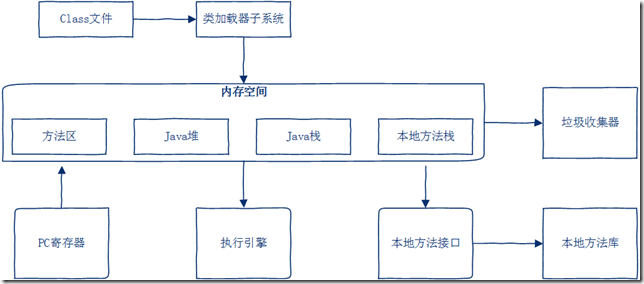
**[JAVA语言主要特点有哪些？](https://www.cnblogs.com/jay36/p/7762535.html)**

1.简单   
  
Java最初是为对家用电器进行集成控制而设计的一种语言，因此它必须简单明了。Java语言的简单性主要体现在以下三个方面：   
  
1) Java的风格类似于C++，因而C++程序员是非常熟悉的。从某种意义上讲，Java语言是C及C++语言的一个变种，因此，C++程序员可以很快就掌握Java编程技术。   
  
2) Java摒弃了C++中容易引发程序错误的地方，如指针和内存管理。   
  
3) Java提供了丰富的类库。   
  
2.面向对象   
  
面向对象可以说是Java最重要的特性。Java语言的设计完全是面向对象的，它不支持类似C语言那样的面向过程的程序设计技术。Java支持静态和动态风格的代码继承及重用。单从面向对象的特性来看，Java类似于Small Talk，但其它特性、尤其是适用于分布式计算环境的特性远远超越了Small Talk。   
  
3.分布式   
  
Java包括一个支持HTTP和FTP等基于TCP/IP协议的子库。因此，Java应用程序可凭借URL打开并访问网络上的对象，其访问方式与访问本地文件系统几乎完全相同。为分布环境尤其是Internet提供的动态内容无疑是一项非常宏伟的任务，但Java的语法特性却使我们很容易地实现这项目标。   
  
4.健壮   
  
Java致力于检查程序在编译和运行时的错误。类型检查帮助检查出许多开发早期出现的错误。Java自已操纵内存减少了内存出错的可能性。Java还实现了真数组，避免了覆盖数据的可能。这些功能特征大大缩短了开发Java应用程序的周期。Java提供Null指针检测数组边界检测异常出口字节代码校验。   
  
5.结构中立   
  
另外，为了建立Java作为网络的一个整体，Java将它的程序编译成一种结构中立的中间文件格式。只要有Java运行系统的机器都能执行这种中间代码。现在，Java运行系统有Solaris2.4(SPARC),Win32系统(Windows95和WindowsNT)等.Java源程序被编译成一种高层次的与机器无关的byte-code格式语言，这种语言被设计在虚拟机上运行，由机器相关的运行调试器实现执行。   
  
6.安全   
  
Java的安全性可从两个方面得到保证。一方面，在Java语言里，象指针和释放内存等C++功能被删除，避免了非法内存操作。另一方面，当Java用来创建浏览器时，语言功能和浏览器本身提供的功能结合起来，使它更安全。Java语言在你的机器上执行前，要经过很多次的测试。它经过代码校验，检查代码段的格式，检测指针操作，对象操作是否过分以及试图改变一个对象的类型。   
  
7.可移植的   
  
这句话一直是Java程序设计师们的精神指标，也是Java之所以能够受到程序设计师们喜爱的原因之一，最大的功臣就是JVM的技术。大多数编译器产生的目标代码只能运行在一 种CPU上(如Intel的x86系列)，即使那些能支持多种CPU的编译器也不能同时产生适合多 种CPU的目标代码。如果你需要在三种CPU( 如x86、SPARC 和MIPS)上运行同一程序, 就必须编译三次。   
  
但JAVA编译器就不同了。JAVA编译器产生的目标代码(J-Code) 是针对一种并不 存在的CPU--JAVA虚拟机(JAVA Virtual Machine)，而不是某一实际的CPU。JAVA虚拟机能掩盖不同CPU之间的差别，使J-Code能运行于任何具有JAVA虚拟机的机器上。   
  
虚拟机的概念并不AVA 所 特 有 的：加州大学几年前就提出了PASCAL虚拟机的概念；广泛用于Unix服务器的Perl脚本也是产生与机器无关的中间代码用于执行。但针对Internet应用而设计的JAVA虚拟机的特别之处在于它能产生安全的不受病毒威胁的目标代码。正是由于Internet对安全特性的特别要求才使得JVM能够迅速被人们接受。 当今主 流的操作系统如OS/2、MacOS、Windows95/NT都已经或很快提供对J-Code的支持。   
  
作为一种虚拟的CPU，JAVA 虚拟机对于源代码(Source Code) 来说是独立的。我们不仅可以用JAVA语言来生成J-Code，也可以用Ada95来生成。事实上，已经有了针对若干种源代码的J-Code 编译器，包括Basic、Lisp 和Forth。源代码一经转换成J-Code以后，JAVA虚拟机就能够执行而不区分它是由哪种源代码生成的。这样做的结果就是CPU可移植性。 将源程序编译为J-Code的好处在于可运行于各种机器上，而缺点是它不如本机代码运行的速度快。   
  
同体系结构无关的特性使得Java应用程序可以在配备了Java解释器和运行环境的任何计算机系统上运行，这成为Java应用软件便于移植的良好基础。但仅仅如此还不够。如果基本数据类型设计依赖于具体实现，也将为程序的移植带来很大不便。例如在Windows3.1中整数(Integer)为16bits，在Windows95中整数为32bits，在DECAlpha中整数为64bits，在Intel486中为32bits。通过定义独立于平台的基本数据类型及其运算，Java数据得以在任何硬件平台上保持一致。Java语言的基本数据类型及其表示方式如下：byte8-bit二进制补码short16-bit二进制补码int32-bit二进制补码long64-bit二进制补码float32-bitIEEE754浮点数double32-bitIEEE754浮点数char16-bitUnicode字符在任何Java解释器中，数据类型都是依据以上标准具体实现的。因为几乎目前使用的所有CPU都能支持以上数据类型、8～64位整数格式的补码运算和单/双精度浮点运算。Java编译器本身就是用Java语言编写的。Java运算系统的编制依据POSIX方便移植的限制，用ANSIC语言写成。Java语言规范中也没有任何"同具体实现相关"的内容。   
  
8.解释的   
  
Java解释器(运行系统)能直接运行目标代码指令。链接程序通常比编译程序所需资源少，所以程序员可以在创建源程序上花上更多的时间。   
  
9.高性能   
  
如果解释器速度不慢，Java可以在运行时直接将目标代码翻译成机器指令。Sun用直接解释器一秒钟内可调用300,000个过程。翻译目标代码的速度与C/C++的性能没什么区别。   
  
10.多线程   
  
多线程功能使得在一个程序里可同时执行多个小任务。线程－－有时也称小进程－－是一个大进程里分出来的小的独立的进程。因为Java实现的多线程技术，所以比C和C++更键壮。多线程带来的更大的好处是更好的交互性能和实时控制性能。当然实时控制性能还取决于系统本身(UNIX,Windows,Macintosh等)，在开发难易程度和性能上都比单线程要好。任何用过当前浏览器的人，都感觉为调一副图片而等待是一件很烦恼的事情。在Java里，你可用一个单线程来调一副图片，而你可以访问HTML里的其它信息而不必等它。   
  
11.动态   
  
Java的动态特性是其面向对象设计方法的发展。它允许程序动态地装入运行过程中所需要的类，这是C++语言进行面向对象程序设计所无法实现的。在C++程序设计过程中，每当在类中增加一个实例变量或一种成员函数后，引用该类的所有子类都必须重新编译，否则将导致程序崩溃。Java从如下几方面采取措来解决这个问题。Java编译器不是将对实例变量和成员函数的引用编译为数值引用，而是将符号引用信息在字节码中保存下传递给解释器，再由解释器在完成动态连接类后，将符号引用信息转换为数值偏移量。这样，一个在存储器生成的对象不在编译过程中决定，而是延迟到运行时由解释器确定的。这样，对类中的变量和方法进行更新时就不至于影响现存的代码。解释执行字节码时，这种符号信息的查找和转换过程仅在一个新的名字出现时才进行一次，随后代码便可以全速执行。在运行时确定引用的好处是可以使用已被更新的类，而不必担心会影响原有的代码。如果程序连接了网络中另一系统中的某一类，该类的所有者也可以自由地对该类进行更新，而不会使任何引用该类的程序崩溃。Java还简化了使用一个升级的或全新的协议的方法。如果你的系统运行Java程序时遇到了不知怎样处理的程序，没关系，Java能自动下载你所需要的功能程序。四.与C和C++语言的异同 Java提供了一个功能强大语言的所有功能，但几乎没有一点含混特征。C++安全性不好，但C和C++还是被大家所接受，所以Java设计成C++形式，让大家很容易学习。Java去掉了C++语言的许多功能，让Java的语言功能很精炼，并增加了一个很有用的功能，Java去掉了以下几个C和C++功能和特征：指针运算结构typedefs#define需要释放内存全局变量的定义这个功能都是很容易引起错误的地方。   
  
12. Unicode   
  
Java使用Unicode作为它的标准字符，这项特性使得Java的程序能在不同语言的平台上都能撰写和执行。简单的说，你可以把程序中的变量、类别名称使用中文来表示<注>，当你的程序移植到其它语言平台时，还是可以正常的执行。Java也是目前所有计算机语言当中，唯一天生使用Unicode的语言。

JVM学习记录：

JVM的内部结构



JVM内存空间：

JVM内存空间包含：方法区、java堆、java栈、本地方法栈。

方法区是各个线程共享的区域，存放类信息、常量、静态变量。

java堆也是线程共享的区域，我们的类的实例就放在这个区域，可以想象一个系统会产生很多实例，因此java堆的空间也是最大的。如果java堆空间不足了，程序会抛出OutOfMemoryError异常。

java栈是每个线程私有的区域，它的生命周期与线程相同，一个线程对应一个java栈，每执行一个方法就会往栈中压入一个元素，这个元素叫“栈帧”，而栈帧中包括了方法中的局部变量、用于存放中间状态值的操作栈。如果java栈空间不足了，程序会抛出StackOverflowError异常，递归如果深度很深，就会执行大量的方法，方法越多java栈的占用空间越大。

本地方法栈角色和java栈类似，只不过它是用来表示执行本地方法的，本地方法栈存放的方法调用本地方法接口，最终调用本地方法库，实现与操作系统、硬件交互的目的。

PC寄存器，类已经加载了，实例对象、方法、静态变量都去了自己该去的地方，那么问题来了，程序该怎么执行，哪个方法先执行，哪个方法后执行，这些指令执行的顺序就是PC寄存器在管，它的作用就是控制程序指令的执行顺序。

执行引擎当然就是根据PC寄存器调配的指令顺序，依次执行程序指令。

JVM配置参数分为三类参数：

1、跟踪参数

2、堆分配参数

3、栈分配参数

这三类参数分别用于跟踪监控JVM状态，分配堆内存以及分配栈内存。

跟踪参数

跟踪参数用于跟踪监控JVM，往往被开发人员用于JVM调优以及故障排查。

1、当发生GC时，打印GC简要信息

使用-XX:+PrintGC或-verbose:gc参数

这两个配置参数效果是一样的，都是在发生GC时打印出简要的信息，例如执行代码：

1: public static void main(String[] args) 2: { 3: byte[] bytes =null; 4: for(int i=0;i<100;i++){ 5: bytes = new byte[1 \* 1024 \* 1024]; 6: } 7: }

这个程序连续创建了100个1M的数组对象，使用-XX:+PrintGC或-verbose:gc参数执行该程序，即可查看到GC情况：

1: [GC (Allocation Failure) 32686K->1648K(123904K), 0.0007230 secs] 2: [GC (Allocation Failure) 34034K->1600K(123904K), 0.0009652 secs] 3: [GC (Allocation Failure) 33980K->1632K(123904K), 0.0005306 secs]

我们可以看到程序执行了3次GC（minor GC），这三次GC都是新生代的GC，因为这个程序每次创建新的数组对象，都会把新的对象赋给bytes变量，而老的对象没有任意对象引用它，老对对象会变的不可达，这些不可达的对象在新生代minor GC时候被回收掉。

32686K表示回收前，对象占用空间。1648K表示回收后，对象占用空间。123904K表示还有多少空间可用。0.0007230 secs表示这次垃圾回收花的时间。

2、打印GC的详细信息以及堆使用详细信息

使用-XX:+PrintGCDetails参数

1: [GC (Allocation Failure) [PSYoungGen: 32686K->1656K(37888K)] 32686K->1664K(123904K), 0.0342788 secs] [Times: user=0.00 sys=0.00, real=0.03 secs] 2: [GC (Allocation Failure) [PSYoungGen: 34042K->1624K(70656K)] 34050K->1632K(156672K), 0.0013466 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] 3: Heap 4: PSYoungGen total 70656K, used 43118K [0x00000000d6100000, 0x00000000dab00000, 0x0000000100000000) 5: eden space 65536K, 63% used [0x00000000d6100000,0x00000000d8985ac8,0x00000000da100000) 6: from space 5120K, 31% used [0x00000000da600000,0x00000000da796020,0x00000000dab00000) 7: to space 5120K, 0% used [0x00000000da100000,0x00000000da100000,0x00000000da600000) 8:ParOldGen total 86016K, used 8K [0x0000000082200000, 0x0000000087600000, 0x00000000d6100000) 9: object space 86016K, 0% used [0x0000000082200000,0x0000000082202000,0x0000000087600000) 10: Metaspace used 2669K, capacity 4486K, committed 4864K, reserved 1056768K 11: class space used 288K, capacity 386K, committed 512K, reserved 1048576K

我们看到除了打印GC信息之外，还显示了堆使用情况，堆分为新生代、老年代、元空间。注意这里没有永久区了，永久区在java8已经移除，原来放在永久区的常量、字符串静态变量都移到了元空间，并使用本地内存。

新生代当中又分为伊甸区（eden）和幸存区（from和to），从上面打印的内容可以看到新生代总大小为70656K，使用了43118K，细心的同学的可能会发现eden+from+to=65536K+5120K+5120K=75776 并不等于总大小70656K，这是为什么呢？这是因为新生代的垃圾回收算法是采用复制算法，简单的说就是在from和to之间来回复制（复制过程中再把不可达的对象回收掉），所以必须保证其中一个区是空的，这样才能有预留空间存放复制过来的数据，所以新生代的总大小其实等于eden+from（或to）=65536K+5120K=70656k。

3、使用外部文件记录GC的日志

还有一个非常有用的参数，它可以把GC的日志记录到外部文件中，这在生产环境进行故障排查时尤为重要，当java程序出现OOM时，总希望看到当时垃圾回收的情况，通过这个参数就可以把GC的日志记录下来，便于排查问题，当然也可以做日常JVM监控。

-Xloggc:log/gc.log

4、监控类的加载

-XX:+TraceClassLoading

使用这个参数可以监控java程序加载的类：

堆配置参数

指定最大堆，最小堆：Xmx、Xms

这两个参数是我们最熟悉最常用的参数，可以用以下代码打印出目前内存使用的情况：

1: public static void main(String[] args) 2: { 3: System.out.println("最大堆："+Runtime.getRuntime().maxMemory()/1024/1024+"M"); 4: System.out.println("空闲堆："+Runtime.getRuntime().freeMemory()/1024/1024+"M"); 5: System.out.println("总的堆："+Runtime.getRuntime().totalMemory()/1024/1024+"M"); 6: }

最大堆也就是Xmx参数指定的大小，表示java程序最大能使用多少内存大小，如果超过这个大小，那么java程序会报：out of memory

（OOM错误），空闲堆表示程序已经分配的内存大小减去已经使用的内存大小，而总的堆表示目前程序已经配置到多少内存大小，一般而言程序一启动，会按照-Xms5m先分配5M的空间，这时总的堆大小就是5M。

指定新生代内存大小:Xmn，例如我们指定-Xmx20m -Xms5m -Xmn2m -XX:+PrintGCDetails

1: 最大堆：19.5M 2: 空闲堆：4.720428466796875M 3: 总的堆：5.5M 4: Heap 5: PSYoungGen total 1536K, used 819K [0x00000000ffe00000, 0x0000000100000000, 0x0000000100000000) 6:eden space 1024K, 79% used [0x00000000ffe00000,0x00000000ffeccc80,0x00000000fff00000) 7: from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000) 8: to space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000) 9: ParOldGen total 4096K, used 0K [0x00000000fec00000, 0x00000000ff000000, 0x00000000ffe00000) 10: object space 4096K, 0% used [0x00000000fec00000,0x00000000fec00000,0x00000000ff000000) 11:Metaspace used 2723K, capacity 4486K, committed 4864K, reserved 1056768K 12: class space used 293K, capacity 386K, committed 512K, reserved 1048576K

可以看到新生代总大小为eden+from+to=1024k+512k+512k=2M，和我们设置的-Xmn相对应。

新生代（eden+from+to）和老年代（不包含永久区）的比值：-XX:NewRatio

例如我们设置参数：-Xmx20m -Xms20m -XX:NewRatio=4 -XX:+PrintGCDetails（注意这里改参数为4表示新生代和老年代比值为1:4）

1: 最大堆：19.5M 2: 空闲堆：8.665084838867188M 3: 总的堆：19.5M 4: Heap 5: PSYoungGen total 3584K, used 916K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)6: eden space 3072K, 29% used [0x00000000ffc00000,0x00000000ffce52f8,0x00000000fff00000) 7: from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000) 8: to space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000) 9: ParOldGen total 16384K, used 10240K [0x00000000fec00000, 0x00000000ffc00000, 0x00000000ffc00000) 10: object space 16384K, 62% used [0x00000000fec00000,0x00000000ff600010,0x00000000ffc00000) 11:Metaspace used 2723K, capacity 4486K, committed 4864K, reserved 1056768K 12: class space used 293K, capacity 386K, committed 512K, reserved 1048576K

可以看到新生代：eden+from+to=3072+512+512=4096k，老年代：16384k，新生代：老年代=4096k：16384k=1:4 和-XX:NewRatio=4吻合。

Eden区与Survivor区（from、to）的大小比值：-XX:SurvivorRatio（如设置为8,则两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10）

例如设置参数-Xmx20m -Xms20m -Xmn8m -XX:SurvivorRatio=6 -XX:+PrintGCDetails

这个参数设置了新生代内存大小为8m，并设置Survivor区与一个Eden区的比值为2:6，来看看打印信息：

1: 最大堆：19.0M 2: 空闲堆：8.104576110839844M 3: 总的堆：19.0M 4: Heap 5: PSYoungGen total 7168K, used 1040K [0x00000000ff800000, 0x0000000100000000, 0x0000000100000000)6: eden space 6144K, 16% used [0x00000000ff800000,0x00000000ff904090,0x00000000ffe00000) 7: from space 1024K, 0% used [0x00000000fff00000,0x00000000fff00000,0x0000000100000000) 8: to space 1024K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x00000000fff00000) 9: ParOldGen total 12288K, used 10240K [0x00000000fec00000, 0x00000000ff800000, 0x00000000ff800000) 10: object space 12288K, 83% used [0x00000000fec00000,0x00000000ff600010,0x00000000ff800000) 11:Metaspace used 2723K, capacity 4486K, committed 4864K, reserved 1056768K 12: class space used 293K, capacity 386K, committed 512K, reserved 1048576K

Survivor区=from+to=2048，Eden区=6144K，Survivor区：Eden区=2：6，和-XX:SurvivorRatio=6吻合。

其他还有-XX:+HeapDumpOnOutOfMemoryError、-XX:+HeapDumpPath这两个参数可以实现在发生OOM异常时把堆栈信息打印到外部文件。

堆分配参数的总结

根据实际事情调整新生代和幸存代的大小

官方推荐新生代占堆的3/8

幸存代占新生代的1/10

在OOM时，记得Dump出堆，确保可以排查现场问题

永久区分配参数

-XX:PermSize -XX:MaxPermSize

用于设置永久区的初始空间和最大空间，他们表示一个系统可以容纳多少个类型，一般空间比较小。在java1.8以后，永久区被移到了元数据区，使用本地内存，所以这两个参数也不建议再使用。

栈大小分配参数

栈大小参数为-Xss，通常只有几百k，决定了函数调用的深度，每个线程都有自己独立的栈空间。如果函数调用太深，超过了栈的大小，则会抛出java.lang.StackOverflowError，通常我们遇到这种错误，不是去调整-Xss参数，而是应该去调查函数调用太深的原理，是否使用递归，能不能保证递归出口等。

小结

本文讲解了JVM常用的参数，涉及跟踪、堆、永久区、栈的分配，其中最重要最常用的是跟踪、堆的分配参数，他们也和调优、故障排查息息相关。

在介绍垃圾回收算法之前，我们需要先了解一个词“stop the world”，stop the world会在执行某一个垃圾回收算法的时候产生，JVM为了执行垃圾回收，会暂时java应用程序的执行，等垃圾回收完成后，再继续运行。如果你使用JMeter测试过java程序，你可能会发现在测试过程中，java程序有不规则的停顿现象，其实这就是“stop the world”，停顿的时候JVM是在做垃圾回收。所以尽可能减少stop the world的时间，就是我们优化JVM的主要目标。接下来我们看一下目前有哪些常见垃圾回收的算法。

《深入理解Java虚拟机》一书中是这么说的，一下几种对象可以作为GC Root：

虚拟机栈中的引用对象  
方法区中类静态属性引用的对象  
方法区中常量引用对象  
本地方法栈中JNI引用对象

标记清除算法，它是很多垃圾回收算法的基础，简单来说有两个步骤：标记、清除。

标记：遍历所有的GC Roots，并将从GC Roots可达的对象设置为存活对象；

清除：遍历堆中的所有对象，将没有被标记可达的对象清除；

这里需要注意的是标记清除算法执行过程中，会产生“stop the world”，让java程序暂停等待以保证在标记清除的过程中，不会有新的对象产生。为什么必须暂停java程序呢？举个例子，如果在标记过程完成后，又新产生了一个对象，而该对象已经错过了标记期，那么在接下来的清除流程中，这个新产生的对象因为未被标记，所以将被视为不可达对象而被清除，这样程序就会出错，因此标记清除算法在执行时，java程序将被暂停，产生“stop the world”。

接下来我们总结一下标记清除算法：

1、因为涉及大量的内存遍历工作，所以执行性能较低，这也会导致“stop the world”时间较长，java程序吞吐量降低；

2、我们注意到对象被清除之后，被清除的对象留下内存的空缺位置，造成内存不连续，空间浪费。

标记压缩算法就是在标记清除算法的基础上，增加了压缩过程。

在进行完标记清除之后，对内存空间进行压缩，节省内存空间，解决了标记清除算法内存不连续的问题。

注意标记压缩算法也会产生“stop the world”，不能和java程序并发执行。在压缩过程中一些对象内存地址会发生改变，java程序只能等待压缩完成后才能继续。

复制算法简单来说就是把内存一分为二，但只使用其中一份，在垃圾回收时，将正在使用的那份内存中存活的对象复制到另一份空白的内存中，最后将正在使用的内存空间的对象清除，完成垃圾回收。

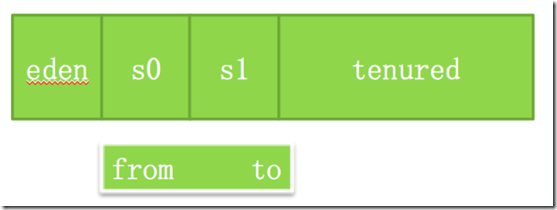
 复制算法相对标记压缩算法来说更简洁高效，但它的缺点也显而易见，它不适合用于存活对象多的情况，因为那样需要复制的对象很多，复制性能较差，所以复制算法往往用于内存空间中新生代的垃圾回收，因为新生代中存活对象较少，复制成本较低。它另外一个缺点是内存空间占用成本高，因为它基于两份内存空间做对象复制，在非垃圾回收的周期内只用到了一份内存空间，内存利用率较低。

垃圾回收器

不同的算法各有各的优缺点，在JVM中并不是单纯的使用某一种算法进行垃圾回收，而是将不同的垃圾回收算法包装在不同的垃圾回收器当中，用户可以根据自身的需求，使用不同的垃圾回收器，以便让自己的java程序性能到达最佳。

在介绍垃圾回收器之前，我们先回顾一下java堆的结构。

堆内存回顾

[](http://images2017.cnblogs.com/blog/352511/201708/352511-20170821094636277-1803221359.png)

java堆内存结构包括：新生代和老年代，其中新生代由一个伊甸区和2个幸存区组成，2个幸存区是大小相同，完全对称的，没有任何差别。我们把它们称为S0区和S1区，也可以称为from区和to区。

JVM的垃圾回收主要是针对以上堆空间的垃圾回收，当然其实也会针对元数据区（永久区）进行垃圾回收，主要介绍对堆空间的垃圾回收。

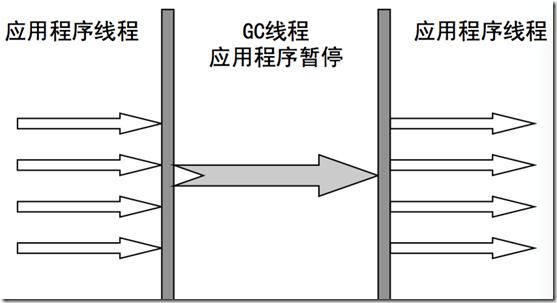
串行收集器

顾名思义，串行收集器就是使用单线程进行垃圾回收。对新生代的回收使用复制算法，对老年代使用标记压缩算法，这也和我们上一篇介绍的算法优势是相吻合的。

串行收集器是最古老最稳定的收集器，尽管它是串行回收，回收时间较长，但其稳定性是优于其他回收器的，综合来说是一个不错的选择。要使用串行收集器，可以在启动配置时加上以下参数：

-XX:+UseSerialGC

串行回收器的执行流程如下所示：

[](http://images2017.cnblogs.com/blog/352511/201708/352511-20170821094637230-1672876723.png)

执行垃圾回收时，应用程序线程暂停，GC线程开始（开始垃圾回收），垃圾回收完成后，应用程序线程继续执行。注意：在GC线程运行过程中使用单线程进行串行回收。

并行回收器

并行回收器你可能已经猜到就是使用多线程并行回收，不过这里需要注意的是，针对新生代和老年代，是否都使用并行，有不同的回收器选择：

1、 ParNew回收器

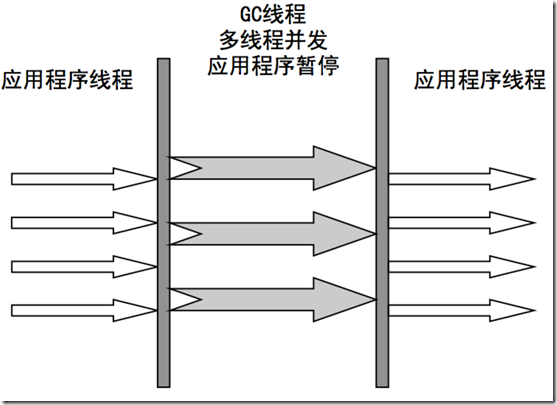
这个回收器只针对新生代进行并发回收，老年代依然使用串行回收。回收算法依然和串行回收一样，新生代使用复制算法，老年代使用标记压缩算法。在多核条件下，它的性能显然优于串行回收器，如果要使用这种回收器，可以在启动参数中配置：

-XX:+UseParNewGC

如果要进一步指定并发的线程数，可以配置一下参数：

-XX:ParallelGCThreads

ParNew回收器的流程如下图所示：

[](http://images2017.cnblogs.com/blog/352511/201708/352511-20170821094638668-1000778792.png)

在进行垃圾回收时应用程序线程依然被暂停，GC线程并行开始执行垃圾回收，垃圾回收完成后，应用程序线程继续执行。

2、 Parallel回收器

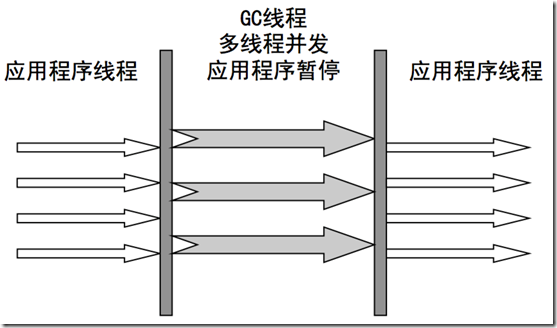
依然是并行回收器，但这种回收器有两种配置，一种类似于ParNEW：新生代使用并行回收、老年代使用串行回收。它与ParNew的不同在于它在设计目标上更重视吞吐量，可以认为在相同的条件下它比ParNew更优。要使用这种回收器可以在启动程序中配置：

-XX:+UseParallelGC

Parallel回收器另外一种配置则不同于ParNew，对于新生代和老年代均适应并行回收，要使用这种回收器可以在启动程序中配置：

XX:+UseParallelOldGC

Parallel回收器的流程和ParNew的流程是一致的：

[](http://images2017.cnblogs.com/blog/352511/201708/352511-20170821094639808-1638675726.png)

在进行回收时，应用程序暂停，GC使用多线程并发回收，回收完成后应用程序线程继续运行。

CMS回收器

CMS回收器: Concurrent Mark Sweep，并发标记清除。注意这里注意两个词：并发、标记清除。

并发表示它可以与应用程序并发执行、交替执行；标记清除表示这种回收器不是使用的是标记压缩算法，这和前面介绍的串行回收器和并发回收器有所不同。需要注意的是CMS回收器是一种针对老年代的回收器，不对新生代产生作用。这种回收器优点在于减少了应用程序停顿的时间，因为它不需要应用程序完成暂定等待垃圾回收，而是与垃圾回收并发执行。要执行这种垃圾回收器可以在启动参数中配置：

-XX:+UseConcMarkSweepGC

CMS回收机运行机制非常复杂，我们简单的将他的运行流程分为以下几步：

**初始标记**

标记从GC Root可以直接可达的对象；

**并发标记（和应用程序线程一起）**

主要标记过程，标记全部对象；

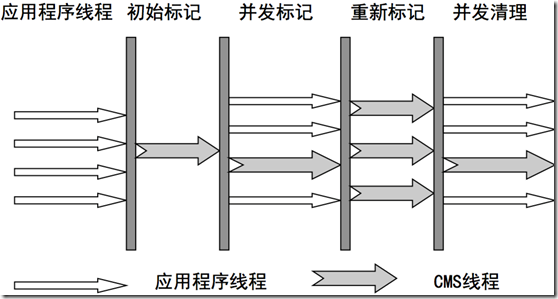
重新标记

由于并发标记时，用户线程依然运行，因此在正式清理前，再做依次重新标记，进行修正。

并发清除（和用户线程一起）

基于标记结果，直接清理对象。

流程如下图所示：

[](http://images2017.cnblogs.com/blog/352511/201708/352511-20170821094641011-247590646.png)

从上图可以看到标记过程分三步：初始标记、并发标记、重新标记，并发标记是最主要的标记过程，而这个过程是并发执行的，可以与应用程序线程同时进行，初始标记和重新标记虽然不能和应用程序并发执行，但这两个过程标记速度快，时间短，所以对应用程序不会产生太大的影响。最后并发清除的过程，也是和应用程序同时进行的，避免了应用程序的停顿。

CMS的优点显而易见，就是减少了应用程序的停顿时间，让回收线程和应用程序线程可以并发执行。但它也不是完美的，从他的运行机制可以看出，因为它不像其他回收器一样集中一段时间对垃圾进行回收，并且在回收时应用程序还是运行，因此它的回收并不彻底。这也导致了CMS回收的频率相较其他回收器要高，频繁的回收将影响应用程序的吞吐量。

G1回收器

G1回收器是jdk1.7以后推出的回收器，试图取代CMS回收器。

不同于其他的回收器、G1将堆空间划分成了互相独立的区块。每块区域既有可能属于老年代、也有可能是新生代，并且每类区域空间可以是不连续的（对比CMS的老年代和新生代都必须是连续的）。这种将老年代区划分成多块的理念源于：当并发后台线程寻找可回收的对象时、有些区块包含可回收的对象要比其他区块多很多。虽然在清理这些区块时G1仍然需要暂停应用线程、但可以用相对较少的时间优先回收包含垃圾最多区块。这也是为什么G1命名为Garbage First的原因：第一时间处理垃圾最多的区块。要使用G1回收器需要在启动是配置以下参数：

-XX:+UseG1GC

G1相对CMS回收器来说优点在于：

1、因为划分了很多区块，回收时减小了内存碎片的产生；

2、G1适用于新生代和老年代，而CMS只适用于老年代。

小结

    本文简要介绍了JVM中的垃圾回收器，主要包括串行回收器、并行回收器以及CMS回收器、G1回收器。他们各自都有优缺点，通常来说你需要根据你的业务，进行基于垃圾回收器的性能测试，然后再做选择。下面给出配置回收器时，经常使用的参数：

-XX:+UseSerialGC：在新生代和老年代使用串行收集器

-XX:+UseParNewGC：在新生代使用并行收集器

-XX:+UseParallelGC ：新生代使用并行回收收集器，更加关注吞吐量

-XX:+UseParallelOldGC：老年代使用并行回收收集器

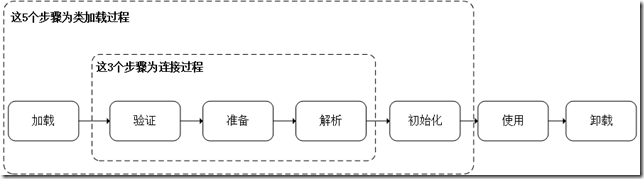
-XX:ParallelGCThreads：设置用于垃圾回收的线程数

-XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用CMS+串行收集器

-XX:ParallelCMSThreads：设定CMS的线程数量

-XX:+UseG1GC：启用G1垃圾回收器

类装载流程

[](http://images2017.cnblogs.com/blog/352511/201708/352511-20170825174440105-2124553043.png)

1、加载

加载是类装载的第一步，首先通过class文件的路径读取到二进制流，并解析二进制流将里面的元数据（类型、常量等）载入到方法区，在java堆中生成对应的java.lang.Class对象。

2、连接

连接过程又分为3步，验证、准备、解析

2.1、验证

验证的主要目的就是判断class文件的合法性，比如class文件一定是以0xCAFEBABE开头的，另外对版本号也会做验证，例如如果使用java1.8编译后的class文件要再java1.6虚拟机上运行，因为版本问题就会验证不通过。除此之外还会对元数据、字节码进行验证，具体的验证过程就复杂的多了，可以专门查看相关资料去了解。

2.2、准备

准备过程就是分配内存，给类的一些字段设置初始值，例如：

public static int v=1;

这段代码在准备阶段v的值就会被初始化为0，只有到后面类初始化阶段时才会被设置为1。

但是对于static final（常量），在准备阶段就会被设置成指定的值，例如：

public static final  int v=1;

这段代码在准备阶段v的值就是1。

2.3、解析

解析过程就是将符号引用替换为直接引用，例如某个类继承java.lang.object，原来的符号引用记录的是“java.lang.object”这个符号，凭借这个符号并不能找到java.lang.object这个对象在哪里？而直接引用就是要找到java.lang.object所在的内存地址，建立直接引用关系，这样就方便查询到具体对象。

3、初始化

初始化过程，主要包括执行类构造方法、static变量赋值语句，staic{}语句块，需要注意的是如果一个子类进行初始化，那么它会事先初始化其父类，保证父类在子类之前被初始化。所以其实在java中初始化一个类，那么必然是先初始化java.lang.Object，因为所有的java类都继承自java.lang.Object。

说完了类加载过程，我们来介绍一下这个过程当中的主角：类加载器。

类加载器

类加载器ClassLoader，它是一个抽象类，ClassLoader的具体实例负责把java字节码读取到JVM当中，ClassLoader还可以定制以满足不同字节码流的加载方式，比如从网络加载、从文件加载。ClassLoader的负责整个类装载流程中的“加载”阶段。

ClassLoader的重要方法：

1: public Class<?> loadClass(String name) throws ClassNotFoundException

载入并返回一个类。

1: protected final Class<?> defineClass(byte[] b, int off, int len)

定义一个类，该方法不公开被调用。

1: protected Class<?> findClass(String name) throws ClassNotFoundException

查找类，loadClass的回调方法

1: protected final Class<?> findLoadedClass(String name)

查找已经加载的类。

系统中的ClassLoader

BootStrap Classloader (启动ClassLoader)

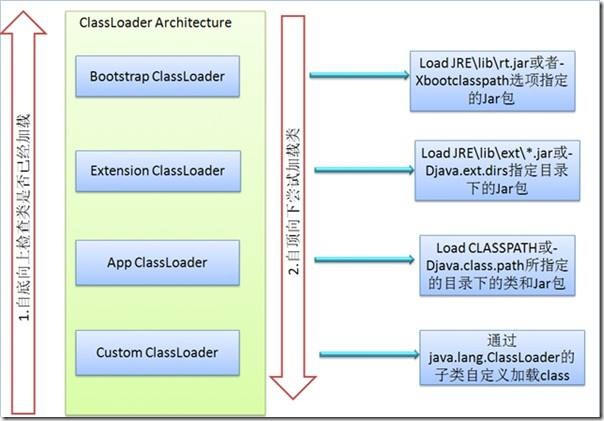
Extension ClassLoader （扩展ClassLoader）

App ClassLoader(应用 ClassLoader)

Custom ClassLoader(自定义ClassLoader)

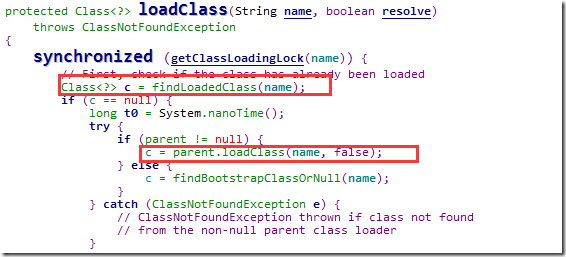
每个ClassLoader都有另外一个ClassLoader作为父ClassLoader，BootStrap Classloader除外，它没有父Classloader。

ClassLoader加载机制如下：

[](http://images2017.cnblogs.com/blog/352511/201708/352511-20170825174114043-87434631.jpg)

自下向上检查类是否被加载，一般情况下，首先从App ClassLoader中调用findLoadedClass方法查看是否已经加载，如果没有加载，则会交给父类，Extension ClassLoader去查看是否加载，还没加载，则再调用其父类，BootstrapClassLoader查看是否已经加载，如果仍然没有，自顶向下尝试加载类，那么从 Bootstrap ClassLoader到 App ClassLoader依次尝试加载。

值得注意的是即使两个类来源于相同的class文件，如果使用不同的类加载器加载，加载后的对象是完全不同的，这个不同反应在对象的 equals()、isAssignableFrom()、isInstance()等方法的返回结果，也包括了使用 instanceof 关键字对对象所属关系的判定结果。

[](http://images2017.cnblogs.com/blog/352511/201708/352511-20170825174115152-692730444.png)

从代码上可以看出，首先查看这个类是否被加载，如果没有则调用父类的loadClass方法，直到BootstrapClassLoader（没有父类），我们把这个过程叫做双亲模式，

双亲模式的问题

**顶层ClassLoader，无法加载底层ClassLoader的类**

Java框架(rt.jar)如何加载应用的类？

比如：javax.xml.parsers包中定义了xml解析的类接口  
Service Provider Interface SPI 位于rt.jar   
即接口在启动ClassLoader中。  
而SPI的实现类，在AppLoader。

这样就无法用BootstrapClassLoader去加载SPI的实现类。

##### 解决

JDK中提供了一个方法：

1: Thread. setContextClassLoader()

用以解决顶层ClassLoader无法访问底层ClassLoader的类的问题；  
基本思想是，在顶层ClassLoader中，传入底层ClassLoader的实例。

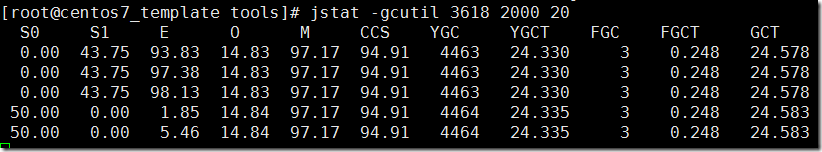
##### 双亲模式的破坏

双亲模式是默认的模式，但不是必须这么做；  
Tomcat的WebappClassLoader 就会先加载自己的Class，找不到再委托parent；  
OSGi的ClassLoader形成网状结构，根据需要自由加载Class。

下面再介绍几个JDK自带有用的工具：**jps、jstat、jmap、jstack**。

jps：上面我们已经使用过了，他可以罗列出目前再服务器上运行的java程序及进程ID；

jstat：用于输出java程序内存使用情况，包括新生代、老年代、元数据区容量、垃圾回收情况。

[](http://images2017.cnblogs.com/blog/352511/201709/352511-20170901174149233-65271705.png)

上述命令输出进程ID为3618的内存使用情况（每2000毫秒输出一次，一共输出20次）

* S0：幸存1区当前使用比例
* S1：幸存2区当前使用比例
* E：伊甸园区使用比例
* O：老年代使用比例
* M：元数据区使用比例
* CCS：压缩使用比例
* YGC：年轻代垃圾回收次数
* FGC：老年代垃圾回收次数
* FGCT：老年代垃圾回收消耗时间
* GCT：垃圾回收消耗总时间

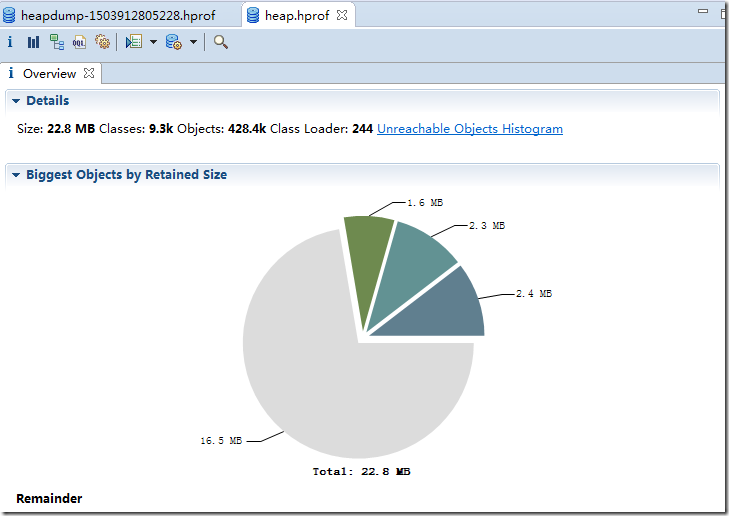
jmap：用于输出java程序中内存对象的情况，包括有哪些对象，对象的数量。

jmap -histo 3618

上述命令打印出进程ID为3618的内存情况。但我们常用的方式是将指定进程的内存heap输出到外部文件，再由专门的heap分析工具进行分析,例如mat（Memory Analysis Tool），所以我们常用的命令是：

jmap -dump:live,format=b,file=heap.hprof 3618

将heap.hprof传输出来到window电脑上使用mat工具分析：

[](http://images2017.cnblogs.com/blog/352511/201709/352511-20170901171922358-1722726117.png)

jstack：用户输出java程序线程栈的情况，常用于定位因为某些线程问题造成的故障或性能问题。

jstack 3618 > jstack.out

上述命令将进程ID为3618的栈信息输出到外部文件，便于传输到windows电脑上进行分析。

1. 静态块：用static申明，JVM加载类时执行，仅执行一次

构造块：类中直接用{}定义，每一次创建对象时执行

执行顺序优先级：静态块>main()>构造块>构造方法

2. HttpServletResponse完成：设置http头标，设置cookie，设置返回数据类型，输出返回数据；读取路径信息是HttpServletRequest做的

设置HTTP头标

response.setHeader("Refresh","3"); //三秒刷新页面一次

设置cookie

Cookie c1 = new Cookie("username","only");

response.addCookie(c1);

读取路径信息,request读取路径信息

从request获取各种路径总结

request.getRealPath("url"); // 虚拟目录映射为实际目录

request.getRealPath("./"); // 网页所在的目录

request.getRealPath("../"); // 网页所在目录的上一层目录

request.getContextPath(); // 应用的web目录的名称

输出返回数据

HttpServleteResponse.getOutputStream().write();

3. [Java](http://lib.csdn.net/base/17)把内存分成两种，一种叫做栈内存，一种叫做堆内存。

在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。

堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈0内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。

引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放。而数组＆对象本身在堆中分配，即使程序运行到使用new产生数组和对象的语句所在地代码块之外，数组和对象本身占用的堆内存也不会被释放，数组和对象在没有引用变量指向它的时候（比如先前的引用变量x=null时），才变成垃圾，不能再被使用，但是仍然占着内存，在随后的一个不确定的时间被垃圾回收器释放掉。这个也是java比较占内存的主要原因。

总结起来就是对象存储在堆内存，引用变量存储在栈内存。栈内存指向堆内存。

4. 有四种方法可以实现会话跟踪技术：URL重写、隐藏表单域、Cookie、Session。  
1）.隐藏表单域：<input type="hidden">，非常适合步需要大量数据存储的会话应用。  
2）.URL 重写:URL 可以在后面附加参数，和服务器的请求一起发送，这些参数为名字/值对。  
3）.Cookie:一个 Cookie 是一个小的，已命名数据元素。服务器使用 SET-Cookie 头标将它作为 HTTP  
响应的一部分传送到客户端，客户端被请求保存 Cookie 值，在对同一服务器的后续请求使用一个  
Cookie 头标将之返回到服务器。与其它技术比较，Cookie 的一个优点是在浏览器会话结束后，甚至  
在客户端计算机重启后它仍可以保留其值  
4）.Session：使用 setAttribute(String str,Object obj)方法将对象捆绑到一个会话

关于Java并发的说法

A，CopyOnWriteArrayList适用于**写少读多**的并发场景

B，ReadWriteLock即为读写锁，他要求写与写之间互斥，读与写之间互斥，

   读与读之间可以并发执行。在读多写少的情况下可以提高效率

C，ConcurrentHashMap是同步的HashMap，读写都加锁

D，volatile只保证多线程操作的可见性，不保证原子性

**关于抽象类：**

A.抽象类可以有非抽象的方法，而接口中的方法都是抽象方法

B.java中类只能单继承，接口可以‘继承’多个接口

C.抽象类必须有构造方法，接口一定没有构造方法

D.实例化一般指new一个对象，所以抽象类不能实例化

A、Semaphore：类，控制某个资源可被同时访问的个数;

B、ReentrantLock：类，具有与使用synchronized方法和语句所访问的隐式监视器锁相同的一些基本行为和语义，但功能更强大；

C、 Future：接口，表示异步计算的结果；

D、 CountDownLatch： 类，可以用来在一个线程中等待多个线程完成任务的类。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **访问级别** | **访问控制修饰符** | **同类** | **同包** | **子类** | **不同的包** |
| 公开 | public | **√** | **√** | **√** | **√** |
| 受保护 | protected | **√** | **√** | **√** | **--** |
| 默认 | 没有访问控制修饰符 | **√** | **√** | **--** | **--** |
| 私有 | private | **√** | **--** | **--** | **--** |

Servlet接口定义了5个方法，其中**前三个方法与Servlet生命周期相关**：

* **void init(ServletConfig config) throws ServletException**
* **void service(ServletRequest req, ServletResponse resp) throws ServletException, java.io.IOException**
* **void destory()**
* java.lang.String getServletInfo()
* ServletConfig getServletConfig()

**生命周期：** **Web容器加载Servlet并将其实例化后，Servlet生命周期开始**，容器运行其**init()方法**进行Servlet的初始化；请求到达时调用Servlet的**service()方法**，service()方法会根据需要调用与请求对应的**doGet或doPost**等方法；当服务器关闭或项目被卸载时服务器会将Servlet实例销毁，此时会调用Servlet的**destroy()方法**。**init方法和destory方法只会执行一次，service方法客户端每次请求Servlet都会执行**。Servlet中有时会用到一些需要初始化与销毁的资源，因此可以把初始化资源的代码放入init方法中，销毁资源的代码放入destroy方法中，这样就不需要每次处理客户端的请求都要初始化与销毁资源。

## Arraylist 与 LinkedList 异同

* **1. 是否保证线程安全：** ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
* **2. 底层数据结构：** Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向链表数据结构（JDK1.6之前为循环链表，JDK1.7取消了循环。注意双向链表和双向循环链表的区别：）； 详细可阅读[JDK1.7-LinkedList循环链表优化](https://www.cnblogs.com/xingele0917/p/3696593.html)
* **3. 插入和删除是否受元素位置的影响：** ① **ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。** 比如：执行add(E e) 方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element) ）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ② **LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。**
* **4. 是否支持快速随机访问：** LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index) 方法)。
* **5. 内存空间占用：** ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

数组天然支持随机访问，时间复杂度为 O（1），所以称为快速随机访问。链表需要遍历到特定位置才能访问特定位置的元素，时间复杂度为 O（n），所以不支持快速随机访问

## HashMap 和 Hashtable 的区别

1. **线程是否安全：** HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 synchronized修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
2. **效率：** 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；
3. **对Null key 和Null value的支持：** HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。
4. **初始容量大小和每次扩充容量大小的不同 ：** ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的tableSizeFor()方法保证，下面给出了源代码）。也就是说 HashMap 总是使用2的幂作为哈希表的大小,后面会介绍到为什么是2的幂次方。
5. **底层数据结构：** JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

### synchronized关键字最主要的三种使用方式的总结

* **修饰实例方法，作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁**
* **修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁** 。也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份，所以对该类的所有对象都加了锁）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，**因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁**。
* **修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。** 和 synchronized 方法一样，synchronized(this)代码块也是锁定当前对象的。synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。这里再提一下：synchronized关键字加到非 static 静态方法上是给对象实例上锁。另外需要注意的是：尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓冲功能！

### JDK1.6 之后的底层优化

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

**①偏向锁**

**引入偏向锁的目的和引入轻量级锁的目的很像，他们都是为了没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。但是不同是：轻量级锁在无竞争的情况下使用 CAS 操作去代替使用互斥量。而偏向锁在无竞争的情况下会把整个同步都消除掉**。

偏向锁的“偏”就是偏心的偏，它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步！关于偏向锁的原理可以查看《深入理解Java虚拟机：JVM高级特性与最佳实践》第二版的13章第三节锁优化。

但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。

**② 轻量级锁**

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)。**轻量级锁不是为了代替重量级锁，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗，因为使用轻量级锁时，不需要申请互斥量。另外，轻量级锁的加锁和解锁都用到了CAS操作。** 关于轻量级锁的加锁和解锁的原理可以查看《深入理解Java虚拟机：JVM高级特性与最佳实践》第二版的13章第三节锁优化。

**轻量级锁能够提升程序同步性能的依据是“对于绝大部分锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。但如果存在锁竞争，除了互斥量开销外，还会额外发生CAS操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢！如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁！**

**③ 自旋锁和自适应自旋**

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

互斥同步对性能最大的影响就是阻塞的实现，因为挂起线程/恢复线程的操作都需要转入内核态中完成（用户态转换到内核态会耗费时间）。

**一般线程持有锁的时间都不是太长，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。** 所以，虚拟机的开发团队就这样去考虑：“我们能不能让后面来的请求获取锁的线程等待一会而不被挂起呢？看看持有锁的线程是否很快就会释放锁”。**为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋**。

百度百科对自旋锁的解释：

何谓自旋锁？它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

自旋锁在 JDK1.6 之前其实就已经引入了，不过是默认关闭的，需要通过--XX:+UseSpinning参数来开启。JDK1.6及1.6之后，就改为默认开启的了。需要注意的是：自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。如果锁被占用的时间短，那么效果当然就很好了！反之，相反！自旋等待的时间必须要有限度。如果自旋超过了限定次数任然没有获得锁，就应该挂起线程。**自旋次数的默认值是10次，用户可以修改--XX:PreBlockSpin来更改**。

另外,**在 JDK1.6 中引入了自适应的自旋锁。自适应的自旋锁带来的改进就是：自旋的时间不在固定了，而是和前一次同一个锁上的自旋时间以及锁的拥有者的状态来决定，虚拟机变得越来越“聪明”了**。

**④ 锁消除**

锁消除理解起来很简单，它指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

**⑤ 锁粗化**

原则上，我们再编写代码的时候，总是推荐将同步快的作用范围限制得尽量小——只在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。

大部分情况下，上面的原则都是没有问题的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，那么会带来很多不必要的性能消耗。

### Synchronized 和 ReenTrantLock 的对比

**① 两者都是可重入锁**

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

**② synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API**

synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReenTrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

**③ ReenTrantLock 比 synchronized 增加了一些高级功能**

相比synchronized，ReenTrantLock增加了一些高级功能。主要来说主要有三点：**①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）**

* **ReenTrantLock提供了一种能够中断等待锁的线程的机制**，通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
* **ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。** ReenTrantLock默认情况是非公平的，可以通过 ReenTrantLock类的ReentrantLock(boolean fair)构造方法来制定是否是公平的。
* synchronized关键字与wait()和notify/notifyAll()方法相结合可以实现等待/通知机制，ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition() 方法。Condition是JDK1.5之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），**线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知”** ，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。

如果你想使用上述功能，那么选择ReenTrantLock是一个不错的选择。

**④ 性能已不是选择标准**

在JDK1.6之前，synchronized 的性能是比 ReenTrantLock 差很多。具体表示为：synchronized 关键字吞吐量岁线程数的增加，下降得非常严重。而ReenTrantLock 基本保持一个比较稳定的水平。我觉得这也侧面反映了， synchronized 关键字还有非常大的优化余地。后续的技术发展也证明了这一点，我们上面也讲了在 JDK1.6 之后 JVM 团队对 synchronized 关键字做了很多优化。**JDK1.6 之后，synchronized 和 ReenTrantLock 的性能基本是持平了。所以网上那些说因为性能才选择 ReenTrantLock 的文章都是错的！JDK1.6之后，性能已经不是选择synchronized和ReenTrantLock的影响因素了！而且虚拟机在未来的性能改进中会更偏向于原生的synchronized，所以还是提倡在synchronized能满足你的需求的情况下，优先考虑使用synchronized关键字来进行同步！优化后的synchronized和ReenTrantLock一样，在很多地方都是用到了CAS操作**。

### 何谓悲观锁与乐观锁

乐观锁对应于生活中乐观的人总是想着事情往好的方向发展，悲观锁对应于生活中悲观的人总是想着事情往坏的方向发展。这两种人各有优缺点，不能不以场景而定说一种人好于另外一种人。

#### 悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（**共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程**）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

#### 乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。**乐观锁适用于多读的应用类型，这样可以提高吞吐量**，像数据库提供的类似于**write\_condition机制**，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式**CAS**实现的。

#### 两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像**乐观锁适用于写比较少的情况下（多读场景）**，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以**一般多写的场景下用悲观锁就比较合适。**

### 乐观锁常见的两种实现方式

**乐观锁一般会使用版本号机制或CAS算法实现。**

#### 1. 版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

**举一个简单的例子：** 假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ；而当前帐户余额字段（ balance ）为 $100 。

1. 操作员 A 此时将其读出（ version=1 ），并从其帐户余额中扣除 $50（ $100-$50 ）。
2. 在操作员 A 操作的过程中，操作员B 也读入此用户信息（ version=1 ），并从其帐户余额中扣除 $20 （ $100-$20 ）。
3. 操作员 A 完成了修改工作，将数据版本号加一（ version=2 ），连同帐户扣除后余额（ balance=$50 ），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2 。
4. 操作员 B 完成了操作，也将版本号加一（ version=2 ）试图向数据库提交数据（ balance=$80 ），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2 ，数据库记录当前版本也为 2 ，不满足 “ 提交版本必须大于记录当前版本才能执行更新 “ 的乐观锁策略，因此，操作员 B 的提交被驳回。

这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

#### 2. CAS算法

即**compare and swap（比较与交换）**，是一种有名的**无锁算法**。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。**CAS算法**涉及到三个操作数

* 需要读写的内存值 V
* 进行比较的值 A
* 拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个**自旋操作**，即**不断的重试**。

关于自旋锁，大家可以看一下这篇文章，非常不错：[《 面试必备之深入理解自旋锁》](https://blog.csdn.net/qq_34337272/article/details/81252853):

### 什么是自旋锁？

自旋锁（spinlock）：是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。

获取锁的线程一直处于活跃状态，但是并没有执行任何有效的任务，使用这种锁会造成[busy-waiting](https://en.wikipedia.org/wiki/Busy_waiting" \t "_blank)。

它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，”自旋”一词就是因此而得名。

### 自旋锁存在的问题

1. 如果某个线程持有锁的时间过长，就会导致其它等待获取锁的线程进入循环等待，消耗CPU。使用不当会造成CPU使用率极高。
2. 上面Java实现的自旋锁不是公平的，即无法满足等待时间最长的线程优先获取锁。不公平的锁就会存在“线程饥饿”问题。

### 自旋锁的优点

1. 自旋锁不会使线程状态发生切换，一直处于用户态，即线程一直都是active的；不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快
2. 非自旋锁在获取不到锁的时候会进入阻塞状态，从而进入内核态，当获取到锁的时候需要从内核态恢复，需要线程上下文切换。 （线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能）

### 自旋锁与互斥锁

* 自旋锁与互斥锁都是为了实现保护资源共享的机制。
* 无论是自旋锁还是互斥锁，在任意时刻，都最多只能有一个保持者。
* 获取互斥锁的线程，如果锁已经被占用，则该线程将进入睡眠状态；获取自旋锁的线程则不会睡眠，而是一直循环等待锁释放。

### 总结：

* 自旋锁：线程获取锁的时候，如果锁被其他线程持有，则当前线程将循环等待，直到获取到锁。
* 自旋锁等待期间，线程的状态不会改变，线程一直是用户态并且是活动的(active)。
* 自旋锁如果持有锁的时间太长，则会导致其它等待获取锁的线程耗尽CPU。
* 自旋锁本身无法保证公平性，同时也无法保证可重入性。
* 基于自旋锁，可以实现具备公平性和可重入性质的锁。
* TicketLock:采用类似银行排号叫好的方式实现自旋锁的公平性，但是由于不停的读取serviceNum，每次读写操作都必须在多个处理器缓存之间进行缓存同步，这会导致繁重的系统总线和内存的流量，大大降低系统整体的性能。
* CLHLock和MCSLock通过链表的方式避免了减少了处理器缓存同步，极大的提高了性能，区别在于CLHLock是通过轮询其前驱节点的状态，而MCS则是查看当前节点的锁状态。
* CLHLock在NUMA架构下使用会存在问题。在没有cache的NUMA系统架构中，由于CLHLock是在当前节点的前一个节点上自旋,NUMA架构中处理器访问本地内存的速度高于通过网络访问其他节点的内存，所以CLHLock在NUMA架构上不是最优的自旋锁。

### 乐观锁的缺点

ABA 问题是乐观锁一个常见的问题

#### 1 ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 **"ABA"问题。**

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

#### 2 循环时间长开销大

**自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。** 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

#### 3 只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作。

### CAS与synchronized的使用情景

**简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）**

1. 对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
2. 对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

补充： Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为 **“重量级锁”** 。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 **偏向锁** 和 **轻量级锁** 以及其它**各种优化**之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠 **Lock-Free** 的队列，基本思路是 **自旋后阻塞**，**竞争切换后继续竞争锁**，**稍微牺牲了公平性，但获得了高吞吐量**。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

### 1.3 讲一下 synchronized 关键字的底层原理

**synchronized 关键字底层原理属于 JVM 层面。**

* 1. **synchronized 同步语句块的情况**

**synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。** 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权.当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

* 1. **synchronized 修饰方法的的情况**

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC\_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

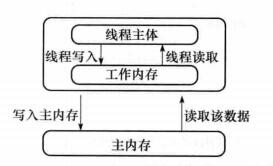
### 1.4 说说 JDK1.6 之后的synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优化吗

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

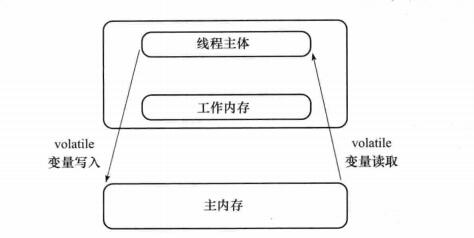
### 2.1 讲一下Java内存模型

在 JDK1.2 之前，Java的内存模型实现总是从**主存**（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存**本地内存**（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成**数据的不一致**。

[](https://camo.githubusercontent.com/5dabf65a6f750c5b767b2e221aaa27eab47d4fa3/68747470733a2f2f757365722d676f6c642d63646e2e786974752e696f2f323031382f31302f33302f313636633436656465343432336261323f773d32373326683d31363626663d6a70656726733d37323638)

要解决这个问题，就需要把变量声明为 **volatile**，这就指示 JVM，这个变量是不稳定的，每次使用它都到主存中进行读取。

说白了， **volatile** 关键字的主要作用就是保证变量的可见性然后还有一个作用是防止指令重排序。

[](https://camo.githubusercontent.com/732fc27aae9ea88870e52523712d4902a540524e/68747470733a2f2f757365722d676f6c642d63646e2e786974752e696f2f323031382f31302f33302f313636633436656465346239663530313f773d34373426683d32333826663d6a70656726733d39393432)

### 2.2 说说 synchronized 关键字和 volatile 关键字的区别

synchronized关键字和volatile关键字比较

* **volatile关键字**是线程同步的**轻量级实现**，所以**volatile性能肯定比synchronized关键字要好**。但是**volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块**。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些**。
* **多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞**
* **volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。**
* **volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。**

### 为什么要用线程池？

线程池提供了一种限制和管理资源（包括执行一个任务）。 每个线程池还维护一些基本统计信息，例如已完成任务的数量。

这里借用《Java并发编程的艺术》提到的来说一下使用线程池的好处：

* **降低资源消耗。** 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
* **提高响应速度。** 当任务到达时，任务可以不需要的等到线程创建就能立即执行。
* **提高线程的可管理性。** 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

### 执行execute()方法和submit()方法的区别是什么呢？

1)**execute() 方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；**

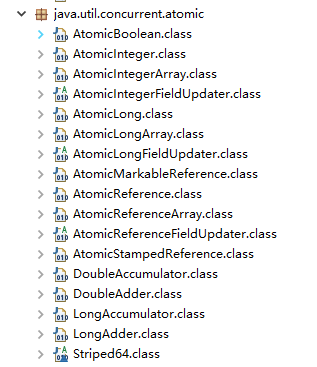
2)**submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功**，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

### 介绍一下Atomic 原子类

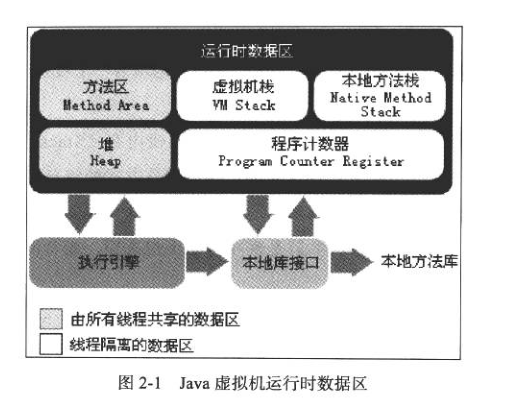
Atomic 翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。在我们这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。

所以，所谓原子类说简单点就是具有原子/原子操作特征的类。

并发包 java.util.concurrent 的原子类都存放在java.util.concurrent.atomic下,如下图所示。

[](https://camo.githubusercontent.com/d97cb5018f858957b4ebcb7abcfa25b081e2df18/68747470733a2f2f757365722d676f6c642d63646e2e786974752e696f2f323031382f31302f33302f313636633461633038643463353534373f773d33313726683d33363726663d706e6726733d3133323637)

## 2 运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。 [](https://camo.githubusercontent.com/7dbe84f3d6186c5f8e2836489ad1dae57f4b2f59/68747470733a2f2f757365722d676f6c642d63646e2e786974752e696f2f323031382f342f32372f313633303661333463643861343335343f773d35313326683d34303426663d706e6726733d313332303638) 这些组成部分一些是线程私有的，其他的则是线程共享的。

**线程私有的：**

* 程序计数器
* 虚拟机栈
* 本地方法栈

**线程共享的：**

* 堆
* 方法区
* 直接内存

### 2.1 程序计数器

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。**字节码解释器工作时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、线程恢复等功能都需要依赖这个计数器来完。**

另外，**为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。**

**从上面的介绍中我们知道程序计数器主要有两个作用：**

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

**注意：程序计数器是唯一一个不会出现OutOfMemoryError的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。**

### 2.2 Java 虚拟机栈

**与程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是 Java 方法执行的内存模型。**

**Java 内存可以粗糙的区分为堆内存（Heap）和栈内存(Stack),其中栈就是现在说的虚拟机栈，或者说是虚拟机栈中局部变量表部分。** （实际上，Java虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。）

**局部变量表主要存放了编译器可知的各种数据类型**（boolean、byte、char、short、int、float、long、double）、**对象引用**（reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

**Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。**

* **StackOverFlowError：** 若Java虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出StackOverFlowError异常。
* **OutOfMemoryError：** 若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出OutOfMemoryError异常。

Java 虚拟机栈也是线程私有的，每个线程都有各自的Java虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

### 2.3 本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是： **虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。** 在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

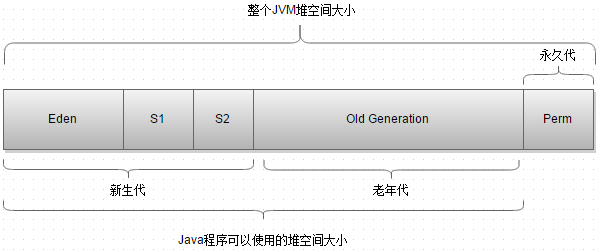
本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 StackOverFlowError 和 OutOfMemoryError 两种异常。

### 2.4 堆

Java 虚拟机所管理的内存中最大的一块，Java 堆是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以及数组都在这里分配内存。**

Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC堆（Garbage Collected Heap）**.从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以Java堆还可以细分为：新生代和老年代：再细致一点有：Eden空间、From Survivor、To Survivor空间等。**进一步划分的目的是更好地回收内存，或者更快地分配内存。**

[](https://camo.githubusercontent.com/05d1fe88e8ac4b03f299af3519b37e76ba4bebf3/68747470733a2f2f757365722d676f6c642d63646e2e786974752e696f2f323031382f382f32352f313635373033343461323963333433333f773d35393926683d32353026663d706e6726733d38393436)

**在 JDK 1.8中移除整个永久代，取而代之的是一个叫元空间（Metaspace）的区域（永久代使用的是JVM的堆内存空间，而元空间使用的是物理内存，直接受到本机的物理内存限制）。**

推荐阅读：

* 《Java8内存模型—永久代(PermGen)和元空间(Metaspace)》：<http://www.cnblogs.com/paddix/p/5309550.html>

### 2.5 方法区

**方法区与 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。**

HotSpot 虚拟机中方法区也常被称为 **“永久代”**，本质上两者并不等价。仅仅是因为 HotSpot 虚拟机设计团队用永久代来实现方法区而已，这样 HotSpot 虚拟机的垃圾收集器就可以像管理 Java 堆一样管理这部分内存了。但是这并不是一个好主意，因为这样更容易遇到内存溢出问题。

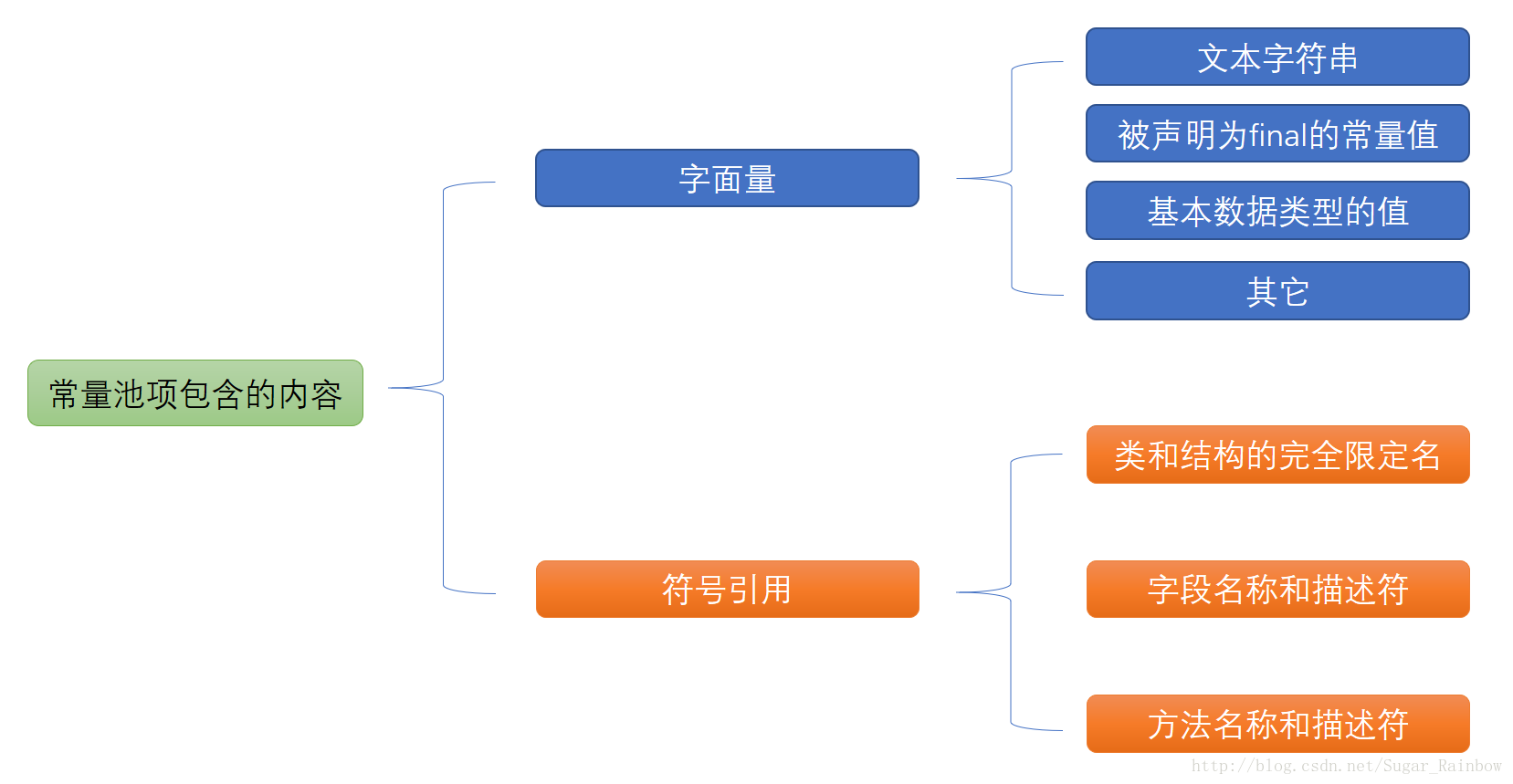
**相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区后就“永久存在”了。**

### 2.6 运行时常量池

运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池信息（用于存放编译期生成的各种字面量和符号引用）

既然运行时常量池时方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

**JDK1.7及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。**

[](https://camo.githubusercontent.com/17620721a9f326a235aeec8956949cec03f3f125/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d392d31342f32363033383433332e6a7067)——图片来源：<https://blog.csdn.net/wangbiao007/article/details/78545189>

推荐阅读：

* 《Java 中几种常量池的区分》： <https://blog.csdn.net/qq_26222859/article/details/73135660>

### 2.7 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致OutOfMemoryError异常出现。

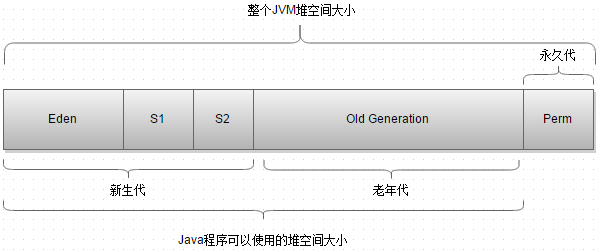
JDK1.4中新加入的 **NIO(New Input/Output) 类**，引入了一种基于**通道（Channel）** 与**缓存区（Buffer）** 的 I/O 方式，它可以直接使用Native函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样就能在一些场景中显著提高性能，因为**避免了在 Java 堆和 Native 堆之间来回复制数据**。

本机直接内存的分配不会收到 Java 堆的限制，但是，既然是内存就会受到本机总内存大小以及处理器寻址空间的限制。

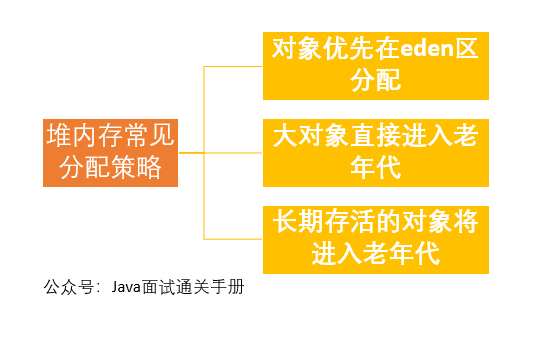
## 1 揭开JVM内存分配与回收的神秘面纱

Java 的自动内存管理主要是针对对象内存的回收和对象内存的分配。同时，Java 自动内存管理最核心的功能是 **堆** 内存中对象的分配与回收。

**JDK1.8之前的堆内存示意图：**

[](https://camo.githubusercontent.com/05d1fe88e8ac4b03f299af3519b37e76ba4bebf3/68747470733a2f2f757365722d676f6c642d63646e2e786974752e696f2f323031382f382f32352f313635373033343461323963333433333f773d35393926683d32353026663d706e6726733d38393436)

从上图可以看出堆内存分为新生代、老年代和永久代。新生代又被进一步分为：Eden 区＋Survivor1 区＋Survivor2 区。值得注意的是，在 JDK 1.8中移除整个永久代，取而代之的是一个叫元空间（Metaspace）的区域（永久代使用的是JVM的堆内存空间，而元空间使用的是物理内存，直接受到本机的物理内存限制）。

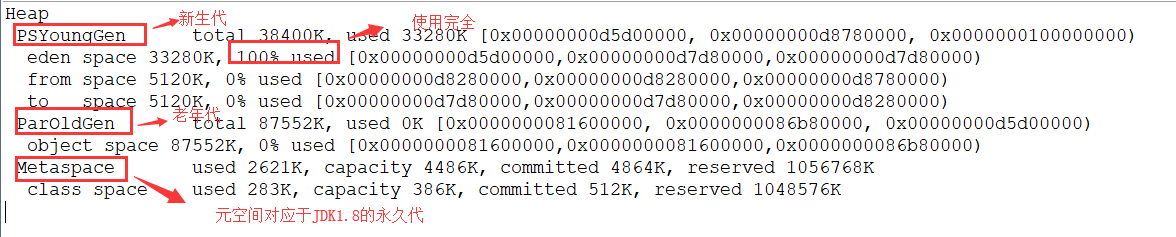
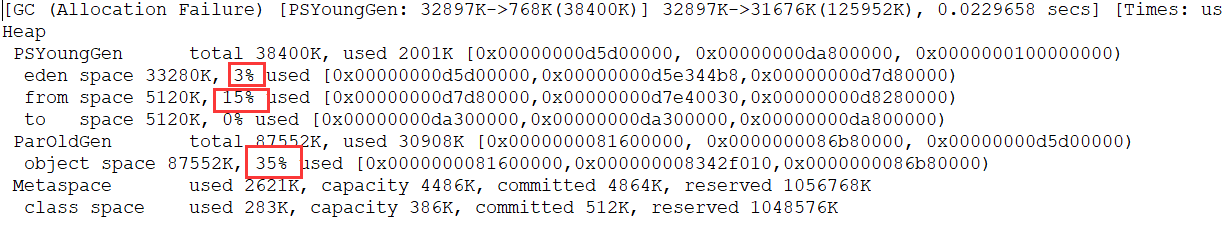
[](https://camo.githubusercontent.com/8da9ff51df49d4cf41107c6fc9db0f782bde9d38/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d382d32372f38393239343534372e6a7067)

### 1.1 对象优先在eden区分配

目前主流的垃圾收集器都会采用分代回收算法，因此需要将堆内存分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

大多数情况下，对象在新生代中 eden 区分配。当 eden 区没有足够空间进行分配时，虚拟机将发起一次Minor GC

**Minor GC和Full GC 有什么不同呢？**

* **新生代GC（Minor GC）**:指发生新生代的的垃圾收集动作，Minor GC非常频繁，回收速度一般也比较快。
* **老年代GC（Major GC/Full GC）**:指发生在老年代的GC，出现了Major GC经常会伴随至少一次的Minor GC（并非绝对），Major GC的速度一般会比Minor GC的慢10倍以上。
* 运行结果：
* [](https://camo.githubusercontent.com/42464b598a233d4319009cc06e3d199b6331c314/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d382d32362f32383935343238362e6a7067)
* 从上图我们可以看出eden区内存几乎已经被分配完全（即使程序什么也不做，新生代也会使用2000多k内存）。假如我们再为allocation2分配内存会出现什么情况呢？
* allocation2 = new byte[900\*1024];
* [](https://camo.githubusercontent.com/215514817e1b01cd89ba942c382d2f64cb6c9e3d/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d382d32362f32383132383738352e6a7067)
* **简单解释一下为什么会出现这种情况：** 因为给allocation2分配内存的时候eden区内存几乎已经被分配完了，我们刚刚讲了当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC.GC期间虚拟机又发现allocation1无法存入Survivor空间，所以只好通过 **分配担保机制** 把新生代的对象提前转移到老年代中去，老年代上的空间足够存放allocation1，所以不会出现Full GC。执行Minor GC后，后面分配的对象如果能够存在eden区的话，还是会在eden区分配内存

### 1.2 大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。

**为什么要这样呢？**

为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

### 1.3 长期存活的对象将进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年代中。为了做到这一点，虚拟机给每个对象一个对象年龄（Age）计数器。

如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为1.对象在 Survivor 中每熬过一次 MinorGC,年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。

### 1.4 动态对象年龄判定

为了更好的适应不同程序的内存情况，虚拟机不是永远要求对象年龄必须达到了某个值才能进入老年代，如果 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需达到要求的年龄。

## 2 对象已经死亡？

堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断那些对象已经死亡（即不能再被任何途径使用的对象）。

### 2.1 引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1；当引用失效，计数器就减1；任何时候计数器为0的对象就是不可能再被使用的。

**这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。** 所谓对象之间的相互引用问题，如下面代码所示：除了对象objA 和 objB 相互引用着对方之外，这两个对象之间再无任何引用。但是他们因为互相引用对方，导致它们的引用计数器都不为0，于是引用计数算法无法通知 GC 回收器回收他们。

public class ReferenceCountingGc {

Object instance = null;

public static void main(String[] args) {

ReferenceCountingGc objA = new ReferenceCountingGc();

ReferenceCountingGc objB = new ReferenceCountingGc();

objA.instance = objB;

objB.instance = objA;

objA = null;

objB = null;

}

}

### 2.2 可达性分析算法

这个算法的基本思想就是通过一系列的称为 **“GC Roots”** 的对象作为起点，从这些节点开始向下搜索，节点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的话，则证明此对象是不可用的。

### 2.3 再谈引用

无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

JDK1.2之前，Java中引用的定义很传统：如果reference类型的数据存储的数值代表的是另一块内存的起始地址，就称这块内存代表一个引用。

JDK1.2以后，Java对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用四种（引用强度逐渐减弱）

**1．强引用**

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于**必不可少的生活用品**，垃圾回收器绝不会回收它。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，也不会靠随意回收具有强引用的对象来解决内存不足问题。

**2．软引用（SoftReference）**

如果一个对象只具有软引用，那就类似于**可有可无的生活用品**。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，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。软引用可用来实现内存敏感的高速缓存。

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，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。

**3．弱引用（WeakReference）**

如果一个对象只具有弱引用，那就类似于**可有可无的生活用品**。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

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。

**4．虚引用（PhantomReference）**

"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。

**虚引用主要用来跟踪对象被垃圾回收的活动**。

**虚引用与软引用和弱引用的一个区别在于：** 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。程序如果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。

特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为**软引用可以加速JVM对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生**。

### 2.4 不可达的对象并非“非死不可”

即使在可达性分析法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程；可达性分析法中不可达的对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize 方法。当对象没有覆盖 finalize 方法，或 finalize 方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。

被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就会被真的回收。

### 2.5 如何判断一个常量是废弃常量

运行时常量池主要回收的是废弃的常量。那么，我们如何判断一个常量是废弃常量呢？

假如在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，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。

注意：我们在 [可能是把Java内存区域讲的最清楚的一篇文章](https://mp.weixin.qq.com/s?__biz=MzU4NDQ4MzU5OA==&mid=2247484303&idx=1&sn=af0fd436cef755463f59ee4dd0720cbd&chksm=fd9855eecaefdcf8d94ac581cfda4e16c8a730bda60c3b50bc55c124b92f23b6217f7f8e58d5&token=506869459&lang=zh_CN#rd) 也讲了JDK1.7及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。

### 2.6 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，那么如何判断一个类是无用的类的呢？

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面3个条件才能算是 **“无用的类”** ：

* 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。
* 加载该类的 ClassLoader 已经被回收。
* 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。

虚拟机可以对满足上述3个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

## 3 垃圾收集算法

### 3.1 标记-清除算法

算法分为“标记”和“清除”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所有被标记的对象。它是最基础的收集算法，效率也很高，但是会带来两个明显的问题：

1. **效率问题**
2. **空间问题（标记清除后会产生大量不连续的碎片）**

### 3.2 复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

### 3.3 标记-整理算法

根据老年代的特点特出的一种标记算法，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回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

### 3.4 分代收集算法

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将java堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

**比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。**

**延伸面试问题：** HotSpot为什么要分为新生代和老年代？

根据上面的对分代收集算法的介绍回答。

## 4 垃圾收集器

**如果说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。**

虽然我们对各个收集器进行比较，但并非要挑选出一个最好的收集器。因为知道现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，**我们能做的就是根据具体应用场景选择适合自己的垃圾收集器**。试想一下：如果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的HotSpot虚拟机就不会实现那么多不同的垃圾收集器了。

### 4.1 Serial收集器

Serial（串行）收集器收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的 **“单线程”** 的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它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（ **"Stop The World"** ），直到它收集结束。

**新生代采用复制算法，老年代采用标记-整理算法。**

虚拟机的设计者们当然知道Stop The World带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

但是Serial收集器有没有优于其他垃圾收集器的地方呢？当然有，它**简单而高效（与其他收集器的单线程相比）**。Serial收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。Serial收集器对于运行在Client模式下的虚拟机来说是个不错的选择。

### 4.2 ParNew收集器

**ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和Serial收集器完全一样。**

**新生代采用复制算法，老年代采用标记-整理算法。**

它是许多运行在Server模式下的虚拟机的首要选择，除了Serial收集器外，只有它能与CMS收集器（真正意义上的并发收集器，后面会介绍到）配合工作。

**并行和并发概念补充：**

* **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。
* **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，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。

### 4.3 Parallel Scavenge收集器

Parallel Scavenge 收集器类似于ParNew 收集器。 **那么它有什么特别之处呢？**

-XX:+UseParallelGC

使用Parallel收集器+ 老年代串行

-XX:+UseParallelOldGC

使用Parallel收集器+ 老年代并行

**Parallel Scavenge收集器关注点是吞吐量（高效率的利用CPU）。CMS等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是CPU中用于运行用户代码的时间与CPU总消耗时间的比值。** 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，如果对于收集器运作不太了解的话，手工优化存在的话可以选择把内存管理优化交给虚拟机去完成也是一个不错的选择。

**新生代采用复制算法，老年代采用标记-整理算法。**

### 4.4.Serial Old收集器

**Serial收集器的老年代版本**，它同样是一个单线程收集器。它主要有两大用途：一种用途是在JDK1.5以及以前的版本中与Parallel Scavenge收集器搭配使用，另一种用途是作为CMS收集器的后备方案。

### 4.5 Parallel Old收集器

**Parallel Scavenge收集器的老年代版本**。使用多线程和“标记-整理”算法。在注重吞吐量以及CPU资源的场合，都可以优先考虑 Parallel Scavenge收集器和Parallel Old收集器。

### 4.6 CMS收集器

**CMS（Concurrent 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。它而非常符合在注重用户体验的应用上使用。**

**CMS（Concurrent Mark Sweep）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。**

从名字中的**Mark Sweep**这两个词可以看出，CMS收集器是一种 **“标记-清除”算法**实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

* **初始标记：** 暂停所有的其他线程，并记录下直接与root相连的对象，速度很快 ；
* **并发标记：** 同时开启GC和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以GC线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。
* **重新标记：** 重新标记阶段就是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，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
* **并发清除：** 开启用户线程，同时GC线程开始对为标记的区域做清扫。

从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：**并发收集、低停顿**。但是它有下面三个明显的缺点：

* **对CPU资源敏感；**
* **无法处理浮动垃圾；**
* **它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。**

### 4.7 G1收集器

**G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗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.**

被视为JDK1.7中HotSpot虚拟机的一个重要进化特征。它具备一下特点：

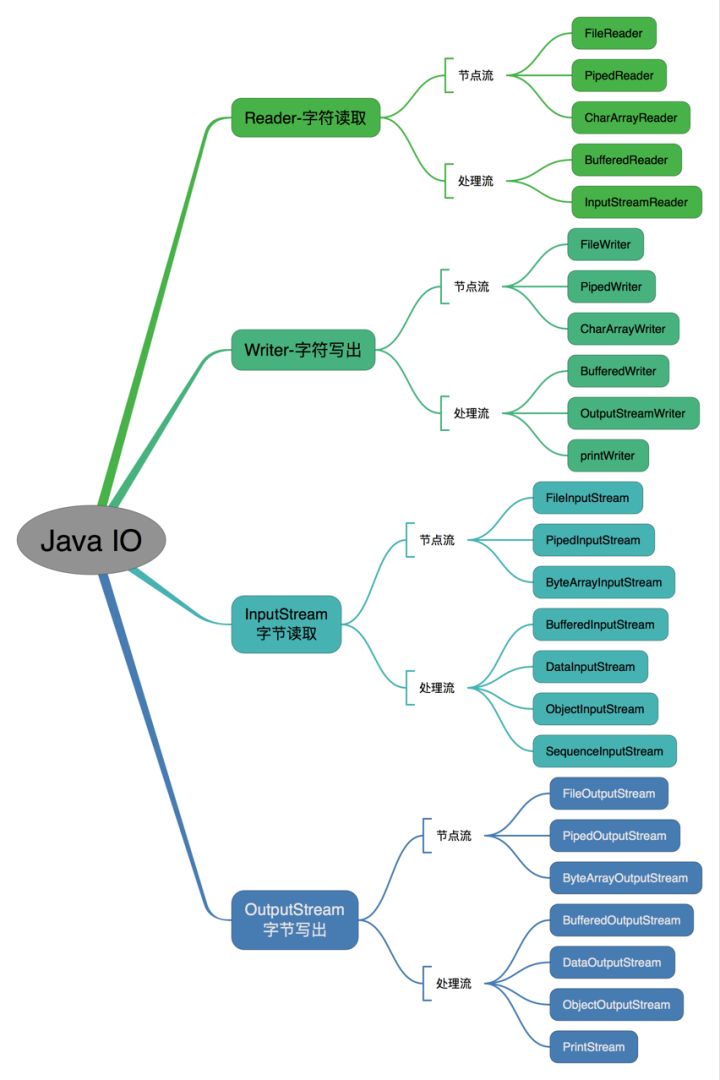
* **并行与并发**：G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短Stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。
* **分代收集**：虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。
* **空间整合**：与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
* **可预测的停顿**：这是G1相对于CMS的另一个大优势，降低停顿时间是G1 和 CMS 共同的关注点，但G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内。

G1收集器的运作大致分为以下几个步骤：

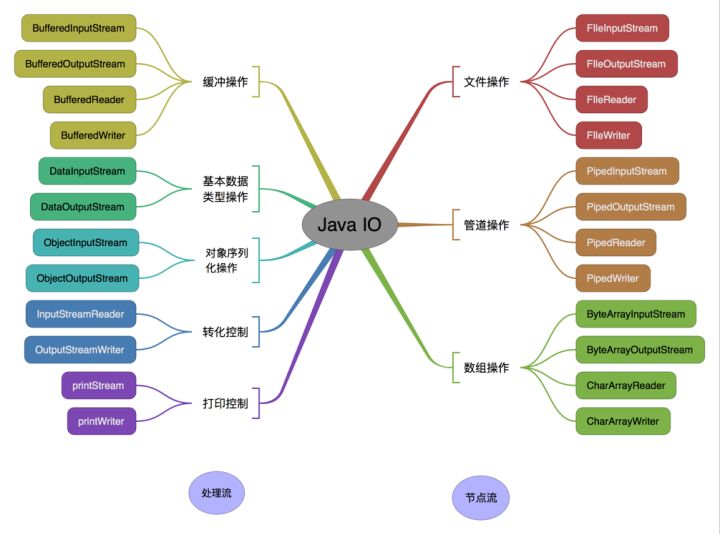
* **初始标记**
* **并发标记**
* **最终标记**
* **筛选回收**

**G1收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的Region(这也就是它的名字Garbage-First的由来)**。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了GF收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）。

**（1） 按操作方式分类结构图：**

[](https://camo.githubusercontent.com/50f105c85f6b42d643d46e1ac7bb0f855b92cd9d/68747470733a2f2f757365722d676f6c642d63646e2e786974752e696f2f323031382f352f31362f313633363764346664316365316234363f773d37323026683d3130383026663d6a70656726733d3639353232)

**（2）按操作对象分类结构图**

[](https://camo.githubusercontent.com/8957efacdf1cd4eac15d844da8353a7f77a3c863/68747470733a2f2f757365722d676f6c642d63646e2e786974752e696f2f323031382f352f31362f313633363764363733623065323638643f773d37323026683d35333526663d6a70656726733d3436303831)

* 红黑树特点:

1. 每个节点非红即黑；
2. 根节点总是黑色的；
3. 每个叶子节点都是黑色的空节点（NIL节点）；
4. 如果节点是红色的，则它的子节点必须是黑色的（反之不一定）；
5. 从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）

* 红黑树的应用：

TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。

* 为什么要用红黑树

简单来说红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构

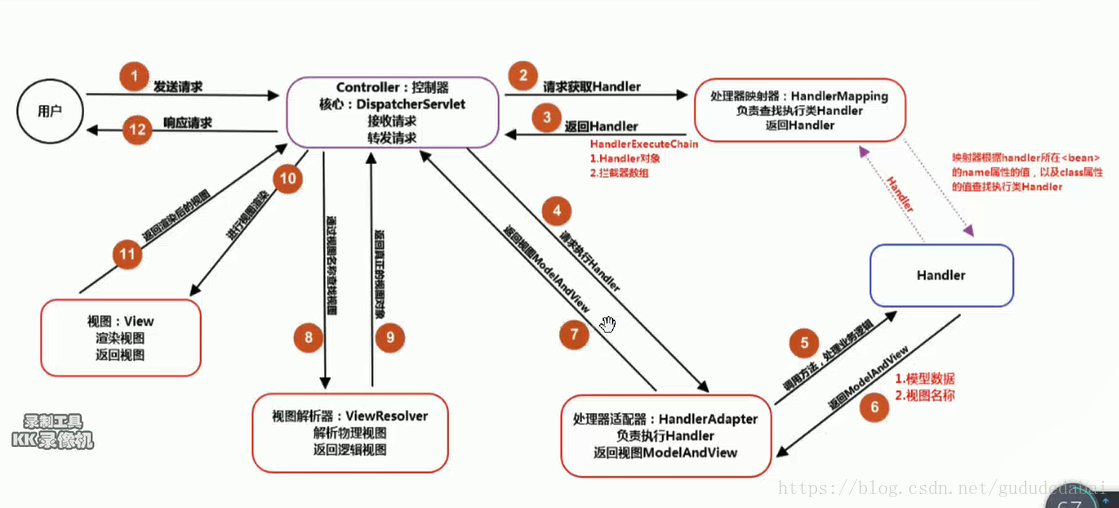
二叉查找树的特点：

1. 若任意节点的左子树不空，则左子树上所有结点的 值均小于它的根结点的值；
2. 若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
3. 任意节点的左、右子树也分别为二叉查找树。
4. 没有键值相等的节点（no duplicate nodes）。

作为平衡二叉查找树里面众多的实现之一，红黑树无疑是最简洁、实现最为简单的。红黑树通过引入颜色的概念，通过颜色这个约束条件的使用来保持树的高度平衡。作为平衡二叉查找树，旋转是一个必不可少的操作。通过旋转可以降低树的高度，在红黑树里面还可以转换颜色。

红黑树里面的插入和删除的操作比较难理解，这时要注意记住一点：操作之前红黑树是平衡的，颜色是符合定义的。在操作的时候就需要向兄弟节点、父节点、侄子节点借调和互换颜色，要达到这个目的，就需要不断的进行旋转。所以红黑树的插入删除操作需要不停的旋转，一旦借调了别的节点，删除和插入的节点就会达到局部的平衡（局部符合红黑树的定义），但是被借调的节点就不会平衡了，这时就需要以被借调的节点为起点继续进行调整，直到整棵树都是平衡的。在整个修复的过程中，插入具体的分为3种情况，删除分为4种情况。

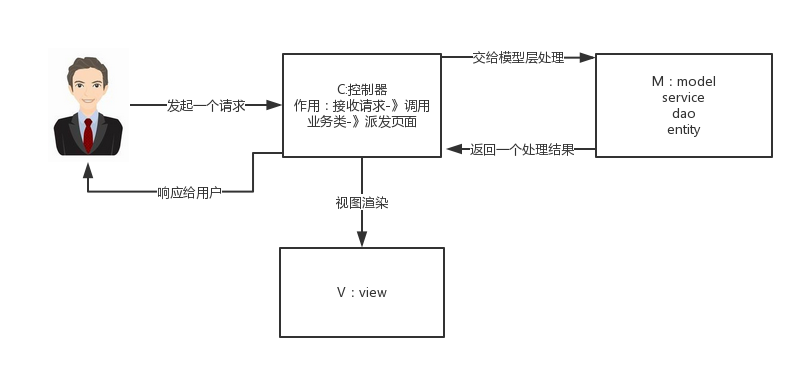
整个红黑树的查找，插入和删除都是O(logN)的，原因就是整个红黑树的高度是logN，查找从根到叶，走过的路径是树的高度，删除和插入操作是从叶到根的，所以经过的路径都是logN。



### 先来看一下什么是 MVC 模式

MVC 是一种设计模式.

**MVC 的原理图如下：**

[](https://camo.githubusercontent.com/0ce15ef13ad65c271362ec48b2ac573a6811c98b/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d31302d31312f36303637393434342e6a7067)

### SpringMVC 简单介绍

SpringMVC 框架是以请求为驱动，围绕 Servlet 设计，将请求发给控制器，然后通过模型对象，分派器来展示请求结果视图。其中核心类是 DispatcherServlet，它是一个 Servlet，顶层是实现的Servlet接口。

### SpringMVC 使用

需要在 web.xml 中配置 DispatcherServlet 。并且需要配置 Spring 监听器ContextLoaderListener

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<!-- 如果不设置init-param标签，则必须在/WEB-INF/下创建xxx-servlet.xml文件，其中xxx是servlet-name中配置的名称。 -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring/springmvc-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

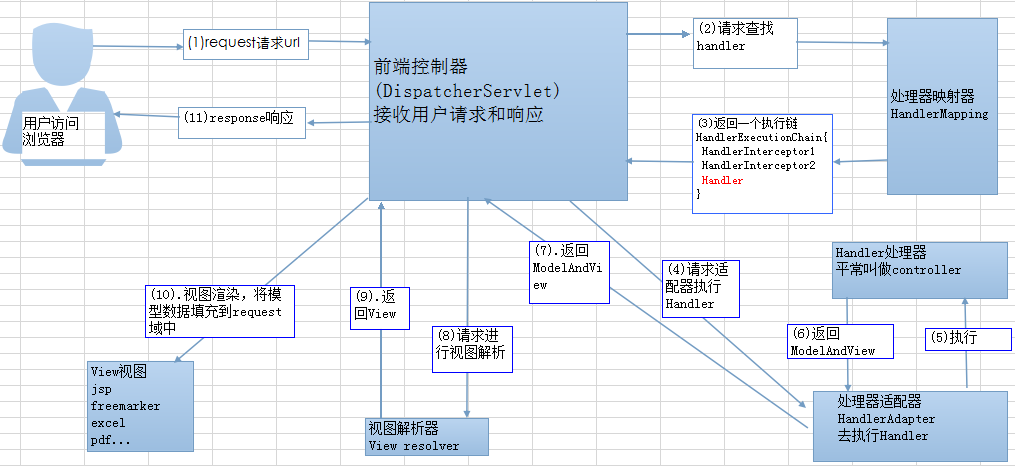
<url-pattern>/</url-pattern>

</servlet-mapping>

### SpringMVC 工作原理（重要）

**简单来说：**

客户端发送请求-> 前端控制器 DispatcherServlet 接受客户端请求 -> 找到处理器映射 HandlerMapping 解析请求对应的 Handler-> HandlerAdapter 会根据 Handler 来调用真正的处理器开处理请求，并处理相应的业务逻辑 -> 处理器返回一个模型视图 ModelAndView -> 视图解析器进行解析 -> 返回一个视图对象->前端控制器 DispatcherServlet 渲染数据（Moder）->将得到视图对象返回给用户

**如下图所示：** [](https://camo.githubusercontent.com/6889f839138de730fce5f6a0d64e33258a2cf9b5/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d31302d31312f34393739303238382e6a7067)

上图的一个笔误的小问题：Spring MVC 的入口函数也就是前端控制器 DispatcherServlet 的作用是接收请求，响应结果。

**流程说明（重要）：**

（1）客户端（浏览器）发送请求，直接请求到 DispatcherServlet。

（2）DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。

（3）解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。

（4）HandlerAdapter 会根据 Handler 来调用真正的处理器开处理请求，并处理相应的业务逻辑。

（5）处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。

（6）ViewResolver 会根据逻辑 View 查找实际的 View。

（7）DispaterServlet 把返回的 Model 传给 View（视图渲染）。

（8）把 View 返回给请求者（浏览器）

### SpringMVC 重要组件说明

**1、前端控制器DispatcherServlet（不需要工程师开发）,由框架提供（重要）**

作用：**Spring MVC 的入口函数。接收请求，响应结果，相当于转发器，中央处理器。有了 DispatcherServlet 减少了其它组件之间的耦合度。用户请求到达前端控制器，它就相当于mvc模式中的c，DispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，DispatcherServlet的存在降低了组件之间的耦合性。**

**2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供**

作用：根据请求的url查找Handler。HandlerMapping负责根据用户请求找到Handler即处理器（Controller），SpringMVC提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

**3、处理器适配器HandlerAdapter**

作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler 通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

**4、处理器Handler(需要工程师开发)**

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。 由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。

**5、视图解析器View resolver(不需要工程师开发),由框架提供**

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view） View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。 springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。 一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

**6、视图View(需要工程师开发)**

View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）

**注意：处理器Handler（也就是我们平常说的Controller控制器）以及视图层view都是需要我们自己手动开发的。其他的一些组件比如：前端控制器DispatcherServlet、处理器映射器HandlerMapping、处理器适配器HandlerAdapter等等都是框架提供给我们的，不需要自己手动开发。**

### DispatcherServlet详细解析

首先看下源码：

package org.springframework.web.servlet;

@SuppressWarnings("serial")

public class DispatcherServlet extends FrameworkServlet {

public static final String MULTIPART\_RESOLVER\_BEAN\_NAME = "multipartResolver";

public static final String LOCALE\_RESOLVER\_BEAN\_NAME = "localeResolver";

public static final String THEME\_RESOLVER\_BEAN\_NAME = "themeResolver";

public static final String HANDLER\_MAPPING\_BEAN\_NAME = "handlerMapping";

public static final String HANDLER\_ADAPTER\_BEAN\_NAME = "handlerAdapter";

public static final String HANDLER\_EXCEPTION\_RESOLVER\_BEAN\_NAME = "handlerExceptionResolver";

public static final String REQUEST\_TO\_VIEW\_NAME\_TRANSLATOR\_BEAN\_NAME = "viewNameTranslator";

public static final String VIEW\_RESOLVER\_BEAN\_NAME = "viewResolver";

public static final String FLASH\_MAP\_MANAGER\_BEAN\_NAME = "flashMapManager";

public static final String WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE = DispatcherServlet.class.getName() + ".CONTEXT";

public static final String LOCALE\_RESOLVER\_ATTRIBUTE = DispatcherServlet.class.getName() + ".LOCALE\_RESOLVER";

public static final String THEME\_RESOLVER\_ATTRIBUTE = DispatcherServlet.class.getName() + ".THEME\_RESOLVER";

public static final String THEME\_SOURCE\_ATTRIBUTE = DispatcherServlet.class.getName() + ".THEME\_SOURCE";

public static final String INPUT\_FLASH\_MAP\_ATTRIBUTE = DispatcherServlet.class.getName() + ".INPUT\_FLASH\_MAP";

public static final String OUTPUT\_FLASH\_MAP\_ATTRIBUTE = DispatcherServlet.class.getName() + ".OUTPUT\_FLASH\_MAP";

public static final String FLASH\_MAP\_MANAGER\_ATTRIBUTE = DispatcherServlet.class.getName() + ".FLASH\_MAP\_MANAGER";

public static final String EXCEPTION\_ATTRIBUTE = DispatcherServlet.class.getName() + ".EXCEPTION";

public static final String PAGE\_NOT\_FOUND\_LOG\_CATEGORY = "org.springframework.web.servlet.PageNotFound";

private static final String DEFAULT\_STRATEGIES\_PATH = "DispatcherServlet.properties";

protected static final Log pageNotFoundLogger = LogFactory.getLog(PAGE\_NOT\_FOUND\_LOG\_CATEGORY);

private static final Properties defaultStrategies;

static {

try {

ClassPathResource resource = new ClassPathResource(DEFAULT\_STRATEGIES\_PATH, DispatcherServlet.class);

defaultStrategies = PropertiesLoaderUtils.loadProperties(resource);

}

catch (IOException ex) {

throw new IllegalStateException("Could not load 'DispatcherServlet.properties': " + ex.getMessage());

}

}

/\*\* Detect all HandlerMappings or just expect "handlerMapping" bean? \*/

private boolean detectAllHandlerMappings = true;

/\*\* Detect all HandlerAdapters or just expect "handlerAdapter" bean? \*/

private boolean detectAllHandlerAdapters = true;

/\*\* Detect all HandlerExceptionResolvers or just expect "handlerExceptionResolver" bean? \*/

private boolean detectAllHandlerExceptionResolvers = true;

/\*\* Detect all ViewResolvers or just expect "viewResolver" bean? \*/

private boolean detectAllViewResolvers = true;

/\*\* Throw a NoHandlerFoundException if no Handler was found to process this request? \*\*/

private boolean throwExceptionIfNoHandlerFound = false;

/\*\* Perform cleanup of request attributes after include request? \*/

private boolean cleanupAfterInclude = true;

/\*\* MultipartResolver used by this servlet \*/

private MultipartResolver multipartResolver;

/\*\* LocaleResolver used by this servlet \*/

private LocaleResolver localeResolver;

/\*\* ThemeResolver used by this servlet \*/

private ThemeResolver themeResolver;

/\*\* List of HandlerMappings used by this servlet \*/

private List<HandlerMapping> handlerMappings;

/\*\* List of HandlerAdapters used by this servlet \*/

private List<HandlerAdapter> handlerAdapters;

/\*\* List of HandlerExceptionResolvers used by this servlet \*/

private List<HandlerExceptionResolver> handlerExceptionResolvers;

/\*\* RequestToViewNameTranslator used by this servlet \*/

private RequestToViewNameTranslator viewNameTranslator;

private FlashMapManager flashMapManager;

/\*\* List of ViewResolvers used by this servlet \*/

private List<ViewResolver> viewResolvers;

public DispatcherServlet() {

super();

}

public DispatcherServlet(WebApplicationContext webApplicationContext) {

super(webApplicationContext);

}

@Override

protected void onRefresh(ApplicationContext context) {

initStrategies(context);

}

protected void initStrategies(ApplicationContext context) {

initMultipartResolver(context);

initLocaleResolver(context);

initThemeResolver(context);

initHandlerMappings(context);

initHandlerAdapters(context);

initHandlerExceptionResolvers(context);

initRequestToViewNameTranslator(context);

initViewResolvers(context);

initFlashMapManager(context);

}

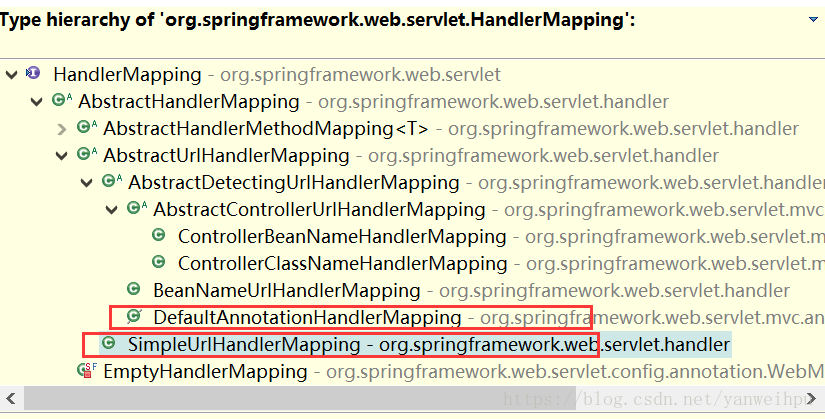
}

DispatcherServlet类中的属性beans：

* HandlerMapping：用于handlers映射请求和一系列的对于拦截器的前处理和后处理，大部分用@Controller注解。
* HandlerAdapter：帮助DispatcherServlet处理映射请求处理程序的适配器，而不用考虑实际调用的是 哪个处理程序。- - -
* ViewResolver：根据实际配置解析实际的View类型。
* ThemeResolver：解决Web应用程序可以使用的主题，例如提供个性化布局。
* MultipartResolver：解析多部分请求，以支持从HTML表单上传文件。-
* FlashMapManager：存储并检索可用于将一个请求属性传递到另一个请求的input和output的FlashMap，通常用于重定向。

在Web MVC框架中，每个DispatcherServlet都拥自己的WebApplicationContext，它继承了ApplicationContext。WebApplicationContext包含了其上下文和Servlet实例之间共享的所有的基础框架beans。

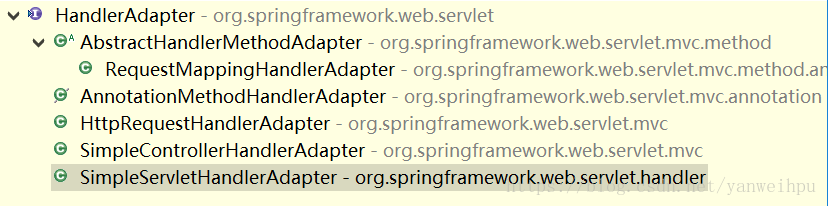
**HandlerMapping**

[](https://camo.githubusercontent.com/fa01f5e13dd0380697bfcac4116d5d3a3c456dd2/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d31302d31312f39363636363136342e6a7067)

HandlerMapping接口处理请求的映射HandlerMapping接口的实现类：

* SimpleUrlHandlerMapping类通过配置文件把URL映射到Controller类。
* DefaultAnnotationHandlerMapping类通过注解把URL映射到Controller类。

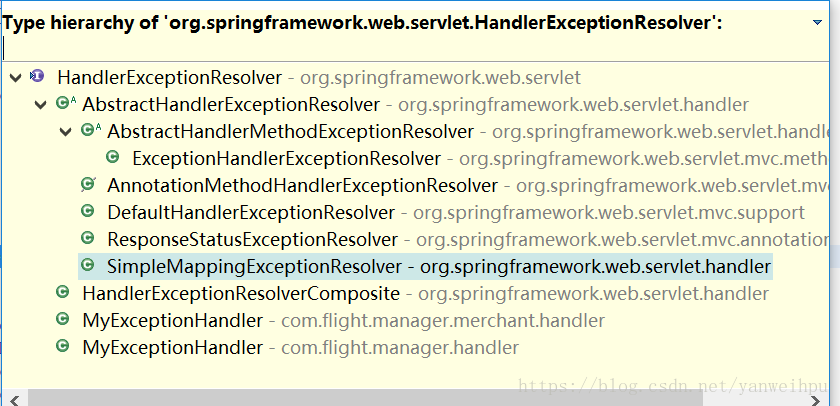
**HandlerAdapter**

[](https://camo.githubusercontent.com/ef6a14e4c8748d0e281f69f280461ab254925c2b/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d31302d31312f39313433333130302e6a7067)

HandlerAdapter接口-处理请求映射

AnnotationMethodHandlerAdapter：通过注解，把请求URL映射到Controller类的方法上。

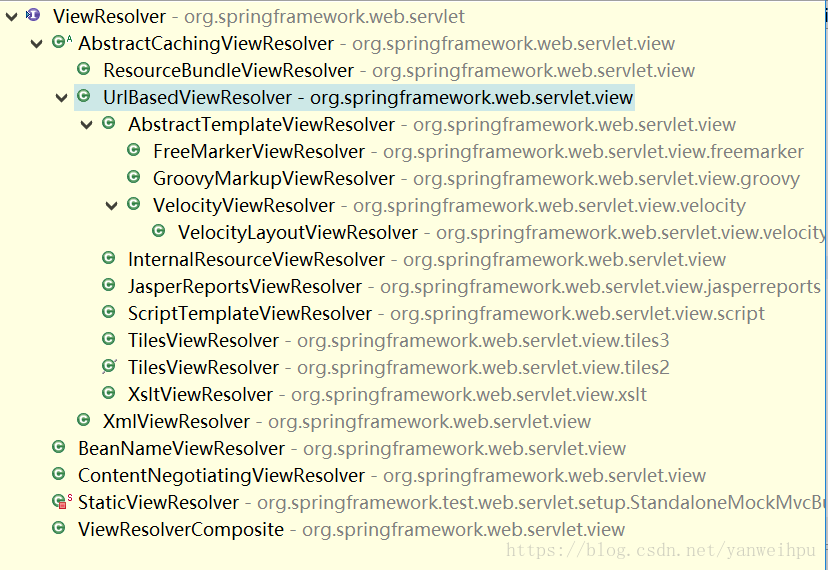
**HandlerExceptionResolver**

[](https://camo.githubusercontent.com/f141ed48dc7da1b82884970d2c398494a2afeeca/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d31302d31312f35303334333838352e6a7067)

HandlerExceptionResolver接口-异常处理接口

* SimpleMappingExceptionResolver通过配置文件进行异常处理。
* AnnotationMethodHandlerExceptionResolver：通过注解进行异常处理。

**ViewResolver**

[](https://camo.githubusercontent.com/7c2907a6ca88d25ebea98d488318f72b89de5176/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d31302d31312f34393439373237392e6a7067)

ViewResolver接口解析View视图。

UrlBasedViewResolver类 通过配置文件，把一个视图名交给到一个View来处理。