总结了JVM一些经典面试题，分享出我自己的解题思路，希望对大家有帮助，有哪里你觉得不正确的话，欢迎指出，后续有空会更新。

1.什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。

我的答案：

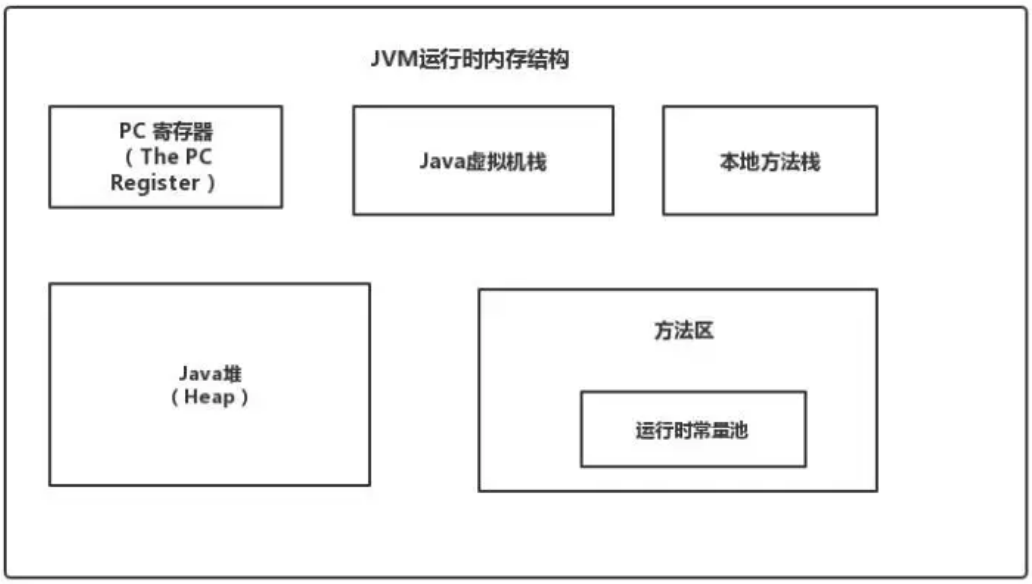
- 栈是线程私有的，他的生命周期与线程相同，每个方法在执行的时候都会创建一个栈帧，用来存储局部变量表，操作数栈，动态链接，方法出口等信息。局部变量表又包含基本数据类型，对象引用类型
- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常，方法递归调用产生这种结果。
- 如果Java虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时候没有足够的内存去创建对应的虚拟机栈，那么Java虚拟机将抛出一个OutOfMemory 异常。(线程启动过多)
- 参数 -Xss 去调整JVM栈的大小

2.详解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。

我的答案：

- JVM内存结构



程序计数器：当前线程所执行的字节码的行号指示器，用于记录正在执行的虚拟机字节指令地址，线程私有。

Java虚拟栈：存放基本数据类型、对象的引用、方法出口等，线程私有。

Native方法栈：和虚拟栈相似，只不过它服务于Native方法，线程私有。

Java堆：java内存最大的一块，所有对象实例、数组都存放在java堆，GC回收的地方，线程共享。

方法区：存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码数据等。（即永久带），回收目标主要是常量池的回收和类型的卸载，各线程共享

3.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。

我的答案：

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，Edem : 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）

4. 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。**

5.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。

我的答案：

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；

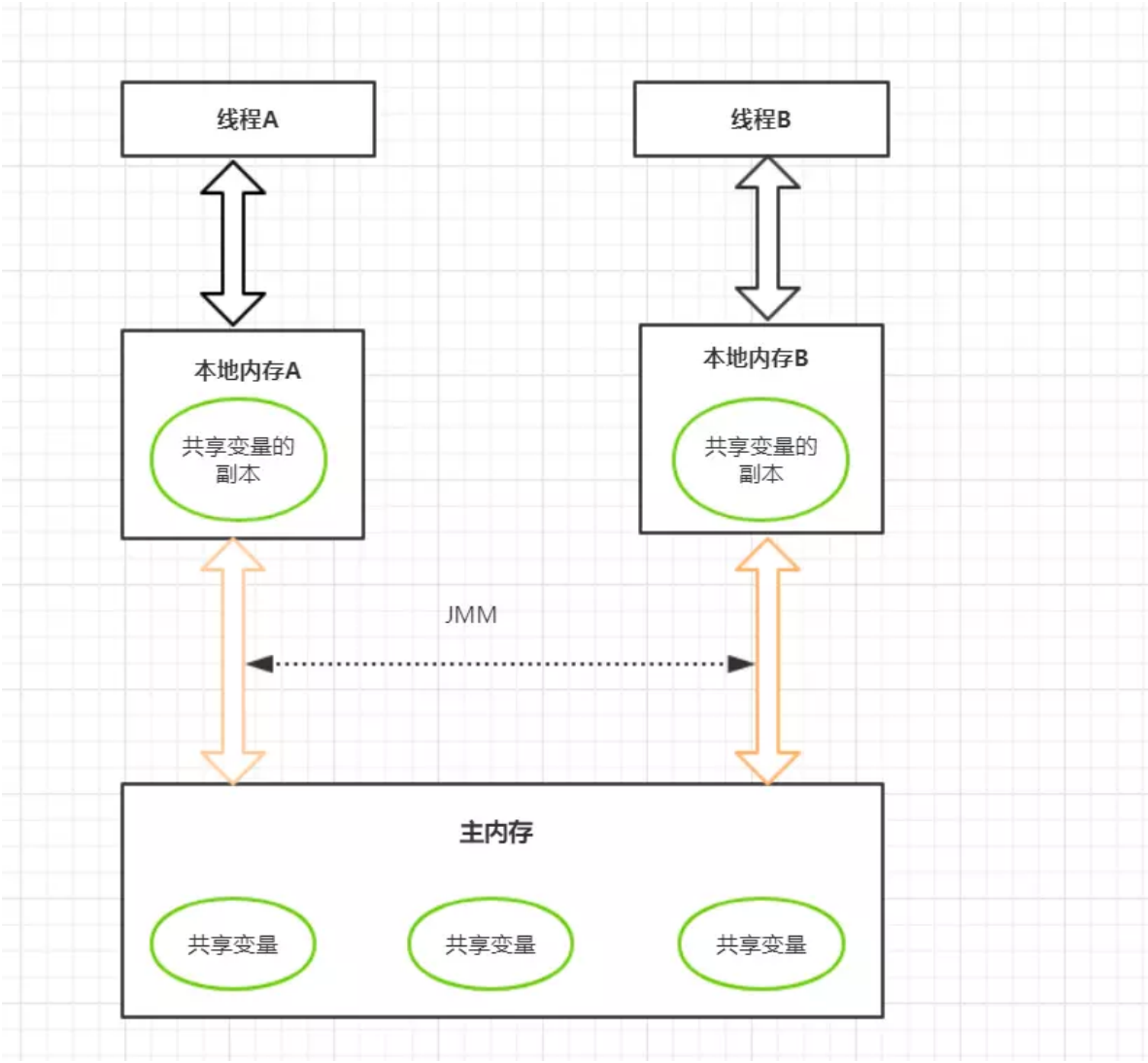
- CMS收集器以最小的停顿时间为目标的收集器；
- G1收集器可预测垃圾回收的停顿时间
- CMS收集器是使用“标记-清除”算法进行的垃圾回收，容易产生内存碎片
- G1收集器使用的是“标记-整理”算法，进行了空间整合，降低了内存空间碎片。

6.JVM内存模型的相关知识了解多少，比如重排序，内存屏障，happen-before，主内存，工作内存。

思路： 先画出Java内存模型图，结合例子volatile ，说明什么是重排序，内存屏障，最好能给面试官写以下demo说明。

我的答案：

1）Java内存模型图：



Java内存模型规定了所有的**变量都存储在主内存**中，每条**线程还有自己的工作内存**，线程的工作内存中保存了该线程中是用到的变量的主内存副本拷贝，**线程对变量的所有操作都必须在工作内存中**进行，**而不能直接读写主内存。**不同的线程之间也**无法直接访问对方工作内存中的变量**，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。

2）指令重排序。

在这里，先看一段代码

```
public class PossibleReordering {
static int x = 0, y = 0;
static int a = 0, b = 0;

public static void main(String[] args) throws InterruptedException {
 Thread one = new Thread(new Runnable() { public void run() { a = 1; x = b; }
});
 Thread other = new Thread(new Runnable() { public void run() { b = 1; y = a; }
}); one.start();other.start(); one.join();other.join(); System.out.println("(" +
x + "," + y + ")");}
```

运行结果可能为(1,0)、(0,1)或(1,1)，也可能是(0,0)。因为，在实际运行时，代码指令可能并不是严格按照代码语句顺序执行的。大多数现代微处理器都会采用将指令乱序执行（out-of-order execution，简称OoOE或OOE）的方法，在条件允许的情况下，直接运行当前有能力立即执行的后续指令，避开获取下一条指令所需数据时造成的等待3。通过乱序执行的技术，处理器可以大大提高执行效率。而这就是**指令重排**。

3）内存屏障

内存屏障，也叫内存栅栏，是一种CPU指令，用于控制特定条件下的重排序和内存可见性问题。

- **LoadLoad屏障**：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
- **StoreStore屏障**：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
- **LoadStore屏障**：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
- **StoreLoad屏障**：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。 在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

4）happen-before原则

- **单线程happen-before原则**：在同一个线程中，书写在前面的操作happen-before后面的操作。锁的happen-before原则：同一个锁的unlock操作happen-before此锁的lock操作。
- **volatile的happen-before原则**：对一个volatile变量的写操作happen-before对此变量的任意操作(当然也包括写操作了)。
- **happen-before的传递性原则**：如果A操作 happen-before B操作，B操作happen-before C操作，那么A操作happen-before C操作。
- **线程启动的happen-before原则**：同一个线程的start方法happen-before此线程的其它方法。
- **线程中断的happen-before原则** ：对线程interrupt方法的调用happen-before被中断线程的检测到中断发送的代码。
- **线程终结的happen-before原则**： 线程中的所有操作都happen-before线程的终止检测。
- **对象创建的happen-before原则**： 一个对象的初始化完成先于他的finalize方法调用。

7.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。

我的答案：

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。

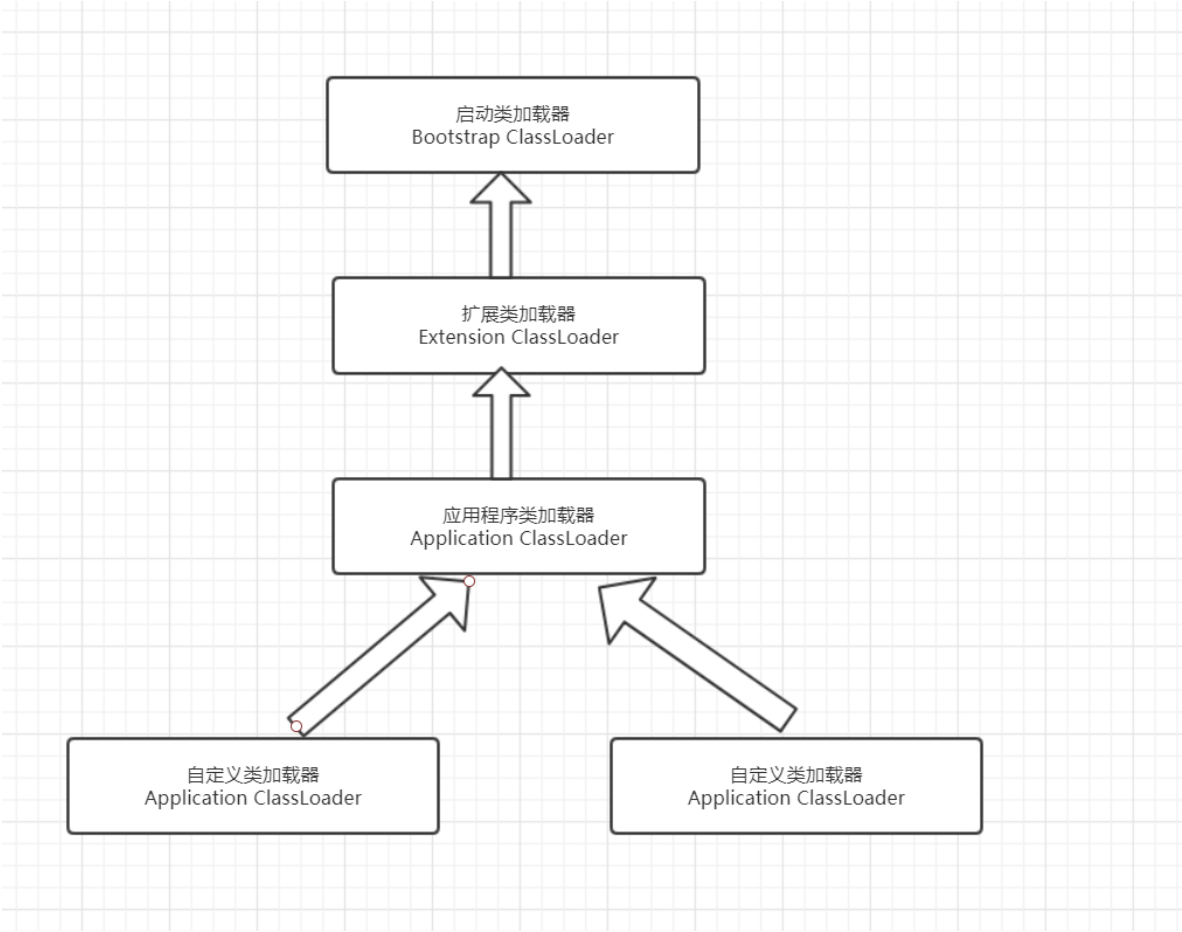
> - 启动类加载器（Bootstrap ClassLoader）：由C++语言实现（针对HotSpot）,负责将存放在<JAVA_HOME>\lib目录或-Xbootclasspath参数指定的路径中的类库加载到内存中。
> - 其他类加载器：由Java语言实现，继承自抽象类ClassLoader。如：
>
> > - 扩展类加载器（Extension ClassLoader）：负责加载<JAVA_HOME>\lib\ext目录或java.ext.dirs系统变量指定的路径中的所有类库。
> > - 应用程序类加载器（Application ClassLoader）。负责加载用户类路径（classpath）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：



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。

8.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。

我的答案:

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

9.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。

我的答案:

- 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/

10.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。

我的答案：

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。