



下载APP



34 | 内存管理第1关：Arena技术和元数据

2021-11-01 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 15:07 大小 13.85M



你好，我是宫文学。

通过前面 8 节课的学习，我们实现了对浮点数、字符串、数组、自定义对象类型和函数类型的支持，涵盖了 TypeScript 的一些关键数据类型，也了解了实现这些语言特性所需要的一些关键技术。

在这些数据类型中，字符串、数组、class 实例，还有闭包，都需要从堆中申请内存，但我们目前还没有实现内存回收机制。所以，如果用我们现在的版本，长时间运行某些需要在堆中申请内存的程序，可能很快会就把内存耗光。



所以，接下来的两节课，我们就来补上这个缺陷，实现一个简单的内存管理模块，支持内存的申请、内存垃圾的识别和回收功能。在这个过程中，你会对内存管理的原理产生更加

清晰的认识，并且能够自己动手实现基本的内存管理功能。

那么，首先我们要分析一下内存管理涉及的技术点，以此来确定我们自己的技术方案。

内存管理中的技术点

计算机语言中的内存管理模块，能够对内存从申请到回收进行全生命周期的管理。

内存的申请方面，一般不会为每个对象从操作系统申请内存资源，而是要提供自己的内存分配机制。

而垃圾回收技术则是内存管理中的难点。垃圾回收有很多个技术方案，包括标记 - 清除、标记 - 整理、停止 - 拷贝和自动引用计数这些基础的算法。在产品级的实现里，这些算法又被进一步复杂化。比如，你可以针对老的内存对象和新内存对象，使用不同的回收算法，从而形成分代管理的方案。又比如，为了充分减少由于垃圾收集所导致的程序停顿，发展出来了增量式回收和并行回收的技术。

关于这些算法的介绍，你可以参考《编译原理之美》的 33 节，里面介绍了各种垃圾收集算法。还有《编译原理实战课》的第 32 节，里面分析了 Python、Java、JavaScript、Julia、Go、Swift、Objective-C 等各种语言采用的内存管理技术的特点，也讨论了这些技术与语言特性的关系。在这节课里，我就不重复介绍这些内容了。

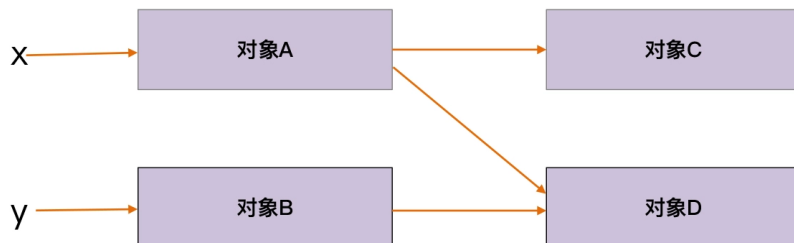
垃圾收集对语言运行的影响是很大的，因此我们希望垃圾回收导致的程序停顿越短越好，消耗的系统资源越少越好。这些苛刻的要求，导致在很多现代语言中，垃圾回收器（GC）成了运行时中技术挑战很高的一个模块。不过，再难的技术都是一口口吃下的。在这节课里，我们先不去挑战那些特别复杂的算法，而是选择一个最容易上手的、入门级的算法，**标记 - 清除算法**来做示范。

标记 - 清除算法的思路比较简单，只需要简单两步：

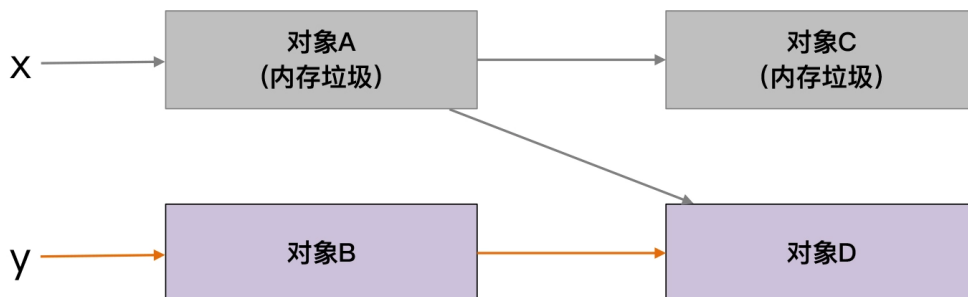
首先，我们要找出哪些内存对象不是垃圾，并进行标记；

第二，回收掉所有没做标记的对象，也就是垃圾对象。

我们通过一个例子来看一下。在下图中，x 和 y 变量分别指向了两个内存对象，这两个内存对象可能是自定义类的实例，也有可能是闭包、字符串或数组。这些对象中的字段，又可能会引用另外的对象。



在图中，当变量 x 失效以后，它直接引用和间接引用的对象就会成为内存垃圾，你就可以回收掉它了。这就是标记 - 清除算法的原理，非常简单。



在这个图里，变量 x 和 y 叫做 GC 的根（GC root）。算法需要从这些根节点出发，去遍历它直接或间接引用的对象。这个过程，实际上就是图的遍历算法。

好了，算法上大的原理我们就搞清楚了。那接下来，我们需要讨论一些实现上的技术点，包括如何管理内存的申请和释放、如何遍历所有的栈帧和内存对象，等等。

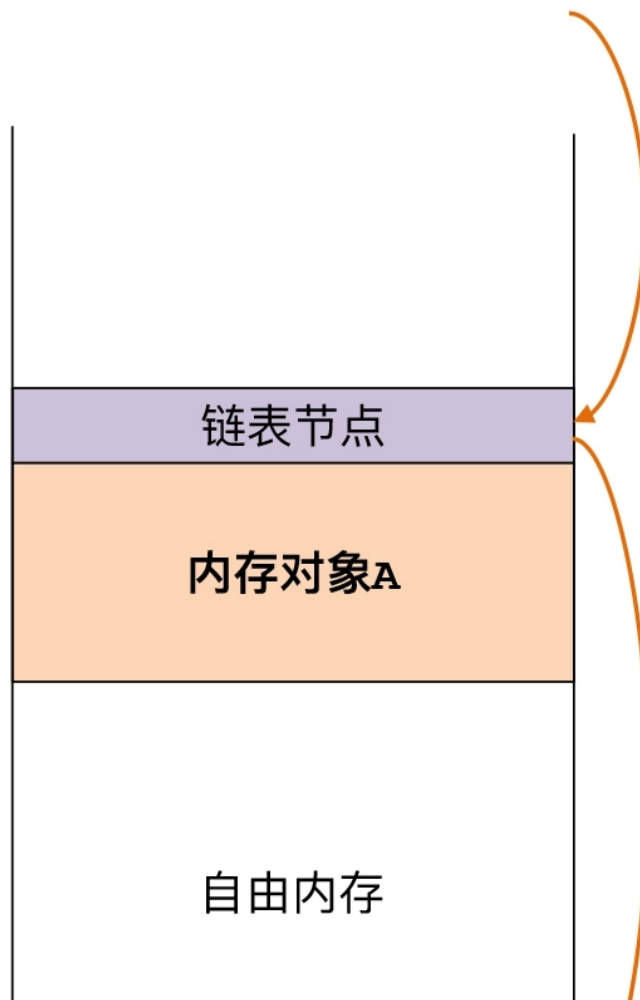
首先说一下如何管理内存的申请和释放。

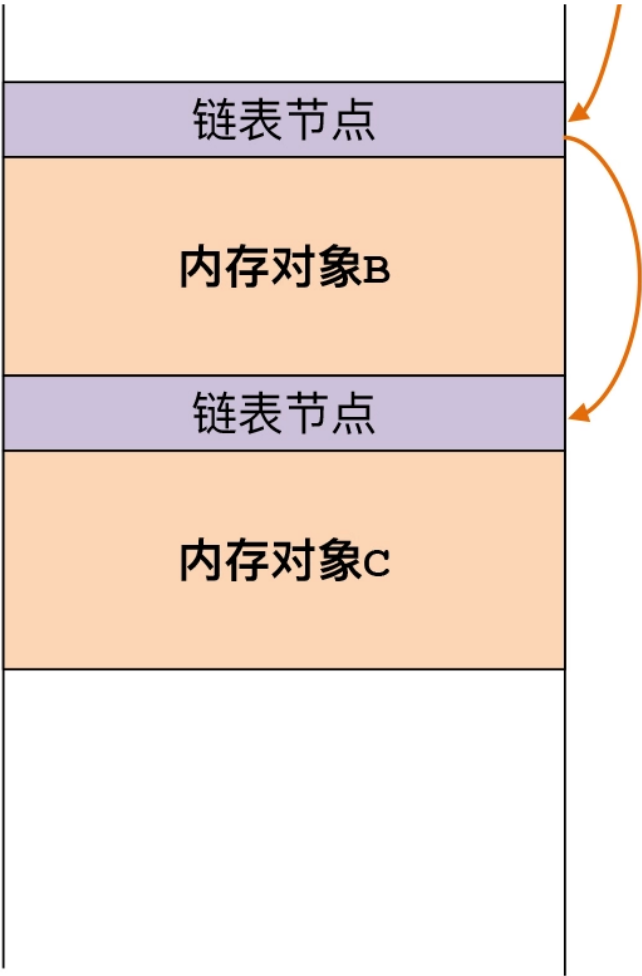
内存的申请和释放

在我们前面实现的、C 语言版本的字节码虚拟机中，我们就曾经讨论过如何高效申请内存的问题。我们发现，如果调用操作系统的接口频繁地申请和释放小的内存块，会大大降低系统的整体性能。所以，我们采用了 Arena 技术，也就是一次性地从操作系统中申请比较大块的内存，然后再自行把这块大内存划分成小块的内存，给自己的语言使用。

在今天这节课，我们仍然使用 Arena 技术来管理内存：**当我们创建新的内存对象的时候，就从 Arena 中找一块未被占用的内容空间；而在回收内存对象的时候，就把内存对象占的内存区域标记成自由空间。**

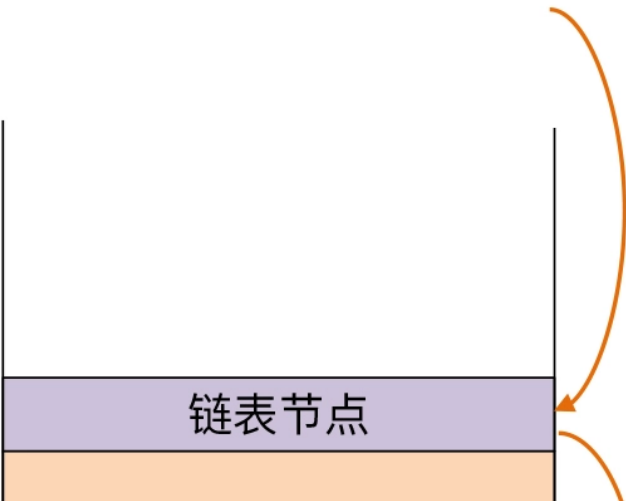
在这里你会发现，为了记住哪些内存是被分配出去的，那些内存是可用的，我们需要一个数据结构来保存这些信息。在我的参考实现里，我用了一个简单的链表来保存这些信息。每块被分配出去的内存，都是链表的一个节点。节点里保存了当前内存对象的大小，以及下一个节点的地址。

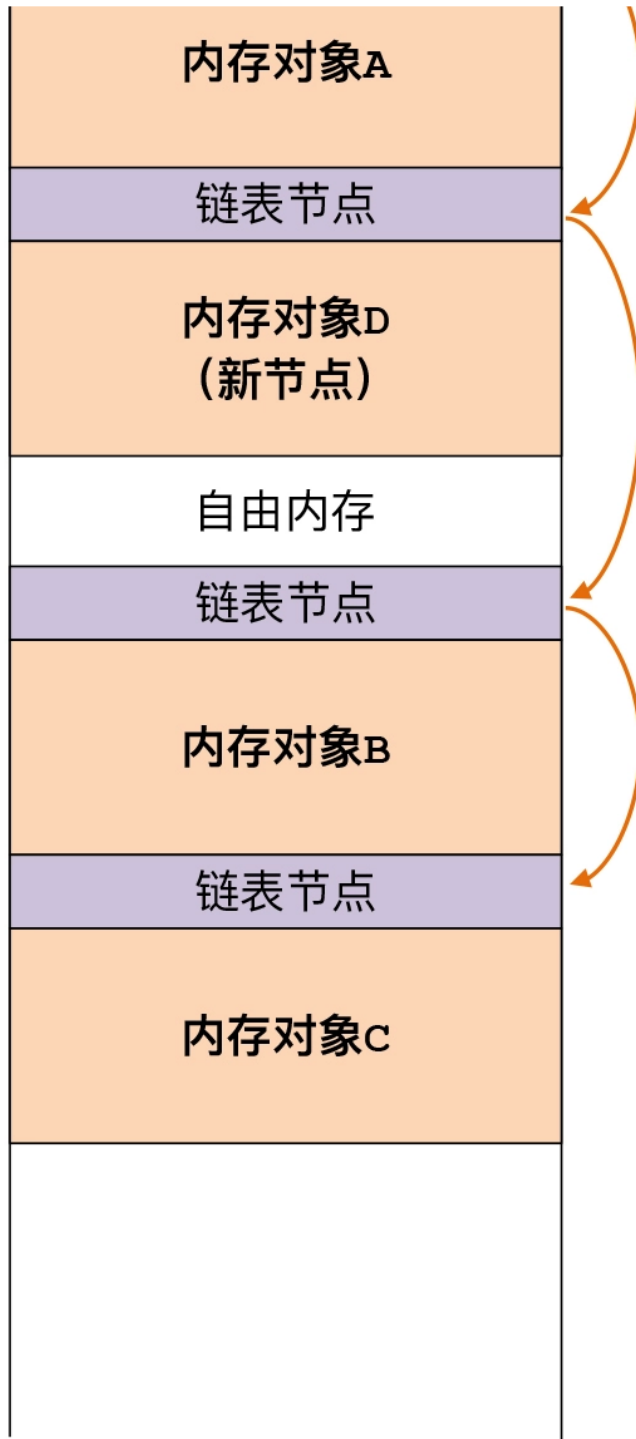




顺着这个链表，你可以查找出自由的内存。假设节点 1 的地址是 80，对象大小是 48 字节，节点 2 地址是 180，那么节点 1 和 2 之间就有 52 个字节的自由空间。

当我们要申请内存的时候，如果我们要申请的对象大小低于 52 个字节，那就可以把这块空间分配给它。这个时候，我们就要修改链表的指针，把新的节点插入到节点 1 和节点 2 之间。





如果要回收内存呢？也比较简单，我们就从链表中去掉这个节点就好了。

了解了内存申请和释放的内容后，接下来，我们就需要查找并标记哪些内存是仍然被使用的，从而识别出内存垃圾。这就需要程序遍历所有栈帧中的 GC 根引用的对象，以及这些对象引用的其他对象。而要完成这样的遍历，我们需要知道函数、类和闭包等的元数据信息才可以。


管理元数据

我们前面说过，**GC 根就是那些引用了内存对象的变量**。而我们知道，我们的程序中用到的变量，有可能是在栈中的，也有可能是在寄存器里的。那到底栈里的哪个位置是变量，哪个寄存器是变量呢？另外，如何遍历所有的栈帧呢？如何知道每个栈帧的开头和结尾位置？又如何知道哪个栈帧是第一个栈帧，从而结束遍历呢？这些都是需要解决的技术问题，我们一个一个来看。

首先，我们要确定栈帧和寄存器里，哪些是变量，也就是 GC 根。

这就需要我们保存变量在栈帧中的布局信息。对于每个函数来说，这些布局信息都是唯一的。这些信息可以看做是函数的元数据的一部分。其他元数据信息包括函数的名称，等等。

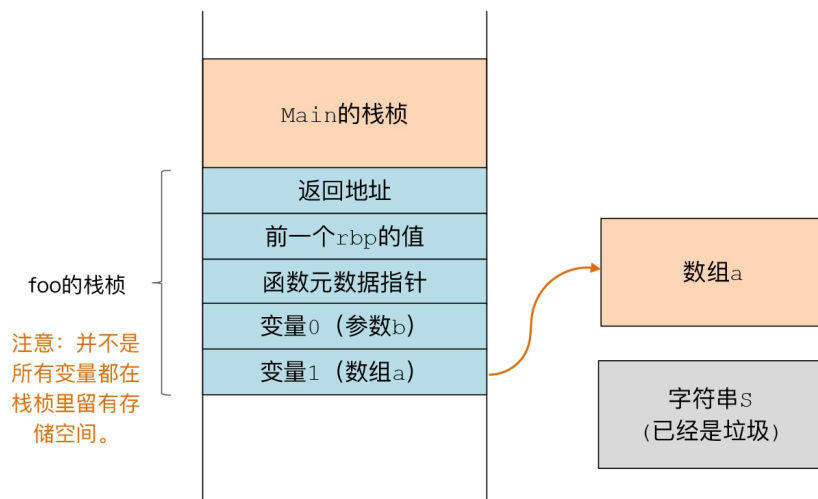
我们用一个例子来分析一下变量布局情况。下面的 foo 函数的栈帧里，包括几个本地变量和几个临时变量。基于我们的寄存器分配算法，这些变量有些会被 Spill 到栈帧中。比如，如果某个变量使用的寄存器是需要 Caller 保护的，那么在调用另一个函数的时候，这些变量就会被 Spill 到内存中。

 复制代码

```
1 function foo(b:number):number{
2     let a:number[] = [1,2,b];
3     let s:string = "Hello PlayScript!";
4     println(s);
5     println(a[2]);
6     return b*10;
7 }
8
9 println(foo(2));
```

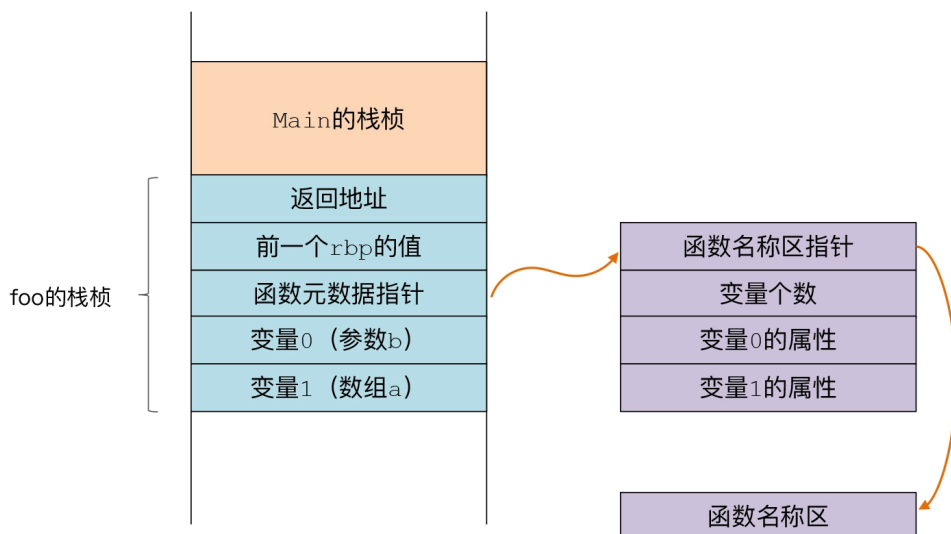
另外，如果一个函数用到了需要 Callee 保护的寄存器，那么这些寄存器的信息也会被写入到栈帧，这些寄存器的值也可能是调用者的某个变量。算法可以查询调用者的变量布局信息来确认这一点。

最终，对于 foo 函数来说，这些变量在栈帧中的布局如下：




那包含了变量布局的元数据信息，应该保存到哪里呢？你可能已经想到了，它们可以被保存在可执行文件的数据区呀，就像之前我们保存 vtable 那样。

在具体实现的时候，这个数据区可以分成多个组成部分。像 vtable 这样的数据，出于性能上的要求，我们最好能够比较快捷地访问，所以我们让程序通过“1 跳”，也就是只做一次获取地址的操作，就能查到方法的入口地址。而对于其他元数据信息，由于数据类型跟 vtable 的不一样，可以安排到另一个数据区中，并从第一个数据区链接过去。元数据在静态数据区的布局如下图所示：



它们在汇编代码中可以写成下面的样子：

 复制代码

```
1  .section    __DATA,__const
2      .globl  _foo.meta                ## can be accessed globally
3      .p2align    3                    ## 8 byte alignment
4  _foo.meta:
5      .quad    _foo.name                ## link to function name section
6      .quad    2                        ## var count
7      .quad    0x0000000001000010      ## var0, type: 1, address offset: 16
8      .quad    0x0000000106000018      ## var1, type: 6, address offset: 24
9
10     .section    __TEXT,__const
11     .globl  _foo.name                ## can be accessed globally
12     .p2align    3                    ## 8 byte alignment
13 _foo.name:
14     .asciz    "foo"
```

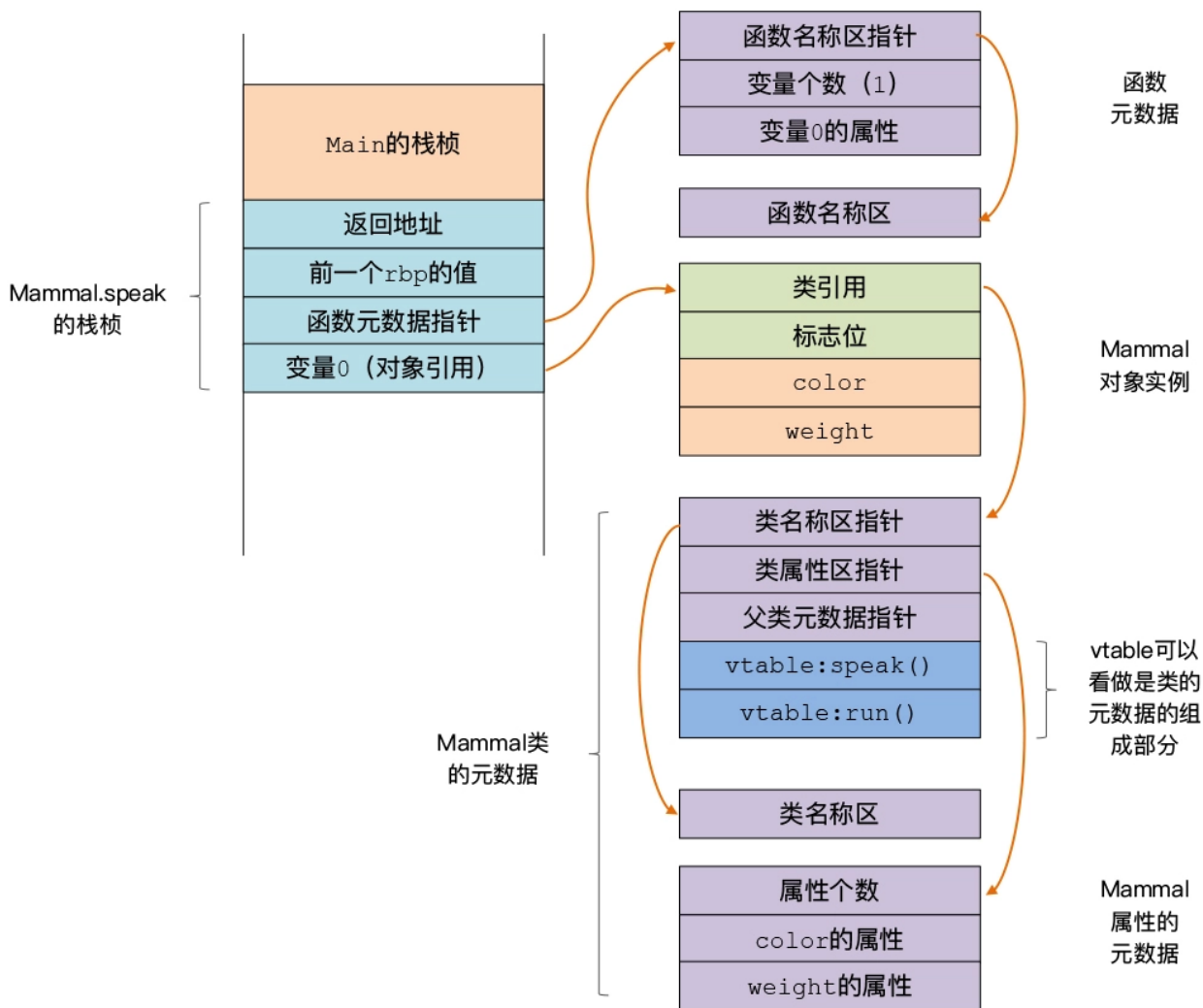
好了，通过函数的元数据，我们已经可以知道栈帧内每个变量的地址是什么，以及哪些变量才是对象引用。

可是，如果内存对象是一个 class 实例，或者是一个数组、一个闭包，那它们还可以引用其他的内存对象。所以我们还要继续往下查找并做标记。

对于对象实例来说，我们需要知道对象的属性都是一些什么类型，是否是对象引用，它们的地址又是什么。所以，我们需要扩展刚才的元数据区，保存 class 的元数据，包括这个 class 有哪些属性、每个属性的类型，以及该属性在对象数据中的位置。另外，我们还需要记录该 class 的父类，用一个指针指向父类的元数据，从而能够访问从父类继承下来的属性的信息。

那如何基于对象实例访问到这些元数据信息呢？

这个简单，因为我们已经在每个对象的对象头都预留了 8 字节的一个位置，用来保存类引用。那么，在每次创建对象的时候，我们就在这个位置存上类的元数据信息的起始地址就好了。

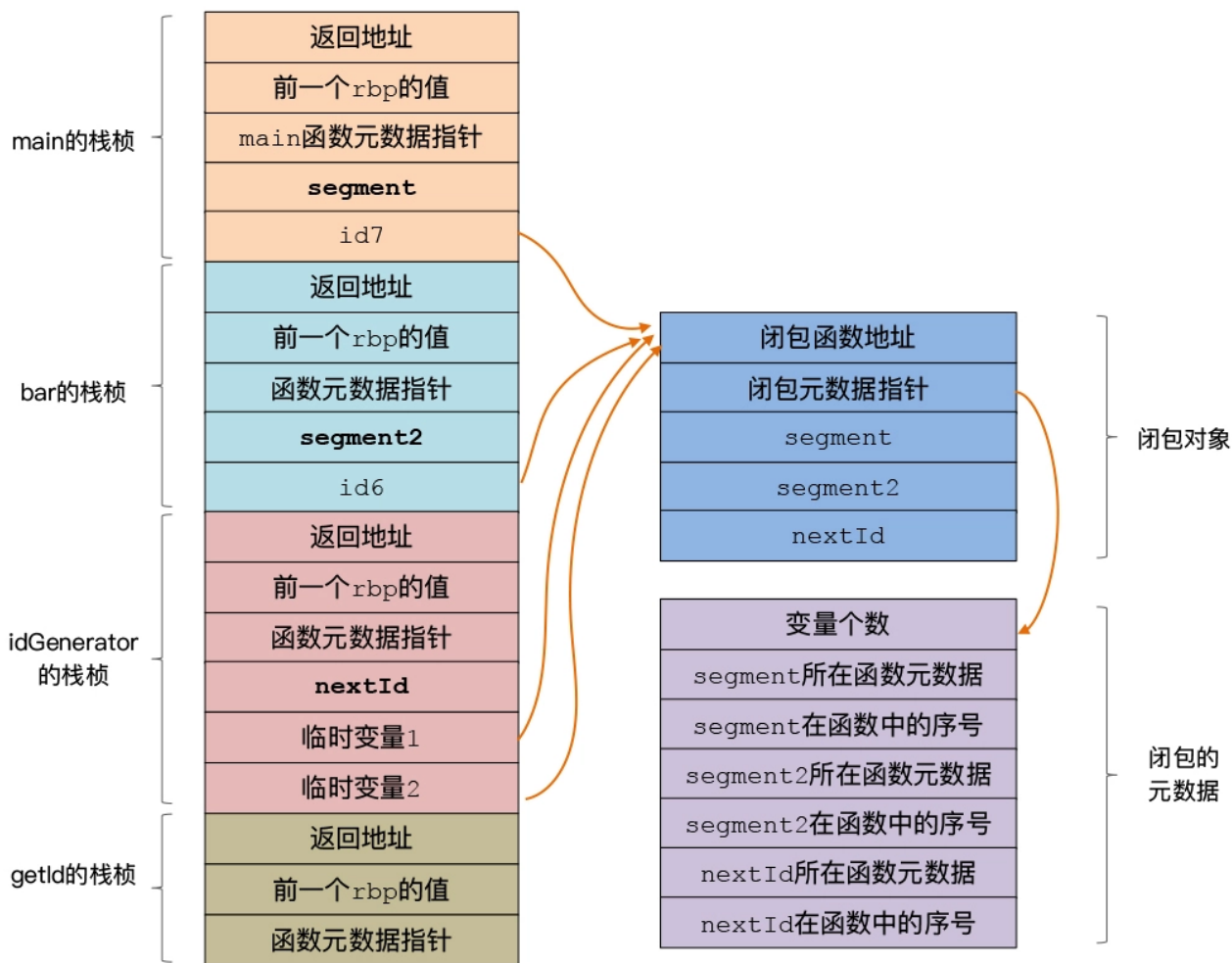


我们再来看看如何处理数组对象。如果数组中存放的数据是对象引用，那我们需要遍历数组的元素。不过因为一个数组里每个元素的类型是一样的（目前为了简单，暂且不支持类型为 `any` 的数组），所以我们可以偷懒，让数组对象头里的类引用也指向元素类型所对应的元数据就好了。对于系统内置的类型，比如 `number`、`string` 等，我们可以特殊处理，建立特殊的元数据信息。

最后，我们再看看闭包。我们知道，闭包也会在堆里形成对象，所以我们也需要知道闭包对象里的数据是否引用了其他对象。不过，处理闭包的复杂之处在于，对于闭包所引用的变量，如果它所在的函数的生存期还没有结束，那么要从该函数的栈帧里访问变量数据；而如果它所在的函数的生存期已经结束，那么这个变量的数据是保存在闭包对象中的。所


以，在运行期，当我们要访问一个闭包数据的时候，总是要先从现有栈帧中去查找，之后才在闭包对象中查找。

为了支持上面这些的数据查找过程，我们需要设计与闭包有关的元数据信息。这个倒也简单，就是把函数引用的所有外部作用域中的变量，以及该变量所在的函数信息保存起来就行。你可以看一下我画的图示。

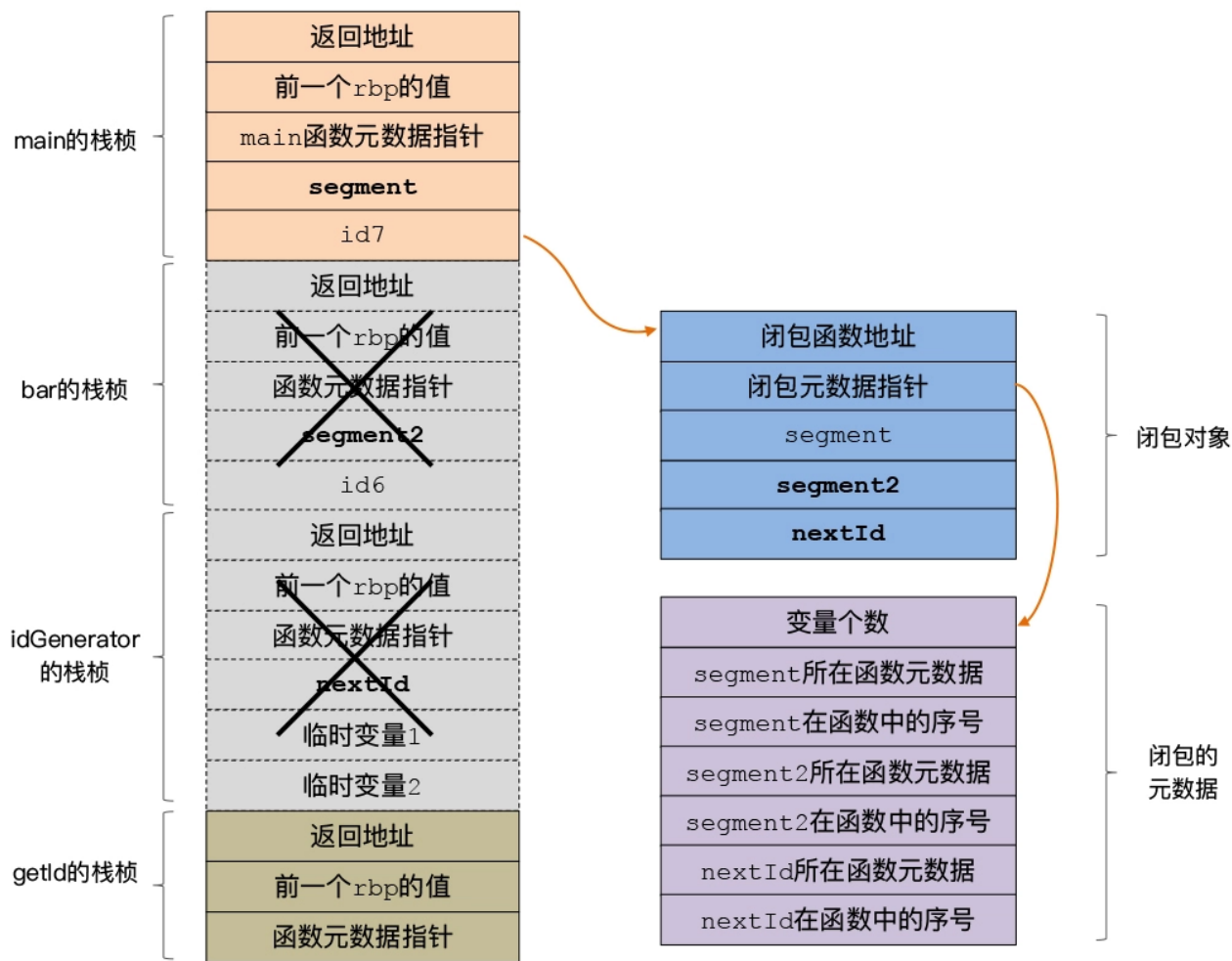


这样的话，你通过闭包的元数据，就能找到闭包中的每个变量所在的函数，从而确定它的类型信息了。你还可以遍历栈帧，来找到该变量具体的值。如果我们在栈帧里找不到这个变量，那么到闭包对象中去找就好了。

不过这里要注意一下，在闭包对象中，我们为每个闭包变量都预留了一个位置，即使这个位置有可能用不上。比如，在下面闭包的例子中，当闭包位于 `bar()` 函数中的时候，它是可以访问 `segment2` 变量的。但如果它到了 `main` 函数中，就不能在栈帧中访问 `segment2` 了，而是要从闭包对象中访问。所以，我们要提前在闭包对象中预留这个内存空间才可以。

 复制代码

```
1 //编号的组成部分
2 let segment:number = 1000;
3
4 function bar():()=>number{
5     //编号的另一个组成部分
6     let segment2:number = 100;
7     function idGenerator():()=>number{
8         let nextId = 0;
9
10        function getId(){
11            return segment + segment2 + nextId++; //访问了3个外部变量
12        }
13
14        //在与getId相同的作用域中调用它
15        println("getId in bar:" + getId());
16        segment2 += 100;
17        println("getId in bar:" + getId());
18
19        //恢复nextId的值
20        nextId = 0;
21
22        return getId;
23    }
24
25    //在bar函数中调用，这时候可以看到segment2变量
26    println("\nid6:");
27    let id6 = idGenerator();
28    println("\nid6:");
29    println(id6());
30    println(id6());
31
32    return id6;
33 }
34
35 //在main函数中调用，这时候可以看到segment变量
36 //而segment2和nextId都保存在闭包对象里了。
37 println("\nid7:");
38 let id7 = bar();
39 println(id7());
40 println(id7());
```



在main函数中执行id7()时，getId函数可以找到segment变量，但找不到segment2和nextId，需要到闭包对象中去寻找。



好了，关于元数据的讨论就是这些。你可以运行"node play example_metadata.ts --dumpAsm"命令，生成带有元数据信息的汇编文件，看看各类元数据信息都是如何保存的。

课程小结

这节课的内容就是这些。关于内存管理和垃圾回收技术，我希望你能对下面这些知识点留下深刻的印象。

首先，垃圾收集的算法是很多的，GC 也是现代语言运行时中的一个重要组成部分。GC 中可能会综合采用多种算法。但只有先掌握像标记 - 清除、停止 - 拷贝这样的基础算法，才

能进一步去掌握更复杂的算法。

第二，内存管理技术不仅包括垃圾收集功能，还包括语言自己的内存分配功能。内存分配模块要能记住哪些内存是被分配出去的，还要能够找到合适大小的可用内存给新对象，也要能够把回收后的内存释放出来。

第三，为了遍历所有的栈帧中的 GC 根，以及内存对象所引用的其他对象，我们必须保存函数、类和闭包的元数据信息，元数据信息保存在可执行文件的数据区中。在函数的栈帧和内存对象中，我们都要保存指向这些元数据的引用。

你应该也注意到了，我们花了大量的篇幅讨论元数据，也讨论了如何把它们编译到可执行文件中去。这项技术很重要，如果我们要 debug 程序，就非常依赖这些信息。而且，如果我们未来还要支持一些元编程功能，比如像 Java 的 Reflection 机制那样，去在运行时动态调用类的方法，那也需要依靠这些元数据信息。

最后，我再补充一点。在 C、C++ 这些语言的工具链中，我们今天提到的这些元数据信息，跟符号信息其实是差不多的意思。而 Java 等语言，似乎更愿意用"元数据"这样的术语。你只要理解它们是差不多的就行了。

在下一节课里，我们将利用现在保存好的元数据信息，去遍历所有的栈帧和对象，识别内存垃圾，并进行回收。

思考题

为了在汇编代码中保存一些静态数据，我们用到了越来越多的伪指令。所以，我又要建议你多熟悉手册。这次，我希望你把 [GNU 汇编器的手册](#) 熟悉起来。那么今天的思考题呢，就要让你查查手册，看看我们这节课生成的汇编代码中，.asciz、.p2align 和 .quad 都是什么意思。另外，如果我要向数据区写 4 字节的整型数据，应该用什么伪指令呢？

欢迎把这节课分享给更多感兴趣的朋友。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 33 | 函数式编程第2关：实现闭包特性

下一篇 35 | 内存管理第2关：实现垃圾回收

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 (4)

写留言



奋斗的蜗牛

2021-11-02

.asciz用来声明字符串，.p2align用来声明下一条指令所在位置的对齐要求，.quad声明8字节的数



1



...

2021-11-02

是否选择使用C语言编写才需要考虑内存管理 使用JS编译的话内存管理会由JS完成



奋斗的蜗牛

2021-11-01

这节课知识量很大，看来汇编语言还是很重要

展开 ∨



有学识的兔子

2021-11-07

.asciz 代表以"\0"作为结束符的字符串； .p2align代表数据边界需要满足对齐的字节倍数； .

quad:代表大数，8字节的整数；

要数据区写4个字节整数，是不是类似如下，声明数据段然后填入数据？

```
.section __DATA,__const
```

```
.globl _foo.meta ## can be accessed globally...
```

展开 ∨

