1. JVM和操作系统的关系
2. JVM和内核的关系
3. JVM和I/O的关系
4. 老生常谈：JVM需要清楚的几个要点
5. JMM：java内存模型

用户线程：内核线程=1:1

1. Volatile原语
2. AQS
3. 强引用、软引用、弱引用、虚引用
4. 类加载、类卸载
5. GC：垃圾收集
6. 算法
7. 可触及性

根对象

　　什么是根对象呢？主要有以下三类：

　　1、栈中引用的对象；

　　2、方法区静态成员或者常量引用的对象（全局变量）；

　　3、JNI方法栈中引用的对象

1. 收集器类型
2. Parallel

2）大火的G1

1. 并发编程
2. 线程池

ThreadPoolExecutor、Executors工具类、Fork/Join

1. 多线程的利与弊
2. 线程同步与常见的锁
3. 常见的编程工具：Semorphore|CountDownLatch|BlockBarrier
4. 可不可以不用并发提高性能？

当然可以。

1）纤程：

java原生不支持纤程，介绍一款第三方实现：Quarsar。

2）Actor模型。介绍一种在工业上大规模使用的成熟实现Akka/Actor

1. JVM发展方向

2、

5）类加载、类卸载

我们编写完一个.java结尾的源文件后，经过编译后生成对应的一个或多个.class后缀结尾的文件。该文件也称为字节码文件，能在java虚拟机中运行。而类的生命周期正是：从类（.class文件）被加载到虚拟机内存，到从内存中卸载为止。整个周期一共分为7个阶段：

加载，验证，准备，解析，初始化，使用，卸载

类加载

Java中的代码都写在类里面，源代码（.java文件）经过编译后会变成字节码（.class文件），当Java应用程序运行时，JVM会加载要用到的类。  
负责类加载的是java.lang.ClassLoader的对象，ClassLoader的对象会根据类的binary name（不知道怎么翻译，二进制名？）来读取类的字节码，然后把这些字节码加载到JVM的方法区里，同时会在heap里面创建一个Class对象，这个Class对象是类的元数据和它的对象通信的接口。类在heap中的Class对象和加载它的ClassLoader对象之间有强引用。

类卸载

不像类加载，Java中没有提供显式进行类卸载的API，但是如果加载类的ClassLoader对象被垃圾回收器回收的话，这个类就会被卸载。所以我们可以自己实现ClassLoader，自己加载类，然后对ClassLoader对象的引用赋值为null，等ClassLoader对象剩下的引用数量为0时会被回收，这样就达到卸载类的目的了。

详细介绍

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。

4、

3）锁

从源码看lock。

private void printStackTrace(PrintStreamOrWriter s) {  
 // Guard against malicious overrides of Throwable.equals by  
 // using a Set with identity equality semantics.  
 Set<Throwable> dejaVu =  
 Collections.*newSetFromMap*(new IdentityHashMap<Throwable, Boolean>());  
 dejaVu.add(this);  
  
 synchronized (s.lock()) {  
 // Print our stack trace  
 s.println(this);  
 StackTraceElement[] trace = getOurStackTrace();  
 for (StackTraceElement traceElement : trace)  
 s.println("\tat " + traceElement);  
  
 // Print suppressed exceptions, if any  
 for (Throwable se : getSuppressed())  
 se.printEnclosedStackTrace(s, trace, *SUPPRESSED\_CAPTION*, "\t", dejaVu);  
  
 // Print cause, if any  
 Throwable ourCause = getCause();  
 if (ourCause != null)  
 ourCause.printEnclosedStackTrace(s, trace, *CAUSE\_CAPTION*, "", dejaVu);  
 }  
}

由源码可知，e.printStackTrace使用了锁机制，因此在多线程并发环境下出现锁的争抢，从而降低性能。因此并发环境下不要使用该方法，而应该使用专门的日志工具来记录异常，例如：log4j2或logback。

JVM官方文档

https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html

OpenJDK官方文档

https://openjdk.java.net/groups/build/doc/building.html

如何编译自己的Openjdk

https://blog.csdn.net/wd2014610/article/details/81664062

<https://blog.csdn.net/wd2014610/article/details/81703203>

知乎高分回答JVM理解

<https://zhuanlan.zhihu.com/p/102914946>

https://zhuanlan.zhihu.com/p/208531033

Jdk源码学习方法：

https://www.zhihu.com/question/19840538

### **二、垃圾回收相关文档**

#### **官方文档**

1. <https://blogs.oracle.com/poonam/entry/understanding_cms_gc_logs>
2. <http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>

除了上面这些参数，还有可以查看弱引用的参数：-XX:+PrintReferenceGC。它跟踪软引用、弱引用、虚引用和Finallize队列的信息，但是使用场景较为狭窄。基本上掌握上面的几个常用的 GC 日志参数就足够排查使用，最重要的是弄清楚每个参数的作用和用法。

最后用列表的形式总结一下，加深一下印象。

| **参数** | **含义** |
| --- | --- |
| -XX:PrintGC | 打印GC日志 |
| -XX:+PrintGCDetails | 打印详细的GC日志。还会在退出前打印堆的详细信息。 |
| -XX:+PrintHeapAtGC | 每次GC前后打印堆信息。 |
| -XX:+PrintGCTimeStamps | 打印GC发生的时间。 |
| -XX:+PrintGCApplicationConcurrentTime | 打印应用程序的执行时间 |
| -XX:+PrintGCApplicationStoppedTime | 打印应用由于GC而产生的停顿时间 |
| -XX:+PrintReferenceGC | 跟踪软引用、弱引用、虚引用和Finallize队列。 |
| -XLoggc | 将GC日志以文件形式输出。 |