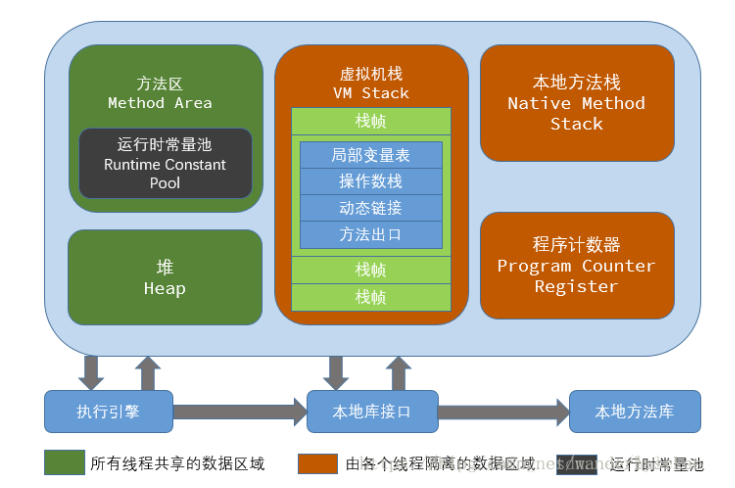
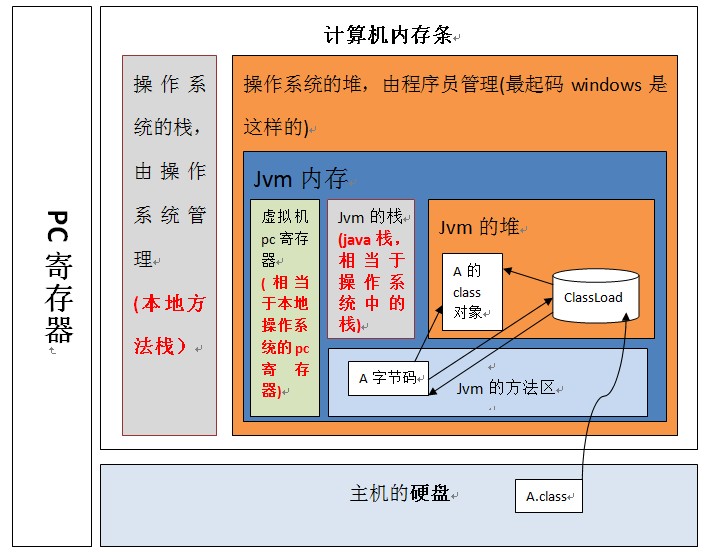
**Jvm内存模型：**



**Jvm的栈就相当于操作系统中的栈，都是自己管理的，计算机的堆是程序员管理的，jvm的堆是gc回收的，还有方法区，jvm的方法区就是计算机的硬盘，这部分的内容属于不断电的永久区。**

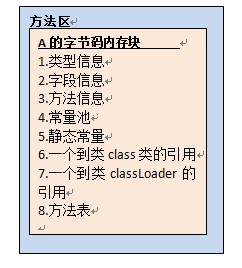
****

**Jvm里还有一个pc寄存器，保存的是下一条将要执行的指令的地址**

**ClassLoader生存在堆中，他会把A.class装载到jvm的方法区（这一步的操作相当于把class文件从操作系统复制到jvm的硬盘中）。这个class文件会被虚拟机解释执行，生成一个A的字节码文件，保存在方法区中，然后再堆内存中生成了一个A字节码的class对象。**

**这个存储在方法区中的A字节码文件中保存有2个引用，一个是指向A的class对象，一个指向加载自己的classLoader**

**这个字节码文件除了保存这2个，还保存了其他什么东西呢？他几乎把类的结构完整的保存到内存中了**

****

**package test;import java.io.Serializable;public final class ClassStruct extends Object implements Serializable {//1.类信息**

**//2.对象字段信息**

**private String name;**

**private int id;**

**//4.常量池**

**public final int CONST\_INT=0;**

**public final String CONST\_STR="CONST\_STR";**

**//5.类变量区**

**public static String static\_str="static\_str";**

**//3.方法信息**

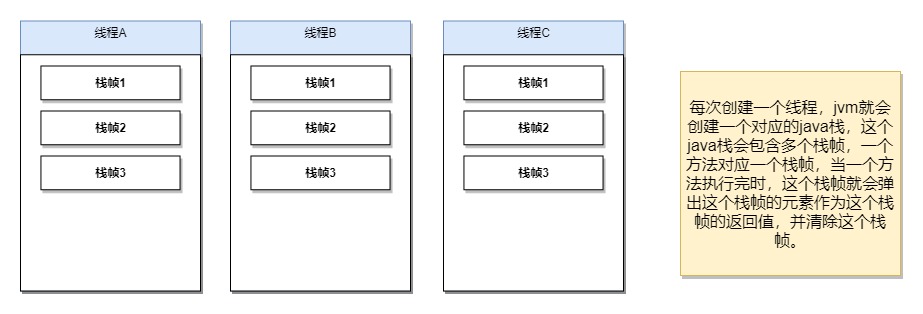
**public static final String getStatic\_str ()throws Exception{**

**return ClassStruct.static\_str;**

**}}**

**A字节码文件中保存了class对象的引用，所以想获得类的信息只能先获得这个对象，所以就得this,getClass().getName**

**每次创建一个线程，jvm就会创建一个对应的java栈，这个java栈会包含多个栈帧，一个方法对应一个栈帧，当一个方法执行完时，这个栈帧就会弹出这个栈帧的元素作为这个栈帧的返回值，并清除这个栈帧。**



**最先被调用的函数的栈帧会最先被压进栈，这个时候只有这个栈帧的本地变量可以被操作栈使用，当这个栈帧需要调用其他方法时，另一个方法的栈帧被压进栈，第二个方法的栈帧变成栈顶，只有他的本地变量能被使用，当第二个函数处理完并返回，他的返回值会变成第一个栈帧的操作栈中的一个操作数**

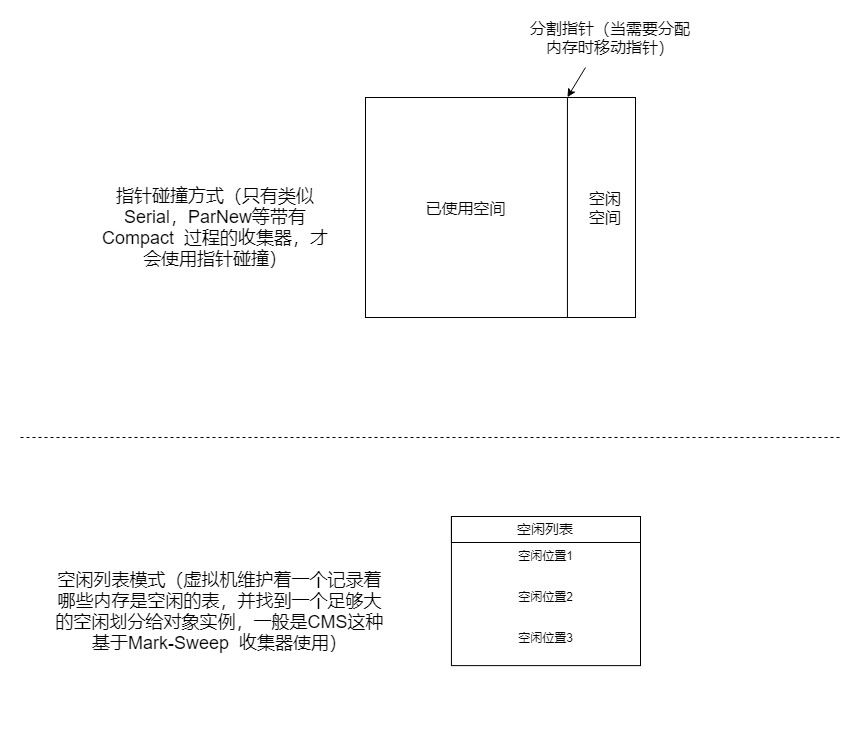
**新生代（Eden，From Survivor，To Survivor）使用Minor GC---> 年老区（Old）使用Full GC=8:1:1:20**

**方法区：储存类结构，属于堆的一部分**

**运行时常量池：是方法区的一部分。存储常量，运行时期常量池也是可以改变大小的，当string调用intern时，会现在常量池中找有没有，如果没有加入常量池并返回，这时可改变常量池大小，常量大小是不可改变的。**

**本地方法栈：为java中的那些native方法服务**

**创建对象时，需要从堆中先划分一份内存，一般有2种方式，一种是已使用内存存放在一边，空闲内存存放在另一边，划分内存时，分隔指针只需要移动一段所需要的距离即可，这种方式称为指针碰撞，另一种方式是，将所有空闲的空间记录在一个表内，划分时在表中寻找一份足够大的内存分配出来。这种方式称为空闲列表**



**分配对象空间不是线程安全的，因为在并发情况下有可能出现同一块内存被同时分配的情况，一般解决办法2种，1是采用CAS配上失败重试保证更新的原子性（要么全部执行，要么全部不执行），2是采用本地线程分配缓冲（TLAB），就是每个线程预先分配一块不重叠的空间，这样只有在预空间分配完了才需要同步锁定，是否使用TLAB由-XX:+/-UseTLAB设定**

**分配空间后会初始化为零值，这表明对象的实例字段可以不被赋予初始值就可以直接使用。**

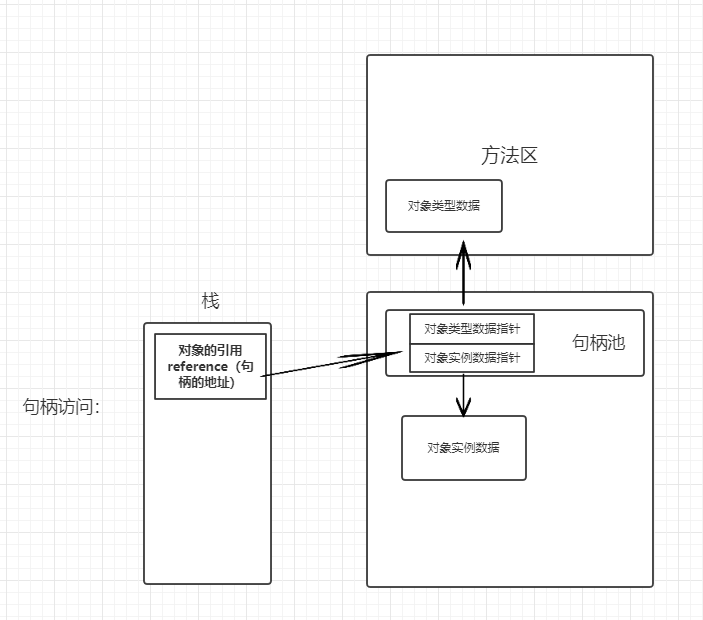
**对象的内存布局：**

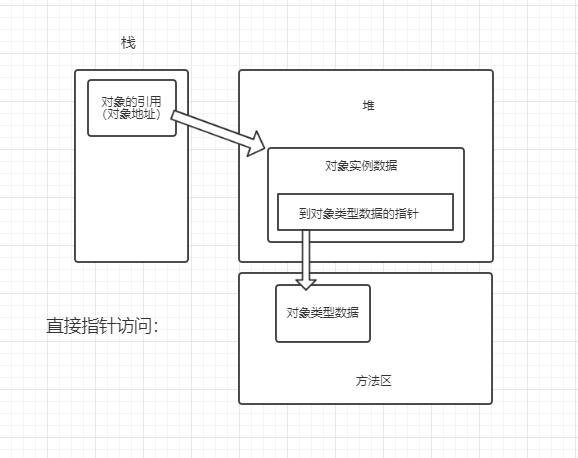
**对象头（Header）：哈希码，GC分代年龄，等等**

**实例数据（Instance Data）：类的字段内容，父类的字段内容，都会被记录，并且相同长度的数据类型会被放在一起，满足这个前提，父类的变量会出现在子类之前**

**对齐填充（Padding）：对象起始地址必须为8字节的整数倍，对象头一般正好是整数倍，当实例数据没有对齐时，使用对齐填充补全**

**对象被分配后，如何使用呢？**





**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。**

**如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常。**

**问题：栈是不是有一个最大值，在栈到达最大值之前会一直进行扩展，当实际内存已满，却还没有到达最大值时抛出OutOfMemoryError，当栈已经到达最大值时，再进行请求生成栈就会抛出StackOverflowError**

**一般来说，虚拟机提供了参数控制java堆和方法区2部分内存的最大值，所以**

**虚拟机栈和本地方法栈的大小=2GB（32bit）-Xmx-MaxPermSize-程序计数器（忽略不计）**

**当出现OutOfMemoryError时，可以通过减少栈容量或者减少最大堆来换取更多线程**

**在jdk1.7版本以上，字符串常量已经从永久代转移到堆中，并且jdk1.8不存在永久代的说法，只有元空间，他并不存在虚拟机中，而是存在本地内存中。**

**垃圾回收机制的原理**

**1：引用计数法：给对象添加一个引用计数器，每当一个地方引用它，计数器值就加1，引用失效就-1任何时刻计数器为0的对象就是不可能再被使用的**

**2：可达性分析法： 通过一系列称为GCRoots的对象作为起始点，向下搜索，当一个对象无法连接到GCRoots时说明该对象不可达**

**可作为GCRoots的对象包括以下：**

**虚拟机栈中引用的对象**

**方法区中类静态属性引用的对象**

**方法区中常量引用的对象**

**本地方法栈中Native方法引用的对象**

**引用的定义：**

**如果reference类型的数据中存储的数值代表的是另一块内存的起始地址，就称这块内存代表着一个引用**

**垃圾回收算法：**

**标记清除算法：先标记后清除，效率低下，而且存在大量内存碎片**

**复制算法：**

**将内存分为大小相等的两块，每次只使用其中一块，当这块使用完了，再将存活着的对象复制到另一块上，然后再把已使用过的内存空间一次清理掉，这样每次都是对整个半区进行内存回收，不需要考虑内存碎片。**

**由于大部分对象都会被回收，所以不需要分成1:1，而是分成Eden和2块survivor，比例是8:1:1每次只使用Eden和其中一块survivor，内存满了之后将存货对象复制到另一块survivor，这样就优化了空间浪费的问题，但是由于不能保证内存存活对象大小小于10%，所以需要向老年代申请分配担保，即超出部分进入老年代**

**标记整理算法：标记并让所有存货对象都向一端移动，再清除掉边界外边的内存，也可达到内存整理的效果**

**分析对象是否存活并进行GC这一操作：**

**是会造成GC停顿的，因为GC必须在一个能确保一致性的快照中进行，一致性是指，在整个分析过程中整个系统的对象引用关系必须停止变化**

**概念：**

**并行：多条垃圾收集线程并行工作，但此时用户线程仍然等待。**

**并发：用户线程和垃圾收集线程同时执行，用户程序继续运行，垃圾收集程序运行在另一个cpu**

**停顿时间：当垃圾收集器进行GC时，用户线程暂停时间**

**吞吐量：描述的是一个可用率，吞吐量=用户代码运行时间/(用户代码运行时间+垃圾回收时间)**

**查看jvm当前参数：java -XX:+PrintFlagsFinal -version**

**垃圾回收器：**

**Serial收集器：单线程运行，并且在进行垃圾回收时会停止所有正常工作的线程**

**ParNew收集器：是Serial的并行多线程版本，只有在多cpu的情况下的性能才比Serial好**

**Parallel Scavenge收集器(PS Scavenge)：类似ParNew收集器，但是注重高可用，不注重停顿时间，可以设置最大停顿时间以及吞吐量大小，还可以设置自适应调整策略，虚拟机会自动调整以提供最合适的停顿时间或者最大吞吐量**

**=========================新生代老年代分割线=============**

**Parallel Old收集器(PS MarkSweep)：是Parallel Scavenge收集器的老年代版本，使用多线程和标记整理算法，与Parallel Scavenge搭配使用追求高可用性**

**CMS（Concurrent Mark Sweep）：分为4个步骤：初始标记（标记GC Roots能关联到的对象），并发标记（GC RootsTracing），重新标记（修正并发标记期间的变化），并发清除（多线程清除），是一个注重低停顿的并发垃圾收集器，但是他的缺点是：占用cpu资源导致应用程序变慢，无法处理浮动垃圾（需要提前预留空间给新产生的垃圾，当空间不够，临时启动Serial Old进行老年代GC），产生大量空间碎片**

**G1收集器：并行并发，分代收集，空间整合，可预测的停顿。G1将整个java堆分成多个大小相等的独立区域（Region），并跟踪各个Region里面垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），维护一个优先列表，每次优先回收价值最大的Region。步骤是：初始标记（标记GC Roots能直接关联到的对象），并发标记（对所有对象进行可达性分析），最终标记（修正并发标记期间的变化），筛选回收（进行价值排序并回收）。**

**Jvm常用参数：**

**-Xmx100m:最大堆内存**

**-Xms100m:初始堆内存，如果设置与-Xmx相同，可以避免gc后jvm重新分配内存**

**-Xmn2g:she**

**-Xss128k：设置每个线程的栈大小，jdk1.5后每个线程栈是1m，这个值越小，就能产生更多线程，但不是无限的，一般在3000-5000，如果这个值过大，会影响系统性能**

**-XX:NewSize=1024m:设置年轻代初始值1024m**

**-XX:MaxNewSize=1024m:设置年轻代最大值为1024m**

**(-Xmn=以上2个相加，而且2个值相等)**

**-XX:PermSize=256m:年老代初始值为256m**

**-XX:MaxPermSize=256m：年老代最大值为256m**

**-XX:NewRatio=4:年轻代与年老代的比值，即年轻代是年老代的4倍**

**-XX:MaxTenuringThreshold=7:一个对象在经过一次MinorGC后还存活，即年龄加1，表示一个对象如果在Survivor复制移动了7次还没有被垃圾回收旧进入年老代，如果一个应用有大量常驻内存对象，可以直接设置为0直接进入年老代提高效率，如果想增加对象在年轻代存活时间或者在年轻代被回收的概念，减少Full GC的概率**

**-XX:PretenureSizeThreshold=3145728(3m)大于这个值的对象直接进入年老代（只有Serial和ParNew认识这个参数）**

**动态年龄判定：**

**不是所有对象只有年龄超过了-XX:MaxTenuringThreshold=15才能进入年老代，当相同年龄的对象超过survivor的一半，所有等于或大于该年龄的对象就可以进入年老代**

**空间分配担保：**

**在MinorGC前，会检查年老代最大可用连续空间是否大于新生代所有对象的总空间，如果大就是保证安全的，如果小，就检查HandlePromotionFailure是否允许失败，如果允许，就检查年老代中最大可用连续空间是否大于历代晋升到年老代对象的平均大小，如果大于，就进行一次有风险的MinorGC，如果不大于，就要进行一个FullGC**

**在jdk1.6后，HandlePromotionFailure不会再影响空间分配担保策略，而是直接判断年老代可用大小是否大于新生代对象总大小或者历代晋升平均大小就会进行MinorGC，否则进行FullGC**