



下载APP



35 | 内存管理第2关：实现垃圾回收

2021-11-03 宫文学

《手把手带你写一门编程语言》

课程介绍 >



讲述：宫文学

时长 14:32 大小 13.32M



你好，我是宫文学。

今天这节课，我们继续上一节课未完成的内容，完成垃圾回收功能。

在上一节课，我们已经实现了一个基于 Arena 做内存分配的模块。并且，我们还在可执行程序里保存了函数、类和闭包相关的元数据信息。

有了上一节课的基础之后，我们这节课就能正式编写垃圾回收的算法了。算法思路是这样的：



首先，我们要有一个种机制来触发垃圾回收，进入垃圾回收的处理程序；

第二，我们要基于元数据信息来遍历栈帧，找到所有的 GC 根；

第三，从每个 GC 根出发，我们需要去标记 GC 根直接和间接引用的内存对象；

最后，我们再基于对象的标记信息，来回收内存垃圾。

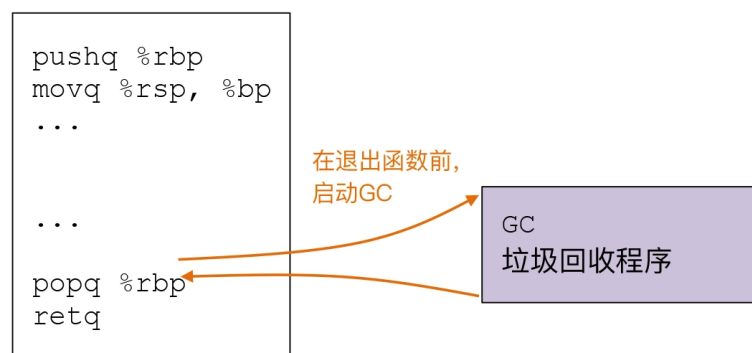
在今天这节课，你不仅仅会掌握标记 - 清除算法，其中涉及的知识点，也会让你能够更容易地实现其他垃圾回收算法，并且让我们的程序能更好地与运行时功能相配合。

那接下来，我们就顺着算法实现思路，看看如何启动垃圾回收机制。


启动垃圾回收机制

在现代的计算机语言中，我们可以有各种策略来启动垃圾回收机制。比如，在申请内存时，如果内存不足，就可以触发垃圾回收。甚至，你也可以每隔一段时间就触发一下垃圾收集。不过不论采取哪种机制，我们首先要有办法从程序的正常执行流程，进入垃圾回收程序才行。

进入垃圾回收程序，其实有一个经常使用的时机，就是在**函数返回**的时候。这个时候，我们可以不像平常那样，使用 `retq` 跳回调用者，而是先去检查是否需要做垃圾回收：如果需要做垃圾回收，那就先回收完垃圾，再返回到原来函数的调用者；如果不需要做垃圾回收，那就直接跳转到函数的调用者。



实现这个功能很简单，只需要在 `return` 语句之前调用 `frame_walker` 这个内置函数，并把当前 `%rbp` 寄存器的值作为参数传进去就好了：

 复制代码

```
1 visitReturnStatement(rtnStmt:ReturnStatement):any{
2     if (rtnStmt.exp!=null){
3         let ret = this.visit(rtnStmt.exp) as Oprand;
4
5         //调用一个内置函数，来做垃圾回收
6         this.callBuiltIns("frame_walker",[Register.rbp]); //把当前%rbp的值传进去
7
8         //把返回值赋给相应的寄存器
9         let dataType = getCpuDataType(rtnStmt.exp.theType as Type);
10        this.movIfNotSame(dataType, ret, Register.returnReg(dataType));
11    ...
12    }
13 }
```

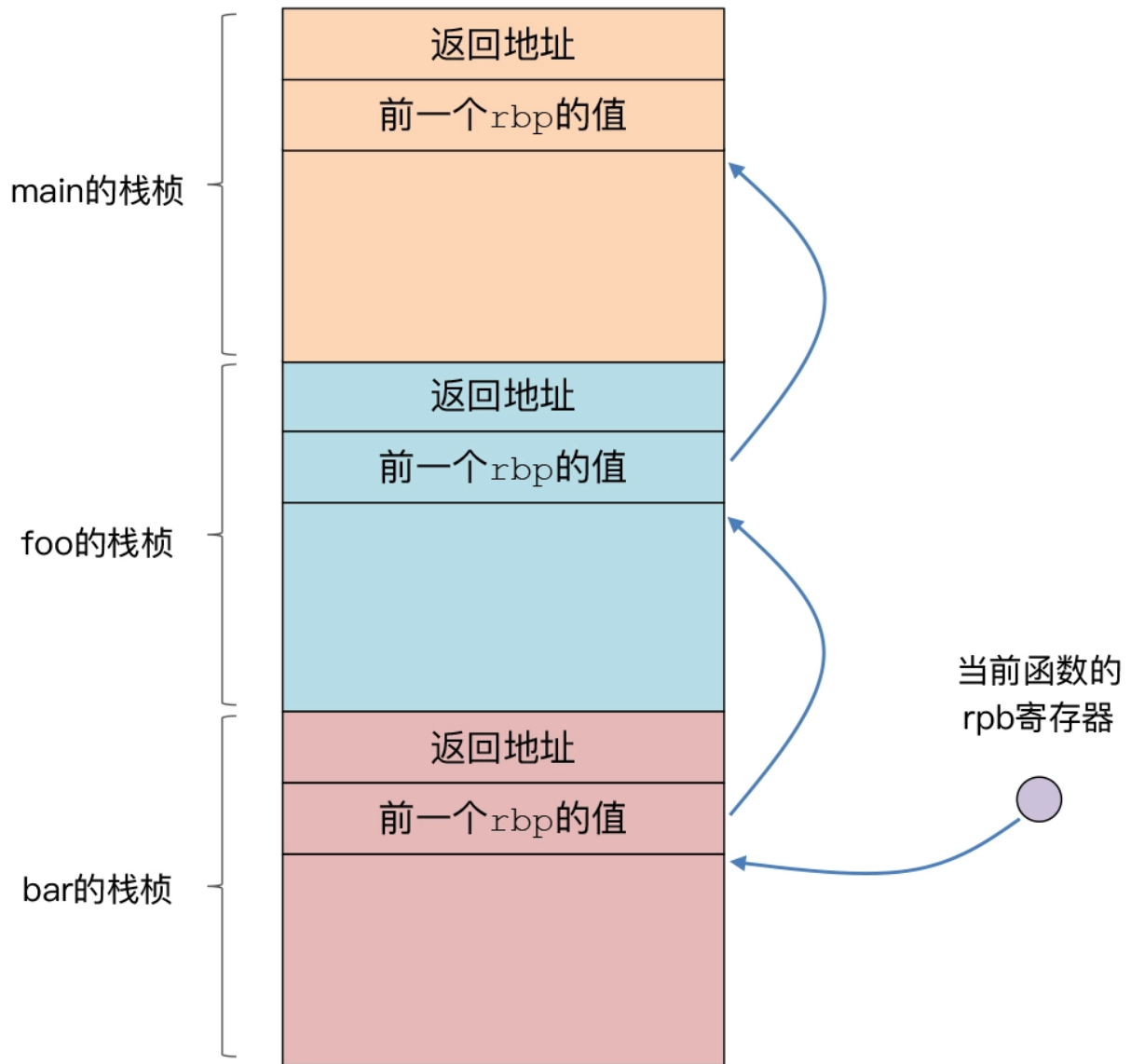
这样，我们就能获得调用 GC 程序的时机。

在这段代码中，frame_walker 内置函数的功能是遍历整个调用栈。这就是我们启动垃圾回收机制后，要进行的下一个任务。接下来我们就来分析一下具体怎么做。

遍历栈帧和对象

遍历栈帧其实很简单，因为我们能够知道每个栈帧的起始地址。从哪里知道呢？就是 rbp 寄存器。rbp 寄存器里保存的是每个栈帧的底部地址。

每次新建一个栈帧的时候，我们总是把前一个栈帧的 rbp 值保护起来，这就是你在每个函数开头看到的第一行指令：pushq %rbp。因此，我们从栈帧里的第一个 8 字节区域，就可以读出前一个栈帧的 %rbp 值，这就意味着我们得到了前一个栈帧的栈底。然后你可以到这个位置，再继续获取更前一个栈帧的地址。具体你可以看下面这张图：



那这个思路真的有用吗？我们直接动手试一下！

首先，我写了一个简单的测试程序。在这个程序里，main 函数调用了 foo，foo 又调用了 bar。这样，在 foo 和 bar 返回前，我们都可以启动垃圾回收，但从 main 函数返回的时候就没有必要启动了，因为这个时候进程结束，进程从操作系统申请的所有内存，都会还给操作系统。

复制代码

```
1 function foo(a:number):string{
2     let s:string = "PlayScript!"
3     let b:number = bar(a+5);
4     return s;
5 }
```

```
6
7 function bar(b:number):number{
8     let a:number[] = [1,2,b];
9     let s:string = "Hello";
10    println(s);
11    println(a[2]);
12    b = b*10;
13    return b;
14 }
15
16 println(foo(2));
17
```

接下来，我又写了一个 `frame_walker.c` 的程序。这个程序很简单，就是依次打印出每个栈帧所保存的 `rbp` 寄存器的地址。在程序里，`rbpValue0` 是调用 `frame_walker.c` 程序的函数的 `rbp` 值，这个值是当前栈帧的底部。在这个地址里，保存的是前一个栈帧原来的 `rbp` 值，也就是前一个栈帧的 `rbp` 的地址，依次类推。

[复制代码](#)

```
1 //遍历栈帧
2 void frame_walker(unsigned long rbpValue0){
3     //当前栈帧的rbp值
4     printf("\nrbpValue0:\t0x%x(%ld)\n", rbpValue0, rbpValue0);
5
6     //往前一个栈帧的rbp值
7     unsigned long rbpValue1 = *(unsigned long*)rbpValue0;
8     printf("rbpValue1:\t0x%x(%ld)\n", rbpValue1, rbpValue1);
9
10    //再往前一个栈帧
11    unsigned long rbpValue2 = *(unsigned long*)rbpValue1;
12    printf("rbpValue2:\t0x%x(%ld)\n", rbpValue2, rbpValue2);
13
14    //再往前一个栈帧
15    unsigned long rbpValue3 = *(unsigned long*)rbpValue2;
16    printf("rbpValue3:\t0x%x(%ld)\n", rbpValue3, rbpValue3);
17
18    //再往前一个栈帧
19    unsigned long rbpValue4 = *(unsigned long*)rbpValue3;
20    printf("rbpValue4:\t0x%x(%ld)\n", rbpValue4, rbpValue4);
21 }
```

最后，你可以用 `make example_gc` 命令，编译成可执行文件。运行这个可执行文件，会打印出下面的输出内容：

```

→ 34 git:(master) x ./example_gc
Hello PlayScript!
7.000000

rbpValue0:      0x7ff7b1d376e0(140701817075424)
rbpValue1:      0x7ff7b1d37700(140701817075456)
rbpValue2:      0x7ff7b1d37720(140701817075488) ← 从bar函数调用
rbpValue3:      0x7ff7b1d37830(140701817075760)   frame_walker
rbpValue4:      0x7ff7b1d37848(140701817075784)

rbpValue0:      0x7ff7b1d37700(140701817075456)
rbpValue1:      0x7ff7b1d37720(140701817075488)
rbpValue2:      0x7ff7b1d37830(140701817075760) ← 从foo函数调用
rbpValue3:      0x7ff7b1d37848(140701817075784)   frame_walker
rbpValue4:      0x0(0)
70.000000

```

你可能注意到了，frame_walker 一共打印了 5 个栈帧的 rbp 值。但是我们的程序最多的时候也只用到了 3 个栈帧呀，那之前更多的那些栈帧是什么呢？这个问题我留给你，看看你能不能用操作系统的知识来回答这个问题。

但目前，我们还只是能找到每个栈帧，接下来我们还要进一步解析栈帧里的内容，从而找到 GC 根，并经由 GC 跟查找到内存对象。**这就需要我们查找函数的元数据了。**

这时候你会发现，为了查找函数的元数据，我们必须知道每个栈帧是由哪个函数生成的。这个问题我们在闭包的那一节课也曾经讨论过，正好在这里实现。

具体实现起来倒也不复杂。我们可以约定，在栈帧的一开头、紧挨着保存 %rbp 寄存器的信息的下面，保存该函数的元数据区的指针。这样，每个函数的汇编代码的序曲部分就会像下面那样。其中，后两行代码就是用来保存 meta 区的地址的：

```

1 pushq    %rbp
2 movq     %rsp, %rbp
3 movq     _foo.meta@GOTPCREL(%rip), %rax
4 movq     %rax, -8(%rbp)

```

复制代码

在这里，“_foo.meta@GOTPCREL(%rip)” 是使用相对于 %rip 寄存器的偏移量，也就是通过下一行代码地址的偏移量来确定 _foo.meta 标签的地址的。

我们曾经也用过相对于 %rip 寄存器寻址的方式，比如获取 double 型字面量。你会发现这里有一点不同，当时我们并没有“@GOTPCREL”这一小段文字。这里会涉及到链接器和代码定位的一些技术细节。在这里，你可以理解为，因为我们把这些元数据声明在数据段，而不是像 double 字面量一样声明在文本段，所以就要用到一个不同的定位方式就行了。

好了，现在我们可以从每个函数的栈帧出发，来找到函数的元数据了。所以，我又写了一个新版本的 frame_walker 函数，来访问函数元数据，打印出每个栈帧里的变量的情况。我们给这个函数传递的参数，是当前栈帧 rbp 的值。

 复制代码

```
1 void frame_walker(unsigned long rbpValue){
2     printf("\nwalking the stack:\n");
3     while(1){
4         //rbp寄存器的值，也就是栈底的地址
5         printf("rbp value:\t\t\t0x%lx(%ld)\n", rbpValue, rbpValue);
6
7         //函数元数据区的地址
8         unsigned long metaAddress = *(unsigned long*)(rbpValue -8);
9         printf("address of function meta:\t0x%lx(%ld)\n", metaAddress, metaAdd
10
11         //函数名称的地址，位于元数据区的第一个位置
12         unsigned long pFunName = *(unsigned long*)metaAddress;
13         printf("address of function name:\t0x%lx(%ld)\n", pFunName,pFunName);
14
15         //到函数名称区，去获取函数名称
16         const char* funName = (const char*)pFunName;
17         printf("function name:\t\t\t%s\n", funName);
18
19         //变量数量，位于元数据区的第二个位置
20         unsigned long numVars = *(unsigned long*)(metaAddress+8);
21         printf("number of vars:\t\t\t%ld\n", numVars);
22
23         //遍历所有的变量
24         for (int i = 0; i< numVars; i++){
25             //获取变量属性的压缩格式，3个属性压缩到了8个字节中
26             unsigned long varAttr = *(unsigned long*)(metaAddress+8*(i+2));
27             printf("var attribute(compact):\t\t0x%lx(%ld)\n", varAttr,varAttr)
28
29             //拆解出变量的属性
30             unsigned long varIndex = varAttr>>32; //变量下标，4个字节
31             unsigned long typeTag = (varAttr>>24) & 0x00000000000000ff; //变量
32             unsigned long offset = varAttr & 0x0000000000ffff; //变量在栈帧
33
34             printf("var attribute(decoded):\t\tvarIndex:%ld, typeTag:%ld, offs
35     }
```



```

36      //去遍历上一个栈帧
37      rbpValue = *(unsigned long*)rbpValue;
38
39      printf("\n");
40
41      //如果遇到main函数，则退出
42      if (strcmp(funName, "main")==0) break;
43  }
44 }
45
46
47
48

```

再次编译并运行，就会打印出每个函数栈帧里的信息。这些信息是在 bar 函数快要返回的时候，遍历各个栈帧打印出来的。从这些信息里，你能够清晰地看到每个函数的栈帧里保存了哪些变量，还有每个变量在栈帧中的位置和每个变量的类型。

```

→ 34 git:(master) ✖ ./example_gc
Hello
7.000000

walking the stack:
rbp value:                0x7ff7ba9fa6d0(140701964674768)
address of function meta: 0x105509030(4384133168)
address of function name: 0x105508c08(4384132104)
function name:            bar
number of vars:           2
var attribute(compact):   0x1000010(16777232)
var attribute(decoded):   varIndex:0, typeTag:1, offset:16 ← 参数b, number型, 比rbp的值小16字节
var attribute(compact):   0x106000018(4395630616)
var attribute(decoded):   varIndex:1, typeTag:6, offset:24 ← 变量a, 数组型, 比rbp的值小24字节

rbp value:                0x7ff7ba9fa700(140701964674816)
address of function meta: 0x105509010(4384133136)
address of function name: 0x105508c00(4384132096)
function name:            foo
number of vars:           2
var attribute(compact):   0x1000010(16777232)
var attribute(decoded):   varIndex:0, typeTag:1, offset:16 ← 参数a, 字符串型, 比rbp的值小16字节
var attribute(compact):   0x105000018(4378853400)
var attribute(decoded):   varIndex:1, typeTag:5, offset:24 ← 变量s, 字符串型, 比rbp的值小24字节

rbp value:                0x7ff7ba9fa720(140701964674848)
address of function meta: 0x105509050(4384133200)
address of function name: 0x105508c10(4384132112)
function name:            main
number of vars:           0

```

接着我们来找 GC 根和它直接引用的对象。你可以看到，bar 函数的栈帧里一共有 2 个变量，分别是参数 b 和变量 a。其中参数 b 是整型的，所以不是 GC 根。而变量 a 是一个数

组，它是 GC 根，所以我们要标记该数组对象为活跃的。而 foo 函数的栈帧里也有两个变量，分别是参数 a 和变量 s。

不过，看到这个结果，你肯定会产生一些疑问。首先，为什么 bar 函数里字符串变量 s 没有出现在栈帧里，而 foo 函数里同样的字符串变量 s 却出现在了栈帧里呢？

这是因为，在 bar 函数要返回的时刻，字符串 s 已经被 bar 函数使用完毕了，变成内存垃圾了。而 foo 函数现在是执行到第二个语句，并且调用了 bar 函数。在从 bar 函数返回以后，它还要在第三条语句中用到字符串变量 s，所以 s 仍然是 alive 的，需要被保存下来。

bar 函数里的参数 b 也是一样的，因为它是被 return 语句引用的，所以在返回之前，仍然是 alive。

那可能你又要问了，不对呀，按照这么说，bar 函数中的数组变量 a 和 foo 函数中的参数 a，应该都没有用了呀，不用去遍历了呀。特别是 bar 函数中的数组，应该作为内存垃圾回收了才对。

没错，这是我们当前算法的缺陷。我们没有去保存变量活跃性分析的结果，也就是说在每一行代码处，到底哪些变量是活跃的，我们是不知道的。这样，我们也就没有办法精确地判断哪些变量其实已经失效了。并且，其实我们也没有办法精确地判断，每个函数现在执行到了哪一行代码，哪些代码是活跃的。

所以，我们这里用了一个近似的算法：只要是曾经被 spill 到栈帧里的变量，我们都认为它们是值得被保护的，还是活跃的。这个算法其实这在实际的应用中已经足够了。

那既然找到了对象引用，接下来我们就可以标记这个对象了。在每个对象的头部，我们都预留了 8 个字节的空间，用来设置各种标志位。这 8 个字节，在一开始申请内存的时候，都被设置成了 0。我们现在要做的也比较简单，只需要把第一位作为“是否是内存垃圾”的标志位就好了。如果该标志位为 1，那就说明这对象是有用的，不能回收。如果是 0，那么就说明它是内存垃圾。

好了，现在我们已经完成了最复杂的任务，也就是遍历栈帧，找到并标记对象引用。**接下来，我们还要基于这些对象，进一步找出它们所引用的其他内存对象。**

在这一部分，针对自定义对象、数组和闭包，我们都需要编写相对应的遍历方法，但原理跟前面我们遍历函数寻找 GC 根是一致的，都是去查找元数据，这里我就不一一展开了，你可以去看 mem.c 中的源代码。

现在标记完所有的对象以后，接下来的事情就简单了。接下来，我们可以遍历 Arena 中的每个内存块，然后再在内存块里遍历每一个对象，把没有标记的对象删掉。对于标记过的对象，我们就要把它的标记去掉，为下一次垃圾收集做好准备。

好了到这里，我们相关算法的实现思路已经梳理清楚了。现在是时候验证我们的成果了，我们内存管理的相关算法到底是否能成功呢？

测试垃圾回收算法

我们直接运行程序，来检验一下。你可以运行命令 `make example_gc`，生成可执行程序，并运行它，会得到如下输出结果：

```

→ 34-35 git:(master) x ./example_gc
obj allocated: 140514728869904
obj allocated: 140514728869948 } ← 用PlayMalloc()申请的
obj allocated: 140514728870004 } ← 三个对象的地址
Hello
7.000000

walking the stack: ← 在退出bar函数前启动的垃圾回收

before gc, dump the arena:
Arena: 1 block(s), blocksize=40960, total memory occupied: 40968
  Block 0: maxFreeSpace=40822
    Node: offset=0, size=44, nextOffset=44
    Node: offset=44, size=56, nextOffset=100
    Node: offset=100, size=38, nextOffset=138
address of obj to set flag: 140514728869948 ← 标记出了数组变量
Flag set for function: bar, varIndex:1, typeTag:6, offset:24
address of obj to set flag: 140514728869904 ← 标记出了字符串变量
Flag set for function: foo, varIndex:1, typeTag:5, offset:24

collecting object: 140514728870004 ← 回收了bar中的字符串对象s

after gc, dump the arena:
Arena: 1 block(s), blocksize=40960, total memory occupied: 40968
  Block 0: maxFreeSpace=40822
    Node: offset=0, size=44, nextOffset=44
    Node: offset=44, size=56, nextOffset=100
walking the stack: ← 在退出foo函数前又一次启动垃圾回收

before gc, dump the arena:
Arena: 1 block(s), blocksize=40960, total memory occupied: 40968
  Block 0: maxFreeSpace=40822
    Node: offset=0, size=44, nextOffset=44
    Node: offset=44, size=56, nextOffset=100
address of obj to set flag: 140514728869904 ← foo中的字符串对象s
Flag set for function: foo, varIndex:1, typeTag:5, offset:24 仍然活跃

collecting object: 140514728869948 ← 把bar函数中的数组对象回收了

after gc, dump the arena:
Arena: 1 block(s), blocksize=40960, total memory occupied: 40968
  Block 0: maxFreeSpace=40822
    Node: offset=0, size=44, nextOffset=44
    Node: offset=44, size=56, nextOffset=100
PlayScript! ← 最后只剩下foo中的字符串对象s

```

你可以看到，程序 bar 函数和 foo 函数结束的时候，分别释放了一些内存垃圾。并且，每次申请内存和垃圾回收，GC 都打印出了当前 Arena 中的信息，你会看到 Arena 中内存分配的详细信息。在做完垃圾回收以后，相关内存空间确实被释放掉了。

也就是说，我们的垃圾回收算法算是圆满成功了！

课程小结

今天这一节课，我们接着上一节完成了垃圾回收算法。我希望你能记住几个要点：

第一，我们有很多种方式可以启动垃圾回收机制。在这里，我们用了最简单的一种方式，也就是在退出函数前触发垃圾回收机制。类似的，你还可以在循环语句的地方检查是否需要做垃圾回收，因为程序中的循环可能会耗费很长的时间，积累很多内存垃圾。这里，我们的程序和运行时就形成了一种协作机制。

其实这种协作机制，也是实现另一个重要的运行时功能，并发管理的基础，特别是协程机制，需要当前程序与协程的调度器主动配合，才能完成细颗粒度的并发调度。你从这节课中，应该会直观地感受到这些运行时的调度机制是怎么运作的。

第二，我们是通过查询栈帧的布局信息去寻找 GC 根的。我们可以认为，没有出现在栈帧布局中的变量，肯定是已经失效了的。而出现在栈帧布局中的变量，有些其实也已经失效了，但我们当前没有必要去做更加精细化地管控。

第三，这节课示范了如何具体访问各种元数据信息。这些技能很重要，能让你可以在可执行文件里自由地存放各种自己需要的静态数据，让程序的运行机制拥有更多的可能性。

最后，我们通过两节课实现了内存分配和垃圾回收的工作。虽然实现得比较简单，但却是一个良好的开始。你在此基础上，可以更快地实现其他的算法。比如，你可以把内存整理一下，减少内存碎片，这样就是实现了标记 - 整理算法。你还可以把所有分配了的内存对象拷贝到一个新的区域，这就实现了停止 - 拷贝算法，也就是 Java 等语言普遍采用的算法。

不过，当你在内存中移动对象的时候，又会遇到新的技术挑战。比如，你需要知道栈帧中的哪些变量是引用了这个对象，再来更新这些值。再比如，当你的程序正在读写一个数组的时候，整个数组却被 GC 挪到了另一个位置，那与数组处理有关的功能就可能会出错。当然，这些技术点也都有解决办法，相关的书籍和文章已经有很多。但有了我们这两节的基础，你可以自己动手来验证这些知识点了。

在作为开源项目的 PlayScript 代码库中，我会继续迭代、修改与垃圾收集有关的算法，如果你有兴趣可以继续一起研究。

思考题

在这节课的示例程序中，为了启动垃圾回收机制，我采用了一个普通的函数调用的方式。但是，有没有办法对这个方式进行优化？比如，因为当前函数要退出了，除了返回值，其

他变量都不用检查了。再比如，这个函数的栈帧也可以被垃圾回收函数复用，就像尾调用优化那样。我想问一下，你要如何修改现有的调用垃圾回收函数的机制，才能实现上面的优化呢？

欢迎把这节课分享给更多对内存管理感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

🔗 这节课的示例代码目录在这里！

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

📄 生成海报并分享

👍 赞 0

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 34 | 内存管理第1关：Arena技术和元数据

下一篇 36 | 节点之海：怎么生成基于图的IR？

11.11 全年底价

VIP 年卡限定 3 折

畅学 200 门课程 & 新课上线即解锁

超值拿下 ¥999



精选留言 (2)

写留言



奋斗的蜗牛

2021-11-04

太强了，轻轻松松就把这么复杂的知识讲明白

展开



1



写点啥呢

2021-11-03

请问宫老师，为何GC操作的实现frame_walker只检查栈中的对象root，存在于寄存器中引用的对象我理解也应该是有效live的。请老师指点，谢谢

展开



1