GC总结：

说到GC，肯定绕不过JVM，我们先来说一下JVM。JVM是java实现跨平台运行的基础，主要包含三大块，类加载机制、内存模型，GC。

一 类加载机制：

1 主要工作：

将字节码文件读取并生成VM能够识别的运行时数据结构并初始化对象

2 主要阶段：

加载、校验、准备、解析、初始化、使用、卸载七个阶段。

3 各阶段的主要工作：

加载：根据类的全限定名读取字节流，并在方法区生成运行时数据结构，并且在堆区创建从class对象

校验：（1）文件格式校验，校验是否是.class文件，如：是否是字节码格式例如文件开头的魔数以及注册版本号，常量池的常量类型是否存在不合法类型等；（2）元数据校验，主要对类进行语义校验，比如类是否是父类，是否继承了不允许被继承的类等；（3）字节码校验，主要是校验字节码是否存在具有危害的命令，语句或是违法语句（4）符号校验：主要检查类，字段，方法的访问域

准备：主要是为类的静态属性分配空间，并设置默认值，类的Final属性这时设置代码中设定的初始值

解析：将常量池中的符号引用替换成直接引用

初始化：执行类的<clinit>方法，但是类和接口的执行方式不一样，类需要先执行父类的<clinit>, 接口则不需要

4 加载时机：

创建对象，使用静态变量和静态方法，反射调用类和动态代理，初始化类父类也会初始化，初始化接口实现类，接口也会初始化

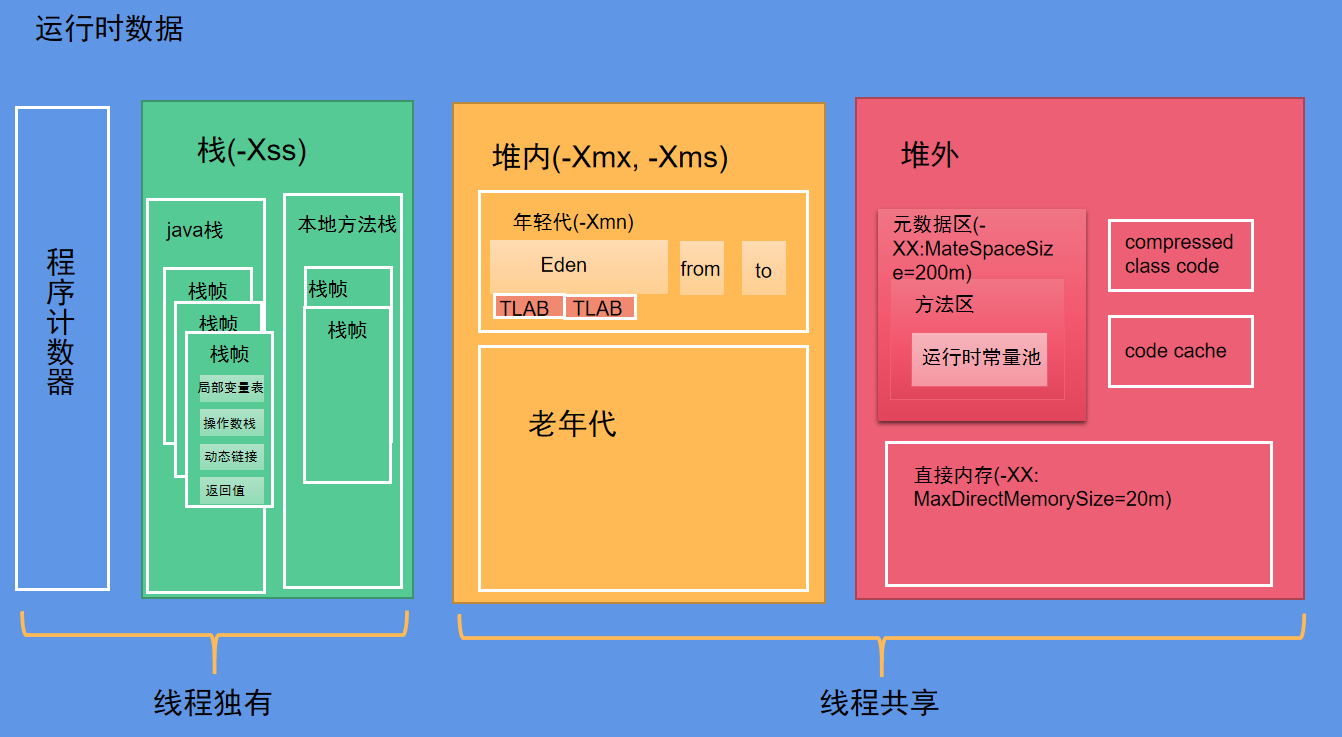
5 类加载器种类

包含启动类加载器， 扩展类加载器（已被平台类加载器替代），应用类加载器

1. 双亲委派模型

类加载器收到类加载请求，会首先将请求委托给父类加载器去实现。历史上出现了三次了这种模型的破坏：第一次，在classLoader 出现之前就存在了自定义类加载器，为了兼容，在ClassLoader类中添加了protect findclass(), 引导用户重写该方法来做自定义类加载；第二次，JNDI主要任务做资源查找与管理，但是启动类加载器并不认识他，通过引入线程上下文类加载器解决这个问题，用Thread Context ClassLoader 去解决SPI加载问题；第三次，OSGI模块儿化热部署用的是自定义的类加载机制

二 内存模型



三 GC

1. GC的工作：

GC的主要工作就是对堆内存进行管理，通过回收废弃对象来实现，包括YGC，FullGC.

针对对象管理，将对象分为两种，朝生夕灭对象与存活时间很久的对象。因此将堆内存分为两块新生代，老年代。新生代存储朝生夕灭对象，老年代存储存活时间比较长的对象。当新生代的对象经历一段时间的YGC仍然存活，那么将其提升到老年代。针对新生代和老年代对象存活时间的长短，对新生代采用复制算法，对老年代采用标记清除或是标记整理。

1. GC算法：

复制算法：新生代对象朝生夕灭，发生YGC时，存活的对象很少，对对象进行复制，然后在将整块儿eden, from survior 区整块清除效率比较高

标记-清除：主要针对老年代，发生FullGC时，老年代依然有很多存活对象，这时采用清除算法效率比较高

标记-整理：实际是标记-清除-整理，主要解决标记清除产生的大量碎片空间的问题

3 GC种类：

Serial GC :串行GC，STW时间最长，集中精力进行垃圾清除

ParallelGC:并行GC，多线程同时进行清理，吞吐量最高的垃圾回收器组合

CMS：并发GC，CMS主要处理老年代回收，年轻代使用ParNew, 开始考虑低延迟的一款收集器

G1：第一款全功能收集器，第一次引入Region分区模式，实现STW可控的一款收集器

ZGC/Shenandoah: 追求更低的延迟甚至0延迟，这意味着回收算法会更复杂

4 GCG工具：

|  |  |  |
| --- | --- | --- |
| 工具 | 命令 | 展示内容 |
| jps | Jps -l | 查看进程信息 |
| jinfo | Jinfo -flags pid | 查看进程信息 |
| Jstat | Jstat -gc pid | 查看JVM内部GC信息： |
| jmap （PS：生产环境Jmap 慎用  ） | Jmap -heap(histo) pid | 查看堆信息或是对象创建排序 |
| Jstack | Jstack -l pid | 查看堆栈信息： |

还有一些可视化工具 jconsole, jmc, visualGC

5 GC实践

系统的两个指标： 吞吐量，延迟时间

（1）根据业务实践，追求吞吐量高，而对延迟没有太大要求可以选择使用并行GC；如果追求低延迟，则在服务器内存<8G, JDK版本为JDK8时，可以选用并发GC（ParNew + CMS）,如果内存大于8G，可以尝试采用G1，甚至zgc/ shenandual

（2）GCLog分析：

分配速率问题：

正常系统： 分配速率较低 ~回收速率 ->系统健康

内存泄漏：分配速率持续大于回收速率 -> OOM

性能劣化： 分配速率较高 ~回收速率 -> 亚健康状态（CMSGC ->SerialGC）

确认问题方案：

1. 通过jmap -heap pid 或是ps-ef|grep str 查看进程的配置
2. 通过jstat -gc -pid 查看回收情况，查看回收YGC发生的次数，每次暂停时间， FullGC发生次数，暂停时间，如果有异常，比如FullGC 特别频繁， 或是停顿 时间特别长，这时影响因素，老年代Size，程序创建对象Size，FullGC时间过 程是否发生异常，以及是否出现回收器退化问题等
3. 用jmap -histo pid 查询进程创建对象的排序，是否会出现自定义引用类型排 名靠前，如果出现，考虑是否需要优化（PS：可以将生产环境的堆内存的状况 档下来在本地用jmc进行分析， jmap-dump:format=b,file=heap.info fileName）

过早提升问题：将长时间存活的对象由年轻代提升到老年代。提升速率会影响GC赞提 速率，从而严重影响系统吞吐量。

症状：

短时间频繁FullGC，

每次FullGC后老年代使用率很低，在10%~20%，甚至更低

提升速率接近分配速率

解决方案：让临时对象能够在年轻代

①增加年轻代的大小

②减少每次批处理的数量，这里指业务批处理逻辑

1. 具体事例分析 通过测试串行/并行/CMS/G1 回收器，做了两个表格：

第一张表：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 收集器 | 命令 | 堆内存 | YGC次数/平均时长 | FullGC次数/平均时长 | 生成对象 | 异常 |
| SerialGC | java -XX:+UseSerialGC -Xmx120m -Xms120m -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails GCLogAnalysis | 120m | 8/10ms内 | 16/15ms |  | 堆溢出 |
| SerialGC | java -XX:+UseSerialGC -Xmx512m -Xms512m -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 512m | 9/25ms | 7/1ms | 8837 |  |
| SerialGC | java -XX:+UseSerialGC -Xmx1g -Xms1g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 1g | 10/47ms | 0 | 10585 |  |
| SerialGC | java -XX:+UseSerialGC -Xmx2g  -Xms2g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 2g | 5/80ms | 0 | 11181 |  |
| SerialGC | java -XX:+UseSerialGC -Xmx4g  -Xms4g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 4g | 2/150ms | 0 | 8499 |  |
| ParallelGC | java -XX:+UseParallelGC -Xmx120m -Xms120m -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 120m | 8/6ms | 15/20ms |  | 堆溢出 |
| ParallelGC | java -XX:+UseParallelGC -Xmx512m -Xms512m -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 512m | 14/15ms | 9/50ms | 8910 |  |
| ParallelGC | java -XX:+UseParallelGC -Xmx1g -Xms1g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 1g | 14/20ms | 2/50ms | 14636 |  |
| ParallelGC | java -XX:+UseParallelGC -Xmx2g -Xms2g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 2g | 8/35ms | 0 | 15781 |  |
| ParallelGC | java -XX:+UseParallelGC -Xmx4g -Xms4g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 4g | 3/45ms | 0 | 13098 |  |
| ParallelGC | java -XX:+UseParallelGC -Xmx8g -Xms8g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 8g | 0 | 0 | 7509 |  |
| CMSGC | java -XX:+UseConcMarkSweepGC -Xmx256m -Xms256m -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 256m | 6/10ms | 5/ | 4113 | 出现cms降级到serial |
| CMSGC | java -XX:+UseConcMarkSweepGC -Xmx512m -Xms512m -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 512m | 13/20ms | 3/ | 8501 | 出现cms降级到serial |
| CMSGC | java -XX:+UseConcMarkSweepGC -Xmx1g -Xms1g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 1g | 9/45ms | 2/ | 11653 |  |
| CMSGC | java -XX:+UseConcMarkSweepGC -Xmx2g -Xms2g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 2g | 7/60ms | 0 | 12582 |  |
| CMSGC | java -XX:+UseConcMarkSweepGC -Xmx4g -Xms4g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 4g | 6/70ms | 0 | 13075 |  |
| CMSGC | java -XX:+UseConcMarkSweepGC -Xmx8g -Xms8g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 8g | 6/80ms | 0 | 12280 |  |
| G1GC | java -XX:+UseG1GC -Xmx256m -Xms256m -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 256m | 7/5ms | 1/14ms | 1451 |  |
| G1GC | java -XX:+UseG1GC -Xmx512m -Xms512m -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 512m | 11/8ms | 2/7ms | 3718 |  |
| G1GC | java -XX:+UseG1GC -Xmx1g -Xms1g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 1g | 5/6ms | 0 | 3638 |  |
| G1GC | java -XX:+UseG1GC -Xmx2g -Xms2g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 2g | 7/12ms | 0 | 4126 |  |
| G1GC | java -XX:+UseG1GC -Xmx4g -Xms4g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 4g | 7/12ms | 0 | 4762 |  |
| G1GC | java -XX:+UseG1GC -Xmx8g -Xms8g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 8g | 4/20ms | 0 | 5524 |  |
| G1GC | java -XX:+UseG1GC -Xmx16g -Xms16g -XX:-UseAdaptiveSizePolicy -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis | 16 | 2/45ms | 0 | 6885 |  |

将上面表做一个总结：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 收集器 | 512m | 1g | 2g | 4g | 8g | 12g |
| Serial | 10000 | 13000 | 12000 | 9000 | 7000 | 7300 |
| Parallel | 10000 | 14000 | 14000 | 12000 | 7400 | 7000 |
| CMS | 10000+ | 12000+ | 11000+ | 9000+ | 5000+ | 4000+ |
| G1 | 10000~12000 | 12000~15000 | 9000~15000 | 11000~17000 | 17381 | 17000 |

先看整张表看，发现并不是内存越大，吞吐量越高，此外每个收集器表现最好的吞吐量的内存并不统一。由此得出来一个结论，并不是内存越大越好，能够满足需求，内存越小越好。

SerialGC 表现最好的内存是1g, 因为Serial是单线程进行收集，当进行垃圾收集的时候，需要业务线程进行停顿，当内存太小的时候，会发生多次GC，影响吞吐量， 当内存太大，因为年轻代使用复制算法，内存的增大，代表需要收集的范围增大，需要复制的对象也会增多，从而增大STW时间，导致吞吐量下降.

ParallelGC 表现最好的内存是1g~2g, 发现吞吐量明显高于SerialGC, parallel是并行多线程去做处理，因此STW时间明显减少，出现的结果是吞吐量提升

CMSGC，表现最好的内存是1g, 内存太小，GC次数增加，GC时间可能也会有提升，倒是吞吐量下去，而且，当内存小的时候CMSGC可能退化到SerialGC ,垃圾收集的时间会有特别明显的增加，甚至最后内存分配速率明显大于回收速率的时候会发生OOM。

G1GC 有表中数据可以发现，当内存越大的时候吞吐量越高，这和G1的模式有关，采用Region块形势进行回收，通过空闲列表统计region的垃圾情况，来控制垃圾回收，使垃圾回收处于可控的情况中。

此外通过表一可以看出，当内存越小时，发生垃圾回收的次数越多，每次的STW相对较小。当随着内存增大，垃圾回收的次数减少，甚至不发生垃圾回收，但是，在伴随垃圾回收次数减小的同时，每次STW停顿时间增大。此外，当内存不足，并且回收的空间不多的情况下，频繁FullGC， CMS， G1很容易发生降级，使用串行回收器进行垃圾收集，这个是在生产中需要预防的。

1. 使用压测工具（wrk或sb），演练gateway-server-0.0.1-SNAPSHOT.jar示例

根据运行结果做了张表：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 收集器类型 | 命令 | P50 | P95 | P99 | 堆内存 | QPS |
| SerialGC | java -Dserver.port=8888 -server -XX:+UseSerialGC -Xmx1g -Xms1g -jar gateway-server-0.0.1-SNAPSHOT.jar | 20ms | 25ms | 29ms | 1g | 4730 |
| SerialGC | java -Dserver.port=8888 -server -XX:+UseSerialGC -Xmx2g -Xms2g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 33ms | 61ms | 2g | 4066 |
| SerialGC | java -Dserver.port=8888 -server -XX:+UseSerialGC -Xmx4g -Xms4g -jar gateway-server-0.0.1-SNAPSHOT.jar | 22ms | 36ms | 54ms | 4g | 4047 |
| SerialGC | java -Dserver.port=8888 -server -XX:+UseSerialGC -Xmx8g -Xms8g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 32ms | 50ms | 8g | 4272 |
| ParallelGC | java -Dserver.port=8888 -server -XX:+UseParallelGC -Xmx1g -Xms1g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 35ms | 63ms | 1g | 4272 |
| ParallelGC | java -Dserver.port=8888 -server -XX:+UseParallelGC -Xmx2g -Xms2g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 34ms | 55ms | 2g | 4201 |
| ParallelGC | java -Dserver.port=8888 -server -XX:+UseParallelGC -Xmx4g -Xms4g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 32ms | 41ms | 4g | 4408 |
| ParallelGC | java -Dserver.port=8888 -server -XX:+UseParallelGC -Xmx8g -Xms8g -jar gateway-server-0.0.1-SNAPSHOT.jar | 22ms | 35ms | 53ms | 8g | 4002 |
| CMSGC | java -Dserver.port=8888 -server -XX:+UseConcMarkSweepGC -Xmx1g -Xms1g -jar gateway-server-0.0.1-SNAPSHOT.jar | 20ms | 33ms | 51ms | 1g | 4347 |
| CMSGC | java -Dserver.port=8888 -server -XX:+UseConcMarkSweepGC -Xmx2g -Xms2g -jar gateway-server-0.0.1-SNAPSHOT.jar | 22ms | 33ms | 57ms | 2g | 3981 |
| CMSGC | java -Dserver.port=8888 -server -XX:+UseConcMarkSweepGC -Xmx4g -Xms4g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 32ms | 43ms | 4g | 4101 |
| CMSGC | java -Dserver.port=8888 -server -XX:+UseConcMarkSweepGC -Xmx8g -Xms8g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 35ms | 31ms | 8g | 4612 |
| G1GC | java -Dserver.port=8888 -server -XX:+UseG1GC -Xmx1g -Xms1g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 33ms | 67ms | 1g | 4006 |
| G1GC | java -Dserver.port=8888 -server -XX:+UseG1GC -Xmx2g -Xms2g -jar gateway-server-0.0.1-SNAPSHOT.jar | 22ms | 34ms | 59ms | 2g | 3905 |
| G1GC | java -Dserver.port=8888 -server -XX:+UseG1GC -Xmx4g -Xms4g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 32ms | 44ms | 4g | 4094 |
| G1GC | java -Dserver.port=8888 -server -XX:+UseG1GC -Xmx8g -Xms8g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 26ms | 29ms | 8g | 4592 |
| G1GC | java -Dserver.port=8888 -server -XX:+UseG1GC -Xmx12g -Xms12g -jar gateway-server-0.0.1-SNAPSHOT.jar | 21ms | 26ms | 29ms | 12g | 4646 |

PS以上表格数据存在偏差

总结：

首先我们看到表格的一个大趋势是每个垃圾收集器的P50,P95,P99针对不同内存表现出来的差异并不大，但是最终的QPS是不同的。

SerialGC，QPS最好的是1g, 随着内存增大QPS下降

ParallelGC, QPS 最好的是4g, 表现出来的并不是内存越大越好，也不是内存越小越好， 而是有个合适的内存，在生产环境也是，追求吞吐量，并行GC是最优选择，但是要注意内存配置

CMSGC，QPS最好的是8g,但是它并不是随着内存增大吞吐量增大的，我们在表中可以看出， 1G的吞吐量高于1g,4g。但是明显可以看出，延迟时间是随内存增加而减少的，所以在追求低延迟的业务场景，既满足用户需要，同时满足高吞吐，4~8G最合适。

G1： 随着内存增大的过程中，吞吐量明显是上升趋势，而且当内存大于8g的时候延迟有比较明显的变化，所以当内存比较大，并且追求低延迟，G1是最好的选择