

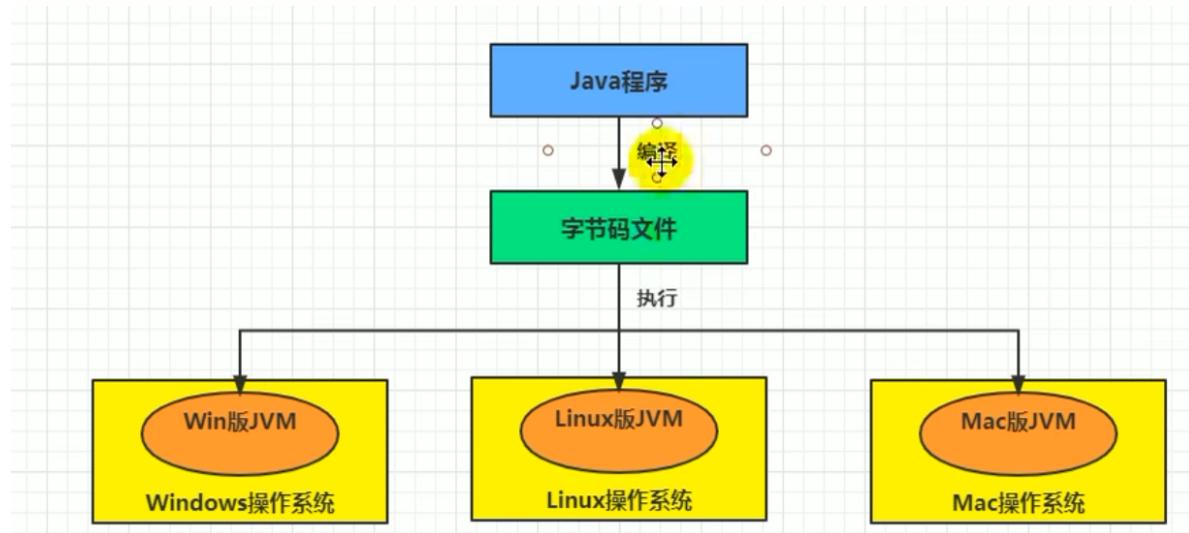
jvm

虚拟机就是一台虚拟的电脑，java虚拟机jvm是程序虚拟机而不是系统虚拟机，专门执行单个计算机程序，执行的指令是jvm字节码指令。

无关编程语言，只要是jvm字节码就可以。所以就有了一次编程，到处运行的说法，也就是编程后编译成字节码文件。有jvm虚拟机就可以运行。

为什么可以一次编译到处运行

因为每个操作系统都有规范的jvm



编程和编译

编程是.java文件，.java文件编译成字节码文件是.class文件

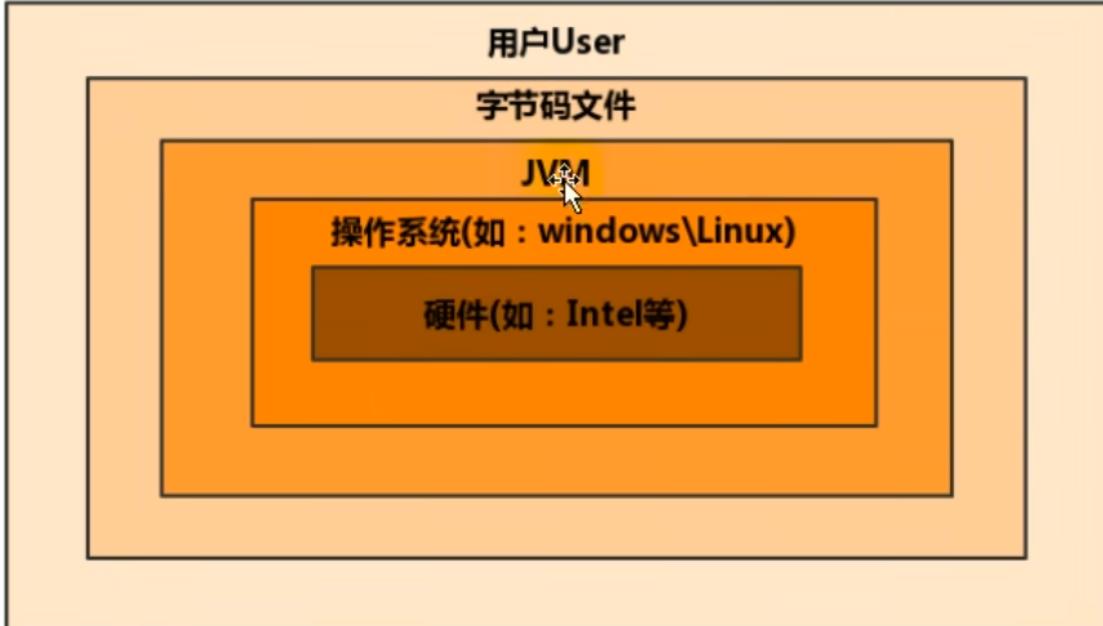
特点

一次编译到处运行

自动内存管理和垃圾回收

位置

jvm是运行在操作系统之上的，没有与硬件进行交互



jvm架构

由于跨平台的设计，对于指令的操作是基于栈式结构

	栈式结构	寄存器结构
依赖	不依赖硬件 (耦合性弱但性能差)	依赖硬件 (因为指令集的问题) (耦合性强但性能好)
优点	跨平台, 指令集小	非跨平台, 指令集多
缺点	指令更多, 执行性能差	指令更少, 执行性能好

举例1：

同样执行 $2+3$ 这种逻辑操作，其指令分别如下：
基于栈的计算流程（以Java虚拟机为例）：

```
1  iconst_2 //常量2入栈  
2  istore_1  
3  iconst_3 //常量3入栈  
4  istore_2  
5  iload_1  
6  iload_2  
7  iadd      //常量2、3出栈，执行相加  
8  istore_0 //结果5入栈
```

而基于寄存器的计算流程：

```
1  mov eax,2 //将eax寄存器的值设为1  
2  add eax,3 //使eax寄存器的值加3
```

jvm的生命周期

虚拟机的启动

我们运行一个类的时候是有其他类来调用该类的，不同类的类加载器不同，但最开始是由引导类加载器来创建一个初始类。

可以理解为你想吃苹果，但也得有苹果树，所以jvm虚拟机就用引导类加载器先创建苹果树。

虚拟机的执行

执行一个java程序其实是执行一个java虚拟机进程

虚拟机的退出

程序正常结束，程序异常，操作系统错误，主动结束程序。

虚拟机版本

虚拟机很多，学的是oracle hotspot

类加载器

功能

负责加载Class文件，将类信息存放到方法区。至于能否运行由执行引擎决定。

注意：一个Class文件只会被类加载器器加载一次！（这跟static知识点有关）。

过程

加载

1. 通过一个类的全限定名获取定义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，注意是static！！！
3. 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

链接

链接分为三个子阶段：验证 → 准备 → 解析

验证：Class文件符合虚拟机要求，不危害虚拟机安全

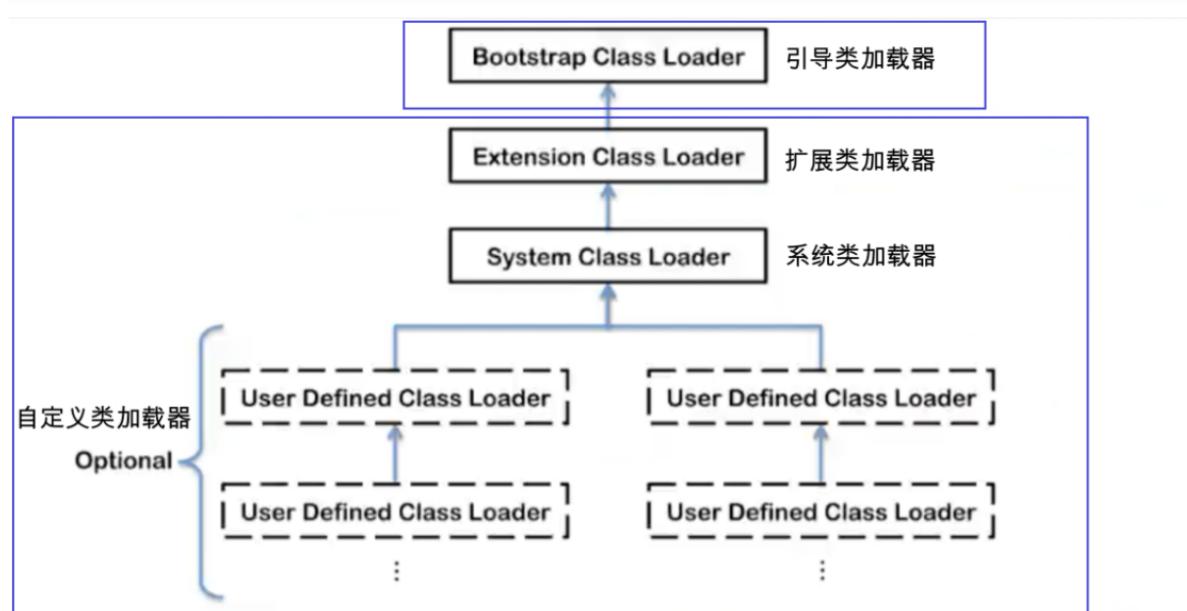
准备：对类变量默认赋值，对final的话是直接赋值

解析：指针的引用

初始化

执行类的构造器方法，若该类具有父类，会先执行父类的构造器方法。 对类变量显式赋值即静态代码块赋值

类加载器分类



这里的四者之间的关系是包含关系。不是上层下层，也不是子父类的继承关系。

	编写	加载的类
启动类加载器（引导类加载器，嵌套在JVM内部）	C/C++语言	加载Java的核心库
扩展类加载器	Java	加载 Java 的扩展库
系统类加载器	Java	Java 应用的类

我们尝试获取引导类加载器，获取到的值为 `null`，这并不代表引导类加载器不存在，因为引导类加载器是由 C/C++ 语言构成的，所以我们是获取不到

还可以自定义类加载器，继承java.lang.ClassLoader

用于

- 隔离加载类
- 修改类加载的方式
- 扩展加载源
- 防止源码泄露

双亲委派机制

如果一个类加载器收到类加载请求，先让父类加载器执行，不可以再让自己执行。

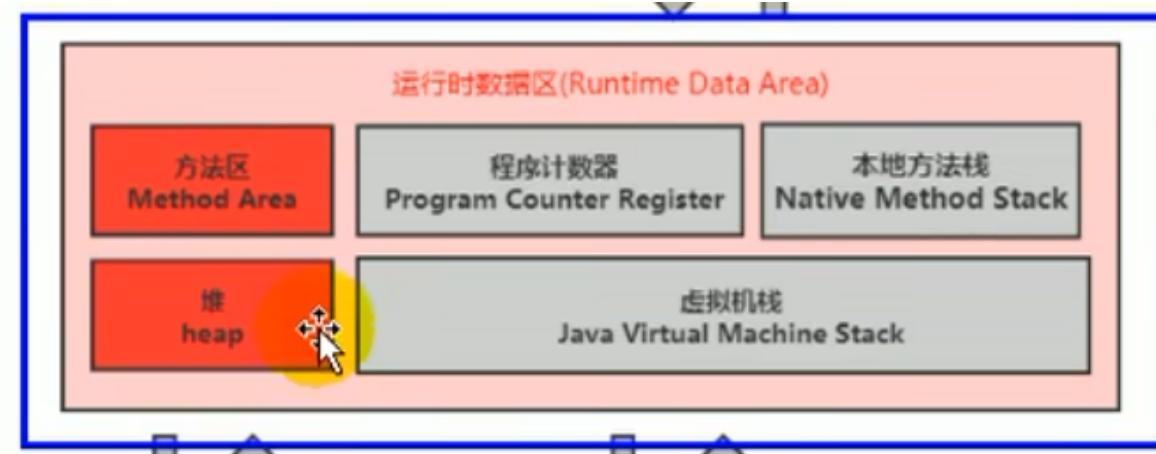
优势：避免类的重复加载，保护程序安全，防止核心api被串改。

沙箱安全机制

自定义String类，但是在加载自定义String类的时候会率先使用引导类加载器加载，而引导类加载器在加载的过程中会先加载jdk自带的文件(rt.jar包中java\lang\String.class)，报错信息说没有main方法，就是因为加载的是rt.jar包中的String类。这样可以保证对java核心源代码的保护，这就是**沙箱安全机制**。

运行时数据区

结构



方法区和堆空间对于线程来说是公用的。线程之间有各自的程序计数器，本地方法栈，虚拟机栈

程序计数寄存器 (pc寄存器) 不会GC/OOM

存储当前指令的地址。让执行引擎根据地址执行

问题

为什么要存地址？

因为cpu不断切换线程，没有存地址，执行引擎不知道从哪开始执行

虚拟机栈不会GC/会OOM

栈最下面的方法先是构造方法再是main方法

生命周期：线程一致

作用：保存方法局部变量，对象的引用地址和结果

优点：是一种快速有效的分配存储方式（分配能力很快），访问速度仅次于程序计数栈

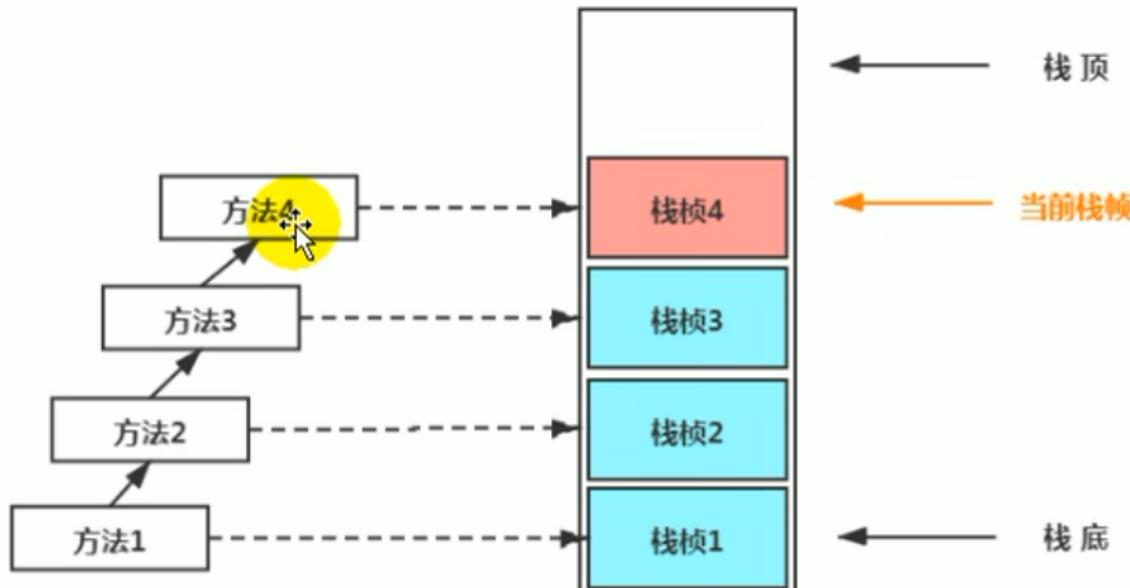
可设置栈大小：不扩展

异常

stackOverflowError（栈溢出异常），栈帧数量太多

outOfMemoryError（内存溢出异常），内存不足，栈内存扩的比内存还大

栈的存储单位



栈的数据以栈帧格式存在

Object Oriented Programming oop面向对象的方法

执行方法：入栈

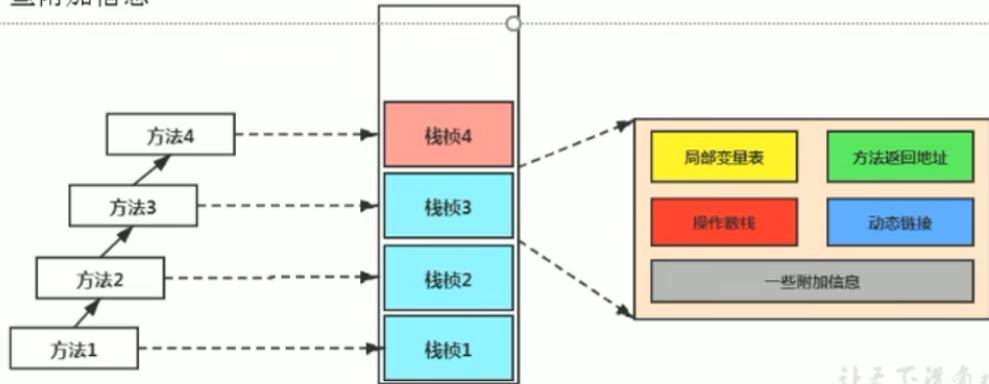
结束方法（包含return）或者抛出异常：出栈

栈帧结构

栈帧的数量取决于栈帧的大小，栈帧的大小取决于局部变量表和操作数栈大小

局部变量表和操作数栈式重点

- 每个栈帧中存储着：
- 局部变量表 (Local Variables)
 - 操作数栈 (Operand Stack) (或表达式栈)
 - 动态链接 (Dynamic Linking) (或指向运行时常量池的方法引用)
 - 方法返回地址 (Return Address) (或方法正常退出或者异常退出的定义)
 - 一些附加信息



让天下没有难学的技术

局部变量表的重要性

只要被局部变量表直接或者间接引用的对象都不会被回收

局部变量表：local variables

结构：以数组的形式存放，有参数，局部变量，（byte, short, char, boolean都以int保存）返回值，
默认this也是一个变量存放在index0，double和long要占据两个槽slot，占据两个数据元素，一个槽4个字节

节省空间：代码块

```
public static void main(String[] args) {
{
    int a=0;
}
//此时b会复用a的槽位
int b=0;
}
```

注意：静态方法的局部变量表没有this变量！！！

生命周期：方法结束，局部变量表销毁

题外话变量

按照数据类型可分为基本数据类型和引用数据类型

按照位置可分为成员变量

成员变量又可分为类变量：在链接阶段的准备会给类赋初始值，在初始化阶段会调用静态代码块给类变量赋值

实例变量：随着对象的创建，在堆空间会分配实例变量空间并有默认值

局部变量：在使用前要显示赋值，否则编译不通过

操作数栈

数组实现，其实就是在方法里面指令执行也是按栈的形式入栈出栈



关于指令执行过程看网课比较形象

https://www.bilibili.com/video/BV1PJ411n7xZ/?p=53&spm_id_from=pageDriver&vd_source=003e9f54d7ed05b77aee32ac0f194b0a

动态链接

当你用到一些符号引用的方法或者常量，你就需要动态链接去方法区的运行时常量池将符号引号转成直接引用。

方法返回地址

将上一个方法所执行的地址存到本方法的方法返回地址，以便本方法执行完，执行引擎知道上一个方法所执行的位置。

一些附加信息

可选的虚拟机栈信息，了解即可

栈的面试题

栈溢出的情况StackOverflow：栈帧太多了

调整栈大小就能保证不出现溢出？不能，因为也不一定

垃圾回收会涉及虚拟机栈？进栈出栈不会gc

分配的栈内存越大越好？不是，内存固定的，会影响其他的内存

方法中定义的局部变量是否线程安全？不是的，方法参数传进来，局部变量指针指向，该参数是共享的就不是安全的。

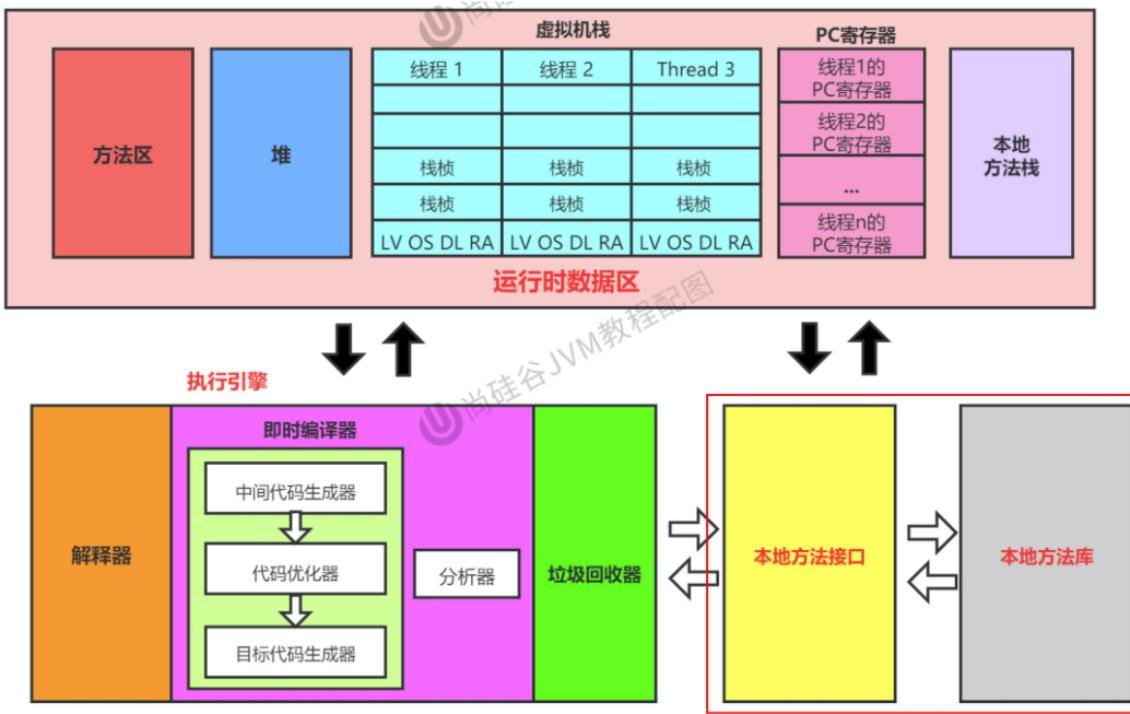
早期绑定和晚期绑定

编译可以确定就是早期绑定，运行期间才确定是晚期绑定

类的虚方法表

多态的知识点，每个类都有一个虚方法表，就不用每次对不确定的方法进行判断。

本地方法接口和本地方法库



方法的实现不是由java实现的由c/c++实现的，即有native关键字，也就是本地方法。

jvm是依赖具体的操作系统，操作系统又是由c或c++写的，所以有时用c或者c++更好

本地方方法栈

管理本地方方法的调用，跟虚拟机栈一样。

堆GC/OOM

jvm启动就创建堆，内存大小也就确定（可调节），是jvm管理最大的一块内存空间。

线程共享java堆，所有的对象实例以及数组都分配在堆上。栈上的栈帧保存的是引用。类及其方法实现是在方法区。

垃圾回收

对象和数组没有引用，垃圾回收启动时候就会垃圾回收。（频繁gc会影响性能）

分区

逻辑上分为新生区，养老区，元空间（java7叫的是永久区），我们设计起始内存大小和最大内存大小管理的是新生区养老区，默认初始内存大小是（电脑物理内存/64），最大内存大小是（电脑物理内存/4）。一般我们设置初始内存和最大内存一致，防止扩容影响压力。

查看内存不详细讲，用的时候查。

新生区中又分为eden, (survivor0) from, (survivor1) to。from和to计算内存是二选一

新生代和老年代的比例可以手动设置

异常

throw可抛出有两个子类，error错误和exception异常

对象分配和垃圾回收

对象会分配在新生代的伊甸园区，当满的时候会触发YGC再放对象，没有被清理的对象会放在幸存者0区。当伊甸园区再次满，幸存者0区和伊甸园区没有被清理都会放在幸存者1区。注意移动时候年龄会+1，初始年龄为0。当伊甸园区再次满，幸存者1区和伊甸园区没有被清理都会放在幸存者0区。也就是幸存者0区和1区谁空谁会被放。谁空谁是to。当年龄达到15（默认值，也可以修改），或者是相同年龄大于from一半，则大于等于该年龄，下一次放就会放到老年区。

注意只有伊甸区满才会触发垃圾回收，然后同时也会回收from区！

如果伊甸园区清完垃圾还放不下对象，就会直接放到老年区，老年区放不下会fgc，再放不下就内存溢出OOM。

如果to区也放不下不用年龄达到阈值也直接放到老年区。

垃圾回收区别

yong garbage collection等同minor gc

major gc等同于old gc

full gc

调优的目的是为了gc次数少一些

为什么堆要分代

分代就是优化gc性能，不分代则所有对象一起

缓存问题

每个线程分配了一个私有缓存区域tLab，包含在伊甸园区内，默认仅占1%。

常用参数

- * 测试堆空间常用的jvm参数：
 - * `-XX:+PrintFlagsInitial` : 查看所有的参数的默认初始值
 - * `-XX:+PrintFlagsFinal` : 查看所有的参数的最终值（可能会存在修改，不再是初始值）
 - * `-Xms`: 初始堆空间内存（默认为物理内存的1/64）
 - * `-Xmx`: 最大堆空间内存（默认为物理内存的1/4）
 - * `-Xmn`: 设置新生代的大小。（初始值及最大值）
 - * `-XX:NewRatio`: 配置新生代与老年代在堆结构的占比
 - * `-XX:SurvivorRatio`: 设置新生代中Eden和S0/S1空间的比例
 - * `-XX:MaxTenuringThreshold`: 设置新生代垃圾的最大年龄
 - * `-XX:+PrintGCDetails`: 输出详细的GC处理日志
 - * 打印gc简要信息：① `-XX:+PrintGC` ② `-verbose:gc`
 - * `-XX:HandlePromotionFailure`: 是否设置空间分配担保

堆是分配对象存储的唯一选择？

不是，使用逃逸分析

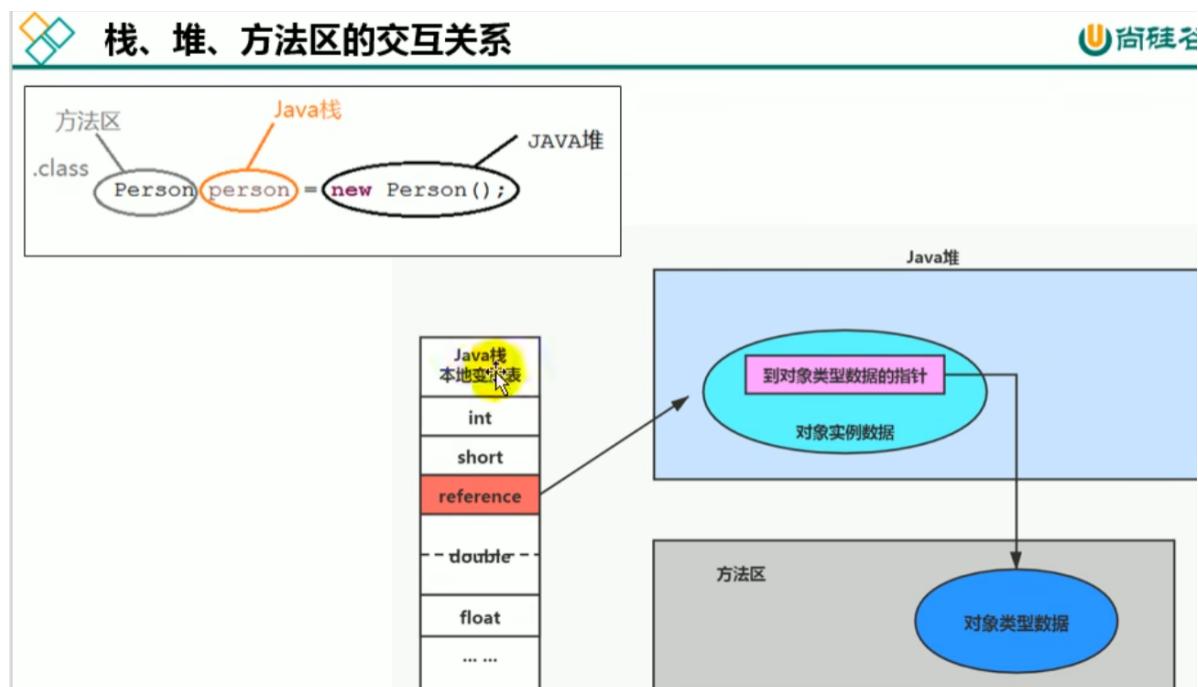
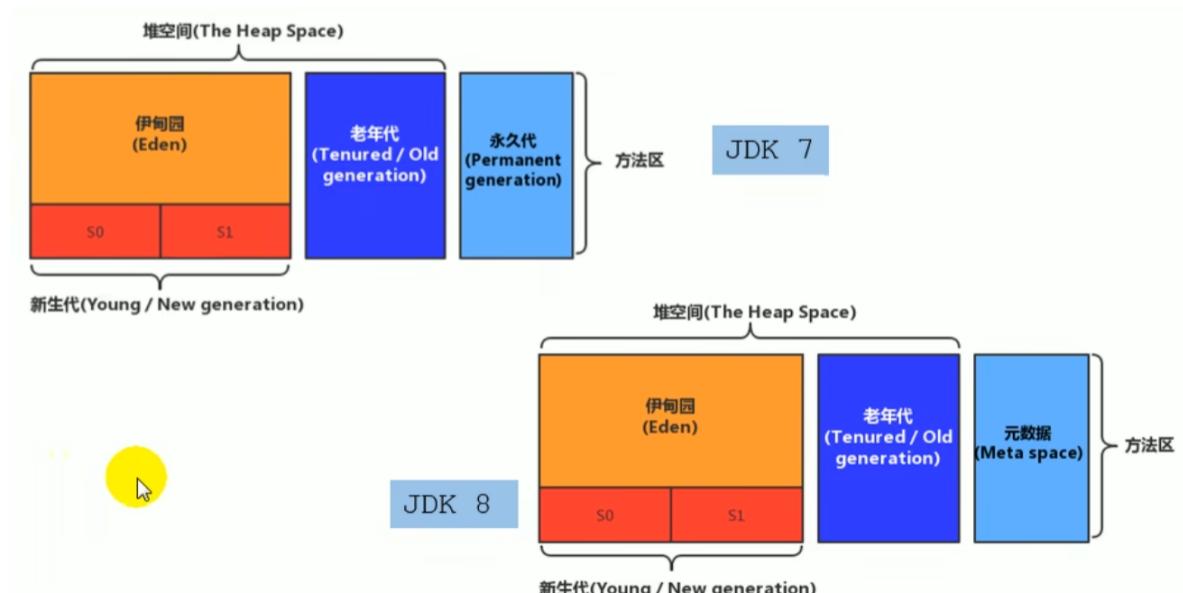
栈上分配：局部变量是则会优化成栈上分配，则减少堆垃圾回收，但是如果局部变量是作为返回值或者被其他方法调用，即逃逸出方法（逃逸是可以设置的），则堆分配。即开发中能使用局部变量，就不要在方法外定义。

标量替换：new一个对象，调用对象的参数也会被优化成直接用参数，也就是标量替换。所以对象还是在堆上，只不过new一个对象，对象变成参数了。

同步省略：一个对象只能从一个线程访问，则对对象即使用编译同步代码块，运行也不会用。

目前没有真正做到栈上分配，是做标量替换。

方法区GC/OOM



方法区（非堆）是一个概念，元空间是方法区的实现。

方法区存放的是类的信息，类太多也会导致OOM，即java.lang.OutOfMemoryError: Metaspace
(jdk7及以下是PerGen space)

元空间和永久代区别在于元空间不在虚拟机设置的内存，而是使用本地内存。

为什么使用本地内存?

设置空间很难确定，调优很难。

存放

类型信息（类的各种信息），常量，静态变量，即时编译器的代码缓存，类加载器也有

字节码文件内部有常量池（相当于一张表，根据这张表可以去找数量值，字符串值，类引用，字段引用，方法引用）

对所有字节码常量池的加载，就是方法区内部的运行时常量池，注意动态性，a找b，b找c，在运行时常量池就是直接a找c，也就是符号地址会变真实地址。

垃圾回收

不回收不行，回收又麻烦，所以主要回收常量池的常量被final修饰。

对象的实例化

先检查该对象的类在元空间的常量池是否有类的符号引用，接着检查类是否加载（加载链接初始化）。没有加载的话在双亲委派模式下加载class文件。没有就报错classnotfoundexception。

为对象在堆中划分内存。内存规整使用指针碰撞法



内存不规整就用一个列表（操作系统的知识）

每个线程预先分配自己的区域tlab。不足的话，堆是共享的，要处理并发安全问题。对区域进行加锁。

接着对属性

1设置默认值

2显式赋值例如int a=2

3代码块赋值

设置对象的对象头。

有哈希值，GC分代年龄，锁状态标志，线程持有的锁，

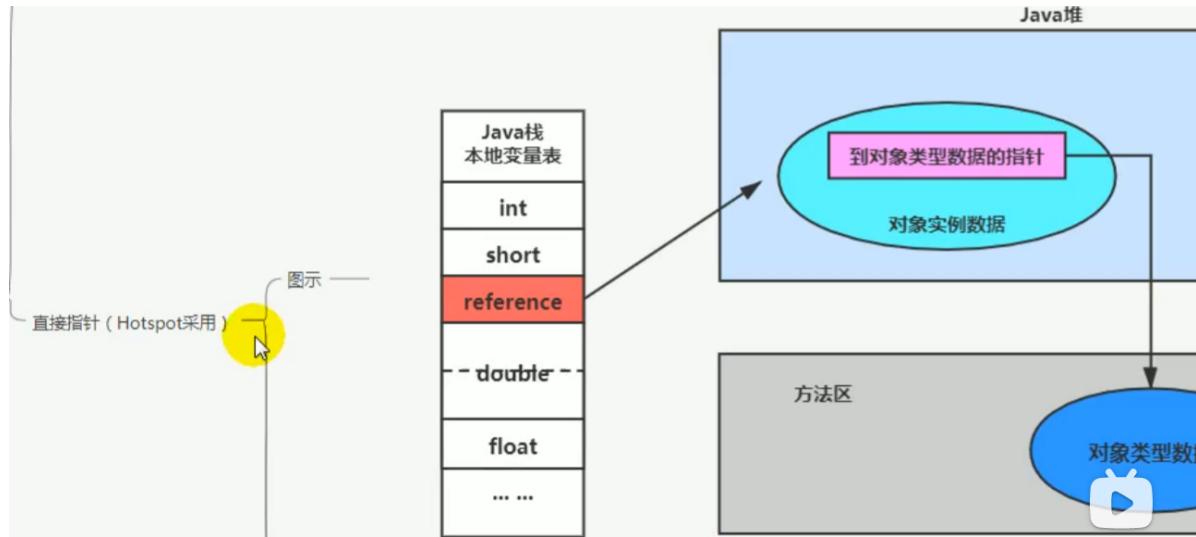
还有类型指针，指向类元数据，确定对象所属的类型。

如果是数组会记录长度。

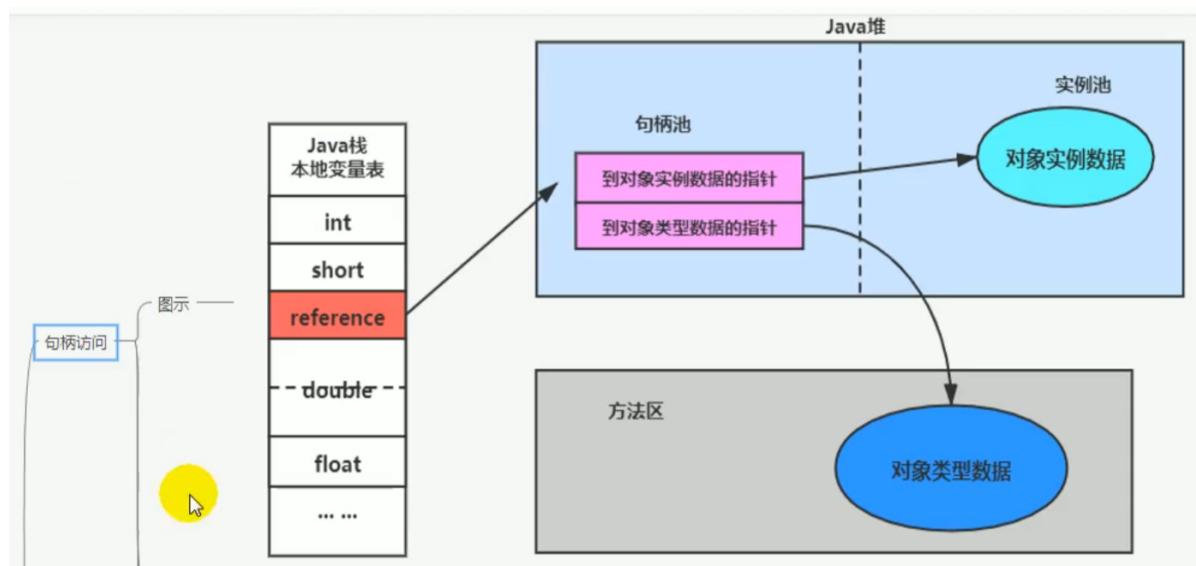
执行初始化方法，构造器方法。

对象访问有两种方式

直接访问（重点）和句柄访问



堆的句柄池有对象实例数据的指针和对象类型数据的指针。



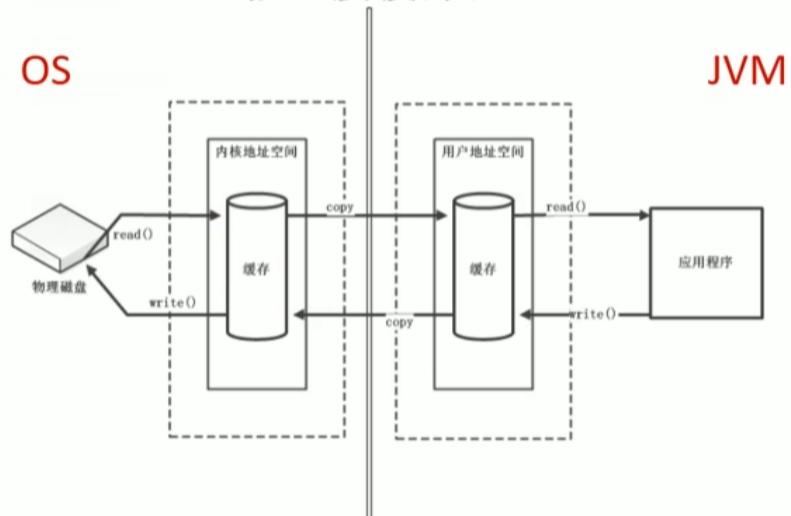
本地内存

元空间不是用虚拟机内存，而是本地内存，也会OOM

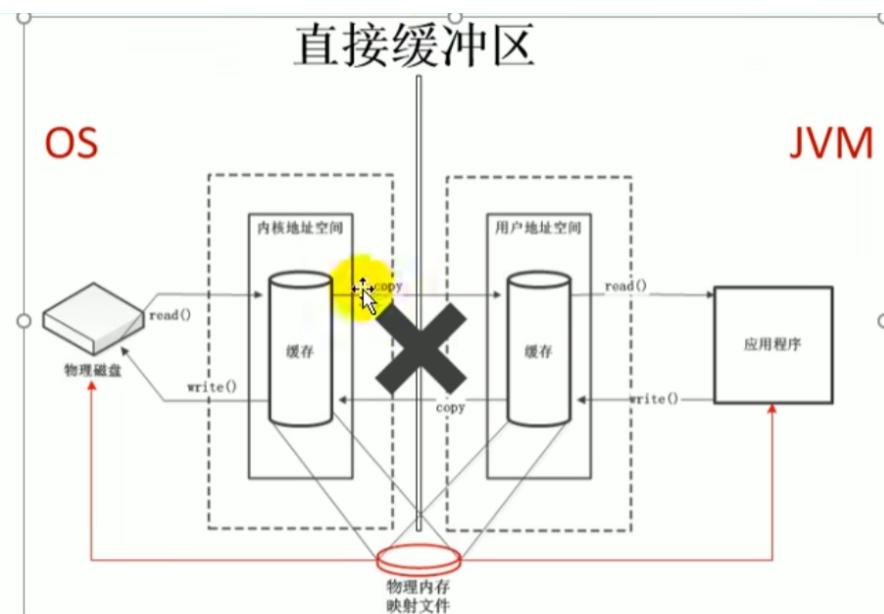
传统io

非直接缓冲区

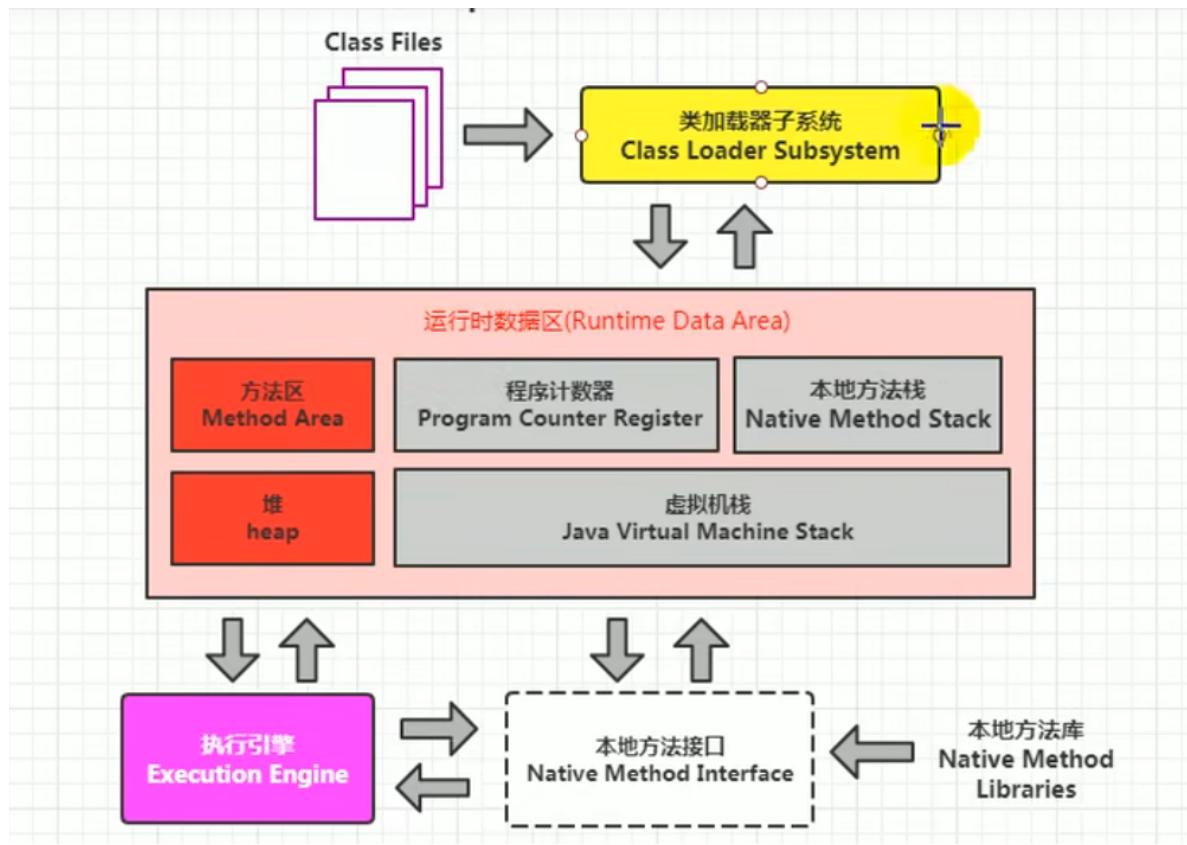
读写文件，需要与磁盘交互，
需要由用户态切换到内核态。
在内核态时，需要内存如右
图的操作。
使用IO，见右图。这里需要
两份内存存储重复数据，效
率低。



nio(即New IO)，是同步非阻塞的



执行引擎



执行引擎是将字节码指令解释/编译为对应平台的本地机器指令才可以，就是将高级语言翻译成机器语言

为什么说java是半编译半解释型语言？

执行java代码时候，解释和编译执行两者结合进行。

解释器将字节码翻译成本地机器指令，即时编译器是将（热点代码）整个函数体编译成机器码，效率更高

各有各存在的道理

当虚拟机启动的时候，**解释器可以首先发挥作用**，而不必等待即时编译器全部编译完成再执行，这样可以**省去许多不必要的编译时间**。并且随着程序运行时间的推移，即时编译器逐渐发挥作用，根据热点探测功能，**将有价值的字节码编译为本地机器指令**，以换取更高的程序执行效率。

编译器分类，优化策略不同

在HotSpot VM中内嵌有两个JIT编译器，分别为Client Compiler和Server Compiler，但大多数情况下我们简称为C1编译器和C2编译器。开发人员可以通过如下命令显式指定Java虚拟机在运行时到底使用哪一种即时编译器，如下所示：

- **-client:** 指定Java虚拟机运行在Client模式下，并使用C1编译器；
➢ C1编译器会对字节码进行**简单和可靠的优化，耗时短**。以达到更快的编译速度。
- **-server:** 指定Java虚拟机运行在Server模式下，并使用C2编译器。
➢ C2进行**耗时较长的优化，以及激进优化**。但优化的代码执行效率更高。

C1和C2编译器不同的优化策略：

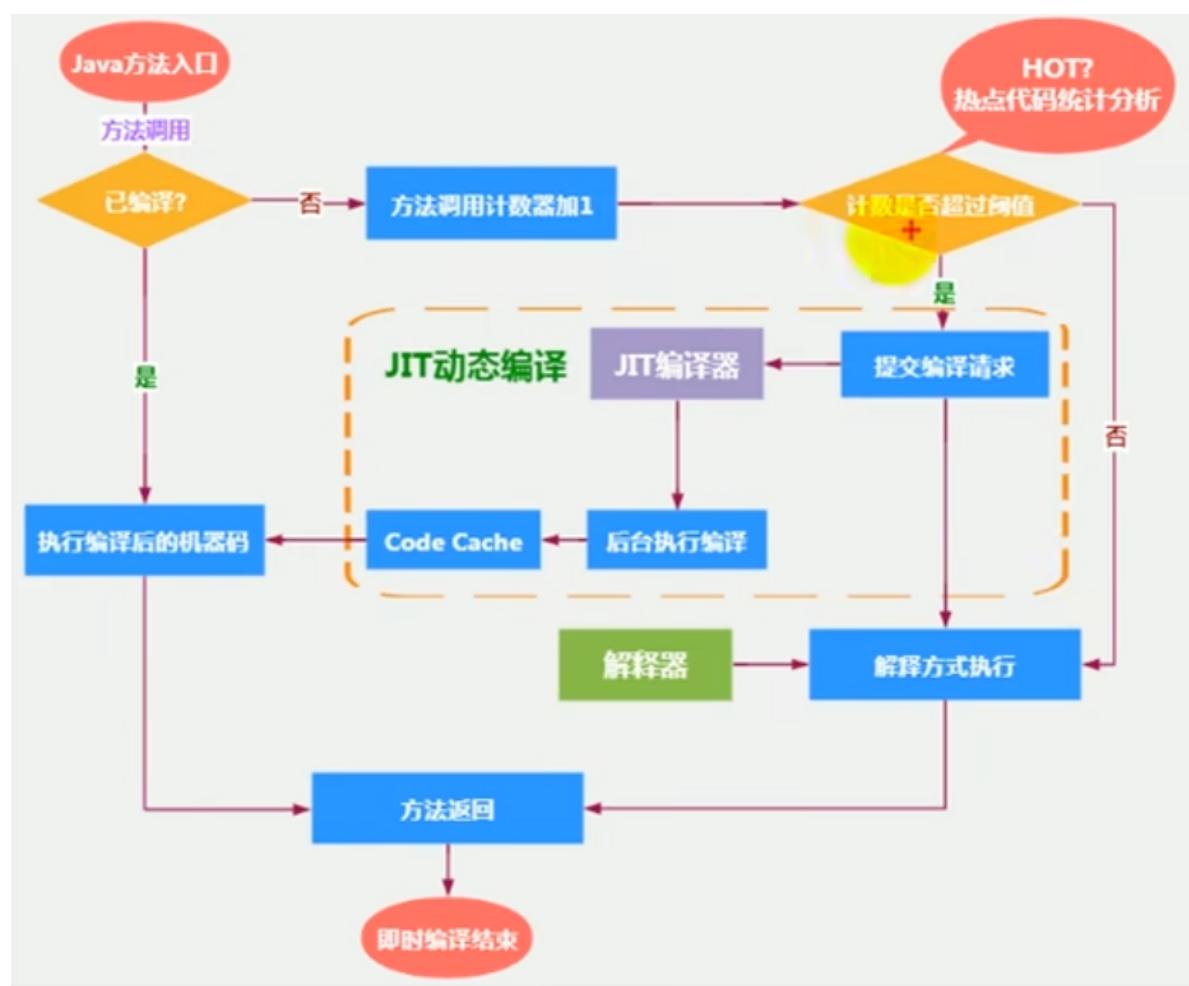
- 在不同的编译器上有不同的优化策略，C1编译器上主要有方法内联，去虚拟化、冗余消除。
 - 方法内联：将引用的函数代码编译到引用点处，这样可以减少栈帧的生成，减少参数传递以及跳转过程
 - 去虚拟化：对唯一的实现类进行内联
 - 冗余消除：在运行期间把一些不会执行的代码折叠掉
- C2的优化主要是在全局层面，逃逸分析是优化的基础。基于逃逸分析在C2上有如下几种优化：
 - 标量替换：用标量值代替聚合对象的属性值
 - 栈上分配：对于未逃逸的对象分配对象在栈而不是堆
 - 同步消除：清除同步操作，通常指synchronized

热点代码的探测

目前HotSpot VM所采用的热点探测方式是基于计数器的热点探测。

采用基于计数器的热点探测，HotSpot VM将会为每一个方法都建立2个不同类型的计数器，分别为方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter）。

- 方法调用计数器用于统计方法的调用次数
- 回边计数器则用于统计循环体执行的循环次数



还会有热度衰减

热度衰减

- 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绝对次数，而是一个相对的执行频率，即一段时间之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即时编译器编译，那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热度的衰减（Counter Decay），而这段时间就称为此方法统计的半衰周期（Counter Half Life Time）。
- 进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数 `-XX:-UseCounterDecay` 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。
- 另外，可以使用 `-XX:CounterHalfLifeTime` 参数设置半衰周期的时间，单位是秒。

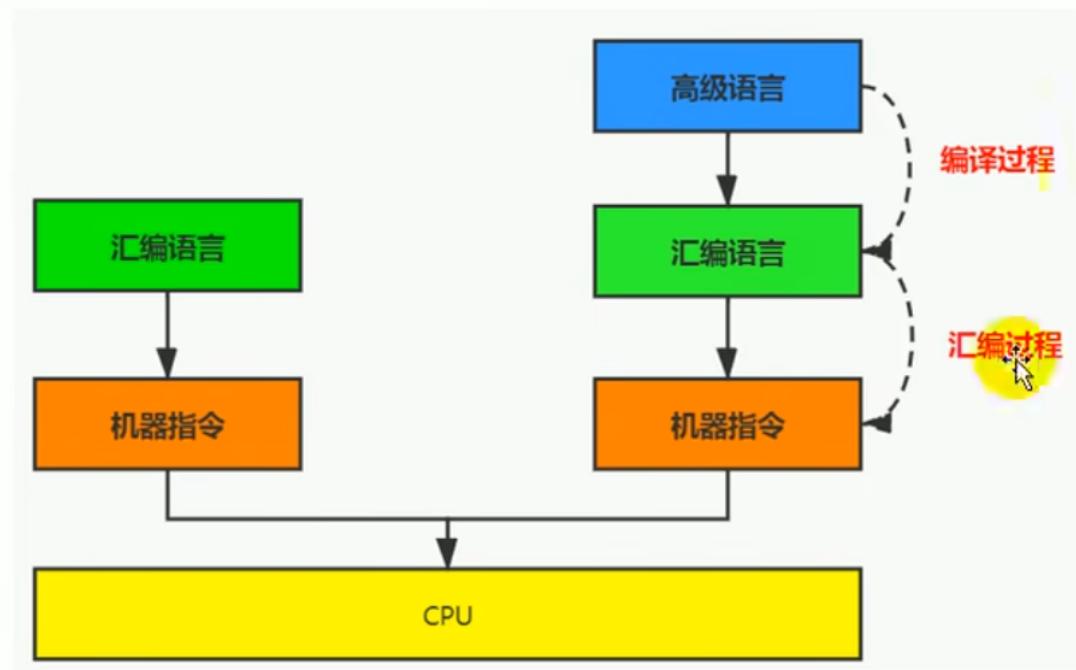
方法区会存即时编译器编译的代码缓存

《深入理解Java虚拟机》书中对方法区（Method Area）存储内容描述如下：

它用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等。



机器码--指令--汇编语言--高级语言



string

- String: 字符串，使用一对""引起来表示。
 - `String s1 = "atguigu"; //字面量的定义方式`
 - `String s2 = new String("hello");`
- String 声明为 final 的，不可被继承
- String 实现了 Serializable 接口：表示字符串是支持序列化的。
实现了 Comparable 接口：表示 String 可以比较大小
- String 在 jdk8 及以前内部定义了 `final char[] value` 用于存储字符串数据。jdk9 时改为 `byte[]`

byte 是一字节

char 是二字节

汉字需要两字节，大部分需要一字节，所以后面改 byte，汉字就用两个 byte 存

字符串常量池，是一个固定大小的 hashtable，默认长度 1009

- 字符串常量池中是不会存储相同内容的字符串的。

- String 的 String Pool 是一个固定大小的 Hashtable，默认值大小长度是 1009。如果放进 String Pool 的 String 非常多，就会造成 Hash 冲突严重，从而导致链表会很长，而链表长了后直接会造成的影响就是当调用 `String.intern()` 时性能会大幅下降。
- 使用 `-XX:StringTableSize` 可设置 StringTable 的长度
- 在 jdk6 中 StringTable 是固定的，就是 1009 的长度，所以如果常量池中的字符串过多就会导致效率下降很快。StringTableSize 设置没有要求
- 在 jdk7 中，StringTable 的长度默认值是 60013，1009 是可设置的最小值。

为什么字符串常量池要从方法区移到堆空间（1.8）

因为方法区回收效率低，但是开发中有大量字符串被创建，所以为了及时回收就放到堆。

静态变量放在堆空间

地址问题

1. 常量与常量的拼接结果在常量池，原理是编译期优化
2. 常量池中不会存在相同内容的常量。
3. 只要其中有一个是变量，结果就在堆中。变量拼接的原理是 `StringBuilder`
4. 如果拼接的结果调用 `intern()` 方法，则主动将常量池中还没有的字符串对象放入池中，并返回此对象地址。jdk7/8 如果堆有对象，常量池的地址是堆的，而不是自己新创建一个字符串对象放入池中

```
String s = new String( original: "1");
```

```

s.intern(); //调用此方法之前，字符串常量池中已经存在了"1"
String s2 = "1";
System.out.println(s == s2);
//jdk6: false jdk7/8: false

String s3 = new String( original:"1") + new String( original: "1");
//执行完上一行代码以后，字符串常量池中，是否存在"11"呢？答案：不存在!!因为stringbuilder的
toString常量池不会有
s3.intern();
//jdk7/8如果堆有对象，常量池的地址是堆的，而不是自己新创建一个字符串对象放入池中
String s4="11";
//s4变量记录的地址：使用的是上一行代码代码行时，在常量池中生成的"11"的地址，它的地址是用堆的，而
不是新生成的
System.out.println(s3 == s4);
//jdk6: false jdk7/8: true

```

```

String s1 ="javaEE";
String s2 = "hadoop";
String s3 ="javaEEhadoop";
String s4 ="javaEE"+ "hadoop"; //编译期优化
//如拼接符号的前后出现了变量，则相当于在堆空间中new String()，具体的内容为拼接的结果：
javaEEhadoop
String s5 = s1 +"hadoop";

String s6 ="javaEE" + s2;
String s7 = s1 + s2;

//intern判断字符串常量池中是否存在javaEEhadoop值，如果存在，则返回常量池中javaEEhadoop的地
址；

String s8 = s6.intern();

s3==s4 true

s3 ==s5 false

s3 ==s6 false

s3 ==s7 false

s5 ==s6 false

s5 ==s7 false

s6 ==s7 false

s3 ==s8 true

```

+

```

string s1 ="a";
string s2 ="b";
String s3 = "ab";
/***

```

```

如下的s1 + s2 的行节:
stringBuilder s = new stringBuilder();
s.append("a")
s.append( "b")
s.toString()--> 类似于 new string("ab")
*/
String s4 = s1 + s2;

//字符串拼接操作不一定使用的是stringBuilder!
//如果拼接符号左右两边都是字符串常量或常量引用则仍然使用编译期优化，即stringBuilder的方式。
final String s1 = "a";
final string s2 ="b";
String s3 = "ab";
String s4 = s1 + s2;
System.out.println(s3 == s4); //true

```

//练习:

@Test

```

public void test5(){
    String s1 = "javaEEhadoop";
    String s2 = "javaEE";
    String s3 = s2 + "hadoop";
    System.out.println(s1 == s3); //false

    final String s4 = "javaEE"; //s4: 常量
    String s5 = s4 + "hadoop";
    System.out.println(s1 == s5); //true
}


```

对象的个数

思考：

```

* new string("a") + new string("b")呢？
对象1: new stringBuilder()
对象2: new string("a")
对象3: 常量池中的"a"
对象4: new string("b")
对象5: 常量池中的"b"
深入剖析: stringBuilder的toString():
对象6: new string("ab")
强调一下, toString()的调用, 在字符串常量池中, 没有生成"ab"

```

stringbuilder的tostring

```
@Override  
public String toString() {  
    // Create a copy, don't share the array  
    return new String(value, offset: 0, count);  
}
```

构造器不同

重复的数据用intern

大的网站平台，需要内存中存储大量的字符串。比如社交网站，很多人都存储：北京市、海淀区等信息。这时候如果字符串都调用 intern()方法，就会明显降低内存的大小。

垃圾

什么是垃圾？

垃圾是指运行程序中没有任何指针指向的对象。

为什么要了解自动内存管理？

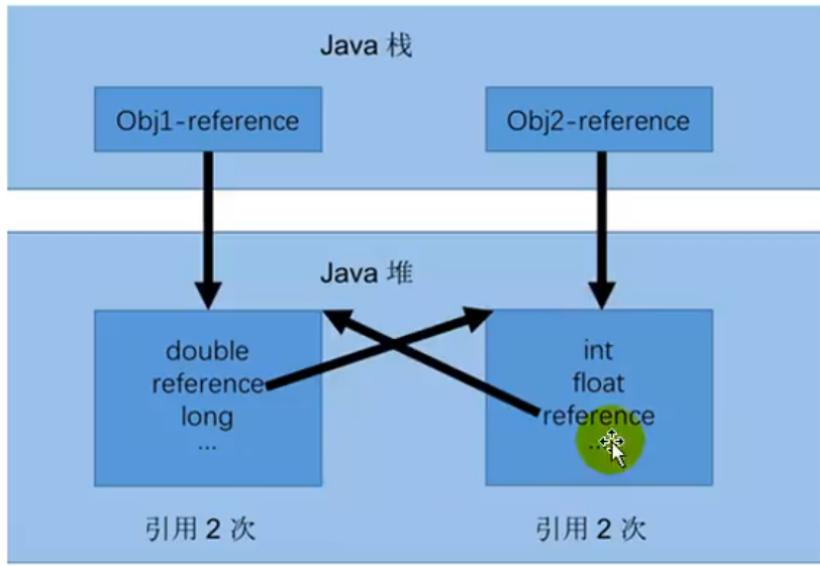
自动内存管理就像黑匣子，过度依赖会导致内存泄漏，内存溢出

垃圾回收的地方？

堆空间是垃圾回收的主要阵地

引用计数器

- 引用计数算法 (Reference Counting) 比较简单，对每个对象保存一个整型的引用计数器属性。用于记录对象被引用的情况。
- 对于一个对象A，只要有任何一个对象引用了A，则A的引用计数器就加1；当引用失效时，引用计数器就减1。只要对象A的引用计数器的值为0，即表示对象A不可能再被使用，可进行回收。
- 优点：实现简单，垃圾对象便于辨识；判定效率高，回收没有延迟性。
- 缺点：
 - 它需要单独的字段存储计数器，这样的做法增加了存储空间的开销。
 - 每次赋值都需要更新计数器，伴随着加法和减法操作，这增加了时间开销。
 - 引用计数器有一个严重的问题，即无法处理循环引用的情况。这是一条致命缺陷，导致在Java的垃圾回收器中没有使用这类算法。



如果不小心直接把
Obj1-reference 和
Obj2-reference 置
null。则在 Java 堆当
中的两块内存依然保持着
互相引用，无法回收。

Java并没有选择引用计数，是因为其存在一个基本的难题，也就是很难处理循环引用关系。

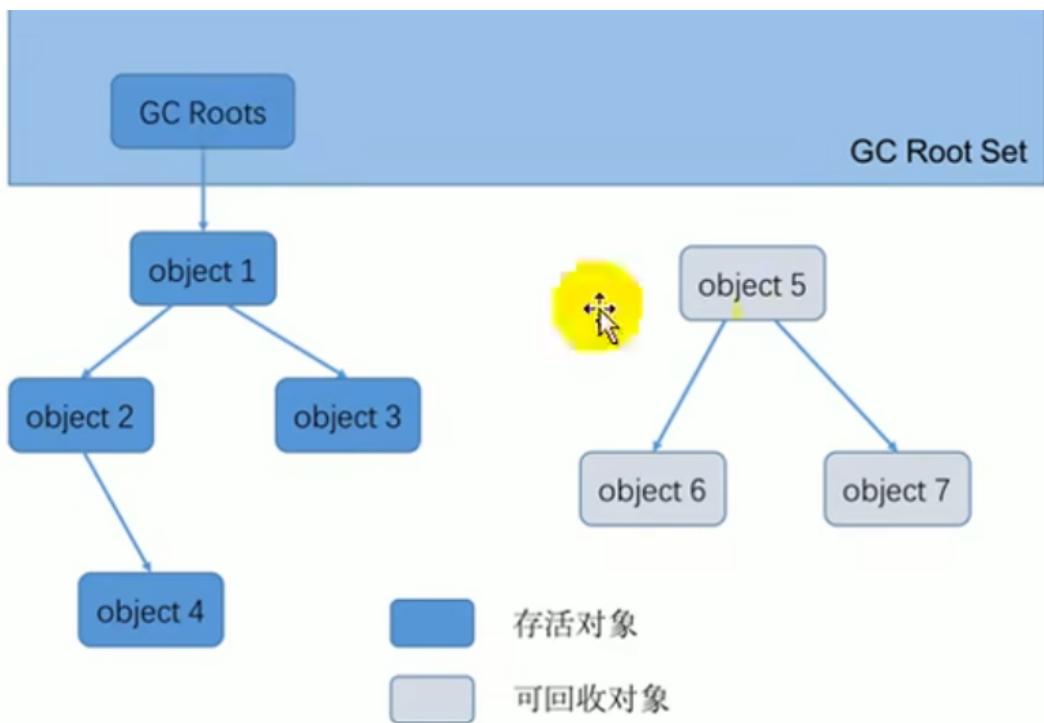
Python如何解决循环引用？

- 手动解除：很好理解，就是在合适的时机，解除引用关系。
- 使用弱引用weakref，weakref是Python提供的标准库，旨在解决循环引用。

可达性分析/根搜索算法/追踪性垃圾回收

基本思路：

- 可达性分析算法是以根对象集合(GC Roots)为起始点，按照从上至下的方式**搜索被根对象集合所连接的目标对象是否可达**。
- 使用可达性分析算法后，内存中的存活对象都会被根对象集合直接或间接连接着，搜索所走过的路径称为**引用链(Reference Chain)**
- 如果目标对象没有任何引用链相连，则是**不可达的**，就意味着该对象已经死亡，可以标记为垃圾对象。
- 在可达性分析算法中，只有能够被根对象集合直接或者间接连接的对象才是存活对象。



对象的finalize方法

垃圾回收，不要主动去调，垃圾回收前只调用一次，只可以复活一次！！！

- 在`finalize()`时可能会导致对象复活。
- `finalize()`方法的执行时间是没有保障的，它完全由GC线程决定，极端情况下，若不发生GC，则`finalize()`方法将没有执行机会。
- 一个糟糕的`finalize()`会严重影响GC的性能。|
- 如果从所有的根节点都无法访问到某个对象，说明对象已经不再使用了。一般来说，此对象需要被回收。但事实上，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段。**一个无法触及的对象有可能在某一个条件下“复活”自己**，如果这样，那么对它的回收就是不合理的，为此，定义虚拟机中的对象可能的三种状态。如下：
 - **可触及的：**从根节点开始，可以到达这个对象。
 - **可复活的：**对象的所有引用都被释放，但是对象有可能在`finalize()`中复活。
 - **不可触及的：****对象的`finalize()`被调用，并且没有复活，那么就会进入不可触及状态。** 不可触及的对象不可能被复活，因为`finalize()`只会被调用一次。
- 以上3种状态中，是由于`finalize()`方法的存在，进行的区分。只有在对象不可触及时才可以被回收。

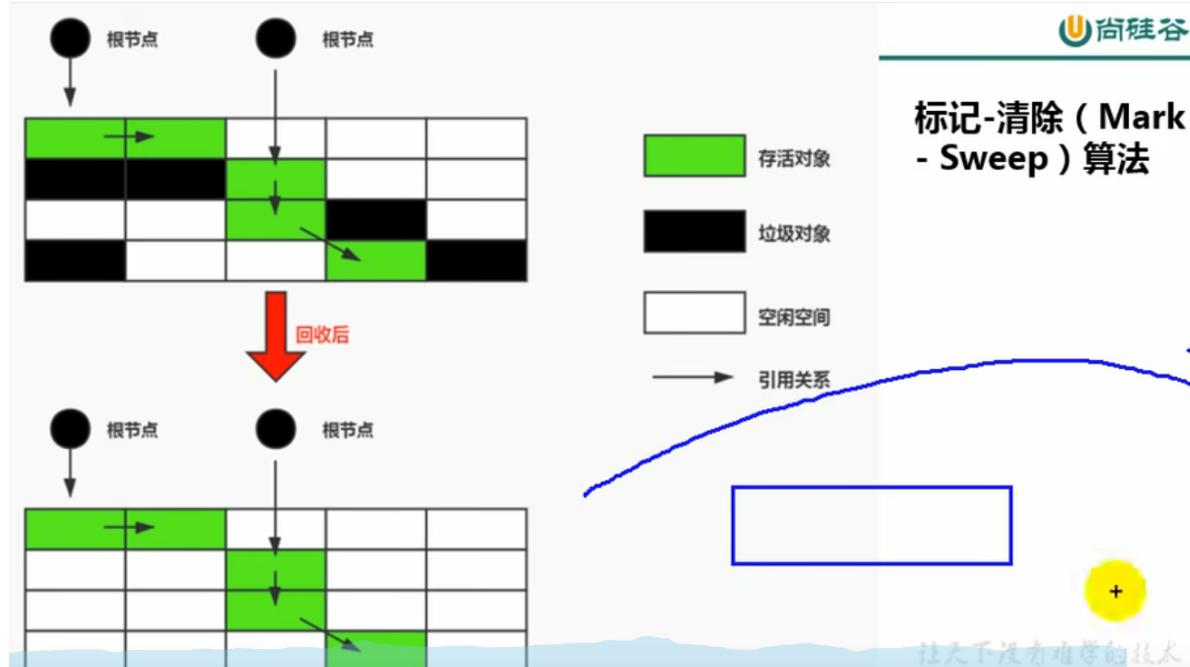
垃圾清除阶段

标记清除算法

执行过程:

当堆中的有效内存空间 (available memory) 被耗尽的时候，就会停止整个程序（也被称为stop the world），然后进行两项工作，第一项则是标记，第二项则是清除。

- **标记:** Collector从引用根节点开始遍历，标记所有被引用的对象。一般是在对象的Header中记录为可达对象。
- **清除:** Collector对堆内存从头到尾进行线性的遍历，如果发现某个对象在其Header中没有标记为可达对象，则将其回收。



• 缺点

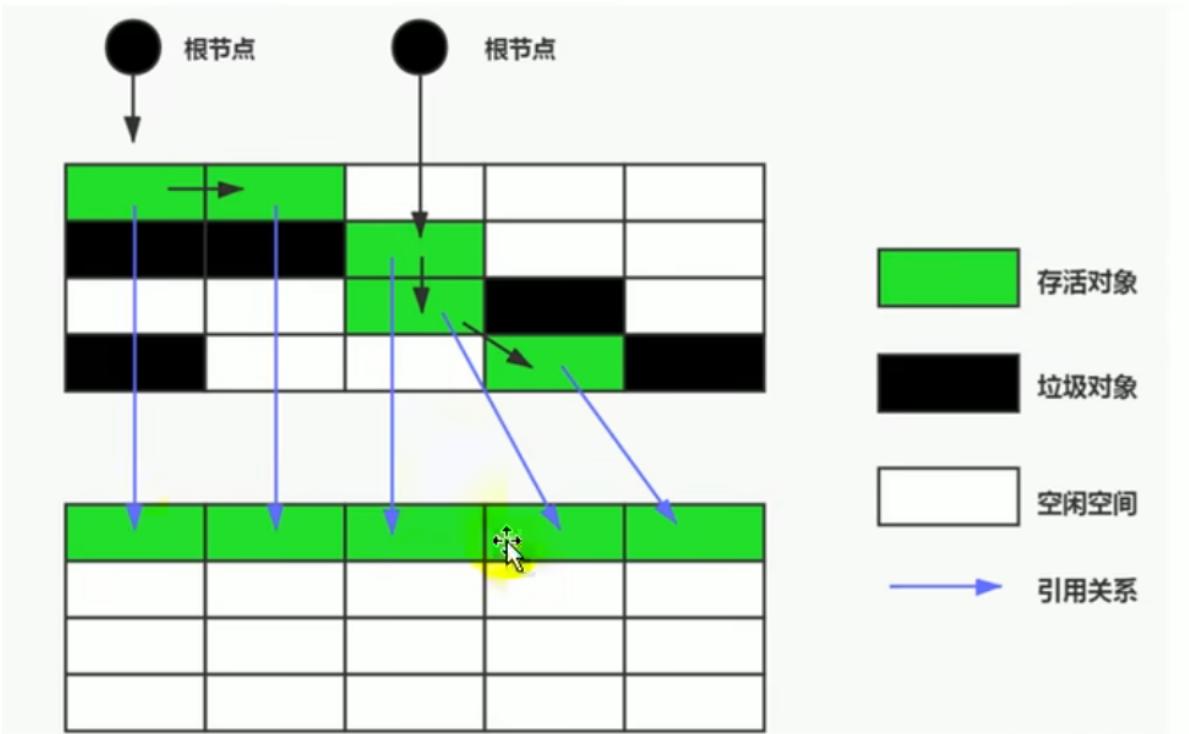
- 效率不算高
- 在进行GC的时候，需要停止整个应用程序，导致用户体验差
- 这种方式清理出来的空闲内存是不连续的，产生内存碎片。需要维护一个空闲列表

• 注意：何为清除？

- 这里所谓的清除并不是真的置空，而是把需要清除的对象地址保存在空闲的地址列表里。下次有新对象需要加载时，判断垃圾的位置空间是否够，如果够，就存放。

复制算法

from和to区就是用这种算法



优点:

- 没有标记和清除过程，实现简单，运行高效
- 复制过去以后保证空间的连续性，不会出现“碎片”问题。

缺点:

- 此算法的缺点也是很明显的，就是需要两倍的内存空间。
- 对于G1这种分拆成为大量region的GC，复制而不是移动，意味着GC需要维护region之间对象引用关系，不管是内存占用或者时间开销也不小。

特别的:

- 如果系统中的垃圾对象很多，复制算法需要复制的存活对象数量并不会太大，或者说非常低才行。

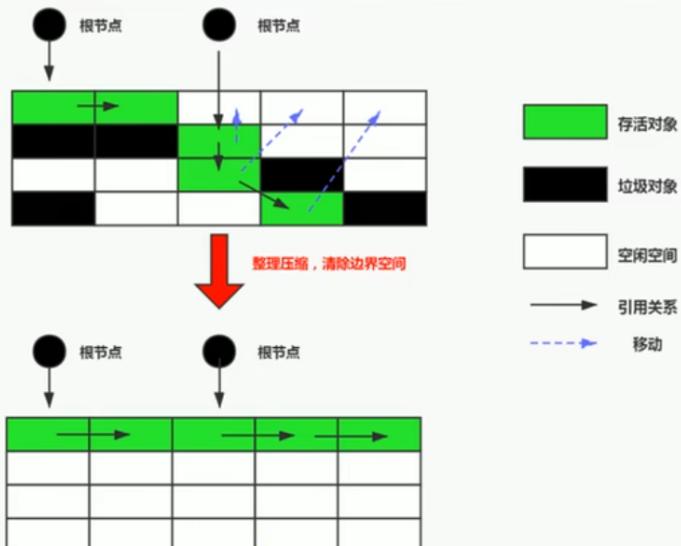
标记压缩算法

执行过程:

第一阶段和标记-清除算法一样，从根节点开始标记所有被引用对象

第二阶段将所有的存活对象压缩到内存的一端，按顺序排放。

之后，清理边界外所有的空间。



优点:

- 消除了标记-清除算法当中，内存区域分散的缺点，我们需要给新对象分配内存时，JVM只需要持有一个内存的起始地址即可。
- 消除了复制算法当中，内存减半的高额代价。

缺点:

- 从效率上来说，标记-整理算法要低于复制算法。
- 移动对象的同时，如果对象被其他对象引用，则还需要调整引用的地址。
- 移动过程中，需要全程暂停用户应用程序。即：STW

分代收集算法

年轻代由于回收频繁，对象小用的是复制算法

老年代由于回收不频繁，对象大用的是标记清除或者标记压缩混合实现

增量收集算法

分区算法将整个堆空间分成连续不同小区间一点一点回收

间接性垃圾回收，减少系统的停顿时间

System.gc()

无法保证对垃圾收集器的调用，只是提醒

内存溢出是因为即使垃圾回收还是没有空闲内存。

当然，也不是在任何情况下垃圾收集器都会被触发的

➤ 比如，我们去分配一个超大对象，类似一个超大数组超过堆的最大值，JVM可以判断出垃圾收集并不能解决这个问题，所以直接抛出`OutOfMemoryError`。

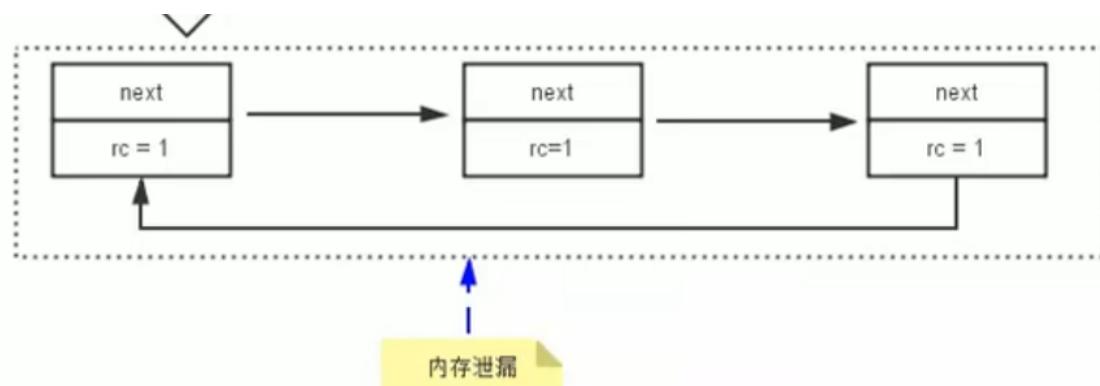
内存泄漏

需要close的资源又close会导致内存泄漏，单例模式的对象。

对象不再被程序用到但GC又回收不了

生命周期很长导致OOM是宽泛意义上的内存泄漏

注意！！！java没有用引用计数器



STW

Stop-the-World，简称STW，指的是GC事件发生过程中，会产生应用程序的停顿。停顿产生时整个应用程序线程都会被暂停，没有任何响应，有点像卡死的感觉，这个停顿称为STW。I

- 可达性分析算法中枚举根节点（GC Roots）会导致所有Java执行线程停顿。
 - ✓ 分析工作必须在一个能确保一致性的快照中进行
 - ✓ 一致性指整个分析期间整个执行系统看起来像被冻结在某个时间点上
 - ✓ 如果出现分析过程中对象引用关系还在不断变化，则分析结果的准确性无法保证

被STW中断的应用程序线程会在完成GC之后恢复，频繁中断会让用户感觉像是网速不快造成电影卡带一样，所以我们需要减少STW的发生。

垃圾回收并发或者并行

安全点，安全区域

程序执行时并非在所有地方都能停顿下来开始 GC，只有在特定的位置才能停顿下来开始GC，这些位置称为“安全点（Safepoint）”。

Safe Point 的选择很重要，如果太少可能导致GC等待的时间太长，如果太频繁可能导致运行时的性能问题。大部分指令的执行时间都非常短暂，通常会根据“是否具有让程序长时间执行的特征”为标准。比如：选择一些执行时间较长的指令作为**Safe Point**，如方法调用、循环跳转和异常跳转等。

如何在GC发生时，检查所有线程都跑到最近的安全点停顿下来呢？

- 抢先式中断：（目前没有虚拟机采用了）

首先中断所有线程。如果还有线程不在安全点，就恢复线程，让线程跑到安全点。

- 主动式中断：

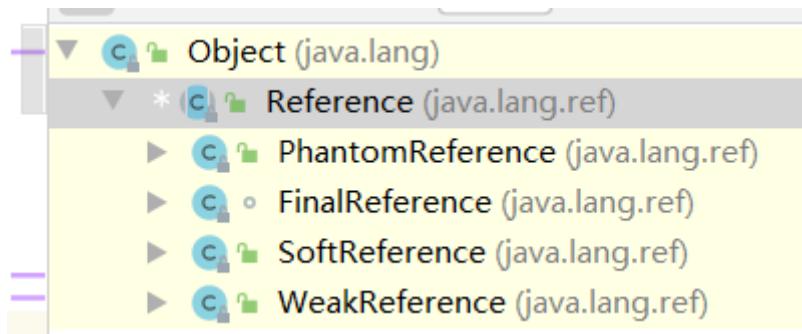
设置一个中断标志，各个线程运行到Safe Point的时候主动轮询这个标志，如果中断标志为真，则将自己进行中断挂起。

Safepoint 机制保证了程序执行时，在不太长的时间内就会遇到可进入 GC 的 Safepoint。但是，程序“不执行”的时候呢？例如线程处于 Sleep 状态或 Blocked 状态，这时候线程无法响应 JVM 的中断请求，“走”到安全点去中断挂起，JVM 也不太可能等待线程被唤醒。对于这种情况，就需要安全区域（Safe Region）来解决。I

安全区域是指在一段代码片段中，对象的引用关系不会发生变化，在这个区域中的任何位置开始GC都是安全的。我们也可以把 Safe Region 看做是被扩展了的 Safepoint。

引用

强软弱虚



引用是指有指针指向

强引用不回收，我们默认就是用强引用

软引用内存不足就回收

```
Object o = new Object();
SoftReference<Object> objectSoftReference = new SoftReference<>(o);
//销毁强引用
objectSoftReference = null;
//引用软引用
Object o1 = objectSoftReference.get();
```

弱引用垃圾收集就回收

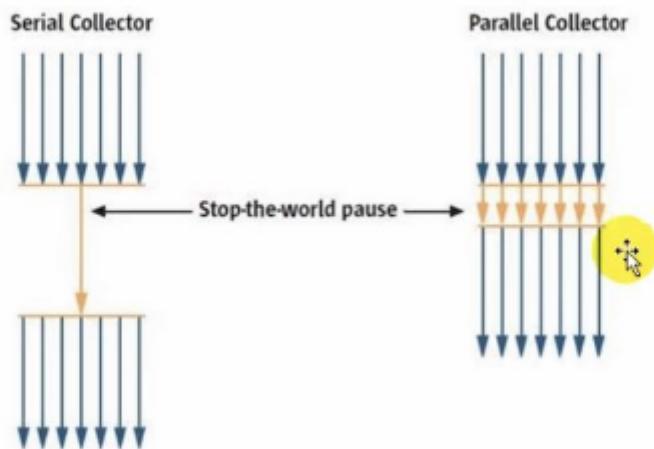
虚引用，它的get也获取不到对象，创建时候必须提供一个引用队列，如果发现它是虚引用，会将虚引用加入引用队列

```
Object o = new Object();
ReferenceQueue<Object> queue = new ReferenceQueue<>();
PhantomReference<Object> objectPhantomReference = new PhantomReference<>(o,
queue);
o = null;
System.out.println( objectPhantomReference.get());
System.gc();
Thread.sleep(1000);
Reference<?> poll = queue.poll();
System.out.println(poll);
```

垃圾回收器分类

按线程数分

- 按线程数分，可以分为串行垃圾回收器和并行垃圾回收器。



让天下没有难
的面试

按工作模式分

- 按照**工作模式**分，可以分为并发式垃圾回收器和独占式垃圾回收器。
 - 并发式垃圾回收器与应用程序线程交替工作，以尽可能减少应用程序的停顿时间。
 - 独占式垃圾回收器(Stop the world)一旦运行，就停止应用程序中的所有用户线程，直到垃圾回收过程完全结束。



让天下没有难
的面试

按碎片处理方式

按工作的内存区间分

- 按**碎片处理方式**分，可分为压缩式垃圾回收器和非压缩式垃圾回收器。
 - 压缩式垃圾回收器会在回收完成后，对存活对象进行压缩整理，消除回收后的碎片。
 - 非压缩式的垃圾回收器不进行这步操作。
- 按**工作的内存区间**分，又可分为年轻代垃圾回收器和老年代垃圾回收器。

如何评估GC性能

重点！！

吞吐量：运行用户代码的时间占总运行时间的比例 a

暂停时间：执行垃圾暂停工作的时间

内存占用：堆内存的大小

垃圾收集开销：吞吐量的补数 1-a

收集频率：垃圾收集频率

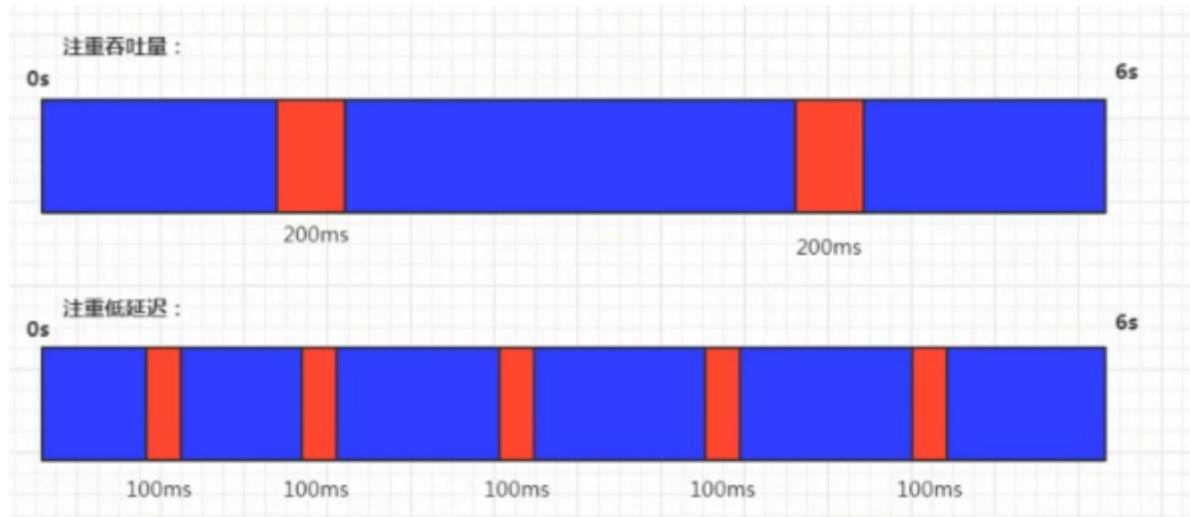
快速：对象诞生到回收的时间

注重吞吐量

回收频率少但时间长

注重低延迟（线程切换需花费时间）

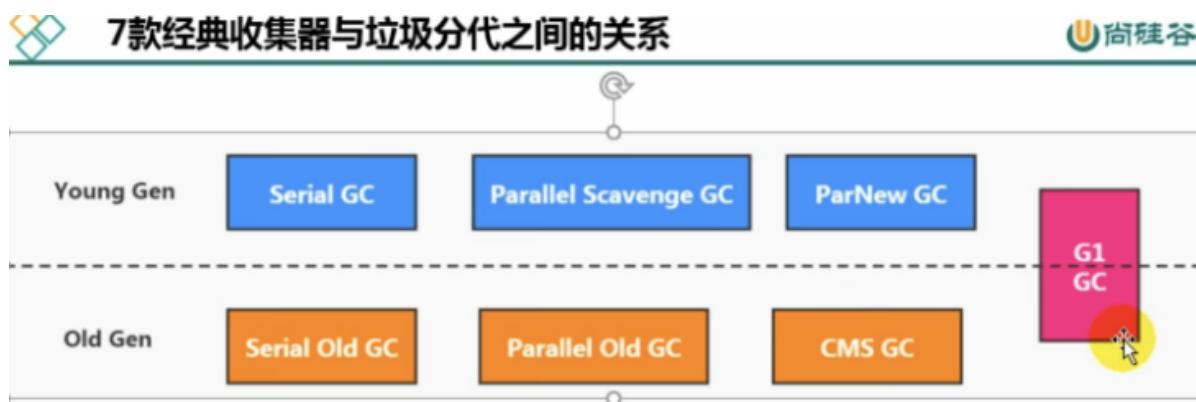
回收频率高但时间短



垃圾回收只能针对低延迟或者吞吐量

垃圾回收器和分区的关系

针对java使用场景不同，提供不同垃圾回收器。



- 新生代收集器: **Serial**、**ParNew**、**Parallel Scavenge**;
- 老年代收集器: **Serial Old**、**Parallel Old**、**CMS**;
- 整堆收集器: **G1**;

serial

针对新生代，收集器采用复制算法，串行回收,STW

serial old

针对老年代，采用标记压缩算法，串行回收,STW

以上串行垃圾回收器现在是不用了，针对单核

以下针对多核

parallel scavenge

针对新生代，收集器采用复制算法，并行回收，STW，特点在于针对吞吐量优先，适合交互不强的场景。

jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

有参数可以设置停顿时间，会导致收集器调整java堆大小或其他参数

设置吞吐量

parallel old

针对老年代，采用标记压缩算法，并行回收,STW

parallel New

ParNew垃圾收集器其实跟Parallel垃圾收集器很类似，区别主要在于它可以和CMS收集器配合使用

针对新生代，收集器采用复制算法，并行回收，STW

CMS

针对老年代，尽可能缩短停顿时间（一定会STW,缩短时间而已），采用标记-清除算法，第一次实现垃圾收集和用户线程同时工作。STW，若干次GC后进行碎片整理

弊端：产生内存碎片

在并发阶段，虽然不会导致用户线程停顿，但是会占用CPU资源而导致引用程序变慢

由于垃圾回收阶段没有中断用户线程，如果堆内存到达一定阈值，则突然临时启用serial old收集老年代



初始标记：需要STW，标记出GC ROOTS能直接关联到的对象

并发标记：遍历对象

重新标记：需要STW修正，为了安全管理

并发清除：清除对象

总结

最小化使用内存使用serial gc

最大化使用吞吐量使用parallel gc

最小化gc中断或停顿时间使用CMS GC

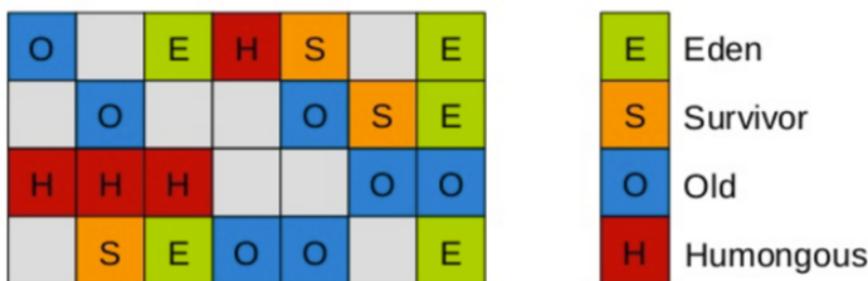
垃圾收集器	分类	作用位置	使用算法	特点	适用场景
Serial	串行运行	作用于新生代	复制算法	响应速度优先	适用于单CPU环境下的client模式
ParNew	并行运行	作用于新生代	复制算法	响应速度优先	多CPU环境Server模式下与CMS配合使用
Parallel	并行运行	作用于新生代	复制算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
Serial Old	串行运行	作用于老年代	标记-压缩算法	响应速度优先	适用于单CPU环境下的Client模式
Parallel Old	并行运行	作用于老年代	标记-压缩算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
CMS	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

G1/garbage first

标记-整理算法收集后不会产生内存碎片

jdk9的默认垃圾回收器，延迟可控的情况下提高高的吞吐量

1、分区Region:



G1 垃圾收集器将堆内存^Q划分为若干个 Region，每个 Region 分区只能是一种角色，Eden区、S区、老年代O区的其中一个，空白区域代表的是未分配的内存，最后还有个特殊的区域H区（Humongous），专门用于存放巨型对象，如果一个对象的大小超过Region容量的50%以上，G1 就认为这是个巨型对象。在其他垃圾收集器中，这些巨型对象默认会被分配在老年代，但如果它是一个短期存活的巨型对象，放入老年代就会对垃圾收集器造成负面影响，触发老年代频繁GC。为了解决这个问题，G1划分了一个H区专门存放巨型对象，如果一个H区装不下巨型对象，那么G1会寻找连续的H分区来存储，如果寻找不到连续的H区的话，就不得不启动 Full GC 了。

以region为基本单位进行标记压缩

参数

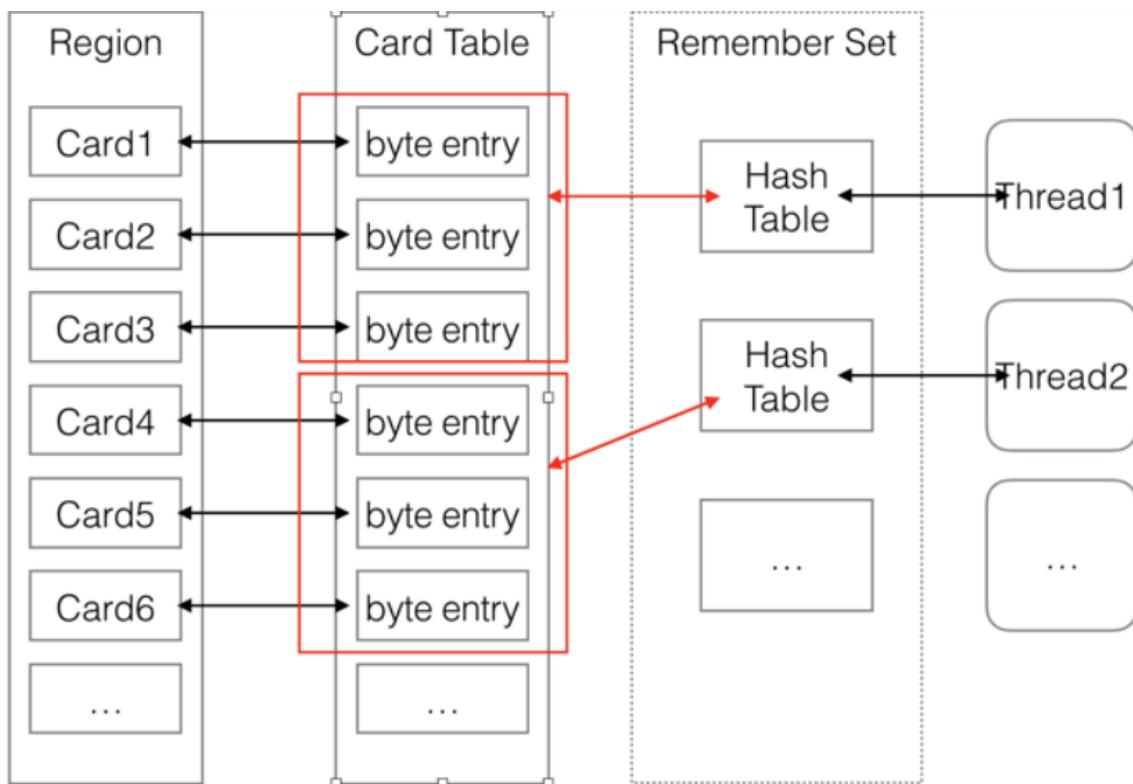
- **-XX: +UseG1GC** 手动指定使用G1收集器执行内存回收任务。
- **-XX: G1HeapRegionSize** 设置每个Region的大小。值是2的幂，范围是1MB到32MB之间，目标是根据最小的Java堆大小划分出约2048个区域。默认是堆内存的1/2000。
- **-XX: MaxGCPauseMillis** 设置期望达到的最大GC停顿时间指标 (JVM会尽力实现，但不保证达到)。默认值是200ms
- **-XX: ParallelGCThread** 设置STW时GC线程数的值。最多设置为8
- **-XX: ConcGCThreads** 设置并发标记的线程数。将n设置为并行垃圾回收线程数(ParallelGCThreads)的1/4左右。
- **-XX: InitiatingHeapOccupancyPercent** 设置触发并发GC周期的Java堆占用率阈值。超过此值，就触发GC。默认值是45。

一个region不可能是孤立的，一个region中的对象可能被其他任意region中对象引用，判断对象存活时，是否需要扫描整个java堆才能保证准确？例如回收新手代也不得不同时扫描老年代？

无论是哪种分代收集器，JVM都是使用remember set避免全局扫描

每个region都有自己的对应的remembered set

每次引用类型数据写操作时，都会产生一个write barrier写屏障暂时中断操作，检查写入的引用指向的对象是否和该引用类型数据在不同的region区域。如果不同，通过cardTable把相关引用信息记录到引用指向对象的所在的region对象的remembered set中

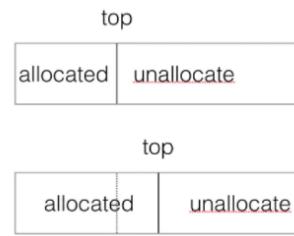


为了避免fullGC!!!!!!

有如下三种回收垃圾模式

前沿知识

每一个分配的 Region 都可以分成两个部分，已分配的和未被分配的，它们之间的界限被称为top。总体上来说，把一个对象分配到 Region内，只需要简单增加top的值。过程如下：



线程本地分配缓冲区 Thread Local allocation buffer (TLab): 为了减少并发冲突损耗的同步时间，G1为每个应用线程和GC线程分配了一个本地分配缓冲区TLAB，分配对象内存时，就在这个 buffer 内分配，线程之间不再需要进行任何的同步，提高GC效率。但是当线程耗尽了自己的Buffer之后，需要申请新的Buffer。这个时候依然会带来并发的问题。G1回收器采用的是CAS (Compart And Swap) 操作。

Eden区中分配：对TLAB空间中无法分配的对象，JVM会尝试在Eden空间中进行分配。如果Eden空间无法容纳该对象，就只能在老年代中进行分配空间。

Humongous区分配：巨型对象会独占一个、或多个连续分区，其中第一个分区被标记为开始巨型(StartsHumongous)，相邻连续分区被标记为连续巨型(ContinuesHumongous)。由于无法享受 TLab 带来的优化，并且确定一片连续的内存空间需要扫描整堆，因此确定巨型对象开始位置的成本非常高，如果可以，应用程序应避免生成巨型对象。

Young GC

Mixed GC

当老年代占用空间超过整堆比 IHOP 阈值 -XX:InitiatingHeapOccupancyPercent(默认45%)时，G1就会启动一次混合垃圾回收Mixed GC，Mixed GC不仅进行正常的新生代垃圾收集，同时也回收部分后台扫描线程标记的老年代分区，注意是一部分！！！

步骤主要分为两步：全局并发标记

拷贝存活对象（等待触发一次年轻代收集，在这次STW中，G1将开始整理混合收集周期）

只有当 Mixed GC 来不及回收old region，也就是说在需要分配老年代的对象时，但发现没有足够的空间，这个时候就会触发一次 Full GC

前端编译器javac

不涉及编译优化

idea用javac编译器，全量编译

eclipse用的是ecj，增量式编译器，ctrl+s编译一部分

Integer

在[-128, 127]从缓冲池取或者就new一个，所以如果Integer 128 == Integer 128 就是false

Integer 5 == int 5 是关于自动装箱拆箱的问题

Class文件的结构

魔数

确定这个文件是否为一个能被虚拟机接受的Class文件。

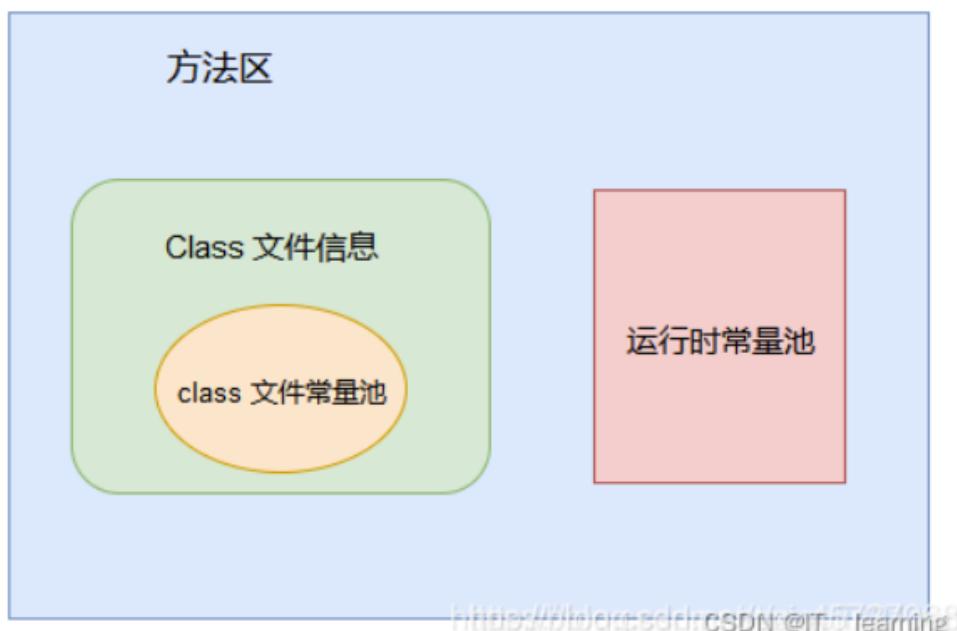
class文件版本

确定class文件的版本

常量池

常量池是资源仓库，运行时常量池是当 Class 文件被加载到内存后，Java 虚拟机会将 Class 文件常量池里的内容转移到运行时常量池里，**能在加载的时候就可以将符号引用转变为直接引用**

(1) 方法区的 Class 文件信息，Class 常量池和运行时常量池的三者关系：



访问标志

识别一些类或者接口层次的访问信息

类索引，父类索引，接口索引集合

类索引用于确定这个类的全限定名

字段表集合

方法表集合

属性表集合

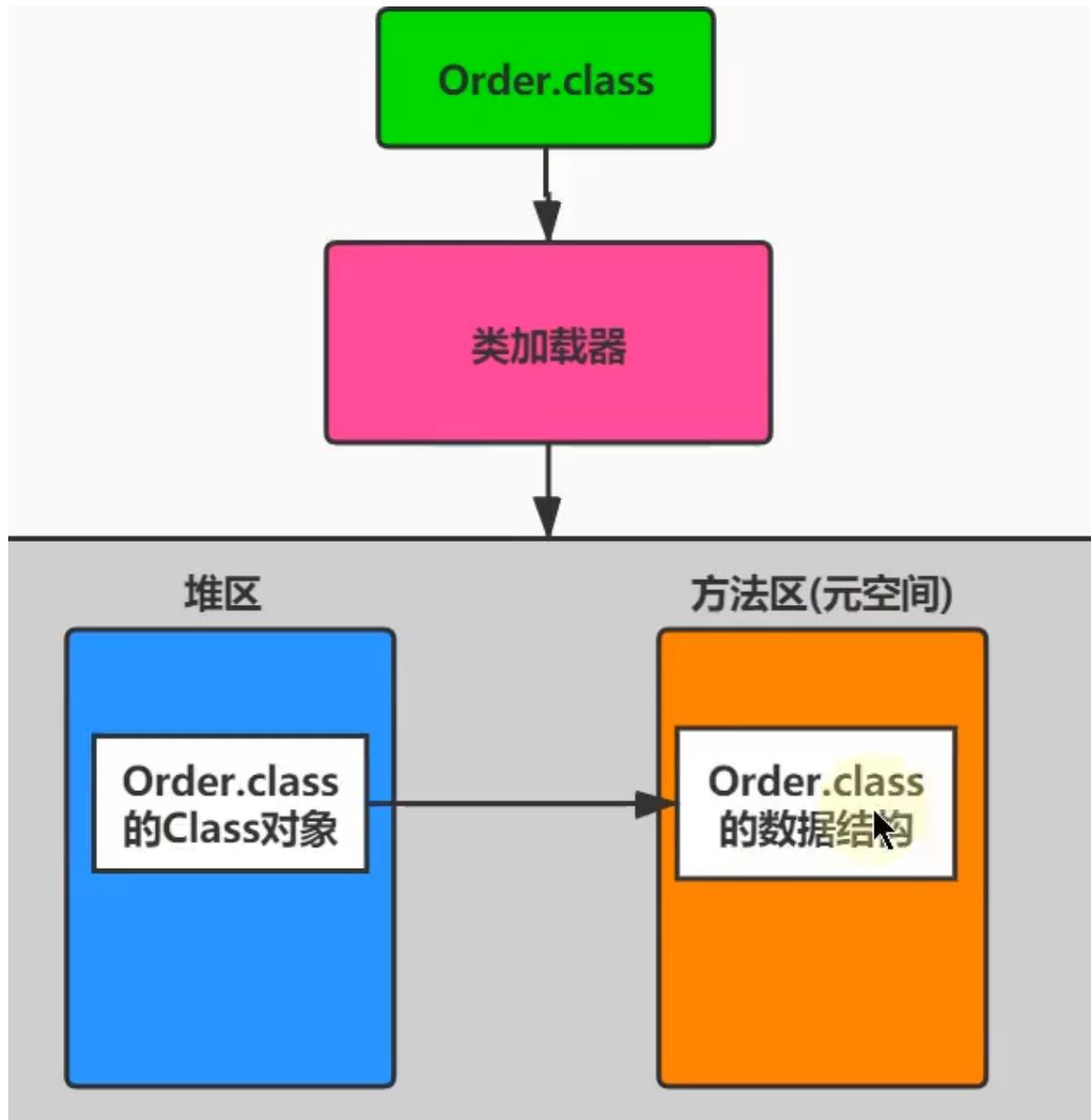
全限定名是全类名的.换成/

javap可以反编译

类加载的过程

加载

通过类的全名找到class文件基于类的二进制数据流生成的类结构回存储在方法区，然后在堆创建一个class对象。



链接

验证

验证：验证和加载是同时进行，只有验证成功才会加载。验证字节码是否合法和语义规范。

无法判断符号引用是否正确

准备

给类的静态变量分配内存赋默认值。

如果是final的静态常量和string常量会显式赋值

解析

将类、接口、字段和方法的符号引用转为直接引用

初始化

其实就是执行代码

先加载该类的父类

对静态变量显示赋值以及执行代码块

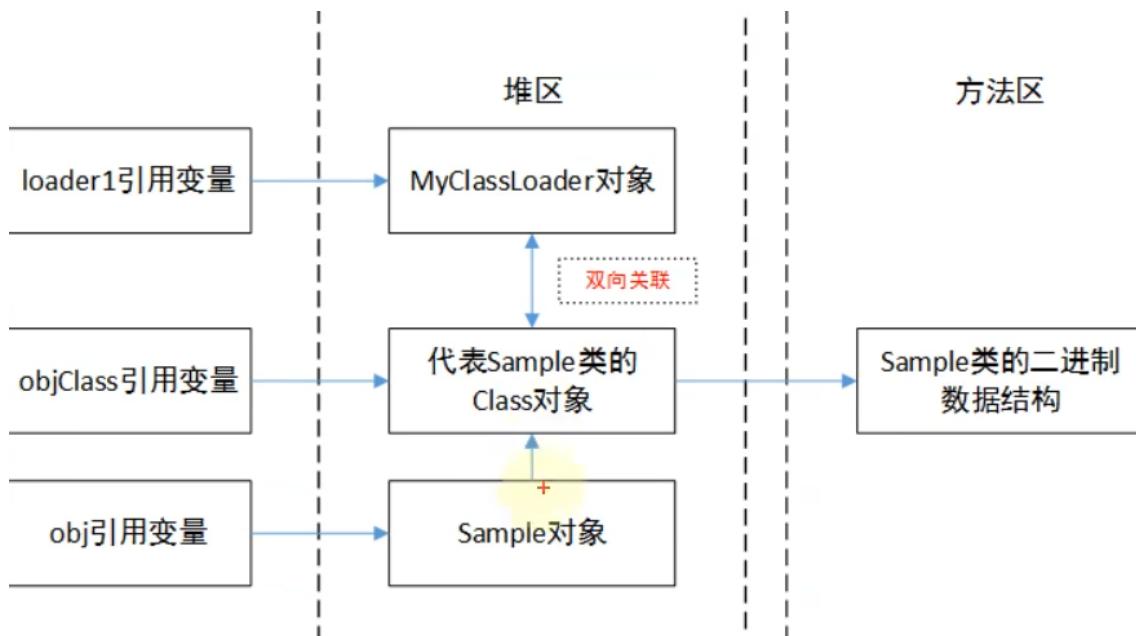
为什么类的初始化是安全的？

因为初始化其实是调用该类的clinit函数，这是带锁线程安全的

类的被动使用不会调用类加载的初始化

例如类的加载器加载一个类就是被动使用

反射为什么找的到class



比较两个类

比较两个类相等得在同一个类加载器才能比较

每个类加载器都有自己命名空间，命名空间由它及父加载器组成

双亲委派模型

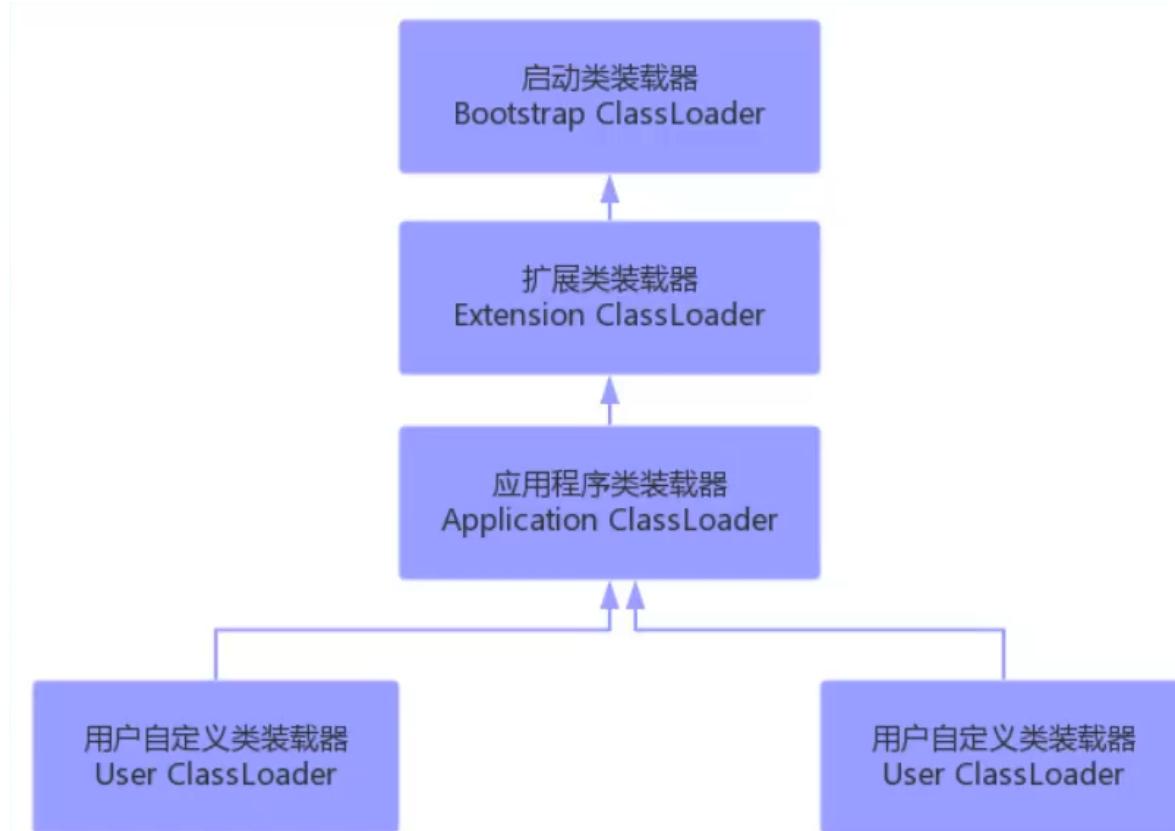
父类加载不了再由子类加载，

可以防止重复加载

防止核心api被篡改，就例如你自己写了一个java.lang.String，加载时候会报错禁止你加载同全类名的类
不是继承关系是包含，也就是聚合模式

```
class
{
    ClassLoader parent; //父类加载器
    public ClassLoader(ClassLoader parent){
        this.parent = parent;
    }
}
```

启动类是由c++写的，获取不到，拓展类加载器，系统类加载器，是由引导类加载器加载



自定义类加载器：loadclass是双亲委派机制，findclass是最终加载类，所以重写findclass就好了，在findclass将数据流通过defindclass方法写到jvm

弊端：下层可以用上层，上层用不了下层

为什么要有findclass：使得程序员可以自定义类加载

怎么打破双亲委派模型：有线程上下文加载器，一般指向系统类加载器

了解类加载器有什么用户：可以对代码进行热替换（同一个类加载器内，不能重复加载同一个类）

为什么要自定义类加载器：隔离加载类，防止源码泄漏

为什么要调优

防止OOM和减少Full GC

jps查看进程

jstat查看jvm统计信息

jinfo查看参数，可实时修改信息

可视化工具

jconsole

jvisual

jprofiler

浅堆和深堆

浅堆是对象本身的内存，不包括引用对象大小

对象的保留集是指对象被垃圾回收后，可以释放的所有对象集合，包括本身。

深堆是对象保留集所有浅堆的大小之和。注意保留集中的的可释放！！！

对象实际大小，对象和对象的指向对象。

内存泄漏

被使用着即指针指向，但不需要用到。

或者宽泛意义上对象生命周期特别长

内存泄漏最终会导致内存溢出

内存泄漏的情况

静态集合存放局部变量

单例模式持有外部对象引用

各种连接，如数据库连接，IO流忘记关闭了

变量不合理的作用域

存放到哈希表后，改变哈希值，就是类的hashCode方法有属性参与哈希化，改变属性则会改变哈希值，用哈希表就获取不到，这个时候删除哈希表的对象就没有用，因为删除会用到hashCode方法，获取的哈希值与原本不同，所以不建议重写hashCode方法。

出栈的时候不是抛出对象而是指针下移

jvm参数

-X比较稳定参数

-Xms20m, -Xmx20m, -Xmn20m, -Xss128k.....

-XX不稳定的参数 用的最多：JVM调优

-XX:+PrintGCDetails, -XX:-UseParallelGC, -XX:+PrintGCTimeStamps.....

<https://www.cnblogs.com/edwardlauxh/archive/2010/04/25/1918603.html>

OOM

-XX:+HeapDumpOnOutOfMemoryError 表示在内存出现QOM的时候，把Heap转存(Dump)到文件以便后续分析

-XX:+HeapDumpBeforeFullGC 表示在出现FullGC之前，生成Heap转储文件

-XX:HeapDumpPath= 指定heap转存文件的存储路径

Xx:OnOutOfMemoryError 指定一个可行性程序或者脚本的路径，当发生OOM的时候，去执行这个脚本

堆比例

新生代跟老年代比例为1比2

伊甸园区和服务区比例是8比2，但是伊甸园区和服务器会有一个自动分配

MemoryMXBean

从java层面来了解堆的情况

FullGC

内存严重不足会执行fullgc

如老年代空间不足，方法区空间不足，显示调用System.gc ()

堆内存总容量

9/10新生代+老年代<初始化的内存大小