浅谈JVM与垃圾回收


SH的全栈笔记 关注
2020.07.03 09:15:42 字数 5,833 阅读 91

写在前面

简单的介绍一下JVM（Java Virtual Machine）吧，它也叫Java虚拟机。虽然它叫虚拟机，但是实际上不是我们所理解的虚拟机，它更像操作系统中的一个进程。JVM屏蔽了各个操作系统底层的相关的东西，Java程序只需要生成对应的字节码文件，然后由JVM来负责解释运行。

介绍几个容易混淆的概念，JDK（Java Development Kit) 可以算是整个Java的核心，其中有编译、调试的工具包和基础类库，它也包含了JRE。

JRE（Java Runtime Environment），包含了JVM和基础类库。而JVM就是我们今天要聊的主角，开篇聊到，JVM负责解释运行，它会将自己的指令映射到当前不同设备的CPU指令集上，所以只需要在不同的操作系统上装不同版本的虚拟机即可。这也给了Java跨平台的能力。

JVM的发展

就跟我们用三方库一样，同样的功能有不同的实现。JVM也是一样的，第一款JVM是Sun公司的Classic VM，JDK1.2之前JVM都是采用的Classic VM，而之后，逐渐被我们都知道的HotSpot给替代，直到JDK1.4，Classic VM才完全被弃用。

HotSpot应该是目前使用最广泛的虚拟机（自信点，把应该去掉），也是OpenJDK中所带的虚拟机。但是你可能不知道，HotSpot最开始并不是由Sun公司开发，而是由一家小公司设计并实现的，而且最初也不是为Java语言设计的。Sun公司看到了这个虚拟机在JIT上的优势，于是就收购了这家公司，从而获得了HotSpot VM。

运行时内存区域

可能你经历过被灵魂拷问是什么滋味，如果线上发生了OOM（Out Of Memory），该怎么排查？如果要你来对一个JVM的运行参数进行调优，你该怎么做？

不像C++可以自己来主宰内存，同时扮演上帝和最底层劳工的角色，Java里我们把内存管理交给了JVM，如果我们不能了解其中具体的运行时内存分布以及垃圾回收的原理，那等到问题真正出现了，很可能就无从查起。这也是要深入的了解JVM的必要性。

Java在运行时会将内存分成如下几个区域进行管理，**堆**、**方法区**、**虚拟机栈**、**本地方法栈**和**程序计数器**。

堆

堆（Java Heap）是JVM所管理的内存中最大的一块了。我们平常开发中使用`new`关键字来进行实例化的对象**几乎**都会在堆中分配内存，所有线程都可以共享被分配在堆上的对象。

堆也是JVM垃圾回收的主要区域，正因为垃圾回收的分代机制，其实堆中还可以分为更细的新生代、老年代。GC这块后面会细讲。

那为什么是几乎呢？在JVM本身的规范中是规定了所有的对象都会在堆上分配内存的，但是随着JIT（Just In Time）编译器和逃逸分析技术的成熟，所有对象都在堆上分配内存就变得没有那么绝对了。

JIT编译器

不知道你有没有听说过，二八定律在我们的程序中也同样适用，那就是20%的代码占用了系统运行中80%的资源。在我们写的代码中，就可能会存在一些热点代码，频繁的被调用。除了被频繁的调用的代码，还有被执行多次的循环体也算热点代码。

那此时JIT编译器就会对这部分的代码进行优化，将它们编译成Machine Code，并做一些对应的优化。不熟悉的同学可能会说，我们的代码不都已经被编译成了字节码了吗？怎么又被编译成了Machine Code？

因为字节码只是一个中间状态，真正的运行是JVM在运行的时候，就跟解释型语言一样将字节码逐条的翻译成了Machine Code，这个Machine Code才是操作系统能够识别直接运行的指令。而JIT就会把编译好的热点代码所对应的Machine Code保存下来， 下载再调用时就省去了从字节码编译到Machine Code的过程，效率自然也就提高了。

逃逸分析

我们刚刚提到过，Java中几乎所有的对象都在堆上分配空间，堆中的内存空间是所有线程共享的，所以在多线程下才需要去考虑同步的相关问题。那如果这个变量是个局部变量，只会在某个函数中被访问到呢?

这种局部变量就是**未逃逸的变量**，而这个变量如果在其他的地方也能被访问到呢？这说明这个变量逃逸出了当前的作用域。通过逃逸分析我们可以知道哪些变量没有逃逸出当前作用域，那这个对象内存就可以在栈中分配，随着调用的结束，随着线程的继续执行完成，栈空间被回收，这个局部变量分配的内存也会一起被回收。

方法区

方法区存放了被加载的Class信息、常量、静态变量和JIT编译之后的结果等数据，与堆一样，方法区也是被所有线程共享的内存区域。但与堆不同，相对于堆的GC力度，这块的垃圾回收力度可以说是小了非常多，但是仍然有针对常量的GC。

虚拟机栈

虚拟机栈是线程私有的，所以在多线程下不需要做同步的操作，是线程安全的。当每个方法执行时，就会在当前线程中虚拟机栈中创建一个栈帧，每个方法从调用到结束的过程，就对应了栈帧在虚拟机栈中的入栈、出栈的过程。那自然而然，栈帧中应该存放的就是方法的**局部变量**、**操作数栈**、**动态链接**和对应的**返回信息**。

不知道你遇到过在方法内写递归时，由于退出条件一直没有达到，导致程序陷入了无限循环，然后就会看到程序抛出了一个`StackOverflow`的错误。其所对应的栈就是上面提到的操作数栈。

当然这是在内存足够的情况下，如果内存不够，则会直接抛出`OutOfMemory`，也就是常说的OOM。

本地方法栈

本地方法栈的功能与虚拟机栈类似，区别在于虚拟机栈是服务于JVM中的Java方法，而本地方法栈则服务于Native的方法。

GC

其实堆中的区域还可以划分为新生代和老年代，再分割的细一点，可以到Eden、From Survivor、To Survivor。首先分配的对象实例会到Eden区，在新生代这块区域一般是最大的，与From Survivor的比例是8:1，当然这个比例可以通过JVM参数来改变。而且当分配的对象实体很大的时候将会直接进入到老年代。

为什么要对堆进行更加细致的内存区域划分，其实是为了让垃圾回收更加的高效。

垃圾识别机制

那JVM是如何判断哪些对象是"垃圾"需要被回收呢？我们就需要来了解一下JVM是如何来判断哪些内存需要进行回收的。

引用计数

实现的思路是，给每个对象添加一个**引用计数器**，每当有其他的对象引用了这个对象，就把引用计数器的值+1，如果一个对象的引用计数为0则说明没有对象引用它。

乍一看是没有问题的，那为什么Java并没有采取这种呢?

想象一下这个场景，一个函数中定义了两个对象O1和O2，然后O1引用了O2，O1又引用了O1，这样一来，两个对象的引用计数器都不为0，但是实际上这两个对象再也不会被访问到了。

所以我们需要另外一种方案来解决这个问题。

可达性分析

可达性分析可以理解为一棵树的遍历，根节点是一个对象，而其子节点是引用了当前对象的对象。从根节点开始做遍历，如果发现从所有根节点出发的遍历都已经完成了，但是仍然有对象没有被访问到，那么说明这些对象是不可用的，需要将内存回收掉。

这些根节点有个名字叫做GC Roots，哪些资源可以被当作GC Roots呢?

- 栈帧中的局部变量所引用的对象
- 方法区中类静态属性所引用的对象
- 方法区中常量所引用的对象
- 本地方法栈所引用的对象

我们刚刚聊过，在引用计数中，如果其引用计数器的值为0，则占用的内存会被回收掉。而在可达性分析中，如果没有某个对象没有任何引用，它也不一定会被回收掉。

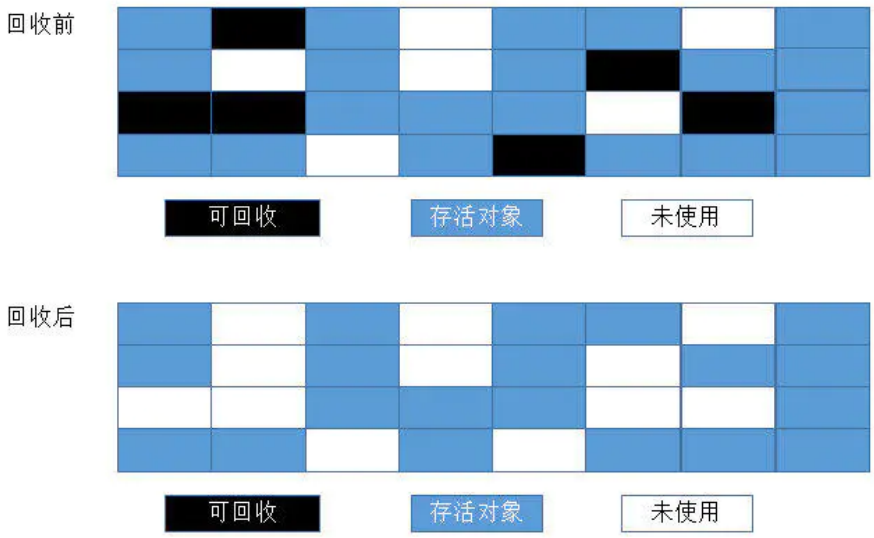
垃圾回收算法

聊完了JVM如何判断一个对象是否需要回收，接下来我们再聊一下JVM是如何进行回收的。

标记-清除

顾名思义，其过程分为两个阶段，分别是**标记**和**清除**。首先标记出所有需要回收的对象，然后统一对标记的对象进行回收。这个算法的十分的局限，首先标记和清除的两个过程效率都不高，而且这样的清理方式会产生大量的内存碎片，什么意思呢?

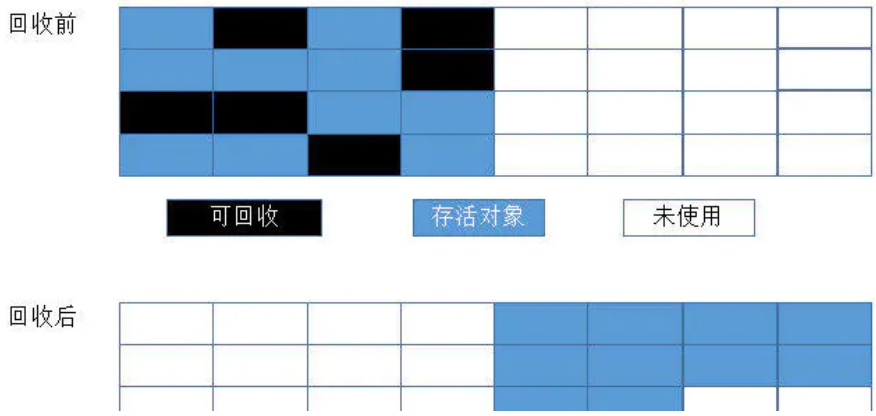
就是虽然总体看起来还有足够的剩余内存空间，但是他们都是以一块很小的内存分散在各个地方。如果此时需要为一个大对象申请空间，即使总体上的内存空间足够，但是JVM无法找到一块这么大的连续内存空间，就会导致触发一次GC。



标记-清除

复制

其大致的思路是，将现有的内存空间分为两半A和B，所有的新对象的内存都在A中分配，然后当A用完了之后，就开始对象存活判断，将A中还存活的对象复制到B去，然后一次性将A中的内存空间回收掉。



复制算法

这样一来就不会出现使用**标记-清除**所造成的内存碎片的问题了。但是，它仍然有自己的不足。那就是以内存空间缩小了一半为代价，而在某些情况下，这种代价其实是很高的。

堆中新生代就是采用的复制算法。刚刚提到过，新生代被分为了Eden、From Survivor、To Survivor，由于几乎所有的新对象都会在这里分配内存，所以Eden区比Survivor区要大很多。因此Eden区和Survivor区就不需要按照复制算法默认的1:1的来分配内存。

在HotSpot中Eden和Survivor的比例默认是8:1，也就意味着只有10%的空间会被浪费掉。

看到这你可能会发现一个问题。

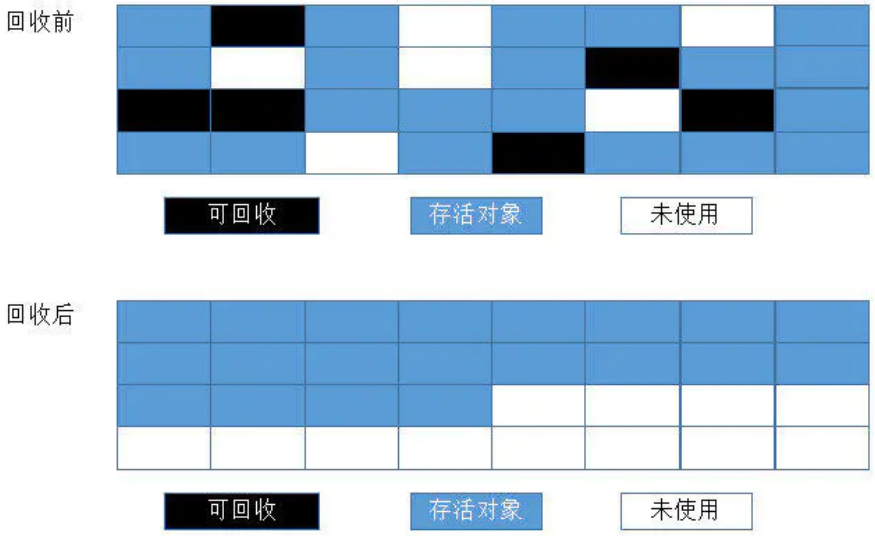
> 既然你的Eden区要比Survivor区大这么多，要是一次GC之后的存活对象的大小**大于**Survivor区的总大小该怎么处理?

的确，在新生代GC时，最坏的情况就是Eden区的所有对象都是存活的，那这个JVM会怎么处理呢？这里需要引入一个概念叫做**内存分配担保。**

当发生了上面这种情况，新生代需要老年代的内存空间来做担保，把Survivor存放不下的对象直接存进老年代中。

标记-整理

标记-整理其GC的过程与标记-清楚是一样的，只不过会让所有的存活对象往同一边移动，这样一来就不会像标记-整理那样留下大量的内存碎片。



249993-20170308200502734-920263398

分代收集

这也是当前主流虚拟机所采用的算法，其实就是针对不同的内存区域的特性，使用上面提到过的不同的算法。

例如新生代的特性是大部分的对象都是需要被回收掉的，只有少量对象会存活下来。所以新生代一般都是采用**复制算法**。

而老年代属于对象存活率都很高的内存空间，则采用**标记**-**清除**和**标记**-**整理**算法来进行垃圾回收。

垃圾收集器

新生代收集器

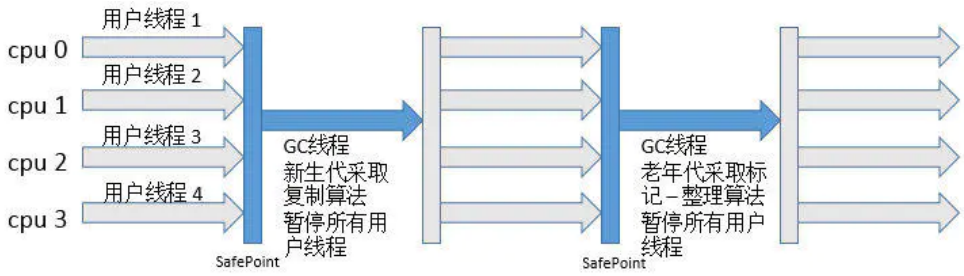
聊完了垃圾回收的算法，我们需要再了解一下GC具体是通过什么落地的， 也就是上面的算法的实际应用。

Serial

Serial采用的是**复制算法**的垃圾收集器，而且是**单线程**运作的。也就是说，当Serial进行垃圾收集时，必须要暂停其他所有线程的工作，直到垃圾收集完成，这个动作叫STW（Stop The World） 。Golang中的GC也会存在STW，在其标记阶段的准备过程中会暂停掉所有正在运行的Goroutine。

而且这个暂停动作对用户来说是不可见的，用户可能只会知道某个请求执行了很久，没有经验的话是很难跟GC挂上钩的。

但是从某些方面来看，如果你的系统就只有单核，那么Serial就不会存在线程之间的交互的开销，可以提高GC的效率。这也是为什么Serial仍然是Client模式下的默认新生代收集器。



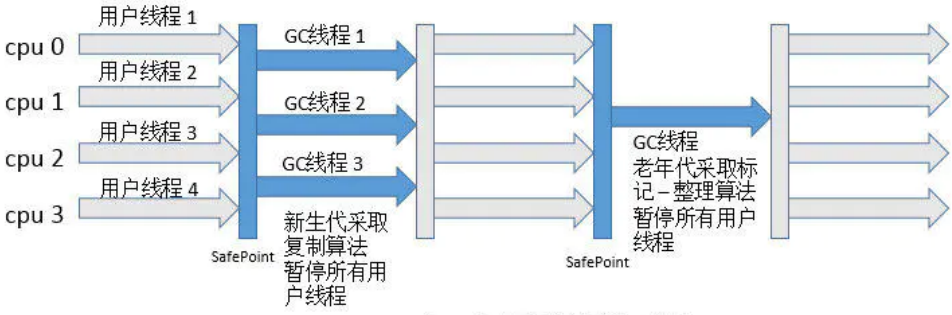
Serial/Serial Old 收集器运行示意图

249993-20170308204330750-898195038

ParNew

ParNew与Serial只有一个区别，那就是ParNew是**多线程**的，而Serial是**单线程**的。除此之外，其使用的垃圾收集算法和收集行为完全一样。

该收集器如果在单核的环境下，其性能可能会比Serial更差一些，因为单核无法发挥多线程的优势。在多核环境下，其默认的线程与CPU数量相同。



ParNew /Serial Old 收集器运行示意图

249993-20170308210151797-1882924644

Parallel Scavenge

Parallel Scavenge是一个多线程的收集器，也是在server模式下的默认垃圾收集器。上面的两种收集器关注的重点是如何减少STW的时间，而Parallel Scavenge则更加关注于系统的**吞吐量**。

例如JVM已经运行了100分钟，而GC了1分钟，那么此时系统的**吞吐量**为`(100 - 1)/100 = 99%`。

吞吐量和**短停顿时间**其侧重的点不一样，需要根据自己的实际情况来判断。

高吞吐量

GC的总时间越短，系统的吞吐量则越高。换句话说，高吞吐量则意味着，STW的时间可能会比正常的时间多一点，也就更加适合那种不存在太多交互的后台的系统，因为对实时性的要求不是很高，就可以高效率的完成任务。

短停顿时间

STW的时间短，则说明对系统的响应速度要求很高，因为要跟用户频繁的交互。因为低响应时间会带来较高的用户体验。

老年代收集器

Serial Old

Serial Old是Serial的老年代版本，使用的**标记**-**整理**算法， 其实从这看出来，新生代和老年代收集器的一个差别。

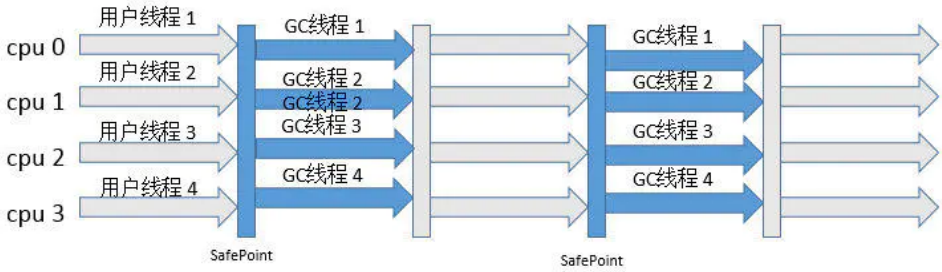
> 新生代：大部分的资源都是**需要**被回收
>
> 老年代：大部分的资源都**不需要**被回收

所以，新生代收集器**基本**都是用的**复制**算法，老年代收集器基本都是用的**标记**-**整理**算法。

Serial Old也是给Client模式下JVM使用的。

Parallel Old

Parallel Old是Parallel Scavenge的老年代版本，也是一个多线程的、采用**标记**-**整理**算法的收集器，刚刚讨论过了系统吞吐量，那么在对CPU的资源十分敏感的情况下， 可以考虑Parallel Scavenge和Parallel Old这个新生代-老年代的垃圾收集器组合。

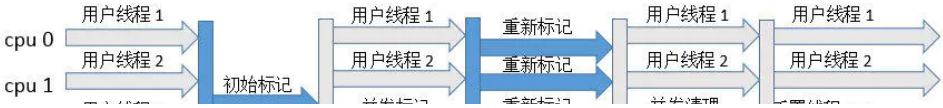


Parallel Scavenge/Parallel Old 收集器运行示意图

249993-20170309210552797-797186750

CMS

CMS全称（Concurrent Mark Sweep），使用的是**标记**-**清除**的收集算法。重点关注于最低的STW时间的收集器，如果你的应用非常注重与响应时间，那么就可以考虑使用CMS。



249993-20170312201047482-791570909

从图中可以看出其核心的步骤：

- 首先会进行**初始标记**，标记从GCRoots出发能够关联到的所有对象，此时需要STW，但是不需要很多时间
- 然后会进行**并发标记**，多线程对所有对象通过GC Roots Tracing进行**可达性分析**，这个过程较为耗时
- 完成之后会**重新标记**，由于在并发标记的过程中，程序还在正常运行，此时有些对象的状态可能已经发生了变化，所以需要STW，来进行重新标记，所用的时间大小关系为 `初始标记 < 重新标记 < 并发标记` 。
- 标记阶段完成之后，开始执行**并发清楚。**

CMS是一个优点很明显的的垃圾收集器，例如可以多线程的进行GC，且拥有较低的STW的时间。但是同样的，CMS也有很多缺点。

缺点

我们开篇也提到过，使用标记-清除算法会造成不连续的内存空间，也就是内存碎片。如果此时需要给较大的对象分配空间，会发现内存不足，重新触发一次Full GC。

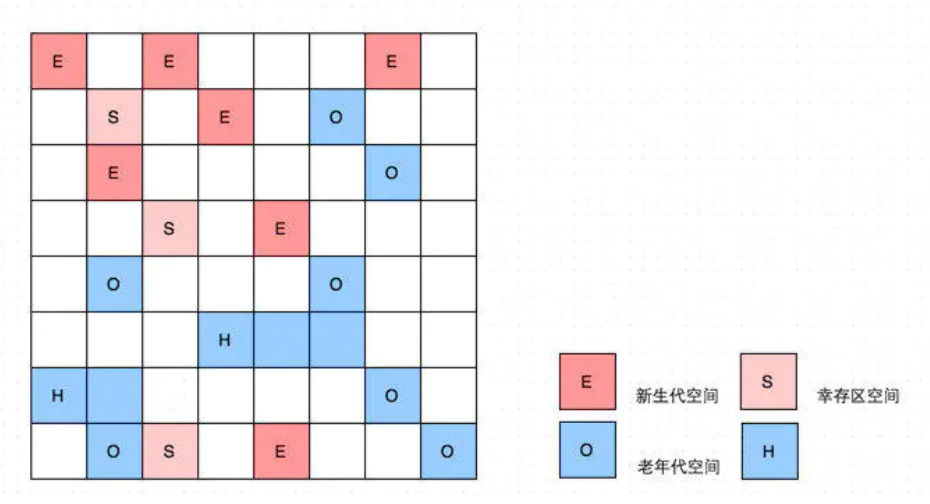
其次，由于CMS可能会比注重**吞吐量**的收集器占用更多的CPU资源，但是如果应用程序本身就已经对CPU资源很敏感了，就会导致GC时的可用CPU资源变少，GC的整个时间就会变长，那么就会导致系统的**吞吐量**降低。

G1

G1全称Garbage First，业界目前对其评价很高，JDK9中甚至提议将其设置为默认的垃圾收集器。我们前面讲过，Parallel Scavenge更加关注于吞吐量，而CMS更加关注于更短的STW时间，那么G1就是在实现高吞吐的同时，尽可能的减少STW的时间。

我们知道，上面聊过的垃圾收集器都会把连续的堆内存空间分为新生代、老年代，新生代则被划分的更加的细，有Eden和两个较小的Survivor空间，而且都是**连续的内存空间。**而G1则与众不同，它引入了新的概念，叫Region。

Region是一堆**大小相等**但是**不连续**的内存空间，同样是采用了分代的思想，但是不存在其他的收集器的物理隔离，属于新生代和老年代的region分布在堆的各个地方。



8ca16868

上面H则代表**大对象**，也叫Humongous Object。为了防止大对象的频繁拷贝，会直接的将其放入老年代。G1相比于其他的垃圾收集器有什么特点呢?

从宏观上来看，其采用的是**标记-整理**算法， 而从region到region来看，其采用的是复制算法的，所以G1在运行期间不会像CMS一样产生内存碎片。

除此之外，G1还可以通过多个CPU，来缩短STW的时间，与用户线程并发的执行。并且可以建立可预测的停顿时间模型，让使用者知道在某个时间片内，消耗在GC上的时间不得超过多少毫秒。之所以G1能够做到这点，是因为没像其余的收集器一样收集整个新生代和老年代，而是在有计划的避免对整个堆进行全区域的垃圾收集。

总结

这个图来自于参考中的博客，总结的很到位。

收集器	**串行、并行**or**并发**	**新生代/老年代**	**算法**	**目标**	**适用场景**
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

参考

- Java垃圾回收（GC）机制详解
- 深入理解JVM(3)——7种垃圾收集器