垃圾回收

1. JVM 运行时内存数据区域

- 方法区域，存放的是类的元数据。线程共享如一个类在JVM的类加载器至于只有一个实例，所有的类共享
    - 存放了每个类的结构信息，包括常量池、字段描述、方法描述
    - GC的非主要工作区域
    - 大量字节码增强技术需要调优方法区
- 堆，存放的对象的信息。创建对象后其他的线程可以使用
    - 堆存放的是对象的实例
    - 是Java虚拟机管理内存中最大的一块
    - GC主要的工作区域，为了高效的GC，会把堆细分更多的子区域
    - 线程共享
- Java虚拟机栈（JVM Stack），存放的是方法调用的相关信息
    - 虚拟机栈描述的是Java方法的执行模型：每个方法执行的时候都会创建一个帧(Frame)栈用于存方局部变量表，操作栈，动态连接，方法出口等信息。 一个方法执行过程就是这个方法对于帧栈的出栈入栈的过程
    - 线程隔离
- 本地方法栈，类型的数据信息
- 程序计数器，存放程序每次执行顺序
- JVM 运行时数据区域-例子

```
public void method1 {
    Object obj = new Object();
}
```

生成了2部分的内存区域：

    - obj 这个引用变量，因为是方法内的变量，放到JVM Stack里面
    - 真正Object class 实例对象，方法Heap 里面
- 上述的new 语句一共消耗了12个bytes, JVM 规定引用占用4个bytes(在JVM Stack),而费控对选哪个是8个bytes (在Heap)
- 方法调用结束后, 对应Stack中的变量马上回收，但是Heap中的对象要等到GC来回收

垃圾回收算法

- 引用计数器算法 (Reference Counting)
    - 给对象添加一个引用计数器，当有一个地方引用了它，计数器加1，当引用失效，计数器减1， 任何时刻计数器为0的对象就是不可能被使用的。
    - 引用计数器算法无法解决对象循环引用的问题
- 根搜索算法 (GC Roots Tracing)
    - 在时机的生产语言中(Java、C#等)，都是使用根搜索算法判定对象是否存活
    - 算法基本思路是通过一系列的称为"GC Roots" 没有任何引用链 (Reference Chain) 相连，则证明此对象是不可用的。
    - 在Java语言中，GC Roots包括
        - 在VM栈（帧中的本地变量）中的引用
        - 方法区中的静态引用
        - JNI(及一般说的Native方法)中的引用

方法区（存放类的元数据）

- JVM 虚拟规范表示可以不要求虚拟机在这区实现GC，这区GC的"性价比" 一般比较低.
- 在堆中，尤其是在新生代，常规进行一次GC一般可以回收70% ~ 95%的空间，而方法区的GC远小于此。

- 当前的商业JVM都有实现方法区的GC，主要回收两部分内容：废弃常量与无用类。
- 主要回收两部分内容： 废弃常量与无用类
    - 该类所有的实例都被GC，也就是JVM中不存在该Class的实例
    - 加载该类的ClassLoader已经被GC
    - 该类对应的java.lang.Class 对象没在任何地方被引用，如不能在任何地方通过反射访问该类的方法。
- 在大量使用反射、动态代理、GCLib等字节码框架、动态生成JSP以及OSGi这类频繁自定义ClassLoader的场景都需要JVM具备类卸载的支持以保证方法区不被溢出。

JVM常见的GC算法

- 标记-清除算法（Mark-Sweep）
    - 算法分为"标记"和"清除"两阶段，首先标记出所有需要回收的对象，然后挥手所有需要回收的对象
    - 缺点
        - 效率问题, 标记和清理两个过程效率都不高
        - 空间问题, 标记清理之后会产生大量不连续的内存碎片，空间碎片太多可能会导致后续使用中无法找到足够的连续内存而提前触发一次的垃圾收集动作。
    - 效率不高，需要扫描所有的对象，堆越大，GC越慢
    - 存在内存碎片问题。GC次数越多，碎片越严重
- 标记-整理算法 (Mark-Compact)
    - 标记过程仍然一样，但后续步骤不是进行直接清理，而是令所有存活的对象一端移动，然后直接清理掉这端边

界意外的内存。
    - 没有内存碎片
    - 对Mark-Sweep耗费更多的事件进行compact
- 复制收集算法 (Copying)
    - 将可用内存划分为2块，每次只使用其中的一块，当半区内存用完了，仅将还存活的独享复制到另外一块上，然后就把原来整块内存空间一次性清理掉
    - 这样使得每次内存对是对真个半区的回收，内存分配时也就不用考虑内存碎片等复杂情况，只要一动堆顶指针，按循序分配就可以了，实现简单，运行效率高。 只是这种算法的代价是将内存缩小为原来的一半。代价高昂。
    - 现在的商业虚拟机中都是采用这种算法来回收新生代
    - 将内存分一块较大的eden空间和2块较少的survivor空间，每次使用eden和其中一块survivor（存活者） 空间，当回收时将eden和survivor还存活的对象一次性拷贝到另外一个块survivor 空间上，然后清理掉eden和用过的survivor
    - Oracle Hotspot 虚拟机默认eden 和 survivor 的大小比例是8:1 也就是每次只有10%的内存是"浪费"的。
    - 复制收集算法在独享存活率高的时候，效率有所下降
    - 如果不想浪费50%的空间，就需要有额外的空间进行分配担保用于应付半区内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。
    - 只需要扫描存活对象，效率更高。
    - 不会产生碎片
    - 需要浪费额外的内存作为复制区

- 复制算法非常适合生命周期比较短的对象，因为每次GC总能回收大部分的对象，复制的开销比较小。
- 根据IBM的专门研究， 98%的java 对象只会存活1个GC周期，对这些对象很适合复制算法。而且不用1:1 划分工作区和复制区的空间。
- 新创建的对象在放在那里?
  - 对象创建是放在eden区域
  - 每次复制的时候将eden区和from survivor区中的数据拷贝到to survivor中。然后清除eden区和from survivor。
  - 如果to survivor 中的区域放不下那么就有一个[分配配担保]的过程来将，存活时间比较久的数据拷贝到Old区

- 分代算法 (Generational)
  - 当前商业虚拟机的垃圾收集都是采用"分代收集"（Generational Collecting）算法，根据独享不同的存活周期将内存划分为几块
  - 一般是把Java堆分作新生代和老年代, 这样就可以根据各个年代的特点采用最适当的收集算法，譬如新生代每次GC都有大批量对象死去，只有烧流量存活， 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。
  - 综合前面几种GC算法的优缺点，针对不同生命周期的对象采用不同的GC算法
    - 新生代采用 Copying

        - 老年代采用 Mark-Sweep or Mark-Compact
    - Hotspot JVM 6中共分为三个代： 年轻代(Young Generation)、年老代（Old Generation）和永久代（Permanent Generation）.
    - 年轻代（Young Generation）
        - 新生成的对象都放在新生代。年轻代用复制算法记性GC（理论上，年轻代对象生命周期非常短，所以适合复制算法）
        - 年青代采用复制算法进行回收，假如当前正在使用的是eden及survivor1区，大部分新生对象都会放在eden区（大对象会直接放在老年代），当eden区内存满了以后，会将存活对象保存到survivor2区，eden进行Minor GC。survivor1区的存活对象根据年龄（默认是15，每经历一轮GC，年龄加1）决定去向，年龄达到阀值，复制到老年代，若年龄未到，复制到survivor2区，survivor1区清空。如果survivor2区内存不足以存放所有的存活对象，则需要以来老年代的担保机制将部分对象复制到老年代。
        - Eden 和2个Survivor的缺省比例是8:1:1，也就是10%的空间将被浪费，可以根据GC log的信息调整大小的比例。

- 老年代（Old Generation）
  - 存放了经过一次或多次GC还存活的对象
  - 一般采用 Mark-Sweep或 Mark-Compact 算法进行GC
  - 有很多垃圾收集器可以选择。每种垃圾收集器可以看做是一个GC算法的具体实现。可以更具具体应用的需求选用合适的垃圾收集器（追求吞吐量？追求最短的响应时间）。
- ~~永久代~~
  - 并不属于堆 (Heap) 但是GC也会涉及到这个区域
  - 存放了每个Class的结构信息， 包括常量池、字段描述、方法描述。与垃圾收集要收集的Java对象关系不大。
- 内存结构

内存分配

1. 堆上分配 大多数情况是在eden上分配，偶尔会直接在old 上分配，细节取决于GC的实现
2. 栈上分配 原子类型的局部变量

内存回收

- GC要做的事情是将那些dead的对象所占用的内存回收掉
  - Hotspot 认为没有引用的对象是dead 的
  - Hotspot 将引用分为四种: Strong(强引用)、Soft(软引用)、Weak(弱引用)、Phantom(虚引用)

        - Strong 即默认通过 Object obj = new Object()这种方式赋值引用
        - Soft、Weak、Phantom 这三种都是继承 Reference
    - 在Full GC时会对Reference类型的引用进行特殊处理
        - Soft: 内存不够时一定会被GC、长期不用也会被GC
        - Weak: 一定会被GC，当被mark为dead，会在ReferenceQueue中通知
        - Phantom: 本来就没有引用，当从jvm heap 中释放会通知

垃圾收集器

- 年轻代
    - Serial、 ParNew、 Parallel Scavenge
- 年老代
    - CMS、Serial Old(MSC)、Parallel Old
- JVM提供了垃圾收集器的实现

GC 的时机

- 在分代模型的基础上，GC从实际上分为两种：Scavenge GC 和 Full GC
- Scavenge GC(Minor GC)
    - 触发时机: 新对象生成时，Eden空间满了。
    - 理论上Eden 区大多数独享会在 Scavenge GC 回收，复制算法的执行效率会很高，Scavenger GC时间比较短。
- Full GC
    - 对整个JVM惊醒整理，包括Young、Old 和 Perm

- 主要的触发时机：1）Old 满了、 2) Perm 满了、3）System.gc()
- 效率很低，尽量减少Full GC

垃圾回收器（Garbage Collector）

- 分代模型: GC的宏观愿景；
- 垃圾回收器: GC的具体实现
- Hotspot JVM 提供多种垃圾回收器，我们需要更具具体应用的需要采用不同的回收器
- 没有万能的垃圾回收器，每种回收器都有它的适用场景

垃圾收集器的"并行"和"并发"

- 并行（Parallel）：指多个收集器的线程同时工作，但是用户线程处于等待状态
- 并发（Concurrent）：指收集器在工作的同时，可以允许用户线程工作
    - 并发不代表解决了GC停顿的问题，在关键的步骤还是要停顿。比如在收集器标记垃圾的时候。 但在清除垃圾的时候，用户线程可以和GC线程并发执行。

Serial 收集器

- 单线程收集器，收集时会暂停所有工作线程（Stop The World, 简称STW），使用复制收集算法，虚拟机运行在Client模式的默认新生代收集器
    - 最早的收集器，单线程进行GC
    - New 和 Old Generation 都可以使用
    - 在新生代, 采用复制算法; 在老年代，采用Mark-Compact算法
    - 因为使用单线程GC，没有多线程切换的额外开销，简单实用。
    - Hotspot Client 模式缺省的收集器
    - Safepoint 安全点

PerNew 收集器

- ParNew 收集器就是Serial 的多线程版本，除了使用多个收集线程外，其余行为包括算法、STW、对象分配规则、回收策略等都与Serial 收集器一模一样
- 对应的这种收集器是虚拟机运行在Server模式的默认新生代收集器，在单CPU的环境下，ParNew 收集器的效果并不会比Serial收集器有更好的效果
    - Serial 收集器的在新生代的多线程版本
    - 使用复制算法（因为针对新生代）
    - 只有在多CPU的环境下，效率才会比Serial收集器高
    - 可以通过-XX:ParallelGCThreads来控制GC线程数的多少。需要结合具CPU的个数
    - Server模式下新生代的缺省收集器。

Parallel Scavenge 收集器

- Parallel Scavenge 收集器也是一个多线程收集器，也是使用复制算法，但它的对象分配规则与收集策略都与ParNew收集器有所不同，它是以吞吐量最大化（ 即GC时间占总运行时间最小）为目标的收集器实现，它允许较长的STW换取总吞吐量最大化。

Serial Old 收集器

- Serial Old收集器是单线程收集器，使用标记-整理算法，是老年代的收集器

Parallel Old 收集器(JDK1.8默认的)

- 老年代版本吞吐量优先的收集器，使用多线程和标记-整理算法，JVM 1.6提供，在此之前，新生代使用了PS收集器的话，老年代除Serial Old外别无选择， 因为PS无法与CMS收集器配合工作。
    - Parallel Scavenge 在老年代的实现
    - 在JVM 1.6 才出现Parallel Old
    - 采用多线程，Mark-Compact 算法
    - 更注重吞吐量

- Parallel Scavenge + Parallel Old = 高吞吐量，但是GC停顿可能不理想

CMS（Concurrent Mark Sweep）收集器

- CMS 是一种以最短停顿时间为目标的收集器，使用CMS并不能达到GC效率最高（总体GC时间最小），但它能尽可能降低GC时服务的停顿时间， CMS收集器使用的是 标记-清除算法
- 特点
  - 最求最短停顿时间，非常适合Web应用
  - 针对老年代，一般结合ParNew使用
  - Concurrent, GC 线程和用户线程并发工作（尽量并发）
  - Mark-Sweep
  - 只有在多CPU环境下才有意义
  - 使用-XX:+UseConcMarkSweepGC启用
- CMS收集器缺点
  - CMS以牺牲CPU资源的代价来减少用户线程的停顿。当CU个数小于4的时候，可能对吞吐量影响非常大
  - CMS在并发清理的过程中，用户线程还在跑，这时候需要预留一部分空间给用户线程。
  - CMS用Mark-Sweep, 会带来碎片问题。碎片过多的时候会容易平凡触发Full GC

GC 垃圾收集器的JVM参数定义

参数	描述
UseSerialGC	虚拟机运行在Client模式下的默认值 Serial Old 的收集器组合进行内存回
UseParNewGC	打开此开关后，使用PerNew + Se
UseConcMarkSweepGC	打开此快关后，使用PerNew + CM 内存回收。Serial Old 收集器作为C
UseParallelGC	失败后的后备收集器使用
UseParalleOldGC	虚拟机运行的Server模式下的默认

	Scavenge + Serial Old (PS Mark 收
SurvivorRatio	新生代中Eden区域与Surivivor 区 Eden: Surivivor = 8:1
PretenureSizeThreshold	直接晋升到老年代的对象大小， 设 象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄。每个对 龄就增加1，当超过这个参数值就
UseAdaptiveSizePolicy	动态调整Java堆中各个区域的大小
HandlePromotionFailure	是否允许分配担保失败，即老年代 个Eden和Survivor区的所有对象都
ParallelGCThreads	设置并行GC时惊醒内存回收的线程
GCTimeRatio	GC时间占总时间的比率，默认值为 用Parallel Scavenge收集器时生效
MaxGCPauseMillis	设置GC的最大停顿时间。仅在使用
CMSInitialingOccupancyFraction	设置CMS收集器在老年代空间被使 68%，仅在使用CMS收集器时生效
UseCMSCompactionAtFullCollection	设置CMS收集器在完成垃圾回收后 仅在使用CMS收集器时生效
CMSFullGCsBeforeCompaction	设置CMS后机器在进行若干次垃圾 仅在使用CMS收集器时生效

Java 内存泄漏的经典原因

- 对象定义在错误的范围（Wrong Scope）
  - 如果 Foo 实例对象的生命较长，会导致临时性内存泄漏。（这里names 变量其实只有临时作用）

```
class Foo {
   private String[] names;
   public void doIt(int length) {
      if (names == null || names.length < length) {
         names = new String[length];
      }
      populate(names);
      print(names);
   }
}
```

  - JVM 喜欢生命周期短的独享，这样做已经足够高效

```
class Foo {
   public void doIt(int length) {
      String[] names = new String[length];
      populate(names);
      print(names);
```

```
    }
}
```

- 异常（Exception）处理不当
    - 错误的做法

```
Conection conn = DriverManager.getConnection(url, name, passwd);
try {
    String sql = "do a query sql";
    PreparendStatement stmt = conn.prepareStatement(sql);
    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
        doSomeStuff();
    }
    rs.close();
    conn.close();
} catch (Exception e) {
    //如果doSomeStuff()抛出异常
    //rs.close和conn.close不会被调用
    //会导致内存泄漏和db连接泄漏
}
```

    - 正确的做法

```
PreparendStatement stmt = null;
ResultSet rs = null;
try {
    String sql = "do a query sql";
    stmt = conn.prepareStatement(sql);
    rs = stmt.executeQuery();
    while (rs.next()) {
        doSomeStuff();
    }
    rs.close();
    conn.close();
} catch (Exception e) {
    //handle exception
} finally {
    //永远用finally去关闭资源，避免资源泄漏
    if (rs != null) {
      rs.close();
    }
    if (stmt != null) {
      stmt.close();
    }
    conn.close();
}
```

集合数据管理不当

- 当我们使用Array-based 的数据结构（ArrayList，HashMap等）时，尽量减少reszie
  - 比如 new ArrayList 时， 尽量估算size，在创建的时候吧size确定
  - 减少 resize 可以避免没有必要的array copying, gc 碎片等问题
- 如果List只需要顺序访问，不需要随机访问（Random Access），用 LinkedList 替代 ArrayList
  - LinkedList本质是链表， 不需要resize，但只适用顺序访问。

打印JDK启动参数信息

```
java -XX:+PrintCommandLineFlags -version

-XX:G1ConcRefinementThreads=4 -XX:GCDrainStackTargetSize=64 -
XX:InitialHeapSize=134217728
-XX:MaxHeapSize=2147483648 -XX:MinHeapSize=6815736 -XX:+PrintCommandLineFlags
-XX:ReservedCodeCacheSize=251658240 -XX:+SegmentedCodeCache -
XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC
java version "13.0.2" 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing)
```

垃圾回收JDK参数

PretenureSizeThreshold: 设置对象超过多大直接在老年代进行分配 （需要在 -XX:+UseSerialGC 配合使用）

-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=5 //可以自动调节对象晋升(Promote)到老年代阈值的GC中，设置该阈值的最大值，该参数的默认值是15，CMS 中默认值为6。G1中默认值为15（在JVM中，该数值是由4个bit来表示，所以最大值为1111， 即15）。

经历了多次GC后，存活的对象会在From Survivor 与 To Survivor 之间存放，而这里面的一个提前将这两个空间有猪狗的大小来存放这些数据，在GC算法中，会 计算每个对象年龄的大小，如果达到某个年龄后发现总大小已经大于了Survivor空间的50%，那么这时就需要调整阈值，不能再继续等到默认的15此GC后才能完成晋升 因为这样会导致Survivor空间不足， 所以需要调整阈值，让这些存活对选哪个尽快完成晋升。

-XX:+PrintTenuringDistribution:输出显示在survivor空间里面有效的对象的岁数情况。

新生代晋升老年代

GC 日志打印

-verbose:gc -Xmx200M -Xmn50M -XX:TargetSurvivorRatio=60 //当Survivor区中的数据大于 60%的时候数据将进入老年代。 -XX:+PrintTenuringDistribution -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:MaxTenuringThreshold=3

总结

- 新生代采用复制算法老年代采用标记清除算法
  - 对象生存周期，新生代复制算法、分代算法老年代的对象，被多次回收存活下来，然后使标记算法
  - 内存碎片的问题，老年代长期存活内存碎片比较多。