class文件

概述

八个字节为单位的二进制流 其中只有两种类型 无符号数和表

常量池从1开始，第0项给匿名内部类或者object类的父类索引指向

java和jvm体系结构

jvm的架构模型

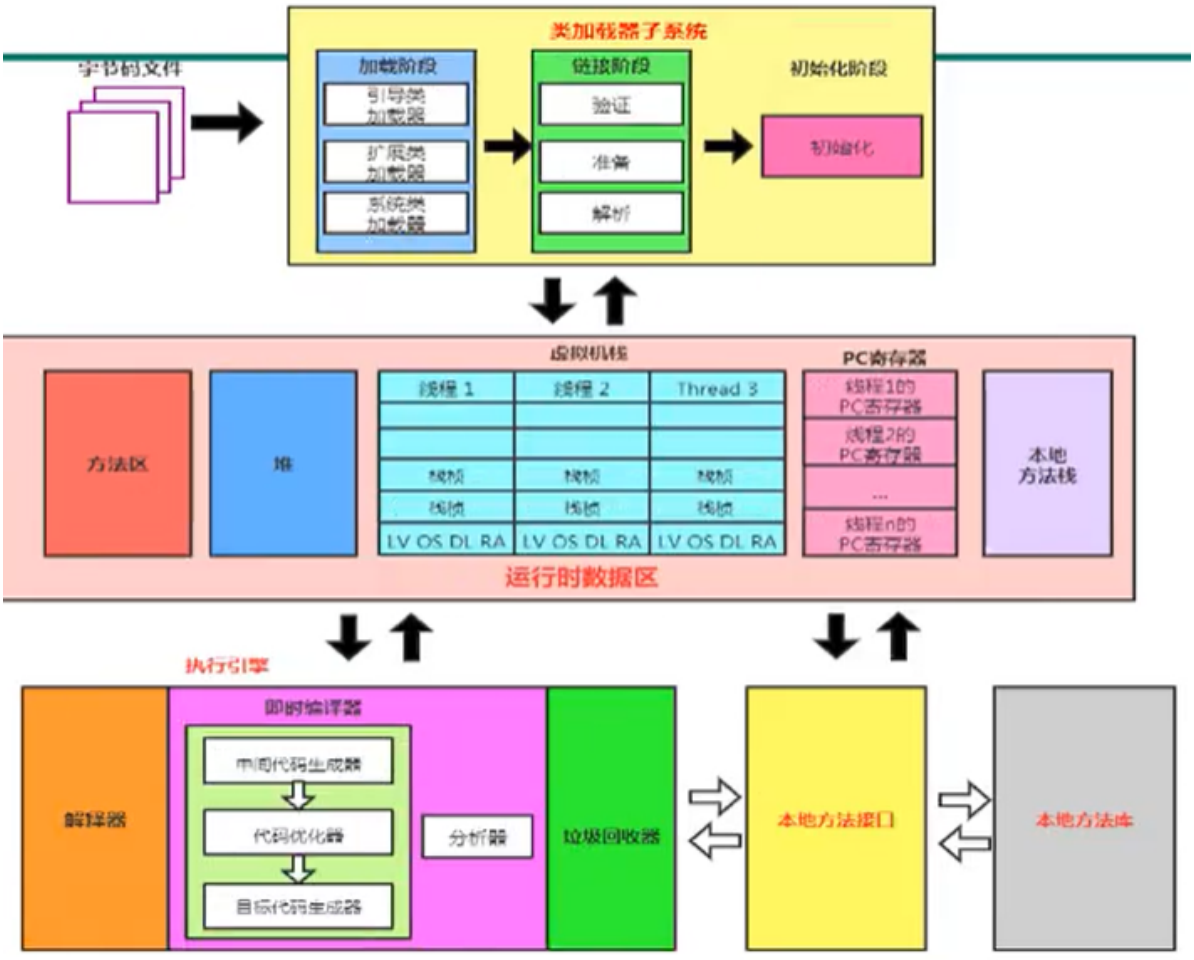
基于栈

简单，指令集少，不需要硬件支持，可移植性好

基于寄存器

相反

内存结构



.class字节码文件

类加载子系统

运行时数据区(pc寄存器 栈 本地方法栈 **方法区 堆** 后两个是线程公有的)

执行引擎

本地方法接口-----库

类加载子系统

类加载器分类

引导类加载器 **Bootstrap ClassLoader**

c++实现，不需要继承java.lang.ClassLoader,加载核心类库,提供jvm自身需要的类

扩展类加载器 **Extension ClassLoader**

继承ClassLoader类

系统类加载器 **AppClassLoader**

默认的类加载器

双亲委派机制及作用如何打破

类加载时，先让父类加载器加载，然后子类再加载

防止用户自定义核心类库，无法保证类的唯一性，即沙箱安全机制

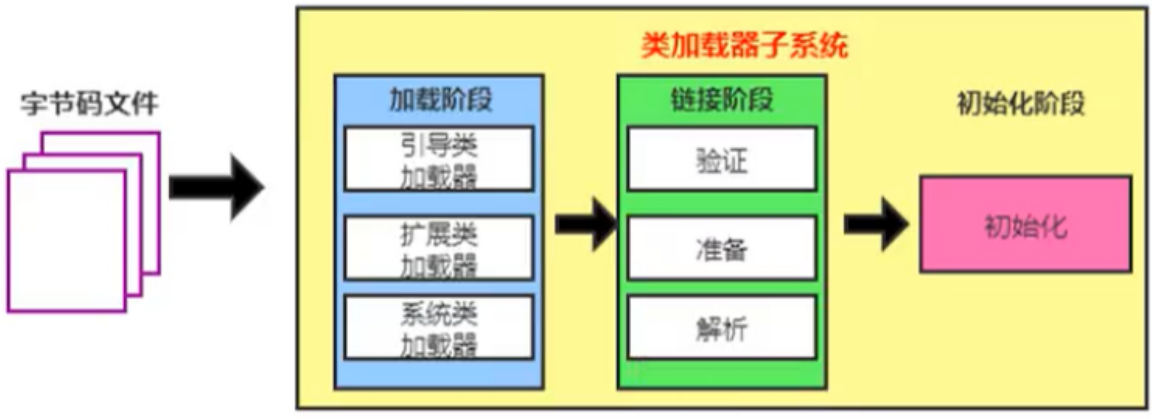
如何打破

自定义类加载器，继承ClassLoader类，重写loadClass方法和findClass方法。

举例

tomcat为了实现多个应用程序部署依赖同一三方库不同版本

作用



将class文件加载到内存?

类信息存放在方法区

加载阶段

通过类的全类名获取二进制字节流

获取类**类信息、常量、静态变量、即时编译器编译后的代码等数据**存放到方法区

在内存中生成一个代表这个类的**java.lang.Class**对象，作为方法区这个类的**各种数据的访问入口**

综述：加载二进制流获取类信息到方法区，在内存中生成对象，作为方法区调用该类的入口

链接阶段

验证（Verify）：安全性验证

准备（Prepare）： 为类变量分配内存**赋0值**，对于static final 常量不会分配，而是在编译时分配

解析（Resolve）： 将常量池内的**符号引用转换为直接引用**的过程。这里只会将可以确定的如静态方法，静态变量进行转换

初始化阶段

初始化阶段就是的过程。

静态代码块和静态变量初始化

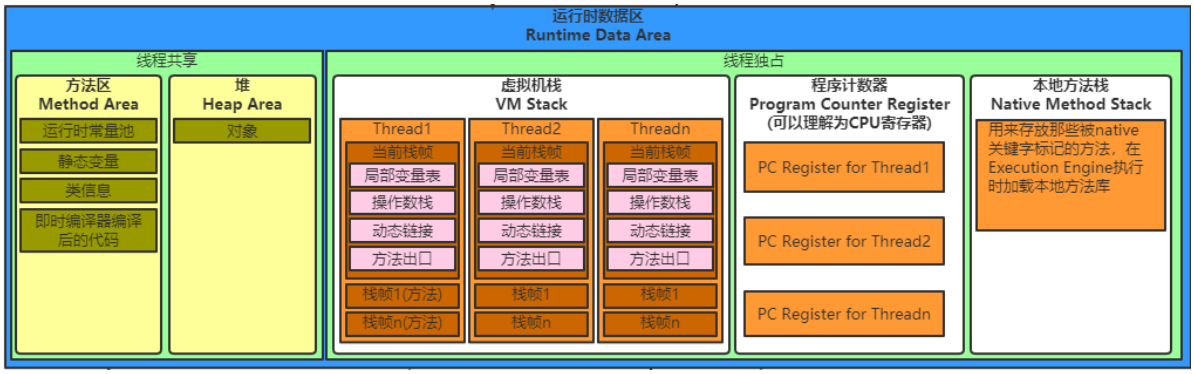
什么时候调用初始化

1)遇到new、getstatic、 putstatic或invokestatic即new 对象 static 修饰的属性（非编译期间存入常量池如" String"） static方法 2)使用java.lang.reflect包的方法对类型进行反射调用的时候，如果类型没有进行过初始化，则需要先触发其初始化。 3)当初始化类的时候，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。 4)当虚拟机启动时,加载主类 5反射

三个常量池

- **Class文件常量池。**class文件是一组以字节为单位的二进制数据流，在java代码的编译期间，我们编写的java文件就被编译为.class文件格式的二进制数据存放在磁盘中，其中就包括class文件常量池。
- **运行时常量池**：运行时常量池相对于class常量池一大特征就是具有动态性，java规范并不要求常量只能在运行时才产生，也就是说运行时常量池的内容并不全部来自class常量池，在运行时可以通过代码生成常量并将其放入运行时常量池中，这种特性被用的最多的就是String.intern()。
- **全局字符串常量池**：字符串常量池是JVM所维护的一个字符串实例的引用表，在HotSpot VM中，它是一个叫做StringTable的全局表。在字符串常量池中维护的是字符串实例的引用，底层C++实现就是一个Hashtable。这些被维护的引用所指的字符串实例，被称作”被驻留的字符串”或”interned string”或通常所说的”进入了字符串常量池的字符串”。
- 基本类型包装类对象常量池：java中基本类型的包装类的大部分都实现了常量池技术，这些类是Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另外上面这5种整型的包装类也只是在对应值小于等于127时才可使用对象池，也即对象不负责创建和管理大于127的这些类的对象。

运行时数据区



PCR

一小段内存空间， 无GC无OOM线程私有，记录当前线程执行的字节码指令地址

虚拟机栈

每个线程有一个栈，有Stack Overflow以及OutofMemory的问题，栈中存的是栈帧，是调用的方法

> -Xss选项来设置线程的最大栈空间

栈帧的结构

局部变量表

大小在**编译器确定**，数字数组存储局部变量，方法参数，返回类型

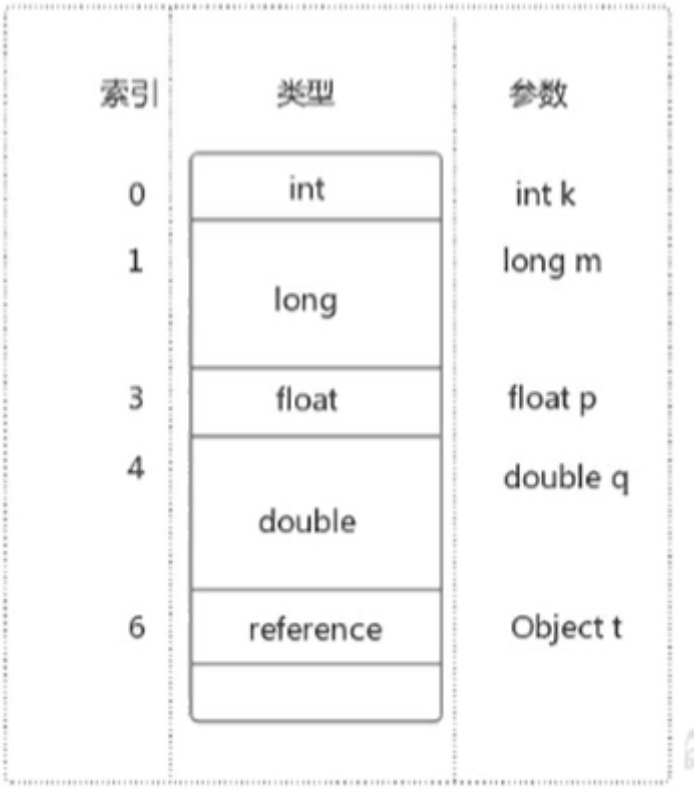
基本单位是slot，32位1个slot，64位2个slot

byte、short、char 在存储前被转换为int，boolean也被转换为int

在构造方法和实例方法(非static)中还需在index0处多一个this对象引用

局部变量表中的变量不存在**链接时准备**和**初始化**阶段，必须人为赋值

局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引用的对象都不会被回收。



操作数栈

32bit的类型占用一个栈单位深度，64bit的类型占用两个栈单位深度

栈顶缓存技术，将栈顶元素全部缓存在物理CPU的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率。

方法出口

存调用者的PC计数器的值

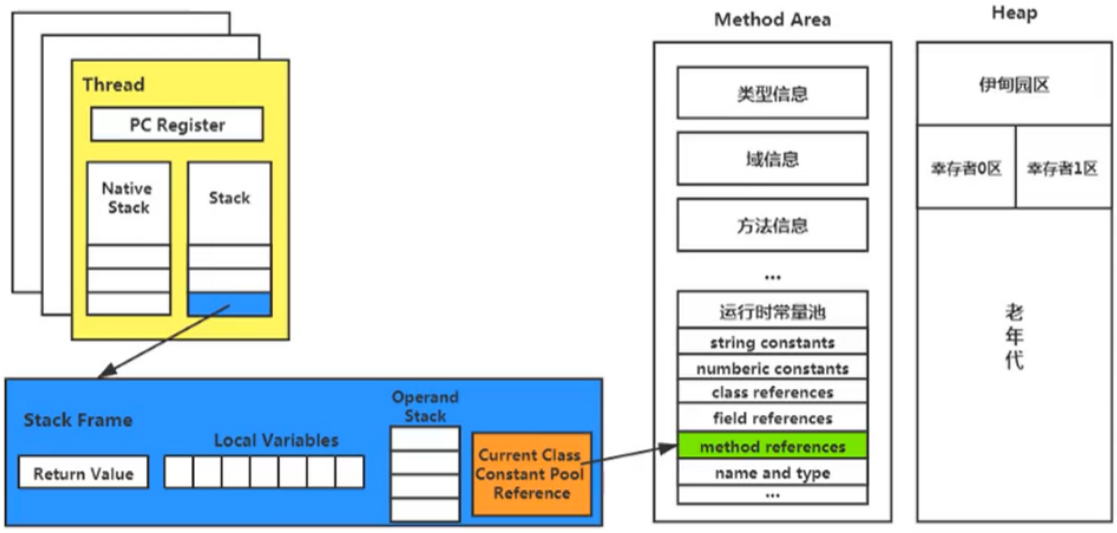
只有返回和抛异常会弹栈

通过异常退出的，返回地址是要通过异常表来确定，栈帧中一般不会保存这部分信息。

动态链接

指向方法区中运行时常量池中该栈帧所属的方法引用

动态链接的作用就是为了将这些**符号引用**转换为调用方法的**直接引用**。



普通调用指令：

- invokestatic：调用静态方法，解析阶段确定唯一方法版本
- invokespecial：调用方法、私有及父类方法，解析阶段确定唯一方法版本

- invokevirtual：调用所有虚方法
- invokeinterface：调用接口方法
- invokedynamic：动态解析出需要调用的方法，然后执行 lambda表达式

方法重写和重载原理

重载

静态分派

依据静态类型，将方法的符号引用指向invokevirtual指令参数中

重写

动态分派

调用invokevirtual

过程

1. 找到操作数栈顶的第一个元素所执行的对象的实际类型，记作C。
2. 如果该类的虚方法表中有该方法，就调用否则
3. 从下到上找到同名方法调用

这样就把常量池中的类方法符号引用解析到了不同的直接引用上

JVM采用在类的方法区建立一个虚方法表 （virtual method table）来实现，虚方法表中存储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的方法地址入口是一致的，都是指向父类的实现入口。如果子类重写了这个方法，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

本地方法栈

堆

jdk 1.7 年轻代 老年代 永久代(使用虚拟机内存)

jdk 1.8 年轻代 老年代 元空间(使用本地内存)

年轻代中分为 eden survivor0 survivor1

> “-Xms"用于表示堆区的起始内存，等价于`-XX:InitialHeapSize`
>
> “-Xmx"则用于表示堆区的最大内存，等价于`-XX:MaxHeapSize`

GC分类

- 部分收集：不是完整收集整个Java堆的垃圾收集。其中又分为：
- - 新生代收集（Minor GC / Young GC）：只是新生代（eden s0 s1)的垃圾收集
- - 老年代收集（Major GC / Old GC）：只是老年代的圾收集。
- - - 目前，只有CMS GC会有单独收集老年代的行为。
- - - 注意，很多时候Major GC会和Full GC混淆使用，需要具体分辨是老年代回收还是整堆回收。
- - 混合收集（MixedGC）：收集整个新生代以及部分老年代的垃圾收集。
- - - 目前，只有G1 GC会有这种行为
- 整堆收集（Full GC）：收集整个java堆和方法区的垃圾收集。

对象分配与分代GC过程

现在eden中new对象，如果满了触发minorGC(YOUNG GC)，将剩余的对象放到幸存者0区

1之后每次触发minorGC 都会将幸存者区中有的和eden剩余的放到另一个幸存者区中，并对每个对象标记次数超过15次 进入老年代

2如果幸存者区不足发生担保分配，历次晋升老年代对象的平均大小是否小于老年代空间，小于说明有可能youngGC后eden存活对象小于老年代从而可以直接升入老年代，否则直接fullGC

3大对象直接升级老年代

4 Survivor 区相同年龄所有对象大小的总和 >Survivor 区内存大小 * 50%， 减少复制次数

当老年代满了进行 MajorGC(OLD GC),若执行后依然空间不够，触发OOM

进入老年代的几种情况

分代超过15 1

担保分配 2

大对象直接进入3

动态对象判断 4

多线程下分配对象--TLAB

JVM为每个线程分配了一个私有缓存区域**TLAB**（Thread Local Allocation Buffer），它包含在Eden空间内。 作用是减少并发堆分配内存时加锁的次数

编译优化:逃逸分析

- 栈上分配
- 同步省略 锁消除
- 标量替换 经过逃逸分析，发现一个对象不会被外界访问的话，那么经过JIT优化，就会把这个对象拆解成若干个其中包含的若干个成员变量来代替。

StringTable

字符串常量池 hashtable

- 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。 除了“abc”+"def"这种会被编译期间优化
- 直接new出来的String("abc")，因为new了会在堆上创建，因为“abc"会在常量池创建
- 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

方法区

内部结构

jdk1.7后静态变量在堆上

运行时常量池

- 运行时常量池中包含多种不同的常量，包括**编译期就已经明确**的数值字面量和符号引用，也包括到**运行期解析后**才能够获得的方法或者字段引用。此时不再是常量池中的符号地址了，这里换为真实地址。

与字节码文件中的常量池区别

- 常量池表（Constant Pool Table）是Class文件的一部分，用于存放编译期生成的各种字面量与符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

方法区jdk 1.678的演进

6 永久代 静态变量在方法区 stringtable在运行时常量池

7 永久代 静态变量 stringtable到堆上 方便GC

8 元空间 静态变量 stringtable到堆上

为什么从永久代变为元空间

永久代调优困难

永久代空间大小难以确定，元空间使用本地直接内存

直接内存

- 因此出于性能考虑，读写频繁的场合可能会考虑使用直接内存。
- Java的NIO库允许Java程序使用直接内存，用于数据缓冲区

对象内存

创建对象的步骤

1类加载检查

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

指针碰撞 （内存整齐） 和空闲列表（内存有间隙）

3.并发分配内存

CAS+失败重试

TLAB eden中对于每个线程有个私有区域，减少加锁

4.默认初始化

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。

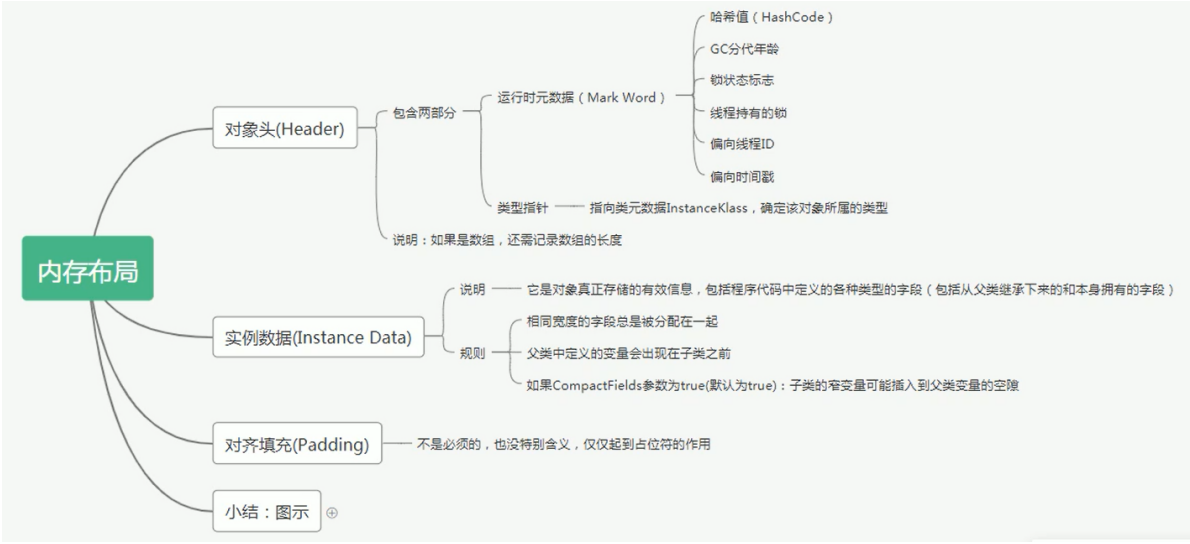
如果静态变量则会在解析的准备过程中初始化 ，常量则在编译时初始化

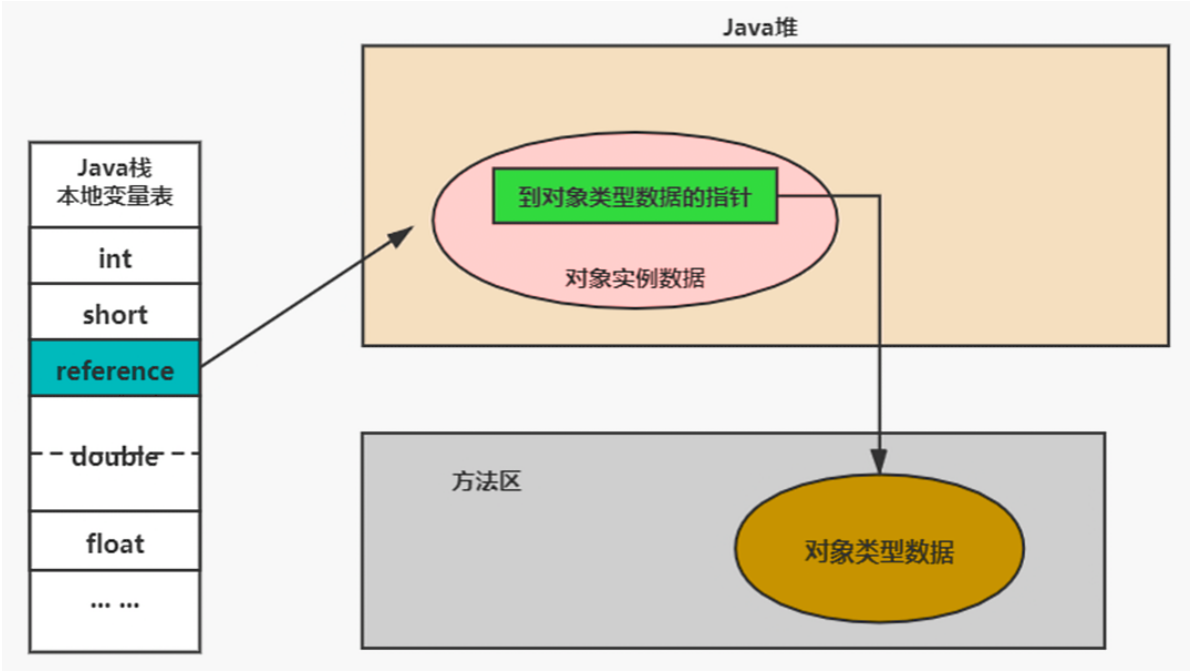
5.设置对象头

6.init初始化

初始化成员变量，执行实例化代码块，调用类的构造方法，并把堆内对象的首地址赋值给引用变量。

内存布局





引用指向堆对象头的类型指针，然后指向方法区对象类型数据

垃圾回收

相关算法

标记阶段

判断是否可以被回收

引用计数

计数该对象有多少个引用

无法解决循环引用问题 即自环之后无其他引用， java不用

可达性分析

从GCRoots上搜索连接对象是否可达

三色标记法

白：未标记的垃圾

黑：已标记，且属性已标记完

灰：已标记，但属性未标记完

过程：将gcroot加入白色集合，根节点开始遍历，遍历到的对象从白色加入灰色，遍历灰色对象的属性对象，

如果是白色加入灰色，如果该灰色对象和属性全灰，加入黑色，将白色GC

缺点：

1. 浮动垃圾，上层黑灰对象突然变白，
2. 漏标 两个条件**灰色断开白色同时**，**黑色对象引用此白色对象**，导致黑色不会再标记此白色对象，而白色对象被GC

CMS优化

增量更新（解决第二个条件)

在一个未被标记的对象（白色对象）被重新引用后，**引用它的对象若为黑色则要变成灰色**

不解决根本问题

G1优化

SATB（解决第一个条件）

1. 在开始标记的时候生成一个快照图标记存活对象
2. 在一个引用断开后，要将此引用推到GC的堆栈里，保证白色对象（垃圾）还能被GC线程扫描到(在**write barrier(写屏障)**里把所有旧的引用所指向的对象都变成非白的)。
3. 配合`Rset`，去扫描哪些Region引用到当前的白色对象，若没有引用到当前对象，则回收

GCRoots分类

- 虚拟机栈中引用的对象 比如：各个线程被调用的方法中使用到的参数、局部变量等。
- 本地方法栈内JNI（通常说的本地方法）引用的对象
- 方法区中类静态属性引用的对象 比如：Java类的引用类型静态变量
- 方法区中常量引用的对象 比如：字符串常量池（String Table）里的引用
- 所有被同步锁synchronized持有的对象
- Java虚拟机内部的引用。 基本数据类型对应的Class对象，一些常驻的异常对象（如：NullPointerException、OutOfMemoryError），系统类加载器。
- 反映java虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等。

由于finalization的存在，对象可能“复活”，所有对象状态有三种 可达，可复活，不可达

当obj不可达GCRoots,第一次标记，如果无finalize方法，直接判定为不可达，如果重写了finalize方法，且未执行,插入F-queue线程，触发finalize方法执行

GC对F-queue中的对象进行第二次标记，如果有引用指向，被移除F-queue，之后再出现不可达时，不会再调用finalize直接GC,如果没有则直接GC

清除阶段

标记清除算法

标记 从GCRoot开始遍历，标记所有被引用的对象，在header中记录为可达对象

清除 线性遍历内存，如果某对象不可达，将其回收

缺点

效率低，有内存碎片，需维护空闲列表，需STW(stop the world)

复制算法

s0 s1 建立在存活对象少的基础上效率很高

两块内存，每次使用一块，将存活对象复制到未被使用的内存块中

优点

无碎片 高效

缺点

两块内存

标记压缩算法

清除后移动位置，使其内存紧凑

优点

内存不需减半 且不需要维护空闲列表

缺点

效率低，需STW(stop the world)

相关概念

system.gc()或者Runtime.getRuntime().gc() 的调用，会显式触发Full GC

system.gc()不会直接调用

强引用、软引用、弱引用、虚引用

强 `Object obj = new Object()`”

软 只有在内存不足时，系统则会回收软引用对象 JDK1.2版之后提供了java.lang.ref.SoftReference类来实现软引用

```
Object obj = new Object(); // 声明强引用
SoftReference<Object> sf = new SoftReference<>(obj); 软引用
WeakReference<Object> sf = new WeakReference<>(obj); 弱引用
obj = null; //销毁强引用
```

弱 当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。

虚 用来跟踪对象被垃圾回收的活动。必须和引用队列一起用 用来追踪GC

OopMap

全称是 `Ordinary Object Pointer` `Map`

在 `OopMap` 的协助下，HotSpot 就能快速准确地完成 `GC Roots` 枚举啦。

什么是安全点 安全区域?

安全点

如果每个指令都更新OOPMAP,会很浪费时间和空间，在安全点上记录

可以停下来GC的位置称为安全点，是否具有让程序长时间执行的特征为标准

方法调用、循环跳转、异常跳转等这些地方才会产生安全点

抢先式中断

先全中断线程，然后让不在安全点的跑到安全点

主动式中断

设置中断标志，运行到安全点主动轮询，如果真则中断挂起

安全区域

如果有线程挂起状态，无法在安全点主动挂起，则可以标志自己进入安全区域，这样如果从挂起态变为运行态时，则会检查是否GC结束

垃圾回收器

垃圾收集器	分类	作用位置	使用算法	特点	适用场景
Serial	串行运行	作用于新生代	复制算法	响应速度优先	适用于单CPU环境下的client模式
ParNew	并行运行	作用于新生代	复制算法	响应速度优先	多CPU环境Server模式下与CMS配合使用
Parallel Scavenge	并行运行	作用于新生代	复制算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
Serial Old	串行运行	作用于老年代	标记-压缩算法	响应速度优先	适用于单CPU环境下的Client模式
Parallel Old	并行运行	作用于老年代	标记-压缩算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
CMS	并发运行	作用于老年代	标记-清除算法	响应速度优先	适用于互联网或B / S业务
G1	并发、并行运行	作用于新生代、老年代	标记-压缩算法、复制算法	响应速度优先	面向服务端应用

jdk1.8默认 Parallel Scavenge Parallel Old

CMS垃圾回收器

过程

初始标记 标记出GCRoots能直接关联到的对象 有STW

并发标记 直接关联对象开始遍历整个对象图的过程 无STW

重新标记 修正已标记对象中被引用的那部分 有STW

并发清除

缺点

会产生内存碎片

占用cpu

浮动垃圾：并发过程中产生新垃圾

G1垃圾回收器

Card Table（多种垃圾回收器均具备）

- 由于在进行`YoungGC`时，我们在进行对一个对象是否被引用的过程，需要扫描整个Old区，所以JVM设计了`CardTable`，将Old区分为一个一个Card，一个Card有多个对象；如果一个Card中的对象有引用指向Young区，则将其标记为`Dirty Card`，下次需要进行`YoungGC`时，只需要去扫描`Dirty Card`
- Card Table 在底层数据结构以 `Bit Map`实现。

RSet(Remembered Set)

Card Table则是一种points-out（我引用了谁的对象）的结构，每个Card 覆盖一定范围的Heap（一般为512Bytes）。G1的RSet是在Card Table的基础上实现的：每个Region会记录下别的Region有指向自己的指针，并标记这些指针分别在哪些Card的范围内。

CSet(Collection Set)

一组可被回收的分区Region的集合, 是多个对象的集合内存区域。

不物理分代，而是逻辑分代

G1不再坚持固定大小以及固定数量的分代区域划分，而是把连续的Java堆划分为多个大小相等的独立区域（Region），每一个Region都可以根据需要，扮演新生代的Eden空间、Survivor空间，或者老年代空间

初始标记（会STW）

并发标记

最终标记

清理阶段

调优

jps 查看java进程

jstat jInfo 监控

jmap 生成dump

jhat 分析dump

jstack Thread类中有getAllStackTrace可以代替 知道线程锁的情况

经验

1在大访问压力下，MinorGC 频繁，MinorGC 是针对新生代进行回收的，每次在MGC存活下来的对象，会移动到Survivor1区。先到这里为止，大访问压力下，MGC频繁-十些是正常的，只要MGC延迟不导致停顿时间过长或者引发FGC，那可以适当的增大Eden空间大小，降低频繁程度，同时要保证，空间增大对垃圾回收时间产生的停顿时间增长也是可以接受的。 如果MinorGC 频繁，且容易引发FullGC。需要从如下几个角度进行分析。

a:每次MGC存活的对象的大小，是否能够全部移动到S1区，如果S1区大小< MGC存活的对象大小，这批对象会直接进入老年代。注意 了， 这批对象的年龄才1岁，很有可能再多等1次MGC 就能被回收了，可是却进入了老年代，只能等到FullGC进行回收，很可怕。这种情况下，应该在系统压测的情况下，实时监控MGC存活的对象大小，并合理调整eden和s区的大小以及比例。

b:还有一种情况会导致对象在未达到15岁之前，直接进入老年代，就是S1区的对象，相同年龄的对象所占总空间大小>s1区空间大小的一半，所以为了应对这种情况，对于S区的大小的调整就要考虑:尽量保证峰值状态下，S1区的对象所占空间能够在MGC的过程中，相同对象年龄所占空间不大于S1区空间的一-半,因此对于S1空间大小的调整，也是十分重要的。

如果是一次fullgc后，剩余对象不多。那么说明eden区设置太小，导致短生命周期短的对象进入了old区。

如果一次fullgc后，old区回收率不大，那么说明old区太小。

2.由于大对象创建频繁，导致Full GC频繁。对于大对象，JVM专门有参数进行控制，-XX:PretenureSizeThreshold。超过这个参数值的对象，会直接进入老年代，只能等到full GC进行回收，所以在系统压测过程中，要重点监测大对象的产生。如果能够优化对象大小，则进行代码层面的优化，优化如:根据业务需求看是否可以将该大对象设置为单例模式下的对象，或者该大对象是否可以进行拆分使用，或者如果大对象确定使用完成后[将该对象赋值为null，方便垃圾回收。如果代码层面无法优化，则需要考虑: a:调高-XX:PretenureSizeThreshold参数的大小，使对象有机会在eden区创建，有机会经历MGC以被回收。但是这个参数的调整要结合MGC过程中Eden区的大小是否能够承载，包括S1区的大小承载问题。b:这是最不希望发生的情况，如果必须要进入老年代，也要尽量保证，该对象确实是长时间使用的对象，放入老年代的总对象创建量不会造成老年代的内存空间迅速长满发生Full GC，在这种情况下，可以通过定时脚本，在业务系统不繁忙情况下，主动触发ull gC。

大对象拆分 单例 调高阈值 定时脚本FULLGC