

系统软件的质量保障

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组





Quiz: 白板面试题 @ Google (25分钟)

- 实现正则表达式匹配函数(成功返回0, 否则返回非0)

```
int match(const char *pattern, const char *str);
```

- 其中pattern是包含通配符的正则表达式模板串
 - * (星号)代表0个或任意多个字符
 - ? (问号)代表恰好1个任意字符
 - *.out匹配a.out; a*?*a匹配aba
- 拿出一张纸, 写出match函数的代码
 - 实际上发生在白板上(看着你写)

Quiz: 面试题的另一部分 (10分钟)

- 你会怎样测试你的代码?
 - 写在白纸的后半部分
- 这才是杀招啊!
 - 瞬间，你可能就觉得你前面写的代码弱爆了
 - 你不仅要能写对，还要向别人说明你写得对



当你在写的时候，面试官在看什么？

- 你的习惯

- 用什么样的方式考虑问题
- 用什么样的方式写代码(今天的关注点)

- 具体而言

- 怎么组织解题逻辑
- 怎么给变量命名
- 怎么处理每一个细节
 - 变量怎么定义？
 - 这个函数需要malloc(), 怎么处理？(嘿嘿嘿)

```
int match(const char *pattern,
          const char *str) {
    int f[strlen(pattern)][strlen(str)];
    ...
    return ...;
}
```



你写测试的时候，面试官在看什么？

- Basically, 看笑话
 - 他的大脑里准备了一万种tricky的情况，等你来掉坑
 - 但如果你都handle了，他会觉得你做得很不错
- 那怎么在面试中得到120分的表现呢？

软件测试



留意问题：怎样测试你的代码？

- 怎样测试代码 \neq 怎样设计测试用例
- 我们希望增加 “**代码正确解决了问题**” 的信心
- 我们想要得到哪方面的信心？
 - 对于不匹配的情况，真的是不匹配
 - 对于匹配的情况，真的是匹配

废话，我要知道这个成立不就证明代码是对的了么

怎么测试我们的代码？

- 对于手工构造的测试用例，我们知道它的结果

```
assert(match("a*?b", "ab") == 0);
```

- 结果也许的确对了，但我们怎么知道**计算过程也是对的**？
 - “代码是否正确实现了算法？”
 - 即便代码是错的，还是有可能输出对的结果
 - 比如程序写错了，就等价于return 0

测试：给我们更多的信心 (1)

- 在匹配的时候，每个Pattern字符都映射了 $[l_i, r_i)$ 的字符串
 - $l_1 = 1, l_n = m, r_i = l_{i+1}$ (首尾相接)
 - $l_i \leq r_i$ ($l_i = r_i$ 意味着不匹配)
 - 根据pattern字符讨论
 - 星号：无限制
 - 问号： $l_i + 1 = r_i$
 - 普通字符： $l_i + 1 = r_i, P(i) = S(l_i)$
- 这么一个Test Oracle给我们什么信心？
 - 任何返回匹配成功的测试用例都找到了合法的匹配
 - 当然我们的check code也可能写错
 - 假设出错是小概率独立事件，还是增加了信心



测试：给我们更多的信心 (2)

- 我们已经增加了一个Test Oracle，如果它实现得正确
 - 任何返回匹配成功的测试用例都找到了合法的匹配
- 接下来，我们如果能无限地生成有匹配的测试输入.....
 - 至少在有匹配的时候我们信心很足！
- 怎么办呢.....
 - 任何一个(随机)字符串，把里面任何一些部分用*或者?代替
 - 这样就获得了几乎无限生成匹配的字符串
 - 此外，随机的字符串和模板串匹配的可能性很低
 - 这样也获得了几乎无限“不匹配”的字符串(偶尔有匹配)

小结：好的测试

- 好的测试未必是“几个测试用例”
- 几个测试用例也很重要
 - 出错了帮助我们快速调试
 - 在版本更新后进行回归(regression)测试
- 系统化的测试
 - 生成大量不同类型的测试输入(test input)
 - 自动判断其是否有出错的可能(test oracle)

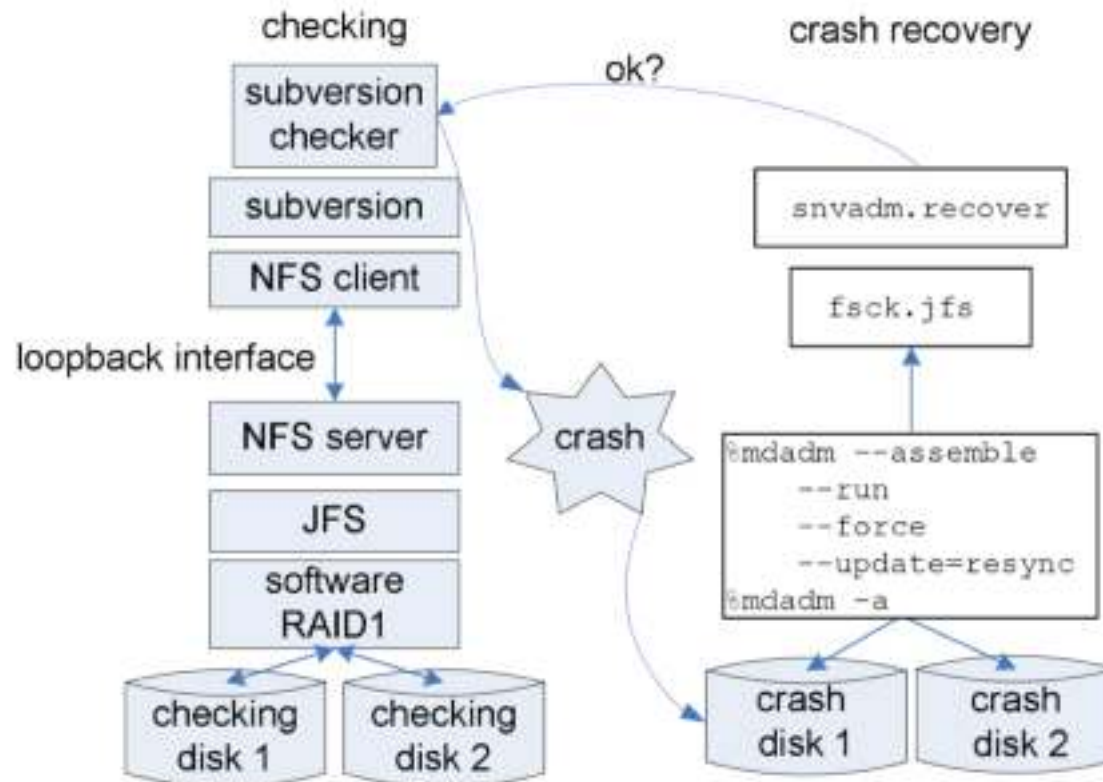
Differential Testing

- 再写一份解决同样问题的代码
 - 通常使用不同的算法、实现(独立性)
 - OI竞赛中常用的“代码对拍”技术
 - 写一个暴力搜索，弄一个随机数据生成器，后台跑一小时
- 这个技术在研究界挺常用的
 - 编译器、文件系统、NEMU (PA)、...

案例：测试存储系统

存储系统的测试

- 问题：我们要用模拟崩溃测试存储系统
 - 存储系统有很多层级(RAID → JFS → NFS → subversion)



基本原则: Try Everything¹

- Explore all choices
 - 如果某个程序点可能发生n种情况, 那就把这些情况都试一遍
 - malloc返回NULL, ...
- Exhaust states
 - 把所有可能的状态都尝试一遍
- Touch nothing
- Report only true errors, deterministically

¹ J. Yang, C. Sar, and D. Engler. EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors. In *Proc. of OSDI*, 2006.

The Magical “choose()”

- 遍历
 - 各种文件系统操作序列
 - 各种内部函数内可能发生的状况
- 模拟崩溃
- 检查错误

```
void * kcalloc(size_t size, int flags) {
    if((flags & __GFP_NOFAIL) == 0)
        if(choose(2) == 0)
            return NULL;
    ...
}
```

```
1: const char *dir = "/mnt/sbd0/test-dir";
2: const char *file = "/mnt/sbd0/test-file";
3: static void do_fsync(const char *fn) {
4:     int fd = open(fn, O_RDONLY);
5:     fsync(fd);
6:     close(fd);
7: }
8: void FsChecker::mutate(void) {
9:     switch(choose(4)) {
10:    case 0: systemf("mkdir %s%d", dir, choose(5)); break;
11:    case 1: systemf("rmdir %s%d", dir, choose(5)); break;
12:    case 2: systemf("rm %s", file); break;
13:    case 3: systemf("echo \"test\" > %s", file);
14:        if(choose(2) == 0)
15:            sync();
16:        else {
17:            do_fsync(file);
18:            // fsync parent to commit the new directory entry
19:            do_fsync("/mnt/sbd0");
20:        }
21:        check_crash_now(); // invokes check() for each crash
22:        break;
23:    }
24: }
25: void FsChecker::check(void) {
26:     ifstream in(file);
27:     if(!in)
28:         error("fs", "file gone!");
29:     char buf[1024];
30:     in.read(buf, sizeof buf);
31:     in.close();
32:     if(strncmp(buf, "test", 4) != 0)
33:         error("fs", "wrong file contents!");
34: }
```


小结

- 理想的目标
 - 能枚举所有输入、都运行一遍——并且看起来都对
 - Explore all choices and states
- 现实
 - 我们受到各种限制(人力、计算资源.....) → 生成丰富的输入
 - EXPLODE: choose()遍历各种可能性
 - 运行的测试用例也不知道是否真正正确 → 找好的test oracle
 - EXPLODE: 立足文件系统(有可检查性)

案例：测试GNU Coreutils

Coreutils (Busybox)的测试

- Linux系统的命令大家每天都在用
 - 那有没有bug呢？该怎么测试呢？
- Coreutils和文件系统的不同
 - 没有checker和明确的语义
 - 文件系统是数据结构
 - 命令行选项非常复杂
 - 各种复杂组合和出错处理

grep的参数选项

```
--after-context=NUM      --files-without-match
--text                  --files-with-matches
--before-context=NUM    --max-count=NUM
--byte-offset           --mmap
--binary-files=TYPE     --line-number
--color[=WHEN]         --only-matching
--context=NUM          --label=LABEL
--count                --line-buffered
--devices=ACTION       --perl-regexp
--directories=ACTION    --quiet
--extended-regexp      --recursive
--regexp=PATTERN       --include=PATTERN
--fixed-strings        --exclude=PATTERN
--file=FILE           --no-messages
--basic-regexp         --binary
--with-filename        --unix-byte-offsets
--no-filename         --version
--help               --invert-match
-I                  --word-regexp
--ignore-case        --line-regexp
-y
-z
--n
```

¹ C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of OSDI*, 2008. (Best Paper Award Winner)

测试输入：自动生成

- 我们希望走到程序的“每一行代码”
 - 但如果我们随机输入(例如x, y), 效率可能很低

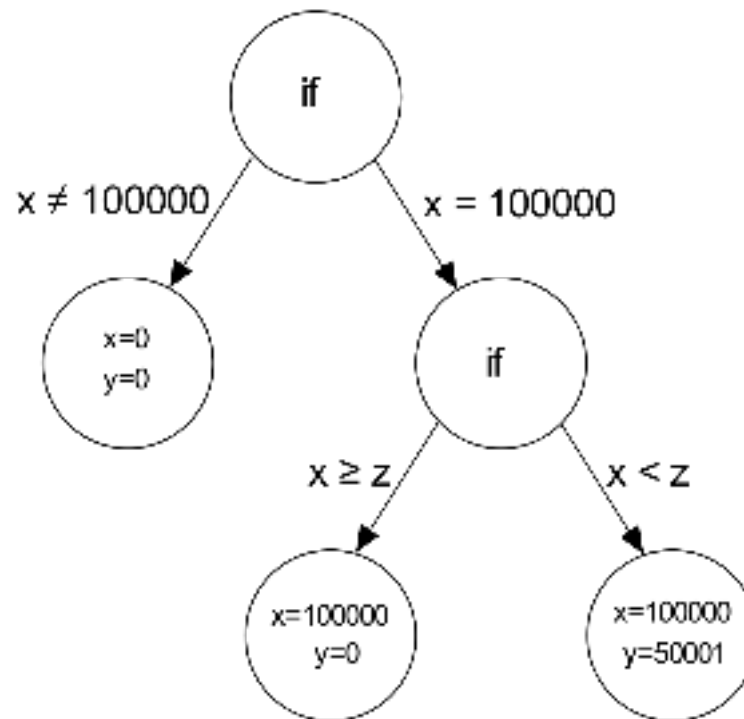
```
int f(int x) { return 2 * x; }
```

```
int h(int x, int y) {  
    if (x != y)  
        if (f(x) == x + 10)  
            bug();  
    return 0;  
}
```


- 有没有可能遍历所有的分支跳转？

符号执行(Symbolic Execution)

- 假装执行一条程序路径
 - 每一个分支都有一个方向
 - 在分支执行时，所有变量的值都可以写成一个表达式



隐藏的分支

`int t = x / y;`  `if (y == 0)
 bug();
int t = x / y;`

`int x = *ptr;`  `if (!valid_check(ptr))
 bug();
int x = *ptr;`

`int nread =
 read(fd, buf, size)`  `nread \in $\{-1, 0, 1, 2, \dots, size\}$`

似曾相识的
choose()

覆盖率：与开发者测试用例对比

- 自动测试竟然能达到90%以上的代码覆盖.....

Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

- 但也有一些难覆盖的代码
 - (我们正在进行这部分研究)