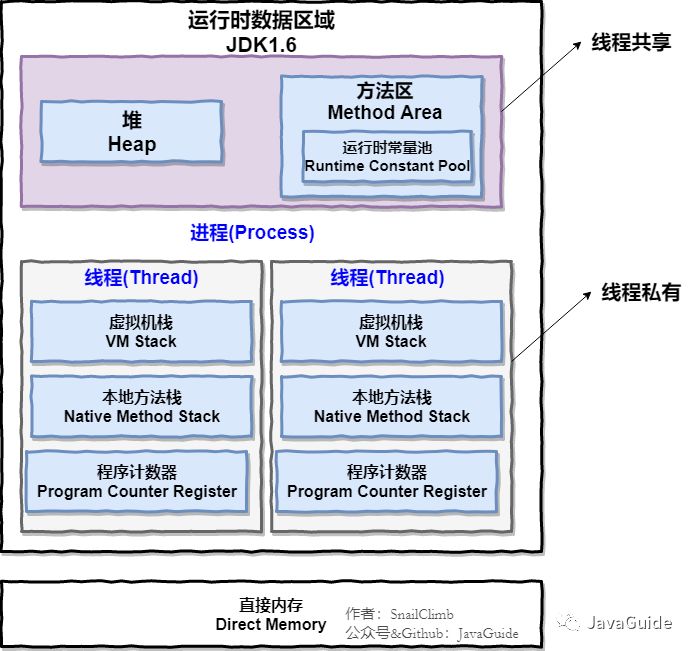
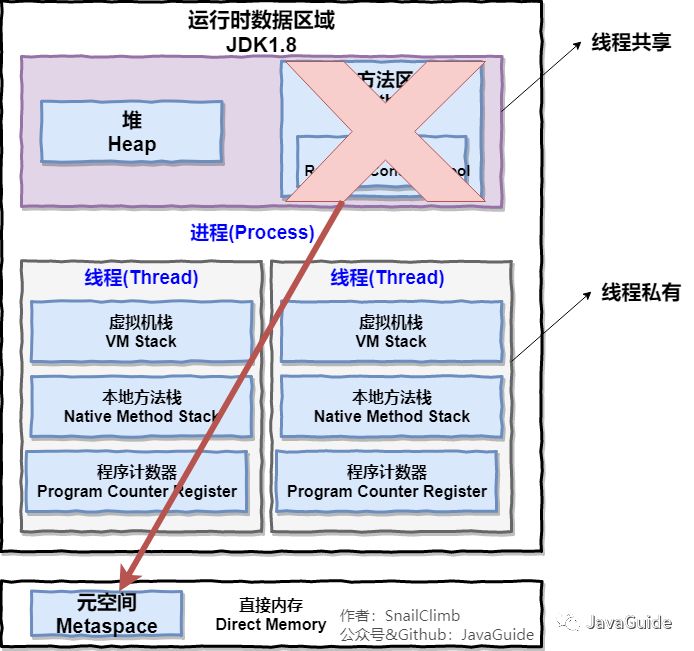
# JVM

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。JDK. 1.8 和之前的版本略有不同





## 程序计数器

从上面的介绍中我们知道程序计数器主要有两个作用：

•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。

•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。

## 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，每次方法调用的数据都是通过栈传递的。

（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方法有两种返回方式：

•return 语句。

•抛出异常。

不管哪种返回方式都会导致栈帧被弹出。

## 本地方法栈

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。

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。

## 堆

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区、s0区、s1区都属于新生代，tentired 区属于老年代。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。

## 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”，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。

JDK 1.8 的时候，方法区被彻底移除了（JDK1.7就已经开始了），取而代之是元空间，元空间使用的是直接内存。

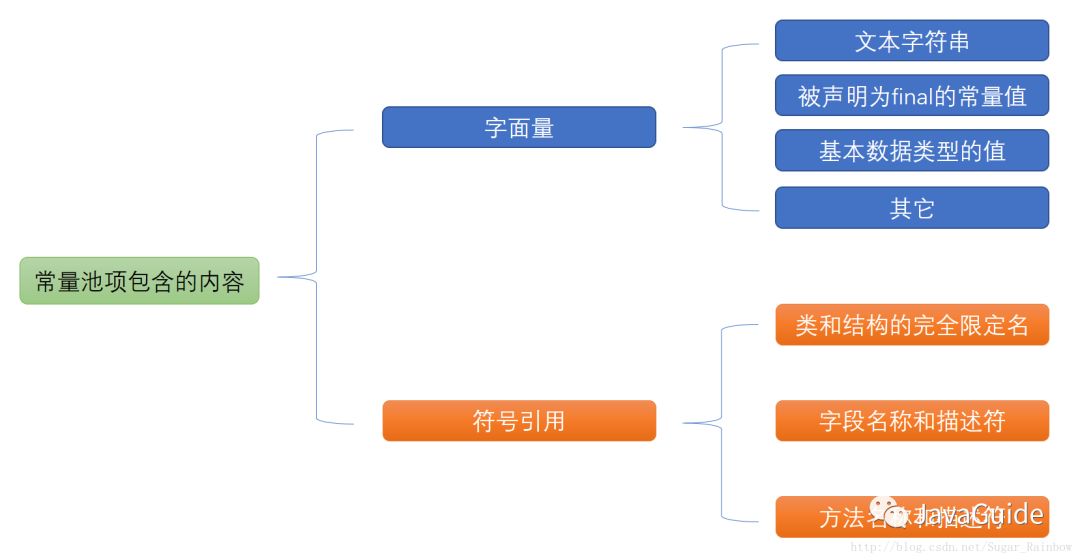
我们可以使用参数： -XX:MetaspaceSize 来指定元数据区的大小。与永久区很大的不同就是，如果不指定大小的话，随着更多类的创建，虚拟机会耗尽所有可用的系统内存。

## 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。



## 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。

## JAVA内存模型

当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。

我们说，并发编程，为了保证数据的安全，需要满足以下三个特性：

原子性是指在一个操作中就是cpu不可以在中途暂停然后再调度，既不被中断操作，要不执行完成，要不就不执行。

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性即程序执行的顺序按照代码的先后顺序执行。

有没有发现，缓存一致性问题其实就是可见性问题。而处理器优化是可以导致原子性问题的。指令重排即会导致有序性问题。

**计算机内存模型：**

为了保证共享内存的正确性（可见性、有序性、原子性），内存模型定义了共享内存系统中多线程程序读写操作行为的规范。通过这些规则来规范对内存的读写操作，从而保证指令执行的正确性。它与处理器有关、与缓存有关、与并发有关、与编译器也有关。他解决了CPU多级缓存、处理器优化、指令重排等导致的内存访问问题，保证了并发场景下的一致性、原子性和有序性。

内存模型解决并发问题主要采用两种方式：限制处理器优化和使用内存屏障。

**什么是Java内存模型**

Java内存模型（Java Memory Model ,JMM）就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。

Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中是用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。

而JMM就作用于工作内存和主存之间数据同步过程。他规定了如何做数据同步以及什么时候做数据同步。

特别需要注意的是，主内存和工作内存与JVM内存结构中的Java堆、栈、方法区等并不是同一个层次的内存划分，无法直接类比。《深入理解Java虚拟机》中认为，如果一定要勉强对应起来的话，从变量、主内存、工作内存的定义来看，主内存主要对应于Java堆中的对象实例数据部分。工作内存则对应于虚拟机栈中的部分区域。

所以，再来总结下，JMM是一种规范，目的是解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。目的是保证并发编程场景中的原子性、可见性和有序性。

**Java内存模型的实现**

了解Java多线程的朋友都知道，在Java中提供了一系列和并发处理相关的关键字，比如volatile、synchronized、final、concurren包等。其实这些就是Java内存模型封装了底层的实现后提供给程序员使用的一些关键字。

## JAVA内存分区

根据存储数据的不同，java内存通常被划分为5个区域：程序计数器（Program Count Register）、栈（Stack）、堆（Heap）、本地方法栈（Native Stack）、方法区（Methon Area）。

程序计数器:记录当前线程执行的下一个指令

栈：栈是运行时单位，JVM为每个新创建的线程都分配一个栈，栈以帧为单位保存线程的状态。当线程激活一个Java方法，JVM就会在线程的Java栈里新压入一个帧，栈帧中存储的有局部变量表、操作站、动态链接、方法出口等。

而本地方法栈跟栈类似，栈执行的是java方法，本地方法栈则为虚拟机使用到的native方法服务，存储了每个native方法调用的状态。

堆是存储单位，此区域的唯一目的就是存放对象实例。

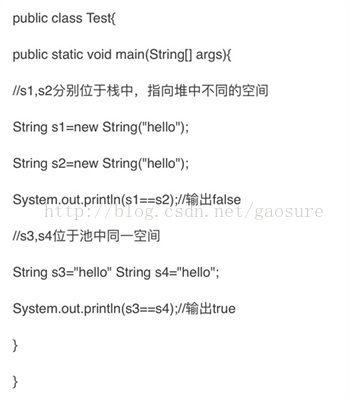
而方法区用于存放类的信息(如类名，修饰符)、类中的静态变量、final定义的常量、类中的field、方法信息。

而方法区中还有一块特殊的内存叫运行时常量池，用于存放字面常量、符号引用、翻译出来的直接引用, 也可以存储在运行时间产生的常量。

JDK1.8已经没有方法区了，改为了元空间。元空间的本质和方法区类似，都是对JVM规范中方法区的实现。不过元空间与方法区之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。

**常量池的作用**非常重要，常量池中除了包含代码中所定义的各种基本类型（如int、long等等）和对象型（如String及数组）的常量值外，还包含一些以文本形式出现的符号引用，比如：类和接口的全限定名，字段的名称和描述符，方法和名称和描述符。

Double包装类没有实现常量池技术，因此Double d1 = 1.0; 相当于Double d1 = new Double(1.0);，是从堆new一个对象，



结果分析：  用new String() 创建的字符串不是常量，不能在编译器就能确定，所以new String()创建的字符串不放入常量池中，他们有自己的地址空间。  String对象(内存)的不变性机制会使修改String字符串时，产生大量的对象，因为每次改变字符串，都会生成一个新的String。Java为了更有效的使用内存，常量池在编译期遇见String字符串时，它会检查该池内是否已经存在相同的String字符串，如果找到，就把新变量的引用指向现有的字符串对象，不创建任何新的String常量对象，没找到再创建新的。所以对一个字符串对象的任何修改，都会产生一个新的字符串对象，原来的依然存在，等待垃圾回收。  String a = "test";  String b = "test";  String b = b + "java";  a、b同时指向常量池中的常量值”test“，b = b + "java"之后，b原先指向一个常量，内容为"test"，通过对b进行+"java"操作后，b之前所指向的那个值没有改变，但此时b不指向原来那个变量值了，而指向了另一个String变量，内容为”test java“。原来那个变量还存在于内存之中，只是b这个变量不再指向它了。

小结：

1、**常量池维护的常量仅仅是【-128至127】这个范围内的常量，如果常量值超过这个范围，就会从堆中创建对象，不再从常量池中取。**如：Integer i1 = 400；Integer i2 = 400；很明显超过了127，无法从常量池中获取常量，就用从堆中new新的Integer对象，这是i1和i2就不相等了。

**2、String类型也实现了常量池技术，但是稍微有点不同，String型是先检测常量池中有没有对应字符串，如果有，则取出来，如果没有，则把当前的添加进去。**

## JVM中会抛出的异常

虚拟机栈中定义了两种异常，如果线程调用的栈深度大于虚拟机允许的最大深度，则抛出**StatckOverFlowError**（栈溢出）；不过多数Java虚拟机都允许动态扩展虚拟机栈的大小(有少部分是固定长度的)，所以线程可以一直申请栈，直到内存不足，此时，会抛出**OutOfMemoryError**（内存溢出）。

年老代被占满：java.lang.OutOfMemoryError:java heap space

持久代被占满：java.lang.OutOfMemoryError:permGen space

## JVM调优

调优工具：Jconsole,jProfile,visualVM

如何调优：

堆信息查看：

可查看堆空间大小分配（年轻代，年老代，持久代）

提供即时的垃圾回收功能

垃圾监控

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

年老代年轻代大小划分是否合理

内存泄漏

垃圾回收算法设置是否合理

查看FullGC的频率

Linux使用jstat命令查看jvm的GC情况，通常运行命令如下：

jstat -gc 12538 5000

即会每5秒一次显示进程号为12538的java进成的GC情况

性能调优建议： jvm调优没有一个固定模板配置说必须如何操作，它需要根据系统的情况不同对待。 但是可以有如下建议：

1、初始化内存和最大内存尽量保持一致，避免内存不够用继续扩充内存。最大内存不要超过物理内存，例如内存8g，你可以设置最大内存4g/6g但是不能超过8g否则加载类的时候没有空间会报错。

2、gc/full gc频率不要太高、每次gc时间不要太长、根据系统应用来定。

## JVM垃圾回收方式

1 引用计数收集器

引用计数器采用分散式管理方式，通过计数器记录对象是否被引用。当计数器为0时，说明此对象已经不再被使用，可进行回收

2**复制（Copying）、**

**3标记-清除（Mark-Sweep）**

**标记失去引用的对象并回收，这个算法会产生碎片**

**4标记-整理（Mark-Compact）**

在标记清除的基础上整理内存空间

**计算对象是否引用**

**1.采用标记计数的方法：**给内存中的对象给打上标记，对象被引用一次，计数就加1，引用被释放了，计数就减一，当这个计数为0的时候，这个对象就可以被回收了。当然，这也就引发了一个问题：循环引用的对象是无法被识别出来并且被回收的。所以就有了第二种方法：

**2.采用根搜索算法：**从一个根出发，搜索所有的可达对象，这样剩下的那些对象就是需要被回收的判断完了哪些对象是没用的，这样就可以进行回收了

## 虚拟机中的GC过程

分代回收

Eden跟Survivor0,Survivor1采用的垃圾回收方式是著名的**“停止-复制（Stop-and-copy）”**清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中）

老年代用的算法是标记-整理算法（串行收集器）

关于方法区即永久代的回收，永久代的回收有两种：常量池中的常量，无用的类信息，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1. 类的索引实例都已经被回收。2.加载类的ClassLoader已经被回收。3.类对象的Class对象没有被引用（即没有通过反射引用该类的地方）

## 垃圾收集器

1 串行（Serial）收集器：一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。

2并行收集器：多条垃圾收集线程并行工作，而用户线程仍处于等待状态。标记整理，复制算法都是采用并行收集器。

3并发收集器：垃圾收集线程与用户线程一段时间内同时工作（不是并行，而是交替执行）

4 CMS（Concurrent Mark Sweep）收集器（并发收集器的一种）

5 G1收集器

G1即Garbage First，它是在java 7中出现的新的收集器，采用了另外一种完全不同的方式组织堆内存，堆内存被划分为多个大小相等的内存块（Region），每个Region是逻辑连续的一段内存。G1具有并行、并发、增量压缩、暂停时间段等特点

CMS收集器的GC周期由6个阶段组成。其中4个阶段(名字以Concurrent开始的)与实际的应用程序是并发执行的，而其他2个阶段需要暂停应用程序线程。

初始标记：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。  
  并发标记：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用。  
  并发预清理：改变当运行第二阶段时，由应用程序线程产生的对象引用，以更新第二阶段的结果。  
  重标记：由于第三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。  
  并发清理：所有不再被应用的对象将从堆里清除掉。

并发重置：收集器做一些收尾的工作，以便下一次GC周期能有一个干净的状态。

一个常见的误解是,CMS收集器运行是完全与应用程序并发的。我们已经看到，事实并非如此，即使“stop-the-world”阶段相对于并发阶段的时间很短。  
  
应该指出，尽管CMS收集器为老年代垃圾回收提供了几乎完全并发的解决方案，然而年轻代仍然通过“stop-the-world”方法来进行收集。对于交互式应用，停顿也是可接受的，背后的原理是年轻代的垃圾回收时间通常是相当短的。  
挑战

当我们在真实的应用中使用CMS收集器时，我们会面临两个主要的挑战，可能需要进行调优：  
堆碎片;对象分配率高

CMS收集器并没有任何碎片整理的机制。

第二个挑战就是应用的对象分配率高。如果获取对象实例的频率高于收集器清除堆里死对象的频率，并发算法将再次失败。

G1算法：

G1把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位；

增量收集把对堆空间划分成一系列内存块，使用时，先使用其中一部分（不会全部用完），垃圾收集时 把之前用掉的部分中的存活对象再放到后面没有用的空间中

回收步骤：

初始标记（Initial Marking） stop the world

并发标记（Concurrent Marking）

最终标记暂停（Final Marking Pause） stop the world

存活对象计算及清除（Live Data Counting and Cleanup）

## JVM编译优化

当虚拟机发现某个方法或代码块的运行特别频繁的时候，就会把这些代码认定为“热点代码”。为了提高热点代码的执行效率，在运行时，即时编译器（Just In Time Compiler ）会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化。

## JVM内存分配

Java对象所占用的内存主要在堆上实现，因为堆是线程共享的，因此在堆上分配内存时需要进行加锁，这就导致了创建对象的开销比较大。当堆上空间不足时，会出发GC，如果GC后空间仍然不足，则会抛出OutOfMemory异常。

为了提升内存分配效率，在年轻代的Eden区HotSpot虚拟机使用了两种技术来加快内存分配 ，分别是**bump-the-pointer**和**TLAB（Thread-Local Allocation Buffers）**。由于Eden区是连续的，因此bump-the-pointer技术的核心就是跟踪最后创建的一个对象，在对象创建时，只需要检查最后一个对象后面是否有足够的内存即可，从而大大加快内存分配速度；而对于TLAB技术是对于多线程而言的， 它会为每个新创建的线程在新生代的Eden Space上分配一块独立的空间，这块空间称为TLAB（Thread Local Allocation Buffer），在TLAB上分配内存不需要加锁，一般JVM会优先在TLAB上分配内存，如果对象过大或者TLAB空间已经用完，则仍然在堆上进行分配。

## JDK7和8中内存模型变化

9.JDK7和8中内存模型变化：JDK7中把String常量池从永久代移到了堆中，并通过intern方法来保证不在堆中重复创建一个对象；JDK7开始使用G1收集器替代CMS收集器。JDK8使用元空间来替代原来的方法区，并且提供了字符串去重功能，也就是G1收集器可以识别出堆中那些重复出现的字符串并让他们指向同一个内部char[]数组，而不是在堆中存在多份拷贝

## 垃圾收集器配置

深入JVM-垃圾收集器常用的GC参数

1.与串行回收器相关的参数

-XX:+UseSerialGC:在新生代和老年代使用串行收集器

-XX:SurvivorRatio:设置eden区大小和survivor区大小的比例

-XX:PretenureSizeThreshold:设置大对象直接进入老年代的阈值。当对象的大小超过这个值时，将直接在老年代分配。

-XX:MaxTenuringThreshold:设置对象进入老年代的年龄的最大值。每一次Minor GC后，对象年龄就加1。任何大于这个年龄的对象，一定会进入老年代。

2.与并行GC相关的参数

-XX:+UseParNewGC:在新生代使用并行收集器

-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。通常情况下可以和CPU数量相等，但在CPU数量较多的情况下，设置相对较小的数值也是合理的。

-XX:MaxGCPauseMillis:设置最大垃圾收集停顿时间。他的值是一个大于0的整数。收集器在工作时，会调整Java堆大小或者其他参数，尽可能把停顿时间控制在MaxGCPauseMillis以内。

-XX:GCTimeRatio:设置吞吐量大小。它是0-100的整数。假设GCTimeRatio的值为n，那么系统将花费不超过1/(1+n)的时间用于垃圾收集。

-XX:+UseAdaptiveSizePolicy:打开自适应GC策略。在这种模式下，新生代的大小、eden和survivor的比例、晋升老年代的对象年龄等参数会被自动调整，已达到在堆大小、吞吐量和停顿时间之间的平衡点。

3.与CMS回收期相关的参数

-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:CMSInitiatingOccupancyFraction:设置CMS收集器在老年代空间被使用多少后触发，默认为68%

-XX:+UseCMSCompactAtFullCollection:设置CMS收集器完成垃圾收集后是否要进行一次内存碎片的整理

-XX:CMSFullGCsBeforeCompaction:设定进行多少次CMS垃圾回收后，进行一次内存压缩

-XX:+CMSClassUnloadingEnabled:允许对类元数据区进行回收

-XX:CMSInitiatingPermOccupancyFraction:当永久区占用率达到这一百分比时，启动CMS回收(前提是-XX:+CMSClassUnloadingEnabled激活了)

-XX:UseCMSInitiatingOccupancyOnlyn:表示只在到达阈值的时候才进行CMS回收

-XX:+CMSIncrementalMode:使用增量模式，比较适合单CPU。增量模式在JDK 8中标记为废弃，并将在JDK 9中彻底移除。

4.与G1回收期相关的参数

-XX:+UseG1GC:使用G1回收器

-XX:MaxGCPauseMillis:设置最大垃圾收集停顿时间

-XX:GCPauseIntervalMillis:设置停顿间隔时间

5.TLAB相关

-XX:+UseTLAB:开启TLAB分配

-XX:+PrintTLAB:打印TLAB相关分配信息

-XX:TLABSize:设置TLAB大小

-XX:+ResizeTLAB:自动调整TLAB大小

6.其他参数

-XX:+DisableExplicitGC:禁用显式GC

-XX:+ExplicitGCInvokesConcurrent:使用并发方式处理显式GC

## 其它

Class a = new Class();此时a叫实例，而不能说是对象。实例在栈中，对象在堆中，操作实例实际上是通过实例的指针间接操作对象。多个实例可以指向同一个对象。

JVM类加载过程：加载（将class相关信息载入持久代）->链接（验证+准备+解析）->初始化（调用构造函数实例化）->使用->卸载

Java对象访问方式

1 通过句柄访问

通过句柄访问的实现方式中，JVM堆中会划分单独一块内存区域作为句柄池，句柄池中存储了对象实例数据（在堆中）和对象类型数据（在方法区中）的指针。这种实现方法由于用句柄表示地址，因此十分稳定。

2通过直接指针访问

通过直接指针访问的方式中，reference中存储的就是对象在堆中的实际地址，在堆中存储的对象信息中包含了在方法区中的相应类型数据。这种方法最大的优势是速度快，在HotSpot虚拟机中用的就是这种方式。

内存泄露的两个条件：无用，无法回收

长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露因为长生命周期对象持有它的引用而导致不能被回收。其中一个例子：HashSet中存储的对象改变了其用来计算hashcode的字段

新生代的垃圾收集器命名为“minor gc”，老生代的GC命名为"Full Gc 或者Major GC".其中用System.gc()强制执行的是Full Gc.

判断对象是否需要回收的方法有两种： 1**.**引用计数 2.对象引用遍历

符号引用和直接引用的概念：

符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

双亲委派过程：

当一个类加载器收到类加载任务时，立即将任务委派给它的父类加载器去执行，直至委派给最顶层的启动类加载器为止。如果父类加载器无法加载委派给它的类时，将类加载任务退回给它的下一级加载器去执行。

变量覆盖：ParentClass subClass = new SubClass();

子类跟父类有相同普通变量的情况下，subClass.protectedField输出的是父类定义的变量

Linux下面查看Jvm性能信息的命令

1. **jstat**： 用于查看Jvm的堆栈信息，能够查看eden,survivor,old,perm等堆区的的容量，利用率信息，对于查看系统是不是有内存泄漏以及参数设置是否合理有不错的意义。例如’’’ jstat -gc 12538 5000 —- 即会每5秒一次显示进程号为12538的java进成的GC情况 ‘’’
2. **jstack**：用来查看Jvm当前的线程dump的，可以看到当前Jvm里面的线程状况，对于查找blocked线程比较有意义
3. **jmap**：用来查看Jvm当前的heap dump的，可以看出当前Jvm中各种对象的数量，所占空间等等；尤其值得一提的是这个命令可以导出一份binary heap dump的bin文件，这个文件能够直接用Eclipse Memory Anayliser来分析，并找出潜在的内存泄漏的地方。
4. **非jvm命令—netstat**：通过这个命令可以看到Linux系统当前在各个端口的链接状态，比如查看数据库连接数等

内存相关问题

**1.内存泄露**是指分配出去的内存没有被回收回来，由于失去了对该内存区域的控制(例如你把它的地址给弄丢了)，因而造成了资源的浪费。Java 中一般不会产生内存泄露，因为有垃圾回收器自动回收垃圾，但这也不绝对，Java堆内也可能发生内存泄露（Memory Leak; 当我们 new 了对象，并保存了其引用，但是后面一直没用它，而垃圾回收器又不会去回收它，这边会造成内存泄露

**2.内存溢出**是指程序所需要的内存超出了系统所能分配的内存（包括动态扩展）的上限

**3.符号引用**：符号引用以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，只要使用时能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已经加载到了内存中。

**4.直接引用**：直接引用可以是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。直接引用是与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，那说明引用的目标必定已经存在于内存之中了。

**5.双亲委派模型**：表示类加载器之间的加载顺序从顶至下的层次关系，加载器之间的父子关系一般都是通过组合来实现，而不是继承。可以防止内存中出现多份同样的字节码，并确保加载顺序

**6.双亲委派模型的工作过程**是：在loadClass函数中，首先会判断该类是否被加载过，加载过则进行下一步—-解析，否则进行加载；如果一个类加载器收到了类加载器的请求，先不会自己尝试加载这个类，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，只有当父类加载器反馈自己无法完成这个加载请求（它的搜说范围中没有找到所需的类时，子加载类才会尝试自己去加载）

**7.静态分派和动态分派**：静态分派发生在编译阶段，是指依据静态类型(变量声明时定义的变量类型)来决定方法的执行版本，例如方法重载中依据参数的定义类型来定位具体应该执行的方法；动态分派发生在运行期，根据变量实例化时的实际类型来决定方法的执行版本，例如方法重写；目前的 Java 语言（JDK1.6）是一门**静态多分派、动态单分派**的语言。

**8.动态分派具体实现**Java虚拟机是通过在方法区中建立一个虚方法表，通过使用方法表的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，如果子类没有覆盖父类的方法，那么子类的虚方法表里面的地址入口与父类是一致的；如果重写父类的方法，那么子类的方法表的地址将会替换为子类实现版本的地址。方法表是在类加载的连接阶段（验证、准备、解析）进行初始化，准备了子类的初始化值后，虚拟机会把该类的虚方法表也进行初始化。

# 类加载

## 类的初始化顺序

1. 父类--静态变量

2. 父类--静态初始化块

3. 子类--静态变量

4. 子类--静态初始化块

5. 父类--变量

6. 父类--初始化块

7. 父类--构造器

8. 子类--变量

9. 子类--初始化块

10.子类--构造器

## 类加载七个阶段以及案例分析

而 JVM 虚拟机执行 class 字节码的过程可以分为七个阶段：加载、验证、准备、解析、初始化、使用、卸载。

加载：在这个阶段，JVM 的主要目的是将字节码从各个位置（网络、磁盘等）转化为二进制字节流加载到内存中，接着会为这个类在 JVM 的方法区创建一个对应的 Class 对象，这个 Class 对象就是这个类各种数据的访问入口。

验证：JVM规范校验，代码逻辑校验。

准备（重要）：

1.内存分配的对象：为类成员变量分配即静态变量内存，在准备阶段，JVM 只会为「类变量」分配内存，而不会为「类成员变量」分配内存。

2.初始化的类型。在准备阶段，JVM 会为类变量分配内存，并为其初始化。但是这里的初始化指的是为变量赋予 Java 语言中该数据类型的零值，而不是用户代码里初始化的值。

例如下面的代码在准备阶段之后，sector 的值将是 0，而不是 3。public static int sector = 3;

但如果一个变量是常量（被 static final 修饰）的话，那么在准备阶段，属性便会被赋予用户希望的值。

例如下面的代码在准备阶段之后，number 的值将是 3，而不是 0。

public static final int number = 3;

解析：当通过准备阶段之后，JVM 针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类引用进行解析。这个阶段的主要任务是将其在常量池中的符号引用替换成直接其在内存中的直接引用。

初始化（重要）：到了初始化阶段，用户定义的 Java 程序代码才真正开始执行。在这个阶段，JVM 会根据语句执行顺序对类对象进行初始化，一般来说当 JVM 遇到下面 5 种情况的时候会触发初始化：

遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译器把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

当使用 JDK1.7 动态语言支持时，如果一个 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初始化，则需要先出触发其初始化。

使用：当 JVM 完成初始化阶段之后，JVM 便开始从入口方法开始执行用户的程序代码。这个阶段也只是了解一下就可以。

卸载：当用户程序代码执行完毕后，JVM 便开始销毁创建的 Class 对象，最后负责运行的 JVM 也退出内存。这个阶段也只是了解一下就可以。

案例分析1:

class Grandpa

{

static

{

System.out.println("爷爷在静态代码块");

}

public Grandpa() {

System.out.println("我是爷爷~");

}

}

class Father extends Grandpa

{

static

{

System.out.println("爸爸在静态代码块");

}

public Father()

{

System.out.println("我是爸爸~");

}

}

class Son extends Father

{

static

{

System.out.println("儿子在静态代码块");

}

public Son()

{

System.out.println("我是儿子~");

}

}

public class InitializationDemo

{

public static void main(String[] args)

{

new Son(); //入口

}

}

输出结果：

爷爷在静态代码块

爸爸在静态代码块

儿子在静态代码块

我是爷爷~

我是爸爸~

我是儿子~

静态字段，只有直接定义这个字段的类才会被初始化（执行静态代码块），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。

案例分析2

public class Book {

public static void main(String[] args)

{

staticFunction();

}

static Book book = new Book();

static

{

System.out.println("书的静态代码块");

}

{

System.out.println("书的普通代码块");

}

Book()

{

System.out.println("书的构造方法");

System.out.println("price=" + price +",amount=" + amount);

}

public static void staticFunction(){

System.out.println("书的静态方法");

}

int price = 110;

static int amount = 112;

}

上面这个例子的输出结果是：

书的普通代码块

书的构造方法

price=110,amount=0

书的静态代码块

书的静态方法

下面我们一步步来分析一下代码的整个执行流程。

在上面两个例子中，因为 main 方法所在类并没有多余的代码，我们都直接忽略了 main 方法所在类的初始化。但在这个例子中，main 方法所在类有许多代码，我们就并不能直接忽略了。

当 JVM 在准备阶段的时候，便会为类变量分配内存和进行初始化。此时，我们的 book 实例变量被初始化为 null，amount 变量被初始化为 0。

当进入初始化阶段后，因为 Book 方法是程序的入口，根据我们上面说到的类初始化的五种情况的第四种：当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。JVM 会对 Book 类进行初始化。

JVM 对 Book 类进行初始化首先是执行类构造器（按顺序收集类中所有静态代码块和类变量赋值语句就组成了类构造器），后执行对象的构造器（先收集成员变量赋值，后收集普通代码块，最后收集对象构造器，最终组成对象构造器）。

对于 Book 类，其类构造方法可以简单表示如下：

static Book book = new Book();

static

{

System.out.println("书的静态代码块");

}

static int amount = 112;

于是首先执行static Book book = new Book();这一条语句，这条语句又触发了类的实例化。与类构造器不同，于是 JVM 执行 Book 类的成员变量，再搜集普通代码块，最后执行类的构造方法，于是其执行语句可以表示如下：

int price = 110;

{

System.out.println("书的普通代码块");

}

Book()

{

System.out.println("书的构造方法");

System.out.println("price=" + price +", amount=" + amount);

}

于是此时 price 赋予 110 的值，输出：「书的普通代码块」、「书的构造方法」。而此时 price 为 110 的值，而 amount 的赋值语句并未执行，所以只有在准备阶段赋予的零值，所以之后输出「price=110,amount=0」。

当类实例化完成之后，JVM 继续进行类构造器的初始化：

static Book book = new Book(); //完成类实例化

static

{

System.out.println("书的静态代码块");

}

static int amount = 112;

即输出：「书的静态代码块」，之后对 amount 赋予 112 的值。

到这里，类的初始化已经完成，JVM 执行 main 方法的内容。

public static void main(String[] args)

{

staticFunction();

}

即输出：「书的静态方法」。

分析方法论

从上面几个例子可以看出，分析一个类的执行顺序大概可以按照如下步骤：

确定类变量的初始值。在类加载的准备阶段，JVM 会为类变量初始化零值，这时候类变量会有一个初始的零值。如果是被 final 修饰的类变量，则直接会被初始成用户想要的值。

初始化入口方法。当进入类加载的初始化阶段后，JVM 会寻找整个 main 方法入口，从而初始化 main 方法所在的整个类。当需要对一个类进行初始化时，会首先初始化类构造器，之后初始化对象构造器。

初始化类构造器。初始化类构造器是初始化类的第一步，其会按顺序收集类变量的赋值语句、静态代码块，最终组成类构造器由 JVM 执行。

初始化对象构造器。初始化对象构造器是在类构造器执行完成之后的第二部操作，其会按照执行类成员变成赋值、普通代码块、对象构造方法的顺序收集代码，最终组成对象构造器，最终由 JVM 执行。

如果在初始化 main 方法所在类的时候遇到了其他类的初始化，那么继续按照初始化类构造器、初始化对象构造器的顺序继续初始化。如此反复循环，最终返回 main 方法所在类。

## 类加载机制

Java虚拟机与程序的生命周期

**在如下几种情况下，Java虚拟机将结束生命周期**

– 执行了System.exit()方法

– 程序正常执行结束

– 程序在执行过程中遇到了异常或错误而异常终止

– 由于操作系统出现错误而导致Java虚拟机进程终止

**将class文件加载到内存到过程：**

加载：查找并加载类的二进制数据，加载到内存里面就会生成class对象，即调用ClassLoader的loaderClass方法。

连接

– 验证：确保被加载的类的正确性

– 准备：为类的 静态变量分配内存，并将其初始化为默认值

– 解析： 把类中的符号引用转换为直接引用

初始化：为类的静态变量赋予正确的初始值，这个正确的初始值，指我们赋予静态变量的值。如：

private static int a = 3;在准备阶段会被a设为默认值0，执行初始化时把3赋予了a。

静态代码块从上到下执行

**Java程序对类的使用方式可分为两种**

– 主动使用

– 被动使用

所有的Java虚拟机实现必须在每个类或接口被Java程序“ 首次主动使用”时才初始化他们

**主动使用（ 六种）**

– 创建类的实例

– 访问某个类或接口的静态变量，或者对该静态变量赋值。（int c = Test.a;或Test.a=b）

– 调用类的静态方法

– 反射（如Class.forName(“com.shengsiyuan.Test”)）

– 初始化一个类的子类

– Java虚拟机启动时被标明为启动类的类（JavaTest）。比如说有Test.java这个文件，编译后有Test.class,Parent.class,Child.class，然后启动时使用java Test（Test有main方法），这就是被标明为启动类的类

除了以上六种情况，其他使用Java类的方式都被看作是对类的被动使用，都不会导致类的初始化

**类的加载指的是**将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。一个类只有一个Class对象，只有JAVA虚拟机才能创建Class对象。

**加载.class文件的方式**

– 从本地系统中直接加载

– 通过网络下载.class文件。使用java.netURLClassLoader加载。

– 从zip ，jar 等归档文件中加载.class 文件

– 从专有数据库中提取.class文件

– 将 Java源文件动态编译为.class文件

包名跟类名重复时，ClassLoader如何处理？？

**类的加载的最终产品是位于堆区中的**

**Class 对象**

Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口，这个接口是反射的接口。

**有两种类型的类加载器**

– Java 虚拟机自带的加载器

根类加载器（Bootstrap，使用C++编写，程序员无法在java代码中获得该类）

扩展类加载器（Extension，使用java代码实现）

系统类加载器（System，使用java代码实现）

– 用户自定义的类加载器

必须继承java.lang.ClassLoader类

用户可以定制类的加载方式

Class的getClassLoader()方法如果这个类是根类加载器加载会返回一个空

Class clazz = Class.forName("java.lang.String");

clazz.getClassLoader();//返回null

**类加载器** 并不需要等到某个类被“首次主动使用”时再加载它。JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）。如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。

类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

类的验证的内容

– 类文件的结构检查

– 语义检查

– 字节码验证

– 二进制兼容性的验证

**类的准备**：在准备阶段，java虚拟机为类的静态变量分配内存，并设置默认的初始值。例如对于以下Sample类，在准备阶段，将为Int类型的静态变量a分配4个字节的内存空间，并且赋予默认值0，为long类型的静态变量b分配8个字节的内存空间，并且赋予默认值0.

Public Class Sample{

Private static int a = 1;

Private static long b;

Static{

B=2;

}

}

**类的解析：** 在解析阶段，java虚拟机会吧类的二进制数据中的符号引用替换为直接引用。例如在Worker类的gotoWork()方法中会引用Car类的run()方法。

Pubic void gotoWork(){

Car.run();//这段代码在worker类的二进制数据中表示为符号引用

}

在worker类的二进制数据中，包含类一个对car类对run方法对符号引用，它由run()方法的全名和相关描述符组成。在解析阶段，java虚拟机会把这个符号引用替换为一个指针，该指针指向car类的run()方法在方法区内的内存位置，这个指针就是直接引用。

**类的初始化**: 在初始化阶段，java虚拟机执行类的初始化语句，为类的静态变量赋予初始值。在程序中，静态变量的初始化有两种途径：1.在静态变量的声明处进行初始化；2.在静态代码块中进行初始化。例如在以下代码中，静态变量a和b都被显示初始化，而静态变量c没有被显示初始化，它将保持默认值0.

public class Sample{

private static int a=1;//在静态变量的声明处进行初始化

public static long b;

public static long c;

static{

b=2; //在静态代码块中进行初始化

}

}

**类的初始化步骤**：

1.这个类还没有被加载和连接，那就先进行加载和连接

2.假如类存在直接的父类，并且这个父类还没有被初始化，那就先初始化直接父类

3.假如类中存在初始化语句，那就依次执行这些初始化语句。

**类的初始化时机**

当java虚拟机初始化一个类时，要求它的所有父类都已经被初始化，但是这条规则并不适用于接口

在初始化一个类时，并不会先初始化它所实现的接口。

在初始化一个接口时，并不会先初始化它的父接口

因此，一个父接口并不会因为它的子接口或实现类的初始化而初始化。只有当程序首次使用特定接口的静态变量时，才会导致该接口的初始化。

**面试题一**

class Singleton{

//private static Singleton singleton = new Singleton();

public static int counter1;

public static int counter2 = 0;

private static Singleton singleton = new Singleton();

private Singleton(){

counter1++;

counter2++；

｝

public static Singleton getInstance(){

return singleton;

}

}

public class MyTest

{

public static void main(String[] args){

Singleton singleton = Singleton.getInstance();

输出counter1跟counter2;

}

}

如果把private static Singleton singleton = new Singleton();放在最前面的话是输出1跟0。

原因：类加载器在准备阶段会先将counter1跟counter2都赋予默认值0，这时再依次执行静态代码块，首先实例化singleton把值都设为了1，然后发现counter2有显示赋值，所以将用户自己设定的值0赋予counter2，导致counter2最后的结果是0

**面试题二**

**注意：此处是针对final的静态变量**

class FinalTest1{

public static final int x = 6/3;

static {

System.out.println("FinalTest2.enclosing\_method()");

}

public FinalTest1() {

System.out.println("FinalTest2.FinalTest2()");

}

}

class FinalTest2{

public static final int x = new Random().nextInt(100);

static {

System.out.println("FinalTest2.enclosing\_method()");

}

public FinalTest2() {

System.out.println("FinalTest2.FinalTest2()");

}

}

public class Test1 {

public static void main(String[] args) {

//只输出2，因为x在编译期就已确定，所以使用的时候不会去初始化类

System.out.println(FinalTest1.x);

System.out.println("==============");

//输出x的随机数跟FinalTest2.enclosing\_method()，因为x需要运行时才能确定，所以要对类进行初始化

//进不进行初始化就看这个静态代码块是否会执行。

System.out.println(FinalTest2.x);

}

}

**面试题三：与上题做对比**

class Parent{

static int a = 4;

static {

System.out.println("Parent.enclosing\_method()");

}

}

class Child extends Parent{

static int a = 5;

static {

System.out.println("Child.enclosing\_method()");

}

}

public class Test3 {

static {

System.out.println("Test3.enclosing\_method()");

}

public static void main(String[] args) {

/\*

\* 输出：

\* Test3.enclosing\_method()

Parent.enclosing\_method()

Child.enclosing\_method()

5

\*/

System.out.println(Child.a);

}

}

同一个类加载器不允许对一个类初始化两次

程序中对子类对主动使用会导致父类被初始化；但父类但主动使用并不会导致子类初始化（不可能说生成一个object类会导致所以子类初始化。）

只有当程序访问当静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口对主动使用。如：

**面试题四：只是对父类对主动使用，没有对子类主动使用**

class Parent1{

static int a = 4;

static {

System.out.println("Parent.enclosing\_method()");

}

static void doSomething() {

System.out.println("Parent.doSomething()");

}

}

class Child1 extends Parent1{

static {

System.out.println("Child.enclosing\_method()");

}

}

public class Test4 {

/\* 输出次序：子类不会被初始化

Parent.enclosing\_method()

4

Parent.doSomething()

\*/

public static void main(String[] args) {

System.out.println(Child1.a);

Child1.doSomething();

}

}

Binary names：类的全称，简单的说就是包名加类名，如果是内部类加内部类名称，如：java.net.URLClassLoader$3$1。其中$3$1指第三个匿名内部类中的第一个匿名内部类

调用ClassLoader类的loadClass方法加载一个类，并不是对类的主动使用，不会导致类的初始化。即指执行到加载，之后的连接与初始化都没有发生。如下例子：

class CL{

static {

System.out.println("CL.enclosing\_method()");

}

}

public class Test5 {

/\* 输出：

---------

CL.enclosing\_method()

\*/

public static void main(String[] args) throws ClassNotFoundException {

ClassLoader loader = ClassLoader.getSystemClassLoader();

Class<?> clazz = loader.loadClass("com.classloader.CL");

System.out.println("---------");

clazz = Class.forName("com.classloader.CL");

}

}

**类加载器**

Java虚拟机自带了以下几种加载器：

根(Bootstrap)加载器;扩展(Extension)类加载器；系统(System)类加载器

系统类加载器是用户自定义的类加载器的默认父加载器。所有用户自定义的类加载器应该继承ClassLoader类。

在父亲委托机制中，各个加载器按照父子关系形成类树形结构，除了根类加载器以为，其余的类加载器都有且只有一个父加载器。

若有一个类加载器能成功加载Sample类，那么这个类加载器被称为定义类加载器，所有能成功返回Class对象的引用的类加载器（包括定义类加载器）都被称为初始类加载器

需要指出的是，加载器之间的父子关系实际上指的是加载器对象之间的包装关系，而不是类之间的继承关系。一对父子加载器可能是同一个加载器类的两个实例，也可能不是。如：

ClassLoader loader1 = new MyClassLoader();

//参数loader1将作为loader2的父加载器

ClassLoader loader2 = new MyClassLoader(loader1);

当生成一个自定义的类加载器实例时，如果没有指定它的父加载器，那么系统类加载器就将成为该类加载器的父加载器。

每个类加载器都有自己的命名空间，命名空间由该加载器及所有父加载器所加载的类组成。在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类；在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类。

同一个类加载器加载的属于相同包的类组成类运行时包。决定两个类是不是属于同一个运行时报，不仅要看它们的包名是否相同，还要看定义类加载器是否相同。只有属于同一运行时包的类才能互相访问包可见（即默认访问级别）的类和类成员。这样的限制能避免用户自定义的类冒充核心类库的类，去访问核心类库的包可见成员。

同一个命名空间内的类是相互可见的。子加载器的命名空间包含所有父加载器的命名空间。因此由子加载器加载的类能看见父加载器加载的类。由父加载器加载的类不能看见子加载器加载的类。如果两个加载器之间没有直接或间接的父子关系，那么它们各自加载的类相互不可见。

可以通过反射解决这个界限，让两个不同命名空间下的类互相访问。而且也只能通过反射来做。

**类的卸载**

当Sample类被加载，连接和初始化后，它的生命周期就开始类。当代码Sample类的Class对象不再不引用，即不可触及时，Class对象就会被结束生命周期，Sample类在方法区内的数据也会被卸载，从而结束Sample类的生命周期。由此可见，一个类何时结束生命周期，取决于代表它的Class对象何时结束生命周期。

由Java虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。Java虚拟机自带类加载器包括根类加载器，扩展类加载器和系统类加载器。Java虚拟机本身会始终引用这些类加载器，而这些类加载器则会始终引用它们所加载的类的Class对象，因此这些Class对象始终是可触及的。由用户自定义的类加载器所加载的类是可以被卸载的。

# HotSpot 虚拟机对象探秘

## 对象的创建

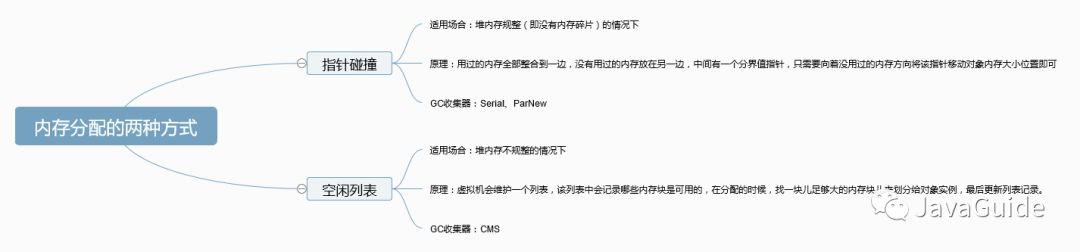


①类加载检查： 虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

②分配内存： 在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存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。分配方式有 “指针碰撞” 和 “空闲列表” 两种，选择那种分配方式由 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。

内存分配的两种方式：（补充内容，需要掌握）

选择以上两种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是"标记-清除"，还是"标记-整理"（也称作"标记-压缩"），值得注意的是，复制算法内存也是规整的



内存分配并发问题（补充内容，需要掌握）

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

•CAS+失败重试： CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。

•TLAB： 为每一个线程预先在Eden区分配一块儿内存，JVM在给线程中的对象分配内存时，首先在TLAB分配，当对象大于TLAB中的剩余内存或TLAB的内存已用尽时，再采用上述的CAS进行内存分配

③初始化零值： 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（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

④设置对象头： 初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希吗、对象的 GC 分代年龄等信息。 这些信息存放在对象头中。 另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

⑤执行 init 方法： 在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚开始，<init> 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 <init> 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

## 对象的内存布局

在 Hotspot 虚拟机中，对象在内存中的布局可以分为3块区域：对象头、实例数据和对齐填充。

Hotspot虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的自身运行时数据（哈希码、GC分代年龄、锁状态标志等等），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那个类的实例。

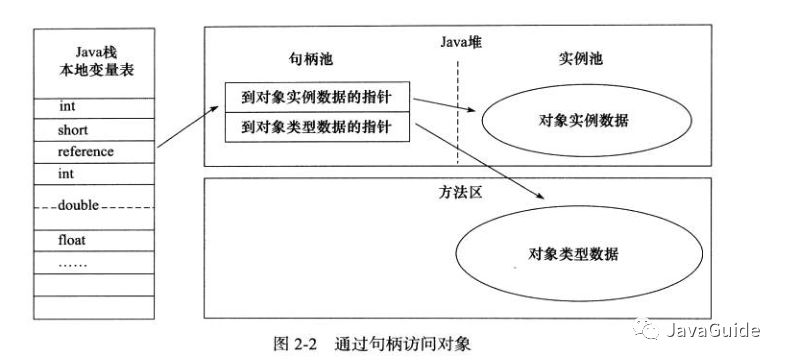
实例数据部分是对象真正存储的有效信息，也是在程序中所定义的各种类型的字段内容。

对齐填充部分不是必然存在的，也没有什么特别的含义，仅仅起占位作用。 因为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，换句话说就是对象的大小必须是8字节的整数倍。而对象头部分正好是8字节的倍数（1倍或2倍），因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

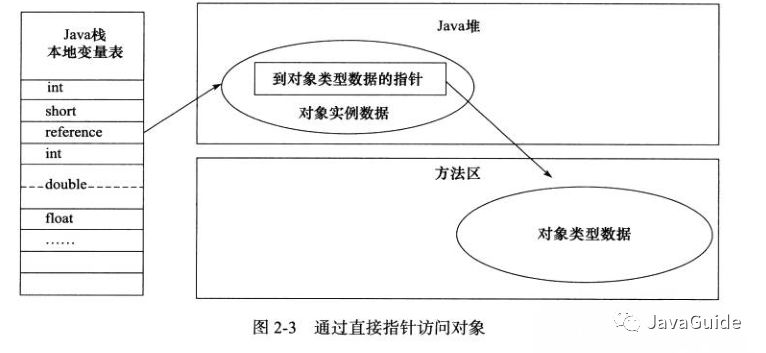
对象的访问定位

建立对象就是为了使用对象，我们的Java程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式有虚拟机实现而定，目前主流的访问方式有①使用句柄和②直接指针两种：

•句柄： 如果使用句柄的话，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与类型数据各自的具体地址信息；



直接指针： 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的地址。



这两种对象访问方式各有优势。使用句柄来访问的最大好处是 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最大的好处就是速度快，它节省了一次指针定位的时间开销。

# String 类和常量池

**String 对象的两种创建方式**：

String str1 = "abcd";

String str2 = new String("abcd");

这两种不同的创建方法是有差别的，第一种方式是在常量池中拿对象，第二种方式是直接在堆内存空间创建一个新的对象。只要使用new方法，便需要创建新的对象。

**String 类型的常量池比较特殊。它的主要使用方法有两种：**

•直接使用双引号声明出来的 String 对象会直接存储在常量池中。

•如果不是用双引号声明的 String 对象，可以使用 String 提供的 intern 方法。String.intern() 是一个 Native 方法，它的作用是：如果运行时常量池中已经包含一个等于此 String 对象内容的字符串，则返回常量池中该字符串的引用；如果没有，则在常量池中创建与此 String 内容相同的字符串，并返回常量池中创建的字符串的引用。

String s1 = new String("计算机");

String s2 = s1.intern();

String s3 = "计算机";

System.out.println(s2);//计算机

System.out.println(s1 == s2);//false，因为一个是堆内存中的String对象一个是常量池中的String对象，

System.out.println(s3 == s2);//true，因为两个都是常量池中的String对象

**String 字符串拼接**

String str1 = "str";

String str2 = "ing";

String str3 = "str" + "ing";//常量池中的对象

String str4 = str1 + str2; //在堆上创建的新的对象

String str5 = "string";//常量池中的对象

System.out.println(str3 == str4);//false

System.out.println(str3 == str5);//true

System.out.println(str4 == str5);//false

**String s1 = new String("abc");这句话创建了几个对象？**

创建了两个对象:先有字符串"abc"放入常量池，然后 new 了一份字符串"abc"放入Java堆(字符串常量"abc"在编译期就已经确定放入常量池，而 Java 堆上的"abc"是在运行期初始化阶段才确定)，然后 Java 栈的 str1 指向Java堆上的"abc"。

**8种基本类型的包装类和常量池**

•Java 基本类型的包装类的大部分都实现了常量池技术，即Byte,Short,Integer,Long,Character,Boolean；这5种包装类默认创建了数值[-128，127]的相应类型的缓存数据，但是超出此范围仍然会去创建新的对象。

•两种浮点数类型的包装类 Float,Double 并没有实现常量池技术。

Integer i1 = 33;

Integer i2 = 33;

System.out.println(i1 == i2);// 输出true

Integer i11 = 333;

Integer i22 = 333;

System.out.println(i11 == i22);// 输出false

Double i3 = 1.2;

Double i4 = 1.2;

System.out.println(i3 == i4);// 输出false

应用场景：

•Integer i1=40；Java 在编译的时候会直接将代码封装成Integer i1=Integer.valueOf(40);，从而使用常量池中的对象。

•Integer i1 = new Integer(40);这种情况下会创建新的对象。

Integer i1 = 40;

Integer i2 = new Integer(40);

System.out.println(i1==i2);//输出false

Integer比较更丰富的一个例子:

Integer i1 = 40;

Integer i2 = 40;

Integer i3 = 0;

Integer i4 = new Integer(40);

Integer i5 = new Integer(40);

Integer i6 = new Integer(0);

System.out.println("i1=i2 " + (i1 == i2));

System.out.println("i1=i2+i3 " + (i1 == i2 + i3));

System.out.println("i1=i4 " + (i1 == i4));

System.out.println("i4=i5 " + (i4 == i5));

System.out.println("i4=i5+i6 " + (i4 == i5 + i6));

System.out.println("40=i5+i6 " + (40 == i5 + i6));

结果：

i1=i2 true

i1=i2+i3 true

i1=i4 false

i4=i5 false

i4=i5+i6 true

40=i5+i6 true

语句i4 == i5 + i6，因为+这个操作符不适用于Integer对象，首先i5和i6进行自动拆箱操作，进行数值相加，即i4 == 40。然后Integer对象无法与数值进行直接比较，所以i4自动拆箱转为int值40，最终这条语句转为40 == 40进行数值比较。

# 垃圾回收器

## G1

G1垃圾回收器是为多处理器和大内存的服务器而设计的，它根据运行JVM过程中构建的停顿预测模型（Pause Prediction Model）计算出来的历史数据来预测本次收集需要选择的Region数量，然后尽可能（不是绝对）满足GC的停顿时间，G1期望能让JVM的GC成为一件简单的事情。G1旨在延迟性和吞吐量之间取得最佳的平衡，它尝试解决有如下问题的Java应用：

堆大小能达到几十G甚至更大，超过50%的堆空间都是存活的对象；

对象分配和晋升的速度随着时间的推移有很大的影响；

堆上严重的碎片化问题；

可预测的停顿时间，避免长时间的停顿。

开启G1

在JDK9之前，JDK7和JDK8默认都是ParallelGC垃圾回收。到了JDK9，G1才是默认的垃圾回收器。所以如果JDK7或者JDK8需要使用G1的话，需要通过参数（-XX:+UseG1GC）显示执行垃圾回收器。

# ClassLoader

## ClassLoader 做什么的？

顾名思义，它是用来加载 Class 的。它负责将 Class 的字节码形式转换成内存形式的 Class 对象。字节码的本质就是一个字节数组 []byte，它有特定的复杂的内部格式。

每个 Class 对象的内部都有一个 classLoader 字段来标识自己是由哪个 ClassLoader 加载的。

## 延迟加载

JVM 运行并不是一次性加载所需要的全部类的，它是按需加载，也就是延迟加载。程序在运行的过程中会逐渐遇到很多不认识的新类，这时候就会调用 ClassLoader 来加载这些类。加载完成后就会将 Class 对象存在 ClassLoader 里面，下次就不需要重新加载了。

比如你在调用某个类的静态方法时，首先这个类肯定是需要被加载的，但是并不会触及这个类的实例字段，那么实例字段的类别 Class 就可以暂时不必去加载，但是它可能会加载静态字段相关的类别，因为静态方法会访问静态字段。而实例字段的类别需要等到你实例化对象的时候才可能会加载。

## 各司其职

JVM 运行实例中会存在多个 ClassLoader，不同的 ClassLoader 会从不同的地方加载字节码文件。

JVM 中内置了三个重要的 ClassLoader，分别是 BootstrapClassLoader、ExtensionClassLoader 和 AppClassLoader。

BootstrapClassLoader 负责加载 JVM 运行时核心类，这些类位于 JAVA\_HOME/lib/rt.jar 文件中，比如 java.util.\*、java.io.\*、java.nio.\*、java.lang.\* 等等。

ExtensionClassLoader 负责加载 JVM 扩展类，比如 swing 系列、内置的 js 引擎、xml 解析器 等等，这些库名通常以 javax 开头，它们的 jar 包位于 JAVA\_HOME/lib/ext/\*.jar 中。

AppClassLoader 才是直接面向我们用户的加载器，它会加载 Classpath 环境变量里定义的路径中的 jar 包和目录。

那些位于网络上静态文件服务器提供的 jar 包和 class文件，jdk 内置了一个 URLClassLoader，ExtensionClassLoader 和 AppClassLoader 都是 URLClassLoader 的子类。

## ClassLoader 传递性

程序在运行过程中，遇到了一个未知的类，它会选择哪个 ClassLoader 来加载它呢？虚拟机的策略是使用调用者 Class 对象的 ClassLoader 来加载当前未知的类。何为调用者 Class 对象？就是在遇到这个未知的类时，虚拟机肯定正在运行一个方法调用（静态方法或者实例方法），这个方法挂在哪个类上面，那这个类就是调用者 Class 对象。前面我们提到每个 Class 对象里面都有一个 classLoader 属性记录了当前的类是由谁来加载的。

因为 ClassLoader 的传递性，所有延迟加载的类都会由初始调用 main 方法的这个 ClassLoader 全全负责，它就是 AppClassLoader。

## 双亲委派

前面我们提到 AppClassLoader 只负责加载 Classpath 下面的类库，如果遇到没有加载的系统类库怎么办，AppClassLoader 必须将系统类库的加载工作交给 BootstrapClassLoader 和 ExtensionClassLoader 来做，这就是我们常说的「双亲委派」。

## Class.forName

当我们在使用 jdbc 驱动时，经常会使用 Class.forName 方法来动态加载驱动类。

其原理是 mysql 驱动的 Driver 类里有一个静态代码块，它会在 Driver 类被加载的时候执行。这个静态代码块会将 mysql 驱动实例注册到全局的 jdbc 驱动管理器里。

class Driver {

static {

try {

java.sql.DriverManager.registerDriver(new Driver());

} catch (SQLException E) {

throw new RuntimeException("Can't register driver!");

}

}

...

}

forName 方法同样也是使用调用者 Class 对象的 ClassLoader 来加载目标类。不过 forName 还提供了多参数版本，可以指定使用哪个 ClassLoader 来加载

通过这种形式的 forName 方法可以突破内置加载器的限制，通过使用自定类加载器允许我们自由加载其它任意来源的类库。根据 ClassLoader 的传递性，目标类库传递引用到的其它类库也将会使用自定义加载器加载。

## 自定义加载器

ClassLoader 里面有三个重要的方法 loadClass()、findClass() 和 defineClass()。

loadClass() 方法是加载目标类的入口，它首先会查找当前 ClassLoader 以及它的双亲里面是否已经加载了目标类，如果没有找到就会让双亲尝试加载，如果双亲都加载不了，就会调用 findClass() 让自定义加载器自己来加载目标类。ClassLoader 的 findClass() 方法是需要子类来覆盖的，不同的加载器将使用不同的逻辑来获取目标类的字节码。拿到这个字节码之后再调用 defineClass() 方法将字节码转换成 Class 对象。

自定义类加载器不易破坏双亲委派规则，不要轻易覆盖 loadClass 方法。否则可能会导致自定义加载器无法加载内置的核心类库。在使用自定义加载器时，要明确好它的父加载器是谁，将父加载器通过子类的构造器传入。如果父类加载器是 null，那就表示父加载器是「根加载器」。

## Class.forName vs ClassLoader.loadClass

这两个方法都可以用来加载目标类，它们之间有一个小小的区别，那就是 Class.forName() 方法可以获取原生类型的 Class，而 ClassLoader.loadClass() 则会报错。

URLClassLoader v1 = new URLClassLoader(new URL[]{new URL(v2dir)});

Class<?> depv1Class = v1.loadClass("Dep");

Object depv1 = depv1Class.getConstructor().newInstance();

depv1Class.getMethod("print").invoke(depv1);

即使是同样的字节码用不同的 ClassLoader 加载出来的类都不能算同一个类