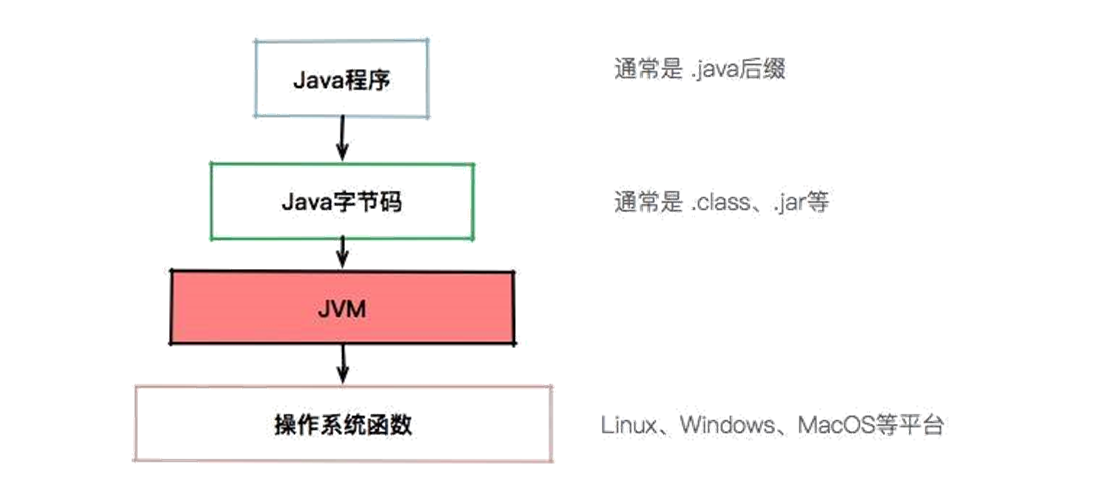
JVM全称java virtual machine，java虚拟机，能够识别.class文件，并且能够解析它的指令，最终调用操作系统的函数，完成想要的操作；有了jvm，java就可以实现跨平台，jvm只需要保证能够正确的执行.class文件，就可以允许在诸如linux，windows，macos等平台上, JVM 与操作系统之间的关系：JVM 上承开发语言，下接操作系统，它的中间接口就是字节码。

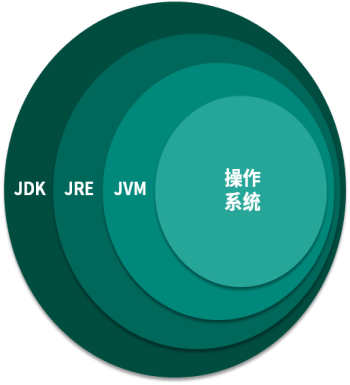


Javac将.java文件编译成class文件；

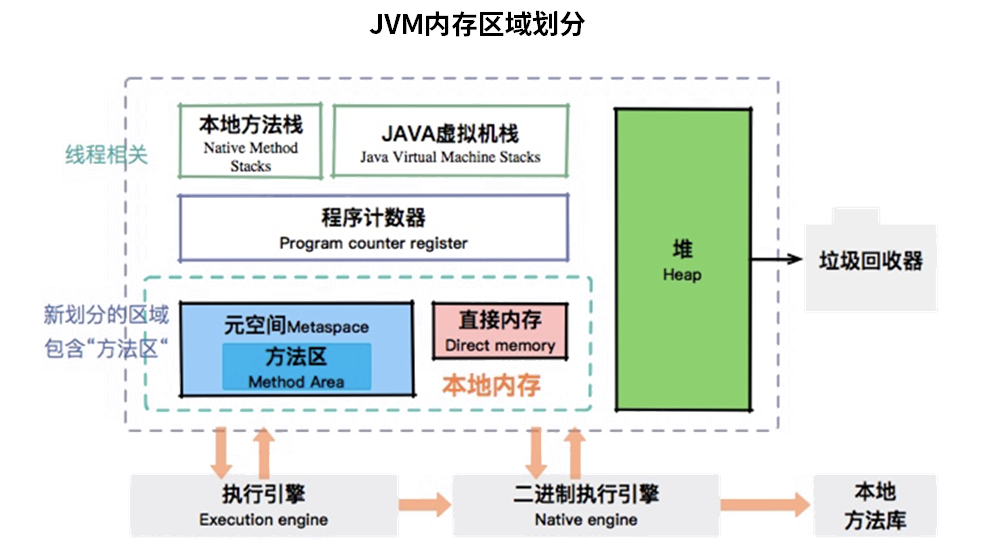
Jvm，jre,jdk的关系

Jvm是运行java程序的核心，但是仅仅只有jvm是不能实现一次编译到处运行的；还需要一个基本的类库，比如怎么操作文件，怎么连接网络等，java提供了jvm运行所需要的全部类库；所以jvm加上运行时需要的类库就组成的jre（java Runtime environment），要运行java程序只需要一个jre就可以了

对于jdk（java development kit）,更加庞大些，他还包括了开发中一些工具，比较javac，java，jar;它们都是java开发的核心;这三者之间的关系，jdk > jre > jvm



# Jvm内存布局



JVM内存区域划分如图所示，从图中我们可以看出：

JVM堆中的数据是共享的，是占用内存最大的一块区域。

可以执行字节码的模块叫作执行引擎。

执行引擎在线程切换时怎么恢复？依靠的就是程序计数器。

JVM 的内存划分与多线程是息息相关的。像我们程序中运行时用到的栈，以及本地方法栈，它们的维度都是线程。

本地内存包含元数据区和一些直接内存。

**虚拟机栈**

在方法执行时，参与计算的数据会频繁的入栈和出栈，栈里每条数据就是栈帧，每个方法被调用的时候都会创建一个栈帧并且入栈，一旦完成调用就出栈；栈帧包含局部变量表，操作数栈，动态链接，返回地址；

对于 JVM 来说，程序就是存储在方法区的字节码指令，而 returnAddress 类型的值就是指向特定指令内存地址的指针。所有的字节码指令，其实都会抽象成对栈的入栈出栈操作。执行引擎只需要傻瓜式的按顺序执行，就可以保证它的正确性

栈帧，对应着方法；第二层是方法的执行，对应着操作数。注意千万不要搞混了。

**本地方法栈**

本地方法栈是和虚拟机栈非常相似的一个区域，它服务的对象是 native 方法。你甚至可以认为虚拟机栈和本地方法栈是同一个区域，这并不影响我们对 JVM 的了解。

**程序计数器**

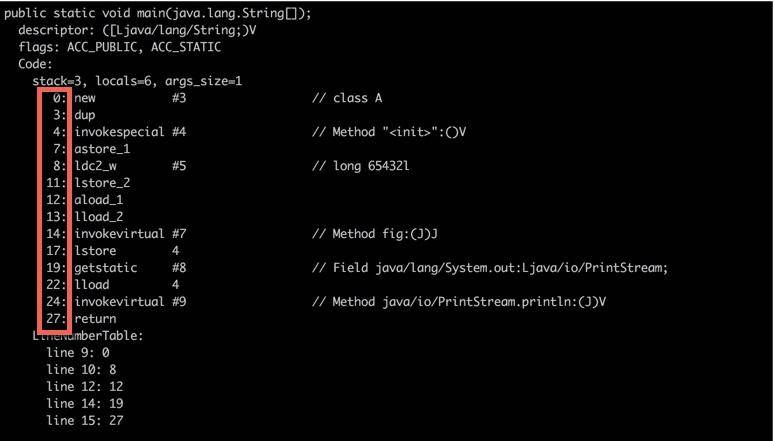
是一块较小的内存空间，作用是当前线程运行的字节码的行号指示器，也就是存放的是当前线程执行的进度

如果我们的程序在线程之间进行切换，凭什么能够知道这个线程已经执行到什么地方呢？

既然是线程，就代表它在获取CPU时间片上，是不可预知的，需要有一个地方，对线程正在运行的点位进行缓冲记录，以便在获取 CPU 时间片时能够快速恢复。

程序计数器是随着线程而产生的，与虚拟机栈配合完成计算操作；还存储了当前正在运行的流程，包括执行的指令，跳转，分支，循环，异常处理等

我们可以看一下程序计数器里面的具体内容。下面这张图，就是使用javap命令输出的字节码。大家可以看到在每个opcode前面，都有一个序号。就是图中红框中的偏移地址，你可以认为它们是程序计数器的内容



当执行的是java方法时，计数器存放的就是指令地址，执行native方法时，则为空

**堆**

堆是jvm内存中最大的一块区域，几乎所有的对象都在堆中，我们常说的垃圾回收操作的就是这块区域；

堆空间是在程序启动是就创建的

对于一个对象的创建，是在堆上分配还是栈上分配，这和2个方面有关：对象的类型和java类中存在的位置，

Java中的对象分为基本数据类型对象和普通对象，对于普通对象首先会在堆上创建对象，然后在其他地方使用其实就是它的引用，比如把这个引用保存到虚拟机栈的局部变量表中；

对于基本数据类型有2种情况

在方法体中声明基本数据类型的对象，它会直接在栈上分配，其他情况是在堆上分配

对于基本类型的数组，是直接在堆上分配，数组不是基本数据类型

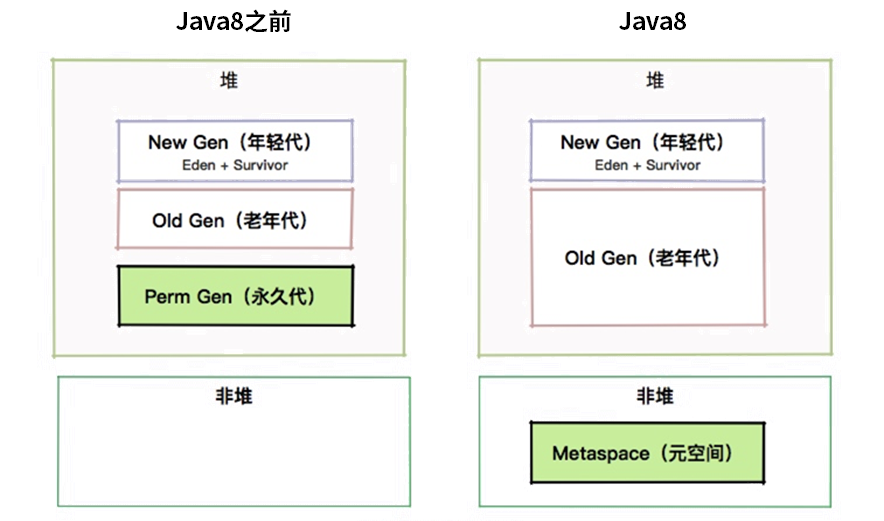
堆上的数据是线程共享的，如果多线程访问会涉及到数据安全和同步的问题

元空间（方法区）

[JDK8的JVM内存结构，元空间替代永久代成为方法区及常量池的变化](https://www.cnblogs.com/shen-qian/p/11277085.html)

在Java8之前，类的信息是放在一个叫Perm区（永久代）的内存里面的。更早版本，甚至String.intern相关的运行时常量池也放在这里。这个区域有大小限制，很容易造成JVM内存溢出，从而造成 JVM 崩溃。Java 8开始Perm区（永久代）就被废除了，取而代之的就是MetaSpace，现在元空间是在非堆上，使用非堆可以使用操作系统的内存，JVM 不会再出现方法区的内存溢出；但是，无限制的使用会造成操作系统的死亡。所以，一般也会使用参数 -XX:MaxMetaspaceSize 来控制大小。

方法区，作为一个概念，依然存在。它的物理存储的容器，就是 Metaspace。这个区域存储的内容，包括：类的信息、常量池、方法数据、方法代码



# JVM类加载机制

类加载过程

加载，验证，准备，解析，初始化

加载：主要是将.class文件加载到方法区，加载类主要是找到并加载类的二进制数据，

验证：不是所有的.class文件都能够加载，那样太不安全，容易受到恶意代码攻击

目的是区别class文件的字节流中包含的信息符合java虚拟机规范，不会给虚拟机带来危害；因为class文件并不一定是有java源码编译生成的，可以通过其他途径生成，所以如果虚拟机不进行检查验证的话，那么就可能引入有害的字节流而导致系统崩溃

准备：将类变量（静态变量）分配内存，并初始化默认值，此时还没有个实例对象分配内存，所以这些操作是在方法区上进行的,final修饰的static变量会赋值

解析：将符号引用替换为直接引用的过程；符号引用是一种定义，可以是任何字面上的含义（用来描述指向的目标），而直接引用就是直接指向目标的指针、相对偏移量，直接引用的对象都存在于内存中，你可以把通讯录里的女友手机号码，类比为符号引用，把面对面和你吃饭的人，类比为直接引用。

**解析阶段负责把整个类激活，串成一个可以找到彼此的网**，过程不可谓不重要。那这个阶段都做了哪些工作呢？大体可以分为：

类或接口的解析

类方法解析

接口方法解析

字段解析

初始化

这阶段才是执行字节码

static 语句块，只能访问到定义在 static 语句块之前的变量。所以下面的代码是无法通过编译的。JVM 会保证在子类的初始化方法执行之前，父类的初始化方法已经执行完毕。

初始化阶段就是执行类构造器<clint>()方法的过程，

<clinit>()方法是编译器自动收集类的所有类（静态）变量的赋值动作和所有静态语句块中的语言合并生成的，编译器收集的顺序是由语句在源文件中的顺序决定的，静态语句块只能访问定义在该语句快之前的变量，定义在静态语句块之后的变量，在前面的语句块可以赋值，但是不能访问；

而对象初始化就不一样了。通常，我们在 new 一个新对象的时候，都会调用它的构造方法，就是 <init>，用来初始化对象的属性。每次新建对象的时候，都会执行

对象的创建

Java语言层面上的创建使用new关键字创建对象就可以了

虚拟机中的创建过程：虚拟机遇到一个new指令时，

1首先检查这个指令的参数是否能在定位到一个类的符号引用（可以是全类名），并且检查这个符号引用对应的类是否已经被加载、解析和初始化过；如果没有那么先要进行类初始化过程；

2类加载检查过后，虚拟机就会为新对象分配内存空间，就是在堆内存中划分一块确定大小的内存；分配内存的方法有指针碰撞和空闲列表

指针碰撞：当堆内存空间绝对规整时，所有用过的内存分配到一边，空闲的在另一边，中间放着一个指针作为分界点指示器，分配内存就是将指针向空闲的一边移动对象大小相等的距离

空闲列表：堆内存不规整，使用的和空闲的内存交错，不能使用指针碰撞；虚拟机会维护一个空闲内存表，记录那些内存块时可用的；虚拟机就从空闲表中找到一个合适大小的区域划分给对象

选择哪种方式是由java堆内存空间是否规则来决定的，java堆是否规则又由采用的垃圾回收算法是否有压缩整理功能决定；serial，parnew等带有compact过程的收集器时，系统采用指针碰撞；使用CMS这种基于Mark-Sweep（标记清除）算法的通常采用空闲列表

3内存空间分配完成后，虚拟机需要将分配到的空间都初始化为零值（不包括对象头）

4然后对对象进行必要的设置，例如这个对象是哪个实例，如何才能找到类的元数据信息，对象的哈希码，对象的gc分代年龄等信息；这些信息存放在对象头中

上面这些工作完成后，从虚拟机角度来看，一个新对象就生成了；

5但是从java程序的角度来看，对象创建才刚开始----<init>方法还没有执行，字段还都是零；执行new指令后一般都会执行<init>方法，把对象按程序员的意愿进行初始化，这样一个真正的对象就生成了

**对象内存布局**

对象在内存中存储的布局可以分为3个部分：对象头（），实例数据（）和对象填充（）

**对象头**

Hotspot虚拟机的对象头包含2部分信息：第一部分用于存储对象自身运行时的数据，如哈希码，GC分代年龄，锁状态标志，线程持有锁，偏向线程Id,偏向时间戳，这部分数据在32位和64位虚拟机中分别是32bit和64bit,这部分叫Mark word；markword是一个非固定大小的数据结构，一边能够存储更多的信息，它会根据对象状态复用自己的存储空间；

对象的另一部分是类型指针，即对象指向他的类元数据指针，虚拟机通过这个指针来确定对象是哪个类的实例，但是并不是所有对象有这个指针，也就是说查询对象元数据信息并不一定要经过对象本身；比较java数组对象：对象头必须有一块记录数组长度的数据，虚拟机可以通过普通对象的元信息来确定java对象的大小，但是从数组对象的元数据中不能确定的大小

**实例数据：**

这部分是对象真正存储的有效信息，也就是程序代码中定义的所有字段，无论是父类继承的还是在子类中 定义的都会记录下来；这部分的存储顺序会受到虚拟机分配参数策略和字段在java代码中的定义顺序的影响；相同宽度的字段会被分配到一起（long/double,shot/char,int,byte/Boolean,）

**对象填充：**没有实际意义，用来起占位符的作用；虚拟机中对象在内存中的位置必须是8位的整数倍，如果一个对象地址不是8的整数倍，那么对象填充就将剩余的位置填起来

# 类加载器

Bootstrap classloader

这是加载器中的大 Boss，任何类的加载行为，都要经它过问。它的作用是加载核心类库，也就是 rt.jar、resources.jar、charsets.jar 等。

当然这些 jar 包的路径是可以指定的，-Xbootclasspath 参数可以完成指定操作。

这个加载器是 C++ 编写的，随着 JVM 启动。

Extention classloader

扩展类加载器，主要用于加载 lib/ext 目录下的 jar 包和 .class 文件。同样的，通过系统变量 java.ext.dirs 可以指定这个目录. 这个加载器是个 Java 类，继承自 URLClassLoader。

App classloader

这是我们写的 Java 类的默认加载器，有时候也叫作 System ClassLoader。一般用来加载 classpath 下的其他所有 jar 包和 .class 文件

Consum classloader

自定义类加载器，支持一个自定义的扩展功能

双亲委派模式

双亲委派机制的意思是除了顶层的启动类加载器以外，其余的类加载器，在加载之前，都会委派给它的父加载器进行加载。这样一层层向上传递，直到祖先们都无法胜任，它才会真正的加载。

这个模型的好处在于Java类有了一种优先级的层次划分关系。比如Object类，这个毫无疑问应该交给最上层的加载器进行加载，即使是你覆盖了它，最终也是由系统默认的加载器进行加载的。如果没有双亲委派模型，就会出现很多个不同的 Object 类，应用程序会一片混乱。

# GC

## 回收对象

Gc不受程序的控制，满足条件后自动触发

一个对象总能够找到他的祖先，如果找到最后祖先不存在，那么这个对象就会被清理掉

判断对象是否能被回收算法：可达性分析算法：从GC roots对象为起点，从这个节点向下搜索，搜索经过的路径称为引用链，当一个对象到GC ROOT没有任何引用链相连时则可以证明这个对象是不可使用的，那么这个对象可以被回收；

GCRoots对象包括：

栈中的引用对象（栈帧中本地变量表的引用）

方法区中静态属性引用的对象（static修饰的变量）

常量引用

Jvm专门有一个线程来做gc,当内存空间达到一定条件时，会自动触发；目前建成使用的垃圾回收器是cms和g1(jdk11版本才有的收集器)

## 回收算法

### 标记清理

1. 标记：标记出那些是可以追溯到gc root的对象
2. 清理：清理那么没有被标记的对象

效率一般，缺点是会造成内存碎片问题。

### 复制算法：

将内存分为2块，只使用其中的一块，当内存空间不够时，将存活的对象转移到另一块内存区域，清除当前使用的区域即可；

效率最高的，缺点是会造成一定的空间浪费

这种算法实现简单，运行效率很高，但是只使用一半内存，太浪费资源了；当存活的对象过多时，copying算法的效率就会大大降低

### 标记整理算法（回收老年代）：

标记过程和标记清除算法一样，但是后续不是直接回收对象，而是让存活的对象向一端移动然后清理端边界以外的内存，

就是移动所有存活的对象，且按照内存地址顺序依次排列，然后将末端内存地址以后的内存全部回收。

效率比前两者要差，但没有空间浪费，也消除了内存碎片问题。

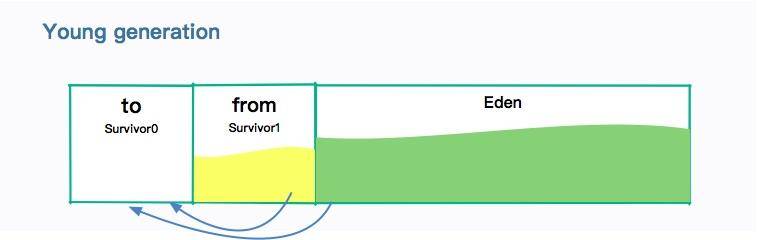
### 分代收集算法

根据对象的存活周期，将内存划分为几块；一般把java堆分为老年代和新生代这样可以根据各个年代的特点来使用不同算法回收；

新生代中每次垃圾回收都会有大批的对象被回收，只有少量存活，使用复制算法；

老年代因为对象的存活率高，没有额外的空间进行分配担保，就必须使用标记清除算法或者标记整理算法

**年轻代使用的垃圾回收算法是复制算法**。因为年轻代发生 GC 后，只会有非常少的对象存活，复制这部分对象是非常高效的。



年轻代分为：一个伊甸园空间（Eden ），两个幸存者空间（Survivor ）

当年轻代中的 Eden 区分配满的时候，就会触发年轻代的 GC（Minor GC）

在Eden区执行了第一次GC之后，存活的对象会被移动到其中一个Survivor分区（以下简称from）；

Eden区再次GC，这时会采用**复制算法**，将 Eden 和 from 区一起清理。存活的对象会被复制到 to 区；

接下来，只需要清空Eden和 from 区就可以了，然后将survivor的from和to进行切换

所以在这个过程中，总会有一个 Survivor 分区是空置的。Eden、from、to 的默认比例是 8:1:1，所以只会造成 10% 的空间浪费。

老年代

**老年代一般使用 “标记-整理”算法**，

因为老年代的对象存活率一般是比较高的，空间又比较大，拷贝起来并不划算，还不如采取就地收集的方式

对象进入老年代的途径

1. 升级

对象经过一次monitor gc，存活下来的对象年龄+1，当年龄到达设定的阈值后，就会升级为老年代

1. 分配担保

看一下年轻代的图，每次存活的对象，都会放入其中一个幸存区，这个区域默认的比例是 10%。但是我们无法保证每次存活的对象都小于 10%，当 Survivor 空间不够，就需要依赖其他内存（指老年代）进行分配担保。这个时候，对象也会直接在老年代上分配。

1. 大对象直接在老年代分配

超出某个大小的对象将直接在老年代分配。这个值是通过参数 -XX:PretenureSizeThreshold 进行配置的。默认为 0，意思是全部首选 Eden 区进行分配。

1. 动态对象年龄判断

有的垃圾回收算法，并不要求age必须达到15才能晋升到老年代，它会使用一些动态的计算方法。比如，如果幸存区中相同年龄对象大小的和，大于幸存区的一半，大于或等于age的对象将会直接进入老年代

Minor GC：发生在年轻代的 GC。

触发条件  
1、Eden区域满

2、新创建的对象大小 > Eden所剩空间

Major GC：发生在老年代的 GC。

Major GC清理Tenured区，用于回收老年代，出现Major GC通常会出现至少一次Minor GC。

Full GC：全堆垃圾回收。比如 Metaspace 区引起年轻代和老年代的回收。System.gc()

## 垃圾回收器

年轻代垃圾回收器

**Serial收集器**（新生代收集器）：

处理 GC 的只有一条线程，并且在垃圾回收的过程中暂停一切用户线程

这是一个单线程收集器，只会使用一个cpu或者一个线程去收集，进行收集时必须暂停其他所有的线程直到这个收集器执行完成，收集工作由虚拟机发起，会将用户正在执行的正常的线程暂停，这就会影响应用的响应时间；它是虚拟机运行在client模式下的默认新生代收集器，简单而且高效（与其他单线程收集器相比），对于单个cpu来说，serial收集器没有线程交互的开销只做垃圾回收，就可以获得最高的单线程收集效率

**ParNew收集器**（新生代收集器）：

ParNew 是 Serial 的多线程版本。由多条 GC 线程并行地进行垃圾清理。清理过程依然要停止用户线程。

其实就是serial收集器的多线程版，除了使用多线程收集垃圾之外，其余行为包括serial收集器可用的控制参数，收集算法，stop world,对象分配规则，回收策略都和serial收集器一样，它是在server模式下首选的新生代收集器，（目前只它能够和CMS收集器配合使用），在单线程环境中收集效果没有serial收集器好，因为多线程之间的存在交互的开销；

ParNew收集器是使用-XX:UseConcMarkSweepGC选项后的默认收集器，也可以使用-XX:ParNewGC选项来强制指定它，

**Parallel scavenge收集器**（新生代收集器）：

另一个多线程版本的垃圾回收器。它与 ParNew 的主要区别是：

Parallel Scavenge：追求 CPU 吞吐量，能够在较短时间内完成指定任务，适合没有交互的后台计算。弱交互强计算。

ParNew：追求降低用户停顿时间，适合交互式应用。强交互弱计算。

它是使用复制算法又是并行的多线程收集器；这个收集器的目标是到达一个可控制的吞吐量，所谓吞吐量是cpu用于运行代码的时间与cpu总消耗时间的比值，吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）

停顿时间越短就越适合需要和用户交互的程序，良好的响应时间提升了用户的体验；而高效的吞吐量则可以高效的利用cpu时间，尽快的完成运算，主要适合在后台运算不需要太多的交互；

Parallel 收集器提供了2个参数来控制吞吐量：控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数和直接设置吞吐量大小的-XX:GCTimeRatio参数；

MaxPauseMillis运行的值是一个大于0的毫秒值，收集器尽可能的保证收集时间在设置值之内完成；并不是将这个值设置的越小收集速度就越快，GC停顿时间是牺牲吞吐量和新生代空间换取的；系统把新生代调小一些，收集300M新生代比收集500M快，这也导致了收集的频繁发生，原来10s收集一次，停顿100毫秒，线性变成5秒收集一次，停顿79毫秒，停顿时间是在减少，但是吞吐量也下降了；

GCTimeRatio参数的值应该是大于0小于100的整数，垃圾收集时间占总时间的比率，相当于吞吐量的总数，如果把参数设置为19，那运行最大GC时间就占总时间的5%

（即1/(1+19)）默认值是99 ，运行最大1%的收集时间

Parallel收集器也被称为吞吐量优先级，

Parallel收集器还有一个参数-XX:UserAdaptiveSizePolicy:这是一个开关参数，当打开这个参数后就不需要设置Eden区，survivor区的比例，晋升为老年代的等细节参数，虚拟机会根据运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大吞吐量，这种调节方式成为GC自适应的调整策略

**Serial old 收集器**（老年代收集器）：serial收集器的老年代版本，也是一个单线程收集器，使用标记-整理算法；这个收集器的意义主要是在client模式下给虚拟机使用；

**Parallel old 收集器**（老年代收集器）parallel收集器的老年代版本，使用多线程和标记整理算法

**CMS收集器**（CMS concurrent Mark sweep）是一个获取最短回收停顿时间为目标的收集器（响应速度快，系统停顿时间短），它在垃圾收集时使得用户线程和 GC 线程能够并发执行，因此在垃圾收集过程中用户也不会感到明显的卡顿；使用标记清除算法

运作过程：

1初始标记，仅仅是标记GC roots能够关联到的对象，速度很快（会暂停用户线程）

2并发标记，进行GC Roots tracing的过程，用于标记所有可达的对象

3重新标记，为并发标记期间因用户程序继续运行而导致标记发送变动的那部分对象进行标记，标记时间比初始时间长，但是比并发标记时间短（会暂停用户线程）

4并发清除

不建议在使用cms收集器的时候把堆空间设置过大，使用标记清除算法，效率较低（标记和清除）

初始标记和重新标记需要stop the world（也就是停止用户线程），耗时最长的并发标记和并发清除可以和用户线程一起工作，从总体上来看，CMS收集器回收过程与用户线程是并发执行的；

这个收集器的优点：并发收集，低停顿

缺点

1. 对cpu资源比较敏感:并发阶段，它虽然不会导致用户线程停顿，但是会占用一部分线程而导致应用程序变慢，总吞吐量降低
2. 无法处理浮动垃圾：由于CMS清理阶段用户线程正在运行，所有不断的产生新的垃圾对象，这部分出现在标记过后，就不会在本次垃圾收集被回收，只有在下次被回收，这部分垃圾对象称为浮动垃圾
3. 使用标记清除法：那么清理完成后就会出现不连续空间，不利于分配大对象空间；在没有足够大的连续空间来分配大对象是会触发full gc

**G1收集器**(Garbage-first)

并行并发：利用多cpu，多核的环境优势，利用多cpu来缩短stop-the-world的时间；g1收集器可用在收集的同时让执行java程序继续执行

分代收集：能够处理新对象，已存活一段时间的对象，经过多次gc存活下来的对象

空间整合：整体采用标记整理算法，局部采用复制算法，保证内存的规则，不会有不连续的空间

可预测停顿：能够让使用者在明确的M毫秒内消耗在回收的时间不会超过N毫秒

G1收集器将java堆划分为多个大小相同的区域虽然保留了老年代和新生代，但是新生代和老年代不在是隔离的，他们都是一块区域的集合；g1收集器之所以能够可预测停顿时间，是因为它可以有计划的避免在整个java堆中进行全区域的收集；g1收集器跟踪各个区域垃圾堆的大小，在后台维护一张优先级列表，每次根据允许的时间，优先回收价值最大的区域；这种使用region划分内存空间及优先级的区域回收方式，保证了G1收集器在短时间内获取更高的收集效率

怎么判断各个Region中的对象是否可达

回收新生代是补不得不扫描老年代，Minor gc的效率将会减少；

在g1收集器中，Region之间的对象引用以及其他收集器的新生代和老年代的对象引用，虚拟机使用Remembered Set来避免全堆扫描；每个region都有一个与之对应的Remembered Set，虚拟机发现程序对reference类型数据进行操作时，会产生一个write barrier暂时中断写操作，判断reference类型是否处于不同region之中，如果是就记录到Remember set中，在gc根节点的枚举范围中加入Remembered set即可保证不对全堆扫描也不会有遗漏；

如果不计算维护Remembered set的操作，g1收集器包含以下几个步骤：

1. 初始标记：标记GC roots能够直接关联的对象，并且修改TAMS(Next top at Mark start)的值,让下一阶段用户程序并发运行时，能在正确可用的region中创建对象，这阶段需要停顿线程，但是时间很短；
2. 并发标记：冲GC roots开始对堆中的对象进行可达性分析，找到存活的对象，耗时较长，可与用户程序并发执行
3. 最终标记：修正并发标记期间因程序继续执行而产生标记发送变动的那部分标记记录；虚拟机将这段时间对象变化记录在线程remenber set logs里面；最终标记会把Remember set log中的数据合并到Remembered set中，这阶段需要停顿线程，但是可以并发执行，
4. 筛选回收：首先对各个region的回收价值和成本进行排序，根据用户期望的GC停顿时间来制定回收计划，

Gc配置参数

-XX:+UseSerialGC年轻代和老年代都用串行收集器

-XX:+UseParNewGC年轻代使用ParNew，老年代使用SerialOld

-XX:+UseParallelGC 年轻代使用 ParallerGC，老年代使用 Serial Old

-XX:+UseParallelOldGC 新生代和老年代都使用并行收集器

-XX:+UseConcMarkSweepGC，表示年轻代使用 ParNew，老年代的用 CMS

-XX:+UseG1GC 使用 G1垃圾回收器

-XX:+UseZGC 使用 ZGC 垃圾回收器

线上使用最多的垃圾回收器，就有 CMS 和 G1，以及 Java8 默认的 Parallel Scavenge

Stop the world

为了保证程序不会乱套，最好的办法就是暂停用户的一切线程。也就是在这段时间，你是不能 new 对象的，只能等待。表现在 JVM 上就是短暂的卡顿，什么都干不了。这个头疼的现象，就叫作 Stop the world。简称 STW。

标记阶段，大多数是要 STW 的。如果不暂停用户进程，在标记对象的时候，有可能有其他用户线程会产生一些新的对象和引用，造成混乱

现在的垃圾回收器，都会尽量去减少这个过程。但即使是最先进的 ZGC，也会有短暂的 STW 过程。我们要做的就是在现有基础设施上，尽量减少 GC 停顿

如何输出gc日志，以及如何使用这些日志的支撑工具解决问题。

GC 日志输出

-verbose:gc 打印 GC 日志

PrintGCDetails 打印详细 GC 日志

PrintGCDateStamps 系统时间，更加可读，PrintGCTimeStamps 是 JVM 启动时间

PrintGCApplicationStoppedTime 打印 STW 时间

PrintTenuringDistribution 打印对象年龄分布，对调优 MaxTenuringThreshold 参数帮助很大

Loggc 将以上 GC 内容输出到文件中

HeapDumpOnOutOfMemoryError OOM 时 Dump 信息，非常有用

HeapDumpPath Dump 文件保存路径

ErrorFile 错误日志存放路径

-verbose:gc

-XX:+PrintGCDetails

-XX:+PrintGCDateStamps

-XX:+PrintGCApplicationStoppedTime

-XX:+PrintTenuringDistribution

-Xloggc:/tmp/logs/gc\_%p.log

-XX:+HeapDumpOnOutOfMemoryError dump出hprof文件，用eclipse memory analyzer查看分析

-XX:HeapDumpPath=/tmp/logs

-XX:ErrorFile=/tmp/logs/hs\_error\_pid%p.log

-XX:-OmitStackTraceInFastThrow

Xms

# 引用类型

强引用：代码中普遍存在的形式：Object obj=new Object();

强引用Strongreferences当内存空间不足，系统撑不住了，JVM就会抛出OutOfMemoryError错误。

即使程序会异常终止，这种对象也不会被回收。这种引用属于最普通最强硬的一种存在，只有在和 GC Roots 断绝关系时，才会被消灭掉。

软引用

用于维护一些可有可无的对象，当内存足够时不会回收，当内存不够时会回收软引用对象，在jvm抛出oom时之前，清理回收软引用对象；如果软引用对象回收内存还不够那么就会抛出内存溢出

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。

将创建好的对象传给softreference

Object object=new Object();

SoftReference<Object> softRef =new SoftReference(object);通过get方法获取对象

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。

应用场景：实现内存敏感的缓存，就是如果还有空闲的空间就缓存对象，没有就清理掉，这样就保证在使用缓存的同时，不会耗尽内存

弱引用

是一种更加无用的引用方式，生命周期更短

当垃圾回收时，无论内存是否充足，弱引用关联的对象都会被回收

Object object=new Object();

WeakReference<Object> softRef = new WeakReference(object);

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。

应用场景：弱引用同样可用于内存敏感的缓存。

虚引用

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回收；

不会调用对象的方法和属性，虚引用主要用来跟踪对象被垃圾回收的活动。

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，就会在回收对象之前，把这个虚引用加入到与之关联的引用队列中。

应用场景：可用来跟踪对象被垃圾回收器回收的活动，当一个虚引用关联的对象被垃圾收集器回收之前会收到一条系统通知。

# OOM



OOM的起因

所以OOM到底是什么引起的呢？有几个原因：

内存的容量太小了，需要扩容，或者需要调整堆的空间。（-xms启动时占用的内存大小，-xmx最大的堆内存空间）

错误的引用方式，发生了内存泄漏。没有及时的切断与GCRoots的关系。比如线程池里的线程，在复用的情况下忘记清理 ThreadLocal 的内容。

接口没有进行范围校验，外部传参超出范围。比如数据库查询时的每页条数等。

对堆外内存无限制的使用。这种情况一旦发生更加严重，会造成操作系统内存耗尽。

排查oom

Jvm参数

-XX:+HeapDumpOnOutOfMemoryError dump出hprof文件，用eclipse memory analyzer查看分析，查找到那里的对象导致oom

-XX:HeapDumpPath=/tmp/logs 指定路径

Tomcat默认可以使用的内存为128MB，在较大型的应用项目中，这点内存是不够的，需要调大。

Windows下，在文件/bin/catalina.bat，Unix下，在文件/bin/catalina.sh的前面，增加如下设置：

JAVA\_OPTS='-Xms【初始化内存大小】

-Xmx【可以使用的最大内存】'

stop-the-world:

Stop-the-world会在任何一种GC算法中发生。Stop-the-world意味着 JVM 因为要执行GC而停止了应用程序的执行。当Stop-the-world发生时，除了GC所需的线程以外，所有线程都处于等待状态，直到GC任务完成。GC优化很多时候就是指减少Stop-the-world发生的时间。

minor gc：发生在新生代的gc（minor：次要的）。

major gc：发生在老年代的gc（major：主要的）也叫full gc。

如何确保新生代对象被老年代引用的时候不被gc？（查询老年代对象来确认对新生代对象的引用避免误回收）

机制：当老年代存活对象多时，每次minor gc查询老年代所有对象影响gc效率（因为gc stop-the-world），所以在老年代有一个write barrier（写屏障）来管理的card table（卡表），card table存放了所有老年代对象对新生代对象的引用。

所以每次minor gc通过查询card table来避免查询整个老年代，以此来提高gc性能。

Java线程消耗过大的cpu

1使用top命令，展示出cpu消耗高的线程

2使用 shift+H切换到线程模式 找到执行cpu高的线程号pid

3使用Jstack pid > p.txt 用jstack dump 线程的日志

1. 把线程号转换为16进制
2. 根据线程id在dump出的文件中找到对应的线程日志

jstack Dump 日志文件中的线程状态

**dump 文件里，值得关注的线程状态有**：

死锁，Deadlock（重点关注）

执行中，Runnable

等待资源，Waiting on condition（重点关注）

等待获取监视器，Waiting on monitor entry（重点关注）

暂停，Suspended

对象等待中，Object.wait() 或 TIMED\_WAITING

阻塞，Blocked（重点关注）

停止，Parked

根据什么来决定堆空间大下

有多少物理内存可以供JVM使用？

是部署多个JVM或者单个JVM？

一个机器上面只是部署一个JVM，且就一个应用使用？如果是这种情况，那么机器的所有物理内存可以供JVM使用。

一个机器上部署了多个JVM？或者一个机器上部署了多个应用？如果是这两个中的任何一种情况，你就必须要决定每一个JVM或者应用需要分配多少内存了。

根据什么来选择垃圾收集器

Jdk 11版本开始使用g1收集器

1.7或者1.8的版本一般在cms和parallel scavenger之间选择

低延迟选择cms

1，初始标记：简单标记下GC Roots能直接关联到的对象，需要“Stop The World“  
2，并发标记：进行GC Roots Tracing  
3，重新标记：修正并发标记期间用户程序继续运行而导致标记发生变动那一部分对 象标记记录，需要“Stop The World“  
4，并发清除

缺点无法清除浮动垃圾，并发清理阶段用户线程还在运行，程序自然会有新的垃圾产生，那么CMS将无法在这次收集中处理掉它们，只有等到下次

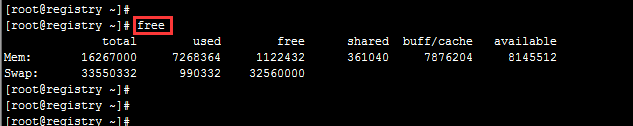
对CPU资源敏感。在并发阶段虽然不会导致用户线程停顿，但是会因为占用一部分线程（或者说CPU资源）而导致应用程序变慢，吞吐量降低，默认是启动（CPU数量＋3）／4的线程数。

CMS是基于标记－清除算法实现的，会产生大量的空间碎片

高吞吐量选择parallel scavenge

Linux使用free命令查看实际内存占用

 默认情况下，即在没有选项的情况下，"free"命令显示内存的使用信息。默认按照k（b）的计数单位统计。



* total：表示 总计物理内存的大小。
* used：表示 已使用多少。
* free：表示 可用内存多少。
* Shared：表示多个进程共享的内存总额。
* Buffers/cached：表示 磁盘缓存的大小。