkedList mockedList = mock(LinkedList.class);

//存根(stubbing)
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

//下面会打印 "first"
System.out.println(mockedList.get(0));

//下面会抛出运行时异常
System.out.println(mockedList.get(1));

//下面会打印"null" 因为get(999)没有存根(stub)
System.out.println(mockedList.get(999));

// 虽然可以验证一个存根的调用，但通常这是多余的
// 如果你的代码关心get(0)返回什么，那么有某些东西会出问题(通常在verify()被调用之前)
// 如果你的代码不关系get(0)返回什么，那么它不需要存根。如果不确信，那么还是验证吧
verify(mockedList).get(0);
```

- 默认情况， 对于返回一个值的所有方法， mock对象在适当的时候要不返回null，基本类型/基本类型包装类，或者一个空集合。比如int/Integer返回0, boolean/Boolean返回false。
- 存根(stub)可以覆盖： 例如通用存根可以固定搭建但是测试方法可以覆盖它。请注意覆盖存根是潜在的代码异味(code smell)，说明存根太多了
- 一旦做了存根，方法将总是返回存根的值，无论这个方法被调用多少次
- 最后一个存根总是更重要 - 当你用同样的参数对同一个方法做了多次存根时。换句话说：存根顺序相关，但是它只在极少情况下有意义。例如，当需要存根精确的方法调用次数，或者使用参数匹配器等。

3. 参数匹配器

mockito使用java原生风格来验证参数的值： 使用equals()方法。有些时候，如果需要额外的灵活性，应该使用参数匹配器：

```
//使用内建anyInt()参数匹配器
when(mockedList.get(anyInt())).thenReturn("element");

//使用自定义匹配器(这里的isValid()返回自己的匹配器实现)
when(mockedList.contains(argThat(isValid()))).thenReturn("elemen
t");

//下面会打印 "element"
System.out.println(mockedList.get(999));

//同样可以用参数匹配器做验证
verify(mockedList).get(anyInt());
```

参数匹配器容许灵活验证或存根。点击 这里 查看更多内建的pipil和自定义参数匹配器/hamcrest 匹配器的例子。

更多的关于自定义参数匹配器的单独的信息，请见类 ArgumentMatcher 的javadoc。

请合理使用复杂的按数匹配器。使用带有少量anyX()的equals()的原生匹配风格易于提供整洁而简单的测试。有时更应该重构代码以便容许equals()匹配，甚至实现equals()方法来帮助测试。

另外，请阅读 第15章 或者 类ArgumentCaptor的javadoc。ArgumentCaptor 是一个特殊的参数匹配器实现，为后面的断言捕获参数的值。

参数匹配警告：

如果使用参数匹配器，所有的参数都不得不通过匹配器提供。

下面的例子：

```
verify(mock).someMethod(anyInt(), anyString(), eq("third argumen
t"));
//上面是正确的 - eq(0也是参数匹配器)

verify(mock).someMethod(anyInt(), anyString(), "third argument")
;
//上面不正确 - 会抛出异常因为第三个参数不是参数匹配器提供的
```

类似anyObject(), eq() 的匹配器方法 不 返回匹配器。实际上，他们在栈上记录一个匹配器并返回一个虚假值(dummy，通常是null)。这个实现为了java编译器的静态类型安全。推论是不能在verified/stubbed 方法外使用anyObject(), eq()方法。

4. 验证精确调用次数/至少X次/从不

```
//使用mock
mockedList.add("once");

mockedList.add("twice");
mockedList.add("twice");

mockedList.add("three times");
mockedList.add("three times");
mockedList.add("three times");

//下面两个验证是等同的 - 默认使用times(1)
verify(mockedList).add("once");
verify(mockedList, times(1)).add("once");

//验证精确调用次数
verify(mockedList, times(2)).add("twice");
verify(mockedList, times(3)).add("three times");

//使用using never()来验证. never()相当于 times(0)
verify(mockedList, never()).add("never happened");

//使用 atLeast()/atMost()来验证
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("five times");
verify(mockedList, atMost(5)).add("three times");
```

默认**times(1)**。因此可以不用写times(1)。

5. 使用exception做void方法的存根

```
doThrow(new RuntimeException()).when(mockedList).clear();

//下面会抛出 RuntimeException:
mockedList.clear();
```

在段落12 中查看更多关于 doThrow|doAnswer 方法家族的信息。

最初， stubVoid(Object) 方法被用于存根void方法。目前stubVoid(Object)被废弃，被doThrow(Throwable...)取代。这是为了提高doAnswer(Answer) 方法家族的可读性和一致性。

6. 验证顺序

```
// A. 单个Mock，方法必须以特定顺序调用
List singleMock = mock(List.class);

//使用单个Mock
singleMock.add("was added first");
singleMock.add("was added second");

//为singleMock创建 inOrder 检验器
InOrder inOrder = inOrder(singleMock);

//下面将确保add方法第一次调用是用"was added first",然后是用"was added second"
inOrder.verify(singleMock).add("was added first");
inOrder.verify(singleMock).add("was added second");

// B. 多个Mock必须以特定顺序调用
List firstMock = mock(List.class);
List secondMock = mock(List.class);

//使用mock
firstMock.add("was called first");
secondMock.add("was called second");

//创建 inOrder 对象，传递任意多个需要验证顺序的mock
InOrder inOrder = inOrder(firstMock, secondMock);

//下面将确保firstMock在secondMock之前调用
inOrder.verify(firstMock).add("was called first");
inOrder.verify(secondMock).add("was called second");

// Oh, 另外 A + B 可以任意混合
```

顺序验证是灵活的 - 不需要逐个验证所有交互 ，只需要验证那些你感兴趣的需要在测试中保持顺序的交互。

7. 确保交互从未在mock对象上发生

```
//使用mock - 仅有mockOne有交互
mockOne.add("one");

//普通验证
verify(mockOne).add("one");

//验证方法从未在mock对象上调用
verify(mockOne, never()).add("two");

//验证其他mock没有交互
verifyZeroInteractions(mockTwo, mockThree);
```

8. 发现冗余调用

```
//使用mock
mockedList.add("one");
mockedList.add("two");

verify(mockedList).add("one");

//下面的验证将会失败
verifyNoMoreInteractions(mockedList);
```

警告：默写做过很多经典的 expect-run-verify mock的用户倾向于非常频繁的使用verifyNoMoreInteractions()，甚至在每个测试方法中。不推荐在每个测试中都使用verifyNoMoreInteractions()。verifyNoMoreInteractions()是交互测试工具集中的便利断言。仅仅在真的有必要时使用。滥用它会导致定义过度缺乏可维护性的测试。可以在这里找到更多阅读内容。

可以看 never() - 这个更直白并且将意图交代的更好。

9. 创建mock的捷径 - @mock注解

- 最大限度的减少罗嗦的创建mock对象的代码
- 让测试类更加可读
- 让验证错误更加可读因为 field name 被用于标志mock对象

```
public class ArticleManagerTest {

   @Mock private ArticleCalculator calculator;
   @Mock private ArticleDatabase database;
   @Mock private UserProvider userProvider;

   private ArticleManager manager;
```

重要的是： 需要在基类或者test runner中的加入：

```
    MockitoAnnotations.initMocks(testClass);
```

可以使用内建的runner: MockitoJUnitRunner 或者 rule: MockitoRule.

更多内容请看这里: MockitoAnnotations

Mockito 10-19节

> 内容翻译自 Mockito Javadoc首页 的10-19节。

10. 存根连续调用(游历器风格存根)

有时我们需要为同一个方法调用返回不同值/异常的存根。典型使用场景是mock游历器。早期版本的mockito没有这个特性来改进单一模拟。例如，为了替代游历器可以使用Iterable或简单集合。那些可以提供存根的自然方式（例如，使用真实的集合）。在少量场景下存根连续调用是很有用的，如：

```
when(mock.someMethod("some arg"))
.thenThrow(new RuntimeException())
.thenReturn("foo");

//第一次调用：抛出运行时异常
mock.someMethod("some arg");

//第二次调用: 打印 "foo"
System.out.println(mock.someMethod("some arg"));

//任何连续调用: 还是打印 "foo" (最后的存根生效).
System.out.println(mock.someMethod("some arg"));
```

可供选择的连续存根的更短版本：

```
when(mock.someMethod("some arg"))
.thenReturn("one", "two", "three");
```

11. 带回调的存根

容许用一般的[Answer]接口做存根。

还有另外一种有争议的特性，最初没有包含的mockito中。推荐简单用 thenReturn() 或者 thenThrow() 来做存根， 这足够用来测试/测试驱动任何干净而简单的代码。然而，如果你对使用一般Answer接口的存根有需要，这里是例子：

```
when(mock.someMethod(anyString())).thenAnswer(new Answer() {
 Object answer(InvocationOnMock invocation) {
     Object[] args = invocation.getArguments();
     Object mock = invocation.getMock();
     return "called with arguments: " + args;
 }
});

//下面会 "called with arguments: foo"
System.out.println(mock.someMethod("foo"));
```

12. doReturn()|doThrow()|doAnswer()|doNothing()|doCallRealMethod() 方法家族

存根void方法需要when(Object)之外的另一个方式，因为编译器不喜欢括号内的void方法.....

doThrow(Throwable...) 替代 stubVoid(Object) 方法来存根void. 主要原因是改善和doAnswer()方法的可读性和一致性。

当想用异常来存根void方法时使用 doThrow():

```
doThrow(new RuntimeException()).when(mockedList).clear();

//下面会抛出 RuntimeException:
mockedList.clear();
```

可以使用 doThrow(), doAnswer(), doNothing(), doReturn() 和 doCallRealMethod() 代替响应的使用when()的调用， 用于任何方法。下列情况是必须的：

- 存根void方法
- 在spy对象上存根方法 (看下面)
- 多次存根相同方法, 在测试中间改变mock的行为

如果喜欢可以用这些方法代替响应的 when()，用于所有存根调用。

13. 监视(Spy)实际对象

可以创建实际对象的间谍(spy)。当使用spy时，真实方法被调用(除非方法被存根)。

只能小心而偶尔的使用spy，例如处理遗留代码。

在真实对象上做spy可以和"部分模拟"的概念关联起来。在1.8版本之前， mockito spy不是真实的部分模拟。理由是我们觉得部分mock是代码异味。在某一时刻我们发现了部分模拟的合法使用场景（第三方接口，遗留代码的临时重构，完整的话题在这里.

```
List list = new LinkedList();
List spy = spy(list);

//随意的存根某些方法
when(spy.size()).thenReturn(100);

//使用spy调用真实方法
spy.add("one");
spy.add("two");

//打印 "one" - 列表中的第一个元素
System.out.println(spy.get(0));

//size() 方法是被存根了的 - 打印100
System.out.println(spy.size());

//随意验证
verify(spy).add("one");
verify(spy).add("two");
```

Spy实际对象时的重要提示！

1. 有时使用when(Object) 来做spy的存根是不可能或者行不通的。在这种情况下使用sky请考虑doReturn|Answer|Throw() 方法家族来做存根。例如：

```
List list = new LinkedList();
List spy = spy(list);

//不可能: 真实方法被调用因此spy.get(0) 会抛出IndexOutOfBoundsException (列表现在还是空的)
when(spy.get(0)).thenReturn("foo");

//可以使用 doReturn() 来做存根
doReturn("foo").when(spy).get(0);
```

2. mockito 不会 将调用代理给被传递进去的实际实例，取而代之的是创建它的一个拷贝。因此如果你持有真实实例并和它交互，不要期待spy会感知到这些交互和实际实例的状态影响。推论是说，当一个非存根方法在sky上被调用，而不是在真实实例上调用，真实实例不会有任何影响。

3. 对final方法保持警惕。mockito不mock final方法，因此底线是：当你在一个真实对象上sky + 你想存根一个final方法 = 问题。同样也无法验证这些方法。

14. 改变未存根调用的默认返回值(Since 1.7)

可以创建返回值使用特殊策略的mock。这是一个高级特性，写正统测试时通常不需要。然后，在处理遗留系统时有用。

这是默认应答，因此仅当你没有存根方法调用时使用：

```
Foo mock = mock(Foo.class, Mockito.RETURNS_SMART_NULLS);
Foo mockTwo = mock(Foo.class, new YourOwnAnswer());
```

更多信息请见 Answer: RETURNS_SMART_NULLS。

15. 为进一步断言捕获参数 (Since 1.8.0)

mockito用自然java风格验证参数的值：使用equals()方法。同样这也是推荐的参数匹配的方式因为它使得测试干净而简单。但是在某些情况下，在实际验证之后对特定参数做断言是很有用的。例如：

```
ArgumentCaptor<Person> argument = ArgumentCaptor.forClass(Person
.class);
verify(mock).doSomething(argument.capture());
assertEquals("John", argument.getValue().getName());
```

警告：推荐在验证时使用ArgumentCaptor，而不是存根。在存根时使用ArgumentCaptor会减少测试的可读性，因为捕获器是在断言(验证或者'then')块的外面创建。也可能会降低defect localization(?)因为如果存根方法没有被调用那么就没有参数被捕获。

某种程度上，ArgumentCaptor 和自定义参数匹配器有关联。这两个技术都被用于确认传递给mock的特定参数。但是，ArgumentCaptor在下列情况下可能会更适合些：

- 自定义参数匹配器不适合重用。
- 仅仅需要用它来断言参数的值以便完成验证

16. 眞实部分模拟(Since 1.8.0)

最终，在邮件列表中做了大量内部辩论和讨论之后，部分模拟的支持被添加到mockito中。开始我们认为部分模拟是代码异味。然后，我们发现了一个部分模拟的合法使用场景 - 更多内容请见这里.

在release 1.8之前 sky()并没有产生眞正的部分模拟，而对一些用户它是令人困惑的。更多关于spy的请见这里,或者spy(Object)方法的 javdoc) 。

```
//用spy()方法创建partial mock:
List list = spy(new LinkedList());

//可以有选择的在mock上开启partial mock
Foo mock = mock(Foo.class);
//确保实际实现是'安全的'。
//如果实际实现抛出异常或者依赖对象的特定状态，那么将会遇到麻烦
when(mock.someMethod()).thenCallRealMethod();
```

常用需要阅读partial mock的警告：面对对象编程或多或少都是通过将复杂问题分解为独立，特定，SRP风格(Single Responsibility Principle/单一职责原则)的对象来解决复杂度。partial mock符合这种规范吗？恩，当然不是... partial mock通常意味找

复杂度已经被转移到同一个对象的一个不同方法。在大多数情况下，这不是设计应用的好方式。

但是，还是有极少场景适合用partial mock：处理不能轻易改动的代码（第三方接口，遗留代码临时重构等）。但是，不要在测试驱动和良好设计的新代码上用partial mock.

17. 重置mock(Since 1.8.0)

聪明的mockito用户几乎不用这个特性，因为他们知道这是糟糕设计的信号。通常，不需要重置mock，只需要为每个测试方法创建新的mock。

以其使用reset()方法，不如考虑编写简单，短小而专注的测试方法，胜过冗长的过定义的测试。首要的潜在代码异味是reset()在测试方法的中间。这可能意味着有太多测试。听从测试方法的细语："请让我们保持短小而专注于单一行为"。在mockito的邮件列表中有一些讨论。

添加reset()方法的唯一理由是让和容器注入的mock一起工作变得可能。看 issue 55 或者 FAQ.

不要自残！在测试方法中的reset()是代码异味(可能测试的太多了).

```
List mock = mock(List.class);
when(mock.size()).thenReturn(10);
mock.add(1);

reset(mock);
//在这里mock忘记了所有的交互和存根
```

18. 故障诊断 & 校验框架使用 (Since 1.8.0)

首先，对于任何问题，鼓励阅读mockito FAQ：
http://code.google.com/p/mockito/wiki/FAQ

有问题也可以发邮件到 mockito 邮件列表: http://groups.google.com/group/mockito

其次，如果你使用正确使用，你应该了解Mockito validates。可以阅读validateMockitoUsage() 的 javadoc)

19. 行为驱动开发的别名 (Since 1.8.0)

编写测试的行为驱动开发(Behavior Driven Development)风格使用 //given //when //then 注释作为测试方法的基本部分。这正式我们编写测试的方式，强烈推荐大家这么做！

在这里开始学习 BDD : http://en.wikipedia.org/wiki/Behavior_Driven_Development

问题在于当前的stubbing api, 使用when关键字的权威规则, 不能很好的和//given //when //then 注释集成。这是因为stubbing属于测试的given模块而不是测试的when模块。为此BDDMockito类引入了一个别名来通过BDDMockito.given(Object)方法来存根方法调用。现在它真的可以非常好的和BDD风格的测试的given模块集成在一起：

测试看上去会是这个样子：

```
import static org.mockito.BDDMockito.*;

Seller seller = mock(Seller.class);
Shop shop = new Shop(seller);

public void shouldBuyBread() throws Exception {
    //given
    given(seller.askForBread()).willReturn(new Bread());

    //when
    Goods goods = shop.buyBread();

    //then
    assertThat(goods, containBread());
}
```

Mockito 20-29节

> 内容翻译自 Mockito Javadoc首页 的10-19节。

TBD: 内容实在太多了，后面内容慢慢再学习/翻译。

Mockito 30-35节

> 内容翻译自 Mockito Javadoc首页 的30-35节。

TBD: 内容实在太多了，后面内容慢慢再学习/翻译。

PowerMock

背景

由于 Mockito 不提供对静态方法、构造方法、私有方法， Final 方法和 Final 类的模拟支持，因此在不得已的时刻，我们不得不在 mockito 之外寻求其他工具。

PowerMock 这样介绍自己的：

> PowerMock 是一个java框架，容许你以普通的方式对被视为不可测试的代码做单元测试。

这里会有一个争议：代码的可测试性和良好设计之间的平衡。

PowerMock 对此的解释是：

> 编写单元测试可以是很困难的，而有时只考虑可测试性的话好的设计会被扼杀。通常可测试性对应到好的设计，但是并不是每次都这样。例如final class和方法无法使用，私有方法有些需要改成 protected 或者不必要的移动到一个协作的静态的方法，这些应该完全避免。诸如此类，仅仅是因为现有(mock)框架的限制。

介绍

PowerMock 是一个扩展其他mock类库如 EasyMock (还有mocktito)的框架，提供更加强大的能力。PowerMock 使用定制的类加载器和字节码操作来实现静态方法、构造函数、fianl 类和方法，private方法，移除静态初始化和其他。通过使用定制的类装载器，不需要修改IDE或者持续集成服务器，这简化了使用。熟悉支持mock框架的开发者将发现PowerMock易于使用，因为整个 expectation API是相同的，都可以用于静态方法和构造函数。PowerMock致力于使用少量方法和注解来扩展现有API以实现额外的功能。目前，PowerMock 仅支持 EasyMock 和 Mockito。

> 注： 因为技术选型原因，后面只展示 Junit4 + Mockito + PowerMock 的使用。

在编写单元测试时，跳过封装通常是很有用的，因此 PowerMock 包含了一些可以简化反射的功能，对于测试非常有用。这容许轻易的访问内部状态，但是也同样简化了partical mock和私有mock。

警告

注意：PowerMock 主要为在单元测试领域有熟练知识的人准备。初级开发者使用它可能弊大于利。

资料

- PowerMock@github

参考资料

- 使用 PowerMock 以及 Mockito 实现单元测试
- PowerMock介绍

官方文档

在 PowerMock 的 github 首页，有一份文档：

https://github.com/jayway/powermock

这里将翻译这封文档(仅仅限于Junit 4.12 + mockito 1.8 + powermock相关的内容)。

该文件修订时间： 2017-03-13 05:37:11

开始使用

> 注： 内容翻译自官网文档 Getting Started

API's

PowerMock 包含两个扩展API。一个用于EasyMock，一个用于Mockito。为了使用PowerMock，需要依赖这两个API中的其中一个，此外还有测试框架。

当前 PowerMock 支持 Junit 和 TestNG。有三个不同的Junit test executor可用，一个用于Junit4.4+，一个用于Junit4.0到4.3,还有一个用于Junit 3.

有一个test executor用于TestNG，要求版本5.11+，取决于使用哪个版本的PowerMock。

> 注：由于我们现在使用的是junit最新版本4.12,因此只关注用于Junit4.4+的test executor。

编写测试

像这样编写测试：

```
@RunWith(PowerMockRunner.class)
@PrepareForTest( { YourClassWithEgStaticMethod.class })
public class YourTestCase {
...
}
```

> 注:PowerMock的案例代码是采用easymock编写，因此后面会自己用mockito编写一套自己的案例。

搭建Maven

> 注：重申，只关注 junit4.12 + mocktito 1.× + powermock 最新版本(1.6.5)

```
<properties>
    <powermock.version>1.6.5</powermock.version>
</properties>
<dependencies>
   <dependency>
      <groupId>org.powermock</groupId>
      <artifactId>powermock-module-junit4</artifactId>
      <version>${powermock.version}</version>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.powermock</groupId>
      <artifactId>powermock-api-mockito</artifactId>
      <version>${powermock.version}</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

其他特殊用法

> 注：以下内容还没有仔细研究，后续再来一一琢磨。

需要和其他Junit Runner联合使用？

- 首先尝试 JUnit Delegation Runner，如果不能工作再尝试 PowerMockRule 或者 PowerMock Java Agent.

需要使用Juit Rule引导？

- PowerMockRule 将用于此处

基于Java Agent的引导？

- 使用 PowerMock Java Agent，当使用PowerMock遇到类装载问题时

动机

> 注： 内容翻译自官方文档 Motivation

TBD.....

Mockito的使用

> 注：内容翻译自官方文档 Mockito Usage

TBD.....

使用案例

模拟 final 类和方法

非final的情况

假定我们有这样一个类(和它的方法getSomething())需要mock，首先看如果类不是final的通常情况：

```
public class FinalClassDemo {
    public int getSomething() {
        return 0;
    }
}
```

用mockito就足以轻易搞定：

```
@RunWith(MockitoJUnitRunner.class)
public class FinalClassDemoTest {
    @Mock
    private FinalClassDemo demo;

    @Test
    public void getSomething() throws Exception {
        when(demo.getSomething()).thenReturn(5);
        assertThat(demo.getSomething()).isEqualTo(5);
    }
}
```

模拟 final 类

但是当类变成 final 之后：

```
public final class FinalClassDemo {}
```

mockito就无能为例，上面的测试代码报错如下:

```
org.mockito.exceptions.base.MockitoException:
Cannot mock/spy class com.github.skyao.test.FinalClassDemo
Mockito cannot mock/spy following:
  - final classes
  - anonymous classes
  - primitive types

    at org.mockito.internal.runners.JUnit45AndHigherRunnerImpl$1
.withBefores(JUnit45AndHigherRunnerImpl.java:27)
    ......
```

此时需要引入PowerMock来解决对类FinalClassDemo的mock问题，需要：

1. 将~~@RunWith(MockitoJUnitRunner.class)~~ 替换为 `@RunWith(PowerMockRunner.class)`
2. 增加 `@PrepareForTest(FinalClassDemo.class)` 指定需要特别处理的mock类

代码如下：

```
import org.powermock.core.classloader.annotations.PrepareForTest
;
import org.powermock.modules.junit4.PowerMockRunner;

@RunWith(PowerMockRunner.class)
@PrepareForTest(FinalClassDemo.class)
public class FinalClassDemoTest {
    @Mock
    private FinalClassDemo demo;

    @Test
    public void getSomething() throws Exception {
        when(demo.getSomething()).thenReturn(5);
        assertThat(demo.getSomething()).isEqualTo(5);
    }
}
```

比较有意思的是，上述代码中的 `when()` 方法还是之前的代码，即用的是 `Mockito.when()` ，无需修改。这也提现了 PowerMock 的设计思想：只要修改少量的代码，通过注解引入少许 PowerMock 的内容，就可以让原有mock框架(这

里是Mockito)继续按照它原有的方式继续工作。

当然，测试验证，这里的 when() 修改为 `PowerMockito.when()` 也是可以同样跑起来的。

```
@Test
public void getSomething() throws Exception {
    PowerMockito.when(demo.getSomething()).thenReturn(5);
    ......
}
```

模拟 **final** 方法

类似的，如果类是非final的，但是方法是final的方法：

```
public class FinalMethodDemo {
    public final int getSomething() {
        return 0;
    }
}
```

mockto默认情况下也是无能为力，报错如下：

```
org.mockito.exceptions.misusing.MissingMethodInvocationException
:
when() requires an argument which has to be 'a method call on a
mock'.
For example:
    when(mock.getArticles()).thenReturn(articles);

Also, this error might show up because:
1. you stub either of: final/private/equals()/hashCode() methods
.
   Those methods *cannot* be stubbed/verified.
   Mocking methods declared on non-public parent classes is not
supported.
2. inside when() you don't call method on mock but on some other
 object.


    at com.github.skyao.test.FinalMethodDemoTest.getSomething(Fi
nalMethodDemoTest.java:23)
```

注意上述错误信息明确指出：`final/private/equals()/hashCode()` 这些方法无法支持。

同样引入PowerMock之后，就可以解决问题，代码和上面final类时完全相同。

模拟 final 类加 final 方法

补充一下，上述两种情况叠加，class是final的，方法也是final的：

```
public final class FinalMethodDemo {
    public final int getSomething() {
        return 0;
    }
}
```

PowerMock同样有效，使用方式没有变化。

模拟 **private** 方法

TBD

AssertJ

AssertJ是一个Java库，提供了流式断言,更接近自然语言。其主要目标是提高测试代码的可读性，使测试更容易维护。

AssertJ提供了JDK标准类型的断言,可以让JUnit或者TestNG使用 AssertJ的版本取决于java版本:

- AssertJ 1.x要求java6以上版本
- AssertJ 2.x要求java7以上版本
- AssertJ 3.x要求java8以上版本

maven依赖

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <!-- use 2.5.0 for Java 7 projects -->
  <version>3.5.2</version>
  <scope>test</scope>
</dependency>
```

Gradle

```
testCompile 'org.assertj:assertj-core:3.5.2'
```

资料收集

- 官网地址
- 代码仓库

AssertJ核心功能

AssertJ有很多很有用的API，你未必知道，下面是这些api的列表:

> 由于内容较多,所以这里只讲最新版本的内容

基本技巧:

java8新增的断言,请查看这里

配置IDE自动提示assertThat

如果你想输入assert就让ide自动帮你完成assertThat(并且是assertj的)的输入．你可以配置你的ide

- Eclipse配置
- Go to : preferences > Java > Editor > Content assist > Favorites > New Type
- Enter : org.assertj.core.api.Assertions
- You should see : org.assertj.core.api.Assertions.* in a the list.
- intellij idea无需特殊配置，自动会提示

用as(String description,Object... agrs)可以描述你的断言

默认的错误消息只是简单提示你预期的值和真实的值．没有具体的业务描述.使用它可以对你的断言进行描述,尤其是布尔类型的断言 你可以用as(String description,Object... agrs)设置你的描述信息,但是一定在调用断言之前设置,否则将会被忽略

```
@Test
public void test_as(){
    Jedi jedi = new Jedi("Judi", "green");
    assertThat(jedi.getName()).as("检查%s的颜色", jedi.getName()).
isEqualTo("blue");
}
```

提示信息会变成这样:

```
org.junit.ComparisonFailure: [检查Judi的颜色]
Expected :"blue"
Actual   :"Judi"
```

迭代器或者数组的过滤和断言

过滤有以下这些方式:

- java8的Predicate
- condition (org.assertj.core.api.Condition)
- 数据/迭代中的元素的一些属性/字段的操作

使用**Predicate**过滤:

```
@Test
public void test_Predicate(){
    assertThat(fellowshipOfTheRing).filteredOn( character -> cha
racter.getName().contains("o") )//过滤name属性中包含o的对象
            .containsOnly(aragorn, frodo, legolas, boromir);//断言

}
```

过滤属性或者字段：

首先你指定属性/字段名称通过给定的预期值进行筛选,该过滤器会先尝试从属性中获取value,然后再去字段中获取．默认情况下是可以读取私有字段的，你可以通过Assertions.setAllowExtractingPrivateFields(false)禁止.

过滤器支持读取嵌套属性/字段,但是如果嵌套的属性/字段有一个是null那么整个嵌套都会被认为是null.例如:如果"address.street.name"会返回null，那么"address.street"也是返回null 过滤器的基本操作:not,in,notIn

```
List<TolkienCharacter> fellowshipOfTheRing = Lists.newArrayList(
);
@Test
public void test_filters() {
    //筛选出TolkienCharacter中的race属性是HOBBIT的对象,断言包含sam,
frodo, pippin, merry
    assertThat(fellowshipOfTheRing).filteredOn("race", HOBBIT).c
ontainsOnly(sam, frodo, pippin, merry);
    //嵌套属性
    assertThat(fellowshipOfTheRing).filteredOn("race.name", "Man"
).containsOnly(aragorn, boromir);
    //筛选出race属性不是HOBBIT和MAN的对象
    assertThat(fellowshipOfTheRing).filteredOn("race", notIn(HOB
BIT, MAN)).containsOnly(gandalf, gimli, legolas);
    //筛选出race属性是MAIA和MAN的对象
    assertThat(fellowshipOfTheRing).filteredOn("race", in(MAIA,
MAN)).containsOnly(gandalf, boromir, aragorn);

    //筛选出race属性不是HOBBIT的对象
    assertThat(fellowshipOfTheRing).filteredOn("race", not(HOBBI
T))
            .containsOnly(gandalf, boromir, aragorn, gimli, lego
las);

    //支持多次过滤
    assertThat(fellowshipOfTheRing).filteredOn("race", MAN)
            .filteredOn("name", not("Boromir"))
            .containsOnly(aragorn);
}
```

为了方便理解将所有涉及到的类都列出如下:

```
public class TolkienCharacter {

  // public to test extract on field
  public int age;
  private String name;
  private Race race;
  // not accessible field to test that field by field comparison
 does not use it
  @SuppressWarnings("unused")
  private long notAccessibleField = 1;
```

```
public enum Race {

  HOBBIT("Hobbit", false, GOOD), MAIA("Maia", true, GOOD), MAN("
Man", false, NEUTRAL), ELF("Elf", true, GOOD), DWARF("Dwarf", fa
lse, GOOD), ORC("Orc", false, EVIL);

  private final String name;
  public final boolean immortal;
  private Alignment alignment;

  Race(String name, boolean immortal, Alignment alignment) {
    this.name = name;
    this.immortal = immortal;
    this.alignment = alignment;
  }
```

```
public enum Alignment {
  SUPER_EVIL, EVIL, NEUTRAL, GOOD, SUPER_GOOD;
}
```

用Condition过滤

过滤器只会保留迭代/数组中满足匹配Condition的元素 有两个方法可用:being(Condition)和having(Condition):

```
 @Test
public void test_condition_BasketBallPlayer() {
```

```
    BasketBallPlayer rose = new BasketBallPlayer(new Name("Derri
ck", "Rose"), "Chicago Bulls");
    rose.setAssistsPerGame(8);
    rose.setPointsPerGame(25);
    rose.setReboundsPerGame(5);
    BasketBallPlayer  lebron = new BasketBallPlayer(new Name("To
ny", "Parker"), "Spurs");
    lebron.setAssistsPerGame(9);
    lebron.setPointsPerGame(21);
    lebron.setReboundsPerGame(5);
    BasketBallPlayer james = new BasketBallPlayer(new Name("Lebr
on", "James"), "Miami Heat");
    james.setAssistsPerGame(6);
    james.setPointsPerGame(27);
    james.setReboundsPerGame(8);
    BasketBallPlayer dwayne = new BasketBallPlayer(new Name("Dwa
yne", "Wade"), "Miami Heat");
    dwayne.setAssistsPerGame(16);
    dwayne.setPointsPerGame(55);
    dwayne.setReboundsPerGame(16);
    BasketBallPlayer  noah = new BasketBallPlayer(new Name("Joac
him", "Noah"), "Chicago Bulls");
    noah.setAssistsPerGame(4);
    noah.setPointsPerGame(10);
    noah.setReboundsPerGame(11);

    dwayne.getTeamMates().add(james);
    james.getTeamMates().add(dwayne);

    Condition<BasketBallPlayer> mvpStats = new Condition<BasketB
allPlayer>() {
        @Override
        public boolean matches(BasketBallPlayer player) {
            return player.getPointsPerGame() > 20 && (player.get
AssistsPerGame() >= 8 || player.getReboundsPerGame() >= 8);
        }
    };

    List<BasketBallPlayer> players = newArrayList();
    players.add(rose);
    players.add(lebron);
    players.add(noah);
```

```
    assertThat(players).filteredOn(mvpStats).containsOnly(rose,
lebron);
}
```

为了方便理解列出BasketBallPlayer类:

```
public class BasketBallPlayer {

    private Name name;
    public double size;
    private float weight;
    private int pointsPerGame;
    private int assistsPerGame;
    private int reboundsPerGame;
    private String team;
    private boolean rookie;
    private List<BasketBallPlayer> teamMates = new ArrayList<Bas
ketBallPlayer>();
    private List<int[]> points = new ArrayList<>();

    public BasketBallPlayer(Name name, String team) {
        setName(name);
        setTeam(team);
    }
```

获取迭代/数组中元素的属性/字段

比如说,你有请求一个service/dao然后得到一个TolkienCharacters的集合(或者数组),想要检查结果,你需要先建立一个预期TolkienCharacters(s)·这个工作量可能会很大

```
List<TolkienCharacter> fellowshipOfTheRing = tolkienDao.findHero
es();  // 集合中包含frodo, sam, aragorn ...

// 这里你需要创建预期的TolkienCharacter:frodo,aragorn
assertThat(fellowshipOfTheRing).contains(frodo, aragorn);
```

然而,通常情况下我们只是检查集合中元素的某些属性或字段,这也你需要在断言之前写一段代码获取这些字段或者属性,比如:

```
//获取name属性
List<String> names = new ArrayList<String>();
for (TolkienCharacter tolkienCharacter : fellowshipOfTheRing) {
  names.add(tolkienCharacter.getName());
}
// ... 然后断言
assertThat(names).contains("Boromir", "Gandalf", "Frodo", "Legol
as");
```

现在你可以用assertj帮你提取这些属性了,这样做:

```
//"name"必须是TolkienCharacter的属性或者字段
assertThat(fellowshipOfTheRing).extracting("name")
                               .contains("Boromir", "Gandalf", "
Frodo", "Legolas")
                               .doesNotContain("Sauron", "Elrond"
);
```

而且你可以同时获取多个属性/字段,比如这样:

```
// 如果你想要同时检查多个属性,你必须使用tuple
import static org.assertj.core.api.Assertions.tuple;

//获取 name, age 和嵌套属性 race.name
assertThat(fellowshipOfTheRing).extracting("name", "age", "race.
name")
                               .contains(tuple("Boromir", 37, "M
an"),
                                         tuple("Sam", 38, "Hobbi
t"),
                                         tuple("Legolas", 1000,
"Elf"));
```

当前元素的name,age还有race.name的值会分组到tuple中,所以你需要用tuple来获取这些值

在只检查一个属性的时候,你可以指定这个属性的类型

```
assertThat(fellowshipOfTheRing).extracting("name", String.class)
                               .contains("Boromir", "Gandalf", "
Frodo", "Legolas")
                               .doesNotContain("Sauron", "Elrond"
);
```

更多的用法请查看这里

获取flatMap

准备:

```
private Extractor<BasketBallPlayer,List<BasketBallPlayer>>  team
Mates =new PlayerTeammatesExtractor();

public class PlayerTeammatesExtractor implements Extractor<Baske
tBallPlayer,List<BasketBallPlayer>> {

    @Override
    public List<BasketBallPlayer> extract(BasketBallPlayer input)
 {
        return input.getTeamMates();
    }
}
```

```
@Test
public void iterable_assertions_on_flat_extracted_values_examples
(){
    BasketBallPlayer rose = new BasketBallPlayer(new Name("Derri
ck", "Rose"), "Chicago Bulls");
    rose.setAssistsPerGame(8);
    rose.setPointsPerGame(25);
    rose.setReboundsPerGame(5);

    BasketBallPlayer noah = new BasketBallPlayer(new Name("Joach
im", "Noah"), "Chicago Bulls");
    noah.setAssistsPerGame(4);
    noah.setPointsPerGame(10);
    noah.setReboundsPerGame(11);

    BasketBallPlayer james = new BasketBallPlayer(new Name("Lebr
on", "James"), "Miami Heat");
    james.setAssistsPerGame(6);
    james.setPointsPerGame(27);
    james.setReboundsPerGame(8);

    BasketBallPlayer  dwayne = new BasketBallPlayer(new Name("Dw
ayne", "Wade"), "Miami Heat");
    dwayne.setAssistsPerGame(16);
    dwayne.setPointsPerGame(55);
    dwayne.setReboundsPerGame(16);

    noah.getTeamMates().add(rose);
    rose.getTeamMates().add(noah);
    james.getTeamMates().add(dwayne);

    ArrayList<BasketBallPlayer> basketBallPlayers = newArrayList
(noah, james);
    //通过指定teamMates属性,获取所有的getTeamMates()返回的集合
    assertThat(basketBallPlayers).flatExtracting("teamMates").co
ntains(dwayne, rose);
    //这里需要你实现Extractor
    assertThat(basketBallPlayers).flatExtracting(teamMates).cont
ains(dwayne, rose);
}
```

关于iterable/数组元素的返回值的断言

从iterable/数组中获取出来的对象的调用方法会被放入一个新的iterable/数组中，变成被测试的对象. 这样可以用来测试元素的调用方法的结果而不是测试元素本身.这种方式对于不符合java bean的getter规范的属性特别有意义(比如toString或者String status())

```
@Test
public void iterable_assertions_on_extracted_method_result_example() {
    // 获取'toString'返回的结果
    assertThat(fellowshipOfTheRing).extractingResultOf("getRace").contains("Frodo 33 years old Hobbit",
            "Aragorn 87 years old Man");

    assertThat(fellowshipOfTheRing).extractingResultOf("getSurname").contains("Sam the Hobbit",
            "Merry the Hobbit");
}
```

用soft断言收集所有错误

一般情况下在你的断言中如果有一个断言错误,那么整个测试就会停止,使用soft断言会运行完你的所有断言之后才停止运行,并且收集你的错误信息 如果你使用的是标准的普通的断言,例如:

```
@Test
public void host_dinner_party_where_nobody_dies() {
   Mansion mansion = new Mansion();
   mansion.hostPotentiallyMurderousDinnerParty();
   assertThat(mansion.guests()).as("Living Guests").isEqualTo(7)
;
   assertThat(mansion.kitchen()).as("Kitchen").isEqualTo("clean"
);
   assertThat(mansion.library()).as("Library").isEqualTo("clean"
);
   assertThat(mansion.revolverAmmo()).as("Revolver Ammo").isEqua
lTo(6);
   assertThat(mansion.candlestick()).as("Candlestick").isEqualTo(
"pristine");
   assertThat(mansion.colonel()).as("Colonel").isEqualTo("well k
empt");
   assertThat(mansion.professor()).as("Professor").isEqualTo("we
ll kempt");
}
```

你会得到这样的错误提示信息:

```
org.junit.ComparisonFailure: [Living Guests] expected:<[7]> but
was<[6]>
```

如果使用soft断言你就可以收集所有的失败的断言信息:

```
@Test
public void host_dinner_party_where_nobody_dies() {
   Mansion mansion = new Mansion();
   mansion.hostPotentiallyMurderousDinnerParty();
   // use SoftAssertions instead of direct assertThat methods
   SoftAssertions softly = new SoftAssertions();
   softly.assertThat(mansion.guests()).as("Living Guests").isEqu
alTo(7);
   softly.assertThat(mansion.kitchen()).as("Kitchen").isEqualTo(
"clean");
   softly.assertThat(mansion.library()).as("Library").isEqualTo(
"clean");
   softly.assertThat(mansion.revolverAmmo()).as("Revolver Ammo")
.isEqualTo(6);
   softly.assertThat(mansion.candlestick()).as("Candlestick").is
EqualTo("pristine");
   softly.assertThat(mansion.colonel()).as("Colonel").isEqualTo(
"well kempt");
   softly.assertThat(mansion.professor()).as("Professor").isEqua
lTo("well kempt");
   //这一步一定要写
   softly.assertAll();
}
```

这时候你会看到这样的提示信息:

```
org.assertj.core.api.SoftAssertionError:
     The following 4 assertions failed:
     1) [Living Guests] expected:<[7]> but was:<[6]>
     2) [Library] expected:<'[clean]'> but was:<'[messy]'>
     3) [Candlestick] expected:<'[pristine]'> but was:<'[bent]'>
     4) [Professor] expected:<'[well kempt]'> but was:<'[bloodied
 and dishevelled]'>
```

assertj还提供了JUnit的规则(rule),它会自动调用soft断言来全局收集错误信息

```
@Rule
public final JUnitSoftAssertions softly = new JUnitSoftAssertion
s();

@Test
public void host_dinner_party_where_nobody_dies() {
   Mansion mansion = new Mansion();
   mansion.hostPotentiallyMurderousDinnerParty();
   // use SoftAssertions instead of direct assertThat methods
   softly.assertThat(mansion.guests()).as("Living Guests").isEqu
alTo(7);
   softly.assertThat(mansion.kitchen()).as("Kitchen").isEqualTo(
"clean");
   softly.assertThat(mansion.library()).as("Library").isEqualTo(
"clean");
   softly.assertThat(mansion.revolverAmmo()).as("Revolver Ammo")
.isEqualTo(6);
   softly.assertThat(mansion.candlestick()).as("Candlestick").is
EqualTo("pristine");
   softly.assertThat(mansion.colonel()).as("Colonel").isEqualTo(
"well kempt");
   softly.assertThat(mansion.professor()).as("Professor").isEqua
lTo("well kempt");
   // 这里就不需要再写softly.assertAll()了
}
```

这里只能用于junit,testNG没办法做全局收集信息,因为testNG只要抛出一个异常就会跳过后面的测试

用字符串断言,断言文件内容

```
File xFile = writeFile("xFile", "The Truth Is Out There");

// 典型的文件断言
assertThat(xFile).exists().isFile().isRelative();
//  文件内容断言
assertThat(Assertions.contentOf(xFile)).startsWith("The Truth").
contains("Is Out").endsWith("There");
```

异常断言

如何断言异常被抛出，并检查它是你所预期的？ 在java8中测试断言是非常优雅的,使用assertThartThrownBy(ThrowingCallable)来捕获异常,断言Throwable.ThrowingCallable是一个功能接口,可以通过lambda来调用 例如:

```
@Test
public void testException() {
   assertThatThrownBy(() -> { throw new Exception("boom!"); }).i
sInstanceOf(Exception.class)
                                                              .ha
sMessageContaining("boom");
}
```

还有一种更自然的语法:

```
@Test
public void testException() {
   assertThatExceptionOfType(IOException.class).isThrownBy(() ->
 { throw new IOException("boom!"); })
                                                    .withMessage("%s!"
, "boom")
                                                    .withMessageConta
ining("boom")
                                                    .withNoCause();
}
```

BDD爱好者还可以这样写:

```
@Test
public void testException3() {
    //given
    List<String> list = Lists.newArrayList();

    // when
    Throwable thrown = catchThrowable(() -> { list.get(1); });

    // then
    assertThat(thrown).isInstanceOf(IndexOutOfBoundsException.cl
ass)
            .hasMessageContaining("Index");
}
```

> 官网原文还有java7的异常断言,由于我们使用的是java8这里不再翻译有兴趣的可以看这里

使用自定义的comparison来做比较断言

有的适合你不想用equalse放来比较对象,那么你可以使用以下两种方法:

- usingComparator(Comparator) : 关注对象本身的断言
- usingElementComparator(Comparator) :关注iterable/数组的元素的断言

usingComparator(Comparator)例子:

```
//frodo和sam都是race为Hobbit类型的TolkienCharacter的实例,很明显他们是
不相等的
assertThat(frodo).isNotEqualTo(sam);

//但是,如果我们仅是比较他们的race属性,那么他们是相等的
//sauron在集合fellowshipOfTheRing中
assertThat(frodo).usingComparator(raceComparator).isEqualTo(sam)
;
```

usingElementComparator(Comparator)的例子

```
//普通的比较,fellowshipOfTheRing中包含gandalf不包含sauron
assertThat(fellowshipOfTheRing).contains(gandalf).doesNotContain
(sauron);

//但是只比较race属性的话,sauron就在fellowshipOfTheRing集合中了
assertThat(fellowshipOfTheRing).usingElementComparator(raceCompa
rator).contains(sauron);
```

属性比较

assertj为属性/字段比较提供了以下几种方法:

- isEqualToComparingFieldByField : 比较各字段/属性包括继承的-不是递归
- isEqualToComparingOnlyGivenFields :仅仅比较指定的字段/属性-非递归
- isEqualToIgnoringGivenFields : 比较除了指定的字段/属性以外的字段/属性-非递归
- isEqualToIgnoringNullFields : 只比较非空字段/属性-非递归
- isEqualToComparingFieldByFieldRecursively : 比较所有的字段/属性-递归

除了isEqualToComparingFieldByFieldRecursively其他的都不是递归的.递归是指:如果对象的属性也是一个对象,那么assertj会自动调用该属性的equalse方法进行比较.另外isEqualToComparingFieldByFieldRecursively默认使用字段的比较,除非你有自定义的equalse方法实现.

以下例子中TolkienCharacter的equalse方法没有被覆盖
isEqualToComparingFieldByField例子:

```
TolkienCharacter frodo = new TolkienCharacter("Frodo", 33, HOBBI
T);
TolkienCharacter frodoClone = new TolkienCharacter("Frodo", 33,
HOBBIT);

// equalse比较的是对象引用,所以是失败的
assertThat(frodo).isEqualsTo(frodoClone);

//两者只是比较了属性值而已,所以是相等的
assertThat(frodo).isEqualToComparingFieldByField(frodoClone);
```

isEqualToComparingOnlyGivenFields例子:

```
TolkienCharacter frodo = new TolkienCharacter("Frodo", 33, HOBBI
T);
TolkienCharacter sam = new TolkienCharacter("Sam", 38, HOBBIT);

// 只是比较race属性的话,它们俩都是HOBBIT
assertThat(frodo).isEqualToComparingOnlyGivenFields(sam, "race")
; // OK

// 嵌套属性比较
assertThat(frodo).isEqualToComparingOnlyGivenFields(sam, "race.n
ame"); // OK

// name属性两者不相等
assertThat(frodo).isEqualToComparingOnlyGivenFields(sam, "name",
"race"); // FAIL
```

isEqualToIgnoringGivenFields例子:

```
TolkienCharacter frodo = new TolkienCharacter("Frodo", 33, HOBBI
T);
TolkienCharacter sam = new TolkienCharacter("Sam", 38, HOBBIT);

// 如果忽略name和age属性那么两者的race属性都是HOBBIT
assertThat(frodo).isEqualToIgnoringGivenFields(sam, "name", "age"
); // OK both are HOBBIT

// ... 如果只忽略age属性,那么就不相等了
assertThat(frodo).isEqualToIgnoringGivenFields(sam, "age"); // F
AIL
```

isEqualToIgnoringNullFields例子:

```
TolkienCharacter frodo = new TolkienCharacter("Frodo", 33, HOBBI
T);
TolkienCharacter mysteriousHobbit = new TolkienCharacter(null, 33
, HOBBIT);

// mysteriousHobbit的name属性是null所以会被忽略,只比较age和race
assertThat(frodo).isEqualToIgnoringNullFields(mysteriousHobbit);
// OK

// ... 两者的位置是不可以更换的！
assertThat(mysteriousHobbit).isEqualToIgnoringNullFields(frodo);
// FAIL
```

isEqualToComparingFieldByFieldRecursively例子:

```
public class Person {
  public String name;
  public double height;
  public Home home = new Home();
  public Person bestFriend;
  // 简洁起见,构造函数和get set方法都省略了
}

public class Home {
  public Address address = new Address();
}

public static class Address {
  public int number = 1;
}

Person jack = new Person("Jack", 1.80);
jack.home.address.number = 123;

Person jackClone = new Person("Jack", 1.80);
jackClone.home.address.number = 123;

jack.bestFriend = jackClone;
jackClone.bestFriend = jack;
```

```
// 直接比较会失败
assertThat(jack).isEqualsTo(jackClone);

// jack and jackClone是递归比较属性的
assertThat(jack).isEqualToComparingFieldByFieldRecursively(jackC
lone);


jack.height = 1.81;

//因为height属性不相等,所以断言会失败
assertThat(jack).isEqualToComparingFieldByFieldRecursively(jackC
lone);

//使用usingComparatorForType指定比较类型,这样就可以只比较height了
//由于DoubleComparator允许0.5的误差,所以断言会成功
assertThat(jack).usingComparatorForType(new DoubleComparator(0.5
), Double.class)
                .isEqualToComparingFieldByFieldRecursively(jackC
lone);

// 使用 usingComparatorForFields 指定哪些属性比较(支持嵌套属性)
assertThat(jack).usingComparatorForFields(new DoubleComparator(0
.5), "height")
                .isEqualToComparingFieldByFieldRecursively(jackC
lone);
```

Java8断言

由于内容较多这里只列举了2.5.x版本新增的特性.你可以在这里查看更多更早版本的内容

新增Predicate断言

jdk中的Predicate接口是一个标准接口，应用程序可以实现它来定义希望应用于 FilteredRowSet 对象的过滤器

下面这些Predicate断言可用

- accepts(T... values) : 如果所有给定的值匹配Predicate则表示成功.
- rejects(T... values) : 如果所有给定的值都不匹配Predicate则表示失败.
- acceptsAll(Iterable) : 如果给定的可迭代的所有元素都匹配Predicate则表示成功
- rejectsAll(Iterable) : 如果给定的可迭代的所有元素都不匹配Predicate则表示成功

accepts 和　acceptsAll的例子

```
    @Test
    public void test_accepts_and_acceptsAll() {
        Predicate<String> ballSportPredicate = sport -> sport.co
ntains("ball");
        // assertion succeeds:
        assertThat(ballSportPredicate).accepts("football").accep
ts("football", "basketball", "handball");
        assertThat(ballSportPredicate).acceptsAll(newArrayList("
football", "basketball", "handball"));

        // assertion fails because of curling :p
        assertThat(ballSportPredicate).accepts("curling");
        assertThat(ballSportPredicate).accepts("football", "bask
etball", "curling");
        assertThat(ballSportPredicate).acceptsAll(newArrayList("
football", "basketball", "curling"));
    }
```

rejects和rejectsAll的例子

```
     @Test
    public void test_rejectsAll_and_rejects(){
        Predicate<String> ballSportPredicate = sport -> sport.co
ntains("ball");

        // assertion succeeds:
        assertThat(ballSportPredicate).rejects("curling").reject
s("curling", "judo", "marathon");
        assertThat(ballSportPredicate).rejectsAll(newArrayList("
curling", "judo", "marathon"));

        // assertion fails because of football:
        assertThat(ballSportPredicate).rejects("football");
        assertThat(ballSportPredicate).rejects("curling", "judo"
, "football");
        assertThat(ballSportPredicate).rejectsAll(newArrayList("
curling", "judo", "football"));
    }
```

新增**satisfies**基础断言,它可以运行一组断言

> jdk Consumer的操作可能会更改输入参数的内部状态

验证实际的对象是否满足Consumer的特定需求. 可以对对个对象执行一组断言,也可以避免为了单个对象上使用对个断言而声明一个局部变量

多个对象执行一组断言的例子:

```
@Test
public void test_satisfied() {
    // second constructor parameter is the light saber color
    Jedi yoda = new Jedi("Yoda", "Green");
    Jedi luke = new Jedi("Luke Skywalker", "Green");

    Consumer<Jedi> jediRequirements = jedi -> {
        assertThat(jedi.getLightSaberColor()).isEqualTo("Green")
;
        assertThat(jedi.getName()).doesNotContain("Dark");
    };

    // assertions succeed:
    assertThat(yoda).satisfies(jediRequirements);
    assertThat(luke).satisfies(jediRequirements);

    // assertions fails:
    Jedi vader = new Jedi("Vader", "Red");
    assertThat(vader).satisfies(jediRequirements);
}
```

在没有局部变量的声明的情况下多次断言:

```
@Test
public void test_satisfied_no_need_define_local_variable(){
    Player team = new Player();
    Stats stats1 = new Stats();
    stats1.assistsPerGame = 8.5;
    stats1.pointPerGame = 26;
    stats1.reboundsPerGame = 9;

    ArrayList<Stats> statses = Lists.newArrayList(stats1);
    team.setPlayers(statses);
    // no need to define team.getPlayers().get(0)as a local vari
able
    assertThat(team.getPlayers().get(0)).satisfies(stats -> {
        assertThat(stats.pointPerGame).isGreaterThan(25.7);
        assertThat(stats.assistsPerGame).isGreaterThan(7.2);
        assertThat(stats.reboundsPerGame).isBetween(9.0, 12.0);
    });
}
```

registerFormatterForType可以控制错误消息的格式

断言有不同类型的错误消息格式化,registerFormatterForType提供了特定的格式化器,控制一个给定类型的格式:

- StandardRepresentation
- UnicodeRepresentation
- HexadecimalRepresentation
- BinaryRepresentation

例如:

```
@Test
public void test_registerFormatterForType(){
    // 没有具体的格式
    assertThat(STANDARD_REPRESENTATION.toStringOf(123L)).isEqual
To("123L");

    //注册一个Long的格式化
    Assertions.registerFormatterForType(Long.class, value -> "$"
 + value + "$");
    //成功
    assertThat(STANDARD_REPRESENTATION.toStringOf(123L)).isEqual
To("$123$");

    //失败
    assertThat(123L).isEqualTo(456L);
}
```

控制台的错误提示信息:

```
Expected :$456$
Actual   :$123$
```

新增hasOnlyOneElementSatisfying做迭代/数组断言

验证迭代器/数组值包含一个元素并且该元素满足断言所描述的内容,否则会报错,但是只有第一个报告错误(用SoftAssertions来获取所有错误信息). 例如:

```
@Test
public void test_hasOnlyOneElementSatisfying(){
    List<Jedi> jedis = Lists.newArrayList(new Jedi("Yoda", "red"
));

    //这些可以通过
    assertThat(jedis).hasOnlyOneElementSatisfying(yoda -> assert
That(yoda.getName()).startsWith("Y"));
    assertThat(jedis).hasOnlyOneElementSatisfying(yoda -> {
        assertThat(yoda.getName()).isEqualTo("Yoda");
        assertThat(yoda.getLightSaberColor()).isEqualTo("red");
    });
```

```
    //这些会失败
    assertThat(jedis).hasOnlyOneElementSatisfying(yoda -> assert
That(yoda.getName()).startsWith("Vad"));
    assertThat(jedis).hasOnlyOneElementSatisfying(yoda -> {
        assertThat(yoda.getName()).isEqualTo("Yoda");
        assertThat(yoda.getLightSaberColor()).isEqualTo("purple"
);
    });

    //失败,但是只会报告第一条错误信息
    assertThat(jedis).hasOnlyOneElementSatisfying(yoda -> {
        assertThat(yoda.getName()).isEqualTo("Luke");
        assertThat(yoda.getLightSaberColor()).isEqualTo("green")
;
    });

    //  使用SoftAssertions可以收集显示所有的错误信息
    assertThat(jedis).hasOnlyOneElementSatisfying(yoda -> {
        SoftAssertions softly = new SoftAssertions();
        softly.assertThat(yoda.getName()).isEqualTo("Luke");
        softly.assertThat(yoda.getLightSaberColor()).isEqualTo("
green");
        softly.assertAll();
    });

    jedis.add(new Jedi("Luke", "green"));
    //失败:虽然断言是满足的,但是它有两个jedis
    assertThat(jedis).hasOnlyOneElementSatisfying(yoda -> assert
That(yoda.getName()).startsWith("Yo"));
}
```

可以直接用**lambda**表示式提取几个可迭代的元素例如

```
@Test
public void test_fellowshipOfTheRing() {
    List<Jedi> fellowshipOfTheRing = Lists.newArrayList(new Jedi(
"Luke", "green"));
    assertThat(fellowshipOfTheRing).flatExtracting(Jedi::getName
,
            Jedi::getLightSaberColor)
            .contains("Luke", "green",
                    "Luke", "green1",
                    "Luke", "green2");
}
```

上面例子断言集合fellowshipOfTheRing包含了三个元素,但是实际上只有一个元素是存在的所以报错了

Assertj断言生成器

通过一个简单的命令行工具，第三方的插件Maven插件或gradle,使用断言生成器创建特定于您自己的类的断言。

比如说有一个Player类，有name,team和teamMaster属性:

```
public class Player {

  private String name;
  private String team;
  private List<String> teamMates;

  // 构造函数和setter方法省略
  public String getName() {
    return name;
  }
  public String getTeam() {
    return team;
  }
  public List<String> getTeamMates() {
    return teamMates;
  }

  @Override
  public String toString() {
    return "Player[name="+name+"]";
  }
}
```

创建一个有hasName(),hasTeam()的PlayerAssert断言类之后可以这样用:

```
assertThat(mj).hasName("Michael Jordan")
              .hasTeam("Chicago Bulls")
              .hasTeamMates("Scottie Pippen", "Tony Kukoc");
```

一个失败的断言错误消息会反映出所检查的属性,例如:如果hasName失败,你会得到错误的信息:

```
Expecting name of:
  <Player[name=Air Jordan]>
to be:
  <Michael Jordan>
but was:
  <Air Jordan>
```

创建断言

生成器可以为每个属性和公共的字段创建断言,创建出来的断言是特定的属性/断言类型,下面列表显示了断言给定类型的创建:

- boolean:
  - isX()
  - isNotX()
- int, long, short, byte, char :
  - hasX(value) - 用==做value的比较
- float and double:
  - hasX(value) - 用==做value的比较
  - hasXCloseTo(value, delta), diff < delta check
- Iterable and T[]:
  - hasX(T... values)
  - hasOnlyX(T... values)
  - doesNotHaveX(T... values)
  - hasNoX() 元素的比较是用equalse()方法
- Object:
  - hasX(value)基于equalse比较

生成器内部会检查属性和公共字段,大多数具体的字段类型都是在生成断言时选择的,对象断言是默认的选择. double和Double会生成同样类型的断言,唯一的区别是他们的比较用的是:==或者equals 你也可以更改模板来修改断言的产生,模板位于模板目录.

例子

```
public class Player {
  // 公共的属性是没有get,set方法的
  // 私有属性的get和set方法省略

  public String name; // 生成对象断言
  private int age; // 生成整数断言
  private double height; // 生成小数断言
  private boolean retired; // 生成boolean断言
  private List<String> teamMates;  // 生成Iterable断言
}
```

创建断言

```
public class PlayerAssert extends AbstractAssert<PlayerAssert, P
layer> {


  public PlayerAssert hasAge(int age) { ... }

  public PlayerAssert hasHeight(double height) { ... }

  public PlayerAssert hasHeightCloseTo(double height, double off
set) { ... }

  public PlayerAssert hasTeamMates(String... teamMates) { ... }

  public PlayerAssert hasOnlyTeamMates(String... teamMates) { ..
. }

  public PlayerAssert doesNotHaveTeamMates(String... teamMates)
{ ... }

  public PlayerAssert hasNoTeamMates() { ... }

  public PlayerAssert isRetired() { ... }

  public PlayerAssert isNotRetired() { ... }

  public PlayerAssert hasName(String name) { ... }

  public PlayerAssert(Player actual) {
    super(actual, PlayerAssert.class);
  }

  public static PlayerAssert assertThat(Player actual) {
    return new PlayerAssert(actual);
  }
}
```

生成切入点(entry point)类 生成器还可以创建入口类,比如Assertions,BddAssertiions和SoftAssertions/JUnitSoftAssertions简化访问每个生成的*Assert类 比如说,例如,该生成器已被应用在Player和Game类,它会创建PlayerAssert和GameAssert类，并且有一个Assertions看起来像:

```
public class Assertions {

  /**
   * 创建一个新的实例 <code>{@link GameAssert}</code>.

   */
  public static GameAssert assertThat(Game actual) {
    return new GameAssert(actual);
  }

  /**
   * 创建一个新的实例 <code>{@link PlayerAssert}</code>.

   */
  public static PlayerAssert assertThat(Player actual) {
    return new PlayerAssert(actual);
  }

  /**
   * 创建一个新的实例<code>{@link Assertions}</code>.
   */
  protected Assertions() {
    // empty
  }
}
```

现在你只需要导入org.player.Assertions.assertThat就可以做断言了: import static org.player.Assertions.assertThat;

```
assertThat(mj).hasName("Michael Jordan")
              .hasTeam("Chicago Bulls")
              .hasTeamMates("Scottie Pippen", "Tony Kukoc");
```

断言模板

任何一种模板断言类或者切入点模板,都可以用断言模板来生成.模板位于这里 下面是一个用模板生成Player的hasName(String name)方法的断言

```
/**
 * 校验实际${class_to_assert}类的 ${property}属性是不是等于给定的值.
 * @param ${property_safe}  ${property}方法会比较 ${class_to_assert}类的 ${property}属性的值.
 * @return 断言类.
 * @throws AssertionError - 如果实际的 ${class_to_assert}类的${property}属性不等于给定的参数${throws_javadoc}就抛出这个异常
 */
public ${self_type} has${Property}(${propertyType} ${property_safe}) ${throws}{
  //检查实际的Player,因为制作断言不能为null
  isNotNull();

  //覆盖默认的错误消息,更加个性化
  String assertjErrorMessage = "\nExpecting ${property} of:\n  <%s>\nto be:\n  <%s>\nbut was:\n  <%s>";

  // 安全检查
  ${propertyType} actual${Property} = actual.get${Property}();
  if (!Objects.areEqual(actual${Property}, ${property_safe})) {
    failWithMessage(assertjErrorMessage, actual, ${property_safe}, actual${Property});
  }

  // 返回当前断言的引用
  return ${myself};
}
```

使用下面这些属性生成hasName(String name)断言:

属性	用法	例子
${property}	生成的断言的属性/字段	name
${propertyType}	生成断言的属性/字段的类型	String
${Property}	生成的断言的属性/字段（大写开头）	Name
${property_safe}	如果${property}等于java的保留关键字,就会使用${Propertiy}	name
${throws}	调用属性的getter抛出的异常(如果有)	-
${throws_javadoc}	调用属性的getter抛出的异常的javadoc(如果有)	-
${self_type}	声明类名	PlayerAssertion
${class_to_assert}	生成的断言类	Player
${myself}	代表用于执行断言断言类的实例	this

最后生成的断言:

```
/**
 * 校验实际${class_to_assert}类的 ${property}属性是不是等于给定的值.
 * @param ${property_safe}  ${property}方法会比较 ${class_to_asser
t}类的 ${property}属性的值.
 * @return 断言类.
 * @throws AssertionError - 如果实际的 ${class_to_assert}类的${pro
perty}属性不等于给定的参数${throws_javadoc}就抛出这个异常
 */
public PlayerAssert hasName(String name) {
  //检查实际的Player,因为制作断言不能为null
  isNotNull();

   //覆盖默认的错误消息,更加个性化
  String assertjErrorMessage = "\nExpecting name of:\n  <%s>\nto
 be:\n  <%s>\nbut was:\n  <%s>";

  // null safe check
  String actualName = actual.getName();
  if (!Objects.areEqual(actualName, name)) {
    failWithMessage(assertjErrorMessage, actual, name, actualNam
e);
  }

     // 返回当前断言的引用
  return this;
}
```

模板列表

方法断言模板:

方法断言模板文件	用法
has_assertion_template.txt	对象断言
is_assertion_template.txt	boolean断言
is_wrapper_assertion_template.txt	Boolean断言
has_assertion_template_for_char.txt	char类型断言
has_assertion_template_for_character.txt	Character断言
has_assertion_template_for_whole_number.txt	int, long, short 和byte类型断言
has_assertion_template_for_whole_number_wrapper.txt	Integer, Long, Short 和 Byte类型断言
has_assertion_template_for_real_number.txt	float 和 double类型断言
has_elements_assertion_template_for_array.txt丨数组类型断言	hasTeamMastes("mj"
has_elements_assertion_template_for_iterable.txt	Iterable断言

类断言模板

类方法模板文件	用法
custom_assertion_class_template.txt丨断言类框架	
custom_abstract_assertion_class_template.txt	base assertion class in in hierarchical assertion
custom_hierarchical_assertion_class_template.txt	concrete assertion class in in hierarchical assertion

切入点(netry point)模板

切入点(netry point)模板文件	使用
standard_assertions_entry_point_class_template.txt	断言类框架
standard_assertion_entry_point_method_template.txt	像Assertions类中的assertThat()放的切入点模板
bdd_assertions_entry_point_class_template.txt	BddAssertions类框架
bdd_assertion_entry_point_method_template.txt	template for then method in BddAssertions entry point class
soft_assertions_entry_point_class_template.txt	SoftAssertions类框架
soft_assertion_entry_point_method_template.txt	template for "soft" assertThat method in SoftAssertions entry point class
junit_soft_assertions_entry_point_class_template.txt	JUnitSoftAssertions类框架

布尔型/predicate型断言的属性

${predicate}	predicate断言方法名，例如isRookie()
${neg_predicate}	negative predicate断言方法名.例如:isNotRookie()
${predicate_for_error_message_part1}	用于构建predicate错误信息的第一部分
${predicate_for_error_message_part2}	用于构建predicate错误信息的第二部分
${negative_predicate_for_error_message_part1}	用于构建negative predicate错误消息的第一部分
${negative_predicate_for_error_message_part2}	用于构建negative predicate错误消息的第二部分
${predicate_for_javadoc}	predicate断言方法的javadoc
${negative_predicate_for_javadoc}	negative predicate断言方法的javadoc

数组断言的属性: ${elementType}:数组中元素的类型.比如String[]的属性是String型

类断言的属性

${custom_assertion_class}	断言类类名.如：PlayerAssert
${super_assertion_class}	used for the super class in hierarchical assertions.
${imports}	导入当前生成的断言类
${package}	断言类的包名

用condition拓展断言

AssertJ可以通过拓展condition提供断言.创建condition你只需要实现org.assertj.assertions.core.Condition并且重写matches方法. 一旦你的condition创建了,那么你可以使用它的is(myCOndition)或者has(myCondition).

下面这些可以结合使用:

- not(Condition) : 给定的condition必须都不满足
- allOf(Condition...) : 给定的condition必须都满足
- anyOf(Condition...) : 给定的condition必须至少有一个满足

你也可以使用一下方法验证集合中的元素是否满足condition:

- are(condition) / have(condition) : 所有的元素都必须满足给定的condition
- areAtLeast(n, condition) / haveAtLeast(n, condition) : 至少有n个元素必须满足给定条件
- areAtMost(n, condition) / haveAtMost(n, condition) : 满足条件的元素不超过n个
- areExactly(n, condition) / haveExactly(n, condition) ：必须是n个元素满足给定条件

创建condition

```
private static final LinkedHashSet<String> JEDIS = newLinkedHash
Set("Luke", "Yoda", "Obiwan");
// Java 8的实现方式 :)
Condition<String> jediPower = new Condition<>(JEDIS::contains, "
jedi power");

// Java 7 的实现方式:(
Condition<String> jedi = new Condition<String>("jedi") {
    @Override
    public boolean matches(String value) {
        return JEDIS.contains(value);
    }
};
```

使用is/isNot的例子

```
@Test
public void test_is_and_isNot() {
    assertThat("Yoda").is(jedi);
    assertThat("Vader").isNot(jedi);
}
```

使用**has/doesNotHave**的例子

```
@Test
public void test_has_and_doesNotHave(){
    assertThat("Yoda").has(jediPower);
    assertThat("Solo").doesNotHave(jediPower);
}
```

验证集合中元素的例子

```
@Test
public void test_colletion_elements(){
    assertThat(newLinkedHashSet("Luke", "Yoda")).are(jedi);
    assertThat(newLinkedHashSet("Leia", "Solo")).areNot(jedi);

    assertThat(newLinkedHashSet("Luke", "Yoda")).have(jediPower)
;
    assertThat(newLinkedHashSet("Leia", "Solo")).doNotHave(jediP
ower);

    assertThat(newLinkedHashSet("Luke", "Yoda", "Leia")).areAtLe
ast(2, jedi);
    assertThat(newLinkedHashSet("Luke", "Yoda", "Leia")).haveAtL
east(2, jediPower);

    assertThat(newLinkedHashSet("Luke", "Yoda", "Leia")).areAtMo
st(2, jedi);
    assertThat(newLinkedHashSet("Luke", "Yoda", "Leia")).haveAtM
ost(2, jediPower);

    assertThat(newLinkedHashSet("Luke", "Yoda", "Leia")).areExac
tly(2, jedi);
    assertThat(newLinkedHashSet("Luke", "Yoda", "Leia")).haveExa
ctly(2, jediPower);
}
```

结合使用conditions

allOf(Condition...)表示逻辑与,anyOf(Condition...)表示逻辑或

```
private final Condition<String> sith = new Condition<String>("si
th") {
    private final Set<String> siths = newLinkedHashSet("Sidious"
, "Vader", "Plagueis");

    @Override
    public boolean matches(String value) {
        return siths.contains(value);
    }
};

private final Condition<String> sithPower = new Condition<String
>("sith power") {
    private final Set<String> siths = newLinkedHashSet("Sidious"
, "Vader", "Plagueis");

    @Override
    public boolean matches(String value) {
        return siths.contains(value);
    }
};
@Test
public void test_combining(){
    assertThat("Vader").is(anyOf(jedi, sith));
    assertThat("Solo").is(allOf(not(jedi), not(sith)));
}
```

Assert断言生成器的maven插件

快速开始

您需要配置项目的pom.xml，以使用Maven断言发生器插件。

1. 添加assertj的核心依赖

```
<dependency>
 <groupId>org.assertj</groupId>
 <artifactId>assertj-core</artifactId>
 <!-- use version 2.0.0 or higher -->
 <version>2.0.0</version>
 <scope>test</scope>
</dependency>
```

2. 在你的pom.xml的build/plugins部分配置插件

```
<plugin>
 <groupId>org.assertj</groupId>
 <artifactId>assertj-assertions-generator-maven-plugin</arti
factId>
 <version>2.0.0</version>
 <configuration>
     <packages>
         <param>your.first.package</param>
         <param>your.second.package</param>
     </packages>
     <classes>
         <param>your.third.package.YourClass</param>
     </classes>
 </configuration>
</plugin>
```

1. 启动maven插件 执行命令

```
mvn assertj:generate-assertions
```

默认情况下在target/generated-test-sources/assertj-assertions/目录生成断言

这是一个assertj-examples中的例子

```
====================================
AssertJ assertions generation report
====================================

--- Generator input parameters ---

The following templates will replace the ones provided by Assert
J when generating AssertJ assertions :
- Using custom template for 'object assertions' loaded from ./te
mplates/my_has_assertion_template.txt
- Using custom template for 'hierarchical concrete class asserti
ons' loaded from ./templates/my_assertion_class_template.txt

Generating AssertJ assertions for classes in following packages
and subpackages:
- org.assertj.examples.data

Input classes excluded from assertions generation:
- org.assertj.examples.data.MyAssert

--- Generator results ---

Directory where custom assertions files have been generated :
- /assertj/assertj-examples/assertions-examples/target/generated
-test-sources/assertj-assertions

# full path truncated for to improve clarity in the website.
Custom assertions files generated :
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/AlignmentAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/BasketBallPlayerAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/BookAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/BookTitleAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/EmployeeAssert.java
```

```
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/EmployeeTitleAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/MagicalAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/MansionAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/NameAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/PersonAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/RaceAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/RingAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/TeamAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/TolkienCharacterAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/movie/MovieAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/neo4j/DragonBallGraphAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/service/GameServiceAssert.java
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/service/TeamManagerAssert.java

No custom assertions files generated for the following input cla
sses as they were not found:
- com.fake.UnknownClass1
- com.fake.UnknownClass2

Assertions entry point class has been generated in file:
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/Assertions.java

Soft Assertions entry point class has been generated in file:
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/SoftAssertions.java

JUnitSoftAssertions entry point class has been generated in file
:
```

```
- .../generated-test-sources/assertj-assertions/org/assertj/JUni
tSoftAssertions.java

BDD Assertions entry point class has been generated in file:
- .../generated-test-sources/assertj-assertions/org/assertj/exam
ples/data/BddAssertions.java
```

插件配置

```
<plugin>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-assertions-generator-maven-plugin</artifac
tId>
  <version>2.0.0</version>

  <!-- 每次 build生成断言 -->
  <executions>
    <execution>
      <goals>
        <goal>generate-assertions</goal>
      </goals>
    </execution>
  </executions>

  <configuration>

    <!-- 列出要生成断言其类的包 -->
    <packages>
      <param>org.assertj.examples.rpg</param>
      <param>org.assertj.examples.data</param>
      <param>com.google.common.net</param>
    </packages>

    <!-- 列出要生成断言类 -->
    <classes>
      <param>java.nio.file.Path</param>
      <param>com.fake.UnknownClass</param>
    </classes>

    <!-- wether generated assertions classes can be inherited wi
th consistent assertion chaining -->
```

```
    <hierarchical>true</hierarchical>

    <!-- 在哪生成切入点断言类 -->
    <entryPointClassPackage>org.assertj</entryPointClassPackage>

    <!-- 用正则表带是来现在段眼类的生成 -->
    <includes>
      <param>org\.assertj\.examples\.rpg.*</param>
    </includes>

    <!-- 从代正则表达式类排除 -->
    <excludes>
      <param>.*google.*HostSpecifier</param>
      <param>.*google.*Headers</param>
      <param>.*google.*MediaType</param>
      <param>.*google.*Escaper.*</param>
      <param>.*Examples.*</param>
    </excludes>

    <!-- 生成断言类的目录 -->
    <targetDir>src/test/generated-assertions</targetDir>

    <!-- 选择哪个断言中的切入点类来生成 -->
    <generateAssertions>true</generateAssertions>
    <generateBddAssertions>true</generateBddAssertions>
    <generateSoftAssertions>true</generateSoftAssertions>
    <generateJUnitSoftAssertions>true</generateJUnitSoftAssertio
ns>

  </configuration>
</plugin>
```

更多内容请查看原文

Assertj的Guava断言

Guava AssertJ断言是Guava提供的Guava类型断言,提供了例如:Multimap,Table,Optional,Range或者ByteSource的断言. 如果你认为欠缺了一些断言,你可以提出问题,也可以做出一些贡献 Guava AssertJ断言的代码仓库

快速入门

1. 添加assertj-guava的maven依赖

```
<dependency>
<groupId>org.assertj</groupId>
<artifactId>assertj-guava</artifactId>
<!-- Use 2.x version if you rely on Java 7 / AssertJ Core 2.
x -->
<version>3.0.0</version>
<scope>test</scope>
</dependency>
```

如果你有其他的依赖请查看这里添加你所需要的工具

2. 静态导入import org.assertj.guava.api.Assertions.assertThat

```
 import static org.assertj.guava.api.Assertions.assertThat;
 import static org.assertj.guava.api.Assertions.entry;

 // Multimap assertions
 Multimap<String, String> actual = ArrayListMultimap.create(
);
 actual.putAll("Lakers", newArrayList("Kobe Bryant", "Magic
Johnson", "Kareem Abdul Jabbar"));
 actual.putAll("Spurs", newArrayList("Tony Parker", "Tim Dun
can", "Manu Ginobili"));

 assertThat(actual).containsKeys("Lakers", "Spurs");
 assertThat(actual).contains(entry("Lakers", "Kobe Bryant"),
 entry("Spurs", "Tim Duncan"));

 // Range assertions
```

```
 Range<Integer> range = Range.closed(10, 12);

 assertThat(range).isNotEmpty()
                  .contains(10, 11, 12)
                  .hasClosedLowerBound()
                  .hasLowerEndpointEqualTo(10)
                  .hasUpperEndpointEqualTo(12);

 // Table assertions
 Table<Integer, String, String> bestMovies = HashBasedTable.
create();

 bestMovies.put(1970, "Palme d'Or", "M.A.S.H");
 bestMovies.put(1994, "Palme d'Or", "Pulp Fiction");
 bestMovies.put(2008, "Palme d'Or", "Entre les murs");
 bestMovies.put(2000, "Best picture Oscar", "American Beauty"
);
 bestMovies.put(2011, "Goldener Bär", "A Separation");

 assertThat(bestMovies).hasRowCount(5).hasColumnCount(3).has
Size(5)
                       .containsValues("American Beauty", "A
 Separation", "Pulp Fiction")
                       .containsCell(1994, "Palme d'Or", "Pu
lp Fiction")
                       .containsColumns("Palme d'Or", "Best
picture Oscar", "Goldener Bär")
                       .containsRows(1970, 1994, 2000, 2008,
2011);

 // Optional assertions
 Optional<String> optional = Optional.of("Test");
 assertThat(optional).isPresent().contains("Test");
```

同时使用**guava**和**assertj**的核心功能做断言

```
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.guava.api.Assertions.assertThat;
...
// assertThat comes from org.assertj.guava.api.Assertions.assert
That static import
Multimap<String, String> actual = ArrayListMultimap.create();
actual.putAll("Lakers", newArrayList("Kobe Bryant", "Magic Johns
on", "Kareem Abdul Jabbar"));
actual.putAll("Spurs", newArrayList("Tony Parker", "Tim Duncan",
"Manu Ginobili"));

assertThat(actual).hasSize(6);
assertThat(actual).containsKeys("Lakers", "Spurs");

// assertThat comes from org.assertj.core.api.Assertions.assertT
hat static import
assertThat("hello world").startsWith("hello");
```

Multimap断言:hasSameEntryesAs(Multimap other)

允许两个Multimap的内容，因为是不同的类型，如SetMultimap和ListMultimap的,所以谁也不会直接调用equlase比较

```
Multimap<String, String> actual = ArrayListMultimap.create();
listMultimap.putAll("Spurs", newArrayList("Tony Parker", "Tim Du
ncan", "Manu Ginobili"));
listMultimap.putAll("Bulls", newArrayList("Michael Jordan", "Sco
ttie Pippen", "Derrick Rose"));

Multimap<String, String> setMultimap = TreeMultimap.create();
setMultimap.putAll("Spurs", newHashSet("Tony Parker", "Tim Dunca
n", "Manu Ginobili"));
setMultimap.putAll("Bulls", newHashSet("Michael Jordan", "Scotti
e Pippen", "Derrick Rose"));

// listMultimap和setMultimap有相等的内容
assertThat(listMultimap).hasSameEntriesAs(setMultimap);

// 即便它们有相同的内容也会失败
assertThat(listMultimap).isEqualTo(setMultimap);
```

Multimap断言:containsAllEntriesOf(Multimap other).

验证实际Multimap中是否包含给定other参数的所有内容

```
Multimap<String, String> actual = ArrayListMultimap.create();
actual.putAll("Spurs", newArrayList("Tony Parker", "Tim Duncan",
"Manu Ginobili"));
actual.putAll("Bulls", newArrayList("Michael Jordan", "Scottie P
ippen", "Derrick Rose"));

Multimap<String, String> other = TreeMultimap.create();
other.putAll("Spurs", newHashSet("Tony Parker", "Tim Duncan"));
other.putAll("Bulls", newHashSet("Michael Jordan", "Scottie Pipp
en"));

//other是actual的子集,断言会通过
assertThat(actual).containsAllEntriesOf(other);

//不通过
assertThat(other).containsAllEntriesOf(actual);
```

断言Optional的extractingValue()

用 extractingValue()可以对Optional的内容断言

```
Optional<Number> optional = Optional.of(12L);
assertThat(optional).extractingValue()
                    .isInstanceOf(Long.class)
                    .isEqualTo(12L);

Optional<String> optional = Optional.of("Bill");
assertThat(optional).extractingCharSequence()
                    .startsWith("Bi");
```

Asserj Joda-Time

Joda-Time提供了joda-time类型断言,比如DateTime和LocalDateTime Assertj Joda-Time的代码仓库

Joda-Time断言的快速开始

1. 添加Joda-Time需要的依赖

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-joda-time</artifactId>
  <!-- use 1.1.0 for Java 7 projects -->
  <version>2.0.0</version>
  <scope>test</scope>
</dependency>
```

如果你要添加其他工具的依赖你可以查看这里

2. 静态导入org.assertj.jodatime.api.Assertions.assertThat

```
 import static org.assertj.jodatime.api.Assertions.assertTha
t;
 ...
 assertThat(dateTime).isBefore(firstDateTime);
 assertThat(dateTime).isAfterOrEqualTo(secondDateTime);

 // 你可以在比较中使用字符串,而不需要转换
 assertThat(new DateTime("2000-01-01")).isEqualTo("2000-01-0
1");

 // 比较DateTime是否忽略秒和毫秒
 dateTime1 = new DateTime(2000, 1, 1, 23, 50, 0, 0, UTC);
 dateTime2 = new DateTime(2000, 1, 1, 23, 50, 10, 456, UTC);
 // assertion succeeds
 assertThat(dateTime1).isEqualToIgnoringSeconds(dateTime2);
```

对于日期时间断言，比较在日期时间的DateTimeZone测试执行，结果如下断言会通过:

```
 DateTime utcTime = new DateTime(2013, 6, 10, 0, 0, DateTime
Zone.UTC);
 DateTime cestTime = new DateTime(2013, 6, 10, 2, 0, DateTim
eZone.forID("Europe/Berlin"));

 assertThat(utcTime).as("in UTC time").isEqualTo(cestTime);
```

技巧

使用日期字符串表示为了更容易使用，可以与他们的字符串表示指定DateTime或LocalDateTime避免手工字符串转换，如下面的例子:

```
//你不需要写这么复杂
assertThat(dateTime).isBefore(new DateTime("2004-12-13T21:39:45.
618-08:00"));
// ... 值需要简单这样写
assertThat(dateTime).isBefore("2004-12-13T21:39:45.618-08:00");
```

assert的断言和joda-time断言一起用

```
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.jodatime.api.Assertions.assertThat;
...
assertThat(new DateTime("2000-01-01")).isAfter(new DateTime("199
9-12-31"));
assertThat("hello world").startsWith("hello");
```

AssertJ-DB

介绍

AssertJ-DB提供了对数据库中数据的断言。它需要Java 7或更高版本，可与JUnit的或TestNG的使用

- 代码仓库

数据库断言的快速开始

假设数据库包中有下面这个表: MEMBERS

ID	NAME	FIRSTNAME	SURNAME	BIRTHDATE	SIZE
1	'Hewson'	'Paul David'	'Bono'	05-10-60	1.75
2	'Evans'	'David Howell'	'The Edge'	08-08-61	1.77
3	'Clayton'	'Adam'		03-13-60	1.78
4	'Mullen'	'Larry'		10-31-61	1.70

1. 添加assertj-db的maven依赖

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-db</artifactId>
  <version>1.1.1</version>
  <scope>test</scope>
</dependency>
```

   如果你需要依赖其他的工具,你可以查看这里

2. 静态导入org.assertj.db.api.Assertions.assertThat

```
import static org.assertj.db.api.Assertions.assertThat;

import org.assertj.db.type.DateValue;
import org.assertj.db.type.Table;

Table table = new Table(dataSource, "members");

//校验name字段的值
assertThat(table).column("name")
                 .value().isEqualTo("Hewson")
                 .value().isEqualTo("Evans")
                 .value().isEqualTo("Clayton")
                 .value().isEqualTo("Mullen");

// 校验下标为1的行的值
assertThat(table).row(1)
                 .value().isEqualTo(2)
                 .value().isEqualTo("Evans")
                 .value().isEqualTo("David Howell")
                 .value().isEqualTo("The Edge")
                 .value().isEqualTo(DateValue.of(1961, 8, 8
))
                 .value().isEqualTo(1.77);
```

AssertJ-DB概念

假设有下面两张表: MEMBERS:

ID	NAME	FIRSTNAME	SURNAME	BIRTHDATE	SIZE
1	'Hewson'	'Paul David'	'Bono'	05-10-60	1.75
2	'Evans'	'David Howell'	'The Edge'	08-08-61	1.77
3	'Clayton'	'Adam'		03-13-60	1.78
4	'Mullen'	'Larry'		10-31-61	1.70

ALBUMS

ID	RELEASE	TITLE	NUMBEROFSONGS	DURATION	LIVI
1	10-20-80	'Boy'	12	42:17	
2	10-12-81	'October'	11	41:08	
3	02-28-83	'War'	10	42:07	
4	11-07-83	'Under a Blood Red Sky'	8	33:25	true
5	10-01-84	'The Unforgettable Fire'	10	42:42	
6	06-10-85	'Wide Awake in America'	4	20:30	true
7	03-09-87	'The Joshua Tree'	11	50:11	
8	10-10-88	'Rattle and Hum'	17	72:27	
9	11-18-91	'Achtung Baby'	12	55:23	
10	07-06-93	'Zooropa'	10	51:15	
11	03-03-97	'Pop'	12	60:08	
12	10-30-00	'All That You Can't Leave Behind'	11	49:23	
13	11-22-04	'How to Dismantle an Atomic Bomb'	11	49:08	
14	03-02-09	'No Line on the Horizon'	11	53:44	
15	09-09-14	'Songs of Innocence'	11	48:11	

链接数据库

使用数据库的断言，有必要进行连接。无论是使用DataSource或Source

DataSource

DataSource是经典的java连接数据库方式

Source

Source是你不想访问数据库也不想使用DataSource的一种方式,它允许通过构造方法传递必要的链接信息.下面这个例子是使用Source链接到H2数据库。

```
Source source = new Source("jdbc:h2:mem:test", "sa", "");
```

DataSource与LetterCase

DataSourceWithLetterCase是一个DataSource的实例,它允许指定tables, columns和primary keys是哪个LetterCase

```
DataSource ds = new DataSourceWithLetterCase(dataSource, tableLe
tterCase, columnLetterCase,pkLetterCase);
```

想要了解更多,请查看这里

数据库元素

Table

Table表示的是数据中的表 一个Table需要一种方法来链接数据库,还需要指定一个表明

```
// Get a DataSource
DataSource dataSource = ...
// 声明一个dataSource下的名为members的表
Table table1 = new Table(dateSource, "members");
//  声明一个Source下的名为members的表
Table table2 = new Table(source, "members");
```

以上代码中的table1和table2都表示的是: MEMBERS:

ID	NAME	FIRSTNAME	SURNAME	BIRTHDATE	SIZE
1	'Hewson'	'Paul David'	'Bono'	05-10-60	1.75
2	'Evans'	'David Howell'	'The Edge'	08-08-61	1.77
3	'Clayton'	'Adam'		03-13-60	1.78
4	'Mullen'	'Larry'		10-31-61	1.70

你也可以在构造方法中指定包含或者排除的字段

```
//获取包含id和name字段的members表
Table table3 = new Table(source, "members", new String[] { "id",
"name" }, null);
//获取不包含birthdate字段的members表
Table table4 = new Table(source, "members", null, new String[] {
"birthdate" });
// 获取只包含name字段的members表 (如果id又包含又排除,那么最终会被排除)
Table table5 = new Table(source, "members", new String[] { "id",
"name" }, new String[] { "id" });
```

"table3"表示:

ID	NAME
1	'Hewson'
2	'Evans'
3	'Clayton'
4	'Mullen'

table4表示:

ID	NAME	FIRSTNAME	SURNAME	SIZE
1	'Hewson'	'Paul David'	'Bono'	1.75
2	'Evans'	'David Howell'	'The Edge'	1.77
3	'Clayton'	'Adam'		1.78
4	'Mullen'	'Larry'		1.70

table5表示:

NAME
'Hewson'
'Evans'
'Clayton'
'Mullen'

Request

Request表示数据库中的SQL语句

```
DataSource dataSource = ...
Request request1 = new Request(source,"select name, firstname fr
om members where id = 2 or id = 3");

Request request2 = new Request(dataSource,"select name, firstnam
e from members where id = 2 or id = 3");
```

以上代码得到以下结果：

NAME	FIRSTNAME	SURNAME
'Evans'	'David Howell'	'The Edge'
'Clayton'	'Adam'	

Request可以有一个或多个参数

```
Request request3 = new Request(dataSource,
                               "select name, firstname from memb
ers " +
                               "where name like ?;",
                               "%e%");

Request request4 = new Request(dataSource,
                               "select name, firstname from memb
ers " +
                               "where name like ? and firstname
like ?;",
                               "%e%",
                               "%Paul%");
```

Changes

Changes是数据库在起点(start point)和终点(end point)之间的状态:

假设在起点到终点之间有这些SQL语句:

```
DELETE FROM ALBUMS WHERE ID = 15;
INSERT INTO MEMBERS(ID, NAME, FIRSTNAME) VALUES(5, 'McGuiness',
'Paul');
UPDATE MEMBERS SET SURNAME = 'Bono Vox' WHERE ID = 1;
UPDATE ALBUMS SET NAME = 'Rattle & Hum', LIVE = true WHERE ID = 8
;
```

```
// Get a DataSource
DataSource dataSource = ...
// 这些change可以在dataSource也可以在source
Changes changes1 = new Changes(dataSource);
Changes changes2 = new Changes(source);
// change可以发生在table也可以发生在request
Changes changes3 = new Changes(table4);
Changes changes4 = new Changes(request3);
Changes changes5 = new Changes(request4);
//name是表中主键字段的值
// request中需要设置主键
Changes changes6 = new Changes(request4).setPksName("name");
```

chang1和chang2相等,chang4和chang5也相等 (下面这段太难懂了所以就直接复制过来了,希望以后理解了再来处理) The changes are ordered :

First by the type of the change : creation, modification and after deletion After if it a change on a table by the name of the table To finish by the values of the primary key and if there are no primary key by the values of the row (for a modification) As indicated above, the primary key is used to order the changes. But more important, the primary key is used to determinate which rows at the same with modifications. In Representation of "changes4" or "changes5" the modification of first row of the table become a creation and deletion.

Representation of "changes1" or "changes2"

Creation	"MEMBERS" table	5 as PK	

	ID	NAME	FIRSTNAME	SURNAME	BIRTHDATE	SIZE
At start point						
At end point	5	'McGuiness'	'Paul'			

Modification	"ALBUMS" table	8 as PK	

	ID	RELEASE	TITLE	NUMBEROFSONGS	DURATION	LIVE
At start point	8	10-10-88	'Rattle and Hum'	17	72:27	
At end point	8	10-10-88	'Rattle & Hum'	17	72:27	true

Modification	"MEMBERS" table	1 as PK	

	ID	NAME	FIRSTNAME	SURNAME	BIRTHDATE	SIZE
At start point	1	'Hewson'	'Paul David'	'Bono'	05-10-60	1.75
At end point	1	'Hewson'	'Paul David'	'Bono Vox'	05-10-60	1.75

Deletion	"ALBUMS" table	15 as PK	

	ID	RELEASE	TITLE	NUMBEROFSONGS	DURATION	LIVE
At start point	15	09-09-14	'Songs of Innocence'	11	48:11	
At end point						

Representation of "changes3"

Creation	"MEMBERS" table	5 as PK	

	ID	NAME	FIRSTNAME	SURNAME	SIZE
At start point					
At end point	5	'McGuiness'	'Paul'		

Modification	"MEMBERS" table	1 as PK	

	ID	NAME	FIRSTNAME	SURNAME	SIZE
At start point	1	'Hewson'	'Paul David'	'Bono'	1.75
At end point	1	'Hewson'	'Paul David'	'Bono Vox'	1.75

Representation of "changes4" or "changes5"

Creation		No PK	

	NAME	FIRSTNAME	SURNAME
At start point			
At end point	'Hewson'	'Paul David'	'Bono Vox'

Creation		No PK	

	NAME	FIRSTNAME	SURNAME
At start point			
At end point	'McGuiness'	'Paul'	

Deletion		No PK	

	NAME	FIRSTNAME	SURNAME
At start point			
At end point	'Hewson'	'Paul David'	'Bono Vox'

Representation of "changes6"

Creation		'McGuiness' as PK		NAME	FIRSTNAME	SURNAME
			At start point			
			At end point	'McGuiness'	'Paul'	
Modification		'Hewson' as PK		NAME	FIRSTNAME	SURNAME
			At start point	'Hewson'	'Paul David'	'Bono'
			At end point	'Hewson'	'Paul David'	'Bono Vox'

Change

Change是Changes中的一个元素 下面changes3的图片中的红色部分就是一个Change：

Representation of "changes3"

Creation	"MEMBERS" table	5 as PK		ID	NAME	FIRSTNAME	SURNAME	SIZE
			At start point					
			At end point	5	'McGuiness'	'Paul'		
Modification	"MEMBERS" table	1 as PK		ID	NAME	FIRSTNAME	SURNAME	SIZE
			At start point	1	'Hewson'	'Paul David'	'Bono'	1.75
			At end point	1	'Hewson'	'Paul David'	'Bono Vox'	1.75

Row

Row可以表示一个Table中的行,也可以是一个Request或者一个Change中的行 下面table4中的红色部分就是一个Row:

Representation of "table4"

ID	NAME	FIRSTNAME	SURNAME	SIZE
1	'Hewson'	'Paul David'	'Bono'	1.75
2	'Evans'	'David Howell'	'The Edge'	1.77
3	'Clayton'	'Adam'		1.78
4	'Mullen'	'Larry'		1.70

下面request3中的红色部分表示一个Row

Representation of "request3"

NAME	FIRSTNAME	SURNAME
'Hewson'	'Paul David'	'Bono'
'Evans'	'David Howell'	'The Edge'
'Mullen'	'Larry'	

下面changes3中的红色部分表示一个Row

Representation of "changes3"

Creation	"MEMBERS" table	5 as PK
Modification	"MEMBERS" table	1 as PK

	ID	NAME	FIRSTNAME	SURNAME	SIZE
At start point					
At end point	5	'McGuiness'	'Paul'		

	ID	NAME	FIRSTNAME	SURNAME	SIZE
At start point	1	'Hewson'	'Paul David'	'Bono'	1.75
At end point	1	'Hewson'	'Paul David'	'Bono Vox'	1.75

Column

Column可以表示一个Table中的列,也可以是一个Request或者一个Change的列 下面table4中的红色部分就是一个Column

Representation of "table4"

ID	NAME	FIRSTNAME	SURNAME	SIZE
1	'Hewson'	'Paul David'	'Bono'	1.75
2	'Evans'	'David Howell'	'The Edge'	1.77
3	'Clayton'	'Adam'		1.78
4	'Mullen'	'Larry'		1.70

下面request3中的红色部分就是一个Column

Representation of "request3"

NAME	FIRSTNAME	SURNAME
'Hewson'	'Paul David'	'Bono'
'Evans'	'David Howell'	'The Edge'
'Mullen'	'Larry'	

下面changes3中的红色部分表示一个Column

Representation of "changes3"

Creation	"MEMBERS" table	5 as PK
Modification	"MEMBERS" table	1 as PK

	ID	NAME	FIRSTNAME	SURNAME	SIZE
At start point					
At end point	5	'McGuiness'	'Paul'		

	ID	NAME	FIRSTNAME	SURNAME	SIZE
At start point	1	'Hewson'	'Paul David'	'Bono'	1.75
At end point	1	'Hewson'	'Paul David'	'Bono Vox'	1.75

Value

Value可以在Row中也可以在Column中 table4中的第二row的第二个value和table4中的第二column的第三个value

Representation of "table4"

ID	NAME	FIRSTNAME	SURNAME	SIZE
1	'Hewson'	'Paul David'	'Bono'	1.75
2	'Evans'	'David Howell'	'The Edge'	1.77
3	'Clayton'	'Adam'		1.78
4	'Mullen'	'Larry'		1.70

request3中的第二row的第二个value和第二column的第二个value:

Representation of "request3"

NAME	FIRSTNAME	SURNAME
'Hewson'	'Paul David'	'Bono'
'Evans'	'David Howell'	'The Edge'
'Mullen'	'Larry'	

changes3第二个change的第四个row的value和第二个change的第第个column的value

Representation of "changes3"

Creation	"MEMBERS" table	5 as PK	
Modification	"MEMBERS" table	1 as PK	

	ID	NAME	FIRSTNAME	SURNAME	SIZE
At start point					
At end point	5	'McGuiness'	'Paul'		

	ID	NAME	FIRSTNAME	SURNAME	SIZE
At start point	1	'Hewson'	'Paul David'	'Bono'	1.75
At end point	1	'Hewson'	'Paul David'	'Bono Vox'	1.75

Type

1. DataType 之前提到的三个数据库的根元素,只有Table和Request是数据元素. 所有可能的数据类型被包含在DataType枚举中 这个类型可以是:
    - Table
    - Request
2. ChangeType change的类型可以是创建,修改或者删除。 所有的change类型都包含在ChangeType枚举中 change类型根数据库的操作类型对应:
    - CREATION是insert SQL请求
    - MODIFICATION是upadte　SQL请求
    - DELETION是delete　SQL请求
3. ValueType value的类型可以是日期型,布尔型或者文本型 所有可能的数据类型被包含在ValueType枚举中 value的类型取决与从数据库读取出来的对象的类型:
    - BYTES　是一个byte数组(byte[])
    - BOOLEAN 是java.lang.Boolean
    - Text 是java.lang.String
    - DATE 是java.sql.Date
    - TIME 是java.sql.time

- DATE_TIME　是java.sql.Timestamp
- UUID 是java.util.UUID
- NUMBER 是 java.lang.Byte,java.lang.Short,java.lang.Integer,java.lang.Long,java.lang.Float或者java.math.BigDecimal
- NOT_IDENTIFIED　是其他的类型(比如说null)

Navigation(导航)

navigation是可以提供不能的级别和指示到达子元素内，并且返回到根元素能力的断言。请看Navigation例子

Table或者Request做根断言

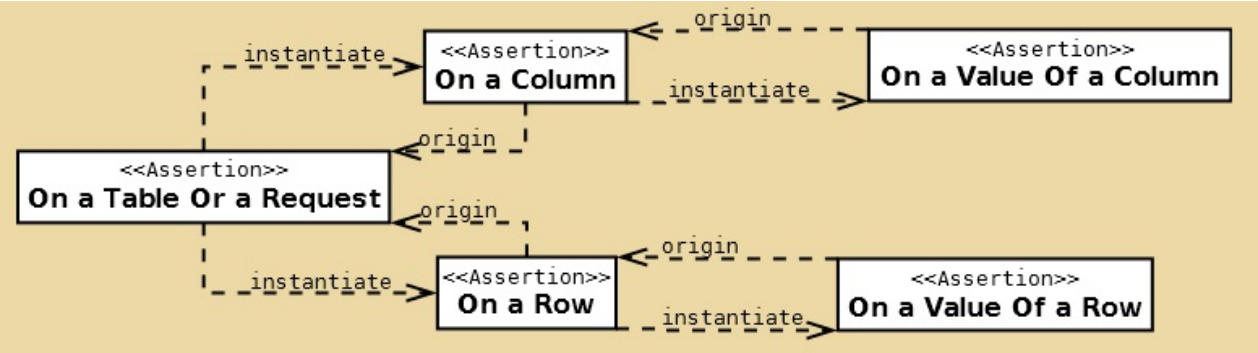
可以在Table上创建一个根断言

```
import static org.assertj.db.api.Assertions.assertThat;

assertThat(table)...
```

也可以在Request上创建:

```
import static org.assertj.db.api.Assertions.assertThat;

assertThat(request)...
```

从这些根断言，能够导航到子元素，并返回到根元素如下面的图片:

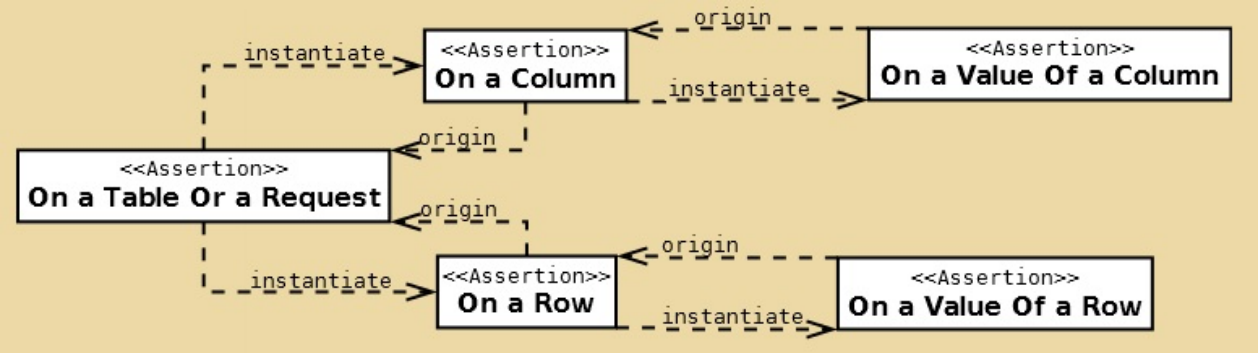


更多细节请查看这里

Changes做根断言

```
import static org.assertj.db.api.Assertions.assertThat;

assertThat(changes)...
```

从该根断言，能够导航到子元素，并返回到根元素如下面的图片。



更多细节请查看这里

DateValue, TimeValue 和 DateTimeValue

有4种静态方法来实例化这些值：

- of接收int类型参数

```
  DateValue dateValue = DateValue.of(2007, 12, 23);

  // 小时和分钟
  TimeValue timeValue1 = TimeValue.of(9, 1);
  // 时分秒
  TimeValue timeValue2 = TimeValue.of(9, 1, 6);
  // 时分秒纳秒
  TimeValue timeValue3 = TimeValue.of(9, 1, 6, 3);

  // 日期(小时是0,即午夜0时)
  DateTimeValue dateTimeValue1 = DateTimeValue.of(dateValue)
;
  //日期和时间
  DateTimeValue dateTimeValue2 = DateTimeValue.of(dateValue,
 timeValue1);
```

- from等效于从java.sql 包(java.sql.Date, java.sql.Time 和 java.sql.Timestamp)或者java.util.Calendar包接收参数

```
  Date date = Date.valueOf("2007-12-23");
  DateValue dateValue = DateValue.from(date);

  Time time = Time.valueOf("09:01:06");
  TimeValue timeValue = TimeValue.from(time);

  Timestamp timestamp = Timestamp.valueOf("2007-12-23 09:01:
06.000000003");
  DateTimeValue dateTimeValue = DateTimeValue.from(timestamp
);

  Calendar calendar = Calendar.getInstance();
  DateValue dateValueFromCal = DateValue.from(calendar);
  TimeValue timeValueFromCal = TimeValue.from(calendar);
  DateTimeValue dateTimeValueFromCal = DateTimeValue.from(ca
lendar);
```

- parse接收字符串(该方法可能会抛出一个ParseException异常)

```
  DateValue dateValue = DateValue.parse("2007-12-23");

  // 小时分钟
  TimeValue timeValue1 = TimeValue.parse("09:01");
  // 时分秒
  TimeValue timeValue2 = TimeValue.parse("09:01:06");
  // 时分秒纳秒
  TimeValue timeValue3 = TimeValue.parse("09:01:06.000000003"
);

  //日期(小时是0,即午夜0时)
  DateTimeValue dateTimeValue1 = DateTimeValue.parse("2007-1
2-23");
  //日期和时间
  DateTimeValue dateTimeValue2 = DateTimeValue.parse("2007-1
2-23T09:01");
  DateTimeValue dateTimeValue2 = DateTimeValue.parse("2007-1
2-23T09:01:06");
  DateTimeValue dateTimeValue2 = DateTimeValue.parse("2007-1
2-23T09:01:06.000000003");
```

- now创建当前时间的实例

```
  DateValue dateValue = DateValue.now();                    /
/ The current date
  TimeValue timeValue = TimeValue.now();                    /
/ The current time
  DateTimeValue dateTimeValue = DateTimeValue.now();        /
/ The current date/time
```

默认的描述

在assertj中，你可以实现Descriptable接口来增加一个描述,如果断言失败了那么这个描述会显示在错误信息中. 由于navigation在asserj-Db的存在,是的assertj-DB更加难以知道是哪个元素引发了错误,因此有一些默认的描述来帮助测试人员. 例如:

- "members table"一个在table中的断言
- "'select * from actor' request"一个在request中的断言
- "'select id, name, firstname, bi...' request"一个更多内容的request中的断言
- "Row at index 0 of members table"在一个table中的一个row的断言
- "Column at index 0 (column name : ID) of 'select * from members' request"一个在request中的一个column的断言
- "Value at index 0 of Column at index 0 (column name : ID) of 'select * from members' request"一个在request中的column中的断言
- "Value at index 0 (column name : ID) of Row at index 0 of 'select * from members' request"一个在request中的一个row中的一个value的断言
- "Value at index 0 (column name : ID) of Row at end point of Change at index 0 (on table : MEMBERS and with primary key : [4]) of Changes on tables of 'sa/jdbc:h2:mem:test' source" 一个在table中的change的end point的rou的value的断言

Ouput

有时候debug的时候你想要查看数据库的数据,用output可以做到这点,例如:

```
import static org.assertj.db.output.Outputs.output;

Table table = new Table(dataSource, "members");

//内容会输出到控制台
output(table).toConsole();
```

输出的内容如下:

```
[MEMBERS table]
|-----------|---------|-----------|-----------|--------------|-----------|-----------|-----------|
|           |         | *         |           |              |           |           |           |
|           | PRIMARY | ID        | NAME      | FIRSTNAME    | SURNAME   | BIRTHDATE | SIZE      |
|           | KEY     | (NUMBER)  | (TEXT)    | (TEXT)       | (TEXT)    | (DATE)    | (NUMBER)  |
|           |         | Index : 0 | Index : 1 | Index : 2    | Index : 3 | Index : 4 | Index : 5 |
|-----------|---------|-----------|-----------|--------------|-----------|-----------|-----------|
| Index : 0 | 1       | 1         | Hewson    | Paul David   | Bono      | 05-10-60  | 1.75      |
| Index : 1 | 2       | 2         | Evans     | David Howell | The Edge  | 08-08-61  | 1.77      |
| Index : 2 | 3       | 3         | Clayton   | Adam         |           | 03-13-60  | 1.78      |
| Index : 4 | 4       | 4         | Mullen    | Larry        |           | 10-31-61  | 1.70      |
|-----------|---------|-----------|-----------|--------------|-----------|-----------|-----------|
```

上面的例子是控制台中显示纯文本,你也可以改变它

输出类型

有两个已经实现的类型:

- PLAIN：文本内容，如上面例子所示(默认输出类型)
- HTML:HTML文档

```
output(table).withType(OutputType.HTML).....;
```

输出目标

有以下三种选择:

- 控制台输出(toConsole()方法)
- 输出到文件(toFile(String fileName)方法)
- 输出到流中(toStream(OutputStream outputStream)方法)

```
output(table).toConsole().withType(OutputType.HTML).toFile("
test.html");
```

更多内容请查看原文

Gitbook该文件修订时间： 2017-03-13 05:37:11

AssertJ-DB特性

看这边文章之前，建议先阅读概念 下面所有的例子都用的是概念文章中的两张表

Navigation

Table或者Request作为根断言

assertThat(...)方法表示一个table或一个request开始的断言 从table或者request开始基本都是以下面这种方式调用:

```
assertThat(tableOrRequest)...
```

如果是有区别的将会被指定

所以的navigation方法都是从origin 点开始工作的

to Row

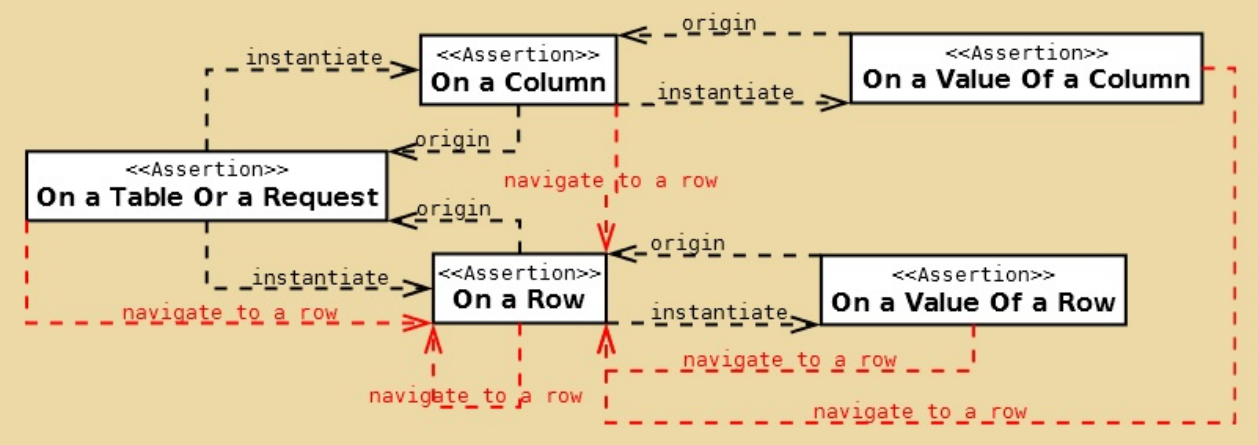
ToRow接口的详情请查看这里

```
@Test
public void testNavigation(){
    Source source = new Source("jdbc:h2:mem:user-db;MODE=Postgre
SQL;INIT=RUNSCRIPT FROM " +
            "'/home/oem/javaProject/other/java-test/assertj-core
/src/test/resources/create_table.sql'", "sa", "");
    Table table = new Table(source, "MEMBERS");

    output(table).toConsole();
    assertThat(table).row().row().value("name").isEqualTo("Evans"
);
}
```

row(int index)方法允许导航到index指定的行

```
// 导航到下标为 2 的行
assertThat(tableOrRequest).row(2)...

// 导航到下标为6的行
assertThat(tableOrRequest).row(2).row(6)...

// 下标为 2 的行的下一行,下标为3的行
assertThat(tableOrRequest).row(2).row()...
```

下面这个图红色部分表示从哪里开始可以导航到row



origin point是Table或者Request的row(...)方法,所以如果是从row,column或者value执行的,那么它们看起来就像从Table或者Request执行

当位置是在row,有可能回到origin

```
//从table中的row中返回table
assertThat(table).row().returnToTable()...
//从request中的row中返回request
assertThat(request).row().returnToRequest()...
```

下面两个是等价的:

```
assertThat(table).row().returnToTable().row()...

assertThat(table).row().row()...
```

to Column

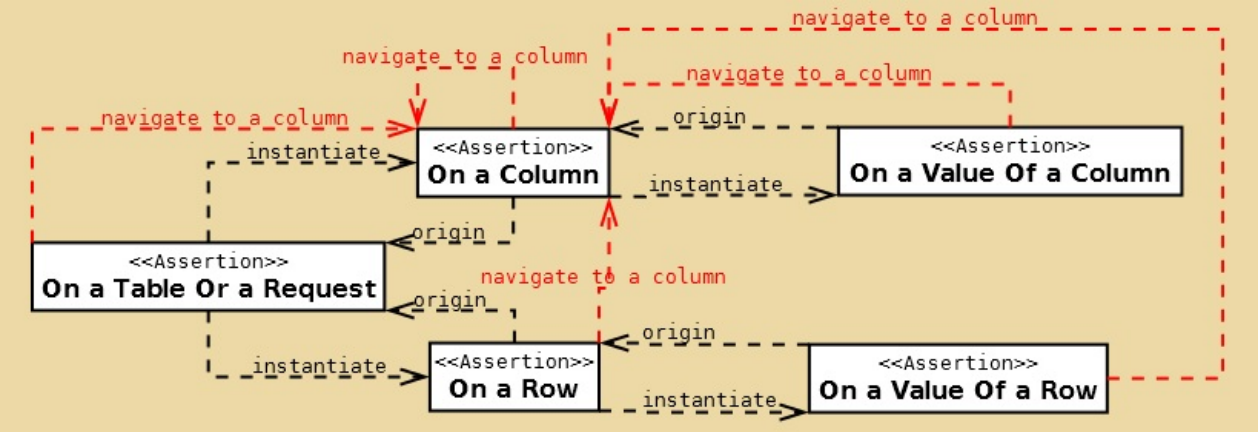
ToColumn接口的详情请查看这里

```
// 导航到第一列
Source source = new Source("jdbc:h2:mem:user-db;MODE=PostgreSQL;
INIT=RUNSCRIPT FROM " +
        "'/home/oem/javaProject/other/java-test/assertj-core/src
/test/resources/create_table.sql'", "sa", "");
Table table = new Table(source, "MEMBERS");

assertThat(table).column().column().hasColumnName("name");
//导航到第二列
assertThat(tableOrRequest).column().column()...
```

```
//导航到第二列
assertThat(table).column().column(1).hasColumnName("name");
//导航到第7列
assertThat(tableOrRequest).column(2).column(6)...
//导航到第4列
assertThat(tableOrRequest).column(2).column()...
//导航第1列
assertThat(tableOrRequest).row(2).column()...
//第4列
assertThat(tableOrRequest).row(2).column(3)...
//第五列
assertThat(tableOrRequest).column(3).row(2).column()...
```

```
//name字段
assertThat(tableOrRequest).column("name")...
//id字段
assertThat(tableOrRequest).column("name").column().column(6).col
umn("id")...
```

下图红色部分是表示从那些点开始执行可以导航到列上




```
assertThat(table).column().returnToTable()...

assertThat(request).column().returnToRequest()...
```

下面两者相等:

```
assertThat(table).column().returnToTable().column()...

assertThat(table).column().column().
```

to a value

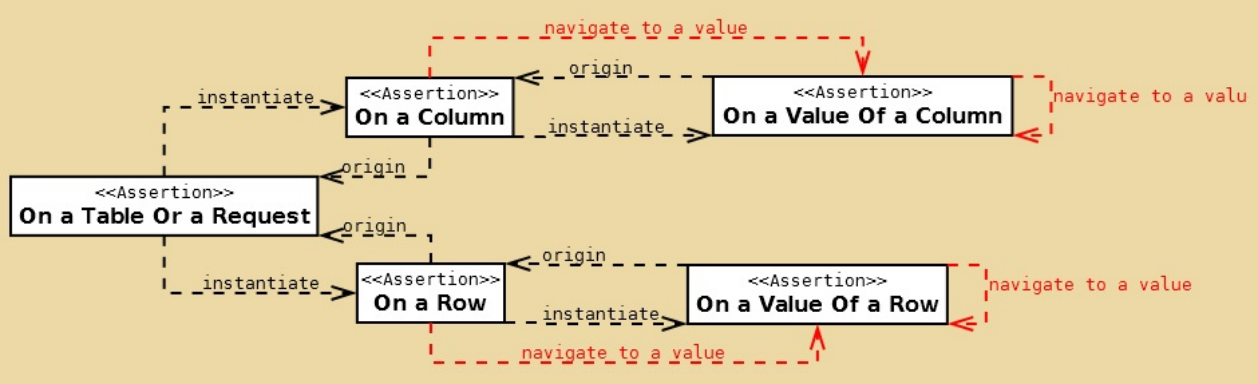
ToValue接口的详情可以看这里,ToValueFromRow接口的详情可以查看这里

```
//第一个value
assertThat(tableOrRequest).row().value()...
//第二个value
assertThat(tableOrRequest).column().value().value()...
```

```
//第三个value
assertThat(tableOrRequest).column().value(2)...
//第七个
assertThat(tableOrRequest).row(4).value(2).value(6)...
//第 4 个value
assertThat(tableOrRequest).column(4).value(2).value()...
//第5个
assertThat(tableOrRequest).column().value(3).row(2).column(0).va
lue()...
```

```
//为name的value
assertThat(tableOrRequest).row().value("name")...
//为id的value
assertThat(tableOrRequest).row().value("surname").value().value(6
).value("id")...
```

下图红色部分表示哪些点开始可以调用到value:




```
assertThat(table).column().value().returnToColumn()...

assertThat(request).row().value().returnToRow()...
```

下面二者相等:

```
assertThat(table).column().value().returnToColumn().value()...

assertThat(table).column().value().value()...
```

Changes作为根断言

to Change

```
//导航到创建
assertThat(changes).ofCreation()...
```

- 由于后续内容都比较简单,相信大家看到这里也应该明白它的大概用法了,若有兴趣可以请查看原文
- 这里是使用案例,大家过一遍结合文章运用起来问题不大

Gitbook该文件修订时间： 2017-03-13 05:37:11

H2介绍

介绍

H2 是一个Java编写的SQL数据库,主要特性有:

- 开源
- java编写
- ++支持标准SQL，JDBC API++
- 支持嵌入式模式和服务器模式，支持集群
- 兼容模式，适用于IBM DB2, Apache Derby, HSQLDB, MS SQL Server, MySQL, Oracle, and PostgreSQL

而我们对H2的使用，通常不是用替代Mysql/Oracle，而是在测试期间通过兼容模式模拟前者，以方便实现自动化测试以至持续继承。

H2数据库介绍

前言

> H2的中文资料不多，索性直接把官网资料中基本感兴趣的部分翻译成中文。

H2特性

> 注：中文翻译版本，原文来自H2官方文档--Feature List

H2 是一个Java编写的SQL数据库.

H2的主要特性有:

- 非常快的数据库引擎
- 开源
- java编写
- ++支持标准SQL，JDBC API++
- 支持嵌入式模式和服务器模式，支持集群
- 强大的安全特性
- PostgreSQL ODBC 驱动可以使用
- 并行多版本

额外特性：

- 基于硬盘或者内存的数据库和表，支持只读数据库，临时表
- 事务支持(read committed)，2-phase-commit
- 多连接，表级锁
- 基于消耗的优化器(Cost based optimizer)，为复杂查询使用演变算法(genetic algorithm?)，支持零管理(zero-administration)的可滚动和可更新result set，大型result set，外部结果排序，可返回result set的函数
- 加密数据库(AES)， SHA-256密码加密，加密函数， SSL

SQL 支持：

- Support for multiple schemas, information schema
- Referential integrity / foreign key constraints with cascade, check constraints
- Inner and outer joins, subqueries, read only views and inline views

- Triggers and Java functions / stored procedures
- Many built-in functions, including XML and lossless data compression
- Wide range of data types including large objects (BLOB/CLOB) and arrays
- Sequence and autoincrement columns, computed columns (can be used for function based indexes)
- ++ORDER BY, GROUP BY, HAVING, UNION, LIMIT, TOP++
- Collation support, including support for the ICU4J library
- Support for users and roles
- 兼容模式，适用于**IBM DB2, Apache Derby, HSQLDB, MS SQL Server, MySQL, Oracle, and PostgreSQL**

安全特性：

- Includes a solution for the SQL injection problem
- User password authentication uses SHA-256 and salt
- For server mode connections, user passwords are never transmitted in plain text over the network (even when using insecure connections; this only applies to the TCP server and not to the H2 Console however; it also doesn't apply if you set the password in the database URL)
- All database files (including script files that can be used to backup data) can be encrypted using the AES-128 encryption algorithm
- The remote JDBC driver supports TCP/IP connections over TLS
- The built-in web server supports connections over TLS
- Passwords can be sent to the database using char arrays instead of Strings

其他特性和工具：

- ++小巧(jar文件大概1.5 MB大小), 内存要求低++
- Multiple index types (b-tree, tree, hash)
- Support for multi-dimensional indexes
- CSV (comma separated values) file support
- Support for linked tables, and a built-in virtual 'range' table
- Supports the EXPLAIN PLAN statement; sophisticated trace options
- Database closing can be delayed or disabled to improve the performance
- 提供基于浏览器的控制台应用(翻译成多个语言)，支持自动完成
- The database can generate SQL script files
- Contains a recovery tool that can dump the contents of the database
- Support for variables (for example to calculate running totals)
- Automatic re-compilation of prepared statements
- Uses a small number of database files

- Uses a checksum for each record and log entry for data integrity
- Well tested (high code coverage, randomized stress tests)

连接模式

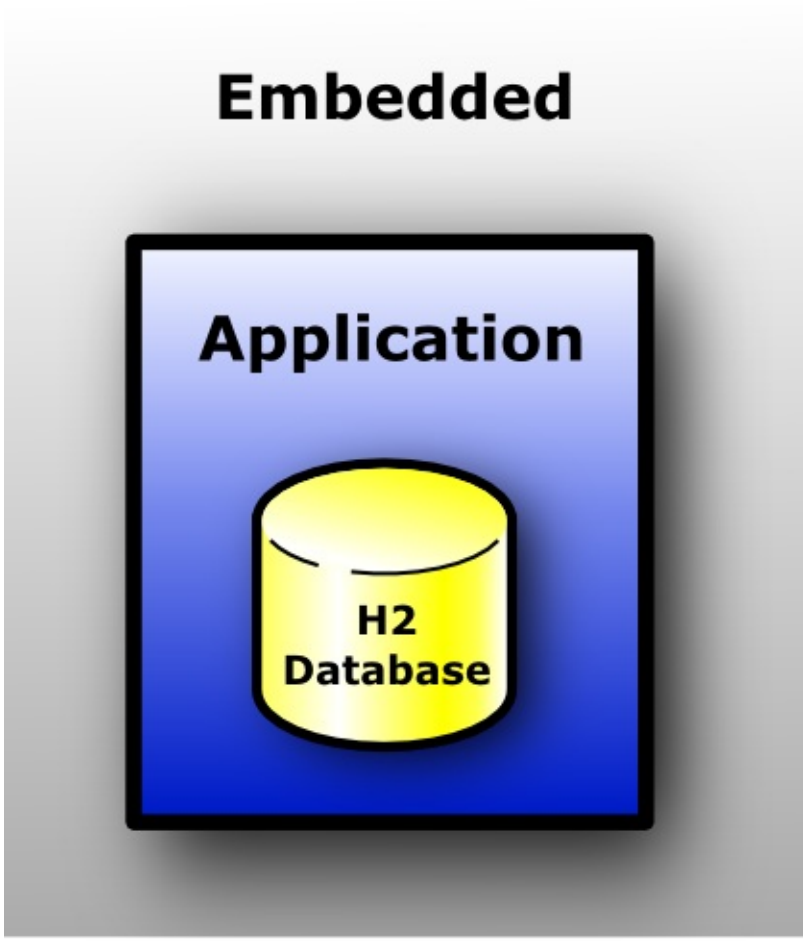
> 注：中文翻译版本，原文来自H2官网文档---Connection Modes.

H2支持三种连接模式：

- 内嵌模式：本地连接，使用JDBC
- 服务器模式：远程连接，在 TCP/IP使用 JDBC 或 ODBC
- 混合模式：同时支持本地连接和远程连接

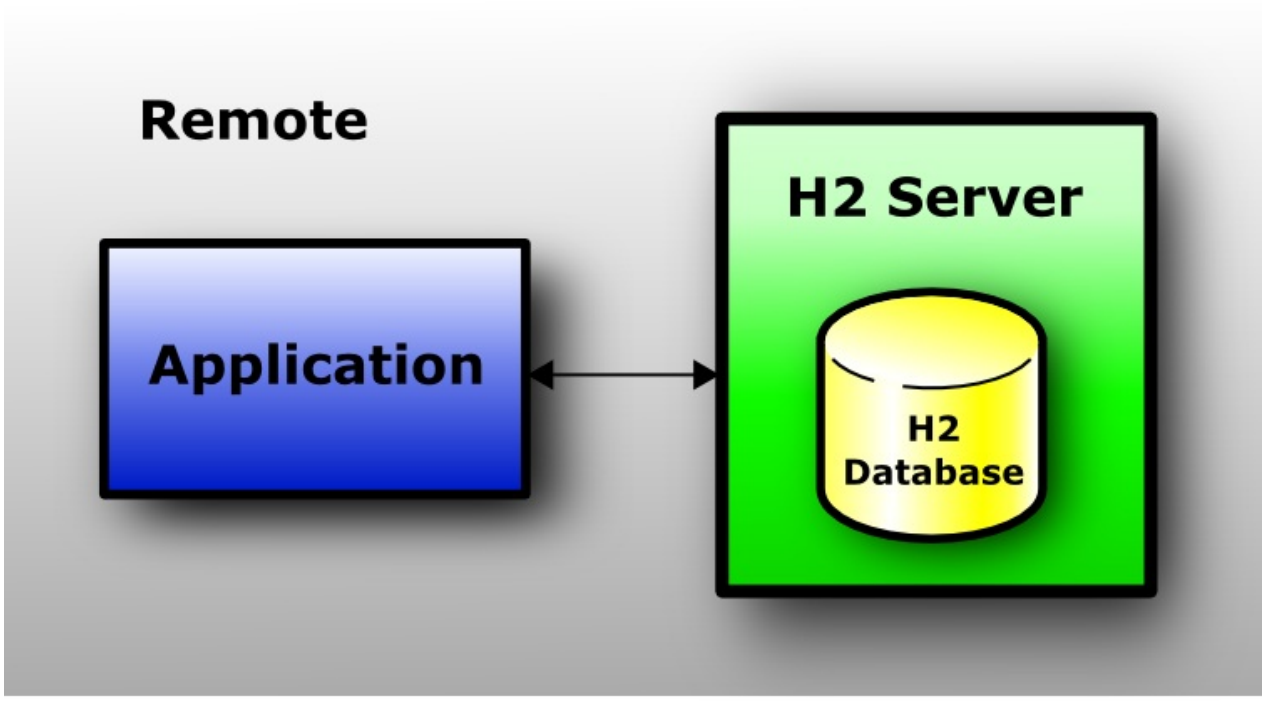
内嵌模式

在内嵌模式中，应用使用JDBC打来在同一个JVM中的数据库。 这是最快和最简单的连接模式。缺点是任何时间这个数据库都只能被一个虚拟机(或者class leader)打开。



服务器模式

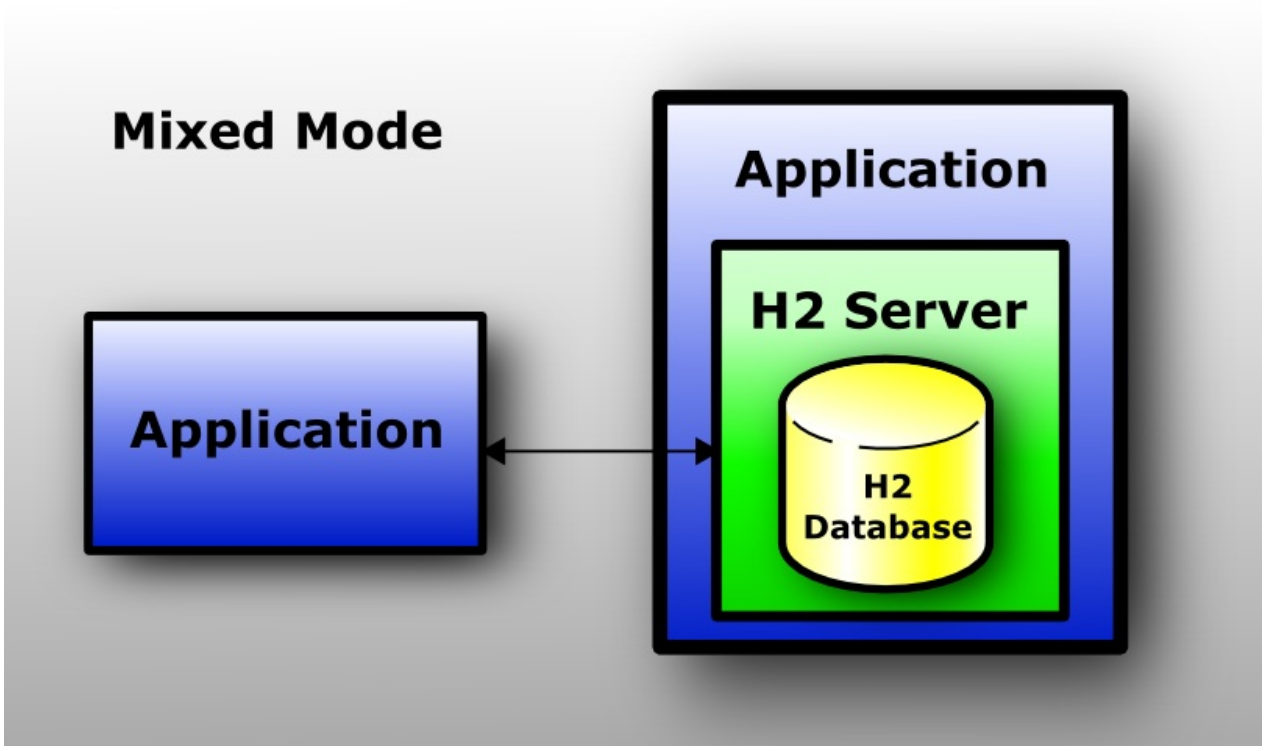
当使用服务器模式时(有时成为远程模式或者客户端/服务器模式)，应用通过使用JDBC 或 ODBC API来远程访问数据库。需要启动一个服务器，可以在同一个虚拟机或者其他虚拟机中，也可以在其他机器上。通过连接到这个服务器，可以有很多应用同时连接到同一个数据库。服务器进程在内部使用内嵌模式打开数据库。



混合模式

混合模式是内嵌模式和服务器模式的组合。第一个连接到数据库的应用采用内嵌模式，但同时开启服务器模式，以便其他应用（运行在不同的进程或者虚拟机中）可以并发的访问同样的数据。

服务器可以在应用中（使用服务器API）启动和停止，或者自动完成（自动混合模式）。当使用自动混合模式时，所有的试图连接到数据库的客户端（不管它是本地还是远程连接）都可以使用完全相同的数据库URL。



特性

> 注：中文翻译版本，原文来自H2官方文档--Database URL

H2数据库支持多种连接模式和连接设置，也提供很多使用的特性，这些都是通过使用不同的数据库URL来设置。

设置运行模式

1. 内嵌模式： 包括本地文件连接和内存数据库连接
2. 服务器模式：包括使用TCP和TLS连接
3. 混合模式

数据库设置

1. 指定用户名和密码
2. 调试跟踪设置
3. 忽略未知设置
4. 兼容模式
5. 当虚拟机退出时不要关闭数据库

数据库文件设置

1. 文件加密
2. 文件加锁方法
3. 定制文件访问模式
4. 将数据库存放在zip文件中

数据库连接设置

1. 在连接时执行SQL
2. 自动重连

运行模式

设置运行模式

内嵌模式

本地文件连接

连接到本地数据库的URL是"jdbc:h2:[file:][]"。其中前缀"file:"是可选的。如果没有设置路径或者只使用了相对路径，则当前工作目录将被作为起点使用。路径和数据库名称的大小写敏感取决于操作系统， 不过推荐只使用小写字母。数据库名称必须最少三个字母（File.createTempFile的限制）。数据库名字不容许包含分号";"。可以使用"~/"来指向当前用户home目录，例如"jdbc:h2:~/test"。

内嵌模式下的本地文件连接URL的格式是：

```
jdbc:h2:[file:][<path>]<databaseName>
```

例如：

```
jdbc:h2:~/test
jdbc:h2:file:/data/sample
jdbc:h2:file:C:/data/sample (仅仅用于Windows)
```

内存数据库连接

对于特殊使用场景(例如：快速原型开发，测试，高性能操作，只读数据库)，可能不需要持久化数据或数据的改变。H2数据库支持内存模式，数据不被持久化。

内存数据库资料存储有两种方式：

1. private/私有

   在一些场景中，要求仅仅有一个到内存数据库的连接。这意味着被开启的数据库是私有的(private)。在这个场景中，数据库URL是"jdbc:h2:mem:"。在同一个虚拟机中开启两个连接意味着打开两个不同的（私有）数据库。

2. named/命名： 其他应用可以通过使用命令来访问

   有时需要到同一个内存数据库的多个连接，在这个场景中，数据库URL必须包含一个名字。例如："jdbc:h2:mem:db1"。仅在同一个虚拟机和class loader下可以通过这个URL访问到同样的数据库。

对应的内存数据库连接URL的格式是：

- 私有格式：jdbc:h2:mem:
- 命名格式：jdbc:h2:mem:< databaseName >

命名格式的例子如下：

```
jdbc:h2:mem:test
```

默认，最后一个连接到数据库的连接关闭时就会关闭数据库。对于一个内存数据库，这意味着内容将会丢失。为了保持数据库开启，可以添加"DB_CLOSE_DELAY=-1"到数据库URL中。为了让内存数据库的数据在虚拟机运行时始终存在，请使用"jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"。

服务器模式

为了从其他进程或者其他机器访问一个内存数据库，需要在创建内存数据库的进程中启动一个TCP服务器。其他进程就需要通过TCP/IP 或者 TLS来访问数据库，使用URL类似"jdbc:h2:tcp://localhost/mem:db1".

H2支持两种连接方式， TCP和TLS：

使用**TCP**

格式如下，通过指定server和port连接到以内嵌模式运行的H2服务器，参数path和databaseName对应该内嵌数据库启动时的参数：

```
jdbc:h2:tcp://<server>[:<port>]/[<path>]<databaseName>
```

范例：

```
jdbc:h2:tcp://localhost/~/test          （对应内嵌模式下的jdbc:h2:~/test)
jdbc:h2:tcp://dbserv:8084/~/sample （对应内嵌模式下的jdbc:h2:~/sample)
jdbc:h2:tcp://localhost/mem:test     （对应内嵌模式下的jdbc:h2:mem:test)
```

使用**TLS**

可以通过采用SSL在传输层对数据库访问内容做加密，URL格式和TCP相同，只是TCP替换为ssl：

```
jdbc:h2:ssl://<server>[:<port>]/<databaseName>
```

范例：

```
jdbc:h2:ssl://localhost:8085/~/sample;
```

混合模式

通过设置AUTO_SERVER=TRUE可以开启自动混合模式：

```
jdbc:h2:<url>;AUTO_SERVER=TRUE
```

范例：

```
jdbc:h2:~/test;AUTO_SERVER=TRUE
```

数据库设置

数据库设置

指定用户名和密码

```
jdbc:h2:<url>[;USER=<username>][;PASSWORD=<value>]
```

范例：

```
jdbc:h2:file:~/sample;USER=sa;PASSWORD=123
```

调试跟踪设置

通过在URL中指定TRACE_LEVEL_FILE可以设置debug级别，取值为：

- 0：（TODO，未知）
- 1：
- 2：
- 3：

格式如下：

```
jdbc:h2:<url>;TRACE_LEVEL_FILE=<level 0..3>
```

范例：

```
jdbc:h2:file:~/sample;TRACE_LEVEL_FILE=3
```

忽略未知设置

某些应用(例如OpenOffice.org)在连接数据库时会传递一些额外参数。不清楚为什么会传递这些参数。例如参数PREFERDOSLIKELINEENDS 和 IGNOREDRIVERPRIVILEGES。为了改善和OpenOffice.org的兼容性他们被简单忽略。如果一个应用在连接到数据库时传递其他参数，通常数据库会抛出异常表示参数不被支持。可以通过在数据库URL中添加 ;IGNORE_UNKNOWN_SETTINGS=TRUE 来忽略这些参数：

格式如下：

```
jdbc:h2:<url>;IGNORE_UNKNOWN_SETTINGS=TRUE
```

兼容模式

注：这是一个至关重要的特性，设置兼容模式后可以非常大程度的模拟该数据库的一些常见语法和行为，以便mock，格式如下：

```
jdbc:h2:<url>;MODE=<databaseType>
```

范例：

```
jdbc:h2:~/test;MODE=MYSQL
```

对于某些特性，H2数据库可以默认特定数据库的行为。当然，这种情况下仅仅实现数据库之间差异的一个小的子集。下面是目前支持的模式和与正常默认的差别。

> 这里仅列出部分比较关注的数据库，详细列表请参考原文

MySQL 兼容模式

为了使用mysql模式，使用数据库URL jdbc:h2:~/test;MODE=MySQL 或者 SQL statement 设置MODE MySQL。

- 当插入数据时，如果列被定义为NOT NULL 和 插入的是NULL， 那么将会使用0值(或者空字符串，或者对于timestamp列使用当前timestamp)。通常，这种操作是不容许的并且会抛出异常。
- 在 CREATE TABLE 语句中创建索引容许使用INDEX(..) 或者 KEY(..)。例如：

```
  create table test(id int primary key, name varchar(255), k
ey idx_name(name));
```

- mata data 调用返回使用小写的标识符
- 当转换浮点数到整型时, the fractional digits are not truncated, but the value is rounded.
- 连接NULL和其他值将得到其他值

Mysql中文本比较默认大小写不敏感， 而在H2(同样在大多数其他数据库)中是大小写敏感。H2支持大小写不敏感的文本比较，但是需要通过设置 IGNORECASE TRUE 单独设置。这会影响使用 =, LIKE, REGEXP 的比较。

> TODO: 如何设置IGNORECASE TRUE?

Oracle 兼容模式

为了使用Oracle模式，使用数据库URL jdbc:h2:~/test;MODE=Oracle 或者 SQL statement 设置MODE Oracle。

- 对于别名列, ResultSetMetaData.getColumnName() 返回别名而 getTableName() 返回 null.
- 当使用唯一索引时, 容许在所有列中有多个行是NULL值, 但是不容许多行有同样的值.
- 连接NULL和其他值将得到其他值
- 空字符串被处理为 like NULL 值

PostgreSQL 兼容模式

为了使用 PostgreSQL 模式，使用数据库URL jdbc:h2:~/test;MODE=PostgreSQL 或者 SQL statement 设置MODE PostgreSQL。

- 对于别名列, ResultSetMetaData.getColumnName() 返回别名而 getTableName() 返回 null.
- 当转换浮点数到整型时, the fractional digits are not truncated, but the value is rounded.
- 支持系统列 CTID 和 OID
- 在这种模式下LOG(x) is base 10

当虚拟机退出时不要关闭数据库

默认情况下H2在虚拟机退出时是会自动关闭数据库， 通过在URl中设置DB_CLOSE_ON_EXIT=FALSE可以改变这个行为（注：暂不清楚申明情况下需要不关闭？），格式：

```
jdbc:h2:<url>;DB_CLOSE_ON_EXIT=FALSE
```

数据库文件设置

文件加密

数据库文件可以加密。H2支持AES加密算法。为了使用文件加密，需要指定加密算法（参数cipher）和连接到数据库时的文件密码（在用户密码之外）。

可以对数据库文件做加密，通过在URL中指定"CIPHER=AES":

```
jdbc:h2:<url>;CIPHER=AES
```

范例：

```
jdbc:h2:ssl://localhost/~/test;CIPHER=AES
jdbc:h2:file:~/secure;CIPHER=AES
```

创建一个文件加密的新数据库

默认，如果数据库不存在会自动创建一个新的数据库。为了创建一个加密的数据库，连接它就像它好像已经存在一样。

连接到加密数据库

连接到一个加密后的数据库， 除了设置CIPHER=AES外，还需要在密码字段中设置文件密码，在用户密码前。在文件密码和用户密码之间用一个空格简单分割，因此文件密码是不容许包含空格的。文件密码和用户密码都是大小写敏感。

这里是连接到加密数据库的例子，java代码如下：

```
Class.forName("org.h2.Driver");
String url = "jdbc:h2:~/test;CIPHER=AES";
String user = "sa";
String pwds = "filepwd userpwd";
conn = DriverManager.getConnection(url, user, pwds);
```

注意pwds的格式，空格分开，前面是文件密码，后面是用户密码。

> TODO：文件密码是哪里设置的？猜测是第一次创建数据库时使用这里的文件密码，之后就依靠这个密码继续访问。

加密或者解密数据库

为了加密一个已经存在的数据库，可以使用ChangeFileEncryption工具。这个工作也同样用于解密数据库，或者修改文件加密的密钥。这个工作在H2的console上的工具区中可以得到，或者你可以在命令行运行它。下面的命令行将加密user home目录下的test数据库，使用文件密码filepwd和AES加密算法：

```
java -cp h2*.jar org.h2.tools.ChangeFileEncryption -dir ~ -db test -cipher AES -encrypt filepwd
```

文件加锁方法

H2可以对数据库文件做加锁操作，指定FILE_LOCK：

```
jdbc:h2:<url>;FILE_LOCK={FILE|SOCKET|FS|NO}
```

范例：

```
jdbc:h2:file:~/private;CIPHER=AES;FILE_LOCK=SOCKET
```

细节见H2文档 database_file_locking。

仅当数据库存在时打开

默认情况下，当URL指定的数据库文件不存在时，H2会自动创建文件并打开该数据库。可以通过设置IFEXISTS=TRUE来改变这个行为，仅当指定数据库已经存在时才打开：

```
jdbc:h2:<url>;IFEXISTS=TRUE
```

范例：

```
jdbc:h2:file:~/sample;IFEXISTS=TRUE
```

细节见文档。

定制文件访问模式

通常，数据库用rw(读写)模式打开数据库文件(除非是只读数据库，那会使用r/只读模式)。为了用只读模式打开一个文件不是只读的数据库，需要设置ACCESS_MODE_DATA=r。也支持rws和rwd。这个设置必须在数据库URL中指明：

```
String url = "jdbc:h2:~/test;ACCESS_MODE_DATA=rws";
```

rwd和rws的含义：

- rwd: 文件内容的每个更新都被同步写入底层存储设备.
- rws: 在rwd的基础上，metadata的每个更新都被同步写入.

具体内容参考 H2文档 Durability Problems.

将数据库存放在zip文件中

可以通过这个方式将数据库文件存放在一个zip文件中，虽然不清楚何时适合如此做，格式如下：

```
jdbc:h2:zip:<zipFileName>!/<databaseName>
```

范例：

```
jdbc:h2:zip:~/db.zip!/test
```

数据库连接设置

在连接时执行SQL

可以通过下面的方式在连接到数据库时立即执行保存在文件中的SQl文件：

```
jdbc:h2:<url>;INIT=RUNSCRIPT FROM '~/create.sql'
```

如果需要执行多个SQL文件，则可以重复多次：

```
jdbc:h2:file:~/sample;INIT=RUNSCRIPT FROM '~/create.sql'\;RUNSCR
IPT FROM '~/populate.sql'
```

特别是内存数据库，当客户端连接到数据库时可以自动执行DDL或者DML命令是非常实用的。这个功能可以通过INIT参数开启。注意可以给INIT传递多个命令，但是分号";"必须转义，如下面的例子：

```
String url = "jdbc:h2:mem:test;INIT=runscript from '~/create.sql
'\\;runscript from '~/init.sql'";
```

注意双反斜杠仅仅是在java或者属性文件中需要。在GUI，或者XML文件中，仅仅需要一个反斜杠：

```
<property name="url" value=
"jdbc:h2:mem:test;INIT=create schema if not exists test\;runscri
pt from '~/sql/init.sql'"
/>
```

init脚本中的反斜杠(例如在runscript中，指定windows下的一个目录名)也需要转义(用第二个反斜杠)。因为这个原因可以简单的替代使用前向的斜杠，避免使用反斜杠。

自动重连

```
jdbc:h2:<url>;AUTO_RECONNECT=TRUE
```

范例：

```
jdbc:h2:tcp://localhost/~/test;AUTO_RECONNECT=TRUE
```

Page size jdbc:h2:;PAGE_SIZE=512 Changing other settings jdbc:h2:;=[;=...] jdbc:h2:file:~/sample;TRACE_LEVEL_SYSTEM_OUT=3

H2在测试中的使用

关注特性

前面我们介绍了H2的众多特性，其中有部分特性是我们关注的重点：

- java编写
- ++支持标准SQL，JDBC API++
- 支持嵌入式模式和服务器模式，支持集群
- 兼容模式，适用于**IBM DB2, Apache Derby, HSQLDB, MS SQL Server, MySQL, Oracle, and PostgreSQL**
- ++小巧(jar文件大概1.5 MB大小), 内存要求低++

适用场景

这几个特性，使得H2非常的适合用于数据库访问层的自动化，典型场景：

- 以嵌入式模式在Java测试案例中启动
- 模拟其他数据库如mysql/Oracle/PostgreSQL
- 提供标准JDBC支持，构建DataSource/Connection等
- 使得数据库访问层的代码可以在H2的基础上运行/测试/验证

在使用H2模拟mysq/oracle/postgresql等真实数据库之后，涉及数据库访问的功能模块的测试就变得方便而且容易自动化，在此基础上，持续集成/CI也就方便了。

限制

H2对其他数据库的替代，毕竟停留在模拟的层次上，不可能在所有的细节上都和原数据库完全一致。因此如果涉及到比较偏门的数据库独有特性，可能无法模拟。

Demo

为了展示H2的测试，提供了一个代码Demo，地址在 https://github.com/skyao/leaning-h2/。

请在clone仓库之后，切换到demo branch。

> git chekcout demo

标准的maven项目，直接倒入IDE即可。

目前提供以下demo：

- purejdbc： 纯JDBC的demo
- openjpa： OpenJPA的demo

纯JDBC

对于pure jdbc的测试场景，我们需要做的是创建连接到H2的DataSource，然后test case就可以将这个datasource用于DAO。

> 注：完整代码请见purejdbc项目。

测试基类

deme中提供的测试基类如下：

```
public class H2TestBase {

    protected DataSource dataSource;

    @Before
    public void prepareH2() throws Exception {
        JdbcDataSource ds = new JdbcDataSource();
        // choose file system or memory to save data files
        // ds.setURL("jdbc:h2:~/test;MODE=POSTGRESQL;INIT=runscr
ipt from './src/test/resources/createTable.sql'");
        // ds.setURL("jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;MODE=PO
STGRESQL;INIT=runscript from './src/test/resources/createTable.s
ql'");
        ds.setURL("jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;MODE=POSTG
RESQL;INIT=runscript from './src/test/resources/createTable.sql'"
);

        ds.setUser("sa");
        ds.setPassword("");

        dataSource = ds;
    }
}
```

注意在URL中通过设置"INIT=runscript from './src/test/resources/createTable.sql'"，在打开数据库时执行建表语句。而"MODE=POSTGRESQL;"指明了当前兼容的数据库模式是POSTGRESQL。

测试案例

各个子类，就是是具体的test case，通过简单继承就可以方便的使用datasource，比如创建connection：

```
public class AddressTest extends H2TestBase {
    private Connection connection;

    @Before
    public void prepareConnection() throws SQLException {
        connection = dataSource.getConnection();
    }
}
```

然后就可以继续编写具体的测试方法：

```
    @Test
    public void testPureJdbc() throws Exception {
        PreparedStatement statement = connection.prepareStatemen
t("INSERT INTO address(id, name) VALUES(1, 'Miller');");
        assertThat(statement.executeUpdate()).isEqualTo(1);

        statement = connection.prepareStatement("select id, name
 from address");
        ResultSet executeQuery = statement.executeQuery();
        executeQuery.next();
        int id = executeQuery.getInt(1);
        String name = executeQuery.getString(2);

        assertThat(id).isEqualTo(1);
        assertThat(name).isEqualTo("Miller");

        //System.out.println("id=" + id + ", name=" + name);
    }
```

OpenJPA

对于OpenJPA的测试场景，稍微复杂一些。

> 注：完整代码请见openjpa项目。

测试基类

测试基类如下，这里要得到的是EntityManager，openjpa中DAO需要的是这个EntityManager对象：

```
public class AbstractH2DaoTest {

    private static EntityManagerFactory emf;
    protected EntityManager entityManager;

    @Before
    public void prepareH2() throws Exception {
        if (emf == null) {
            try {
                emf = Persistence.createEntityManagerFactory("Em
beddedH2");
            } catch (Throwable e) {
                System.out.printf("******  fail to initial openJ
pa and H2    *******");
                e.printStackTrace();
                Assert.fail("fail to initial openJpa and H2");
            }
        }

        entityManager = emf.createEntityManager();
        entityManager.getTransaction().begin();
    }

    @After
    public void tearDown() {
        if (entityManager != null) {
            entityManager.getTransaction().commit();
            entityManager = null;
        }
    }
}
```

注意这里的@Before和@After两个方法中，分别做了Transaction的begin和commit。后面会介绍为什么需要如此。

persistence.xml

对应上面创建EntityManagerFactory的语句：

> Persistence.createEntityManagerFactory("EmbeddedH2");

需要准备文件 src/test/resources/META-INF/persistence.xml， 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence h
ttp://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="EmbeddedH2"
        transaction-type="RESOURCE_LOCAL">
        <provider>org.apache.openjpa.persistence.PersistenceProv
iderImpl</provider>
        <class>com.github.skyao.h2demo.entity.AddressEntity</cla
ss>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        <properties>
            <property name="javax.persistence.jdbc.driver" value=
"org.h2.Driver" />
            <property name="javax.persistence.jdbc.url"
                value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;MODE=P
OSTGRESQL" />
            <property name="javax.persistence.jdbc.user" value="
sa" />
            <property name="javax.persistence.jdbc.password" val
ue="" />
            <property name="openjpa.RuntimeUnenhancedClasses" va
lue="supported" />
            <property name="openjpa.jdbc.SynchronizeMappings" va
lue="buildSchema(ForeignKeys=true)" />
        </properties>
    </persistence-unit>
</persistence>
```

这里的 persistence-unit name="EmbeddedH2" 中的name需要和前面Persistence.createEntityManagerFactory()保持名字一致。在这种情况下，transaction-type只能填写"RESOURCE_LOCAL"。

RESOURCE_LOCAL会带来一个问题，由于没有完整的oepanjpa支持，数据库事务需要自己手工提交。因此我们不得已，只好如上面的基类代码中那样在每个测试案例执行前(@Before)begin transaction，在每个测试案例执行后(@After)commit

transaction。

注意这里的H2 数据库URL中不再有建表语句的sql需要执行，这是因为openjpa自身提供自动建表的功能，因此可以不想纯JDBC下面那边需要自己动手。

测试案例

测试案例通过继承基类得到EntityManager，注入给要测试的DAO：

```
public class AddressDaoImplTest extends AbstractH2DaoTest {

    private AddressDaoImpl dao;

    @Before
    public void prepareDao() {
        dao = new AddressDaoImpl();
        dao.setEntityManager(this.entityManager);
    }

    @Test
    public void testSaveAndQuery() {
        AddressEntity addressEntity =new AddressEntity();
        addressEntity.setId(100);
        addressEntity.setName("sky");
        int id = dao.save(addressEntity);
        //System.out.println(addressEntity.toString());

        AddressEntity queryEntity = dao.query(id);
        //System.out.println(queryEntity.toString());
        assertThat(queryEntity).isEqualToComparingFieldByField(a
ddressEntity);
    }
}
```

DbUnit

DbUnit是一个JUnit扩展,针对数据库驱动的项目. DbUnit支持数据库中的数据导出和导入和XML数据集的能力。从2.0版本开始，DbUnit也支持非常大的数据集流模式下使用。 DbUnit的还可以帮助你验证数据是否符合预期

Maven依赖

```
<dependency>
  <groupId>org.dbunit</groupId>
  <artifactId>dbunit</artifactId>
  <version>${dbunitVersion}</version>
  <scope>test</scope>
</dependency>
```

官网

入门

分别介绍以下几种方案:

1. 编写一个DbTestCase的子类设置数据库
2. 使用你写的TestCase子类设置数据库
3. 无父类设置数据库
4. 数据库校验
5. 数据文件加载
6. 使用DbUnit的Ant任务做webTest

DBTestClass的子类设置数据库

1. 首先你需要创建一个数据集，比如说xml的方式，你可以手动创建一个xml或者从数据库到处一些数据来创建xml
2. 继承一个DBTestCase类 现在你需要创建一个类，让它继承DBTestCase类。DBTestCase拓展了JUnit的TestCase类,需要实现一个模板方法:getDataSet()用来返回在步骤1中创建好的数据集,DbTestCase依靠IDatabaseTester工作，它默认使用
   PropertiesBasedJdbcDatabaseTester,PropertiesBasedJdbcDatabaseTester可以定位DriverManager类中的系统配置。你可以通过测试类的构造方法实现,也可以通过重写getDatabaseTeser()方法实现,也可以通过以下提供的类来实现:

   - JdbcBasedDBTestCase : 使用DriverManager创建链接
   - DataSourceBasedDBTestCase : 使用 javax.sql.DataSource创建链接
   - JndiBasedDBTestCase : 使用javax.sql.DataSource定位JNDI
   - DefaultPrepAndExpectedTestCase ： 使用可配置的IDatabaseTester(可以是任何类型的链接)准备明确分离和预期的数据集 以下是返回到Hypersonic数据库的连接和xml数据集，示例实现：

```
public class SampleTest extends DBTestCase{
  public SampleTest(String name){
      super( name );
             System.setProperty(PropertiesBasedJdbcDataba
seTester.DBUNIT_DRIVER_CLASS, "org.h2.Driver");
  System.setProperty(PropertiesBasedJdbcDatabaseTester.DB
UNIT_CONNECTION_URL,
          "jdbc:h2:mem:user-db;MODE=PostgreSQL;INIT=RUNSC
RIPT FROM './src/test/resources/sql/create_table.sql'");
  System.setProperty(PropertiesBasedJdbcDatabaseTester.DB
UNIT_USERNAME, "sa");
  System.setProperty(PropertiesBasedJdbcDatabaseTester.DB
UNIT_PASSWORD, "sa");
  // System.setProperty( PropertiesBasedJdbcDatabaseTeste
r.DBUNIT_SCHEMA, "" );
  }

  protected IDataSet getDataSet() throws Exception{
      return new FlatXmlDataSetBuilder().build(new FileIn
putStream("dataset.xml"));
  }
}
```

3. 实现getSetUpOeration()和getTearDownOperation()方法 在默认情况下DbUnit在执行方法之前都会执行CLEAN_INSERT操作,之后不再执行任何清除操作.你可以通过重写getSetUpOeration()和getTearDownOperation()方法来修改此行为.

```
 public class SampleTest extends DBTestCase{
      ...
      protected DatabaseOperation getSetUpOperation() throws
Exception{
          return DatabaseOperation.REFRESH;
      }

      protected DatabaseOperation getTearDownOperation() thro
ws Exception{
          return DatabaseOperation.NONE;
      }
      ...
 }
```

4. 重写setUpDatabaseConfig(DatabaseConfig config)方法 使用它可以更改DbUnit DatabaseCOnfig的一些配置

```
public class SampleTest extends DBTestCase{
      ...
      /**
       * Override method to set custom properties/features
       */
      protected void setUpDatabaseConfig(DatabaseConfig confi
g) {
          config.setProperty(DatabaseConfig.PROPERTY_BATCH_SI
ZE, new Integer(97));
          config.setFeature(DatabaseConfig.FEATURE_BATCHED_ST
ATEMENTS, true);
      }
      ...
 }
```

5. 实现你的testXXX()方法

用TestCase的子类做数据库设置

你可以覆盖标准的JUnit的setUp()方法,然后执行你所需要的数据库操作,并且在类似teardown()的方法中做清理工作 比如:

```
public class TestCaseSubClass extends TestCase {
    public TestCaseSubClass(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        super.setUp();
        super.setUp();

        // initialize your database connection here
        IDatabaseConnection connection = null;
        // ...

        // initialize your dataset here
        IDataSet dataSet = null;
        // ...

        try {
            DatabaseOperation.CLEAN_INSERT.execute(connection, d
ataSet);
        } finally {
            connection.close();
        }
    }
}
```

从2.2版本开始使用新的IDatabaseTester来提供相似功能.如前文所提DbTesteCase使用IDatabaseTester做内部实现,也可以在你的测试类中使用它来处理DataSets.现在有如下四中实现:

- JdbcDatabaseTester: 使用DriverManager创建连接
- PropertiesBasedJdbcDatabaseTester: 也是使用DriverManager,但是配置是从系统属性中读取,这也是DbTestCase的默认实现
- DataSourceDatabaseTester: 使用 javax.sql.DataSource创建连接

- JndiDatabaseTester: 使用javax.sql.DataSource定位JDNI

你也可以自己提供IDatabaseTester实现,它建议使用AbstractDatabaseTester做为起始点 例如:

```
public class TestIDatabaseTester extends TestCase {
    private IDatabaseTester databaseTester;

    public TestIDatabaseTester(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        databaseTester = new JdbcDatabaseTester("org.h2.Driver",
                "jdbc:h2:mem:user-db;MODE=PostgreSQL;INIT=RUNSCR
IPT FROM './src/test/resources/sql/create_table.sql'",
                "sa", "sa");

        IDataSet dataSet = new FlatXmlDataSetBuilder().build(new
 FileInputStream("./src/test/resources/dataset.xml"));

        databaseTester.setDataSet(dataSet);
        // will call default setUpOperation
        databaseTester.onSetup();
    }

    protected void tearDown() throws Exception {
        // will call default tearDownOperation
        databaseTester.onTearDown();
    }

}
```

非继承实现方式

简单的配置DBTestCase子类的实例,无论是直接实例化或者依赖于测试类的注解.比如使用PrepAndExpectedTestCase:

```
public class TestNoParent {
    private PrepAndExpectedTestCase tc; // injected or instantia
ted, already configured

    @Test
    public void testExample() throws Exception {
        final String[] prepDataFiles = {}; // define prep files
        final String[] expectedDataFiles = {}; // define expecte
d files
        final VerifyTableDefinition[] tables = {}; // define tab
les to verify
        final PrepAndExpectedTestCaseSteps testSteps = () -> {
            return null; // or an object for use outside the Ste
ps
        };

        tc.runTest(tables, prepDataFiles, expectedDataFiles, tes
tSteps);
    }
}
```

数据库数据验证

DbUnit提供了用于验证两个表或者数据集是否含有相同的数据的支持.以下两种方式可以用来验证数据库是否包含测试用例执行期间的预期数据。

```
public class Assertion{
    public static void assertEquals(ITable expected, ITable actu
al)
    public static void assertEquals(IDataSet expected, IDataSet
actual)
}
```

下面这个例子告诉大家如何将数据库表中的快照跟xml表中的数据进行比较:

```
public class TestVerification extends DBTestCase {
    public TestVerification(String name) {
        super(name);
    }

    @Override
    protected IDataSet getDataSet() throws Exception {
        return null;
    }

    public void testMe() throws Exception {
        // Execute the tested code that modify the database here

        // Fetch database data after executing your code
        IDataSet databaseDataSet = getConnection().createDataSet
();
        ITable actualTable = databaseDataSet.getTable("TABLE_NAM
E");

        // Load expected data from an XML dataset
        IDataSet expectedDataSet = new FlatXmlDataSetBuilder().b
uild(new File("expectedDataSet.xml"));
        ITable expectedTable = expectedDataSet.getTable("TABLE_N
AME");

        // Assert actual database table match expected table
        Assertion.assertEquals(expectedTable, actualTable);
    }
}
```

actualTable(实际数据)是你想要验证的数据库的数据快照，expectedTable里面存放的是预期数据 预期数据必须跟你设置的数据库不同,因此你需要两个dataset。一个在你的测试代码之前设置你的数据库,一个在测试期间提供预期的数据

使用query读取你的数据库快照

你也可以用query语句验证你结果是否符合预期,语句可以select一张表或者join多张表：

```
    ITable actualJoinData = getConnection().createQueryTable("RE
SULT_NAME", "SELECT * FROM TABLE1, TABLE2 WHERE ...");
```

忽略比较某些字段

有的时候是需要胡烈一些字段来进行比较的，尤其是主键，日期或者由测试代码生成的时间.你可以在预期表中声明忽略的字段,然后你就可以过滤掉实际数据表中的字段了，只需要暴露预期表中的字段 下面这个代码展现了如何筛选实际表中的字段.在实际工作当中,实际表中的字段必须包含预期表中的所有字段.也可以有一些字段是预期表中没有的.

```
    ITable filteredTable = DefaultColumnFilter.includedColumnsTa
ble(actual,expected.getTableMetaData().getColumns());
    Assertion.assertEquals(expected, filteredTable);
```

可以用你自己定义的IColumnFilter或者DbUnit自带的DefaultColumnFilter来实现以上功能;DefaultColumnFilter支持通配符,还提供了一些方便的方法.includedColumnsTable()和excludedColumnsTable()可以简单的创建过滤字段。

下面这个例子展示了DefaultColumnFilter过滤ＰＫ为前缀或者以TIME为后缀的字段

```
    DefaultColumnFilter columnFilter = new DefaultColumnFilter()
;
    columnFilter.excludeColumn("PK*");
    columnFilter.excludeColumn("*TIME");

    FilteredTableMetaData metaData = new FilteredTableMetaData(o
riginalTable.getTableMetaData(), columnFilter);
```

或者

```
    ITable filteredTable = DefaultColumnFilter.excludedColumnsTa
ble(originalTable, new String[]{"PK*", "*TIME"});
```

字段排序

DbUnit默认情况下获取到的数据库快照是使用主键来排序的.若一个表没有主键或者主键由数据库自动生成的,字段排序是不可预测的华那么assertEquals将会失败. 这种情况你必须使用IDatabaseConnection来定制你的数据库快照.例如带"order by"子句的createQueryTable,或者使用SortedTable修饰:

```
    Assertion.assertEquals(new SortedTable(expected),
    new SortedTable(actual, expected.getTableMetaData()));
```

SortedTable在默认情况下是使用字段的字符串值来做排序的,所以如果你是使用的数字字段排序,你得到的排序序号可能是:1,10,11,12,2,3,4,如果你想按照字段的数据类型排序（获得排序顺序:1,2,3,4,10,11,12）你可以像下面这样做:

```
    SortedTable sortedTable1 = new SortedTable(table1, new Strin
g[]{"COLUMN1"});
    sortedTable1.setUseComparable(true); // must be invoked imme
diately after the constructor
    SortedTable sortedTable2 = new SortedTable(table2, new Strin
g[]{"COLUMN1"});
    sortedTable2.setUseComparable(true); // must be invoked imme
diately after the constructor
    Assertion.assertEquals(sortedTable1, sortedTable2);
```

断言和收集差异

在默认情况下,DbUnit在犯下第一个数据差异时就会立即失败.从2.4版本开始,可以注册一个自定的 FailureHandler,它可以制定抛出的各种异常,以及如何处理数据差异的出现。使用DiffCollectingFailureHandler可以避免数据不匹配的抛出,这样可以评估比较的所有结果

```
IDataSet dataSet = getDataSet();
DiffCollectingFailureHandler myHandler = new DiffCollectingFailu
reHandler();
//invoke the assertion with the custom handler
assertion.assertEquals(dataSet.getTable("TEST_TABLE"),
                       dataSet.getTable("TEST_TABLE_WITH_WRONG_V
ALUE"),
                       myHandler);
// Evaluate the results and throw an failure if you wish
List diffList = myHandler.getDiffList();
Difference diff = (Difference)diffList.get(0);
...
```

数据文件加载

几乎所有的测试都需要从一个或多个文件中加载数据,特别是预期的数据。DbUnit有一组数据文件加载工具辅助类,可以从classpath加载数据集.例如:

```
DataFileLoader loader = new FlatXmlDataFileLoader();
IDataSet ds = loader.load("/the/package/prepData.xml");
```

最佳实践

1. 使用一个数据库实例作为开发 一个数据库应该只用于一个测试,否则数据库状态不能guarantied.因此多个开发人员在开发同一个项目时应该由自己的数据库实例以防止数据损坏.这也简化了数据库清理,因为你并不一定需要把数据库恢复到初始状态

2. 好的设置不需要清理 如果你每个测试使用一个数据库实例,那么你不需要担心你的测试留下痕迹.如果你总是在已知情况下的测试之前创建数据库实例，那么就不需要每次都清理数据库,这样可以简化测试维护，节省了清理数据库的开销，对于手动验证数据库也是有很大的帮助

3. 使用多个小型数据集 大部分测试不要求整个数据库重新初始化,因此,你可以尝试把一个大型的数据集分解成许多个小的数据集 这些小的数据集可以大致对应一个逻辑单元或者组件,减少了由数据库初始化对于每个测试的开销.对于集成测试,还可以用CompositeDataSet类在运行时将多个数据集合并成一个大的数据集

4. 整个测试类或者测试套件执行时数据库值设置一次 如果多次测试使用相同的只读数据,该数据可以一次行对整个测试类或者测试套件进行初始化,但是你需要确保永远不会修改这些数据，虽然减少了测试运行的时间,但是也加大了风险

5. 链接管理策略 以下是推荐的链接管理策略:

   i. 远程客户端DatabaseTestCase 尽量重用测试套件中相同的链接,以减少每个测试都创建一个新的链接的开销,从１．１版本开始DatabaseTestCase在setUp()和tearDown()关闭链接,你可以通过覆盖closeConnection()方法体为空实现来修改这个行为

   ii. 在容器内链接数据库 如果你选择了在容器内的策略,应该使用DatabaseDataSourceConnection类访问应用服务器配置的数据源。

```
 IDatabaseConnection connection = new DatabaseDataSourceC
onnection(new InitialContext(), "jdbc/myDataSource");
```

   2.2版本的替代方案是继承JndiBasedDBTestCase并指定jndi查找名称

```
 public class MyJNDIDatabaseTest extends JndiBasedDBTestC
ase {
    protected String getLookupName(){
       return "jdbc/myDatasource";
    }
    ...
 }
```

DatabaseOperation

DatabaseOperation定义了对数据库进行的操作，它是一个抽象类，通过静态字段提供了几种内置的实现：

- NONE：不执行任何操作，是getTearDownOperation的默认返回值。
- UPDATE：将数据集中的内容更新到数据库中。它假设数据库中已经有对应的记录，否则将失败。
- INSERT：将数据集中的内容插入到数据库中。它假设数据库中没有对应的记录，否则将失败。
- REFRESH：将数据集中的内容刷新到数据库中。如果数据库有对应的记录，则更新，没有则插入。
- DELETE：删除数据库中与数据集对应的记录。
- DELETE_ALL：删除表中所有的记录，如果没有对应的表，则不受影响。
- TRUNCATE_TABLE：与DELETE_ALL类似，更轻量级，不能rollback。
- CLEAN_INSERT：是一个组合操作，是DELETE_ALL和INSERT的组合。是getSetUpOeration的默认返回值。

ReplacementDataSet

该类可以用来替换数据集中的占位符,实现动态数据传入.例如有如下数据集:

```
    <?xml version='1.0' encoding='UTF-8'?>
    <dataset>
        <table_rule id="${ruleId}" description="description" gro
up_id="group_id" app_id="app_id" window_unit="${window_unit}" is
_sensitive="1" rate="${rate}" window_interval="${window_interval
}"/>
    </dataset>
```

用下面代码就可以将占位符号${xxx}动态换成你想要的数据,然后执行就可以将数据插入数据库了

```
    IDataSet dataSet = new FlatXmlDataSetBuilder().build(new Fil
e
            ("/home/oem/javaProject/dolphin/throttle-service/fun
ction-test/src/test/resources/dataset/dataset" +
                    ".xml"));
    ReplacementDataSet replacementDataSet = new ReplacementDataS
et(dataSet);
    replacementDataSet.addReplacementSubstring("${ruleId}", rule
Id);
    replacementDataSet.addReplacementObject("${window_interval}"
, windowInterval);
    replacementDataSet.addReplacementObject("${window_unit}", wi
ndowUnit);
    replacementDataSet.addReplacementObject("${rate}", maxNum);

    DatabaseOperation.CLEAN_INSERT.execute(databaseTester.getCon
nection(), replacementDataSet);
```

Cucumber介绍

介绍

BDD 就是 Behavior Driven Development / 行为驱动开发，是一种软件开发流程。

Cucumber 是一个经典的并广为使用的 BDD 框架，可以说是 BDD 框架的先驱。cucumber 移植到java之后被称为 cucumber-jvm。

该文件修订时间： 2017-03-13 05:37:11

我们的目标

目标

引入 BDD 和 cucumber 是希望用 BDD 的方式来解决产品开发测试中的一些问题，主要是打通各个环节：

1. 需求

   需求通常由产品人员提出。

2. 功能

   每个需求展开为若干个功能/feature。

   从产品人员的角度，表明为了满足该需求，需要实现这些功能。

   从开发人员的角度，表明只要实现了这些功能，则视为已经满足该需求。

   可以将 `功能` 视为产品人员和开发人员针对需求达成的共识或者说契约。

   从需求到功能的展开，由开发人员主导，产品人员review，测试人员可以参与（强烈建议参与）。

3. 测试点

   每个功能展开为若干个测试点 / test point。

   从测试人员的角度，表明为了验证该功能可以工作，需要通过这些测试点的检验。

   从开发人员的角度，表明只要通过了这些测试点的检验，则视为功能已经实现。

   可以将 `测试点` 视为测试人员和开发人员针对如何做功能测试达成的共识或者说契约。

   从功能到测试点的展开，由测试人员主导，开发人员提供必要的案例补充，并做review，建议产品人员适当review。

   测试点是可读性的文本描述，不是代码，不涉及到具体编程语言。

4. 测试案例

每个测试点对应到一个测试案例。

每个测试案例都明确描述具体的输入，操作行为，和期待的输出（包括错误和异常）。

测试案例是可执行的代码，实现的是从可读性的文本描述到编程代码的转换。

5. 测试代码

   测试代码是测试案例中眞正执行具体测试功能的代码，需要和被测试对象进行实际交互。

对应关系

在cucumber中,有以下对应关系：

- Feature 对应于 功能
- Scenario/场景 对应于 测试点
- step class 对应于测试案例
- 场景中的每个Step 对应于 测试案例的输入/输出等，同时映射到测试案例中的step方法
- step方法中的具体代码实现 对应于 测试代码

资料收集整理

官方资料

- 官网
- 官网文档
  - Reference参考文档
  - Reference参考文档(中文翻译)
- 官网Blog

教程

- Cucumber java从入门到精通:一个写的不错的个人blog

安装使用

暂时只考虑java下的安装，即cucumber-java。

参考资料

- https://cucumber.io/docs/reference/jvm#java

maven依赖

需要引入cucumber-java的依赖，另外如果测试框架采用的是testng，则需要多一个cucumber-testng：

```
<dependencies>
        <dependency>
            <groupId>info.cukes</groupId>
            <artifactId>cucumber-java</artifactId>
            <version>1.2.4</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>info.cukes</groupId>
            <artifactId>cucumber-testng</artifactId>
            <version>1.2.4</version>
            <scope>test</scope>
        </dependency>
        ......
    </dependencies>
```

为了方便管理，还需要引入spring相关的依赖：

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-spring</artifactId>
    <version>1.2.4</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
```

Jdk8下的问题

cucumber支持JDK8：

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java8</artifactId>
    <version>1.2.4</version>
    <scope>test</scope>
</dependency>
```

容许使用jdk8的lambdas表达式：

```
public class MyStepdefs implements En {
    public MyStepdefs() {
        Given("I have (\\d+) cukes in my belly", (Integer cukes)
 -> {
            System.out.format("Cukes: %n\n", cukes);
        });
    }
}
```

问题

但是cucumber对jdk8的支持有一个重大问题：

不支持**jdk8 U51**之后的新版本！

每次都报错说"Wrong type at constant pool index":

```
Caused by: java.lang.IllegalArgumentException: Wrong type at con
stant pool index
    at sun.reflect.ConstantPool.getMemberRefInfoAt0(Native Metho
d)
    at sun.reflect.ConstantPool.getMemberRefInfoAt(ConstantPool.
java:47)
    at cucumber.runtime.java8.ConstantPoolTypeIntrospector.getTy
peString(ConstantPoolTypeIntrospector.java:37)
    at cucumber.runtime.java8.ConstantPoolTypeIntrospector.getGe
nericTypes(ConstantPoolTypeIntrospector.java:27)
    at cucumber.runtime.java.Java8StepDefinition.<init>(Java8Ste
pDefinition.java:45)
    at cucumber.runtime.java.JavaBackend.addStepDefinition(JavaB
ackend.java:162)
    at cucumber.api.java8.En.Given(En.java:190)
```

这个bug由来已久，从2015年9月就被报告，但是一直没有fix。

> 注： 尝试过jdk u6 / *u7* / u91,都报错，无奈放弃，等待官方修复。

这个bug的相关信息:

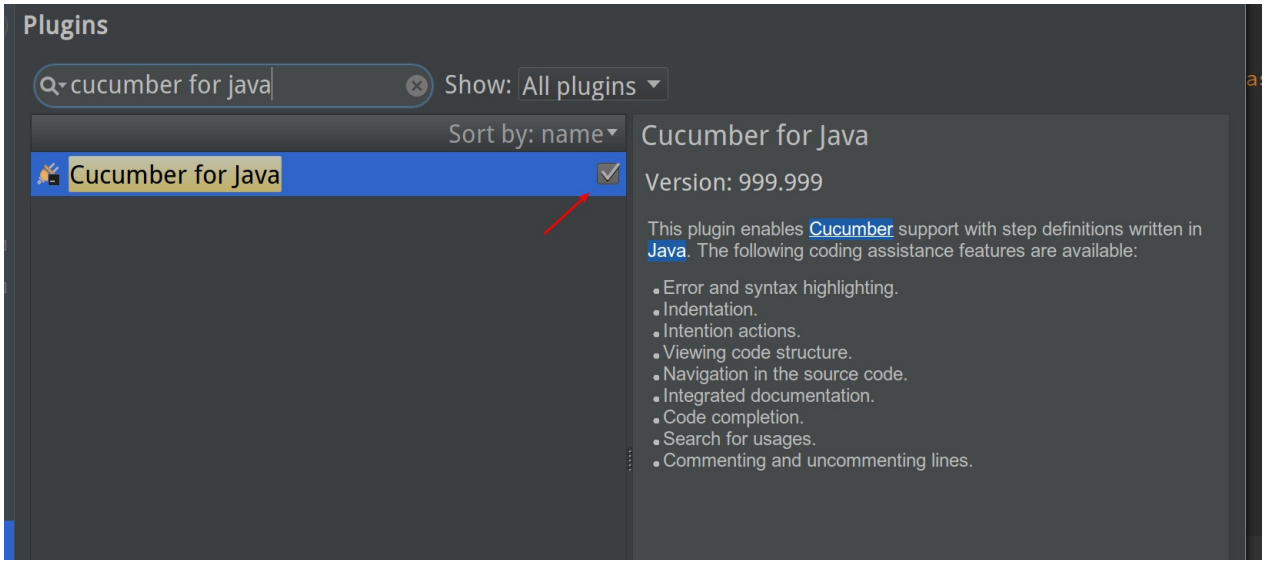
- https://github.com/cucumber/cucumber-jvm/issues/937
- https://github.com/cucumber/cucumber-jvm/issues/912

IntelliJ Idea

> 注: 内容参考 https://www.jetbrains.com/help/idea/2016.1/cucumber.html

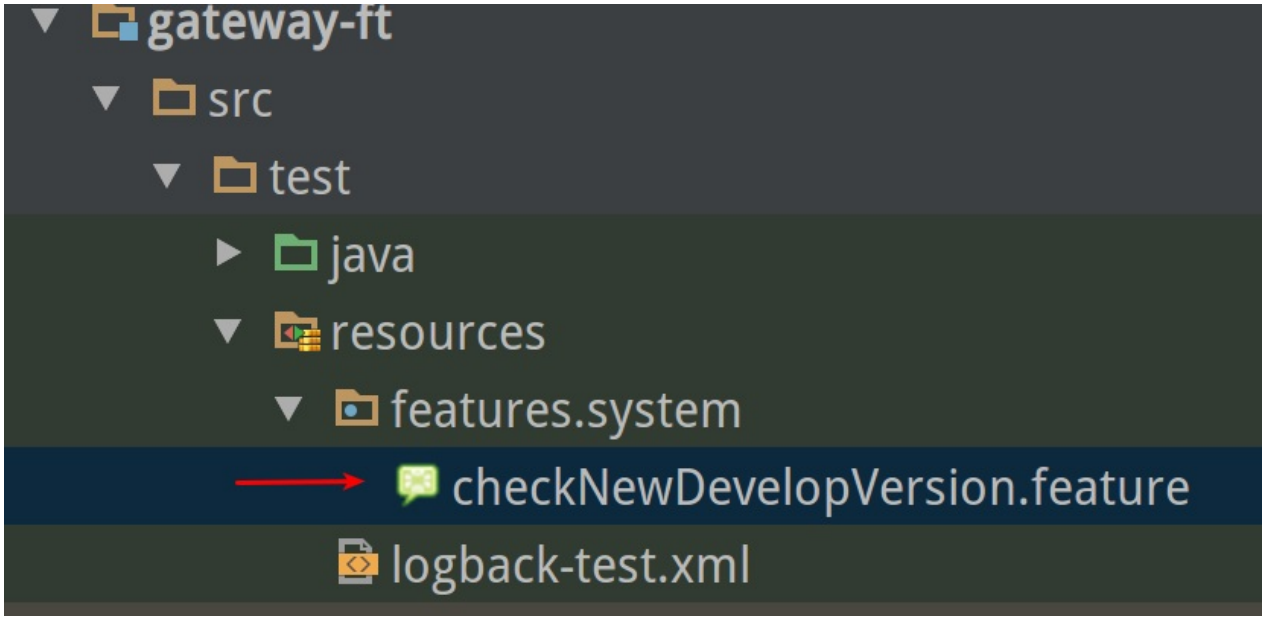
前提

IntelliJ Idea下默认安装有cucumber的插件，如果安装时没有勾选，可以再自行安装。



IntelliJ IDEA 支持 Cucumber for Java， cucumber的feature文件将被标记为

如图：



Cucumber 的支持包括:

- 语法和错误高亮
- "create step definitions/创建步骤定义"的快速修复

  即当feature文件中定义的step没有对应的定义时，可以"alt + enter"然后快速创建

- 在步骤和步骤定义之间导航

  在feature文件中按住"ctl"键，用鼠标点步骤，就可以跳到对应的步骤定义的java文件的具体代码行。

- 可以用英语或者任何其他在 `#language:` 注释中定义的语言来描述步骤
- Run/debug 配置
- 可以运行文件夹中的所有feature，单个feature，或者一个feature中的单独一个步骤。

创建步骤定义

如果 `.feature` 文件引用到一个不存在的步骤， IntelliJ IDEA的代码检验(code inspection)会觉察并高亮这个步骤，并提供一个引导行为(intention action)来帮助创建丢失的步骤定义。

创建一个丢失的步骤定义：

1. 在编辑 `.feature` 文件时，鼠标指到一个步骤定义的引用，IntelliJ IDEA会高亮步骤为未定义，并在tooltip中给出详细信息

2. 按 `Alt+Enter` 来显示 `Create Step Definition` 的intention action

   > 特别提醒：在这个 `Create Step Definition` 的提示出来之后，直接敲回车就可以出来下面的步骤，不要理会这个三角形。因为进去之后反而找不到后面的步骤。

3. 从下拉列表中选择目标步骤定义文件：

   可以从建议列表中选择一个已经存在的步骤定义文件，或者创建一个新的。

   如果想创建一个新的步骤定义文件，请指定名称，类型，和父目录。

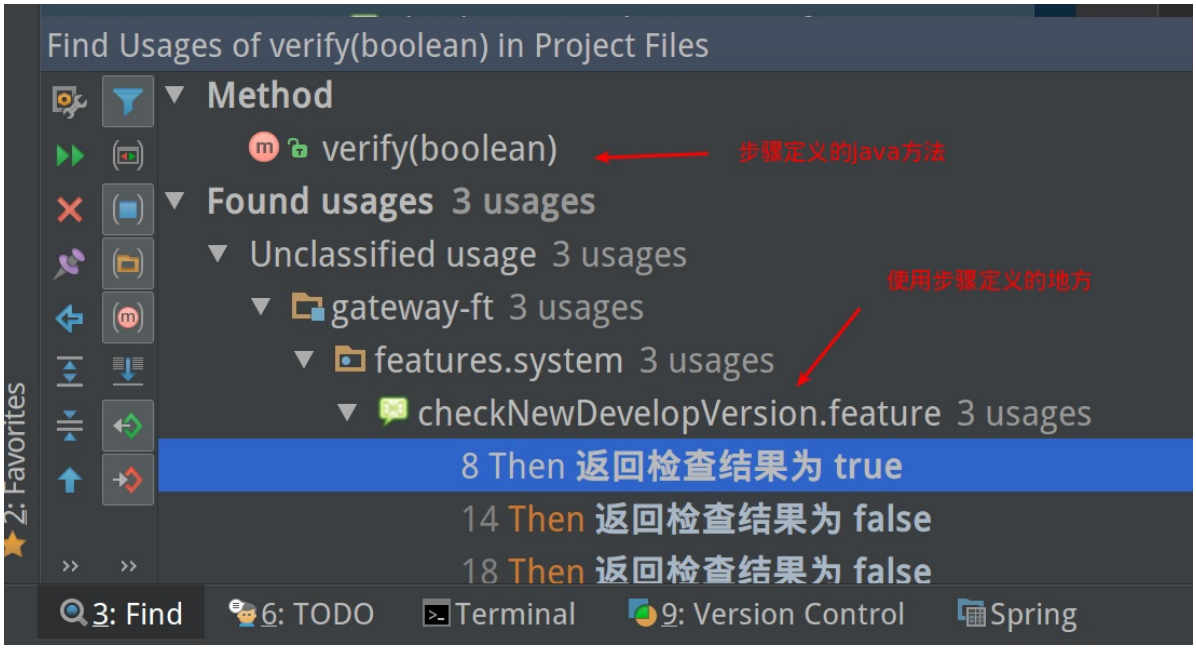
4. 在编辑器打开的被选择的步骤定义文件中，键入要求的代码。

   注意：编辑器会转为模板编辑模式，并用红框显示第一个高亮的输入字段。

TIPs

- 查找使用步骤定义的地方

  和普通java代码类似，使用 `alt + F7` 或者 鼠标右键点击步骤定义的java方法，就可以查找出所有使用这个步骤定义的



- IntelliJ IDEA 会始终关注步骤定义的唯一性。如果有相同名字的步骤定义会高亮显示。

  > 注：有点疑惑，测试中试过没有看到所谓高亮。待确认...

从.feature 文件导航到步骤定义

1. 在编辑器中打开 `.feature` 文件
2. 如下操作中的任意一个
   - 一直按着 `ctrl` 键，将鼠标移动到步骤上面。步骤会变成一个超链接，它的引用信息会在tooltip中显示：

     点击超链接。步骤定义会在编辑器中打开。

- 在主菜单中，选择 Navigate | Declaration
- 直接用快捷键 `Ctrl+B`

在Scenario Outline中创建Examples Table

IntelliJ IDEA 提供对Scenario Outline的支持，可以通过带有占位符的模板来描述多个情景。这些支持包括：

- 关键字的代码完成
- 关键字，占位符和属性的语法高亮
- 代码检测，用于探测找不到的examples，和生成example table stub的快速修改

参考文档

> 注： 内容翻译自cucumber官网 reference文档

Gherkin

Cucumber 执行 .feature 文件，而这些文件包含可执行的规范，用被称为 Gherkin 的语言写成。

Gherkin 是带有一点额外结构的纯文本的英文(或者替他60多种语言). Gherkin 设计为容易被非编程人员学习，但有足够的组织结构来容许简洁的范例描述，以说明大多数实际领域中的业务规则。

这里是Gherkin文档的一个例子：

```
Feature: Refund item

  Scenario: Jeff returns a faulty microwave
    Given Jeff has bought a microwave for $100
    And he has a receipt
    When he returns the microwave
    Then Jeff should be refunded $100
```

在 Gherkin 中, 每个不是空行的行必须以 Gherkin 关键字开头, 然后跟随有任意的文本。主要的关键字有：

- Feature / 特性
- Scenario / 场景
- Given, When, Then, And, But (Steps/步骤)
- Background / 背景
- Scenario Outline / 场景大纲
- Examples / 示例

还有其他一些额外的关键字:

- `"""` (Doc Strings)
- `|` (Data Tables)
- `@` (Tags)
- `#` (Comments)

Feature / 特性

.feature 文件用来描述系统的一个单一特性，或者某个特性的一个独特方面。这仅仅是一个提供软件特性的高级描述的方法，并用于组织相关的场景(scenarios)。

feature有三个基本元素：

1. `Feature:` 关键字
2. name： 名称, 在同一行
3. description：描述, 可选（但是强烈推荐），可以占据多行

Cucumber 并不介意名称或者描述 --- 目的只是简单的提供一个地方可以用来描述特性的重要方面，例如一个简短的解释和一个业务规则列表(常规验收标准).

这里有一个例子：

```
Feature: Refund item

  Sales assistants should be able to refund customers' purchases
.
  This is required by the law, and is also essential in order to
  keep customers happy.

  Rules:
  - Customer must present proof of purchase
  - Purchase must be less than 30 days ago
```

除了 name 和 description， feature还包含一个 Scenarios 或者 Scenario Outlines 列表，还有一个可选的 Background。

Descriptions / 描述

Gherkin文档的某些部分并非以关键字开始。

在 Feature, Scenario, Scenario Outline 或者 Examples 后面的行中，可以写任意内容，只要没有行是以关键字开头。

Scenario / 场景

Scenario 是具体的实例，描述一个业务规则。它由步骤列表组成。

可以有任意多个步骤，但是推荐数量保持在每个场景3-5个步骤。如果太长，他们将丧失作为规范和文档的表单能力。

在作为规范和文档之外，场景也同样是测试。作为一个整体，场景是系统的可执行规范。

场景遵循同样的模式：

- 描述一个初始化上下文
- 描述一个时间
- 描述一个期望的产出

这些是通过步骤来完成。

Steps / 步骤

步骤通常以 Given, When 或 Then 开头。如果有多个 Given 或者 When 步骤连在一起，可以使用 And 或者 But。Cucumber不区分这些关键字，但是选择正确的关键字对于场景整体的可读性很重要。

Given / 假设

Given 步骤用于描述系统的初始化上下文 - 场景的一幕(scene of Scenario)。它通常是某些已经发生在过去的东西。

当cucumber执行 Given 步骤时，它将配置系统到一个定义良好的状态，例如创建并配置对象或者添加数据到测试数据库。

可以有多个 Given 步骤（可以使用 And 或者 But 来变的更可读）

When / 当

When 步骤用来描述一个事件， 或者一个动作。这可以是一个人和系统交互，或者可以是其他系统触发的事件。

强烈推荐每个场景仅有一个单一的 When 步骤。如果感觉不得不添加更多，这通常是应该拆分场景到多个场景的信号。

Then / 那么

Then 步骤用于描述期望的产出，或者结果。

Then 步骤的 步骤定义 应该使用断言来比较实际产出(系统实际行为)和期待产出(步骤所述的系统应有的行为)。

Background / 背景

发现一个feature文件中的所有场景都在重复同样的 Given 步骤。既然它在每个场景

可以将这样的 Given 步骤移动到background中，在第一个场景之前，用一个 Background 块组织他们：

```
Background:
  Given a $100 microwave was sold on 2015-11-03
  And today is 2015-11-18
```

Scenario Outline / 场景大纲

当有复杂业务规则，带有多个输入或者输出，可以最终创建仅仅是值有差别的多个场景。

举个例子(feed planning for cattle and sheep)：

```
Scenario: feeding a small suckler cow
  Given the cow weighs 450 kg
  When we calculate the feeding requirements
  Then the energy should be 26500 MJ
  And the protein should be 215 kg

Scenario: feeding a medium suckler cow
  Given the cow weighs 500 kg
  When we calculate the feeding requirements
  Then the energy should be 29500 MJ
  And the protein should be 245 kg

# 还有两个例子 --- 已经令人厌烦了
```

如果有很多例子，将会很乏味。可以通过使用场景大纲来简化：

```
Scenario Outline: feeding a suckler cow
  Given the cow weighs <weight> kg
  When we calculate the feeding requirements
  Then the energy should be <energy> MJ
  And the protein should be <protein> kg

  Examples:
    | weight | energy | protein |
    |    450 |  26500 |     215 |
    |    500 |  29500 |     245 |
    |    575 |  31500 |     255 |
    |    600 |  37000 |     305 |
```

这更易于阅读。

场景大纲步骤中的变量通过使用 `<` 和 `>` 来标记。

Examples / 示例

场景大纲部分总被带有一个或者多个 Examples / 示例 部分，用于包含一个表格。

表格必须有header 行，对应场景大纲步骤中的变量。

下面的每一行将创建一个新的场景，使用变量的值填充。

场景大纲和UI

使用UI自动化例如 Selenium WebDriver 来做自动化场景大纲被认为是一个坏的实践。

使用场景大纲的唯一的好理由是用来验证业务规则的实现，这些业务规则基于多个输入参数有不同的行为。

通过UI来验证业务规则很慢， 而且如果有错误，很难确定错误在哪里。

场景大纲的自动化代码应该直接和业务规则实现直接通讯，通过的层尽可能的少。这样不仅快，而且容易诊断修复。

步骤参数

在一些案例中，可能想传递一大段文本或者一个数据表格到步骤 --- 而这些有时不适合用在一个单行上。

为此 Gherkin 提供了 Doc Strings / 文档字符串 和 Data Tables / 数据表格.

Doc Strings / 文档字符串

文档字符串方便传递大段文本到步骤定义。语法受python的 Docstring 语法启发。

文本应该在由三个双引号组成的分隔符中：

```
Given a blog post named "Random" with Markdown body
  """
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor
 sit amet,
  consectetur adipiscing elit.
  """
```

在步骤定义中，不需要查找这个文本并在正则表达式中匹配它。它将被自动传递给步骤定义中的最后一个参数。

开始"""前面的空缺不重要，当然通常的实践是在步骤下面缩进两个空格。三个引号内的空缺是有意义的。文档字符串的每行都将对应开始的"""来取出空缺。开始的"""之外的空缺将被保留。

Data Tables / 数据表格

Data Table易于传递值列表到步骤定义：

```
Given the following users exist:
  | name   | email              | twitter         |
  | Aslak  | aslak@cucumber.io  | @aslak_hellesoy |
  | Julien | julien@cucumber.io | @jbpros         |
  | Matt   | matt@cucumber.io   | @mattwynne      |
```

非常类似文档字符串，数据表格将被传递给步骤定义作为最后一个参数。

这个参数的类型是 DataTable。请查看 API 文档来获取如何访问行和列的更多细节。

Tags / 标签

标签是组织场景的一种方法。他们用@做前缀，而且可以放置任何多个标签在 Feature, Scenario, Scenario Outline 或 Examples关键字上。空格在标签中是非法的，但是可以用来分隔他们。

标签可以从父元素中继承。例如，如果你在Feature上放置一个标签，这个feature中的所有场景将得到这个标签。

类似的，如果放置标签在 Scenario Outline 或 Examples 关键字上，所有衍生自实例行的场景将继承这个标签。

可以告诉cucumber仅仅跑带有特定标签的场景，或者忽略代用特定标签的场景。

Cucumber可以在每个场景前后基于场景上的具体标签执行不同操作。

查看 tagged hooks/标签钩子 来得到更多细节。

> 注： 标签钩子 一节目前还是只有一个简单的TODO :)

Comments / 注释

Gherkin 提供很多多个场合可以用来为文档化feature和scennario。最合适的地方是 descriptions (注：见最上面的description一节)。选择好的名字同样有用。

如果这些场合都不适合，可以用一个 `#` 来告诉cucumber这行剩余的部分是注释，并不能执行。

步骤定义

cucumber不知道如何开箱即用的执行场景。它需要步骤定义来将纯文本的gherkin步骤转化为和系统交互的行为。

当cucumber执行场景中的步骤时，它将查找匹配的步骤定义来执行。

步骤定义是带有正则表达式的小段代码。正则表达式用于连接步骤定义到所有匹配的步骤，而代码是cucumber要执行的内容。

为了理解步骤定义如何工作，考虑下列场景：

```
Scenario: Some cukes
  Given I have 48 cukes in my belly
```

步骤的 `I have 48 cukes in my belly` 部分（Given关键字后面的文本）将匹配下面的步骤定义：

```
@Given("I have (\\d+) cukes in my belly")
public void I_have_cukes_in_my_belly(int cukes){
    System.out.format("Cukes: %n\n", cukes);
}
```

当cucumber匹配步骤到一个步骤定义中的正则表达式时，它传递所有捕获组（capture group）的值到步骤定义的参数。

捕获组是字符串(即使他们匹配数字如 `\d+`)。对于静态类型语言，cucumber将自动转换这些字符串到合适的类型。对于动态类型语言，默认不转换，因为他们没有类型信息。

Cucumber不区分这五个步骤关键字 Given， When， Then， And 和 But。

Simple Arguments

TODO

Argument Transformations

TODO

Doc String Argument

TODO

Data Table Argument

TODO

Diff comparison

TODO

Data Table Transformation

TODO

Hooks

TODO

Tagged Hooks

TODO

Command line

TODO

> 注：以上TODO是原文如此，只能等待他们更新内容。

报告

cucumber可以以多个不同格式报告结果，使用 formatter 插件。可用的 formatter 插件有：

- Pretty
- HTML
- JSON
- Progress
- Usage
- JUnit
- Rerun

注意有些cucumber实现可能不提供所有这些formatter插件，而有些实现可能提供额外的。

Pretty / 漂亮

打印 Gherkin 原代码到 STDOUT （标准输出），带有额外的配色和错误栈轨迹（stack trace）。

HTML

生成HTML文件，使用发布。

JSON

生成JSON文件，使用后处理来生成自定义报告。

Progress

报告打印结果到 STDOUT， 每次一个字符。看上去像这样：

```
....F--U.......
```

Usage

打印统计到STDOUT。程序员会发现它对发现慢或者未使用的步骤定义会有帮助。

Junit

生成类似 Apache Ant 的 junitreport 任务的xml文件。这个xml格式可以被大多数持续继承服务器理解，将使用它来生成可视化的报告。

Rerun

retun 报告是一个列举失败场景位置的文件。这个可以用于后续的cucumber运行：

```
cucumber @rerun.txt
```

当修复有误场景时有用，因为仅有上次运行失败的场景将会在此运行。这可以减少修复bug的花费时间，当运行所有场景费时时。

如果要在同一个cucumber运行中寻找方法来自动重运行非确定性的，或者闪烁的场景，那么return报告无法提供帮助。rerun用于场景失败可以确定的工作流，而且可以修改场景或者系统来让他们在每次cucumber运行中通过。

非确定性场景是cucumber无法解决的深层问题。只能自行检测和定位非确定性的根本原因。

报告附件

文本，图片甚至视频可以内嵌在特定的报告中，通过步骤定义和钩子的API。

这可以更简便的诊断失败。某些报告会忽略内嵌数据而有些会包含他们。

> 注： 好像用不到的样子，跳过先不翻译。

集成

Spring集成

关于如何与spring集成，cucumber上没有资料，只是简单的说要用cucumber-spring：

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-spring</artifactId>
    <version>1.2.4</version>
    <scope>test</scope>
</dependency>
```

> 注: 见 https://cucumber.io/docs/reference/java-di#spring

依赖注入

如果编程语言是java，将用plain old java class来编写glue代码(步骤定义和Hooks)。

在每个场景前，cucumber将为每个 glue 代码类创建新的实例。如果所有 glue 代码类都有空构造函数，则不需要担心其他。否则，大多数项目将从依赖注入模块中收益，来更好的管理代码和在步骤定义之间分享状态。

可用的依赖注入模块是：

- PicoContainer (如果应用没有使用其他依赖注入，则推荐用这个)
- Spring
- Guice
- OpenEJB
- Weld
- Needle

类SpringFactory

介绍

> 注： 以下内容来自类SpringFactory的 javadoc

基于spring的ObjectFactory实现。

使用TestContextManager来管理spring context。

通过 @ContextConfiguration 或 @ContextHierarcy 注解来配置。

在步骤定义类上至少需要一个 @ContextConfiguration 或 @ContextHierarcy 注解。如果有超过一个步骤定义类有这样的注解，这些不同步骤定义类上的注解必须相等。如果没有步骤定义类有@ContextConfiguration 或 @ContextHierarcy 注解注解，则将尝试从classpath下装载 cucumber.xml 。

带有 @ContextConfiguration 或 @ContextHierarcy 注解的步骤定义类，可以也有 @WebAppConfiguration 或 @DirtiesContext 注解。

步骤定义添加到 TestContextManagers context 并为每个场景重新装载(reload)。

application bean 可以在步骤定义中通过自动装配(autowiring)访问。

集成testng

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-testng</artifactId>
    <version>1.2.4</version>
    <scope>test</scope>
</dependency>
```

其他参考资料

- https://github.com/talios/cucumber-testng-factory

实践

使用中文

cucumber支持使用40多种语言来描述feature文件，也包括中文。

对照表

中英文关键字的对照表

英文关键字	中文关键字
feature	"功能"
background	"背景"
scenario	"场景", "剧本"
scenario_outline	"场景大纲", "剧本大纲"
examples	"例子"
given	"* ", "假如", "假设", "假定"
when	"* ", "当"
then	"* ", "那么"
and	"* ", "而且", "并且", "同时"
but	"* ", "但是"
given (code)	"假如", "假设", "假定"
when (code)	"当"
then (code)	"那么"
and (code)	"而且", "并且", "同时"
but (code)	"但是"

范例

以下是用英文编写的feature：

```
Feature: 检查是否有新的开发版本

  Scenario: 有新的开发版本
    Given 有版本文件 a-10001.akg
    And 有版本文件 a-10002.akg
    And 有版本文件 a-10003.akg
    When 客户端检查新版本而它的版本号为10002
    Then 返回检查结果为 true

  Scenario: 没有新的开发版本
    Given 有版本文件 a-10001.akg
    And 有版本文件 a-10002.akg
    When 客户端检查新版本而它的版本号为10002
    Then 返回检查结果为 false

  Scenario: 没有新的开发版本(文件列表为空)
    When 客户端检查新版本而它的版本号为10001
    Then 返回检查结果为 false
```

下面是它的中文版本，注意关键字都变成中文了：

```
#language: zh-CN
功能: 检查是否有新的开发版本

  场景: 有新的开发版本
    假设 有版本文件 a-10001.akg
    同时 有版本文件 a-10002.akg
    同时 有版本文件 a-10003.akg
    当 客户端检查新版本而它的版本号为10002
    那么 返回检查结果为 true

  场景: 没有新的开发版本
    假设 有版本文件 a-10001.akg
    同时 有版本文件 a-10002.akg
    当 客户端检查新版本而它的版本号为10002
    那么 返回检查结果为 false

  场景: 没有新的开发版本(文件列表为空)
    当 客户端检查新版本而它的版本号为10001
    那么 返回检查结果为 false
```

为了支持中文关键字，需要注明语言为zh-CN，所以需要在前面加入：

```
#language: zh-CN
```

传递参数给步骤定义

简单参数的传递

参数类型	正则表达式
整型	`(\\d+)`
字符串	`((\S*)`
布尔值	`(true OR false)`

传递整型

对于feature中定义的整型内容：

```
When 客户端检查新版本而它的版本号为10002
```

可以用 `(\\d+)` 来捕获：

```
@When("^客户端检查新版本而它的版本号为(\\d+)$")
    public void checkWith(long timestamp) {}
```

传递字符串

对于feature中定义的字符串内容：

```
Given 有版本文件 a-10001.akg
```

可以用 `((\S*)` 来捕获：

```
@Given("^有版本文件 (\\S*)$")
public void prepareExistApkFile(String apkFile) {}
```

传递布尔值

对于feature中定义的布尔值内容：

```
Then 返回检查结果为 false
```

可以用 `(true|false)` 来捕获：

```
@Then("^返回检查结果为 (true|false)$")
public void verify(boolean expectedHasNewVersion) {}
```

复杂类型

传递一维列表

cucumber 支持 One-dimensional lists / 一维列表。

> 注：内容来自 https://cucumber.io/docs/reference/jvm#java

- 最简单的方法

  传递一个 List 到步骤定义的最简单的方式是使用逗号：

  ```
  Given the following animals: cow, horse, sheep
  ```

  简单将参数定义为List：

  ```
  @Given("the following animals: (.*)")
  public void the_following_animals(List<String> animals) {
  }
  ```

  使用 @Delimiter 注解可以定义","/逗号之外的分隔符。

- 使用Data Table

  如果更喜欢使用Data Table来定义列表，也可以这样做：

```
Given the following animals:
  | cow   |
  | horse |
  | sheep |
```

简单定义参数为List(但是不要在正则表达式中定义capture group):

```
@Given("the following animals:")
public void the_following_animals(List<String> animals) {
}
```

在这个案例中，在调用步骤定义前，DataTable 被Cucumber(使用DataTable.asList(String.class))自动转为 List。