

## 11 | 堆和栈：函数调用是如何影响到内存布局的？

2020-04-09 李兵

图解 Google V8

[进入课程 >](#)



**讲述：李兵**

时长 17:25 大小 15.96M



你好，我是李兵。


相信你在使用 JavaScript 的过程中，经常会遇到栈溢出的错误，比如执行下面这样一段代码：

```
1 function foo() {
2   foo() // 是否存在堆栈溢出错误？
3 }
4 foo()
```

复制代码




V8 就会报告**栈溢出**的错误，为了解决栈溢出的问题，我们可以在 foo 函数内部使用 setTimeout 来触发 foo 函数的调用，改造之后的程序就可以正确执行。

 复制代码

```
1 function foo() {  
2   setTimeout(foo, 0) // 是否存在堆栈溢出错误?  
3 }
```

如果使用 Promise 来代替 setTimeout，在 Promise 的 then 方法中调用 foo 函数，改造的代码如下：

 复制代码

```
1 function foo() {  
2   return Promise.resolve().then(foo)  
3 }  
4 foo()
```

在浏览器中执行这段代码，并没有报告栈溢出的错误，但是你会发现，执行这段代码会让整个页面卡住了。

为什么这三段代码，第一段造成栈溢出的错误，第二段能够正确执行，而第三段没有栈溢出的错误，却会造成页面的卡死呢？

其主要原因是这三段代码的底层执行逻辑是完全不同的：

第一段代码是在同一个任务中重复调用嵌套的 foo 函数；

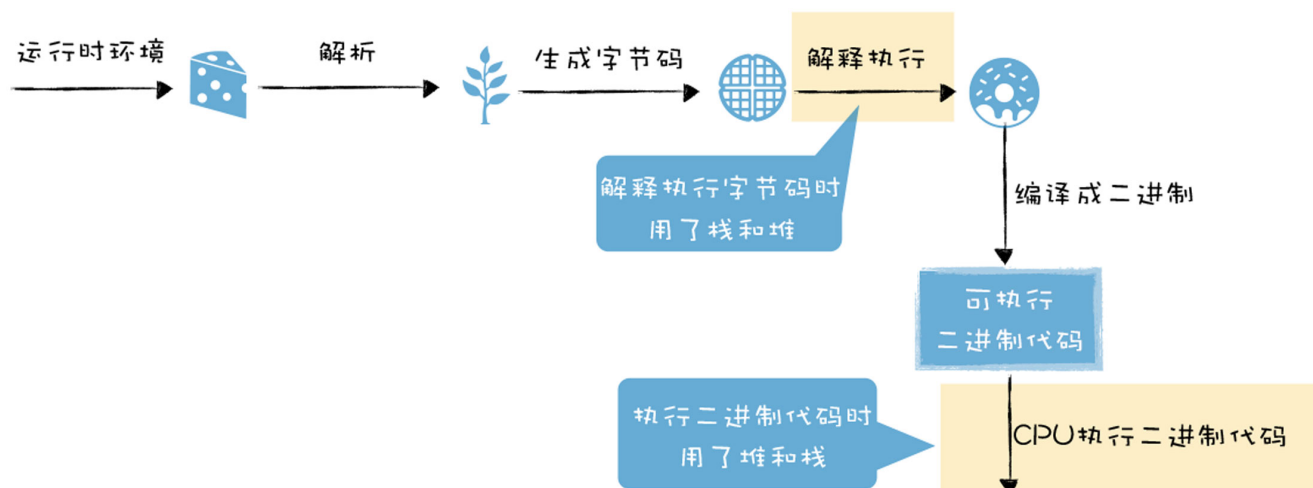
第二段代码是使用 setTimeout 让 foo 函数在不同的任务中执行；

第三段代码是在同一个任务中执行 foo 函数，但是却不是嵌套执行。

这是因为，V8 执行这三种不同代码时，它们的内存布局是不同的，而不同的内存布局又会影响到代码的执行逻辑，因此我们需要了解 JavaScript 执行时的内存布局。

这节课，我们从函数特性入手，来一步步延伸出通用的函数调用模型，进而来分析不同的函数调用方式是如何影响到运行时内存布局的。

下图是本文的主要内容在编译流水线中的位置，因为解释执行和直接执行二进制代码都使用了堆和栈，虽然它们在执行细节上存在着一定的差异，但是整体的执行架构是类似的。



## 为什么使用栈结构来管理函数调用？

我们知道，大部分高级语言都不约而同地采用栈这种结构来管理函数调用，为什么呢？这与函数的特性有关。通常函数有两个主要的特性：

1. 第一个特点是函数**可以被调用**，你可以在一个函数中调用另外一个函数，当函数调用发生时，执行代码的控制权将从父函数转移到子函数，子函数执行结束之后，又会将代码执行控制权返还给父函数；
2. 第二个特点是函数**具有作用域机制**，所谓作用域机制，是指函数在执行的时候可以将定义在函数内部的变量和外部环境隔离，在函数内部定义的变量我们也称为**临时变量**，临时变量只能在该函数中被访问，外部函数通常无权访问，当函数执行结束之后，存放在内存中的临时变量也随之被销毁。

我们可以看下面这段 C 代码：

```
1 int getZ()  
2 {  
3     return 4;
```

复制代码

```

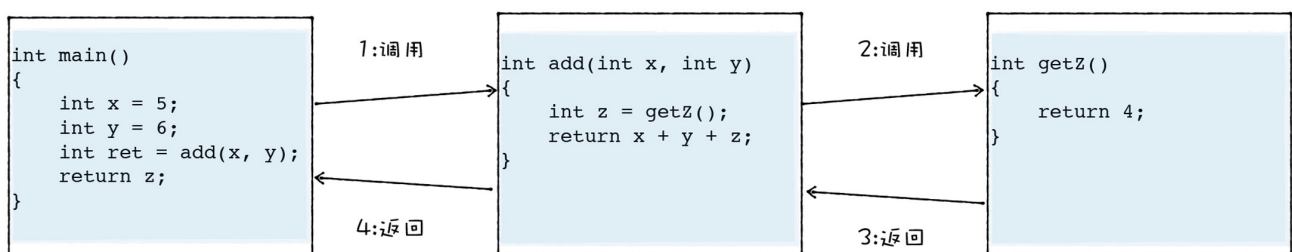
4  }
5  int add(int x, int y)
6  {
7      int z = getZ();
8      return x + y + z;
9  }
10 int main()
11 {
12     int x = 5;
13     int y = 6;
14     int ret = add(x, y);
15 }

```

观察上面这段代码，我们发现其中包含了多层函数嵌套调用，其实这个过程很简单，执行流程是这样的：

1. 当 main 函数调用 add 函数时，需要将代码执行控制权交给 add 函数；
2. 然后 add 函数又调用了 getZ 函数，于是又将代码控制权转交给 getZ 函数；
3. 接下来 getZ 函数执行完成，需要将控制权返回给 add 函数；
4. 同样当 add 函数执行结束之后，需要将控制权返还给 main 函数；
5. 然后 main 函数继续向下执行。

具体的函数调用示意图如下：

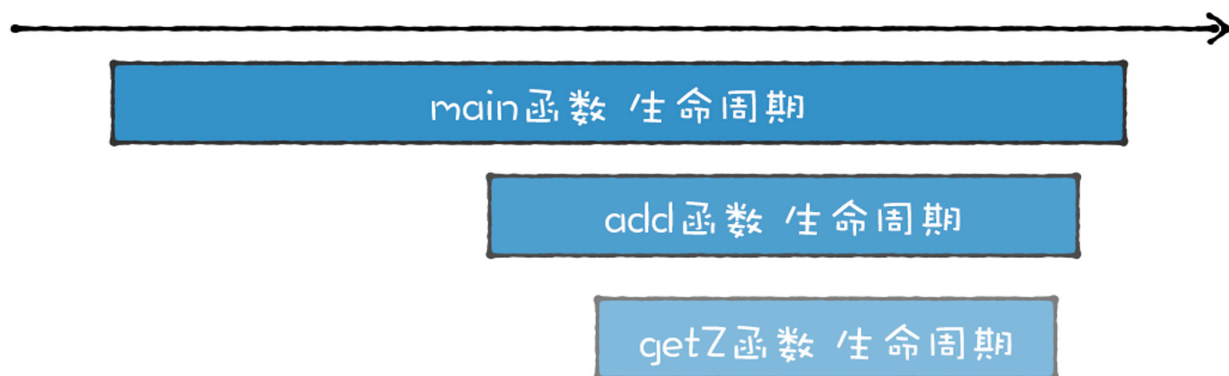


## 函数调用示意图

通过上述分析，我们可以得出，**函数调用者的生命周期总是长于被调用者（后进），并且被调用者的生命周期总是先于调用者的生命周期结束（先出）。**

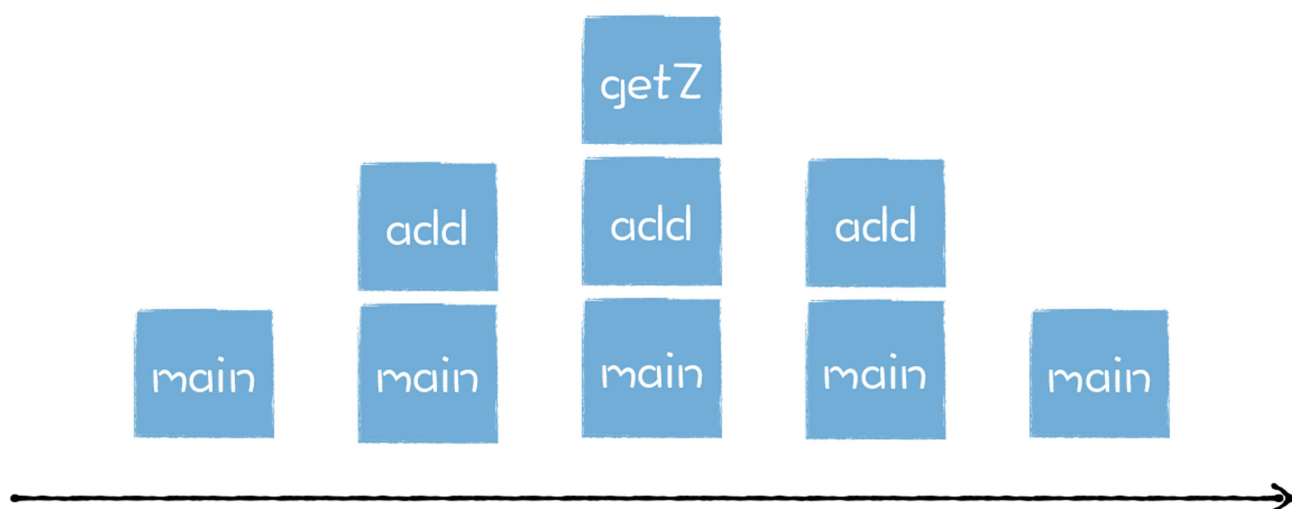
在执行上述流程时，各个函数的生命周期如下图所示：

## 时间线



嵌套调用时函数的生命周期

因为函数是有作用域机制的，作用域机制通常表现在函数执行时，会在内存中分配函数内部的变量、上下文等数据，在函数执行完成之后，这些内部数据会被销毁掉。**所以站在函数资源分配和回收角度来看，被调用函数的资源分配总是晚于调用函数 (后进)，而函数资源的释放则总是先于调用函数 (先出)。**如下图所示：



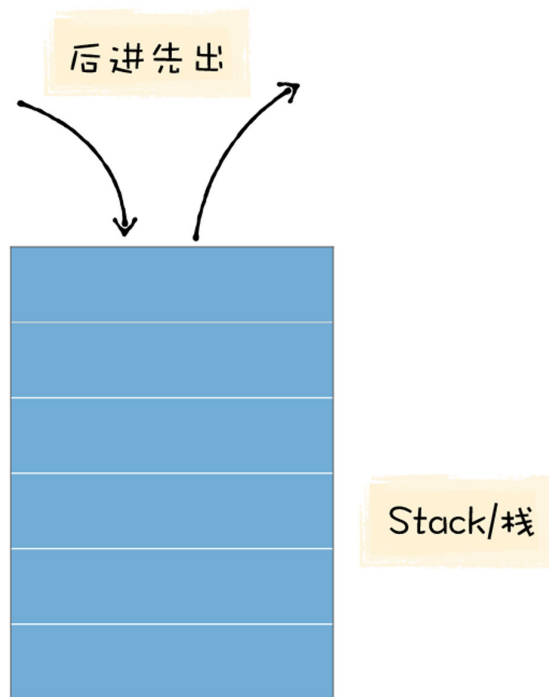
函数资源分配流程

通过观察函数的生命周期和函数的资源分配情况，我们发现，它们都符合**后进先出 (LIFO)** 的策略，而栈结构正好满足这种后进先出 (LIFO) 的需求，所以我们选择栈来管理函数调用关系是一种很自然的选择。

关于栈，你可以结合这么一个贴切的例子来理解，一条单车道的单行线，一端被堵住了，而另一端入口处没有任何提示信息，堵住之后就只能后进去的车子先出来（后进先出），这时

这个堵住的单行线就可以被看作是一个**栈容器**，车子开进单行线的操作叫做**入栈**，车子倒出去的操作叫做**出栈**。

在车流量较大的场景中，就会发生反复地入栈、栈满、出栈、空栈和再次入栈，一直循环。你可以参看下图：



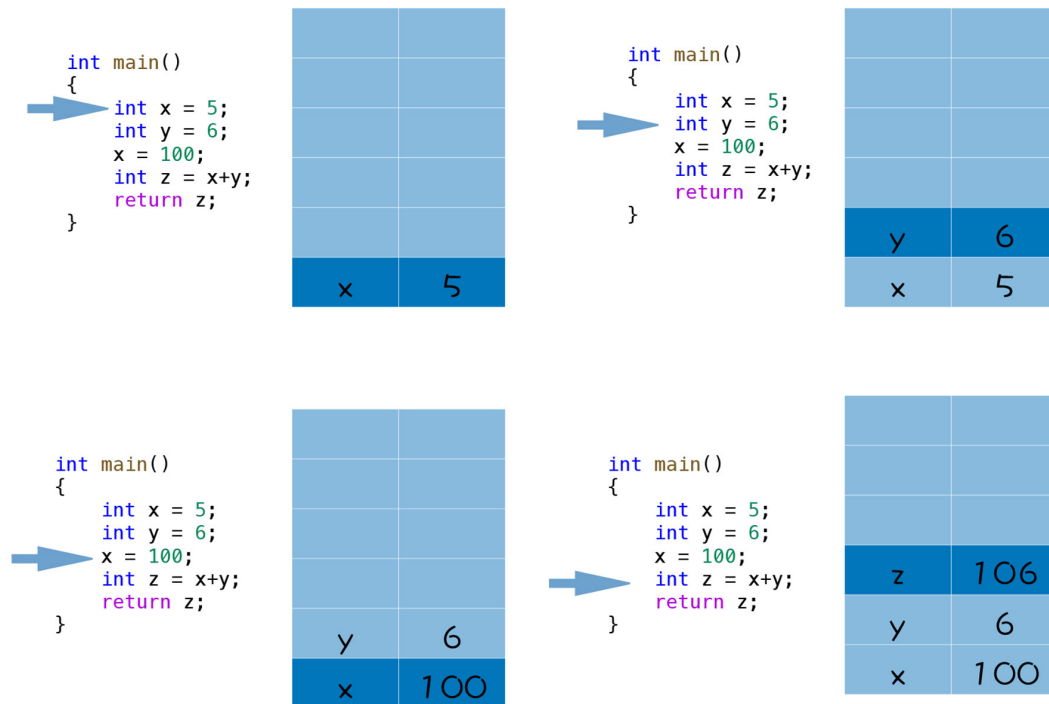
数据结构层面的栈

## 栈如何管理函数调用？

了解了栈的特性之后，我们就来看看栈是如何管理函数调用的。

首先我们来分析最简单的场景：当执行一个函数的时候，栈怎么变化？

当一个函数被执行时，函数的参数、函数内部定义变量都会依次压入到栈中，我们结合实际的代码来分析下这个过程，你可以参考下图：



函数内部变量压栈状态

上图展示的是一段简单的 C 代码的执行过程，可以看到：

当执行到函数的第一段代码的时候，变量 x 第一次被赋值，且值为 5，这时 5 会被压入到栈中。

然后，执行第二段代码，变量 y 第一次被赋值，且值为 6，这时 6 会被压入到栈中。

接着，执行到第三段代码，注意这里变量 x 是第二次被赋值，且新的值为 100，那么这时并不是将 100 压入到栈中，而是替换以前压入栈的内容，也就是将栈中的 5 替换成 100。

最后，执行第四段代码，这段代码是 `int z = x + y`，我们会先计算出来 `x+y` 的值，然后再将 `x+y` 的值赋值给 `z`，由于 `z` 是第一次被赋值，所以 `z` 的值也会被压入到栈中。

你会发现，**函数在执行过程中，其内部的临时变量会按照执行顺序被压入到栈中。**

了解了这一点，接下来我们就可以分析更加复杂一点的场景了：当一个函数调用另外一个函数时，栈的变化情况是怎样的？我们还是先看下面这段代码：

```
1 int add(num1,num2){
2     int x = num1;
3     int y = num2;
```

复制代码

```

4     int ret = x + y;
5     return ret;
6 }
7
8
9 int main()
10 {
11     int x = 5;
12     int y = 6;
13     x = 100;
14     int z = add(x+y);
15     return z;
16 }

```

观察上面这段代码，我们把上段代码中的 `x+y` 改造成了一个 `add` 函数，当执行到 `int z = add(x,y)` 时，当前栈的状态如下所示：

```

int main()
{
    int x = 5;
    int y = 6;
    x = 100;
    int z = add(x+y);
    return z;
}

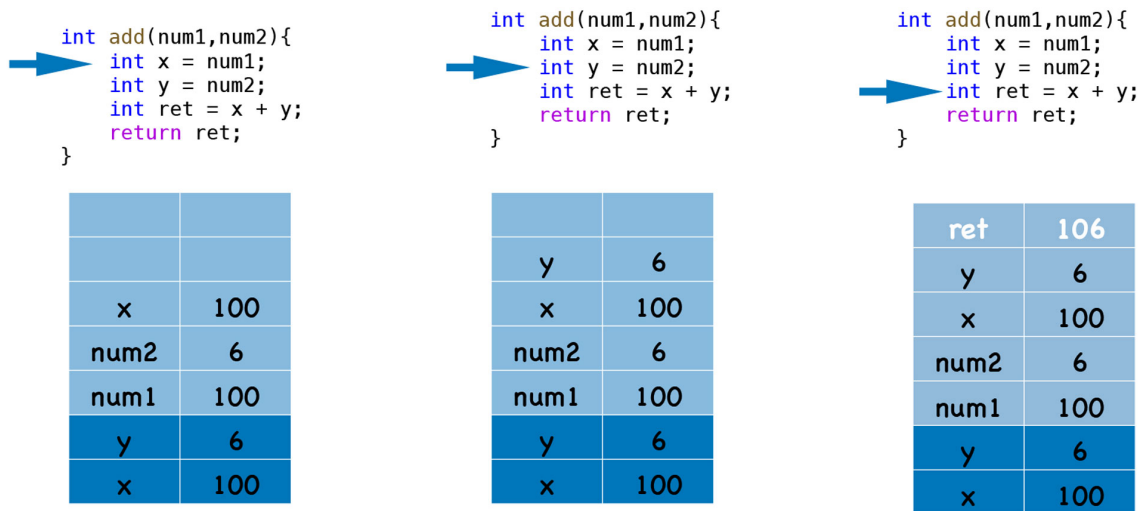
```

➡

y	6
x	100

接下来，就要调用 `add` 函数了，理想状态下，执行 `add` 函数的过程是下面这样的：

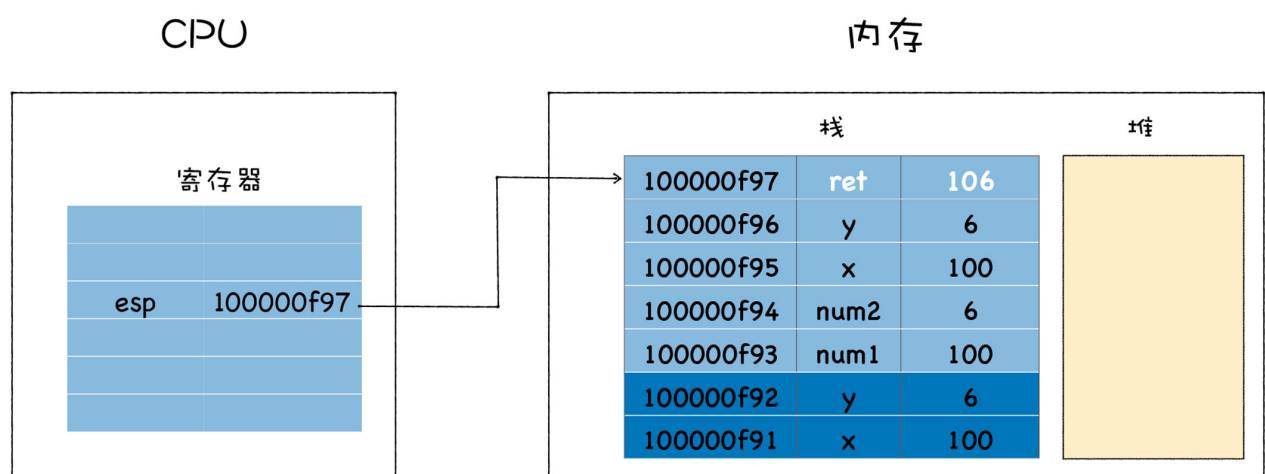




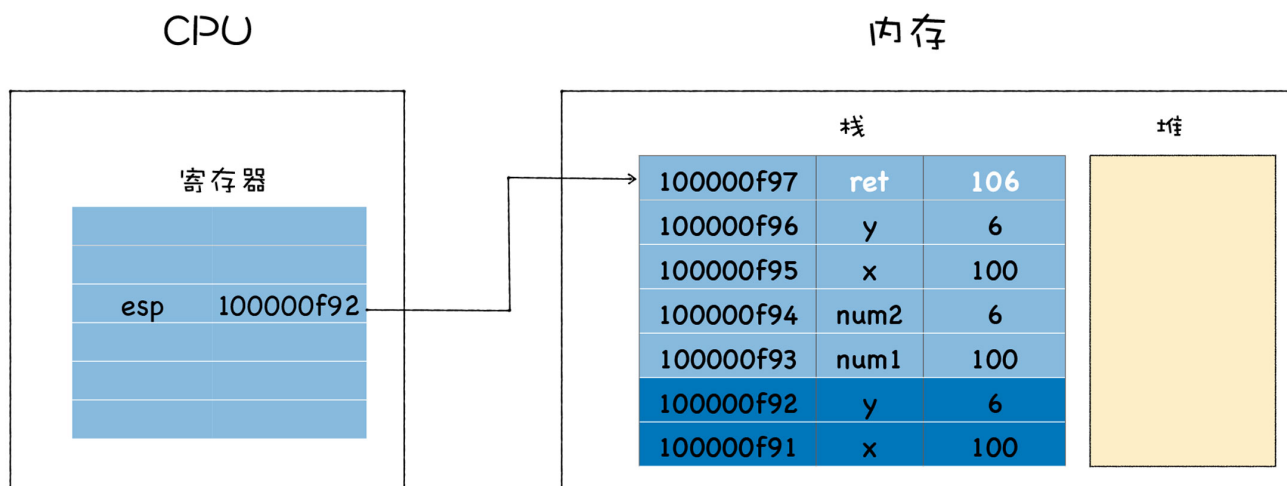
当执行到 `add` 函数时，会先把参数 `num1` 和 `num2` 压栈，接着我们再把变量 `x`、`y`、`ret` 的值依次压栈，不过执行这里，会遇到一个问题，那就是当 `add` 函数执行完成之后，需要将执行代码的控制权转交给 `main` 函数，这意味着需要将栈的状态恢复到 `main` 函数上次执行时的状态，我们把这个过程叫**恢复现场**。那么应该怎么恢复 `main` 函数的执行现场呢？

其实方法很简单，只要在寄存器中保存一个永远指向当前栈顶的指针，栈顶指针的作用就是告诉你应该往哪个位置添加新元素，这个指针通常存放在 `esp` 寄存器中。如果你想往栈中添加一个元素，那么你需要先根据 `esp` 寄存器找到当前栈顶的位置，然后在栈顶上方添加新元素，新元素添加之后，还需要将新元素的地址更新到 `esp` 寄存器中。

有了栈顶指针，就很容易恢复 `main` 函数的执行现场了，当 `add` 函数执行结束时，只需要将栈顶指针向下移动就可以了，具体你可以参看下图：



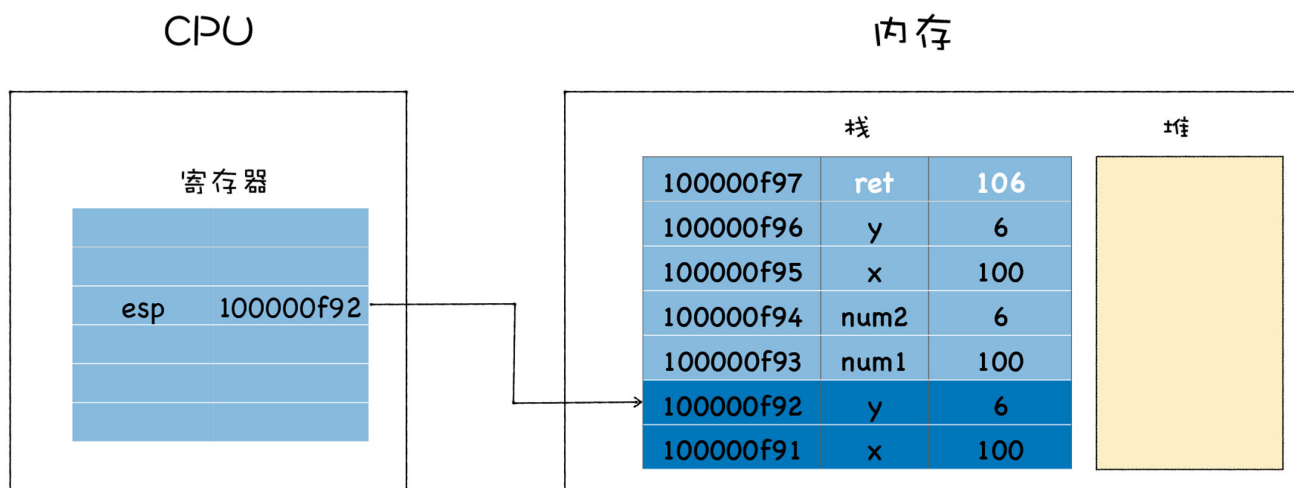
add函数即将执行结束的状态



恢复mian函数执行现场

观察上图，将 `esp` 的指针向下移动到之前 `main` 函数执行时的地方就可以，不过新的问题又来了，CPU 是怎么知道要移动到这个地址呢？

CPU 的解决方法是增加了另外一个 `ebp` 寄存器，用来保存当前函数的起始位置，我们把一个函数的起始位置也称为栈帧指针，`ebp` 寄存器中保存的就是当前函数的栈帧指针，如下图所示：



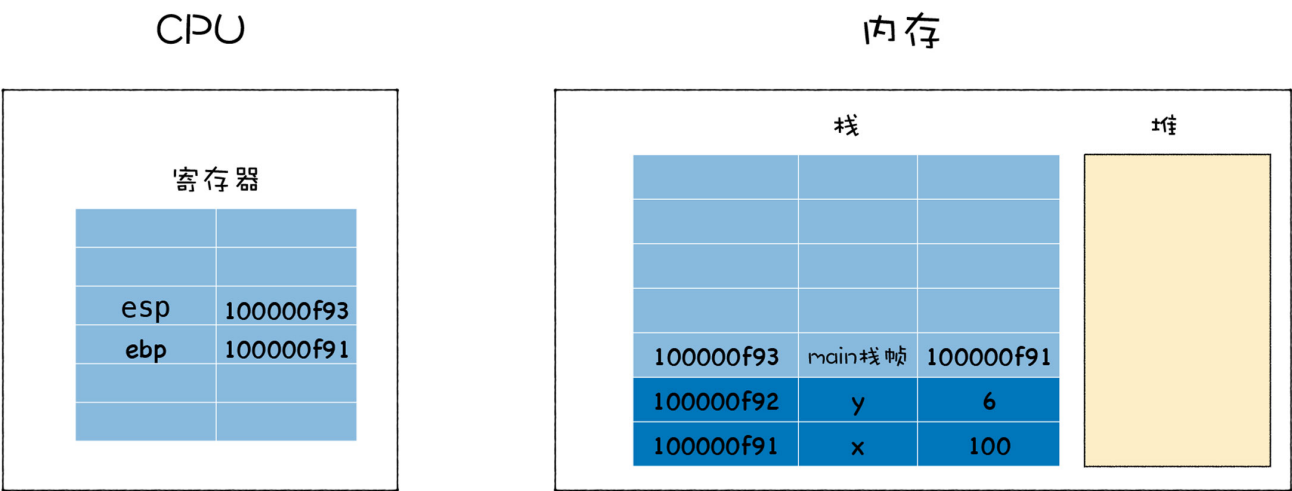
ebp寄存器保存了栈帧指针

在 `main` 函数调用 `add` 函数的时候，`main` 函数的栈顶指针就变成了 `add` 函数的栈帧指针，所以需要将 `main` 函数的栈顶指针保存到 `ebp` 中，当 `add` 函数执行结束之后，我需要销毁 `add` 函数的栈帧，并恢复 `main` 函数的栈帧，那么只需要取出 `main` 函数的栈顶指针写到 `esp` 中即可 (`main` 函数的栈顶指针是保存在 `ebp` 中的)，这就相当于将栈顶指针移动到 `main` 函数的区域。

那么现在，我们可以执行 main 函数了吗？

答案依然是“不能”，这主要是因为 main 函数也有它自己的栈帧指针，在执行 main 函数之前，我们还需恢复它的栈帧指针。如何恢复 main 函数的栈帧指针呢？

通常的方法是在 main 函数中调用 add 函数时，CPU 会将当前 main 函数的栈帧指针保存在栈中，如下图所示：



当函数调用结束之后，就需要恢复 main 函数的执行现场了，首先取出 ebp 中的指针，写入 esp 中，然后从栈中取出之前保留的 main 的栈帧地址，将其写入 ebp 中，到了这里 ebp 和 esp 就都恢复了，可以继续执行 main 函数了。

另外在这里，我们还需要补充下**栈帧**的概念，因为在很多文章中我们会看到这个概念，每个栈帧对应着一个未运行完的函数，栈帧中保存了该函数的返回地址和局部变量。

以上我们详细分析了 C 函数的执行过程，在 JavaScript 中，函数的执行过程也是类似的，如果调用一个新函数，那么 V8 会为该函数创建栈帧，等函数执行结束之后，销毁该栈帧，而栈结构的容量是固定的，所有如果重复嵌套执行一个函数，那么就会导致栈会栈溢出。

了解了这些，现在我们再回过头来看下这节课开头提到的三段代码。

第一段代码由于循环嵌套调用了 foo，所以当函数运行时，就会导致 foo 函数会不断地调用 foo 函数自身，这样就会导致栈无限增，进而导致栈溢出的错误。

第二段代码是在函数内部使用了 `setTimeout` 来启动 `foo` 函数，这段代码之所以不会导致栈溢出，是因为 `setTimeout` 会使得 `foo` 函数在消息队列后面的任务中执行，所以不会影响到当前的栈结构。也就不会导致栈溢出。关于消息队列和事件循环系统，我们会在最后一单元来介绍。

最后一段代码是 `Promise`，`Promise` 的情况比较特别，既不会造成栈溢出，但是这种方式会导致主线的卡死，这就涉及到了微任务，关于微任务在这里我们先不展开介绍，我会在微任务这一节来详细介绍。

## 既然有了栈，为什么还要堆？

好了，我们现在理解了栈是怎么管理函数调用的了，使用栈有非常多的优势：

1. 栈的结构和非常适合函数调用过程。
2. 在栈上分配资源和销毁资源的速度非常快，这主要归结于栈空间是连续的，分配空间和销毁空间只需要移动下指针就可以了。

虽然操作速度非常快，但是栈也是有缺点的，其中最大的缺点也是它的优点所造成的，那就是栈是连续的，所以要想在内存中分配一块连续的大空间是非常难的，因此栈空间是有限的。

因为栈空间是有限的，这就导致我们在编写程序的时候，经常一不小心就会导致栈溢出，比如函数循环嵌套层次太多，或者在栈上分配的数据过大，都会导致栈溢出，基于栈不方便存放大的数据，因此我们使用了另外一种数据结构用来保存一些大数据，这就是**堆**。

和栈空间不同，存放在堆空间中的数据是不要求连续存放的，从堆上分配内存块没有固定模式的，你可以在任何时候分配和释放它，为了更好地理解堆，我们看下面这段代码是怎么执行的：

```
1 struct Point
2 {
3     int x;
4     int y;
5 };
6
7
```

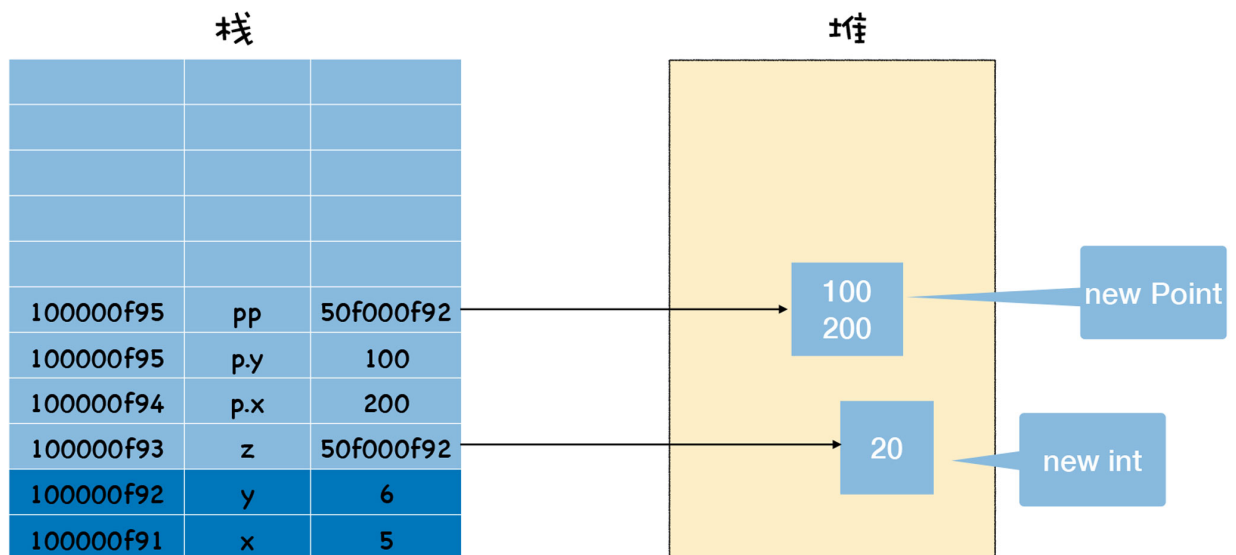
 复制代码

```

8  int main()
9  {
10     int x = 5;
11     int y = 6;
12     int *z = new int;
13     *z = 20;
14
15
16     Point p;
17     p.x = 100;
18     p.y = 200;
19
20
21     Point *pp = new Point();
22     pp->y = 400;
23     pp->x = 500;
24     delete z;
25     delete pp;
26     return 0;


```

观察上面这段代码，你可以看到代码中有 `new int`、`new Point` 这种语句，当执行这些语句时，表示要在堆中分配一块数据，然后返回指针，通常返回的指针会被保存到栈中，下面我们来看看当 `main` 函数快执行结束时，堆和栈的状态，具体内容你可以参看下图：



观察上图，我们可以发现，当使用 `new` 时，我们会在堆中分配一块空间，在堆中分配空间之后，会返回分配后的地址，我们会把该地址保存在栈中，如上图中的 `p` 和 `pp` 都是地址，它们保存在栈中，指向了在堆中分配的空间。

通常，当堆中的数据不再需要的时候，需要对其进行销毁，在 C 语言中可以使用 free，在 C++ 语言中可以使用 delete 来进行操作，比如可以通过：

 复制代码

```
1 delete p;  
2 delete pp;
```

来销毁堆中的数据，像 C/C++ 这种手动管理内存的语言，如果没有手动销毁堆中的数据，那么就会造成内存泄漏。不过 JavaScript，Java 使用了自动垃圾回收策略，可以实现垃圾自动回收，但是事情总有两面性，垃圾自动回收也会给我们带来一些性能问题。所以不管是自动垃圾回收策略，还是手动垃圾回收策略，要想写出高效的代码，我们都需要了解内存的底层工作机制。

## 总结


因为现代语言都是基于函数的，每个函数在执行过程中，都有自己的生命周期和作用域，当函数执行结束时，其作用域也会被销毁，因此，我们会使用栈这种数据结构来管理函数的调用过程，我们也把管理函数调用过程的栈结构称之为**调用栈**。

因为栈在内存中连续的数据结构，所以在通常情况下，栈都有最大容量限制的，这也就意味着，函数的嵌套调用次数过多，就会超出栈的最大使用范围，从而导致栈溢出。

为了解决栈溢出的问题，我们可以使用 setTimeout 将要执行的函数放到其他的任务中去执行，也可以使用 Promise 来改变栈的调用方式，这涉及到了事件循环和微任务，我们会在后续课程中再来介绍。

## 思考题

你可以分析一下，在浏览器中执行文中的这段代码，为什么不会造成栈溢出，但是却会造成页面的卡死？欢迎你在留言区与我分享讨论。

 复制代码

```
1 function foo() {  
2     return Promise.resolve().then(foo)  
3 }  
4 foo()
```



感谢你的阅读，如果你觉得这篇文章对你有所启发，也欢迎把它分享给你的朋友。

# 图解 Google V8

一门课搞懂 JavaScript 执行逻辑

李兵

前盛大创新院高级研究员



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 10 | 机器代码：二进制机器码究竟是如何被CPU执行的？

下一篇 12 | 延迟解析：V8是如何实现闭包的？

## 精选留言 (11)

💬 写留言



Geek\_49d301

2020-04-09

没写过js代码，猜测js应该也有两个队列，一个微任务队列一个事件队列，然后微任务队列的优先级高于事件队列，由于微任务队列一直被占用导致后面事件队列永远无法执行直到卡死

作者回复: 微任务队列属于当前任务，也就是说，微任务队列中的微任务、一定会在当前正在执行的任务退出之前执行完毕！

💬 1

👍 2



**Peter Cheng**

2020-04-09

JS的事件机制有宏任务和微任务。

宏任务是setTimeout、requestAnimationFrame、用户输入事件（I/O）等，它是由浏览器的队列完成的，在浏览器的主进程中进行，页面不会卡死。

...

展开 ∨

作者回复: 主线程通过一个while循环一直拉取微任务

这句话可以修改下：主线程在当前任务快要执行结束之前，检查微任务队列中是否存在微任务，如果有，那么那么那么当前任务会依次取出微任务队列中的微任务，并一一执行！

补充一点，微任务队列是属于当前宏任务的，所以一个宏任务中产生的微任务，只会放到它自己的微任务队列中！



1



**刘大夫**

2020-04-10

这相当于在当前这一轮任务里不停地创建微任务，执行，创建，执行，创建.....虽然不会爆栈，但也无法去执行下一个任务，主线程被卡在这里了，所以页面会卡死

作者回复: 理解的很透彻了 🍷



1



**wuqilv**

2020-04-10

"答案依然是“不能”，这主要是因为 main 函数也有它自己的栈帧指针，在执行 main 函数之前，我们还需恢复它的栈帧指针。如何恢复 main 函数的栈帧指针呢？通常的方法是在 main 函数中调用 add 函数时，CPU 会将当前 main 函数的栈顶指针保存在栈中，" 这里前面说的是 main 函数栈帧，后面就变成了main 函数的栈顶，看着有点迷糊。

展开 ∨

作者回复: 我写错了，已经改正过来了 🙄







一步

2020-04-09

Promise属于微任务会在当前事件循环执行，一直会占用主线程，导致页面无法进行渲染，所以导致页面卡死

作者回复: 会在当前的任务中重复执行，导致当前任务无法退出，阻塞消息队列中的其它任务的执行！

我把你这句中的/当前事件循环\修改了下



王楚然

2020-04-09

思考题：

不知道对不对，

1. 不会栈溢出，是因为Promise类似setTimeout将foo放入了任务队列。
2. 会卡死，是因为，Promise会将foo放入微任务队列，该队列在每次事件循环中都需要清空才能进行下一次事件循环。对比的setTimeout是将foo放入宏任务队列，该队列每次事...  
展开

作者回复: 微任务并没有进消息队列，它只是当前任务的一个未完成部分。



冯剑

2020-04-09

请问下栈帧是一个逻辑内存还是物理内存？

展开

作者回复: 程序使用了自己的进程空间，64位系统，每个程序的虚拟进程空间是 $2^{64}$ ，实际上使用时，才会在物理内存中开辟空间！

所以可以说，栈空间都是程序的虚拟空间，使用的地址和实际内存中的地址是不一样的，中间还做了一层映射！



桃翁

2020-04-09

老师，我有一个疑惑哈，就是我看你图中这些变量是按照栈来存的，那么当访问先入栈的变量的时候岂不是要把后入栈的弹出去才能访问？但是我觉得肯定不会这么做，老师能解释下怎么访问栈底部的变量么？

作者回复: 不需要弹出来啊，比如栈帧指针地址是1000，那么变量x对应栈顶的偏移是16，那么变量x的内存位置就是1000-16(通常情况下，栈的方向是向下增长的)

2



伏枫

2020-04-09

1. promise不会造成栈溢出，是因为foo函数在执行到promise时，会将then的回调函数放到可执行任务队列中，然后就会执行完毕，直接出栈。虽然then的回调函数也是foo，但是栈中永远只有一个foo的栈。

2. 关于为啥会造成主线卡死，我不是很确定，只能猜测一下。卡死的原因是浏览器的页面渲染和js执行共用一个线程，js如果一直执行下去，会导致无法渲染，造成页面卡死的现...

展开

作者回复: 因为promise触发的微任务会一直在当前的任务中执行，这回导致其他的宏任务无法被执行！所以就造成了页面的卡死

2



luckyone

2020-04-09

快速扫了一遍文章，发现有些地方不能理解，然后仔细看，发现图画错了-\_-||

作者回复: 能够告知下哪张图哈

2



zz

2020-04-09

虽然是异步方式调用，js主线程中任务结束后，开始执行事件循环，导致event queue中不断的进和出队列。

2



