SPASVO

Unit test & memory leak test &other

目录

- 单元测试方法和流程
- 程序中的log
- 内存跟踪工具

例子

- Java的例子

```
class adds {
    public static final int cycle=1;
    public static final int rect = 2;
        public int area(int x, int y, int type) {
            assert(x<0 && y<0);
          if(type==adds.cycle)
              return 3.14*x*x;
          else if(type==add.rect)
              return x*y;
          }
          Public int addone(int x) {
              return x+1;
          }
}
```

```
class  oper {
    public int cacu(String sx, String sy) {
            int x;
            Int y;
            int ret =0;
            try {
                    x = Integer.parse(sx);
                    y = Integer.parse(sy);
            }catch(Exception ex) {
                    Return 0;
                    }
            if(x<y)
                    Return x;
            for(int i=x; i<y; i++j) {
                    ret +=I;
            }
            return ret;
            }
    public int cacu2(String sx) {
    {
            adds instance = new adds();
            return cacu(instance.area(Integer.parse(sx), 1),
                "100");
    }
```

- **测试class adds**

```
public class testadds {
    public boolean test() {
        int x=0;
        int y=0;                                        设置输入
        int type = adds.cycle;
        adds instance = new adds();                     执行
        int ret = Instance.area(x,y,type);
        if(ret == 0)
            return true;                                预期结果比对
        else
            return false;
    }
    ....
```

设置多组覆盖用例进行测试

```
public int cacu2(String sx) {
{
      adds instance = new
    adds();
      return
         cacu(instance.area(Integer.
         parse(sx), 1), “100”);
}
```

- **编写桩**

```
class adds {
    public area(int x,y) {
        return 20;
    }
```

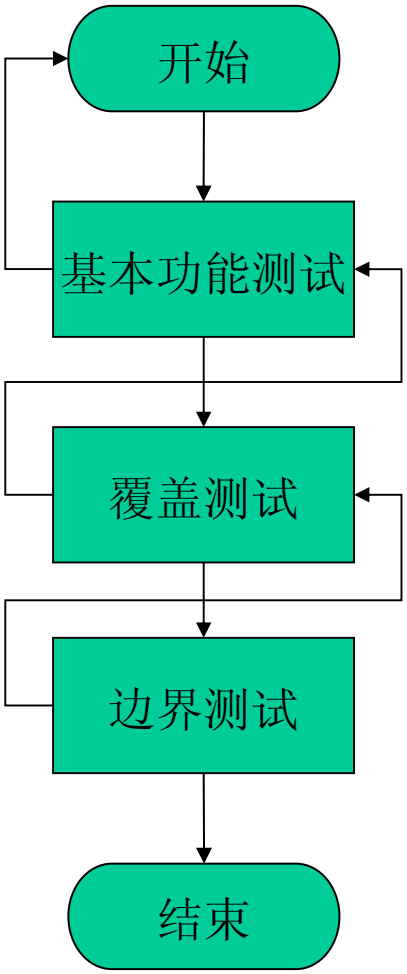
- **测试代码**

```
class testoper {
    public boolean test() {
        …..
    }
```

单元测试目标

- 测试目标
  - 实现100%的代码覆盖、分支覆盖
  - 边界条件覆盖
- 技术
  - 代码覆盖
  - 分支覆盖
  - 路径覆盖
  - 断言(assert)

单元测试方法



- 基本功能测试
  - 主要的功能数据输入
  - 主要的功能执行正确
  - 主要的输出数据检查正确
- 覆盖测试
  - 分支分析
  - 编写对分支进行覆盖的驱动和测试数据
  - 达到100%覆盖
- 边界测试
  - 设计数据边界条件
  - 执行测试

单元测试方法

- 基本功能测试
  - 描述功能
  - 主要功能数据分析
  - 编写测试框架
    - 试用现有的测试工具：junit，c++unit等
    - 自己编写驱动模块
    - 自己开发工具来实现（代码扫描工具）
  - 编写必要的桩模块
  - 测试用例设计与编写
    - 输入数据
    - 预期输出
  - 通过测试框架来执行测试用例
    - 执行多个测试用例
    - 分析测试结果

单元测试方法

- 覆盖测试
  - 分析单元的分支（根据流程图/程序代码/试用工具），结果：单元分支文档
  - 根据分支来设计测试用例
    - 输入数据
    - 预期结果
  - 方法（如何知道分支被测试？）
    - 插入log代码标记分支
    - Debug
    - 测试工具

单元测试方法

- 边界测试
  - 根据边界数据来设计测试用例
    - 输入数据
    - 预期结果
  - 方法（如何知道分支被测试？）
    - 插入log代码标记分支
    - Debug
    - 测试工具

测试用例

- 测试用例汇总:
  - 基本功能测试用例
  - 分支覆盖测试用例
  - 边界条件测试用例

覆盖设计技术分类

- 针对单元接口规则的用例设计方法
  - 等价类分析法
  - 边界值分析法
  - 错误猜测法
- 针对单元内部逻辑控制流程的用例设计方法：
  - 语句覆盖
    - 测试用例覆盖所有的执行语句
  - 分支
    - 测试用例覆盖所有执行分支
  - 条件
    - 覆盖条件的各种数据情况
  - 分支条件
    - 分支+条件
  - 路径
    - 根据流程图设计路径覆盖

测试用例编写基本方法

- 冒烟测试
- 正向测试
- 逆向测试
- 特殊需求测试
- 代码覆盖
- 覆盖率指标测试

SPASVO

防止接口错误技术

- 定义Interface:

- Public interface term{
  Public boolean connet(String ip,
  int port);
  Public boolean disconnect();
  Public void type(int key);
  Public int getCursor();
  }

- 调用类:
  - Term ss = new VT100Term();
  - ss.connect(localhost, 1010);
  - Ss.type(10);
  - Ss.disconnect();
- 定义类:
  - Public class VT100Term implements term {
    - Public boolean connect(..)..
    - Public boolean disconnect()…
    - Public void type(..)…
    - Public int getCursor()…

Sample：使用Junit进行单元测试

- JUnit 是java单元测试工具
- 面向class, 自动生成测试框架
- JUnit不能生成测试用例、不能进行测试用例辅助设计
- JUnit不能够进行桩模块的自动生成

目录

- 单元测试方法和流程
- 程序中的log
- 内存跟踪工具

使用log

- 为什么要使用Log?
  - 执行分支的标志：程序执行的时候，执行到了哪些分支？
  - 关键数据察看：数据是否和预期一致
  - Runtime bug分析：运行系统出错了，log可以知道：
    - 执行到什么地方出错了？
    - 什么环境下出错了？
    - 可能会知道：为什么出错了？
  - Warnning

LOG的级别

- Error：针对出现的错误，记录日志
  - 空指针
  - 数据库异常
  - 逻辑错误
  - 导致系统即将退出的error
- Warning
  - 边界（memory边界、hashtable边界等）
- Log
  - 变量的value
  - 状态的改变
  - Event发生

LOG常见问题

- LOG太多，很难看出问题;
- LOG太乱，不知道在做什么？
- LOG会影响效率

SPASVO

C++ Log solution

- Java的solution: log4j
- C++的solution:
  - 面向class的log控制

```
#ifndef _MENU_CLASS_
#define _MENU_CLASS_


class TMenu{
public:
    static bool                         debug;
    static char
    command[MAXNAMELEN_MENU];

    private:
    static char * get_key(char *name);
    void SelectNextItem();
    int MenuAction();
    int PickPreItem();
    int PickNextItem();
    void SetSubMenuSelected(char cFlag);
    void BottomMenu();
    int menu_action(char *action_buffer);
    int nAcclKeyProc(int nkey );

public:

    TMenu(int nX0, int nY0, int itemNum, int
nSpaceWidth, int nItemsNumInWin);
    TMenu();
    ~TMenu();
    。。。。};


#endif
```

SPASVO

C++ Log solution

- 面向class的log参数管理
- 面向class方法的标准log格式
- 使用条件编译控制log的出现频率，可以得到多个不同的运行版本:
  - 开发版
  - 运行版

```
void TField::setValue(int value)
{
        #ifdef LX_DEBUG
        if(my_debug && debug) {
                fprintf(dbp,
"TField(%s)::setValue(%d) ,start...\n",
                                this->getName(),
value);
                fflush(dbp);
        }
        #endif

        if(this->cValueType == INTEGER_MARK)
                memcpy(this->pvFieldValue, &value,
sizeof(int));
        else if(this->cValueType == DECIMAL_MARK)
        {
                TDecimal dd;
                char tmp[100];
                sprintf(tmp, "%d", value);

                dd.parse(tmp, strlen(tmp));
                memcpy(this->pvFieldValue, &dd,
sizeof(TDecimal));
        }
        else
                MessageBox(10, 10, "setValue error:
not int type");

        #ifdef LX_DEBUG
        if(my_debug && debug) {
                fprintf(dbp, "TField::setValue(%d)
end\n", value);
                fflush(dbp);
        }
        #endif
}
```

Log sample：控制log

- 主程序启动之后设置log:

```
commAdapter::debug = false;
        TField::debug = 1;
        TScreen::debug = 0;
        TGrid::debug = 1;
        TRuntime::debug = 1;
        editdebug=1;
        TFieldEditor::debug = 1;
```

SPASVO

Log samples

- TField::NewFieldByName(TXPROP_SUBSYS) start...
- pattern.cpp::GetDefaultPattern(,S,8,0,) start...
- string|dbchar|hex
- nLength:8
- string|dbchar|hex
- nLength:8
- pattern.cpp::GetDefaultPattern(,S,8,0,&&&&&&&&&) end
- pattern.cpp::GetDefaultPattern()=>0
- delete bundle(/home/cbod/app/dec/fld/TXPROP_SUBSYS) start
- delete bundle ok
- TField::NewFieldByName(TXPROP_SUBSYS) end

SPASVO

Log samples——通讯错误

- Connected on Terminal: NOJS
- System:CICSTCP
- send buffer----------
- 84[T] 67[C] 67[C] 66[B] 57[9] 57[9] 57[9] 57[9] 32[ ] 0[^@] 111[o] 5
- 1[3] 49[1] 48[0] 50[2] 56[8] 48[0] 48[0] 48[0] 48[0] 65[A] 48[0] 49[1
-] 48[0] 48[0] 49[1] 67[C] 77[M] 48[0] 57[9] 57[9] 57[9] 57[9] 48[0]
- 48[0] 48[0] 48[0] 51[3] 49[1] 48[0] 50[2] 56[8] 48[0] 48[0] 48[0] 48[
- 0] 48[0] 48[0] 49[1] 0[^@] 0[^@] 0[^@] 0[^@] 48[0] 49[1] 0[^@] 0[^@]
- 0[^@] 0[^@] 0[^@] 0[^@] 0[^@] 0[^@] 56[8] 48[0] 48[0] 48[0] 32[ ] 32
- [ ] 32[ ] 32[ ] 32[ ] 32[ ] 32[ ] 32[ ] 32[ ] 32[ ] 32[ ] 50[2] 32[ ]
- 32[ ] 32[ ] 32[ ] 32[ ] 32[ ] 32[ ] 32[ ] 0[^@] 0[^@] 4[^D] 0[^@] 5
- 1[3] 49[1] 48[0] 50[2] 56[8] 48[0] 48[0] 48[0] 48[0] 48[0] 48[0] 49[1
-] 57[9] 48[0] 53[5] 53[5] 54[6] 57[9] 54[6] 62[>] ----------

- Starting TCCB on CICSTCP
- Event type CICS_EPI_EVENT_SEND Received from CICS
- End of transaction; end reason CICS_EPI_TRAN_NO_ERROR
- Terminal deleted; end reason CICS_EPI_END_SIGNOFF
- EPI returned: CICS_EPI_NORMAL
- bHaveRecvData:1
- bHaveRecvData:1

使用log的意义

- Runtime下，知道错误发生时刻的信息:
  - 当前运行到什么地方（代码行、函数）;
  - 数据是什么;
  - 大概的错误原因;
  - 对于维护人员而言，log具有非常重要的意义;
- 开发环境下:
  - 错误原因;
  - 程序执行分支;
  - 执行测序是否和预期一致?
  - 执行分支是否和预期一致?
- 根据log进行单元测试（比对期望值）

目录

- 单元测试方法和流程
- 程序中的log
- 内存跟踪工具

Unix系统下的开发

- 内存导致的错误:
  - 越界：数据错误
  - 访问非法空间：coredump
  - 没有释放内存：out of memory

sampe1

```
void main() {
    char s1[10];
    char s2[8];

    strcpy(s2, “12345”);
    strcpy(s1, “0123456789”);
    printf(“s1:%s\n”, s1);
    printf(“s2:%s\n”, s2);
    }
```

sample2

- void fun1(int a, int b, int c) {
  - struct st1 *S1 = (struct st1) malloc(sizeof(struct st1));
  - If(S1==NULL)
    - Return NULL;
  - S1.a = a;
  - S1.b = b;
  - S1.c = c;
  - Return s1;
- }
- void call() {
  - Struct st1 *s1 = fun1(1,2,3);
  - If(s1->a + s1->b+s1->c > 10) {
    - pirntf(“ok”);
    - free(s1);
  - }
  - else
    - printf(“false”);
- }

内存跟踪工具

- Valgrind（linux）

- boid f() {
  - Int* x= malloc(10 * sizeof(int));
  - x[10] = 0;
- }
- Int main(void) {
  - f();
  - return 0;
- }

valgrind

- gcc –Wall example.c –g –o example
- valgrind –tool=memcheck –leak-check=yes ./example

运行，得到日志

- 上面的C程序存在两个错误：1. 数组下标越界；2. 分配的内存没有释放，存在内存泄露的问题。

```
==6742== Invalid write of size 4
==6742==    at 0x8048384: f (example.c:6)
==6742==    by 0x80483AC: main (example.c:12)
==6742==  Address 0x1B908050 is 0 bytes after a block of size 40 alloc'd
==6742==    at 0x1B904984: malloc (vg_replace_malloc.c:131)
==6742==    by 0x8048377: f (example.c:5)
```

```
==6742== malloc/free: 1 allocs, 0 frees, 40 bytes allocated.

.....

==6742== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6742==    at 0x1B904984: malloc (vg_replace_malloc.c:131)
==6742==    by 0x8048377: f (example.c:5)
==6742==    by 0x80483AC: main (example.c:12)
```

使用内存跟踪程序

- 内存跟踪程序：
  - 跟踪每一块被分配的内存（给内存命名）
  - 发现没有被释放的内存
  - 发现重复释放的内存错误
  - 发现越界的操作

SPASVO

C++内存跟踪类

```
#ifndef _DEBUG_CLASS_H
#define _DEBUG_CLASS_H

#include <stdio.h>

#define DFREE(a) { int rr = debug::dfree(a); \
                                        if(my_debug && rr !=0 ) { \
                                        fprintf(dbp, "%s:%d ptr:%d, dfree ret:%d, \n", \
                                                                __FILE__, __LINE__, (int)a, rr); \
                                        fflush(dbp); } }\


#define MAXFILEPATH 128
//#define MAXMEMSIZE                    200000
#define MAXMEMSIZE                      20000

typedef struct MemInfo_t {
                bool    ibUsed;
                char                    *ptr;
                //char                  name[128];
                char                    name[32];
                size_t                  size;
} MEMINFO;


class debug {
public:
                static bool sbInit;
                static char sszDebugFile[MAXFILEPATH];
                static FILE *sfpdb;
                static int debug_mem;

                #ifdef MEM_DEBUG
                static MEMINFO          ameminfo[MAXMEMSIZE];
                static MEMINFO  meminfo[MAXMEMSIZE];
                static int                                              nMaxCount;
                #endif

public:
                static bool IsDebugInit();
                static void DebugInit();
                static void DebugStart(char *iFileName);
                static void DebugEnd();
                static void DebugOutputLevel(int level, char *fmt,...);
                static void DebugOutput(char *fmt,...);
                static void init_meminfo();
                static void * dcalloc(size_t len,size_t size,char *name);
                static void * dmalloc(size_t size,char *name);
                static void * realloc(void* ptr, size_t newsize);
                static int dfree(void *buff);
                static int get_free_mem_index();
                static int get_mem_from_ptr(void *ptr);
                static int check_ptr(void *ptr, char *msg);
                static void check_meminfo();
private:
                static int debug::AdjustStringPos(int width, char *src, char *dst);
};

#endif
```

Debug.cpp

- void debug::init_meminfo()---初始化内存跟踪模块
- void * debug::dcalloc(size_t len,size_t size,char *name)----代替calloc（ ）;
- void * debug::realloc(void* ptr, size_t newsize)---代替recalloc（ ）;
- void * debug::dmalloc(size_t size,char *name)----代替malloc（ ）
- int debug::dfree(void *ptr)---代替free（一般使用DFREE宏）

内存跟踪程序的功能

- 发现内存泄露
- 发现内存重复释放
- 发现数据越界、数据错误

- 实际使用效果：开发了300个交易，发现1个内存错误(与动态连接库相关的资源未释放).

SPASVO

其他: 使用STL减少代码错误

- 以前:
- Class aa {
  - Private:
    - Int *link;
  - Public:
  - void push_back(int i);

  }

  Class bb {

  private:

  string *link;

  public:

  void push_back(string ss) ..

  }

- STL:

- Vector<int> aa;
- aa.push_back(10);
- Aa.push_back(11);

- Vector<string> bb;
- bb.push_back(“first”);
- bb.push_back(“second”);