

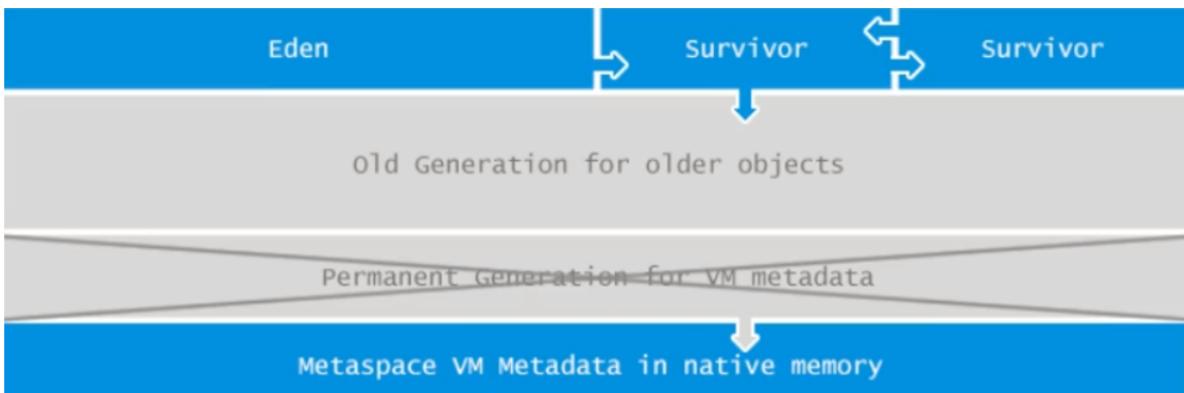
JVM核心知识点

- [Java8 JVM内存结构](#)
- [GC Roots](#)
 - [如果判断一个对象可以被回收?](#)
 - [引用计数算法](#)
 - [可达性分析算法](#)
 - [哪些对象可以作为GC Roots?](#)
- [JVM参数](#)
 - [JVM 三种类型参数](#)
 - [标配参数](#)
 - [X参数](#)
 - [XX参数](#)
 - [JVM XX参数](#)
 - [布尔类型](#)
 - [KV键值类型](#)
 - [JVM Xms/Xmx参数](#)
 - [JVM 查看参数](#)
 - [查看某个参数](#)
 - [查看所有参数](#)
 - [查看修改后的参数](#)
 - [查看常见参数](#)
 - [JVM 常用参数](#)
 - [-Xmx/-Xms](#)
 - [-Xss](#)
 - [-Xmn](#)
 - [-XX:MetaspaceSize](#)
 - [-XX:+PrintGCDetails](#)
 - [-XX:SurvivorRatio](#)
 - [-XX:NewRatio](#)
 - [-XX:MaxTenuringThreshold](#)
- [四大引用](#)
 - [强引用](#)
 - [软引用](#)
 - [弱引用](#)
 - [WeakHashMap](#)
 - [虚引用](#)
 - [引用队列](#)
- [OutOfMemoryError](#)
 - [StackOverflowError](#)
 - [OOM—Java head space](#)
 - [OOM—GC overhead limit exceeded](#)
 - [OOM—GC Direct buffer memory](#)
 - [OOM—unable to create new native thread](#)

- [OOM—Metaspace](#)
- [JVM垃圾收集器](#)
 - [四大垃圾收集算法](#)
 - [标记整理](#)
 - [标记清除](#)
 - [复制算法](#)
 - [分代收集算法](#)
 - [四种垃圾收集器](#)
 - [串行收集器Serial](#)
 - [并行收集器Parallel](#)
 - [并发收集器CMS](#)
 - [G1收集器](#)
 - [默认垃圾收集器](#)
 - [默认收集器有哪些?](#)
 - [查看默认垃圾修改器](#)
 - [七大垃圾收集器](#)
 - [体系结构](#)
 - [Serial收集器](#)
 - [ParNew收集器](#)
 - [Parallel Scavenge收集器](#)
 - [SerialOld收集器](#)
 - [ParallelOld收集器](#)
 - [CMS收集器](#)
 - [过程](#)
 - [优缺点](#)
 - [G1收集器](#)
 - [特点](#)
 - [过程](#)
- [附—Linux相关指令](#)
 - [top](#)
 - [vmstat](#)
 - [pidstat](#)
 - [free](#)
 - [df](#)
 - [iostat](#)
 - [ifstat](#)
- [CPU占用过高原因定位](#)
- [JVM性能调优和监控工具](#)
 - [jps](#)
 - [jstack](#)
 - [jinfo/jstat](#)
 - [jmap](#)

Java8 JVM内存结构

基本结构与之前类似，只是Java8取消了之前的“永久代”，取而代之的是“元空间”——**Metaspace**，两者本质是一样的。“永久代”使用的是JVM的堆内存，而“元空间”是直接使用的本机物理内存。



GC Roots

如果判断一个对象可以被回收？

引用计数算法

维护一个计数器，如果有对该对象的引用，计数器+1，反之-1。无法解决循环引用的问题。

可达性分析算法

从一组名为“GC Roots”的根节点对象出发，向下遍历。那些没有被遍历到、与GC Roots形成通路的对象，会被标记为“回收”。

哪些对象可以作为GC Roots？

1. 虚拟机栈（栈帧中的局部变量）中引用的对象。
2. 本地方法栈（native）中引用的对象。
3. 方法区中常量引用的对象。
4. 方法区中类静态属性引用的对象。

JVM参数

JVM 三种类型参数

标配参数

比如 `-version`、`-help`、`-showversion` 等，几乎不会改变。

X参数

用得不多，比如 `-xint`，解释执行模式；`-xcomp`，编译模式；`-xmixed`，开启混合模式（默认）。

```
C:\Users\MaJesTySA>java -version  
java version "1.8.0_201"  
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)  
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)  
  
C:\Users\MaJesTySA>java -Xint -version  
java version "1.8.0_201"  
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)  
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, interpreted mode)  
  
C:\Users\MaJesTySA>java -Xcomp -version  
java version "1.8.0_201"  
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)  
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, compiled mode)
```

XX参数

重要，用于JVM调优。

JVM XX参数

布尔类型

公式：`-xx:+某个属性`、`-xx:-某个属性`，开启或关闭某个功能。比如`-xx:+PrintGCDetails`，开启GC详细信息。

KV键值类型

公式：`-xx:属性key=值value`。比如`-xx:Metaspace=128m`、`-xx:MaxTenuringThreshold=15`。

JVM Xms/Xmx参数

`-xms` 和 `-xmx` 十分常见，用于设置**初始堆大小**和**最大堆大小**。第一眼看上去，既不像X参数，也不像XX参数。实际上 `-xms` 等价于`-xx:InitialHeapSize`，`-xmx` 等价于`-xx:MaxHeapSize`。所以 `-xms` 和 `-xmx` 属于XX参数。

JVM 查看参数

查看某个参数

使用`jps -l`配合`jinfo -flag JVM参数 pid`。先用`jps -l`查看java进程，选择某个进程号。

```
1 17888 org.jetbrains.jps.cmdline.Launcher  
2 5360 org.jetbrains.idea.maven.server.RemoteMavenServer  
3 18052 demo3.demo3
```

`jinfo -flag PrintGCDetails 18052`可以查看18052 Java进程的`PrintGCDetails`参数信息。

```
1 -xx:-PrintGCDetails
```

查看所有参数

使用 `jps -l` 配合 `jinfo -flags pid` 可以查看所有参数。

也可以使用 `java -XX:+PrintFlagsInitial`

```
1 [Global flags]
2     intx ActiveProcessorCount          = -1
3 {product}
4     uintx AdaptiveSizeDecrementScaleFactor = 4
5 {product}
6     uintx AdaptiveSizeMajorGCDecayTimescale = 10
7 {product}
8     uintx AdaptiveSizePausePolicy      = 0
9 {product}
10 .....
11     uintx YoungPLABSize             = 4096
12 {product}
13     bool zeroTLAB                 = false
14 {product}
15     intx hashCode                  = 5
16 {product}
17
18
```

查看修改后的参数

使用 `java -XX:PrintFlagsFinal` 可以查看修改后的参数，与上面类似。只是修改过后是 `:=` 而不是 `=`。

查看常见参数

如果不想查看所有参数，可以用 `-XX:+PrintCommandLineFlags` 查看常用参数。

```
1 -XX:InitialHeapSize=132375936 -XX:MaxHeapSize=2118014976 -
  XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -
  XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -
  XX:+UseParallelGC
```

JVM 常用参数

-Xmx/-Xms

最大和初始堆大小。最大默认为物理内存的1/4，初始默认为物理内存的1/64。

-Xss

等价于 `-xx:Thresholdstacksize`。用于设置单个线程栈的大小，一般是512K-1024K。系统默认值是0，**不代表栈大小为0**。而是根据操作系统的不同，有不同的值。比如64位的Linux系统是1024K，而Windows系统依赖于虚拟内存。

-Xmn

新生代大小，一般不调。

-XX:MetaspaceSize

设置元空间大小。

-XX:+PrintGCDetails

输出GC收集信息，包含 GC 和 Full GC 信息。

-XX:SurvivorRatio

新生代中，Eden 区和两个 survivor 区的比例，默认是 8:1:1。通过 -xx:SurvivorRatio=4 改成

4:1:1

-XX:NewRatio

老生代和新年代的比列，默认是2，即老年代占2，新生代占1。如果改成 -xx:NewRatio=4，则老年代占4，新生代占1。

-XX:MaxTenuringThreshold

新生代设置进入老年代的时间，默认是新生代逃过15次GC后，进入老年代。如果改成0，那么对象不会在新生代分配，直接进入老年代。对于老年代较多的应用可以提高效率。如果将此值设值为较大值，则年轻代对象会在survivor区进行多次复制，这样可以增加对象在年轻代的生存时间，增加在年轻代被回收的概率，可以使full gc概率降低。

四大引用

以下Demo都需要设置 -xmx 和 -xms，不然系统默认很大，很难演示。

强引用

使用 new 方法创造出来的对象，默认都是强引用。GC的时候，就算内存不够，抛出 OutOfMemoryError 也不会回收对象，死了也不回收。详见[StrongReferenceDemo](#)。

软引用

需要用 Object.Reference.SoftReference 来显示创建。如果内存够，GC的时候不回收。内存不够，则回收。常用于内存敏感的应用，比如高速缓存。详见[SoftReferenceDemo](#)。

弱引用

需要用 Object.Reference.WeakReference 来显示创建。无论内存够不够，GC的时候都回收，也可以用在高速缓存上。详见[WeakReferenceDemo](#)

WeakHashMap

传统的 HashMap 就算 key==null 了，也不会回收键值对。但是如果是 WeakHashMap，一旦内存不够用时，且 key==null 时，会回收这个键值对。详见[WeakHashMapDemo](#)。

虚引用

软引用和弱引用可以通过 `get()` 方法获得对象，但是虚引用不行。虚引用形同虚设，在任何时候都可能被 GC，不能单独使用，必须配合 **引用队列（ReferenceQueue）** 来使用。设置虚引用的**唯一目的**，就是在这个对象被回收时，收到一个**通知**以便进行后续操作，有点像 spring 的后置通知。详见 [PhantomReferenceDemo](#)。（GC之后才会放入到引用队列后面）

引用队列

弱引用、虚引用被回收后，会被放到引用队列里面，通过 `poll` 方法可以得到。关于引用队列和弱、虚引用的配合使用，见[ReferenceQueueDemo](#)。

OutOfMemoryError

StackOverflowError

栈满会抛出该错误。无限递归就会导致 `StackOverflowError`，是

`java.lang.Throwable → java.lang.Error → java.lang.VirtualMachineError` 下的错误。详见 [StackOverflowErrorDemo](#)。

OOM—Java head space

栈满会抛出该错误。详见 [JavaHeapSpaceDemo](#)。

OOM—GC overhead limit exceeded

这个错误是指：GC的时候会有“Stop the World”，STW越小越好，正常情况是GC只会占到很少一部分时间。但是如果用超过98%的时间来做GC，而且收效甚微，就会被VM叫停。下例中，执行了多次 `Full GC`，但是内存回收很少，最后抛出了 `oom:GC overhead limit exceeded` 错误。详见 [GCOverheadDemo](#)。

```
1 [GC (Allocation Failure) [PSYoungGen: 2048K->496K(2560K)] 2048K-
>960K(9728K), 0.0036555 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
2 [GC (Allocation Failure) [PSYoungGen: 2544K->489K(2560K)] 3008K-
>2689K(9728K), 0.0060306 secs] [Times: user=0.08 sys=0.00, real=0.01 secs]
3 [GC (Allocation Failure) [PSYoungGen: 2537K->512K(2560K)] 4737K-
>4565K(9728K), 0.0050620 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
4 [GC (Allocation Failure) [PSYoungGen: 2560K->496K(2560K)] 6613K-
>6638K(9728K), 0.0064025 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
5
6 [Full GC (Ergonomics) [PSYoungGen: 2048K->860K(2560K)] [ParOldGen: 6264K-
>7008K(7168K)] 8312K->7869K(9728K), [Metaspace: 3223K->3223K(1056768K)],
0.1674947 secs] [Times: user=0.63 sys=0.00, real=0.17 secs]
7 [Full GC (Ergonomics) [PSYoungGen: 2048K->2006K(2560K)] [ParOldGen: 7008K-
>7008K(7168K)] 9056K->9015K(9728K), [Metaspace: 3224K->3224K(1056768K)],
0.1048666 secs] [Times: user=0.45 sys=0.00, real=0.10 secs]
8 [Full GC (Ergonomics) [PSYoungGen: 2047K->2047K(2560K)] [ParOldGen: 7082K-
>7082K(7168K)] 9130K->9130K(9728K), [Metaspace: 3313K->3313K(1056768K)],
0.0742516 secs] [Times: user=0.28 sys=0.00, real=0.07 secs]
9
10 .....
11
12 [Full GC (Ergonomics) [PSYoungGen: 2047K->2047K(2560K)] [ParOldGen: 7084K-
>7084K(7168K)] 9132K->9132K(9728K), [Metaspace: 3313K->3313K(1056768K)],
0.0738461 secs] [Times: user=0.36 sys=0.02, real=0.07 secs]
13
```

```
14 Exception in thread "main" [Full GC (Ergonomics) [PSYoungGen: 2047K->0K(2560K)] [ParOldGen: 7119K->647K(7168K)] 9167K->647K(9728K), [Metaspace: 3360K->3360K(1056768K)], 0.0129597 secs] [Times: user=0.11 sys=0.00, real=0.01 secs]
15 java.lang.OutOfMemoryError: GC overhead limit exceeded
16     at java.lang.Integer.toString(Integer.java:401)
17     at java.lang.String.valueOf(String.java:3099)
18     at jvm.GCOverheadDemo.main(GCOverheadDemo.java:12)
```

OOM—GC Direct buffer memory

在写 NIO 程序的时候，会用到 `ByteBuffer` 来读取和存入数据。与 Java 堆的数据不一样，`ByteBuffer` 使用 `native` 方法，直接在 **堆外分配内存**。当堆外内存（也即本地物理内存）不够时，就会抛出这个异常。详见[DirectBufferMemoryDemo](#)。

OOM—unable to create new native thread

在高并发应用场景时，如果创建超过了系统默认的最大线程数，就会抛出该异常。Linux 单个进程默认不能超过 1024 个线程。**解决方法**要么降低程序线程数，要么修改系统最大线程数 `vim /etc/security/limits.d/90-nproc.conf`。详见[UnableCreateNewThreadDemo](#)

OOM—Metaspace

元空间满了就会抛出这个异常。永久代（1.8 后被 metaspace 取代）存放了一下信息：

- 虚拟机加载的类信息
- 常量池（这里的常量池指的是运行时常量池，而非字符串常量池（jdk1.6 以后，字符串常量池放在了堆中））

```
1 class常量池、字符串常量池和运行时常量池的区别
2 https://blog.csdn.net/xiaojin21cen/article/details/105300521
3 下面有一篇文章写的是比较好的
4 http://blog.csdn.net/vegetable_bird_001/article/details/51278339
5
6 1. String s1 = new String("xyz");
7
8 考虑类加载阶段和实际执行时。
9 (1) 类加载对一个类只会进行一次。"xyz"在类加载时就已经创建并驻留了（如果该类被加载之前已经有"xyz"字符串被驻留过则不需要重复创建用于驻留的"xyz"实例）。驻留的字符串是放在全局共享的字符串常量池中的。
10 (2) 在这段代码后续被运行的时候，"xyz"字面量对应的String实例已经固定了，不会再被重复创建。所以这段代码将常量池中的对象复制一份放到heap中，并且把heap中的这个对象的引用交给s1 持有。
11 这条语句创建了2个对象。
12
13 String test = "a" + "b" + "c";
14 会创建几个字符串对象，在字符串常量池中保存几个引用么？
15
16 答案是只创建了一个对象，在常量池中也只保存一个引用。
17 看到了么，实际上在编译期间，已经将这三个字面量合成一个。这样做实际上是一种优化，避免了创建多余的字符串对象，也没有发生字符串拼接问题。
18
19 java.lang.String.intern()
20 运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用比较多的就是String类的intern()方法。
```

```

21  String的intern()方法会查找在常量池中是否存在一份equal相等的字符串，如果有则返回该字符串的
22  引用，如果没有则添加自己的字符串进入常量池。
23
24  public static void main(String[] args) {
25      String str1 = "123";
26      String str2 = "123";
27
28      String a = new String("123");
29      String b = new String("123");;
30
31      System.out.println(str1 == str2); //true
32      System.out.println(a == b); //false
33
34      System.out.println(str1 == b); //false
35      String bb=b.intern();
36      System.out.println((str1== bb)); //true
37  }
38
39  true
40  false
41  false
42  true
43
44
45  https://www.cnblogs.com/holos/p/6603379.html

```

- 静态变量（静态变量在方法区中，它引用的对象在堆里）

```

1 方法区和元空间是一回事吗？
2 严格来说，不是。首先，方法区是JVM规范的一个概念定义，并不是一个具体的实现，每一个JVM的实现都可以有各自的实现；然后，在Java官方的HotSpot 虚拟机中，Java8版本以后，是用元空间来实现的方法区；在Java8之前的版本，则是用永久代实现的方法区；也就是说，“元空间”和“方法区”，一个是HotSpot 的具体实现技术，一个是JVM规范的抽象定义；所以，并不能说“JVM的元空间是方法区”，但是可以说在Java8以后的HotSpot 中“元空间用来实现了方法区”。然后多说一句，这个元空间是使用本地内存（Native Memory）实现的，也就是说它的内存是不在虚拟机内的，所以可以理论上物理机器还有多个内存就可以分配，而不用再受限于JVM本身分配的内存了
3
4 作者：Butters
5 链接：https://www.zhihu.com/question/358312524/answer/965401488
6 来源：知乎
7 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

```

- 即时编译的代码

JVM垃圾收集器

四大垃圾收集算法

标记整理

回收前的状态								
		Yellow	Yellow	Yellow	Grey	Yellow	Yellow	Yellow
Blue	Yellow	Grey	Blue	Grey	Blue	Grey	Blue	Yellow
Grey	Yellow	Grey	Yellow	Yellow	Yellow	Grey	Blue	Yellow
Yellow	Yellow	Yellow	Blue	Yellow	Grey	Blue	Grey	Grey

解释说明								
		存活对象		未使用		可回收对象		
		Grey		Blue		Yellow		

回收后的状态								
			Blue	Blue	Blue	Blue	Blue	Blue
Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue

解释说明								
		存活对象		未使用		可回收对象		
		Grey		Blue		Yellow		

标记清除

回收前的状态								
		Yellow	Yellow	Yellow	Grey	Yellow	Yellow	Yellow
Blue	Yellow	Grey	Blue	Grey	Blue	Grey	Blue	Yellow
Grey	Yellow	Grey	Yellow	Yellow	Yellow	Grey	Blue	Yellow
Yellow	Yellow	Yellow	Blue	Yellow	Grey	Blue	Grey	Grey

解释说明								
		存活对象		未使用		可回收对象		
		Grey		Blue		Yellow		

回收后的状态								

解释说明

存活对象	未使用	可回收对象
------	-----	-------

复制算法

回收前的状态								

解释说明

存活对象	未使用	
可回收对象	保留区域	

回收后的状态							

解释说明							
		存活对象	未使用				
		可回收对象	保留区域				

分代收集算法

准确来讲，跟前面三种算法有所区别。分代收集算法就是根据对象的年代，采用上述三种算法来收集。

1. 对于新生代：每次GC都有大量对象死去，存活的很少，常采用复制算法，只需要拷贝很少的对象。
2. 对于老年代：常采用标整或者标清算法。

四种垃圾收集器

Java 8可以将垃圾收集器分为四类。

串行收集器Serial

为单线程环境设计且只使用一个线程进行GC，会暂停所有用户线程，不适用于服务器。就像去餐厅吃饭，只有一个清洁工在打扫。

并行收集器Parallel

使用多个线程并行地进行GC，会暂停所有用户线程，适用于科学计算、大数据后台，交互性不敏感的场合。多个清洁工同时在打扫。

并发收集器CMS

用户线程和GC线程同时执行（不一定是并行，交替执行），GC时不需要停顿用户线程，互联网公司多用，适用对响应时间有要求的场合。清洁工打扫的时候，也可以就餐。

G1收集器

对内存的划分与前面3种很大不同，将堆内存分割成不同的区域，然后并发地进行垃圾回收。

默认垃圾收集器

默认收集器有哪些？

有 `Serial`、`Parallel`、`ConcMarkSweep` (CMS)、`ParNew`、`ParallelOld`、`G1`。还有一个 `SerialOld`，快被淘汰了。

查看默认垃圾修改器

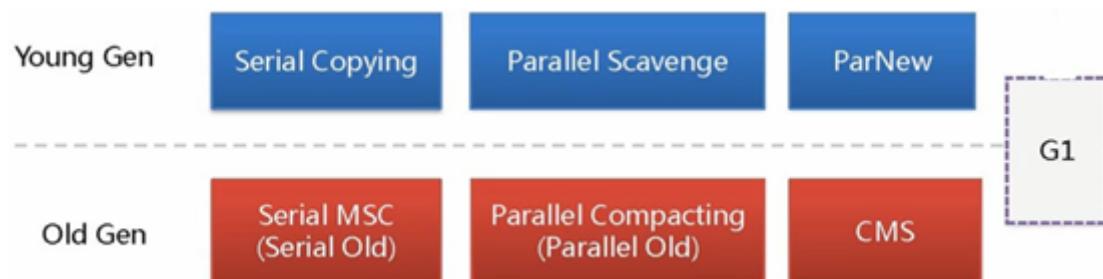
使用 `java -XX:+PrintCommandLineFlags` 即可看到，Java 8 默认使用 `-XX:+UseParallelGC`。

```
1 | -XX:InitialHeapSize=132375936 -XX:MaxHeapSize=2118014976 -
| XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -
| XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -
| XX:+UseParallelGC
```

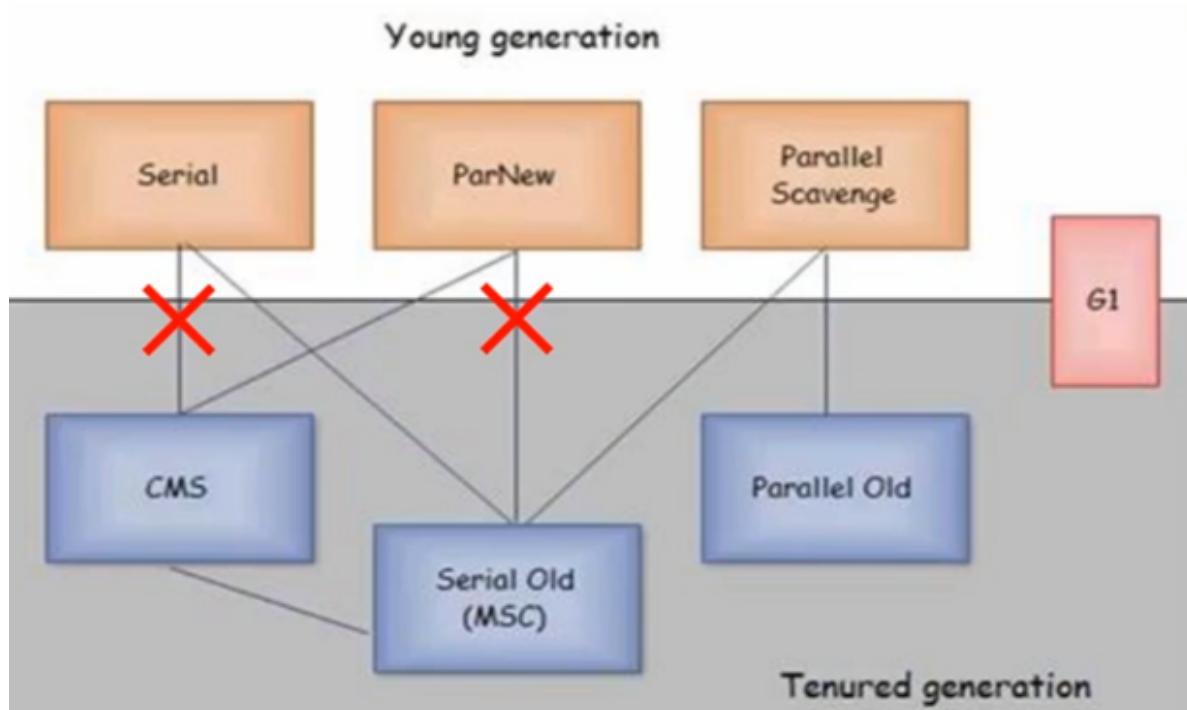
七大垃圾收集器

体系结构

`Serial`、`Parallel Scavenge`、`ParNew` 用户回收新生代；`SerialOld`、`ParallelOld`、`CMS` 用于回收老年代。而 `G1` 收集器，既可以回收新生代，也可以回收老年代。



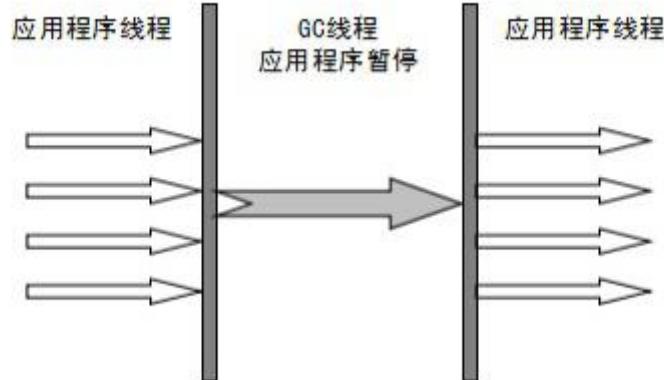
连线表示可以搭配使用，红叉表示不推荐一同使用，比如新生代用 `serial`，老年代用 `CMS`。



Serial收集器

年代最久远，是 `client VM` 模式下的默认新生代收集器，使用 **复制算法**。优点：单个线程收集，没有线程切换开销，拥有最高的单线程GC效率。缺点：收集的时候会暂停用户线程。

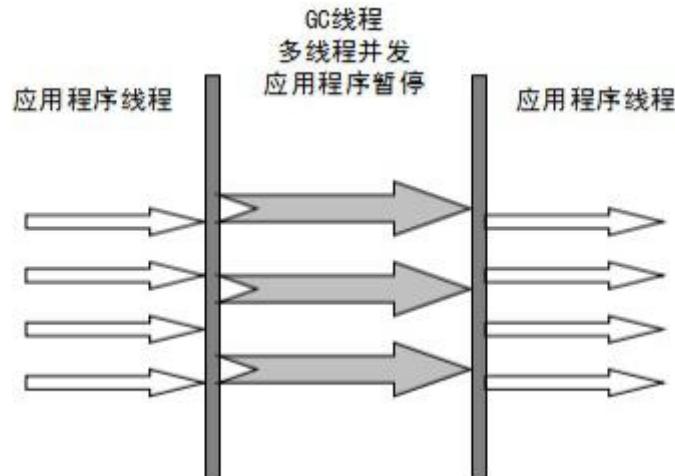
使用 `-XX:+UseSerialGC` 可以显式开启，开启后默认使用 `Serial + SerialOld` 的组合。



ParNew收集器

也就是 `Serial` 的多线程版本，GC的时候不再是一个线程，而是多个，是 `Server VM` 模式下的默认新生代收集器，采用 **复制算法**。

使用 `-XX:+UseParNewGC` 可以显式开启，开启后默认使用 `ParNew + SerialOld` 的组合。但是由于 `SerialOld` 已经过时，所以建议配合 `CMS` 使用。



Parallel Scavenge收集器

`ParNew` 收集器仅在新生代使用多线程收集，老年代默认是 `SerialOld`，所以是单线程收集。而 `Parallel Scavenge` 在新、老两代都采用多线程收集。`Parallel Scavenge` 还有一个特点就是**吞吐量优先收集器**，可以通过自适应调节，保证最大吞吐量。采用 **复制算法**。

使用 `-XX:+UseParallelGC` 可以开启，同时也会使用 `ParallelOld` 收集老年代。其它参数，比如 `-XX:ParallelGCThreads=N` 可以选择N个线程进行GC，`-XX:+UseAdaptiveSizePolicy` 使用自适应调节策略。

SerialOld收集器

`serial`的老年代版本，采用**标整算法**。JDK1.5之前跟`Parallel Scavenge`配合使用，现在已经不了，作为`CMS`的后备收集器。

ParallelOld收集器

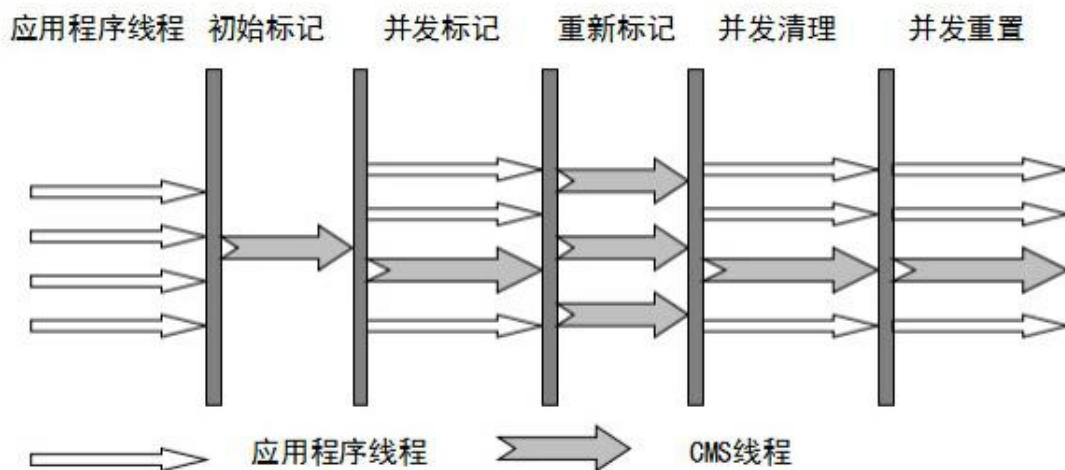
`Parallel`的老年代版本，JDK1.6之前，新生代用`Parallel`而老年代用`serialold`，只能保证新生代的吞吐量。JDK1.8后，老年代改用`Parallelold`。

使用`-XX:+UseParallelOldGC`可以开启，同时也会使用`Parallel`收集新生代。

CMS收集器

并发标记清除收集器，是一种以获得**最短GC停顿**为目标的收集器。适用在互联网或者B/S系统的服务器上，这类应用尤其重视服务器的**响应速度**，希望停顿时间最短。是G1收集器出来之前的首选收集器。使用**标清算法**。在GC的时候，会与用户线程并发执行，不会停顿用户线程。但是在**标记**的时候，仍然会**STW**（stop the world：使工作线程暂停，只允许GC线程）。

使用`-XX:+UseConcMarkSweepGC`开启。开启过后，新生代默认使用`ParNew`，同时老年代使用`serialold`作为备用。



过程

1. **初始标记**：只是标记一下GC Roots能直接关联的对象，速度很快，需要**STW**。
2. **并发标记**：主要标记过程，标记全部对象，和用户线程一起工作，不需要**STW**。
3. **重新标记**：修正在并发标记阶段出现的变动，需要**STW**。
4. **并发清除**：和用户线程一起，清除垃圾，不需要**STW**。

优缺点

优点：停顿时间少，响应速度快，用户体验好。

缺点：

1. 对CPU资源非常敏感：由于需要并发工作，多少会占用系统线程资源。
2. 无法处理浮动垃圾：由于标记垃圾的时候，用户进程仍然在运行，无法有效处理新产生的垃圾。
3. 产生内存碎片：由于使用**标清算法**，会产生内存碎片。

```

4
5  /**
6   * @author ZZYY
7   * @create 2019-04-01 12:24
8   * 1
9   * -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseSerialGC (DefNew+Tenured)
0

* 2
* -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseParNewGC (ParNew+Tenured)
*
备注情况: Java HotSpot(TM) 64-Bit Server VM warning:
Using the ParNew young collector with the Serial old collector is deprecated
and will likely be removed in a future release
*
* 3
* -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseParallelGC (PSYoungGen+ParOldGen)
*
* 4
* 4.1
* -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseParallelOldGC (PSYoungGen+ParOldGen)
* 4.2 不加就是默认UseParallelGC
* -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags (PSYoungGen+ParOldGen)
*
* 5
* -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseConcMarkSweepGC (par new generation+ concurrent)
* 6
* -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseG1GC 后面单独讲解G1
*
* 7 (理论知道即可, 实际中java已经被优化掉了, 没有了。)
* -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseSerialOldGC

```

如何选择+小总结

组合的选择

- 单CPU或小内存，单机程序
-XX:+UseSerialGC
- 多CPU，需要最大吞吐量，如后台计算型应用
**-XX:+UseParallelGC 或者
-XX:+UseParallelOldGC**
- 多CPU，追求低停顿时间，需快速响应如互联网应用
-XX:+UseConcMarkSweepGC
-XX:+ParNewGC

参数	新生代垃圾收集器	新生代算法	老年代垃圾收集器	老年代算法
-XX:+UseSerialGC	SerialGC	复制	SerialOldGC	标整
-XX:+UseParNewGC	ParNew	复制	SerialOldGC	标整
-XX:+UseParallelGC/-XX:+UseParallelOldGC	Parallel[Scavenge]	复制	Parallel Old	标整
-XX:+UseConcMarkSweepGC	ParNew	复制	CMS + Serial Old的收集器组合 (Serial Old作为CMS出错的后备收集器)	标清
-XX:+UseG1GC	G1整体上采用标记-整理算法 局部是通过复制算法，不会产生内存碎片。			

G1收集器

G1收集器与之前垃圾收集器的一个显著区别就是——之前收集器都有三个区域，新、老两代和元空间。而G1收集器只有G1区和元空间。而G1区，不像之前的收集器，分为新、老两代，而是一个一个Region，每个Region既可能包含新生代，也可能包含老年代。

G1收集器既可以提高吞吐量，又可以减少GC时间。最重要的是**STW可控**，增加了预测机制，让用户指定停顿时间。

使用**-XX:+UseG1GC**开启，还有**-XX:G1HeapRegionSize=n**、**-XX:MaxGCPauseMillis=n**等参数可调。

特点

1. **并行和并发**: 充分利用多核、多线程CPU, 尽量缩短STW。
2. **分代收集**: 虽然还保留着新、老两代的概念, 但物理上不再隔离, 而是融合在Region中。
3. **空间整合**: G1 整体上看是标整算法, 在局部看又是复制算法, 不会产生内存碎片。
4. **可预测停顿**: 用户可以指定一个GC停顿时间, G1 收集器会尽量满足。

过程

与 CMS 类似。

1. 初始标记。
2. 并发标记。
3. 最终标记。
4. 筛选回收。

JVMGC结合SpringBoot微服务的生产部署和调参优化:

公式: java -server jvm各种参数 -jar jar包或war包名字

java -server -Xms1024 -Xmx1024G -XX:+UseG1GC -jar xxx.jar

附一Linux相关指令

top

主要查看 %CPU、%MEM, 还有load average。load average 后面的三个数字, 表示系统1分钟、5分钟、15分钟的平均负载值。如果三者平均值高于0.6, 则复杂比较高了。当然, 用 uptime 也可以查看(简约版)。

vmstat

查看进程、内存、I/O等多个系统运行状态。2表示每两秒采样一次, 3表示一共采样3次。procs 的 r 表示运行和等待CPU时间片的进程数, 原则上1核CPU不要超过2。b 是等待资源的进程数, 比如磁盘I/O、网络I/O等。

```
1 [root@ ~]# vmstat -n 2 3
2 procs -----memory----- --swap-- -----io---- -system-- -----cpu--
3   r   b   swpd   free   buff   cache   si   so     bi     bo   in    cs us sy id wa
4   2   0      0 173188 239748 1362628   0   0     0     3    17     8   0   0 99
5   0   0      0 172800 239748 1362636   0   0     0     0   194   485   1   1 99
6   1   0      0 172800 239748 1362640   0   0     0     0   192   421   1   1 99
7   0   0
```

pidstat

查看某个进程的运行信息。

free

查看内存信息。

df

查看磁盘信息。

iostat

查看磁盘I/O信息。比如有时候MySQL在查表的时候，会占用大量磁盘I/O，体现在该指令的%util字段很大。对于死循环的程序，CPU占用固然很高，但是磁盘I/O不高。

ifstat

查看网络I/O信息，需要安装。

CPU占用过高原因定位

先用 top 找到CPU占用最高的进程，然后用 ps -mp pid -o THREAD,tid,time，得到该进程里面占用最高的线程。这个线程是10进制的，将其转成16进制，然后用 jstack pid | grep tid 可以定位到具体哪一行导致了占用过高。

JVM性能调优和监控工具

jps

Java版的 ps -ef 查看所有JVM进程。

jstack

查看JVM中运行线程的状态，比较重要。可以定位CPU占用过高位置，定位死锁位置。

jinfo/jstat

jinfo 查看JVM的运行环境参数，比如默认的JVM参数等。jstat 统计信息监视工具。

jmap

JVM内存映像工具。

=====强软弱虚参考文章<https://mp.weixin.qq.com/s/ZflBpn2TBzTNv-G-zZxNg>

小心点，别被当成垃圾回收了。

原创 cxuan Java建设者 2020-04-22

收录于话题

#Java

31个



今天的我又是如此迷人

我们说的不同的引用类型其实都是逻辑上的，而对于虚拟机来说，主要体现的是对象的不同的可达性(reachable)状态和对垃圾收集(garbage collector)的影响。

初识引用

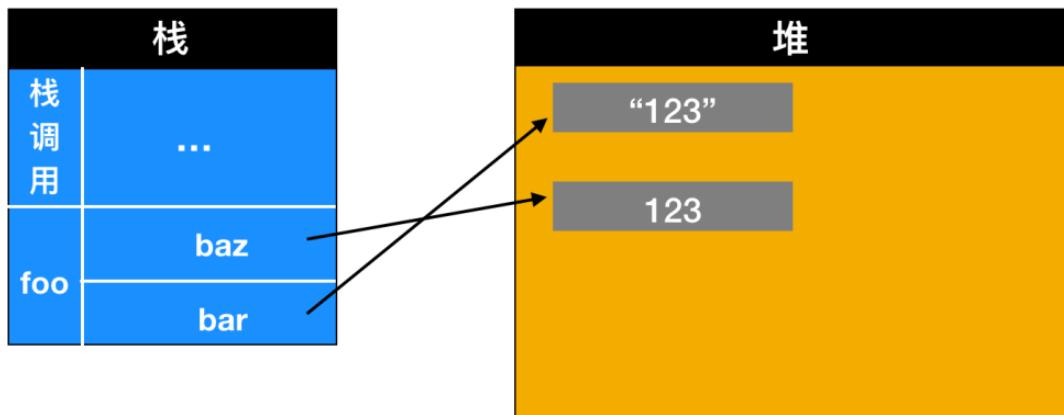
对于刚接触 Java 的 C++ 程序员而言，理解栈和堆的关系可能很不习惯。在 C++ 中，可以使用 new 操作符在堆上创建对象，或者使用自动分配在栈上创建对象。下面的 C++ 语句是合法的，但是 Java 编译器却拒绝这么写代码，会出现 syntax error 编译错误。

```
1 | Integer foo = Integer(1);
```

Java 和 C 不一样，Java 中会把对象都放在堆上，需要 new 操作符来创建对象。本地变量存储在栈中，它们持有一个指向堆中对象的引用(指针)。下面是一个 Java 方法，该方法具有一个 Integer 变量，该变量从 String 解析值

```
1 | public static void foo(String bar){  
2 |     Integer baz = new Integer(bar);  
3 | }
```

这段代码我们使用堆栈分配图可以看一下它们的关系



首先先来看一下 `foo()` 方法，这一行代码分配了一个新的 Integer 对象，JVM 尝试在堆空间中开辟一块内存空间。如果允许分配的话，就会调用 Integer 的构造方法把 String 字符串转换为 Integer 对象。JVM 将指向该对象的指针存储在变量 `baz` 中。

上面这种情况是我们乐意看到的情况，毕竟我们不想在编写代码的时候遇到阻碍，但是这种情况是不可能出现的，当堆空间无法为 bar 和 baz 开辟内存空间时，就会出现 `OutOfMemoryError`，然后就会调用 垃圾收集器(garbage collector) 来尝试腾出内存空间。这中间涉及到一个问题，垃圾收集器会回收哪些对象？

垃圾收集器

Java 给你提供了一个 `new` 操作符来为堆中的对象开辟内存空间，但它没有提供 `delete` 操作符来释放对象空间。当 `foo()` 方法返回时，如果变量 `baz` 超过最大内存，但它所指向的对象仍然还在堆中。如果没有垃圾回收器的话，那么程序就会抛出 `OutOfMemoryError` 错误。然而 Java 不会，它会提供垃圾收集器来释放不再引用的对象。

当程序尝试创建新对象并且堆中没有足够的空间时，垃圾收集器就开始工作。当收集器访问堆时，请求线程被挂起，试图查找程序不再主动使用的对象，并回收它们的空间。如果垃圾收集器无法释放足够的内存空间，并且JVM 无法扩展堆，则会出现 `OutOfMemoryError`，你的应用程序通常在这之后崩溃。还有一种情况是 `StackOverflowError`，它出现的原因是因为线程请求的栈深度要大于虚拟机所允许的深度时出现的错误。

标记 - 清除算法

Java 能永久不衰的一个原因就是因为垃圾收集器。许多人认为 JVM 会为每个对象保留一个引用计数，当每次引用对象的时候，引用计数器的值就 + 1，当引用失效的时候，引用计数器的值就 - 1。而垃圾收集器只会回收引用计数器的值为 0 的情况。这其实是 `引用计数法(Reference Counting)` 的收集方式。但是这种方式无法解决对象之间相互引用的问题，如下

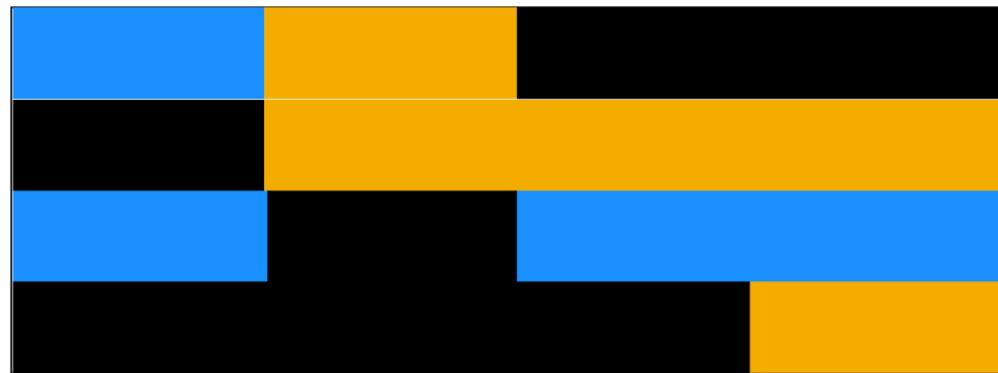
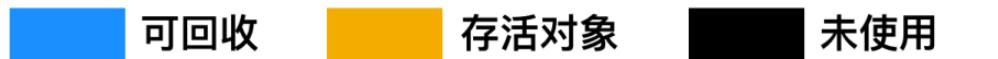
```
1 class A{
2     public B b;
3
4 }
5 class B{
6     public A a;
7 }
8 public class Main{
9     public static void main(String[] args){
10         A a = new A();
11         B b = new B();
12         a.b=b;
13         b.a=a;
14     }
15 }
```

然而实际上，JVM 使用一种叫做 `标记-清除(Mark-Sweep)` 的算法，标记清除垃圾回收背后的想法很简单：程序无法到达的每个对象都是垃圾，可以进行回收。

标记-清除收集具有如下几个阶段

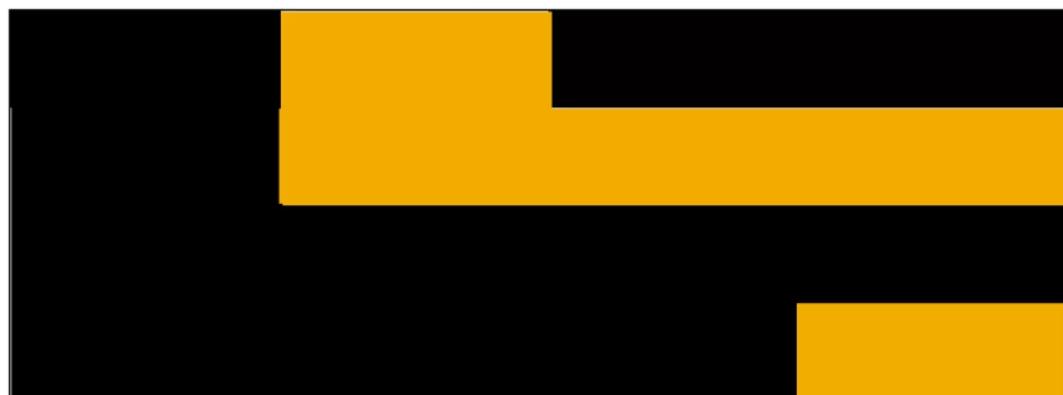
- 阶段一：标记

垃圾收集器会从 `根(root)` 引用开始，标记它到达的所有对象。如果用老师给学生判断卷子来比喻，这就相当于是给试卷上的全部答案判断正确还是错误的过程。



- 阶段二：清理

在第一阶段中所有可回收的内容都能够被垃圾收集器进行回收。如果一个对象被判定为是可以回收的对象，那么这个对象就被放在一个 `finalization queue`(回收队列) 中，并在稍后会由一个虚拟机自动建立的、低优先级的 `finalizer` 线程去执行它。



- 阶段三：整理 (可选)

一些收集器有第三个步骤，整理。在这个步骤中，GC 将对象移动到垃圾收集器回收完对象后所留下的自由空间中。这么做可以防止堆碎片化，防止大对象在堆中由于堆空间的不连续性而无法分配的情况。



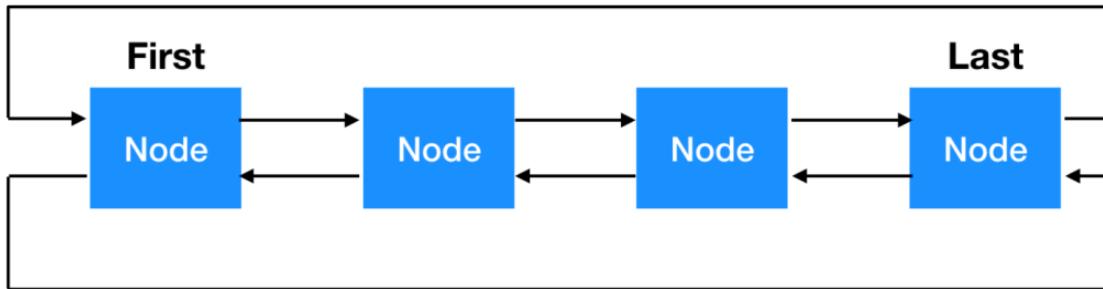
所以上面的过程中就涉及到一个 `GC Roots` 来判断是否存在需要回收的对象。这个算法的基本思想就是通过一系列的 `GC Roots` 作为起始点，从这些节点向下搜索，搜索所走过的路径称为 `引用链 (Reference Chain)`，当一个对象到 `GC Roots` 之间没有任何引用链相连的话，则证明此对象不可用。引用链上的任何一个能够被访问的对象都是 `强引用` 对象，垃圾收集器不会回收强引用对象。

因此，返回到 `foo()` 方法中，仅在执行方法时，参数 `bar` 和局部变量 `baz` 才是强引用。一旦方法执行完成，它们都超过了作用域的时候，它们引用的对象都会进行垃圾回收。

下面来考虑一个例子

```
1 | LinkedList foo = new LinkedList();
2 | foo.add(new Integer(111));
```

变量 `foo` 是一个强引用，它指向一个 `LinkedList` 对象。`LinkedList` (JDK.18) 是一个链表的数据结构，每一个元素都会指向前驱元素，每个元素都有其后继元素。



当我们调用 `add()` 方法时，我们会增加一个新的链表元素，并且该链表元素指向值为 111 的 `Integer` 实例。这是一连串的强引用，也就是说，这个 `Integer` 的实例不符合垃圾收集条件。一旦 `foo` 对象超出了程序运行的作用域，`LinkedList` 和其中的引用内容都可以进行收集，收集的前提是没有强引用关系。

Finalizers

C++ 允许对象定义析构函数方法：当对象超出作用范围或被明确删除时，会调用析构函数来清理使用的资源。对于大多数对象来说，析构函数能够释放使用 `new` 或者 `malloc` 函数分配的内存。在 Java 中，垃圾收集器会为你自动清除对象，分配内存，因此不需要显式析构函数即可执行此操作。这也是 Java 和 C++ 的一大区别。

然而，内存并不是唯一需要被释放的资源。考虑 `FileOutputStream`：当你创建此对象的实例时，它从操作系统分配文件句柄。如果你让流的引用在关闭前超过了其作用范围，该文件句柄会怎么样？实际上，每个流都会有一个 `finalizer` 方法，这个方法是垃圾回收器在回收之前由 JVM 调用的方法。对于 `FileOutputStream` 来说，`finalizer` 方法会关闭流，释放文件句柄给操作系统，然后清除缓冲区，确保数据能够写入磁盘。

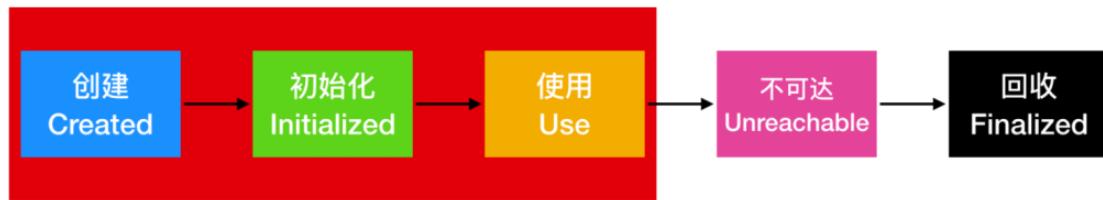
任何对象都具有 `finalizer` 方法，你要做的就是声明 `finalize()` 方法。如下

```
1 | protected void finalize() throws Throwable
2 | {
3 |     // 清除对象
4 | }
```

虽然 `finalizers` 的 `finalize()` 方法是一种好的清除方式，但是这种方法产生的负面影响非常大，你不应该依靠这个方法来做任何垃圾回收工作。因为 `finalize` 方法的运行开销比较大，不确定性强，无法保证各个对象的调用顺序。`finalize` 能做的任何事情，可以使用 `try-finally` 或者其他方式来做，甚至做的更好。

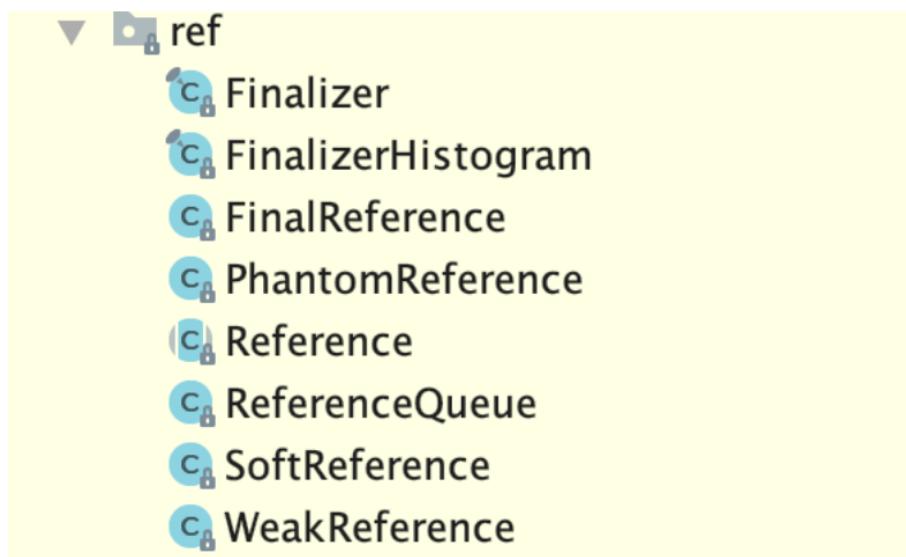
对象的生命周期

综上所述，可以通过下面的流程来对对象的生命周期做一个总结



对象被创建并初始化，对象在运行时被使用，然后离开对象的作用域，对象会变成不可达并会被垃圾回收器回收。图中用红色标明的区域表示对象处于强可达阶段。

JDK1.2 介绍了 `java.lang.ref` 包，对象的生命周期有四个阶段：□强可达□(`Strongly Reachable`)、软可达(`Soft Reachable`)、弱可达(`Weak Reachable`)、幻象可达(`Phantom Reachable`)。



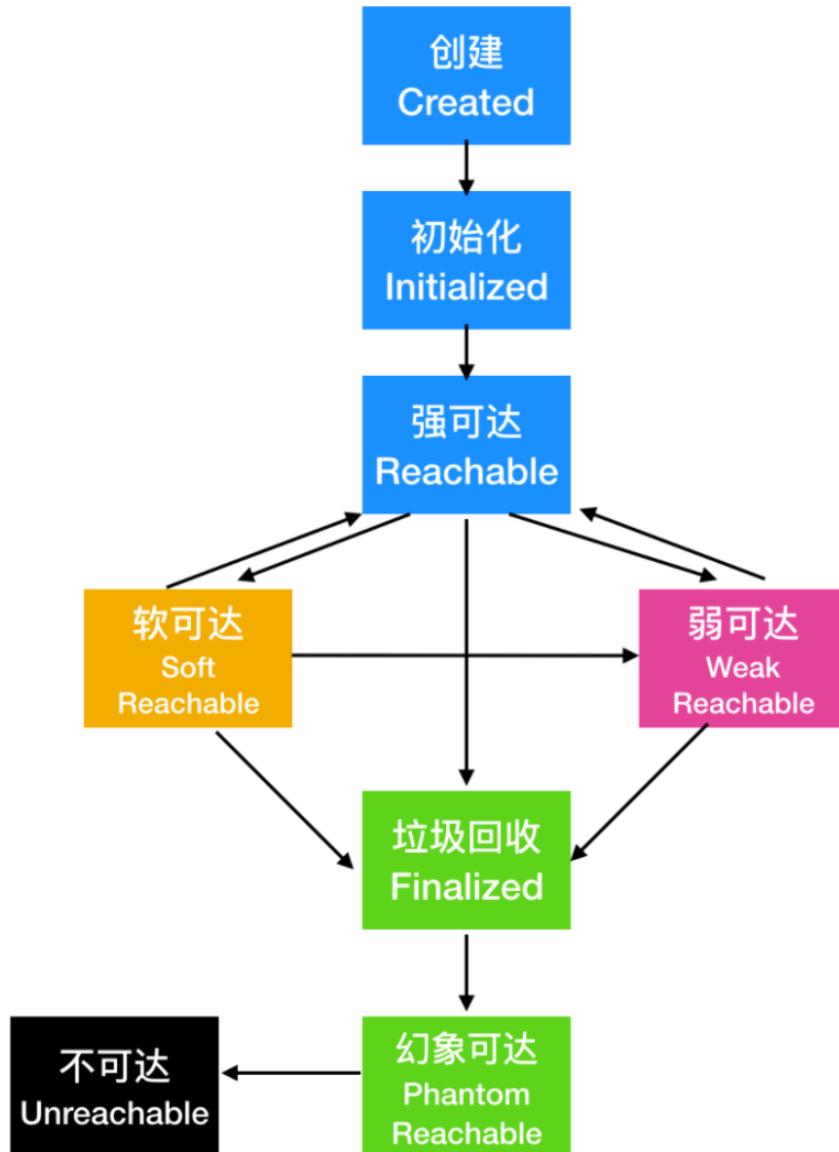
如果只讨论符合垃圾回收条件的对象，那么只有三种：软可达、弱可达和幻象可达。

- 软可达：软可达就是□我们只能通过软引用□才能访问的状态，软可达的对象是由 `SoftReference` 引用的对象，并且没有强引用的对象。软引用是用来描述一些还有用但是非必须的对象。垃圾收集器会尽可能长时间的保留软引用的对象，但是会在发生 `OutOfMemoryError` 之前，回收软引用的对象。如果回收完软引用的对象，内存还是不够分配的话，就会直接抛出 `OutOfMemoryError`。
- 弱可达：弱可达的对象是 `WeakReference` 引用的对象。垃圾收集器可以随时收集弱引用的对象，不会尝试保留软引用的对象。
- 幻象可达：幻象可达是由 `PhantomReference` 引用的对象，幻象可达就是没有强、软、弱引用进行关联，并且已经被 `finalize` 过了，只有幻象引用指向这个对象的时候。

除此之外，还有强可达和不可达的两种可达性判断条件

- 强可达：就是一个对象刚被创建、初始化、使用中的对象都是处于强可达的状态
- 不可达(unreachable)：处于不可达的对象就意味着对象可以被清除了。

下面是一个不同可达性状态的转换图



判断可达性条件，也是 JVM 垃圾收集器决定如何处理对象的一部分考虑因素。

所有的对象可达性引用都是 `java.lang.ref.Reference` 的子类，它里面有一个 `get()` 方法，返回引用对象。如果已通过程序或垃圾收集器清除了此引用对象，则此方法返回 `null`。也就是说，除了幻象引用外，软引用和弱引用都是可以得到对象的。而且这些对象可以人为 拯救，变为强引用，例如把 `this` 关键字赋值给对象，只要重新和引用链上的任意一个对象建立关联即可。

ReferenceQueue

引用队列 又称为 `ReferenceQueue`，它位于 `java.lang.ref` 包下。我们在构建各种引用（软引用，弱引用，幻象引用）并关联到响应对象时，可以选择是否需要关联引用队列。JVM 会在特定的时机将引用入队到队列中，程序可以通过判断引用队列中是否已经加入引用，来了解被引用的对象是否被GC回收。

Reference

`java.lang.ref.Reference` 为软 (soft) 引用、弱 (weak) 引用、虚 (phantom) 引用的父类。因为 `Reference` 对象和垃圾回收密切配合实现，该类可能不能被直接子类化。

文章参考：

<https://www.jianshu.com/p/f86d3a43eec5>

《深入理解Java虚拟机》第二版

<http://www.kdgregory.com/index.php?page=java.refobj>