深入学习Java单元测试(Junit+Mock+代码覆盖率)

在做单元测试时，代码覆盖率常常被拿来作为衡量测试好坏的指标，甚至，用代码覆盖率来考核测试任务完成情况，比如，代码覆盖率必须达到80%或 90%。下面我们就来详细学习下java单元测试吧

前言

单元测试是编写测试代码，用来检测特定的、明确的、细颗粒的功能。单元测试并不一定保证程序功能是正确的，更不保证整体业务是准备的。

单元测试不仅仅用来保证当前代码的正确性，更重要的是用来保证代码修复、改进或重构之后的正确性。

一般来说，单元测试任务包括

- 1.接口功能测试：用来保证接口功能的正确性。
- 2.局部数据结构测试（不常用）：用来保证接口中的数据结构是正确的
    - 1.比如变量有无初始值
    - 2.变量是否溢出
- 3.边界条件测试
    - 1.变量没有赋值（即为NULL）
    - 2.变量是数值（或字符)
        - 1.主要边界：最小值，最大值，无穷大（对于DOUBLE等）
        - 2.溢出边界（期望异常或拒绝服务）：最小值-1，最大值+1
        - 3.临近边界：最小值+1，最大值-1
    - 3.变量是字符串
        - 1.引用“字符变量”的边界
        - 2.空字符串
        - 3.对字符串长度应用“数值变量”的边界
    - 4.变量是集合
        - 1.空集合
        - 2.对集合的大小应用“数值变量”的边界
        - 3.调整次序：升序、降序
    - 5.变量有规律
        - 1.比如对于Math.sqrt，给出n^2-1，和n^2+1的边界
- 4.所有独立执行通路测试：保证每一条代码，每个分支都经过测试
    - 1.代码覆盖率
        - 1.语句覆盖：保证每一个语句都执行到了
        - 2.判定覆盖（分支覆盖）：保证每一个分支都执行到
        - 3.条件覆盖：保证每一个条件都覆盖到true和false（即if、while中的条件语句）
        - 4.路径覆盖：保证每一个路径都覆盖到
    - 2.相关软件
        - 1.Cobertura：语句覆盖
        - 2.Emma: Eclipse插件Eclemma
- 5.各条错误处理通路测试：保证每一个异常都经过测试

JUNIT

JUnit是Java单元测试框架，已经在Eclipse中默认安装。目前主流的有JUnit3和JUnit4。JUnit3中，测试用例需要继承TestCase类。JUnit4中，测试用例无需继承TestCase类，只需要使用@Test等注解。

Junit3

先看一个Junit3的样例

```
// 测试java.lang.Math
// 必须继承TestCase
public class Junit3TestCase extends TestCase {
public Junit3TestCase() {
super();
}
// 传入测试用例名称
public Junit3TestCase(String name) {
super(name);
}
// 在每个Test运行之前运行
@Override
protected void setUp() throws Exception {
System.out.println("Set up");
}
// 测试方法。
// 方法名称必须以test开头，没有参数，无返回值，是公开的，可以抛出异常
```

```
// 也即类似public void testXXX() throws Exception {}
public void testMathPow() {
System.out.println("Test Math.pow");
Assert.assertEquals(4.0, Math.pow(2.0, 2.0));
}
public void testMathMin() {
System.out.println("Test Math.min");
Assert.assertEquals(2.0, Math.min(2.0, 4.0));
}
// 在每个Test运行之后运行
@Override
protected void tearDown() throws Exception {
System.out.println("Tear down");
}
}
```

如果采用默认的TestSuite，则测试方法必须是public void testXXX() [throws Exception] {}的形式，并且不能存在依赖关系，因为测试方法的调用顺序是不可预知的。

上例执行后，控制台会输出

```
Set up
Test Math.pow
Tear down
Set up
Test Math.min
Tear down
```

从中，可以猜测到，对于每个测试方法，调用的形式是：

```
testCase.setUp();
testCase.testXXX();
testCase.tearDown();
```

运行测试方法

在Eclipse中，可以直接在类名或测试方法上右击，在弹出的右击菜单中选择Run As -> JUnit Test。

在Mvn中，可以直接通过mvn test命令运行测试用例。

也可以通过Java方式调用，创建一个TestCase实例，然后重载runTest()方法，在其方法内调用测试方法（可以多个）。

```
TestCase test = new Junit3TestCase("mathPow") {
// 重载
protected void runTest() throws Throwable {
testMathPow();
};
};
test.run();
```

更加便捷地，可以在创建TestCase实例时直接传入测试方法名称，JUnit会自动调用此测试方法，如

```
TestCase test = new Junit3TestCase("testMathPow");
test.run();
```

Junit TestSuite

TestSuite是测试用例套件，能够运行过个测试方法。如果不指定TestSuite，会创建一个默认的TestSuite。默认TestSuite会扫描当前内中的所有测试方法，然后运行。
如果不想采用默认的TestSuite，则可以自定义TestSuite。在TestCase中，可以通过静态方法suite()返回自定义的suite。

```
import junit.framework.Assert;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
public class Junit3TestCase extends TestCase {
//...
public static Test suite() {
System.out.println("create suite");
TestSuite suite = new TestSuite();
suite.addTest(new Junit3TestCase("testMathPow"));
return suite;
}
}
```

允许上述方法，控制台输出

```
写道
create suite
```

```
Set up
Test Math.pow
Tear down
```

并且只运行了testMathPow测试方法，而没有运行testMathMin测试方法。通过显式指定测试方法，可以控制测试执行的顺序。

也可以通过Java的方式创建TestSuite，然后调用TestCase，如

```
// 先创建TestSuite，再添加测试方法
TestSuite testSuite = new TestSuite();
testSuite.addTest(new Junit3TestCase("testMathPow"));
// 或者 传入Class，TestSuite会扫描其中的测试方法。
TestSuite testSuite = new TestSuite(Junit3TestCase.class,Junit3TestCase2.class,Junit3TestCase3.class);
// 运行testSuite
TestResult testResult = new TestResult();
testSuite.run(testResult);
```

testResult中保存了很多测试数据，包括运行测试方法数目(runCount)等。

JUnit4

与JUnit3不同，JUnit4通过注解的方式来识别测试方法。目前支持的主要注解有：

- @BeforeClass 全局只会执行一次，而且是第一个运行
- @Before 在测试方法运行之前运行
- @Test 测试方法
- @After 在测试方法运行之后允许
- @AfterClass 全局只会执行一次，而且是最后一个运行
- @Ignore 忽略此方法

下面举一个样例：

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Assert;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;
public class Junit4TestCase {
@BeforeClass
public static void setUpBeforeClass() {
System.out.println("Set up before class");
}
@Before
public void setUp() throws Exception {
System.out.println("Set up");
}
@Test
public void testMathPow() {
System.out.println("Test Math.pow");
Assert.assertEquals(4.0, Math.pow(2.0, 2.0), 0.0);
}
@Test
public void testMathMin() {
System.out.println("Test Math.min");
Assert.assertEquals(2.0, Math.min(2.0, 4.0), 0.0);
}
// 期望此方法抛出NullPointerException异常
@Test(expected = NullPointerException.class)
public void testException() {
System.out.println("Test exception");
Object obj = null;
obj.toString();
}
// 忽略此测试方法
@Ignore
@Test
public void testMathMax() {
Assert.fail("没有实现");
}
// 使用“假设”来忽略测试方法
@Test
public void testAssume(){
System.out.println("Test assume");
// 当假设失败时，则会停止运行，但这并不会意味测试方法失败。
Assume.assumeTrue(false);
Assert.fail("没有实现");
```

```
}
@After
public void tearDown() throws Exception {
System.out.println("Tear down");
}
@AfterClass
public static void tearDownAfterClass() {
System.out.println("Tear down After class");
}
}
```

如果细心的话，会发现Junit3的package是junit.framework，而Junit4是org.junit。
执行此用例后，控制台会输出

```
写道
Set up before class
Set up
Test Math.pow
Tear down
Set up
Test Math.min
Tear down
Set up
Test exception
Tear down
Set up
Test assume
Tear down
Tear down After class
```

可以看到，执行次序是@BeforeClass -> @Before -> @Test -> @After -> @Before -> @Test -> @After -> @AfterClass。
@Ignore会被忽略。

运行测试方法

与Junit3类似，可以在Eclipse中运行，也可以通过mvn test命令运行。

Assert

Junit3和Junit4都提供了一个Assert类（虽然package不同，但是大致差不多）。Assert类中定义了很多静态方法来进行断言。列表如下：

- assertTrue(String message, boolean condition) 要求condition == true
- assertFalse(String message, boolean condition) 要求condition == false
- fail(String message) 必然失败，同样要求代码不可达
- assertEquals(String message, XXX expected,XXX actual) 要求expected.equals(actual)
- assertArrayEquals(String message, XXX[] expecteds,XXX [] actuals) 要求expected.equalsArray(actual)
- assertNotNull(String message, Object object) 要求object!=null
- assertNull(String message, Object object) 要求object==null
- assertSame(String message, Object expected, Object actual) 要求expected == actual
- assertNotSame(String message, Object unexpected,Object actual) 要求expected != actual
- assertThat(String reason, T actual, Matcher matcher) 要求matcher.matches(actual) == true

Mock/Stub

Mock和Stub是两种测试代码功能的方法。Mock测重于对功能的模拟。Stub测重于对功能的测试重现。比如对于List接口，Mock会直接对List进行模拟，而Stub会新建一个实现了List的TestList，在其中编写测试的代码。

强烈建议优先选择Mock方式，因为Mock方式下，模拟代码与测试代码放在一起，易读性好，而且扩展性、灵活性都比Stub好。

比较流行的Mock有：

JMock

EasyMock

Mockito

powermock

其中EasyMock和Mockito对于Java接口使用接口代理的方式来模拟，对于Java类使用继承的方式来模拟（也即会创建一个新的Class类）。Mockito支持spy方式，可以对实例进行模拟。但它们都不能对静态方法和final类进行模拟，powermock通过修改字节码来支持了此功能。

EasyMock

EasyMock把测试过程分为三步：录制、运行测试代码、验证期望。

录制过程大概就是：期望method(params)执行times次（默认一次），返回result（可选），抛出exception异常（可选）。

验证期望过程将会检查方法的调用次数。

一个简单的样例是：

```
@Test
public void testListInEasyMock() {
List list = EasyMock.createMock(List.class);
// 录制过程
// 期望方法list.set(0,1)执行2次，返回null，不抛出异常
expect1: EasyMock.expect(list.set(0, 1)).andReturn(null).times(2);
// 期望方法list.set(0,1)执行1次，返回null，不抛出异常
expect2: EasyMock.expect(list.set(0, 1)).andReturn(1);
// 执行测试代码
EasyMock.replay(list);
// 执行list.set(0,1)，匹配expect1期望，会返回null
Assert.assertNull(list.set(0, 1));
// 执行list.set(0,1)，匹配expect1（因为expect1期望执行此方法2次），会返回null
Assert.assertNull(list.set(0, 1));
// 执行list.set(0,1)，匹配expect2，会返回1
Assert.assertEquals(1, list.set(0, 1));
// 验证期望
EasyMock.verify(list);
}
```

EasyMock还支持严格的检查，要求执行的方法次序与期望的完全一致。

Mockito

Mockito是Google Code上的一个开源项目，Api相对于EasyMock更好友好。与EasyMock不同的是，Mockito没有录制过程，只需要在“运行测试代码”之前对接口进行Stub，也即设置方法的返回值或抛出的异常，然后直接运行测试代码，运行期间调用Mock的方法，会返回预先设置的返回值或抛出异常，最后再对测试代码进行验证。

官方提供了很多样例，基本上包括了所有功能，可以去看看。

这里从官方样例中摘录几个典型的：

验证调用行为

```
import static org.mockito.Mockito.*;
//创建Mock
List mockedList = mock(List.class);
//使用Mock对象
mockedList.add("one");
mockedList.clear();
//验证行为
verify(mockedList).add("one");
verify(mockedList).clear();
```

对Mock对象进行Stub

```
//也可以Mock具体的类，而不仅仅是接口
LinkedList mockedList = mock(LinkedList.class);
//Stub
when(mockedList.get(0)).thenReturn("first"); // 设置返回值
when(mockedList.get(1)).thenThrow(new RuntimeException()); // 抛出异常
//第一个会打印 "first"
System.out.println(mockedList.get(0));
//接下来会抛出runtime异常
System.out.println(mockedList.get(1));
//接下来会打印"null",这是因为没有stub get(999)
System.out.println(mockedList.get(999));
// 可以选择性地验证行为，比如只关心是否调用过get(0)，而不关心是否调用过get(1)
verify(mockedList).get(0);
```

代码覆盖率

比较流行的工具是Emma和Jacoco,Ecliplse插件有eclemma。eclemma2.0之前采用的是Emma，之后采用的是Jacoco。这里主要介绍一下Jacoco。Eclmama由于是Eclipse插件，所以非常易用，就不多做介绍了。

Jacoco

Jacoco可以嵌入到Ant、Maven中，也可以使用Java Agent技术监控任意Java程序，也可以使用Java Api来定制功能。

Jacoco会监控JVM中的调用，生成监控结果（默认保存在jacoco.exec文件中），然后分析此结果，配合源代码生成覆盖率报告。

需要注意的是：监控和分析这两步，必须使用相同的Class文件，否则由于Class不同，而无法定位到具体的方法，导致覆盖率均为0%。

Java Agent嵌入

首先，需要下载jacocoagent.jar文件，然后在Java程序启动参数后面加上 -javaagent:[yourpath/]jacocoagent.jar=[option1]=[value1],[option2]=[value2]，具体的options可以在此页面找到。默认会在JVM关闭时（注意不能是kill -9），输出监控结果到jacoco.exec文件中，也可以通过socket来实时地输出监控报告（可以在Example代码中找到简单实现）。

Java Report

可以使用Ant、Mvn或Eclipse来分析jacoco.exec文件，也可以通过API来分析。

```
public void createReport() throws Exception {
// 读取监控结果
final FileInputStream fis = new FileInputStream(new File("jacoco.exec"));
final ExecutionDataReader executionDataReader = new ExecutionDataReader(fis);
// 执行数据信息
ExecutionDataStore executionDataStore = new ExecutionDataStore();
// 会话信息
SessionInfoStore sessionInfoStore = new SessionInfoStore();
executionDataReader.setExecutionDataVisitor(executionDataStore);
executionDataReader.setSessionInfoVisitor(sessionInfoStore);
while (executionDataReader.read()) {
}
fis.close();
// 分析结构
final CoverageBuilder coverageBuilder = new CoverageBuilder();
final Analyzer analyzer = new Analyzer(executionDataStore, coverageBuilder);
// 传入监控时的Class文件目录，注意必须与监控时的一样
File classesDirectory = new File("classes");
analyzer.analyzeAll(classesDirectory);
IBundleCoverage bundleCoverage = coverageBuilder.getBundle("Title");
// 输出报告
File reportDirectory = new File("report"); // 报告所在的目录
final HTMLFormatter htmlFormatter = new HTMLFormatter(); // HTML格式
final IReportVisitor visitor = htmlFormatter.createVisitor(new FileMultiReportOutput(reportDirectory));
// 必须先调用visitInfo
visitor.visitInfo(sessionInfoStore.getInfos(), executionDataStore.getContents());
File sourceDirectory = new File("src"); // 源代码目录
// 遍历所有的源代码
// 如果不执行此过程，则在报告中只能看到方法名，但是无法查看具体的覆盖（因为没有源代码页面）
visitor.visitBundle(bundleCoverage, new DirectorySourceFileLocator(sourceDirectory, "utf-8", 4));
// 执行完毕
visitor.visitEnd();
}
```

以上就是本文的全部内容，希望对大家的学习有所帮助，也希望大家多多支持我们。