**一、背景**

在计算机程序运行过程中，为了能够加快运行速度，提高运行效率，都会提前将指令及数据加载到内存中。对加载到内存中的数据来说，一部分是需要常驻内存的，而另一部分则是在计算结束后，不会被再次使用，因此，这部分数据被称之为垃圾数据。在程序运行过程中，为了防止内存空间被耗尽，需要对内存进行管理，对于这些已经不再使用的空间进行释放。不同的编程语言，内存管理的方式是不同的。对于C/C++来说，内存管理交由程序员手动完成。而对于Java或者其他运行在JVM上的编程语言来说，程序员不需要直接参与内存的管理，这些工作直接由JVM自动完成，这就是JVM的垃圾回收机制（Garbage Collection）。这种方式极大的提高了编程的效率，也降低了程序员手动管理内存带来的诸多Bug及风险。

尽管JVM可以帮助我们完成内存管理，但这并不意味着，作为程序员的我们可以完全不需要关心JVM内存回收的机制，因为，由于程序设计的不合理，导致JVM执行GC不符合预期引发的线上故障不计其数。具体列举如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **故障名称** | **故障定级** | **故障影响** | **原因简介** | **COE链接** |
|  | 快驴商城不可用 | S3 | * 快驴商城不可用，页面无法正常打开。 * 快驴商城页面打开正常后，商品促销价无法正常展示。 | * 依赖的上游服务故障，导致快驴营销服务打印了大量异常栈信息，线程Block。 * 大量线程Block，造成下游服务大量超时，而在超时情况下，会触发下游的重试机制，营销服务进一步劣化。 * 由于Block进程无法释放，导致线程不断创建，内存消耗过大，触发FGC。 * 频繁FGC导致CPU消耗过大。 | [COE](https://coe.mws.sankuai.com/detail/203229) |
|  | 到综鲜花详情页展示下架 | S3 | * 订单损失预估0.57%。 * 金额损失预估：37810元。 * 造成商诉39个。 * 影响商品数：40+商品，170+SKU，涉及384客户。 | 商品下SKU价格版本数量太多，导致的Full GC和慢查询 | [COE](https://coe.mws.sankuai.com/detail/143918) |
|  | 商品展示productShelf服务不可用 | S9 | * 服务不可用时长：1小时。 * 影响订单量为：300单。 | * 商品展示服务出现Metaspace空间耗尽，出现OutOfMemory * 该服务加载的类较多，正常情况下，Metaspace空间的使用空间为200M左右。 * 新扩容出来的机器只有256M，容易触发Metaspace的FGC。 * Metaspace内存空间的回收极为苛刻，触发OutOfMemory。 | [COE](https://coe.mws.sankuai.com/detail/198119) |
|  | devicehub服务频繁发生fullGC引发服务不可用 | S9 | * 影响设备消息订阅。 | * 设备重复订阅及设备解除订阅后，内存未完全释放，导致内存泄漏。 * 设备订阅请求增加，导致内存一直无法释放，出现FGC。 | [COE](https://coe.mws.sankuai.com/detail/159038) |
|  | 投线列表查询fullgc | S9 | * 暂不明确。 | * 接口查询过程中，查询结果返回了大对象。 * 接口流量上升，导致短时间内生成大量大对象，触发FGC。 | [COE](https://coe.mws.sankuai.com/detail/77848) |

从历史的一些线上事故中，可以发现，常见的GC问题有很多，但根因却千差万别，很多与JVM的GC并没有直接的关系，如何才能透过GC指标异常这个现象看透问题的根因这个是本篇的核心目标。通过本篇的分析，希望可以提升在编码过程中的质量意识，从而在开发阶段消灭一些线上可能出现的隐患，提升系统的稳定性。

# 二、GC原理

在上一个章节中，我们已经介绍了，需要进行垃圾回收的原因——在程序执行过程中对内存进行管理，对不再使用的空间进行释放。显而易见，各种垃圾回收算法的核心目标就是为了判断出死亡的对象，并清除死亡的对象，从而释放对应的内存空间。如何判断对象是否存活呢？一般来说，主要有两种算法：一个是引用计数法，另一个是可达性分析算法。

//

### 3）对象存活识别算法对比

|  |  |  |
| --- | --- | --- |
| **算法** | **优势** | **劣势** |
| 引用计数法 | * 原理简单，实现方便 * 判定效率高 | * Java中定义了多种不同的对象引用方式，软引用、弱引用、虚引用。不同的引用类型，JVM行为不同，引用计数法无法解决这种多种类型引用的问题。 * 该算法无法解决对象循环引用场景下的对象存活识别的问题。 |
| 可达性分析算法 | * 结果准确可靠， * 对于不同的引用类型，该算法也可以很好的执行。 * 可以解决对象循环引用的问题。 | * 判定效率一般 |

引用计数法目前在Python等脚本语言上被使用。而可达性分析算法被应用于主流的JVM虚拟机上。

### 4）对象回收时机

目前，主流的商用语言（比如Java、C#等）都采用的是可达性分析算法来判定对象是否存活。但是一个对象是否真的可以被回收，并不仅仅依赖于可达性分析算法的计算结果。

在一次可达性分析算法结束后，对于那些不可达的对象并不会被立刻回收，而是会先判定当前对象是否有必要执行finalize()。一般来说，如果对象并没有覆盖finalize()方法，或者当前对象已经执行过该方法，则认为当前没有必要执行此方法。如果当前对象被认为有必要执行finalize()方法，则会将这个对象放入一个叫做F-Queue的队列之中，并在之后会由虚拟机创建一个低优先级的Finalizer线程执行该队列中每一个对象的finalize()方法。在执行这个finalize()方法的过程中，可以将自己与GC Root引用链的任何一个对象相连，从而达到将自己解救的目的。需要注意的是，Finalizer线程在执行finalize()方法时，并不会承诺一定会等待该方法返回。这样设计主要是考虑到可能会存在一些finalize()方法执行速度较慢，或者由于用户代码设计问题，导致其他对象永远处于等待状态，在内存耗尽后，虚拟机崩溃。

## 2.3、常见基础垃圾回收算法

### 1）名词介绍

|  |  |  |
| --- | --- | --- |
|  | **名词** | **解释** |
|  | Stop-the-world（STW） | 在STW发生后，除了JVM的GC所需要的线程外，其他线程都会暂停工作，这些被暂停的线程只有在GC结束后，才能继续它们的任务。无论何种GC算法，都会存在STW现象。降低STW的时间是GC调优重要的工作之一。 |
|  | 响应速度 | 应用系统对一个请求的响应的时间。 |
|  | 吞吐量 | 在一个特定的时间段内，应用系统的最大工作量。 |

### 2）标记清除法（Mark-Sweep）

标记清除法，是一种基础常见的垃圾收集算法，该算法于1960年代被提出，并应用于Lisp语言。

//

### 2）分代区域描述

|  |  |  |
| --- | --- | --- |
|  | **分代名** | **描述** |
|  | 新生代 | * 新生代主要分为两个部分：Eden区和Survivor区，其中Survivor区又可以分为两个部分，S0和S1。 * 该区域中，相对于老年代空间较小，对象的生存周期短，GC频繁。因此在该区域通常使用标记复制算法。 |
|  | 老年代 | 老年代整体空间较大，对象的生命周期长，存活率高，回收不频繁。因此更适合标记整理算法。 |
|  | 永久代 | 永久代又称为方法区，存储着类和接口的元信息以及interned的字符串信息。在JDK8中被元空间取代。 |
|  | 元空间 | JDK8以后引入，方法区也存在于元空间。 |

### 3）分代垃圾回收算法执行过程

* 初始态：对象分配在Eden区，S0、S1区几乎为空。
* 随着程序的运行，越来越多的对象被分配在Eden区。
* 当Eden放不下时，就会发生MinorGC（即YoungGC），此时，会先标识出不可达的垃圾对象，然后将可达的对象移动到S0区，并将不可达的对象清理掉。这时候，Eden区就是空的了。在这个过程中，使用了标记清理算法及标记复制算法。
* 随着Eden放不下时，会再次触发minorGC，和上一步一样，先标记。这个时候，Eden和S0区可能都有垃圾对象了，而S1区是空的。这个时候，会直接将Eden和S0区的对象直接搬到S1区，然后将Eden与S0区的垃圾对象清理掉。经历这一轮的MinorGC后，Eden与S0区为空。
* 随着程序的运行，Eden空间会被分配殆尽，这时会重复刚才MinorGC的过程，不过此时，S0区是空的，S0和S1区域会互换，此时存活的对象会从Eden和S1区，向S0区移动。然后Eden和S1区中的垃圾会被清除，这一轮完成之后，这两个区域为空。
* 在程序运行过程中，虽然大多数对象都会很快消亡，但仍然存在一些存活时间较长的对象，对于这些对象，在S0和S1区中反复移动，会造成一定的性能开销，降低GC的效率。因此引入了对象晋升的行为。
* 当对象在新生代的Eden、S0、S1区域之间，每次从一个区域移动到另一个区域时，年龄都会加一，在达到一定的阈值后，如果该对象仍然存活，该对象将会晋升到老年代。
* 如果老年代也被分配完毕后，就会出现MajorGC（即Full GC），由于老年代通常对象比较多，因此标记-整理算法的耗时较长，因此会出现STW现象，因此大多数应用都会尽量减少或着避免出现Full GC的原因。

需要特别说明的是：对于一个新分配的对象，如果Eden区放不下，但是老年代可以放下时，该对象会被直接分配到老年代，不会按照上述从新生代晋升到老年代。

### 4）基于分代算法的垃圾回收器

|  |  |  |  |
| --- | --- | --- | --- |
|  | **垃圾回收器名** | **应用场景** | **描述** |
|  | Serial收集器 | 主要用于年轻代的垃圾回收 | 基于单线程的标记复制算法，主要用于古早时期的单核机器。在执行期间会出现STW。 |
|  | ParNew收集器 | 主要用于年轻代的垃圾回收 | Serial算法的多线程版本，在多核心机器上更加适用。在执行期间，同样会出现STW。 |
|  | Parallel Scavenge收集器 | 主要用于年轻代的垃圾回收 | ParNew的升级版本，主要提供了两个参数：   * -XX:MaxGCPauseMillis：最大垃圾回收停顿时间 * -XX:GCTimeRatio：垃圾回收时间与总时间占比   通过这两个参数，可以调节GC的执行节奏，从而对系统的吞吐率进行调优。 |
|  | Serial Old收集器 | 主要用于老年代的垃圾回收 | 单线程的标记整理算法，由于老年代对象比较多，占用的空间一般较大，如果采用标记复制算法，需要预留50%的空间用于复制，因此空间利用率相当低，而复制过程也会耗时较长，因此采用标记整理算法。此算法为单线程实现，执行过程会出现STW。 |
|  | Parallel Old 收集器 | 主要用于老年代的垃圾回收 | Serial Old的多线程版本。 |
|  | CMS收集器 | 主要用于老年代的垃圾回收 | Concurrent Mark Sweep：在JDK7中广泛使用的垃圾收集器。多线程实现，可以充分利用多核处理器。该算法的执行流程如下：  1）Initial Mark初始标记：主要是从GC Root开始的下一级对象，这个过程会出现STW，但由于与GC Root直接关联的下级对象并不会很多，因此这个过程持续时间很短暂。  2）Concurrent Mark并发标记：根据上一步的结果，继续向下标所有关联的对象，直到每一条链的末尾。这个过程是多线程的，虽然耗时较长，但由于是多线程的，因此其他工作线程并不会阻塞，不会出现STW的现象。  3）Remark再标记：由于第二步进行的过程中，并没有阻塞其他工作线程，可能会产生新的垃圾，因此需要再标记一次。  4）Concurrent Sweep并行清理：这一步采用的是标记清除算法。其他的工作线程还在继续执行，因此不会出现STW。 |

# 三、常见GC问题分析

## 3.1、GC评价指标

### 1）延迟（Latency）

可以理解为最大停顿时间，即垃圾回收过程中，一次STW的最长时间。我们希望这个指标越短越好，在实际生产环境中，我们在一定程度上可以接受STW的频次增加。

### 2）吞吐量（Throughput）

在应用系统的生命周期内，由于GC线程在执行过程中，会占用应用线程的CPU时钟周期和资源。而吞吐量即为应用线程执行的时间占系统总运行时间的百分比。譬如，系统运行了100min，GC线程执行的耗时为1min，则系统的吞吐量为99%。

在各大互联网公司的线上系统，基本上都追求更低的延时，避免一次过长的GC停顿，对用户体验造成较大的负面影响。

## 3.2、理解GC日志

在JVM的启动参数中，如果添加如下参数，则会按照配置在指定的目录下打印GC相关的日志

JVM GC日志相关配置参数

-XX:+PrintGC 输出GC日志

-XX:+PrintGCDetails 输出GC的详细日志

-XX:+PrintGCTimeStamps 输出GC的时间戳（以基准时间的形式）

-XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）

-XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息

-Xloggc:../logs/gc.log 日志文件的输出路径

一般来说，无论线上还是线下环境，上述参数并不需要我们特别关注，只需要切换到目录**/opt/logs/${AppKey}/**下，找到**gc.log**或者**gc.log.0.current**相关的文件，就可以看到JVM的GC日志了。

### 常见的GC原因列表

|  |  |
| --- | --- |
| **GC原因** | **描述** |
| GC (Allocation Failure) | 表示发生的GC的类型为非停顿类型，发生GC的原因是由于分配内存失败导致。 |
| Full GC (Metadata GC Threshold) | 表示发生的GC类型为停顿类型，发生GC的原因是由于元数据区超过阈值（这个阶段不会清理软引用）。 |
| Full GC（Last ditch collection） | 经过Metadata GC Threshold触发的full gc后还是不能满足条件，这个时候会触发再一次的gc cause为Last ditch collection的full gc，这次full gc会清理掉软引用。 |

### GC日志分析

**GC日志**

Plain Text

[GC (Allocation Failure) [PSYoungGen: 65536K->10417K(76288K)] 65536K->10497K(251392K), 0.0092401 secs] [Times: user=0.04 sys=0.01, real=0.00 secs]

GC (Allocation Failure) 表示发生的GC的类型为非停顿类型，发生GC的原因是由于分配内存失败导致。

PSYoungGen 表示GC发生在新生代，且使用的垃圾收集器为 Parallel Scavenge 。

65536K->10417K(76288K) 表示该区域内存一共76288K，回收前已使用65536K，回收后已使用10417K。

65536K->10497K(251392K) 表示堆大小一共251392K，回收前已使用65536K，回收后已使用10417K。

0.0092401 secs 表示该内存区域GC所占用的内存时间。

[Times: user=0.04 sys=0.01, real=0.00 secs] 分别对应 用户态消耗CPU时间，内核态消耗CPU时间 和 操作从开始到结束经过的墙钟时间（包含非运算的等待耗时）

**代码块**

Plain Text

[Full GC (Metadata GC Threshold) [PSYoungGen: 71045K->0K(1305088K)] [ParOldGen: 107466K->85284K(179712K)] 178511K->85284K(1484800K),

[Metaspace: 20805K->20805K(1069056K)], 0.0726323 secs] [Times: user=0.37 sys=0.00, real=0.07 secs]

Full GC (Metadata GC Threshold) 表示发生的GC类型为停顿类型，发生GC的原因是由于元数据区超过阈值。

ParOldGen 表示GC发生在老年代，且使用的垃圾收集器为 Parallel Old。

Metaspace 表示GC发生在元数据区。

## 3.3、GC是因还是果？

我们在线上收到GC指标异常的时候，首先需要判断是GC导致的故障，还是系统其他问题引发GC指标异常。这里主要对前人的一些经验做一个总结：

* **时序分析**：一般来说，先出现异常的指标，是问题的根因的概率更大，通过监控工具，分析各个异常指标的时间点，还原整体的时间线。如果先观察到CPU负载过高，并且其后的指标与该指标存在比较明显的时间差距，那么问题发生的时间线就是：CPU负载高 -> 慢查询增加 -> GC耗时增加 -> Block线程数增多 -> 接口TP线增加。
* **概率分析**：对于一个项目，如果过往慢查询问题较多，那么可以考虑从慢查询入手开始排查。可能的问题发生的过程是这样的：慢查询增加 -> GC耗时增加 -> CPU负载高 -> Block线程数增多 -> 接口TP线增加。
* **实验分析**：通过故障演练、在LiteSet中进行压测等方式对问题现场进行模拟，尝试出发其中的条件，观察是否会发生问题。

在排查过程中，可以综合机器的各项性能指标及代码逻辑进行分析。对于不同的根因，一般来说后续的分析方法是完全不同的。比如，如果是CPU负载高，则可以通过性能诊断工具中的火焰图查看热点代码逻辑、如果慢查询增多，可以看下SQL的执行效率、如果是Block线程数增加，则需要考虑是否是存在锁竞争的情况。

//

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **场景** | **描述** | **现象** | **原因** | **方案** |
|  | 动态扩容引起的空间震荡 | 服务刚启动时，GC次数较多，最大空闲空间很多，但是依然发生GC。  通过监控工具及GC日志可以发现GC Cause一般为Allocation Failure，之后堆内的各个空间的大小会被调整 | 当JVM参数中，-Xms和-Xmx设置的不一致，在初始化时只会按照-Xms的大小来初始化存储空间信息，当空间不够的时，才会向OS申请，这样会触发一次GC。  另外，如果剩余空间较多时，也会进行缩容操作，JVM是通过-XX:MinHeapFreeRatio和-XX:MaxHeapFreeRatio | 尽量将成对出现的空间大小配置参数设置成固定的，比如-Xms 和 -Xmx，-XX:MaxNewSize 和 -XX:NewSize，-XX:MetaSpaceSize 和 -XX:MaxMetaSpaceSize 等，保证这些成对的参数设置的值是一致的，这样JVM可以获得一个稳定的堆。 |
|  | 显式GC问题 | 在没有 动态扩容、Old区域达到回收阈值、MetaSpace空间不足、Young区晋升失败大对象等情况下，却触发了GC。 | 可能是因为在代码中手动调用了System.gc方法。 | 在代码中，规范使用System.gc方法的使用，另外，考虑到Netty等组件大量使用堆外空间，DirectByteBuffer在分配空间的过程中都会显式调用System.gc，希望通过FGC的方式来迫使无用的DirectByteBuffer对象释放掉他们关联的Native Memory。 |
|  | MetaSpace区的OOM | JVM在启动后，或者某个时间点开始，MetaSpace区中，已用空间持续增长，同时每次GC也无法释放，增加MetaSpace空间也没法彻底解决。 | * 类及其元数据的生命周期与类加载器相同，只要类的加载器存活，则在MetaSpace的类元数据也是存活的，不能被回收。 * 一般来说，按照场景一，为了避免频繁的弹性伸缩带来的额外GC消耗，我们会将-XX:MetaSpaceSize 和 -XX:MaxMetaSpaceSize设置为相同的固定值，这就会导致空间不足时无法扩容，然后频繁触发GC，最终导致OOM。 * MetaSpace空间不断增长的主要原因是ClassLoader不断加载新的类，一般主要发生在动态类加载的场景下。 | * 给MetaSpace的使用情况增加监控告警，对于该区域使用率上升的问题 |
|  | 过早晋升 | * 对象晋升年龄较小。 * FGC比较频繁，且经过一次GC后Old区的变化比例较大。   YGC频繁导致系统的吞吐量下降，FGC频繁导致系统停顿时间增加。 | * 新生代（Eden区）过小：该区域过小的直接后果就是Eden区很快就被装满，本应该被回收的对象直接参与到了晋升，没有来得及回收的对象增加了回收的代价，所以导致YGC时间增加，同时也无法快速释放空间，导致YGC的次数增加。 * 分配速度过大：可以观察Mutator的分配速度是否有明显的波动，如果有，则可以观察网卡流量、存储类中间件的慢查询日志等信息，观察是否有大量的数据被加载到内存中。 | * 增加新生代的空间，一般老年代的大小应该为活跃对象的2～3倍左右。剩下的空间都可以分配给新生代。 * 分配速度过大：   + 偶发过大：通过内存分析工具找到问题代码，从业务逻辑上做优化。   + 一直较大：如果垃圾收集器已经无法满足Mutator的需求，则这个时候要么对空间进行扩容，要么调整垃圾收集器的类型。 |
|  | CMS Old GC频繁 | 老年代频繁GC，但是每次耗时不长，最大STW也在可接受范围内，由于GC较为频繁导致系统吞吐量下降。 | 一般来说，这种现象基本上都是一次YGC之后，负责处理CMS GC的一个后台线程不断轮询，调用shouldConcurrentCollect()方法做一次检测，判断是否达到了回收条件，如果达到条件，则会进行一次GC。间隔周期为-XX:CMSWaitDuration，默认为2s。  这种现象一般比较常见的一个业务场景就是，与过早晋升不同的是，有一些对象确实存活了一段时间，存活时间超过了TP9999，但是又达不到长期存活。 | * 做内存的Dump diff，在CMS GC发生前后各Dump一次。 * 观察不可达对象的变化情况。 * 结合代码逻辑分析代码问题，可以采用控制变量法。 |
|  | 单次CMS Old GC耗时长 | CMS GC 单次STW最大超过1000ms，在某些场景下，会导致系统雪崩。在生产环境中，我们应该尽量避免出现这类问题。 | STW主要发生在Init Mark和Final Remark这两个阶段，这也是导致Old GC最多的原因。一般来说，Init Mark的执行速度较快。一般都是Final Remark阶段耗时长导致的，通过GC日志可以看到具体耗时长的阶段。 | * 在GC动作被触发前，对JVM内存进行Dump。 * FinalReference：需要观察java.lang.ref.Finalizer对象的支配树，然后找到内存泄漏的来源。 * Scrub Symbol table表示清理元数据符号引用的耗时，观察MetaSpace区历史使用情况曲线，以及每次GC前后的回收情况，如果没有使用动态类加载或者DSL的话，MetaSpace空间的使用情况不会有变化。 * 对于G1，有同样的FinalReference问题，可以在GC日志中观察Ref Proc，处理的方法与CMS类似 |
|  | 内存碎片/收集器退化 | 并发的CMS GC算法退化为单线程串行GC模式，STW时间有时候会长达十几秒。退化后的单线程串行GC算法主要有两种：   * 带压缩动作的算法MCS，采用标记-清理-压缩的方式。 * 不带压缩动作的算法。 | * **晋升失败**：在进行YoungGC的时候，Survivor放不下，对象只能放入老年代，但是此时老年代也放不下。就会出现晋升失败的问题。引起晋升失败的原因主要是由于老年代内存碎片导致的。内存碎片主要会导致两个问题：   + **空间分配效率较低**：如果内存空间是连续的，则在JVM中可以采用Pointer Bumping方式来分配内存，对于有大量内存碎片的空闲链表则需要逐个访问freelist中的项来查找可以存放新建对象的地址。   + **空间利用效率较低**：即使老年代空间足够，由于该区域碎片化，导致空间不连续，无法存放新对象，所以会出现晋升失败的问题。 * **增量收集担保失败**：在分配内存失败后，会判断统计得到的YoungGC晋升到老年代的平均大小，以及当前新生代已经使用的大小，是否大于老年代的剩余空间，只要剩余空间比两者任意一个大，则CMS认为晋升就是安全的；否则就是不安全的，会直接进行FGC。 * **显式GC**：参见场景2 * **并发模式失败**：在GC日志中，会看到Concurrent Mode Failure。这是由于并发CMS GC在执行的过程中，又有YoungGC晋升的对象需要放入老年代，而此时老年代的空间不足。出现这种情况的原因是CMS无法处理浮动垃圾，这些浮动垃圾不在清理标记的范围内，无法在本次GC中清除掉。这种情况主要发生在老年代回收阈值过高，预留的内存空间不够的情况下。 | * **内存碎片**：通过配置-XX:UseCMSCompactAtFullCollection=true参数来控制FGC是否进行空间整理，并通过-XX: CMSFullGCsBeforeCompaction=${}参数来控制多少次FGC之后进行一次压缩。 * **增量收集**：通过调节-XX:CMSInitiatingOccupancyFraction参数来降低CMS GC的阈值，让CMS GC动作尽早执行，保证有足够的连续空间，也减小老年代空间的使用大小。另外，需要使用-XX:+UseCMSInitiatingOccupancyOnly来配合使用，否则JVM仅在第一次使用设定值，后续会自动调整。 * **浮动垃圾**：核心是控制每次晋升对象的大小，或者是缩短每次CMS GC的时间，并利用参数-XX:+CMSScavengeBeforeRemark提前触发一次YoungGC，防止后续晋升过多对象。   在编码习惯上，也要避免出现需要连续空间的大对象的产生，比如过长的字符串、存放文件或者序列化的Byte数组等。 |
|  | 堆外内存OOM | 内存使用率不断上升，内存近乎耗尽之后，开始使用SWAP内存，同时出现GC时间徒增，线程被Block的现象，通过top命令查看Java进程的RES（常驻内存）甚至超出了-Xmx设置的大小。 | * 通过UnSafe#allocateMemory，ByteBuffer#allocateDirect主动申请了堆外内存，并没有释放，比如NIO、Netty等组件。 * 代码中有通过JNI调用Native Code申请的内存并没有被释放。 | * 主动申请堆外内存并未释放：走查代码，找到代码中是如何使用对外内存的，NIO或者是Netty，可以获取对应组件中的计数器字段，可以在代码中对该字段进行打点，从而准确监控到这部分的堆外内存的使用情况。 * 通过JNI调用Native Code申请的内存并未释放：采用Google perftools + Btrace工具定位问题代码。 |
|  | JNI引发的内存问题 | 在GC日志中，出现GC Cause为GCLocker Initiated GC | JNI（Java Native Inteface）意为Java本地调用，允许Java代码与其他语言写的代码进行交互。JNI如果需要获取JVM中的变量数据的时候，主要有两种方式：   1. 拷贝传递 2. 共享指针   由于Native代码直接使用了JVM堆区的指针，如果此时触发GC，则会导致数据异常。因此，在JNI调用的时候，会禁止GC，并禁止其他线程进入JNI临界区，直到最后一个线程从临界区退出后，触发一次GC。  GC Locker导致的负面影响：   * 当新生代空间不足导致的GC，由于此时无法GC，会导致对象被直接分配到老年代。 * 如果老年代也没有足够的空间，此时则会等待锁释放，导致线程阻塞。 * JDK的Bug（[JDK-8048556](https://bugs.openjdk.java.net/browse/JDK-8048556)），重复YGC | * 如果项目中使用了JNI特性，则可以添加-XX+PrintJNIGCStalls参数，打印出发生JNI调用时的线程进一步分析，找到引发问题的JNI调用。 * JNI调用需要谨慎使用，使用不当不但不能提升性能，反而可能造成性能及系统问题。 |

**四、GC常用工具**

|  |  |  |
| --- | --- | --- |
|  | **工具名** | **作用&使用方法** |
|  | jps | jdk自带命令。查看JVM中运行的进程状态信息。  **代码块**  Plain Text  jps [options] [hostid] |
|  | jstat | jdk自带命令。可以观察到classloader，compiler，gc相关信息。可以时时监控资源和性能 。  jstat -gc -t ${pid} ${time} |
|  | jmap | jdk自带命令。jmap是一个多功能的命令，查看JVM内存使用情况。它可以生成 java 程序的 dump 文件， 也可以查看堆内对象示例的统计信息、查看 ClassLoader 的信息以及 finalizer 队列。  **代码块**  Plain Text  jmap -dump:live,format=b,file=myjmapfile.txt ${pid} |
|  | GCViewer | GCViewer是一个GC日志的可视化分析工具。由于JVM的GC日志每秒钟都会生产大量的日志，这种大量的文本形式的日志对我们分析GC问题，调优JVM性能造成很大的不便，因此可以通过这类可视化分析工具可以大大提高我们分析GC日志的效率，从而协助我们分析并找到问题根因。  GCViewer工具主要有以下几个标签： |

//

**六、GC编程建议**

本文花了大量的篇幅介绍了JVM的GC的原理、常见的垃圾收集器、GC问题的一般排查方案以及典型问题的优化方案。对于这些原理性的知识，是我们在遇到具体问题时，排查解决的基础。而典型问题的优化方案，则给了我们一些解决问题的思路和指引。但无论如何，这些排查的手段，都是在问题发生之后的补救措施，如何在问题发生前就尽力避免问题的发生呢？这就需要我们在编码的过程中遵守一系列规范，防患于未然。

1. **【建议】对外接口的出参包含列表或者Map类型的属性的，可以采用分页的形式返回。**

说明：这些类型的属性可能包含大量的元素，导致内存开销过大，造成GC，在大客场景下，这种情况更加常见。另外，基于Thrift的RPC服务对接口的返回最大字节数有限制，默认为16384000（[RPC框架-Thrift](https://km.sankuai.com/page/164817650)），超过限制请求将失败。

正例：对外接口的出参包含列表或者Map类型属性的，经过评估可能存在包含大量元素的场景时，需要采用分页的形式进行返回，对可能的大对象进行拆分。

反例：对外接口的出参包含列表或者Map类型的属性，不做任何考虑，直接返回。

1. **【强制】对外接口的入参包含列表或者Map类型的属性的，必须限制长度。**

说明：这些类型的属性的入参数量如果不加以控制，一旦客户端传入大量对象，看你会造成服务端内存开销过大。对于部分批量查询的场景，可能简介造成查询结果对象过多，对服务端稳定性造成很大隐患。

正例：在与客户端事先约定的前提下，所有对外接口需要对列表或者Map类型的入参或者属性的长度做限制。对于不合法的参数应直接返回错误。

反例：1）未与客户端事先约定入参最大长度，或者未对入参最大长度做打点统计，直接对入参长度进行限制。2）不对入参长度做任何限制措施。

1. **【建议】在存储层，需要对返回结果的长度做限制。**

说明：在项目中，涉及到对存储层访问的，返回结果应该增加长度限制，包括但不限于MySQL、ES、Redis、Cellar，防止存储层返回大量结果，造成服务端内存开销过大。

正例：在执行DB数据查询、使用ES搜索，及Redis缓存等时，增加limit关键字，对返回结果的数量增加限制。

反例：编码时，对存储层返回长度不做任何评估，也不加以限制，直接进行后续的业务逻辑。

1. **【强制】代码中配置的扫描的包路径需要按需配置，防止配置范围过大。**

说明：代码中配置扫描包路径时禁止配置“com.”、“org.”、“com.sankuai.”，防止出现包扫描返回过大的情况，造成服务启动过慢，或者启动时内存开销过大的问题。在使用通配符时，也需要明确具体含义，确定扫描范围是否符合预期。

正例：按需配置扫描包的包路径。

反例：随意配置，或者不加思索地直接从别的项目里拷贝包路径。

1. **【建议】谨慎构造、序列化、反序列化大对象**

说明：在代码逻辑中，尽量减少构造大对象，尽量减少对大对象的序列化操作。

正例：针对可能出现的大对象，分页或者按照领域进行拆分。

反例：随意构造大对象、超长字符串等。

1. **【强制】禁止手动GC**

说明：在代码中禁止调用System.gc方法手动GC。

正例：在提交的代码中，无System.gc方法调用。

反例：在代码中随意使用System.gc方法。

1. **【强制】-Xms和-Xmx设置的值保持一致。**

说明：在JVM启动参数中，如果-Xms和-Xmx参数配置的不一致，当JVM内存空间不足时，会像操作系统申请内存，从而触发一次GC，导致JVM空间震荡。

正例：-Xms和-Xmx参数配置的值保持完全一致。

反例：-Xms和-Xmx参数配置的值不一致。

1. **【建议】主动GC**

说明：对于个别对GC动作比较敏感的系统，通过定时摘掉服务的流量，手动出发一次MajorGC，从而减少JVM自动GC带来的停顿及不可控。

正例：在服务保护平台（Rhino）上，配置主动GC任务。

反例：在代码中随意调用System.gc方法。

1. **【建议】完善相关监控能力**

说明：大对象、大入参、大返回、长耗时、流量突增等场景，导致服务容量不足，从而导致JVM GC指标异常。

正例：针对大对象、大入参、大返回、长耗时、流量突增等场景，完善相关监控、限流、弹性扩容等能力，避免服务被打崩。

反例：系统服务仅有基本的监控能力，无针对上述场景配置专门的监控告警、限流降级、弹性扩容能力。甚至无基本的监控能力，系统仿佛在裸奔。