1.jvm发展史

1.1sun Classic/Exact VM

1. 世界上第一款商用java虚拟机
2. sun公司发布jdk1.0 jdk中的虚拟机就是sun Classic VM
3. sun解决sun Classic VM 执行效率慢的问题, 在1.2 的时候在Solaris平台发布一款Exact VM 的虚拟机,他已经具备了高性能虚拟机的雏形,两级即时编译器,编译器与解释器混合工作模式,使用**准确式内存管理**

缺点:

1. 每行代码都需要编译,导致执行效率相比 C/C++慢

1.2Sun HotSpot VM

1. 最初是由Longview Technologies 一家小公司研发,后被sun收购
2. javaOne大会,开源jdk,在GPL 协议下公开源码,并在基础上建立OpenJDK ,HotSpot VM 成为Sun jdk 和 Open jdk两个极度接近的jdk项目共同虚拟机
3. 2018 2019 oracle 收购了BEA SUN oracle 拥有两款优秀的java虚拟机, JRockit VM 和 HotSpot VM, oracle 宣布在java8 整合两款虚拟机的优势

1.3Sun Mobile-Embeded VM/Meta-Circular

1. **KVM** 简单,高移植性,但是运行慢,在Anddroid IOS 中得到广泛运用
2. **CDC/CLDC HotSpot Impementation** 他希望在手机,电子书,PDA建立统一java编程接口,CDC/CLDC式整个javame重要支柱,
3. **Squawk VM** 运行于Sun Spot(一种手持wifi设备),也曾用于java card是一个java比重比较高的vm,类加载器,字节码验证器,垃圾收集器,编译器,解释器都是java , 仅仅用c写i/o
4. **javaInjava** 试图使用java语言来实现java语言本身运行环境,"元循环",这种东西速度可想而知
5. **Maxine VM** 几乎全部用java实现,只用启动jvm的加载器用c写,有先进的jit编译器和垃圾收集器,没有解释器,执行效率接近hotspot VM

1.4 BEA JRockit/IBM J9 vm

1. BEA将其发展为一款专门为服务器硬件和服务器端应用场景的高度优化虚拟机,
2. jrockit 的垃圾收集器和MissionControl服务套件等部分,在众多java虚拟机中一直处于领先
3. **j9 vm**最初是由IBM Ottawa 实验室一个命名为SmallTalk的虚拟机拓展而来,虚拟机最初有一个bug是由8k值造成,
4. IBM J9 VM市场定位和HotStop比较接近,他是一款从服务端到嵌入式桌面应用都涉及的vm

1.5 Azul VM/BEA Liquid Vm

1. "高性能java虚拟机" 一般至hotspot jrockit j9 ,但其实 Azul Vm /BEA Liquid Vm这类特定硬件平台专有的虚拟机才是高性能武器
2. Azul VM 是Azul Systems公司在hotspot基础上做出大量的改进,运行在Azul Systems公司专有硬件Vega系统上的java虚拟机,每个Azul Vm 至少管理是个CUP和数百个GB内存资源,系统巨大范围可控gc时间垃圾垃圾收集器

3. Liquid VM是现在Jrockit VE 是由BEA 公司开发,不需要操作系统支持,或者说自身实现了一个专用的操作系统必要功能,文件系统,网络,由虚拟机直接调用操作硬件可以获得很多好处,比如线程调度, 不需要内核态/用户态的切换,大大提高硬件效率,

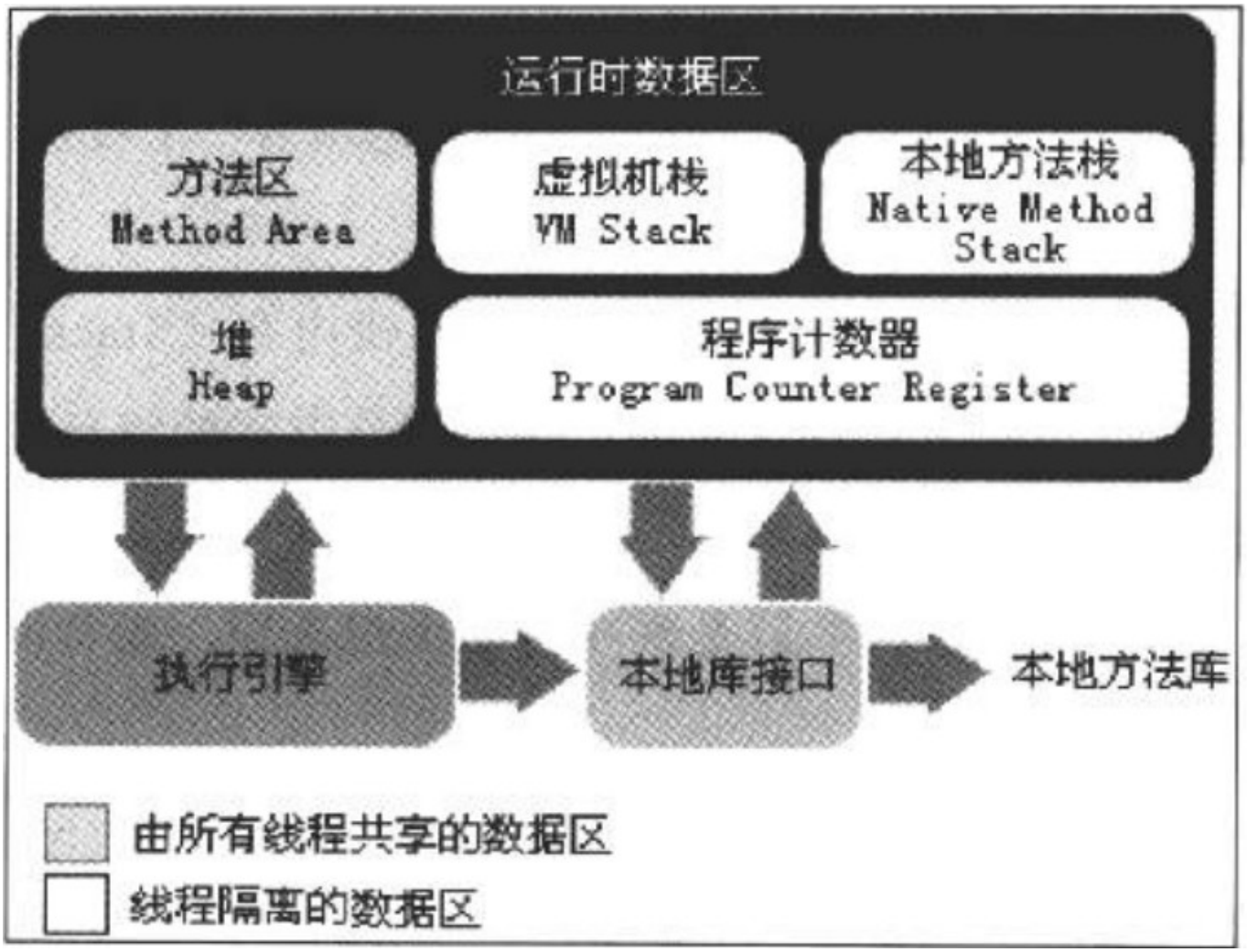
1.6Apache Harmony /Google Android Dalvik VM

1. Harmony Vm Dalvik Vm 只能称之为虚拟机,不能称之为java虚拟机,对近几年的java产生非常大的影响和挑战

1.7Microsoft JVM其他

1. 微软在ie3中支持 Java Applets 基础,而开发自己的虚拟机,这款虚拟机实在window性能最好的虚拟机,但是sun控告Microsoft 最后赔偿sun公司10亿美元,微软退出java研发
2. 别的jvm不说

2.运行时数据区域



2.1程序计数器

1. 一块小的内存,当前线程执行字节码的行号指示器,字节码解析器就是通过改变计数器来取下一条要执行的字节码指令,分支,循环,跳转,异常处理,线程恢复,
2. 每个线程拥有一个程序计数器互不影响

3. 如果线程正在执行一个方法,这个计数器记录正在执行虚拟机字节码指令的地址
4. 如果执行native方法,计数器为空(undefind)此区域没有outofmemoryError

2.2Java 虚拟机栈

1. 和程序计数器一样,线程私有,他的生命周期和线程相同,
2. 每个方法执行的同时会创建栈帧,用于存储局部变量表,操作数栈,动态连接,方法出口,
3. **局部变量表**存放编译器可知的各种基本类型 和引用类型
4. java虚拟机规范中规定两种异常状况 1,StackOverflowError OutOfMemoryError

2.3本地方法栈

1. 本地方法栈和java虚拟机方法栈,类似,但是java虚拟机方法栈执行的java方法,但是本地方法栈执行的是native方法,
2. 和java虚拟机栈一样,会出现两种异常状况 StackOverflowError OutOfMemoryError

2.4java堆

1. java heap是jvm管理的内存中最大的一块,是被所有线程共享 的,
2. jvm 的gc主要发生在这里
3. jvm规定,java heap处于物理上不连续的内存空间中,但是只要逻辑上是连续的,
4. 如果堆没有内存完成实例分配,堆无法拓展,排除outofmemoryerror异常

2.5方法区

1. 线程共享,存放虚拟机加载的类的信息,常量,静态变量,编译后的代码等数据,
2. hotspot上很多人叫永久代,jdk1.7hotspot把原本放在永久代中的字符串常量池移除
3. java8废除永久代,为了让hotspot 和jrockit结合,现实中永久代容易outofmemoryError

2.6元空间(Metaspace)

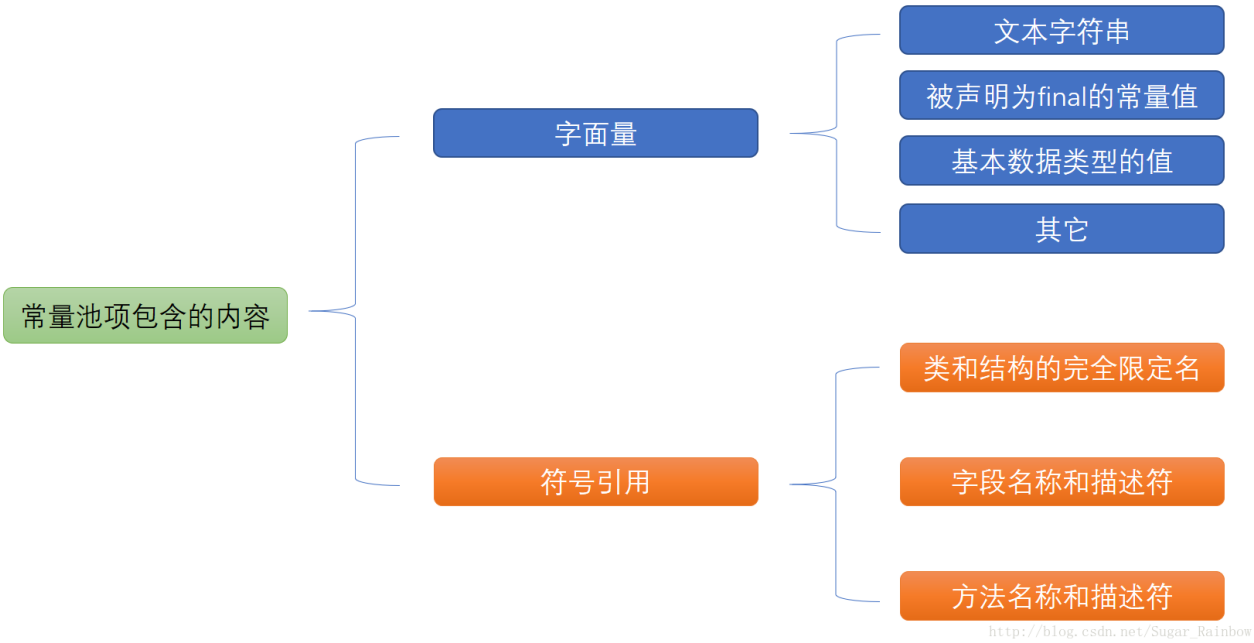
1. 元空间本质和永久代类似,
2. **最大的区别在于,元空间不在jvm中,使用本地内存**理论上取决于32/64系统可虚拟内存大小可以配置

2.7运行时常量池

1. 运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池信息（用于存放编译期生成的各种字面量和符号引用）

   既然运行时常量池时方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

   JDK1.7及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。



——图片来源：https://blog.csdn.net/wangbiao007/article/details/78545189

推荐阅读：

- 《Java 中几种常量池的区分》： [https://blog.csdn.net/qq_26222859/article/details/73135660](

2.8直接内存

不解释

3.hotspot 对象探秘

3.1对象创建

1. jvm遇到new指令,首先 检测这个指令能否在常量池定位到一个类的引用,并检测这个符号是否被加载,解析初始化过,如果没有必须执行相应的类加载过程
2. 分配内存,对象需要的内存大小在类加载完就知道了,此时虚拟机内部维护一个表,记录那些内存是可用的,那些是不可用的,分配的时候找出可用的分配,这种方式叫做"空闲分配"
3. 在分配内存的时候也是不安全的,因为对象分配内存这是经常发生的, 因此使用cas配上失败重试保证每次分配更新的原子性,
4. 内存分配完之后需要把分配的内存都初始化为0值(不包括对象头)
5. 虚拟机对对象进行必要的设置,例如对象是那个类的实例, 如何才能找到元数据信息,对象的哈希码,gc年龄,这些信息存放在对象头中,根据jvm的运行状态不同是否使用偏向锁,
6. 从jvm角度看对象创建完成,但是还要执行init方法,按照程序员的一项进行初始化

3.2对象内存布局

1. 对象头,实例数据,对齐填充
2. 对象头分为两部分,一部分存放运行时数据,哈希码,gc分代年龄,锁状态,偏向锁id,偏向时间戳等
3. 另一部分存放类型指针,对象指向他的类元数据指针,虚拟机通过指针确定对象是那个类的实例,
4. 实例数据是对象的真正有效数据,存放各种字段的内容,无论是父类继承下来还是子类的,
5. 对齐填充不是必然存在,仅仅占位符作用

3.3对象访问定位

1. java程序员通通过栈上的reference 数据来操作堆上的具体对象,由于reference类型在java虚拟机中规定了一个执行对象的引用并没有如何定位,访问堆对象的位置,所以访问对象方式取决虚拟自己实现,主流两种使用句柄和指针
2. 使用句柄的话,堆中会划分出一部分内存作为句柄池,reference 中存放的就是对象句柄的地址,
3. 如果是用指针访问,java堆对象就要考虑存放类型数据信息,reference中直接存放内存地址
4. 使用句柄优势是,gc 的时候只会改变句柄中实例数据的指针,而reference不受影响
5. 使用指针的优势,速度快,节省一次指针定位时间开销,但是对于对象的频繁访问开销积少成多,
6. hotspot使用两种方式对象访问,从各语言框架句柄还是常见的

4.垃圾回收,内存分配策略

4.1对象已死吗?

1. 引用计数算法

- 判定效率高,微软COM技术,ActionScript3的FlashPlayer python语言,和Squirrel都是用计数算法
- 如果两个对象都有字段,instance赋值令,objA.instance=objeB ,objeB.instance=objeA ,除此之外对象再无任何引用,但是因为双方互相引用,导致引用技术都不为0 ,引用计数算法不会通过gc回收

2.可达性分析算法

- 主流的商用程序语言 java Csharp lisp,都是使用可达性分析算法判断对象是否存活,
- 算法的核心思想通过一系列称为"**GC Roots**"对象作为起点,从这些节点向下搜索,走过的路径叫做**引用链**当一个对象到GC Roots没有任何引用链相连接就证明这个对象是不可用的,(对象不可达)
- 在java语言中,可作为GC Roots的对象包含 **虚拟机栈(栈中的本地变量表)中引用的对象, 方法区中类静态属性引用的对象,本地方法栈JNI** 引用的对象

3.引用类型

- jdk1.2后将引用分为 **强引用(strong reference),软引用(Soft Reference),弱引用(Weak Reference),虚引用(Phantom Reference) **
- obejct obj = new object() 这种引用 强引用, GC永远不会回收被引用的对象
- 软引用是用来描述一些**有用但是非必须的**对象,如果在将要发生内存溢出的情况,这些对象将会被回收掉
- 弱引用也是用来描述非必须对象的,他的强度比软引用更弱一点,当GC 工作的时候不论内存是否充足都会回收掉弱引用的对象 jdk1.2 后提供WeakReference 实现弱引用
- 最弱的引用关系,一个对象是否有虚引用的存在,完全不会堆生存时间造成影响,无法通过虚引用获取对象实例,唯一目的就是能在这个对象被收集的时候收到一个系统通知,jdk1.2 后使用phantomreference 实现虚引用

4.生存还是死亡

- 即使一个对象不可达,也并非非死不可,要宣告死亡至少经历两次标记过程,
- 如果发现此对象和GCRoots没有相关联的引用链,他会被第一次标记筛选,筛选是否覆盖finalize() 方法,如果没有覆盖,或者finalize() 方法被虚拟机调用过,这个对象必须死
- 如果被判为有必要执行finalize() 方法,这个对象会被放入一个叫做F-Queue队列中,并稍后由虚拟机自己建立的低优先级的Finalizer线程去执行他,
- 如果一个对象在finalize() 方法中执行缓慢,发生死循环,可能导致整个gc崩溃

```
package com.jvm;

/**
 * @author ZZQ
 * @Title: java8_demo
 * @Package com.jvm
 * @date 2018/7/14 16:48
 */
public class FinalizeEscapeGC {

    public static FinalizeEscapeGC SAVE_HOOK= null;

    public void isAlive(){
        System.out.println("yes,i am still alive :)");
    }

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("finalize method executed!");
        FinalizeEscapeGC.SAVE_HOOK=this;
    }


    public static void main(String[] args) throws  Exception{
        SAVE_HOOK=new FinalizeEscapeGC();

        // 对象第一次拯救自己
        SAVE_HOOK =null;
        System.gc();

        //因为finalize 方法执行的优先级很低，要暂停0.5 等待他
        Thread.sleep(500);
        if(SAVE_HOOK !=null){
            SAVE_HOOK.isAlive();
        }else {
            System.out.println("no , i am dead :(");
        }

    }
}
```

回收方法区

方法区回收需要满足三个条件

- 该类的所有实例都已经回收
- 该类的ClassLoader 都已经被回收
- 该类对应的java.lang.Class对象没有被任何地方引用,无任何地方可以通过反射访问此对象的方法

这个时候可以回收,是否堆类进行回收,hotspot提供了参数

在大量使用反射、 动态代理、 CGLib等ByteCode框架、 动态生jsp 以及OSGii这类 频繁自定义 ClassLoader 的场景都需要虚拟机具备类卸载的功能， 以保证永久代不会溢出。

4.2垃圾收集算法(新生代使用复制算法,老年代使用mark-sweep/mark-整理)

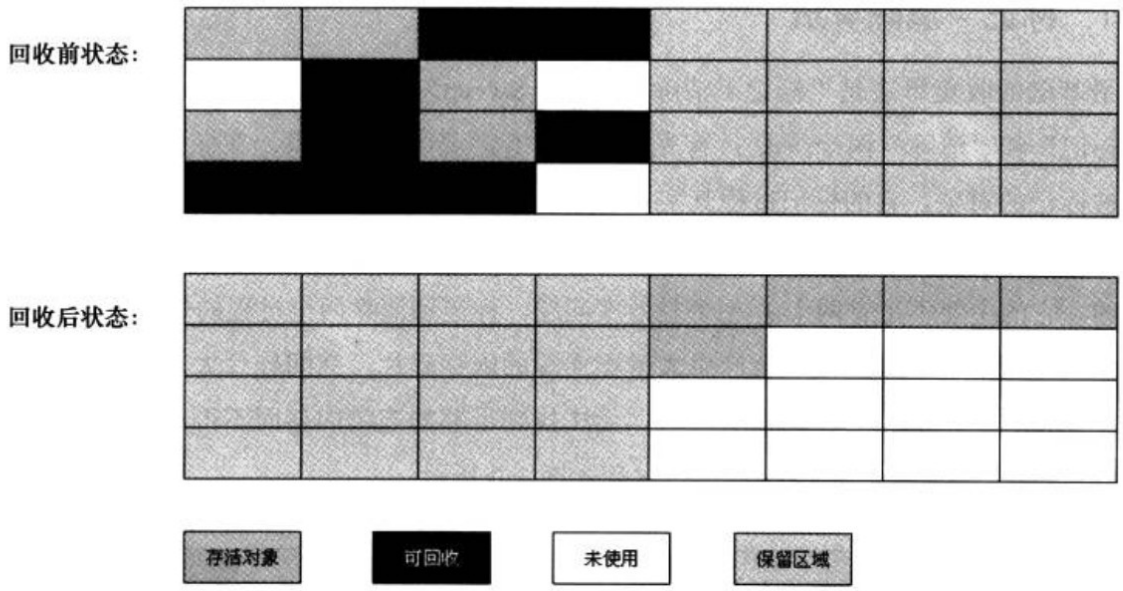
4.2.1标记清除(Mark-Sweep)

对需要回收的对象进行标记,然后统一回收内存

缺点1:标记清除效率都不高

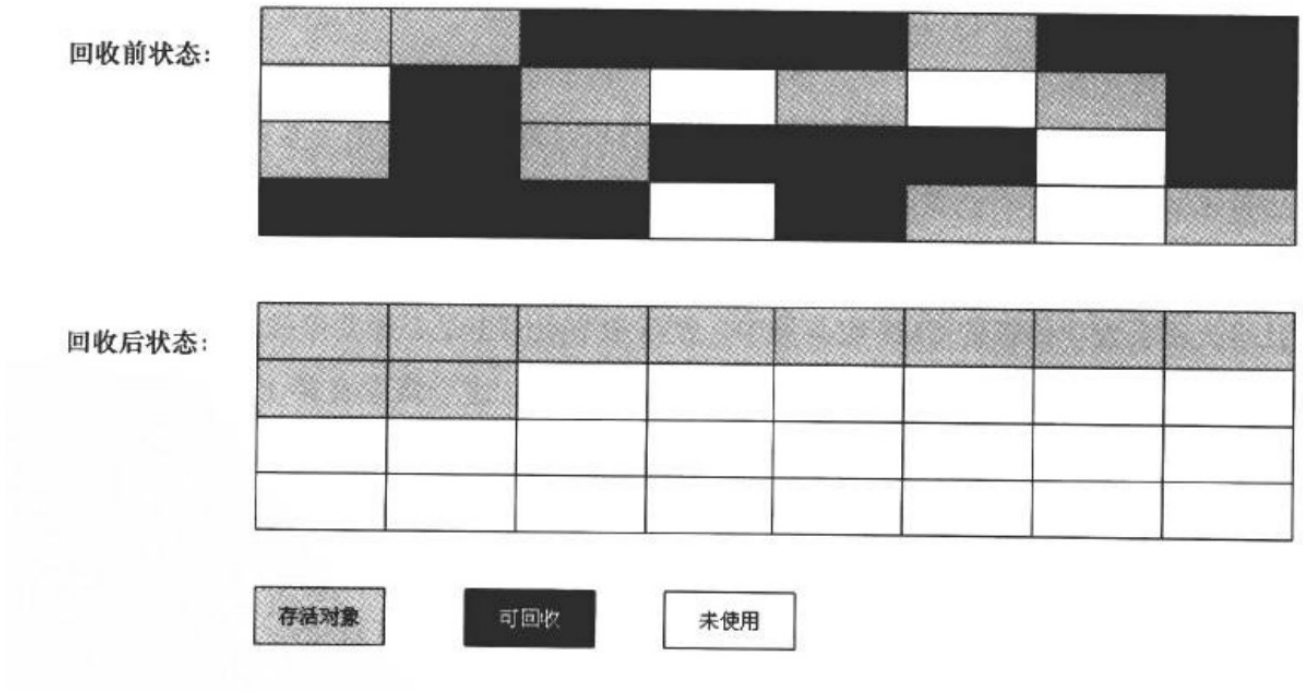
缺点2:标记清除会产生大量不连续的内存碎片,会导致分配大的对象找不到连续的内存,导致体现gc

4.2.2复制算法(Copying)



- 效率高,实现简单
- 内存缩小为原来的一般,这样代价太高
- 商业虚拟机都采用这种收集算法,回收新生代,
- 将内存分为一个较大的Eden空间和两个较小的Survivor空间,回收时,将eden空间和Survivor空间上的存活者复制到另一块Survivor内存上
- HotSpot VM 默认Eden Survivor比例是8:1
- 如果生存下来的对象比较多,大于Survivor ,这些对象将通过**分配担保机制**被送入老年代

4.2.3标记-整理算法



老年代死的比较少,所以应对100%存活这种情况,老年代采用标记整理算法,标记清除,然后整理内存向一段移动

4.2.4分代收集算法(GC)

根据对象存活周期将不同内存划分几块,一般把java堆划分为新生代,和老年代

- 在新生代中' 每次垃圾收集时都发现有大批对象死去' 只有少量存活' 那就选用复制算法只需要付 出少量存活对象的复制成本就可以完成收集。
- 而老年代中因为对象存活率高 没有额外空间 对它进行分配担保' 就必须使用 “标记一清理” 或者 “标记_整理” 算法来进行回收。

4.3HotSpot的算法实现

4.4垃圾收集器

4.4.1Serial(单线程)(新生代)

- 单线程新生代收集器,垃圾回收的时候会发生"stop the world"
- serial 的单线程相比其他收集齐,做的很好,简单高效,
- 运行在Client模式jvm是一个很好的选择

4.4.2parNew(并行)(多线程)(新生代)

- Serial 多线程版,是运行在Server模式中中新生代
- 是除了Serial 之外在jdk1.8 之前唯一能和CMS收集齐结合的 (CMS是1.5老年代)

4.4.3parallel scavenge(并行)(多线程)(新生代)/'pærəlɛl/ /'skævɪndʒ/

- 新生代收集器
- 和cms收集器的关注点不同,cms 是关注的是gc停顿时间

- parallel scavenge 关注的是达到一个可控制的**吞吐量**(吞吐量=运行用户代码/(运行用户代码+运行gc)) 时间
- **自适应调调节**也是parallel scavenge 和parnew一个重要的区别

4.4.4serial old(单线程)(老年代)(mark-Compact算法)

- 使用**标记-整理算法**
- 主要意义是给client端使用的
- 如果用于server 主要两大用途,jdk5之前和parallel Scavenge 配合使用,作为cms的备用

4.4.5parallel old(并行)(多线程)(老年代)(mark-Compact算法)

- jdk1.6开始提供,因为parallel scavenge 无法和cms配合工作,只能用serial old 和parallel old 作为老年代收集器
- 使用serial old 拖累cms ,还不如用parnew

4.4.6cms(并发)(老年代)(mark-sweep算法)

- cms非常复合b/s架构系统的要求
- **初始标记** 需要stop the world 标记 GC Roots
- **并发标记**需要stop the world 对GC Roots 追踪
- **重新标记**修正并发标记期间用户程序继续运行导致变动的一部分标记记录
- **并发清除**
- 低停顿,并发收集,
- **缺点: **对cpu敏感,面向并发设计的程序都会对cpu敏感,
- **缺点:无法处理浮动垃圾**,只好等待下次gc清理调的垃圾叫做浮动垃圾
- **缺点:**基于mark-sweep算法实现,会导致空间碎片,会导致提前full GC

4.4.7G1(使用Region)(并发)(mark-Compact)

- 是一款面向服务器应用的垃圾收集齐,
- 并发与并行,能充分利用CPU,多核环境下硬件优势,使用多个cpu缩短stop-the-world,通过并发实现gc
- 分代收集
- 标记-整理算法,不会产生空间碎片
- 可预测停顿,这是比cms 最大的优势,这几乎已经实时java(rtsj)垃圾收集器的特征
- 后台维护一个优先列表,先回收价值最大的Region
- G1把内存"化整为零"
- g1把每个region都有一个对应的rememberered set来避免全堆扫描
- 如果追求低停顿,g1是一个非常好的选择,如果追求吞吐量,g1不会带特别好处

4.4.8gc日志

4.5内存分配和回收分配

MinorGC 指的是gc 新生代 , **Major GC** 是指gc老年代 ,Full GC 是指gc整个堆

- 对象主要分配在堆上新生代的Eden区上(也可能经过jit编译后拆散标量类型并间接栈上分配)
- 如果启动本地线程分配缓冲,按线程优先在tlab上分配
- 少数情况分配在老年代

- **大对象直接分配在老年代**最典型就是字符串和数组
- **长期存活对象直接晋级老年代**,虚拟机给每个对象定义一个Age计数器,如果对象在Eden出生并经过一次Minor GC并存活,仍然能够在Suvivor 存放下,将被移动到Suvivor,设置年龄为1,对象在Suvivor区中每次熬过一次MinorGc Age增加1 **,当年龄>15(jvm默认值)**就可以晋级老年代,这个值可以调节
- **动态对象年龄判断**如果Survivor 区的某个年龄段的对象所在区域大于整个Survivor 的一半,那么大于此年龄的对象直接晋级老年代
- **空间分配担保**

5.虚拟机加载机制

计算机只能识别**机械码(二进制)** jvm识别 **字节码(class)**

5.1类加载器的时机

- 加载,验证,准备,初始化,卸载,这五个个过程一定是按照顺序执行的
- 解析过程不一定是按照顺序执行的,在某些时候可以在初始化之后开始**,是为了支持java语言运行时绑定(动态绑定)**
- jvm规范对类进行初始化,严格规定
    - 1.遇到new(**new**关键字) getstatic putstatic(读取和设置一个静态常量的时候) invokestatic(调用一个类的静态方法的时候) 四条字节码执行的时候如果类没有初始化,有限初始化类
    - 使用**java.lang.reflect**包对类进行反射调用的时候
    - 初始化一个类,但此类父类未初始化,先初始化该父类
    - **jvm**启动,指定**main**方法类,优先加载
    - jdk1.7动态语言支持时,如果一个java.lang.invoke.MethodHandle实例最后解析为REF_getStatic\ REF_putStatic\ REF_invokeSwtic方法句柄,并且这些方法对应的句柄都没有初始化,需对其进行初始化操作

1.静态字段,只有定义了这个字段的类才会被初始化

```
//父类
class SuperClass {
    static {
        System.out.println("SupperClass init");
    }
    public static  final  String HelloWorld ="Hello world" ;
    public static  int value = 3 ;
}

//子类
class SubClass extends  SuperClass{
    static {
        System.out.println("SubClass init");
    }
}
```

```
public static void main(String[] args){
    System.out.println(SuperClass.value);    //调用父类
}
```

2.通过数组定义引用类,不会触发类的初始化,会触发一个名为jvm产生的一个object的子类,创建动作有newarray指令完成

```
public static void main(String[] args) {
       SuperClass[] sca =new SuperClass[10];
 }

 class SuperClass {

    static {
        System.out.println("SupperClass init");
    }
    public static  final  String HelloWorld ="Hello world" ;
    public static  int value = 3 ;
}
```

3.常量在编译的时候会直接存入调用类的常量池中,本质上没有引用到定义的类,因此不会触发类的初始化(常量传播优化)

```
 class SuperClass {

    static {
        System.out.println("SupperClass init");
    }

    // 常量在编译阶段进行了常量传播优化，会调入调用类的常量池中，本质上没有引用定义常量的类，因此不会触发类的初始化

    public static  final  String HelloWorld ="Hello world" ;
    public static  int value = 3 ;

}


    public static void main(String[] args) {
         System.out.println( SuperClass.HelloWorld);
    }
}
```

5.2类加载器的过程

5.2.1加载

- 通过类的全限定名,获取此类的二进制字节流(很灵活)开发人员可以自定义类加载器去控制字节流的获取方式
- 将字节流所代表静态存储结构转化为方法区运行时数据结构
- 在内存中生成代表此类的Class对象,作为方法区这个类的各种数据的访问入口
- 数组类直接由**jvm**创建不由类加载器创建,但数组类的原数据类型还是靠类加载器创建的
    - 一个数组类,将在加载该组件类型的类加载器的类名称空间被标识
    - if(组件类型不是引用类型)then(jvm将数组类标记为引导类加载器相关)
    - 数组类的可见性和组件可见性一致,如果组件类型不为引用类型,数组类默认可见性为public
- j**ava.lang.Class 类比较特殊,虽然是对象,但是放在方法区,**这个对象作为程序访问方法区的这类数据的外部接口

5.2.2验证

bytecode不一定要java编译而成,所以,jvm需要在运行class文件的时候,进行验证

- 如果输入的字节流不符合class文件格式,jvm抛出**java.lang.VerfyError**错误

1.文件格式验证

- 是否以魔数开头
- 主,次版本号是否在当前jvm 的处理范围内
- 常量池中是否不被支持的常量类型
- 指向常量池 的索引值中是存在不存在的常量
- 文件本身各部分是否有残缺

.....

2.元数据验证

- 是否存在object除外的父类
- 存在父类是否被final修饰
- 如果父类不是抽象类,是否实现了父类中的方法
- 类中的字符,方法是否冲突

....

3.字节码验证

- 保证跳转指令不会跳转到方法体以外的字节码指令上
- 保证方法体种类型转换有效的,
- 保证任意时刻操作栈的数据类型与字节码指令序列都能配合工作

这里涉及到一个离散数学中的一个问题:"Halting problem"通过程序检验程序逻辑是否准确,无法做到绝对准确

.....

4.符号引用验证

- 通过全限定名,是否可以找到对应的类
- 在指定的类中是否存在符合方法的符号描述和简单名称所描述的方法字段
- 符号应用的类中的字段,方法,是否可以被当前类访问到(修饰符)

....

5.2.3准备

准备是正式为类分配内存,并设置类变量初始化值的阶段,这些变量使用的内存将在**方法区**分配(这些变量是**静态变量**实例变量将随对象实例化在堆内存上)

5.2.4解析

1.类,接口解析

- 如果类引用了一个未解析的类将先解析为解析类,然后解析引用类
- 如果类引用了一个未解析的数组,将区加载数组的组件类型,然后有jvm生成数组类型,然后解析引用类
- 最后检查引用权限,如果引用权限不足,抛出java.lang.illegalAccessError异常

2.字段解析

- 解析该字段所处的类,接口,解析失败,直接抛出异常
- 如果引用字段确实存在符号描述的类中,返回直接引用
- 如果引用字段不存在符号描述的类中,会按照继承顺序,在类或者接口中查找,找到之后,返回直接引用
- 查找失败返回NoSuchFieldError

3.类方法解析**(静态方法)**

- 解析类方法表中的class-index中的索引方法所属类或者接口引用
- 如果在类方法表中发现class_index中索引的c是一个接口,直接抛出异常
- 直接在引用类中查找对应的方法,找到方法返回直接引用
- 在引用类中找不到方法,递归引用类父类,是否存在此方法,如果存在,返回直接引用,
- 在引用类的实现的接口,或者父接口中查找是否有匹配的方法,如果匹配,说明此类为**抽象类**抛出异常

4.接口方法解析

- 解析接口方法表中的class_index项中的索引方法所属类或者接口的符号引用,如果解析成功,下一步操作
- 如果引用类是一个接口,返回直接引用,如果是一个类直接抛出异常
- 在接口类中是否存在描述符相同的方法,不匹配返回异常
- 递归查找付接口,是否存在描述符相同的方法,没有返回异常

5.2.5初始化

类加载的最后一步

- 初始化,是对在准备阶段已经赋默认值的变量,**进行用户赋值操作*

- 或者说是执行的过程,

    - 是编译器自动收集所有变量赋值操作,和static{}中的语句合并产生,编译器收集顺序,按照源文件中的顺序收集

- 和(类构造器)不同他不需要显示调用父类构造器,clinit在jvm确保子类执行clinit之前进行执行,所以说,jvm中最先下执行clinit的是java.lang.Object类
- 所以父类的static{} 优先执行
- 如果一个类中没有static{} 编译器不为其生成()
- jvm保证一个类的() 执行过程中的线程安全;

5.3类加载器

类加载器最初是为了**java Applet**基础开发出来的,虽然java Applet已经死掉,但是类加载器在类层次划分,**osgi,热部署,代码就加密**,大方光彩

5.3.1类与类加载器

- 类加载器用于实现加载动作,但不限于加载阶段
- 类和类的类加载一同确立此类在jvm中的唯一性
- 两个类如果相等**,必须通过同一个类加载器,同一个class文件**

```
public class ClassLoaderTest {

    public static void main(String[] args) throws Exception {

        ClassLoader myLoader = new ClassLoader() {

            @Override
            public Class<?> loadClass(String name) throws ClassNotFoundException {
                try {
                    String fileName = name.substring(name.lastIndexOf(".") + 1) +
".class";

                    InputStream is = getClass().getResourceAsStream(fileName);

                    if (is == null) {
                        return super.loadClass(name);
                    }
                    byte[] b = new byte[is.available()];

                    is.read(b);
                    return defineClass(name, b, 0, b.length);
                } catch (IOException e) {
                    throw new ClassNotFoundException(name);
                }
            }
        };


        Object obj =
```

```
myLoader.loadClass("com.jvm.ClassLoaderDemo.ClassLoaderTest").newInstance();


        System.out.println(obj);
       System.out.println(obj instanceof com.jvm.ClassLoaderDemo.ClassLoaderTest);
    }
}
```

一个是由**jvm**的默认类加载器加载**,**一个是由我们自定义的类加载器加载返回**false**

5.3.2双亲委派模型

java中存在两种不同的类加载器一种 启动类加载器(Bootstrap ClassLoader)这个类加载器用cpp写,一个是java字节写的,都继承与抽象类java.llang.ClassLoader

1.启动类加载器**(bootstrap ClassLoader)**

- 将<JAVA_HOME>\lib目录中的类库加载虚拟机内存中
- 将rt.jar中的一些虚拟机按照文件名识别的类库加载虚拟机内存中
- 将通过**-Xbootclasspath**参数指定的类库加载进jvm内存中
- 如果自定义类加载器 需要加载请求委派bottstrap classloader直接返回null即可

2.拓展类加载器**(Extension ClassLoader)**

- 加载<JAVA_HOME>\bin\ext中,或者java.ext.dirs系统变量所制定路径下的类库

3.引用程序类加载器**(Application ClassLoader)**

- 这个类加载器是ClassLoader 中的getSystemClassLoader() 方法的返回值,
- 负责加载classpath上的类库,如果用户不自定义类加载器,使用默认的类加载器

- 这里的类加载器之间的关系一般不是继承,是以组合的方式复用类加载器的代码
- 工作方式
    - 如果一个类加载器收到了类加载请求,他首先不会自己尝试加载
    - 委派父类进行加载,加载请求最终传递到顶层,
    - 当父类加载器无法完成加载请求,子类**尝试加载**
- 这种方式好处是,比如rt.jar中的java.lang.Object 都是委托给最顶层的bootstarp classloader 加载,这样保证了**class**在**jvm**中的唯一性
- 这种方式保证java程序稳定运行,如果重写一个和rt.jar命名相同的类,可以编译,但是永远不能加载
- 实现双亲委派代码都集中在java.lang.ClassLoader上的loadClass() 方法中

5.3.2破坏双亲委派加载模型

jndi

osgi

6.内存模型与线程

7.线程安全锁优化

8.类文件结构

8.1无关性基石(bytecode 字节码文件)

- java中的各种变量,关键字,运算符号的语义最终都是由多条字节码命令组合而成,
- 字节码命令所提供的语义描述能力肯定比java本身强大
- java语言无法支持的特性不代表字节码不支持

8.2class文件结构

- class文件是一组以**8 个字节**为基础单位的**二进制流**
- 各数据项目**严格**按照顺序排列,中间没有间隔符
- 遇到8位以上空间数据项目,会按照高位在前方式分割

8.2.1魔数与class文件版本

- class文件的头**4个字节**为魔数
- 魔数主要式基于安全性考虑问题,如果通过后缀名很容易被改变
- 紧接着魔术的后**4位是class文件的版本号,**5 6 为次版本号, 78 为主版本号
- java的版本号是从**45 开始**(jdk1.1能支持45~45.65535)(jdk1.7主版本号最大值为51)1.8 为52

8.2.2常量池(javap 分析字节码工具)

- 紧接着主版本后是**常量池**(class文件的资源仓库)入口
- 容量计数是从1 开始不是从0
- 设计者将0项常量空出来是有特殊考虑的,如果后面某些指向常量池的索引值在特定情况下需要表达"不引用任何一个常量池项目"
- 常量池主要放两类常量: **字面量(Literal) 符号引用(Symbolic References)`**
    - **字面量**接近于java 的概念,文本字符串,final修饰的变量
    - 符号引用属于编译原理包含三类常量**,类和接口的全限定名 ,字段的名称和描述,方法的名称描述符**
- jvm加载Class文件的时候进行**动态连接** class文件中**不会保存方法,字段的最终内存布局信息**
- jvm运行时,从常量池获取对应的符号引用,再在类创建时解析,翻译到具体内存地址中
- 常量池中每一项常量都是一个表, jdk1.7 之前有11中不同的表结构,1.7 新增三种,更好的支持动态语言调用
- 14个表有共同特点,表开始的第一位**表示常量属于那种类型**

8.2.3访问标识

- 常量池结束后,紧接着两个字节代表访问标识,**用于识别一些类,接口层次的访问信息**(是类还是接口,是public 还是static 还是final...)

8.2.4类索引,父类索引,接口索引集合

- 类索引,和父类索引,都是一个**u2类型的数据**,而接口索引集合是一个**u2类型的数据集合**(只要是在jvm上跑的语言都不支持多继承) class文件通过三项数据确定继承关系
    - 类索引确定类的全限名,
    - 父类索引用确定父类的全限名
- 所有的Class父类索引都不为0
- 被实现的接口,从左到右在接口索引集合中排列
- 接口索引集合,入口第一项,**u2类型的数据为接口计数器**,如果该类没有实现任何接口,则计数器为0,后面所以表不占用任何字节

8.2.5字段表集合

- 用于描述接口,或类中声明的变量,**不包括方法内部声明的局部变量**
- 字段表集合中**不会列出从超类,父类继承下来的字段**,但可能列出原本java代码中不存在的字段(在内部类中为了保持对外部类的访问性,**会自动添加外部类的实例字段**)
- java中不允许字段的重载,但是对字节码来说,两个字段的描述符不一致呢字段重名是合法的

8.2.6方法表集合

- 方法的定义可以通过访问标识,名称索引,描述符表达清楚,代码放在**方法属性表**中的一个**名为Code的属性中**
- 属性表是class文件中最具有拓展的一个数据项目
- 父类的方法如果不被重写**,方法表中的集合就不会出现来自父类的方法信息,但是这个可以由编译器添加进来,比如java中的类构造器**
- 重载一个方法,除了要求与原方法有相同的方法名,**还要有方法的各项参数在常量池中字段符号引用集合**返回值不会包含在特征标签中(下面例子编译器报错)

```
public String hello(){
     return "1";
}
public int hello(){
     return  1;
}
```

- 但是class文件格式中,特征标签的范围更大,只要描述符不是完全一**致,就可以共存一个class文件中**

8.2.7属性表集合

- 在class文件,字段表,方法表都可以携带自己的属性集合

Code属性

- java程序方法体中的代码经过javac编译之后最终编程字节码指令存放在code属性中,Code属性不是必要的,比如接口,抽象类就没有方法体
- code属性是class文件中最重要的属性,java程序中的信息分为两部分, 代码,和元数据
    - 代码: 方法体中的代码
    - 类名,字段,方法

Exceptions属性

- 方法描述时throws后列举的异常

LocalVariableTable属性

- 描述帧栈中局部变量表中的变量,和java源码中定义变量的关系

SourceFile属性

- 记录生成这个Class文件的源码名称

ConstanValue 属性

- 通知jvm自动给静态变量赋值

InnerClasses属性

- 记录内部类和宿主类之间的关联

Deprecated 及Synthetic属性

- 属于标志性的布尔属性,Deprecated 标识某类方法字段,不在推荐使用
- java 的编译器自动添加,添加构造方法

StackMapTable属性

- 会被jvm类加载字节码验证阶段被类检查器使用,

.....

9.jdk性能监视故障处理工具

9.1jps虚拟机进程状况工具

1. jps [options] hostId
2. jps -q 只输出LVMID 胜率主类的名称
3. jps -m 只输出进行启动传递给主类main方法的参数
4. jps -l 输出类的全名, 如果是jar包,输出jar包的路径
5. jps -v 输出jvm进程启动的jvm参数

9.2 jstat虚拟机统计信息监视工具

1. jstat [option vmid [interval [s|ms]] [count]]

9.3 jinfo java信息配置工具

查看具体参数值

1. jinfo [option] pid

9.4jmap java内存映像工具

将堆转化为快照

1. jmap [option] vmid

选　项	作　用
-dump	生成 Java 堆转储快照。格式为：-dump:[live，]format=b，file=<filename>，其中 live 子参数说明是否只 dump 出存活的对象
-finalizerinfo	显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对象。只在 Linux / Solaris 平台下有效
-heap	显示 Java 堆详细信息，如使用哪种回收器、参数配置、分代状况等。只在 Linux / Solaris 平台下有效
-histo	显示堆中对象统计信息，包括类、实例数量、合计容量
-permstat	以 ClassLoader 为统计口径显示永久代内存状态。只在 Linux / Solaris 平台下有效
-F	当虚拟机进程对 -dump 选项没有响应时，可使用这个选项强制生成 dump 快照。只在 Linux / Solaris 平台下有效

9.5jhat 虚拟机堆转快照分析工具

和jmap配合分析快照,jhat内置http服务,生成dump分析结果后可以在浏览器中查看

9.6jstack,java堆栈跟踪工具

生成当前虚拟机当前时刻的线程快照

jstack [option] vmid

1. -F 当正常请求不被相应,强制输出线程堆栈
2. -l 除了堆栈附加关于锁的信息
3. -m 如果调用navtive方法显示 c/c++堆栈

9.7HSDIS JIT生成代码反汇编

这个操作纯属装逼没啥用个人感觉

10.不懂的

- hotsop算法实现,的讲解