参考：<https://www.cnblogs.com/little-YTMM/p/5613642.html>

## GC定义

<https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)> 旧版

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **garbage collection** (**GC**) is a form of automatic [memory management](https://en.wikipedia.org/wiki/Memory_management). The *garbage collector*, or just *collector*, attempts to reclaim [*garbage*](https://en.wikipedia.org/wiki/Garbage_(computer_science)), or memory occupied by [objects](https://en.wikipedia.org/wiki/Object_(computer_science)) that are no longer in use by the [program](https://en.wikipedia.org/wiki/Application_software).

即一种自动内存管理机制。垃圾回收器试图回收**垃圾**（在程序中不再被使用的对象，数据，或其他系统资源占用的内存）。

<https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)> 新版

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **garbage collection** (**GC**) is a form of automatic [memory management](https://en.wikipedia.org/wiki/Memory_management). The *garbage collector* attempts to reclaim memory which was allocated by the program, but is no longer referenced; such memory is called [*garbage*](https://en.wikipedia.org/wiki/Garbage_(computer_science)). 在计算机科学中，垃圾收集 (GC) 是一种自动内存管理形式。垃圾收集器试图回收由程序分配但不再被引用的内存；这样的内存称为**垃圾**。

## 引入GC原因

1. 安全性
2. 减少内存泄露
3. 减少程序员工作量

## 如何触发GC

<https://blog.csdn.net/wangxuelei036/article/details/108180150>

GC触发的条件有两种：

（1）程序调用System.gc时可以触发。

（2）系统自身来决定GC触发的时机。

## （What）哪些内存要回收

### （1）JVM运行时数据区

内存运行时JVM会有一个运行时数据区来管理内存。主要5大部分：程序计数器（Program Counter Register），虚拟机栈（VM Stack），本地方法栈（Native Method Stack），方法区（Method Area），堆（Heap）。

而其中程序计数器，虚拟机栈，本地方法栈是每个线程私有的内存空间，随线程而生，随线程而亡。例如栈中每一个栈帧中分配多少内存基本上在类结构确定后就已知了，因此这三个区域的内存分配和回收都是确定的，无需考虑内存回收的问题。

但方法区和堆就不同了，一个接口的多个实现类需要的内存可能不同，我们只能在程序运行期间才会知道会创建哪些对象，这部分内存的分配和回收都是动态的，GC主要关注的是这部分内存。

### （2）总结

GC主要进行回收的内存是JVM中的方法区和堆，涉及到多线程（指堆），多个对该对象不同类型的引用（指方法区），才会涉及GC的回收。

## （When）回收时机

### 堆

### 如何判断一个对象已经死去

在面试中经常会碰到这样一个问题：**如何判断一个对象已经死去？**

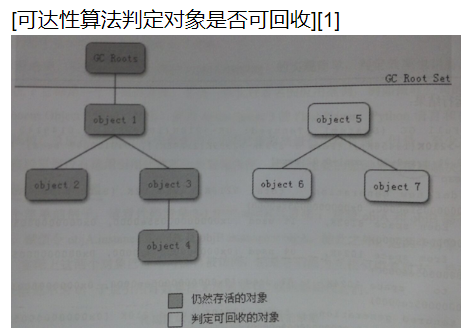
容易想到一个答案：对一个对象添加引用计数器。每当有地方引用它时，计数器值加1；当引用失效时，计数器值减1。而当计数器的值为0时该对象就不会再被使用，判断为已死。但该方法是错误的！事实上，用引用计数法确实在大多数情况下是一个不错的解决方案，但它无法解决对象之间的循环引用问题。比如对象A中有一个字段指向对象B，而对象B中也有一个字段指向对象A，而事实上他们两都不再使用，但计数器的值永远不可能为0，也就不会被回收，然后就发生了内存泄露。

所以，正确的做法应该怎样？

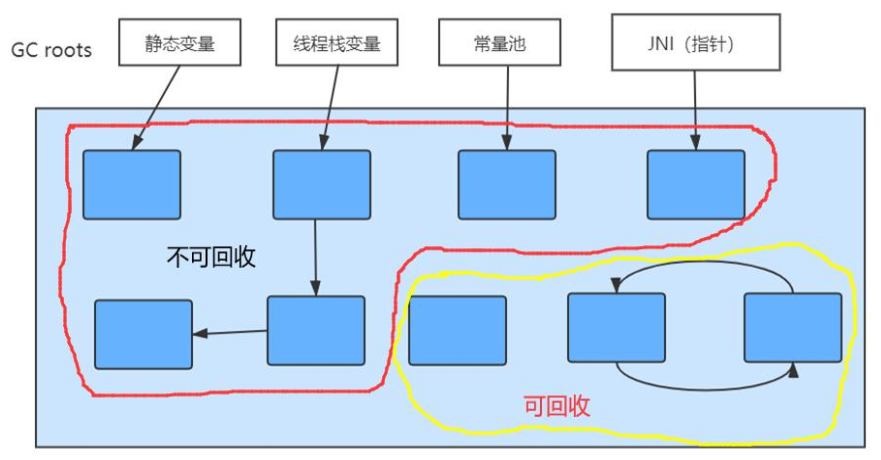
在Java，C#等语言中，比较主流的判定一个对象已死的方法：可达性分析（Reachability Analysis）。

所有生成的对象都是一个称为“GC Roots”的根的子树。从GC Roots开始向下搜索，搜索所经过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链可达时，就称这个对象是不可达的（不可引用的），也就是可以被GC回收。

<https://www.cnblogs.com/little-YTMM/p/5613642.html>



<https://ke.qq.com/webcourse/index.html?r=1678119046269#cid=347420&term_id=103755197&taid=12286522749701404&type=3072&source=PC_COURSE_DETAIL&vid=3701925922528318833>



#### 可作为gc roots的对象类型

<https://time.geekbang.org/column/article/40961> 评论1中的子评论

<https://www.lubojian.cn/post/112.html>

<https://blog.csdn.net/zhangvalue/article/details/101034370>

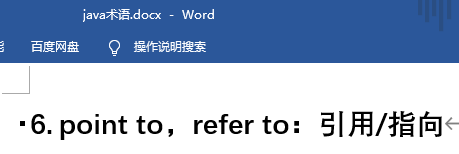
<https://ke.qq.com/webcourse/index.html?r=1678119046269#cid=347420&term_id=103755197&taid=12286522749701404&type=3072&source=PC_COURSE_DETAIL&vid=3701925922528318833> 57:49

可作为gc roots的对象主要有以下几种：

1.Java虚拟机栈中引用的对象  
2.方法区类的静态属性引用的对象  
3.方法区常量（池）引用的对象  
4.本地方法栈中JNI引用的对象

### （2） 四种引用

同参考：享学vip-2021.8.12-对象分配及垃圾回收机制下的4:22

注：这里所说的**引用**指引用类型变量的值，即必须要涉及引用类型变量，具体见。

#### 【1】 4种引用+引用队列

<https://www.cnblogs.com/little-YTMM/p/5613642.html>

<https://blog.csdn.net/coding_or_coded/article/details/6603549>

<https://blog.csdn.net/u014086926/article/details/52106589>

无论是引用计数器还是可达性分析，判定对象是否存活都与引用有关。

Java对象的引用有4种，按级别由高到低依次为：强引用、软引用、弱引用和虚引用。

根据引用类型不同，GC回收时也会有不同操作。

##### 强引用（Strong Reference）

###### 是什么

1. **强引用**是指创建一个对象并把这个对象赋给一个引用变量。

如：

Object object = new Object();

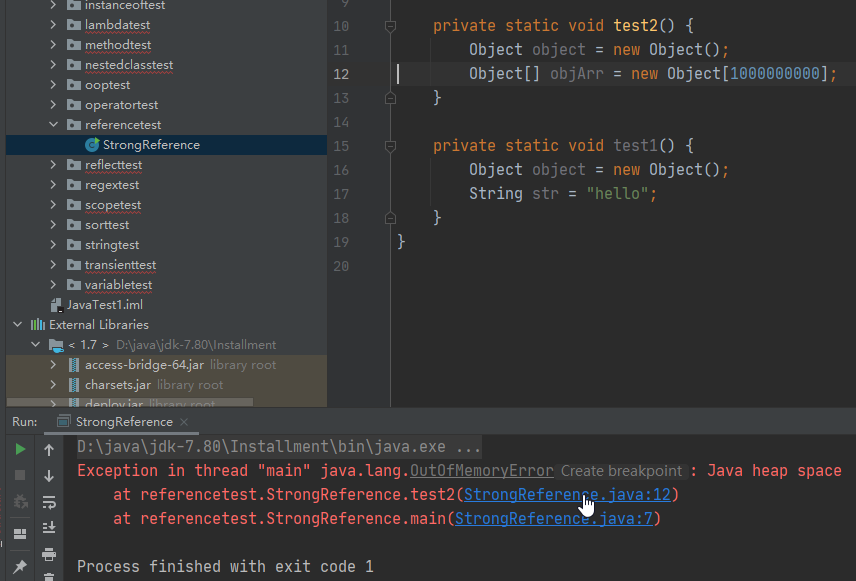
String str = "hello";

1. 只要强引用还存在，GC永远不会回收掉被引用的对象。

当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会回收具有强引用的对象来解决内存不足的问题。

如下：

|  |
| --- |
| Object object = new Object(); Object[] objArr = new Object[1000000000]; |

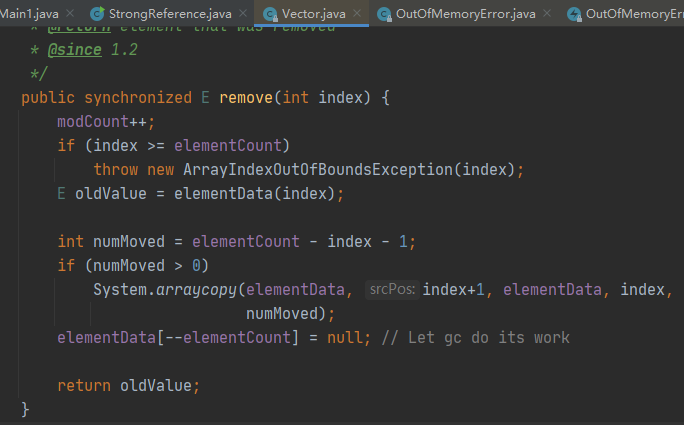


当执行到Object[] objArr = new Object[1000000000];这句，如果内存不足，JVM会抛出OOM错误也不会回收object指向的对象。

不过要注意的是，当test2()运行完之后，object和objArr都已经不存在了，所以它们指向的对象都会被JVM回收。

1. 如果想中断强引用和某个对象之间的关联，可以显示地将引用赋值为null，这样一来的话，JVM在合适的时间就会回收该对象。

比如Vector类的remove方法中就是通过将引用赋值为null来实现移除工作的：



###### 使用场景

<https://blog.51cto.com/u_12358040/5407761>

该变量指向的对象不希望被垃圾回收器回收。

##### 软引用（Soft Reference）

###### 是什么

1. 如果一个对象只具有**软引用**，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了【即系统将会发生内存溢出】，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。
2. 用SoftReference类来表示软引用。SoftReference的特点是它的一个实例保存对一个Java对象的软引用， 该软引用的存在不妨碍垃圾收集线程对该Java对象的回收。

也就是说，一旦SoftReference保存了对一个Java对象的软引用后，在垃圾收集线程对这个Java对象回收前，SoftReference类所提供的get()方法返回Java对象的强引用。另外，一旦垃圾收集线程回收该Java对象之后，get()方法将返回null。

举个例子：

|  |
| --- |
| public class SoftReferenceTest {   public static void main(String[] args) {  MyObject aRef = new MyObject();  SoftReference aSoftRef = new SoftReference(aRef);  } }  class MyObject {  } |

此时，对于这个MyObject对象，有两个引用路径，一个是来自SoftReference对象的软引用，一个来自变量aRef的强引用，所以这个MyObject对象是强可及对象。

随即，我们可以结束aRef对这个MyObject实例的强引用：

|  |
| --- |
| aRef = null; |

此后，这个MyObject对象成为了软引用对象。如果垃圾收集线程进行内存垃圾收集，并不会因为有一个SoftReference对该对象的引用而始终保留该对象。

Java虚拟机的垃圾收集线程对软可及对象和其他一般Java对象进行了区别对待：软可及对象的清理是由垃圾收集线程根据其特定算法按照内存需求决定的。也就是说，垃圾收集线程会在虚拟机抛出OutOfMemoryError之前回收软可及对象，而且虚拟机会尽可能优先回收长时间闲置不用的软可及对象，对那些刚刚构建的或刚刚使用过的“新”软可及对象会被虚拟机尽可能保留。

在回收这些对象之前，我们可以通过：

|  |
| --- |
| MyObject anotherRef = (MyObject) aSoftRef.get(); |

重新获得对该实例的强引用。而回收之后，调用get()方法就只能得到null

了。

【3】 软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用

所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

###### 使用场景

<https://blog.csdn.net/u014086926/article/details/52106589>

软引用可用来实现内存敏感的高速缓存，比如网页缓存、图片缓存等。

使用软引用能防止内存泄露，增强程序的健壮性。

<https://blog.csdn.net/u014086926/article/details/52106589>

利用软引用和弱引用解决OOM问题。

<https://m.imooc.com/wiki/jvm-fourgets>

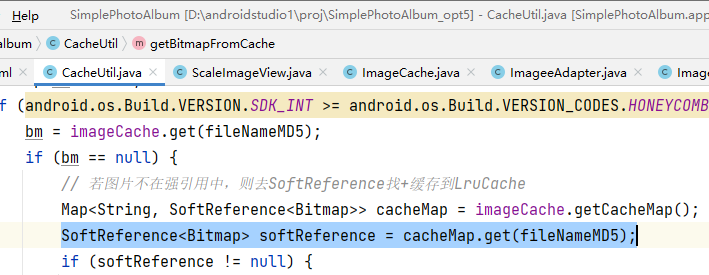
**软引用使用场景**：Android应用图片  
软引用主要应用于内存敏感的高速缓存，在 Android 系统中经常使用到。一般情况下，Android 应用会用到大量的默认图片，这些图片很多地方会用到。如果每次都去读取图片，由于读取文件需要硬件操作，速度较慢，会导致性能较低。所以我们考虑将图片缓存起来，需要的时候直接从内存中读取。

但是，由于图片占用内存空间比较大，缓存很多图片需要很多的内存，就可能比较容易发生 OutOfMemory 异常。这时，我们可以考虑使用软引用技术来避免这个问题发生。

SoftReference 可以解决 OOM 的问题，每一个对象通过软引用进行实例化，这个对象就以cache的形式保存起来，当再次调用这个对象时，那么直接通过软引用中的 get() 方法，就可以得到对象中的资源数据，这样就没必要再次进行读取了，直接从 cache 中就可以读取得到，当内存将要发生 OOM 的时候，GC 会迅速把所有的软引用清除，防止 OOM 发生。

###### 实例

Oppo应届生培养期的相册优化任务：



##### 弱引用（Weak Reference）

###### 是什么

1. 只具有**弱引用**的对象只能生存到下次GC之前。当GC工作时，无论内存是否足够都会将其回收。（即只要进行GC，就会对他们进行回收）
2. 弱引用和软引用的关联
3. 它们都是用来描述非必需对象的，但是仅被软引用关联的对象只有在内存不足时才会被回收，而仅被弱引用关联的对象在JVM进行垃圾回收时总会被回收。即只具有弱引用的对象拥有更短暂的生命周期。
4. 在SoftReference/WeakReference类都有两个构造器。
5. 在SoftReference/WeakReference类中get方法用来获取与软/弱引用关联的对象的强引用，如果该对象被回收了，则返回null。
6. 在使用软引用和弱引用的时候，我们可以显式地通过System.gc()来通知JVM进行垃圾回收，但是要注意的是，虽然发出了通知，JVM不一定会立刻执行，也就是说这句是无法确保此时JVM一定会进行垃圾回收的。
7. 用WeakReference类来表示弱引用。

示例：

|  |
| --- |
| public class WeakReferenceTest {   public static void main(String[] args) {  WeakReference<People> reference = new WeakReference<People>(new People("zhouqian", 20));  System.*out*.println(reference.get());  System.*gc*();//通知JVM回收资源  System.*out*.println(reference.get());  } }  class People {  public String name;  public int age;   public People(String name, int age) {  this.name = name;  this.age = age;  }   @Override  public String toString() {  return "[name:" + name + ",age:" + age + "]";  } } |

输出：



第二个输出结果是null，这说明只要JVM进行垃圾回收，被弱引用关联的对象必定会被回收掉。

不过要注意的是，这里所说的被弱引用关联的对象是指只有弱引用与之关联，如果存在强引用同时与之关联，则进行垃圾回收时也不会回收该对象（软引用也是如此）。

比如：将代码做一点小更改：

|  |
| --- |
| public class WeakReferenceTest {   public static void main(String[] args) {  People people=new People("zhouqian",20);  WeakReference<People>reference=new WeakReference<People>(people); //关联强引用  System.*out*.println(reference.get());  System.*gc*();  System.*out*.println(reference.get());  } }  class People {  public String name;  public int age;   public People(String name, int age) {  this.name = name;  this.age = age;  }   @Override  public String toString() {  return "[name:" + name + ",age:" + age + "]";  } } |

输出变成：

1. 弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

###### 使用场景

参1：<https://blog.51cto.com/u_12358040/5407761>

防止内存泄漏，保证内存被JVM回收。

参2：<https://www.cnblogs.com/minikobe/p/11929737.html>

弱引用的使用：

1、WeakHashmap的key源码分析：<https://www.jianshu.com/p/8660087c09e3>

2、ThreadLocal.ThreadLocalMap的key源码分析：<https://blog.csdn.net/hohoo1990/article/details/106356145>

3、jdk动态代理中缓存代理类的WeakCache：

<https://www.cnblogs.com/liuyun1995/p/8144676.html>

参3：<https://blog.csdn.net/u014086926/article/details/52106589>

**使用弱引用构建非敏感数据的缓存。**

利用软引用和弱引用解决OOM问题。

参4：<https://blog.csdn.net/hohoo1990/article/details/106356145>

个人理解，是在较为复杂的数据结构中，为了避免内存泄露而使用的一种引用方式。

参5：享学vip-2021.8.12-对象分配及垃圾回收机制下的13:27

弱引用一般使用场景比软引用要广泛点，因为造成oom【即软引用的触发时机】的几率并不大，但gc能将一些该回收的东西都回收掉。

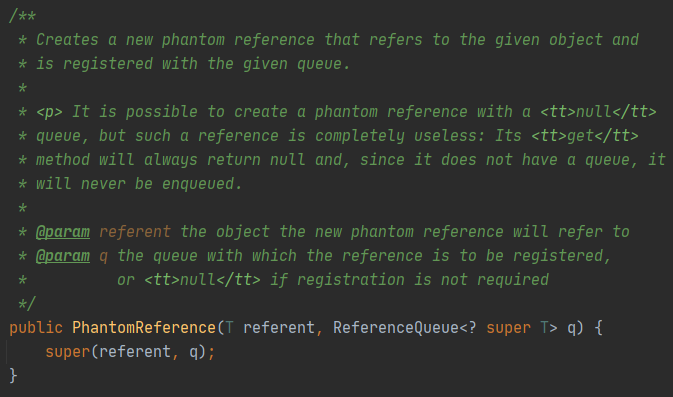
参6：Glide

Glide的活动缓存用的就是弱引用。

##### 虚引用（Phantom Reference）

###### 是什么

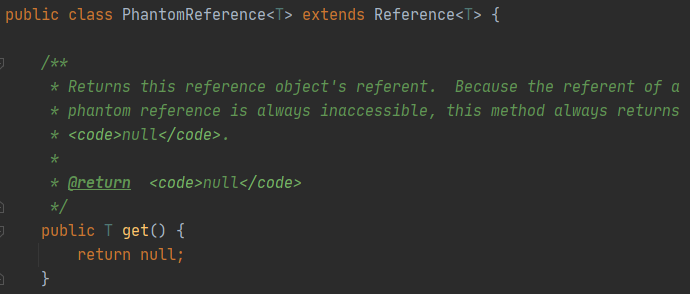
1. “虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，**虚引用**并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。
2. 用PhantomReference类表示虚引用。
3. 要注意的是，虚引用必须和引用队列关联使用，因其只有1个构造器



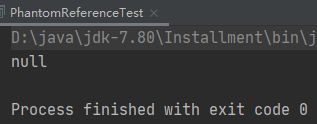
|  |
| --- |
| ReferenceQueue queue = **new** ReferenceQueue (); PhantomReference pr = **new** PhantomReference (object, queue); |

当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

1. PhantomReference的get方法永远返回null：



|  |
| --- |
| ReferenceQueue<String> queue = new ReferenceQueue<String>(); PhantomReference<String> pr = new PhantomReference<String>(new String("hello"), queue); System.*out*.println(pr.get()); |

输出：

<https://www.cnblogs.com/minikobe/p/11929737.html>

也就是说将永远无法通过虚引用来获取对象。

###### 使用场景

参1：<https://blog.csdn.net/hohoo1990/article/details/106356145>

使用虚引用的目的就是为了得知对象被GC的时机，所以可以利用虚引用来进行对象销毁前的一些操作，比如说资源释放等。这个虚引用对于对象而言完全是无感知的，有没有完全一样，但是对于虚引用的使用者而言，就像是待观察的对象的把脉线，可以通过它来观察对象是否已经被回收，从而进行相应的处理。

事实上，虚引用有一个很重要的用途就是用来做堆外内存的释放，DirectByteBuffer就是通过虚引用来实现堆外内存的释放的。

参2：<https://blog.51cto.com/u_12358040/5407761>

跟踪对象被GC回收的活动。

通过判断引用队列中是否有虚引用来了解被引用对象是否被回收。

参3：<https://juejin.cn/post/7075224868655988743>

主要是**管理堆外内存**的，主要是给写jvm的使用。

主要检测引用队列配合使用，虚引用API无法get到值，主要是通知对象已经被回收，去清理堆外的内存。

参4：<https://www.jianshu.com/p/32e3ca1ef157>

##### 引用队列（ReferenceQueue）

###### 概述：

参1：<https://www.cnblogs.com/minikobe/p/11929737.html>

引用队列可以与软引用、弱引用以及虚引用一起配合使用。

当垃圾回收器准备回收一个对象时，如果发现它还有以上3种引用，那么就会在回收对象之前，把这个引用加入到与之关联的引用队列中去。程序可以通过判断引用队列中是否已经加入了引用，来判断被引用的对象是否将要被垃圾回收，这样就可以在对象被回收之前采取一些必要的措施。

与软引用、弱引用不同，虚引用必须和引用队列一起使用。

参2：<https://juejin.cn/post/6844903728500178952#heading-6>

前面说到在使用软引用、虚引用、幻象引用的时候可以指定一个引用队列，在引用所引用的对象被回收后引用本身就会进入引用队列。

###### 示例：

<https://blog.csdn.net/coding_or_coded/article/details/6603549>



##### 总结：

###### 功能总结：

<https://xie.infoq.cn/article/83777ef09f349643cee6b5cd7>

* 强引用：发生 gc 的时候不会被回收。
* 软引用：有用但不是必须的对象，在发生内存溢出之前会被回收。
* 弱引用：有用但不是必须的对象，在下一次 GC 时会被回收。
* 虚引用（幽灵引用/幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。

<https://juejin.cn/post/7075224868655988743>

| **引用类型** | **被垃圾回收时间** | **用途** | **生存时间** |
| --- | --- | --- | --- |
| 强引用 | 从来不会 | 对象的一般状态 | JVM停止运行时终止 |
| 软引用 | 当内存不足时 | 对象缓存 | 内存不足时终止 |
| 弱引用 | 正常垃圾回收时 | 对象缓存 | 垃圾回收后终止 |
| 虚引用 | 正常垃圾回收时 | 跟踪对象的垃圾回收 | 垃圾回收后终止 |

###### 使用频度总结：

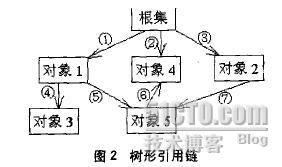
参考：享学vip-2021.8.12-对象分配及垃圾回收机制下的17:12

使用频度由高到低：强>弱（在gc时必回收）>软，虚

#### 【2】 4种引用与对象可及性判断

<https://blog.csdn.net/u014086926/article/details/52106589>

在很多时候，一个对象并不是从根集直接引用的，而是一个对象被其他对象引用，甚至同时被几个对象所引用，从而构成一个以根集为顶的树形结构。如图2所示



在这个树形的引用链中，箭头的方向代表了引用的方向，所指向的对象是被引用对象。由图可以看出，从根集到一个对象可以由很多条路径。比如到达对象5的路径就有①-⑤，③-⑦两条路径。由此带来了一个问题，那就是某个对象的可及性如何判断：

◆ 单条引用路径可及性判断：在这条路径中，最弱的一个引用决定对象的可及性。

◆ 多条引用路径可及性判断：几条路径中，最强的一条的引用决定对象的可及性。

比如，我们假设图2中引用①和③为强引用，⑤为软引用，⑦为弱引用，对于对象5按照这两个判断原则，路径①-⑤取最弱的引用⑤，因此该路径对对象5的引用为软引用。同样，③-⑦为弱引用。在这两条路径之间取最强的引用，于是对象5是一个软可及对象。

### 方法区

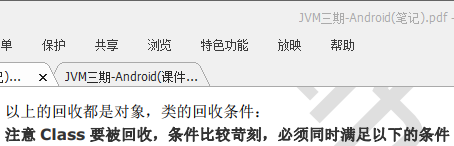
方法区中需要回收的是一些废弃的常量和无用的类。

### 废弃常量回收

这里看引用计数就可以了。没有对象引用该常量就可以放心的回收。

### 无用的类回收

参考：

1. <https://www.cnblogs.com/little-YTMM/p/5613642.html>
2. 
3. 

什么是无用的类：

1. 该类所有的实例都已经被回收。也就是Java堆中不存在该类的任何实例；
2. 加载该类的ClassLoader已经被回收；
3. 该类对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

### 总结：

对于**堆中的对象**，主要用可达性分析判断一个对象是否还存在引用，如果该对象没有任何引用就应该被回收。而根据我们实际对引用的不同需求，又分为4种引用，每种引用的回收机制也是不同的。

对于**方法区中的常量和类**，当一个常量没有任何对象引用它，它就可以被回收了。而对于类，如果可以判定它为无用类，就可以被回收了。

## （How）如何回收？

### 6.1 回收算法

参考：

1. 主：<https://www.cnblogs.com/little-YTMM/p/5613642.html>
2. 辅：

享学vip 3期-对象的分配及垃圾回收机制下的51:52

<https://www.lubojian.cn/post/112.html>

<https://xie.infoq.cn/article/83777ef09f349643cee6b5cd7>

#### （1） 标记-清除（Mark-Sweep）算法

##### 【1】 是什么

###### 参1：

<https://www.cnblogs.com/little-YTMM/p/5613642.html>

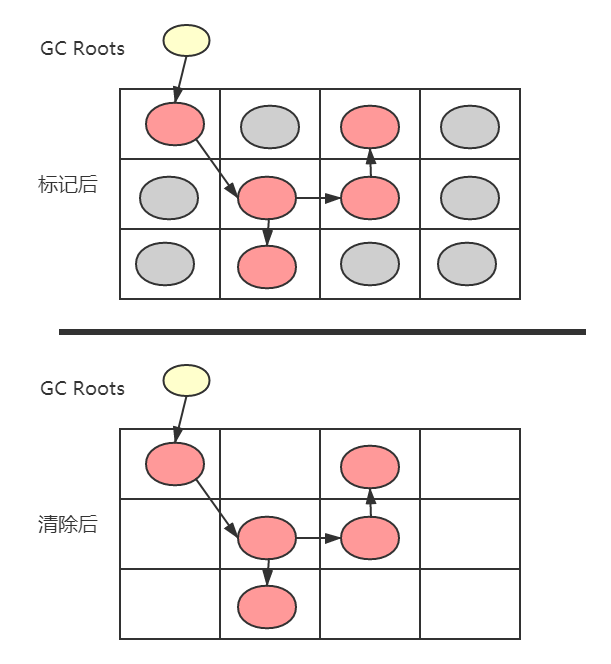
<https://www.lubojian.cn/post/112.html>

分为两阶段：首先标记出所有需要回收/无用的对象，在标记完成后统一回收所有被标记的对象。

###### 参2：<https://time.geekbang.org/column/article/40961> 评论

大多数主流虚拟机采用可达性分析算法来判断对象是否存活，在标记阶段，会遍历所有 GC ROOTS，将所有 GC ROOTS 可达的对象标记为存活。只有当标记工作完成后，清理工作才会开始。

###### 参3：享学vip 3期-对象的分配及垃圾回收机制下的1:00:17



##### 【2】 优缺点

###### 优点：

实现简单，不需要对象进行移动。

###### 缺点：

效率问题：标记和清除两个过程的效率都不高；

空间问题：会产生很多碎片。

#### （2） 复制（Copying）算法

##### 【1】 原始算法

###### 是什么

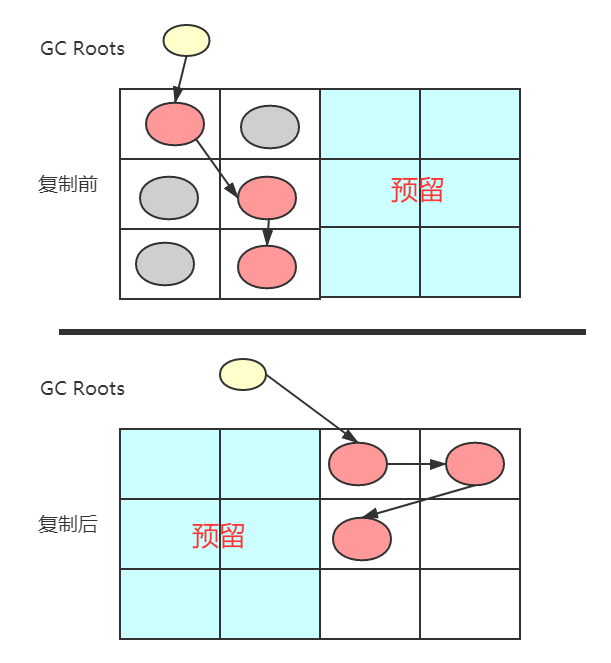
参1：

<https://www.cnblogs.com/little-YTMM/p/5613642.html>

<https://www.lubojian.cn/post/112.html>

将可用内存按容量划分为大小相等的两块，每次只用其中一块。当这一块用完了【个人理解是没有（连续）内存可分配了】，就将还存活的对象复制到另一块上面，然后把原始空间全部回收【包括回收所有可回收对象】。

参2：享学vip 3期-对象的分配及垃圾回收机制下的52:22



###### 优缺点

优点：

实现简单、运行高效；

没有内存碎片。

缺点：

内存使用率不高，只有原来的一半。

##### 【2】 Appel式的复制回收算法

###### 是什么

参1：<https://www.lubojian.cn/post/112.html>

参1-1：

由于原始复制算法内存的代价太高，每次基本上都要浪费一半的内存。于是将原始算法进行了改进，内存区域不再是按照 1：1 去划分，而是将内存划分为 8:1:1 三部分，较大那份内存叫 Eden 区，其余是两块较小的内存区叫 Survior 区。每次都会优先使用 Eden 区，若 Eden 区满，就将对象复制到第二块内存区上，然后清除 Eden 区，如果此时存活的对象太多，以至于 Survivor 不够时，会将这些对象通过分配担保机制复制到老年代中。(java堆又分为新生代和老年代)

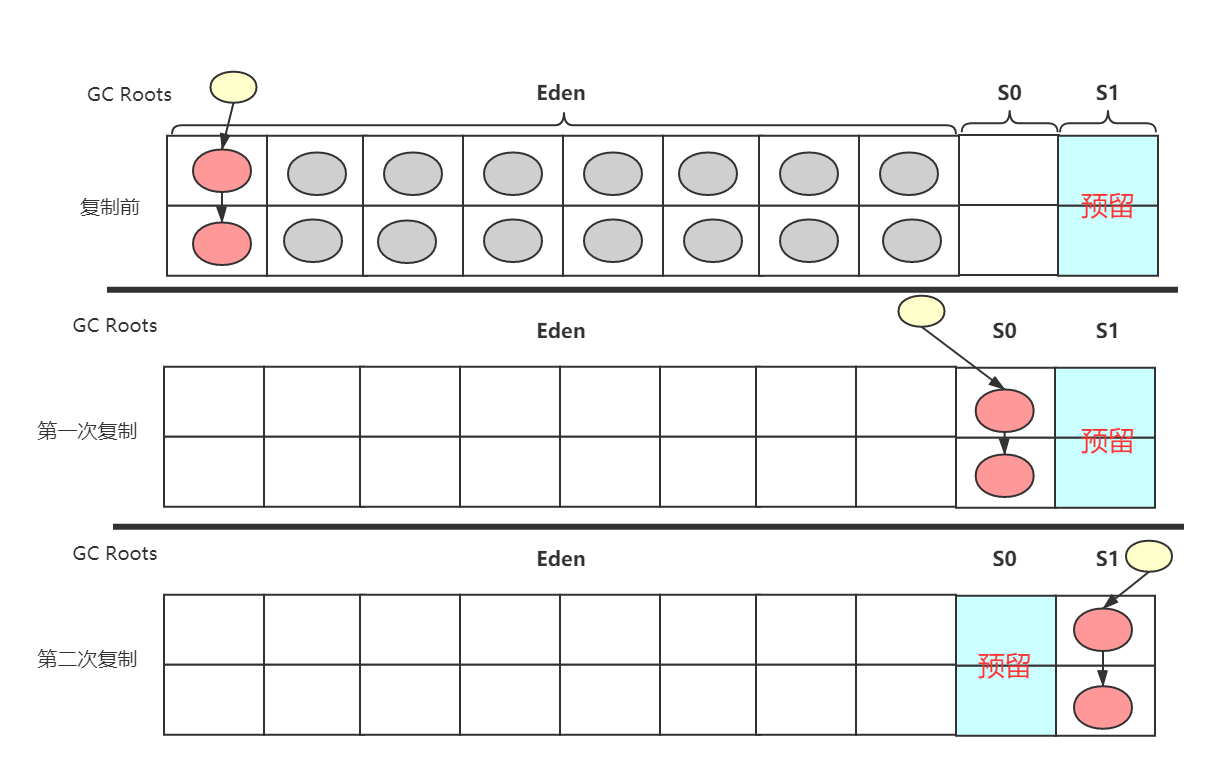
参1-2：

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

* 把 Eden + From Survivor 存活的对象放入 To Survivor 区；
* 清空 Eden 和 From Survivor 分区；
* From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

参2：享学vip 3期-对象的分配及垃圾回收机制下的55:02



一种更加优化的复制回收分代策略：具体做法是分配一块较大的 Eden 区和两块较小的 Survivor 空间（你可以叫做 From 或者 To， 也可以叫做 Survivor1和 Survivor2）。  
专门研究表明， 新生代中的对象 98%是“朝生夕死” 的， 所以并不需要按照 1:1 的比例来划分内存空间， 而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间， 每次使用 Eden 和其中一块 Survivor。 当回收时， 将 Eden 和 Survivor 中还存活着的对象一次性地复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。  
HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1， 也就是每次新生代中可用内存空间为整个新生代容量的 90%（ 80%+10%） ， 只有 10%的内存会被“浪费”【空间利用率达到90%】。 当然， 98%的对象可回收只是一般场景下的数据， 我们没有办法保证每次回收都只有不多于 10%的对象存活， 当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代） 进行分配担保（Handle Promotion）。

###### 《2》 优缺点

优点：

提高空间利用率和空间分配担保。

缺点：

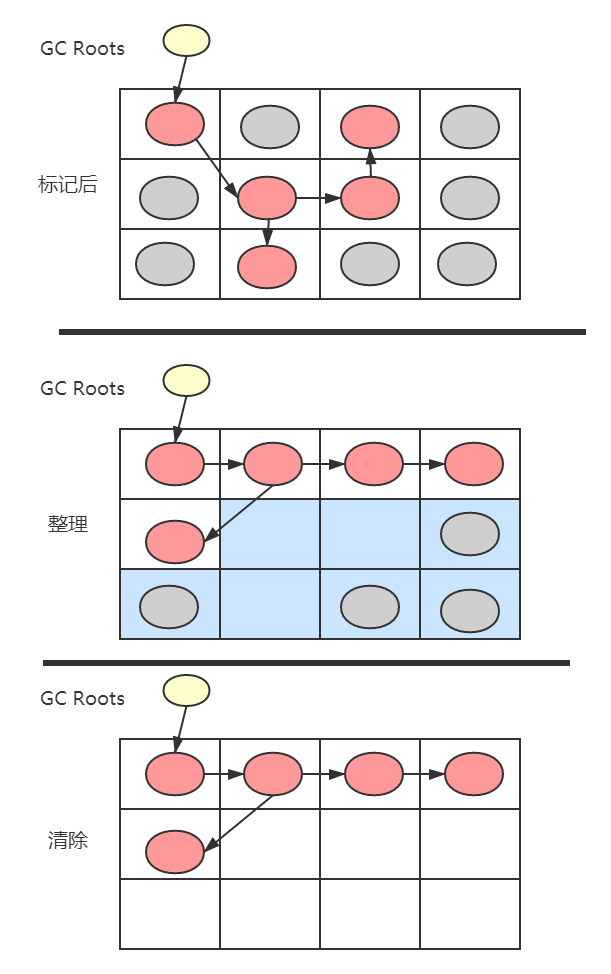
#### （3） 标记-整理（Mark-Compact）算法

##### 【1】 是什么

###### 参1：<https://www.cnblogs.com/little-YTMM/p/5613642.html>

标记过程与标记-清除算法过程一样，但后面不是简单的清除，而是让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。

###### 参2：享学vip 3期-对象的分配及垃圾回收机制下的1:03:22



##### 【2】 优缺点

###### 优点：

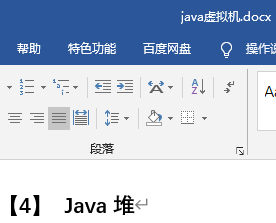
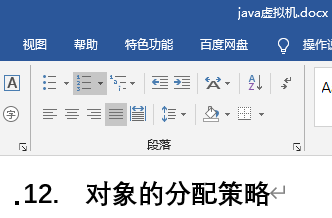
没有内存碎片。

###### 缺点：

仍需要进行局部对象移动，一定程度上降低了效率。

#### （4） 分代收集（Generational Collection）算法

##### 【1】 是什么

下面的新生代和老年代的概念见和。

###### 参1：

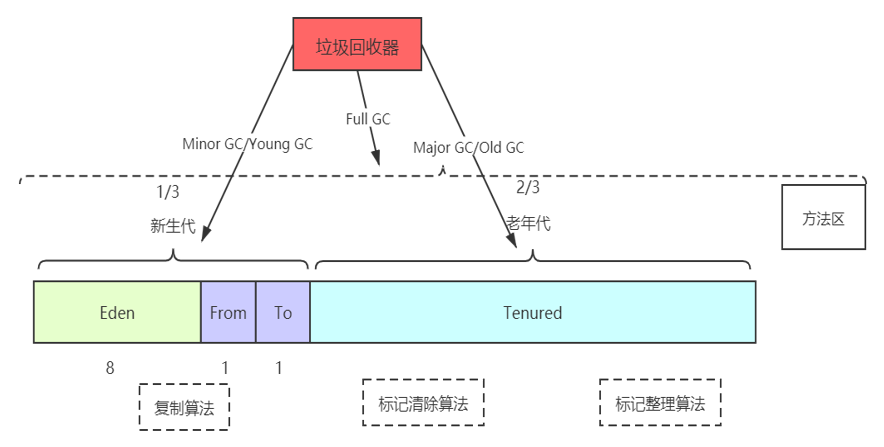
<https://www.cnblogs.com/little-YTMM/p/5613642.html>

<https://www.lubojian.cn/post/112.html>

**新生代**中，对象生存期短，每次垃圾收集时都有大批对象死去，只有少量存活，就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。

**老年代**中，对象存活率较高，没有额外空间进行分配担保，就应该使用“标记-整理”或“标记-清除”算法回收。

###### 参2：享学vip 3期-对象的分配及垃圾回收机制下的47:27+58:54



新生代+老年代=堆

新生代用改进的复制算法，即Appel式的复制算法；老年代用标记清除或标记整理算法。

### 6.2 GC分类

参考：

1. 享学vip 3期-对象的分配及垃圾回收机制下的47:27
2. Full GC：

<https://blog.csdn.net/kep159/article/details/103088438?utm_source=app&app_version=4.21.0&code=app_1562916241&uLinkId=usr1mkqgl919blen>

1. <https://www.cnblogs.com/ZJOE80/p/12931989.html>
2. <https://www.cnblogs.com/little-YTMM/p/5613642.html>
3. <https://www.cnblogs.com/dalianpai/p/14147534.html>

#### （1） Minor GC/Young GC【新生代GC】

##### 【1】 是什么

只回收新生代。

##### 【2】 触发时机

<https://blog.csdn.net/wangxuelei036/article/details/108180150>

<https://www.cnblogs.com/little-YTMM/p/5613642.html>

系统自动触发的时机只有一个，就是 Eden 区没有足够的空间分配给新创建的对象。

注：只是Eden 区。对于S区，一个存放着存活对象，一个空值等待下一次Minor GC 完成后，存活对象的转移，并不用于分配给新生对象。

#### （2） Major GC/Old GC【老年代GC】

##### 【1】 是什么

只回收老年代（CMS垃圾回收器）。

目前这种很少见了，因为绝大部分情况只要回收老年代，基本都会触发Full GC来回收新生代。

出现了Major GC，经常会伴随至少一次的Minor GC。

##### 【2】 触发时机

<https://blog.csdn.net/qq_29860591/article/details/111307416>

<https://cloud.tencent.com/developer/article/1939522>

在老年代空间不足时触发。

#### （3） Full GC

##### 【1】 是什么

回收新生代，老年代，方法区。

##### 【2】 触发时机

<https://blog.csdn.net/wangxuelei036/article/details/108180150>

Full GC 系统触发条件相对来说要复杂一点：

* 老年代空间不足，这个很简单，就是字面上的不足，例如：大对象不停的直接进入老年代，最终造成空间不足。
* 方法区空间不足。
* Minor GC 引发 Full GC。

### 6.3 一些收集器

参考：

1. 主

<https://ke.qq.com/webcourse/index.html?r=1677559851652#cid=347420&term_id=103755197&taid=12286527044668700&type=3072&source=PC_COURSE_DETAIL&vid=3701925922460860215> 和

1. 辅

<https://www.lubojian.cn/post/112.html>

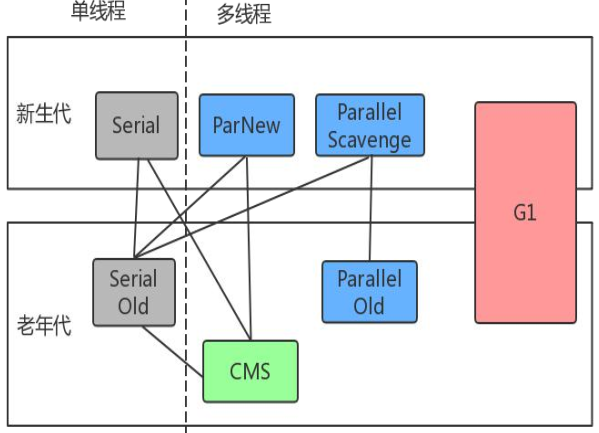
<https://xie.infoq.cn/article/83777ef09f349643cee6b5cd7>

<https://www.cnblogs.com/little-YTMM/p/5613642.html>

#### （1） 概述

如果说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

下图展示了7种收集器，其中用于回收新生代的收集器包括Serial、ParNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。不同收集器之间的连线表示它们可以搭配使用。



一般都是一个新生代和一个老年代收集器搭配使用。Serial Old和CMS的连线称为“CMS的退化”，见

<https://ke.qq.com/webcourse/index.html?r=1677559851652#cid=347420&term_id=103755197&taid=12286527044668700&type=3072&source=PC_COURSE_DETAIL&vid=3701925922460860215> 1:50:03。

#### （2） 详解

##### 【1】 Serial+Serial Old

1. 相同

都是单线程串行。即在它（GC线程）进行垃圾收集时，必须暂停其他所有的工作/用户线程，直到它收集结束【Stop the world】。

1. 不同

Serial：新生代+复制算法

Serial Old：老年代+标记整理算法

##### 【2】 Parallel Scavenge+Parallel Old

1. 相同

都是多线程并行，吞吐量优先。

1. 不同

Parallel Scavenge：新生代+复制算法

Parallel Old：老年代+标记整理算法

##### 【3】 ParNew

《1》 多线程并行+新生代+复制算法

《2》 是Serial收集器的多线程版本，主要用于配套CMS

##### 【4】 CMS《Concurrent Mark Sweep》

1. 多线程并发+老年代+标记清除算法
2. 以获取最短回收停顿时间为目标的收集器
3. 发展历程

Java9：进入废弃倒计时，因为问题太多-》java14：正式移除

##### 【5】 G1

1. 多线程并发+{跨新生代和老年代}+{标记整理算法+化整为零}
2. jdk1.9开始，默认的垃圾回收器是G1

##### 【6】 ZGC

<https://ke.qq.com/webcourse/index.html?r=1677559851652#cid=347420&term_id=103755197&taid=12286527044668700&type=3072&source=PC_COURSE_DETAIL&vid=3701925922460860215> 

<https://tech.meituan.com/2020/08/06/new-zgc-practice-in-meituan.html>

1. 是一款低延迟垃圾回收器，它的设计目标包括：

* 停顿时间不超过10ms；
* 停顿时间不会随着堆的大小，或者活跃对象的大小而增加；
* 支持8MB~4TB级别的堆（未来支持16TB）。

《2》 发展历程

Java11：开始加入-》java13：ZGC最大的堆空间从4TB提升到16TB

#### （3） 一些总结

【1】 从CMS开始的垃圾回收器的优化方向都是减少STW《Stop the world》，即减少暂停/停顿。

### 总结：

该部分主要分析了三种不同的GC算法：Mark-Sweep，Copy，Mark-Compact。每种算法都有不同的优缺点，也有不同的适用范围。而JVM中对垃圾回收器并没有严格的要求，不同的收集器会结合多个算法进行垃圾回收。

## 相关API

### System.gc()

|  |
| --- |
| /\*\*  \* Runs the garbage collector.  \* <p>  \* Calling the <code>gc</code> method suggests that the Java Virtual  \* Machine expend effort toward recycling unused objects in order to  \* make the memory they currently occupy available for quick reuse. \* When control returns from the method call, the Java Virtual  \* Machine has made a best effort to reclaim space from all discarded  \* objects.  \* <p>  \* The call <code>System.gc()</code> is effectively equivalent to the  \* call:  \* <blockquote><pre>  \* Runtime.getRuntime().gc()  \* </pre></blockquote>  \*  \* @see java.lang.Runtime#gc()  \*/  public static void gc() {  Runtime.getRuntime().gc();  } |

注释看出该方法用于运行Garbage Collector。调用该方法JVM将努力回收未使用的对象来使它们目前占用的内存能尽快被重用。当方法被调用结束时，JVM已经尽最大努力回收所有被废弃对象的空间。

该方法等价于：Runtime.getRuntime().gc();

### System.runFinalization()

|  |
| --- |
| /\*\*  \* Runs the finalization methods of any objects pending finalization.  \* <p>  \* Calling this method suggests that the Java Virtual Machine expend  \* effort toward running the <code>finalize</code> methods of objects  \* that have been found to be discarded but whose <code>finalize</code>  \* methods have not yet been run. When control returns from the  \* method call, the Java Virtual Machine has made a best effort to  \* complete all outstanding finalizations.  \* <p>  \* The call <code>System.runFinalization()</code> is effectively  \* equivalent to the call:  \* <blockquote><pre>  \* Runtime.getRuntime().runFinalization()  \* </pre></blockquote>  \*  \* @see java.lang.Runtime#runFinalization()  \*/  public static void runFinalization() {  Runtime.getRuntime().runFinalization();  } |

从注释看出，运行所有正等待被终止对象的finalization方法。

调用该方法表明JVM将努力运行那些已经被认定是废弃的但finalize方法还未被调用的对象的 finalize 方法。当该方法调用结束，JVM已经尽最大努力完成了所有需要完成的finalization方法。

该方法等价于：

Runtime.getRuntime().runFinalization()。

## 总结

主要讲了Java的GC原理，从What，When，How三方面对如何进行垃圾回收做了分析。

简而言之：

What —— 堆和方法区；

When —— 已死的对象（引用不可达）；

How —— 标记-清除-整理-复制算法；

关于GC，要牢牢把握这三个问题，然后发散性思维。

最后对Java对对象的内存分配策略进行了介绍：新生代Eden区 -- Survivor区 -- 老年代

## 补充

### jvm的自动垃圾回收和c++的手动回收对比

享学vip 3期：20210812的垃圾回收机制上的36:05

### finalize方法

享学vip 3期：对象的分配及垃圾回收机制上的59:06+对象的分配及垃圾回收机制下的0:00开始

### Stop the world<STW>

<https://www.jianshu.com/p/d686e108d15f?utm_campaign=haruki&utm_content=note&utm_medium=reader_share&utm_source=weixin>

<https://blog.csdn.net/sinat_25306771/article/details/52374498?utm_source=app&app_version=4.21.0&code=app_1562916241&uLinkId=usr1mkqgl919blen>

<https://ke.qq.com/webcourse/index.html?r=1677559851652#cid=347420&term_id=103755197&taid=12286527044668700&type=3072&source=PC_COURSE_DETAIL&vid=3701925922460860215> 

## 面试题

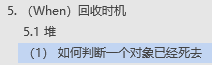
### 如何判断对象是否可以被回收/对象是否存活/GC对象的判定方法

参考：

1. <https://xie.infoq.cn/article/83777ef09f349643cee6b5cd7>
2. <https://www.lubojian.cn/post/112.html>
3. <https://blog.csdn.net/zhangvalue/article/details/101034370>
4. 和享学alvin老师的qq对话

有2种方法：

1. 引用计数法：存在循环引用问题，jvm未采用
2. 可达性分析法：jvm采用

详细解释见上面。