图文 057、案例实战：每秒十万QPS的社交APP 如何优化GC性能提升3倍?

详情 评论

案例实战：每秒十万QPS的社交APP，

如何优化GC性能提升3倍?

狸猫技术窝专栏上新，基于**真实订单系统**的消息中间件（mq）实战，重磅推荐：

未来3个月，我的好朋友原子弹大侠将带你一起，全程实战，360度死磕MQ

（点击下方蓝字进行试听）

从 0 开始带你成为消息中间件实战高手

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情在评论区留言提问，我都会逐一答疑

（**ps**：评论区还精选了一些小伙伴对**专栏每日思考题的作答**，有的答案真的非常好！大家可以通过看别人的思路，启发一下自己，从而加深理解）

如何加群：购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**。

（群里有不少**一二线互联网大厂的助教**，大家可以一起讨论交流各种技术）

具体**加群方式**请参见文末。

（**注**：以前通过其他专栏加过群的同学就不要重复加了）

写在前面：

本来打算今天更新两篇文章，但是考虑到大家的学习效果和学习节奏，最后决定今天还是发一篇文章。

本周文章，将依次顺延一天，即从原来的周一到周五发文，改成周二到周六发文

好，通知结束，正文开始：

1、引子

这篇文章开始，我们会开始用一系列真实的生产案例给大家还原出来各种各样不同的JVM优化场景

力求让大家在不同的业务背景下，对不同的原因产生的JVM性能问题进行分析和处理，进而积累出来大量不同场景下的JVM性能优化经验。

所有的JVM优化案例都是基于之前几十篇文章教给大家的核心原理以及优化手段来展开的，因此大家可以认为，之前几十篇文章学习完过后，你就已经有能力在生产系统上手进行JVM的性能分析以及优化了。

只不过大家如果再经过几十个真实的生产案例的锤炼，就能够有能力对未来自己可能遇到的各种不同的情况下的JVM生产问题进行分析和处理。

因为这些案例都是真实的生产案例，所以有的案例我们会用一些模拟代码还原出来当时的故障场景，并且带着大家用各种工具来进行分析，然后进行优化。

但是有的案例我们很难用模拟代码还原出来，此时还会延续采用一步一图的方式，尽量用大量的手绘图让大家理解JVM故障的产生背景、发生原因以及优化的手段。

跟之前一样，我们开始先用两三个较为简单的案例引入，让大家找找感觉，接着再切入比较复杂的案例。

2、案例背景

本案例的背景是一个有高峰期每秒十万QPS的社交APP。这是我曾经帮助一个朋友的公司处理过的一个JVM优化的案例。

大家都知道，其实现在社交APP有很多种，不光是大家熟悉的微信、QQ之类的，还有很多细分领域的明星社交APP，诸如陌生人社交，基于地理位置的社交，等等。

其实很多明星创业社交APP产品，也有每日数百万的日活用户，尤其在晚上高峰期的时候，APP的QPS也是很高的。

附带一句，可能有的同学不知道QPS是什么，其实英文全称就是“Query Per Second”，也就是每秒钟的查询数量，大致可以理解为是APP每秒钟的访问数量。

对于这个APP而言，流量最大的是哪个功能模块？

不知道大家有没有玩儿过陌生人社交类的APP，在这种APP中操作最多的就是浏览某个陌生人的个人页面。

一般这类APP都会通过各种方式来给你推荐一些周边的陌生人，然后你可能会看到一些感兴趣的人，就会进入他/她的个人主页去看一看。个人主页里可能就包含了那个人的一些自我介绍，照片之类的东西。

所以这类APP，在晚上高峰期，流量最大的一个模块，其实就是个人主页模块

会有大量的活跃用户在一个集中的时间段内频繁的访问各种个人主页数据，而且这类数据的量还通常很大，要包含很多的信息，通常一个个人主页的数据甚至可能有几MB。

随便给大家举个例子，一个个人主页里，可能有这个人每天发的一些心情、感悟之类的东西

那么一旦要把这个个人主页加载出来，必然会加载出来这个人最近N多天发的一些心情感悟之类的文字，这个文字的量还是比较多的。

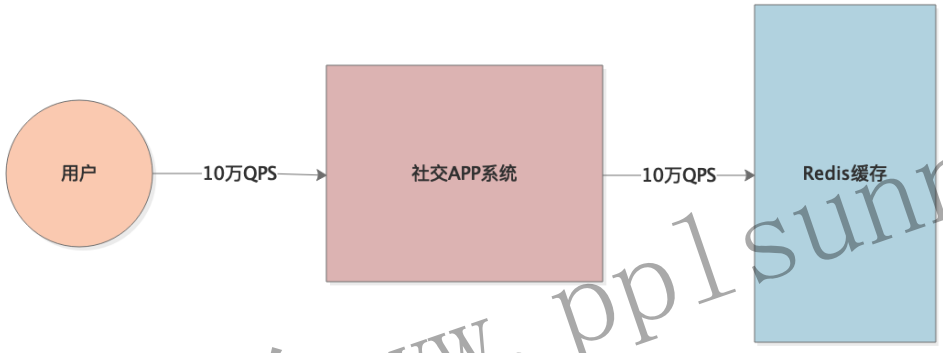
所以我们大致可以认为，一次个人主页的查询，就会加载出来比如5MB的数据。

而且一般在高峰期内，有可能一些活跃用户他可能会连续不断的去点击他感兴趣的人的个人主页，比如连续1个小时都在不停的点击。

所以其实这类社交APP他的高峰期QPS时很高的。在当时的场景中，这个社交APP流量最大的个人主页模块高峰期最多每秒会有10万+的QPS。

当然在底层存储中，这些个人主页数据一定是基于缓存来存放的，也就是基于Redis缓存来查询这些个人主页数据。

所以，综合我们分析出来的这个背景，我们在这里可以用下面一幅图来让大家了解一下这个社交APP的一个情况。

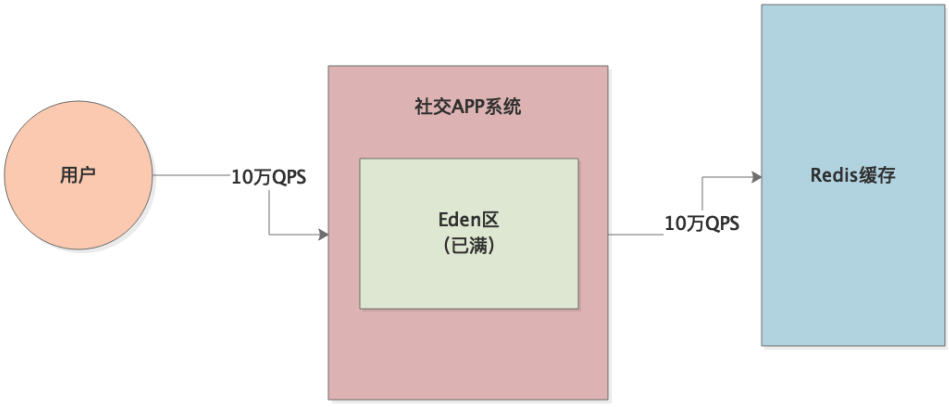


3、高并发查询导致对象快速进入老年代

之前已经用大量的图示给大家分析过了JVM的运行原理了，所以在这个案例中就不再给大家描述的过于细节。

这里就直接给大家分析一下当时案例发生的一个场景，因为这个社交APP的日活用户涨的很快，所以导致他的高峰期QPS很快就飙升到了10万。

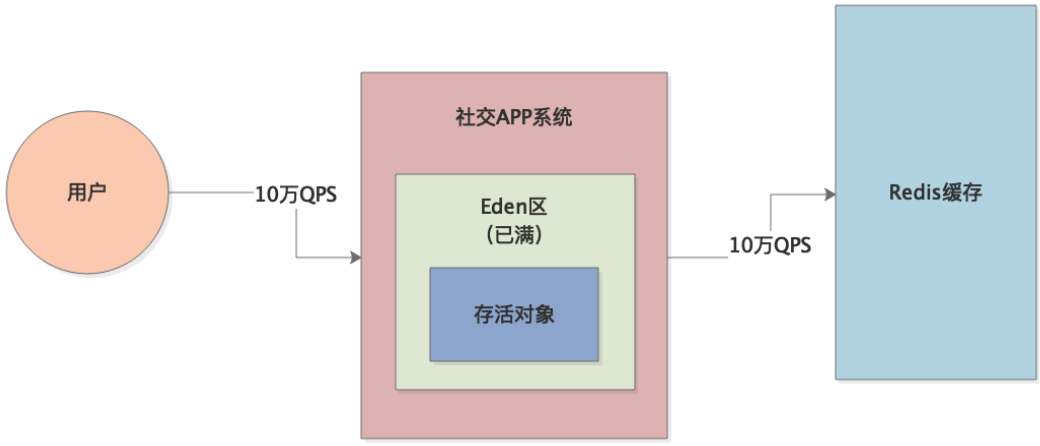
正是因为每秒并发量太高，这也直接导致了这个系统在高峰期的时候，年轻代的Eden区会迅速的被填满，并且频繁的触发Young GC，如下图所示。



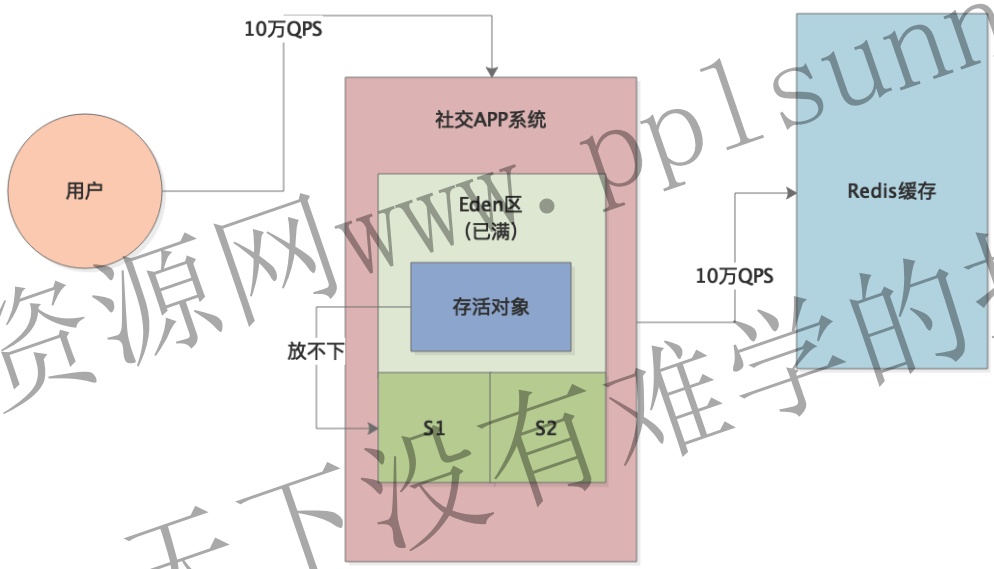
而且每次在Young GC的时候，实际上还有很多请求是没处理完毕的，没办法，因为每秒请求量太多了，所以在你触发Young GC的时候，就这一瞬间，必然有很多请求是还没处理完毕的。

这就导致Eden区中其实每次触发Young GC的时候，都有很多对象是需要存活下来的

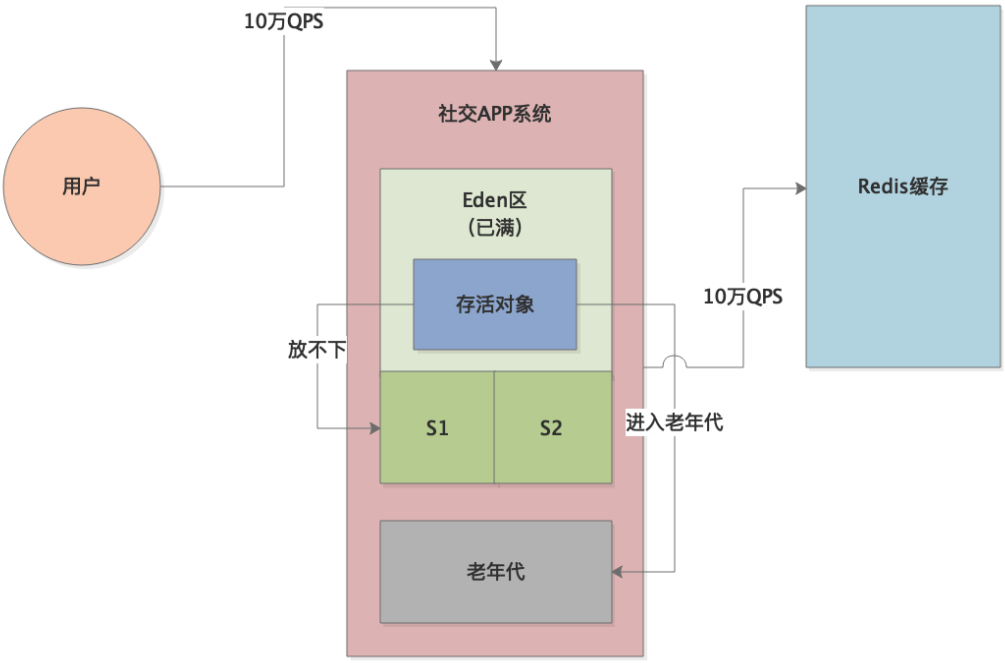
如下图所示：



因此在高峰期的时候，其实经常会出现Young GC过后存活对象较多，在Survivor区中放不下的问题，如下图所示。



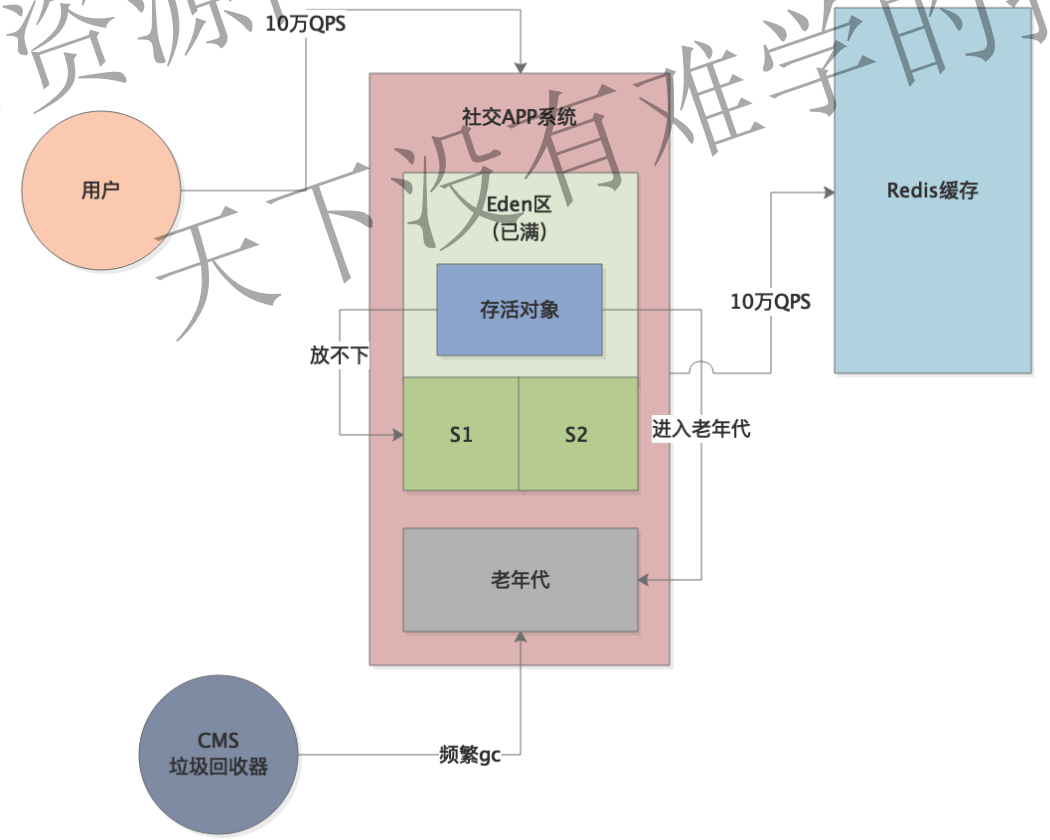
所以此时必然会导致大量的对象快速的进入老年代中，如下图所示。



4、老年代必然会触发频繁GC

其实根据之前学过的知识，大家都清楚了一点，那就是一旦在高并发场景下Young GC后存活对象过多，导致对象快速进入老年代，必然会频繁触发老年代的GC，对老年代进行垃圾回收。

所以在上述社交APP高峰期高并发场景下，必然会导致个人主页服务对应的JVM频繁的发生老年代的GC，如下图所示。



5、优化前的线上系统JVM参数

其实大家都知道，针对上述场景，最核心的优化点，主要应该是增加机器，尽量让每台机器承载更少的并发请求，减轻压力。

同时，给年轻代的Survivor区域更大的内存空间，让每次Young GC后的存活对象务必停留在Survior中，别进入老年代。

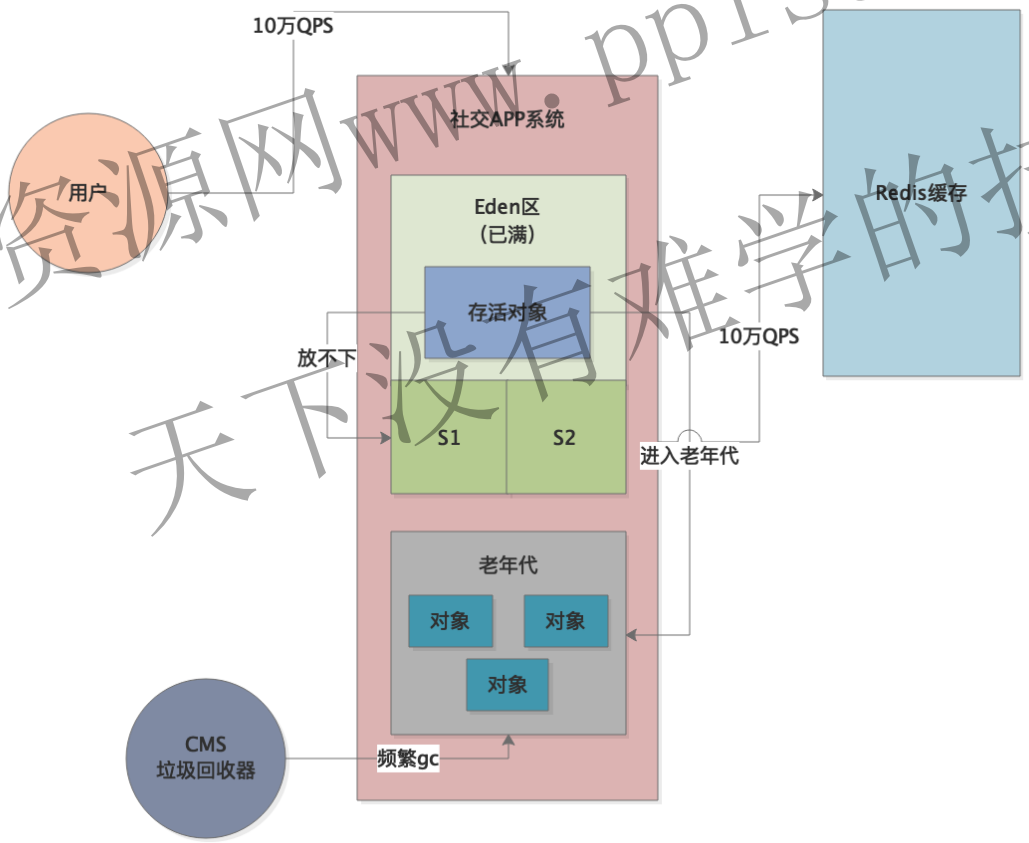
但是在这里我们先不考虑上述优化，在优化前的线上系统中，对JVM有两个比较关键的参数大家可以看一下：

-XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=5

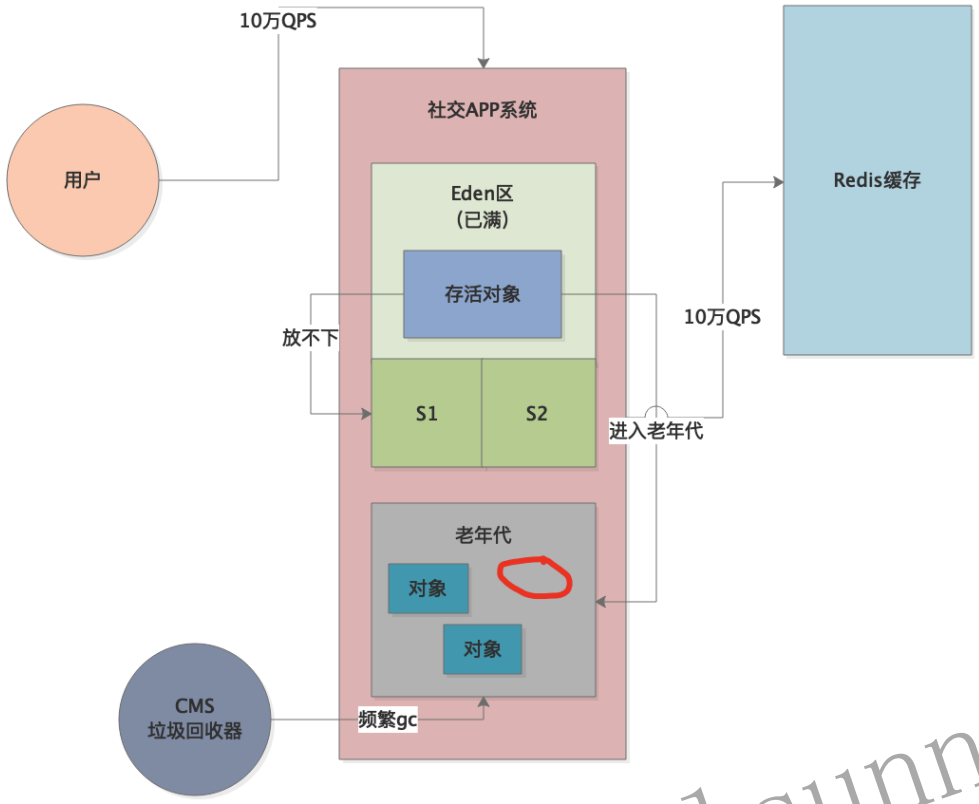
大家可以看到这个是什么意思，其实非常明显，我们之前讲过这两个参数的含义。CMS垃圾回收器默认是采用标记-清理算法，所以是会造成大量的内存碎片的。

什么叫内存碎片？我们再来看下图回顾一下

比如现在老年代内存里有一些垃圾对象。



然后CMS垃圾回收器一次垃圾回收过后，回收掉了一些垃圾对象，此时可能内存里看起来跟下面这样：



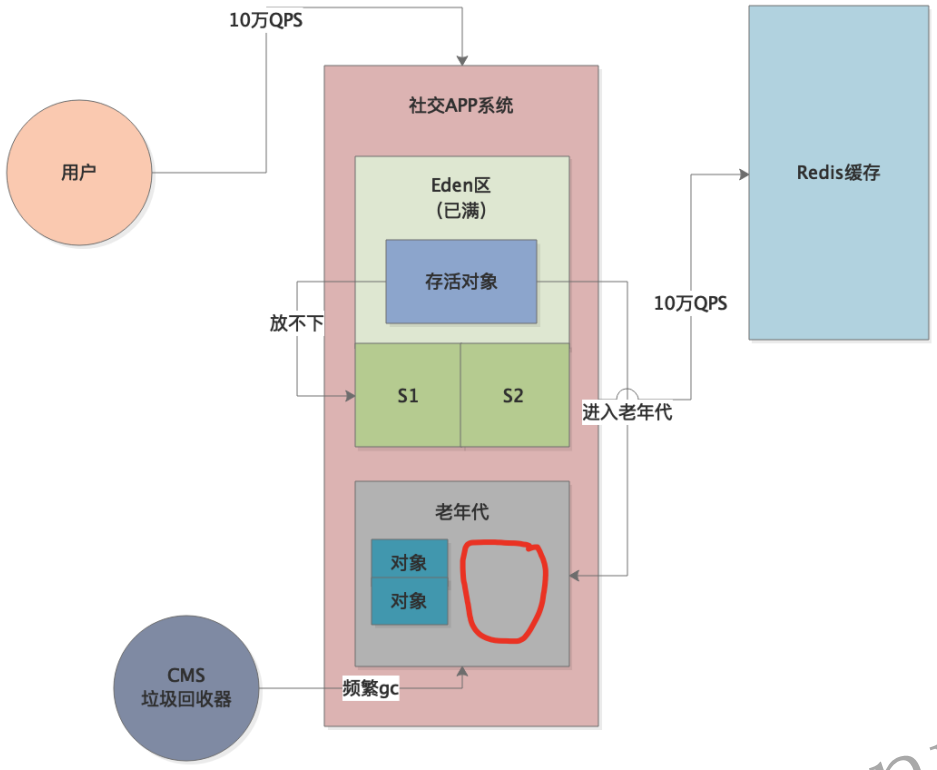
大家注意上面那个红圈的地方，因为回收掉了一个对象，所以那里出现了一个内存碎片

虽然这里是空白内存，但是假如此时你要是要分配一个对象比较大，没法再上面红圈处放进去呢？那么红圈的那个内存碎片不就没任何意义了？

所以CMS正常垃圾回收，因为使用标记-清理算法，所以必然导致大量的内存碎片。

所以“-XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=5”两个参数的含义，就是在5次Full GC之后会触发一次Compaction操作，也就是压缩操作

这个操作会把存活对象放到紧邻在一起，避免大量的内存碎片，如下图所示。



大家看下上图，是不是发现两个存活对象被挤压在一起了？然后红圈地方是不是多出来一大块连续可用的内存空间？不再是之前的一小片内存碎片了吧？

6、频繁Full GC导致的大量内存碎片

但是大家现在要明白一点，上述两个参数“-XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=5”是设置的5次Full GC之后才会进行一次压缩操作，解决内存碎片的问题，空出来大片的连续可用内存空间。

所以这就直接导致在这5次Full GC的过程中，每一次Full GC之后都会产生大量的内存碎片。

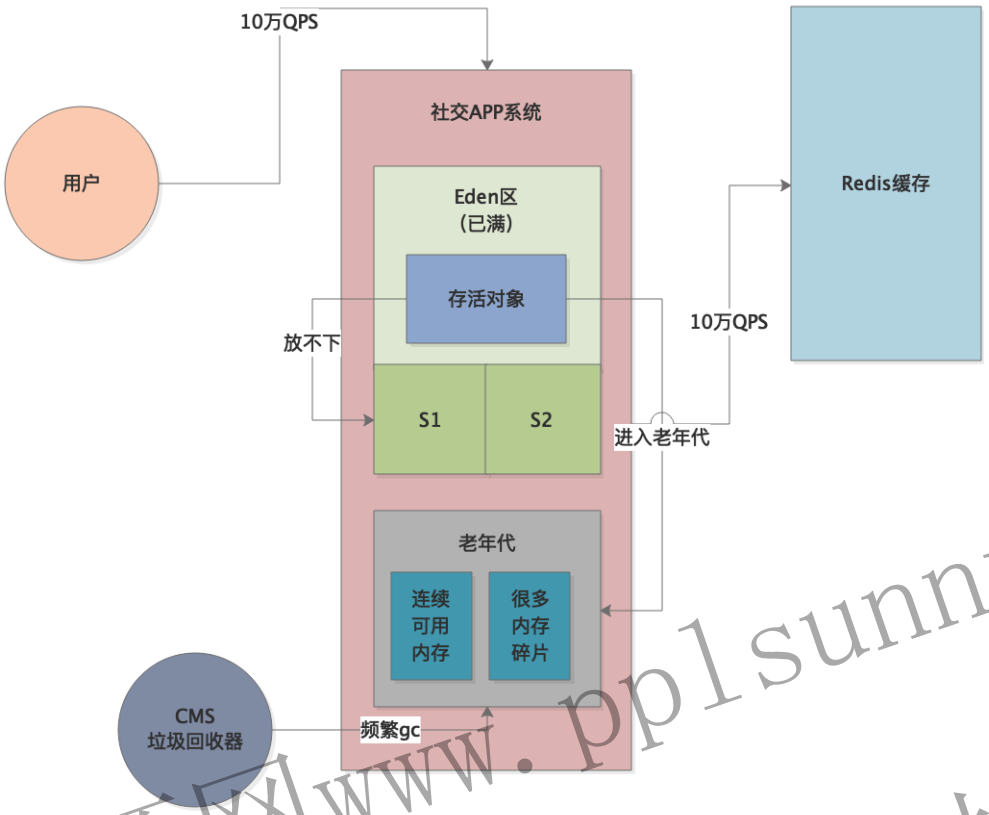
大量的内存碎片会导致很多问题，其中一个问题，就是提高Full GC的频率。

为什么呢？因为大家之前应该还记得我们讲过，触发老年代GC的一个非常重要的条件，就是Young GC后的存活对象无法放入Survivior，就要放入老年代。

但是此时老年代假设也没足够内存放这些对象了，就必须触发Full GC了。

所以大家考虑一个场景，假设如下图所示，一次Full GC过后，老年代中有一部分内存里都是大量的内存碎片，没法放入完整的一些大对象了，只有部分内存是连续可用的内存空间。

如下图所示：



这个时候，随着大量对象快速进入老年代，会导致一旦老年代的那块连续可用内存满了，此时很多内存碎片是无法放入更多对象的，就会立马触发下一次Full GC。

比如老年代有2G的内存，其中1.5G是连续可用内存，0.5G是很多内存碎片。

本来老年代如果都是连续空内存的话，那么可能可以对象占用到将近2G才会触发Full GC。

结果现在就是对象占用到了1.5G就需要触发Full GC了，剩下0.5G是没法放任何对象的。

所以这就会导致随着一次一次Full GC导致老年代产生更多的内存碎片，连续可用内存越来越少，触发下一次FUll GC的速度就会越快。

直到几次Full GC之后，才会触发一次Compaction操作去整理内存碎片。

7、这个案例如何进行优化?

其实对这个案例进行优化，非常的简单，无法就是用之前讲过的jstat分析一下各个机器上的jvm的运行状况，判断出来每次Young GC后存活对象有多少，然后就是增加Survivor区的内存，避免对象快速进入老年代。

另外一个，在当时对那个系统优化之后，增加了年轻代和Survivor区的大小，但还是会慢慢的有对象进入老年代里，毕竟系统负载很高，彻底让对象不进入老年代也很难做到。所以当时调优过后每小时还是会有一次Full GC。

所以当时第二个参数的优化就是针对CMS内存碎片问题的

在降低了Full GC频率之后，务必设置如下参数“-XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0”，每次Full GC后都整理一下内存碎片。

否则如果每次Full GC过后，都造成老年代里很多内存碎片，那么必然导致下一次Full GC更快到来，因为内存碎片会导致老年代可用内存变少。

也许第一次Full GC是一小时才有，第二次Full GC也许是40分钟之后，第三次Full GC可能就是20分钟之后，要是不解决CMS内存碎片问题，必然导致Full GC慢慢变得越来越频繁。

8、今日思考题

今天给大家留一个小的思考题：去看看你们线上系统的JVM参数，如果用的是CMS垃圾回收器的话，压缩这块是如何设置的？

考虑一下如果不是每次FUll GC后都压缩一次解决内存碎片，会对你们的系统有什么影响？

End
