链滴

初探 JVM

作者：Hefery

原文链接：https://ld246.com/article/1644156069883

来源网站：链滴

<h2 id="JVM概述">JVM 概述</h2>
JVM：Java Virtual Machine，Java 程序的运行环境，Java 虚拟机通过软件来模拟 Java 字节码指令集
虚拟机：通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的计算机系统</>
JVM 功能：虚拟机是 Java 平台无关的保障

通过 Classloader 寻找和装载 class 文件
解释字节码成为指令并执行，提供 class 文件的运行环境
进行运行期间的内存分配和垃圾回收
提供与硬件交互的平台

<h2 id="JVM规范">JVM 规范</h2>
Java 虚拟机规范为不同的硬件平台提供了一种编译 Java 技术代码的规范
JVM 规范作用

Java 虚拟机规范为不同的硬件平台提供了一种编译 Java 技术代码的规范
该规范使 Java 软件独立于平台，因为编译是针对作为虚拟机的“一般机器”而做，这个“一般器”可用软件模拟并运行于各种现存的计算机系统，也可用硬件来实现

规范主要内容

字节码指令集（相当于中央处理器 CPU）
Class 文件的格式
数据类型和值
运行时数据区
栈帧
特殊方法
类库
异常
虚拟机的启动、加载、链接和初始化

官方文档：https://docs.oracle.co/javase/specs/index.html
<h3 id="字节码指令集">字节码指令集</h3>
Java 虚拟机的指令由一个字节长度的、代表着某种特定操作含义的操作码（opcode）以及跟随后的零至多个代表此操作所需参数的操作数（operand）所构成。虚拟机中许多指令并不包含操作数只有一个操作码
<h4 id="加载和存储指令">加载和存储指令</h4>
加载和存储指令用于将数据从栈帧的本地变量表和操作数栈之间来回传递
将一个本地变量加载到操作数栈的指令：
iload、iload_<code><n></code>、lload、lload_<code><n></code>、 fload、floa_<code><n></code>、 dload、dload_<code><n></code>、 aload、aload_<cod><n></code>
将一个数值从操作数栈存储到局部变量表的指令：
istore、 istore_<code><n></code>、 lstore、lstore_<code><n></code>、 fstor、fstore_<code><n></code>、 dstore、dstore_<code><n></code>、 astore、asore_<code><n></code>
将一个常量加载到操作数栈的指令：
bipush、sipush、ldc、ldc_w、ldc2_w、aconst_null、iconst_ml、iconst_<code><i></cod>、lconst_<code><l></code>、fconst_<code><f></code>、 dconst_<code><></code>
<h4 id="算术指令">算术指令</h4>

加法指令：iadd、ladd、fadd、dadd
减法指令：isub、lsub、fsub、dsub
乘法指令：imul、lmul、fmul、dmul
除法指令：idiv、lliv、fdiv、ddiv
求余指令：irem、lrem、frem、drem
求负值指令：ineg、lneg、fneg、dneg
移位指令：ishl、ishr、iushr、lshl、lshr、lusr
按位或指令：ior、lor
按位与指令：iand、land
按位异或指令：ixor、lxor
局部变量自增指令；iinc
比较指令：dcmpg、dcmpl、fcmpg、fcmpl、lcmp
<h4 id="类型转换">类型转换</h4>
宽化类型转换：i2l、i2f、i2d、l2f、l2d、f2d
int => long、float、double
long => float、double
float => double
窄化类型转换：i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l、d2f
int => btye、short、char
long => int
float => int、long
double => int、long、float
<h4 id="对象的创建与操作">对象的创建与操作</h4>
创建类实例的指令：new
创建数组的指令：newarray、 anewarray、 multianewarray
访问类字段和类实例字段指令：getfield、putfield、getstatic、putstatic
把一个数组元素加载到操作数栈的指令：baload、caload、saload、laload、faload、daload、aalo
d
将一个操作数栈的值存储到数组元素中的指令：bastore、castore、sastore、iastore、lastore、fast
re、dastor、aastore
取数组长度的指令：arraylength
检查类实例或数组类型的指令：instanceof、checkcast
<h4 id="方法调用和返回指令">方法调用和返回指令</h4>

invokevirtual 指令：调用对象的实例方法，根据对象的实际类型进行分派（虚方法分派）。这
是 Java 语言中最常见的方法分派方式
invokeinterface 指令：调用接口方法，它会在运行时搜索由特定对象所实现的这个接口方法，
找出适合的方法进行调用
invokespecial 指令：调用一些需要特殊处理的实例方法，包括实例初始化方法、私有方法和父
方法
invokestatic 指令：调用命名类中的类方法（static 方法）
invokedynamic 指令：调用以绑定了 invokedynamic 指令的调用点对象（call site object）作
目标的方法。调用点对象是一个特殊语法结构，当一条 invokedynamic 指令首次被 Java 虚拟机执
前，Java 虚拟机将会执行一个引导方法（bootstrap method）并以这个方法
运行结果作为调用点对象。因此，每条 invokedynamic 指令都有独一无二的链接状态，这是它与其
方法调用指令的一个差异

<h3 id="Class文件格式">Class 文件格式</h3>
Class 文件是 JVM 的输入，Java 虚拟机规范中定义了 Class 文件的结构。 Class 文件是 JVM
现平台无关、技术无关的基础

Class 文件是一组以 8 字节为单位的字节流，各个数据项目按顺序紧凑排列

<p>对于占用空间大于 8 字节的数据项，按照高位在前的方式分割成多个 8 字节进行存储</p>

<p>Class 文件格式里面只有两种类型：无符号数、表
无符号数：基本数据类型，以 u1、u2、u4、u8 来代表几个字节的无符号数
表：由多个无符号数和其它表构成的复合数据类型，通常以 _info 结尾</p>

<p>javap 工具生成非正式的“虚拟机汇编语言”，格式如下</p>
<p><index><opcode>[<operand1> [<operand2>...]][]>
<index>：指令操作码在数组中的下标，该数组以字节形式来存储当前方法的 JVM 代码；也可
是相对于方法起始处的字节偏移量
<opcode>：指令的助记码、<operand> 是操作数、<comment> 是行尾的注释 <
ode><opcode></code>[<code><operand1></code> [<code><operand2><
code>...]][]>
<index>：指令操作码在数组中的下标，该数组以字节形式来存储当前方法的 JVM 代码；也可
是相对于方法起始处的字节偏移量
<opcode>：指令的助记码、<code><operand></code> 是操作数、<code><com
ent></code> 是行尾的注释 <code><opcode></code>[<code><operand1></c
de> [<code><operand2></code>...]][]>
<index>：指令操作码在数组中的下标，该数组以字节形式来存储当前方法的 JVM 代码；也可
是相对于方法起始处的字节偏移量
<opcode>：指令的助记码、<code><operand></code> 是操作数、<code><com
ent></code> 是行尾的注释</p>

<pre><code class="language-java highlight-chroma"><span c
ass="highlight-cl">ClassFile

   <span class="h
ghlight-n">u4   magic<span class="highli
ht-o">;
   <span class="h
ghlight-n">u2   minor_version<span class
"highlight-o">;
   <span class="h
ghlight-n">u2   major_version<span class
"highlight-o">;
   <span class="h
ghlight-n">u2   constant_pool_count<spa
 class="highlight-o">;
   <span class="h
ghlight-n">cp_info   constant_pool<span clas
="highlight-o">[constant_pool_count<span clas
="highlight-o">-1]

   <span class="h
ghlight-n">u2   access_flags<span class="
ighlight-o">;
   <span class="h
ghlight-n">u2   this_class<span class="hi
hlight-o">;
<span class="highlight-line"><span class="highlight-cl">   <span class="h

ghlight-n">u2 super_class<span class="h
ghlight-o">;
  <span class="h
ghlight-n">u2 interfaces_count<span cla
s="highlight-o">;
  <span class="h
ghlight-n">u2 interfaces<span class="hi
hlight-o">[interfaces_count<span class="highli
ht-o">];
  <span class="h
ghlight-n">u2 fields_count<span class="
ighlight-o">;
  <span class="h
ghlight-n">field_info fields<span class="highl
ght-o">[fields_count]

  <span class="h
ghlight-n">u2 methods_count<span clas
="highlight-o">;
  <span class="h
ghlight-n">method_info methods<span class
"highlight-o">[methods_count<span class="hig
light-o">];
  <span class="h
ghlight-n">u2 attributes_count<span cla
s="highlight-o">;
  <span class="h
ghlight-n">attribute_info attributes<span class=
highlight-o">[attributes_count<span class="hig
light-o">];
<span class="high
ight-o">}
</code></pre>

magic：魔数，唯一作用是确定这个文件是否为一个能被虚拟机所接受的 c1ass 文件。魔数值固为 0XCAFEBABE，不会改变
minor_version（副版本号）、major_version（主版本号）：分别表示 c1ass 文件的副、主版。共同构成了 c1ass 文件的格式版本号
constant_pool_count：常量池计数器，值等于常量池表中的成员数 +1
常量池表的索引值只有在 >0 且小 <constant_pool_count 才会认为是有效的，对于 1ong 和 double 类型有例外情况
constant_poo1：常量池，一种表结构，包含 c1ass 文件结构及其子结构中引用的所有字符串常、类或接口名、字段名和其他常量
常量池中的每一项都具备相同的特征作为类型标记，用于确定该项的格式，这个字节称为 tag_byte 标记字节、标签字节）
access_flags：访问标志，一种由标志所构成的掩码，用于表示某个类或者接口的访问权限及属。每个标志的取值及其含义
this_class：类索引，值必须是对常量池表中某项的一个有效索引值。常量池在这个索引处的成必须为 CONSTANT_Class_info 类型结构体，该结构体表示这个 c1ass 文件所定义的类或接口
super_class：父类索引，对于类来说，super_class 的值要么是 0，要么是对常量池表中某项的个有效索引值。如果它的值不为 0，那么常量池在这个索引处的成员必须为 CONSTANT_Class_info 型常量，它表示这个 class 文件所定义的类的直接超类。在当前类的直接超类，以及它所有间接超类的 ClassFi1e 结构体中，access_flags 里面均不能带有 ACC_FINAL 标志
interfaces_count：接口计数器，值表示当前类或接口的直接超接口数量

interfaces：接口表，每个成员的值必须是对常量池表中某项的有效索引值，它的长度 interfaces_count。每个成员 interfaces[i] 必须为 CONSTANT_C1ass_info 结构，其中 0≤i<interfaces_coun 。在 interfaces[]中，各成员所表示的接口顺序和对应的源代码中给定的接口顺序（从左至右）一样即 interfaces[0]对应的是源代码中最左边的接口
fields_count：字段计数器，值表示当前 class 文件 fields 表的成员个数。fields 表中每个成员是一个 field_info 结构,用于表示该类或接口所声明的类字段或者实例字段
fields：字段表，fields 表中的每个成员都必须是一个 fields_info 结构的数据项，用于表示当前或接口中某个字段的完整描述,fields 表描述当前类或接口声明的所有字段，但不包括从父类或父接口承的那些字段
methods_count：方法计数器，值表示当前 class 文件 methods 表的成员个数。methods 表每个成员都是一个 method_info 结构_
methods：方法表，methods 表中的每个成员都必须是一个 method_info 结构，用于示当前类或接口中某个方法的完整描述，如果某个 method_info 结构的 access_fags 项既没有设置 CC NATIVE 标志也没有设置 ACC ABSTRACT 标志，那么该结构中也应包含实现这个法所用的 Java 虚拟机指令
attributes count：属性计数器，值表示当前 class 文件属性表的成员个数。属性中每项都是一个 attribute_info 结构
attributes：属性，每个项的值必须是 attribute_info 结构

<h2 id="类加载-链接和初始化">类加载、链接和初始化</h2>
加载类的流程

加载：查找并加载类文件的二进制数据
连接：就是将已经读入内存的类的二进制数据合并到 MM 运行时环境中去

验证：确保被加载类的正确性
准备：为类的静态变量分配内存，并初始化它们
解析：把常量池中的符号引用转换成直接引用

初始化：为类的静态变量赋初始值

<h3 id="类加载和类加载器">类加载和类加载器</h3>
<h4 id="类加载">类加载</h4>
类加载要完成的功能：

通过类的全限定名来获取该类的二进制字节流
把二进制字节流转化为方法区的运行时数据结构
在堆上创建一个 java.lang.Class 对象，用来封装类在方法区内的数据结构，并向外提供了访问法区内数据结构的接口

加载类的方式

最常见的方式：本地文件系统中加载、从 jar 等归档文件中加载
动态的方式：将 java 源文件动态编译成 lass
其它方式：网络下载、从专有数据库中加载等等

<h4 id="类加载器">类加载器</h4>

JVM 自带的加载器：

启动类加载器：BootstrapClassLoader
扩展类加载器：ExtensionClassLoader JDK8
平台类加载器：PlatformClassLoader JDK9 之后的版本

应用程序类加载器：AppClassloader

用户自定义的加载器：java.lang.ClassLoader 子类，用户定制类的加载方式；其加载的顺序是所有系统类加载器的最后

>
启动类加载器：用于加载启动的基础模块类，比如 java base、 Java.management、 Java.xml 等
平台类加载器：用于加载一些平台相关的模块，比如：java. scripting、java. compiler*、 java.crba 等等
应用程序类加载器：用于加载应用级别的模块，比如 jdk.compiler、jdk. jartool、jdk. jshe||等；还加载 classpath 路径中的所有类库
JDK8：

启动类加载器：负责将 < JAVA_HOME>/ib，或者 Xbootclasspath 参数指定的路径中的且是虚拟机识别的类库加载到内存中（按照名字识别，比如 rt.jar，对于不能识别的文件不予装载）</l>
扩展类加载器：负责加载 < JRE_HOME>/lib/ext，或者 java.ext.dirs 系统变量所指定路径的所有类库
应用程序类加载器：负责加载 classpath 路径中的所有类库

Java 程序不能直接引用启动类加载器，直接设置 classLoader 为 null，默认就使用启动类加载器
类加载器并不需要等到某个类“首次主动使用”的时候才加载它，JVM 规范允许类加载器在预料到某类将要被使用的时候就预先加载它
如果在加载的时 class 文件缺失，会在该类首次主动使用时报告 LinkageError，如一直没有被使用，不会报错
<h4 id="双亲委派模型">双亲委派模型</h4>
JVM 中的 ClassLoader 通常采用双亲委派模型，要求除了启动类加载器外，其余类加载器都应自己父级加载器
这里的父子关系是组合而不是继承
工作流程：

一个类加载器接收到类加载请求后，首先搜索它的内建加载器定义的所有"具名模块"
如果找到了合适的模块定义，将会使用该加载器来加载
如果 class 没有在这些加载器定义的具名模块中找到，那么将会委托给父级加载器，直到启动类载器
如果父级加载器反馈不能完成加载请求，如在它的搜索路径下找不到这个类，那子类加载器才自来加载
在类路径下找到的类将成为这些加载器的无名模块

双亲委派模型对于保证 Java 程序的稳定运作很重要
实现双亲委派的代码在 java. lang.classloader 的 loadClass()方法中，如果自定义类加载器的话推荐覆盖实现 findClass()
如果有一个类加载器能加载某个类，称为定义类加载器，所有能成功返回该类的 class 的类加载都被称为初始类加载器
如果没有指定父加载器，默认就是启动加载器
每个类加载器都有自己的命名空间，命名空间由该加载器及其所有父加载器所加载的类构成，不的命名空间，可以出现类的全路径名相同的情况
运行时包由同一个类加载器的类构成，决定两个类是否属于同一个运行时包，不仅要看全路径名否一样，还要看定义类加载器是否相同。只有属于同一个运行时包的类才能实现相互包内可见

<h4 id="破坏双亲委派模型">破坏双亲委派模型</h4>
双亲模型有个问题：父加载器无法向下识别子加载器加载的资源
为了解决这个问题，引入了线程上下文类加载器，可以通过 Thread 的 setContextClassLoader() 进行设置
另外一种典型情况就是实现热替换，比如 OSGI 的的模块化热部署，它的类加载器就不再是严格照双亲委派模型，很多可能就在平级的类加载器中执行了
<h3 id="类连接">类连接</h3>
<h4 id="类连接主要验证的内容">类连接主要验证的内容</h4>

类文件结构检查：按照 JVM 规范规定的类文件结构进行
元数据验证：对字节码描述的信息进行语义分析，保证其符合 Java 语言规范要求
字节码验证：对数据流和控制流进行分析，确保程序语义是合法和符合逻辑的。主要对方法体进校验
符号引用验证：对类自身以外的信息，也就是常量池中的各种符号引用，进行匹配校验

<h4 id="类连接中的解析">类连接中的解析</h4>

解析：把常量池中的符号引用转换成直接引用的过程，包括：符号引用：以一组无歧义的符号来述所引用的目标，与虚拟机的实现无关
直接引用：直接指向目标的指针、相对偏移量、或是能间接定位到目标的句柄，是和虚拟机实现关的
主要针对：类、接口、字段、类方法、接口方法、方法类型方法句柄、调用点限定符
如果类中存在初始化语句，就依次执行这些初始化语句
如果是接口的话：
初始化一个类的时候，并不会先初始化它实现的接口
初始化一个接口时，并不会初始化它的父接口
只有当程序首次使用接口里面的变量或者是调用接口方法的时候，才会导致接口初始化
调用 Classloader 类的 loadClass 方法来装载一个类，并不会初始化这个类，不是对类的主动使

<h4 id="类的初始化时机">类的初始化时机</h4>
Java 程序对类的使用方式分成：主动使用和被动使用
JVM 必须在每个类或接口“首次主动使用”时才初始化它们；被动使用类不会导致类的初始化
主动使用：

创建类实例
访问某个类或接口的静态变量
调用类的静态方法
反射某个类
初始化某个类的子类，而父类还没有初始化
JVM 启动的时候运行的主类
定义了 default 方法的接口，当接口实现类初始化时

<h4 id="类的卸载">类的卸载</h4>
当代表一个类的 Class 对象不再被引用，那 Class 对象的生命周期就结束了，对应在方法区中的据也会被卸载
JVM 自带的类加载器装载的类，是不会卸载的，由用户自定义的类加载器加载的类是可以卸载的
<h3 id="类的初始化">类的初始化</h3>
类的初始化：为类的静态变量赋初始值，或者说是执行类构造器 <code><clinit></code> 方法的过程
如果类还没有加载和连接，就先加载和连接
如果类存在父类，且父类没有初始化，就先初始化父类
<h2 id="Java内存分配">Java 内存分配</h2>
JVM 的简化架构

<h3 id="运行时数据区">运行时数据区</h3>
<h4 id="PC寄存器-程序计数器">PC 寄存器=程序计数器</h4>
PC 寄存器：Program Counter

每个线程拥有一个 PC 寄存器，是线程私有的，用来存储指向下一条指令的地址
在创建线程的时候，创建相应的 PC 寄存器
执行本地方法时，PC 寄存器的值为 undefined
一块较小的内存空间，是唯一一个在 JVM 规范中没有规定 OutOfMemoryError 的内存区域

<h4 id="Java栈">Java 栈</h4>
栈：由一系列帧（Frame）组成（因此 Java 栈也叫做帧栈），是线程私有的

帧用来保存一个方法的局部变量、操作数栈（Java 没有寄存器，所有参数传递使用操作数栈）、量池指针、动态链接、方法返回值等
每一次方法调用创建一个帧，并压栈，退出方法的时候，修改栈顶指针就可以把栈帧中的内容销
局部变量表存放了编译期可知的各种基本数据类型和引用类型，每个 slot 存放 32 位的数据，lon、double 占两个槽位

栈的优点：存取速度比堆快，仅次于寄存器
栈的缺点：存在栈中的数据大小、生存期是在编译期决定的，缺乏灵活性
<h4 id="Java堆">Java 堆</h4>

用来存放应用系统创建的对象和数组（new），所有线程共享 Java 堆
Java 垃圾回收（GC）主要就管理堆空间，对分代 GC 来说，堆也是分代的
Java 堆是在运行期动态分配内存大小，自动进行垃圾回收

堆的优点：运行期动态分配内存大小，自动进行垃圾回收
堆的缺点：效率相对较慢
Java 堆的结构：

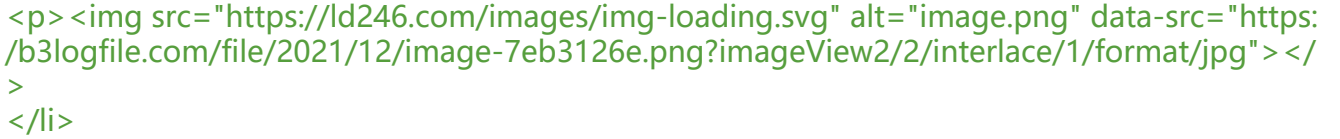
整个堆大小=新生代 + 老年代　新生代=Eden+ 存活区
新生代用来放新分配的对象；新生代中经过垃圾回收，没有回收掉的对象，被复制到老年代
老年代存储对象比新生代存储对象的年龄大得多，老年代存储一些大对象
从前的持久代，用来存放 Class、 Method 等元信息的区域，从 JDK8 开始去掉了，取而代之的元空间（ MetaSpace），元空间并不在虚拟机里面，而是直接使用本地内存
对象的内存布局
对象在内存中存储的布局（这里以 HotSpot 虚拟机为例来说明）分为：对象头、实例数据和对填充

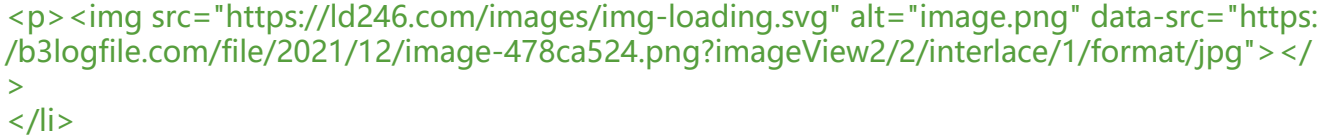
对象头
Mark Word：存储对象自身的运行数据，如：Hash Code、GC 分代年龄、锁状态标志等

类型指针：对象指向它的类元数据的指针

实例数据：真正存放对象实例数据的地方
对齐填充：这部分不一定存在，也没有什么特别含义，仅仅是占位符。因为 HotSpot 要求对象始地址都是 8 字节的整数倍，如果不是，就对齐

对象的访向定位
在 JVM 规范中只规定了 reference 类型是一个指向对象的引用，但没有规定这个引用具体如何定位、访问堆中对象的具体位置
因此对象的访问方式取决于 JVM 的实现，目前主流的有：使用句柄或使用指针两种方式

使用句柄：Java 堆中会划分出一块内存来做为句柄池 reference 中存储句柄的地址，句柄中存对象的实例数据和类元数据的地址


使用指针：Java 堆中会存放访问类元数据的地址， reference 存储的就直接是对象的地址


<h4 id="方法区">方法区</h4>

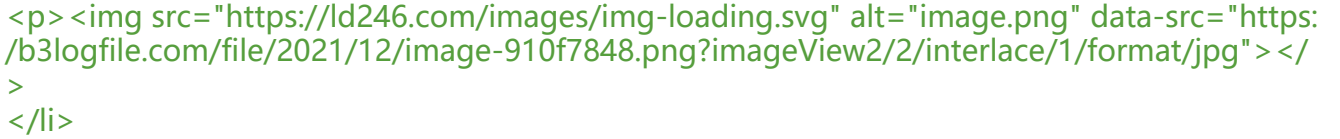
方法区是线程共享的，通常用来保存装载的类的结构信息
通常和元空间关联在一起，但具体的跟 JVM 实现和版本有关
JVM 规范把方法区描述为堆的一个逻辑部分，但它有一个别名称为 Non-heap（非堆），应是了与 Java 堆区分开

<h4 id="运行时常量池">运行时常量池</h4>

Class 文件中每个类或接口的常量池表，在运行期间的表示形式，通常包括：类的版本、字段、法、接口等信息
在方法区中分配
通常在加载类和接口到 JVM 后，就创建相应的运行时常量池

<h4 id="本地方法栈">本地方法栈</h4>

在 JVM 中用来支持 native 方法执行的栈就是本地方法栈

栈、堆、方法区之间的交互关系


<h3 id="JVM内存分配参数">JVM 内存分配参数</h3>
<h4 id="Trace跟踪参数">Trace 跟踪参数</h4>
打印 GC 的信息：-Xlog:gc*
指定 GClog 的位置，以文件输出：-X1og:gc:garbage-collection.log
每一次 GC 后，都打印堆信息：-Xlog:gc+heap=debug
GC 日志格式

GC 发生的时间，也就是 JVM 从启动以来经过的秒数

日志级别信息、日志类型标记
GC 识别号
GC 类型和说明 GC 的原因
容量：GC 前容量 -> GC 后容量该区域总容量）
GC 持续时间，单位秒。有的收集器会有更详细的描述，比如：user 表示应用程序消耗的时间，s
s 表示系统内核消耗的时间、real 表示操作从开始到结束的时间

<h4 id="Java堆的参数">Java 堆的参数</h4>
-Xms：初始堆大小，默认物理内存的 1/64
-Xmx：最大堆大小，默认物理内存的 1/4
-Xmn：新生代大小，默认整个堆的 3/8
-XX:+HeapDumpOnOutOfMemoryError：OOM 时导出堆到文件
-XX:HeapDumpPath=path：导出 OOM 的路径
-XX:OnOutOfMemoryError：OOM 时，执行一个脚本
-XX:NewRatio：老年代与新生代的比值，如果 Xms=xmx，且设置了 Xmn 的情况下，该参数
用设置
-XX:SurvivorRatio：Eden 区和 Survivor 区的大小比值，设置为 8，则两个 Survivo 区与一个 E
en 区的比值为 2:8，一个 Survivor 占整个新生的 1/10
<pre><code class="highlight-chroma"><span class="highlight
cl">-XX:+UseConcMarkSweepGC
-XX:+UseG1GC

-XX:MinHeapSize
8m
-XX:InitialHeapSiz
=9m
</code></pre>
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space：堆内存不够</
>
<h4 id="Java栈的参数">Java 栈的参数</h4>
-Xss：通常只有几百 K，决定了函数调用的深度
java.lang.StackOverflowError：方法递归调用
<h4 id="元空间的参数">元空间的参数</h4>
-XX:MetaspaceSize：初始空间大小
-XX:MaxMetaspaceSize：最大空间，默认是没有限制的
-XX:MinMetaspaceFreeRatio：在 GC 之后，最小的 Metaspace 剩余空间容量的百分比
-XX:MaxMetaspaceFreeRatio：在 GC 之后，最大的 Metaspace 剩余空间容量的百分比
<h2 id="字节码执行引擎">字节码执行引擎</h2>
<h3 id="字节码执行引擎概述">字节码执行引擎概述</h3>
JVM 的字节码执行引擎

基本功能：输入字节码文件，然后对字节码进行解析并处理，最后输出执行的结果
实现方式：
通过解释器直接解释执行字节码

通过即时编译器产生本地代码，也就是编译执行
两者皆有

栈帧概述

栈帧用于支持 JVM 进行方法调用和方法执行的数据结构
栈帧随着方法调用而创建，随着方法结束而销毁

栈帧里面存储了方法的局部变量、操作数栈、动态连接、方法返回地址等信息

栈帧结构

局部变量表：用来存放方法参数和方法内部定义的局部变量的存储空间
以变量槽 slot 为单位，目前一个 slot 存放 32 位以内的数据类型

对于 64 位的数据占 2 个 slot
对于实例方法，第 0 位 slot 存放的是 this，然后从 1 到 n，依次分配给参数列表
然后根据方法体内部定义的变量顺序和作用域来分配 slot
slot 是复用的，以节省栈帧的空间，这种设计可能会影响到系统的垃圾收集行为

操作数栈：用来存放方法运行期间，各个指令操作的数据
操作数栈中元素的数据类型必须和字节码指令的顺序严格匹配

虚拟机在实现栈帧的时候可能会做一些优化，让两个栈帧出现部分重叠区域，以存放公用的数据

动态连接：每个栈帧持有一个指向运行时常量池中该栈帧所属方法的引用，以支持方法调用过程动态连接
静态解析：类加载的时候，符号引用就转化成直接引用

动态连接：运行期间转换为直接引用

方法返回地址 ： 方法执行后返回的地址

<h3 id="方法调用-静态分派和动态分派">方法调用、静态分派和动态分派</h3>
方法调用：方法调用就是确定具体调用那-个方法，并不涉及方法内部的执行过程

部分方法是直接在类加载的解析阶段，就确定了直接引用关系
对于实例方法，也称虚方法，因为重载和多态，需要运行期动态委派

分派：单分派和多分派：就是按照分派思考的纬度，多余一个的就算多分派，只有一个的称为单派

静态分派：所有依赖静态类型来定位方法执行版本的分派方式，比如：重载方法
动态分派：根据运行期的实际类型来定位方法执行版本的分派方式，比如：覆盖方法

<h2 id="垃圾回收">垃圾回收</h2>
<h3 id="垃圾回收基础">垃圾回收基础</h3>
<h4 id="什么是垃圾">什么是垃圾</h4>
简单说就是内存中已经不再被使用到的内存空间就是垃圾
<h4 id="如何判定是垃圾">如何判定是垃圾</h4>

引用计数法

原理：给对象添加一个引用计数器，有访问就加 1，引用失效就减 1，计数器有值说明还在被引，就不是垃圾

优点：实现简单、效率高
缺点：不能解决对象之间循环引用的问题

根搜索算法

原理：从根(GC Roots)节点向下搜索对象节点，搜索走过的路经称为引用链，当一个对象到根之没有连通的话，则该对象不可用

可作为 GC Roots 对象

虚拟机栈（栈帧局部变量）中引用的对象

方法区类静态属性引用的对象
方法区中常量引用的对象
本地方法栈中 JNI(Native 方法) 引用的对象
被同步锁 synchronized 修饰的对象

OopMap

Hotspot 使用了一组叫做 OopMap 的数据结构达到准确式 GC 的目的，不用每次从根节点查找

在 OopMap 的协助下，JVM 可以很快的做完 GC Roots 枚举。但是 JVM 并没有为每一条指令成一个 OopMap
记录 OopMap 的这些“特定位置”被称为安全点，即当前线程执行到安全点后才允许暂停进行C
如果一段代码中，对象引用关系不会发生变化，这个区域中任何地方开始 GC 都是安全的，那么个区域称为安全区域

引用分类

强引用：类似于 Object a=newA()，不会被回收

软引用：还有用但并不必须的对象。用 SoftReference 来实现软引用
弱引用：非必须对象，比软引用还要弱，垃圾回收时会回收掉。用 WeakReference 来实现弱引
虚引用：也称为幽灵引用或幻影引用，是最弱的引用。垃圾回收时会回收掉。用 PhantomReference 来实现虚引用

跨代引用：也就是一个代中的对象引用另一个代中的对象

跨代引用假说：跨代引用相对于同代引用来说只是极少数

隐含推论：存在互相引用关系的两个对象，是应该倾向于同时生存或同时消亡

记忆集（Remembered set）：一种用于记录从非收集区域指向收集区域的指针集合的抽象数结构（全局）

精度

字长精度：每个记录精确到一个机器字长，该字包含跨代指针

对象精度：每个记录精确到一个对象，该对象里有字段含有跨代指针
卡精度：每个记录精确到一块内存区域，该区域内有对象含有跨代指针

卡表（Card Table）：记忆集的一种具体实现，定义了记忆集的记录精度和与堆内存的映射关等

卡表的每个元素都对应着其标识的内存区域中一块特定大小的内存块，这个内存块称为卡页（Cad Page）

写屏障维护卡表状态：记忆集数据发生改变时
写屏障可以看成是 JVM 对“引用类型字段赋值”这个动作的 AOP

通过写屏障来实现当对象状态改变后，维护卡表状态

判断是否垃圾的步骤

根搜索算法判断不可用
看是否有必要执行 finalize 方法
两个步骤走完后对象仍然没有人使用，那就属于垃圾

<h4 id="如何回收">如何回收</h4>

GC 类型

MinorGC/YoungGC：发生在新生代的收集动作
MajorGC/OldGC：发生在老年代的 GC，目前只有 CMS 收集器会有单独收集老年代的行为
MixedGC：收集整个新生代以及部分老年代，目前只有 G1 收集器会有这种行为
FullGC：收集整个 Java 堆和方法区的 GC

Stop-The-World

STW 是 Java 中一种全局暂停的现象，多半由于 GC 引起。所谓全局停顿，就是所有 Java 代码止运行， native 代码可以执行，但不能和 JVM 交互
危害是长时间服务停止，没有响应；对于 HA 系统，可能引起主备切换，严重危害生产环境

垃圾收集类型

串行收集：GC 单线程内存回收、会暂停所有的用户线程，如：Serial
并行收集：多个 GC 线程并发工作，此时用户线程是暂停，如：Parallel
并发收集：用户线程和 GC 线程同时执行（不一定是并行，可能交替执行），不需要停顿用户线，如：CMS

判断类无用的条件

JVM 中该类的所有实例都已经被回收
加载该类的 Classloader 已经被回收
没有任何地方引用该类的 class 对象
无法在任何地方通过反射访问这个类

<h3 id="垃圾回收算法-方法">垃圾回收算法-方法</h3>
<h4 id="标记清除法-Mark-Sweep-">标记清除法（Mark-Sweep）</h4>

原理：标记和清除两个阶段，先标记出要回收的对象，然后统一回收这些对象
优点：简单
缺点：
效率不高，标记和清除的效率都不高

标记清除后会产生大量不连续的内存碎片，从而导致在分配大对象时触发 GC

<h4 id="复制算法-Copying-">复制算法（Copying）</h4>

原理：把内存分成两块完全相同的区域每次使用其中一块，当一块使用完了，就把这块上还存活对象拷贝到另外一块，然后把这块清除掉
优点：实现简单，运行高效，不用考虑内存碎片问题
缺点：内存有些浪费
实际
JVM 实际实现中，是将内存分为一块较大的 Eden 区和两块较小的 Survivor 空间，每次使用 Edn 和块 Survivor，回收时，把存活的对象复制到另一块 Survivor

Hotspot 默认的 Eden 和 Survivor 比是 8:1，也就是每次能用 90% 的新生代空间
如果 Survivor 空间不够，就要依赖老年代进行分配担保，把放不下的对象直接进入老年代
分配担保：当新生代进行垃圾回收后，新生代的存活区放置不下，那么需要把这些对象放置到老代去的策略，也就是老年代为新生代的 GC 做空间分配担保
在发生 MinorgGC 前，JVM 会检查老年代的最大可用的连续空间，是否大于新生代所有对象的空间，如果大于，可以确保 MinorGC 安全

如果小于，那么 JVM 会检查是否设置了允许担保失败，如果允许，则继续检查老年代最大可用连续空间，是否大于历次晋升到老年代对象的平均大小

如果大于，则尝试进行一次 MinorGC；如果不大于，则改做一次 FullGC

<h4 id="标记整理法-Mark-Compact-">标记整理法（Mark-Compact）</h4>

原理：

由于复制算法在存活对象比较多的时候，效率较低，且有空间浪费，因此老年代般不会选用复制法，老年代多选用标记整理算法

标记过程跟标记清除一样，但后续不是直接清除可回收对象而是让所有存活对象都向一端移动，后直接清除边界以外的内存

<h3 id="垃圾收集器-实现">垃圾收集器-实现</h3>
HotSpot 中的收集器

新生代 Young generation：Serial、ParNew、Parallel Scavenge、G1
老年代 Tenured generation：CMS、Serial old、Parallel old、G1

<h4 id="串行收集器">串行收集器</h4>
Serial（串行）收集器/ Serial old 收集器，是一个单线程的收集器，在垃圾收集时，会 Stop-the World
Serial/ Serial Old 收集器运行示意图：
优点：简单，对于单 CPU，由于没有多线程的交互开销，可能更高效，是默认的 Client 模式下

新生代收集器
使用：-XX:+UseSerialGC 来开启，会使用:Serial+ Serial Old 的收集器组合（新生代使用复制法，老年代使用标记-整理算法）
<h4 id="并行收集器">并行收集器</h4>
ParDew（并行）收集器：使用多线程进行垃圾回收，在垃圾收集时，会 Stop-the-World
ParDew 收集器运行示意图：

在并发能力好的 CPU 环境里，它停顿的时间要比串行收集器短；但对于单 CPU 或并发能力较弱 CPU，由于多线程的交互开销，可能比串行回收器更差
Server 模式下首选的新生代收集器，且能和 CMS 收集器配合使用
不再使用-XX:+UseParNewGo 来单独开启，使用 CMS 即可
-XX:ParallelGCThreads：指定线程数，最好与 CPU 数量一致
<h4 id="新生代-Parallel-Scavenge收集器">新生代 Parallel Scavenge 收集器</h4>
新生代 Parallel Scavenge 收集器 / Parallel old 收集器：是一个应用于新生代的、使用复制算的、并行的收集器
跟 ParNew 很类似，但更关注吞吐量，能最高效率的利用 CPU，适台运行后台应用
新生代 Parallel Scavenge/Parallel old 收集器运行示意图：

使用：-XX:+UseParallelGC 来开启；使用-XX:+UseParallelOldGC 来开启老年代使用 Parallel Od 收集器，使用 Parallel Scavenge+ Parallel old 的收集器组合
-XX:MaxGCPauseMillis：设置 GC 的最大停顿时间
<h4 id="CMS">CMS</h4>
CMS（Concurrent Mark and Sweep 并发标记清除）收集器分为：
初始标记：只标记 GC Roots 能直接关联到的对象
并发标记：进行 GC Roots Tracing 的过程
重新标记：修正并发标记期间，因程序运行导致标记发生变化的那一部分对象
并发清除：并发回收垃圾对象
CMS 收集器运行示意图：

在初识标记和重置标记阶段还是可能发生 Stop-the-World
使用标记-清除算法，多线程并发收集的垃圾收集器
重置线程是清空跟收集相关数据并重置，为下一次收集做准备
优点：低停顿，并发执行
缺点：

并发执行，对 CPU 资源压力大
无法处理在处理过程中产生的垃圾，可能导致 FullGC
采用标记-清除算法会产生大量碎片，从而在分配大对象时可能会触发 FullGC

使用：
-XX:+UseConcMarkSweepGC：使用 ParNew + CMS + Serial old 的收集器组合，Serial Old 将作为 CMS 出错的后备收集器
-XX:CMSInitiatingOccupancyFraction：设置 CMS 收集器在老年代空间被使用多少后触发回，默认 80%
<h4 id="G1">G1</h4>
G1（Garbage-Frst）收集器：一款面向服务端应用的收集器
特点

G1 把内存划分成多个独立的区域（Region）

G1 仍采用分代思想，保留了新生代和老年代，但它们不再是物理隔离的，而是一部分 Region 集合，且不需要 Region 是连续的

G1 能充分利用多 CPU、多核环境硬件优势，尽量缩短 STW

G1 整体上采用标记-整理算法，局部是通过复制算法，不会产生内存碎片

G1 的停顿可预测，能明确指定在一个时间段内，消耗在垃圾收集上的时间不能超过多长时间</p

G1 跟踪各个 Region 里面垃圾堆的价值大小（回收能清理除更多空间），在后台维护一个优先表，每次根据允许的时间来回收价值最大的区域，从而保证在有限时间内的高效收集

G1 收集器新生代回收过程：
G1 收集器老年代回收过程：
原理

初始标记：只标记 GC Roots 能直接关联到的对象
并发标记：进行 GC Roots Tracing 的过程
最终标记：修正并发标记期间，因程序运行导致标记发生变化的那一部分对象
筛选回收：根据时间来进行价值最大化的回收

开启 G1：-XX:+UseG1GC
-XX:MaxGCPauseMillis=time：最大 GC 停顿时间，这是个软目标，M 将尽可能（但不保证）顿小于这个时间
-XX:InitiatingHeapOccupancyPercent：堆占用了多少的时候就触发 GC，默认为 45
-XX:NewRatio=ratio：默认为 2
-XX:SurvivorRatio=ratio：默认为 8
-XX:MaxTenuringThreshold=threshold：新生代到老年代的岁数，默认是 15
-XX:ParallelGCThreads=threads：并行 GC 的线程数，默认值会根据平台不同而不同
-XX:ConcGCThreads=threads：并发 GC 使用的线程数
-XX:G1ReservePercent=percent：设置作为空闲空间的预留内存百分比，以降低目标空间溢出风险，默认值是 10%
-XX:G1HeapRegionSize=size：设置的 G1 区域的大小。值是 2 的幂，范围是 1MB 到 32MB 目标是根据最小的 Java 堆大小划分出约 2048 个区域
<h4 id="ZGC收集器">ZGC 收集器</h4>
ZGC 收集器：JDK11 加入的具有实验性质的低延迟收集器
设计目标：支持 TB 级内存容量，暂停时间低（<10ms），对整个程序吞吐量的影响小于 15%
新技术：着色指针和读屏障
<h3 id="GC性能指标和JVM内存配置原则">GC 性能指标和 JVM 内存配置原则</h3>
GC 性能指标

吞吐量 = 应用代码执行的时间 / 运行的总时间
GC 负荷：与吞吐量相反，是 GC 时间 / 运行的总时间
暂停时间：发生 Stop-the-World 的总时间
GC 频率：GC 在一个时间段发生的次数
反应速度：从对象成为垃圾到被回收的时间（交互式应用通常希望暂停时间越少越好）

JVM 内存配置原则

新生代：尽可能设置大点，如果太小会导致
新生代垃圾回收(YGC)次数更加频繁

可能导致 YGC 后的对象进入老年代，如果此时老年代满了，会触发 FGC

老年代
针对响应时间优先的应用：由于老年代通常采用并发收集器，因此其大小要综合考虑发量和并发持续时间等参数.如果设置小了，可能会造成内存碎片，回收频率会导致应用暂停;如果设置大了，会需要较长的回收时间

针对吞吐量优先的应用：通常设置较大的新生代和较小的老年代，这样可以尽可能回收大部分短对象，减少中期对象，而老年代尽量存放长期存活的对象

依据对象的存活周期进行分类，对象优先在新生代分配，长时间存活的对象进入老年代
根据不同代的特点，选取合适的收集算法：少量对象存活，适合复制算法；大量对象存活，适合记清除或者标记整理

<h2 id="JVM支持高效并发">JVM 支持高效并发</h2>
<h3 id="内存模型">内存模型</h3>
内存模型：在特定的操作协议下，对特定的内存或高速缓存进行读写访问的过程抽象
Java 内存模型主要关注 JVM 中把变量值存储到内存和从内存中取出变量值这样的底层细节
所有变量（共享的）都存储在主内存中，每个线程都有自己的工作内存；工作内存中保存该线程用到的变量的主内存副本拷贝
线程对变量的所有操作（读、写）都应该在工作内存中完成
不同线程不能相互访问工作内存，交互数据要通过主内存
<h3 id="内存间的交互操作">内存间的交互操作</h3>
Java 内存模型规定了一些操作来实现内存间交互，JVM 会保证它们是原子的
lock：锁定，把变量标识为线程独占，作用于主内存变量
unlock：解锁，把锁定的变量释放，别的线程才能使用，作用于主内存变量
read：读取，把变量值从主内存读取到工作内存
load：载入，把 read 读取到的值放入工作内存的变量副本中
use：使用，把工作内存中一个变量的值传递给执行引擎
assign：赋值，把从执行引擎接收到的值赋给工作内存里面的变量
store：存储，把工作内存中一个变量的值传递到主内存中
write：写入，把 store 进来的数据存放如主内存的变量中

<h3 id="内存间交互的规则">内存间交互的规则</h3>

不允许 read 和 load、 store 和 write 操作单独出现，以上两个操作必须按顺序执行，不保证连执行，也就是说 read 与 load 之间、 store 与 write 之间是可插入其他指令
不允许一个线程丢弃它的最近的 assign 操作，即变量在工作内存中改变了之后必须把该变化同

回主内存
不允许一个线程无原因地（没有发生过任何 assign 操作）把数据从线程的工作内存同步回主内中
一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化的变量，就是对一个变量实施 use 和 store 操作之前，必须先执行过了 assign 和 load 操作
一个变量在同一个时刻只允许一条线程对其执行 lock 操作，但 lock 操作可以被同一个条线程重执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁
如果对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行 load 或 assign 操作初始化变量的值
如果一个变量没有被 lock 操作锁定，则不允许对它执行 unlock 操作，也不能 unlock 个被其他程锁定的变量
对一个变量执行 unlock 操作之前，必须先把此变量同步回主内存（执行 store 和 write 操作）</i>

<h3 id="多线程的可见性">多线程的可见性</h3>
可见性：一个线程修改了变量，其他线程可以知道
保证可见性的常见方法：volatile、 synchronized、final（一旦初始化完成，其他线程就可见）/p>
<h4 id="volatile">volatile</h4>
volatile：JVM 提供的最轻量级的同步机制，用 volatile 修饰的变量，对所有的线程可见，即对olatile 变量所做的写操作能立即反映到其它线程中。用 volatile 修饰的变量，在多线程环境下仍然是安全的
volatile 修饰的变量，是禁止指令重排优化
适合使用 valatile 的场景：
运算结果不依赖变量的当前值
或者能确保只有一个线程修改变量的值
指令重排：JVM 为了优化，在条件允许的情况下，对指令进行一定的重新排列，直接运行当前够立即执行的后续指令，避开获取下一条指令所需的等待时间
线程内串行语义，不考虑多线程间的语义
不是所有的指令都能重拍，比如：
写后读 a=1；b=a；写一个变量之后，再读这个位置
写后写 a=1；a=2；写一个变量之后，再写这个变量
读后写 a=b；b=1；读一个变量之后，再写这个变量
a=1；b=2；是可以重排的
指令重拍的基本规则

程序顺序原则：一个线程内保证语义的串行性
volatile 规则：volatile 变量的写，先发生于读
锁规则：解锁（unlock）必然发生在随后的加锁（lock）前
传递性：A 先于 B，B 先于 C 那么 A 必然先于 C
线程的 start 方法先于它的每一个动作
线程的所有操作先于线程的终结（Thread.join()）
线程的中断 interrupt() 先于被中断线程的代码
对象的构造函数执行结束先于 finalize()

Java 线程安全的处理方法

不可变是线程安全的

互斥同步（阻塞同步）：synchronized、java.util.concurrent.ReentrantLock。目前这两个方性能已经差不多了，建议优先选用 synchronized
ReentrantLock 增加了如下特性：

等待可中断：当持有锁的线程长时间不释放锁，正在等待的线程可以选择放弃等待
公平锁：多个线程等待同一个锁时，须严格按照申请锁的时间顺序来获得锁
锁绑定多个条件：一个 ReentrantLock 对象可以绑定多个 condition 对象，而 synchronized 针对一个条件的，如果要多个，就得有多个锁

非阻塞同步：一种基于冲突检查的乐观锁定策略，通常是先操作，如果没有冲突，操作就成功了有冲突再采取其它方式进行补偿处理

无同步方案：其实就是在多线程中，方法并不涉及共享数据，自然也就无需同步了

锁优化之自旋锁与自适应自旋
自旋：如果线程可以很快获得锁，那么可以不在 OS 层挂起线程，而是让线程做几个忙循环，这是自旋
自适应自旋：自旋的时间不再固定，而是由前一次在同一个锁上的自旋时间和锁的拥有者状态来定
如果锁被占用时间很短，自旋成功，那么能节省线程挂起、以及切换时间，从而提升系统性能</>
如果锁被占用时间很长，自旋失败，会白白耗费处理器资源，降低系统性能
锁消除
在编译代码的时候，检测到根本不存在共享数据竞争，自然也就无需同步加锁了；通过-XX:+EliminateLocks 来开启。同时要使用-XX:+DoEscapeAnalysis 开启逃逸分析，所谓逃逸分析：1.如果一方法中定义的一个对象，可能被外部方法引用，称为方法逃逸；2.如果对象可能被其它外部线程访问称为线程逃逸，比如赋值给类变量或者可以在其它线程中访问的实例变量
锁粗化
通常我们都要求同步块要小，但一系列连续的操作导致对一个对象反复的加锁和解锁，这会导致必要的性能损耗。这种情况建议把锁同步的范围加大到整个操作序列
轻量级锁
轻量级是相对于传统锁机制而言，本意是没有多线程竞争的情况下，减少传统锁机制使用 OS 实互斥所产生的性能损耗。实现原理很简单，就是类似乐观锁的方式
如果轻量级锁失败，表示存在竞争，升级为重量级锁，导致性能下降
偏向锁
偏向锁是在无竞争情况下，直接把整个同步消除了，连乐观锁都不用，从而提高性能；所谓的偏，就是偏心，即锁会偏向于当前已经占有锁的线程
只要没有竞争，获得偏向锁的线程，在将来进入同步块，也不需要做同步
当有其它线程请求相同的锁时，偏向模式结束
如果程序中大多数锁总是被多个线程访问的时候，也就是竞争比较激烈，偏向锁反而会降低性能
使用-XX:-UseBiasedLocking 来禁用偏向锁，默认开启
JVM 中获取锁的步骤
会先尝试偏向锁；然后尝试轻量级锁
再然后尝试自旋锁
最后尝试普通锁，使用 OS 互斥量在操作系统层挂起
同步代码的基本规则
尽量减少锁持有的时间
尽量减小锁的粒度
<h2 id="JVM性能监控">JVM 性能监控</h2>
JVM 监控工具的作用

对 JVM 运行期间的内部情况进行监控，比如：对 JVM 参数、CPU、内存、堆等信息的查看

辅助进行性能调优
辅助解决应用运行时的一些问题，比如：OutOfMemoryError、内存泄露、线程死锁、锁争用、Java 进程消耗 CPU 过高等

命令行工具

jps：JVM Process Status Tool，主要用来输出 JVM 中运行的进程状态信息，语法格式如下：jps [options] [hostid]
hostid 字符串的语法与 URI 的语法基本一致：[protocol:][/]hostname][：port][/servername]
如果不指定 hostid，默认为当前主机或服务器

jinfo：打印给定进程或核心文件或远程调试服务器的配置信息。语法格式：jinfo [option] pid #指定进程号（pid）的进程

jstack：主要用来查看某个 Java 进程内的线程堆栈信息。语法格式如下：jstack [option] pid

jmap：用来查看堆内存使用状况，语法格式如下；jmap [option] pid
jmap [option] executable core
jmap [option][server-id@]remote-hostname-or-ip

jstat：JVM 统计监测工具，查看各个区内存和 GC 的情况

jstatd：jstat [generalOption | outputOptions vmid [interval[s|ms][count]]]
虚拟机的 jstat 守护进程，主要用于监控 JVM 的创建与终止，并提供一个接口，以允许远程监工具附加到在本地系统上运行的 JVM

jcmd：JVM 诊断命令工具，将诊断命令请求发送到正在运行的 Java 虚拟机，比如可以用来导堆，查看 java 进程，导出线程信息，执行 GC 等

图形化工具

jconsole：一个用于监视 Java 虚拟机的符合 JMX 的图形工具。它可以监视本地和远程 JVM，可以监视和管理应用程序

jmc：JDK Mission Control，Java 任务控制（JMC）客户端包括用于监视和管理 Java 应用程的工具，而不会引入通常与这些类型的工具相关联的性能开销

visualvm：一个图形工具，它提供有关在 Java 虚拟机中运行的基于 Java 技术的应用程序的详信息
Java VisualVM 提供内存和 CPU 分析，堆转储分析，内存泄漏检测，访问 MBean 和垃圾回收

两种连接方式：JMX、jstatd

JMX 连接：可以查看：系统信息、CPU 使用情况、线程多少、手动执行垃圾回收等比较偏于系级层面的信息
jstatd 连接方式可以提供：JVM 内存分布详细信息、垃圾回收分布图、线程详细信息，甚至可以到某个对象使用内存的大小

<h2 id="JVM性能调优">JVM 性能调优</h2>
JVM 调优：
调什么

内存方面
JVM 需要的内存总大小

各块内存分配，新生代、老年代、存活区
选择合适的垃圾回收算法、控制 GC 停顿次数和时间
解决内存泄露的问题，辅助代码优化
内存热点：检查哪些对象在系统中数量最大，辅助代码优化

线程方面
死锁检查，辅助代码优化

Dump 线程详细信息：查看线程内部运行情况，查找竞争线程，辅助代码优化
CPU 热点：检查系统哪些方法占用了大量 CPU 时间，辅助代码优化

如何调

监控 JVM 的状态，主要是内存、线程、代码、I/O 几部分
分析结果，判断是否需要优化
调整：垃圾回收算法和内存分配；修改并优化代码
不断的重复监控、分析和调整，直至找到优化的平衡点

调的目标是什么

GC 的时间足够的小
GC 的次数足够的少
将转移到老年代的对象数量降低到最小
减少 Full GC 的执行时间
发生 Full GC 的间隔足够的长

JVM 调优策略

减少创建对象的数量
减少使用全局变量和大对象
调整新生代、老年代的大小到最合适
选择合适的 GC 收集器，并设置合理的参数

调优冷思考

多数的 Java 应用不需要在服务器上进行 GC 优化

多数导致 GC 问题的 Java 应用，都不是因为参数设置错误，而是代码问题
在应用上线之前，先考虑将机器的 JVM 参数设置到最优（最适合）
JVM 优化是到最后不得已才采用的手段
在实际使用中，分析 JVM 情况优化代码比优化 JVM 本身要多得多
如下情况通常不用优化：
Minor GC 执行时间不到 50ms

Minor GC 执行不频繁，约 10 秒一次
Full GC 执行时间不到 1s
Full GC 执行频率不算频繁，不低于 10 分钟 1 次

调优经验

要注意 client 模式和 Server 模式的选择
要想 GC 时间小必须要一个更小的堆；而要保证 GC 次数足够少，又必须保证一个更大的堆，这个是有冲突的，只能取其平衡
针对 JVM 堆的设置，一般可以通过-Xms -Xmx 限定其最小、最大值，为了防止垃圾收集器在最、最大之间收缩堆而产生额外的时间，通常把最大、最小设置为相同的值
新生代和老年代将根据默认的比例（1:2）分配堆内存，可以通过调整二者之间的比率 NewRadio 来调整，也可以通过-XX:newSize-XX:MaxNewSize 来设置其绝对大小，同样，为了防止新生的堆收，通常会把 -XX:newSize -XX:MaxNewSize 设置为同样大小
合理规划新生代和老年代的大小
如果应用存在大量的临时对象，应该选择更大的新生代；如果存在相对较多的持久对象，老年代该适当增大。在抉择时应该本着 Full GC 尽量少的原则，让老年代尽量缓存常用对象，JVM 的默认比 1:2 也是这个道理
通过观察应用一段时间，看其在峰值时老年代会占多少内存，在不影响 Full GC 的前提下，根据际情况加大新生代，但应该给老年代至少预留 1/3 的增长空间
线程堆栈的设置：每个线程默认会开启 1M 的堆栈，用于存放栈帧、调用参数、局部变量等，对多数应用而言这个默认值太大了，一般 256K 就足用。在内存不变的情况下，减少每个线程的堆栈，以产生更多的线程

分析和处理内存溢出
内存泄露导致系统崩溃前的一些现象，比如：

每次垃圾回收的时间越来越长，Full GC 时间也延长到好几秒
Full GC 的次数越来越多，最频繁时隔不到 1 分钟就进行一次 Full GC
老年代的内存越来越大，并且每次 Full　GC 后年老代没有内存被释放
老年代堆空间被占满的情况：根据垃圾回收前后情况对比，同时根据对象引用情况分析，辅助去找泄漏点
堆栈溢出的情况：通常抛出 java.lang.StackOverflowError 例外，一般就是递归调用没退出，或循环调用造成

重点是调优的过程、方法和思路
内存调整、数据库连接调整、内存泄漏查找等
<h2 id="面试题">面试题</h2>
知道字节码吗？字节码都有哪些？Integerx=5，inty=5，比较 x==y 都经过哪些步骤？
Java 虚拟机指令集：
类加载、连接和初始化部分
简述 Java 的类加载机制，并回答一个 JVM 中可否存在两个相同的类
1.2.1.9
讲讲类加载机制，都有哪些类加载器，这些类加载器都加载哪些文件？
说说类加载、连接和初始化的过程

内存分配
谈谈 JVM 内存模型
JVM 的数据区有哪些，作用是什么？
Java 堆内存一定是线程共享的吗？
<pre><code class="highlight-chroma">为对象分配内存的基本方法：指针碰撞法、空闲列表法
</code></pre>
<pre><code class="highlight-chroma">内存分配并发问题的解决：CAS、TLAB
</code></pre>
JVM 堆内存结构是怎样的？哪些情况会触发 GC？会触发哪些 GC？
垃圾回收
说一说 JVM 的垃圾回收
JVM 四种引用类型
JVM 回收算法和垃圾收集器
监控工具和实战
如何把 Java 内存的数据全部 dump 出来
Jstack 是做什么的？Jstat 呢？
如何定位问题？如何解决问题？说一下解决思路和处理方法
CPU 使用率过高怎么办？
线上应用频繁 Full GC 如何处理？
如果应用周期性地出现卡顿，你会怎么来排查这个问题？
你有没有遇到过 OutOfMemory 问题？你是怎么来处理这个问题的？
StackOverFlow 异常有没有遇到过？这个异常会在什么情况下被触发？如何指定线程堆栈的大？
JVM 内存结构说一下，各个部分的作用
介绍 JVM 内存模型，Java 运行时的内存模型
垃圾回收算法
垃圾回收期器
G1 垃圾回收器
对象内存分配被垃圾回收器管理吗，为什么
看过 JVM 源码吗
Java 的四种引用类型？
双亲委派模型是什么？有什么好处？
gc 一定会停顿吗，不一定，Epsilongc
JVM 组成，每块功能，new 一个对象过程
GC 有哪些方式
垃圾回收算法，了解到的都说一下
JVM 如何调优
JVM 内存模型
JVM 垃圾回收算法
JVM 垃圾回收器
CMS、G1 的设计思路、关联和区别、垃圾回收阶段的不同
让你设计系统中进行选择其中一个回收器，你的想法是什么
ASM 是什么 （字节码增强器）
Java 里面的类加载器的设计
类加载器的类之间的可见性 （委托机制、单一性、可见性）
如果父级对子级进行调用，会出现什么异常
双亲委派模型
讲下 JVM 内存模型，CMS 垃圾回收器
类加载机制说一说,类加载过程是不是线程安全的
Class.forName()和 ClassLoader 的区别