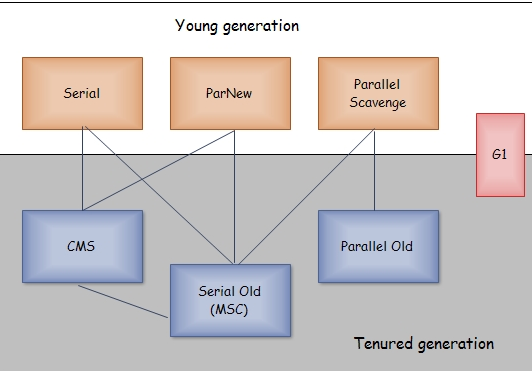
1. 走进java
   1. JDKx
      1. Java程序设计语言、java虚拟机、java API类库
   2. JRE
      1. Java SE API子集、JAVA虚拟机
2. 运行时数据区
   1. 程序计数器
      1. 定义：当前线程所执行字节码的行号指示器
      2. 作用：通过这个行号指示器获取下一条需要执行的字节码指令
      3. 特点：线程私有，各个线程之间的程序计数器互不影响
   2. Java虚拟机栈
      1. 定义：描述Java方法执行时的内存模型
      2. 特点
         1. 一个Java方法开始执行到结束执行的过程对应一个栈帧的入栈到出栈过程
         2. 线程私有
   3. 本地方法栈
      1. 与java虚拟机栈区别：
         1. Java虚拟机栈位java方法服务
         2. 本地方法栈为native方法服务
   4. 堆
      1. 作用：存放对象实例，几乎所有对象的实例都存放于此
      2. 特点
         1. 物理空间可以不连续
         2. 所有线程共享
   5. 方法区
      1. 作用：存放类信息、常量、静态变量等数据
      2. 特点：所有线程共享
   6. 常量池
      1. 作用：存放各种字面量和符号引用
3. 对象的创建
   1. 对象创建过程
      1. 在常量池中查找类的符号引用，并检测其是否被加载过，若未被加载过，执行类的加载过程
      2. 为对象分配内存空间
         1. 指针碰撞法：适用于标识-整理垃圾收集算法
         2. 空闲列表法：适用于标识-清除垃圾收集算法
      3. 执行<init>方法，按照程序员的意愿初始化对象
   2. 对象堆内存空间冲突解决
      1. 问题提出：多线程环境下一块堆内存同时分配为多个线程中的对象内存空间
      2. 解决：
         1. CAS+失败重试方法
         2. 预先分配TLAB（本地线程分配缓冲），只有在TLAB用完并分配新的TLAB时才需要同步
   3. 对象的内存布局
      1. 对象内存分布：对象头+实例数据+对齐填充
         1. 对象头：自身运行时数据+类型指针
            1. 自身运行时数据：hash码、GC分代年龄、锁状态标识等
            2. 类型指针：指向它的类元数据的指针，虚拟机通过这个数据确定对象属于哪一个类
   4. 对象的访问方式
      1. 句柄
         1. 组成：实例数据指针+对象类型数据指针
         2. 优点：对象被移动时只需要改变句柄中的实例数据指针，而reference本身不需要改变
      2. 直接指针
         1. 组成：对象实例数据+对象类型数据指针
         2. 优点：相对于句柄减少了一次指针定位，访问对象速度较快
4. 垃圾收集
   1. 对象状态判断算法
      1. 引用计数法
      2. 可达性分析算法
         1. 虚拟机栈中的引用对象可作为枚举根节点
   2. 引用类型
      1. 强引用
      2. 软引用
         1. 当一个对象只有软引用，且软引用对象在内存空间不足时才会被收集
      3. 弱引用
         1. 当一个对象只有弱引用时，弱引用对象就会被收集
      4. 虚引用
         1. 对于对象的生命周期并影响，主要用来追踪垃圾收集时的活动
      5. 详细介绍

<http://blog.csdn.net/qi183472492/article/details/53668945>

* 1. Finalize
     1. 对象真正“死亡需要经历两次标记”，一是枚举根节点不可达，而是执行finalize方法后对象无“复活”
     2. 一个对象的Finalize方法只会被执行一次
     3. Finalize方法不一定会完整执行（如果执行时间过长会影响后面对象的finalize方法执行）
  2. 垃圾回收算法
     1. 标记清除法
     2. 标记整理法
     3. 复制算法
     4. 分代收集算法
        1. 新生代复制算法，老年代标识-清除或者标识-整理算法
  3. 其他
     1. Stop-the-world
        1. Gc’过程中不应该出现对象引用还在变化的情况，这点不满足的话分析结果的准确性就无法得到保证，所以要暂停所有执行线程
     2. OopMap
        1. 有些应用程序的方法区就数百兆，逐个按顺序扫描肯定很慢，所以出现了OopMap这样的数据结构来保存GcRoot的位置
     3. 安全点
        1. 指的是生成OopMap的位置，也是执行gc的位置（不是每一条指令都会生产OopMap）
     4. Gc中断
        1. 抢先式中断
           1. 先暂停所有线程，然后逐个判断该线程是否在安全点，不是的话重新让线程执行跑到安全点再暂停下来
        2. 主动式中断
           1. 线程执行是轮询安全点标识，到了这个标识自己停下来
     5. 安全区
        1. 可以看作是扩展了的安全点

1. 垃圾收集器
   1. 新生代收集器
      1. Serial
         1. 单线程
      2. ParNew
         1. 多线程
         2. Serial的多线程版本
      3. Parallel
         1. 多线程
         2. 吞吐量优先
      4. 共同特点
         1. 都是基于复制算法实现的
         2. 都需要stop-the-world
   2. 老年代收集器
      1. SerialOld
         1. Serial的老年代版本
         2. 基于标识-整理法
         3. 需要stop-the-world
      2. Parallel Old
         1. Parallel的老年代版本
         2. 基于标记整理法
         3. 需要stop-the-world
      3. CMS收集器
         1. CMS收集器垃圾收集过程
            1. 初始标记：只标记一下GC Root，需要stop-the-world
            2. 并发标记：GC Roots Tracing的过程，不需要stop-the-world
            3. 重新标记：还记得需要Stop-the-world的原因吗？这里的重新标记就是为了修正并发标记过程中因用户线程执行可产生变化的那一部分对象标记。这个过程需要Stop-the-world
            4. 并发清除：不需要stop-the-world
         2. 基于标记-清除算法
         3. 缺点
            1. 影响用户线程的执行效率
            2. 不能处理浮动垃圾
            3. 基于标记-清除算法
      4. G1收集器
         1. 基于标记-整理法
         2. 把堆内存划分为小的region
      5. 收集器关系图



1. 内存分配和回收策略
   1. 堆组成：新生代+老年代
      1. 新生代：Eden+2\*survivor（Eden：survivor = 8：1）
   2. MinorGc和MajorGc
      1. MinorGc指新生代GC，Eden区和其中一个Survivor区存活的到Survivor区去
      2. MajorGc指老年代gc，但通常每次老年代gc伴随着一次新生代gc，所以MajorGC别名FullGC
   3. 对象分配空间策略
      1. 对象优先在Eden分配
      2. 大对象直接进入老年代（原因？）
      3. 长期存活的对象进入老年代（原因？）
   4. 对象年龄的判定
      1. 一般情况：每经过一次MinorGc还存活的对象年龄+1，jvm默认进入老年代的年龄为15，当大于15时对象进入老年代
      2. 动态年龄判定：如果在Survivor空间中相同年龄所有对象的大小总和大于survivor的一半，年龄在这之上的对象直接进入老年代（原因？）
   5. 空间分配担保：老年代剩余空间大于历次平均晋升到老年代空间的内存大小