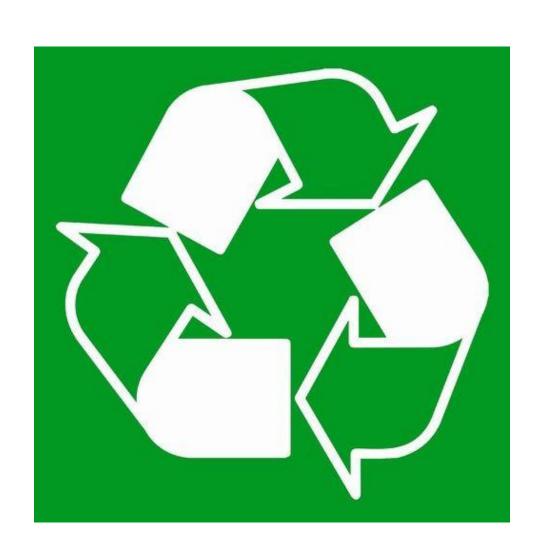
如何实现垃圾一样的垃圾回收——GC的坑爹简介

Liutos mat dot liutos at gmail dot com

什么是垃圾回收?



什么是垃圾回收?

- 维基百科:
 http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- 自动化内存管理的一种方式
- 回收程序中被占据但是不再使用的内存区域

垃圾回收的方法有多少种?



垃圾回收的方法有多少种?

- reference counting
- naive mark-and-sweep
- stop-and-copy
- •

reference counting

- 例子: Unix的硬链接
- 当对象A被对象B引用时,就将A的引用计 数增加一。
- · 当对象B解除对B的引用时,就将A的引用 计数减一。

reference counting

```
    typedef struct foobar t *foobar;

• struct foobar t {
  int ref cnt;
    foobar t foo;
foobar t bar;

    foobar t mk foobar (foobar t, foobar t);

  foobar t a = mk foobar(NULL, NULL); // a->ref cnt = 0
  foobar t b = mk foobar(a, a); // a->ref cnt = 2
  b->ref cnt; // 0
```

reference counting——计数

```
foobar t mk foobar (foobar t foo, foobar t bar) {
  foobar t fb = malloc(sizeof(struct foobar t));
  fb \rightarrow rec cnt = 0;
  fb \rightarrow foo = foo;
  if (foo != NULL)
    foo->ref cnt++;
  fb->bar = bar;
  if (bar != NULL)
    bar->ref cnt++;
  return fb;
```

reference counting——回收

```
// Call it only when fb->rec cnt is zero
void reclaim(foobar t fb) {
  assert(fb->ref cnt == 0);
  foobar t foo = fb->foo;
  foobar t bar = fb->bar;
  free (fb);
  if (--(foo->ref cnt) == 0)
    reclaim (foo);
  if (--(bar->ref cnt) == 0)
    reclaim(bar);
```

reference counting——缺点

- 太麻烦了!
- 遇到循环引用就无法回收了

naïve mark-and-sweep

- 当对象A被对象B引用时,将A标记为已用。
- 扫描所有已分配的内存,回收所有没有在上一个过程中被标记的对象所占据的空间。

naïve mark-and-sweep

```
typedef struct foobaz_t *foobaz;
struct foobaz_t {
    int markp;
    foobaz_t foo;
    foobaz_t baz;
}

#define TRUE 1
#define FALSE 0
```

naïve mark-and-sweep——标记

```
• // Assume the fb has been referenced
• void mark(foobaz_t fb) {
• if (fb == NULL)
• return;
• fb->markp = TRUE;
• mark(fb->foo);
• mark(fb->baz);
• }
```

naïve mark-and-sweep ——标记重捕法

项目	白色蛾			黑色蛾		
地区	释放数	回收数	回收率	释放数	回收数	回收率
伯明翰	64	16	25.0%	154	82	53.2%
多塞特	393	54	13.7%	406	19	4.7%

naïve mark-and-sweep——扫清

```
• #define HEAP SIZE

    struct foobaz t object heap[HEAP SIZE];

    // Scan from the beginning of allocated memory

void sweep (void) {
    for (int i = 0; i < HEAP SIZE; i++) {
      foobaz t fb = object heap[i];
      if (fb->markp = FALSE)
        reclaim(fb); // Another reclaim
```

naïve mark-and-sweep——有错好吗?!

```
• #define HEAP SIZE

    struct foobaz t object heap[HEAP SIZE];

    // Scan from the beginning of allocated memory

void sweep (void) {
    for (int i = 0; i < HEAP SIZE; i++) {
      foobaz t fb = object heap[i];
      if (fb->markp = FALSE)
        reclaim(fb); // Another reclaim
```

naïve mark-and-sweep——回顾

```
foobaz_t mk_foobaz(foobaz_t foo, foobaz_t baz) {
  foobaz_t fb = my_malloc();
  fb->markp = FALSE;
  fb->foo = foo;
  fb->baz = baz;
  return fb;
}
```

• 初始化时是标记为FALSE的,因此在标记阶段过后,光凭markp字段 没办法区分一个内存块是否被分配过。这样到了扫清阶段后,就会有 从来没被用过的内存被进行回收操作。

naïve mark-and-sweep——修改

```
• struct foobaz t {
    int used;
int markp;

    foobaz t foo;

    foobaz t baz;

void sweep (void) {
    for (int i = 0; i < HEAP SIZE; i++) {
      foobaz t fb = object heap[i];
      if (fb->used == TRUE && fb->markp = FALSE)
        reclaim(fb); // Another reclaim
```

相比reference counting的优点

- 平时不需要管
- 可以回收掉循环引用的东东
- 有很大的改进潜力

External Links

http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

END