21 | 运行时机制：突破现象看本质，透过语法看运行时

2019-10-09 宫文学 来自北京

《编译原理之美》

编译器的任务，是要生成能够在计算机上运行的代码，但要生成代码，我们必须对程序的运行环境和运行机制有比较透彻的了解。

你要知道，大型的、复杂一点儿的系统，比如像淘宝一样的电商系统、搜索引擎系统等等，都存在一些技术任务，是需要你深入了解底层机制才能解决的。比如淘宝的基础技术团队就曾经贡献过，Java 虚拟机即时编译功能中的一个补丁。

这反映出掌握底层技术能力的重要性，所以，如果你想进阶成为这个层次的工程师，不能只学学上层的语法，而是要把计算机语言从上层的语法到底层的运行机制都了解透彻。

本节课，我会对计算机程序如何运行，做一个解密，话题分成两个部分：

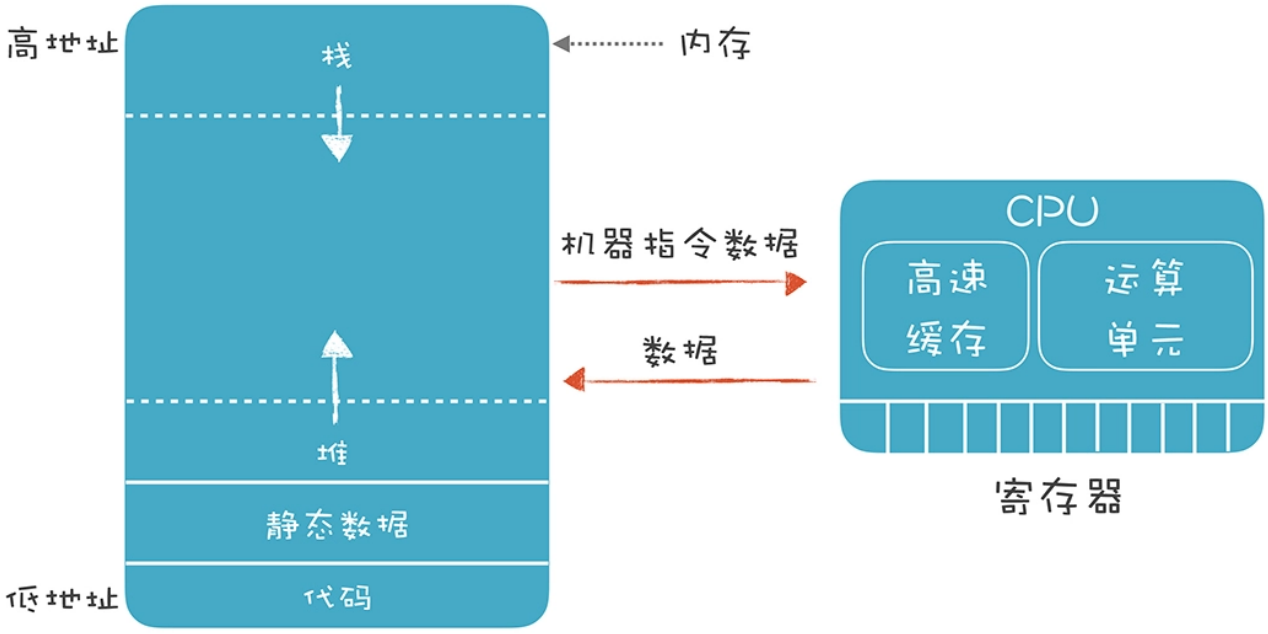
1. 了解程序运行的环境，包括 CPU、内存和操作系统，探知它们跟程序到底有什么关系。

2. 了解程序运行的过程。比如，一个程序是怎么跑起来的，代码是怎样执行和跳转的，又是如何管理内存的。

首先，我们先来了解一下程序运行的环境。

程序运行的环境

程序运行的过程中，主要是跟两个硬件（CPU 和内存）以及一个软件（操作系统）打交道。



本质上，我们的程序只关心 CPU 和内存这两个硬件。你可能说：“不对啊，计算机还有其他硬件，比如显示器和硬盘啊。”但对我们的程序来说，操作这些硬件，也只是执行某些特定的驱动代码，跟执行其他代码并没有什么差异。

1. 关注 CPU 和内存

CPU 的内部有很多组成部分，对于本课程来说，我们重点关注的是**寄存器以及高速缓存，**它们跟程序的执行机制和优化密切相关。

寄存器是 CPU 指令在进行计算的时候，临时数据存储的地方。CPU 指令一般都会用到寄存器，比如，典型的一个加法计算（c=a+b）的过程是这样的：

> 指令 1（mov）：从内存取 a 的值放到寄存器中；
> 指令 2（add）：再把内存中 b 的值取出来与这个寄存器中的值相加，仍然保存在寄存器中；
> 指令 3（mov）：最后再把寄存器中的数据写回内存中 c 的地址。

寄存器的速度也很快，所以能用寄存器就别用内存。尽量充分利用寄存器，是编译器做优化的内容之一。

而高速缓存可以弥补 CPU 的处理速度和内存访问速度之间的差距。所以，我们的指令在内存读一个数据的时候，它不是老老实实地只读进当前指令所需要的数据，而是把跟这个数据相邻的一组数据都读进高速缓存了。这就相当于外卖小哥送餐的时候，不会为每一单来回跑一趟，而是一次取一批，如果这一批外卖恰好都是同一个写字楼里的，那小哥的送餐效率就会很高。

内存和高速缓存的速度差异差不多是两个数量级，也就是一百倍。比如，高速缓存的读取时间可能是 0.5ns，而内存的访问时间可能是 50ns。不同硬件的参数可能有差异，但总体来说是几十倍到上百倍的差异。

你写程序时，尽量把某个操作所需的数据都放在内存中的连续区域中，不要零零散散地到处放，这样有利于充分利用高速缓存。**这种优化思路，叫做数据的局部性。**

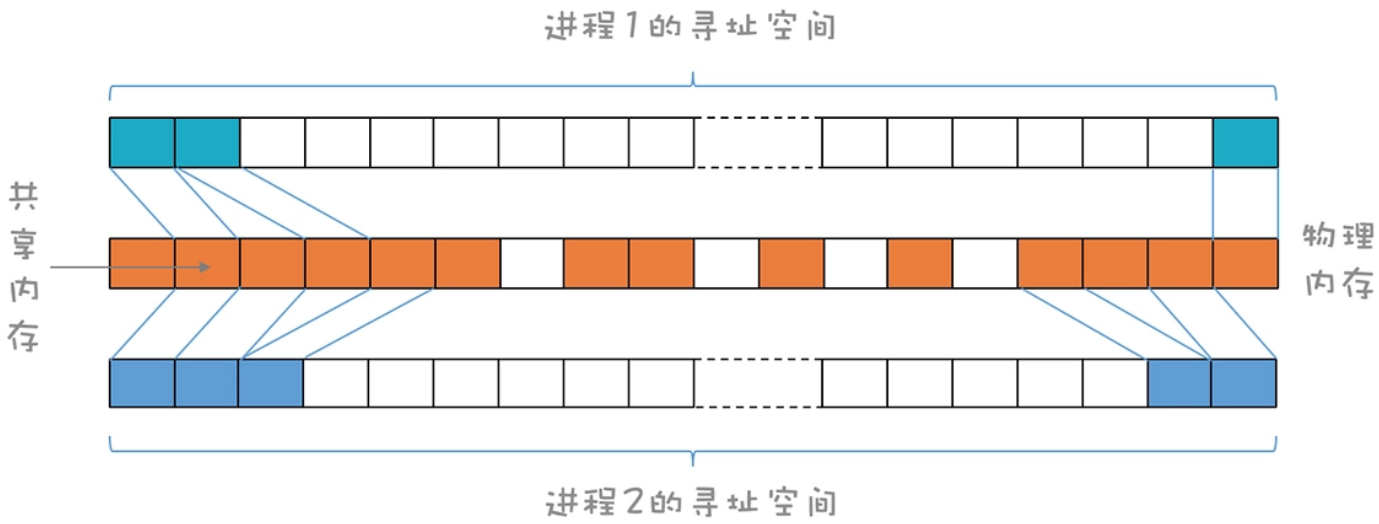
这里提一句，在写系统级的程序时，你要对各种 IO 的时间有基本的概念，比如高速缓存、内存、磁盘、网络的 IO 大致都是什么数量级的。因为这都影响到系统的整体性能，也影响到你如何做程序优化。如果你需要对程序做更多的优化，还需要了解更多的 CPU 运行机制，包括流水线机制、并行机制等等，这里就不展开了。

讲完 CPU 之后，还有内存这个硬件。

程序在运行时，操作系统会给它分配一块虚拟的内存空间，让它在运行期可以使用。我们目前使用的都是 64 位的机器，你可以用一个 64 位的长整型来表示内存地址，它能够表示的所有地址，我们叫做寻址空间。

64 位机器的寻址空间就有 2 的 64 次方那么大，也就是有很多很多个 TB（Terabyte），大到你的程序根本用不完。不过，操作系统一般会给予一定的限制，不会给你这么大的寻址空间，比如给到 100 来个 G，这对一般的程序，也足够用了。

在存在操作系统的情况下，程序逻辑上可使用的内存一般大于实际的物理内存。程序在使用内存的时候，操作系统会把程序使用的逻辑地址映射到真实的物理内存地址。有的物理内存区域会映射进多个进程的地址空间。



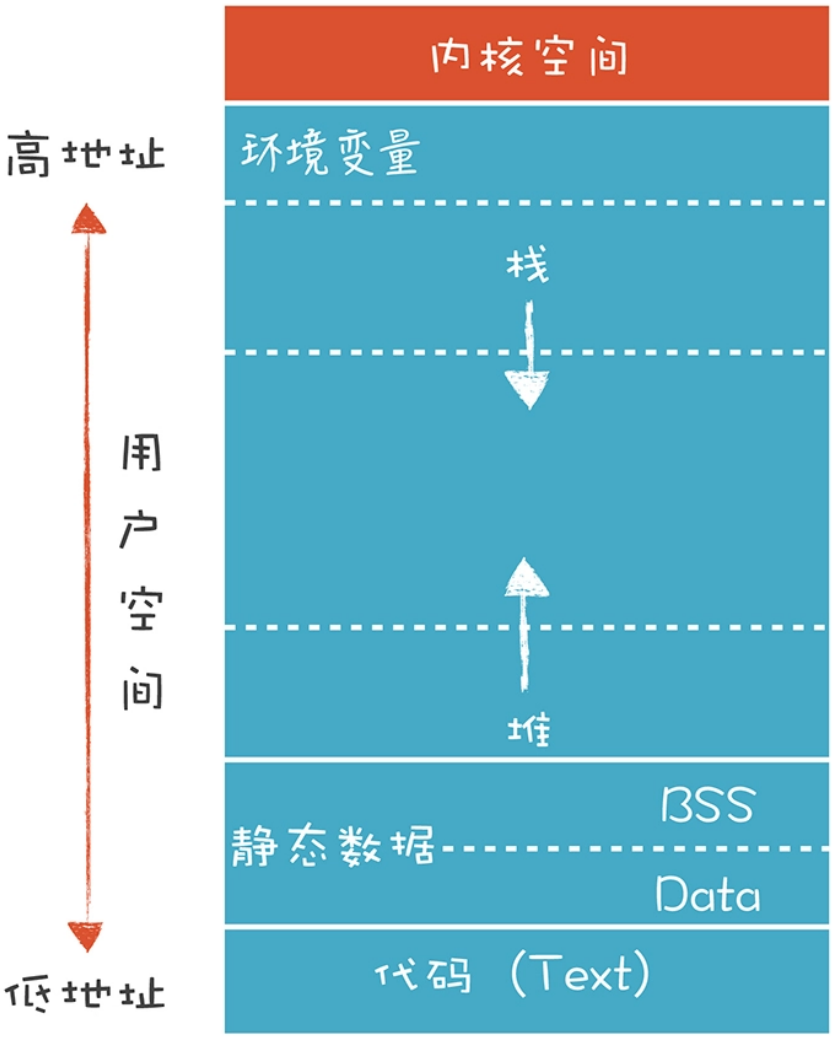
对于不太常用的内存数据，操作系统会写到磁盘上，以便腾出更多可用的物理内存。

当然，也存在没有操作系统的情况，这个时候你的程序所使用的内存就是物理内存，我们必须自己做好内存的管理。

对于这个内存，该怎么用呢?

本质上来说，你想怎么用就怎么用，并没有什么特别的限制。一个编译器的作者，可以决定在哪儿放代码，在哪儿放数据，当然了，别的作者也可能采用其他的策略。实际上，C 语言和 Java 虚拟机对内存的管理和使用策略就是不同的。

尽管如此，大多数语言还是会采用一些通用的内存管理模式。以 C 语言为例，会把内存划分为代码区、静态数据区、栈和堆。



一般来讲，代码区是在最低的地址区域，然后是静态数据区，然后是堆。而栈传统上是从高地址向低地址延伸，栈的最顶部有一块区域，用来保存环境变量。

代码区（也叫文本段）存放编译完成以后的机器码。这个内存区域是只读的，不会再修改，但也不绝对。现代语言的运行时已经越来越动态化，除了保存机器码，还可以存放中间代码，并且还可以在运行时把中间代码编译成机器码，写入代码区。

静态数据区保存程序中全局的变量和常量。它的地址在编译期就是确定的，在生成的代码里直接使用这个地址就可以访问它们，它们的生存期是从程序启动一直到程序结束。它又可以细分为 Data 和 BSS 两个段。Data 段中的变量是在编译期就初始化好的，直接从程序装在进内存。BSS 段中是那些没有声明初始化值的变量，都会被初始化成 0。

堆适合管理生存期较长的一些数据，这些数据在退出作用域以后也不会消失。比如，我们在某个方法里创建了一个对象并返回，并希望代表这个对象的数据在退出函数后仍然可以访问。

而栈适合保存生存期比较短的数据，比如函数和方法里的本地变量。它们在进入某个作用域的时候申请内存，退出这个作用域的时候就可以释放掉。

讲完了 CPU 和内存之后，我们再来看看跟程序打交道的操作系统。

2. 程序和操作系统的关系

程序跟操作系统的关系比较微妙：

> 一方面我们的程序可以编译成不需要操作系统也能运行，就像一些物联网应用那样，完全跑在裸设备上。
>
> 另一方面，有了操作系统的帮助，可以为程序提供便利，比如可以使用超过物理内存的存储空间，操作系统负责进行虚拟内存的管理。

在存在操作系统的情况下，因为很多进程共享计算机资源，所以就要遵循一些约定。这就仿佛办公室是所有同事共享的，那么大家就都要遵守一些约定，如果一个人大声喧哗，就会影响到其他人。

程序需要遵守的约定包括：程序文件的二进制格式约定，这样操作系统才能程序正确地加载进来，并为同一个程序的多个进程共享代码区。在使用寄存器和栈的时候也要遵守一些约定，便于操作系统在不同的进程之间切换的时候、在做系统调用的时候，做好上下文的保护。

所以，我们编译程序的时候，要知道需要遵守哪些约定。因为就算是使用同样的 CPU，针对不同的操作系统，编译的结果也是非常不同的。

好了，我们了解了程序运行时的硬件和操作系统环境。接下来，我们看看程序运行时，是怎么跟它们互动的。

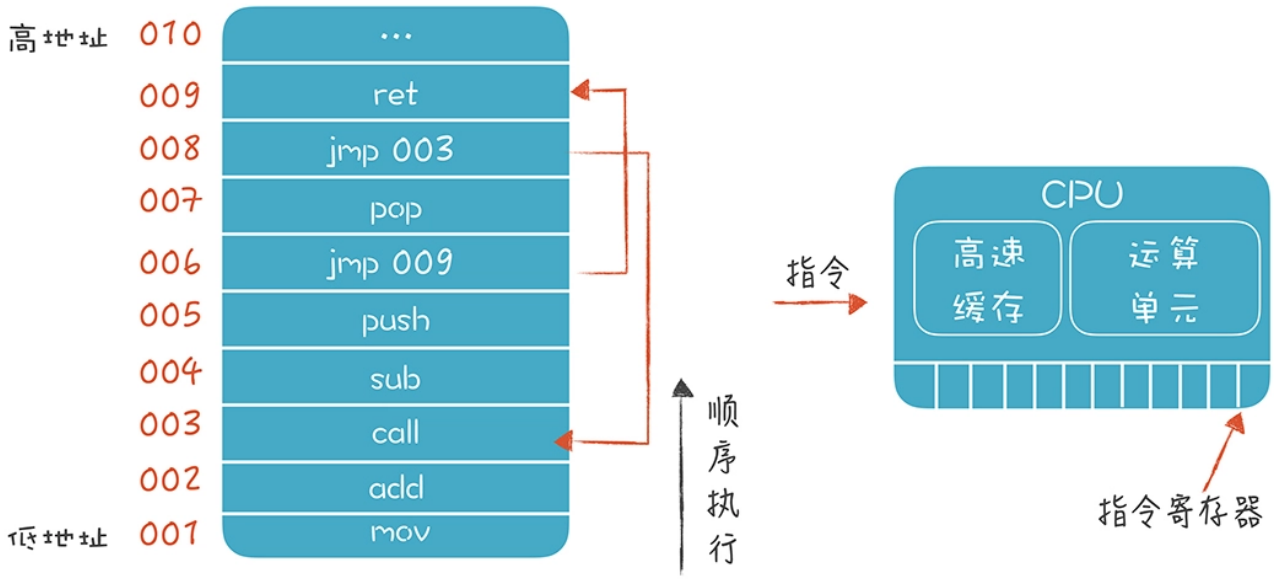
程序运行的过程

你天天运行程序，可对于程序运行的细节，真的清楚吗？

1. 程序运行的细节

首先，可运行的程序一般是由操作系统加载到内存的，并且定位到代码区里程序的入口开始执行。比如，C 语言的 main 函数的第一行代码。

每次加载一条代码，程序都会顺序执行，碰到跳转语句，才会跳到另一个地址执行。CPU 里有一个指令寄存器，里面保存了下一条指令的地址。



假设我们运行这样一段代码编译后形成的程序：

复制代码

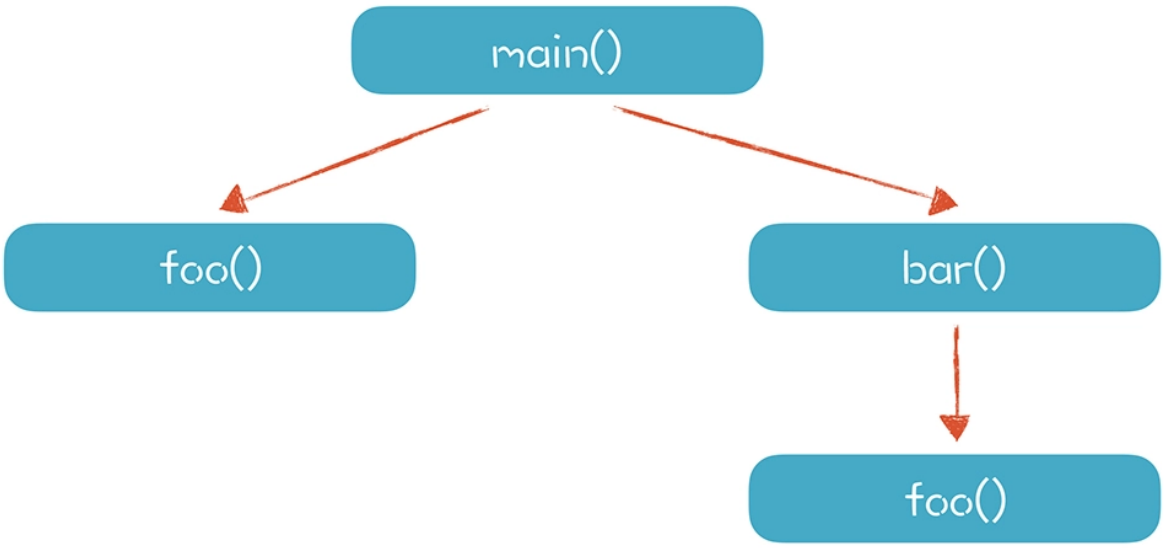
```
int main(){
  int a = 1;
  foo(3);
  bar();
}

int foo(int c){
    int b = 2;
    return b+c;
}

int bar(){
    return foo(4) + 1;
```

```
14 }
```

我们首先激活（Activate）main() 函数，main() 函数又激活 foo() 函数，然后又激活 bar() 函数，bar() 函数还会激活 foo() 函数，其中 foo() 函数被两次以不同的路径激活。



我们把每次调用一个函数的过程，叫做一次活动（Activation）。每个活动都对应一个活动记录（Activation Record），这个活动记录里有这个函数运行所需要的信息，比如参数、返回值、本地变量等。

目前我们用栈来管理内存，所以可以把活动记录等价于栈桢。栈桢是活动记录的实现方式，我们可以自由设计活动记录或栈桢的结构，下图是一个常见的设计：

返回值：一般放在最顶上，这样它的地址是固定的。foo() 函数返回以后，它的调用者可以到这里来取到返回值。在实际情况中，我们会优先通过寄存器来传递返回值，比通过内存传递性能更高。

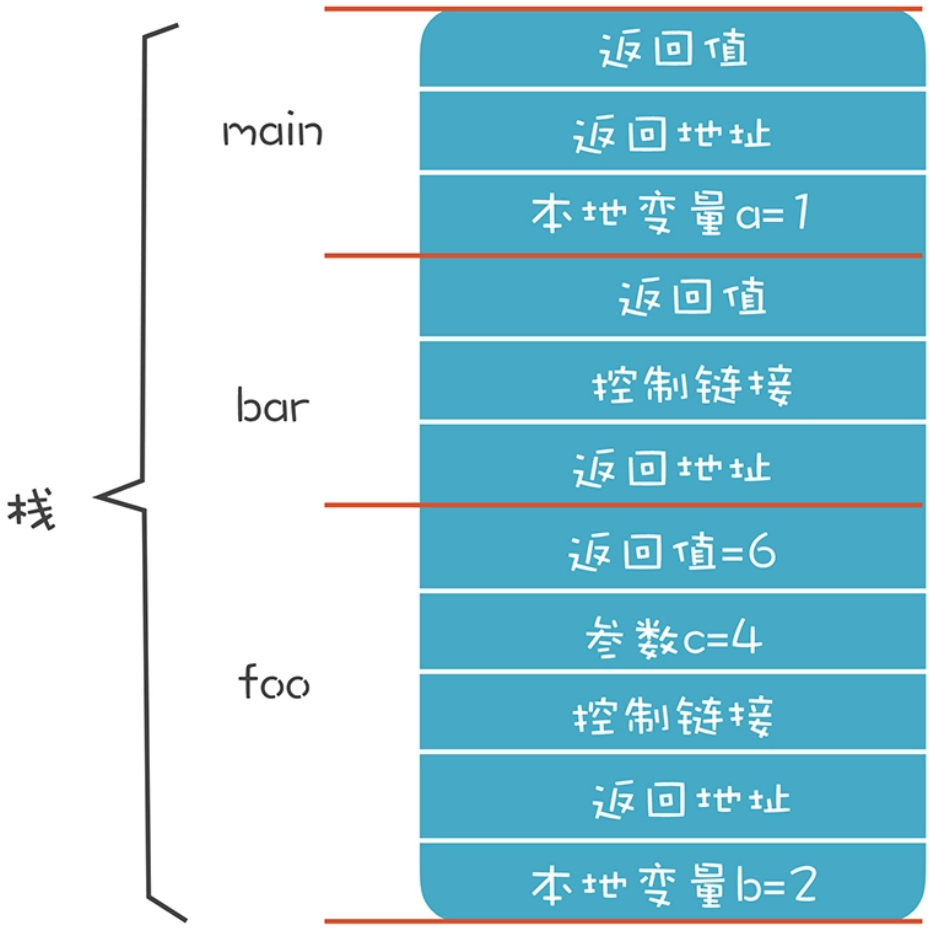
参数：在调用 foo 函数时，把参数写到这个地址里。同样，我们也可以通过寄存器来传递，而不是内存。

控制链接：就是上一级栈桢的地址。如果用到了上一级作用域中的变量，就可以顺着这个链接找到上一级栈桢，并找到变量的值。

返回地址：foo 函数执行完毕以后，继续执行哪条指令。同样，我们可以用寄存器来保存这个信息。

本地变量：foo 函数的本地变量 b 的存储空间。

寄存器信息：我们还经常在栈桢里保存寄存器的数据。如果在 foo 函数里要使用某个寄存器，可能需要先把它的值保存下来，防止破坏了别的代码保存在这里的数据。**这种约定叫做被调用者责任，**也就是使用寄存器的人要保护好寄存器里原有的信息。某个函数如果使用了某个寄存器，但它又要调用别的函数，为了防止别的函数把自己放在寄存器中的数据覆盖掉，要自己保存在栈桢中。**这种约定叫做调用者责任。**



你可以看到，每个栈桢的长度是不一样的。

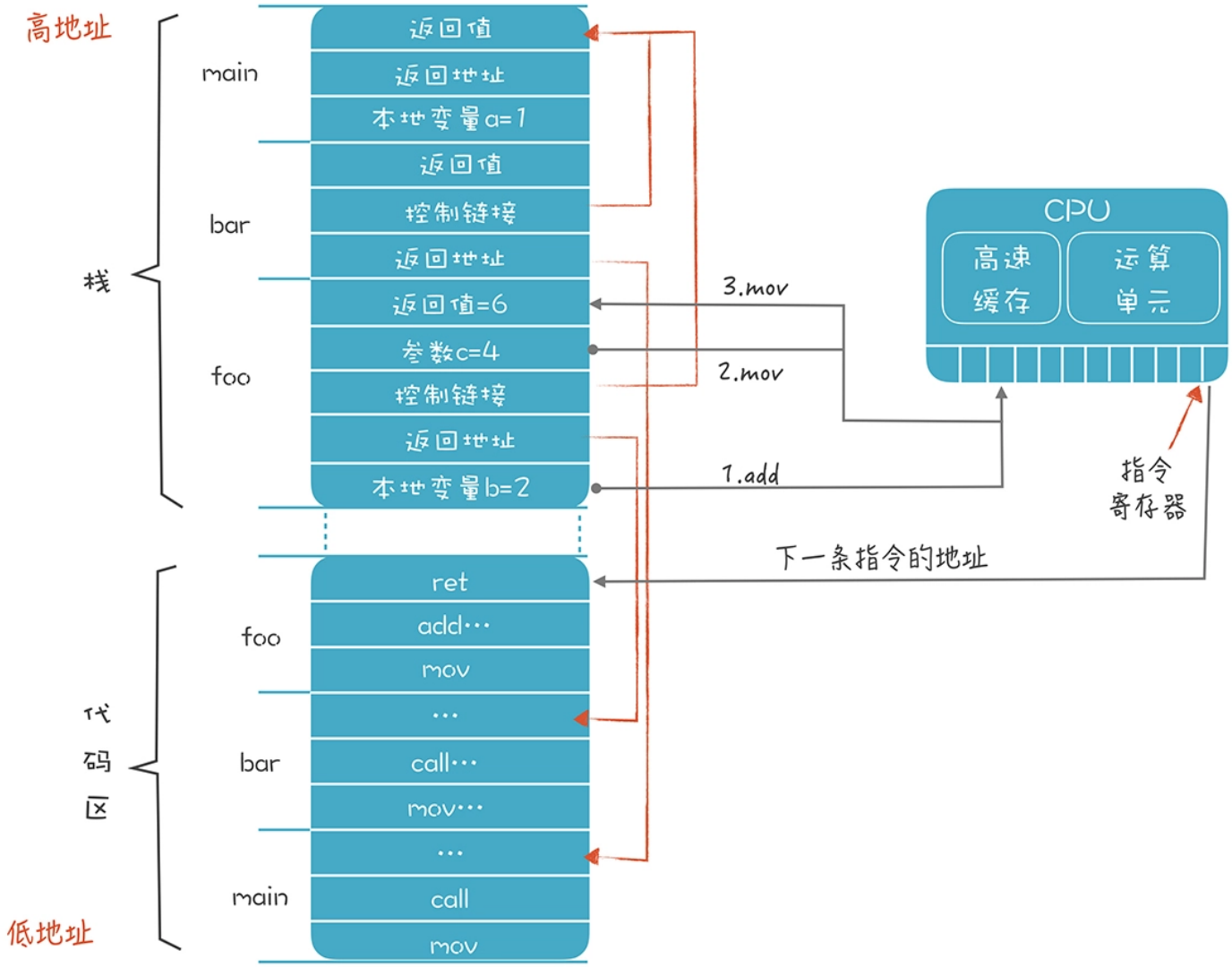
用到的参数和本地变量多，栈桢就要长一点。但是，栈桢的长度和结构是在编译期就能完全确定的。这样就便于我们计算地址的偏移量，获取栈桢里某个数据。

总的来说，栈桢的设计很自由。但是，你要考虑不同语言编译形成的模块要能够链接在一起，所以还是要遵守一些公共的约定的，否则，你写的函数，别人就没办法调用了。

在08 讲，我提到过栈桢，这次我们用了更加贴近具体实现的描述：栈桢就是一块确定的内存，变量就是这块内存里的地址。在下一讲，我会带你动手实现我们的栈桢。

2. 从全局角度看整个运行过程

了解了栈桢的实现之后，我们再来看一个更大的场景，从全局的角度看看整个运行过程中都发生了什么。



代码区里存储了一些代码，main 函数、bar 函数和 foo 函数各自有一段连续的区域来存储代码，我用了一些汇编指令来表示这些代码（实际运行时这里其实是机器码）。

假设我们执行到 foo 函数中的一段指令，来计算“b+c”的值，并返回。这里用到了 mov、add、jmp 这三个指令。mov 是把某个值从一个地方拷贝到另一个地方，add 是往某个地方加一个值，jmp 是改变代码执行的顺序，跳转到另一个地方去执行（汇编命令的细节，我们下节再讲，你现在简单了解一下就行了）。

复制代码

```
mov b的地址 寄存器1
add c的地址 寄存器1
mov 寄存器1 foo的返回值地址
jmp 返回地址  //或ret指令
```

执行完这几个指令以后，foo 的返回值位置就写入了 6，并跳转到 bar 函数中执行 foo 之后的代码。

这时，foo 的栈桢就没用了，新的栈顶是 bar 的栈桢的顶部。理论上讲，操作系统这时可以把 foo 的栈桢所占的内存收回了。比如，可以映射到另一个程序的寻址空间，让另一个程序使用。但是在这个例子中你会看到，即使返回了 bar 函数，我们仍要访问栈顶之外的一个内存地址，也就是返回值的地址。

所以，目前的调用约定都规定，程序的栈顶之外，仍然会有一小块内存（比如 128K）是可以由程序访问的，比如我们可以拿来存储返回值。这一小段内存操作系统并不会回收。

我们目前只讲了栈，堆的使用也类似，只不过是要手工进行申请和释放，比栈要多一些维护工作。

课程小结

本节课，我带你了解了程序运行的环境和过程，我们的程序主要跟 CPU、内存，以及操作系统打交道。你需要了解的重点如下：

- CPU 上运行程序的指令，运行过程中要用到寄存器、高速缓存来提高指令和数据的存取效率。
- 内存可以划分成不同的区域保存代码、静态数据，并用栈和堆来存放运行时产生的动态数据。
- 操作系统会把物理的内存映射成进程的寻址空间，同一份代码会被映射进多个进程的内存空间，操作系统的公共库也会被映射进进程的内存空间，操作系统还会自动维护栈。

程序在运行时顺序执行代码，可以根据跳转指令来跳转；栈被划分成栈桢，栈桢的设计有一定的自由度，但通常也要遵守一些约定；栈桢的大小和结构在编译时就能决定；在运行时，栈桢作为活动记录，不停地被动态创建和释放。

以上这些内容就是一个程序运行时的秘密。你再面对代码时，脑海里就会想象出它是怎样跟 CPU、内存和操作系统打交道的了。而且有了这些背景知识，你也可以让编译器生成代码，按

照本节课所说的模式运行了！

一课一思

本节课，我概要地介绍了程序运行的环境和运行过程。常见的静态编译型的语言，比如 C 语言、Go 语言，差不多都是这个模式。那么你是否了解你所采用的计算机语言的运行环境和运行过程？跟本文描述的哪些地方相同，哪些地方不同？欢迎在留言区分享你的经验。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

精选留言 (16)

鹏 置顶

2019-10-25

宫老师讲的真好！一直以来，都有一种强烈的信念！要学编译原理，要学这种在日新月异的信息技术领域里“亘古不变”的技术原理，以不变应万变。上半年学习了极客时间出的“深入浅出计算机组成原理”，在这一节正好排上用场，对程序运行时机制有了更深入的理解。

> 作者回复: 对的。
> 计算机的组成原理、编译器的基本架构等等内容，其实半个世纪也没有太大的变化，是比较稳定的。
> 并且，真正做一些深入的工作的时候，这些知识仍然非常有价值。

Gopher

2019-10-09

写的真好，一下子就听懂了(￣∀￣)

内存布局：
指令数据，分而治之；
自下而上，由静至动；
栉比鳞次，序从中来。

作者回复: 你不光代码写得好，文采也很好。
新东方的三驾马车之一的王强说到，好的代码就像诗歌一样优美。写完代码要站在远处欣赏一下 :-)

12

刘強
2019-11-05

文章里说操作系统还会自动维护栈。
但我觉得栈的维护是有程序或者编译器来维护的。操作系统只是给程序（进程）分配了栈的起始地址而已，剩下的进栈和出栈操作，都是预先编译好的push和pop指令来完成的。不知理解的对不对。

作者回复: 栈这个东西，如果深入看一下，其实涉及得还挺多的。

操作系统的介入，主要原因是内存管理。因为你的程序所使用的内存，并不是物理内存，都是虚拟出来的。在你使用栈的时候，操作系统要帮你把逻辑的内存映射到物理的内存上去。只不过这个过程对程序是透明的。是CPU和操作系统之间协作完成的。

具体细节是：当你push一个新的变量到栈里的时候，如果超出了当前可用的物理内存，CPU会产生一个page fault（缺页错误），操作系统这个时候介入，调度一页新的物理内存过来。

你可以查看Intel的手册，看看push指令的说明，里面有介绍。关于page fault，你可以参考https://en.wikipedia.org/wiki/Page_fault

当然，如果你的程序直接运行在裸机上，没有使用操作系统，那就没有操作系统什么事了。直接在裸机上编程叫做bare metal programming，在有些领域很有用。

总结起来，栈的管理，与CPU、内存和你的代码都有关。

共 2 条评论>

8

wolfie
2020-12-13

“控制链接”和“上一帧的%rbp的值”两者分列返回地址的两边，它们是怎样的关系？

作者回复: 控制链接通常是指向上一级作用域的函数对应的栈桢，用于查找上一级作用域中变量的值，它不一定是前一个栈桢。

比如，有两个函数，foo和bar，他们都是在全局作用域中声明的，而在foo里调用了bar。那么，bar的控制链接是指向全局作用域（的栈桢），可以访问全局变量，而不是指向foo（的栈桢）。

共 2 条评论 > 5

sunbird
2020-11-21

一节一节，看的根本停不下来。看过才发现，自己根本不会计算机。比MOOC上那些顶尖名牌大学的教授讲的好太多了！！！

有几个问题还是想的不太明白，麻烦宫老师在***有时间的时候***帮忙解惑一下，不胜感激。

1.这个栈是操作系统维护的栈吗?
2.这个栈和数据结构中的栈有什么区别?
3.这个栈是每个程序一个，还是所有的程序共用一个?
4.这个栈是和进程绑定的吗?
5.多线程的时候，每个线程都有自己的方法栈，和这个栈是一个吗？如果不是，他们之间有什么区别?
6.栈为什么是高地址向低地址的延伸？栈顶在高地址还是低地址？栈寄存器指向的是栈顶吗?

作者回复: 谬赞了!

回答一下你的问题:

1.是的，栈是操作系统维护的，所以会降低程序维护内存的负担。但缺点是函数退出后栈里的内存就自动收回，这点是跟堆的区别。

2.数据结构中的栈是一个普遍的概念，可以用在很多地方。内存管理是栈的一个具体应用。

3.在现在操作系统中，是每个线程一个栈。如果一个进程里面有多个线程，那就有多个栈。

另外，如果程序是支持协程的，有些实现机制也会给协程提供单独的栈，用来维护协程的状态。但协程的栈一般是由程序的运行时或库来支持的，而不是由操作系统来维护的。

4.同上，栈是跟线程绑定的。

5.同上，每个线程有自己单独的栈。

6.栈从高地址到低地址延伸，这个时候栈顶是低地址。对于这个问题，没有什么特别的道理，只是一个设计决定而已。并且，并不是所有的架构都这样设计，有的设计恰好是反向的。栈寄存器的设计也是跟具体架构相关的。对于比较新的x86架构来说，我们一般只需要一个指向栈顶的栈寄存器即可。但，由于调用约定通常要向下兼容，所以生成的程序通常也要用到一个寄存器指向栈底。

对于上述问题，我想额外指出几点:

1.上述具体实现，跟CPU架构和操作系统二者都有关系，不是死的。编译器在实现的时候，要生成针对具体架构的代码。不要认为这些都是一成不变的。

2.在《编译原理实战课》中，对于运行时有更多的介绍，比如栈和线程的关系，协程的机制等等。你可以参考。

5

吴小智
2020-01-08

太赞了，老师一文道破计算机专业本科生四年需要学的 70% 专业领域的知识，底层知识扎实很重要。

作者回复: 谬赞了。
不过学习编译原理，确实会用到计算机学科的多方面的知识，如形式语言、数据结构和算法、计算机组成、操作系统等。这也不奇怪，因为你要让一门语言跑起来，就是要涉及方方面面。

5

曾经瘦过
2019-10-09

使用的java 语言。java是运行在jvm虚拟机里面的，是便以为jvm所需的机器码 基本的过程和这个是差不多的。看了这一篇专栏之后 发现基础知识的用处真的很多，操作系统 组成原理 用处真多。

作者回复: 对呀。既然学计算机嘛，就搞到根本上去，心里会比较踏实。而且说实话，基础原理并不多，也不易变。反倒上层各种类库、框架，层出不穷，天天更新。这两头哪边学起来更辛苦，真不一定!

4

westfall
2020-05-21

像js这种非编译性语言又是怎么跑起来的呢?

作者回复: 通常，是要通过一个解释器来执行。而解释器执行的一般是中间代码，比如java和python的字节码。

还有一种情况，就是把js快速编译成机器码，然后执行。因为是快速编译，所以不够优化，代码体积也比较大，但好处是编译时间很短，可以马上跑起来。如果一段代码经常被执行，就意味着这个热代码，那么就进行优化编译，产生更好的代码。这就是早期JavaScript引擎做编译的流程。但最新的v8

引擎，采用的是一个解释器（ignition）加一个优化编译器（turbofan）的结构，看上去跟JVM很像，都是一个解释器加一个优化编译器。

我在第二季《编译原理实战课》的运行环境这一讲，增加了栈机和寄存器机这两种虚拟机的介绍。另外，也有对v8编译器的分析。你可以去看看。

3

阿辉
2021-04-15

老师说的真好，通过老师的讲解，运行时的机制主要是操作作系统系统级别，那编译器起的主要作用是？我们通常所说的runtime到底指的是什么？和vm,engine之类的有啥区别啊？

作者回复: 当我们谈论一门语言的时候，通常三个组成部分是必不可少的：编译器、运行时和标准库。此外，一般还要加上一些工具，比如模块管理工具、调试工具等等。
运行时，顾名思义，就是你编写的程序在运行的时候所需要的那些软件。这个定义比较泛，所以在Windows系统中，把一些标准库也称作运行时，因为确实是在Windows上运行软件所必须的。
而虚拟机，则是比较复杂的那种运行时，典型代表是HotSpot虚拟机和V8虚拟机，它们为程序的运行、内存管理、并发等都提供了支持，程序可以无缝地在解释执行和运行本地代码之间切换、适时地启动自动编译机制等。
你可以看出，这种级别的运行时和操作系统的功能有很多相似之处。比如，要做好内存的管理，其实你需要对操作系统的内存管理也了解得比较深，并使它们二者良好的配合。并发也是如此。

2

微秒
2020-03-16

老师你好，想问下你在文中说的静态数据期的地址在编译时就确定，这里的地址是虚拟地址，还是实际内存地址。

作者回复: 应用程序能够看到的地址，都是操作系统虚拟出来的，除非你直接运行在裸机上。

2

upon you
2022-06-09

老师好，请问如果函数调用返回了一个值，但是这个值没有被使用，那么这个值怎么处理呢？比如下面的形式

```
while(1)
{
   get();  // return 100
   dispatch();
}
```


Join
2021-11-06

太通俗易懂啦，最近刚看完龙书的运行时，再看这节，感觉在复习


淡
2020-04-26

宫老师，你好。文章中对于函数返回后，写到：
”但是在这个例子中你会看到，即使返回了 bar 函数，我们仍要访问栈顶之外的一个内存地址，也就是返回值的地址。“，这里没太明白，bar函数返回后，返回值地址还在bar栈中，因为外层调用函数要用变量（也可以理解为外层栈空间）接受这个返回值，接受完了是不是就可以释放了？是因为有其他比较特殊的场景如外层异步调用内层函数？

共 1 条评论>

宋健
2020-04-04

老师讲的真不错，上学期学校正好学了汇编，也做过实验，果然感觉后端要比前端好理解很多呀

作者回复: 谢谢肯定!
可是很多同学都会觉得后端更难呢:-)


风
2019-10-19

怎么没提push和pop呢

作者回复: 在22讲，汇编语言的部分就有。

D

2019-10-10

有些汇编的语法和上面的是反着的，比如 指令：寄存器 ， 源操作数/地址

> 作者回复: 是的。我们用的都是GNU汇编的语法。第22讲正式讲汇编的时候特别做了说明。看看是不是在21讲提到汇编时也注释一下。