1、什么是垃圾(Garbage)呢？

垃圾是指在**运行程序中没有任何指针指向的对象**，这个对象就是需要被回收的垃圾。如果**不及时**对内存中的**垃圾进行清理**，那么，这些垃圾对象**所占的内存空间**会**一直保留到应用程序结束**，被保留的空间**无法被其他对象使用**。甚至**可能导致内存溢出**。

【补充】如何判断一个常量是废弃常量？

假如在字符串常量池中存在字符串 "abc"，如果**当前没有任何 String 对象引用该字符串常量**的话，就说明常量 "abc" 就是废弃常量。运行时常量池主要回收的是废弃的常量。

【补充】如何判断一个类是无用的类？

方法区主要回收的是无用的类，需要同时满足下面 3 个条件才能算是 “无用的类”：：

1）该类**所有的实例**都已经**被回收**；

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**在任何地方**被引用**，**无法**在任何地方**通过反射访问**该类的方法。

2、为什么需要 GC？

1）如果不进行垃圾回收，**内存**迟早都会被**消耗完**，可能**导致内存溢出**；

2）除了释放没用的对象，垃圾回收也可以**清除内存里的记录碎片**，以便 JVM 将整理出的**内存分配给新的对象**。

3、Java 中垃圾回收的重点区域？

垃圾回收的作用区域就是**堆**和**方法区**，其中堆是重点。从次数上说，频繁收集**新生代**，较少收集**老年代**，基本不动**永久代（元空间）**。

4、垃圾判别阶段的算法

当一个对象已经**不再被任何的存活对象继续引用**时，就可以宣判为死亡对象。

1）引用计数算法

用一个整型的**引用计数器属性**来**记录对象被引用**的情况。对于一个对象 A，只要有任何一个对象引用了 A，则 A 的引用计数器就加 1，当引用失效时，引用计数器就减 1。只要对象 A 的**引用计数器的值为 0**，即表示对象 A 不可能再被使用，**可进行回收**。

优点：实现简单，垃圾对象便于辨识；**判定效率高**，回收没有延迟性。

缺点：1>需要单独的字段存储计数器；（空间）

2>每次赋值都需要更新计数器；（时间开销）

3>最严重问题：**无法处理循环引用**的情况，会导致内存泄漏。

【解决循环引用问题】

1>手动解除：很好理解，就是在**合适的时机，解除引用关系**；

2>使用弱引用 weakref， weakref 是 Python 提供的标准库，旨在解决循环引用。

2）可达性分析算法

将**对象**及其**引用关系**看作**一个图**，以**根对象集合**(GC Roots)为**起始点**，按照**从上至下**的方式**搜索**被根对象集合**所连接的目标对象**是否可达。**内存中的存活对象**都会被**根对象集合直接或间接连接**着，如果目标对象**没有**任何**引用链相连**，可以认为是**可回收对象**。

优点：解决循环引用的问题，防止内存泄漏。

【注意】1>使用可达性分析算法来判断内存是否可回收，那么分析工作必须在一个能保障**一致性的快照中**进行。这也是导致 GC 进行时必须“Stop The World”的一个重要原因。

2>CMS 收集器中，**枚举根节点时**也是必须要**停顿**的。

5、GC Roots 包括以下几类元素：

1）虚拟机栈中引用的对象；

比如：各个线程被调用的方法中使用到的参数、局部变量等。

2）本地方法栈内 JNI(本地方法)引用的对象；

3）类静态属性引用的对象；

比如：Java 类的引用类型静态变量

4）方法区中常量引用的对象；

比如：字符串常量池（String Table）里的引用

5）所有被同步锁 synchronized 持有的对象；

6）Java 虚拟机内部的引用；

基本数据类型对应的 Class 对象，一些常驻的异常对象（如：NullPointerException、OutOfMemoryError），系统类加载器。

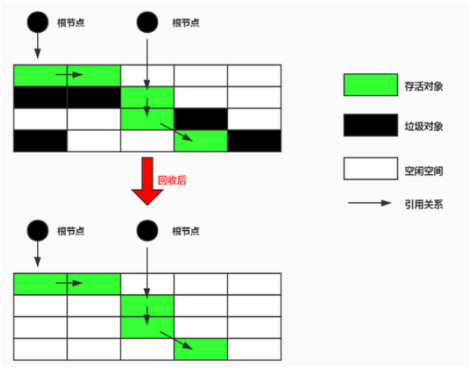
7）反映 java 虚拟机内部情况的 JMXBean、JVMTI 中注册的回调、本地代码缓存等。

6、标记-清除（Mark - Sweep）算法

当堆中的**有效内存空间被耗尽**的时候，就会**停止整个程序**（也被称为 stop the world），然后**进行两项工作**，一是**标记**，二是**清除**。

1）标记：Collector 从**引用根节点**开始**遍历**，**标记所有被引用的对象**。一般是在对象的 Header 中记录为可达对象。（注意不是标记垃圾，是标记不被回收的对象）

2）清除：Collector 对堆内存**从头到尾**进行线性的**遍历**，如果发现某个对象在**其 Header 中没有标记为可达对象**，则将其**回收**。【注意】这里所谓的清除**并不是真的置空**，而是把需要清除的对象地址**保存在空闲的地址列表**里。



缺点：

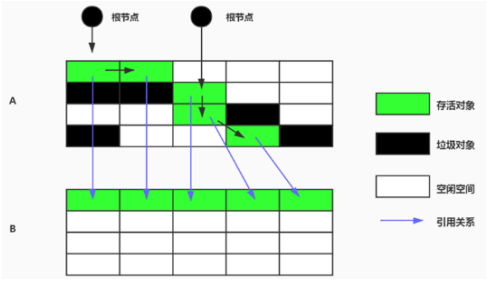
1）**效率比较低：**递归与全堆对象遍历两次；

2)在进行 GC 的时候，需要**停止整个应用程序**，导致用户体验差；

3）这种方式清理出来的**空闲内存是不连续**的，**产生内存碎片**。

7、复制算法

将活着的**内存空间分为两块**，每次**只使用其中一块**，在垃圾回收时**将正在使用的内存**中的**存活对象*复制***到**未被使用的内存块**中，之后**清除正在使用的**内存块中的**所有对象**，**交换**两个内存的**角色**，最后完成垃圾回收。



优点：

1）没有标记和清除过程，实现简单运行高效；

2）复制过去以后保证空间的连续性，**不会出现“碎片”问题**。

缺点：

1）需要**两倍的内存空间**；

2）GC **需要维护 region 之间对象引用关系，内存占用**和**时间开销大**。

3）如果系统中的**存活对象很多**，复制算法不会很理想。**适合存活对象少、垃圾对象多**。

应用场景：

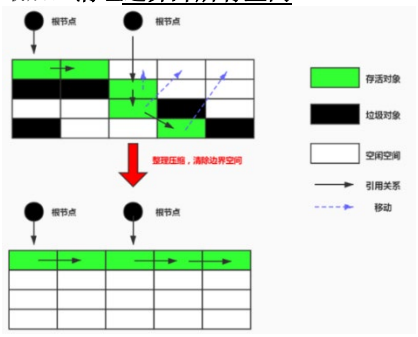
在新生代中的对象大部分都是“朝生夕死”，利用复制算法回收性价比很高。

8、标记-压缩（Mark - Compact）算法

第一阶段和标记-清除算法一样，从**根节点开始标记所有被引用对象**；

第二阶段将**所有的存活对象**压缩到内存的一端，**按顺序排放**。

最后，**清理**边界外**所有空间**。



优点：1）**消除**了标记-清除算法中，**内存碎片的缺点**，我们需要给新对象分配内存时，JVM **只需要**持有一个**内存的起始地址**即可。

2）**消除**复制算法当中，**内存减半的高额代价**。

缺点：1）标记-压缩算法的**效率低**于复制算法。不仅要**标记**所有存活对象，还要**整理**所有存活对象的引用地址。

2）**移动对象的同时**，如果对象被其他对象引用，则还需要**调整引用的地址**。

3）移动过程中，需要**全程暂停用户应用程序**。

9、分代收集算法

不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以**采取不同的收集方式**，以便提高回收效率。

1）年轻代：区域相对较小，对象**生命周期短**、存活率低，**回收频繁**。**适合复制算法速度最快。**（复制算法**内存利用率不高**的问题可以通过**两个 survivor 的设计**得到缓解。）

2）老年代：区域较大，对象**生命周期长**、存活率高，**回收不频繁**。一般使用**标记-清除**或者是**标记-清除与标记-整理的混合**实现。

【总结】

标记阶段的开销**与存活对象的数量**成正比。

清除阶段的开销与**所管理区域大小**成正相关。

整理阶段的开销与**存活对象的数据**成正比。

10、增量收集算法

如果**一次性将所有的垃圾进行处理**，需要造成**系统长时间的停顿**，那么就可以让**垃圾收集线程**和**应用程序线程**交替执行。每次，垃圾收集线程**只收集一小片区域**的内存空间，**接着切换**到应用程序线程。**依次反复**，直到垃圾收集完成。

缺点：因为**线程切换和上下文转换**的消耗，会使得垃圾回收的总体成本上升，**造成系统吞吐量的下降**。

11、分区算法：---G1 GC 使用的算法

按照**对象的生命周期长短**划分成**两个部分**，分区算法将**整个堆空间**划分成**连续的不同小区间**，每一个小区间都独立使用，独立回收。根据**目标的停顿时间，每次合理地回收若干个小区间**，而不是整个堆空间，从而**减少一次 GC 所产生的停顿**。

12、System.gc()方法

通过 System.gc()/Runtime.getRuntime().gc()的调用，会**显式触发 Full GC**，同时对**老年代**和**新生代**进行回收。（注意不一定马上就发生，需要等到安全点）

【注意】1）一般情况下，**垃圾回收**应该是**自动进行**的，无须手动触发，在一些特殊情况下，如我们正在**编写一个性能基准**，我们可以在运行之间**手动调用 System.gc()**。

2）System.gc()中调 Runtime.getRuntime().gc()。

13、finalize()方法

finalize()是 Object 的 **protected 方法**，子类可以覆盖该方法以**实现资源清理工作**，**GC** 在回收对象**之前**调用该方法。

【注意】**finalize()**与 C++中的**析构函数**不是对应的。C++中的析构函数**调用的时机是确定的**（对象离开作用域或 delete 掉），但 Java 中的 **finalize 的调用具有不确定性**。

14、finalize 的执行过程

当**对象**变成**不可达时**，GC 会**判断**该对象是否**覆盖了 finalize 方法**，若**未覆盖**，则直接将其**回收**。否则，若对象**未执行过 finalize 方法**，将**其放入 F-Queue 队列**，由一低优先级线程**执行**该队列中对象的 **finalize 方法**。执行 finalize 方法完毕后，GC 会**再次判断**该对象是否**可达**，若不可达，则进行回收，否则，对象**“复活”**。（可被回收对象一定会被回收吗？不）

15、内存溢出

1）OOM：**没有空闲内存**，并且垃圾收集器也**无法提供更多内存**。

2）内存不够的原因：1）Java 虚拟机的**堆内存设置不够**；2)代码中创建了**大量大对象被引用**，并且长时间不能被垃圾收集器收集。

3）OOM 前必有 GC 吗？（不一定）

在 **java.nio.BIts.reserveMemory()方法**中，能看到 System.gc()会被调用，以清理空间。

【特殊情况】直接**分配一个超大对象（超出堆的最大值）**，JVM 判断垃圾收集解决不了问题便直接抛出 OOM。

16、内存泄漏

对象**不会再被程序用到**，但是 **GC 又不能回收**他们的情况。例如可达性算法判断对象是否还在使用（即**被引用**），**但实际上并不需要**该对象了，此时却回收不了，造成内存泄漏。

【理解】**生命周期长**的对象 X **引用**着**生命周期短**的对象 Y，那么 Y 的生命周期结束后，因为倍 X 引用所以回收不了，造成内存泄漏。

【关系】内存泄漏的增多，最终导致内存溢出。

17、内存泄露的 8 种情况（不需要还被用着）

1）静态集合类：如 HashMap、LinkedList 等，**它们的生命周期与 JVM 程序一致**，则**容器中的对象**在程序结束之前将不能被释放，从而造成内存泄漏。

2）单例模式：因为单例的**静态**特性，**它的生命周期和 JVM 的生命周期一样长**，所以如果**单例对象**如果**持有外部对象的引用**，那么这个**外部对象也不会被回收**，那么就会造成内存泄漏。

3）内部类持有外部类：如果一个**外部类**的实例对象的**方法返回了**一个**内部类的实例对象**。这个**内部类对象被长期引用了**，即使那个**外部类实例对象不再被使用**，但由于**内部类持有外部类的实例对象**，这个外部类对象将不会被垃圾回收，这也会造成内存泄漏。

【理解】页面对象不在需要了，但是其内部对象有延迟消息，此时外部的页面对象虽然不需要但还是被引用，故回收不了。

4）各种连接，如数据库连接、网络连接和 IO 连接等：这些连接不再使用时需调用 close 方法关闭连接，才会被回收，否则内存泄漏。

5）变量不合理的作用域：一个变量的**定义的作用范围**大于**其使用范围**，如果该变量**不再需要时，没有及时**地把对象**设置为 null**，很有可能导致内存泄漏的发生。

6）改变哈希值（找不到了）：当一个对象被**存储进 HashSet 集合**中以后，就不能修改这个对象中的哈希值的字段，修改后并**不能**通过 **contains 方法**使用**当前引用**参数**去检索对象**，也就无法单独删除当前对象了。

7）缓存泄露：当对象引用放入缓存中，当**不需要的时候还始终存在**，造成内存泄漏。（使用 **WeakHashMap** 代表缓存，除了**自身有对 key 的引用**外，此 **key 没有其他引用**那么此 map 会**自动丢弃此值**）

8）监听器和回调：如果客户端在实现的 API 中注册回调，却**没有显式取消**，那么就会积聚。（回调立即被当作垃圾回收：保存弱引用）

18、Stop-the-World ，简称 STW

指的是 **GC 事件发生过程**中，会产生**应用程序的停顿**。停顿产生时整个应用程序线程都会**被暂停**，没有任何响应。

【注意】STW 是 JVM 在后台自动发起和自动完成的。它**和采用哪款 GC 无关**，**所有的 GC** 都有这个事件，只能说尽可能缩短暂停时间。

19、安全点

程序执行时**并非在所有地方**都能停顿下来开始 GC，**只有在特定的位置**才能**停顿**下来**开始 GC**。

安全点如果太**少**可能导致 **GC 等待的时间太长**，如果**太频繁**可能导致**运行时的性能**问题。一般选择**执行时间较长**的指令作为安全点。

20、如何在 GC 发生时，检查所有线程都跑到最近的安全点停顿下来呢？

1）抢先式中断：（目前没有虚拟机采用了）

首先**中断所有线程**。如果还有线程**不在安全点**，就**恢复线程**，让线程**跑到安全点**。

2）主动式中断：

设置一个**中断标志**，**各个线程**运行到 Safe Point 的时候**主动轮询**这个**标志**，如果中断标志**为真**，则将自己进行**中断挂起**。

21、安全区域

对于**程序不执行**的时候（线程睡眠或阻塞）**无法跑到安全点中断挂起**，也不能等到被唤醒，此时需要使用安全区域。

安全区域是指在**一段代码片段中**，对象的**引用关系不会发生变化**，在**这个区域**中的任何位置开始 **GC 都是安全**的。

22、5 种引用类型总结

1）强引用：默认的引用类型，具有强引用的对象是不会被 GC 回收的，宁愿抛出 OOM 异常，也不会回收强引用所指向对象。

2）软引用：描述**还有用，但非必需的对象**。**只被软引用关联着**的对象，在系统将要发生**内存溢出异常前**，会把这些对象**列进回收范围之中**进行**第二次回收**，如果这次回收还没有足够的内存，才会抛出内存溢出异常。

（软引用用来实现内存敏感的高速缓存：如果还**有空闲内存**，暂时**保留缓存**，**内存不足**时**清理掉**，保证使用缓存的同时，不耗尽内存。）

3）弱引用：描述那些**非必需对象，只被弱引用关联的对象**只能**生存到下一次垃圾收集发生为止**。在系统 GC 时，**只要发现**弱引用，**不管**系统堆空间使用**是否充足**，都会**回收掉**只被弱引用关联的对象。（不过，由于**垃圾回收器**是一个**优先级很低的线程，并不一定能很快地发现**持有弱引用的对象）

弱引用对象与软引用对象的最大不同：当 GC 在进行回收时，要通过**算法检查**是否**回收软引用对象**，而对于**弱引用对象，GC 总是进行回收**。弱引用对象**更容易、更快被 GC 回收**。

4）虚引用：虚引用主要用来**跟踪对象被垃圾回收的活动**。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。**无法通过虚引用**来获取**被引用的对象**。

【注意】虚引用与软引用和弱引用的一个区别：**虚引用**必须**和引用队列**（ReferenceQueue）**联合使用**。当垃圾回收器**准备回收一个对象**时，如果发现**它还有虚引用**，就会在**回收**对象的**内存之前**，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**，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**。

5）终结器引用：用以实现对象的 finalize()方法。在 **GC 时，终结器引用入队**。由 **Finalizer 线程**通过**终结器引用**找到被引用对象并**调用它的 finalize()方法**，**第二次 GC 时**才能**回收被引用对象**。

23、Serial 收集器

1）是一个**单线程收集器**。它只会使用**一条垃圾收集线程**去完成垃圾收集工作，在进行垃圾收集工作的时候**必须暂停其他所有的工作线程**（STW），**直到它收集结束**。

2）**新生代**采用**标记-复制算法**，**老年代**采用**标记-整理算法**。

3）**简单而高效**，没有线程交互的开销。

4）适合 **Client 模式下**的虚拟机。

Serial/Serial Old收集器运行示意图

【补充】Serial Old 收集器

Serial 收集器的老年代版本，单线程收集器。两个用途：1）与 **Parallel Scavenge 收集器**搭配使用；2）作为 **CMS 收集器**的后备方案。

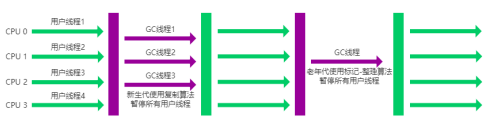
24、ParNew 收集器

1）是 **Serial 收集器的多线程版本**，在多核 CPU 环境下有着比 Serial 更好的表现。

2）**新生代**采用**标记-复制算法**，**老年代**采用**标记-整理算法**。

【补充】**并行**：指**多条垃圾收集线程**并行工作，但此时**用户线程**仍然处于**等待**状态。

并发：指**用户线程与垃圾收集线程同时**执行（但不一定是并行，可能会**交替执行**），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。



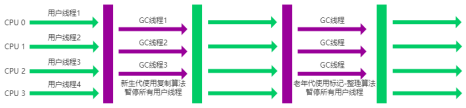
ParNew/Serial Old收集器运行示意图

25、Parallel Scavenge 收集器

1）并行收集器，**追求高吞吐量，高效利用 CPU**。吞吐量 = 用户线程时间/(用户线程时间+GC 线程时间)，高吞吐量可以高效率的利用 CPU 时间，尽快完成程序的运算任务，适合后台应用等**对交互相应要求不高的场景**。

2）**新生代**采用**标记-复制算法**，**老年代**采用**标记-整理算法**。

3）Parallel Scavenge 收集器提供了很多参数供用户**找到最合适的停顿时间或最大吞吐量**。



Parallel Scavenge/Parallel Old收集器运行示意图

【补充】Parallel Old 收集器

Parallel Scavenge 收集器的**老年代版本**。

在**注重吞吐量以及 CPU 资源**的场合，可以优先考虑 Parallel Scavenge 收集器和 Parallel Old 收集器。

26、CMS 收集器（两次停顿）

1）CMS 收集器是一种以**获取最短回收停顿时间**为目标的收集器。它非常符合在**注重用户体验的**应用上使用。

2）CMS 收集器是 HotSpot 虚拟机**第一款**真正意义上的**并发收集器**，它第一次实现了让垃圾收集线程与用户线程（**基本上**）**同时工作**。

3）CMS 收集器是**“标记-清除”算法**实现：

1>初始标记：暂停所有的其他线程（STW 时间非常短），标记与 GC Roots **直接关联**的对象；（注意，后面引用的对象不管）

2>并发标记：从 **GC Roots 的直接关联对象**开始**遍历整个对象图**的过程，这个**过程耗时**较长但是**不需要停顿用户线程**，可以与垃圾收集线程一起并发运行。

3>重新标记：为了**修正并发标记期间**因为用户程序继续运行而**导致标记产生变动的那一部分对象的标记记录**（主要关注**不可达变可达**的对象），**这个阶段的停顿时间**一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短。

4>并发清除：开启用户线程，同时 GC 线程开始对未标记的区域做清扫。

Concurrent Mark Sweep 收集器运行示意图

【注意】标记清理会出现碎片，为啥不用压缩？

当并发清理的时候，如果压缩整理内存的话，**用户线程原来使用的**内存资源**无法访问**使用。

CMS 优点：并发收集、低延迟；

CMS 缺点：1>**产生内存碎片**；2>对 **CPU 资源非常敏感**，总吞吐量降低；3>**无法处理浮动垃圾**，并发标记阶段可能产生新的垃圾对象无法被标记到（可达变不可达），不能被清理。

27、G1 收集器(Garbage-First)

G1 是一款**面向服务器**的垃圾收集器,主要针对**配备多个处理器**及**大容量内存**的机器. 以极高概率**满足 GC 停顿时间**要求的同时,还具备**高吞吐量性能**特征.

1）特点

1>并行与并发：G1 能充分**利用 CPU 多核环境下**的硬件优势，使用**多个 CPU（CPU 核心）**来**缩短 STW 停顿**时间。

2>分区域收集：G1 将整个堆划分为**多个大小相等的独立区域**，每个区域看作 Eden 区、Survivor 区或 Old 区等角色，更加灵活地进行内存管理和垃圾收集。

3>空间整合：与 CMS 的“标记-清除”算法不同，G1 从**整体来看**是**基于“标记-整理”算法**实现的收集器（减少内存碎片）；从**局部来看**是**基于“标记-复制”算法**实现的。

4>可预测的停顿：能让使用者明确指定在一个长度为 **M 毫秒的时间片段内**，消耗在**垃圾收集**上的**时间不得超过 N 毫秒**。

5>优先回收垃圾最多区域：根据**回收价值和成本**进行**排序**，提高垃圾收集的效率。

2）流程：1>初始标记；2>并发标记；3>最终标记；4>筛选回收（并发）：根据每个区的**垃圾堆积情况和回收价值**进行**排序，选择性地**回收**部分区域**。回收过程将存活的对象从一个区域复制或移动到另一个区域，并更新相关引用。

28、怎么选择垃圾收集器？

1）串行收集器：1>**内存小于 100M**；2>**单核**、单机程序，并且**没有停顿时间**的要求；**2）并行收集器**：多 CPU、需要**高吞吐量**、允许停顿时间**超过 1 秒**；**3）并发收集器**：多 CPU、追求**低停顿时间**（不超过 1 秒）