**史上最全 Java 面试题：JVM 篇**

**社群交流的面试真题**

**面试真题1：元空间会产生内存溢出么？在什么情况下会产生内存溢出？**

具体问题：元空间会产生内存溢出么？在什么情况下会产生内存溢出？。

java8 及以后的版本使用Metaspace来代替永久代，Metaspace是方法区在HotSpot中的实现，它与持久代最大区别在于，Metaspace并不在虚拟机内存中而是使用本地内存也就是在JDK8中,classe metadata(the virtual machines internal presentation of Java class),被存储在叫做Metaspace的native memory.  
永久代（java 8 后被元空间Metaspace取代了）存放了以下信息：

* 虚拟机加载的类信息
* 常量池
* 静态变量
* 即时编译后的代码

**出现问题原因**

错误的主要原因, 是加载到内存中的 class 数量太多或者体积太大。

**解决办法**

增加 Metaspace 的大小

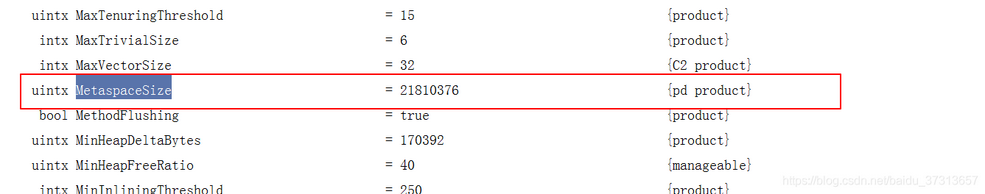
-XX:MaxMetaspaceSize=512m

**代码演示**

模拟Metaspace空间溢出，我们不断生成类往元空间灌，类占据的空间是会超过Metaspace指定的空间大小的

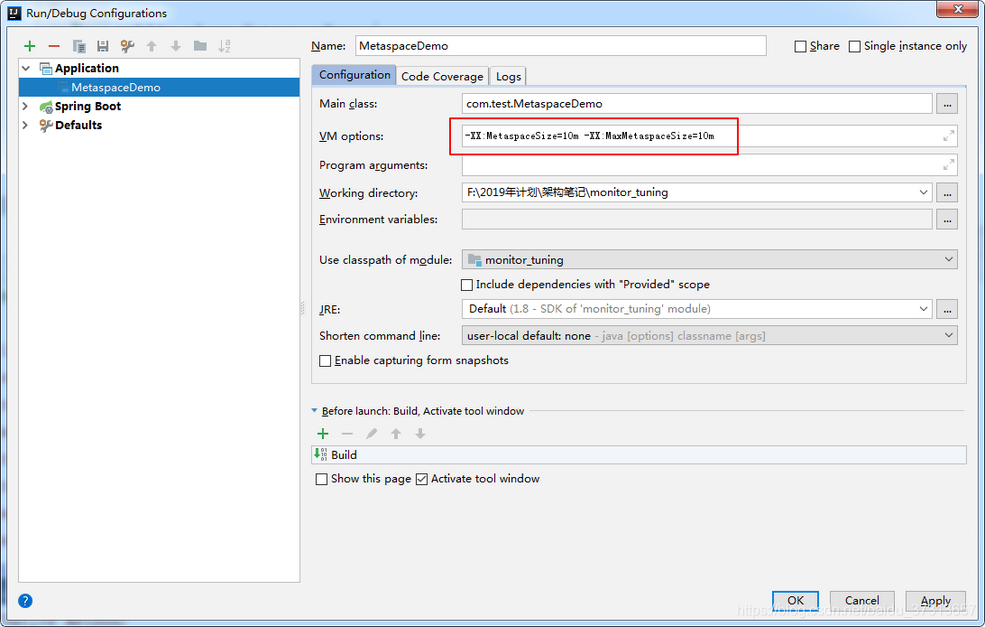
查看元空间大小

java -XX:+PrintFlagsInitial

[](https://img-blog.csdnimg.cn/20210720142156868.png)

设置配置 这里设置10m方便演示效果

-XX:MetaspaceSize=10m -XX:MaxMetaspaceSize=10m

[](https://img-blog.csdnimg.cn/20210720144120153.png)

编写代码

import org.springframework.cglib.proxy.Enhancer;

import org.springframework.cglib.proxy.MethodInterceptor;

import org.springframework.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class MetaspaceDemo {

static class OOM{}

public static void main(String[] args) {

int i = 0;*//模拟计数多少次以后发生异常*

try {

while (true){

i++;

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(OOM.class);

enhancer.setUseCache(false);

enhancer.setCallback(new MethodInterceptor() {

@Override

public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

return methodProxy.invokeSuper(o,args);

}

});

enhancer.create();

}

} catch (Throwable e) {

System.out.println("=================多少次后发生异常："+i);

e.printStackTrace();

}

}

}

运行结果：  
[](https://img-blog.csdnimg.cn/20210720144133550.png)

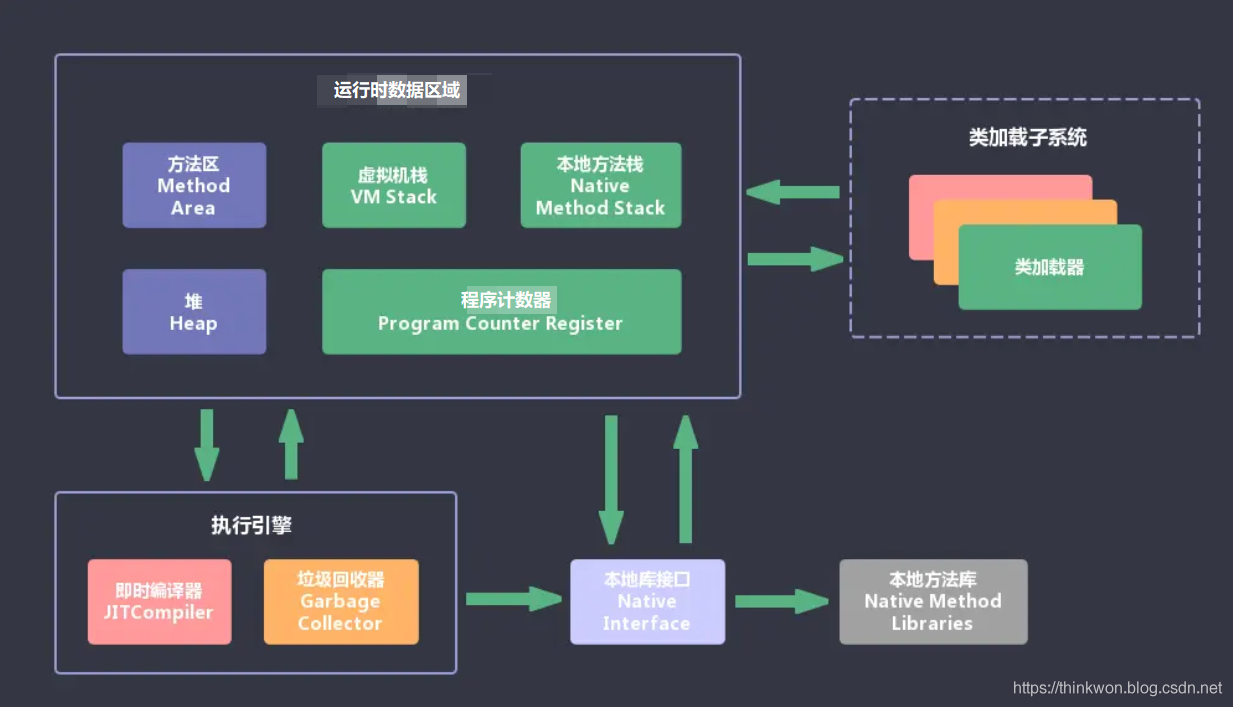
更多面试真题，请来 疯狂创客圈社群的 微信群交流，具体请参考 [**Java 高并发 发烧友社群：疯狂创客圈（总入口）**](https://www.cnblogs.com/crazymakercircle/p/9904544.html)

**正式内容1：Java内存区域**

**解释 Java 堆空间及 GC？**

当通过 Java 命令启动 Java 进程的时候，会为它分配内存。内存的一部分用于创建堆空间，当程序中创建对象的时候，就从对空间中分配内存。GC 是 JVM 内部的一个进程，回收无效对象的内存用于将来的分配。

**说一下 JVM 的主要组成部分及其作用？**

[](https://img-blog.csdnimg.cn/20200103213149526.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

JVM包含两个子系统和两个组件，两个子系统为Class loader(类装载)、Execution engine(执行引擎)；两个组件为Runtime data area(运行时数据区)、Native Interface(本地接口)。

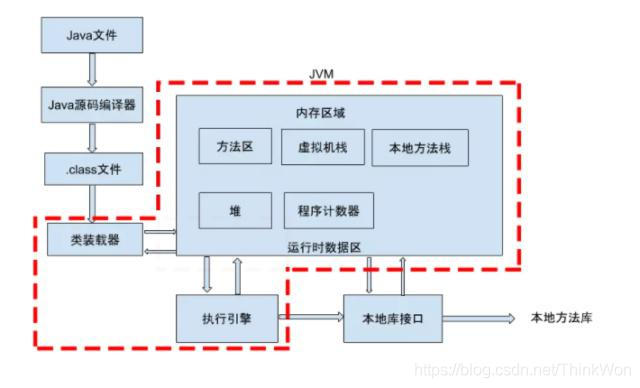
* Class loader(类装载)：根据给定的全限定名类名(如：java.lang.Object)来装载class文件到Runtime data area中的method area。
* Execution engine（执行引擎）：执行classes中的指令。
* Native Interface(本地接口)：与native libraries交互，是其它编程语言交互的接口。
* Runtime data area(运行时数据区域)：这就是我们常说的JVM的内存。

**作用** ：首先通过编译器把 Java 代码转换成字节码，类加载器（ClassLoader）再把字节码加载到内存中，将其放在运行时数据区（Runtime data area）的方法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

**下面是Java程序运行机制详细说明**

Java程序运行机制步骤

* 首先利用IDE集成开发工具编写Java源代码，源文件的后缀为.java；
* 再利用编译器(javac命令)将源代码编译成字节码文件，字节码文件的后缀名为.class；
* 运行字节码的工作是由解释器(java命令)来完成的。

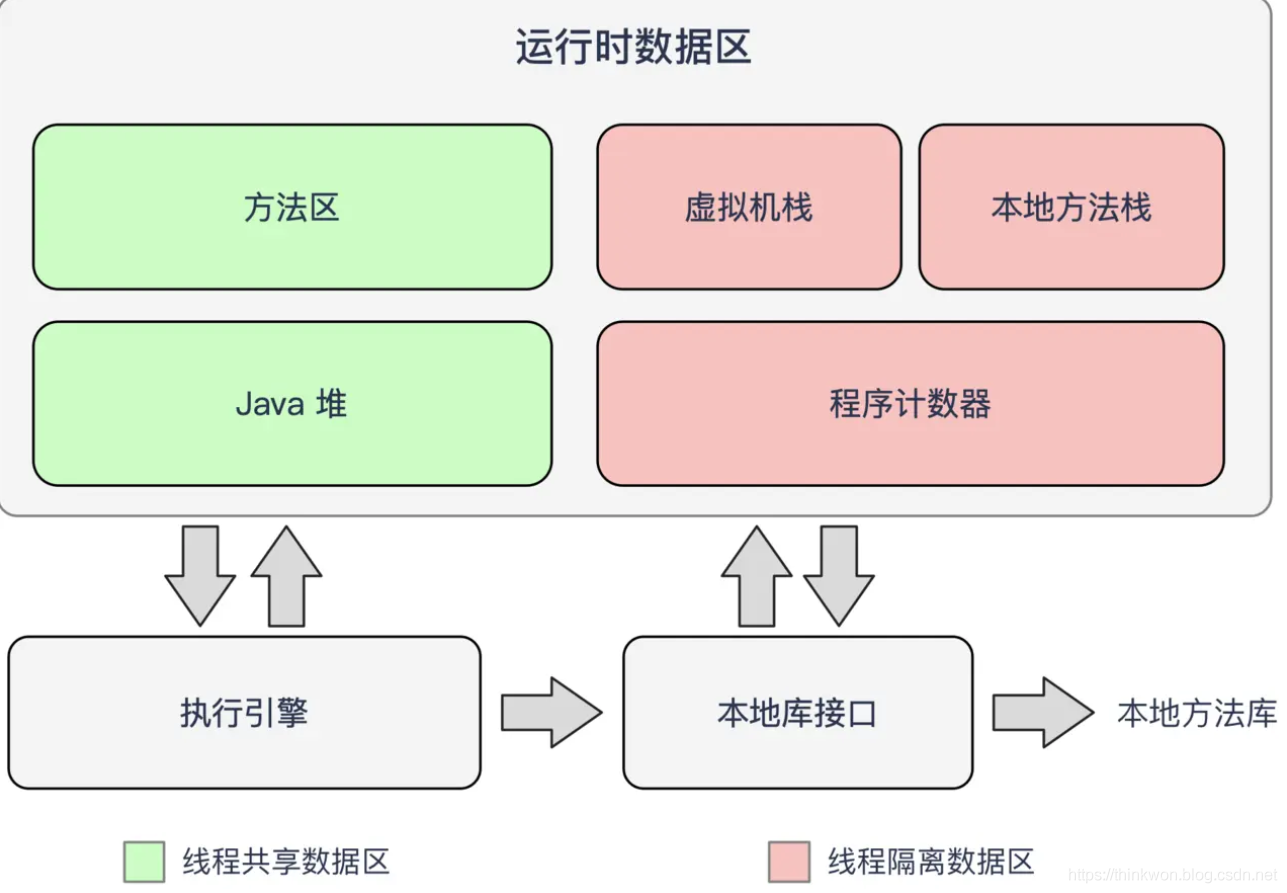
[](https://img-blog.csdnimg.cn/2020031416414486.jpeg?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

从上图可以看，java文件通过编译器变成了.class文件，接下来类加载器又将这些.class文件加载到JVM中。  
其实可以一句话来解释：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。

**说一下 JVM 运行时数据区？ 或者：说一下JVM内存模型？**

**思路：** 给面试官画一下JVM内存模型图，并描述每个模块的定义，作用，以及可能会存在的问题，如栈溢出等。

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存区域划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有些区域随着虚拟机进程的启动而存在，有些区域则是依赖线程的启动和结束而建立和销毁。Java 虚拟机所管理的内存被划分为如下几个区域：

[](https://img-blog.csdnimg.cn/20200103213220764.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

* 程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作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；
* Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息；
* 本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的；
* Java 堆（Java Heap）：Java 虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存；
* 方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

**深拷贝和浅拷贝**

浅拷贝（shallowCopy）只是增加了一个指针指向已存在的内存地址，

深拷贝（deepCopy）是增加了一个指针并且申请了一个新的内存，使这个增加的指针指向这个新的内存，

使用深拷贝的情况下，释放内存的时候不会因为出现浅拷贝时释放同一个内存的错误。

浅复制：仅仅是指向被复制的内存地址，如果原地址发生改变，那么浅复制出来的对象也会相应的改变。

深复制：在计算机中开辟一块**新的内存地址**用于存放复制的对象。

**说一下堆栈的区别？**

物理地址

堆的物理地址分配对对象是不连续的。因此性能慢些。在GC的时候也要考虑到不连续的分配，所以有各种算法。比如，标记-消除，复制，标记-压缩，分代（即新生代使用复制算法，老年代使用标记——压缩）

栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。所以性能快。

内存分别

堆因为是不连续的，所以分配的内存是在**运行期**确认的，因此大小不固定。一般堆大小远远大于栈。

栈是连续的，所以分配的内存大小要在**编译期**就确认，大小是固定的。

存放的内容

堆存放的是对象的实例和数组。因此该区更关注的是数据的存储

栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。

PS：

1. 静态变量放在方法区
2. 静态的对象还是放在堆。

程序的可见度

堆对于整个应用程序都是共享、可见的。

栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同。

**Java 中堆和栈有什么区别？**

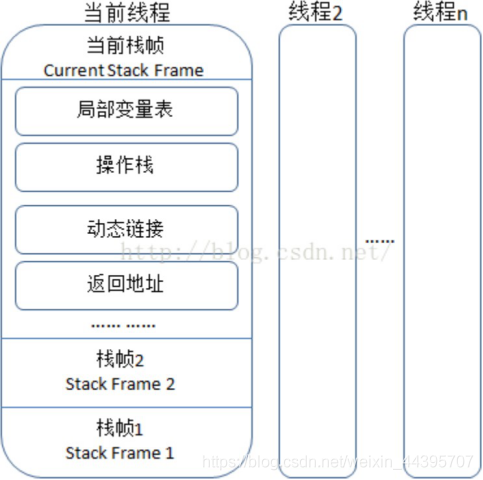
JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。

**队列和栈是什么？有什么区别？**

队列和栈都是被用来预存储数据的。

* 操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。
* 可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。
* 操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列头上（不允许中途离队）。而栈为后进先出（LIFO）,即每次删除（出栈）的总是当前栈中最新的元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。

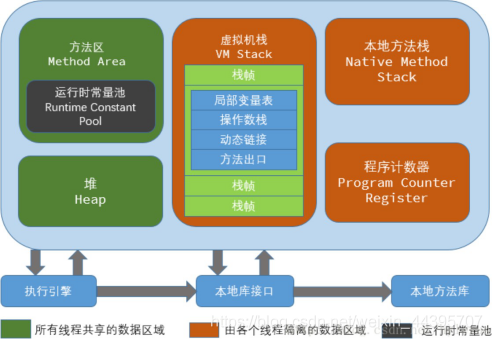
**虚拟机栈(线程私有)**

是描述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（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。 每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。  
栈帧（ Frame）是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态链接(Dynamic Linking)、 方法返回值和异常分派（ Dispatch Exception）。 栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异常）都算作方法结束。  
[](https://img-blog.csdnimg.cn/20200515110720994.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3dlaXhpbl80NDM5NTcwNw==,size_16,color_FFFFFF,t_70)

**程序计数器(线程私有)**

一块较小的内存空间, 是当前线程所执行的字节码的行号指示器，每条线程都要有一个独立的程序计数器，这类内存也称为“线程私有” 的内存。  
正在执行 java 方法的话，计数器记录的是虚拟机字节码指令的地址（当前指令的地址） 。如果还是 Native 方法，则为空。这个内存区域是唯一一个在虚拟机中没有规定任何 OutOfMemoryError 情况的区域。

**什么是直接内存？**

直接内存并不是 JVM 运行时数据区的一部分, 但也会被频繁的使用: 在 JDK 1.4 引入的 NIO 提供了基于 Channel 与 Buﬀer 的 IO 方式, 它可以使用 Native 函数库直接分配堆外内存, 然后使用DirectByteBuﬀer 对象作为这块内存的引用进行操作(详见: Java I/O 扩展), 这样就避免了在 Java堆和 Native 堆中来回复制数据, 因此在一些场景中可以显著提高性能。  
[](https://img-blog.csdnimg.cn/20200515110656779.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3dlaXhpbl80NDM5NTcwNw==,size_16,color_FFFFFF,t_70)

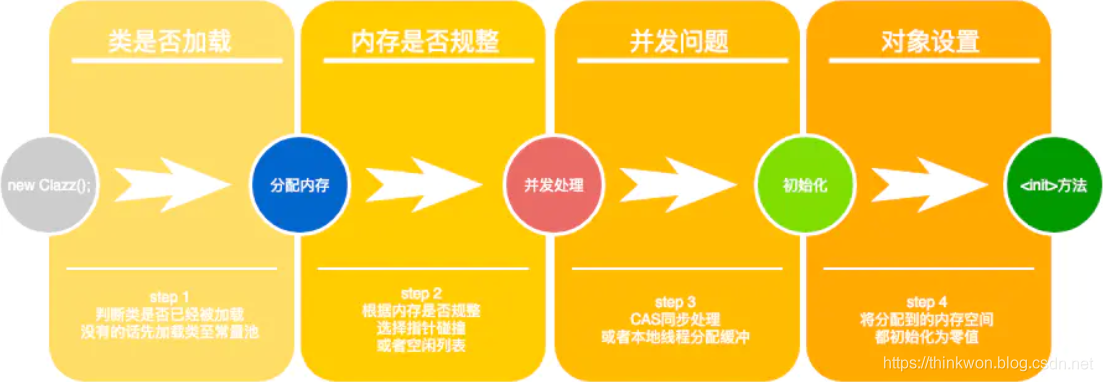
**HotSpot虚拟机对象探秘**

**对象的创建**

说到对象的创建，首先让我们看看 **Java** 中提供的几种对象创建方式：

| **Header** | **解释** |
| --- | --- |
| 使用new关键字 | 调用了构造函数 |
| 使用Class的newInstance方法 | 调用了构造函数 |
| 使用Constructor类的newInstance方法 | 调用了构造函数 |
| 使用clone方法 | 没有调用构造函数 |
| 使用反序列化 | 没有调用构造函数 |

下面是对象创建的主要流程:

[](https://img-blog.csdnimg.cn/20200103213726902.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

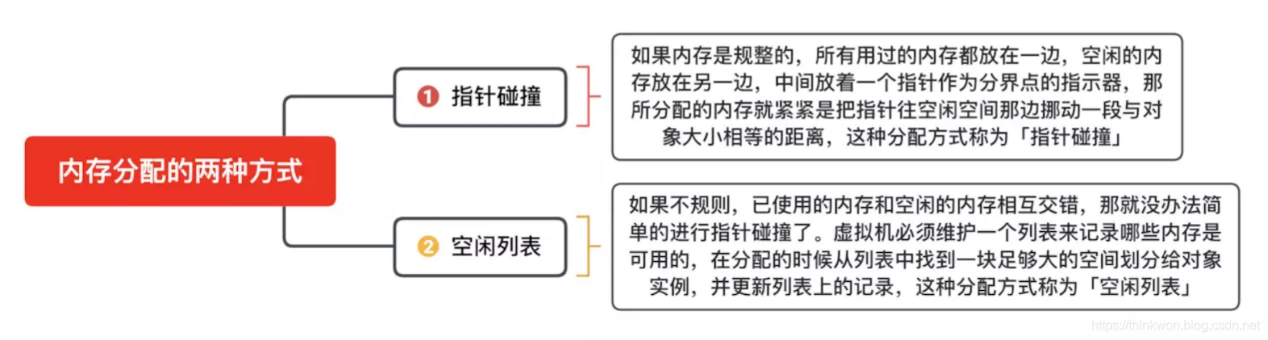
虚拟机遇到一条new指令时，先检查常量池是否已经加载相应的类，如果没有，必须先执行相应的类加载。类加载通过后，接下来分配内存。若Java堆中内存是绝对规整的，使用“指针碰撞“方式分配内存；如果不是规整的，就从空闲列表中分配，叫做”空闲列表“方式。划分内存时还需要考虑一个问题-并发，也有两种方式: CAS同步处理，或者本地线程分配缓冲(Thread Local Allocation Buffer, TLAB)。然后内存空间初始化操作，接着是做一些必要的对象设置(元信息、哈希码…)，最后执行**<init>**方法。

**为对象分配内存**

类加载完成后，接着会在Java堆中划分一块内存分配给对象。内存分配根据Java堆是否规整，有两种方式：

* 指针碰撞：如果Java堆的内存是规整，即所有用过的内存放在一边，而空闲的的放在另一边。分配内存时将位于中间的指针指示器向空闲的内存移动一段与对象大小相等的距离，这样便完成分配内存工作。
* 空闲列表：如果Java堆的内存不是规整的，则需要由虚拟机维护一个列表来记录那些内存是可用的，这样在分配的时候可以从列表中查询到足够大的内存分配给对象，并在分配后更新列表记录。

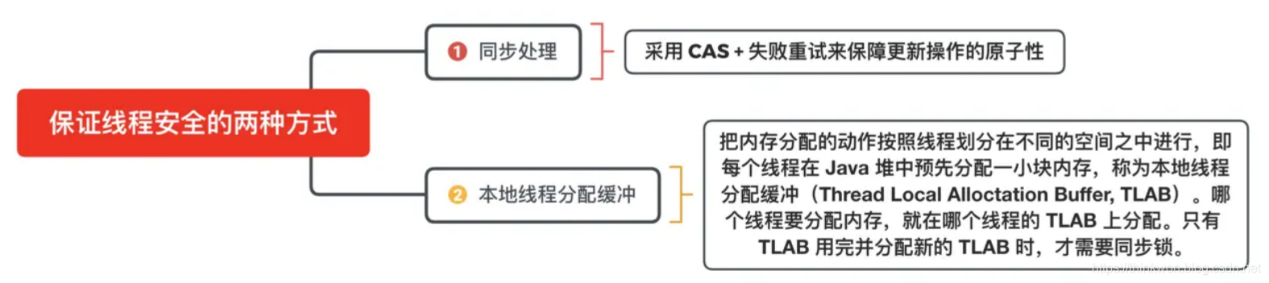
选择哪种分配方式是由 Java 堆是否规整来决定的，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

[](https://img-blog.csdnimg.cn/20200103213812259.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

**处理并发安全问题**

对象的创建在虚拟机中是一个非常频繁的行为，哪怕只是修改一个指针所指向的位置，在并发情况下也是不安全的，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。解决这个问题有两种方案：

* 对分配内存空间的动作进行同步处理（采用 CAS + 失败重试来保障更新操作的原子性）；
* 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（Thread Local Allocation Buffer, TLAB）。哪个线程要分配内存，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。通过-XX:+/-UserTLAB参数来设定虚拟机是否使用TLAB。

[](https://img-blog.csdnimg.cn/20200103213833317.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

**对象的访问定位**

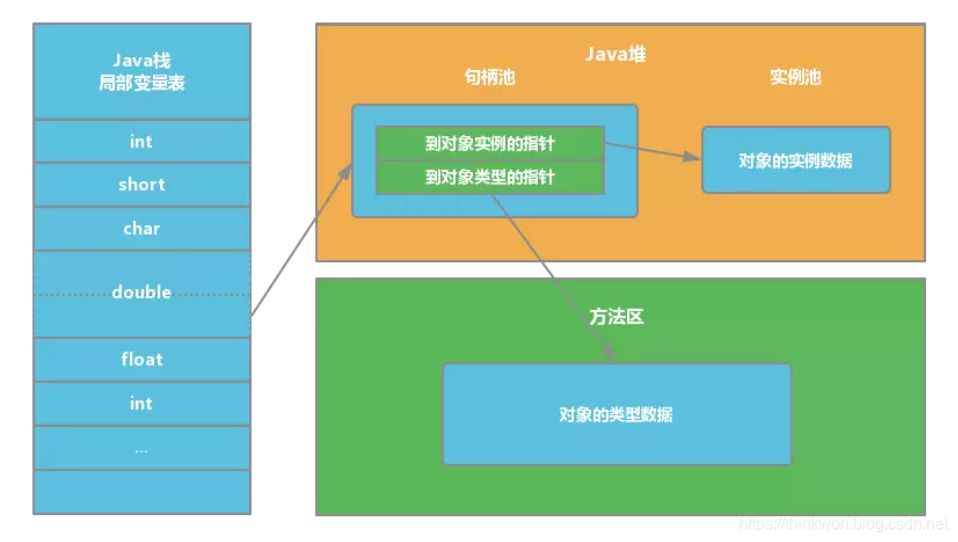
**Java**程序需要通过 **JVM** 栈上的引用访问堆中的具体对象。对象的访问方式取决于 **JVM** 虚拟机的实现。目前主流的访问方式有 **句柄** 和 **直接指针** 两种方式。

**指针：** 指向对象，代表一个对象在内存中的起始地址。

**句柄：** 可以理解为指向指针的指针，维护着对象的指针。句柄不直接指向对象，而是指向对象的指针（句柄不发生变化，指向固定内存地址），再由对象的指针指向对象的真实内存地址。

**句柄访问**

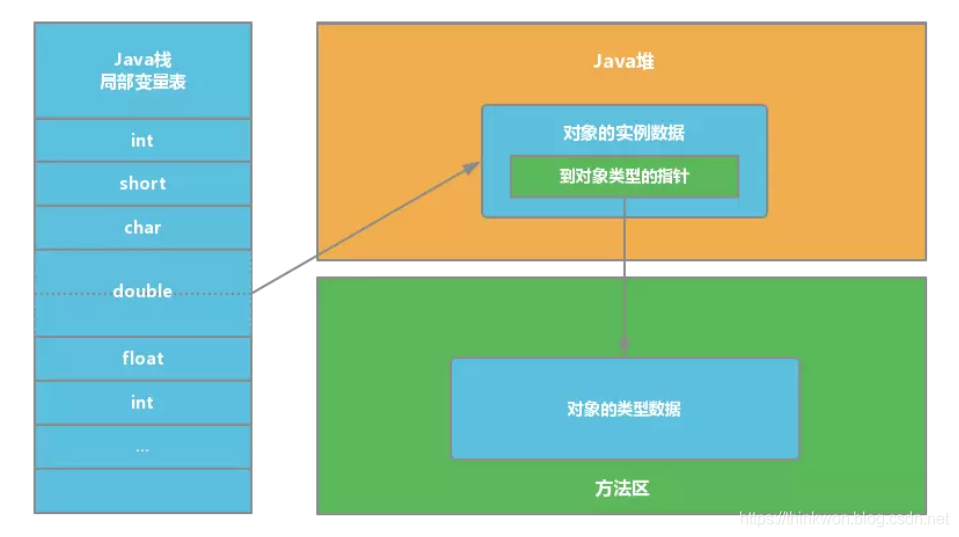
**Java**堆中划分出一块内存来作为**句柄池**，引用中存储对象的**句柄地址**，而句柄中包含了**对象实例数据**与**对象类型数据**各自的**具体地址**信息，具体构造如下图所示：

[](https://img-blog.csdnimg.cn/20200103213926911.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

**优势**：引用中存储的是**稳定**的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变**句柄中**的**实例数据指针**，而**引用**本身不需要修改。

**直接指针**

如果使用**直接指针**访问，**引用** 中存储的直接就是**对象地址**，那么**Java**堆对象内部的布局中就必须考虑如何放置访问**类型数据**的相关信息。

[](https://img-blog.csdnimg.cn/20200103213948956.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

**优势**：速度更**快**，节省了**一次指针定位**的时间开销。由于对象的访问在**Java**中非常频繁，因此这类开销积少成多后也是非常可观的执行成本。HotSpot 中采用的就是这种方式。

**64 位 JVM 中，int 的长度是多数？**

Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位。意思就是说，在 32 位 和 64 位 的 Java 虚拟机中，int 类型的长度是相同的。

**32 位和 64 位的 JVM，int 类型变量的长度是多数？**

32 位和 64 位的 JVM 中，int 类型变量的长度是相同的，都是 32 位或者 4个字节。

**怎样通过 Java 程序来判断 JVM 是 32 位 还是 64位？**

你可以检查某些系统属性如 sun.arch.data.model 或 os.arch 来获取该信息。

**32 位 JVM 和 64 位 JVM 的最大堆内存分别是多数？**

理论上说上 32 位的 JVM 堆内存可以到达 2^32， 即 4GB，但实际上会比这个小很多。不同操作系统之间不同，如 Windows 系统大约 1.5GB，Solaris 大约3GB。64 位 JVM 允许指定最大的堆内存，理论上可以达到 2^64，这是一个非常大的数字，实际上你可以指定堆内存大小到 100GB。甚至有的 JVM，如 Azul，堆内存到 1000G 都是可能的。

**JRE、JDK、JVM 及 JIT 之间有什么不同？**

JRE 代表 Java 运行时（Java run-time），是运行 Java 引用所必须的。

JDK 代表 Java 开发工具（Java development kit），是 Java 程序的开发工具，如 Java编译器，它也包含 JRE。

JVM 代表 Java 虚拟机（Java virtual machine），它的责任是运行 Java 应用。

JIT 代表即时编译（Just In Time compilation），当代码执行的次数超过一定的阈值时，会将 Java 字节码转换为本地代码，如，主要的热点代码会被准换为本地代码，这样有利大幅度提高 Java 应用的性能。

**内存溢出异常**

**Java会存在内存泄漏吗？请简单描述**

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。理论上来说，Java是有GC垃圾回收机制的，也就是说，不再被使用的对象，会被GC自动回收掉，自动从内存中清除。

但是，即使这样，Java也还是存在着内存泄漏的情况，java导致内存泄露的原因很明确：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，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景。

**什么情况下会发生栈内存溢出。**

**思路：** 描述栈定义，再描述为什么会溢出，再说明一下相关配置参数，OK的话可以给面试官手写是一个栈溢出的demo。

**参考答案：**

* 栈是线程私有的，他的生命周期与线程相同，每个方法在执行的时候都会创建一个栈帧，用来存储局部变量表，操作数栈，动态链接，方法出口等信息。局部变量表又包含基本数据类型，对象引用类型
* 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，方法递归调用产生这种结果。
* 如果Java虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时候没有足够的内存去创建对应的虚拟机栈，那么Java虚拟机将抛出一个OutOfMemory 异常。(线程启动过多)
* 参数 -Xss 去调整JVM栈的大小

**垃圾收集器**

**简述Java垃圾回收机制**

在java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

**GC是什么？为什么要GC**

GC 是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存

回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动

回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

**垃圾回收的优点和原理。并考虑2种回收机制**

java语言最显著的特点就是引入了垃圾回收机制，它使java程序员在编写程序时不再考虑内存管理的问题。

由于有这个垃圾回收机制，java中的对象不再有“作用域”的概念，只有引用的对象才有“作用域”。

垃圾回收机制有效的防止了内存泄露，可以有效的使用可使用的内存。

垃圾回收器通常作为一个单独的低级别的线程运行，在不可预知的情况下对内存堆中已经死亡的或很长时间没有用过的对象进行清除和回收。

程序员不能实时的对某个对象或所有对象调用垃圾回收器进行垃圾回收。

垃圾回收有分代复制垃圾回收、标记垃圾回收、增量垃圾回收。

**垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？**

对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。

通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。

可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

**你能保证 GC 执行吗？**

不能，虽然你可以调用 System.gc() 或者 Runtime.gc()，但是没有办法保证 GC的执行。

**Java 中都有哪些引用类型？**

* 强引用：发生 gc 的时候不会被回收。
* 软引用：有用但不是必须的对象，在发生内存溢出之前会被回收。
* 弱引用：有用但不是必须的对象，在下一次GC时会被回收。
* 虚引用（幽灵引用/幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。

**介绍一下强引用、软引用、弱引用、虚引用的区别？**

**思路：** 先说一下四种引用的定义，可以结合代码讲一下，也可以扩展谈到ThreadLocalMap里弱引用用处。

**参考答案：**

1）强引用

我们平时new了一个对象就是强引用，例如 Object obj = new Object();即使在内存不足的情况下，JVM宁愿抛出OutOfMemory错误也不会回收这种对象。

2）软引用

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。

SoftReference<String> softRef=new SoftReference<String>(str); *// 软引用*

**用处：** 软引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建

（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出

如下代码：

Browser prev = new Browser(); *// 获取页面进行浏览*

SoftReference sr = new SoftReference(prev); *// 浏览完毕后置为软引用*

if(sr.get()!=null){

rev = (Browser) sr.get(); *// 还没有被回收器回收，直接获取*

}else{

prev = new Browser(); *// 由于内存吃紧，所以对软引用的对象回收了*

sr = new SoftReference(prev); *// 重新构建*

}

3）弱引用

具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

String str=new String("abc");

WeakReference<String> abcWeakRef = new WeakReference<String>(str);

str=null;

等价于

str = null;

System.gc();

4）虚引用

如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。虚引用主要用来跟踪对象被垃圾回收器回收的活动。

**怎么判断对象是否可以被回收？**

垃圾收集器在做垃圾回收的时候，首先需要判定的就是哪些内存是需要被回收的，哪些对象是「存活」的，是不可以被回收的；哪些对象已经「死掉」了，需要被回收。

一般有两种方法来判断：

* 引用计数器法：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；
* 可达性分析算法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

**在Java中，对象什么时候可以被垃圾回收**

当对象对当前使用这个对象的应用程序变得不可触及的时候，这个对象就可以被回收了。  
垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。

**JVM 运行时堆内存如何分代?**

Java 堆从 GC 的角度还可以细分为: 新生代(Eden 区、 From Survivor 区和 To Survivor 区)和老年代。

参考图1：

[](https://img-blog.csdnimg.cn/20200515110837890.png)

参考图2：

[](https://img-blog.csdnimg.cn/20210203103213658.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2NyYXp5bWFrZXJjaXJjbGU=,size_16,color_FFFFFF,t_70)

从图中可以看出： 堆大小 = 新生代 + 老年代。其中，堆的大小可以通过参数 –Xms、-Xmx 来指定。

默认的，新生代 ( Young ) 与老年代 ( Old ) 的比例的值为 **1:2** ( 该值可以通过参数 –XX:NewRatio 来指定 )，

即：**新生代 ( Young ) = 1/3 的堆空间大小。老年代 ( Old ) = 2/3 的堆空间大小**。

其中，新生代 ( Young ) 被细分为 Eden 和 两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分

默认的，**Eden: from : to = 8 :1 : 1** ( 可以通过参数**–XX:SurvivorRatio** 来设定 )，即： **Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小**。

JVM **每次只会使用 Eden 和其中的一块 Survivor 区域**来为对象服务，所以**无论什么时候，总是有一块Survivor区域是空闲着的**。

因此，新生代实际可用的内存空间为 9/10 ( 即90% )的新生代空间。

**新生代**

是用来存放新生的对象。一般占据堆的 1/3 空间。由于频繁创建对象，所以新生代会频繁触发MinorGC 进行垃圾回收。新生代又分为 Eden区、 ServivorFrom、 ServivorTo 三个区。  
**Eden 区**  
Java 新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年代）。当 Eden 区内存不够的时候就会触发 MinorGC，对新生代区进行一次垃圾回收。  
**Servivor from 区**  
上一次 GC 的幸存者，作为这一次 GC 的被扫描者。  
**Servivor to 区**  
保留了一次 MinorGC 过程中的幸存者。  
**MinorGC 的过程（复制->清空->互换）**  
MinorGC 采用复制算法。

1. eden、 servicorFrom 复制到 ServicorTo，年龄+1  
   首先，把 Eden 和 ServivorFrom 区域中存活的对象复制到 ServicorTo 区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果 ServicorTo 不够位置了就放到老年区）；
2. 清空 eden、 servicorFrom  
   然后，清空 Eden 和 ServicorFrom 中的对象；
3. ServicorTo 和 ServicorFrom 互换  
   最后， ServicorTo 和 ServicorFrom 互换，原 ServicorTo 成为下一次 GC 时的 ServicorFrom区。

**老年代**

主要存放应用程序中生命周期长的内存对象。  
老年代的对象比较稳定，所以 MajorGC （常常称之为 FULL GC）不会频繁执行。在进行 FULL GC前一般都先进行了一次 MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次 MajorGC 进行垃圾回收腾出空间。  
FULL GC 采用标记清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。 ajorGC 的耗时比较长，因为要扫描再回收。 FULL GC 会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。当老年代也满了装不下的时候，就会抛出 OOM（Out of Memory）异常。

**永久代**

指内存的永久保存区域，主要存放 Class 和 Meta（元数据）的信息,Class 在被加载的时候被放入永久区域， 它和和存放实例的区域不同,GC 不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的 Class 的增多而胀满，最终抛出 OOM 异常。

**JVM内存为什么要分成新生代，老年代，持久代。新生代中为什么要分为Eden和Survivor。**

**思路：** 先讲一下JAVA堆，新生代的划分，再谈谈它们之间的转化，相互之间一些参数的配置（如： –XX:NewRatio，–XX:SurvivorRatio等），再解释为什么要这样划分，最好加一点自己的理解。

**参考答案：**

这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。

1）共享内存区划分

* 共享内存区 = 持久带 + 堆
* 持久带 = 方法区 + 其他
* Java堆 = 老年代 + 新生代
* 新生代 = Eden + S0 + S1

2）一些参数的配置

* 默认的，新生代 ( Young ) 与老年代 ( Old ) 的比例的值为 1:2 ，可以通过参数 –XX:NewRatio 配置。
* 默认的，Eden : from : to = 8 : 1 : 1 ( 可以通过参数 –XX:SurvivorRatio 来设定)
* Survivor区中的对象被复制次数为15(对应虚拟机参数 -XX:+MaxTenuringThreshold)

3)为什么要分为Eden和Survivor?为什么要设置两个Survivor区？

* 如果没有Survivor，Eden区每进行一次Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发Major GC.老年代的内存空间远大于新生代，进行一次Full GC消耗的时间比Minor GC长得多,所以需要分为Eden和Survivor。
* Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生，Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。
* 设置两个Survivor区最大的好处就是解决了碎片化，刚刚新建的对象在Eden中，经历一次Minor GC，Eden中的存活对象就会被移动到第一块survivor space S0，Eden被清空；等Eden区再满了，就再触发一次Minor GC，Eden和S0中的存活对象又会被复制送入第二块survivor space S1（这个过程非常重要，因为这种复制算法保证了S1中来自S0和Eden两部分的存活对象占用连续的内存空间，避免了碎片化的发生）

**JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代**

**思路：** 先描述一下Java堆内存划分，再解释Minor GC，Major GC，full GC，描述它们之间转化流程。

**我的答案：**

* Java堆 = 老年代 + 新生代
* 新生代 = Eden + S0 + S1
* 当 Eden 区的空间满了， Java虚拟机会触发一次 Minor GC，以收集新生代的垃圾，存活下来的对象，则会转移到 Survivor区。
* **大对象**（需要大量连续内存空间的Java对象，如那种很长的字符串）**直接进入老年态**；
* 如果对象在Eden出生，并经过第一次Minor GC后仍然存活，并且被Survivor容纳的话，年龄设为1，每熬过一次Minor GC，年龄+1，**若年龄超过一定限制（15），则被晋升到老年态**。即**长期存活的对象进入老年态**。
* 老年代满了而**无法容纳更多的对象**，Minor GC 之后通常就会进行Full GC，Full GC 清理整个内存堆 – **包括年轻代和年老代**。
* Major GC **发生在老年代的GC**，**清理老年区**，经常会伴随至少一次Minor GC，**比Minor GC慢10倍以上**。

**JVM中的永久代中会发生垃圾回收吗**

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。请参考下Java8：从永久代到元数据区  
(译者注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

**JAVA8 与元数据**

在 Java8 中， 永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。元空间的本质和永久代类似，元空间与永久代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。 类的元数据放入native memory, 字符串池和类的静态变量放入 java 堆中， 这样可以加载多少类的元数据就不再由MaxPermSize 控制, 而由系统的实际可用空间来控制。

**如何判断对象可以被回收？**

判断对象是否存活一般有两种方式：

* 引用计数：

每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。

* 可达性分析（Reachability Analysis）：

从GC Roots开始向下搜索，搜索所走过的路径称为引用链。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用的，不可达对象。

**引用计数法**

在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关联的引用， 即他们的引用计数都不为 0， 则说明对象不太可能再被用到，那么这个对象就是可回收对象。

**可达性分析**

为了解决引用计数法的循环引用问题， Java 使用了可达性分析的方法。通过一系列的“GC roots”对象作为起点搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。要注意的是，不可达对象不等价于可回收对象， 不可达对象变为可回收对象至少要经过两次标记过程。两次标记后仍然是可回收对象，则将面临回收。

**Minor GC与Full GC分别在什么时候发生？**

新生代内存不够用时候发生MGC也叫YGC，JVM内存不够的时候发生FGC

**垃圾收集算法有哪些类型？**

GC最基础的算法有三类： 标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

标记 -清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。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，然后再把已使用过的内存空间一次清理掉。

标记-压缩算法，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都向一端移动，然后直接清理掉端边界以外的内存

分代收集算法，“分代收集”（Generational Collection）算法，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

**说一下 JVM 有哪些垃圾回收算法？**

* 标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。
* 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半。
* 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。
* 分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

**标记-清除算法**

标记无用对象，然后进行清除回收。

标记-清除算法（Mark-Sweep）是一种常见的基础垃圾收集算法，它将垃圾收集分为两个阶段：

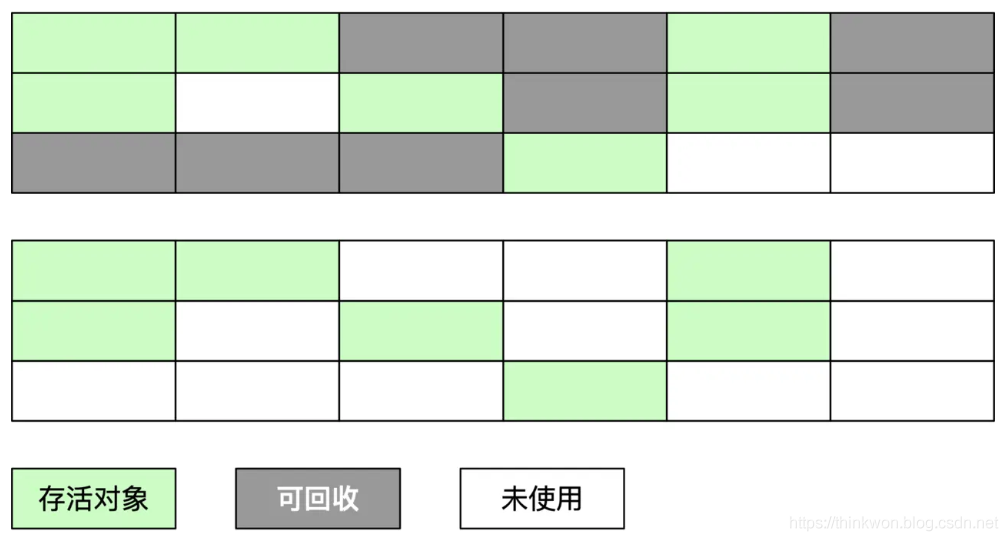
* 标记阶段：标记出可以回收的对象。
* 清除阶段：回收被标记的对象所占用的空间。

标记-清除算法之所以是基础的，是因为后面讲到的垃圾收集算法都是在此算法的基础上进行改进的。

**优点**：实现简单，不需要对象进行移动。

**缺点**：标记、清除过程效率低，产生大量不连续的内存碎片，提高了垃圾回收的频率。

标记-清除算法的执行的过程如下图所示

[](https://img-blog.csdnimg.cn/20200104115917418.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

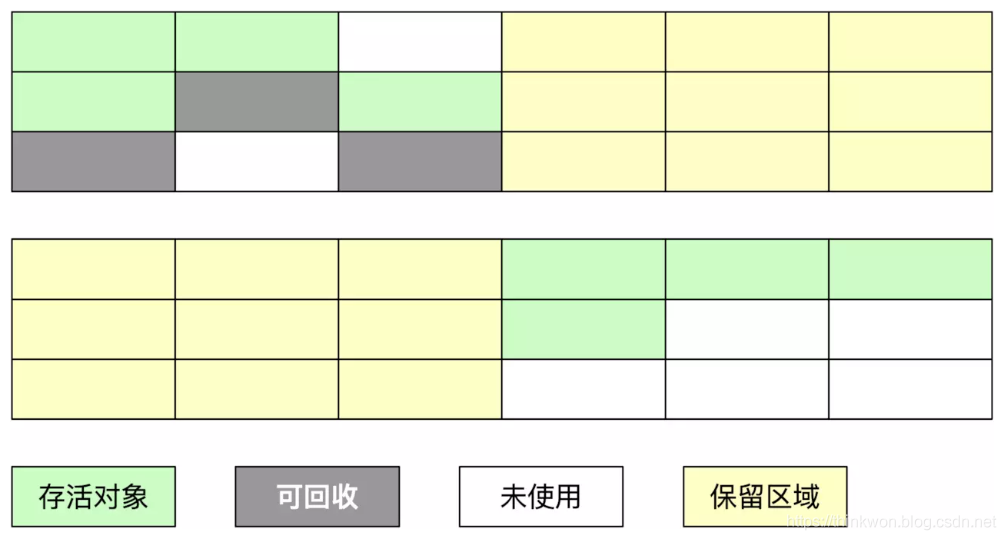
**复制算法**

为了解决标记-清除算法的效率不高的问题，产生了复制算法。它把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾收集时，遍历当前使用的区域，把存活对象复制到另外一个区域中，最后将当前使用的区域的可回收的对象进行回收。

**优点**：按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片。

**缺点**：可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制。

复制算法的执行过程如下图所示

[](https://img-blog.csdnimg.cn/20200104115940771.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

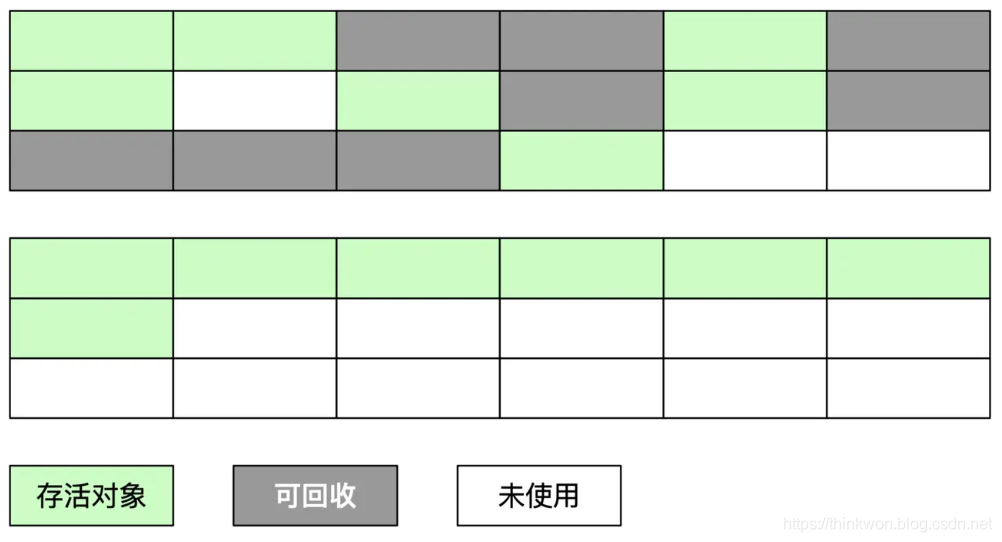
**标记-整理算法**

在新生代中可以使用复制算法，但是在老年代就不能选择复制算法了，因为老年代的对象存活率会较高，这样会有较多的复制操作，导致效率变低。标记-清除算法可以应用在老年代中，但是它效率不高，在内存回收后容易产生大量内存碎片。因此就出现了一种标记-整理算法（Mark-Compact）算法，与标记-整理算法不同的是，在标记可回收的对象后将所有存活的对象压缩到内存的一端，使他们紧凑的排列在一起，然后对端边界以外的内存进行回收。回收后，已用和未用的内存都各自一边。

**优点**：解决了标记-清理算法存在的内存碎片问题。

**缺点**：仍需要进行局部对象移动，一定程度上降低了效率。

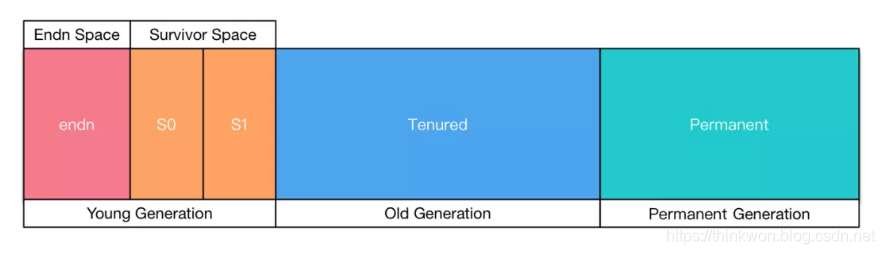
标记-整理算法的执行过程如下图所示

[](https://img-blog.csdnimg.cn/20200104120006513.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

**分代收集算法**

分代收集法是目前大部分 JVM 所采用的方法，其核心思想是根据对象存活的不同生命周期将内存划分为不同的域，一般情况下将 GC 堆划分为老生代(Tenured/Old Generation)和新生代(YoungGeneration)。老生代的特点是每次垃圾回收时只有少量对象需要被回收，新生代的特点是每次垃圾回收时都有大量垃圾需要被回收，因此可以根据不同区域选择不同的算法。

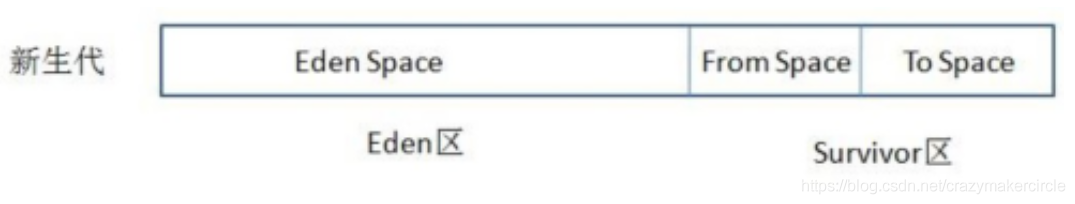
当前商业虚拟机都采用**分代收集**的垃圾收集算法。分代收集算法，顾名思义是根据对象的**存活周期**将内存划分为几块。一般包括**年轻代**、**老年代** 和 **永久代**，如图所示：

[](https://img-blog.csdnimg.cn/20200104120031885.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

当前主流 VM 垃圾收集都采用”分代收集” (Generational Collection)算法, 这种算法会根据对象存活周期的不同将内存划分为几块, 如 JVM 中的 新生代、老年代、永久代， 这样就可以根据各年代特点分别采用最适当的 GC 算法

**新生代与复制算法**

每次垃圾收集都能发现大批对象已死, 只有少量存活. 因此选用复制算法, 只需要付出少量存活对象的复制成本就可以完成收集

目前大部分 JVM 的 GC 对于新生代都采取 Copying 算法，因为新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少，但通常并不是按照 1： 1 来划分新生代。一般将新生代划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另一块 Survivor 空间中。  
[](https://img-blog.csdnimg.cn/20210203104013367.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2NyYXp5bWFrZXJjaXJjbGU=,size_16,color_FFFFFF,t_70)

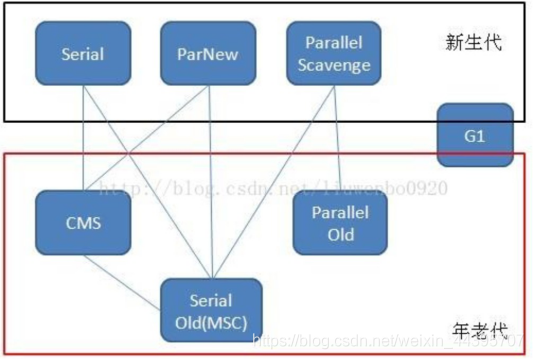
**老年代与标记复制算法**

因为老年代对象存活率高、没有额外空间对它进行分配担保, 就必须采用“标记—清理”或“标记—整理” 算法来进行回收, 不必进行内存复制, 且直接腾出空闲内存。因而采用 Mark-Compact 算法。

1. JAVA 虚拟机提到过的处于方法区的永生代(Permanet Generation)， 它用来存储 class 类，常量，方法描述等。对永生代的回收主要包括废弃常量和无用的类。
2. 对象的内存分配主要在新生代的 Eden Space 和 Survivor Space 的 From Space(Survivor 目前存放对象的那一块)，少数情况会直接分配到老生代。
3. 当新生代的 Eden Space 和 From Space 空间不足时就会发生一次 GC，进行 GC 后， EdenSpace 和 From Space 区的存活对象会被挪到 To Space，然后将 Eden Space 和 FromSpace 进行清理。
4. 如果 To Space 无法足够存储某个对象，则将这个对象存储到老生代。
5. 在进行 GC 后，使用的便是 Eden Space 和 To Space 了，如此反复循环。
6. 当对象在 Survivor 区躲过一次 GC 后，其年龄就会+1。 默认情况下年龄到达 15 的对象会被移到老生代中。

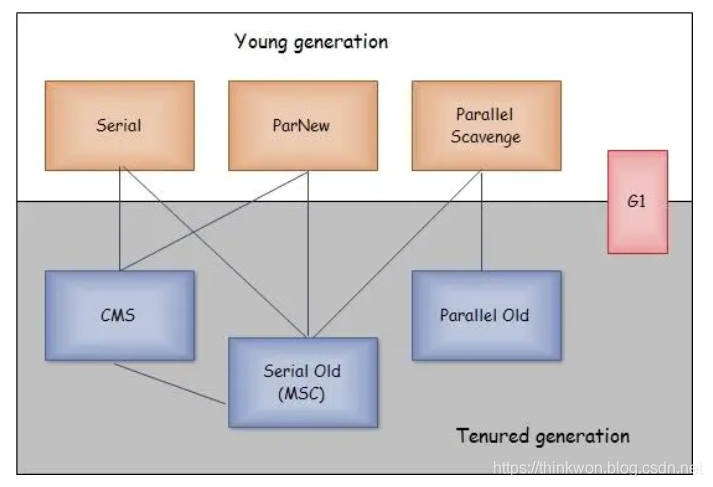
**GC 垃圾收集器**

Java 堆内存被划分为新生代和年老代两部分，新生代主要使用复制和标记-清除垃圾回收算法；年老代主要使用标记-整理垃圾回收算法，因此 java 虚拟中针对新生代和年老代分别提供了多种不同的垃圾收集器， JDK1.6 中 Sun HotSpot 虚拟机的垃圾收集器如下：

[](https://img-blog.csdnimg.cn/20200515111133467.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3dlaXhpbl80NDM5NTcwNw==,size_16,color_FFFFFF,t_70)

**说一下 JVM 有哪些垃圾回收器？**

如果说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。下图展示了7种作用于不同分代的收集器，其中用于回收新生代的收集器包括Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。不同收集器之间的连线表示它们可以搭配使用。

[](https://img-blog.csdnimg.cn/20200104120144820.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

* Serial收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；
* ParNew收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；
* Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；
* Serial Old收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年代版本；
* Parallel Old收集器 (标记-整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本；
* CMS(Concurrent Mark Sweep)收集器（标记-清除算法）： 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。
* G1(Garbage First)收集器 (标记-整理算法)： Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

**Serial 与 Parallel GC 之间的不同之处？**

Serial 与 Parallel 在 GC 执行的时候都会引起 stop-the-world。它们之间主要不同 serial 收集器是默认的复制收集器，执行 GC 的时候只有一个线程，而parallel 收集器使用多个 GC 线程来执行。

**类似的问题：你知道哪几种垃圾收集器，各自的优缺点，重点讲下cms和G1，包括原理，流程，优缺点。**

**思路：** 一定要记住典型的垃圾收集器，尤其cms和G1，它们的原理与区别，涉及的垃圾回收算法。

**参考答案：**

1）几种垃圾收集器：

* **Serial收集器：** 单线程的收集器，收集垃圾时，必须stop the world，使用复制算法。
* **ParNew收集器：** Serial收集器的多线程版本，也需要stop the world，复制算法。
* **Parallel Scavenge收集器：** 新生代收集器，复制算法的收集器，并发的多线程收集器，目标是达到一个可控的吞吐量。如果虚拟机总共运行100分钟，其中垃圾花掉1分钟，吞吐量就是99%。
* **Serial Old收集器：** 是Serial收集器的老年代版本，单线程收集器，使用标记整理算法。
* **Parallel Old收集器：** 是Parallel Scavenge收集器的老年代版本，使用多线程，标记-整理算法。
* **CMS(Concurrent Mark Sweep) 收集器：** 是一种以获得最短回收停顿时间为目标的收集器，**标记清除算法，运作过程：初始标记，并发标记，重新标记，并发清除**，收集结束会产生大量空间碎片。
* **G1收集器：** 标记整理算法实现，**运作流程主要包括以下：初始标记，并发标记，最终标记，筛选标记**。不会产生空间碎片，可以精确地控制停顿。

2）CMS收集器和G1收集器的区别：

* CMS收集器是老年代的收集器，可以配合新生代的Serial和ParNew收集器一起使用；
* G1收集器收集范围是老年代和新生代，不需要结合其他收集器使用；
* CMS收集器以最小的停顿时间为目标的收集器；
* G1收集器可预测垃圾回收的停顿时间
* CMS收集器是使用“标记-清除”算法进行的垃圾回收，容易产生内存碎片
* G1收集器使用的是“标记-整理”算法，进行了空间整合，降低了内存空间碎片。

**详细介绍一下 CMS 垃圾回收器？**

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。在启动 JVM 的参数加上“-XX:+UseConcMarkSweepGC”来指定使用 CMS 垃圾回收器。

CMS 使用的是标记-清除的算法实现的，所以在 gc 的时候回产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低。

**Serial 垃圾收集器（单线程、 复制算法）**

Serial（英文连续） 是最基本垃圾收集器，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， 它不但只会使用一个 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。

**ParNew 垃圾收集器（Serial+多线程）**

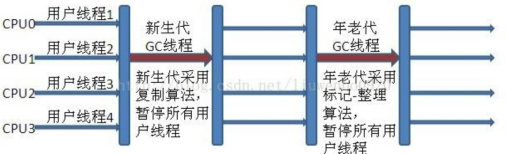
ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃圾收集之外，其余的行为和 Serial 收集器完全一样， ParNew 垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。  
ParNew 收集器默认开启和 CPU 数目相同的线程数，可以通过-XX:ParallelGCThreads 参数来限制垃圾收集器的线程数。 【Parallel：平行的】  
ParNew 虽然是除了多线程外和Serial 收集器几乎完全一样，但是ParNew垃圾收集器是很多 java虚拟机运行在 Server 模式下新生代的默认垃圾收集器。

**Parallel Scavenge 收集器（多线程复制算法、高效）**

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。 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。

**Serial Old 收集器（单线程标记整理算法 ）**

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：

1. 在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。
2. 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：  
   [](https://img-blog.csdnimg.cn/2020051511121357.png)  
   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 ParallelScavenge/ParNew 与年老代 Serial Old 搭配垃圾收集过程图：  
   [](https://img-blog.csdnimg.cn/20200515111222258.png)

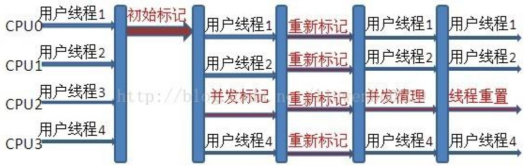
**Parallel Old 收集器（多线程标记整理算法）**

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  
[](https://img-blog.csdnimg.cn/20200515111233186.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3dlaXhpbl80NDM5NTcwNw==,size_16,color_FFFFFF,t_70)

**CMS 收集器（多线程标记清除算法）**

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 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：

**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。  
**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。  
**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。  
**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

[](https://img-blog.csdnimg.cn/20200515111255464.png)

**G1 收集器**

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：  
1.基于标记-整理算法，不产生内存碎片。  
2.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

**新生代垃圾回收器和老年代垃圾回收器都有哪些？有什么区别？**

* 新生代回收器：Serial、ParNew、Parallel Scavenge
* 老年代回收器：Serial Old、Parallel Old、CMS
* 整堆回收器：G1

新生代垃圾回收器一般采用的是复制算法，复制算法的优点是效率高，缺点是内存利用率低；老年代回收器一般采用的是标记-整理的算法进行垃圾回收。

**简述分代垃圾回收器是怎么工作的？**

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

* 把 Eden + From Survivor 存活的对象放入 To Survivor 区；
* 清空 Eden 和 From Survivor 分区；
* From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

**什么时候会触发FullGC？**

除直接调用System.gc外，触发Full GC执行的情况有如下四种。  
**1.旧生代空间不足**  
老生代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出如下错误：  
java.lang.OutOfMemoryError: Java heap space  
为避免以上两种状况引起的FullGC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。  
**2.Permanet Generation空间满**  
PermanetGeneration中存放的为一些class的信息等，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation  
可能会被占满，在未配置为采用CMS GC的情况下会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出如下错误信息：  
java.lang.OutOfMemoryError: PermGen space  
为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。  
**3.CMS GC时出现promotion failed和concurrent mode failure**  
对于采用CMS进行老生代GC的程序而言，尤其要注意GC日志中是否有promotion failed和concurrent mode failure两种状况，当这两种状况出现时可能会触发Full GC。  
promotionfailed是在进行Minor GC时，survivor space放不下、对象只能放入老生代，而此时老生代也放不下造成的；concurrent mode failure是在执行CMS GC的过程中同时有对象要放入老生代，而此时老生代空间不足造成的。  
应对措施为：增大survivorspace、老生代空间或调低触发并发GC的比率，但在JDK 5.0+、6.0+的版本中有可能会由于JDK的bug29导致CMS在remark完毕后很久才触发sweeping动作。对于这种状况，可通过设置-XX:CMSMaxAbortablePrecleanTime=5（单位为ms）来避免。  
**4.统计得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间**  
这是一个较为复杂的触发情况，Hotspot为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行Minor GC时，做了一个判断，如果之前统计所得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发Full GC。  
例如程序第一次触发MinorGC后，有6MB的对象晋升到旧生代，那么当下一次Minor GC发生时，首先检查旧生代的剩余空间是否大于6MB，如果小于6MB，则执行Full GC。  
当新生代采用PSGC时，方式稍有不同，PS GC是在Minor GC后也会检查，例如上面的例子中第一次Minor GC后，PS GC会检查此时旧生代的剩余空间是否大于6MB，如小于，则触发对旧生代的回收。除了以上4种状况外，对于使用RMI来进行RPC或管理的Sun JDK应用而言，默认情况下会一小时执行一次Full GC。可通过在启动时通过- java-Dsun.rmi.dgc.client.gcInterval=3600000来设置Full GC执行的间隔时间或通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc

**内存分配策略**

**简述java内存分配与回收策率以及Minor GC和Major GC**

所谓自动内存管理，最终要解决的也就是内存分配和内存回收两个问题。前面我们介绍了内存回收，这里我们再来聊聊内存分配。

对象的内存分配通常是在 Java 堆上分配（随着虚拟机优化技术的诞生，某些场景下也会在栈上分配，后面会详细介绍），对象主要分配在新生代的 Eden 区，如果启动了本地线程缓冲，将按照线程优先在 TLAB 上分配。少数情况下也会直接在老年代上分配。总的来说分配规则不是百分百固定的，其细节取决于哪一种垃圾收集器组合以及虚拟机相关参数有关，但是虚拟机对于内存的分配还是会遵循以下几种「普世」规则：

**对象优先在 Eden 区分配**

多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行分配时，虚拟机将会发起一次 Minor GC。如果本次 GC 后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。

这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中发现 Major GC/Full GC。

* **Minor GC** 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所有 Minor GC 非常频繁，一般回收速度也非常快；
* **Major GC/Full GC** 是指发生在老年代的 GC，出现了 Major GC 通常会伴随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

**大对象直接进入老年代**

所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对象。

前面我们介绍过新生代使用的是标记-清除算法来处理垃圾回收的，如果大对象直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

**长期存活对象将进入老年代**

虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor 区中每「熬过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15） 就会被晋升到老年代。

**对象分配规则**

1. 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。
2. 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
3. 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。
4. 动态判断对象的年龄。如果Survivor区中相同年龄的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可以直接进入老年代。
5. 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC

**虚拟机类加载机制**

**简述java类加载机制?**

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，解析和初始化，最终形成可以被虚拟机直接使用的java类型。

**描述一下JVM加载Class文件的原理机制**

Java中的所有类，都需要由类加载器装载到JVM中才能运行。类加载器本身也是一个类，而它的工作就是把class文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

类装载方式，有两种 ：

1.隐式装载， 程序在运行过程中当碰到通过new 等方式生成对象时，隐式调用类装载器加载对应的类到jvm中，

2.显式装载， 通过class.forname()等方法，显式加载需要的类

Java类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类(像是基类)完全加载到jvm中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

**描述一下 JVM 加载 class 文件的原理机制**

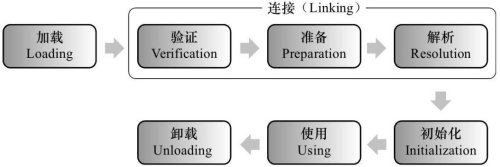
JVM 中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java 中的类加载器是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入类文件中的类。  
由于 Java 的跨平台性，经过编译的 Java 源程序并不是一个可执行程序，而是一个或多个类文件。当 Java 程序需要使用某个类时，JVM 会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class 文件中的数据读入到内存中，通常是创建一个字节数组读入.class 文件，然后产生与所加载类对应  
的 Class 对象。

加载完成后，Class 对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后 JVM 对  
类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。  
类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader 的子类）。

从 Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM 更好的保证了 Java 平台的安全性，在该机制中，JVM 自带的Bootstrap 是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM 不会向 Java 程序提供对 Bootstrap 的引用。下面是关于几个类  
加载器的说明：

1. Bootstrap：一般用本地代码实现，负责加载 JVM 基础核心类库（rt.jar）；
2. Extension：从 java.ext.dirs 系统属性所指定的目录中加载类库，它的父加载器是 Bootstrap；
3. System：又叫应用类加载器，其父类是 Extension。它是应用最广泛的类加载器。它从环境变量 classpath 或者系统属性  
   java.class.path 所指定的目录中记载类，是用户自定义加载器的默认父加载器。

**JVM 类加载机制**

JVM 类加载机制分为五个部分：加载，验证，准备，解析，初始化，下面我们就分别来看一下这五个过程。  
[](https://img-blog.csdnimg.cn/20200515111310928.png)

**加载**  
加载是类加载过程中的一个阶段， 这个阶段会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入口。注意这里不一定非得要从一个 Class 文件获取，这里既可以从 ZIP 包中读取（比如从 jar 包和 war 包中读取），也可以在运行时计算生成（动态代理），也可以由其它文件生成（比如将 JSP 文件转换成对应的 Class 类）。  
**验证**  
这一阶段的主要目的是为了确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。  
**准备**  
准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：  
实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080， 将 v 赋值为 8080 的 put static 指令是程序被编译后， 存放于类构造器方法之中。  
但是注意如果声明为：public static ﬁnal int v = 8080;  
在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v赋值为 8080。  
解析  
解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的：

public static int v = 8080;

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080， 将 v 赋值为 8080 的 put static 指令是程序被编译后， 存放于类构造器方法之中。但是注意如果声明为：  
在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v  
赋值为 8080。解析  
解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的：

public static final int v = 8080;

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v赋值为 8080。  
解析  
解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的：

1. CONSTANT\_Class\_info
2. CONSTANT\_Field\_info
3. CONSTANT\_Method\_info  
   等类型的常量。

**符号引用**  
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**  
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。  
**初始化**  
初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载器以外，其它操作都由 JVM 主导。到了初始阶段，才开始真正执行类中定义的 Java 程序代码。  
**类构造器**  
初始化阶段是执行类构造器方法的过程。 方法是由编译器自动收集类中的类变量的赋值操作和静态语句块中的语句合并而成的。虚拟机会保证子方法执行之前，父类的方法已经执行完毕， 如果一个类中没有对静态变量赋值也没有静态语句块，那么编译器可以不为这个类生成() 方法。注意以下几种情况不会执行类初始化：

1. 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
2. 定义对象数组，不会触发该类的初始化。
3. 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
4. 通过类名获取 Class 对象，不会触发类的初始化。
5. 通过 Class.forName 加载指定类时，如果指定参数 initialize 为 false 时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
6. 通过 ClassLoader 默认的 loadClass 方法，也不会触发初始化动作。

**什么是类加载器，类加载器有哪些?**

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。
2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

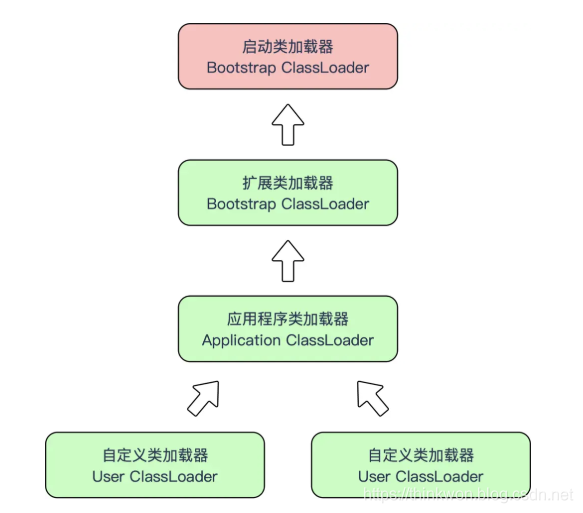
**说一下类装载的执行过程？**

类装载分为以下 5 个步骤：

* 加载：根据查找路径找到相应的 class 文件然后导入；
* 验证：检查加载的 class 文件的正确性；
* 准备：给类中的静态变量分配内存空间；
* 解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；
* 初始化：对静态变量和静态代码块执行初始化工作。

**什么是双亲委派模型？**

在介绍双亲委派模型之前先说下类加载器。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在 JVM 中的唯一性，每一个类加载器，都有一个独立的类名称空间。类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，然后再转化为 class 对象。

[](https://img-blog.csdnimg.cn/20200104165551656.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly90aGlua3dvbi5ibG9nLmNzZG4ubmV0,size_16,color_FFFFFF,t_70)

类加载器分类：

* 启动类加载器（Bootstrap ClassLoader），是虚拟机自身的一部分，用来加载Java\_HOME/lib/目录中的，或者被 -Xbootclasspath 参数所指定的路径中并且被虚拟机识别的类库；
* 其他类加载器：
* 扩展类加载器（Extension ClassLoader）：负责加载\lib\ext目录或Java. ext. dirs系统变量指定的路径中的所有类库；
* 应用程序类加载器（Application ClassLoader）。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

双亲委派模型：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，每一层的类加载器都是如此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加载类。

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

**简单说说你了解的类加载器，可以打破双亲委派么，怎么打破。**

**思路：** 先说明一下什么是类加载器，可以给面试官画个图，再说一下类加载器存在的意义，说一下双亲委派模型，最后阐述怎么打破双亲委派模型。

**参考的答案：**

1. 什么是类加载器？

**类加载器** 就是根据指定全限定名称将class文件加载到JVM内存，转为Class对象。

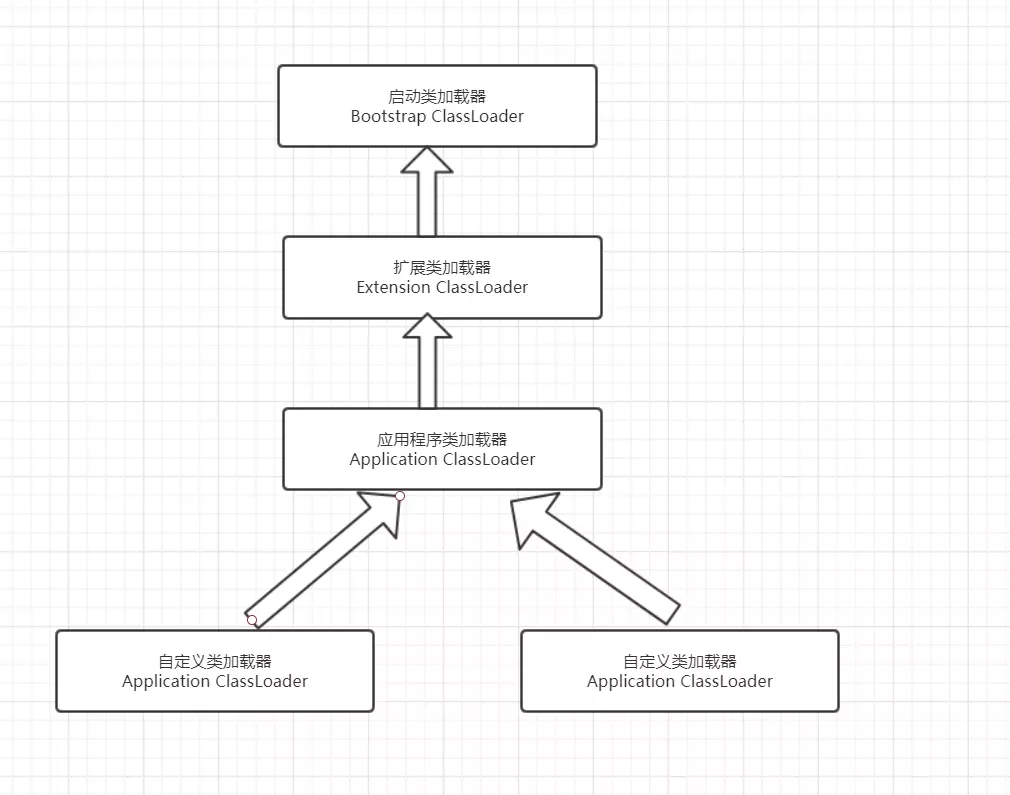
* 启动类加载器（Bootstrap ClassLoader）：由C++语言实现（针对HotSpot）,负责将存放在<JAVA\_HOME>\lib目录或-Xbootclasspath参数指定的路径中的类库加载到内存中。
* 其他类加载器：由Java语言实现，继承自抽象类ClassLoader。如：
* 扩展类加载器（Extension ClassLoader）：负责加载<JAVA\_HOME>\lib\ext目录或java.ext.dirs系统变量指定的路径中的所有类库。
* 应用程序类加载器（Application ClassLoader）。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

2）双亲委派模型

**双亲委派模型工作过程是：**

如果一个类加载器收到类加载的请求，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完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。

双亲委派模型图：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly91c2VyLWdvbGQtY2RuLnhpdHUuaW8vMjAxOS83LzIzLzE2YzFjNTRjZjRhZDg4NmI_aW1hZ2VWaWV3Mi8wL3cvMTI4MC9oLzk2MC9mb3JtYXQvd2VicC9pZ25vcmUtZXJyb3IvMQ?x-oss-process=image/format,png)

3）为什么需要双亲委派模型？

在这里，先想一下，如果没有双亲委派，那么用户是不是可以**自己定义一个java.lang.Object的同名类**，**java.lang.String的同名类**，并把它放到ClassPath中,那么**类之间的比较结果及类的唯一性将无法保证**，因此，为什么需要双亲委派模型？**防止内存中出现多份同样的字节码**

4）怎么打破双亲委派模型？

打破双亲委派机制则不仅**要继承ClassLoader**类，还要**重写loadClass和findClass**方法。

**什么是Java虚拟机？为什么Java被称作是“平台无关的编程语言”？**

Java虚拟机是一个可以执行Java字节码的虚拟机进程。Java源文件被编译成能被Java虚拟机执行的字节码文件。 Java被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java虚拟机让这个变为可能，因为它知道底层硬件平台的 指令长度和其他特性。

**JVM调优**

**说一下 JVM 调优的工具？**

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具。

* jconsole：用于对 JVM 中的内存、线程和类等进行监控；
* jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等。

**常用的 JVM 调优的参数都有哪些？**

* -Xms2g：初始化推大小为 2g；
* -Xmx2g：堆最大内存为 2g；
* -XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；
* -XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；
* –XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；
* -XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合；
* -XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；
* -XX:+PrintGC：开启打印 gc 信息；
* -XX:+PrintGCDetails：打印 gc 详细信息。

**调优命令有哪些？**

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

1. jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。
2. jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。
3. jmap，JVM Memory Map命令用于生成heap dump文件
4. jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看
5. jstack，用于生成java虚拟机当前时刻的线程快照。
6. jinfo，JVM Conﬁguration info 这个命令作用是实时查看和调整虚拟机运行参数

**调优工具**

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory AnalyzerTool)、GChisto。

1. jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存， 线程和类等的监控
2. jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。
3. MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Javaheap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗
4. GChisto，一款专业分析gc日志的工具

**说说你知道的几种主要的JVM参数**

**思路：** 可以说一下堆栈配置相关的，垃圾收集器相关的，还有一下辅助信息相关的。

**参考答案：**

1）堆栈配置相关

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:MaxPermSize=16m -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxTenuringThreshold=0

**-Xmx3550m：** 最大堆大小为3550m。

**-Xms3550m：** 设置初始堆大小为3550m。

**-Xmn2g：** 设置年轻代大小为2g。

**-Xss128k：** 每个线程的堆栈大小为128k。

**-XX:MaxPermSize：** 设置持久代大小为16m

**-XX:NewRatio=4:** 设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。

**-XX:SurvivorRatio=4：** 设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

**-XX:MaxTenuringThreshold=0：** 设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。

2）垃圾收集器相关

-XX:+UseParallelGC-XX:ParallelGCThreads=20-XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5-XX:+UseCMSCompactAtFullCollection：

**-XX:+UseParallelGC：** 选择垃圾收集器为并行收集器。

**-XX:ParallelGCThreads=20：** 配置并行收集器的线程数

**-XX:+UseConcMarkSweepGC：** 设置年老代为并发收集。

**-XX:CMSFullGCsBeforeCompaction**：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

**-XX:+UseCMSCompactAtFullCollection：** 打开对年老代的压缩。可能会影响性能，但是可以消除碎片

3）辅助信息相关

-XX:+PrintGC-XX:+PrintGCDetails

**-XX:+PrintGC 输出形式:**

[GC 118250K->113543K(130112K), 0.0094143 secs] [Full GC 121376K->10414K(130112K), 0.0650971 secs]

**-XX:+PrintGCDetails 输出形式:**

[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs] [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs

**怎么打出线程栈信息。**

**思路：** 可以说一下jps，top ，jstack这几个命令，再配合一次排查线上问题进行解答。

**参考答案：**

* 输入jps，获得进程号。
* top -Hp pid 获取本进程中所有线程的CPU耗时性能
* jstack pid命令查看当前java进程的堆栈状态
* 或者 jstack -l > /tmp/output.txt 把堆栈信息打到一个txt文件。
* 可以使用fastthread 堆栈定位，[fastthread.io/](https://link.juejin.im/?target=http%3A%2F%2Ffastthread.io%2F)