

调试：理论与实践

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



本讲概述

代码出 bug 了？又浪费了一天？太真实了？

- 本讲内容
 - 调试理论
 - 调试理论的应用
 - 调试 “不是代码”
 - 调试 PA/Lab/任何代码
- 为什么 PA 都开始那么久了现在才讲？
 - 因为你们不吃苦头是理解不了方法学的

调试理论

开始调试之前

- 摆正心态 (编程哲♂学)

机器永远是对的

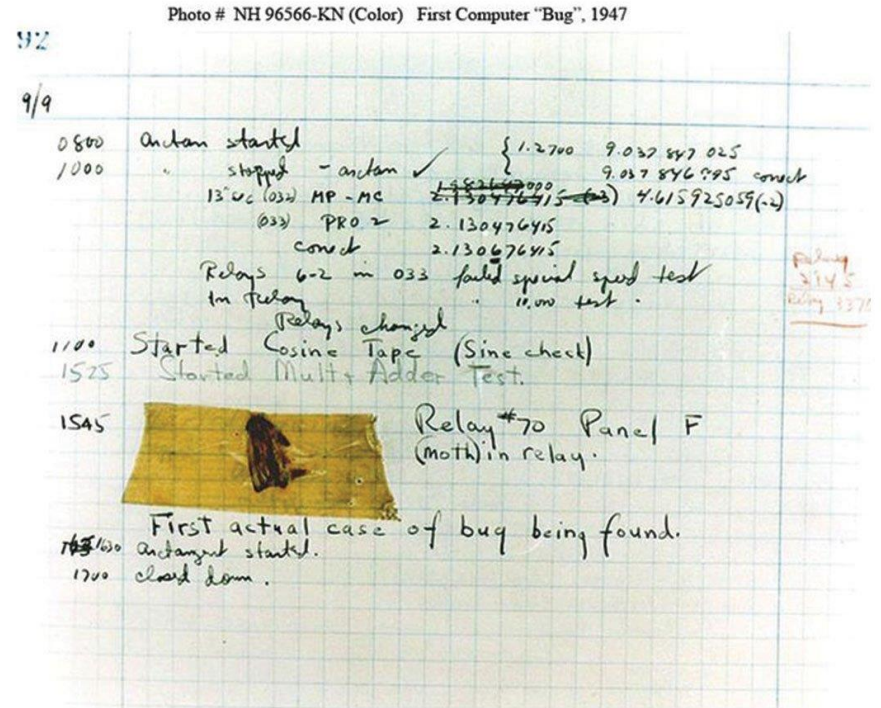
不管是 crash 了，图形显示不正常了，还是 HIT BAD TRAP 了，最后都是你自己背锅

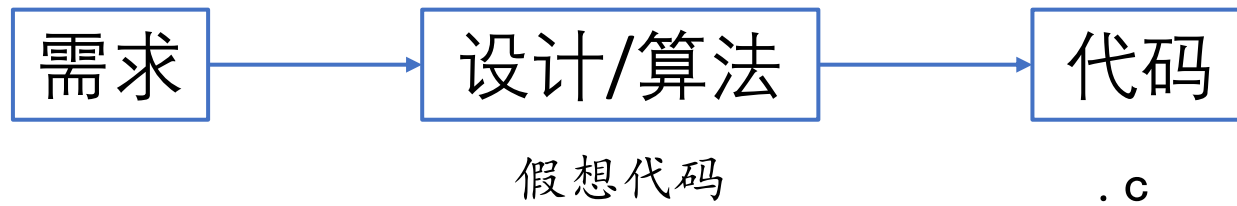
未测代码永远是错的

你以为最不可能出 bug 的地方，往往 bug 就在那躺着

调试理论

- 程序的两个功能
 - 人类世界需求的载体
 - 理解错需求 → bug
 - 计算过程的精确描述
 - 实现错误 → bug
- 调试 (debugging)
 - 已知程序有 bug, 如何找到?



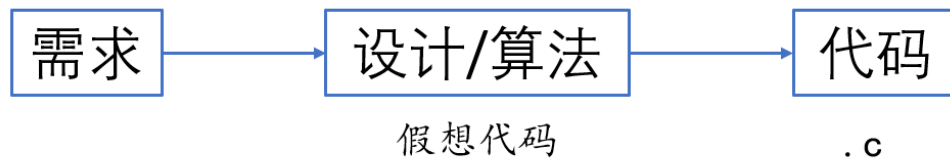


NEMU PA

为什么debug那么困难

- 因为 bug 的触发经历了漫长的过程
 - 需求 → 设计 → 代码 → Fault (bug) → Error (程序状态错) → Failure
 - 我们只能观测到 failure (可观测的结果错)
 - 我们可以检查状态的正确性 (但非常费时)
 - 无法预知 bug 在哪里 (每一行 “看起来” 都挺对的)

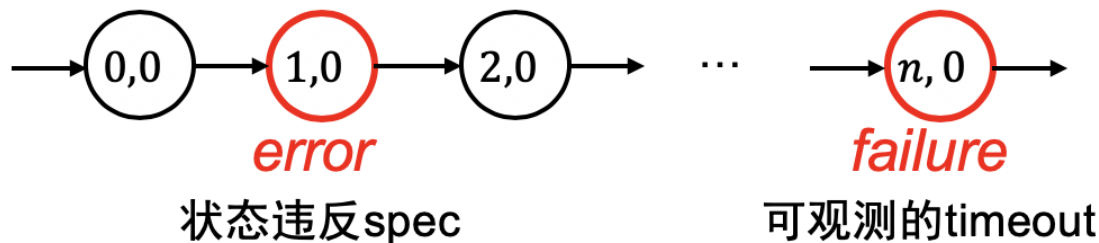
```
1 #include <stdio.h>
2
3 #define n 3
4
5 int main(){
6     int sum = 0;
7     for(int i = 0; i<n; i++){
8         for(int j = 0; j<n; i++){
9             sum += i * j;
10        }
11        printf("sum = %d\n", sum);
12 }
```



为什么debug那么困难

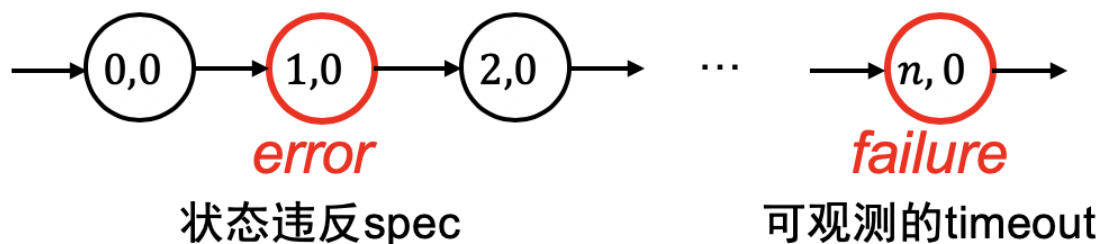
- 因为 bug 的触发经历了漫长的过程
 - 需求 → 设计 → 代码 → Fault (bug) → Error (程序状态错) → Failure
 - 我们只能观测到 failure (可观测的结果错)
 - 我们可以检查状态的正确性 (但非常费时)
 - 无法预知 bug 在哪里 (每一行 “看起来” 都挺对的)

```
for (int i = 0; i < n; i++) fault 程序bug
  for (int j = 0; j < n; i++)
  ...
```



假想执行的困难

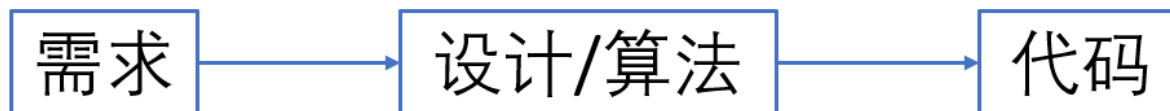
```
for (int i = 0; i < n; i++) fault 程序bug  
  for (int j = 0; j < n; i++)  
  ...
```



实际执行



假想执行

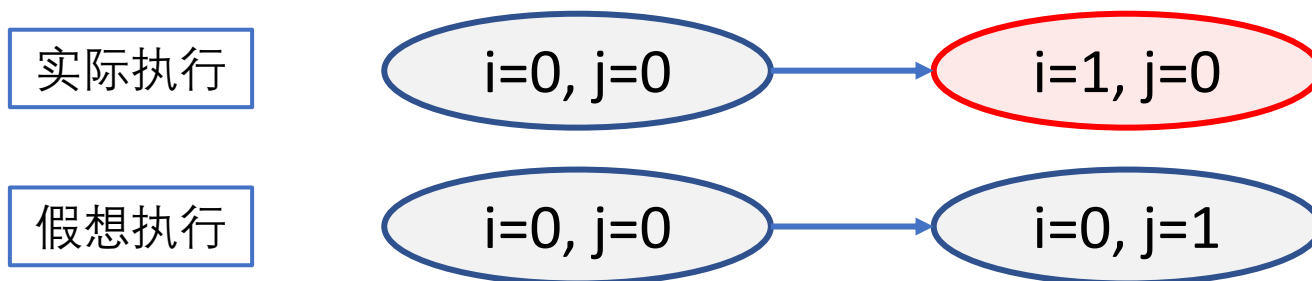


假想代码

.c

调试理论

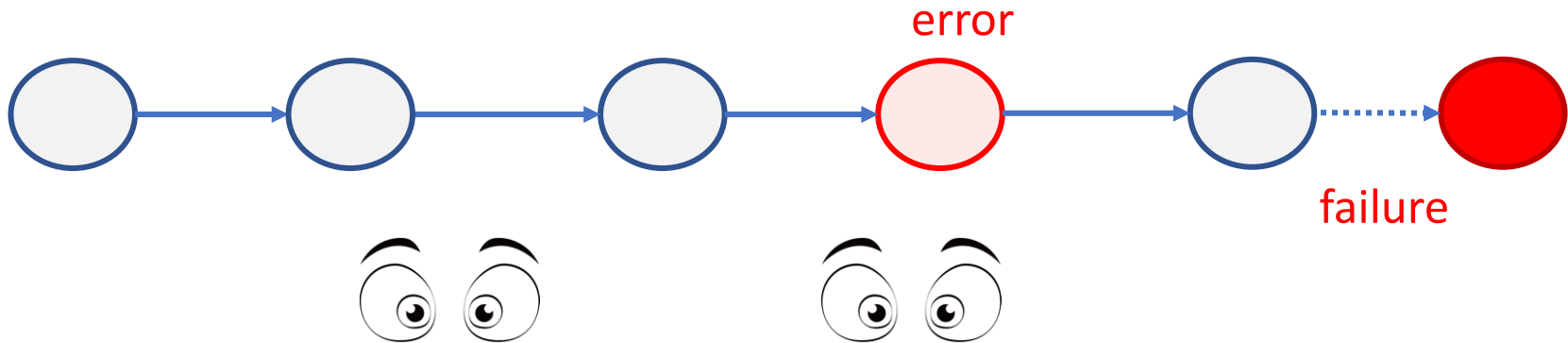
调试理论：如果我们能判定任意程序状态的正确性，那么给定一个 failure，我们可以通过二分查找定位到 **第一个** error 的状态，此时的代码就是 fault (bug)。



```
for (int i = 0; i < n; i++) fault 程序bug  
  for (int j = 0; j < n; i++)
```

Common practice

```
int main() {  
    sort();  
    //printf();  
    find_something();  
    //printf();  
    delete_something();  
    //printf();  
}
```

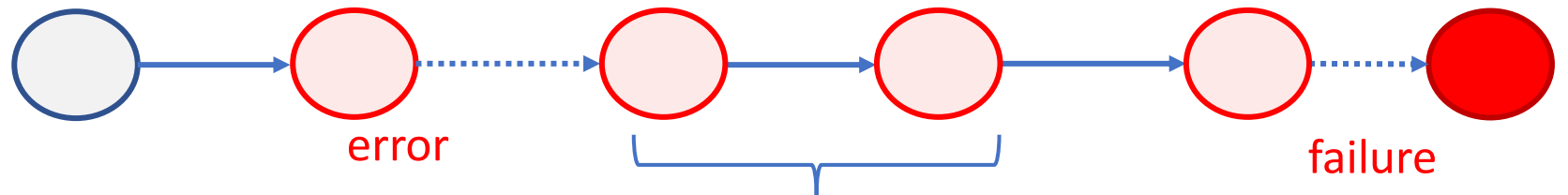
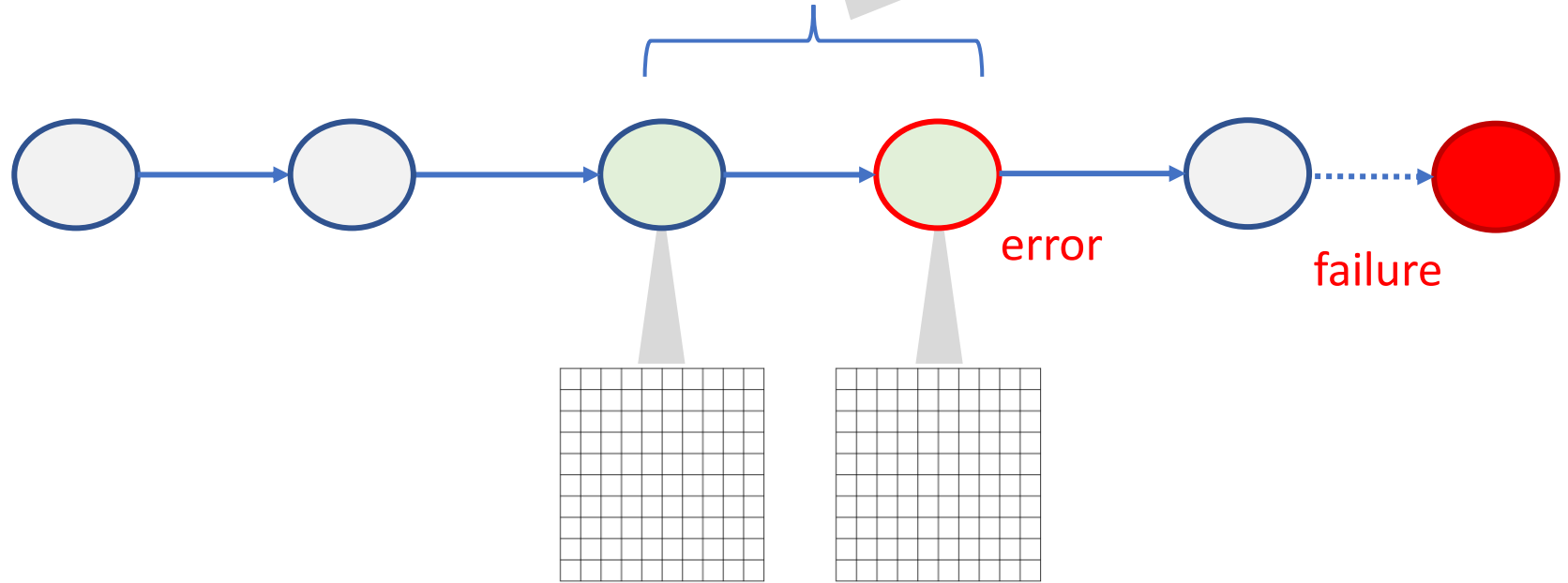


调试理论

调试理论：如果我们能判定任意程序状态的正确性，那么给定一个 failure，我们可以通过二分查找定位到 **第一个** error 的状态，此时的代码就是 fault (bug)。

- 调试理论：推论
 - 为什么我们喜欢 “单步调试”？
 - 从一个假定正确的状态出发
 - 每个语句的行为有限，容易判定是否是 error
 - 为什么调试理论看起来没用？
 - 因为判定程序状态的正确性非常困难
 - (是否在调试 DP 题/图论算法时陷入时间黑洞？)

Common practice: DP

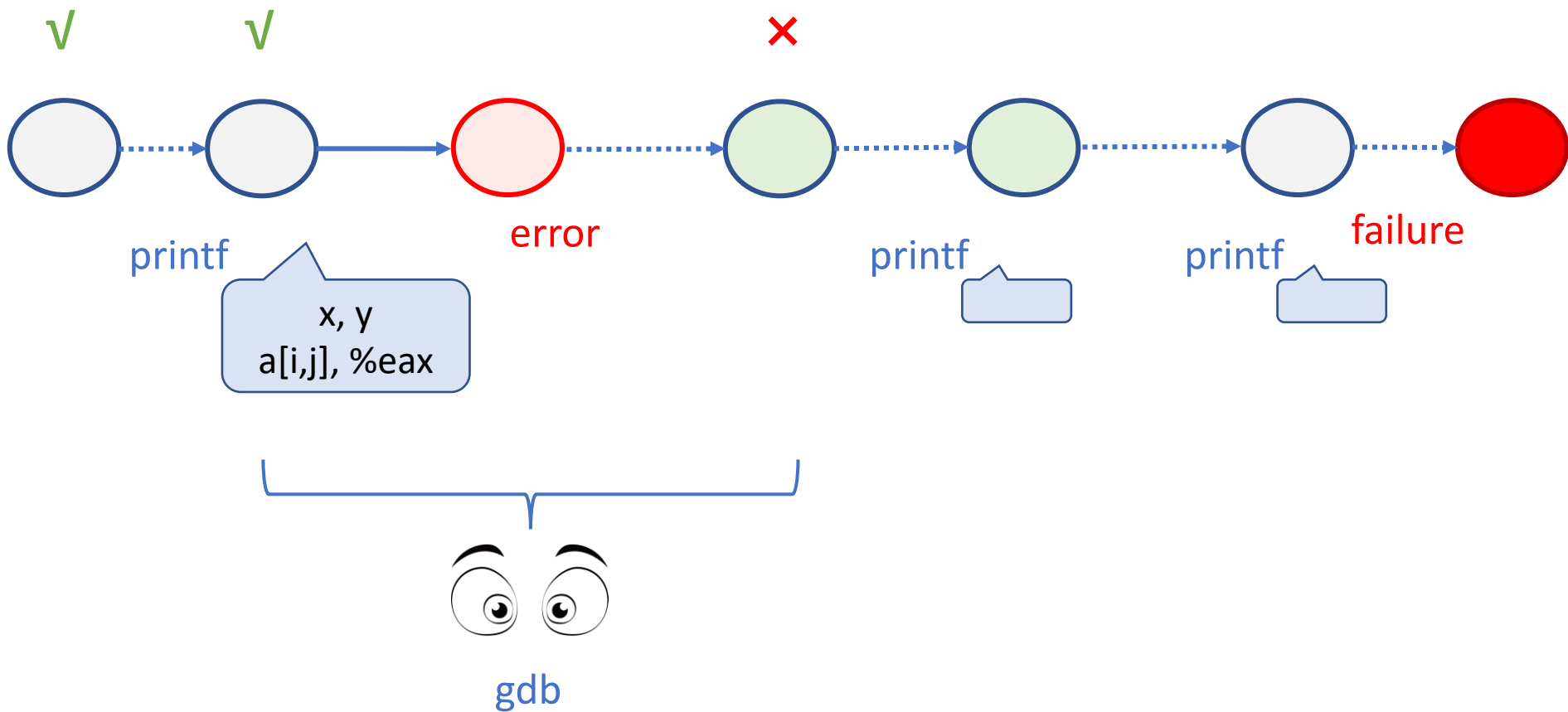


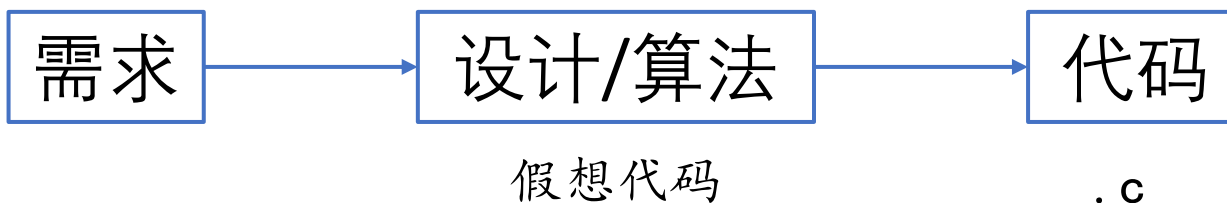
调试理论 (cont'd)

- 实际中的调试：通过观察程序执行的轨迹 (trace)
 - 缩小错误状态 (error) 可能产生的位置
 - 作出适当的假设
 - 再进行细粒度的定位和诊断
- 最重要的两个工具
 - printf → 自定义log的trace
 - + 灵活可控、能快速定位问题大概位置、适用于大型软件
 - - 无法精确定位、大量的logs管理起来比较
 - gdb → 指令/语句级trace
 - + 精确、指令级定位、任意查看程序内部状态
 - - 耗费大量时间

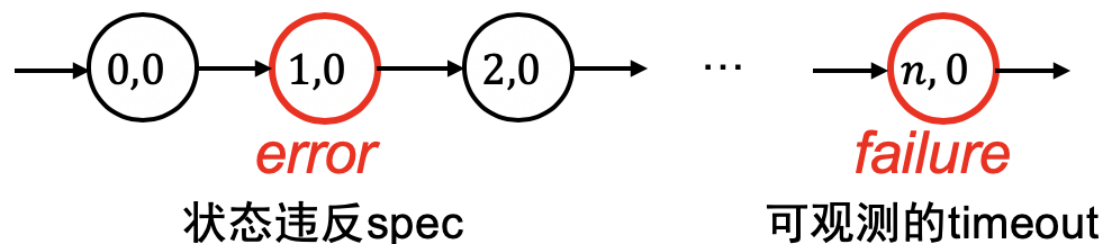
调试理论 (cont'd)

- 尝试接近状态的判定





```
for (int i = 0; i < n; i++) fault 程序bug
  for (int j = 0; j < n; i++)
    ...
```



printf → 自定义log的trace
+ 灵活可控、能快速定位问题大概位置、适用于大型软件
- 无法精确定位、大量的logs管理起来比较



gdb → 指令/语句级trace
+ 精确、指令级定位、任意查看程序内部状态
- 耗费大量时间

调试理论：应用（1）

调试（不是一般意义上“程序”的）bug

PA体验极差

- 做实验会遇到大量与编程无关的问题
 - 也许没有想过：问题诊断其实也是调试
 - 我们应该利用调试理论去解决问题！

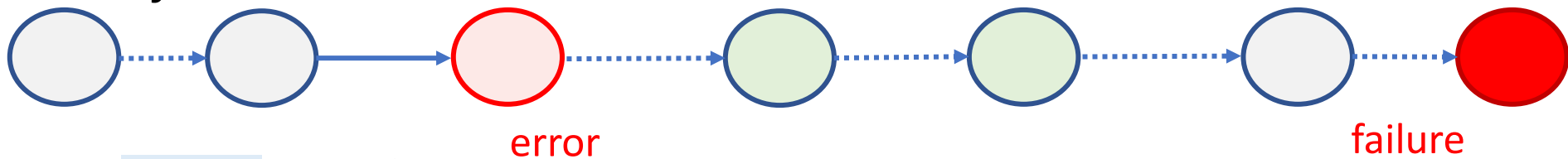
```
bash: curl: command not found
```

```
fatal error: 'sys/cdefs.h': No such file or directory #include  
<sys/cdefs.h>
```

```
make[2]: *** run: No such file or directory. Stop.  
Makefile:31: recipe for target 'run' failed  
make[1]: *** [run] Error 2  
...
```

配置

Faulty code



输入

\$

S 英 简 拼

```
$ bash a.sh  
<html>  
<meta http-equiv="refresh" content="0;url=http://www.baidu.com/">  
</html>  
$ bash a.sh
```

PA体验极差

- 做实验会遇到大量与编程无关的问题
 - 也许没有想过：问题诊断其实也是调试
 - 我们应该利用调试理论去解决问题！

```
bash: curl: command not found
```

```
fatal error: 'sys/cdefs.h': No such file or directory #include  
<sys/cdefs.h>
```

```
make[2]: *** run: No such file or directory. Stop.  
Makefile:31: recipe for target 'run' failed  
make[1]: *** [run] Error 2  
...
```

```
$ ls  
a.c  a.out  a.sh  
$
```

例子：找不到sys/cdefs.h

- 'sys/cdefs.h': No such file or directory, 找不到文件
 - (这看起来是用 perror() 打印出来的哦!)
 - #include = 复制粘贴, 自然会经过路径解析
 - (折腾20分钟) 明明/usr/include/x86_64-linux-gnu/sys/cdefs.h 是存在的 (man 1 locate)
- 推理: #include<>一定有一些搜索路径
 - 为什么两个编译选项, 一个通过, 一个不通过?
 - gcc -m32 -v **V.S.** gcc -v
- 这是标准的解决问题办法: 自己动手排查
 - 在面对复杂/小众问题时比 STFW 有效

PA体验极差

- 做实验会遇到大量与编程无关的问题
 - 也许没有想过：问题诊断其实也是调试
 - 我们应该利用调试理论去解决问题！

```
bash: curl: command not found
```

```
fatal error: 'sys/cdefs.h': No such file or directory #include  
<sys/cdefs.h>
```

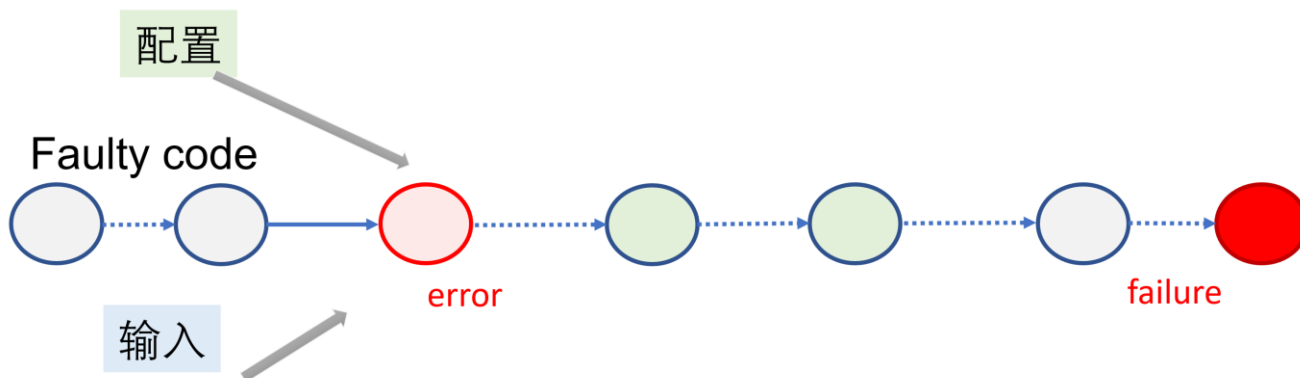
```
make[2]: *** run: No such file or directory. Stop.  
Makefile:31: recipe for target 'run' failed  
make[1]: *** [run] Error 2  
...
```

使用调试理论 (cont'd)

- 正确的方法：理解程序的**执行过程**，弄清楚到底为何导致了bug
 - ssh：使用 -v 选项检查日志
 - gcc：使用 -v 选项打印各种过程
 - make：使用 -n 选项查看完整命令
 - `make -nB | grep -ve '^(echo\|mkdir\|)'` 可以查看完整编译 nemu 的编译过程
- 各个工具普遍提供调试功能，帮助用户/开发者了解程序的**行为**

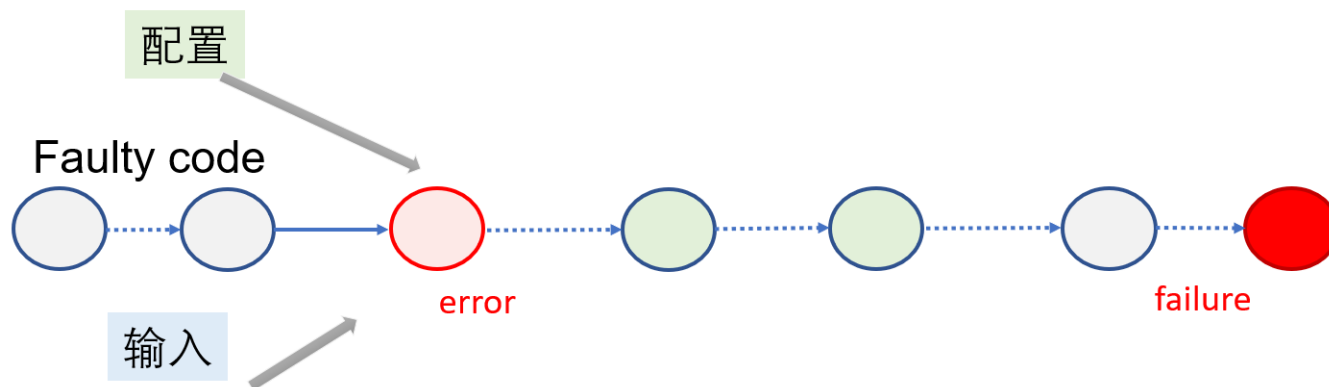
调试（不是程序的） Bug

- UNIX世界里你做任何事情都是在编程
 - 因此配置错、make 错等，都是程序或输入/配置有 bug
 - (输入/配置可以看成是程序的一部分)
- 正确的态度：把所有问题当程序来调试
 - 你写了一个程序，现在这个程序出 bug 了 (例如 Segmentation Fault)，你是怎样排查这个问题的？
 - `curl: command not found`
 - `'sys/cdef.h': No such file or directory`
 - `make run: No such file or directory`



使用调试理论

- Debug (fault localization) 的基本理论回顾：
 - Fault (程序/输入/配置错) → Error → Failure (可观测)
 - 绝大部分工具的 Failure 都有 “原因报告”
 - 因此能帮助你快速定位 fault
 - `man perror`: 标准库有打印error message的函数
- 为什么会抓瞎？
 - 出错原因报告不准确或不够详细
 - 程序执行的过程不详
 - 既然我们有需求，那别人肯定也会有这个需求
 - 一定有信息能帮助我们！



调试理论：应用（2）

调试PA/Lab/任何代码

调试 NEMU 的难处

1. NEMU 没有这些 debug 信息

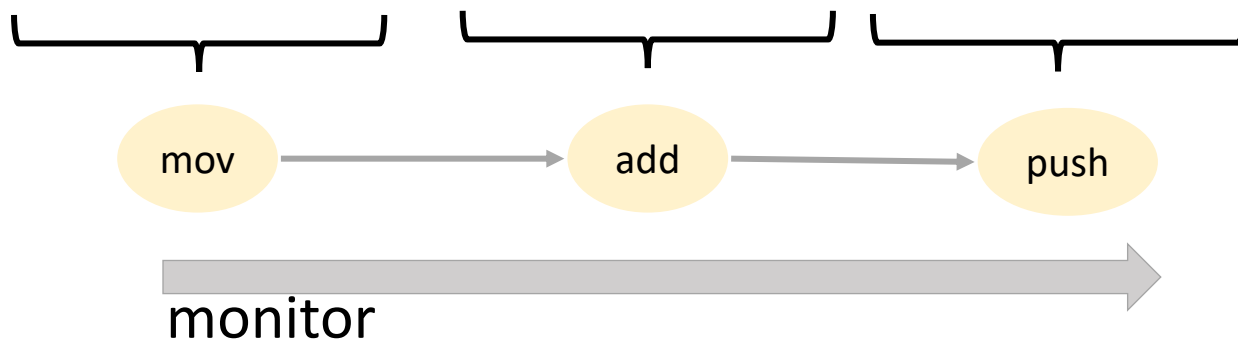
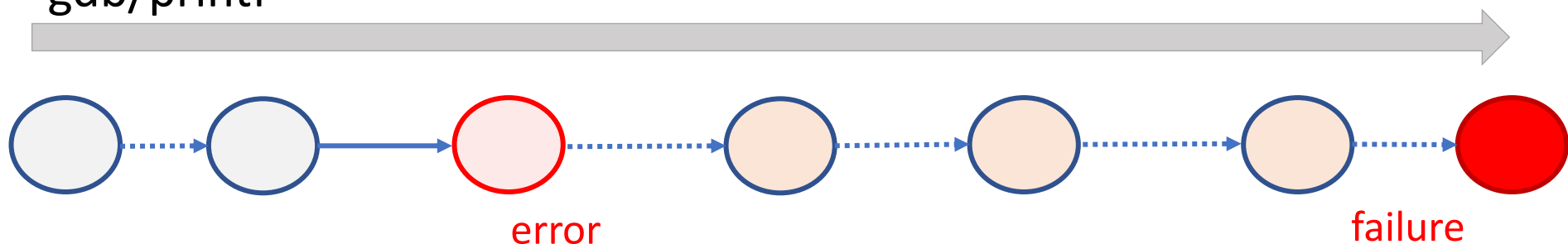
- 指令执行着执行着，悄悄就错了，直到在某个地方死循环
- 到底哪里错了呢？
 - 对于 cpu-test，还能一条一条看指令日志
 - 打字游戏的时候 bug 了，那么多指令.....
 - LiteNES bug 了，这上哪玩啊.....

2. NEMU 有两个层次的状态机

- NEMU binary 本身是个状态机
 - 它模拟了另一个状态机 (指令序列)
 - 这就是 monitor 的用处！



`gdb/printf`



调试理论回顾

需求 → 设计 → 代码 → Fault → Error → Failure

调试理论：应用 (Again)

需求  设计  代码 → Fault → Error → Failure

- “Technical Debt”

每当你写出不好维护的代码，你都在给你未来的调试挖坑。

- 中枪了？

- 为了快点跑程序，随便写的klib
- 为了赶紧实现指令，随手写的代码
-

编程基本准则：回顾

Programs are meant to be read by humans and only incidentally for computers to execute. — D. E. Knuth

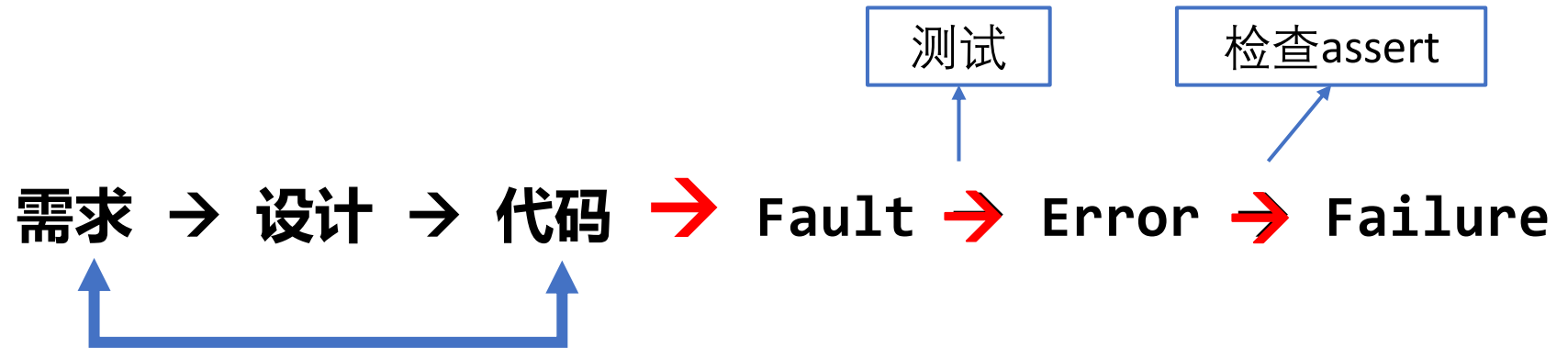
(程序首先是拿给人读的，其次才是被机器执行。)

- 好的程序

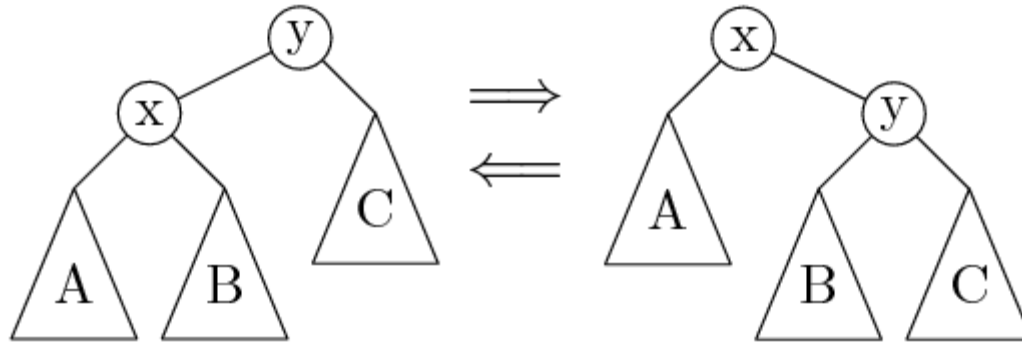
- 不言自明：能知道是做什么的
 - 因此代码风格很重要
- 不言自证：能确认代码和 specification 一致
 - 因此代码中的逻辑流很重要



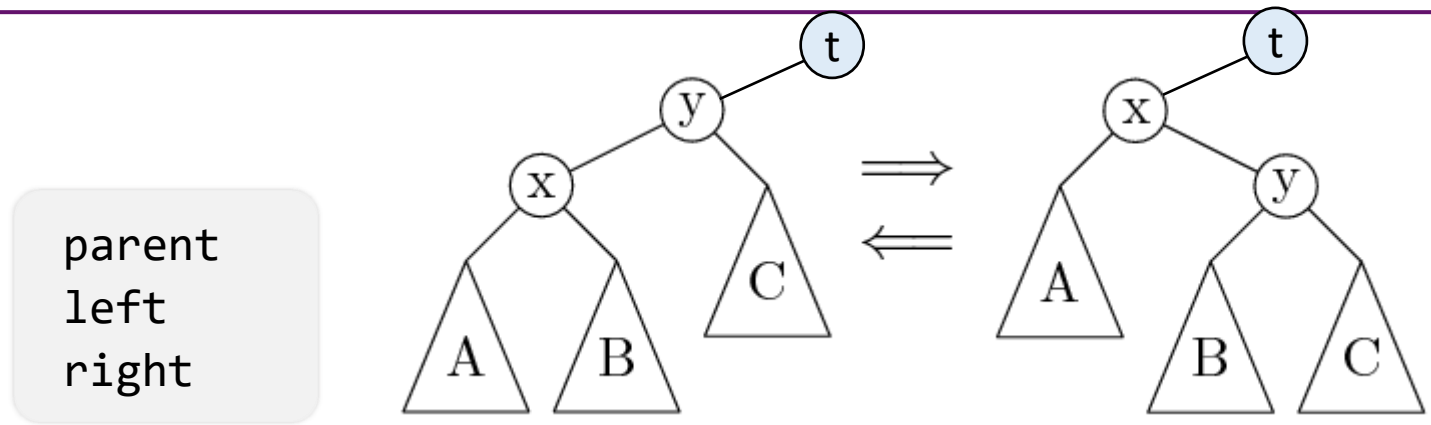
调试理论：



例子：平衡树的测试



例子：维护父亲节点的平衡树



t=y->parent
X
y
A
B
C

parent	left	right

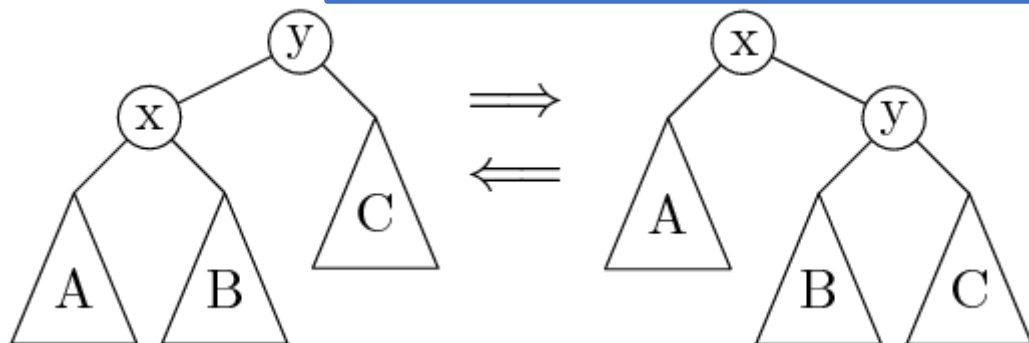


例子：维护父亲节

需求 → 设计 → 代码 → Fault → Error → Failure

测试

检查assert



// 结构约束

```
assert(u->parent == u ||
       u->parent->left == u ||
       u->parent->right == u);
assert(!u->left || u->left->parent == u);
assert(!u->right || u->right->parent == u);
```

// 数值约束

```
assert(!u->left || u->left->val < u->val);
assert(!u->right || u->right->val > u->val);
```

调试理论的最重要应用

写好读、易验证的代码

在代码中添加更多的断言 (assertions)

- 断言的意义

- 把代码中隐藏的 specification 写出来 (回到需求的另一种理解)
 - Fault → Error (靠测试)
 - Error → Failure (靠断言)
 - Error 暴露的越晚, 越难调试
 - 追溯导致 assert failure 的变量值 (slice) 通常可以快速定位到 bug

例子：NEMU 中的断言

- 看起来很没必要，但可以提前拦截一些未知的 bug
 - 例如 memory error

```
static inline int check_reg_index(int index) {  
    IFDEF(CONFIG_RT_CHECK, assert(idx >= 0 && idx < 32));  
    return index;  
}
```

```
#define gpr(idx) (cpu.gpr[check_reg_idx(idx)]._64)
```

```
Assert(map != NULL && addr <= map->high && addr >= map->low,  
    "address (0x%08x) is out of bound {%s} [0x%08x, 0x%08x] at pc = "  
FMT_WORD, addr, (map ? map->name : "???"), (map ? map->low : 0), (map ?  
map->high : 0), cpu.pc);
```

福利：更多的断言

- 你是否希望在每一次指针访问时，都增加一个断言
 - `assert(obj->low <= ptr && ptr < obj->high);`

```
int *ref(int *a, int i) {  
    return &a[i];  
}  
void foo() {  
    int arr[64];  
    *ref(arr, 64) = 1; // bug  
}
```

- 一个神奇的编译选项
 - `-fsanitize=address`
 - **Address Sanitizer; asan** “动态程序分析”

\$

S 英 简 拼

调试理论：应用

Fault → Error → Failure

Fault → Error

- 充分的测试
 - 例子：Lab1 测试

Error → Failure

- 充足的日志
- 及时的检查

用好工具

- monitor, gdb, ...

例子：使用GDB

```
$ ./a.out  
Segmentation fault (core dumped)
```

- GDB: 最常用的命令在 [gdb cheat sheet](#) (unavailable?)
- Segmentation fault 是一个非常好的 failure
 - 某条指令访问了非法内存
 - 无非就是空指针、野指针、栈溢出、堆溢出.....
- 怎样定位 segmentation fault 发生时的语句/指令？
 - (core dumped)
 - gdb - 自带 backtrace, 95% 都能帮你定位到问题

想要更好的体验？ RTFM！

- 每次 gdb 都要输入一堆命令，都烦了
 - 当你感到不爽的时候，一定有机会解决
- gdb 可以支持命令 (-x, -ex, ...)

```
set pagination off
set confirm off
layout asm
file a.out
b main
r
```

总结

残酷的现实和难听的本质

- 道理都懂，但出 bug 了，还是不知道怎么办
 - 一句难听的话
 - 你对代码的关键部分还不熟悉
 - 不知道如何判定程序状态是否正确
 - 对项目缺乏了解
 - 缺乏基础知识
 - 抱有“我不理解这个也行”的侥幸信息
- 《计算机系统基础》PA 给大家最重要的训练
 - 消除畏惧，但有些同学还没能理解
 - 努力做到知晓所有细节
 - 为你自己的代码负责

用好调试理论

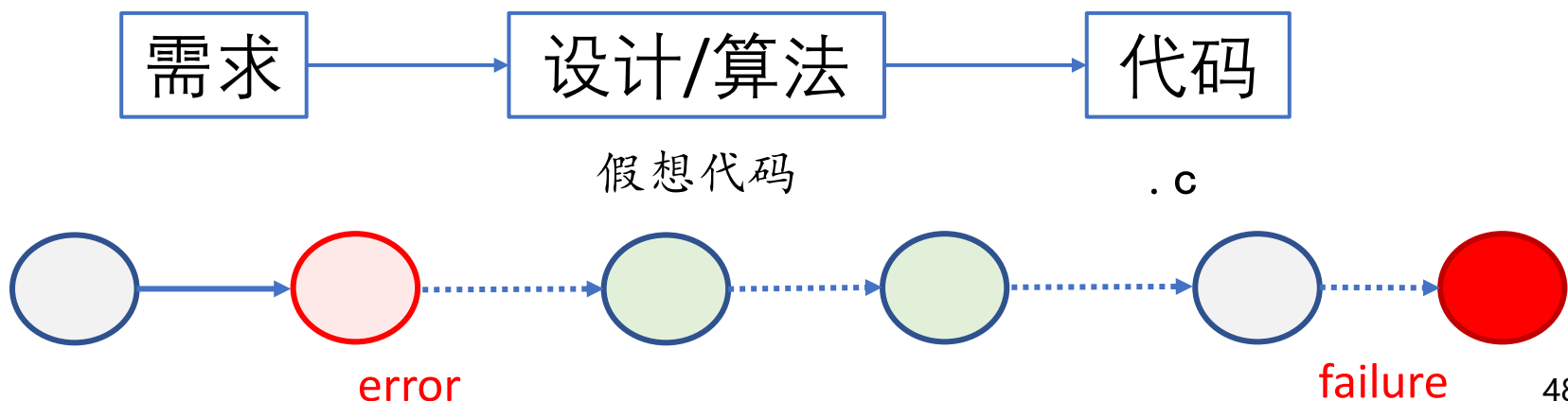
- 调试理论

- Fault → (测试) → Error → (断言) → Failure

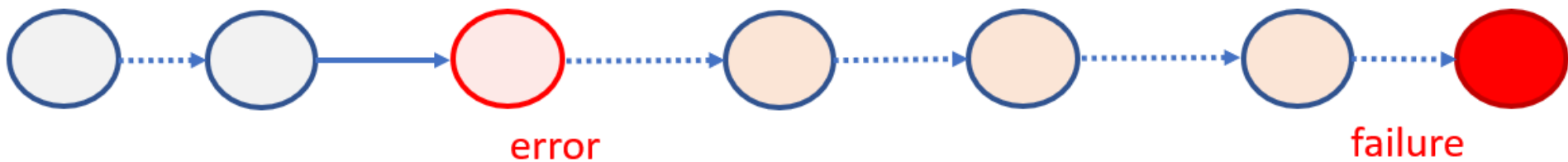
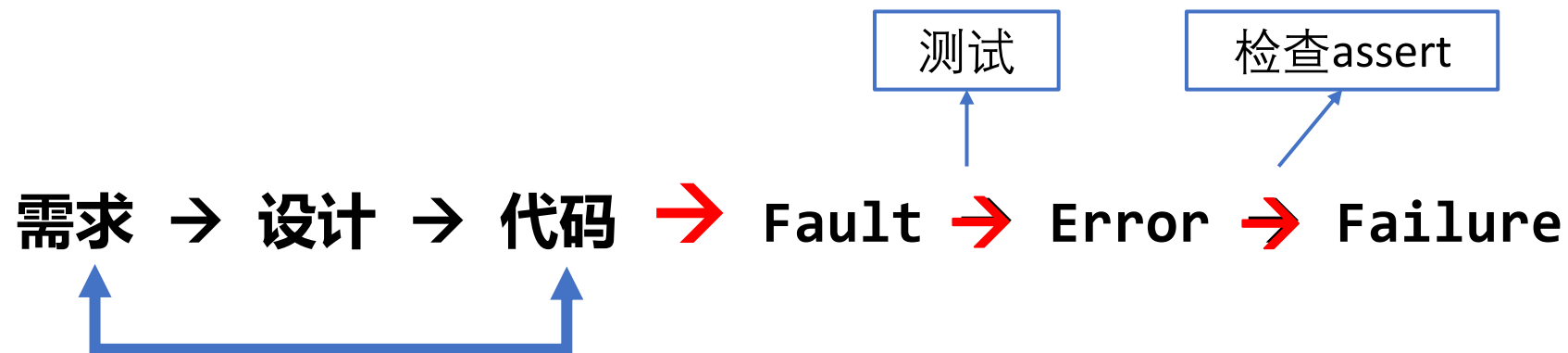
- 二分查找、观察trace、.....

- 调试实践

- 理解程序如何完成对现实世界任务的抽象
 - 我们很容易用一小句话来概括一大段程序执行过程
 - 并且排除/确认其中的问题

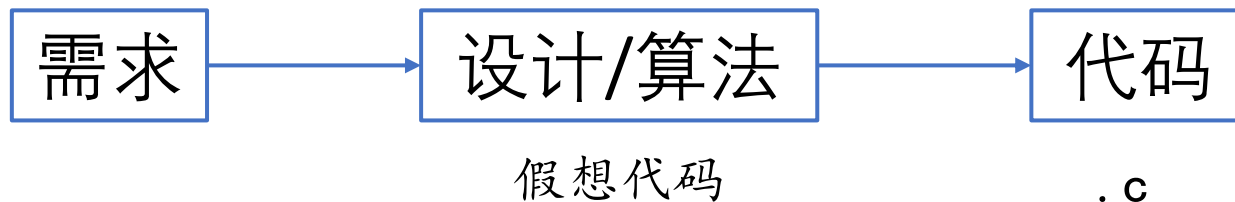
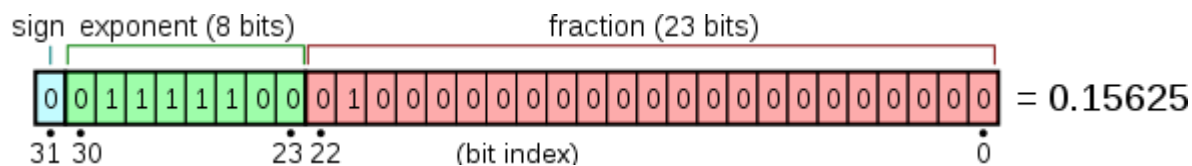


调试理论：



史上著名的bug们

- First bug
- 千年虫
- Intel浮点数bug



End.

不要放弃PA。