Java JUnit 单元测试小结

永顺 发布于 2016-08-26

测试类型

单元测试(Unit test)

单元测试关注单一的类. 它们存在的目的是检查这个类中的代码是否按照期望正确运行.

集成测试(Integration test)

顾名思义, 集成测试是检查开发的模块和其他模块整合时是否正常工作.
虽然集成测试的代码影响范围比单元测试要广, 但是集成测试和单元测试一样, 也是针对于开发者而言的.

端到端测试(End-to-End test)

端到端测试是将整个系统作为一个整体, 然后从用户的角度进行测试的.
端到端测试的目的是测试系统在实际使用的是否正常的, 因此通常来说是不需要测试替身的(Test Double)

单元测试基本概念

什么是单元测试

单元测试的目的: 测试当前所写的代码是否是正确的, 例如输入一组数据, 会输出期望的数据; 输入错误数据, 会产生错误异常等.
在单元测试中, 我们需要保证被测系统是独立的(SUT 没有任何的 DOC), 即当被测系统通过测试时, 那么它在任何环境下都是能够正常工作的. 编写单元测试时，仅仅需要关注单个类就可以了. 而不需要关注例如数据库服务, Web 服务等组件.

被测系统

被测系统(System under test, SUT)表示正在被测试的系统, 目的是测试系统能否正确操作.
根据测试类型的不同, SUT 指代的内容也不同, 例如 SUT 可以是一个类甚至是一整个系统.

测试依赖组件(DOC)

被测系统所依赖的组件, 例如进程 UserService 的单元测试时, UserService 会依赖 UserDao, 因此 UserDao 就是 DOC.

测试替身(Test Double)

一个实际的系统会依赖多个外部对象, 但是在进行单元测试时, 我们会用一些功能较为简单的并且其行为和实际对象类似的假对象来作为 SUT 的依赖对象, 以此来降低单元测试的复杂性和可实现性. 在这里, 这些假对象就被称为 **测试替身(Test Double)**.
测试替身有如下 5 种类型:

- Test stub, 为 SUT 提供数据的假对象.
  我们举一个例子来展示什么是 Test stub.

假设我们的一个模块需要从 HTTP 接口中获取商品价格数据, 这个获取数据的接口被封装为 getPrice 方法. 在对这个模块进行测试时, 我们显然不太可能专门开一个 HTTP 服务器来提供此接口, 而是提供一个带有 getPrice 方法的假对象, 从这个假对象中获取数据.
在这个例子中, 提供数据的假对象就叫做 Test stub.

- Fake object
  实现了简单功能的一个假对象. Fake object 和 Test stub 的主要区别就是 Test stub 侧重于用于提供数据的假对象, 而 Fake object 没有这层含义.

使用 Fake object 的最主要的原因就是在测试时某些组件不可用或运行速度太慢, 因而使用 Fake object 来代替它们.

- Mock object
  用于模拟实际的对象, 并且能够校验对这个 Mock object 的方法调用是否符合预期.

实际上, Mock object 是 Test stub 或 Fake object 一种, 但是 Mock object 有 Test stub/Fake object 没有的特性, Mock object 可以很灵活地配置所调用的方法所产生的行为, 并且它可以追踪方法调用, 例如一个 Mock Object 方法调用时传递了哪些参数, 方法调用了几次等.

- Dummy object: 在测试中并不使用的, 但是为了测试代码能够正常编译/运行而添加的对象. 例如我们调用一个 Test Double 对象的一个方法, 这个方法需要传递几个参数, 但是其中某个参数无论是什么值都不会影响测试的结果, 那么这个参数就是一个 Dummy object.
  Dummy object 可以是一个空引用, 一个空对象或者是一个常量等.

简单的说, Dummy object 就是那些没有使用到的, 仅仅是为了填充参数列表的对象.

- Test Spy
  可以包装一个真实的 Java 对象, 并返回一个包装后的新对象. 若没有特别配置的话, 对这个新对象的所有方法调用, 都会委派给实际的 Java 对象.

mock 和 spy 的区别是: mock 是无中生有地生出一个完全虚拟的对象, 它的所有方法都是虚拟的; 而 spy 是在现有类的基础上包装了一个对象, 即如果我们没有重写 spy 的方法, 那么这些方法的实现其实都是调用的被包装的对象的方法.

Test fixture

所谓 test fixture, 就是运行测试程序所需要的先决条件(precondition). 即对被测对象进行测试时锁需要的一切东西(The test fixture is everything we need to have in place to exercise the SUT). 这个 **东西** 不单单指的是数据, 同时包括对被测对象的配置, 被测对象所需要的依赖对象等.
JUnit4 之前是通过 setUp, TearDown 方法完成, 在 JUnit4这, 我们可以使用@Before 代替 setUp 方法, @After 代替 tearDown 方法.

> 注意, @Before 在每个测试方法运行前都会被调用, @After 在每个测试方法运行后都会被调用.

因为 @Before 和 @After 会在每个测试方法前后都会被调用, 而有时我们仅仅需要在测试前进行一次初始化, 这样的情况下, 可以使用@BeforeClass 和@AfterClass 注解.

测试用例(Test case)

在 JUnit 3中, 测试方法都必须以 test 为前缀, 且必须是 public void 的, JUnit 4之后, 就没有这个限制了, 只要在每个测试方法标注 @Test 注解, 方法签名可以是任意的.

测试套件

通过 TestSuit 对象将多个测试用例组装成一个测试套件, 测试套件批量运行.
通过@RunWith 和@SuteClass 两个注解, 我们可以创建一个测试套件. 通过@RunWith 指定一个特殊的运行器, 几 Suite.class 套件运行器, 并通过@SuiteClasses 注解, 将需要进行测试的类列表作作为参数传入.

JUnit4

HelloWorld 例子

我们已一个简单的例子来快速展示 JUnit4 的基本用法.
首先新建一个名为 JUniTest 的 Maven 工程, 然后添加依赖:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

接着编写测试套件:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestJunit {
    @Test
    public void testingCrunchifyAddition() {
        assertEquals("Here is test for Addition Result: ", 30, addition(27, 3));
    }

    @Test
    public void testingHelloWorld() {
        assertEquals("Here is test for Hello World String: ", "Hello + World", helloWorld());
    }

    public int addition(int x, int y) {
        return x + y;
    }

    public String helloWorld() {
        String helloWorld = "Hello +" + " World";
        return helloWorld;
    }
}
```

随后使用测试用例:

```
public class App {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestJunit.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        if (result.wasSuccessful()) {
            System.out.println("Both Tests finished successfully...");
        }
    }
}
```

这就是一个完整的 JUnit 测试例子了.

定义测试

一个 JUnit 测试是一个在专用于测试的类中的一个方法, 并且这个方法被 **@org.junit.Test** 注解标注. 例如:

```
public class TestJunit {
    @Test
    public void testingCrunchifyAddition() {
        assertEquals("Here is test for Addition Result: ", 30, addition(27, 3));
    }
    ...
}
```

JUnit4 生命周期

JUnit4测试用例的完整的生命周期要经历如下几个阶段:

- 类级初始化资源处理
- 方法级初始化资源处理
- 执行测试用例中的方法
- 方法级销毁资源处理
- 类级销毁资源处理

其中, 类级初始化和销毁资源处理在每一个测试用例类这仅仅执行一次, 方法级初始化, 销毁资源处理方法在执行测试用例这的每个测试方法中都会被执行一次.

JUnit4 注解

- @Test (expected = Exception.class) 表示预期会抛出Exception.class 的异常
- @Ignore 含义是"某些方法尚未完成，暂不参与此次测试"。这样的话测试结果就会提示你有几个测试被忽略，而不是失败。一旦你完成了相应函数，只需要把@Ignore注解删去，就可以进行正常的测试。
- @Test(timeout=100) 表示预期方法执行不会超过 100 毫秒，控制死循环
- @Before 表示该方法在每一个测试方法之前运行，可以使用该方法进行初始化之类的操作
- @After 表示该方法在每一个测试方法之后运行，可以使用该方法进行释放资源，回收内存之类的操
- @BeforeClass 表示该方法只执行一次，并且在所有方法之前执行。一般可以使用该方法进行数据库连接操作，注意该注解运用在静态方法。
- @AfterClass 表示该方法只执行一次，并且在所有方法之后执行。一般可以使用该方法进行数据库连接关闭操作，注意该注解运用在静态方法。

@Test 注解

被@Test 标注的方法就是执行测试用例的测试方法, 例如:

```
public class TestJunit {
    @Test
    public void  myTest() {
        assertEquals("Here is test for Addition Result: ", 30, addition(27, 3));
    }
}
```

方法myTest 被注解@Test 标注, 表示这个方法是一个测试方法, 当运行测试用例时, 会自动调用这个方法 .

@BeforeClass , @AfterClass, @Before, @After

使用@BeforeClass 和 @AfterClass 两个注解标注的方法会在所有测试方法执行前后各执行一次
使用@Before 和 @After 两个注解标注的方法会在每个测试方法执行前后都执行一次.

TestSuite

如果有多个测试类, 可以合并成一个测试套件进行测试, 运行一个 Test Suite, 那么就会运行在这个 Test Suite 中的所用的测试.
例如:

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

@RunWith( Suite.class )
@SuiteClasses( { JUnitTest1.class, JUnitTest2.class } )
public class AllTests {

}
```

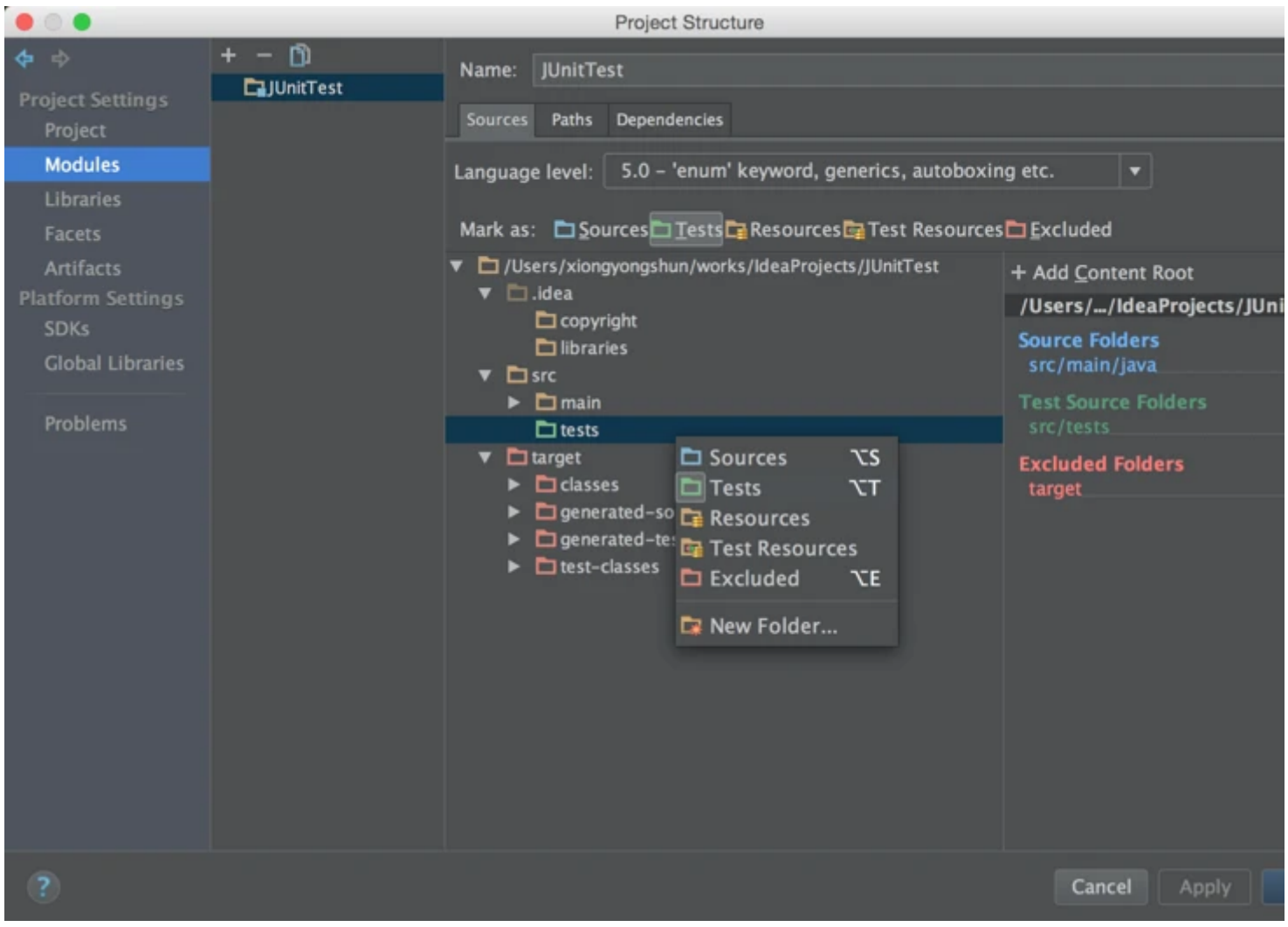
在这个例子中, 我们定义了一个 Test Suite, 这个 Test Suite 包含了两个测试类: JUnitTest1 和 JUnitTest2, 因此运行 这个 Test Suite 时, 就会自动运行这两个测试类了.

在 IntelliJ IDEA 中使用 JUnit 4

创建一个名为 JUnitTest 的 HelloWorld 工程, 添加依赖:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
    </dependency>
</dependencies>
```

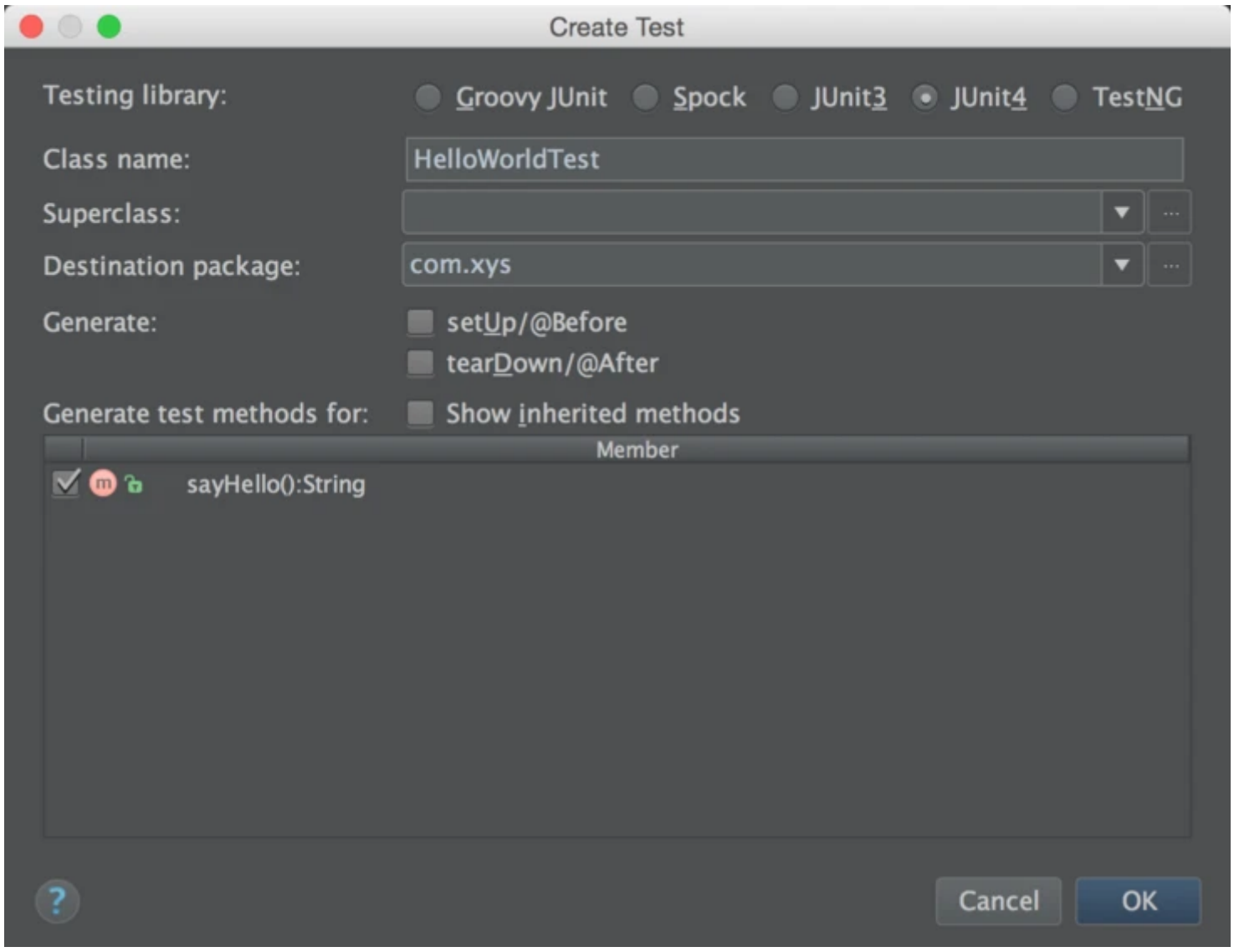
然后创建 src/tests, 在File -> Project Structure -> Modules -> Sources 中, 右键选中 tests, 将其设置为 Test, 此时 tests 目录就变为绿色:

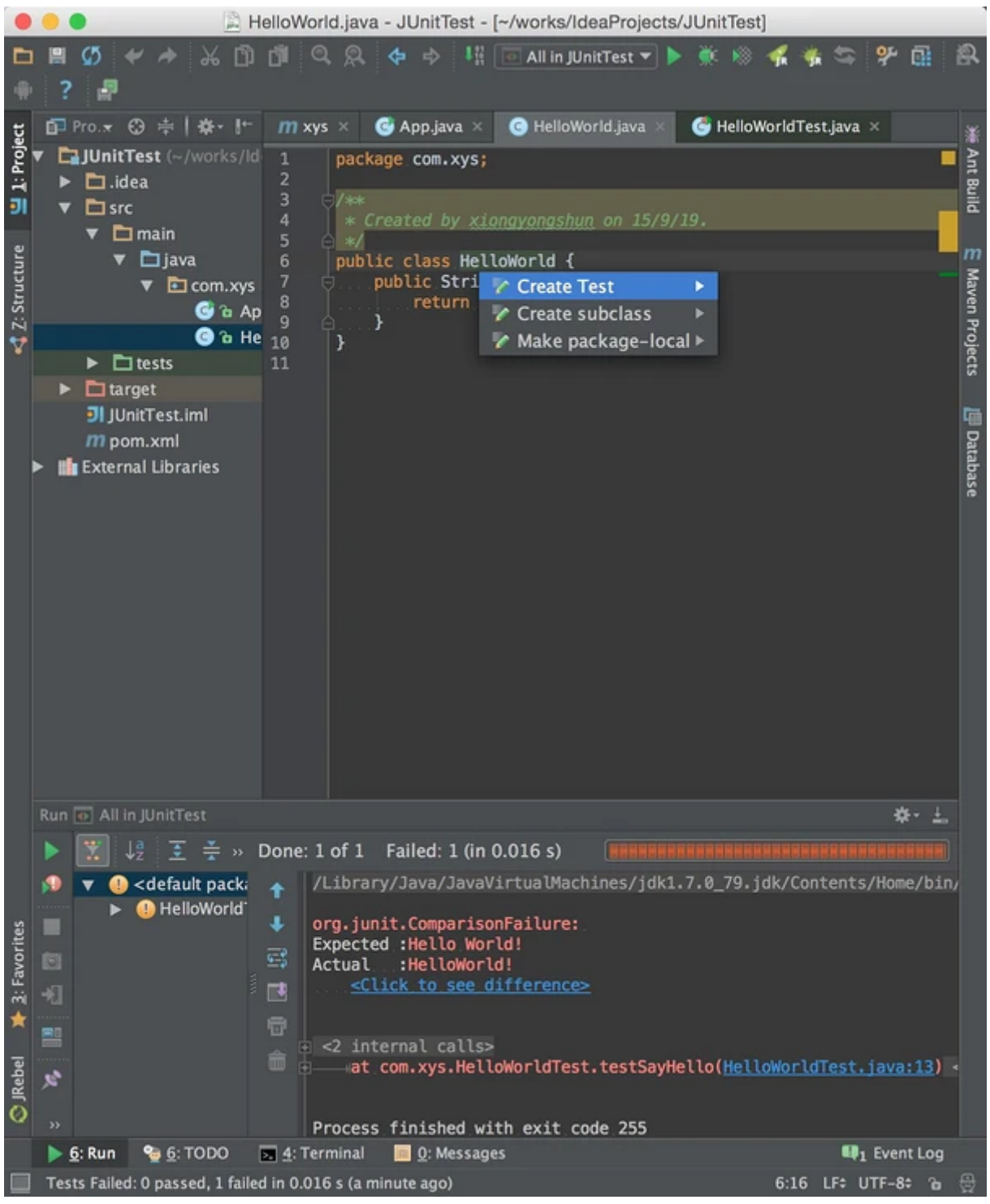


然后创建需要进行测试的类:

```
public class HelloWorld {
    public String sayHello() {
        return "Hello World!";
    }
}
```

在 HelloWorld 类名上按下 **alt + enter** 后, 就可以自动生成测试类了:





IntelliJ 在生tests/ 目录下生成了一个测试类, 我们可以添加自动测试内容:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class HelloWorldTest {

    @Test
    public void testSayHello() throws Exception {
        HelloWorld helloWorld = new HelloWorld();
        assertEquals(helloWorld.sayHello(), "Hello World!");
    }
}
```


本文标题为: Java JUnit 单元测试小结
本文链接为: https://segmentfault.com/a/11...

junit java

阅读 21.1k • 更新于 2017-04-07

永顺

好饿好饿好饿, 我真的好饿.

5.1k 声望 **1.1k** 粉丝

关注作者