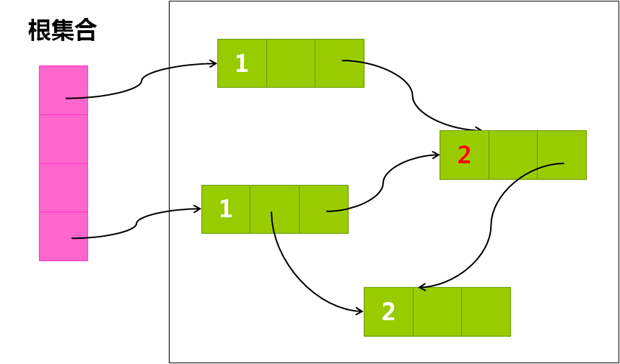
# [垃圾回收系列（2）：几种经典的垃圾回收算法](http://terrylee.me/blog/post/2010/09/26/garbage-collection-and-performance-part2.aspx)

发布者：[TerryLee](http://terrylee.me/blog/author/TerryLee.aspx) | 分类：[ANSI C/C++](http://terrylee.me/blog/category/ANSI-CC2b2b.aspx) | [计算机科学](http://terrylee.me/blog/category/e8aea1e7ae97e69cbae7a791e5ada6.aspx)

本文是关于垃圾回收讲座的第二篇，第一篇为《[垃圾回收系列（1）：没有GC，世界将会怎样](http://terrylee.me/blog/post/2010/09/25/garbage-collection-and-performance-part1.aspx)》。本文主要关注垃圾回收算法。垃圾回收机制，最早出现于世界上第二元老语言Lisp，Jean E. Sammet曾经说过，Lisp语言最长久的共享之一是一个非语言特征，即代表了系统自动处理内存的方法的术语极其技术——垃圾收集（GC，Garbage Collection）。接下来我们介绍几种经典的垃圾回收算法，这些算法尽管出现于60、70年代，但是现在的CLR、JVM等上面的垃圾回收器，仍然使用了它们。

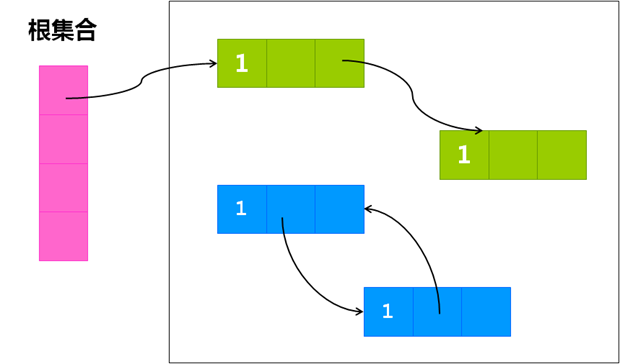
引用计数算法

引用计数（Reference Counting）算法是每个对象计算指向它的指针的数量，当有一个指针指向自己时计数值加1；当删除一个指向自己的指针时，计数值减1，如果计数值减为0，说明已经不存在指向该对象的指针了，所以它可以被安全的销毁了。可以很直观的用下面的图表示：

[](http://terrylee.me/blog/image.axd?picture=gc011.png)

引用计数算法的优点在于内存管理的开销分布于整个应用程序运行期间，非常的“平滑”，无需挂起应用程序的运行来做垃圾回收；而它的另外一个优势在于空间上的引用局部性比较好，当某个对象的引用计数值变为0时，系统无需访问位于堆中其他页面的单元，而后面我们将要看到的几种垃圾回收算法在回收前都回遍历所有的存活单元，这可能会引起换页（Paging）操作；最后引用计数算法提供了一种类似于栈分配的方式，废弃即回收，后面我们将要看到的几种垃圾回收算法在对象废弃后，都会存活一段时间，才会被回收。

引用计数算法有着诸多的优点，但它的缺点也是很明显的。首先能看到的一点是时间上的开销，每次在对象创建或者释放时，都要计算引用计数值，这会引起一些额外的开销；第二是空间上的开销，由于每个对象要保持自己被引用的数量，必须付出额外的空间来存放引用计数值；引用计数算法最大的缺点就在于它无法处理环形引用，如下图所示：

[](http://terrylee.me/blog/image.axd?picture=gc012.png)

此处蓝色的这两个对象既不可达也无法回收，因为彼此之间互相引用，它们各自的计数值都不为0，这种情况对引用计数算法来说是无能为力的，而其他的垃圾回收算法却能很好的处理环形引用。

引用计数算法最著名的运用，莫过于微软的COM技术，大名鼎鼎的IUnknown接口：

interface IUnknown

{

virtual HRESULT \_stdcall QueryInterface

(const IID& iid, void\* \* ppv) = 0;

virtual ULONG \_stdcall AddRef() = 0;

virtual ULONG \_stdcall Release() = 0;

}

其中的AddRef和Release就是用来让组件自己管理其生命周期，而客户程序只关心接口，而无须再去关心组件的生命周期，一个简单的使用示例如下：

int main()

{

IUnknown\* pi = CreateInstance();

IX\* pix = NULL;

HRESULT hr = pi->QueryInterface(IID\_IX, (void\*)&pix);

if(SUCCEEDED(hr))

{

pix->DoSomething();

pix->Release();

}

pi->Release();

}

上面的客户程序在CreateInstance中已经调用过AddRef，所以无需再次调用，而在使用完接口后调用Release，这样组件自己维护的计数值将会改变。下面代码给出一个简单的实现AddRef和Release示例：

ULONG \_stdcall AddRef()

{

return ++ m\_cRef;

}

ULONG \_stdcall Release()

{

if(--m\_cRef == 0)

{

delete this;

return 0;

}

return m\_cRef;

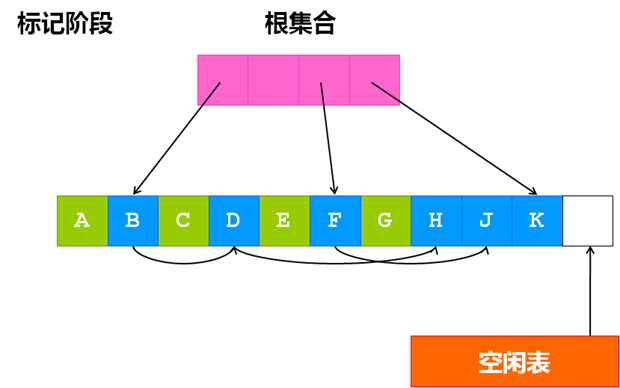
}

在编程语言Python中，使用也是引用计数算法，当对象的引用计数值为0时，将会调用\_\_del\_\_函数，至于为什么Python要选用引用计数算法，据我看过的一篇文章里面说，由于Python作为脚本语言，经常要与C/C++这些语言交互，而使用引用计数算法可以避免改变对象在内存中的位置，而Python为了解决环形引用问题，也引入gc模块，所以本质上Python的GC的方案是混合引用计数和跟踪（后面要讲的三个算法）两种垃圾回收机制。

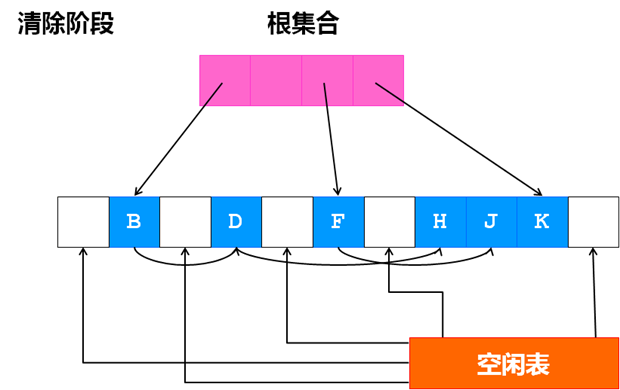
标记-清除算法

标记-清除（Mark-Sweep）算法依赖于对所有存活对象进行一次全局遍历来确定哪些对象可以回收，遍历的过程从根出发，找到所有可达对象，除此之外，其它不可达的对象就是垃圾对象，可被回收。整个过程分为两个阶段：标记阶段找到所有存活对象；清除阶段清除所有垃圾对象。

标记阶段：

[](http://terrylee.me/blog/image.axd?picture=gc013.png)

清除阶段：

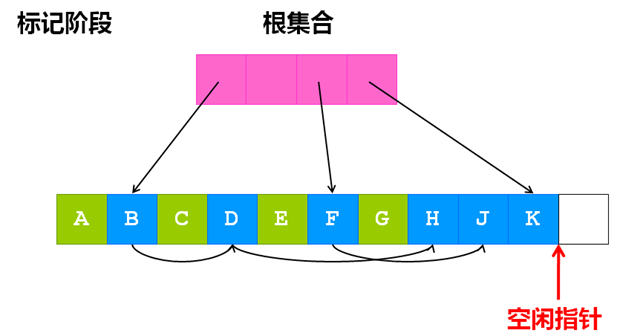
[](http://terrylee.me/blog/image.axd?picture=gc014.png)

相比较引用计数算法，标记-清除算法可以非常自然的处理环形引用问题，另外在创建对象和销毁对象时时少了操作引用计数值的开销。它的缺点在于标记-清除算法是一种“停止-启动”算法，在垃圾回收器运行过程中，应用程序必须暂时停止，所以对于标记-清除算法的研究如何减少它的停顿时间，而分代式垃圾收集器就是为了减少它的停顿时间，后面会说到。另外，标记-清除算法在标记阶段需要遍历所有的存活对象，会造成一定的开销，在清除阶段，清除垃圾对象后会造成大量的内存碎片。

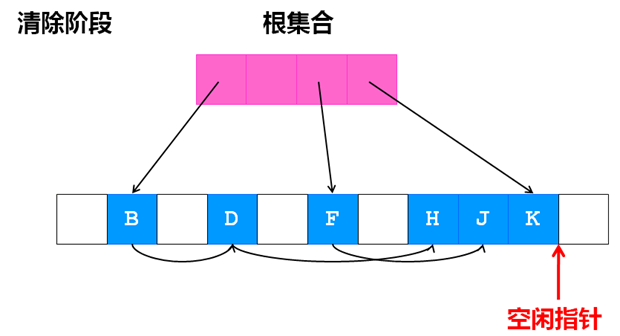
标记-缩并算法

标记-缩并算法是为了解决内存碎片问题而产生的一种算法。 它的整个过程可以描述为：标记所有的存活对象；通过重新调整存活对象位置来缩并对象图；更新指向被移动了位置的对象的指针。

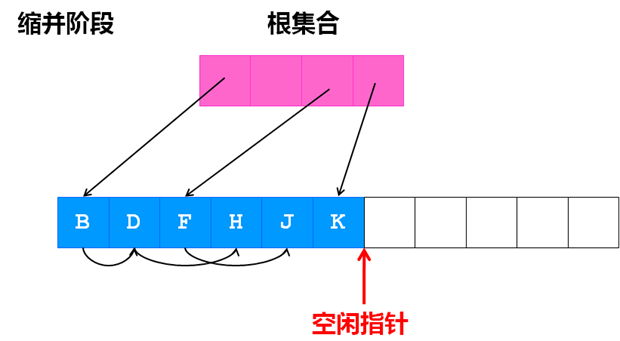
标记阶段：

[](http://terrylee.me/blog/image.axd?picture=gc015.png)

清除阶段：

[](http://terrylee.me/blog/image.axd?picture=gc016.png)

缩并阶段：

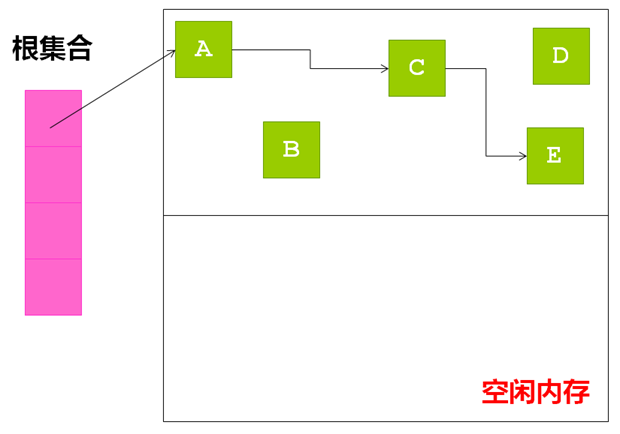
[](http://terrylee.me/blog/image.axd?picture=gc017.png)

标记-压缩算法最大的难点在于如何选择所使用的压缩算法，如果压缩算法选择不好，将会导致极大的程序性能问题，如导致Cache命中率低等。一般来说，根据压缩后对象的位置不同，压缩算法可以分为以下三种：

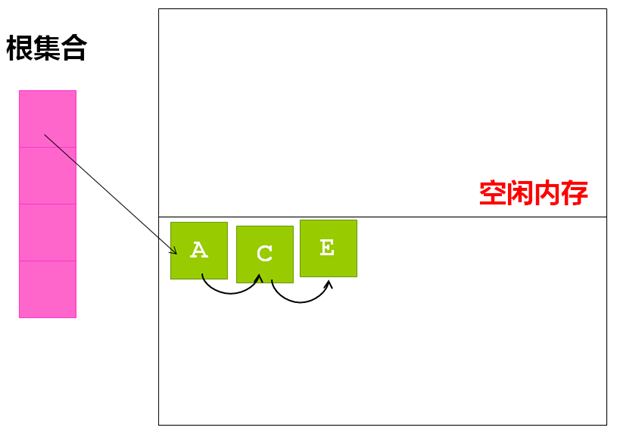
1. 任意：移动对象时不考虑它们原来的次序，也不考虑它们之间是否有互相引用的关系。   
2. 线性：尽可能的将原来的对象和它所指向的对象放在相邻位置上，这样可以达到更好的空间局部性。   
3. 滑动：将对象“滑动”到堆的一端，把存活对象之间的自由单元“挤出去”，从而维持了分配时的原始次序。

节点拷贝算法

节点拷贝算法是把整个堆分成两个半区（From，To）， GC的过程其实就是把存活对象从一个半区From拷贝到另外一个半区To的过程，而在下一次回收时，两个半区再互换角色。在移动结束后，再更新对象的指针引用，GC开始前的情形：

[](http://terrylee.me/blog/image.axd?picture=gc018.png)

GC结束后的情形：

[](http://terrylee.me/blog/image.axd?picture=gc019.png)

节点拷贝算法由于在拷贝过程中，就可以进行内存整理，所以不会再有内存碎片的问题，同时也不需要再专门做一次内存压缩。，而它最大的缺点在于需要双倍的空间。

总结

本文总共介绍了四种经典的垃圾回收算法，其中后三种经常称之为跟踪垃圾回收，因为引用计数算法能够平滑的进行垃圾回收，而不会出现“停止”现象，经常出现于一些实时系统中，但它无法解决环形问题；而基于跟踪的垃圾回收机制，在每一次垃圾回收过程中，要遍历或者复制所有的存活对象，这是一个非常耗时的工作，一种好的解决方案就是对堆上的对象进行分区，对不同区域的对象使用不同的垃圾回收算法，分代式垃圾回收器正是其中一种，CLR和JVM中都采用了分代式垃圾回收机制，但它们在处理上又有些不同，后面的文章再详细介绍这两种垃圾回收器的区别。