垃圾回收器是什么

一眼就应该从名称看出垃圾回收机制的含义-查找垃圾，然后丢弃。事实正好相反。垃圾回收器追踪所有正在使用的对象，将无用对象标记为垃圾。请留意，我们开始研究JVM的“Garbage Collection”的实现细节。

避免仓促进入细节，我们因该从入门开始入手。理解垃圾收集器的一般规律，核心、概念，处理方法。

郑重声明：本文关注的是Oracle的Hotspot和OpenJDK。其他运行环境或者其他的JVM，诸如jRockit，IBM J9、以及本手册提及的一些产品会有一些差异化。

内存管理指南

在我们开始讲解垃圾回收器的工作方式之前，你需要手动为你的数据分配一块可用空间。如果你忘记分配，你将不能重复使用这块空间。这块空间将被声明但是不能被使用。例如，内存泄漏。

下面是一个用C实现的简单内存管理使用示例：

```
int send_request() {
    size_t n = read_size();
    int *elements = malloc(n * sizeof(int));

    if(read_elements(n, elements) < n) {
        // elements not freed!
        return -1;
    }

    // …

    free(elements)
    return 0;
}
```

正如您看到的，它非常容易忘记去释放内存。内存泄漏是一个比现在更加广泛的共性问题。你只能把内嵌在你的代码来克服这个问题。因此，最好的方法是自动回收不用的内存，以完全避免人为错误。例如Garbage Collection(GC)自动化操作。

自动化指针

内存回收自动化的最好方式之一是使用钩子函数。例如，在C++中我们使用vector做同样的事情，当数据在作用域不再使用时被自动调用的一种钩子函数：

```
int send_request() {
    size_t n = read_size();
    vector<int> elements = vector<int>(n);

    if(read_elements(elements.size(), &elements[0]) < n) {
        return -1;
    }

    return 0;
}
```

但是多数情况下更加复杂，特别是对象被多个线程跨线程共享，仅仅使用钩子函数不合适，由此产生最简单的垃圾回收机制：引用计数。对于每一个对象，只要简单知道它被引用了多少次，当计数器归零的时候，它就被安全回收。下面是C++共享指针的著名例子：

```
int send_request() {
    size_t n = read_size();
    auto elements = make_shared<vector<int>>();

    // read elements

    store_in_cache(elements);

    // process elements further

    return 0;
}
```

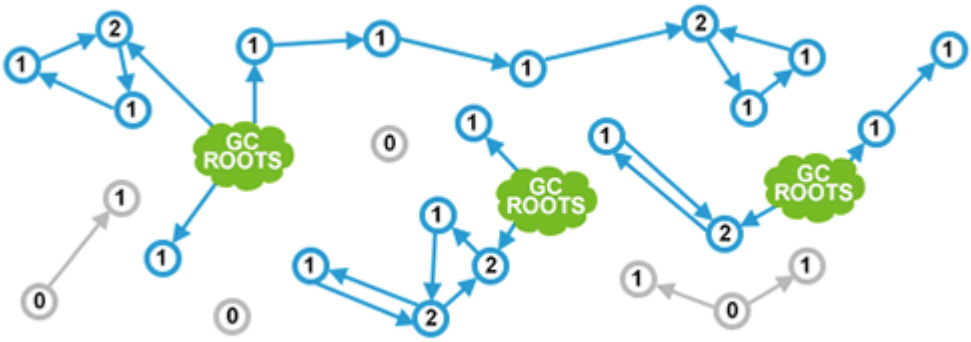
现在，为了避免元素在下一时刻被其他函数调用，我们也许应该将它们缓存起来。在这种情况下，我们不能选择当它出了作用域就销毁vector。因此，我们使用shared_ptr。它记录引用次数。在它的作用域内数量增加，在作用域之外次数减少。一旦引用次数归零，shared_ptr 自动删除其下的vector。

内存自动管理

从上面的C++代码可以看出，我们依然明确提出关注内存管理。但是如果使用这种方式管理所有的内存有会怎样呢？它变得非常方便，自此开发人员不再考虑自己手动清除。运行环境会自动识别不再使用的内存并且释放它。换句话说，它自动回收垃圾。世界上第一款垃圾收集于1959年使用Lisp语言实现，自此相关技术开始向前发展。

引用计数

上文提到我们使用C++共享指针的方式管理所有对象。许多语言，例如 Perl, Python ,PHP都采用这种方式。下图很好说明了这种方法：



绿色云状图标指向的对象说明他们仍然被程序占用。技术上，他们类似于当前执行方法的本地变量或者静态变量或者其他。不同语言之间的差别很大，本文不做关注。

蓝色图标表示内存中存活的对象，里面的数字代表它们被引用的次数。最后，灰色图标表示没有被任何存活对象引用（就是直接被绿色云状代表对象引用的对象）。灰色对象就这样被当成垃圾，被垃圾收集器清除。

这看起来很不错，不是吗？好吧，的确，但是这种方式存在一个巨大的缺点。这些没有在作用域中对象发生环状引用，由于环状引用导致他们的引用计数不归零。下面是示意图：

看到了吗？实际上，红色对象没有被应用程序使用而成为垃圾。由于引用计数的局限性，他们造成了内存泄漏。

有几种方法客服这种情况，例如使用特殊的“”weak“”引用，或者为了环状应用特殊的算法。Perl, Python ， PHP使用哪种方式处理，不在本文的讨论范围之内。相反，我开始讨论更多JVM的实现细节。

标记清除

首先，对于可达对象，JVM有着明确的定义。与上面章节中绿色云状图模糊定义不同的是，我们有一个叫做Garbage Collection Roots（GC Root）的明确定义：

- 局部变量
- 活跃线程
- 静态域
- JNI引用

JVM使用这种方式追踪所有可达（存活）的对象，通过被称作标记清除算法重复扫描确定不可达对象。该算法包含两步：

- 标记，扫描所有的可达对象，在本地内存这些对象的副本。
- 清除，确保所有被不可达对象占据的内存空间可以在下一次重新分配。

JVM中多种不同的算法，例如Parallel Scavenge，Parallel Mark+Copy，CMS，他们的实施阶段不尽相同，但是执行步骤和上面描述的两步类似。

这些算法很重要的一点是保证环形引用不再泄漏：

不太好的一点是，回收操作发生的时候，所有的应用线程被停止。正如它们一直在改变引用导致无法准确计算它们的应用的那样。像这种应用线程被暂时停止，以便JVM活动的晴空称之为“”stop-the-world“”。它们可能因为多种原因发生，但是这种垃圾回收器是最主流的一种。