1 类加载

1.1class文件结构(无符号数和表组成)

U4魔数(CAFEBABE)

U2 Minor verison

U2 Major version 52(1.8)

U2 cp_count

cp_info(CONSTANT_Methodref_info\CONSTANT_Fieldref_info\CONSTANT_Class_info\CONSTANT_Integer_info\CONSTANT_Utf8_info\CONSTANT_NameAndType_Info)-字面量和符号引用

U2 访问权限 PUBLIC FINAL

U2 类名 常量池中的class信息

U2 父类名 常量池中的class信息

U2 接口数量

接口名

U2 域数量

域表(类的静态变量,如果是final类型的变量（基本数据类型，引用类型除外）,则域下有个ConstantValue名字的属性,记录类变量的值)

U2 方法数量

方法表

init方法：实例变量的显示赋值、代码块语句、构造方法语句组成

clinit方法：静态变量的显示赋值、静态代码块语句

init(名字、描述符、访问标识)

code:bytecode\exception table(方法中有try\cathe语句，或者sync关键字)

LineNumberTable

LocalVariableTable

exceptions(方法上写了throws Exception)

U2 属性数量

属性表(sourceFile:文件名字ClassLoaderTest.java\sourceFile的字面量)

1.2 类加载过程

加载

把class文件加载入内存的过程，最终会生成class对象,存在于堆中,代码可以干涉,文件格式的验证与这个阶段同时进行

class对象中有指针指向方法区的类元数据，也有指针指向对应的classLoader对象

classLoader对象中有该Loader加载过的所有Class信息

双亲委派机制(避免重复加载、避免核心类被篡改)

AppClassLoader(加载应用程序的JAR包) extends UrlClassLoader->存在于Launcher类中

ExtClassLoader(加载jar/lib/ext目录下的jar包)extends UrlClassLoader->存在于Launcher类中

BootstrapClassLoader(加载jre/lib/rt.jar、resource.jar，同时为了安全只加载java\javax\sun开头的类)

在Launcher的构造函数中，调用ExtClassLoader的构造方法，然后把获取到的ext传给AppClassLoader的构造方法，生成AppClassLoader,然后设置Thread.contextClassLoader为AppClassLoader。

如果自定义ClassLoader类，extends ClassLoader并重写findClass方法，在ClassLoader类中loadClass方法中实现了双亲委派机制.

synchronized(ClassLoader对象) {

1 从ClassLoader缓存池中查找

2 if(null != parent) {

parent.loadClass();

} else {

bootstrapClassLoader加载class

}

3 if(没有加载过){

调用findClass方法加载

}

}

怎么打破双亲委派(重写loadClass方法、或者使用线程上下文加载器)

SPI（Service Provider Interface）:在META-INF/services/目录下定义一个名字是接口全限定名的文件，内容是接口的实现了该接口的类，多个类之间换行分隔。

Thread类中有一个contentClassLoader（ClassLoader）属性，一个threadLocals（ThreadLocal.ThreadLocalMap）属性（key=定义在业务类中的ThreadLocal变量，value=ThreadLocal中T类型的值）。

连接

验证

1 文件格式验证（CAFEBABE\Minor\Major）

2 语义验证(是否集成Object\是否重写了final方法或集成了final类)

3 字节码验证(iadd后的参数是否是long类型的)

4 符号引用验证（在解析阶段进行，验证是否存在、能否访问）

准备

为类变量赋0值或者NULL值，如果是final基本类型常量，则在此时进行显示赋值。（ConstantValue属性常量）,如果是final引用类型，没有ConstantValue,也不在此时赋值,而是在初始化时赋值。

解析

符号引用转化为直接引用

初始化：执行clinit方法（静态变量的显示赋值、静态代码块语句-可能不存在clinit方法）

clinit方法是由编译器生成的，由所有类变量的赋值动作和静态代码块中的语句合并产生。(父类的clinit方法在子类的clinit方法之前调用) clinit方法不是必须的，没有赋值动作和静态代码块，则没有clinit方法。类执行clinit方法时，不会先执行接口的clinit方法（如果有的话） 只有当接口变量被使用时才会调用.jvm必须保证一个类的clinit方法在多线程环境中被正确的加锁同步。而且clinit只会被jvm加载一次。

主动引用：

1 使用new\getstatic\putstatic\invokestatic指令时，如果没有进行初始化，则需要先触发初始化

2 使用反射方法对类进行反射调用，会触发初始化

3 初始化一个类时，如果父类没有初始化，先调用父类的初始化

4 程序启动的主类（MAIN方法的类），虚拟机会先初始化此类

被动引用（不会执行clinit方法）

1 通过子类引用父类的类变量，不会引起子类初始化

2 通过数组对象来引用类，不会触发此类的初始化

3 引用类中的常量的类变量，不会触发此类的初始化

1.3 字节码指令(操作码和操作数)

存储和加载指令：iload istore const_1 bipush sipush ldc

类型转换指令：i2l i2d i2f l2i d2i

运算指令:iadd dcmpg dcmpl fcmpg fcmpl lcmp

栈操作指令:pop pop2 dup dup2 swap dup_x1 dup_x2

对象创建指令:new newarray iaload iastore instanceof checkcast getstatic putstatic getspecial putspecial

控制转移指令:ifeq ifne ifnull ifgt ifge tableswitch lockswitch if_cmpge goto

异常指令:athrow(异常表、方法的异常属性)

方法执行指令:invokespecial invokestatic invokeinterface invokevirtual invokedynamic return(虚方法和非虚方法)

同步指令:moniterenter moniterexit（对应JMM原子操作的lock unlock）

2 运行时数据区

1 PC计数器-存储的是字节码指令的行号，线程私有，不会OOM，没有GC

2 虚拟机栈，会OOM，StackOverdueflowError，不会GC，每个方法对应一个栈帧，每个栈帧包含如下机构

局部变量表

底层是数组结构，大小在编译后已经确定，用来存储基本数据类型、引用类型、returnAddress类型，基本存储单元是slot，每个slot是4个字节（long和double占用两个slot）,如果是实例方法，则第0个位置是this对象所对应的内存地址的值，此处会有slot复用的情况。

操作数栈

底层是数组结构，大小在变异后已经确定，用来存储操作数（操作码对应的操作指令所需要的参数）。存储单元也是slot。只有入栈和出栈两个操作.此处可以思考i++和++i的问题。栈顶缓存技术。

动态链接

指向常量池中该帧所属方法的引用，持有这个方法是为了支持方法调用过程中的动态链接。

静态解析：类加载阶段转化为直接引用

动态链接：运行期间转化为直接引用

非虚方法（虚方法）：静态方法、私有方法、构造器方法、父类方法、final方法（静态方法、私有方法满足编译期可知，运行时不变，因此加载阶段进行解析）invokestatic invokespecial invokeinterface invokevirtual invokedynamic

方法返回地址

正常返回和异常返回，正常返回，则恢复到上个方法的调用入口位置。异常返回，则通过异常表来执行。需要执行的操作，恢复上个方法的操作数栈和局部变量表，把返回值压入调用者的操作数栈中，调整PC寄存器中的值。

相关参数:-Xss

3 本地方法栈

4 堆

分为：新生代和老年代，新生代包含一个Eden和2个Survivor区（总有一个空闲的）

对象创建的方式:1 new(xxFactory\xxBuilder) 2 反射 3 clone 4 反序列化

对象的创建步骤：

1 加载类信息

2 分配内存：线程安全问题：TLAB，cas+重试的方式,分配方式：指针碰撞、空闲列表

3 初始化内存空间-给属性对象赋0值

4 设置对象头信息

5 执行init方法

对象的访问方式：句柄池和直接指针

Object obj = new Object的字节码指令（理解对象的半初始化和DCL的volatile问题）

new

dum

invokespecial

astore

return

对象分配内存过程：

栈上分配->大对象->TLAB->Eden->survivor->Old

栈上分配（需要开启逃逸分析-XX:+DoEscapeAnalysis和标量替换-XX:+EliminateAllocations）：本质是把对象的属性打散分配到栈中

大对象：-XX:PretenureSizeThreshold(大对象大小，只在serial\parnew垃圾收集器中起作用)

空间分配担保：在发生minor GC之前，虚拟机会检查老年代的最大可用连续空间是都大于新生代所有对象总空间，如果大于，则此次GC是安全的，如果不满足，查看HandlePromotionFailure是否开启，如果开启，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的对象的平均大小，如果大于，进行Minor GC,否则进行Full GC.

正常情况下，新生代中对象大于MaxTenuringThreshold后会进入老年代，但是如果Survivor区中某个相同年龄的对象占整个区域的一半大小时，大于等于这个年龄的对象都会晋升到老年代.

对象如何进入老年代，通常有4种情况会导致对象进入老年代

动态年龄判断

大对象直接进入老年代

超过MaxTenuringThreshold阈值年龄(默认15)的对象进入老年代

空间分配担保机制，如Survivor区无法放入所有的存活对象时

对象的状态：可触及、可复活（finallze方法，只会调用一次）、不可触及

对象的引用：强、软、弱、虚（必须配合reference-queue一起使用，在垃圾回收时接收一些通知）

垃圾标记算法：引用计数（需要空间、循环引用python使用-null\弱引用）和根搜索算法（算法复杂、无循环引用问题）

gc-roots有哪些：栈中指针指向的对象、方法区中常量和静态变量指向的对象（1.7中移至堆中）、JVM所需的对象、锁对象、活动的线程等。

对象标记阶段需要STW，同时为了提高效率维护了OOPMAP，只有在safepoint时才会更新OOPMAP,safepoint只有在方法调用、循环跳转、异常跳转等地方。抢先试中断和主动式中断，saferegion时，OOPMAP不会更新，所以GC时忽略这些线程，当线程走出saferegion时，查看GC时候完成，如果没完成，等待GC完成后执行。

进行YGC时，除了常规的GC-ROOTS外，还会把老年代中的对象当成GC-ROOT，但是为了这种跨代引用是很少的，为了解决这个问题，在新生代维护了RSet集合（为什么老年代没有RSet?），在发生YG时，只有跨代引用的对象才会加入到GC-ROOT中。但是需要在对象引用关系改变时位数RSet数据的正确性，增加一些开销，但是这些开销对扫描整个老年代来说是划算的。

hotspot虚拟机采用cardtable的方式实现RSet，cardtable底层是字节数组，每个字节数组中的元素都对应着内存区中中一块特定的大小，叫做卡页，一般来说卡页大小是2的N次幂，默认是2的9次幂是512个字节。每个卡页中有多个对象，只要有一个对象存在跨代引用，则相应的卡表就变成1，称为变脏。

卡表何时变脏，如何变脏？

引用类型字段赋值时，如果发现有了跨代引用就变脏。

解释执行时好处理，可以在字节码层面操作，编译执行时，都已经是机器指令了，所以必须在机器码层面解决这个问题，吧维护卡表的动作放到每一个赋值动作操作之中。HOTSPOT是通过写屏障（write barrier）维护卡表的。区分写屏障和内存屏障。写屏障可以看做对写操作的环绕通知，在G1垃圾收集器出现之前，其他垃圾收集器只使用了写后屏障。即在赋值动作后，更新RSET，无论是不是老年代对新生代的应用，相对扫描整个老年代来说，性价比高。此处还需要考虑缓存行的问题，为了解决这个问题，先检查卡表记录，只有记录是0是才更新。

三色标记算法：用户解决并发标记的问题

白色：尚为被扫描的对象，或者是不可达对象。

黑色：对象已经扫描过&对象的所有引用也扫描过

灰色：表示对象被垃圾收集器访问过，单这个对象至少存在一个引用还没被扫描

产生问题：浮动垃圾和错误标识

要消除错误标识：1 赋值时插入了一条或者多条从黑色对象到白色对象的引用 2 赋值时删除了全部从灰色对象到白色对象的直接或者间接引用

要解决错误标记问题，只需要破坏一条即可。对应增量更新(CMS采用这种方案)和原始快照(G1采用这种方案)两种方案。

垃圾回收算法：

标记清除算法：执行效率适中、空间要求低、会产生内存碎片

复制算法：执行效率高、空间要求高、不会产生内存碎片

标记整理算法：执行效率低、空间要求低、不会产生内存碎片

垃圾收集器：

serial（def new generation）：复制算法、单线程串行，适用单核小内存的机器上，新生代 -XX:+UseSerialGC

serial old（tenured generation）：标记整理算法、单线程串行，适用单核小内存的机器上，老年代 -XX:+UseSerialOldGC

parallel scavange(PSYoungGen):并行垃圾回收器，采用复制算法，多线程并行，吞吐量优先垃圾回收器，新生代 -XX:+UseParallelGC -XX:ParallelGCThreads -XX:MaxGCPauseMills -XX:GCTimeRatio -XX:+UseAdaptiveSizePolicy(不要主动设置年轻代、老年代、年龄、Eden的大小)

parallel old(ParOldGen):并行垃圾回收器，采用标记整理算法，多线程并行，吞吐量优先垃圾回收器，老年代 -XX:+UseParallelOldGC

parnew(par new generation):并行垃圾回收器，新生代 -XX:+UseParNewGC -XX:ParallelGCThreads(默认是CPU核数)

cms(concurrent mark-sweep generation):并发垃圾收集器，老年代，低延迟，采用标记清除算法，CMS的默认收集线程数量是=(CPU数量+3)/4，会产生内存碎片,会发生Concurrent Mode Failure -XX:UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction（1.8版本是92%） -XX:+UseCMSInitiatingOccupancyOnly 一般和前者配合使用，思考如果不配合使用会怎么样？，-XX:+UseCMSCompactAtFullCollection(默认true) -XX:CMSFullGCsBeforeCompaction -XX:ConcGCThreads

初始标记、并发标记、重新标记、并发清除。采用的是增量更新的形式。

G1(garbage-first heap)：分区模式，region(不要主动设置年轻代、老年代、年龄、Eden的大小)，每个region都可以是eden\survivor\old\humongous,避免使用-Xmn、-XX:NewRatio等显式设置Young区大小，会覆盖暂定时间目标,暂停时间不要太苛刻，吞吐量目标为90%，太严苛影响吞吐量,Rset、CardTable和region关系，更新RSet会放入到缓存中，后续异步更新。衰减标准偏差来预测回收Region所需要的时间 -XX:UseG1GC -XX:G1HeapRegionSize -XX:MaxGCPauseMillis -XX:InitiatingHeapOccupancyPercent -XX:ParallelGCThreads -XX:ConcGCThreads -XX:G1MixedGCCountTarget -XX:G1HeapWastePercent -XX:G1OldCSetRegionThresholdPercent -XX:G1MixedGCLiveThresholdPercent

YGC（STW，复制算法）并发周期 MixedGC FULLGC

上文中，多次提到了global concurrent marking，它的执行过程类似CMS，但是不同的是，在G1 GC中，它主要是为Mixed GC提供标记服务的，并不是一次GC过程的一个必须环节。global concurrent marking的执行过程分为四个步骤： * 初始标记（initial mark，STW）。它标记了从GC Root开始直接可达的对象。 * 并发标记（Concurrent Marking）。这个阶段从GC Root开始对heap中的对象标记，标记线程与应用程序线程并行执行，并且收集各个Region的存活对象信息。 * 最终标记（Remark，STW）。标记那些在并发标记阶段发生变化的对象，将被回收。 * 清除垃圾（Cleanup）。清除空Region（没有存活对象的），加入到free list。第一阶段initial mark是共用了Young GC的暂停。第四阶段Cleanup只是回收了没有存活对象的Region，所以它并不需要STW。

相关参数:-Xms -Xmx -Xmn -XX:NewRatio -XX:SurvivorRatio -XX:+DoEscapeAnalysis -XX:+EliminateAllocations -XX:PretenureSizeThreshold -XX:MaxTenuringThreshold=15 -XX:PermSize -XX:MaxPermSize -XX:MaxMetaspaceSize -XX:MetaspaceSize -XX:+UseTLAB -XX:+HandlePromotionFailure

GC日志分析

```
2020-12-09T19:05:14.542+0800: [GC (Allocation Failure) [PSYoungGen: 2048K->504K(2560K)] 2048K->997K(9728K), 0.1283340 secs]
GC发生时间:[GC类型(GC原因)[垃圾收集器类型:YGC前新生代占用->YGC后新生代占用(新生代总大小)]YGC前堆占用->YGC后堆占用(堆总大小),YGC耗时]
[Times: user=0.00 sys=0.02, real=0.13 secs]
[YGC用户耗时 YGC系统耗时 YGC实际耗时]
2020-12-09T19:05:14.705+0800: [GC (Allocation Failure) [PSYoungGen: 1858K->504K(2560K)] 8496K->7317K(9728K), 0.0007040 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
2020-12-09T19:05:14.706+0800: [Full GC (Ergonomics) [PSYoungGen: 504K->500K(2560K)] [ParOldGen: 6813K->6735K(7168K)] 7317K->7236K(9728K), [Metaspace: 3558K->3558K(1056768K)], 0.0063708 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
2020-12-09T19:05:14.712+0800: [Full GC (Allocation Failure) [PSYoungGen: 500K->493K(2560K)]
GC发生时间:[GC类型(GC原因) [Young区:GC前Young区内存占用->GC后Young区内存占用(Young区总大小)]
[ParOldGen: 6735K->6723K(7168K)] 7236K->7217K(9728K), [Metaspace: 3558K->3558K(1056768K)], 0.0065637 secs]
[old区:GC前old区内存占用->GC后old区内存占用(old区总大小)]GC前堆内存占用->GC后堆内存占用(堆总大小),[元空间:GC前元空间内存占用->GC后元空间内存占用(元空间总大小),GC耗时]


[Times: user=0.09 sys=0.00, real=0.01 secs]
[用户时间 系统时间 实际耗时]


规律：名称：GC前内存占用->GC后内存占用 该区域总大小
real —— 程序从开始到结束所用的时钟时间（进程阻塞的时间）
user —— 进程执行用户态代码（核心之外）所使用的时间
```

```
sys —— 进程在内核态消耗的 CPU 时间
```

5 方法区 会OOM，会GC

类信息 、常量、静态变量、JIT及时编译的代码

其中字符串常量池(intern方法)和静态变量在1.7中都放入了堆中

1.7之前方法区的实现是永久代，1.8之后是元空间

-XX:PermSize -XX:MaxPermSize -XX:MaxMetaspaceSize -XX:MetaspaceSize

类回收的条件：1 ClassLoader不使用 2 Class对象不使用 3 实例对象不使用

6 直接内存（不受JVM管理），会OOM,会发生GC

nio会直接使用直接内存DirectByteBuffer，0copy技术.

相关参数：-XX:MaxDirectMemorySize默认与-Xmx一致

3 执行引擎

解释执行和编译执行（为什么两者都存在？）

4 杂

```
* java spi机制
* spi=service provider interface
* api=application program interface
*
* 站在开发者的角度看，api是开发人员调用之后可以完成某个功能，而spi则是，需要又开发者完成，供框架调用。
* API的实现是框架，SPI的实现是开发者
*
* SPI中一个核心的类     ServiceLoader
* 文件放在META-INF下
* 文件名字是：SPI接口名称
* 文件内容是：实现了SPI接口的实现类（如果有多个，换行表示）
*
* 一个典型的SPI的应用场景是JVM加载JDBC的过程
*       之前的开发方式中，需要程序员手动引入MYSQL类
Class.forName("com.mysql.jdbc.Driver")  这种情况AppClassLoader会加载该类
*
*       现在的方式中，不用再直接引入该类，而在JVM加载DriverManager(rt.jar包中的类)时，在DriverManager的静态代码块中使用ServiceLoader把
*       META-INF/services下的java.sql.Driver文件的com.mysql.jdbc.Driver加载到JVM中
*       此处有一个问题，DriverManager是BootstrapClassLoader加载的，
com.mysql.jdbc.Driver需要AppClassLoader加载，是通过
*       Thread.currentThread().getContextClassLoader()获取线程上下文加载器完成的--这个就是打破了双亲委派机制
*
*
* 总结：SPI是通过ServiceLoader + META-INF + （固定文件格式）完成的
*       打破双亲委派是通过ContextClassLoader实现的
*
*
```

```
*  SPI的问题：
*       不能按需加载，需要遍历所有的实现，并实例化
*
*  延伸一个问题：DUBBO的SPI机制：
*                   通过ExtensionLoader + META-INF/dubbo/internal + (固定文件格式)
 文件名称是:DUBBO的接口名称    文件内容支持多行记录   每行记录key=DUBBO的接口实现类
*               SPRING的SPI机制：
*                   通过SpringFactoriesLoader + META-INF + spring.factories(文件名字是固定的，其它两个都是接口名字) 文件内容是  key=value,value  key是接口的全名  value是接口的实现
```

```
* ThreadLocal<T>
*
* 典型的以空间换时间的多线程并发解决方案
*
* 原理：
* 每个Thread中都有一个ThreadLocalMap属性threadLocals,该对象是一个KV结构对象
* 其中key=ThreadLocal对象本身  Value=T对象，也就是你真正需要的对象
* 注意，key是WeakReference
* 此处可能会产生内存泄漏问题，原因
* 如果线程是一个线程池中的核心线程（长时间存在），那么value这个值会长时间存在内存中，
* 虽然这个值已经没有用了（因为KEY是弱引用，GC的时候已经回收，但是VALUE的指针会一直存在线程中）
```

```
* String 是final的  为什么？
* final关键字    如果是static变量 要么在声明时指定，要么在静态代码块中指定，如果是成员变量 要么在声明时指定，要么在构造方法中指定 final修饰类 final修饰方法 final修饰变量
* 底层结构是什么?
* +操作的原理
* StringBuilder StringBuffer的区别
* 字符串常量池、类的常量池、运行时常量池的关系
* 字符串常量池在运行时常量池中（在1.7之前在运行时常量池中，在1.7之后在堆中）
* intern()方法
* 方法区是接口->永久区和元空间是实现
```

```
* 如果extends  ClassLoader
* ClassLoader.loadClass();双亲委派在这个方法中实现，底层会调用findClass方法
* ClassLoader.findClass();如果想要重写ClassLoader，只需要重写该方法即可
*
*
* AppClassLoader extends URLClassLoader是Launcher静态内部类
*
* ExtClassLoader extends URLClassLoader是Launcher静态内部类
*
* class对象中有指针指向方法区的类元数据，也有指针指向对应的classLoader对象
*
* classLoader对象中有该Loader加载过的所有Class信息
*
*
*
* 加载：(会生成堆中的Class对象)把文件从某个地方加载进JVM中（本地、JAR、网络、数据库等等）
* 如果要对class内容进行加解密，在此处进行，重写findClass方法，数组类型不是通过classloader加载的，
* 是JVM直接在内存中动态构建的，但是数组相关类型是通过classloader加载的
*
*
```

```
* BootstrapClassLoader(加载jre/lib/rt.jar、resources.jar)
* ExtClassLoader(加载jre/lib/ext目录下的jar包)
* AppClassLoader
*
* 涉及到双亲委派:
* 打破双亲委派:线程上下文类加载器
*
* SPI(Service Provider Interface)
* DriverManager静态代码块中进行
* ServiceLoader.load(Driver.class)会使用Thread属性中有一个名字是contextClassLoader的ClassLoader用来加载相关的类
* 调用循环方法时:去META-INF/services/下面找Driver的文件
* 文件的名称是:定义的接口类的名字如java.sq.Driver,文件内容是实现了该接口的类:包+类名称,多个实现类之间换行区分
*
*
* 连接
*       验证:
*            如果校验失败会报错:VerifyError
*            1 文件格式验证:魔数验证、版本验证
*            2 元数据验证:是否有父类、是否继承了final类、是否重写了final方法
*            3 字节码验证:验证方法中Code属性中的字节码是否合法
*            4 符号引用验证(发生在解析阶段):通过全限定名是否能找到相应的类,符号引用对应的类、方法、属性能否被访问等
*       准备:
*            1 一般情况下为静态变量赋0值(静态变量在1.8之后在堆中)
*            2 如果是final常量&是基本数据类型或者字符串字面量,该field有constantvlaue属性,则在此时赋值
*       解析:符号引用转化为直接引用
*            符号引用:CONSTANT_Methodref_info\CONSTANT_Fieldref_info\
*
 CONSTANT_Class_info\CONSTANT_Integer_info\CONSTANT_Utf8_info\CONSTANT_NameAndType_Info
*            直接引用:内存地址
*
* 初始化:执行clinit方法
*       clinit方法是由编译器生成的,由所有类变量的赋值动作和静态代码块中的语句合并产生。(父类的clinit方法在子类的clinit方法之前调用)
*       clinit方法不是必须的,没有赋值动作和静态代码块,则没有clinit方法。类执行clinit方法时,不会先执行接口的clinit方法(如果有的话)
*       只有当接口变量被使用时才会调用.jvm必须保证一个类的clinit方法在多线程环境中被正确的加锁同步。而且clinit只会被jvm加载一次。
*       主动使用
*       被动使用:1 调用父类静态变量 2 调用常量 3 数组对象
```

```
* 创建对象的方式:
* 1 new (XXXBuilder\XXXFactory)
* 2 反射:class.newInstance(),只能是public无参的构造方法,constructor.newInstance()
* 3 clone(),深COPY和浅COPY
* 4 反序列化(文件、网络)
*
* 对象创建的步骤:
*
* 例如:Object obj = new Object();
*  底层字节码指令
*  new #8    // class java/lang/Object
*  dup
```

```
*  invokespecial #1 //Method java/lang/Object."<init>":()V
*  astore_1
*  return
*
*  理解对象的半初始化（以及DCL问题）
*  虚方法和非虚方法（编译期就确定下来的方法，如private方法、构造器、父类方法、静态方法、final方法）
*  执行非虚方法:invokestatic invokespecial
*  执行虚方法:invokevirtual invokeinterface
*  invokedynamic执行lamda表达式的
*
* 1 判断对象的类信息是否加载，如果没有加载（加载、链接（验证、准备、解析）、初始化）
* 2 为对象分配内存 new
*       此处需要考虑两个问题
*           1 采用什么方式分配内存：指针碰撞（内存连续）和空闲列表（内存不连续CMS）
*           2 处理并发问题：CAS+失败重试，TLAB
* 3 初始化内存空间-实例变量赋0值
* 4 设置对象头信息（是否开启了偏向锁、开启了指针压缩、开启了oops压缩）
* 5 执行方法的init方法
*
* 对象访问的方式：
* 1 句柄池
* 2 直接指针
*
* 垃圾收集：
* 1 垃圾收集算法：
*       1 什么是垃圾：不再使用的对象
*       2 如何标记垃圾对象：
*           1 引用计数算法：优点，实现简单、效率高；缺点：需要记录引用次数，浪费空间和时间，无法解决循环引用问题
*           2 根搜索算法：可以解决循环引用问题
*               gc-roots有哪些：（非堆指向堆对象）
*                   1 两个栈中的指针指向的对象
*                   2 方法区中常量、方法区中的静态变量（1.7之后字符串常量池和静态变量移至堆中）
*                   3 JVM所需要的对象（rt.jar中的类生成的对象）
*                   4 锁对象
*                   5 活动的线程对象
*           3 对象标记阶段需要STW,同时为了提高效率维护了OOPMAP,OOPMAP只有到了safepoint才会更改
*       3 如何回收垃圾对象：
*           1 标记-清除算法：效率中等、内存碎片、空间要求小
*           2 复制算法：效率高、内存整齐、空间要求大
*           3 标记-整理算法：效率最低、内存整齐、空间要求小
*           4 分代收集算法：
*           5 增量收集算法：
*           6 分区收集算法：
* 2 垃圾收集器：
*       1 seria（GCDetails中的信息是：def new generation）和serial old（GCDetails中的信息是：tenured generation）
*           -XX:+UseSerialGC
*           适用于单核小内存的场景
*           采用复制和标记-压缩算法
*       2 parNew(par new generation)
*           -XX:+UseParNewGC
*           -XX:ParallelGCThreads
*           serial的多线程版本,用于新生代,采用复制算法
```

```
*       3 parallel scavenge(GCDetails中的信息是：PSYoungGen)和parallel old(GCDetails中的信息是：ParOldGen)
*           -XX:+UseParallelGC
*           -XX:ParallelGCThreads(默认是cup的核数)
*           -XX:MaxGCPauseMillis
*           -XX:GCTimeRatio
*           -XX:+UseAdaptiveSizePolicy
*           与parNew的区别
*               1 吞吐量优先的垃圾收集器
*               2 有自适应策略
*       4 cms(GCDetails中的信息是：concurrent mark-sweep generation)
*           -XX:+UseConcMarkSweepGC
*           -XX:CMSInitiatingOccupancyFraction(1.8默认是92%)
*           -XX:+UseCMSInitiatingOccupancyOnly（默认是true）
*           -XX:UseCMSCompactAtFullCollection
*           -XX:CMSFullGCsBeforeCompaction
*           -XX:ConcGCThreads
*           -XX:ParallelGCThreads
*
*           用于老年代垃圾回收，采用标记-清除算法，会产生内存碎片，并发的垃圾回收器，
*           不能等到内存满了之后再进行gc,，存在回收失败的情况（concurrent mode failure），需要使用serial old作为后备方式
*           低延迟的垃圾收集器
*
*           1 初始标记(STW)
*           2 并发标记
*           3 最终标记(STW，采用增量更新算法-三色标记)
*           4 并发清除
*       7 G1(GCDetails中的信息是：garbage-first heap)
*           -XX:UseG1GC
*           -XX:G1HeapRegionSize(1M-32M,2的N次幂),Region数量在2048左右
*           -XX:MaxGCPauseMillis
*           把堆分成大小相同的N个Region,每个Region都可以是Eden,Survivor,old区，还有一个大对象区，只要超过Region一半都属于大对象。
*           g1把region当成单次回收的最小单元，每次回收都是region的整数倍，G1在后台维护一个优先级列表，根据设置的最大停顿时间，默认200毫秒，
*           优先处理回收价值最大的Region
*           -XX:InitiatingHeapOccupancyPercent（当整个Java堆的占用率达到参数值时，开始并发标记阶段；默认为45）
*           -XX:ParallelGCThreads
*           -XX:ConcGCThreads
*           -XX:G1MixedGCCountTarget默认值8在一次全局并发标记后，最多接着8此Mixed GC也就是会把全局并发标记阶段生成的Cset里的Region拆分为最多8部分，然后在每轮Mixed GC里收集一部分
*           -XX:G1HeapWastePercent默认值5%，也就是在全局标记结束后能够统计出所有Cset内可被回收的垃圾占整对的比例值，如果超过5%，那么就会触发之后的多轮Mixed GC，如果不超过，那么会在之后的某次Young GC中重新执行全局并发标记。可以尝试适当的调高此阈值，能够适当的降低Mixed GC的频率
*           -XX:G1OldCSetRegionThresholdPercent默认10%，也就是每轮Mixed GC附加的Cset的Region不超过全部Region的10%
*           -XX:G1MixedGCLiveThresholdPercent在全局并发标记阶段，如果一个Region的存活对象的空间占比低于此值，则会被纳入Cset
*
*
*           衰减标准偏差-最近的参考价值越大
*
*           新生代的占比是从5%-60%
*
*           缺点是：需要的内存空间更大(RSet需要占用很大的内存空间)，在小内存上性能可能还不如cms（均衡点在6-8G之间）
```

```
*
*           YGC（整个过程STW）：
*               1 扫描GC-ROOTS（包含RSet中的数据）
*               2 读取dirty card queue中的数据，更新RSet(dirty card queue是每个赋值操作都会通过写屏障插入记录,每个Region都会读取queue中的数据更新RSet)
*               3 复制算法收集内存(Collection set就是待回收的Region集合，空闲的Region会被放入到linked list中供JVM使用)
*           Mixed GC(YGC和部分old区Region的回收)---当内存使用默认超过45%的时候触发并发标记然后执行Mixed GC,Mixed GC默认会进行8次，
*               即每次Mixed GC只会回收并发标记的1/8的老年代Region,但是默认当内存使用情况小于10%的时候就结束Mixed GC，每个Region只有垃圾超过65%才会进行回收
*               1 初始标记(STW,会触发一次YGC)
*               2 并发标记
*               3 重新标记（STW，采用原始快照的方式-三色标记）
*               4 筛选回收(STW，通过上面的流程已经能筛选出性价比最高的Region，然后设定的最大停顿时间，进行相应回收)
*
*
*       8 ZGC
*
*       组合方式：
*       1 serial->serial old
*       2 parNew->serial old
*       3 parallel scavenge -> serial old
*       4 serial->cms
*       5 parNew->cms->serial old
*       6 parallel scavenge -> parallel old（1.7 1.8默认垃圾收集器）
*       7 g1(1.9默认垃圾收集器)
* 3 垃圾收集相关概念：
*       oopMap
*           栈中的指针不全都是指向堆中对象的，找出栈中的指向堆中对象的指针，把他们记录到oopMap中，进行minor gc时，只需要扫描这些oopMap对象就可以
*           栈中的操作会改变oopMap,如果每次都去修改oopMap太耗性能，需要设置了safepoint，在安全点修改oopMap,如果安全点太少，线程不太容易执行到安全点，
*           如果太多会频繁修改oopMap
*       Rememberd set
*           为了解决跨代引用，避免把整个老年代加入到GC-ROOTS中（包括各种部分区域收集的垃圾回收器，如g1,zgc等，都有相同问题），
*           有三种精度的实现方案，字长精度，对象精度，卡精度（card table）
*       card tablb(Remembered set的具体实现，一个"字节"数组)：
*           存在与新生代，用于记录老年代中的跨代引用（把老年代按照512byte的大小分成N多份，如果该部分任意一个对象存在跨代引用，
*           就把相对应的card table的值变成1，minor gc进行时，只需要扫描相应的card table就可以）
*           有了卡表之后需要考虑，什么时候更新，怎么更新卡表为dirty的问题？
*           采用的是在执行赋值（包含跨区域赋值和非跨区域赋值）的机器指令的时候（主要是考虑jit即时编译的代码），执行写屏障（写环绕通知），（思考下读写屏障和内存屏的不同）
*           每次引用更新都会产生额外的开销，不过这与minor gc时扫描整个老年代代价还是低。
*           出了这个开销之外，还有一个缓存行的问题，卡表一个元素一个字节，64个卡表共享一个缓存行，如果不通线程更新位于64*512范围内的
*           对象时，就会有问题。采用了先检查，只有当该卡表元素不是dirty时，才更新。
*       safepoint和saferegion：
*           当发生gc时，线程不能马上停止，必须在safepoint点才能停止（方法的调用、循环跳出、异常跳出）
*               线程停止有两种形式：1 抢先式停止（线程先中断，如果没有到安全点，恢复继续跑到安全点停止） 2 主动式停止（设置一个标志，线程运行到安全点，查询标志）
*           如果gc发生时，线程处于block状态（sleep）等时，不能等待线程跑到安全点，但是此时引用关系也不会发生变化，所以进入安全区域。
```

```
*                线程阻塞时，线程标记进入安全区域，当gc发生时，会忽略进入安全区域的线程。但是等线程要出安全区域的时候，会先查看gc时候完成，如果没有完成，则等待gc完成后唤醒线程继续执行。
*       强引用(死也不回收)：
*           Object obj = new Object(); 导致oom的元凶
*       软引用（内存不够就回收，可以通过引用获取对象）：
*           Object obj = new Object();
*           SoftReference softReference = new SoftReference<>(obj);
*           obj = null;    必须释放obj引用
*       弱引用（垃圾收集就回收，可以通过引用获取对象）
*           WeakReference weakReference = new WeakReference<>(new Object());
*       虚引用（不可以通过引用获取对象，必须和ReferenceQueue队列配合使用,垃圾回收发生时，会把对象放入queue中，进行相应处理）
*           Object obj = new Object();
*           ReferenceQueue q = null;
*           PhantomReference phantomReference = new PhantomReference<Object>(object, obj);
*       三色标记（用于标记阶段）：
*           黑：表示已经被垃圾收集器访问过，且这个对象的所有引用（指向别的对象的引用）都已经扫描过。黑色对象表示它已经扫描过，并且是存活的。
*           白：表示未被垃圾收集器访问过。可达性分析刚开始，所有对象都是白色的，分析阶段结束后，白色的代表不可达对象。
*           灰：表示已经被垃圾收集器访问过，但这个对象上至少有一个引用还没扫描过。
*           如果，用户线程stw，则没问题。
*           如果，用户线程和垃圾回收线程并发执行，会出现两种问题。
*               1 浮动垃圾。把原本垃圾对象标记为存活对象。
*               2 存活对象标记为垃圾对象。
*               为了解决问题2，有两种方案：
*                   1 增量更新（黑色对象插入新引用）：cms采用，当黑色对象插入新的指针指向白色对象时，把这个关系记录下来，等并发扫描结束后，再将这些黑色对象为根再扫描一次。
*                   2 原始快照（灰色对象删除新引用）：G1采用，当灰色对象要删除白色对象的引用关系时，把删除的引用关系记录下来，等并发扫描结束后，再将这些灰色对象扫描一次。
*
```

```
* 堆空间分为：新生代和老年代
* 新生代分为：一个Eden区，两个Survivor区
*
* 对象分配流程：
* new一个对象：
*  如果开启了逃逸分析：如果在栈范围内，首先尝试在栈上分配
*  如果不能在栈上分配，首先尝试在Eden区的TLAB(Thread Local Allocation Buffer)尝试分配
*  如果TLAB不能分配，则尝试在Eden区分配，如果Eden空间不足，则进行一次Minor GC，如果GC后还不能存放，则直接放在老年代。
*  如果老年代也放不下，触发一次FULL GC， FULL GC之后仍不能放下，则OOM
*  YGC中，会清空Eden和一个Survivor，然后把存活的对象复制到另一个Survivor中，如果对象的年龄大于阈值，直接进入老年代
*
*
*
* 优先分配Eden
* 大对象直接进入老年代
* 长期存活的对象进入老年代
* 如果Survivor中相同年龄的所有对象的大小的总和大于Survivor空间的一般，年龄大于或者等于该年龄的对象直接进入老年代
* 空间分配担（前提是在要进行Minor GC之（可以理解成eden区满的时候），如果老年代中的连续内存大于新生代的对象总大小或者历次晋升的平均大小就会进行Minor GC,否则进行Full GC）
*
*
```

```
* 什么时候触发Minor GC: Eden满的时候触发，S0和S1并不会触发GC，但是GC的时候会清理Survivor
* 什么时候触发Full GC: 老年代满的时候触发，方法区满的时候触发，Eden区满的时候，老年代的内存空间小与新生代的对象总大小和历次晋升平均大小进行FULL GC
*
* 所有对象都只能在堆上分配么？并不是，还可以在栈上分配
* 堆上的所有对象都是共享的么？并不是，TLAB，线程私有的
*
*
*
*
*
* 指定堆大小
* -Xms（默认情况下是内存的1/64）
* -Xmx（默认情况下是内存的1/4） 生产情况下Xms和Xmx的值是一样的（思考下为什么？）
* -Xmn（新生代大小）
* -XX:NewRatio（老年代与新生代的比例，默认是2）
* -XX:SurvivorRati（Eden与Survivor的比例，默认是8，但是本地测试一般是6，为什么？）
* -XX:+UseTLAB(开启TLAB，默认情况下是eden的1%，可以通过参数设计TLAB的大小)
* -XX:MaxTenuringThreshold(设置晋升到老年代的阈值，默认是15，最大也只能是15？思考为什么？)
* -XX:HandlePromotionFailure（空间担保策略，默认是开启）
* -XX:PrintFlagsInitial（打印参数初始值）
* -XX:PrintFlagsFinal（打印参数最终值）
* -XX:PrintGC         -Xlog: gc(打印GC基本信息)
* -XX:PrintGCDetails -Xlog: gc*(打印GC详情信息)
* -XX:PrintHeapAtGC  -Xlog: gc+heap=debug（查看GC前后方法区、堆的可用容量变化）
* -XX:PrintGCApplicationConcurrentTime -XX:PrintGCApplicationStoppedTime -Xlog:safepoint(查看GC过程中用户并发时间和停顿时间)
* -XX:PrintAdaptiveSizePolicy(自动设置堆空间各分代区域大小、收集目标等内容。从parallel收集器开始)
* -XX:PrintTenuringDistribution -Xlog: gc+age=trace(收集过后，存活对象的年龄分布信息)
* 生产GC日志配置
* -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/home/GCEASY/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSize=20M
*
* -XX:+PrintGCDetails
* -XX:+PrintGCDateStamps
* -Xloggc:/home/GCEASY/gc-%t.log（当JVM重启以后，会生成新的日志文件，新的日志也不会覆盖老的日志，只需要在日志文件名中添加%t的后缀即可，%t会给文件名添加时间戳后缀，格式是YYYY-MM-DD_HH-MM-SS）
*
* -XX:+HeapDumpOnOutOfMemoryError
* -XX:HeapDumpPath
*
* -XX:TraceClassLoading
*
* -XX:PretenureSizeThreshold(大对象大小，只在serial\parnew垃圾收集器中起作用)
*
* java -XX:PrintFlagsFinal | more
* java -XX:PrintFlagsFinal | wc -l
*
* jps
* jinfo -flag flagName pid
* jinfo -flags pid
* jinfo -flag +flagName pid
* jinfo -flag flagName=xx pid
*
* jstat
* jstack
```

```
* jmap
```

```
* 写屏障
* 内存屏障
*
* 主内存->read->load->工作内存->user->工作线程
* 工作线程->assign->工作内存->store->write->主内存
*
* synchronized关键字底层原理：
* 1 用户态、内核态
* 2 cas
*       cas会引起aba问题，解决方案:AtomicStampedReference
*       cas怎么解决原子性问题：lock
```

```
* 类变量初始过程
* 成员变量初始化（new 执行init方法 栈指针指向可能会触发DCL问题）
*
* 给类变量赋值有两种方式：第一是显式赋值，第二是静态代码块赋值     这两种方式最终会变成clinit方法
*
* 给实例变量赋值有三种方式：第一种是显式赋值，第二种是代码块赋值，第三种是构造函数赋值 这三种方式最终会变成init方法
```

```
/**
 * 常量，编译期就已经确定下来了
 * 如果是static final 必须有默认值
 * 思考？是否可以对static final常量重新赋值？静态代码块？-不可以，编译期就已经确定了
 * */
```

```
* 打印类加载情况:-verbose:class
*
* -XX:+TraceClassLoading
*
* -XX:+TraceClassUnLoading
```

```
* 逃逸分析：栈分配、锁消除、标量替换
*
* 方法区：类信息、常量、静态变量、JIT及时编译的代码
*
* 类信息：类基本信息、域信息、方法信息
* 类基本信息：类的全限定名、父类的全限定名、接口的全限定名、类的修饰符
* 域信息：名称、类型、修饰符
* 方法信息：方法名称、方法返回值、方法参数列表、方法的修饰符、方法的字节码指令、异常表
*
* 常量信息：数据类型的常量池（字符串类型的常量池1.6之前在方法区，1.7之后迁移到堆中）
*
* 静态变量：类的静态变量是存在方法区中的（final类型的静态变量是常量，在编译期就已经确定下来）
*
* JIT及时编译的代码：热点代码的编译执行，性能提升的关键
*
* 常用参数
* 1.7之前    -XX:PermSize -XX:MaxPermSize
* 1.8之后    -XX:MetaspaceSize(默认大小21M) -XX:MaxMetaspaceSize(我们项目中的大小在130M左右)
*
```

```
* 主要是JVM Stack的相关知识
*
* 可设置的参数-Xss   stacksize
*
* 此空间不会发生GC，但是会OOM和StackOverFlowError
*
* 每个方法对应一个栈帧,进入一个方法对应着一个栈帧的入栈和出栈
* 每个栈帧包含：局部变量表、操作数栈、动态链接和方法返回地址
*
* 局部变量表存在：基本数据类型、引用类型、ReturnAddress类型(在编译器大小就已经确定)
* (这个是数组结构，里面存储的是数字类型的值，每个单元称为一个Slot，大小是4个字节，
* 所以double和long需要占用两个空间，就是两个Slot,如果是实例方法（包含构造方法），
* 局部变量表中的第一个位置是this对象（所以在这些方法中才可以使用this），静态方法没有this,
* 还有一点是如果超过了作用域(在一个方法中定义代码块)，可能会涉及到slot重复利用的问题)
*
* 操作数栈: 作用是临时存储数据，只有两个操作：入栈和出栈(在编译器大小就已经确定)
* 这个也是数组结构，只是当成栈使用（所以不能指定索引取值,只有入栈和出栈两个操作）
*
* 思考：JVM运行时，局部变量表&操作数栈&执行引擎是怎么配合执行的
*
* 动态链接
*
* 方法返回地址
```

```
/**
 * 可以使用-XX:+PrintFlagsInitial  -XX:+PrintFlagsFinal查看所有属性值
 * -XX:+PrintGCDetails -XX:+PrintCommandLineFlags
 * -XX:+UseCompressedOops  -XX:+UseCompressedClassPointers
 * jps
 * jinfo [option] <pid>
 *      -flag <name>         to print the value of the named VM flag
 *      -flag [+|-]<name>    to enable or disable the named VM flag
 *      -flag <name>=<value> to set the named VM flag to the given value
 *      -flags               to print VM flags
 * jinfo -flag ThreadStackSize 9 打印9这个进程的-XX:ThreadStackSize的值
 * jinfo -flags 9 打印9这个进程的所有flags
 * jinfo -flag +PrintGCDetails 开启(关闭)GC打印
 * jinfo -flag ThreadStackSize=1024 设置变量的值
 *
 * 模拟实现StackOverFlowError
 * 此处可以设置-Xss大小(64位操作系统默认大小是1M)，来延迟StackOverFlowError
```

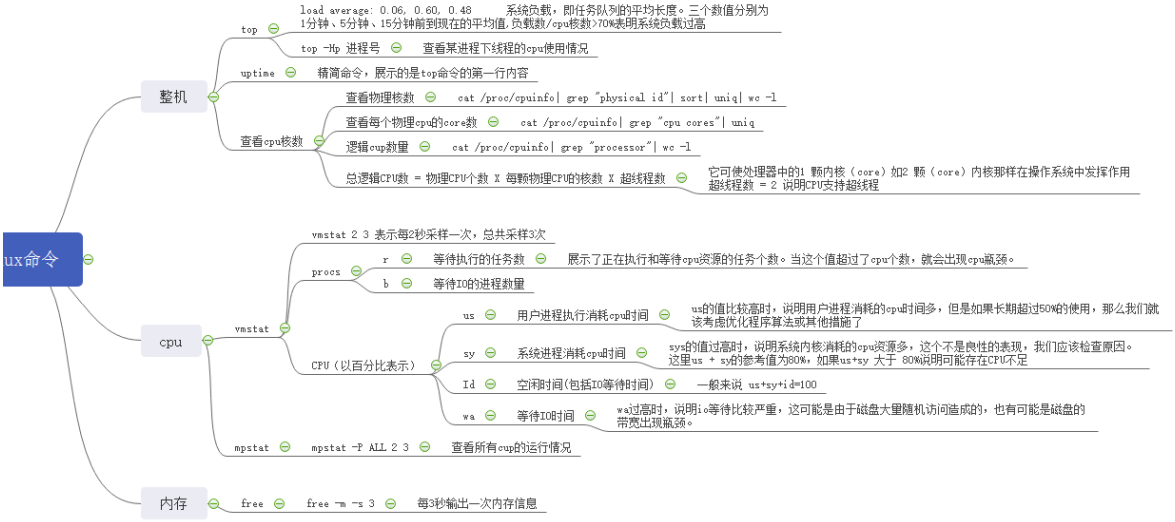
```
* 查看java进程命令
* (ps -ef|grep java)  &  (ls -l)  ->  jps -l
*
* jinfo -flags 线程号
*
* jinfo -flag 参数名称 线程号
* 参数分为三种：
* 标准参数   -version -help
* X参数      -Xint(解释执行\interpreted) -Xcomp(编译执行) -Xmixed(混合模式)
* XX参数     分为boolean参数和key-value参数
*              boolean参数形式：    -XX:[+-]<name>     其中“+”表示开启参数，“-”表示禁用参
数   -XX+UseConcMarkSweepGc
*              key-value参数形式：   -XX:<key>=<value>   -XX:MaxGcPauseMillis=500
* 常用参数名称：
```

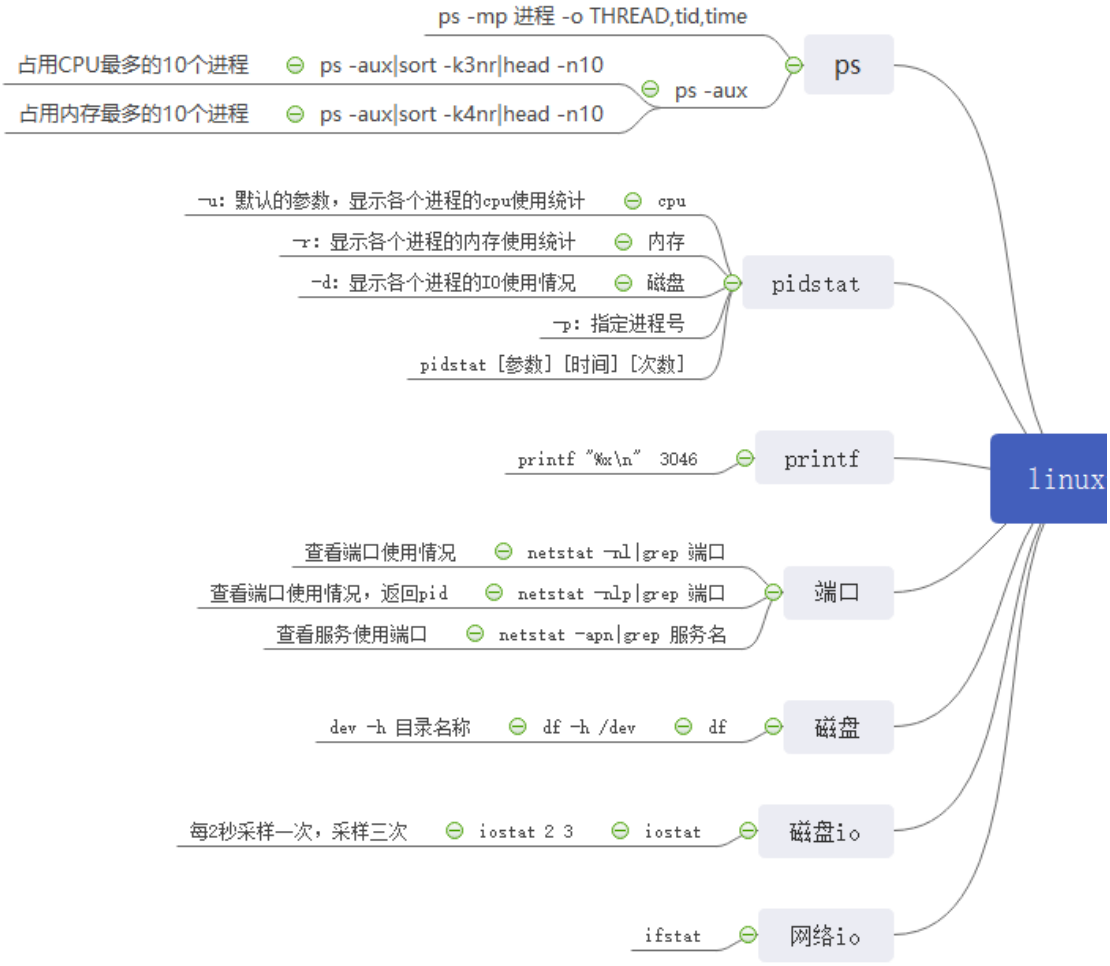
```
*
*    -Xms   初始化heap大小，默认为内存的1/64    相当于 -XX:InitialHeapSize
*    -Xmx   最大heap大小，默认为内存的1/4       相当于 -XX:MaxHeapSize          oom堆空间溢出
*    -Xss   stack空间大小,默认1024k             相当于 -XX:ThreadStackSize    如果超出会导致stackOverFlowError  oom栈空间溢出
*    -Xmn   设置年轻代大小                      相当于 -XX:MaxNewSize
*    -XX:MaxPermSize    -XX:MetaspaceSize     JDK1.7方法区大小    JDK1.8元空间大小
*    -XX:MaxTenuringThreshold=15               存活多少次进入老年代
*    -XX:NewRatio=3                            设置年轻代和年老代的比值   默认是年轻代:年老代=1:3
*    -XX:SurvivorRatio=8                       设置eden区与survivor区的比值      默认是eden:from:to=8:1:1
*
*
*
*
* 查看JVM参数另一种方法
* java -XX:+PrintFlagsInitial                       本机的初始化参数
* java -XX:+PrintFlagsFinal -version                查看JVM
* java -XX:+PrintCommandLineFlags -version
*
*
*
* 查看GC情况
* -XX:PrintGCDetails
*
*
* javac Test.java    生成Test.class文件
*
* javap -c Test.class 查看源码
* javap -verbose Test.class 查看详细源码
*
*
*
*
* .class文件组成包含7个部分
* 1 魔数与class文件版本
* 2 常量池（字面量和符号引用）
*       2.1 字面量：文本字符串、申明为final类型的常量值
*       2.2 符号引用：1类或接口的全限定名 2方法的名称和描述符 3字段的名称和描述符
* 3 访问标志
* 4 类索引、父类索引、接口索引
* 5 字段表
* 6 方法表
* 7 属性表
```

```
* volatile关键字，轻量级的线程同步工具，
* jmm(java memory model) java内存模型(可以画出内存模型)，规定需要满足三个条件
* 1 可见性
* 2 原子性
* 3 有序性
*
*
* volatile满足两种，可见性和有序性，但是不满足原子性
* volatile之所以能满足这两种特性底层是通过memory barrier实现的。
* memory barrier禁止重排序和强制读取主存数据
*
*
* 内存屏障和读写屏障
```

```
* JVM指令总结
* 1 JDK自带
*      所有命令都支持 -help
*      jps -m(main)l(location)v(virtual)
*      jinfo [option] pid
*          option如下：
*          -flag <name>         to print the value of the named VM flag
*          -flag [+|-]<name>    to enable or disable the named VM flag
*          -flag <name>=<value> to set the named VM flag to the given value
*          -flags               to print VM flags
*          -sysprops            to print Java system properties
*          <no option>          to print both of the above
*          -h | -help           to print this help message
*      jstat [option] pid [interval] [count] （实时查看虚拟机状态，可以查看各个区域大
小）
*          option如下：
*          -class(加载或者卸载了多少个类，总大小是多少)
*          -gc（GC信息，每个区域容量大小是多少，使用了多少）
*     jstack [-l|-m] pid -l(lock) -m(mixed包含JVM栈和本地方法栈) 查看这个进程下所有的线
程信息
*          jstack 10 > /logs/catalina_out/jstack.dump可以导出到文件
*          top 查看占用CPU最高的进程
*          top -Hp pid查看pid进程中线程占用CPU的情况
*          通过printf %x 172把10进制转换成16进制，可以查找nid=0x（xxx就是转换的结果）的线
程信息
*     jmap [option] pid （-heap实时查看虚拟机各个区域大小，可以查看所有加载的类信息-
histo(-verbose:class\-XX:+TraceClassLoading)）
*          option如下:
*          -heap                查看各区域大小
*          -histo[:live]        查看所有对象
*          -clstats             查看class loader的统计信息
*          -finalizerinfo       打印finalization队列的对象信息
*          -dump:<dump-options> to dump java heap in hprof binary format
*                           dump-options:
*                             live         dump only live objects; if not
specified,
*                                          all objects in the heap are dumped.
*                             format=b     binary format
*                             file=<file>  dump heap to <file>
*
*  2 标准参数
```

```
*      -version -verbose:[gc|class](可以打印GC信息和类加载信息 jmap -histo:live  -
XX:+TraceClassLoading)
*
*  3 X参数
*      -Xloggc:fileName -Xms -Xmx -Xmn -Xss
*  4 XX参数
*      参数查看相关：
*          -XX:+PrintFlagsInitial -XX:+PrintFlagsFinal -
XX:+PrintCommandLineFlags
*      内存相关：
*          -Xms -Xmx -Xmn -Xss -XX:NewRatio -XX:SurvivorRatio -XX:PermSize
*          -XX:MateSpaceSize -XX:MaxPermSize -XX:MaxMetaSpaceSize -
XX:MaxDirectMemorySize(默认与-Xmx相同)
*      GC相关：
*          GC日志：
*              -XX:+PrintGCDetails -XX:+PrintGCDateStamps -
Xloggc:/home/GCEASY/gc.log
*              -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -
XX:GCLogFileSize=20M
*              -XX:+PrintGCApplicationConcurrentTime -
XX:+PrintGCApplicationStoppedTime
*              -XX:+PrintTenuringDistribution -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath
*          垃圾收集器相关:
*              serial
*                  -XX:+UseSerialGC -XX:UseSerialOldGC
*              parallel
*                  -XX:+UseParallelGC -XX:+UseParalelOldGC -XX:ParallelGCThreads
*                  -XX:MaxGCPauseMills -XX:GCTimeRatio -
XX:+UseAdaptiveSizePolicy
*              par new
*                  -XX:+UseParNewGC -XX:ParallelGCThreads
*              cms
*                  -XX:UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction -
XX:+UseCMSInitiatingOccupancyOnly
*                  -XX:+UseCMSCompactAtFullCollection -
XX:CMSFullGCsBeforeCompaction -XX:ConcGCThreads
*              g1
*                  -XX:UseG1GC -XX:G1HeapRegionSize -XX:MaxGCPauseMillis -
XX:InitiatingHeapOccupancyPercent
*                  -XX:ParallelGCThreads -XX:ConcGCThreads -
XX:G1MixedGCCountTarget -XX:G1HeapWastePercent
*                  -XX:G1OldCSetRegionThresholdPercent -
XX:G1MixedGCLiveThresholdPercent
*          其它:
*              -XX:+UseTLAB -XX:MaxTenuringThreshold -XX:PretenureSizeThreshold -
XX:+DoEscapeAnalysis
*              -XX:+EliminateAllocations -XX:+HandlePromotionFailure -
XX:+HeapDumpOnOutOfMemoryError
*              -XX:+TraceClassLoading
```





5 OMM与内存调优案例

问题一：FASTJSON使用不当导致内存溢出问题

场景：接口对外输出数据时候，输出JSON格式数据，并且要求时间格式是YYYY-MM-DD，但是默认时间的格式返回的是毫秒数

错误写法：

SerializeConfig config = new SerializeConfig();

config.put(Date.class, new SimpleDateFormatSerializer("yyyy-MM-dd HH:mm:ss"));

String res = JSON.toJSONString(srcRes, config);

导致的问题:jvm报错OOM:metaspce

分析思路：元空间溢出，基本可以锁定是加载了过多的类，但是项目已经运行了一段时间，应该是某个功能动态生成了很多类，所以在启动命令里面加了-verbose:class(此处不能使用jinfo动态增加-XX:+TraceClassLoading，这个值不能动态改变)，程序启动之后在catalina.out文件中发现生成了大量的com.alibaba.fastjson.serializer.ASMSerializer,该类是利用asm技术生成的，每次new就生成一个，最终导致内存溢出。解决办法：单例一个对象

问题二：接口访问超时问题

同事反馈接口访问超时，要到具体信息后发现，不是一个接口超时，很多接口超时，基本判断是应用程序问题（服务器问题运维会报警），通过uav查看接口请求时间的服务器运行情况，发现基本与full gc重合，full gc发生6s左右，接口超时时间一般为3s左右，查看gc情况发现,minor gc频繁发生，一分钟25次，cms gc 1个小时一次（别的服务10多个小时一次），gc前老年代内存占用1.8G左右（服务器配置，2c4g，内存设置3g），gc后300M左右，gc日志中有new threshold1(max 6)（思考为什么）,基本判定是过早晋升导致的问题，解决方案，调整新生代大小，规则，老年代大小为gc后活跃对象的3倍，剩余的给新生代，所以新生代2g，老年代1g，调整后，minor gc一分钟10次，cms gc 10多个小时一次，超时情况发生频率减少。

问题三：服务不可用问题

朋友是在小公司，没有系统监控，线上服务经常不可用，问怎么排查。

1 top命令（查看cpu使用情况、内存使用情况）

2 查看cpu 使用情况top -Hp pid 转16进制 导出栈信息 观察是否有死锁 线程等待等情况

3 内存使用情况free -h 如果jvm内存占用使用过大，可以dump下内存文件，使用mat,jvisiualvm进行分析

4 磁盘使用情况df -h du-h

5 iostat

6 iftstat

最终定位发现，他们线上服务器cpu最近飙升100%

解决方案：

线上4台服务器，重启三台解决问题，保留一台，分析问题

1 jps查看进程

2 top查看cpu使用情况

3 使用top -Hp pid查看线程使用cpu情况（查到的线程id转成16晋制）

4 下载几个线程的栈日志 jstack tid > 1.txt

5 dump堆数据

6 使用mat(visualvm)查看堆信息

7 发现有一个BouncyCastleProvider类占用了1.5G左右内存（线上服务器4G内存）

8 原因RSA加密算法是从网上copy过来的，写法有问题，每次加密的时候直接new BouncyCastleProvider()了一个，其实应该使用单例只用一个就行