Lecture5 Code Coverage

1. 介绍

是否有一个标准的对于测试质量的衡量？有的，那就是代码覆盖率

> - https://en.wikipedia.org/wiki/Code_coverage
> - https://en.wikipedia.org/wiki/White-box_testing
> - http://www.eclemma.org/jacoco/
> - http://www.jacoco.org/jacoco/trunk/doc/
> - https://www.atlassian.com/software/clover

什么是代码覆盖率

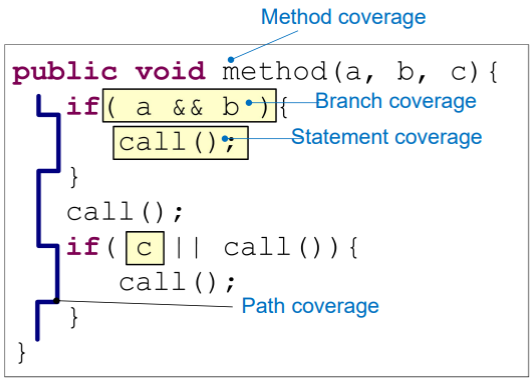
- 代码覆盖率是一种度量，**用来描述在特定测试时程序源代码执行的程度**
- 代码覆盖被归类为**白盒测试 White box testing**
  - 白盒测试：在已知被测试项目的内部结构/设计/实现的地方进行测试

代码覆盖率的优点

- 确定代码库中未测试的部分
- 通过改进的测试覆盖率来提高质量
- 确定测试差距或遗漏的测试
- 识别冗余/无效代码

覆盖标准

为了度量一个测试套件执行了多少百分比的代码，需要使用一个或多个覆盖率标准



- 指令覆盖 Instructions Coverage
  - Method 的字节码流是 JVM 的指令序列
  - 方法的字节码在该方法被调用时执行
- 语句覆盖 Statements Coverage
  - 报告是否执行了每个可执行语句

- 分支覆盖 Branch Coverage
  - 报告布尔表达式的值是否为 true 和 false
- 方法覆盖 Method Coverage
  - 报告在测试应用程序时是否调用了一个方法(函数)
- 类覆盖 Class Coverage
  - 报告所覆盖的代码基类的数量

代码覆盖分析流程

- 编写测试用例并执行它们
- 使用代码覆盖工具查找未覆盖的代码区域
- 为确定的差距创建额外的测试，以增加测试覆盖率
- 确定代码覆盖率的定量度量

使用 JaCoCo 进行代码覆盖率检查

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.abc.sss.services		0%		0%	313	313	738	738	90	90	4	4
com.abc.ppp.utils		0%		0%	181	181	488	488	37	37	2	2
com.abc.sss.export		23%		23%	169	214	645	856	33	55	0	2
com.abc.aa.import		63%		50%	294	491	449	1,388	33	131	5	19
com.abc.aa.export		39%		36%	179	239	451	742	34	60	3	5
com.abc.aa.view		16%		10%	186	203	331	409	27	41	5	9
com.abc.bb.code		34%		38%	142	217	332	559	11	61	0	9
com.abc. xx.utilities		42%		37%	114	160	278	515	15	34	2	4
com.abc.yy. search		37%		28%	141	181	295	489	20	43	2	5
com.abc. zz.ptl		56%		50%	229	410	322	766	121	265	4	16
com.abc.xx.result		40%		30%	95	144	214	374	12	39	0	5

JaCoCo 是一个用于 Java 的开放源代码覆盖工具，它是由 EclEmma 团队创建的，它提供指令覆盖行覆盖分支覆盖和包覆盖等

- 使用系统的JVM配置 JaCoCo 代理，以检测 java 类
- 在系统上执行测试用例时生成 EXEC 文件
- 生成代码覆盖率报告(不同格式)

其它测试代码覆盖率的工具

- Cobertura
- Atlassian Clover
- DevPartner
- JTest
- Bullseye for C++
- Sonar
- Kalistic

2. Evosuite

> - UsingEvoSuite on the command line: http://www.evosuite.org/documentation/tutorial-part-1/
> - UsingEvoSuite with Maven: http://www.evosuite.org/documentation/tutorial-part-2/
> - Running experiments withEvoSuite: http://www.evosuite.org/documentation/tutorial-part-3/
> - ExtendingEvoSuite: http://www.evosuite.org/documentation/tutorial-part-4/

有什么工具可以通过自动生成 JUnit 测试来帮助快速提高覆盖率?

是的，有几种流行的开源测试生成器

- Randoop
- Evosuite

如何工作

EvoSuite 使用**进化算法**来**生成和优化整个测试套件**，以满足覆盖标准

获得 EvoSuite

http://www.evosuite.org/downloads

- Jar 发布（命令行使用）
- Maven 插件
- IntelliJ 插件
- Eclipse 插件
- Jenkins 插件

什么时候使用，什么时候不用

- 测试自己的 Java 代码
  - 使用!
- 在单元测试生成上实现我的想法?
  - 使用!
- 研究开发者行为?
  - 使用!
- 为我在X上的实验生成单元测试?
  - 使用!
- 为不同的语言构建单元测试生成器?
  - 不太行
  - Evosuite 的 90% 是 JVM 来处理的代码
    - 需要重新实现一些表达，例如操作符，适应函数，测试执行等...
- 创建一个 android 测试工具?
  - Android 使用 Java Dalvik 字节码，可以编译成 Java 字节码
  - 但是比较难处理安卓的一些依赖
- 创建一个 GUI 测试工具?
  - 如果你想测试 Java/Swing 程序，可以试试
  - 但是整个测试套件的优化很多都不是正确的选择

EvoSuite 的研究方向

- 增加代码覆盖率
- 可读性优化
- 更好的环境处理
- 找出开发人员如何从使用测试生成中获益最多
- 用户研究，复制

3. 突变测试 Mutation Testing

代码覆盖率的局限

```
@Test
public void add_should_add() {
    new Math().add(1, 1);
}
```

- assert 在哪里呢?
- 尽管看起来代码覆盖率是可以的
- 任何指标都可以被篡改，代码覆盖率是一个度量
  - 代码覆盖可以被篡改
  - 不管是出于故意的还是意外
- 代码覆盖率给你一种虚假的安全感
- 代码覆盖不能确保测试质量
  - 还有别的办法吗
  - 突变测试来拯救我们

什么是突变

我们在下面的情况中使用突变测试

- 使用定好的**规则**（突变操作符）
- 定义在**句法描述上**（语法：Java/C++/...）
- 进行**系统性的改变**（普遍适用的或根据经验验证的分布）
- 指向语法或从语法发展而来的**对象**（地面字符，测试或者程序）

为什么突变

```
public int m1(int i1, int i2) {
    return i1 + i2;
}
```

```
public int m1(int i1, int i2) {
    return i1 - i2;
}
```

上面两个方法，其中一个可能是程序员手误写错了，但假设测试内容如下

sum = m1(1,0) -> sum = 1 两个情况都输出的是 1（再测试时无法区分代码是否写错）

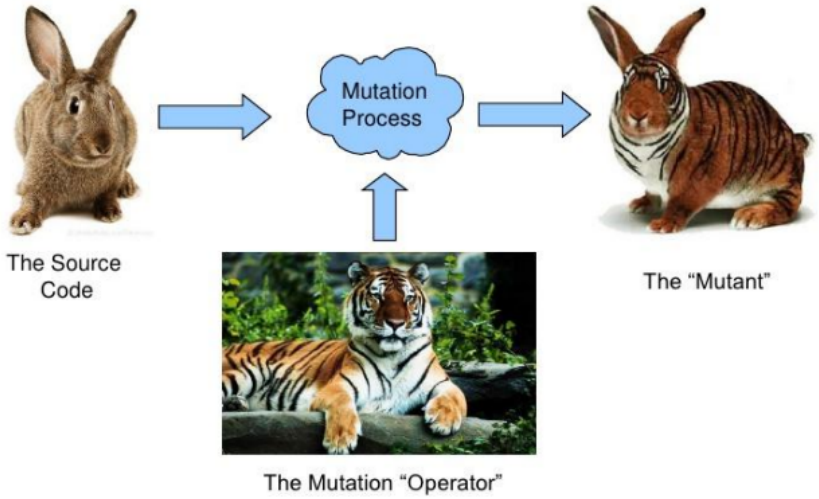
那么可能会补充一个测试样例

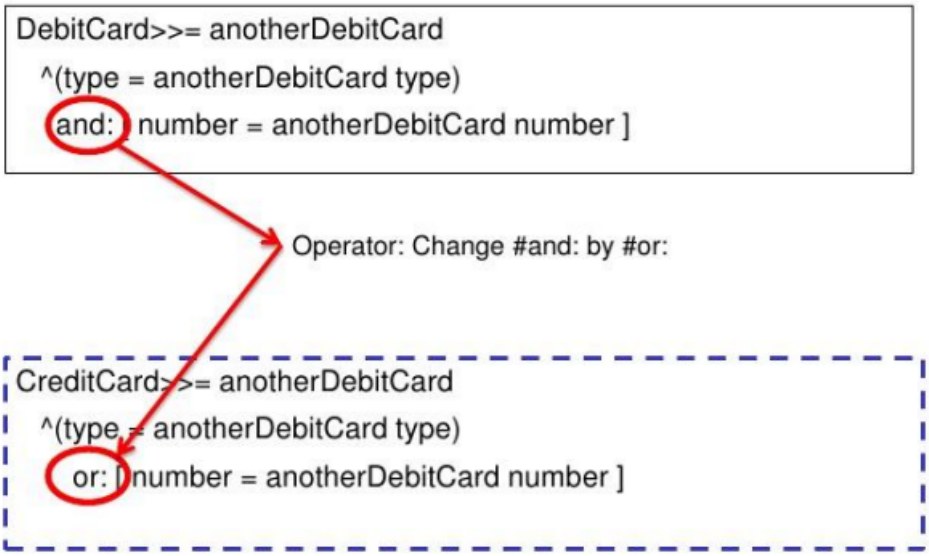
sum = m1(1,2) -> sum = 3 预期输出是 3，可以判断程序员写的方法应该是第一个

- 创建突变进程是为了尝试模仿程序员所犯的典型语法错误
- 许多不同的突变体在指定的测试中运行，以评估测试的质量
- 这些测试的得分介于 0 到 1、至于能否区分原始者和突变者

如何工作

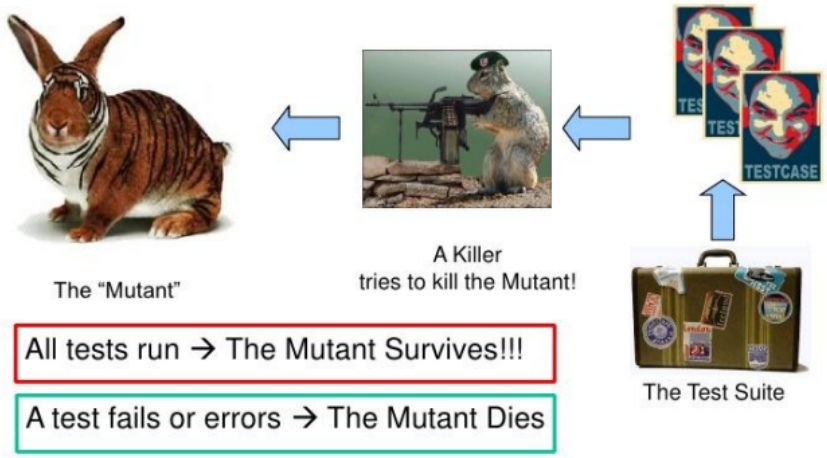
Step1：创建突变体





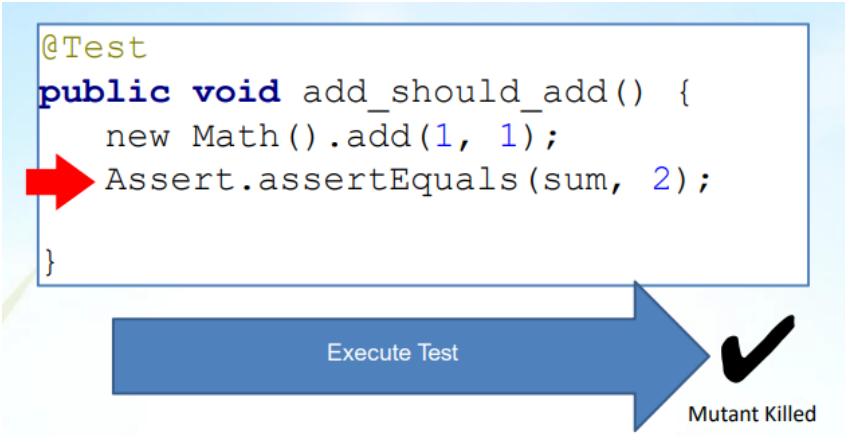
- 一般而言，程序通过更改一些符号来创建突变体

Step2：尝试杀掉突变体



- 如果所有的测试仍然能够通过 -> 突变者存活 -> 意味着由突变体生成的样例没有进行测试（存在问题）
- 如果有测试出错/error -> 突变者被杀掉 -> 意味着由突变体生成的样例被进行测试（通过）

示例：杀掉突变体



在 Java 中的突变测试

PIT 是一个用于突变测试的工具，可作为

- 命令行工具
- Maven 插件

突变器是应用于源代码以产生突变的模式

Name	Example source	Result
Conditionals Boundary	>	>=
Negate Conditionals	==	!=
Remove Conditionals	foo == bar	true
Math	+	-
Increments	foo++	foo--
Invert Negatives	-foo	foo
Inline Constant	static final FOO= 42	static final FOO = 43
Return Values	return true	return false
Void Method Call	System.out.println("foo")	
Non Void Method Call	long t = System.currentTimeMillis()	long t = 0
Constructor Call	Date d = new Date(d);	Date d = null;