## JVM知识点

1. 多语言编程：并行处理使用clojure编写，展示层使用JRuby/Rails,中间层java
2. 栈：跨平台性、指令集小、指令多；执行性能比寄存器差
3. 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

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

4，String类使用引导类加载器进行加载的。Java的核心类库都是使用引导类加载器进行加载的

5，为什么要自定义加载类：

1，隔离加载类

2，修改类的加载方式

3，扩展加载原

4，防止源码泄露

6，双亲委派模式：java虚拟机对class文件采用的是按需加载的方式，也就是说当需要使用该类时才会将他的class文件加载到内存生成class对象。而且加载某个类的class文件时，java虚拟机采用的是双亲委派模式，即把请求交给父类处理，它是一种任务委派模式

优势：避免类的重复加载

保护程序安全，防止核心api被随意篡改比如自定义类java.lang.String

1. 沙箱安全机制：避免被恶意篡改
2. GC:垃圾回收机制
3. OOM：OutOtMemoryError,内存不足异常
4. PC寄存器为什么会被设定为线程私有？

答：因为PC寄存器需要记录一个线程执行到那了，如果公用一个，当两个线程中，第一个执行到第五行的时候，它会记录为5，第二个线程开始执行了，之后执行到第四行的时候，反过来执行第一个线程的时候，会记录第二个线程的执行行数4，这时第一个线程的执行行数被覆盖了，就没办法完整执行线程1了，所以需要私有PC寄存器

1. 并行：一个CPU时间段，同时执行

并发：交替执行

1. 虚拟机栈：

由于跨平台性的设置，java的指令都是根据栈来设计的，不同平台CPU架构不同，所以不能设置基于寄存器的

优点是跨平台，指令集小，编译器容易实现，缺点是性能下降，实现同样的功能需要更多指令

1. 栈是运行时的单位，堆是存储的单位

即：栈解决程序的运行单位，即程序如何执行，或者说如何处理数据，堆解决的是数据存储的问题，即：数据怎么放，放到那

1. 一个栈帧对应一个方法
2. 调整栈的大小：

IDEA：Run -> Edit Configuration : Configuration -> VM options : 输入：-Xss256K

16,解析class文件：javap -v Test.class

17，{}括号中的变量执行完之后会直接销毁

17,一个方法以return结束

1. 每个线程有自己的栈，栈里面可以有多个栈帧
2. 栈帧的内部结构：局部变量表，操作数栈，动态链接，方法返回地址，一些附加信息
3. 变量的分类：按照数据类型分：① 基本数据类型 ② 引用数据类型  
    按照在类中声明的位置分：
4. （1），成员变量：在使用前，都经历过默认初始化赋值  
    （2），类变量： linking的prepare阶段：给类变量默认赋值 ---> initial阶段：给类变量显式赋值即静态代码块赋值  
    实例变量：随着对象的创建，会在堆空间中分配实例变量空间，并进 行默认赋值  
    2， 局部变量：在使用前，必须要进行显式赋值的！否则，编译不通过
5. 操作数栈在编译时期的长度就已经确定，因为是数组实现
6. 寄存器：指令更少，执行速度快
7. 为什么需要常量池：常量池的作用，就是为了提供一些符号和常量，便于指令的识别
8. 虚方法跟非虚方法
   1. 跟多态类型的方法大多都是虚方法
   2. 静态方法，私有方法，final方法，实例构造器，父类方法不能被重写的方法是非虚方法
9. 编译器能被确定的叫非虚方法，不能被确定的叫虚方法
10. OOM异常 垃圾回收GC
    1. 程序计数器 无 无
    2. 虚拟机栈 有 无
    3. 本地方法栈 有 无
    4. 堆 有 有
    5. 方法区 有 有
11. 线程安全：
    1. 如果只有一个线程才可以操作此数据，则必定是线程安全的
    2. 如果有多个线程操作此数据，则此数据是共享数据，如果不考虑同步机制的话，会存在线程安全问题
12. 每个进程表示一个JVM实例，一个JVM实例有一个运行时数据区，一个进程里的多个线程共享方法区跟堆，一个线程都拥有独自的程序计数器，本地方法栈和虚拟机栈，一个JVM实例只存在一个堆
13. Jdk8相对于jdk7堆有哪些变化：jdk7中的永久区改为了元空间
14. 设置堆空间参数
    1. -Xms 用来设置堆空间（新生代+老年代）的初始内存大小
       1. -X 是JVM的运行参数
       2. ms是memory start
    2. -Xmx 用来设置堆空间的（新生代+老年代）最大内存大小
    3. 默认堆空间大小
       1. 初始内存大小：物理电脑内存大小/64
          1. 最大内存大小：物理电脑内存大小/4
    4. 手动设置：-Xms600m -Xmx600m
       1. 开发中建议将初始堆内存和最大堆内存设置为相同的值
    5. 查看设置的参数：
       1. 方式1：jps / jstat -gc 进程id
       2. 方式2：-XX:+PrintGCDetails
15. 参数
    1. -XX:NewRatio ：设置新生代于老年代的比例默认值是2
    2. -XX:SurviorRatio ：设置新生代中Eden区与Survivor区的比例，默认值是8
    3. -XX:-UseAdaptiverSizePolicy ：关闭自适应的内存分配策略（暂时用不到）“-”表示关闭，“+”表示开启
    4. -Xmn ：设置新生代的空间大小（一般不设置）
16. 判断是否发生了逃逸：就看new的对象的实体是否可能在方法的外被调用，能就是逃逸，不能就不是逃逸
    1. 能使用局部变量的，就不要使用在方法外定义
17. GC较多的收集新生代，较少的收集老年代，基本不动方法区（元空间）
18. 方法区看作是一块独立于java堆的内存空间
19. 方法区默认值跟最大值设置：
    1. Jdk8以后
       1. 默认：-XX:MetaspaceSize=100m
       2. 最大值:-XX:MaxMetaspaceSize=100m
    2. Jdk7以前
       1. 默认：-XX:PermSize=100m
       2. 最大值:-XX:MaxPermSize=100m
20. 方法区存储什么：
    1. 已被虚拟机加载的类型信息
    2. 常量
    3. 静态变量
    4. 及时编译器编译后的代码缓存
    5. 域信息，方法信息涵盖在类型信息当中
21. 常量池：
    1. 可以看做是一张表，虚拟机指令根据这张常量表找到要执行的类名，方法名，参数类型，字面量等类型。
22. ByteBuffer对象：直接分配本地内存空间
23. String不可变性，jdk8中字符串常量放在堆，jdk6放在永久代
24. 字符串拼接操作中有变量相当于new StringBuilder()，最后返回一个字符串，类似于new String();
25. String str = “a”；常量池 String str = new String(“s”)；堆
26. 拼接操作不一定是StringBuilder
    1. 如果拼接符号左右两边都是字符常量或常量引用，则仍然使用编译期优化，即非StringBuilder方式
    2. 针对于final修饰类，方法，基本数据类型，引用数据类型的量的结构时，能使用上final就使用final
27. 体会执行效率：通过StringBuilder的append()方法添加字符串的效率要远高于使用String方式拼接字符串

详情

* 1. StringBuilder方式自始至终创建了一个StringBuilder对象，使用String方式拼接创建了多个StringBuilder对象跟String对象
  2. 使用String的字符串拼接方式：由于内存中创建了多个StringBuilder对象和String对象，内存占用更大，会进行更多的GC操作

改进空间

在实际开发中，如果基本确定前前后后添加的字符串长度不高于某个限定值higHLevel的情况下，建议使用构造器：StringBuilder sb = new StringBuilder(higHLevel);//new char[highLevel]

1. StringBuilder不同步，StringBuffer同步没有StringBuilder效率高
2. 如何保证变量指向的是字符串常量池中的数据呢？

两种方式

* 1. String s = “hello”； //通过字面量定义的方式
  2. String s = new String(“sdf”).intern()

StringBuilder sb = new StringBuilder(“sss”).toString().intern();

//调用intern()

1. GC主要关注与方法区跟堆区
2. 引用计数算法不能处理循环引用，会导致内存泄漏，java中没有采用引用计数法
3. 三级缓存：1,内存-->2,本地-->3,网络
4. Java不同版本新特性
   1. 语法层面：Lambda表达式，switch，自动装箱，自动拆箱，enum，<>,......
   2. API层面：Stream API，新的日期时间，Optional，String，框架
   3. 底层优化：JVM优化，GC的变化，元空间，静态域，字符串常量池
5. 查看默认的垃圾回收器
   1. -XX：+PrintCommandLineFlags:查看命令行 相关参数，包含使用的垃圾收集器
   2. 使用命令行指令： jinfo -flag 相关垃圾回收器参数 进程ID