

13-垃圾回收：垃圾数据是如何自动回收的？

在[上一篇文章](#)中，我们提到了JavaScript中的数据是如何存储的，并通过例子分析了**原始数据类型是存储在栈空间中的，引用类型的数据是存储在堆空间中的**。通过这种分配方式，我们解决了数据的内存分配的问题。

不过有些数据被使用之后，可能就不再需要了，我们把这种数据称为**垃圾数据**。如果这些垃圾数据一直保存在内存中，那么内存会越用越多，所以我们需要**对这些垃圾数据进行回收，以释放有限的内存空间**。

不同语言的垃圾回收策略

通常情况下，垃圾数据回收分为**手动回收**和**自动回收**两种策略。

如C/C++就是使用手动回收策略，**何时分配内存、何时销毁内存都是由代码控制的**，你可以参考下面这段C代码：

```
//在堆中分配内存
char* p = (char*)malloc(2048); //在堆空间中分配2048字节的空间，并将分配后的引用地址保存到p中

//使用p指向的内存
{
    //....
}

//使用结束后，销毁这段内存
free(p);
p = NULL;
```

从上面这段C代码可以看出来，要使用堆中的一块空间，我们需要先调用malloc函数分配内存，然后再使用；当不再需要这块数据的时候，就要手动调用free函数来释放内存。如果这段数据已经不再需要了，但是又没有主动调用free函数来销毁，那么这种情况就被称为**内存泄漏**。

另外一种使用的是自动垃圾回收的策略，如JavaScript、Java、Python等语言，**产生的垃圾数据是由垃圾回收器来释放的**，并不需要手动通过代码来释放。

对于JavaScript而言，也正是这个“自动”释放资源的特性带来了很多困惑，也让一些JavaScript开发者误以为可以不关心内存管理，这是一个很大的误解。

那么在本文，我们将围绕“JavaScript的数据是如何回收的”这个话题来展开探讨。因为数据是存储在栈和堆两种内存空间中的，所以接下来我们就来分别介绍“栈中的垃圾数据”和“堆中的垃圾数据”是如何回收的。

调用栈中的数据是如何回收的

首先是调用栈中的数据，我们还是通过一段示例代码的执行流程来分析其回收机制，具体如下：

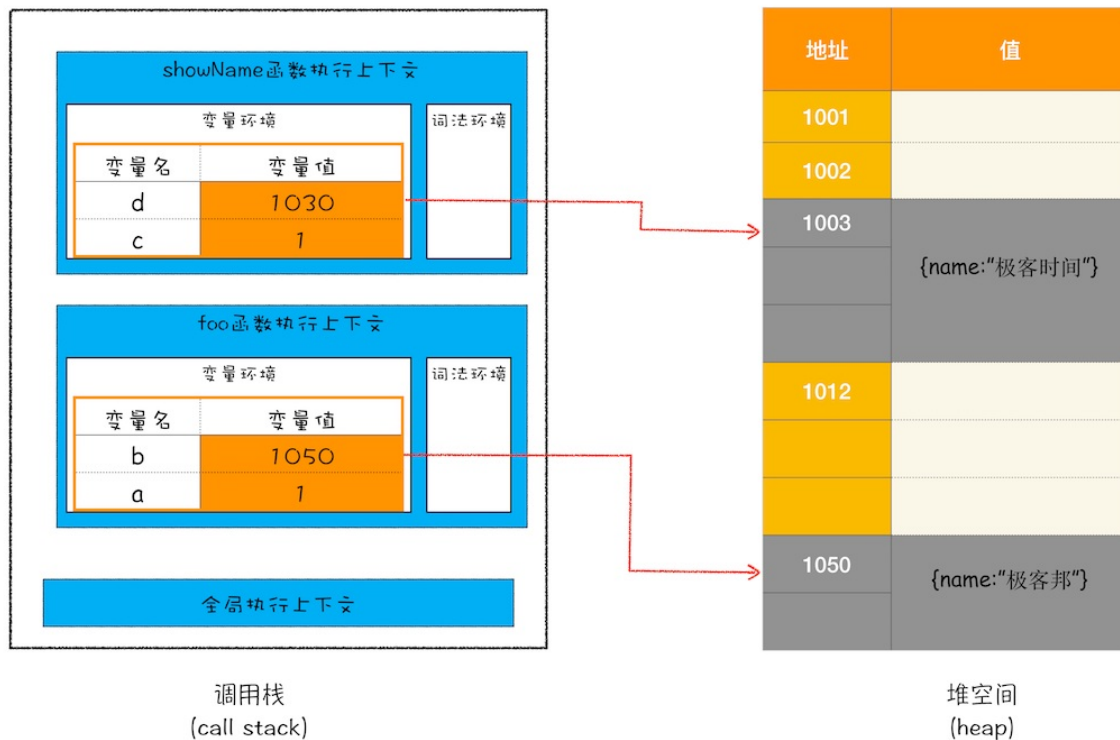
```
function foo(){
```

```

var a = 1
var b = {name:"极客邦"}
function showName(){
  var c = "极客时间"
  var d = {name:"极客时间"}
}
showName()
}
foo()

```

当执行到第6行代码时，其调用栈和堆空间状态图如下所示：



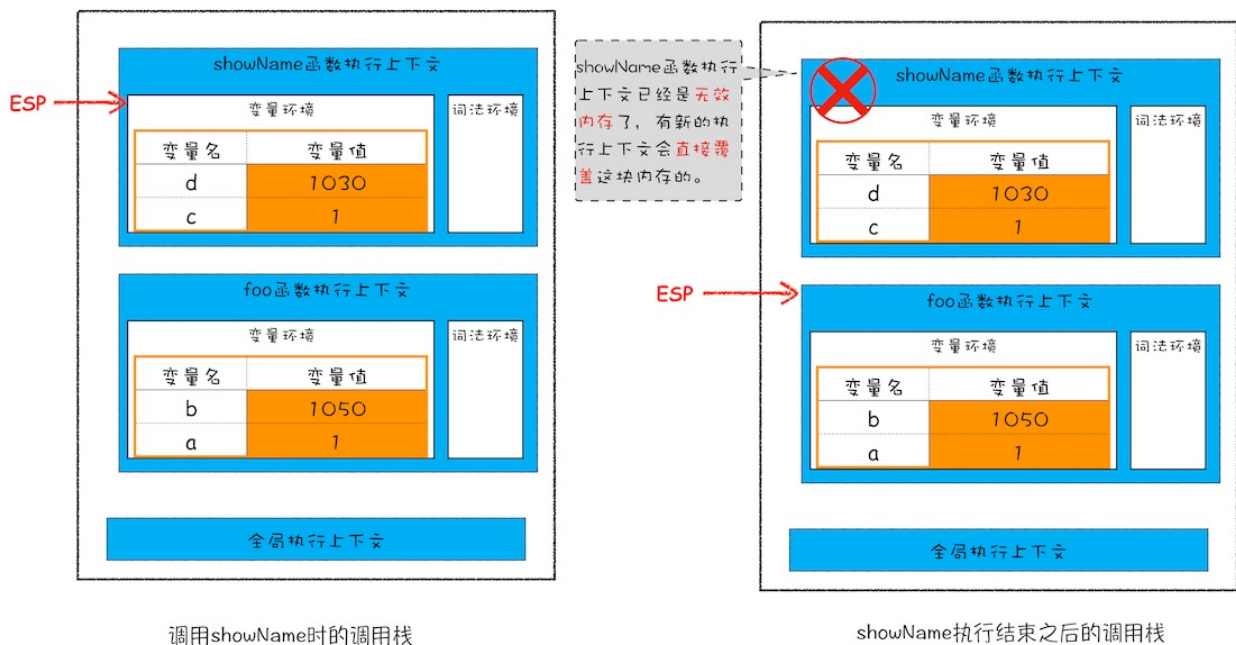
执行到showName函数时的内存模型

从图中可以看出，原始类型的数据被分配到栈中，引用类型的数据会被分配到堆中。当foo函数执行结束之后，foo函数的执行上下文会从堆中被销毁掉，那么它是怎么被销毁的呢？下面我们就来分析一下。

在[上篇文章](#)中，我们简单介绍过了，如果执行到showName函数时，那么JavaScript引擎会创建showName函数的执行上下文，并将showName函数的执行上下文压入到调用栈中，最终执行到showName函数时，其调用栈就如上图所示。与此同时，还有一个**记录当前执行状态的指针（称为ESP）**，指向调用栈中showName函数的执行上下文，表示当前正在执行showName函数。

接着，当showName函数执行完成之后，函数执行流程就进入了foo函数，那这时就需要销毁showName函数的执行上下文了。ESP这时候就帮上忙了，JavaScript会将ESP下移到foo函数的执行上下文，**这个下移操作就是销毁showName函数执行上下文的过程。**

你可能会有点懵，ESP指针向下移动怎么就能把showName的执行上下文销毁了呢？具体你可以看下面这张移动ESP前后的对比图：

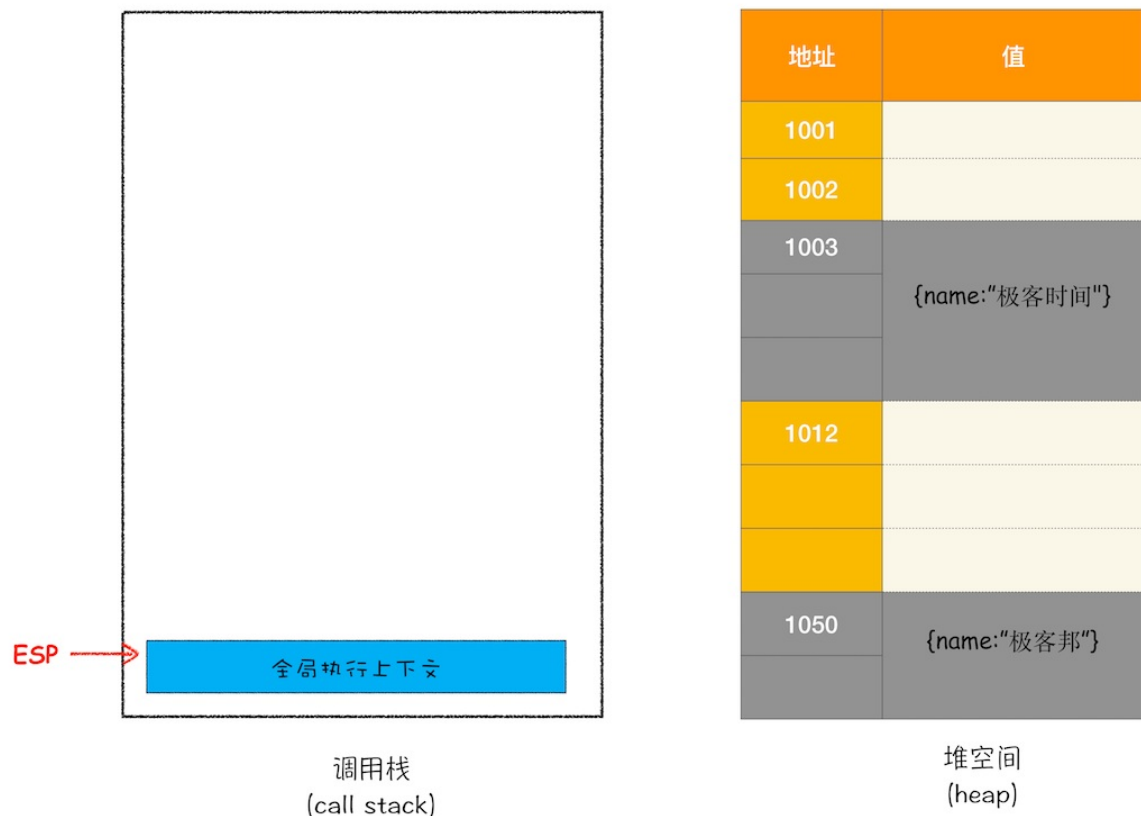


从图中可以看出，当showName函数执行结束之后，ESP向下移动到foo函数的执行上下文中，上面showName的执行上下文虽然保存在栈内存中，但是已经是无效内存了。比如当foo函数再次调用另外一个函数时，这块内容会被直接覆盖掉，用来存放另外一个函数的执行上下文。

所以说，当一个函数执行结束之后，**JavaScript引擎会通过向下移动ESP来销毁该函数保存在栈中的执行上下文。**

堆中的数据是如何回收的

通过上面的讲解，我想现在你应该已经知道，当上面那段代码的foo函数执行结束之后，ESP应该是指向全局执行上下文的，那这样的话，showName函数和foo函数的执行上下文就处于无效状态了，不过保存在堆中的两个对象依然占用着空间，如下图所示：



foo函数执行结束后的内存状态

从图中可以看出，1003和1050这两块内存依然被占用。**要回收堆中的垃圾数据，就需要用到JavaScript中的垃圾回收器了。**

所以，接下来我们就来通过Chrome的JavaScript引擎V8来分析下堆中的垃圾数据是如何回收的。

代际假说和分代收集

不过在正式介绍V8是如何实现回收之前，你需要先学习下**代际假说（The Generational Hypothesis）**的内容，这是垃圾回收领域中一个重要的术语，后续垃圾回收的策略都是建立在该假说的基础之上的，所以很重要。

代际假说有以下两个特点：

- 第一个是大部分对象在内存中存在的时间很短，简单来说，就是很多对象一经分配内存，很快就变得不可访问；
- 第二个是不死的对象，会活得更久。

其实这两个特点不仅仅适用于JavaScript，同样适用于大多数的动态语言，如Java、Python等。

有了代际假说的基础，我们就可以来探讨V8是如何实现垃圾回收的了。

通常，垃圾回收算法有很多种，但是并没有哪一种能胜任所有的场景，你需要权衡各种场景，根据对象的生存周期的不同而使用不同的算法，以便达到最好的效果。

所以，在V8中会把堆分为**新生代**和**老生代**两个区域，**新生代中存放的是生存时间短的对象，老生代中存放**

的生存时间久的对象。

新生区通常只支持1~8M的容量，而老生区支持的容量就大很多了。对于这两块区域，V8分别使用两个不同的垃圾回收器，以便更高效地实施垃圾回收。

- **副垃圾回收器，主要负责新生代的垃圾回收。**
- **主垃圾回收器，主要负责老生代的垃圾回收。**

垃圾回收器的工作流程

现在你知道了V8把堆分成两个区域——新生代和老生代，并分别使用两个不同的垃圾回收器。其实**不论什么类型的垃圾回收器，它们都有一套共同的执行流程。**

第一步是标记空间中活动对象和非活动对象。所谓活动对象就是还在使用的对象，非活动对象就是可以进行垃圾回收的对象。

第二步是回收非活动对象所占据的内存。其实就是在所有的标记完成之后，统一清理内存中所有被标记为可回收的对象。

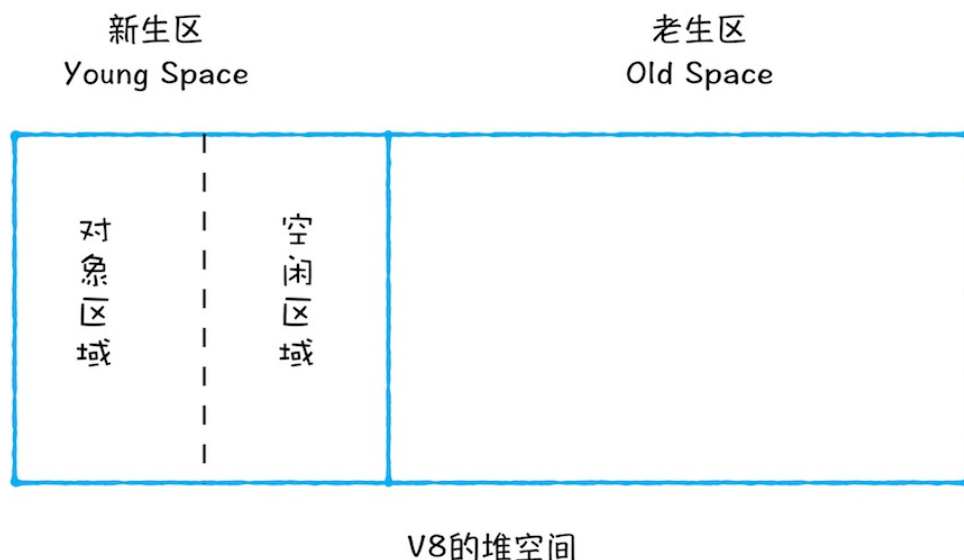
第三步是做内存整理。一般来说，频繁回收对象后，内存中就会存在大量不连续空间，我们把这些不连续的内存空间称为**内存碎片**。当内存中出现了大量的内存碎片之后，如果需要分配较大连续内存的时候，就有可能出现内存不足的情况。所以最后一步需要整理这些内存碎片，但这步其实是可选的，因为有的垃圾回收器不会产生内存碎片，比如接下来我们要介绍的副垃圾回收器。

那么接下来，我们就按照这个流程来分析新生代垃圾回收器（副垃圾回收器）和老生代垃圾回收器（主垃圾回收器）是如何处理垃圾回收的。

副垃圾回收器

副垃圾回收器主要负责新生区的垃圾回收。而通常情况下，大多数小的对象都会被分配到新生区，所以说这个区域虽然不大，但是垃圾回收还是比较频繁的。

新生代中用**Scavenge算法**来处理。所谓Scavenge算法，是把新生代空间对半划分为两个区域，一半是对象区域，一半是空闲区域，如下图所示：



新生区要划分为对象区域和空闲区域

新加入的对象都会存放到对象区域，当对象区域快被写满时，就需要执行一次垃圾清理操作。

在垃圾回收过程中，首先要对对象区域中的垃圾做标记；标记完成之后，就进入垃圾清理阶段，副垃圾回收器会把这些存活的对象复制到空闲区域中，同时它还会把这些对象有序地排列起来，所以这个复制过程，也就相当于完成了内存整理操作，复制后空闲区域就没有内存碎片了。

完成复制后，对象区域与空闲区域进行角色翻转，也就是原来的对象区域变成空闲区域，原来的空闲区域变成了对象区域。这样就完成了垃圾对象的回收操作，同时这种**角色翻转的操作还能让新生代中的这两块区域无限重复使用下去。**

由于新生代中采用的Scavenge算法，所以每次执行清理操作时，都需要将存活的对象从对象区域复制到空闲区域。但复制操作需要时间成本，如果新生区空间设置得太大了，那么每次清理的时间就会过久，所以**为了执行效率，一般新生区的空间会被设置得比较小。**

也正是因为新生区的空间不大，所以很容易被存活的对象装满整个区域。为了解决这个问题，JavaScript引擎采用了**对象晋升策略**，也就是经过两次垃圾回收依然还存活的对象，会被移动到老生区中。

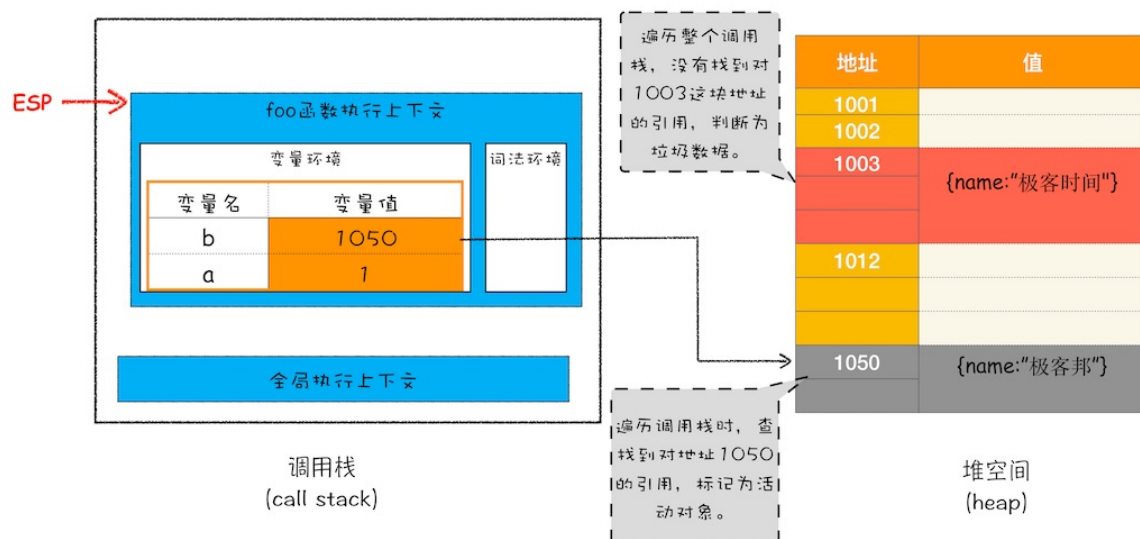
主垃圾回收器

主垃圾回收器主要负责老生区中的垃圾回收。除了新生区中晋升的对象，一些大的对象会直接被分配到老生区。因此老生区中的对象有两个特点，一个是对象占用空间大，另一个是对象存活时间长。

由于老生区的对象比较大，若要在老生区中使用Scavenge算法进行垃圾回收，复制这些大的对象将会花费比较多的时间，从而导致回收执行效率不高，同时还会浪费一半的空间。因而，主垃圾回收器是采用**标记-清除（Mark-Sweep）**的算法进行垃圾回收的。下面我们来看看该算法是如何工作的。

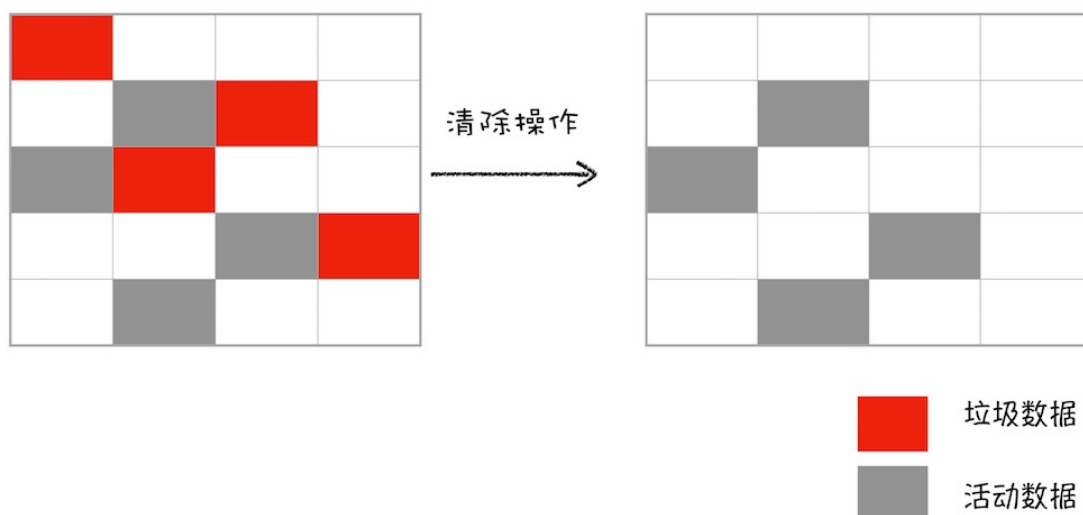
首先是标记过程阶段。标记阶段就是从一组根元素开始，递归遍历这组根元素，在这个遍历过程中，能到达的元素称为**活动对象**，没有到达的元素就可以判断为**垃圾数据**。

比如最开始的那段代码，当showName函数执行退出之后，这段代码的调用栈和堆空间如下图所示：



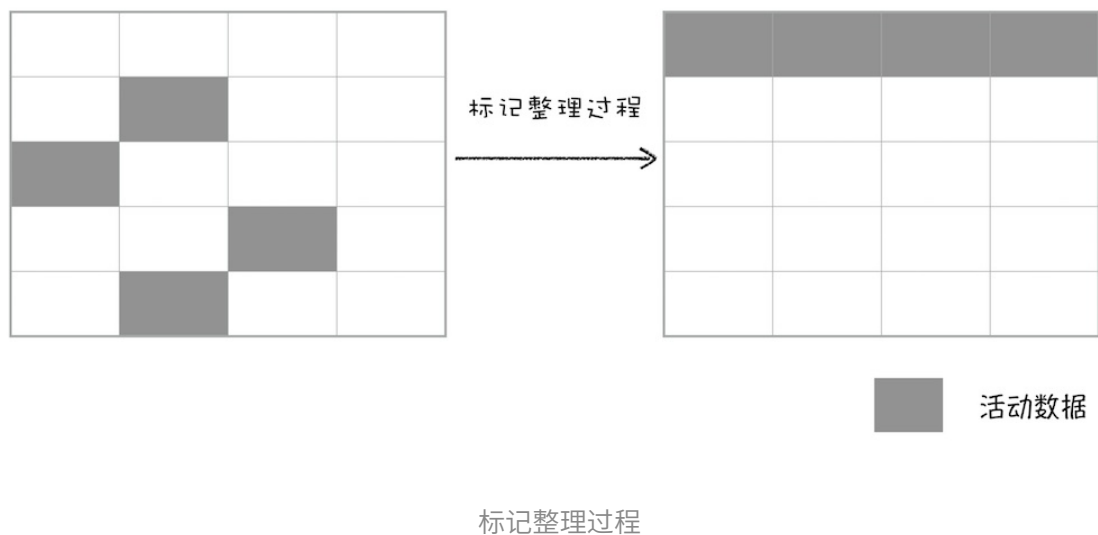
从上图你可以大致看到垃圾数据的标记过程，当showName函数执行结束之后，ESP向下移动，指向了foo函数的执行上下文，这时候如果遍历调用栈，是不会找到引用1003地址的变量，也就意味着1003这块数据为垃圾数据，被标记为红色。由于1050这块数据被变量b引用了，所以这块数据会被标记为活动对象。这就是大致的标记过程。

接下来就是垃圾的清除过程。它和副垃圾回收器的垃圾清除过程完全不同，你可以理解这个过程是清除掉红色标记数据的过程，可参考下图大致了解下其清除过程：



标记清除过程

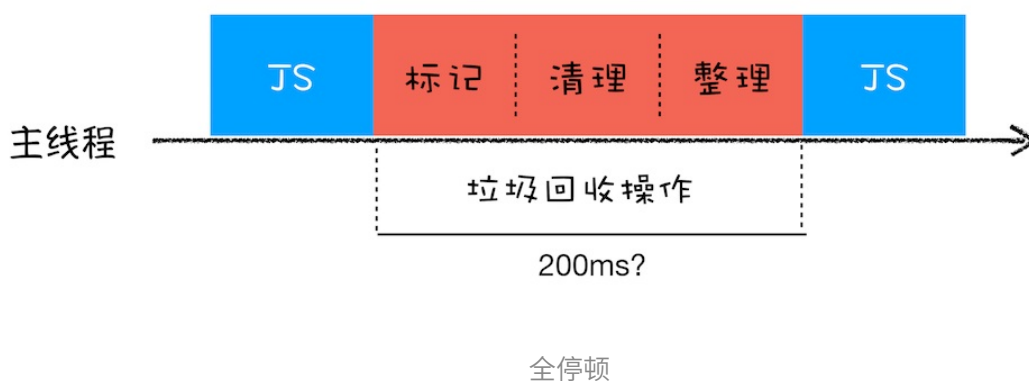
上面的标记过程和清除过程就是标记-清除算法，不过对一块内存多次执行标记-清除算法后，会产生大量不连续的内存碎片。而碎片过多会导致大对象无法分配到足够的连续内存，于是又产生了另外一种算法——**标记-整理 (Mark-Compact)**，这个标记过程仍然与标记-清除算法里的是一样的，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。你可以参考下图：



全停顿

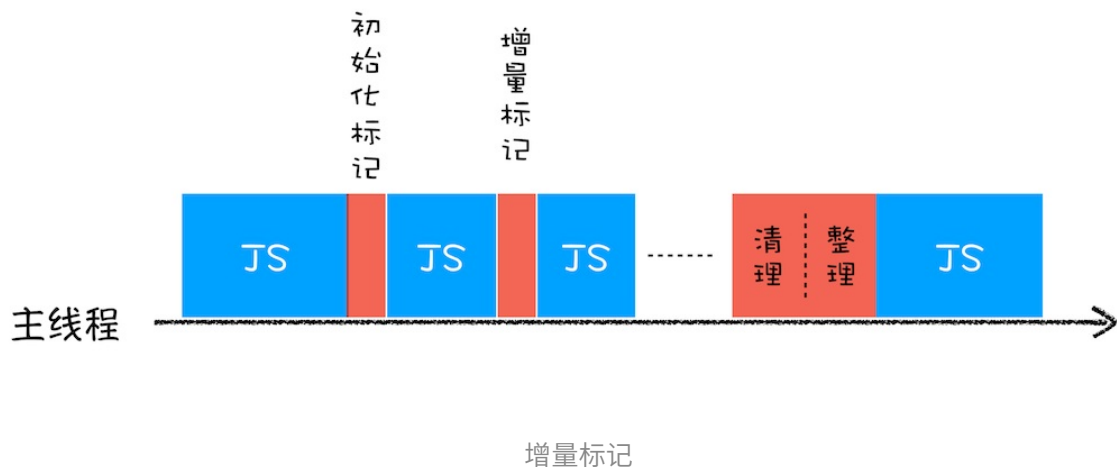
现在你知道了V8是使用副垃圾回收器和主垃圾回收器处理垃圾回收的，不过由于JavaScript是运行在主线程之上的，一旦执行垃圾回收算法，都需要将正在执行的JavaScript脚本暂停下来，待垃圾回收完毕后再恢复脚本执行。我们把这种行为叫做**全停顿 (Stop-The-World)**。

比如堆中的数据有1.5GB，V8实现一次完整的垃圾回收需要1秒以上的时间，这也是由于垃圾回收而引起JavaScript线程暂停执行的时间，若是这样的时间花销，那么应用的性能和响应能力都会直线下降。主垃圾回收器执行一次完整的垃圾回收流程如下图所示：



在V8新生代的垃圾回收中，因其空间较小，且存活对象较少，所以全停顿的影响不大，但老生代就不一样了。如果在执行垃圾回收的过程中，占用主线程时间过久，就像上面图片展示的那样，花费了200毫秒，在这200毫秒内，主线程是不能做其他事情的。比如页面正在执行一个JavaScript动画，因为垃圾回收器在工作，就会导致这个动画在这200毫秒内无法执行的，这将会造成页面的卡顿现象。

为了降低老生代的垃圾回收而造成的卡顿，V8将标记过程分为一个个的子标记过程，同时让垃圾回收标记和JavaScript应用逻辑交替进行，直到标记阶段完成，我们把这个算法称为**增量标记 (Incremental Marking) 算法**。如下图所示：



使用增量标记算法，可以把一个完整的垃圾回收任务拆分为很多小的任务，这些小的任务执行时间比较短，可以穿插在其他的JavaScript任务中间执行，这样当执行上述动画效果时，就不会让用户因为垃圾回收任务而感受到页面的卡顿了。

总结

好了，今天就讲到这里，下面我们就来总结下今天的主要内容。

首先我们介绍了不同语言的垃圾回收策略，然后又说明了栈中的数据是如何回收的，最后重点讲解了JavaScript中的垃圾回收器是如何工作的。

从上面的分析你也能看出来，无论是垃圾回收的策略，还是处理全停顿的策略，往往都没有一个完美的解决方案，你需要花一些时间来做权衡，而这需要牺牲当前某几方面的指标来换取其他几个指标的提升。

其实站在工程师的视角，我们经常需要在满足需求的前提下，权衡各个指标的得失，把系统设计得尽可能适应最核心的需求。

生活中处理事情的原则也与之类似，古人很早就说过“两害相权取其轻，两利相权取其重”，所以与其患得患失，不如冷静地分析哪些才是核心诉求，然后果断决策牺牲哪些以使得利益最大化。

思考时间

今天留给你的思考题是：你是如何判断JavaScript中内存泄漏的？可以结合一些你在工作中避免内存泄漏的方法。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

浏览器工作原理与实践

>>> 透过浏览器看懂前端本质

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。

精选留言：

• mfist 2019-09-03 07:14:46

1. 如何判断内存泄漏的？一般是感官上的长时间运行页面卡顿，猜可能会有内存泄漏。通过DynaTrace（IE）profiles等工具一段时间收集数据，观察对象的使用情况。然后判断是否存在内存泄漏。修改后验证
2. 工作中避免内存泄漏方法：确定不使用的临时变量置为null，当前es6普及场景下少使用闭包也是一种方法。

今日总结

垃圾回收策略一般分为手动回收和自动回收，java python JavaScript等高级语言为了减轻程序员负担和出错概率采用了自动回收策略。JavaScript的原始类型数据和引用数据是分别存储在栈和堆中的，由于栈和堆分配空间大小差异，垃圾回收方式也不一样。栈中分配空间通过ESP的向下移动销毁保存在栈中数据；堆中垃圾回收主要通过副垃圾回收器（新生代）和主垃圾回收器（老生代）负责的，副垃圾回收器采用scavenge算法将区域分为对象区域和空闲区域，通过两个区域的反转让新生代区域无限使用下去。主垃圾回收器采用Mark-Sweep（Mark-Compact Incremental Marking解决不同场景下问题的算法改进）算法进行空间回收的。无论是主副垃圾回收器的策略都是标记-清除-整理三个大的步骤。另外还有新生代的晋升策略（两次未清除的），大对象直接分配在老生代。