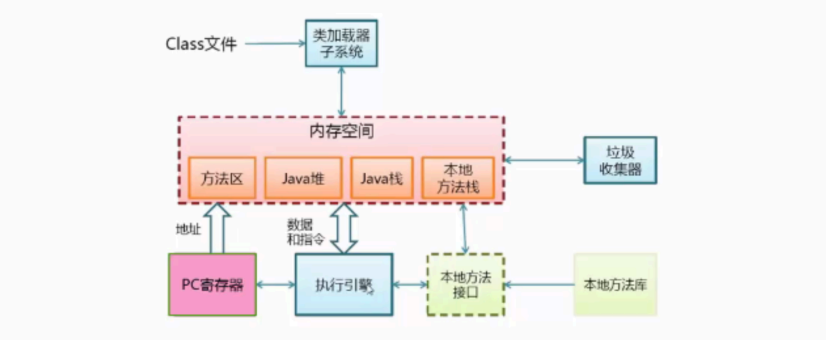
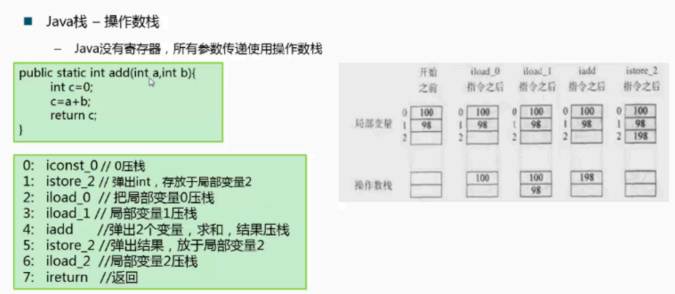
1、jvm结构：



1. PC寄存器：
    a. 每个线程拥有一个pc寄存器；
    b. 指向下一条指令的地址。
2. 方法区：
    a. 保存装载的类的元信息：类型的常量池，字段、方法信息，方法字节码；

> jdk6时，String等常量信息置于方法区，jdk7移到了堆中；

    b. 通常和永久区（Perm）关联在一起；
3. 堆：
    a. 应用系统对象都保存在java堆中；
    b. 所有线程共享java堆；
    c. 对分代GC来说，堆也是分代的；
4. 栈：
    a. 线程私有；
    b. 栈由一系列帧组成（因此java栈也叫做帧栈）；
    c. 帧保存一个方法的局部变量（局部变量表）、操作数栈、常量池指针；



    d. 每一次方法调用创建一个帧，并压栈。

2、jvm内存模型：

1. 每一个线程有一个工作内存，和主存独立；
2. 工作内存存放主存中变量的值的拷贝；
3. 对于普通变量，一个线程中更新的值，不能马上反应在其他变量中；如果需要在其他线程中立即可见，需要使用volatile关键字；
4. volatile不能代替锁，一般认为volatile比锁性能好(不绝对)，使用volatile的条件是语义是否满足应用；

5. 可见性：一个线程修改了变量，其他线程可以立即知道。
    a. volatile；
    b. synchronized（unlock之前，写变量值回主存）；
    c. final（一旦初始化完成，其他线程可见）。

3、java四引用：

1. 强引用：

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

2. 软引用：

如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

3. 弱引用：

弱引用与软引用的区别在于：只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。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4. 虚引用：

虚引用在任何时候都可能被垃圾回收器回收，主要用来跟踪对象被垃圾回收器回收的活动，被回收时会收到一个系统通知。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

4、GC算法分类：

1. 引用计数法(没有被java采用)：
    a. 原理：对于一个对象A，只要有任何一个对象引用了A，则A的引用计数器就加1，当引用失效时，引用计数器就减1，只要对象A的引用计数器的值为0，则对象A就会被回收。
    b. 问题：
        i. 引用和去引用伴随加法和减法，影响性能；
        ii. 很难处理循环引用。
2. 标记清除法：
    a. 原理：现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。一种可行的实现是，在标记节点，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象。然后在清除阶段，清除所有未被标记的对象。
    b. 问题：
        i. 标记和清除两个过程效率不高，产生内存碎片导致需要分配较大对象时无法找到足够的连续内存而需要触发一次GC操作。
3. 标记压缩法：
    a. 原理：适合用于存活对象较多的场合，如老年代。它在标记-清除算法的基础上做了一些优化。标记阶段一样，但之后，将所有存活对象压缩到内存的一端。之后，清除边界外所有的空间。
    b. 优点：
        i. 解决了标记- 清除算法导致的内存碎片问题和在存活率较高时复制算法效率低的问题。
4. 复制算法：
    a. 原理：将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，之后清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。
    b. 问题：
        i. 不适用于存活对象比较多的场合，如老年代。
5. 分代回收法：
    a. 原理：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。

5、MinorGC & FullGC：

1. Minor GC通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生GC的频率较高，回收速度比较快，一般采用复制-回收算法。
2. Full GC/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC，所采用的是标记-清除算法。

6、垃圾收集器：

1. Serial New收集器是针对新生代的收集器，采用的是复制算法；

2. Parallel New（并行）收集器，新生代采用复制算法，老年代采用标记整理；
3. Parallel Scavenge（并行）收集器，针对新生代，采用复制收集算法；
4. Serial Old（串行）收集器，新生代采用复制，老年代采用标记清理；
5. Parallel Old（并行）收集器，针对老年代，标记整理；
6. CMS收集器，基于标记清理；
7. G1收集器(JDK)：整体上是基于标记清理，局部采用复制；

综上：新生代基本采用复制算法，老年代采用标记整理算法。cms采用标记清理；

7、java类加载机制：

1. 概念：虚拟机把描述类的数据文件（字节码）加载到内存，并对数据进行验证、准备、解析以及类初始化，最终形成可以被虚拟机直接使用的java类型（java.lang.Class对象）。
2. 类生命周期：

类加载过程：读取二进制字节流到jvm—>验证格式语义等—>为静态变量分配内存空间—>常量池引用解析—>执行static标识的代码

a. 加载过程：通过一个类的全限定名来获取定义此类的二进制字节流，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。在内存中(方法区)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；

b. 验证过程：为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，文件格式验证、元数据验证、字节码验证、符号引用验证；

c. 准备过程：正式为类属性分配内存并设置类属性初始值的阶段，这些内存都将在方法区中进行分配；

> 准备阶段，static对象会被设置默认值，static final对象会被赋上给予的值。

d. 解析阶段：虚拟机将常量池内的符号引用替换为直接引用的过程。

i. 符号引用：字符串，引用对象不一定被加载；

ii. 直接引用：指针或者地址偏移量，引用对象一定在内存中。

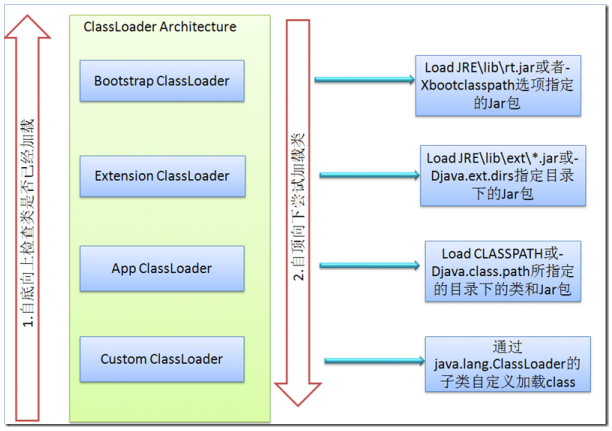
e. 初始化阶段：类初始化阶段是类加载过程的最后一步。初始化阶段就是执行类构造器<clint>()方法的过程。

f. 使用阶段：

g. 卸载阶段：

8、类加载器：

1. java默认提供三个类加载器：



a. BootStrap ClassLoader 启动ClassLoader（sun.boot.class.path）：最顶层的加载类，主要加载jdk中的核心库，%JRE_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。

> Bootstrap ClassLoader不继承自ClassLoader，因为它不是一个普通的Java类，底层由C++编写，已嵌入到了JVM内核当中，当JVM启动后，Bootstrap ClassLoader也随着启动，负责加载完核心类库后，并构造Extension ClassLoader和App ClassLoader类加载器。

b. Extension ClassLoader 扩展ClassLoader（java.ext.dirs）：扩展的类加载器，加载目录%JRE_HOME%\lib\ext目录下的jar包和class文件。还可以加载-D java.ext.dirs选项指定的目录。

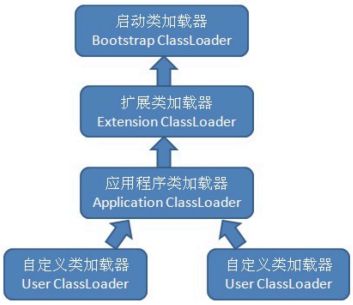
c. App ClassLoader 应用ClassLoader/系统ClassLoader（java.class.path）：也称为SystemAppClass 加载当前应用的classpath的所有类。

> 除了BootStrap ClassLoader，每个ClassLoader都有一个Parent作为父亲。

2. 双亲委派机制：

a. 定义：当一个ClassLoader实例需要加载某个类时，它会试图亲自搜索某个类之前，先把这个任务委托给它的父类加载器，这个过程是由上至下依次检查的，首先由最顶层的类加载器Bootstrap ClassLoader试图加载，如果没加载到，则把任务转交给Extension ClassLoader试图加载，如果也没加载到，则转交给App ClassLoader 进行加载，如果它也没有加载得到的话，则返

回给委托的发起者，由它到指定的文件系统或网络等URL中加载该类。如果它们都没有加载到这个类时，则抛出ClassNotFoundException异常。否则将这个找到的类生成一个类的定义，并将它加载到内存当中，最后返回这个类在内存中的Class实例对象。



b. 作用：

i. 避免重复加载；

ii. 考虑到安全因素，避免自定义的类去替代系统类，如String。

c. jvm如何判定两个class是否相同？JVM在判定两个class是否相同时，不仅要判断两个类名是否相同，而且要判断是否由同一个类加载器实例加载的。只有两者同时满足的情况下，JVM才认为这两个class是相同的。

1. 自底向上检查类是否已经加载；
2. 自顶向下尝试加载类。

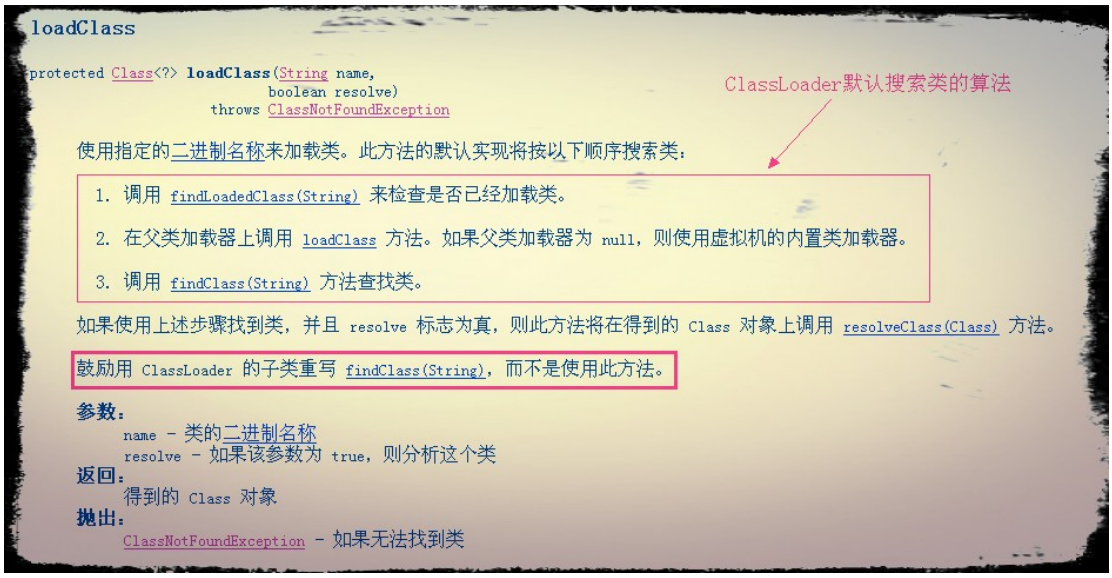
3. custom classloader：自定义classloader

a. Java中提供的默认ClassLoader，只加载指定目录下的jar和class，如果我们想加载其它位置的类或jar时，就需要定义自己的ClassLoader。

b. 步骤：

i. 继承java.lang.ClassLoader

ii. 重写父类的findClass方法



9、引起类加载的五个行为：

1. 遇到new、getstatic、putstatic或invokestatic这四条字节码指令
2. 反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化
3. 子类初始化的时候，如果其父类还没初始化，则需先触发其父类的初始化
4. 虚拟机执行主类的时候(有 main(string[] args))
5. JDK1.7 动态语言支持

10、java对象创建时机：

1. 使用new关键字创建对象
2. 使用Class类的newInstance方法(反射机制)

3. 使用Constructor类的newInstance方法(反射机制)
4. 使用Clone方法创建对象
5. 使用(反)序列化机制创建对象

11、jvm调优：

1. 调优时机：
   a. heap 内存（老年代）持续上涨达到设置的最大内存值；
   b. Full GC 次数频繁；
   c. GC 停顿时间过长（超过1秒）；
   d. 应用出现OutOfMemory 等内存异常；
   e. 应用中有使用本地缓存且占用大量内存空间；
   f. 系统吞吐量与响应性能不高或下降。
2. 调优原则：
   a. 多数的Java应用不需要在服务器上进行JVM优化；
   b. 多数导致GC问题的Java应用，都不是因为我们参数设置错误，而是代码问题；
   c. 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）；
   d. 减少创建对象的数量；
   e. 减少使用全局变量和大对象；
   f. JVM优化是到最后不得已才采用的手段；
   g. 在实际使用中，分析GC情况优化代码比优化JVM参数更好；
3. 调优目标：
   a. GC低停顿；
   b. GC低频率；
   c. 低内存占用；
   d. 高吞吐量；
4. 调优步骤：
   a. 分析GC日志及dump文件，判断是否需要优化，确定瓶颈问题点；
   b. 确定jvm调优量化目标；
   c. 确定jvm调优参数（根据历史jvm参数来调整）；
   d. 调优一台服务器，对比观察调优前后的差异；
   e. 不断的分析和调整，知道找到合适的jvm参数配置；
   f. 找到最合适的参数，将这些参数应用到所有服务器，并进行后续跟踪。

12、jvm调优参数：

1. 设定堆内存大小，这是最基本的。
2. -Xms：启动JVM时的堆内存空间。
3. -Xmx：堆内存最大限制。
4. 设定新生代大小。
5. 新生代不宜太小，否则会有大量对象涌入老年代。
6. -XX:NewRatio：新生代和老年代的占比。
7. -XX:NewSize：新生代空间。
8. -XX:SurvivorRatio：伊甸园空间和幸存者空间的占比。
9. -XX:MaxTenuringThreshold：对象进入老年代的年龄阈值。
10. 设定垃圾回收器

    年轻代：-XX:+UseParNewGC。

    老年代：-XX:+UseConcMarkSweepGC。

    CMS可以将STW时间降到最低，但是不对内存进行压缩，有可能出现“并行模式失败”。比如老年代空间还有300MB空间，但是一些10MB的对象无法被顺序的存储。这时候会触发压缩处理，但是CMS GC模式下的压缩处理时间要比Parallel GC长很多。

    G1采用”标记-整理“算法，解决了内存碎片问题，建立了可预测的停顿时间类型，能让使用者指定在一个长度为M毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。

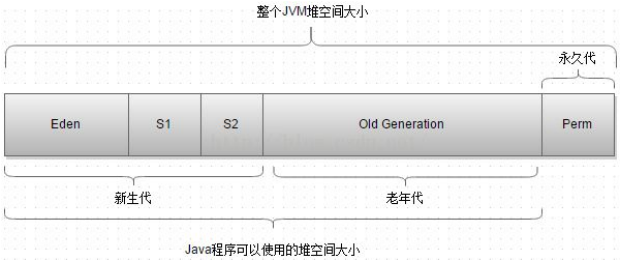
13、触发full gc的场景及应对策略：

1. System.gc()方法的调用，应对策略：通过-XX:+DisableExplicitGC来禁止调用System.gc；
2. 老年代代空间不足，应对策略：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，不要创建过大的对象及数组；
3. 永生区空间不足，应对策略：增大PermGen空间；
4. GC时出现promotionfailed和concurrent mode failure，应对策略：增大survivor space；
5. Minor GC后晋升到旧生代的对象大小大于老年代的剩余空间，应对策略：增大Tenured space 或下调

CMSInitiatingOccupancyFraction=60；

6. 内存持续增涨达到上限导致Full GC，应对策略：通过dumpheap 分析是否存在内存泄漏。

14、jvm堆的基本结构：



1、JVM中堆空间可以分成三个大区，新生代、老年代、永久代

2、新生代可以划分为三个区，Eden区，两个Survivor区，在HotSpot虚拟机Eden和Survivor的大小比例为8:1

15、如何查看jvm内存使用情况：

可以使用JDK自带的JConsole、JVisualVM、JMap、JHat等工具，或者使用第三方工具，比如 Eclipse Memory Analyzer

16、jvm内存溢出例子：

1. 内存溢出，比如给JVM分配的内存不够大，或者程序中存在死循环一直申请内存。

2. 内存泄露，比如下面这段代码，list持有o的引用，o暂时是无法被JVM垃圾回收的，只有当list被垃圾回收或者o从对象list删除掉后，o才能被JVM垃圾回收。

```
List<Object> list = new ArrayList<>();
Object o = new Object();
list.add(o);
o = null;
```

17、常用的GC策略，什么时候会触发YGC，什么时候触发FGC：

1. YGC(Young GC)：

   a. 概念：对新生代堆进行GC。频率比较高，因为大部分对象的存活寿命较短，在新生代里被回收。性能耗费较小。

   b. 触发时机：Eden区空间不足。

2. FGC(Full GC)：

   a. 概念：全堆范围的GC。默认堆空间使用到达80%(可调整)的时候会触发FGC。以我们生产环境为例，一般比较少会触发FGC，有时10天或一周左右会有一次。

   b. 触发时机：

      1、Old空间不足；

      2、Perm空间不足；

      3、显示调用System.gc()，包括RMI等的定时触发；

      4、YGC时的悲观策略；

      5、dump live的内存信息时(jmap –dump:live)。

18、获得Class对象的方式

1. 静态类的.class语法：GuideUtil.class

2. 普通类对象的getClass()方法：new Test().getClass()

3. 通过Class对象的forName()方法：

   Class.forName(“com.zhenai.modules.guide.utils.GuideUtil");

4. 对于包装类,可以通过.TYPE语法方式：Integer.TYPE

19、内存溢出的可能原因和解决方法

A，数据加载过多，如1次从数据库中取出过多数据

B，集合类中有对对象的引用，用完后没有清空或者集合对象未置空导致引用存在等，是的JVM无法回收

C，死循环，过多重复对象

D，第三方软件的bug

E，启动参数内存值设定的过小。

解决方法：修改JVM启动参数，加内存(-Xms，-Xmx)；错误日志，是否还有其他错误；代码走查

20、内存泄漏的原因?

1. 未对作废数据内存单元置为null，尽早释放无用对象的引用，使用临时变量时，让引用变量在推出活动域后自动设置为null，暗示

垃圾收集器收集；

2. 程序避免用String拼接，用StringBuffer，因为每个String会占用内存一块区域；

3. 尽量少用静态变量（全局不会回收）；

4. 不要集中创建对象尤其大对象，可以使用流操作；

5. 尽量使用对象池，不再循环中创建对象，优化配置；

6. 创建对象到单例getInstance中，对象无法回收被单例引用；

7. 服务器session时间设置过长也会引起内存泄漏。

21、方法区oom

1. 方法区用于存放Class的相关信息，如：类名，访问修饰符，常量池，字符描述，方法描述等。

2. 原因：运行时产生大量的类去填满方法区，直到溢出。

22、哪些情况下对象会进入老年代?

1. 新生代对象每次经历一次minor gc，年龄会加1，当达到年龄阈值（默认为15岁）会直接进入老年代；

2. 大对象直接进入老年代；

3. 新生代复制算法需要一个survivor区进行轮换备份，如果出现大量对象在minor gc后仍然存活的情况时，就需要老年代进行分配担保，让survivor无法容纳的对象直接进入老年代；

4. 如果在Survivor空间中相同年龄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就可以直接进入年老代。

23、jvm中哪些地方会出现oom？分别说说oom的可能原因?

1. java堆溢出（heap）：

a. Java堆内存主要用来存放运行过程中所以的对象，该区域OOM异常一般会有如下错误信息：

```
java.lang.OutofMemoryError:Java heap space
```

b. 此类错误一般通过Eclipse Memory Analyzer分析OOM时dump的内存快照就能分析出来，到底是由于程序原因导致的内存泄露，还是由于没有估计好JVM内存的大小而导致的内存溢出。

c. 另外，Java堆常用的JVM参数：

```
-Xms：初始堆大小，默认值为物理内存的1/64(<1GB)，默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会
-Xmx：最大堆大小，默认值为物理内存的1/4(<1GB)，默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减
-Xmn：年轻代大小(1.4or lator)，此处的大小是（eden + 2 survivor space)，与jmap -heap中显示的New gen是不同的
```

2. 栈溢出（stack）：

a. 栈用来存储线程的局部变量表、操作数栈、动态链接、方法出口等信息。如果请求栈的深度不足时抛出的错误会包含类似下面的信息：

```
java.lang.StackOverflowError。
```

b. 另外，由于每个线程占的内存大概为1M，因此线程的创建也需要内存空间。操作系统可用内存-Xmx-MaxPermSize即是栈可用的内存，如果申请创建的线程比较多超过剩余内存的时候，也会抛出如下类似错误：java.lang.OutofMemoryError: unable to create new native thread

c. 相关的JVM参数有：

```
-Xss：每个线程的堆栈大小，JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。
在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~
```

d. 可能原因：

i. 递归：递归里用到的局部变量存储在堆栈中，堆栈的访问效率高，速度快，但空间有限，递归太多变量需要一直入栈而不出栈，导致需要的内存空间大于堆栈的空间，栈空间是2M，堆空间内存空间。

3. 运行时常量溢出（constant）：

a. 运行时常量保存在方法区，存放的主要是编译器生成的各种字面量和符号引用，但是运行期间也可能将新的常量放入池中，比如String类的intern方法。如果该区域OOM，错误结果会包含类似下面的信息：

```
java.lang.OutofMemoryError: PermGen space
```

b. 相关的JVM参数有：

```
-XX:PermSize: 设置持久代(perm gen)初始值，默认值为物理内存的1/64
-XX:MaxPermSize: 设置持久代最大值，默认为物理内存的1/4
```

4. 方法区溢出：

a. 方法区主要存储被虚拟机加载的类信息，如类名、访问修饰符、常量池、字段描述、方法描述等。理论上在JVM启动后该区域大小应该比较稳定，但是目前很多框架，比如Spring和Hibernate等在运行过程中都会动态生成类，因此也存在OOM的风险。如果该区域OOM，错误结果会包含类似下面的信息：

```
java.lang.OutofMemoryError: PermGen space
```

b. 相关的JVM参数有：

```
-XX:PermSize: 设置持久代(perm gen)初始值，默认值为物理内存的1/64
-XX:MaxPermSize: 设置持久代最大值，默认为物理内存的1/4
```

23、如何定位jvm内存信息?

1. 打印日志：

```
-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), 0.043348
-XX:+PrintGCTimeStamps: 打印GC停顿耗时
-XX:+PrintGCApplicationStoppedTime: 打印垃圾回收期间程序暂停的时间.
-XX:+PrintHeapAtGC: 打印GC前后的详细堆栈信息
-Xloggc:filename: 把相关日志信息记录到文件以便分析.
```

2. 错误调试：

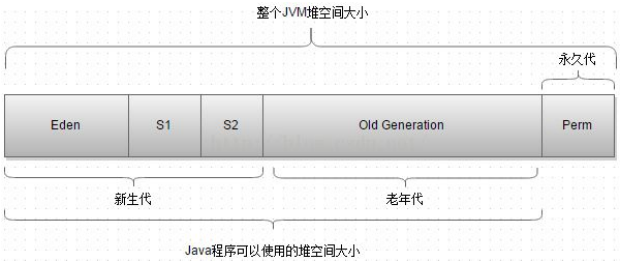
```
-XX:ErrorFile=./hs_err_pid<pid>.log: 如果JVM crashed，将错误日志输出到指定文件路径。
-XX:HeapDumpPath=./java_pid<pid>.hprof: 堆内存快照的存储文件路径。
-XX:-HeapDumpOnOutOfMemoryError: 在OOM时，输出一个dump.core文件，记录当时的堆内存快照
```

3. 类状态器相关：

```
-XX:-TraceClassLoading: 打印class装载信息到stdout。记Loaded状态。
-XX:-TraceClassUnloading: 打印class的卸载信息到stdout。记Unloaded状态。
```

24、当对象A创建之后，对象A在各个区之间的流转过程

1. jvm堆结构图：



a. 新生代通常占JVM堆内存的1/3，因为新生代存储都是新创建的对象，比较小的对象，而老年代存的都是比较大的，活的久的对象，所以老年代占JVM堆内存较大；

b. 新生代里的Eden区通常占年轻代的4/5，两个Survivor分别占新生代的1/10。因为Survivor中存储的是GC之后幸存的对象，实际上只有很少一部分会幸存，所以Survivor占的比例比较小。

2. 对象流转流程（新生代复制算法，可以减少内存碎片）：

a. 对象A被new出来之后，是被存放在Eden（伊甸园）区的。

b. 当发生一次GC后，Eden区存活下来的对象A会被复制到s1区，s0中存活的对象也会被复制到s1中。

> 如果对象年龄超过阈值年龄（默认15岁），会被复制到老年区。
> 部分对象也需要老年代分担。

c. GC会清空Eden和s0中存储的所有对象；

d. 交换s0和s1的角色；

e. 重复上面的步骤。

25、jvm堆持久代：

用于存放静态类型数据，如 Java Class, Method 等。持久代对垃圾回收没有显著影响。但是有些应用可能动态生成或调用一些 Class，例如 Hibernate CGLib 等，在这种时候往往需要设置一个比较大的持久代空间来存放这些运行过程中动态增加的类型。

26、synchronized和ReentrantLock的实现机制?

synchronized用的锁是存在Java对象头里的