Cmocka单元测试实践

莫大

2020/05

https://github.com/kuangtu/cmocka-practice

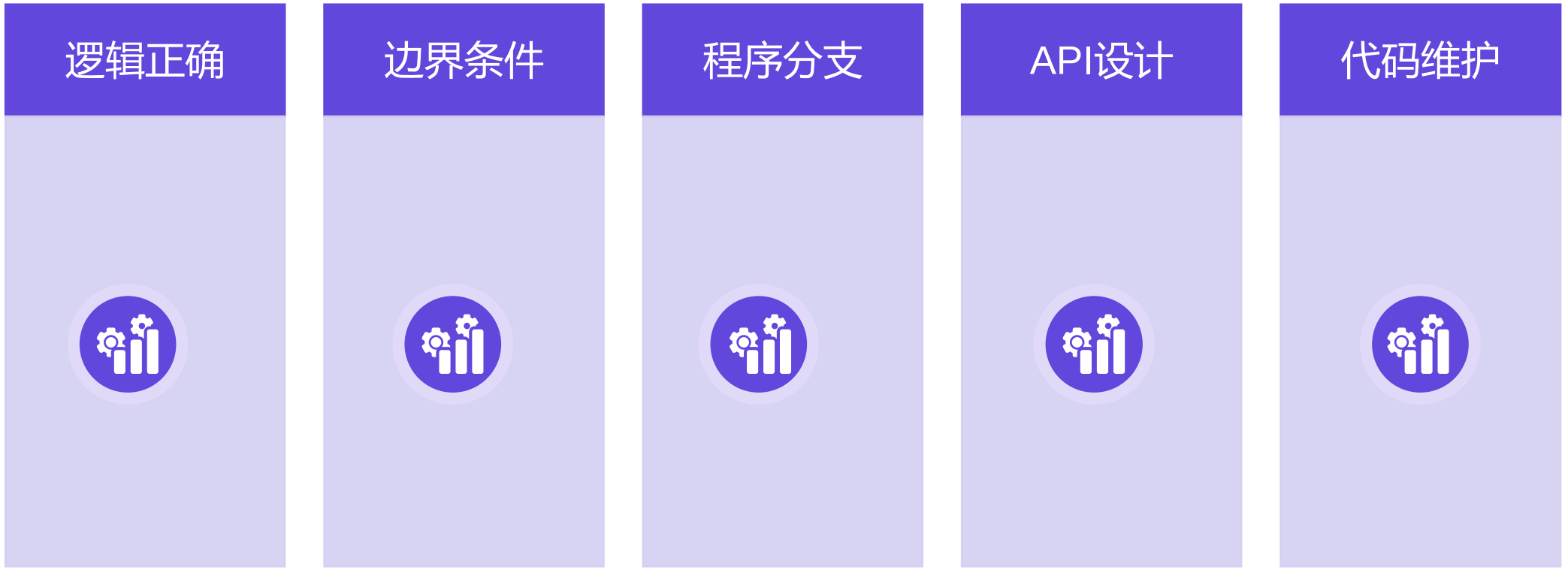
目录

/01

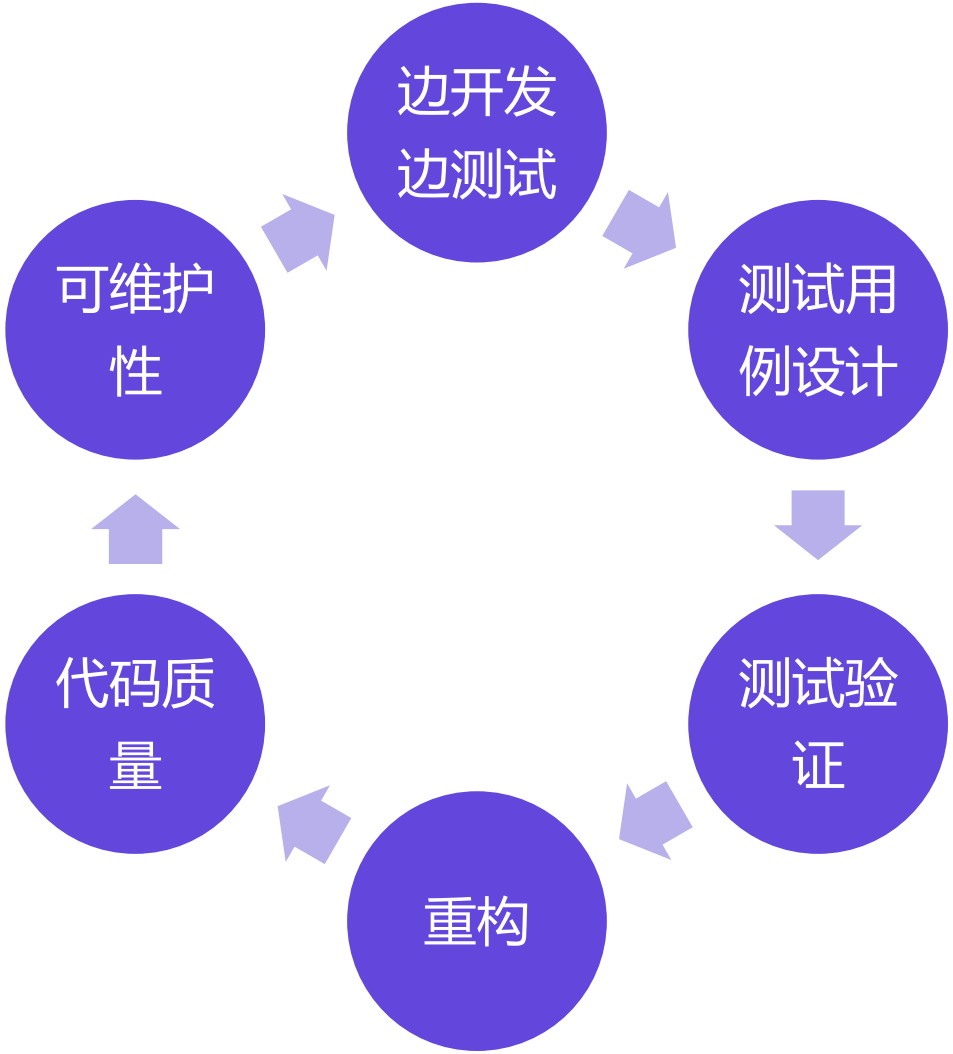
单元测试概述

1.1 什么是单元测试

1.2 为什么需要单元测试



1.3 单元测试过程



1.4 单元测试原则

独立性

- 相互独立
- 不依赖先后顺序

可重复

- 重复运行

BCDE原则

- Border-边界值
- Correct-输出期望
- Design-用例设计
- Error-异常处理

Assert断言

- 避免人工确认
- 通过断言验证

代码重构

- 不可测
- 重构难以避免

/02

Cmocka介绍

2.1Cmocka特性

- 只依赖c库
- 支持mock对象
- 异常处理
- Assert函数
- Setup & teardown
- 跨平台

2.2 基本使用

```c
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
/* A test case that does nothing and succeeds. */
static void null_test_success(void **state) {
    (void) state; /* unused */
}
int main(void) {
    const struct CMUnitTest tests[] = {
        cmocka_unit_test(null_test_success),
    };
    return cmocka_run_group_tests(tests, NULL, NULL);
}
```

/03
工程测试示例

3.1 股票行情接入示例

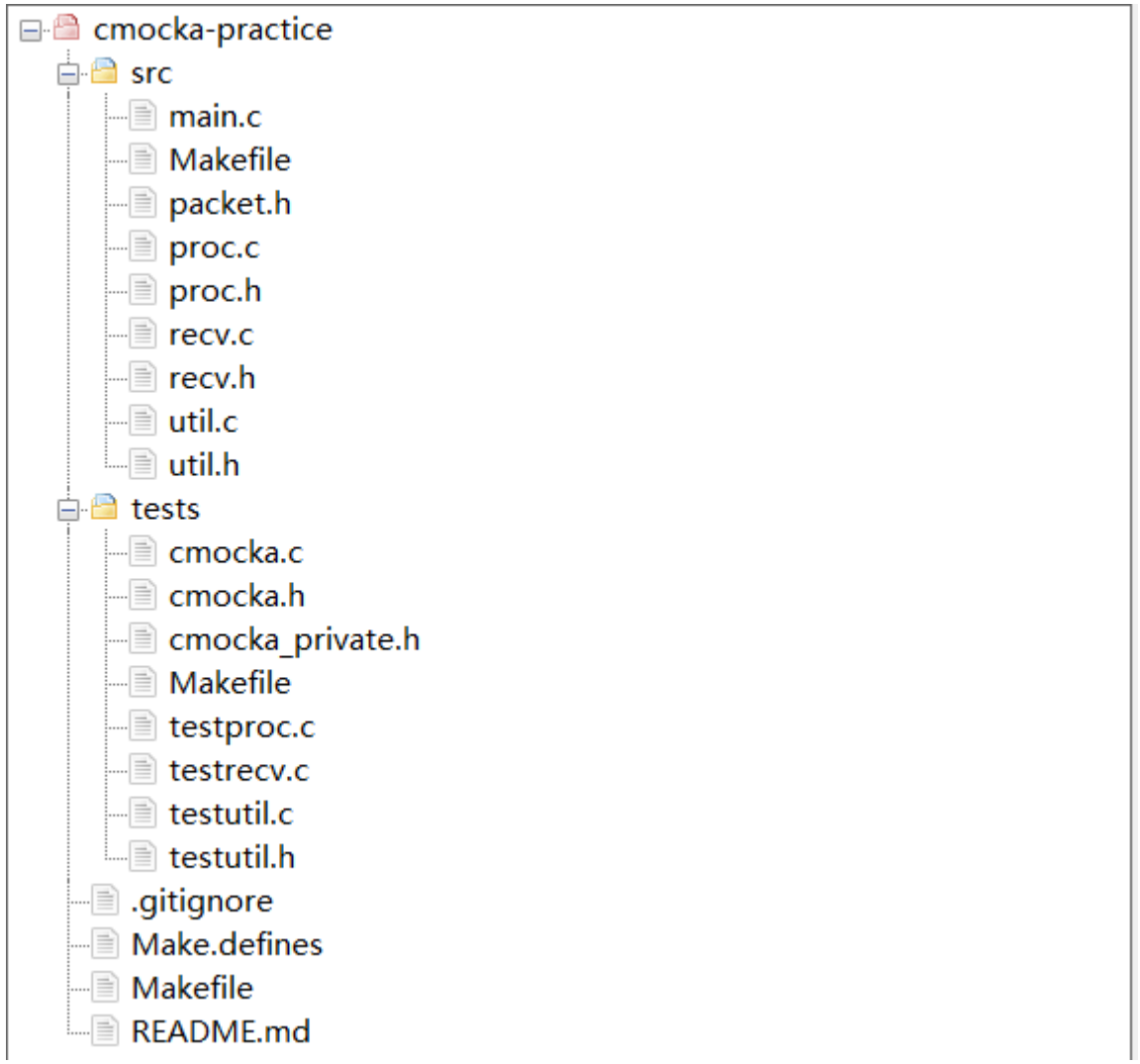
recv.c

- 从socket读取行情
- 判断完整数据包

proc.c

- 行情初始化
- 解析数据包

2.3 工程示例结构



工程结构

- Src & test 分开
- 接收、处理模块不同源文件
- Cmocka文件导入

2.3 makefile结构

```
#通过Make.defines定义对于gcc编译选项
include ../Make.defines
CFLAGS+= -I../src/
PROG=testrecv testproc
#直接将源代码中的.c文件包含进来
MODINC=-I. -I../src
MODLIBS=-lpthread -lrt -ldl
TESTRECVOBJ= testutil.o
TESTPROCOBJ= testutil.o
CMOCKAOBJ=cmocka.o
OBJ1=testrecv.o testproc.o
all:${PROG}
# 包含编译依赖关系
DEPS := ${OBJ1:.o=.d}
-include ${DEPS}
%.o: %.c
        ${CC}  ${CFLAGS} ${MODFLGS} ${MODINC} -MM -MT $@ -MF
$(patsubst %.o,%.d,$@) $<
        ${CC}  ${CFLAGS} ${MODFLGS} ${MODINC} -c $<
# 编译测试模块
testrecv: ${CMOCKAOBJ} ${TESTRECVOBJ} testrecv.o
        ${CC} -Wl,--wrap=ProcHqPkt -o ${DEST}$@ $^ ${CFLAGS}
${MODINC} ${MODLIBS}
testproc: ${CMOCKAOBJ} ${TESTPROCOBJ} testproc.o
        ${CC}  -o ${DEST}$@ $^ ${CFLAGS} ${MODINC} ${MODLIBS}
clean:
        rm -rf *.o
        rm -rf *.d
        rm -rf testrecv
        rm -rf testproc
```

makefile

- 将cmocka源文件编译为.o文件链接
- 包含.c文件相关的依赖关系
- 单元测试模块隔离.
  - 依赖的.o文件通过${TESTRECVOBJ} ${TESTPROCOBJ}分开

2.3 include源文件

- Testrecv.c单元测试文件

```
#include <limits.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <setjmp.h>
#include "cmocka.h"
#include "cmocka_private.h"
#include "packet.h"
#include "proc.h"
#include "testutil.h"
#include "recv.c"
```

测试用例文件include源文件

- 静态函数
- 静态变量
- 其他函数

2.3 函数测试

```
static void
UpdateMsg(MSG_STATS_T *ptMsg){};


static void
testUpdateMsg(void **state)
{
    UNUSED(state);

    MSG_STATS_T tMsg;

    memset(&tMsg, 0, sizeof(MSG_STATS_T));

    tMsg.u32SecurityCode = 3988;
    tMsg.u32LstPrice = 1000;
    tMsg.u32ClsPrice = 1010;
    tMsg.ullSharesTraded = 200;
    tMsg.llTurnover = 180000;

    UpdateMsg(&tMsg);

    SEC_MEM_T *ptSecMem = &gs_tSecMem[3988];

    assert_int_equal(ptSecMem->u32LstPrice, 1000);
    assert_int_equal(ptSecMem->llTurnover, 180000);
}
```

调用函数

输入条件，验证结果

- 构造输入条件
- 函数返回结果（或变量修改后的结果）通过assert断言验证

2.3 mock其他模块的函数

```
int
CopyBuf(char *szBuf, size_t iRead)
{

    uint32_t u32SeqNum = 0;
    //LOG SEQ
    UNUSED(u32SeqNum);

    //判断剩余buffer大小
    if (!IsRemain())
    {
        return BUF_SIZE_ERR;
    }

    //将接收到的数据放置到buffer中
    memcpy(gs_tNetContext.szBuf + gs_tNetContext.iPos, szBuf, iRead);

    //完整数据包的检查
    if(IsFullPkt(gs_tNetContext.szBuf, gs_tNetContext.iPos))
    {

        //更新到内存中
        u32SeqNum = ProcHqPkt(gs_tNetContext.szBuf);

        //TODO根据完整包的大小还需要调整buffer中的长度

        return COPY_BUF_OK;
    }
    else
    {
        return NOT_FULL_PKT;
    }
}
```

测试源文件中调用了模块的函数

- CopyBuf函数中调用ProcHqPkt函数
- ProcHqPkt函数在proc.c源文件中
- CopyBuf函数在recv.c源文件中

2.3 __wrap方式

```
#编译时设置wrap方式

testrecv: ${CMOCKAOBJ} ${TESTRECVOBJ} testrecv.o
          ${CC} -Wl,--wrap=ProcHqPkt -o ${DEST}$@ $^ ${CFLAGS} ${MODINC} ${MODLIBS}
```

```
uint32_t
__wrap_ProcHqPkt(char *szBuf)
{
          UNUSED(szBuf);

          return 0;
}
```

__wrap处理

- 通过gcc编译选项-Wl,--wrap设置
- ProcHqPkt会替换为__wrap_ProcHqPkt
- 单元测试文件中定义__wrap_ProcHqPkt函数
- 原函数仍然可以调用

2.3 mock Objects过程

- Will_return设置函数将数据压入栈
- mock_type函数调用时从栈中取出数据
- 成对出现
- 测试函数中执行will_return
- wrap函数中执行mock_type

2.3 系统调用

```
ssize_t
read(int fd, void *buf, size_t count)
{
    UNUSED(fd);
    UNUSED(count);

    char *szBuf;
    size_t iNum;
    ssize_t iRet;

    szBuf = mock_type(char*);

    iNum = mock_type(size_t);

    iRet = mock_type(ssize_t);

    gs_bRunFlag = mock_type(bool);

    //拷贝到buf中
    memcpy(buf, szBuf, iNum);

    return iRet;

}
```

得到buffer

buffer长度

调用返回

拷贝buffer

```
void
test_RecvFromSrv_ok(void **state)
{
    UNUSED(state);
    int iRet = 0;

    //设置select返回正常
    will_return(select, 1);

    //模拟创建一个消息
    HQ_PKT_HEADER_T *ptPktHeader = (HQ_PKT_HEADER_T*)CreateHqPkt();

    will_return(read, ptPktHeader);

    will_return(read, PKT_HEADER_SIZE + 2 * MSG_STATS_SIZE);

    will_return(read, PKT_HEADER_SIZE + 2 * MSG_STATS_SIZE);

    will_return(read, false);

    //设置read读取的buf
    iRet = ReadFromSrv();
    assert_int_equal(iRet, COPY_BUF_OK);
    FreeHqPkt(&ptPktHeader);

}
```

2.3 边界数据

```
void
testLoadSecData_uncomp(void **state)
{

    UNUSED(state);

    char szFileName[PATH_MAX + 1];
    int iRet = 0;

    //模拟创建一个文件
    memset(szFileName, 0, PATH_MAX + 1);
    strncat(szFileName, DATA_FILE_NAME, strlen(DATA_FILE_NAME));

    FILE *fpData = fopen(szFileName, "w+");
    assert_non_null(fpData);

    char *szLine =
        "00700|40000|0|100|400000|093400\n"
        "03988|12000|0";

    fprintf(fpData, "%s", szLine);
    fclose(fpData);

    //初始化内存行情
    InitSecMem();

    iRet = LoadSecData(szFileName);
    assert_int_equal(iRet, 0);

    uint32_t u32Secode = 3988;
    uint32_t u32LstPrice = 12000;

    SEC_MEM_T secMem = gs_tSecMem[u32Secode];

    //不完整的记录，不会更新
    assert_int_not_equal(secMem.u32LstPrice, u32LstPrice);
    assert_int_equal(secMem.u32LstPrice, 0);

    //删除模拟行情文件
    remove(szFileName);
}
```

记录异常

检验处理结果

删除测试数据文件

模拟输入异常

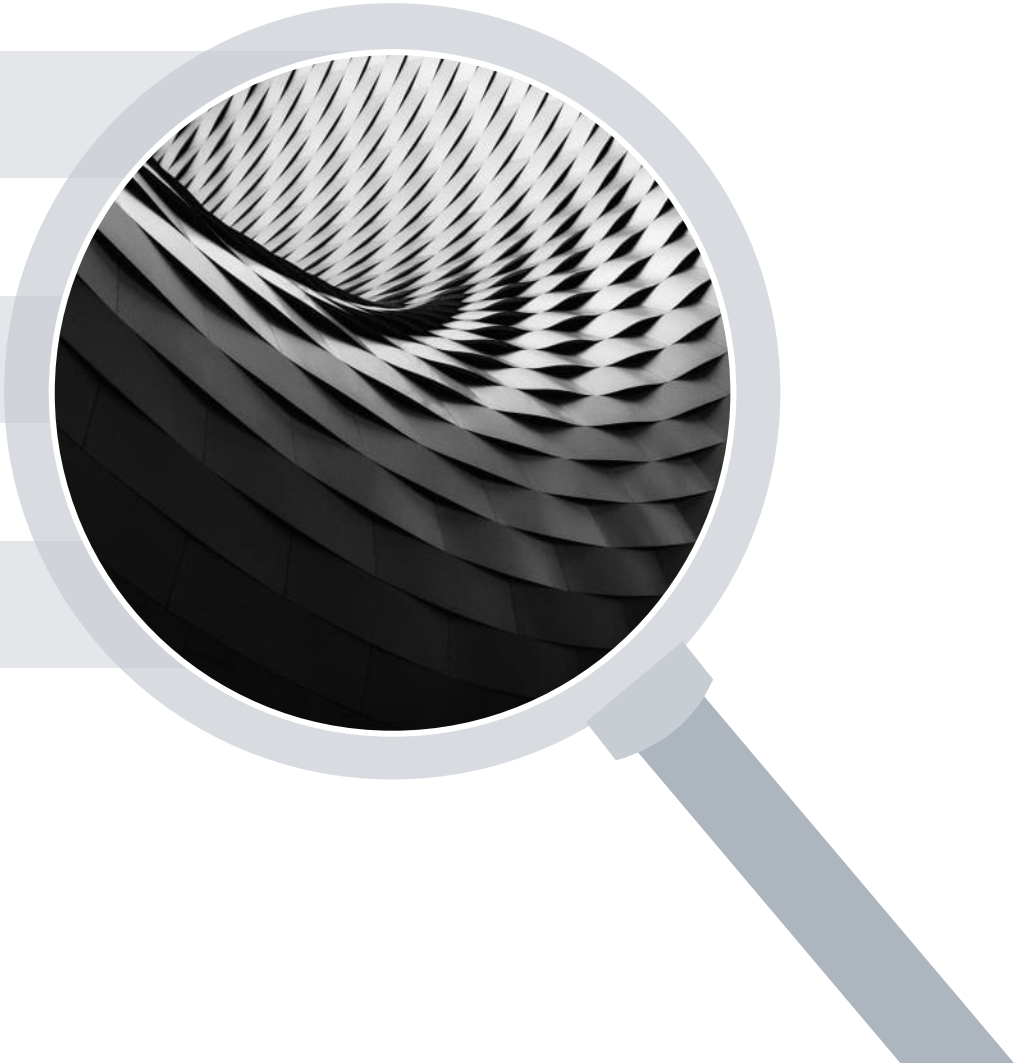
- 构造边界、异常数据测试函数处理的正确性
  - 加载文件完整性
  - 数据范围
  - 数据校验.

/04

单元测试总结

4.1 总结

1. 单元测试支持者
2. 根据需求创建测试用例
3. 不是Test-driven development实践者



GAME OVER