链滴

GC 和 GC 调优

作者：noryar

原文链接：https://ld246.com/article/1618880738985

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

GC和GC Tuning

GC的基础知识

1.什么是垃圾

C语言申请内存：malloc free

C++： new delete

c/C++ 手动回收内存

Java: new ?

自动内存回收，编程上简单，系统不容易出错，手动释放内存，容易出两种类型的问题：

1. 忘记回收
2. 多次回收

没有任何引用指向的一个对象或者多个对象（循环引用）

2.如何定位垃圾

1. 引用计数（ReferenceCount）
2. 根可达算法(RootSearching)

3.常见的垃圾回收算法

1. 标记清除(mark sweep) - 位置不连续 产生碎片 效率偏低（两遍扫描）
2. 拷贝算法 (copying) - 没有碎片，浪费空间
3. 标记压缩(mark compact) - 没有碎片，效率偏低（两遍扫描，指针需要调整）

4.JVM内存分代模型（用于分代垃圾回收算法）

1. 部分垃圾回收器使用的模型

除Epsilon ZGC Shenandoah之外的GC都是使用逻辑分代模型

G1是逻辑分代，物理不分代

除此之外不仅逻辑分代，而且物理分代

2. 新生代 + 老年代 + 永久代（1.7）Perm Generation/ 元数据区(1.8) Metaspace
   1. 永久代 元数据 - Class
   2. 永久代必须指定大小限制 ，元数据可以设置，也可以不设置，无上限（受限于物理内存）
   3. 字符串常量 1.7 - 永久代，1.8 - 堆

原文链接：GC 和 GC 调优

4. MethodArea逻辑概念 - 永久代、元数据

3. 新生代 = Eden + 2个suvivor区

   1. YGC回收之后，大多数的对象会被回收，活着的进入s0
   2. 再次YGC，活着的对象eden + s0 -> s1
   3. 再次YGC，eden + s1 -> s0
   4. 年龄足够 -> 老年代 （15 CMS 6）
   5. s区装不下 -> 老年代

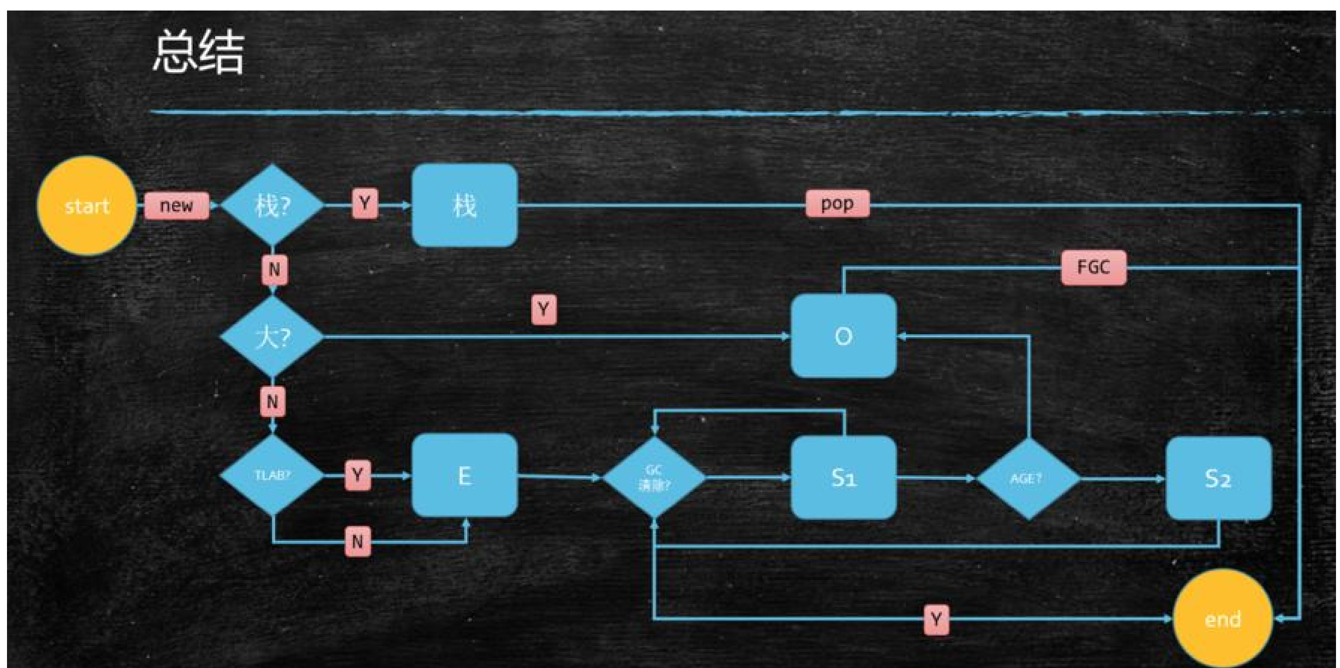
4. 老年代

   1. 顽固分子
   2. 老年代满了FGC Full GC

5. GC Tuning (Generation)

   1. 尽量减少FGC
   2. MinorGC = YGC
   3. MajorGC = FGC

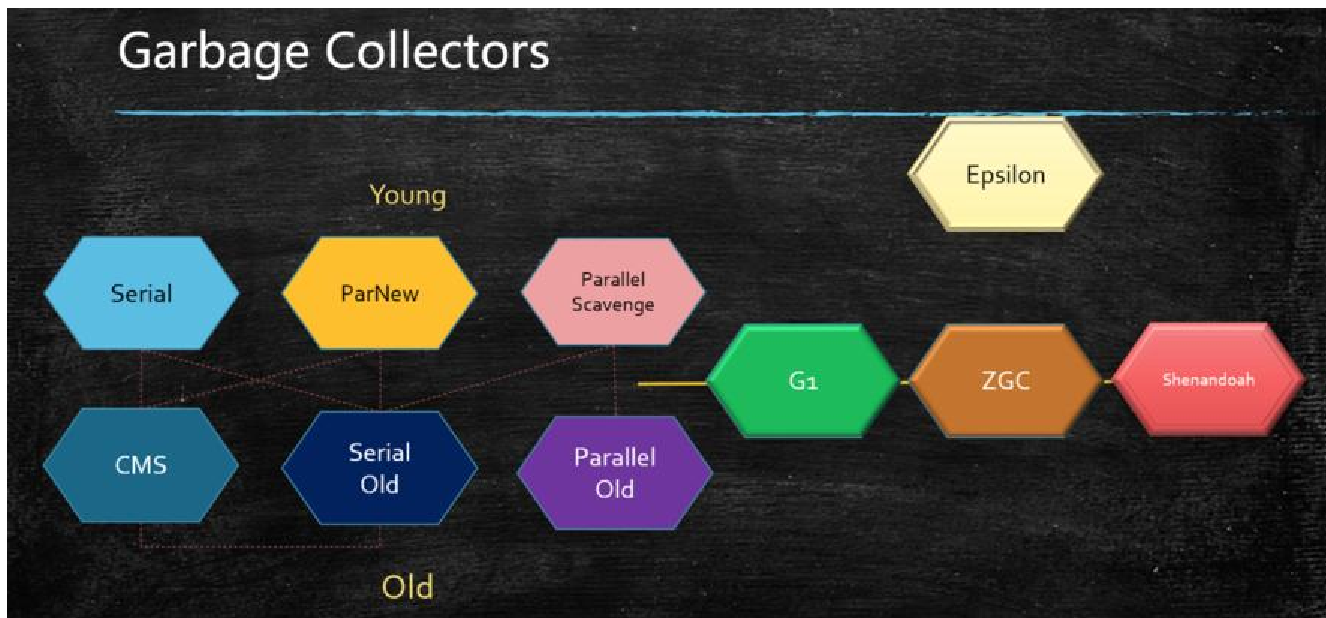
6. 对象分配过程图



7. 动态年龄：（不重要）
https://www.jianshu.com/p/989d3b06a49d

8. 分配担保：（不重要）
YGC期间 survivor区空间不够了 空间担保直接进入老年代
参考：https://cloud.tencent.com/developer/article/1082730

5.常见的垃圾回收器



1. JDK诞生 Serial追随 提高效率，诞生了PS，为了配合CMS，诞生了PN，CMS是1.4版本后期引入，MS是里程碑式的GC，它开启了并发回收的过程，但是CMS毛病较多，因此目前任何一个JDK版本默是CMS

并发垃圾回收是因为无法忍受STW

2. Serial 年轻代 串行回收

3. PS 年轻代 并行回收

4. ParNew 年轻代 配合CMS的并行回收

5. SerialOld

6. ParallelOld

7. ConcurrentMarkSweep 老年代 并发的， 垃圾回收和应用程序同时运行，降低STW的时间(200ms)

CMS问题比较多，所以现在没有一个版本默认是CMS，只能手工指定

CMS既然是MarkSweep，就一定会有碎片化的问题，碎片到达一定程度，CMS的老年代分配对象分不下的时候，使用SerialOld 进行老年代回收

想象一下：

PS + PO -> 加内存 换垃圾回收器 -> PN + CMS + SerialOld（几个小时 - 几天的STW）

几十个G的内存，单线程回收 -> G1 + FGC 几十个G -> 上T内存的服务器 ZGC

算法：三色标记 + Incremental Update

8. G1(10ms)

算法：三色标记 + SATB

9. ZGC (1ms) PK C++

算法：ColoredPointers + LoadBarrier

10. Shenandoah

算法：ColoredPointers + WriteBarrier

11. Eplison

12. PS 和 PN区别的延伸阅读： 点击

13. 垃圾收集器跟内存大小的关系

原文链接：GC 和 GC 调优

1. Serial 几十兆
    2. PS 上百兆 - 几个G
    3. CMS - 20G
    4. G1 - 上百G
    5. ZGC - 4T - 16T（JDK13）

1.8默认的垃圾回收：PS + ParallelOld

常见垃圾回收器组合参数设定：(1.8)

- -XX:+UseSerialGC = Serial New (DefNew) + Serial Old
    - 小型程序。默认情况下不会是这种选项，HotSpot会根据计算及配置和JDK版本自动选择收集器
- -XX:+UseParNewGC = ParNew + SerialOld
    - 这个组合已经很少用（在某些版本中已经废弃）
    - https://stackoverflow.com/questions/34962257/why-remove-support-for-parnewserialold-anddefnewcms-in-the-future
- -XX:+UseConcMarkSweepGC = ParNew + CMS + Serial Old
- -XX:+UseParallelGC = Parallel Scavenge + Parallel Old (1.8默认) 【PS + SerialOld】
- -XX:+UseParallelOldGC = Parallel Scavenge + Parallel Old
- -XX:+UseG1GC = G1
- Linux中没找到默认GC的查看方法，而windows中会打印UseParallelGC
    - java +XX:+PrintCommandLineFlags -version
    - 通过GC的日志来分辨
- Linux下1.8版本默认的垃圾回收器到底是什么？
    - 1.8.0_181 默认（看不出来）Copy MarkCompact
    - 1.8.0_222 默认 PS + PO

JVM调优第一步，了解JVM常用命令行参数

- JVM的命令行参数 参考
- HotSpot参数分类

标准： - 开头，所有的HotSpot都支持
非标准：-X 开头，特定版本HotSpot支持特定命令
不稳定：-XX 开头，下个版本可能取消

java -version
java -X
试验用程序：

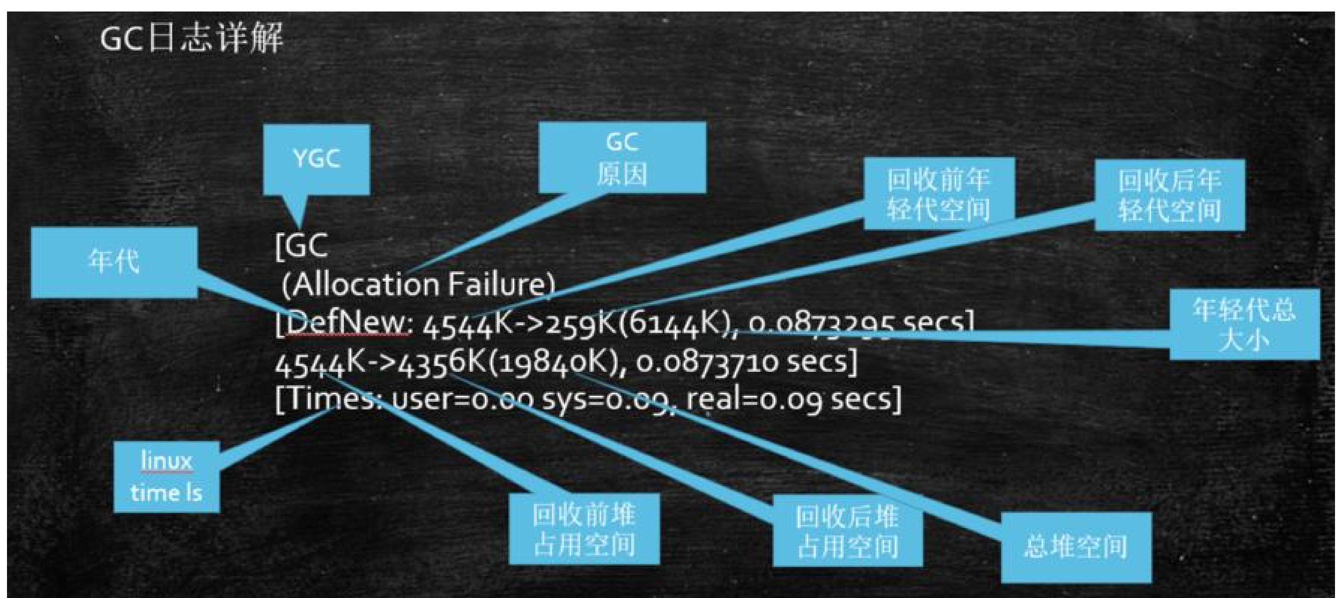
```
import java.util.List;
import java.util.LinkedList;

public class HelloGC {
  public static void main(String[] args) {
    System.out.println("HelloGC!");
    List list = new LinkedList();
    for(;;) {
      byte[] b = new byte[1024*1024];
      list.add(b);
    }
  }
}
```

1. 区分概念：内存泄漏memory leak，内存溢出out of memory
2. java -XX:+PrintCommandLineFlags HelloGC
3. java -Xmn10M -Xms40M -Xmx60M -XX:+PrintCommandLineFlags -XX:+PrintGC HelloGC PrntGCDetails PrintGCTimeStamps PrintGCCauses
4. java -XX:+UseConcMarkSweepGC -XX:+PrintCommandLineFlags HelloGC
5. java -XX:+PrintFlagsInitial 默认参数值
6. java -XX:+PrintFlagsFinal 最终参数值
7. java -XX:+PrintFlagsFinal | grep xxx 找到对应的参数
8. java -XX:+PrintFlagsFinal -version |grep GC

PS GC日志详解

每种垃圾回收器的日志格式是不同的！

PS日志格式



heap dump部分：

eden space 5632K, 94% used [0x00000000ff980000,0x00000000ffeb3e28,0x00000000fff00000

后面的内存地址指的是，起始地址，使用空间结束地址，整体空间结束地址

total = eden + 1个survivor（原因是survivor是复制算法，实际可用空间就只能是1个survivor）

调优前的基础概念：

1. 吞吐量：用户代码时间 /（用户代码执行时间 + 垃圾回收时间）
2. 响应时间：STW越短，响应时间越好

所谓调优，首先确定，追求啥？吞吐量优先，还是响应时间优先？还是在满足一定的响应时间的情况，要求达到多大的吞吐量...

问题：

科学计算，吞吐量。数据挖掘，thrput。吞吐量优先的一般：（PS + PO）

响应时间：网站 GUI API （1.8 G1）

什么是调优?

1. 根据需求进行JVM规划和预调优
2. 优化运行JVM运行环境（慢，卡顿）
3. 解决JVM运行过程中出现的各种问题(OOM)

调优，从规划开始

- 调优，从业务场景开始，没有业务场景的调优都是耍流氓
- 无监控（压力测试，能看到结果），不调优
- 步骤：
    1. 熟悉业务场景（没有最好的垃圾回收器，只有最合适的垃圾回收器）
        1. 响应时间、停顿时间 [CMS G1 ZGC] （需要给用户作响应）
        2. 吞吐量 = 用户时间 /(用户时间 + GC时间) [PS]

2. 选择回收器组合
   3. 计算内存需求（经验值 1.5G 16G）
   4. 选定CPU（越高越好）
   5. 设定年代大小、升级年龄
   6. 设定日志参数
      1. -Xloggc:/opt/xxx/logs/xxx-xxx-gc-%t.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSize=20M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCCause
      2. 或者每天产生一个日志文件
   7. 观察日志情况

- 案例1：垂直电商，最高每日百万订单，处理订单系统需要什么样的服务器配置？

这个问题比较业余，因为很多不同的服务器配置都能支撑(1.5G 16G)

1小时360000集中时间段， 100个订单/秒，（找一小时内的高峰期，1000订单/秒）

经验值，

非要计算：一个订单产生需要多少内存？512K * 1000 500M内存

专业一点儿问法：要求响应时间100ms

压测！

- 案例2：12306遭遇春节大规模抢票应该如何支撑？

12306应该是中国并发量最大的秒杀网站：

号称并发量100W最高

CDN -> LVS -> NGINX -> 业务系统 -> 每台机器1W并发（10K问题） 100台机器

普通电商订单 -> 下单 ->订单系统（IO）减库存 ->等待用户付款

12306的一种可能的模型： 下单 -> 减库存 和 订单(redis kafka) 同时异步进行 ->等付款

减库存最后还会把压力压到一台服务器

可以做分布式本地库存 + 单独服务器做库存均衡

大流量的处理方法：分而治之

- 怎么得到一个事务会消耗多少内存？
   1. 弄台机器，看能承受多少TPS？是不是达到目标？扩容或调优，让它达到
   2. 用压测来确定

优化环境

1. 有一个50万PV的资料类网站（从磁盘提取文档到内存）原服务器32位，1.5G

的堆，用户反馈网站比较缓慢，因此公司决定升级，新的服务器为64位，16G
的堆内存，结果用户反馈卡顿十分严重，反而比以前效率更低了

1. 为什么原网站慢?

很多用户浏览数据，很多数据load到内存，内存不足，频繁GC，STW长，响应时间变慢

2. 为什么会更卡顿?

内存越大，FGC时间越长

3. 咋办?

PS -> PN + CMS 或者 G1

2. 系统CPU经常100%，如何调优？(面试高频)

CPU100%那么一定有线程在占用系统资源，

1. 找出哪个进程cpu高（top）
2. 该进程中的哪个线程cpu高（top -Hp）
3. 导出该线程的堆栈 (jstack)
4. 查找哪个方法（栈帧）消耗时间 (jstack)
5. 工作线程占比高 | 垃圾回收线程占比高

3. 系统内存飙高，如何查找问题？（面试高频）

1. 导出堆内存 (jmap)
2. 分析 (jhat jvisualvm mat jprofiler ...)

4. 如何监控JVM

1. jstat jvisualvm jprofiler arthas top...

解决JVM运行中的问题

一个案例理解常用工具

1. 测试代码：

```
package com.test.jvm.gc;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * 从数据库中读取信用数据，套用模型，并把结果进行记录和传输
 */
public class TestFullGCProblem01 {
```

```
    private static class CardInfo {
        BigDecimal price = new BigDecimal(0.0);
        String name = "张三";
        int age = 5;
        Date birthDate = new Date();

        public void m() {}
    }

    private static ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor
50,
            new ThreadPoolExecutor.DiscardOldestPolicy());

    public static void main(String[] args) throws Exception {
        executor.setMaximumPoolSize(50);

        for (;;){
            modelFit();
            Thread.sleep(100);
        }
    }

    private static void modelFit(){
        List<CardInfo> taskList = getAllCardInfo();
        taskList.forEach(info -> {
            // do something
            executor.scheduleWithFixedDelay(() -> {
                //do sth with info
                info.m();

            }, 2, 3, TimeUnit.SECONDS);
        });
    }

    private static List<CardInfo> getAllCardInfo(){
        List<CardInfo> taskList = new ArrayList<>();

        for (int i = 0; i < 100; i++) {
            CardInfo ci = new CardInfo();
            taskList.add(ci);
        }

        return taskList;
    }
}
```

2. java -Xms200M -Xmx200M -XX:+PrintGC com.test.jvm.gc.TestFullGCProblem01
3. 一般是运维团队首先收到报警信息（CPU Memory）
4. top命令观察到问题：内存不断增长 CPU占用率居高不下
5. top -Hp观察进程中的线程，哪个线程CPU和内存占比高
6. jps定位具体java进程

jstack定位线程状况，重点关注：WAITING BLOCKED

eg.

waiting on <0x0000000088ca3310> (a java.lang.Object)

假如有一个进程中100个线程，很多线程都在waiting on <xx>，一定要找到是哪个线程持有这把锁

怎么找？搜索jstack dump的信息，找<xx> ，看哪个线程持有这把锁RUNNABLE

作业：1：写一个死锁程序，用jstack观察 2 ：写一个程序，一个线程持有锁不释放，其他线程等待

7. 为什么阿里规范里规定，线程的名称（尤其是线程池）都要写有意义的名称。怎么样自定义线程池的线程名称？（自定义ThreadFactory）

8. jinfo pid

9. jstat -gc动态观察gc情况 / 阅读GC日志发现频繁GC / arthas观察 / jconsole/jvisualVM/ Jprofile（最好用）

jstat -gc 4655 500 : 每个500个毫秒打印GC的情况

如果面试官问你是怎么定位OOM问题的？如果你回答用图形界面（错误）

1：已经上线的系统不用图形界面用什么？（cmdline arthas）

2：图形界面到底用在什么地方？测试！测试的时候进行监控！（压测观察）

10. jmap -histo 4655 | head -20，查找有多少对象产生

11. jmap -dump:format=b,file=xxx pid ：

线上系统，内存特别大，jmap执行期间会对**进程产生很大影响**，甚至卡顿（电商不适合）

1：设定了参数HeapDump，OOM的时候会自动产生堆转储文件

2：很多服务器备份（高可用），停掉这台服务器对其他服务器不影响

3：在线定位(一般小点儿公司用不到)

12. java -Xms20M -Xmx20M -XX:+UseParallelGC -XX:+HeapDumpOnOutOfMemoryError com.est.jvm.gc.TestFullGCProblem01

13. 使用MAT / jhat /jvisualvm 进行dump文件分析： https://www.cnblogs.com/baihuitestsoftware/articles/6406271.html

jhat -J-mx512M xxx.dump

http://192.168.17.11

拉到最后：找到对应链接

可以使用OQL查找特定问题对象

14. 找到代码的问题

jconsole远程连接

1. 程序启动加入参数：

```
java -Djava.rmi.server.hostname=192.168.17.11 -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=11111 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false XXX
```

2. 如果遭遇 Local host name unknown：XXX的错误，修改/etc/hosts文件，把XXX加入进去

```
192.168.17.11 basic localhost localhost.localdomain localhost4 localhost4.localdomain4
```

::1   localhost localhost.localdomain localhost6 localhost6.localdomain6

3. 关闭linux防火墙（实战中应该打开对应端口）

service iptables stop
chkconfig iptables off #永久关闭

4. windows上打开 jconsole远程连接 192.168.17.11:11111

jvisualvm远程连接

https://www.cnblogs.com/liugh/p/7620336.html （简单做法）

jprofiler (收费)

arthas在线排查工具

- 为什么需要在线排查？

在生产上我们经常会碰到一些不好排查的问题，例如线程安全问题，用最简单的threaddump或者heapdump不好查到问题原因。为了排查这些问题，有时我们会临时加一些日志，比如在一些关键的函数打印出入参，然后重新打包发布，如果打了日志还是没找到问题，继续加日志，重新打包发布。对于线流程复杂而且审核比较严的公司，从改代码到上线需要层层的流转，会大大影响问题排查的进度。

- jvm观察jvm信息
- thread定位线程问题
- dashboard 观察系统情况
- heapdump + jhat分析
- jad反编译

动态代理生成类的问题定位

第三方的类（观察代码）

版本问题（确定自己最新提交的版本是不是被使用）

- redefine 热替换

目前有些限制条件：只能改方法实现（方法已经运行完成），不能改方法名， 不能改属性

m() -> mm()

- sc - search class
- watch - watch method
- 没有包含的功能：jmap

GC算法的基础概念

Card Table

由于做YGC时，需要扫描整个OLD区（原因是old区可能由指向young区的东西），效率非常低，所以VM设计了CardTable，如果一个OLD区CardTable中有对象指向Y区，就将它设为Dirty，下次扫描时只需要扫描Dirty Card。在结构上，Card Table用BitMap来实现。

并发标记算法

难点：在标记对象过程中，对象引用关系正在发生改变

三色标记法

1. 白色：未被标记的对象
2. 灰色：自身被标记，成员变量未被标记
3. 黑色：自身和成员变量均已标记完成

```
graph TD
A(A:黑色) --> B(B:灰色)
A(A:黑色) --> C(C:灰色)
A(A:黑色) --> |新增| D(D:白色)
B --> |删除| D(D:白色)
```

漏标（2种情况）：

漏标是指，本来时live object，但是由于没有遍历到，被当成垃圾回收了。

1. 在remakr过程中，黑色指向了白色，如果不对黑色重新扫描，则会漏标，会把白色对象当做没有引用指向从而回收掉。
2. 并发标记过程中，删除了所有从灰色到白色的引用，会产生漏标，此时白色对象应该被回收。

怎么解决（2个方式）：

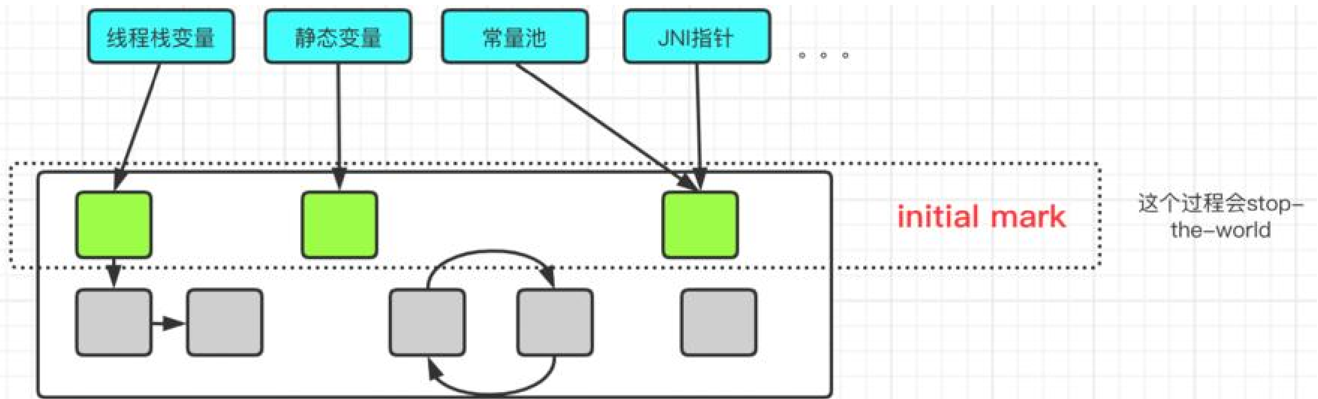
1. incremental update：增量更新，关注引用的增加，把黑色重新标记为灰色，下次重新扫描属性（CMS使用了这种）
2. SATB（snapshot at the beginning）：关注引用的删除，当指向消失时，要把这个 **引用**推到G的堆栈，保证白色还能被GC扫描到。（G1使用了这种方式）

CMS

简介

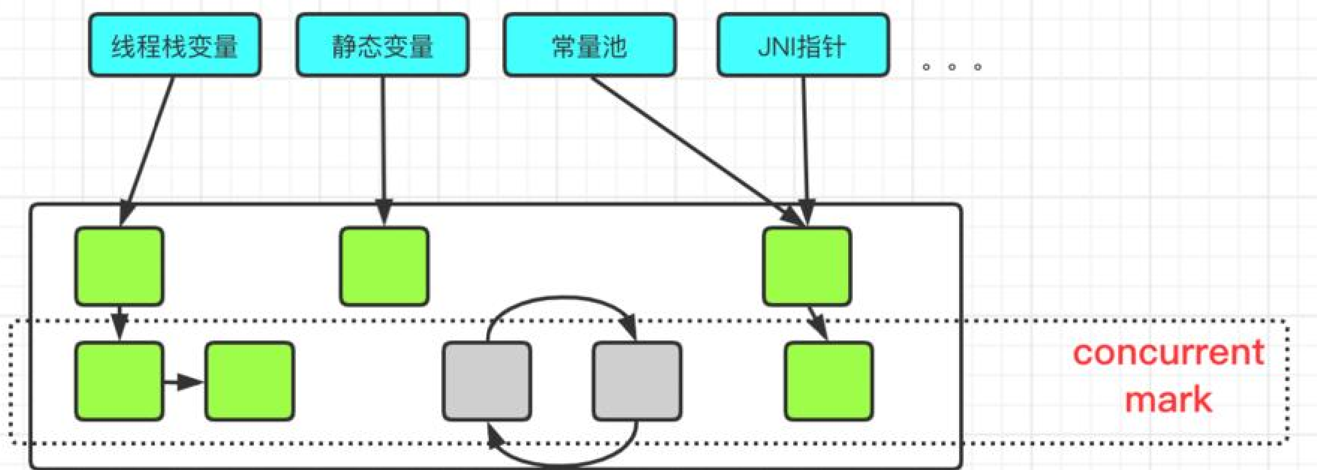
1. Concurrent Mark Sweep
2. a mostly concurrent, low-pause collector
3. 4 phases
    1. initial mark
    2. concurrent mark
    3. remark
    4. concurrent sweep

初始标记



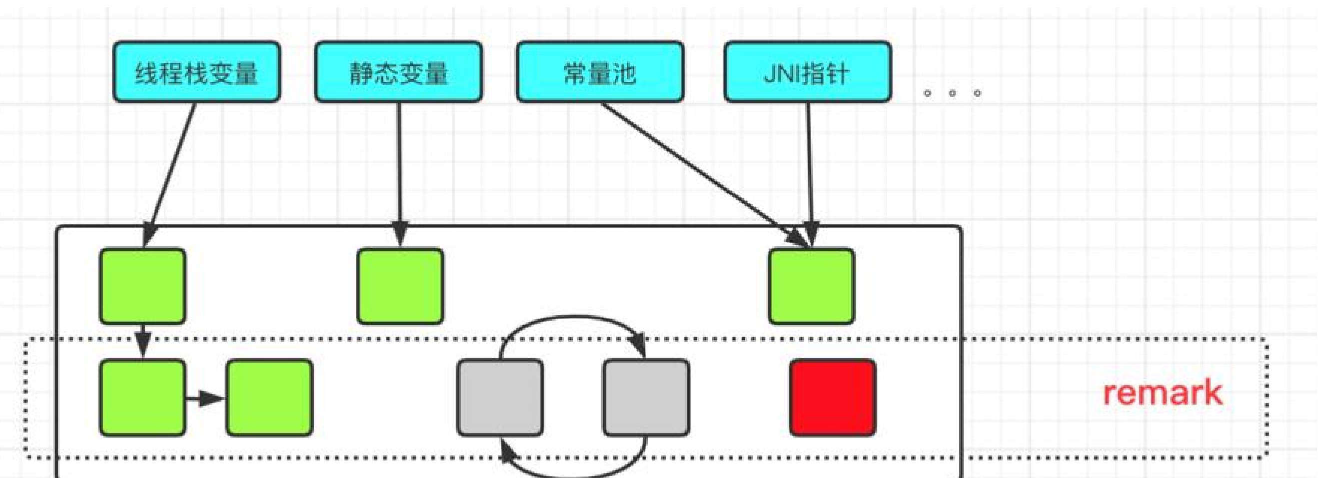
1. 这个过程会发生STW，但是由于找的对象会比较少，因此STW时间会很短

并发标记



1. 这个过程不会发送STW

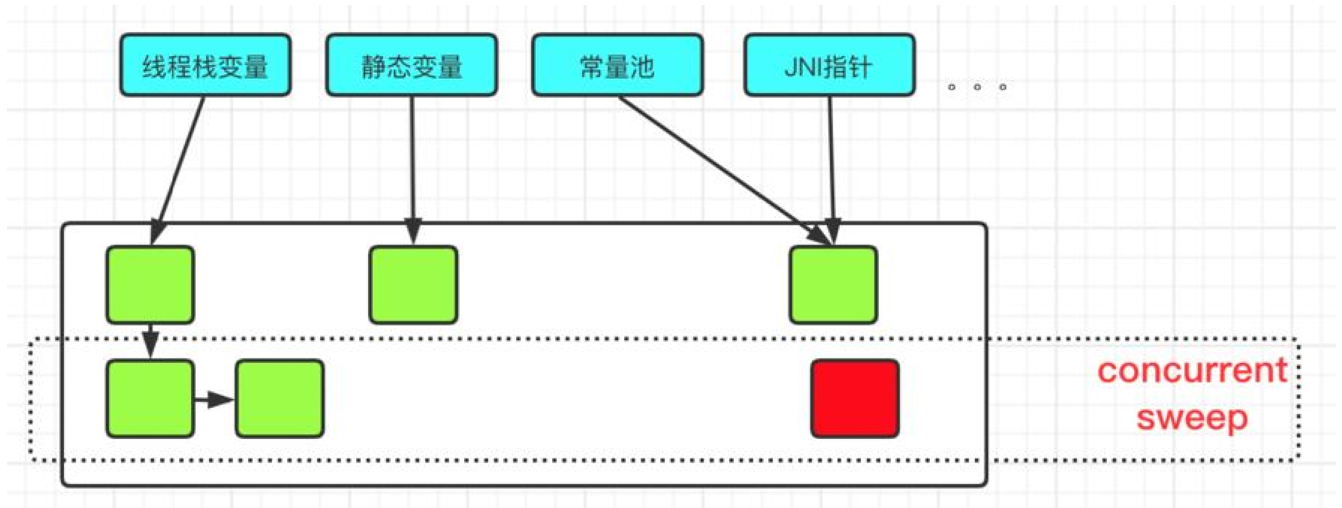
重新标记



1. 为了解决并发标记过程中，某些对象的引用不存在了，所以需要重新标记
2. 这个过程会发生STW

原文链接：GC 和 GC 调优

并发清理



1. 并发清理的过程还是可能会产生垃圾，这部分称为浮动垃圾，会在下次GC的时候清理

CMS的问题

1. Memory Fragmentation

-XX:+UseCMSCompactAtFullCollection

-XX:CMSFullGCsBeforeCompaction 默认为0 指的是经过多少次FGC才进行压缩

2. Floating Garbage

Concurrent Mode Failure

产生：if the concurrent collector is unable to finish reclaiming the unreachable objects before he tenured generation fills up, or if an allocation cannot be satisfiedwith the available free spa e blocks in the tenured generation, then theapplication is paused and the collection is comple ed with all the applicationthreads stopped

解决方案：降低触发CMS的阈值

PromotionFailed

解决方案类似，保持老年代有足够的空间

–XX:CMSInitiatingOccupancyFraction 92%可以降低这个值，让CMS保持老年代足够的空间

CMS日志分析

执行命令：java -Xms20M -Xmx20M -XX:+PrintGCDetails -XX:+UseConcMarkSweepGC com.tes .jvm.gc.TestFullGCProblem01

[GC (Allocation Failure) [ParNew: 6144K->640K(6144K), 0.0265885 secs] 6585K->2770K(1984 K), 0.0268035 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

ParNew：年轻代收集器

6144->640：收集前后的对比

（6144）：整个年轻代容量

6585 -> 2770：整个堆的情况

（19840）：整个堆大小

```
[GC (CMS Initial Mark) [1 CMS-initial-mark: 8511K(13696K)] 9866K(19840K), 0.0040321 secs] [T
mes: user=0.01 sys=0.00, real=0.00 secs]
// 8511 (13696) : 老年代使用（最大）
// 9866 (19840) : 整个堆使用（最大）

[CMS-concurrent-mark-start]
[CMS-concurrent-mark: 0.018/0.018 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
// 这里的时间意义不大，因为是并发执行

[CMS-concurrent-preclean-start]
[CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
// 标记Card为Dirty，也称为Card Marking

[GC (CMS Final Remark) [YG occupancy: 1597 K (6144 K)][Rescan (parallel) , 0.0008396 secs][w
ak refs processing, 0.0000138 secs][class unloading, 0.0005404 secs][scrub symbol table, 0.00
6169 secs][scrub string table, 0.0004903 secs][1 CMS-remark: 8511K(13696K)] 10108K(19840K)
 0.0039567 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
// STW阶段，YG occupancy:年轻代占用及容量
// [Rescan (parallel)：STW下的存活对象标记
// weak refs processing: 弱引用处理
// class unloading: 卸载用不到的class
// scrub symbol(string) table:
    // cleaning up symbol and string tables which hold class-level metadata and
    // internalized string respectively
// CMS-remark: 8511K(13696K): 阶段过后的老年代占用及容量
// 10108K(19840K): 阶段过后的堆占用及容量

[CMS-concurrent-sweep-start]
[CMS-concurrent-sweep: 0.005/0.005 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
// 标记已经完成，进行并发清理

[CMS-concurrent-reset-start]
[CMS-concurrent-reset: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
// 重置内部结构，为下次GC做准备
```

G1

简介

1. 官方简介

2. G1是一种服务端应用使用的垃圾收集器，目标是在多核、大内存的机器上，它在大多数情况下可实现指定GC暂停时间，同时还能保持较高的吞吐量。

3. 与PS相比较，G1大约比其低10%~15%的吞吐量，但是维持响应时间在200ms。

4. G1使用分治的思想来管理内存，把内存分割成一个一个的Region。每个Region可能是某一个代，

如old、survivor、tenured、Humongous。每个区域不是固定的。

5. 特点

    1. 并发收集
    2. 压缩空闲空间不会延长GC的暂停时间
    3. 更易预测的GC暂停时间
    4. 适用不需要实现很高吞吐量的场景

6. 基本概念

    1. CSet = CollectionSet

一组可被回收的分区的集合。在CSet中存活的数据会在GC过程中被移动到另一个可用分区，CSet中分区可以来自Eden空间、Survivor空间或者老年代。CSet会占用不到整个堆空间的1%大小。

    2. RSet = RememberedSet

** 记录了其他region中的对象到本region的引用。其价值在于：垃圾收集器不需要扫描整个堆找到引用了当前分区中的对象，只需要扫描RSet即可。RSet是维护在每个Region中的。

** 由于RSet的存在，那么每次给对象赋引用的时候，就得做一些额外的操作，指的时在RSet中做一额外的记录（在GC中被称为写屏障）。

G1日志详解

```
[GC pause (G1 Evacuation Pause) (young) (initial-mark), 0.0015790 secs]
// young -> 年轻代 Evacuation-> 复制存活对象
// initial-mark 混合回收的阶段，这里是YGC混合老年代回收
   [Parallel Time: 1.5 ms, GC Workers: 1] // 一个GC线程
      [GC Worker Start (ms):  92635.7]
      [Ext Root Scanning (ms):  1.1]
      [Update RS (ms):  0.0]
         [Processed Buffers:  1]
      [Scan RS (ms):  0.0]
      [Code Root Scanning (ms):  0.0]
      [Object Copy (ms):  0.1]
      [Termination (ms):  0.0]
         [Termination Attempts:  1]
      [GC Worker Other (ms):  0.0]
      [GC Worker Total (ms):  1.2]
      [GC Worker End (ms):  92636.9]
   [Code Root Fixup: 0.0 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.0 ms]
   [Other: 0.1 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.0 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.0 ms]
      [Humongous Register: 0.0 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 0.0 ms]
   [Eden: 0.0B(1024.0K)->0.0B(1024.0K) Survivors: 0.0B->0.0B Heap: 18.8M(20.0M)->18.8M(20.0M)]
```

[Times: user=0.00 sys=0.00, real=0.00 secs]
// 以下是混合回收其他阶段
[GC concurrent-root-region-scan-start]
[GC concurrent-root-region-scan-end, 0.0000078 secs]
[GC concurrent-mark-start]
// 无法evacuation，进行FGC
[Full GC (Allocation Failure) 18M->18M(20M), 0.0719656 secs]
   [Eden: 0.0B(1024.0K)->0.0B(1024.0K) Survivors: 0.0B->0.0B Heap: 18.8M(20.0M)->18.8M(20.
M)], [Metaspace: 38
76K->3876K(1056768K)] [Times: user=0.07 sys=0.00, real=0.07 secs]

案例汇总

OOM产生的原因多种多样，有些程序未必产生OOM，不断FGC(CPU飙高，但内存回收特别少）（上案例）

1. 硬件升级系统反而卡顿的问题（见上）
2. 线程池不当运用产生OOM问题（见上）

不断的往List里加对象（实在太LOW）

3. jira问题

实际系统不断重启

解决问题 加内存 + 更换垃圾回收器 G1

真正问题在哪儿？不知道

4. tomcat http-header-size过大问题（Hector）
5. lambda表达式导致方法区溢出问题(MethodArea / Perm Metaspace)

LambdaGC.java   -XX:MaxMetaspaceSize=9M -XX:+PrintGCDetails

"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" -XX:MaxMetaspaceSize=9M -XX:+PrintGCD
tails "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2019.1\lib\idea_rt.
ar=49316:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2019.1\bin" -Dfile.encod
ng=UTF-8 -classpath "C:\Program Files\Java\jdk1.8.0_181\jre\lib\charsets.jar;C:\Program Files\
ava\jdk1.8.0_181\jre\lib\deploy.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\ext\access-bridg
-64.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\ext\cldrdata.jar;C:\Program Files\Java\jdk1.8.
_181\jre\lib\ext\dnsns.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\ext\jaccess.jar;C:\Program
Files\Java\jdk1.8.0_181\jre\lib\ext\jfxrt.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\ext\local
data.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\ext\nashorn.jar;C:\Program Files\Java\jdk1.8
0_181\jre\lib\ext\sunec.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\ext\sunjce_provider.jar;C:
Program Files\Java\jdk1.8.0_181\jre\lib\ext\sunmscapi.jar;C:\Program Files\Java\jdk1.8.0_181\j
e\lib\ext\sunpkcs11.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\ext\zipfs.jar;C:\Program Files
Java\jdk1.8.0_181\jre\lib\javaws.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\jce.jar;C:\Progr
m Files\Java\jdk1.8.0_181\jre\lib\jfr.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\jfxswt.jar;C:\P
ogram Files\Java\jdk1.8.0_181\jre\lib\jsse.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\mana
ement-agent.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\plugin.jar;C:\Program Files\Java\jd
1.8.0_181\jre\lib\resources.jar;C:\Program Files\Java\jdk1.8.0_181\jre\lib\rt.jar;C:\work\ijprojec
s\JVM\out\production\JVM;C:\work\ijprojects\ObjectSize\out\artifacts\ObjectSize_jar\ObjectS
ze.jar" com.mashibing.jvm.gc.LambdaGC
[GC (Metadata GC Threshold) [PSYoungGen: 11341K->1880K(38400K)] 11341K->1888K(1259
2K), 0.0022190 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Metadata GC Threshold) [PSYoungGen: 1880K->0K(38400K)] [ParOldGen: 8K->1777
(35328K)] 1888K->1777K(73728K), [Metaspace: 8164K->8164K(1056768K)], 0.0100681 secs] [T

```
mes: user=0.02 sys=0.00, real=0.01 secs]
[GC (Last ditch collection) [PSYoungGen: 0K->0K(38400K)] 1777K->1777K(73728K), 0.0005698
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Last ditch collection) [PSYoungGen: 0K->0K(38400K)] [ParOldGen: 1777K->1629K(67
84K)] 1777K->1629K(105984K), [Metaspace: 8164K->8156K(1056768K)], 0.0124299 secs] [Tim
s: user=0.06 sys=0.00, real=0.01 secs]
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
3)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at sun.instrument.InstrumentationImpl.loadClassAndStartAgent(InstrumentationImpl.java:3
8)
    at sun.instrument.InstrumentationImpl.loadClassAndCallAgentmain(InstrumentationImpl.ja
a:411)
Caused by: java.lang.OutOfMemoryError: Compressed class space
    at sun.misc.Unsafe.defineClass(Native Method)
    at sun.reflect.ClassDefiner.defineClass(ClassDefiner.java:63)
    at sun.reflect.MethodAccessorGenerator$1.run(MethodAccessorGenerator.java:399)
    at sun.reflect.MethodAccessorGenerator$1.run(MethodAccessorGenerator.java:394)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.reflect.MethodAccessorGenerator.generate(MethodAccessorGenerator.java:393)
    at sun.reflect.MethodAccessorGenerator.generateSerializationConstructor(MethodAccesso
Generator.java:112)
    at sun.reflect.ReflectionFactory.generateConstructor(ReflectionFactory.java:398)
    at sun.reflect.ReflectionFactory.newConstructorForSerialization(ReflectionFactory.java:360)
    at java.io.ObjectStreamClass.getSerializableConstructor(ObjectStreamClass.java:1574)
    at java.io.ObjectStreamClass.access$1500(ObjectStreamClass.java:79)
    at java.io.ObjectStreamClass$3.run(ObjectStreamClass.java:519)
    at java.io.ObjectStreamClass$3.run(ObjectStreamClass.java:494)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.io.ObjectStreamClass.<init>(ObjectStreamClass.java:494)
    at java.io.ObjectStreamClass.lookup(ObjectStreamClass.java:391)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1134)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1509)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
    at javax.management.remote.rmi.RMIConnectorServer.encodeJRMPStub(RMIConnectorSer
er.java:727)
    at javax.management.remote.rmi.RMIConnectorServer.encodeStub(RMIConnectorServer.ja
a:719)
    at javax.management.remote.rmi.RMIConnectorServer.encodeStubInAddress(RMIConnecto
Server.java:690)
    at javax.management.remote.rmi.RMIConnectorServer.start(RMIConnectorServer.java:439)
    at sun.management.jmxremote.ConnectorBootstrap.startLocalConnectorServer(Connector
ootstrap.java:550)
    at sun.management.Agent.startLocalManagementAgent(Agent.java:137)
```

6. 直接内存溢出问题（少见）

《深入理解Java虚拟机》P59，使用Unsafe分配直接内存，或者使用NIO的问题

7. 栈溢出问题

-Xss设定太小

8. 比较一下这两段程序的异同，分析哪一个是更优的写法：

```
Object o = null;
for(int i=0; i<100; i++) {
    o = new Object();
    //业务处理
}

for(int i=0; i<100; i++) {
    Object o = new Object();
}
```

9. 重写finalize引发频繁GC

小米云，HBase同步系统，系统通过nginx访问超时报警，最后排查，C++程序员重写finalize引发频GC问题

为什么C++程序员会重写finalize？（new delete）

finalize耗时比较长（200ms）

10. 如果有一个系统，内存一直消耗不超过10%，但是观察GC日志，发现FGC总是频繁产生，会是什引起的？

System.gc() (这个比较Low)

11. Distuptor有个可以设置链的长度，如果过大，然后对象大，消费完不主动释放，会溢出 (来自 死风情)

12. 用jvm都会溢出，mycat用崩过，1.6.5某个临时版本解析sql子查询算法有问题，9个exists的联合sl就导致生成几百万的对象（来自 死物风情）

13. new 大量线程，会产生 native thread OOM，（low）应该用线程池，

解决方案：减少堆空间（太TMlow了）,预留更多内存产生native thread

JVM内存占物理内存比例 50% - 80%

GC参数

GC常用参数

- -Xmn -Xms -Xmx -Xss

年轻代 最小堆 最大堆 栈空间

- -XX:+UseTLAB

使用TLAB，默认打开

- -XX:+PrintTLAB

打印TLAB的使用情况

- -XX:TLABSize

设置TLAB大小

- -XX:+DisableExplictGC

System.gc()不管用 ，FGC

- -XX:+PrintGC
- -XX:+PrintGCDetails
- -XX:+PrintHeapAtGC
- -XX:+PrintGCTimeStamps
- -XX:+PrintGCApplicationConcurrentTime(低)

打印应用程序时间

- -XX:+PrintGCApplicationStoppedTime （低）

打印暂停时长

- -XX:+PrintReferenceGC（重要性低）

记录回收了多少种不同引用类型的引用

- -verbose:class

类加载详细过程

- -XX:+PrintVMOptions
- -XX:+PrintFlagsFinal和-XX:+PrintFlagsInitial

必须会用

- -Xloggc:opt/log/gc.log
- -XX:MaxTenuringThreshold

升代年龄，最大值15

- 锁自旋次数 -XX:PreBlockSpin 热点代码检测参数-XX:CompileThreshold逃逸分析 标量替换 ...

这些不建议设置

Parallel常用参数

- -XX:SurvivorRatio
- -XX:PreTenureSizeThreshold

大对象到底多大

- -XX:MaxTenuringThreshold
- -XX:+ParallelGCThreads

并行收集器的线程数，同样适用于CMS，一般设为和CPU核数相同

- -XX:+UseAdaptiveSizePolicy

自动选择各区大小比例

CMS常用参数

- -XX:+UseConcMarkSweepGC
- -XX:ParallelCMSThreads

CMS线程数量

- -XX:CMSInitiatingOccupancyFraction

使用多少比例的老年代后开始CMS收集，默认是68%(近似值)，如果频繁发生SerialOld卡顿，应该调，（频繁CMS回收）

- -XX:+UseCMSCompactAtFullCollection

在FGC时进行压缩

- -XX:CMSFullGCsBeforeCompaction

多少次FGC之后进行压缩

- -XX:+CMSClassUnloadingEnabled
- -XX:CMSInitiatingPermOccupancyFraction

达到什么比例时进行Perm回收

- GCTimeRatio

设置GC时间占用程序运行时间的百分比

- -XX:MaxGCPauseMillis

停顿时间，是一个建议时间，GC会尝试用各种手段达到这个时间，比如减小年轻代

G1常用参数

- -XX:+UseG1GC
- -XX:MaxGCPauseMillis

建议值，G1会尝试调整Young区的块数来达到这个值

- -XX:GCPauseIntervalMillis

？GC的间隔时间

- -XX:+G1HeapRegionSize

分区大小，建议逐渐增大该值，1 2 4 8 16 32。

随着size增加，垃圾的存活时间更长，GC间隔更长，但每次GC的时间也会更长

ZGC做了改进（动态区块大小）

- G1NewSizePercent

新生代最小比例，默认为5%

- G1MaxNewSizePercent

新生代最大比例，默认为60%

- GCTimeRatio

GC时间建议比例，G1会根据这个值调整堆空间

- ConcGCThreads

线程数量

- InitiatingHeapOccupancyPercent

启动G1的堆空间占用比例

参考资料

原文链接：GC 和 GC 调优

1. https://blogs.oracle.com/jonthecollector/our-collectors
2. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html
3. http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp
4. JVM调优参考文档： https://docs.oracle.com/en/java/javase/13/gctuning/introduction-garage-collection-tuning.html#GUID-8A443184-7E07-4B71-9777-4F12947C8184
5. https://www.cnblogs.com/nxlhero/p/11660854.html 在线排查工具
6. https://www.jianshu.com/p/507f7e0cc3a3 arthas常用命令
7. Arthas手册：
    1. 启动arthas java -jar arthas-boot.jar
    2. 绑定java进程
    3. dashboard命令观察系统整体情况
    4. help 查看帮助
    5. help xx 查看具体命令帮助
8. jmap命令参考： https://www.jianshu.com/p/507f7e0cc3a3
    1. jmap -heap pid
    2. jmap -histo pid
    3. jmap -clstats pid