第二章：

1.运行时数据区域：

①程序计数器 >选取下一条需要执行的字节码指令，每个线程私有，相互独立，唯一一个jvm没有规定OOM的区域.

②java虚拟机栈 > 存放每个方法创建的一个栈帧，存放局部变量表，操作数栈等信息。线城私有，每个栈帧的大小在编译期就已经确定.运行期不会改变.

③本地方法栈 > native方法

④java堆 > 用于存放对象实例和数组对象，线程共享，内存最大.

⑤方法区 > 存放类相关信息，如类的接口，常量，静态成员，静态方法，类的class对象实例，常量池等信息.线程共享。

2.对象创建分配内存方式：

①如果内存规整，则用指针碰撞法.

②内存不规整，虚拟机会维护一个空闲内存的列表，进行内存分配.

3.堆中对象的结构：

①对象头，存放着对象hash码，该对象线程锁的id，以及该对象对应方法区中class对象的指针.

②实例数据.

③对齐填充.

4.对象实例的访问方式（2种）：

①reference指向java堆中的句柄池中的句柄，句柄中包含了对象实例和对应class对象的指针.

②reference直接指向java堆中的对象，java对象中含有指向对应方法区中的class的指针.

第三章：垃圾收集器和内存分配策略

1.回收的对象有【java堆】和【方法区】中的对象

首先要判断对象是否已死（两种算法）：

①引用计数法：当对象被引用，该对象的计数器就+1，只要计数器为0，则说明该对象不被任何对象引用，说明已死。 【问题】：如果出现循环引用，则没办法判断是否对象已死.如：A->B,B->A,则AB计数器都不为0.

②可达性分析法：通过一系列的GC Root为起点，向下搜索被引用的对象,如果一个对象没有被搜索到，说明其没有被引用，即对象已死.避免了出现循环引用无法判断对象是否已死的问题. 【可以做GC ROOT的有】：本地变量表中引用的对象；方法区中静态属性的引用；常量引用的对象；本地方法栈中的引用

2.引用分为四种：强引用，软引用，弱引用，虚引用

3.finailze()方法的解析..... 待补

4.回收方法区：常量 和 无用的类 ①不在被引用的常量②不再被使用的类：必须满足，类的所有实例已经回收；对应的classload已经回收；class对象没有没引用.

5.垃圾收集算法

5.1 标记-清除算法：【缺点一】效率低下。【缺点二】产生大量不连续的内存碎片.

5.2 复制算法： 把可用空间分配成2等分，每次只使用一份，在进行垃圾收集时，把存活的对象复制到另外一份空的内存中，在全部清除这一份内存.

在java8中，新生代就是这种算法进行收集，不过是分为 E:S0:S1=8:1:1（E:Eden 新生代，S：surrivor 幸存区）的比例分为三块，每次把E和S的存活对象复制到另一个S区中，然后直接清除之前的内存，效率较高. 因为大部分Java对象都是很快就消亡的，所以只有少数幸存，所以如果按照1:1分配，会浪费大量内存，而，按照8:1:1指挥浪费10%。

5.3 标记-整理算法：适用于老年代，因为老年代对象存活率高。

逻辑是：把标记存活的对象移到内存的一边，然后清除其他内存.

5.4 分代收集算法

对于新生代使用复制算法，对于老年代，存活率高的使用标记整理和标记清除算法.

6.安全点：标准是，以程序“是否具有让程序长时间执行的特征”进行选定，如方法调用的位置，循环跳转的位置，异常跳转的位置等...

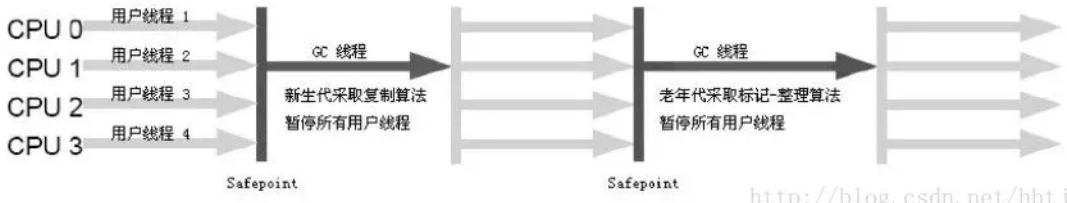
7.垃圾收集器

新生代有：serial，ParNew,Parallel scavenge

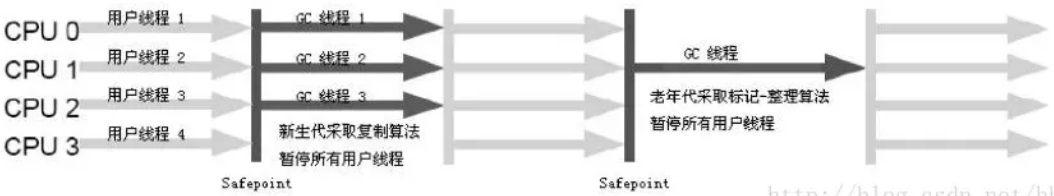
老年代有：【CMS】,serial old , parallel old

【G1】垃圾收集器

7.1 serial收集器 / serial old 收集器 垃圾收集的流程图



7.2 ParNew垃圾收集器



可以看出：采用复制算法，并且并发执行垃圾收集. 主要用于和CMS配合使用.

7.3 Parallel scavenge /parallel old 收集器

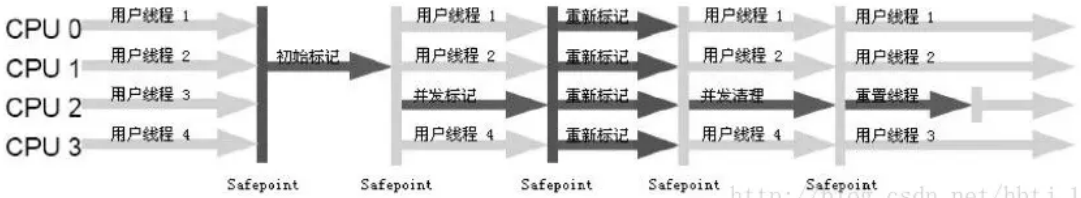
主要就是新生代和老年代都是并发收集

适用于：注重吞吐量以及cpu资源敏感的场合

7.4 【CMS垃圾收集器】老年代

并发标记清除：目的是获取最短回收停顿时间.

> 4个步骤：初始标记，并发标记，重新标记，并发清除



优点：并发标记和并发清除都可以和用户线程一起，所以该收集器停顿的时间比较少.

缺点：i.占用较多cpu资源 ii.无法回收浮动垃圾 iii.基于标记清除算法，收集完成后会产生大量内存碎片.

7.5 【G1收集器】

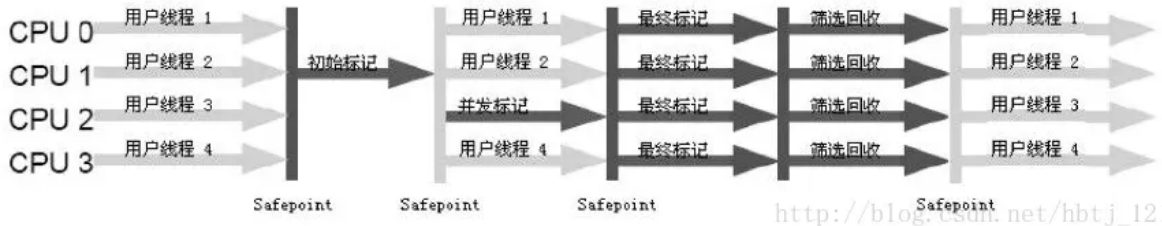
优点：i.充分利用cpu，缩短停顿用户线程的时间.

ii.对存活时间不同的对象使用不同的方式去收集.

iii.G1收集器不会像CMS那样，产生大量的空间碎片.

iv.可以控制停顿时间.

第四



章：jvm性能监控

1.jps:显示系统内所有hotspot虚拟机的进程id，主类等信息

① -q：省略主类名 ② -m 输入main方法的参数 ③-l 输出主类全名，如果是jar，输出jar路径

2.jstat：监视各种运行状态信息的工具

4.jmap：生成堆转储快照，获得dump文件

-dump ： 生成Java堆转储快照

-heap ; 显示Java堆的详细信息，如使用哪种垃圾收集器，参数配置等信息

5.jhat:分析dump文件，一般不用

6.jstack：用于生成虚拟机当前时刻线程的快照

目的是：查看线程相关的信息，可以定位到出现线程长时间停顿的原因.

第七章：类加载机制

1.指虚拟机吧类从class文件或其他二进制流 加载到内存，并对数据进行校验，转换，解析和初始化，最终形成可以被虚拟机直接使用的java类型.

2.五种方式出发类加载：①new 关键字，读取，调用静态字段或者方法. ②java.lang.reflect中的通过反射调用时，出发类加载. ③父类未初始化，先初始化父类 .④虚拟机先初始化执行的主类.⑤。。

3.注意：

①通过子类调用父类的静态字段，不会导致子类的初始化.

②通过数组定义来引用类，不会触发此类的初始化.

③只调用类的常量，不会触发类的初始化.

4.类加载过程：

①加载 : 获取类的二进制字节流，生成方法区的该类的class对象，作为访问该类的入口.

②验证：文件格式校验，语义校验，字节码校验，符号引用的校验

③准备：为类变量分配内存和赋 初始默认值. 在方法区中.

④解析：将常量池中的符号引用转换为直接引用，主要是类的接口，类字段，类方法，接口方法，方法类型，方法句柄，限定符 的符号引用，转换为内存区的直接引用.

⑤初始化：真正的执行类中的java代码，如 给变量赋定义的值，构造函数.

5.双亲委派模型（叫多代委派模型更合适）

①启动类加载器 <--- ②扩展类加载器 <---③应用程序类加载类 <----④自定义 类加载器

①加载JAVA_HOME/lib目录下或 -xbootpath 目录下的类

②加载JAVA_HOME/lib/ext目录下的类

③加载系统变量classpath下的类库

逻辑：如果一个类加载器收到了加载类的请求，他会先让父类尝试加载，如果父类加载失败，再自己尝试加载.

好处是：java的类和他的类加载器有一种层次关系.如，object类是所有类的父类，在rt.jar中，/lib路径下，所以是启动类加载器加载加载，也就是说，只加载一次object类即可，不用重复加载.也更安全.

6.破坏双亲委派模型

①为了兼容jdk1.2之前的版本（那时还没有双亲委派模型），所以破坏

②线程上下文类加载器，父类加载器请求子类加载器加载类.

③热替换，热部署等工作

第八章：

1.栈帧：用于支持虚拟机进行方法调用和执行的数据结构. 存放着局部变量表，操作数栈，返回地址等信息.

2.类加载的解析阶段：包括静态方法，私有方法，构造器方法，父类方法

3.分派　继承，封装，多态中的多态的实现原理

①静态分配：依赖静态类型确定 执行版本的 分派动作. (重载的主要实现)

Human m = new Man(); 中 Human为静态类型，Man为实例类型.

编译期间虚拟机并不知道m的实际类型是什么，而在传递参数时，man的类型为Huamn类型.

②动态分配：在运行期间根据实际类型确定方法执行版本的分派. （重写的主要实现）

Human m = new Man();　m.hello();

m = new Woman(); m.hello();

4.某个父类的方法在子类中没有被重写，那子类的方法表中会有父类该方法的调用地址；如果被重写，则，子类的方法表中保存子类的该方法地址.

方法的调用：先调用子类的方法，如果子类中没有，则会调用父类的方法.

第十二章：java内存模型

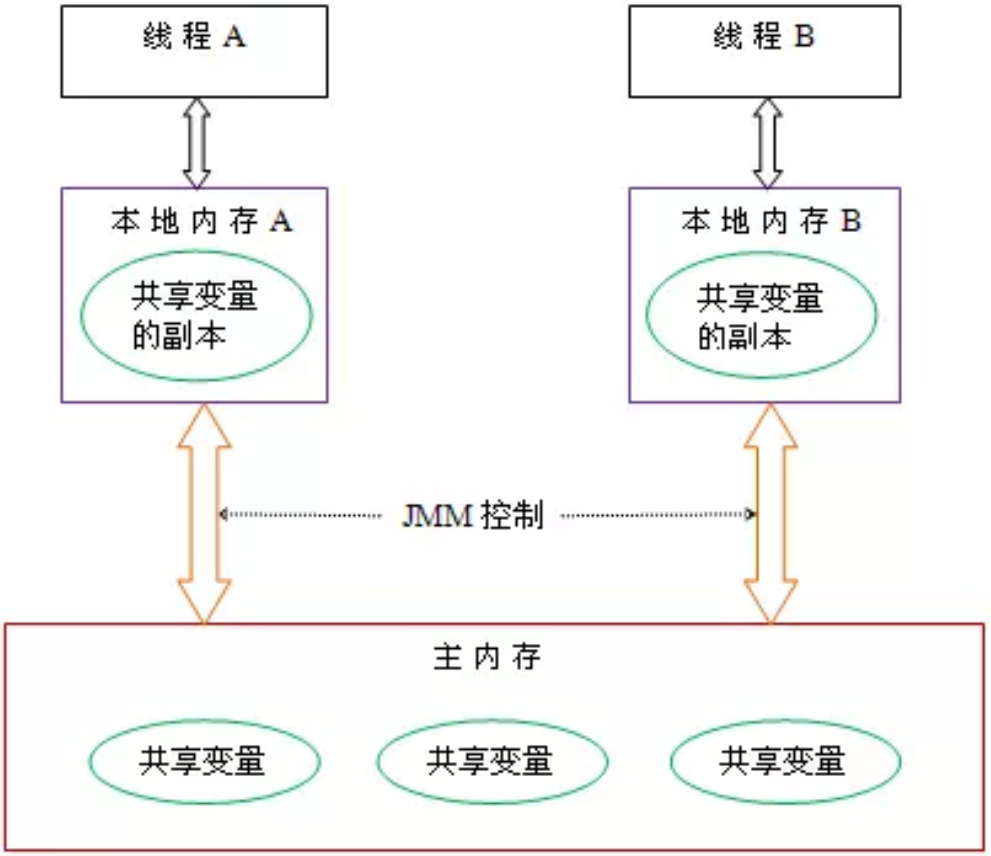
1.缓存一致性：在多处理系统中，每个处理器都有自己的高速缓存，而他们又共享同一个主内存.当多个处理器运行任务涉及代主内存同一内存，可能会导致缓存不一致的情况.

2.内存模型：在特定的操作协议下，对特定的内存或者缓存进行读写访问的过程的抽象.

3.java内存模型：主要目的是定义程序中哥哥变量的访问规整。即在java虚拟机中 把变量存储到内存和从内存中读取的过程的抽象.

4.java内存模型分为：主内存，工作内存，java线程，保存和加载变量的操作。 线程之间的通讯，必需通过主内存.

示意图：



5.volatile关键字：①可见性：读取时，强制从主内存中读取；如果有线程修改，会立马写入到主内存。②禁止指令重排序的优化.