# 类

装载，连接（验证，准备，解析（可选）），初始化。

## 装载

通过类的全限定名来获取其定义的二进制字节流。

将这个字节流所代表的静态存储结构转化为方法区运行时的数据结构。

在堆中创造一个对象，用于访问方法区的数据的访问入口。

通过双亲委派模型加载类到方法区

## 验证

文件格式的验证：验证.class字节流是否规范。主要对魔数、主版本号、常量池等等的校验（都是.class文件里面包含的数据信息）。

元数据验证：进行语义分析，比如说验证这个类是不是有父类，类中的字段方法是不是和父类冲突等等。

字节码验证：主要对类的方法做出分析，保证类的方法在运行时不会损害jvm。

符号引用验证：它是验证的最后一个阶段，发生在虚拟机将符号引用转化为直接引用的时候。主要是对类自身以外的信息进行校验。目的是确保解析动作能够完成。

## 准备

为该类型分配需要的内存，并分配默认初始值，此时并未初始为真正的初始值（final会直接分配相应的值）。

static会分配内存在方法区；但是实例变量不会，因为随着对象的实例化一块分配到堆中。

## 解析（可选）

可选：当程序真正的用到某个符号的引用时才去解析(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。)。

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类符号引用进行。

## 初始化

在这个阶段，java程序代码才开始真正执行。给类的变量赋以适当的初始值。

# 类的其他知识点

## 类的加载器（ClassLoader）

从Java虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用C++语言实现（只限于HotSpot），是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由Java语言实现，独立于虚拟机外部，并且全都继承自抽象类java.lang.ClassLoader。

启动类加载器-BootStarp ClassLoad（加载内置类）：这个类加载器负责将存放在＜JAVA\_HOME＞\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录中也不会被加载）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

扩展类加载器-Extension ClassLoad（加载继承类或者实现类）：这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载＜JAVA\_HOME＞\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器-Applocation ClassLoader（加载自定义的类）：这个类加载器由sun.misc.Launcher$App-ClassLoader实现。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，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。

自定义加载器

## 双亲委派原则

当一个类加载器收到类加载任务，会先交给其父类加载器去完成，因此最终加载任务都会传递到顶层的启动类加载器，只有当父类加载器无法完成加载任务时，才会尝试执行加载任务。

好处：

可以避免重复加载，父类已经加载了，子类就不需要再次加载

很好的解决了各个类加载器的基础类的统一问题，如果不使用该种方式，那么用户可以随意定义类加载器来加载核心api，会带来相关隐患。

## 虚方法表

如果继承层级比较深，调用的方法在很上层的父类，效率会比较低，每次调用都需要多次查找，此时多数系统会采用虚方法表来优化这种调用（这种情况在加载的时候就会创建一个表，记录该类对象所有动态绑定的方法与地址，调用时直接查该表就行了）

# GC

<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

## java内存划分：

线程共享的：堆区，方法区。

***1.堆区：***存放对象实例和对象的实例变量（Mark Word等）。1.8之后删除永久代。

***2.方法区：***用于存储虚拟机加载的：静态变量+常量+类信息+运行时常量池 （类信息：类的版本、字段、方法、接口、构造函数等描述信息 ），默认最小值为16MB，最大值为64MB，可以通过-XX:PermSize 和 -XX:MaxPermSize 参数限制方法区的大小。

线程私有的：虚拟机栈，本地方法栈，程序计数器。

***1.虚拟机栈：***jvm执行的java方法。

***2.本地方法栈：***jvm执行的native方法。HotSpot将虚拟机栈和本地方法栈合并。

***3.程序计数器：***每个线程都有一个程序计算器，就是一个指针，指向方法区中的方法字节码（下一个将要执行的指令代码），由执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不记。由于java多线程是通过线程轮流切换并分配执行时间来实现的，任何时刻只会执行一条线程中的指令。因此为了切换线程后依旧能恢复到正确的位置，需要一个独立的计数器。

## 执行对象

自动释放不再被引用的对象所占的堆内存，整理碎片空间。

可达性分析：从GC Roots开始向下搜索，没有任何引用链相连时，则证明此对象是不可用的。不可达对象，需要被回收。

GC Roots包括：

a．方法区中类静态属性实体引用的对象。

b．方法区中常量引用的对象。

c．虚拟机栈中引用的对象。

d．本地方法栈中JNI（本地接口）引用的对象。

## 优缺点

优点：可以提高生产效率（减少释放空间的编程时间）和增加程序的健壮性（不再因为错误的释放空间导致程序崩溃）。

缺点：影响程序的性能，因为要动态的监视和释放。在明确的内存不再使用内存比起来，这个活动占用了更多的cpu时间。程序员缺乏安排cpu对释放对象的控制。

## 执行时机

程序调用通知System.gc()。

Minor GC：

当Eden区满后。

Major GC：

调用System.GC；老年代空间不足；方法区空间不足；调用Minor GC后，/移至年老区的对象大小大于可用空间大小；

## 流程

绝大多数刚创建的对象被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；

Eden区满，执行Minor GC，清理消亡对象，将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；

此后，每次Eden区满了，就执行一次Minor GC，并将剩余的对象都添加到Survivor0；

当Survivor0也满的时候，将其中仍然活着的对象直接复制到Survivor1，以后Eden区执行Minor GC后，就将剩余的对象添加Survivor1（此时，Survivor0是空白的）。

当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。

## Minor GC

使用停止-复制清理法，用于清理年轻代。

JVM会安全的暂停所有正在执行的线程（Stop The World），来回收内存空间，在这个时间内，所有除了回收垃圾的线程外，其他有关JAVA的程序，代码都会静止，反映到系统上，就会出现系统响应大幅度变慢，卡机等状态。

指针碰撞（bump-the-pointer）用于追踪最后创建的一个对象，在创建对象的时候，只需要检查最后一个对象后边是否有足够的内存即可。

TLAB（thread-local-allocation-buffers）对于多线程而言，将eden区分为若干段，每个线程使用独立的一段。

## Major GC

使用标记整理清理法，用于清理年老代。将存活对象全部移动至一端。

## Full GC

清理整个堆空间—包括年轻代和老年代。

## 空配分配担保

在发生minor gc之前，虚拟机会检测 : 老年代最大可用的连续空间>新生代all对象总空间？

1、满足，minor gc是安全的，可以进行minor gc。

2、不满足，虚拟机查看HandlePromotionFailure参数：

（1）为true，允许担保失败，会继续检测老年代最大可用的连续空间>历次晋升到老年代对象的平均大小。若大于，将尝试进行一次minor gc，若失败（应该时gc之后再上述的检测），则重新进行一次full gc。

（2）为false，则不允许冒险，要进行full gc（对老年代进行gc）。

## 垃圾回收算法

为什么只有ParNew能与CMS收集器配合

CMS是HotSpot在JDK1.5推出的第一款真正意义上的并发（Concurrent）收集器，第一次实现了让垃圾收集线程与用户线程（基本上）同时工作；

CMS作为老年代收集器，但却无法与JDK1.4已经存在的新生代收集器Parallel Scavenge配合工作；

因为Parallel Scavenge（以及G1）都没有使用传统的GC收集器代码框架，而另外独立实现；而其余几种收集器则共用了部分的框架代码；

————————————————

版权声明：本文为CSDN博主「谙忆」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/qq\_26525215/article/details/84294481

## GC垃圾回收器

### Serial（串行-新生代-复制算法）

单行程执行，有单线程带来的简单高效，减少了线程的切换，也有单线程带来的STW时间更长。新生代的收集器。

### Serial Old（Serial老年代-标记整理压缩）

CMS收集器的后备方案，

### ParNew（Serial多线程版本）

除了多线程外，其余的特点，行为和Serial一样。

### Parallel Scavenge（如何高效的使用CPU-新生代-复制算法）

提供很多参数供用户找到合适的停顿时间和最大吞吐量。（控制最大的停顿时间，控制垃圾收集占时间百分比，自适应调节）

### Parallel Old（Parallel Scavenge的老年代版本-标记整理）

用来代替老年代的Serial Old收集器，注重吞吐量以及CPU资源敏感的场景，就有了Parallel Scavenge（新生代）加Parallel Old（老年代）收集器的"给力"应用组合；

### CMS（老年代-并发标记清除）

注重STW的时间，即注重用户感受。

四个步骤：

初始标记：STW，仅仅标记GC Roots能直接关联到的对象，速度很快；

并发标记：不会STW，进行GC Roots Tracing的过程，会记录发生引用更新的地方；

重新标记：STW，多线程对引用更改进行修正再扫描，且停顿时间比初始标记稍长，但远比并发标记短；

并发清除：不会STW，同时GC线程回收所有的垃圾对象；

缺点：占用CPU，解决收集线程和用户线程交替执行，1.6不再提倡；更大的空间，清理需要预留空间（新产生的垃圾存放），解决设置老年代预留空间大小；产生大量碎片，导致是新生代进入老年代失败而引发FULL GC，解决进行碎片整理参数和进行多少次不压缩FULL GC之后对空间进行压缩整理。

增加了新生代垃圾收集时应用暂停的时间、降低了吞吐量而且需要占用更大的堆空间；

### G1

四个步骤

初始标记：STW，标记GC Roots能直接关联到的对象，且修改TAMS（Next Top at Mark Start）,让下一阶段并发运行时，用户程序能在正确可用的Region中创建新对象；

并发标记：从GC Roots开始进行可达性分析，找出存活对象，耗时长，可与用户线程并发执行并不能保证可以标记出所有的存活对象；

最终标记：STW，修正并发标记阶段因用户线程继续运行而导致标记发生变化的那部分对象的标记记录。

上一阶段对象的变化记录在线程的Remembered Set Log；

这里把Remembered Set Log合并到Remembered Set中；

G1采用多线程并行执行来提升效率；且采用了比CMS更快的初始快照算法:Snapshot-At-The-Beginning (SATB)。

筛选回收（Live Data Counting and Evacuation）

首先排序各个Region的回收价值和成本；

然后根据用户期望的GC停顿时间来制定回收计划；

最后按计划回收一些价值高的Region中垃圾对象；

回收时采用"复制"算法，从一个或多个Region复制存活对象到堆上的另一个空的Region，并且在此过程中压缩和释放内存；

可以并发进行，降低停顿时间，并增加吞吐量；

## OutOfMemoryError

**堆：**

堆达到最大的容量限制，需要分析是内存泄漏还

是内存溢出。（GC Roots的可达性分析）

**方法区：**

分配空间不足，常量池分配不足等就会

java.lang.OutOfMemoryError:PermGen space

**虚拟机栈和本地方法栈：**

如果线程请求的栈深度大于虚拟机所允许的深度，将

抛出java.lang.StackOverflowError。

不断创建线程的方法，在这种情况下，为每个线程分配的内存越大，就会越容易产生OOM异常。

**程序计数器：**唯一不会引发OOM问题。

***OOM:GC overhead limit exceeded***

如果系统大量的时间都在GC（98%）而回收的效果不明显（2% heap空间），就会抛出这个异常。

## 各项参数

<https://www.iteye.com/blog/leichenlei-2097797>

|  |  |
| --- | --- |
| -Xms | 堆初始值（建议：操作系统可用的1/64） |
| -Xmx | 堆最大值（建议：操作系统的1/4） |
| -XX:PermSize | 方法区初始值 |
| -XX:MaxPermSize | 方法区最大值 |
| -XX:newSize | 新生代初始值 |
| -XX:MaxnewSize | 新生代最大值 |
| -Xmn | 同时设置新生代初始值和最大值 |
| -XX:PretenureSizeThreshold | 直接就进入老年代的对象大小 |
| -XX:MaxTenuringThreshold | 新生代进入老年代的“岁数” |
| -XX:SurvivorRatio | eden比率 |
| -XX:+PrintGCDetails | 输出GC时信息 |

# 进程、线程

## 进程解释：

一个进程就是一个正在运行的程序，包括代码、数据、cpu寄存器的值

# https://img-blog.csdnimg.cn/20190111092408622.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2xpdWR1bl9jb29s,size_16,color_FFFFFF,t_70锁

## Mark Word

## synchronized

偏向锁：biased\_lock为0则CAS设置当前线程id，不为0则升级锁。默认关闭，因为存在并发的概率是很高的，用户态和内核态的切换很消耗性能。

轻量级锁：升级为轻量级锁时，会先把锁的对象头MarkWord复制一份到线程的栈帧中，建立一个名为锁记录空间（Lock Record），用于存储当前Mark Word的拷贝。

虚拟机使用cas操作尝试将对象的Mark Word指向Lock Record的指针，并将Lock record里的owner指针指对象的Mark Word。

如果cas操作成功，则该线程拥有了对象的轻量级锁。第二个线程cas自选锁等待锁线程释放锁。

如果多个线程竞争锁，轻量级锁要膨胀为重量级锁，Mark Word中存储的就是指向重量级锁（互斥量）的指针。其他等待线程进入阻塞状态。

重量级锁：

代码块：monitorenter/monitorexit，获取monitor

方法：ACC\_SYNCRHONIZED，获取monitor

自旋锁：线程循环等待，1.6之后引入自适应自旋锁。根据前一次在同一个锁上自 旋的时间以及锁的拥有者的状态来决定

消除锁：一个变量没有在其他类被引用，消除自身得锁。

锁粗化：多个锁连续获取释放，则只形成一个更大得锁。

# utp，tcp和websocket

1：UDP是面向报文的，发送方的UDP对应用层交下来的报文，不合并，不拆分，只是在其上面加上首部后就交给了下面的网络层，也就是说无论应用层交给UDP多长的报文，它制统统发送，一次发送一个。而对接收方，接到后直接去除首部，交给上面的应用层就完成任务了。因此，它需要应用层控制报文的大小

2：TCP是面向字节流的，它把上面应用层交下来的数据看成无结构的字节流来发送，知可以想象成流水形式的，发送方TCP会将数据放入“蓄水池”（缓存区），等到可以发送的时候就发送，不能发送就等着，TCP会根据当前网络的拥塞状态来确定每个报文段的道大小。

3：每一条TCP连接只能是点到点的，UDP支持一对一，一对多和多对多的交互通信。

4：TCP是流模式，UDP是数据报模式。

# mysql

## 二叉树

## 红黑树

解决了二叉树搜索出现O(n)的时间复杂度。

插入或删除数据导致不满足平衡二叉树不平衡时，平衡二叉树会进行调整树上的节点来保持平衡。

## B-Tree

解决了红黑树数据过多导致极高的高度，因为IO的存在使查找效率降低。

多路，非二叉树**；**

每个节点既保存索引，又保存数据**；**

搜索时相当于二分查找**；**

MongoDB所有节点都有Data域，只要找到指定索引就可以进行访问，无疑单次查询平均快于Mysql。

## B+Tree

解决：非叶子节点只存储索引，这样查找索引导致的IO操作更少，效率更高。

多路，非二叉树。

只有叶子节点保存数据

搜索时相当于二分查找

增加了相邻接点的双向指针。

Mysql作为一个关系型数据库，数据的关联性是非常强的，区间访问是常见的一种情况，B+树由于数据全部存储在叶子节点，并且通过指针串在一起，这样就很容易的进行区间遍历甚至全部遍历。

## B\*Tree

# hashmap的扩容和红黑树

# Spring代理

jdk动态代理：利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

cglib动态代理，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP，如果目标对象没有实现了接口，必须采用CGLIB库。

动态代理只能对实现了接口的类生成代理，而不能针对类，CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法，因为是继承，所以该类或方法最好不要声明成final。

# 三次握手与四次挥手

## 三次握手（建立连接）

center->server：SYN=1，seq=x

server->center：SYN=1，ACK=1，seq=y，ack=x+1

center->server：ACK=1，seq=x+1，ack=y+1

三次原因：为了确保双方的接受和发送正常，只需要三次即可确认。

## 四次挥手（断开连接）

center->server：FIN=1,seq=u

server->center：ACK=1,seq=v,ack=u+1

server->center：FIN=1,seq=w,ack=u+1

center->server：ACK=1,seq=u+1,ack=w+1

四次原因：服务器收到关闭的请求不能立马关闭连接，所以需要告诉客户端，我收到请求了，等我发送完请求，我会再告诉你我关闭了。

为什么客户端最后需要等待2MSL（Windows :120s,Ubuntu, CentOs:60s,Unix:30s）：第四次挥手可能会丢失，服务器超时未接受到第四次挥手则再发送第三次挥手信息，