Garbage Collection

计64 嵇天颖 2016010308

---

1 手动垃圾回收管理

语言：C++

文件：main.cpp

数据集：wiki0.01.txt

编译指令：

```
//compile
g++ main.cpp -o main.out -std=c++11 -O2
//run
./main.out datasets/datasets/wiki0.01.txt
```

设计思路：

- 首先用 `C++` 实现 `wordcount.go` 的功能
- 用 `MyString` 封装 `char*`，类似 `C++11` 的智能指针实现
  - 用 `MyString` 存储每一个 `word`
  - 构造函数和析构函数处是执行 `malloc` 和 `delete` 的地方
- 插桩法实现 `malloc` 和 `delete` 的时间记录
  - 将 `malloc` 封装为 `newchar`
    - 在执行 `new char[]` 之前和之后分别调用 `c++11` 中 `steady_clock::now()`
    - 用全局变量 `steady_clock::duration mallocTime` 来记录 `malloc` 时间
  - 将 `delete` 封装为 `mydelete`
    - 类似 `malloc` 实现，用 `steady_clock::duration deleteTime` 记录 `delete` 时间

实验结果

- 测试 `Overall Execution Time` 与 `malloc & delete` 开销，结果如下

```
(base) jitianyingdeMacBook-Pro:WordCount jitianying$ time ./main.out datasets/datasets/wiki0.01.txt
num origin words: 19008043
num filtered words: 16419850
num different words:365396
mallocTime:14.4797
deleteTime:12.3931

real    1m12.243s
user    1m11.077s
sys     0m1.061s
```

- 我尝试测试内存占用，但发现在 `MacOS` 上一时无法得知申请的虚拟内存大小，于是我实现了对占用的物理内存的大小的测试
  - 测试占用的物理内存(添加如下代码)：

```
#include <sys/resource.h>
size_t getPeakMemory() {
    struct rusage r_usage;
  getrusage(RUSAGE_SELF, &r_usage);
  return (r_usage.ru_maxrss / 1024L);
}
/*....*/
cout << "Memory used: " << getPeakMemory() << endl;
```

  - 测试结果：

```
(base) jitianyingdeMacBook-Pro:WordCount jitianying$ ./main.out
/*.....*/
Memory used: 1219028
```

---

2 引用计数垃圾回收

2.1 Version1

语言： C++

文件： rc.cpp

数据集： wiki0.01.txt

编译指令：

```
//compile
g++ rc.cpp -o main.out -std=c++11 -O2
//run
./rc.out datasets/datasets/wiki0.01.txt
```

设计思路：

- 沿用手动计数的代码继续操作
- 为 `MyString` 添加成员 `int *refer`，用这个来记录每个指针的引用计数
  - 因为我使用了指针来进行引用计数，所以我也关心这个引用计数指针的代价
  - 我添加了两个函数 `newint` 和 `referDelete` 用于对引用计数进行 `malloc` 和 `delete` 的封装
  - 现在我来考虑引用计数的问题：
    - 每次构建新指针的时候引用计数设置为 `1`
    - 如果用一个 `MyString` 来构建另一个 `String`，就相当于 `new` 了一个 `u =v` 这种操作

      我们需要将 `a` 的 `refer` 赋给 `this->refer`，因为还有 `new` 的操作，所以还要 `++` 一次

      ```
      MyString (const MyString &a) {
              this->s = a.s;
              this->len = a.len;
              this->refer = a.refer;
              (*refer)++;
              referCount++;
      }
      ```
  - 记录改变次数就是在每次 `newint` 和 `referDelete` 调用的时候记录

测试结果：

- 我们测试 `Overall Execution Time` 与 `space usage` 与 `malloc & delete` 开销，结果如下：

  (我认为引用计数这个东西在我的实现中也会给程序带来一定的垃圾，所以我也记录了引用计数的分配和回收所消耗的时间)

  ```
  (base) jitianyingdeMacBook-Pro:WordCount jitianying$ time
  ./rc.out datasets/datasets/wiki0.01.txt
  num origin words: 19008043
  num filtered words: 16419850
  num different words:365396
  ```

```
mallocTime:3.28955
deleteTime:1.8189
referMallocTime:3.40622
referDeleteTime:1.80192
total mallocTime:6.69577
total deleteTime:3.62082
Modify Reference Counting Times:346435746
Memory used: 2079572

real    0m57.165s
user    0m53.915s
sys     0m2.391s
```

- 我们发现效果明显好于了手动管理的GC

2.2 Version2

文件：rcop.out

编译指令：

```
g++ rcop.cpp -o rcop.out -std=c++11 -O2
time ./rcop.out datasets/datasets/wiki0.01.txt
```

设计思路：

我发现上面的结果中引用计数带来的时间消耗有点大，于是我实现了一些魔法操作，优化了这种引用计数的消耗。

这里我直接把引用计数的时间放入 `mallocTime` 中一并计算，直接输出总的 `malloc` 和 `delete` 时间。

我将原本 `word` 指针设置为：

```
|*refer|word|
//其中refer为int长度
//word区域就是存储的单词
```

测试结果：

```
(base) jitianyingdeMacBook-Pro:WordCount jitianying$ time ./rcop.out datasets/datasets/wiki0.01.txt
num origin words: 19008043
num filtered words: 16419850
num different words:365396
mallocTime:3.1791
deleteTime:1.74462
Modify Reference Counting Times:346435746
Memory used: 1523488

real    0m44.869s
user    0m42.845s
sys     0m1.656s
```

惊喜的发现，时间优化成了`version1`的一半左右，空间也减少了不少。

GO程序测试

- 测试空间的办法：

```
ru := syscall.Rusage{}
syscall.Getrusage(syscall.RUSAGE_SELF, &ru)
fmt.Println("memory:", ru.Maxrss)
```

- 测试结果汇总

Parameters	Real	User	Sys	Space
GODEBUG=gctrace=1	35.03	55.53	2.54	2691720 KB
GOGC = off	64.08	36.58	15.90	2768632 KB
GOGC = on	33.78	56.11	1.51	2586456 KB
GOMAXPROCS=1 GOGC=off	65.20	35.16	17.15	3370244 KB
GOMAXPROCS=1 GOGC=on	48.80	45.86	2.27	3904516 KB

- 测试结果分析
    - 多线程，开启GC的时候，我们发现时间和空间指标最好，符合认知
    - 在开启单线程的时候，User的时间少于Real，因为多线程的时候`User`的时间是多个线程的总和，所以经常会比`real`大，但是单线程的时候不会出现这种情况

单线程情况下各个GC比较

GC	Real	User	Sys	Space
手动GC	1m12.243s	1m11.077s	0m1.061s	1219028 KB
引用计数Version1	0m57.165s	0m53.915s	0m2.391s	2079572 KB
引用计数Version2	0m44.869s	0m42.845s	0m1.656s	1523488 KB
GO程序	0m48.80s	0m45.86s	0m2.27s	3904516 KB

我发现我实现的引用计数 `Version2` 的性能是优于 `GO` 的