<https://www.cnblogs.com/zuolun2017/p/6121782.html>

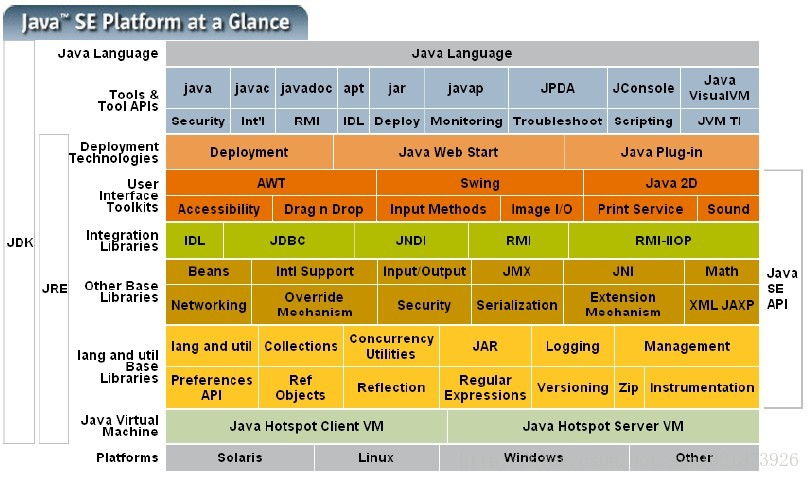
<http://www.importnew.com/23792.html>

**一、什么是JVM**

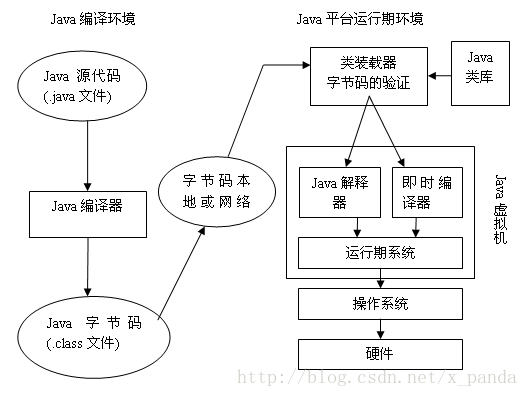
    JVM是Java Virtual Machine（Java虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

    Java语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入Java语言虚拟机后，Java语言在不同平台 上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。这就是Java的能够“一次编译，到处运行”的原因。

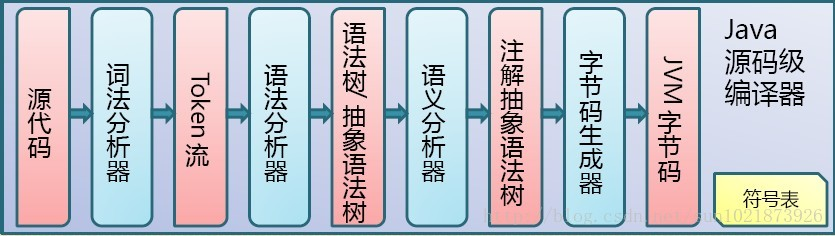
**java平台**



**java程序执行图**



**JAVA代码编译和执行过程**



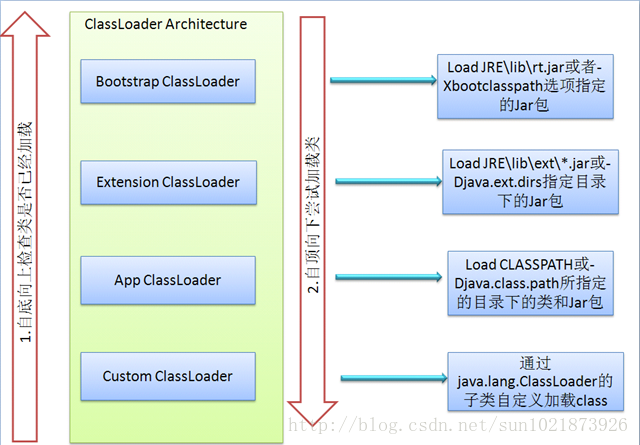
class文件由以下部分组成：

**结构信息**：包括class文件格式版本号及各部分的数量与大小的信息

**元数据**：对应于Java源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池

**方法信息**：对应Java源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息

**类加载机制**



1). Bootstrap ClassLoader

  负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

2). Extension ClassLoader

  负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

3). App ClassLoader

  负责记载classpath中指定的jar包及目录中class

4). Custom ClassLoader（自定义加载器）

  属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

**双亲委派模型**

双亲委派模型要求除了顶层的启动类加载器之外，其余的类加载器都应当有自己的父类加载器。这里的类加载器之间的父子关系一般不会以继承（Inheritance）的关系来实现，而是使用组合（Composition）关系来复用父加载器的代码。

**双亲委派模型类加载过程**：

1.  当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

2.  当前classLoader的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到bootstrp ClassLoader.

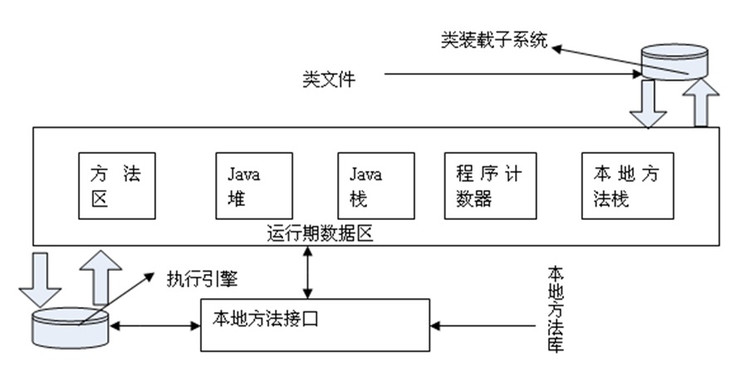
3.  当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

**作用**

**防止重复加载相同类名的类，保证系统库的安全**

**解释：**这种设计有个好处是，如果有人想替换系统级别的类：String.java。篡改它的实现，但是在这种机制下这些系统的类已经被Bootstrap classLoader加载过了，所以并不会再去加载，从一定程度上防止了危险代码的植入。

### 内存模型



**程序计数器**

程序计数器（Program Counter Register）是一块较小的内存空间，是线程私有的，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

**栈**

Java 虚拟机栈（Java Virtual Machine Stacks）是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame ①）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程

**本地方法栈**

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟栈为虚拟机执行Java 方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native 方法服务，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

**堆**

Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC 堆”（Garbage Collected Heap，幸好国内没翻译成“垃圾堆”）。如果从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以Java 堆中还可以细分为：新生代和老年代；再细致一点的有Eden 空间、From Survivor 空间、To Survivor 空间等。

**方法区**

方法区（Method Area）与Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap（非堆），目的应该是与Java 堆区分开来。

垃圾回收

#### 

**判断对象是否存活一般有两种方式：**

* 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。
* 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，不可达对象。

**GC Root**

常说的GC(Garbage Collector) roots，特指的是垃圾收集器（Garbage Collector）的对象，GC会收集那些不是GC roots且没有被GC roots引用的对象。

一个对象可以属于多个root，GC root有几下种：

* **Class** - 由系统类加载器(system class loader)加载的对象，这些类是不能够被回收的，他们可以以静态字段的方式保存持有其它对象。我们需要注意的一点就是，通过用户自定义的类加载器加载的类，除非相应的java.lang.Class实例以其它的某种（或多种）方式成为roots，否则它们并不是roots，.
* **Thread** - 活着的线程
* **Stack Local** - Java方法的local变量或参数
* **JNI Local** - JNI方法的local变量或参数
* **JNI Global** - 全局JNI引用
* **Monitor Used** - 用于同步的监控对象
* **Held by JVM** - 用于JVM特殊目的由GC保留的对象，但实际上这个与JVM的实现是有关的。可能已知的一些类型是：系统类加载器、一些JVM知道的重要的异常类、一些用于处理异常的预分配对象以及一些自定义的类加载器等。**然而，JVM并没有为这些对象提供其它的信息，因此需要去确定哪些是属于"JVM持有"的了。**

#### 

#### 垃圾回收算法

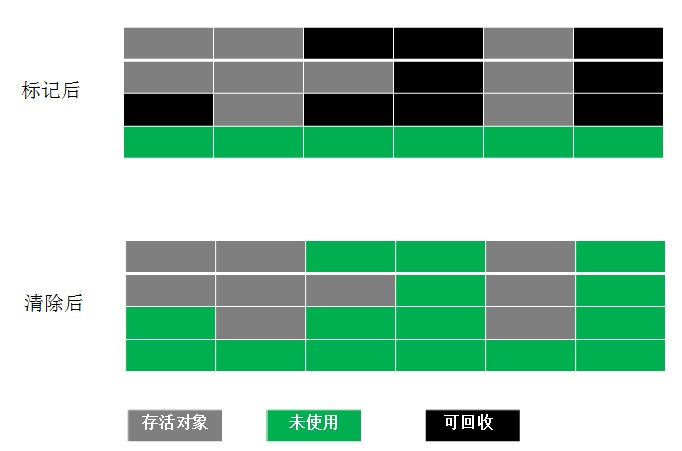
**标记-清除（Mark-Sweep）**

“标记-清除”算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

它的主要缺点有两个：

        （1）效率问题：标记和清除过程的效率都不高；

        （2）空间问题：标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，碎片过多会导致大对象无法分配到足够的连续内存，从而不得不提前触发GC，甚至Stop The World。



**复制（Copying）算法**

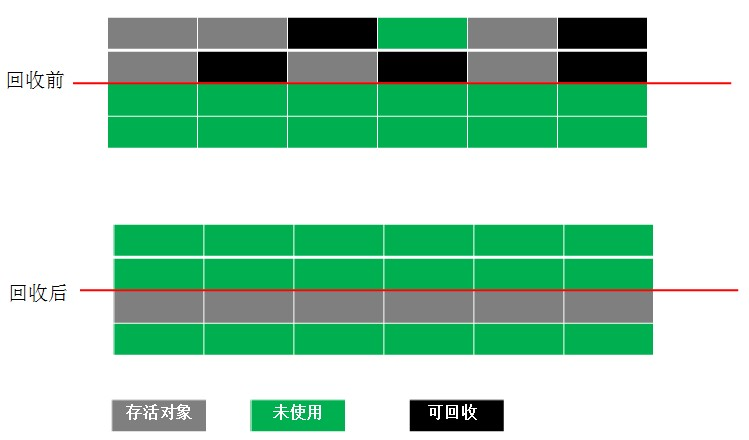
为解决效率问题，“复制”收集算法出现了。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

 它的主要缺点有两个：

        （1）效率问题：在对象存活率较高时，复制操作次数多，效率降低；

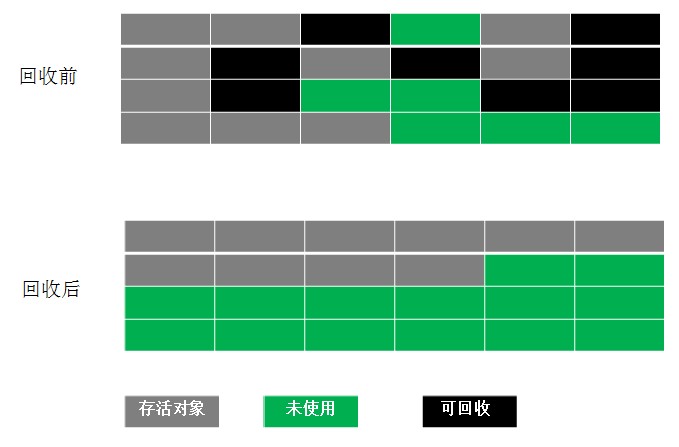
        （2）空间问题：內存缩小了一半；需要額外空间做分配担保(老年代)



**标记-整理（Mark-Compact）**

　 复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

    根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。



**分代法（Java堆采用）**

主要思想是根据对象的生命周期长短特点将其进行分块，根据每块内存区间的特点，使用不同的回收算法，从而提高垃圾回收的效率。比如Java虚拟机中的堆就采用了这种方法分成了新生代和老年代。然后对于不同的代采用不同的垃圾回收算法。新生代使用了复制算法，老年代使用了标记压缩清除算法。

  **JVM中的堆，一般分为三大部分：新生代、老年代、永久代：**



**Minor GC和Full**

（A）、Minor GC

       又称新生代GC，指发生在新生代的垃圾收集动作；

       因为Java对象大多是朝生夕灭，所以Minor GC非常频繁，一般回收速度也比较快；

（B）、Full GC

       又称Major GC或老年代GC，指发生在老年代的GC；

       出现Full GC经常会伴随至少一次的Minor GC（不是绝对，Parallel Sacvenge收集器就可以选择设置Major GC策略）；

      Major GC速度一般比Minor GC慢10倍以上；

**新生代区、老年代、永久代**

一：新生代：主要是用来存放新生的对象。一般占据堆的1/3空间。由于频繁创建对象，所以新生代会频繁触发MinorGC进行垃圾回收。

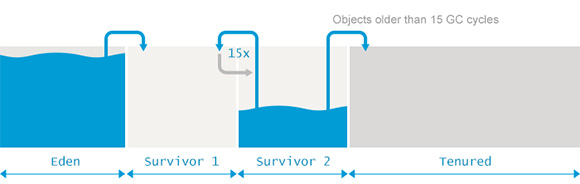
         新生代又分为 Eden区、ServivorFrom、ServivorTo三个区。

         Eden区：Java新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年代）。当Eden区内存不够的时候就会触发MinorGC，对新生代区进行一次垃圾回收。

         ServivorTo：保留了一次MinorGC过程中的幸存者。

         ServivorFrom：上一次GC的幸存者，作为这一次GC的被扫描者。

         MinorGC的过程：MinorGC采用复制算法。首先，把Eden和ServivorFrom区域中存活的对象复制到ServicorTo区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果ServicorTo不够位置了就放到老年区）；然后，清空Eden和ServicorFrom中的对象；最后，ServicorTo和ServicorFrom互换，原ServicorTo成为下一次GC时的ServicorFrom区。



    二：老年代：主要存放应用程序中生命周期长的内存对象。

    老年代的对象比较稳定，所以MajorGC不会频繁执行。在进行MajorGC前一般都先进行了一次MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次MajorGC进行垃圾回收腾出空间。

    MajorGC采用标记—清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。MajorGC的耗时比较长，因为要扫描再回收。MajorGC会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。

     当老年代也满了装不下的时候，就会抛出OOM（Out of Memory）异常。

    三：永久代

    指内存的永久保存区域，主要存放Class和Meta（元数据）的信息,Class在被加载的时候被放入永久区域. 它和和存放实例的区域不同,GC不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的Class的增多而胀满，最终抛出OOM异常。

    **在Java8中，永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。**

 　　元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制，但可以通过以下参数来指定元空间的大小：

　　-XX:MetaspaceSize，初始空间大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。

　　-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

    采用元空间而不用永久代的几点原因：

　　1、为了解决永久代的OOM问题，元数据和class对象存在永久代中，容易出现性能问题和内存溢出。

　　2、类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出（因为堆空间有限，此消彼长）。

　　3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

**并发垃圾收集和并行垃圾收集的区别(了解)**

（A）、并行（Parallel）

       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；

       如ParNew、Parallel Scavenge、Parallel Old；

（B）、并发（Concurrent）

       指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行）；

      用户程序在继续运行，而垃圾收集程序线程运行于另一个CPU上；

       如CMS、G1（也有并行）；

垃圾回收器

　1.Serial/Serial Old

　　Serial/Serial Old收集器是最基本最古老的收集器，它是一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。Serial收集器是针对新生代的收集器，采用的是Copying算法，Serial Old收集器是针对老年代的收集器，采用的是Mark-Compact算法。它的优点是实现简单高效，但是缺点是会给用户带来停顿。

　　2.ParNew

　　ParNew收集器是Serial收集器的多线程版本，使用多个线程进行垃圾收集。

　　3.Parallel Scavenge

　　Parallel Scavenge收集器是一个新生代的多线程收集器（并行收集器），它在回收期间不需要暂停其他用户线程，其采用的是Copying算法，该收集器与前两个收集器有所不同，它主要是为了达到一个可控的吞吐量。

　　4.Parallel Old

　　Parallel Old是Parallel Scavenge收集器的老年代版本（并行收集器），使用多线程和Mark-Compact算法。

　　5**.CMS**

　　CMS（Current Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是Mark-Sweep算法。（老年代收集器）

　　6.G1

　　G1收集器是当今收集器技术发展最前沿的成果，它是一款面向服务端应用的收集器，它能充分利用多CPU、多核环境。因此它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型。

详细查看该篇博文<https://www.cnblogs.com/cxxjohnson/p/8625713.html>

**重点掌握CMS**和G1

java -XX:+PrintCommandLineFlags -version

jdk1.7 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.9 默认垃圾收集器G1

JVM调优

 对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。特别要关注Full GC，因为它会对整个堆进行整理，导致Full GC一般由于以下几种情况：

1)年老代（Tenured）被写满

调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象 。

2)持久代Pemanet Generation空间不足（jdk8已经没有持久代）

增大Perm Gen空间，避免太多静态对象 ， 控制好新生代和旧生代的比例

3)System.gc()被显示调用

垃圾回收不要手动触发，尽量依靠JVM自身的机制

调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现，下面来看看各部分比例不良设置会导致什么后果

1). 新生代设置过小

  一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

2). 新生代设置过大

  一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加

  一般说来新生代占整个堆1/3比较合适

3). Survivor设置过小

  导致对象从eden直接到达旧生代，降低了在新生代的存活时间

4). Survivor设置过大

  导致eden过小，增加了GC频率

  另外，通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收

由内存管理和垃圾回收可知新生代和旧生代都有多种GC策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，JVM提供两种较为简单的GC策略的设置方式

**分析结果，判断是否需要优化**

如果各项参数设置合理，系统没有超时日志出现，GC频率不高，GC耗时不高，那么没有必要进行GC优化，如果GC时间超过1-3秒，或者频繁GC，则必须优化。

注：如果满足下面的指标，则一般不需要进行优化：

* Minor GC执行时间不到50ms；
* Minor GC执行不频繁，约10秒一次；
* Full GC执行时间不到1s；
* Full GC执行频率不算频繁，不低于10分钟1次；

**JVM调优参数参考**

1.针对JVM堆的设置，一般可以通过-Xms -Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，通常把最大、最小设置为相同的值;

2.年轻代和年老代将根据默认的比例（1：2）分配堆内存， 可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代。

比如年轻代，通过 -XX:newSize -XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize设置为同样大小。

3.年轻代和年老代设置多大才算合理

1）更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的Full GC

2）更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短；大的年老代会减少Full GC的频率

如何选择应该依赖应用程序对象生命周期的分布情况： 如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性。

在抉择时应该根 据以下两点：

（1）本着Full GC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理 。

（2）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响Full GC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间。

4.在配置较好的机器上（比如多核、大内存），可以为年老代选择并行收集算法： -XX:+UseParallelOldGC 。

5.线程堆栈的设置：每个线程默认会开启1M的堆栈，用于存放栈帧、调用参数、局部变量等，对大多数应用而言这个默认值太了，一般256K就足用。

理论上，在内存不变的情况下，减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

调优命令

**jstack**

观察jvm中当前所有线程的运行情况和线程当前状态

Jps查看当前机器运行的java程序

jstack -l 52676 > ./mainStack.txt

**查找java占用高cpu的线程**

<https://www.cnblogs.com/tiancai/p/9252780.html>

<https://blog.csdn.net/jijianshuai/article/details/79128014>

查看进程堆内存使用情况，包括使用的GC算法、堆配置参数和各代中堆内存使用情况

jmap -heap 52676

<https://www.cnblogs.com/kongzhongqijing/articles/3621163.html>

使用jmap -histo[:live] pid查看堆内存中的对象数目、大小统计直方图，如果带上live则只统计活对象

jmap -histo:live 52676

用jmap把进程内存使用情况dump到文件中

jmap -dump:format=b,file=./dump.dat 49364

dump出来的文件可以用MAT、VisualVM、jhat等工具查看

jhat -port 9998 ./dump.dat

jstat

<https://www.cnblogs.com/lizhonghua34/p/7307139.html>

jstat -gc 49364 250 4

S0C、S1C、S0U、S1U：Survivor 0/1区容量（Capacity）和使用量（Used）

EC、EU：Eden区 容量和使用量

OC、OU：年老代容量和使用量

PC、PU：永久代容量和使用量

YGC、YGT：年轻代GC次数和GC耗时

FGC、FGCT：Full GC次数和Full GC耗时

GCT：GC总耗时

JVM调参

**最后汇总一下JVM常见配置**

**堆设置**

-Xms:初始堆大小

-Xmx:最大堆大小

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5

-XX:MaxPermSize=n:设置持久代大小

**收集器设置**

-XX:+UseSerialGC:设置串行收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParalledlOldGC:设置并行年老代收集器

-XX:+UseConcMarkSweepGC:设置并发收集器

**垃圾回收统计信息**

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

**并行收集器设置**

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间

-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)

**并发收集器设置**

-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。

-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。

<https://www.jianshu.com/p/fff649585b78>