# JVM

# 运行时数据区

* Java运行时数据区有
* 堆 ，本地方法栈，虚拟机栈，程序计数器，方法区（运行时常量池，属性和方法数据，代码区)
* Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。Java虚拟机所管理的内存将会包括以下集合运行时数据区域：

# 5.1 程序计数器（线程独享)

* 程序计数器（Program Counter Register)是一块很小的内存区域，它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中，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。
* 由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。因此，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，各条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。
* 如果线程正在执行的是Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法， 这个计数器的值为空。此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

# 5.2 虚拟机栈（线程独享)

* Java虚拟机栈也是线程私有的，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：每个方法在执行的同时都会创建一个栈帧（StackFrame)用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

# 5.3 本地方法栈（线程独享)

* 本地方法栈（Native Method Stack)与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行Java方法（字节码)服务，而本地方法栈为虚拟机使用到的Native方法服务。

# 5.4 Java堆

* 对大多数应用来说，Java堆是Java虚拟机所管理的内存中最大的一块。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。
* Java堆是垃圾收集器管理的主要区域。从内存回收的角度来看，由于现在收集器基本都采用分代收集算法，所以Java堆中还可以分为：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。从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer，TLAB)。
* Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。

# 5.5 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。
* “PermGen space”是方法区。不过方法区和“PermGen space”又有着本质的区别。前者是 JVM 的规范，而后者则是 JVM 规范的一种实现，并且只有 HotSpot 才有 “PermGen space”。
* HotSpot虚拟机将GC分代收集拓展至方法区，或者说使用永久代来实现方法区。这样的HotSpot的垃圾收集器可以像管理Java堆一样管理这部分内存，能够省去专门为方法区编写内存管理代码的工作。如果实现方法区属于虚拟机实现细节，不受虚拟机规范约束，但是用永久代实现方法区，并不是一个好主意，因为这样容易遇到内存溢出问题。
* 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就永久存在了。这区域的内存回收目标主要是针对常量池的回收和对类型的卸载。
* 在Java8中，永久代被删除，方法区的HotSpot的实现为Metaspace元数据区，不放在虚拟机中而放在本地内存中，存储类的元信息；
* 而将类的静态变量（放在Class对象中)和运行时常量池放在堆中。
* 为什么？
  + 1)移除永久代是为融合HotSpot JVM与 JRockit VM而做出的努力，因为JRockit没有永久代，不需要配置永久代。
  + 2)现实使用中易出问题
* 由于永久代内存经常不够用或发生内存泄露，出现异常java.lang.OutOfMemoryError: PermGen

## 运行时常量池

* 运行时常量池（Runtime Constant Pool)是方法区的一部分。Class文件中除了有关的描述信息外，还有一项信息是常量池，用于存放编译器生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。一般来说，除了保存Class文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池。
* 运行时常量池相对于Class文件常量池的一个重要特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，运行期间也可能将新的常量放入池中，比如String类的intern方法。

# 5.6 直接内存

* 直接内存（Direct Memory)并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但这部分内存也被频繁使用。JDK的NIO类，引入了一种基于通道和缓冲区的IO方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在堆中的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场合显著提高性能，避免了在Java堆和Native堆来回复制数据。
* 直接内存的分配不会受到Java堆大小的限制，但会受到本机总内存的限制。

# 对象

# 5.7 对象的创建

* 虚拟机遇到一条new指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，那就执行类加载过程。
* 在类加载检查通过后，虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。假设Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一块与对象大小相等的距离，这种分配方式称为指针碰撞。如果Java堆中的内存不是规整的，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为空闲列表（Free List)。选择哪种分配方式由Java堆是否规整决定，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。
* 除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。解决这个问题有两个方案，一种是对分配内存空间的动作进行同步处理，另一种是把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲TLAB。哪个线程分配内存，就在哪个线程的TLAB上分配，只有TLAB用完并分配新的TLAB时，才需要同步锁定。
* 内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值。接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何找到类的元数据等信息。这些信息存放在对象的对象头之中。上述工作完成后，从虚拟机的视角来看，一个新的对象已经产生，但从Java程序的视角来看，构造方法还没有执行，字段都还为0。所以执行new指令之后会接着执行构造方法等，这样一个对象才算真正产生出来。

# 5.8 对象的内存布局

* 在HotSpot虚拟机中，对象在内存中存储的布局可以分为3个区域：对象头（Header)、实例数据（Instance Data)和对齐填充（Padding)。
* 对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志等。对象头的另一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。
* 实例数据是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。无论是从父类继承的，还是在子类中定义的，都需要记录下来。相同宽度的字段总是被分配到一起，在这个前提下，在父类中定义的变量会出现在子类之前。
* 对齐填充并不是必然存在的，它仅仅起着占位符的作用，HotSpot虚拟机的自动内存管理系统要求对象起始地址必须是8字节的整数倍，即对象大小必须是8字节的整数倍，而对象头正好是8字节的整数倍。因此，当对象实例数据部分没有对齐时，就需要对齐填充来补全。

# 5.9 对象的访问定位

* Java程序需要通过栈上的Reference数据来操作堆上的具体对象。由于Reference类型在Java虚拟机规范中只规定了一个指向对象的引用，并没有定义这个引用应该通过何种方式来定位、访问堆中对象的具体位置，所以对象访问方式也是取决于虚拟机实现而定的。目标主流的方式有使用句柄和直接指针两种。
* 如果使用句柄访问的话，那么Java堆中将会划分出一块内存来作为句柄池，Reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息。
* 如果使用直接指针访问，那么Java堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而Reference中存储的直接就是对象地址。
* 使用句柄来访问的最大好处就是Reference中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而Reference本身不需要修改。
* 使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销。

# 内存溢出与内存泄露

* OOM ；方法区OOM时的异常；查看dump 文件，怎么查看，具体命令记得吗，答jstack 具体怎么用的

# 5.10 堆溢出

* Java堆用于存储对象实例，只要不断增加对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在对象数量达到最大堆的容量限制后就会产生OOM异常。

public class HeapOOM {  
 static class OOMObject{  
   
 }   
 public static void main(String[] args) {  
 List<OOMObject> list = new ArrayList<>();  
 while(true){  
 list.add(new OOMObject());  
 }  
 }  
}

* VM Options:
* -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
* java.lang.OutOfMemoryError: Java heap space
* Dumping heap to java\_pid15080.hprof ...
* Heap dump file created [28193498 bytes in 0.125 secs]
* Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
* at java.util.Arrays.copyOf(Arrays.java:3210)
  + at java.util.Arrays.copyOf(Arrays.java:3181)
  + at java.util.ArrayList.grow(ArrayList.java:261)
  + at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:235)
  + at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:227)
  + at java.util.ArrayList.add(ArrayList.java:458)
  + at cn.sinjinsong.se.review.oom.HeapOOM.main(HeapOOM.java:17)
* 要解决这个区域的异常，一般的手段是通过内存映像分析工具对dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要判断是出现来内存泄露还是内存溢出。前者的话要进一步通过工具查看泄露对象到GC Roots的引用链；后者的话可以调大虚拟机的堆参数（-Xms和-Xmx)，或者从代码上检查某些对象生命周期过长等。

# 5.11 栈溢出（虚拟机栈和本地方法栈)

* 对于HotSpot来说，虽然-Xoss参数（设置本地方法栈大小)存在，但实际上是无效的，栈容量只由-Xss参数设定。关于虚拟机栈和本地方法栈，在JVM规范中描述了两种异常：
  + 1) 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。
  + 2)如果虚拟机在扩展栈时无法申请到足够的内存空间，将抛出OutOfMemoryError异常。

public class StackSOF {  
 private int stackLength = -1;  
 public void stackLeak() {  
 stackLength++;  
 stackLeak();  
 }  
  
 public static void main(String[] args) {  
 StackSOF sof = new StackSOF();  
 try {  
 sof.stackLeak();  
 } catch (Throwable e) {  
 System.out.println("stack length:" + sof.stackLength);  
 throw e;  
 }  
 }  
}

* -Xss128k（设置栈容量)
* stack length:998
* Exception in thread "main" java.lang.StackOverflowError
* at cn.sinjinsong.se.review.oom.StackSOF.stackLeak(StackSOF.java:10)
  + at cn.sinjinsong.se.review.oom.StackSOF.stackLeak(StackSOF.java:11)
* ...
* 操作系统分配给每个进程的内存是有限制的，每个线程分配到的栈容量越大，可以建立的线程数量自然就越少，建立线程时就越容易把剩下的内存耗尽。
* 如果线程过多导致SOF，可以通过减少最大堆和减少栈容量来换取更多的线程。

public class StackSOFByThread {  
 public void stackLeakByThread() {  
 while(true) {  
 new Thread(() -> {  
 while (true){}  
 }).start();  
 }  
 }  
  
 public static void main(String[] args) {  
 new StackSOFByThread().stackLeakByThread();  
 }  
}

# 5.12 方法区溢出

* 注意Java8下运行时常量池在堆中，所以运行时常量池过大会体现为OOM：heap；
* 而在此以前是放在永久代中，体现为OOM：PermGen space。

public class RuntimeConstantPoolOOM {  
 public static void main(String[] args) {  
 List<String> list = new ArrayList<>();  
 int i = 0;  
 while (true) {  
 list.add(String.valueOf(i++).intern());  
 }  
 }  
}

* VM Options: -Xms20m -Xmx20m
* Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
  + at java.lang.Integer.toString(Integer.java:401)
  + at java.lang.String.valueOf(String.java:3099)
  + at cn.sinjinsong.se.review.oom.RuntimeConstantPoolOOM.main(RuntimeConstantPoolOOM.java:15)
* 方法区还存放Class的相关信息，运行时产生大量的类也会导致方法区（Java8中放在直接内存中)溢出。

public class MetaspaceOOM {  
 public static void main(String[] args) {  
 while(true){  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(HeapOOM.OOMObject.class);  
 enhancer.setUseCache(false);  
 enhancer.setCallback(new MethodInterceptor() {  
 @Override  
 public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {  
 return proxy.invokeSuper(obj,args);  
 }  
 });  
 enhancer.create();  
 }  
 }  
}

* VM Options: -XX:MetaspaceSize=10m -XX:MaxMetaspaceSize=10m
* Caused by: java.lang.OutOfMemoryError: Metaspace
* at java.lang.ClassLoader.defineClass1(Native Method)
  + at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
* ... 11 more
* 方法区溢出也是一种常见的内存溢出异常，一个类被GC，判定条件是比较苛刻的。在经常生成大量Class的应用中，需要特别注意类的回收情况。这类场景除了动态代理生成类和动态语言外，还有：大量使用JSP、基于OSGi的应用。

# 5.13 直接内存溢出

* 直接内存可以使用-XX:MaxDirectMemorySize指定，如果不指定，则默认与Java堆最大值相同。
* 虽然使用DirectByteBuffer分配内存也会抛出OOM异常，但它抛出异常时并没有真正向OS申请分配内存，而是通过计算得知内存无法分配，于是手动抛出异常。
* 真正申请内存的方法是unsafe.allocateMemory()。

public class DirectMemoryOOM {  
 private static final int \_1MB = 1024 \* 1024;  
  
 public static void main(String[] args) throws IllegalAccessException {  
 Field unsafeField = Unsafe.class.getDeclaredFields()[0];  
 unsafeField.setAccessible(true);  
 Unsafe unsafe = (Unsafe) unsafeField.get(null);  
 while(true) {  
 unsafe.allocateMemory(\_1MB);  
 }  
 }  
}

* VM Options: -XX:MaxDirectMemorySize=10m
* Exception in thread "main" java.lang.OutOfMemoryError
* at sun.misc.Unsafe.allocateMemory(Native Method)
  + at cn.sinjinsong.se.review.oom.DirectMemoryOOM.main(DirectMemoryOOM.java:19)

# 5.14 内存泄露

- 1)非静态内部类  
- 2)连接未关闭：比如数据库连接（dataSourse.getConnection())，网络连接(socket)和io连接，除非其显式的调用了其close（)方法将其连接关闭，否则是不会自动被GC 回收的。

# GC

# 5.15 对象是否存活

## 引用计数算法

* 很多教科书判断对象是否存活的算法是这样的：给对象添加一个引用计数器，每当有一个地方引用它时，计数器就加1；当引用失效时，计数器值就减1；任何时刻计算器为0的对象就是不可能再被使用的。
* 主流的Java虚拟机中没有选用计数算法来管理内存，最主要的原因是它很难就解决对象之间相互循环引用的问题。

## 可达性分析算法

* 主流的商用程序语言的主流实现中，都是称通过可达性分析（Reachability Analysis)来判定对象是否存活的。这个算法的基本思路就是通过一系列的称为GC Roots 的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可达的。下图章，对象object5、object6、object7虽然互相有关联，但是它们到GC Roots时不可达的，所以它们将会被判定为可回收的对象。
* 在Java中，可作为GC Roots的对象包括：  
  - 虚拟机栈中引用的对象  
  - 方法区中类静态属性引用的对象  
  - 方法区中常量引用的对象  
  - 本地方法栈中JNI（一般说的Native方法)引用的对象

## finalize

* 即使在可达性分析中不可达的对象，也并非是非死不可。要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件就是此对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法，或者finalize()已经被虚拟机调用过，虚拟机将这两种情况视为没有必要执行。
* 如果这个对象被判为有必要执行finalize()方法，那么这个对象将会放置在一个叫做F-Queue队列之中，并在稍后由一个虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里所谓的执行是指虚拟机会触发这个方法，但并不承诺会等待它运行结束，这样做的原因是，如果一个对象在finalize()方法中执行缓慢，或者发生了死循环，将很可能会导致F-Queue队列中其他对象永久处于等待，甚至导致整个内存回收系统崩溃。finalize()方法是对象逃脱死亡命运的最后一次机会，稍后GC将会对F-Queue中的对象进行第二次小规模的标记，\如果对象要在finalize()中拯救自己，只要重新与引用链上的任何一个对象建立联系即可，比如把自己this复制给某个类变量或对象的成员变量，那在第二次标记时它将被移出即将回收的集合；如果对象这时候还没有逃脱，那基本上它就真的被回收了。任何一个对象的finalize()方法都只会被系统调用一次，如果对象面临下一次回收，它的finalize()方法不会被再次执行。

## 回收方法区

* 在方法区（永久代)中进行垃圾收集的性价比较低：在堆中，尤其在新生代中，常规应用进行一次垃圾收集一般可以回收70%~95%的空间，而永久代的垃圾收集效率远低于此。
* 永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。回收废弃常量和回收Java堆中的对象类似。以常量池中字面量的回收为例，没有任何String对象引用常量池中的某个字符串常量，这个常量就会被系统清理出常量池。常量池中的其他类、方法、字段的符号引用也与此类似。
* 判定一个类是否是无用的类的条件比较苛刻，需要同时满足以下三个条件：
  + 1)该类的所有实例都已经被回收
  + 2)加载该类的类加载器已经被回收
  + 3)该类对应的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，而不是和对象一样，不适用了就必然会被回收。是否对类回收，HotSpot虚拟机提供了参数进行控制。
* 在大量使用反射、动态代理、CGLib等ByteCode框架，动态生成JSP以及OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

# 5.16 GC算法

## 标记-清除算法

* 最基础的收集算法是标记-清除算法（Mark-Sweep)，算法分为标记和清除两个阶段。首先标记处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。他的不足主要有两个：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清除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以后再程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

## 复制算法

* 为了解决效率问题，出现了复制算法。它将可用内存按容量划分为大小相等的两块，每次只是用其中的一块。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，然后再把使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，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，按顺序分配内存空间，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。
* 现在的商业虚拟机都采用这种收集算法来回收新生代。将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次都使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8：1。当Survivor空间不够用时，需要依赖其他内存（老年代)进行分配担保。

## 标记-整理算法

* 复制收集算法在对象存活率较高时，效率就会变低。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，所以老年代一般不能直接选用这种算法。
* 根据老年代的特点，有人提出一种标记-整理算法（Mark-Compact)，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向一端移动，然后直接清理掉端边界以外的内存。

## 分代收集算法

* 当前商业虚拟机的垃圾收集都采用分代收集算法（Generational Collection)，这种算法是根据对象存活周期的不同将内存划分为适当的几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记-清除或者标记-整理算法来进行回收。

# 5.17 Minor Full GC

## Minor GC

* 从年轻代空间（包括 Eden 和 Survivor 区域)回收内存被称为 Minor GC。
* 非常频繁，回收速度较快。
* 各种Young GC的触发原因都是eden区满了。

## Full GC

* 收集整个堆，包括年轻代、老年代、元数据区等所有部分。
* 速度较慢。
* 触发原因不确定，因具体垃圾收集器而异。
* 比如老年代内存不足，ygc出现promotion failure，System.gc()等。
* CMS垃圾收集器不能像其他垃圾收集器那样等待年老代机会完全被填满之后再进行收集，需要预留一部分空间供并发收集时的使用。

# 5.18 HotSpot的垃圾收集器

* Java虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同厂商、不同版本的虚拟机所提供的垃圾收集器都可能有很大差别，并且一般都会提供参数供用户根据自己得到应用特点和要求组合出各个年代所使用的收集器。这里讨论的收集器基于HotSpot虚拟机，这个虚拟机包含的所有收集器如图所示。

## (1).Serial垃圾收集器

* Serial是最基本、历史最悠久的垃圾收集器，使用复制算法，曾经是JDK1.3.1之前新生代唯一的垃圾收集器。
* Serial是一个单线程的收集器，它不仅仅只会使用一个CPU或一条线程去完成垃圾收集工作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束。
* Serial垃圾收集器虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个CPU环境来说，没有线程交互的开销，可以 获得最高的单线程垃圾收集效率，因此Serial垃圾收集器依然是java虚拟机运行在Client模式下默认的新生代垃圾收集器。

## (2).ParNew垃圾收集器

* ParNew垃圾收集器其实是Serial收集器的多线程版本，也使用复制算法，除了使用多线程进行垃圾收集之外，其余的行为和Serial收集器完全一样，ParNew垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。
* ParNew收集器默认开启和CPU数目相同的线程数，可以通过-XX:ParallelGCThreads参数来限制垃圾收集器的线程数。
* ParNew虽然是除了多线程外和Serial收集器几乎完全一样，但是ParNew垃圾收集器是很多java虚拟机运行在Server模式下新生代的默认垃圾收集器。

## (3).Parallel Scavenge收集器

* Parallel Scavenge收集器也是一个新生代垃圾收集器，同样使用复制算法，也是一个多线程的垃圾收集器，它重点关注的是程序达到一个可控制的吞吐量 （Thoughput，CPU用于运行用户代码的时间/CPU总消耗时间，即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间))，高吞吐量 可以最高效率地利用CPU时间，尽快地完成程序的运算任务，主要适用于在后台运算而不需要太多交互的任务。
* Parallel Scavenge收集器提供了两个参数用于精准控制吞吐量：
* a.-XX:MaxGCPauseMillis：控制最大垃圾收集停顿时间，是一个大于0的毫秒数。
  + b.-XX:GCTimeRation：直接设置吞吐量大小，是一个大于0小于100的整数，也就是程序运行时间占总时间的比率，默认值是99，即垃圾收集运行最大1%（1/(1+99))的垃圾收集时间。
* Parallel Scavenge是吞吐量优先的垃圾收集器，它还提供一个参数：-XX:+UseAdaptiveSizePolicy，这是个开关参数，打开之后就不需 要手动指定新生代大小(-Xmn)、Eden与Survivor区的比例(-XX:SurvivorRation)、新生代晋升年老代对象年龄 (-XX:PretenureSizeThreshold)等细节参数，虚拟机会根据当前系统运行情况收集性能监控信息，动态调整这些参数以达到最大吞吐 量，这种方式称为GC自适应调节策略，自适应调节策略也是ParallelScavenge收集器与ParNew收集器的一个重要区别。

## (4).Serial Old收集器

* Serial Old是Serial垃圾收集器年老代版本，它同样是个单线程的收集器，使用标记-整理算法，这个收集器也主要是运行在Client默认的java虚拟机默认的年老代垃圾收集器。
* 在Server模式下，主要有两个用途：
* a.在JDK1.5之前版本中与新生代的Parallel Scavenge收集器搭配使用。
* b.作为年老代中使用CMS收集器的后备垃圾收集方案。
* 新生代Serial与年老代Serial Old搭配垃圾收集过程图：
* 新生代Parallel Scavenge收集器与ParNew收集器工作原理类似，都是多线程的收集器，都使用的是复制算法，在垃圾收集过程中都需要暂停所有的工作线程。
* 新生代Parallel Scavenge/ParNew与年老代Serial Old搭配垃圾收集过程图：

## (5).Parallel Old收集器

* Parallel Old收集器是Parallel Scavenge的年老代版本，使用多线程的标记-整理算法，在JDK1.6才开始提供。
* 在JDK1.6之前，新生代使用ParallelScavenge收集器只能搭配年老代的Serial Old收集器，只能保证新生代的吞吐量优先，无法保证整体的吞吐量，Parallel Old正是为了在年老代同样提供吞吐量优先的垃圾收集器，如果系统对吞吐量要求比较高，可以优先考虑新生代Parallel Scavenge和年老代Parallel Old收集器的搭配策略。
* 新生代Parallel Scavenge和年老代Parallel Old收集器搭配运行过程图：

## (6).CMS收集器（重点)

* Concurrent mark sweep(CMS)收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间，和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。
* 最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验，CMS收集器是Sun HotSpot虚拟机中第一款真正意义上并发垃圾收集器，它第一次实现了让垃圾收集线程和用户线程同时工作。
* CMS工作机制相比其他的垃圾收集器来说更复杂，整个过程分为以下4个阶段：
* a.初始标记：只是标记一下GC Roots能直接关联的对象，速度很快，仍然需要暂停所有的工作线程。
* b.并发标记：进行GC Roots跟踪的过程，和用户线程一起工作，不需要暂停工作线程。
* c.重新标记：为了修正在并发标记期间，因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，仍然需要暂停所有的工作线程。
* d.并发清除：清除GC Roots不可达对象，和用户线程一起工作，不需要暂停工作线程。
* 由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户现在一起并发工作，所以总体上来看CMS收集器的内存回收和用户线程是一起并发地执行。
* CMS收集器工作过程：
* CMS收集器有以下三个不足：
  + a.CMS收集器对CPU资源非常敏感，其默认启动的收集线程数=(CPU数量+3)/4，在用户程序本来CPU负荷已经比较高的情况下，如果还要分出CPU资源用来运行垃圾收集器线程，会使得CPU负载加重。
* b.CMS无法处理浮动垃圾(Floating Garbage)，可能会导致Concurrent ModeFailure失败而导致另一次Full GC。由于CMS收集器和用户线程并发运行，因此在收集过程中不断有新的垃圾产生，这些垃圾出现在标记过程之后，CMS无法在本次收集中处理掉它们，只好 等待下一次GC时再将其清理掉，这些垃圾就称为浮动垃圾。
* CMS垃圾收集器不能像其他垃圾收集器那样等待年老代机会完全被填满之后再进行收集，需要预留一部分空间供并发收集时的使用，可以通过参数
* -XX:CMSInitiatingOccupancyFraction来设置年老代空间达到多少的百分比时触发CMS进行垃圾收集，默认是68%。
* 如果在CMS运行期间，预留的内存无法满足程序需要，就会出现一次ConcurrentMode Failure失败，此时虚拟机将启动预备方案，使用Serial Old收集器重新进行年老代垃圾回收。
* c.CMS收集器是基于标记-清除算法，因此不可避免会产生大量不连续的内存碎片，如果无法找到一块足够大的连续内存存放对象时，将会触发因此 Full GC。CMS提供一个开关参数-XX:+UseCMSCompactAtFullCollection，用于指定在Full GC之后进行内存整理，内存整理会使得垃圾收集停顿时间变长，CMS提供了另外一个参数-XX:CMSFullGCsBeforeCompaction， 用于设置在执行多少次不压缩的Full GC之后，跟着再来一次内存整理。
* promotion failure 发生在 young gc 阶段，即 cms 的 ParNewGC。promotion failed是在进行Minor GC时，survivor space放不下、对象只能放入老年代，而此时老年代也放不下造成的；
* concurrent mode failure是在执行CMS GC的过程中同时有对象要放入老年代，而此时老年代空间不足造成的

## (7).G1收集器（重点)

* Garbage first垃圾收集器是目前垃圾收集器理论发展的最前沿成果，相比与CMS收集器，G1收集器两个最突出的改进是：
* a.基于标记-整理算法，不产生内存碎片。
* b.可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。
* G1收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间，优先回收垃圾最多的区域。
* 区域划分和优先级区域回收机制，确保G1收集器可以在有限时间获得最高的垃圾收集效率。

# 5.19 内存分配原则

* 对象的内存分配，就是在堆上分配，对象主要分配在新生代的Eden区上，如果启动了本地线程分配缓冲，将按线程优先在TLAB上分配。少数情况下也可能直接分配在老年代中，分配的规则不是固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数设置。
* 下面会讲解几条最普遍的内存分配原则。

## 对象优先在Eden分配

* 大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC。如果GC期间虚拟机发现已有的对象全部无法放入Survivor空间，会通过分配担保机制提前转移至老年代中。

## 大对象直接进入老年代

* 所谓的大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是那种很长的字符串以及数组。经常出现大对象容易导致内存还有不少空间就提前触发垃圾收集以获取足够的连续空间来安置它们。

## 长期存活的对象将进入老年代

* 虚拟机为每个对象定义一个对象年龄（Age)计数器。如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能够被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1.对象在Survivor区中每熬过一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁)，就将会被晋升到老年代。

## 动态对象年龄判定

* 虚拟机并不是永远地要求对象的年龄必须达到MaxTenuringThreshold才能晋升老年代，如果在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。

## 空间分配担保

* 在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险，那此时也要改为进行一次Full GC。
* 冒险是指当出现大量对象在Minor GC后仍然存活的情况，就需要老年代进行分配担保，把Survivor区无法容纳的对象直接进入老年代。老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共会有多少对象会活下来在实际完成内存回收之前是无法明确知道的，所以只好取之间每一次回收晋升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进行Full GC来让老年代腾出更多空间。
* 取平均值进行比较其实仍然是一种动态概率的手段，依然存在担保失败的情况。如果出现了HandlePromotionFailure失败，那就只好在失败后重新发起一次Full GC。

# 5.20 GC相关API

## System.gc

* 建议JVM进行Full GC,虽然只是建议而非一定,但很多情况下它会触发 Full GC,从而增加Full GC的频率,也即增加了间歇性停顿的次数。强烈影响系建议能不使用此方法就别使用，让虚拟机自己去管理它的内存，可通过通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc。

public static void gc() {

* Runtime.getRuntime().gc();
* }
* Runtime.gc的底层实现位于Runtime.c文件中
* JNIEXPORT void JNICALL
* Java*java*lang*Runtime*gc(JNIEnv \*env, jobject this)
* {
* JVM\_GC();
* }
* JVM*ENTRY*NO*ENV(void, JVM*GC(void))
* JVMWrapper("JVM\_GC");
* if (!DisableExplicitGC) {
* Universe::heap()->collect(GCCause::*java*lang*system*gc);
* }
* JVM\_END
* 这里有一个DisableExplicitGC参数，默认是false，如果启动JVM时添加了参数-XX:+DisableExplicitGC，那么JVM\_GC相当于一个空函数，并不会进行GC。
* 其中Universe::heap()返回当前堆对象，由collect方法开始执行GC，并设置当前触发GC的条件为*java*lang*system*gc，内部会根据GC条件执行不同逻辑。
* JVM的具体堆实现，在Universe.cpp文件中的initialize\_heap()由启动参数所设置的垃圾回收算法决定。
* 堆实现和回收算法对应关系：
* 1、UseParallelGC：ParallelScavengeHeap
* 2、UseG1GC：G1CollectedHeap
* 3、默认或者CMS：GenCollectedHeap

# 类文件结构

* Class类文件的结构
* Class文件并不一定定义在文件里，也可以通过类加载器直接生成。
* Class文件是一组以8位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在Class文件之中，中间没有添加任何分隔符。当遇到需要占用8位字节以上空间的数据项时，则会按照高位在前的方式分割成若干个8位字节进行存储。
* Class文件结构采用一种类似于C语言结构体的伪结构来存储数据，这种伪结构只有两种数据类型：无符号数和表。
* 无符号数属于基本的数据类型，以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节、8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照utf-8编码构成字符串值。
* 表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯性地以\_info结尾。表用于描述有层次关系的复合结构的数据。
* 无论是无符号数还是表，当需要描述同一类型但数量不定的多个数据时，经常会使用一个前置的容量计数器加若干个连续的数据项的形式，这时称这一系列连续的某一类型的数据为某一类型的集合。
* 魔数与Class文件的版本
* 每个Class文件的头4个字节称为魔数，它的唯一作用是确定这个文件是否为一个能被虚拟机接收的Class文件。很多文件存储格式都使用魔数来进行身份识别。魔数的值为0xCAFEBABE。
* 紧接着魔数的4个字节存储的是Class文件的版本号：第5和第6个字节是此版本号，第7和第8个字节是主版本号。
* 简单的一段Java代码，后面的内容将以此为例进行讲解：

public class TestClass {

private int m;

public int inc(){

* return m+1;
* }
* }
* 常量池
* 紧接着主次版本号之后的是常量池入口，常量池可以理解为Class文件中的资源仓库，它是Class文件结构中和其他项目关联最多的数据类型，也是占用Class文件空间最大的数据项目之一，同时它还是在Class文件中第一个出现的表类型数据项目。
* 由于常量池中常量的数量是不固定的，所以在常量池入口需要放置一项u2类型的数据，代表常量池容量计数器（从1开始)。对于其他集合类型，包括接口索引集合、字段表集合、方法表集合等的容量计数都与一般习惯相同，是从0开始的。
* 常量池中主要存放两大类常量：字面量（Literal)和符号引用（Symbolic References)。字面量比较接近于Java语言层面的常量概念，如字符串、final常量值。而符号引用则属于编译原理方面的概念，包括了下面三类常量：  
  - 类和接口的全限定名  
  - 字段的名称和描述符  
  - 方法的名称和描述符
* Java代码在javac编译的时候，并没有连接这一步骤，而是在虚拟机加载Class文件的时候进行动态连接。在Class文件中不会保存各个方法、字段的最终内存布局信息，因此这些字段、方法的符号引用不经过运行期转换的话无法得到真正的内存入口地址，也就无法直接被虚拟机使用。当虚拟机运行时，需要从常量池获得对应的符号引用，再在类创建时或运行时解析、翻译到具体的内存地址之中。
* 常量池中每一项常量都是一个表，在JDK1.7之前有11种不同结构的表结构数据。1.7增加了3种。它们的共同特点是表开始的第一位是一个u1类型的标志位，代表当前这个常量属于哪种常量类型。
* 访问标志

在常量池结束之后，紧接着的两个字节代表访问标志，这个标志用于识别一些类或接口层次的访问信息，包括：这个Class是类还是接口；是否定义为public类型；是否定义为abstract类型；如果是类的话，是否被声明为final等。

* 访问标志中一共有16个标志位可以使用，当前只定义了其中8个，没有使用到的标志位一律为0。
* 类索引、父类索引与接口索引集合
* 类索引和父类索引都是一个u2类型数据，而接口索引集合是一组u2类型数据的集合，Class文件中由这3项数据来确定这个类的继承关系。类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名。接口索引集合用来描述这个类实现了哪些接口，这些被实现的接口将按implements 语句后的接口顺序从左到右排列在接口索引集合中。
* 类索引、父类索引和接口索引集合都按顺序排列在访问标志之后，类索引和父类索引用两个u2类型的索引值表示，它们各自指向一个类型为CONSTANT*Class*info的类描述符常量，通过CONSTANT*Class*info类型的常量中的索引值可以找到定义在CONSTANT*Utf8*info类型的常量中的全限定名字符串。
* 对于接口类型集合，入口的第一项---u2类型的数据为接口计数器，表示索引表的容量。如果该类没有实现任何接口，则该计数器值为0，后面接口的索引表不再占用任何字节。
* 字段表集合
* 字段表用于描述接口或者类中声明的变量。字段包括类级变量以及实例级变量，但不包括在方法内部声明的局部变量。可以包括的信息有：作用域（访问权限)、static修饰符、final修饰符、并发可见性、序列化修饰符等。
* 跟随access*flags标志的是两项索引值：name*index and desciptor\_index。它们都是对常量池的引用，分别代表字段的简单名称以及字段和方法的描述符。
* 全限定名和简单名称：org/fenixsoft/clazz/TestClass是这个类的全限定名。简单名称是指没有类型和参数修饰的方法或者字段名称，这个类的inc（)方法和m字段的简单名称分别是inc和m。
* 相对于全限定名和简单名称而言，方法和字段的描述符要复杂一些。描述符的作用是用来描述字段的数据类型、方法的参数列表（数量、类型、顺序)和返回值。基本数据类型（byte、char、double…)以及代表无返回值的void类型都用一个大写字母表示，而对象类型则用字符L加对象的全限定名来表示。
* 对于数组来说，每一维度将使用一个前置的【字符来描述，如定义一个java.lang.String类型的二维数组，将被记录为[[Ljava/lang/String，一个整数数组 int[] 将被记录为[I。
* 用描述符描述方法时，按照先参数列表，后返回值的顺序描述，参数列表按照参数的严格顺序放在一组小括号之内。比如方法void inc（)的描述符为()V，方法java.lang.String.toString()的描述符为()Ljava/lang/String。
* 字段表集合首先是一个容量计数器，说明该类的字段表数据个数，然后是access\_flags标志，然后是其他标志、
* 在descriptor\_index之后跟随着一个属性表集合用于存储一些额外的信息。
* 字段表集合不会列出从超类或者父接口继承而来的字段，但有可能列出原本Java代码中不存在的字段，比如内部类会自动添加指向外部类实例的字段。
* 方法表集合
* 方法表的结构依次包括了访问标志、名称索引、描述符索引、属性表集合。
* 方法里面的Java代码，经过编译器编译为字节码指令后，存放在方法属性表中一个名为Code的属性里面，属性表是Class文件格式中最具拓展性的一种数据项目。
* 与字段表集合对应的，如果父类方法在子类中没有被重写，方法表集合中就不会出现来自父类的方法信息。但同样的，有可能会出现由编译器自动添加的方法，最典型的便是类构造器和实例构造器。
* 要重载一个方法，除了要有和原方法相同的简单名称外，还必须有一个不同的特征签名。特征签名就是一个方法中各个参数在常量池中的字段符号引用的结婚，也就是返回值不会包含在特征签名中。但是在Class文件格式中，特征签名的范围更大一些，只要描述符不是完全一致的两个方法也可以共存。如果两个方法有相同的名称和特征签名，但返回值不同，那么也是可以合法共存在同一个Class文件中的。
* 属性表集合
* 与Class文件中其他的数据项目要求严格的顺序、长度和内容不同，属性表集合的限制宽松一些，不再要求各个属性表具有严格顺序，并且只要不与已有属性名重复，任何人实现的编译器都可以向属性表写入自己定义的属性信息。为了能正确解析Class文件，Java虚拟机规范预定义了9项虚拟机实现应当能识别的属性。（现已增至21项)
* 以上列出其中的5种。
* 对于每个属性，它的名称需要从常量池中引用一个CONSTANT*Utf8*info类型的常量来表示，而属性表的结构则是完全自定义的，只需要通过一个u4的长度属性去说明属性值所占用的位数即可。一个符合规则的属性表应该满足下图所定义的结构：
* 1、Code属性
* Code属性出现在方法表的属性集合中，但并非所有的方法表都必须存在这种属性。
* max\_stack代表了操作数栈深度的最大值。在方法执行的任意时刻，操作数栈都不会超过这个深度。
* max*locals代表了局部变量表所需的存储空间，单位是slot，Slot是虚拟机为局部变量分配内存所使用的最小单位。对于byte、char等长度不超过32位的数据类型，每个局部变量占1个Slot，而long和double占2个Slot。方法参数（包括this)、显式异常处理器参数（catch所定义的异常)、方法体中定义的局部变量都需要使用局部变量表来存放。并不是方法中用到了多少个局部变量，就把这些变量所占Slot之和作为max*locals的值，因为局部变量表中的Slot可以重用，当代码执行超过一个局部变量的作用域时，这个局部变量所占的Slot可以被其他局部变量所使用。
* Code属性表是Class文件中最重要的一个属性，如果把一个Java程序中的信息分为代码和元数据两部分，那么在整个Class文件中，Code属性用于描述代码，所有的其他数据项目都用于描述元数据。
* 在任何实例方法中，都可以通过this关键字访问到此方法所属的对象。它的实现就是通过javac编译器变异的时候把对this关键字的访问转变为对一个普通方法参数的访问，然后在虚拟机调用实例方法时自动传入此参数。因此在实例方法的局部变量表中至少会存在一个指向当前对象实例的局部变量，局部变量表中也会预留出第一个Slot位来存放对象实例的引用。
* 在字节码指令之后的是这个方法的显式异常处理表集合，异常表对Code属性来说并不是必须存在的。
* 异常表包含4个字段，这些字段的含义是：如果当字节码在第start*pc行到第end*pc之间（不含end*pc)出现了类型为catch*type或者其子类的异常，则转到第handler*pc行继续处理。当catch*type的值为0时，代表任意异常情况都需要转向到handler\_pc处进行处理。
* 异常表实际上是Java代码的一部分，编译器使用异常表而不是简单的跳转命令来实现Java异常及finally处理机制。
* 2、Exceptions属性
* Exceptions属性的作用是列举出方法中可能抛出的受检查异常，也就是方法描述时在throws关键字后面列举的异常。它的结构：
* 3、LineNumberTable属性
* LineNumberTable属性用于描述Java源码行号与字节码行号之间的对应关系，是可选的属性。如果选择不生成LineNumberTable属性，对程序运行的最主要的影响就是当跳出异常时，堆栈中将不会显示出错的行号，并且在调试的时候，也无法按照源码行来设置断点。
* 4、LocalVariableTable属性
* LocalVariableTable属性用于描述栈帧中局部变量表中的变量和Java源码中定义的变量之间的关系，它也是可选的属性。
* 5、SourceFile属性
* SourceFile属性用于记录生成这个Class文件的源码文件名称，也是可选的。
* 6、ConstantValue属性
* ConstantValue属性的作用是通知虚拟机自动为静态变量赋值。只有被static修饰的变量才可以使用这项属性。对于非static类型的变量的赋值是在实例构造方法中进行的；而对于类变量，则有两种方式可以选择：在类构造器中或者使用ConstantValue属性。目前Sun Javac编译器的选择是：如果是常量（static final)，并且这个常量的数据类型是基本类型或者String的话，就生成ConstantValue属性来进行初始化，如果这个变量没有被final修饰，或者并非基本类型或字符串，则将会选择在类构造器中进行初始化。
* 字节码指令简介
* Java虚拟机指令是由一个字节长度的、代表着某种特定操作含义的数字（操作码，Opcode)以及跟随其后的零至多个代表此操作所需参数（操作数，Operands)而构成。由于Java虚拟机采用面向操作数栈而不是寄存器的架构，所以大多数的指令都不包含操作数，只有一个操作码。
* 1个字节意味着指令集的操作码总数不能超过256条；又由于Class文件格式放弃了编译后代码的操作数长度对齐，这就意味着虚拟机处理那些超过一个字节数据的时候，不得不在运行时从字节中重建出具体数据的结构。放弃了操作数长度对齐，就意味着可以省略很多填充和间隔符号。
* Java虚拟机的解释器可以使用下面这个伪代码当做最基本的执行模型来理解：
* 字节码与数据类型
* 在Java虚拟机的指令集中，大多数的指令都包含了其操作所对应的数据类型信息。iload指令用于从局部变量表中记载int型的数据到操作数栈中，而fload指令加载的则是float类型的数据。这两条指令的操作在虚拟机内部可能会是由同一段代码来实现的，但在Class文件中它们必须拥有各自独立的操作码。
* 对于大部分与数据类型相关的字节码指令，它们的操作码助记符中都有特殊的字符来表明专门为哪种数据类型服务，i代表对int类型的数据操作，l代表long等。
* 大部分的指令都没有支持整数类型byte、char和short，编译器会在编译时或运行时将byte和short类型的数据带符号拓展为相应的int类型的数据。大多数对于boolean、byte、short和char类型的数据的操作，实际上都是使用相应的int类型作为运算类型。
* 加载和存储指令
* 加载和存储指令用于将数据在栈帧中的局部变量表和操作数之间来回传输。
* 尖括号结尾的指令实际上是代表了一组指令，这几组指令都是某个带有一个操作数的通用指令的特殊形式，对于这若干组的特殊指令来说，它们省略掉了显式地操作数，不需要进行取操作数的动作，实际上操作数就隐含在指令中。
* 运算指令
* 类型转换指令
* 尽管数据类型窄化转换可能会发生上限溢出、下限溢出和精度丢失等情况，但是Java虚拟机规范中明确规定数值类型的窄化转换指令永远不可能导致虚拟机抛出运行时异常。
* 对象创建与访问指令
* 操作数栈管理指令
* 控制转换指令
* 方法调用和返回指令
* 异常处理指令
* 在Java程序中显式抛出的操作都由athrow指令来实现。处理异常（catch)不是由字节码指令来实现的，而是采用异常表来完成的。
* athrow指令与异常表：

public void catchException() {

* try {
* throw new Exception();
  + } catch (Exception var2) {
* ;
* }
* }
* 字节码：

public void catchException();

* Code:
* Stack=2, Locals=2, Args\_size=1
* 0: new #58; //class java/lang/Exception
* 3: dup
* 4: invokespecial #60; //Method java/lang/Exception."":()V
* 7: athrow
* 8: astore\_1
* 9: return
* Exception table:
* from to target type
* 0 8 8 Class java/lang/Exception
* 偏移为7的athrow指令，这个指令运作过程大致是首先检查操作栈顶，这时栈顶必须存在一个reference类型的值，并且是java.lang.Throwable的子类（虚拟机规范中要求如果遇到null则当作NPE异常使用)，然后暂时先把这个引用出栈，接着搜索本方法的异常表，找一下本方法中是否有能处理这个异常的handler，如果能找到合适的handler就会重新初始化PC寄存器指针指向此异常handler的第一个指令的偏移地址。接着把当前栈帧的操作栈清空，再把刚刚出栈的引用重新入栈。如果在当前方法中很悲剧的找不到handler，那只好把当前方法的栈帧出栈（这个栈是VM栈，不要和前面的操作栈搞混了，栈帧出栈就意味着当前方法退出)，这个方法的调用者的栈帧就自然在这条线程VM栈的栈顶了，然后再对这个新的当前方法再做一次刚才做过的异常handler搜索，如果还是找不到，继续把这个栈帧踢掉，这样一直到找，要么找到一个能使用的handler，转到这个handler的第一条指令开始继续执行，要么把VM栈的栈帧抛光了都没有找到期望的handler，这样的话这条线程就只好被迫终止、退出了。
  + 上面的异常表只有一个handler记录，它指明了从偏移地址0开始（包含0)，到偏移地址8结束（不包含8)，如果出现了java.lang.Exception类型的异常，那么就把PC寄存器指针转到8开始继续执行。顺便说一下，对于Java语言中的关键字catch和finally，虚拟机中并没有特殊的字节码指令去支持它们，都是通过编译器生成字节码片段以及不同的异常处理器来实现。
* 同步指令（重点)
* Java虚拟机可以支持方法级的同步和方法内部一段指令序列的同步，这两种同步结构都是使用管程（Monitor)来支持的。
* 方法级的同步是隐式的，即无需通过字节码指令来控制，它实现在方法调用和返回操作之中。当方法调用时，调用指令将会检查方法的ACC\_SYNCHRONIZED访问标志是否被设置，如果设置了，执行线程就要求先成功持有管程。在方法执行期间，执行线程持有了管程，其他任何线程都无法再获取到同一个管程。如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那么这个同步方法所持有的管程将在异常抛到同步方法之外时自动释放。
* 同步一段指令集序列通常是由synchronized语句块来表示的，Java虚拟机的指令集有monitorenter和monitorexit两条指令来支持synchronized关键字的语义，正确实现synchronized关键字需要Javc编译器和虚拟机两者共同协作支持。
* 方法中调用过的每一条monitorenter指令都必须执行其对应的monitorexit指令，而无论这个方法是否正常结束。
* 为了保证在方法异常完成时monitorenter和monitorexit指令异常可以正常配对执行，编译器会自动产生一个异常处理器，这个异常处理器声明可以处理所有的异常，它的目的就是用来执行monitorexit指令。

# 类加载机制

* 概述
* 虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。
* 与那些编译时需要进行连接的语言不同，Java语言里面，类型的加载、连接和初始化都是在程序运行期间完成的，这种策略虽然会令类加载时增加一些性能开销，但是是为Java应用程序提供高度的灵活性，Java里天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。
* 类加载的时机
* 类的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载。
* 加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序开始，而解析阶段在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定（动态绑定)。
* 什么时候开始类加载过程的第一个阶段：加载？
* Java虚拟机规范规定有且只有5种情况必须立即对类进行初始化：
  + 1)遇到new、getstatic、putstatic、invokestatic这4条字节码指令时，如果类没有过初始化，则需要先初始化。生成这4条指令的最常见的Java代码场景是：使用new实例化对象时、读取或设置一个类的静态字段时、调用一个类的静态方法时
  + 2)反射
  + 3)如果一个类的父类尚未初始化，那么先触发其父类的初始化
  + 4)main方法所在类
  + 5)java.lang.invoke.MethodHandle实例最后的解析结果REF*getStatic、REF*putStatic、 REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。
* 只有直接定义一个静态字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。至于是否要触发子类的加载和验证，在虚拟机规范中并未明确确定，这点取决于虚拟机的具体实现。
* 当一个类在初始化时，要求其父类全部都已经初始化过了，但是一个接口在初始化时，并不要求其父类接口全部都完成了初始化，只有在真正使用到父接口（如引用接口中定义的常量)的时候才会初始化。
* 类加载的过程
  + 1) 加载
* 加载是类加载过程的一个阶段。
* 在加载阶段，虚拟机需要完成以下3件事情：
  + 1)通过一个类的全限定名获得此类的二进制字节流
  + 2)将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
  + 3)在内存中生成一个代表这个类的Class对象（HotSpot中放在堆里)，作为方法区这个类的各种数据的访问入口。
* 加载阶段既可以使用系统提供的引导类加载器来完成，也可以由用户自定义的类加载器，开发人员可以通过定义自己的类加载器去控制字节流的获取方式。
* 加载阶段和连接阶段的部分内容是交叉进行的
* java.lang.Class实例并不负责记录真正的类元数据，而只是对VM内部的InstanceKlass对象的一个包装供Java的反射访问用，InstanceKlass放在方法区（Java8HotSpot中放在元数据区)
  + 2) 验证
* 验证是连接阶段的第一步，目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。
* 验证阶段是否严谨直接决定了Java虚拟机是否能承受恶意代码的攻击，从执行性能的角度上讲，验证阶段的工作量在虚拟机的类加载子系统中又占了相当大的一部分。
* 验证阶段大致会完成下面4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。
* 1、文件格式验证
* 验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理。这阶段的验证是基于二进制字节流进行的，只有通过了这个阶段的验证后，字节流才会进入内存的方法区中进行存储，所以后面的3个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流。
* 2、元数据验证
* 第二阶段是对字节码描述的数据进行语义分析，以保证其描述的信息符合Java语言规范的要求。
* 3、字节码验证
* 第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。这个阶段将对类的方法体进行校验分析，保证被校验的类的方法在运行时不会做出危害虚拟机安全的事件。
* 如果一个方法体通过了字节码校验，也不能说明其一定就是安全的，这里涉及一个停机问题，通过程序去校验程序逻辑是无法做到绝对准确的-----不能通过程序准确地检查出程序是否能在有限的时间之内结束运行。
* 4、符号引用验证
* 最后一个阶段发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段—解析阶段中发生。符号引用可以看做是对类自身以外（常量池中的各种符号引用)的信息进行匹配性校验。
  + 3) 准备
* 准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这时候进行内存分配的仅包括类变量，而不包括实例变量；这里所说的初始值，是指0值。
* 如果是static final 常量，那么会被初始化为ConstantValue属性所指定的值。
  + 4) 解析
* 解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。
* 符号引用：符号引用是一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定都已经加载到内存中。
* 直接引用：直接引用是可以直接指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那么引用的目标必须已经在内存中存在。
* 对同一个符号引用进行多次解析请求是很常见的事情，除了invokedynamic指令外，虚拟机实现可以对第一次解析的结果进行缓存，从而避免解析动作重复进行。动态（invokedynamic)的含义是必须等到程序实际运行到这条指令的时候，解析动作才能进行。
  + 5) 初始化
* 类初始化是类加载阶段的最后一步。到了初始化阶段，才真正开始执行类中定义的Java程序代码。
* 初始化阶段是执行类构造器方法的过程。
* 类构造器是由编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序决定的，静态代码块只能访问到定义在静态代码块之间的变量，定义在它之后的变量，在前面的静态代码块可以赋值，但是不能访问。
* 类构造器与类的构造方法不同，它不需要显式调用父类构造器，虚拟机会保证在子类的类构造器执行之前，父类的类构造器已经执行完毕。
* 类构造器对于类或接口不是必需的，如果一个类中没有静态代码块，也没有对变量的赋值操作，那么编译器可以不为这个类生成类构造器。
* 接口中不能使用静态代码块，但是仍然有变量初始化的赋值操作，因此接口和类一样都会生成类构造器。但接口与类不同的是，执行接口的类构造器不需要先执行父接口的类构造器。只有当父接口中定义的变量使用时，父接口才会初始化，另外，接口的实现类在初始化时也一样不会执行接口的类构造器。
* 虚拟机会保证一个类的类构造器在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只有一个线程去执行这个类的类构造器，其他线程都需要阻塞等待，直至活动线程执行类构造器完毕。
* 类的主动引用和被动引用
* 主动引用（一定会发生类的初始化)
* new对象
* 引用静态变量（static非final)和静态方法
* 反射
* main方法所在类
* 当前类的所有父类
* 被动引用（不会发生类的初始化)
* 访问一个类的静态变量，只有真正声明这个静态变量的类才会被初始化
* 通过数组定义类引用
* 引用常量（存在方法区的运行时常量池)
* 类加载器
* 类加载阶段中的“通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为类加载器。
* 类与类加载器
* 类加载器虽然只用于实现类的加载动作，但它在Java程序中的作用不限于类加载阶段。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。换句话说，比较两个类是否相等，只有在这两个类是由同一个类加载器（实例)加载的前提下才有意义，否则，即使这两个来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类必定不相等。
* 这里所指的相等，包括类的Class对象的equals方法等的返回结果，也包括instance of的返回结果。
* 双亲委派模型
* 从Java虚拟机角度来讲，只存在两种不同的类加载器：一种是启动类加载器（bootstrap classloader)，这个类加载器由C++语言实现（HotSpot)，是虚拟机自身的一部分；另一种就是所有的其他类加载器，都由Java语言实现，独立于虚拟机外部。并且全继承自java.lang.ClassLoader。
* 从Java开发人员的角度看，Java程序使用到以下3种系统提供的类加载器：
  + 1)启动类加载器：负责将存放在\lib目录中的类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用。
  + 2)扩展类加载器（Extension ClassLoader)：这个加载器负责加载\lib\ext目录中的所有类库，开发者可以直接使用扩展类加载器。
  + 3)应用程序类加载器（Application ClassLoader)：或称系统类加载器，负责加载用户classpath下所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。
* 类加载器之间的层次关系成为类加载器的双亲委派模型（Parents Delegation Model)。双亲委派模型要求除了顶层了启动类加载器，其他的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不会以继承的关系来实现，而是使用组合的方式来复用父加载器的代码。
* 双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求，子加载器才会尝试自己去加载。
* 使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是Java类随它的类加载器一起具备了一种带有优先级的层次关系。
* 破坏双亲委派模型
* 双亲委派模型并不是一个强制性的约束模型，而是Java设计者推荐给开发者的类加载器实现方式。比如OSGi环境下，类加载不再是双亲委派模型中的树形结构，而是进一步发展为更加复杂的网状结构，当收到类加载请求，OSGi将按照下面的顺序进行类搜索：

## 热部署

* 如果我们希望将java类卸载，并且替换更新版本的java类，该怎么做呢？’
* 1、销毁该自定义ClassLoader
* 2、更新class文件
* 3、创建新的ClassLoader去加载更新后的class文件。

# 5.21 对象初始化的先后顺序

* 单个类：
  + 1)类的静态变量清0
  + 2)静态变量赋值，静态代码块（按照编写顺序调用)
  + 3)成员变量清0
  + 4)成员变量赋值，非静态代码块（按照编写顺序调用)
  + 5)构造方法
* 1、2统称为类的初始化
* 4、5统称为对象初始化
* 带有继承时：
  + 1)父类类初始化
  + 2)子类类初始化
  + 3)成员变量清0，包括父类和子类
  + 3)父类对象初始化
  + 4)子类对象初始化
* 实例一：

public class StaticTest {  
 public static void main(String[] args) {  
 staticFunction();  
 }  
   
 static StaticTest st = new StaticTest();  
  
 static { //静态代码块  
 System.out.println("1");  
 }  
  
 { // 实例代码块  
 System.out.println("2");  
 }  
  
 StaticTest() { // 实例构造器  
 System.out.println("3");  
 System.out.println("a=" + a + ",b=" + b);  
 }  
  
 public static void staticFunction() { // 静态方法  
 System.out.println("4");  
 }  
  
 int a = 110; // 实例变量  
 static int b = 112; // 静态变量  
  
 /\*\*  
 main 方法属于静态方法，主动引用，开始执行类的初始化：按照编写顺序进行静态变量赋值与静态代码块执行  
 1)先初始化StaticTest，对象实例化时，因为类已经被加载，所以执行对象初始化，先对成员变量进行初始化（a赋值为0)，  
 然后按照编写顺序进行非静态变量赋值与非静态代码块执行（打印2，a赋值为110)，  
 再调用构造方法（打印3，打印a=110，b=0)  
 2)再执行静态代码块，打印1  
 3)再赋值b为112,  
 4)至此类加载完毕，执行main方法，打印4  
   
   
 2  
 3  
 a=110,b=0  
 1  
 4  
 \*/  
}

* 实例二：

public class InitializeDemo {  
 private static int k = 1;  
 private static InitializeDemo t1 = new InitializeDemo("t1");  
 private static InitializeDemo t2 = new InitializeDemo("t2");  
 private static int i = print("i");  
 private static int n = 99;  
  
 static {  
 print("静态块");  
 }  
  
 private int j = print("j");  
  
 {  
 print("构造块");  
 }  
  
 public InitializeDemo(String str) {  
 System.out.println((k++) + ":" + str + " i=" + i + " n=" + n);  
 ++i;  
 ++n;  
 }  
  
 public static int print(String str) {  
 System.out.println((k++) + ":" + str + " i=" + i + " n=" + n);  
 ++n;  
 return ++i;  
 }  
  
 public static void main(String args[]) {  
 new InitializeDemo("init");  
 }  
}

* 1:j i=0 n=0
* 2:构造块 i=1 n=1
* 3:t1 i=2 n=2
* 4:j i=3 n=3
* 5:构造块 i=4 n=4
* 6:t2 i=5 n=5
* 7:i i=6 n=6
* 8:静态块 i=7 n=99
* 9:j i=8 n=100
* 10:构造块 i=9 n=101
* 11:init i=10 n=102
* 实例三：

class Glyph {  
 void draw() {  
 System.out.println("Glyph.draw()");  
 }  
  
 Glyph() {  
 System.out.println("Glyph() before draw()");  
 draw();  
 System.out.println("Glyph() after draw()");  
 }  
}  
  
class RoundGlyph extends Glyph {  
 private int radius = 1;  
  
 RoundGlyph(int r) {  
 radius = r;  
 System.out.println("RoundGlyph.RoundGlyph(), radius = " + radius);  
 }  
  
 void draw() {  
 System.out.println("RoundGlyph.draw(), radius = " + radius);  
 }  
}  
  
public class PolyConstructors {  
 public static void main(String[] args) {  
 new RoundGlyph(5);  
 /\*\*  
 \*  
 Glyph() before draw()  
 RoundGlyph.draw(), radius = 0  
 Glyph() after draw()  
 RoundGlyph.RoundGlyph(), radius = 5  
 \*/  
 }  
}

# 字节码执行引擎

* 概述
* 虚拟机的执行引擎不是直接建立在处理器、硬件、指令集和操作系统层面的，而是由自己实现的，因此可以自行制定指令集与执行引擎的结构体系，并能够执行哪些不被硬件直接支持的指令集格式。
* 在不同的虚拟机实现里面，执行引擎在执行Java代码的时候可能会有解释执行和编译执行（通过即时编译器产生本地代码)两种选择，也可能两者兼备。但从外观上看起来，所有的Java虚拟机都是一样的：输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行过程。
* 运行时栈帧结构
* 栈帧（Stack Frame)是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态链接、返回地址等信息。每一个方法从调用开始至执行完成过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。
* 在编译程序代码时，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的Code属性之中。因此，一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。
* 一个线程中的方法调用链可能会很长，很多方法同时处于执行状态。对于执行引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为当前栈帧，与这个栈帧相关联的方法称为当前方法。执行引擎运行的所有字节码指令都只针对当前栈帧进行操作。
* 局部变量表
* 局部变量表（Local Variable Table)是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。方法的Code属性的max\_locals数据项中确定了该方法所需要分配的局部变量表的最大容量。
* 局部变量表的容量以变量槽（Slot)为最小单位，每个Slot都应该能存放一个boolean、byte、char、short、Reference等类型的数据，允许Slot的长度可以随着处理器、操作系统或虚拟机的实现不同而发生变化。
* 一个Slot可以存放一个对象实例的引用，虚拟机能够通过这个引用做到两点：一是从此引用中直接或间接地查找对象在Java堆中的数据存放的起始地址索引，二是此引用中直接或间接地查找到对象所属数据类型在方法区中的存储的类型信息。
* 局部变量表是线程私有的数据，无论读写两个连续的Slot（long、double)是否为原子操作，都不会引起线程安全问题。
* 对于64位的数据类型，虚拟机会以高位对齐的方式为其分配两个连续的Slot空间。
* 虚拟机通过索引定位的方式使用局部变量表，索引值的范围从0开始至局部变量表最大的Slot数量。对于两个相邻的共同存放一个64位数据的两个Slot，不允许采用任何方式单独访问其中的某一个。
* 在方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果执行的是实例方法，那局部变量表的第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字this来访问到这个隐含的参数。其他参数则按照参数表顺序排列，占用从1开始的局部变量Slot，参数分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的Slot。
* 局部变量表中的Slot是可以重用的，方法体中定义的变量，其作用域并不一定会覆盖整个方法体，如果当前字节码PC计数器的值已经超出了某个变量的作用域，那整个变量对应的Slot就可以交给其他变量使用。Slot的复用会直接影响到系统的垃圾收集行为。
* 操作数栈
* 操作数栈（Operand Stack)是一个后进先出栈。操作数栈的最大深度也是在编译的时候就写入到Code属性的max*stacks数据项中。操作数栈的每一个元素可以是任意的Java数据类型，包括long和double。32位数据类型所占的栈容量为1,64位数据类型所占的栈容量为2。在方法执行的任何时候，操作数栈的深入都不会超过max*stacks。
* 在概念模型中，两个栈帧作为虚拟机栈的元素，是完全相互独立的。但在大多数虚拟机的实现里都会做一些优化处理，令两个栈帧出现一部分重叠。让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用时就可以共用一部分数据，无须进行额外的参数复制传递。
* Java虚拟机的解释执行引擎称为基于栈的执行引擎，其中的栈就是操作数栈。
* 动态连接
* 每个栈帧都包含一个执行运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接（Dynamic Linking)。Class文件的常量池存有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或者第一次使用的时候就转化为直接引用，这种转化称为静态解析。另外一部分将在每一次运行期间转化为直接引用，这部分称为动态连接。
* 方法返回地址
* 当一个方法开始执行后，只有两种方式可以退出这个方法。第一种方式是执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者，是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法的方式称为正常完成出口。
* 另外一种退出方式是，在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，无论是Java虚拟机内部产生的异常，还是代码中使用athrow字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方法的方式称为异常完成出口。一个方法使用异常完成出口的方式退出，是不会给它的上层调用者产生任何返回值的。
* 无论何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行。一般来说，方法正常退出时，调用者的PC计数器的值可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是通过异常处理器确定的，栈帧中一般不会保存这部分信息。
* 方法退出的过程实际上就等同于把当前栈帧出栈，因此退出可能的操作有：恢复上层方法的局部变量表和操作数栈，把返回值压入调用者栈帧的操作数栈中，调整PC计数器的值以指向方法调用指令后面的一条指令。

# 5.22 方法调用

* 方法调用并不等同于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本（即调用哪一个方法)，暂时不涉及方法内部的具体执行过程。Class文件的编译过程不包含传统编译中的连接步骤，一切方法调用在Class文件中存储的都是符号引用，而不是方法在实际运行时内存布局中的入口地址。这个特性使得Java方法调用过程变得复杂，需要在类加载器件，甚至到运行期间才能确定目标方法的直接引用。
* 解析
* 符号引用能转为直接引用成立的前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在程序代码写好、编译器进行编译时就必须确定下来。这类方法的调用称为解析。
* 在Java语言中符合“编译期可知，运行期不可变”这个要求的方法，主要包括静态方法和私有方法，前者和类型直接关联，后者在外部不可被访问，这两种方法的特点决定了它们都不可能通过继承或别的方式重写其他版本，因此它们都适合在类加载阶段进行解析。
* 只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段中确定唯一的调用版本，符合这个条件的有静态方法、私有方法、实例构造器、父类方法4类，它们在类加载的时候就会把符号引用解析为该方法的直接引用。这个方法可以称为非虚方法，与之相反，其他方法称为虚方法（除去final)。
* 非虚方法也包含被final修饰的方法。虽然final方法是使用invokevirtual指令来调用的，但是由于它无法被覆盖，没有其他版本，所以也无须对方法调用者进行多态选择，又或者说多态选择的结果肯定是位移的。
* 解析调用一定是个静态过程，在编译期间就完全确定，在类加载的解析阶段就会把涉及的符号引用全部转为可确定的直接引用，不会延迟到运行期再去完成。而分派调用则可能是静态的也可能是动态的，根据分派依据的宗量数可分为单分派和多分派。这两类分派方式的组合就构成了静态单分派、静态多分派、动态单分派、动态多分派这4中分派组合情况。
* 分派
* 分派调用过程将会揭示多态性的一些最基本体现，如重载和重写。
* 1、静态分派
* 上面代码中的Human称为变量的静态类型（Static Type)，或者叫做外观类型，后面的Man则称为变量的实际类型（Actual Type)，静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会发生改变，并且最终的静态类型是在编译期可知的；而实际类型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。
* 代码中刻意定义了两个静态类型相同但是实际类型不同的变量，但编译器在重载时是通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型是编译器可知的，因此，在编译阶段，Javac编译器会根据参数的静态类型决定使用哪个重载版本，，所以选择了sayHello（Human)作为调用目标，并把这个方法的符号引用写到main（)方法里的两条invokevirtual指令的参数中。
* 所有依赖静态类型来定位方法执行版本的分派动作被称为静态分派。静态分派的典型应用是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的。另外，编译器虽然能确定出方法的重载版本，但在很多情况下这个重载版本并不是唯一的，往往只能确定一个更加合适的版本。产生这种模糊结论的主要原因是字面量不需要定义，所以字面量没有显式地静态类型，它的静态类型只能通过语言上的规则去理解和推断。
* 2、动态分派
* 重写
* 导致整个现象的原因很明显，是这两个变量的实际类型不同。
* 以下为字节码
* 由于invokevitual指令执行的第一步就是 在运行期确定接受者的实际类型，所以两次调用中的invokevirtual指令把常量池中的类方法符号解析到了不同的直接引用上，这个过程就是Java语言中方法重写的本质，我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。
* 3、单分派与多分派
* 方法的接收者和方法的参数统称为方法的宗量。根据分派基于多少种宗量，可以将分派划分为单分派和多分派两种。单分派是根据一个宗量对目标方法进行选择，多分派则是根据多于一个宗量对目标方法进行选择。
* 今天的Java语言是一门静态多分派、动态单分派的语言。
* 4、虚拟机动态分派的实现
* 由于动态分派是非常频繁的动作，而且动态分派的方法版本选择过程需要运行时在类的方法元数据中搜索合适的目标方法，因此在虚拟机的实际实现中基于性能的考虑，大部分实现都不会真正地进行如何频繁的搜索。最常用的稳定优化的方法就是为类在方法区中建立一个虚方法表，使用虚方法表索引来代替元数据查找以提高性能。
* 虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类方法表中的地址将会替换为指向子类实现版本的入口地址。
* 为了程序实现上的方便，具有相同签名的方法，在父类、子类的虚方法表中都应当具有一样的索引编号，这样当类型变换时，仅需要变更查找的方法表，就可以从不同的虚方法表中按索引转换出所需的入口地址。
* 方法表一般在类加载的连接阶段进行初始化，准备了类的变量初始值后，虚拟机会把该类的方法表也初始化完毕。
* 虚拟机除了使用方法表之外，在条件允许的情况下，还会使用内联缓存和基于类型继承关系分析技术的守护内联两种非稳定的激进优化手段来获得更高的性能。
* 基于栈的字节码解释执行引擎
* 许多Java虚拟机的执行引擎在执行Java代码的时候都有解释执行和编译执行两种选择。
* 解释执行
* Java语言经常被人们定位为解释执行的语言，但当主流的虚拟机都包含了即时编译器后，Class文件中的代码到底会被解释执行还是编译执行，就成了只有虚拟机自己才能准确判断的事情。只有确定了谈论对象是某种具体的Java实现版本和执行引擎运行模式时，谈解释执行还是编译执行才会比较确切。
* 基于栈的指令集和基于寄存器的指令集
* Java编译器输出的指令流，基本上是一种基于栈的指令集架构（Instruction Set Architecture,ISA)，指令流中的指令大部分都是零地址指令，它们依赖操作数栈进行工作。与之相对的另外一套常用的指令集架构是基于寄存器的指令集。
* 计算1+1：
* 前者：
* 后者：
* 基于栈的指令集主要的优点是可移植，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。栈架构的指令集还有一些其他的优点，如代码相对更加紧凑、编译器实现更加简单（不需要考虑空间分配，都在栈上操作)等。
* 栈架构指令集的主要缺点是执行速度相对来说会稍慢一些。
* 虽然栈架构指令集的代码非常紧凑，但是完成相同功能所需的指令数量一般会比寄存器架构多，因为出栈、入栈操作本身就产生了相当多的指令数量。更重要的是，栈实现在内存之中，频繁的栈访问也就意味着频繁的内存访问，相对于处理器来说，内存始终是执行速度的瓶颈。尽管虚拟机可以采取栈顶缓存的手段，把最常用的操作映射到寄存器中避免直接内存访问，但这也只能是优化措施而不是解决本质问题的方法。由于指令数量和内存访问的原因，所以导致了栈架构指令集的执行速度会相对较慢。

# 程序编译与代码优化

# 5.23 字节码的编译过程（前端编译器)

* Java语言的编译期是一段不确定的操作过程。
  + 1)编译器前端/前端编译器：把java文件转为class文件，比如Sun的Javac
  + 2)编译器后端/后端运行时编译器(JIT Just In Time 编译器)：把字节码转为机器码，比如HotSpot VM的C1、C2编译器
  + 3)静态提前编译器（AOT Ahead Of Time 编译器)：直接把java文件编译为本地机器代码，比如GNU Compiler for the Java（GCJ)。
* 通常意义上的编译器就是前端编译器，这类编译器对代码的运行效率几乎没有任何优化，把对性能的优化集中到了后端编译器，这样可以使其他语言的class文件也同样能享受到编译器优化所带来的好处。
* 但是Javac做了很多针对Java语言编码过程中的优化措施来改善程序员的编码风格和提高编码效率，相当多的新的语法特性都是靠前端编译器的语法糖实现的，而非依赖虚拟机的底层改进来实现。
* Javac的编译过程大致可以分为三个阶段：
  + 1)解析和填充符号表
  + 2)插入式注解处理器的注解处理
  + 3)语义分析与字节码生成

## 解析与填充符号表

- 1)解析包括了词法分析和语法分析两个过程。

* 词法分析是将源代码的字符流变为Token序列；
* 语法分析是根据Token序列构造抽象语法树AST的过程
  + 2)填充符号表
* 符号表是由一组符号地址和符号信息构成的表格。
* 符号表中所登记的信息在编译的不同阶段都要用到。

## 插入式注解处理器的注解处理

* 插入式注解处理器可以视为一组编译器的插件，可以读取、修改、添加AST中的任意元素。如果在处理注解期间对AST进行了修改，那么编译器将回到解析与填充符号表的过程重新处理，每一次循环称为一个Record。

## 语义分析与字节码生成

* 语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查。
* 语义分析的过程分为Token检查和数据及控制流分析两个阶段。
  + 1)Token检查的内容包括变量使用前是否声明、变量和赋值之间的数据类型能否匹配，还有常量折叠等。
  + 2)数据及控制流分析是对程序上下文逻辑进行更进一步的验证，它可以检查出如程序员局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受检异常都被正确处理等。
  + 3)解语法糖：比如泛型、变长参数、自动装箱/拆箱等
  + 4)字节码生成：不仅仅是把签个各个步骤所生成的信息转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作，比如添加实例构造器()和类构造器()。

# 5.24 后端编译器的优化（JIT)

* 解决以下几个问题：
  + 1)为何HotSpot虚拟机要使用解释器和编译器并存的架构
  + 2)为何HotSpot虚拟机要实现两个不同的JIT
  + 3)程序何时使用解释器执行，何时使用编译器执行
  + 4)哪些程序代码会被编译为本地代码，如何编译为本地代码
  + 5)如何从外部观察JIT的编译过程和编译结果

## 编译器与解释器

* 解释器与编译器各有优势，前者节省内存，后者提高效率。
* 在整个虚拟机执行架构中，解释器与编译器经常配合工作。
* HotSpot虚拟机中内置了两个JIT，分别称为Client Compiler和Server Compiler。在虚拟机中习惯将Client Compiler称为C1，将Server Complier 称为C2。目前主流的HotSpot虚拟机中，默认采用解释器与其中一个编译器直接配合的方式工作，程序使用哪个编译器取决于虚拟机运行的模式。HotSpot虚拟机会根据自身版本与机器硬件性能自动选择运行模式，用户也可以使用-client或者-server参数去强制指定虚拟机运行的模式。
* 无论采用哪一种编译器，解释器与编译器搭配使用的方式在虚拟机中称为混合模式，用户可以使用参数-Xint强制虚拟机运行于解释模式，这时编译器完全不介入工作；也可以使用参数-Xcomp强制虚拟机运行于编译模式，这时将优先采用编译方式执行程序，但是解释器仍然要在编译无法进行的情况下介入执行过程。
* 为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot虚拟机会逐渐启用分层编译的策略。
* 第0层：程序解释执行，解释器不开启性能监控功能，可触发第1层编译
* 第1层：也称为C1编译，将字节码编译为本地代码，进行简单、可靠的优化，如有必须将加入性能监控的逻辑
* 第2层（或2层以上)：也称为C2编译，也是将字节码编译为本地代码，但是会启动一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。
* 实时分层编译后，Client Compiler和Server Compiler将会同时工作，很多代码都可能会被多次编译，用Client Compiler获取更高的编译速度，用Server Compile获取更好的编译质量，在解释执行的时候也无需再承担收集性能监控信息的任务。

## 编译对象与触发条件

* 在运行过程中被JIT编译的热点代码有两类：
  + 1)被多次调用的方法
  + 2)被多次执行的循环体

### 热点探测

* 编译器都会以整个方法作为编译对象。这种编译方式因为编译发生在方法执行过程之中，因此形象地成为栈上替换（On Stack Replacement OSR)。
* 判断一段代码是不是热点代码，是不是需要触发JIT，这样的行为称为热点探测。目前主流的热点探测判定方式有两种：
  + 1)基于采样的热点探测：采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这些方法就是热点方法。好处是简单高效，还可以很容易得获取方法调用关系，缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。
  + 2)基于计数器的热点探测：采用这种方法的虚拟机会为每个方法建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值就认为它是热点方法。好处是更加精确演进，缺点是实现较为麻烦。
* HotSpot采用的第二种方法，因为它为每个方法准备了两类计数器：方法调用计数器和回边计数器。这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发JIT编译。

### 方法调用计数器

* 当一个方法被调用时，会先检查该方法是否存在被JIT编译过的版本，如果存在，则优先使用编译后的本地代码来执行。如果不存在已被编译过的版本，则将此方法的调用计数器加一，然后判断方法调用计数器与回边计数器之和是否超过方法调用计数器的阈值。如果已超过阈值，那么会向JIT提交一个该方法的代码编译请求。
* 如果不做任何设置，方法调用计数器统计的并不是方法被调用的总次数，而是一个相对的执行频率，即一段时间内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给JIT编译，则这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器的衰减，而这段时间就称为此方法统计的半衰周期。

### 回边计数器

* 回边计数器的作用是统计一个方法中循环体的代码执行次数，在字节码中遇到控制流向后调换的指令称为回边，回边计数器统计的目的就是为了触发OSR编译。
* 当解释器遇到一条回边指令时，会先查找将要执行的代码片段是否有这已经编译好的版本，如果有，它将会优先执行已编译的代码，否则就把回边计数器的值加一，然后判断方法调用计数器与回边计数器之和是否超过回边计数器的阈值。当超过阈值时，将会提交一个OSR编译请求，并且把回边计数器的值降低一些，以便继续在解释器中执行循环，等待编译器输出编译结果。

## Client Compiler（编译速度快)

* 是一个简单快速的三段式编译器，主要的关注点在于局部性的优化，而放弃了许多耗时较长的全局优化手段。
* 三段式：
  + 1)第一个阶段，一个平台独立的前端会将字节码构造成一种高级中间代码表示（HIR High-Level Intermediate Representation)。HIR使用静态单分配的形式来表示代码值，这使得一些在HIR的构造过程之中和之后进行的优化动作更容易实现。
* 在此之前编译器会在字节码上完成一部分基础优化，如方法内联、常量传播等。
  + 2)第二个阶段，一个平台相关的后端从HIR中产生低级中间代码表示（LIR Low-Level Intermediate Representation)，而在此之前会在HIR上完成另外一些优化，如空值检查消除、范围检查消除等。
  + 3)第三个阶段，一个平台相关的后端使用线性扫描算法在LIR上分配寄存器，并在LIR上做窥孔优化，然后产生机器代码。

## Server Compiler（编译质量高)

* 是专门面向服务端的典型应用并为服务端的性能配置特别调整过的编译器，也是一个充分优化过的编译器。它会执行所有经典的优化动作，如无用代码消除、循环展开、循环表达式外提、消除公共子表达式、常量传播、基本块重排序等，还会实施一些与Java语言特性密切相关的优化技术，如范围检查消除、空值检查消除。另外还可能根据解释器或Client Compiler提供的性能监控信息，进行一些不稳定的激进优化，如守护内联、分值预测检测等。

## 编译优化

* 语言无关的经典优化技术之一：公共子表达式消除
* 语言相关的经典优化技术之一：数组范围检查消除
* 最重要的优化技术之一：方法内联
* 最前沿的优化技术之一：逃逸分析

### 语言相关的优化技术——逃逸分析

* 分析指针动态范围的方法称之为逃逸分析（通俗点讲，当一个对象的指针被多个方法或线程引用时们称这个指针发生了逃逸)。
* 逃逸分析并不是直接的优化手段，而是一个代码分析，通过动态分析对象的作用域，为其它优化手段如栈上分配、标量替换和同步消除等提供依据，发生逃逸行为的情况有两种：方法逃逸和线程逃逸。
* 1、方法逃逸：当一个对象在方法中定义之后，作为参数传递到其它方法中；
* 2、线程逃逸：如类变量或实例变量，可能被其它线程访问到；
* 如果不存在逃逸行为，则可以对该对象进行如下优化：同步消除、标量替换和栈上分配。
* 同步消除
* 线程同步本身比较耗，如果确定一个对象不会逃逸出线程，无法被其它线程访问到，那该对象的读写就不会存在竞争，则可以消除对该对象的同步锁，通过-XX:+EliminateLocks可以开启同步消除。
* 标量替换
* 1、标量是指不可分割的量，如java中基本数据类型和reference类型，相对的一个数据可以继续分解，称为聚合量；
* 2、如果把一个对象拆散，将其成员变量恢复到基本类型来访问就叫做标量替换；
* 3、如果逃逸分析发现一个对象不会被外部访问，并且该对象可以被拆散，那么经过优化之后，并不直接生成该对象，而是在栈上创建若干个成员变量；
* 通过-XX:+EliminateAllocations可以开启标量替换， -XX:+PrintEliminateAllocations查看标量替换情况。
* 栈上分配
* 故名思议就是在栈上分配对象，其实目前Hotspot并没有实现真正意义上的栈上分配，实际上是标量替换。

# 性能监控与故障处理工具

- 如果一个接口调用很慢，原因是，如何定位，没有日志的话：假设一下，复现问题，dump查看内存，查看监控日志  
- 如何把java内存的数据全部dump出来   
- 在生产线Dump堆分析程序是否有内存及性能问题  
- jstack jmap、jconsole 等工具 可视化工具使用；如何线上排查JVM的相关问题？  
- JVM线程死锁，你该如何判断是因为什么？如果用VisualVM，dump线程信息出来，会有哪些信息？  
- 查看jvm虚拟机里面堆、线程的信息，你用过什么命令？  
- 内存泄露如何定位

# 5.25 JPS：显示所有虚拟机进程

# 5.26 JConsole：图形化工具，查询JVM中的内存变化情况。

# 5.27 JVisualVM：图形化工具，分析GC趋势、内存消耗情况

* 可以分析堆dump文件

# 5.28 JMap：命令行工具，查看JVM中各个代的内存状况、JVM中对象的内存的占用状况，以及dump整个JVM中的内存信息。

* jmap –heap [pid] 整个JVM中内存的状况
* jmap –histo [pid] JVM堆中对象的详细占用情况
* jmap –dump:format=b,file=文件名 [pid] 将整个JVM内存拷贝到文件中

# 5.29 JHat 用于分析JVM堆的dump文件：jhat –J-Xmx1024M [file]

* 可以通过浏览器访问，端口号是7000.

# 5.30 JStack：看到JVM中线程的运行状况，包括锁的等待、线程是否在运行等。

* jstack [pid]
* jstack [option] pid
* jstack [option] executable core
* jstack [option] [server-id@]remote-hostname-or-ip
* 命令行参数选项说明如下：
* -l long listings，会打印出额外的锁信息，在发生死锁时可以用jstack -l pid来观察锁持有情况
* -m mixed mode，不仅会输出Java堆栈信息，还会输出C/C++堆栈信息（比如Native方法)
* jstack可以定位到线程堆栈，根据堆栈信息我们可以定位到具体代码，所以它在JVM性能调优中使用得非常多。下面我们来一个实例找出某个Java进程中最耗费CPU的Java线程并定位堆栈信息，用到的命令有ps、top、printf、jstack、grep。
* 第一步先找出Java进程ID，服务器上的Java应用名称为mrf-center：
* root@ubuntu:/# ps -ef | grep mrf-center | grep -v grep （或者直接JPS查看进程PID)
* root 21711 1 1 14:47 pts/3 00:02:10 java -jar mrf-center.jar
* 第二步 top -H -p pid
* 用第三个，输出如下：
* PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
* 21936 root 20 0 1747m 21m 9404 S 0.0 0.6 0:00.00 java
* 21937 root 20 0 1747m 21m 9404 S 0.0 0.6 0:00.14 java
* 21938 root 20 0 1747m 21m 9404 S 0.0 0.6 0:00.00 java
* 21939 root 20 0 1747m 21m 9404 S 0.0 0.6 0:00.00 java
* 21940 root 20 0 1747m 21m 9404 S 0.0 0.6 0:00.00 java
* TIME列就是各个Java线程耗费的CPU时间，CPU时间最长的是线程ID为21742的线程，用
* printf "%x\n" 21742
* 得到21742的十六进制值为54ee，下面会用到。
* OK，下一步终于轮到jstack上场了，它用来输出进程21711的堆栈信息，然后根据线程ID的十六进制值grep，如下：
* root@ubuntu:/# jstack 21711 | grep 54ee
* "PollIntervalRetrySchedulerThread" prio=10 tid=0x00007f950043e000 nid=0x54ee in Object.wait()
* 可以看到CPU消耗在PollIntervalRetrySchedulerThread这个类的Object.wait()，我找了下我的代码，定位到下面的代码：
* // Idle wait
* getLog().info("Thread [" + getName() + "] is idle waiting...");
* schedulerThreadState = PollTaskSchedulerThreadState.IdleWaiting;
* long now = System.currentTimeMillis();
* long waitTime = now + getIdleWaitTime();
* long timeUntilContinue = waitTime - now;
* synchronized(sigLock) {
* try {
* if(!halted.get()) {
* sigLock.wait(timeUntilContinue);
* }
* }
* catch (InterruptedException ignore) {
* }
* }
* 它是轮询任务的空闲等待代码，上面的sigLock.wait(timeUntilContinue)就对应了前面的Object.wait()。

# 5.31 JStat：JVM统计监测工具

* jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]] ]
* vmid是Java虚拟机ID，在Linux/Unix系统上一般就是进程ID。interval是采样时间间隔。count是采样数目。
* 比如下面输出的是GC信息，采样时间间隔为250ms，采样数为4：
* root@ubuntu:/# jstat -gc 21711 250 4
* S0C S1C S0U S1U EC EU OC OU PC PU YGC YGCT FGC FGCT GCT
* 192.0 192.0 64.0 0.0 6144.0 1854.9 32000.0 4111.6 55296.0 25472.7 702 0.431 3 0.218 0.649
* 192.0 192.0 64.0 0.0 6144.0 1972.2 32000.0 4111.6 55296.0 25472.7 702 0.431 3 0.218 0.649
* 192.0 192.0 64.0 0.0 6144.0 1972.2 32000.0 4111.6 55296.0 25472.7 702 0.431 3 0.218 0.649
* 192.0 192.0 64.0 0.0 6144.0 2109.7 32000.0 4111.6 55296.0 25472.7 702 0.431 3 0.218 0.649
* 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

# 5.32 MAT 可视化分析dump文件

* Memory Analyzer Tool

# 性能调优

# 5.33 参数

## 堆设置

* -Xms:初始堆大小
* -Xmx:最大堆大小
* -Xmn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

## 栈设置

* -Xss 设置每个线程的栈大小

## 元数据区设置

* -XX:MetaspaceSize -XX:MaxMetaspaceSize 元数据区的初始大小和最大大小

## 异常设置

* -XX:+HeapDumpOnOutOfMemoryError 使得JVM在产生内存溢出时自动生成堆内存快照（日后再进行分析，写监控脚本，如果发现应用崩溃则重启，并提醒开发人员去查看dump信息)
* -XX:HeapDumpPath 改变默认的堆内存快照生成路径，可以是相对或者绝对路径
* -XX:OnOutOfMemoryError 当内存发生溢出时 执行一串指令

## 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。

# 5.34 调优原则

* JVM的内存参数；xmx,xms,xmn,xss参数你有调优过吗，设置大小和原则你能介绍一下吗？；Xss默认大小，在实际项目中你一般会设置多大
* 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。

## 代大小的调优

* 1. 避免新生代大小设置过小，过小的话一是minor GC更加频繁，二是有可能导致minor GC对象直接进入老年代，此时如进入老年代的对象占据了老年代剩余空间，则触发Full GC。
  2. 避免新生代大小设置过大，过大的话一是老年代变小了，有可能导致Full GC频繁执行，二是minor GC的耗时大幅度增加。
  3. 避免Survivor区过小或过大
  4. 合理设置新生代存活周期，存活周期决定了新生代的对象经过多少次Minor GC后进入老年代，对应的JVM参数是-XX;MaxTenuringThreshold。

## GC策略的调优

* 串行GC性能太差，在实际场景中使用的主要为并行和并发GC。
* 由于CMS GC多数动作是和应用并发进行的，确实可以减小GC给应用带来的暂停。