内存泄漏

内存泄漏的原因？

内存泄漏是在没有自动 gc 的编程语言里面，经常发生的一个问题。因为没有 gc，所以分配的内存需要程序员自己调用释放。其核心原因是调用分配与释放没有符合开闭原则，没有配对，形成了有分配，没有释放的指针，从而产生了内存泄漏

```
{
    void *p1 = malloc(s2);
    void *p2 = malloc(s1);
    free(p1);
}
```

以上代码段，分配了 s1,s2 大小的内存块分配由 p1 与 p2 指向。而代码块执行完以后，释放了 p1，而 p2 没有释放。形成了有分配没有释放的指针，产生了内存泄漏。

内存泄漏会产生哪些后果？

随着工程代码量越来越多，自然内存泄漏的排查就成为了一个很头疼的问题。有分配没有释放，自然会使得进程堆的内存会越来越少，直到耗尽。会造成后面的运行时代码不能成功分配内存。

内存泄漏如何解决？

内存泄漏是没有自动 gc 的编程语言所产生的，解决方案一，引入 gc。这是根治内存泄漏的最好的方案。但是这样的方案有失去了 c/c++语言的优势。方案二，当发生内存泄漏的时候，能够精准的定位代码哪一行所引起的。这也是我们实现内存泄漏检测的如何核心实现需求。

需求 1：能够检测出来内存泄漏

需求 2：能够判断是由代码哪一行引起的内存泄漏

内存泄漏检测如何实现？

内存泄漏是由于内存分配与内存释放，不匹配所引起的。对内存分配函数

malloc/calloc/realloc，以及内存释放 free 进行“劫持”hook。能够统计出内存分配的位置，内存释放的位置，从而判断是否匹配。

方案一：采用__libc_malloc, __libc_malloc 与__builtin_return_address

方案二：采用宏定义

方案三：借助 malloc.h 里面 malloc_hook

方案一：

```
extern void *__libc_malloc (size_t bytes);
int enable_malloc_hook = 1;

extern void __libc_free (void *p);
int enable_free_hook = 1;

void *malloc(size_t bytes) {
    //printf("malloc\n");
    if (enable_malloc_hook) {
        enable_malloc_hook = 0;

        void *p = NULL;
        p = __libc_malloc(bytes);

        char buff[256];
        sprintf(buff, "%p.mem", p);
        printf("malloc: %s\n", buff);

        void *caller =  __builtin_return_address(0);

        FILE *fp = fopen(buff, "w");
        fprintf(fp, "[%p]: addr: %p, size: %lu byte(s)\n", caller, p,
bytes);
        fflush(fp);
        //fclose(fp); //

        enable_malloc_hook = 1;

        return p;

    }

    return __libc_malloc(bytes);

}
```

```
void free(void *p) {
    if (enable_free_hook) {
        enable_free_hook = 0;

        char buff[256];
        sprintf(buff, "%p.mem", p);

        printf("hook_free: %s\n", buff);

        if (unlink(buff) < 0) {
            printf("Double free: %p\n", p);
            return ;
        }

        __libc_free(p);
        enable_free_hook = 1;

    } else {
        __libc_free(p);
    }

}
```

方案二：

```
void *hook_malloc(size_t bytes, const char *file, int line) {

    void *p = malloc(bytes);

    char buff[256];
    sprintf(buff, "%p.mem", p);
    printf("hook_malloc: %s\n", buff);

    //void *caller =  __builtin_return_address(0);

    FILE *fp = fopen(buff, "w");
    fprintf(fp, "[%s:%d]: addr: %p, size: %lu byte(s)\n", file, line, p,
bytes);
    fflush(fp);
```

```
    fclose(fp); //

    return p;
}

void hook_free(void* p, const char *file, int line) {

    char buff[256];
    sprintf(buff, "%p.mem", p);

    printf("hook_free: %s\n", buff);

    if (unlink(buff) < 0) {
        printf("Double free: %p\n", p);
        return ;
    }

    free(p);

}



#define malloc(size)   hook_malloc(size, __FILE__, __LINE__)
#define free(p)        hook_free(p, __FILE__, __LINE__)
```

方案三：

```
typedef void *(*malloc_hook_t)(size_t size, const void *caller);
typedef void  (*free_hook_t)(void *ptr, const void *caller);

malloc_hook_t old_malloc_f = NULL;
free_hook_t old_free_f = NULL;

void *malloc_hook_f(size_t size, const void *caller) {

    mem_untrace();

    void *ptr = malloc(size);
    printf("+%p: addr[%p]\n", caller, ptr);
```

```
    mem_trace();

    return ptr;
}

void free_hook_f(void *ptr, const void *caller) {

    mem_untrace();

    printf("-%p: addr[%p]\n", caller, ptr);
    free(ptr);

    mem_trace();
}

int replaced = 0;

void mem_trace(void) {
    replaced = 1;

    old_malloc_f = __malloc_hook;
    old_free_f = __free_hook;

    __malloc_hook = malloc_hook_f;
    __free_hook = free_hook_f;

}

void mem_untrace(void) {

    __malloc_hook = old_malloc_f;
    __free_hook = old_free_f;

    replaced = 0;
}
```

执行效果

```
king@ubuntu:~/share/0voice2109/3.2.5_memoryleak$ ./mem
+0x40075e: addr[0x1547010]
-0x40076e: addr[0x1547010]
king@ubuntu:~/share/0voice2109/3.2.5_memoryleak$
king@ubuntu:~/share/0voice2109/3.2.5_memoryleak$
king@ubuntu:~/share/0voice2109/3.2.5_memoryleak$ addr2line -f -e ./mem -a 0x40075e
0x000000000040075e
main
/home/king/share/0voice2109/3.2.5_memoryleak/mem.c:187
king@ubuntu:~/share/0voice2109/3.2.5_memoryleak$
```