# 《JUnit实战 第二版》读书笔记：

“本书展示了如何想尽一切办法来进行测试。” --John Griffin

PS: 这句话来自本书的书背面的推荐语。

### JUnit起步

单元测试框架三大规则：

1. 每个单元测试都必须**独立**于其他所有单元测试而运行；
2. 框架应该以单个测试为单位来检测和报告；
3. 应该抑郁定义要运行哪些单元测试。

### JUnit核心

JUnit核心对象：

**断言** Assert 断言，预言成功通过测试，预言失败抛出异常

**注解** 测试 @Test注解测试方法 自动生成测试类实例，运行测试方法

测试类 **@****Test**注册测试类，测试方法容器

**打包** Suite 自动打包，为测试分类

**运行器** Runner 运行测试的加载器

运行器可以理解为，运行整个测试框架的工具，定义了整个测试框架要如何加载测试测试类测试方法。整个流程大致为，JUnit扫描将所有@test注释的需要测试的类全部打包成一个Suite,然后用运行器将对包中的测试类根据运行器规则进行注解解读，并按照规则顺序执行测试方法，检测是否符合预先定义的断言。其中JUnit4中运行器大致分为下面四类：  
 兼容3.8 org.junit.internal.runners.JUnit38.ClassRunner

默认4.0 org.junit.runners.JUnit4

参数化测试 org.junit.runners.Parameterized

容器测试 org.junit.runners.Suite

[**使用格式@Runwith(value=\*\*\*.class)**](mailto:使用格式@Runwith(value=***.class))来注解测试类。JUnit4向后兼容，可以用3.8运行器运行旧版本的测试用例，同时可以使用org.junit.runner.JUnitCore提供的统一接口，由facade决定运行哪个版本运行器，更加方便。同时，可以继承org.junit.runner.Runner类，创建自己的运行器，根据自己的需求定义运行器。

默认Junit会扫描所有的测试类以及其中的测试方法，全部打包自动创建一个Suite,其实这就是容器测试，同时根据自己的需求可以自定义组装测试，将几个测试作为集合一起作为单独的测试集。[使用格式@SuiteClasses(value={\*Test.class,\*Test.class})](mailto:使用格式@SuiteClasses(value=%7b*Test.class,*Test.class%7d)),这样两个类中的测试方法都会被打包测试。可以包含测试集class，也就是多个测试集，可以自由组合成更大的测试集合。

### 掌握JUnit

单元测试的最原始的思路是Stub-隔离策略，用虚拟的资源去替换测试的环境的其他因素，将整个被测试的部分被隔离起来，在可控的环境中去测试测试对象，这样的环境被称作测试状态机。

对于一些会抛出异常的方法，如何测试方法是否能抛出正确的异常呢？JUnit就提供了测试异常的方法，[使用格式@Test(expected=RuntimeException.class)](mailto:使用格式@Test(expected=RuntimeException.class)),来预期将会发生的异常

JUnit还提供了一种用来测试，方法调用执行时间的超时测试，可以轻松对调用发放运行时间过长进行检测，使用格式@Test(timeout=130),单位默认是毫秒，如果测试的方法执行超过设置时间，就会被认为执行用例失败。

对于在测试集合中不再测试的测试方法可以使用格式@Ignore(value=”why ignore”)。

这本书还推荐了一个匹配器-Hamcrest,这个匹配器相当于提供了很多约束词，约束语可以对预言进行描述，相当于提供了比assert更多种的断言，断言更方便。需要静态import匹配器相关的类，org.hamcrest.corematchers.anyOf/equalTo/Instance/not/hasItem等各种匹配器，这里不再详细描述。

测试类中的生命周期：

@Before/After 公有，在每个测试方法前后执行

@BeforeClass/AfterClass 公有且静态，在测试类实例化后，测试方法执行前，无序

Junit断言方法

1. assertArrayEquals("message", A, B) --断言A数组和B数组相等

2. assertEauals("message", A, B) --断言A对象和B对象相等

3. assertSame("message", A, B) --断言A对象和B对象相同

4. assertTrue("message", A) --断言A条件为真

5. assertNotNull("message", A) --断言A对象不为null

### 软件测试原则

单元测试必要性：

1.更高的测试覆盖率；

2.提高团队效率；

3．监控衰退退，降低对调试的需要；

4.能够对重构，改变是否可行有一定的检测；

5.将期望的行为文档化，明确。

测试的类型由小到大依次是：单元测试、集成测试、功能测试、压力负荷测试和验收测试。其中单元测试根据测试的范围粒度，分为逻辑单元测试、集成单元测试和功能单元测试。

推荐了**JMeter**进行压力测试，**Profiler**进行性能测试，**Cobertura**进行覆盖率检测

### 测试与开发

TDD测试驱动开发，他是一项编程实践，要求开发者只有在测试失败的情况下才编写新的代码，先写测试用例，用测试用例来规范代码的运行结果。这样测试不仅仅是一种检测手段，更是一总预先指定好的规则。这样的优点是，程序的执行结果是一直被规范化的，不会出现由程序主导执行结果，使得编码更规范，缺点是，会给前期的编码增加巨大的负担。

迪米特原则，又被称为最少知识原则，它主张一个类只要知道它需要知道的那么多即可。这样的原则更是被应用到设计模式当中。

### 生成测试对象的策略

测试状态机是需要透明已知的，那么如何去模拟出这个测试的对象即环境呢？现在主流的有两种策略：**Stub和Mock** 。

Stub是原始的方法，允许用户在不修改代码的前提下，使用一段代码，来代替真实代码，这样的策略思路大概为-**替换/隔离对象环境，**用其他代码替换测试中需要使用的对象。这样的坏处就是可能会比较复杂难以替换，并且替换的代码同时也需要维护，不同的场景需要不同的Stub，而且部分不能被替换的对象，很可能会造成测试遗漏。比如用Jetty服务器模拟HTTP连接，模拟WEB服务资源、又或者为测试创建自己的eURL协议处理器，这样就可以在没有真实的连接环境也能测试客户端处理结果。

Mock是新兴的测试策略，我觉得他的思路为-**仿造/隔离方法**,他对被测试的对象进行模拟，把测试的部分代码隔离出来，仿造类，并为所有业务逻辑方法行为提供一个空壳。会比较常用到一个设计模式的思路**-IoC控制反转和接口模式**。Mock有两个使用非常广泛的框架：EasyMock和JMock,框架提供了Mock中的标准的仿造类的过程，不要自己去书写仿造类，大大提高了Mock测试的效率。

### 容器内测试

组件和容器指：容器为存放在内的组件提供了各种服务，如生命周期、安全、事务和分布等。在servlet和JSP的情况下，容器就是一个Servlet容器，如Jetty或Tomcat，以及一些其他类型容器，EJB、数据库和OSGi容器。容器在运行期间创建并管理对象，那标准的单元测试就无法测试这些对象。采用一种机制来部署和执行在容器中的测试，就是容器内测试。

### 附录1 JUnit最佳实践：

1. 对还没有实现的测试代码抛出一个异常
2. 选择有意义的测试方法名字
3. 一次只能单元测试一个对象
4. 在assert调用中解释失败的原因
5. 一个单元测试等于一个@Test方法
6. 测试任何可能失败的事物
7. 让测试改善你的代码
8. 使异常测试更易于阅读
9. 总是为跳过测试说明原因
10. 测试方法和被测试的类，相同的包，分离的目录
11. 首先编写失败的测试、
12. 持续回归测试
13. 不要在Mock objects中写入业务逻辑
14. 值测试那些可能出错的代码
15. 创建EasyMock对象