# 保单中心高负载问题分析过程

## 背景：

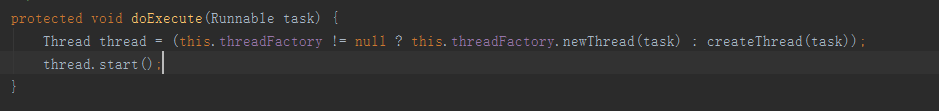
1. 强依赖第三方系统，不稳定导致高耗时：

在以往的报单中心响应高情况中，主要检查的是拉车接口的耗时，因为保单中心强依赖于拉车接口，因此在强依赖第三方的情况下，当系统出现响应慢的问题时，需要留意强依赖系统的耗时情况。

1. 减低对第三方系统的依赖，数据异构方案实施：

后面经过改造，将拉车数据缓存到redis，并通过订阅车中心的车信息变更消息通道更新缓存数据，因此在后续的响应高的情况中，都没再发生是因为拉车接口耗时高导致的问题；但在一段时间里，并没有订阅消息通道，因此通过时间间隔异步控制刷新本地缓存，如请求过了10分钟，再去通过异步队列更新本地缓存，从而减少一定的数据不一致状态。

这里异步使用的是@Async注解，需要注意的是，默认情况下，Spring会用SimpleAsyncTaskExecutor，其每次都会创建一个线程，而且会存在任务丢失的情况。



因此后面采用自定义线程池以及disruptor框架去处理异步任务，并对陆续对之前的如保单全量等数据异构异步任务进行了改造。

此时，系统的耗时也降下来了，相对也比较稳定了，但是随着业务的新增，以及保单数据的新增，这时系统时不时出现高耗时报警，但看监控并没有多大变化的情况，并且越来越频繁。后来演变成了基本每个工作日都会发生的情况，出现了高耗时、高负载。

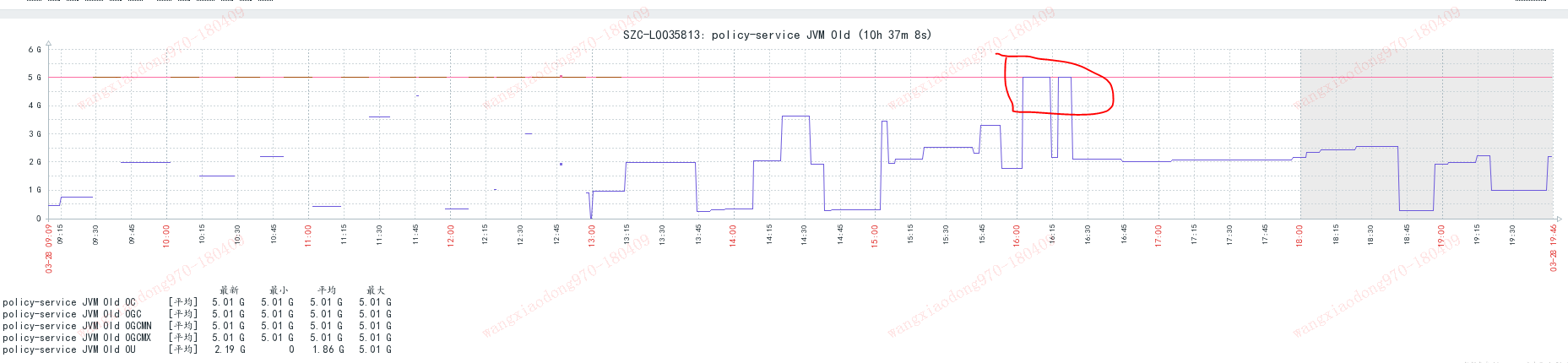
## 高负载问题分析：

1. 从日志分析，找高负载：

刚开始由于没有jvm的权限，因此对于系统的运行情况，也就只能从日志去分析。此时通过日志分析发现每次高负载发生时，zk也伴随着出现连接不上的问题，此时便怀疑是zk的问题，因此便将zk停了，系统也稳定了下来，但隔天发现问题又发生了，因此便将zk问题排除了。

后来从系统可能导致高负载的业务功能分析，定位到系统的图片旋转业务，因此便提出将图片旋转放到前端去实现，但效果并没有很明显。

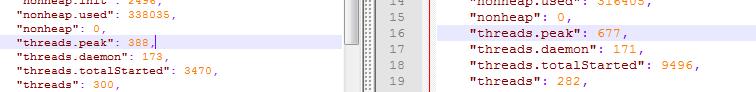
期间也发现存在大文件上传的问题，同时也发现了gc问题GC overhead limit exceeded也就是存在gc回收效益低导致的内存溢出问题，结合监控平台也可以发行堆内存经常爆满。



因此前端再做了限制，如果上传图片超出限制，便不让上传。但后面发现还是存在部分大文件上传的问题。不过此时量已经少了很多。但问题还是存在，并没有解决。

1. 增加端点监控和自定义线程监控：

由于生产没有权限，对于系统的运行情况也无法得知，因此基于保单中心采用springboot框架，将其配有的端点监控进行了集成，对比在高负载时和非高负载时，系统的线程存在比较大的差异。



因此怀疑系统存在执行慢的异步任务，此时运维也给了生产的jvm命令权限，也发现了垃圾收集器对老年代使用的是Serial Old（一个只能在单CPU服务器情况下才能比较有高效益的垃圾收集器）以及开始实现自定义线程池监控，进行异步任务数的统计和耗时情况记录。

1. jvm检查以及自定义线程池监控：

有了生产权限后，在高负载发生时，采用了如下的步骤：

1. 通过top -Hp pid找出高负载的子线程id
2. 再通过printf "%x\n" pid 获得十六进制的子线程id，因为jvm记录的是十六进制的子线程id
3. 再通过jstack pid | grep xpid 获取目前高负载的线程信息。

但是通过此步骤发现，高负载的线程就是老年代的垃圾收集器线程。因此还是无法准确定位问题。

再将jstack命令输出的信息保存到本地，检查自定义线程池监控的信息，通过jmap -dump:live,format=b,file=./dump.bin pid 将jvm的运行情况dump下来(此时文件可能超4G，因此需要拆分对文件拆分才能下载下来)，通过jmap -histo:live pid 将jvm此时的实例信息保存下来，再进行分析。

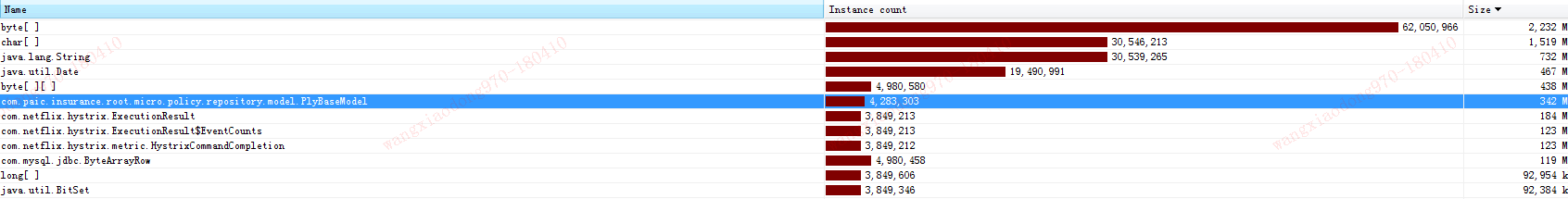
通过自定义线程池发现拉车的异步任务等待数非常大，达到了几十万的情况，大量的异步任务等待，占用了一定的堆内存且会存在老年代中。



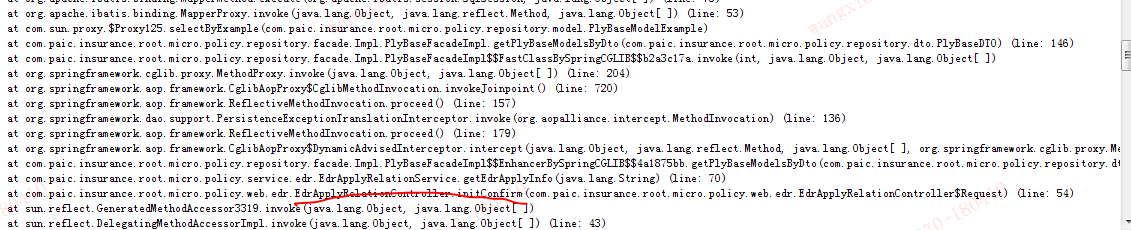
由于拉车此时已经接入了变更消息通道，因此可以将该异步移除。

检查dump和实例信息，发现保单全量的实例同样达到了几十万的情况，由于保单全量的数据信息比较多，一个保单全量大约3KB,检查数据库也发现，保单全量的数据在2月份后增量也比较大，从2月份的单表2千多，到3月份的最高峰9千多,因此也对保单全量进行缩减，只保留系统需要的字段，其他字段都移除；

经过整改后，下个版本，依旧出现了高负载的情况，此时系统能挺过去，一段时间后会降下来，在高负载时，再次检查gc、线程快照、堆存储快照以及实例信息，发现批改中的保单信息，实例达到了4百多万的情况。且基本都是一个业务代码，ByteArrayRow也达到4百多万的实例，说明数据库操作也存在一定的问题。

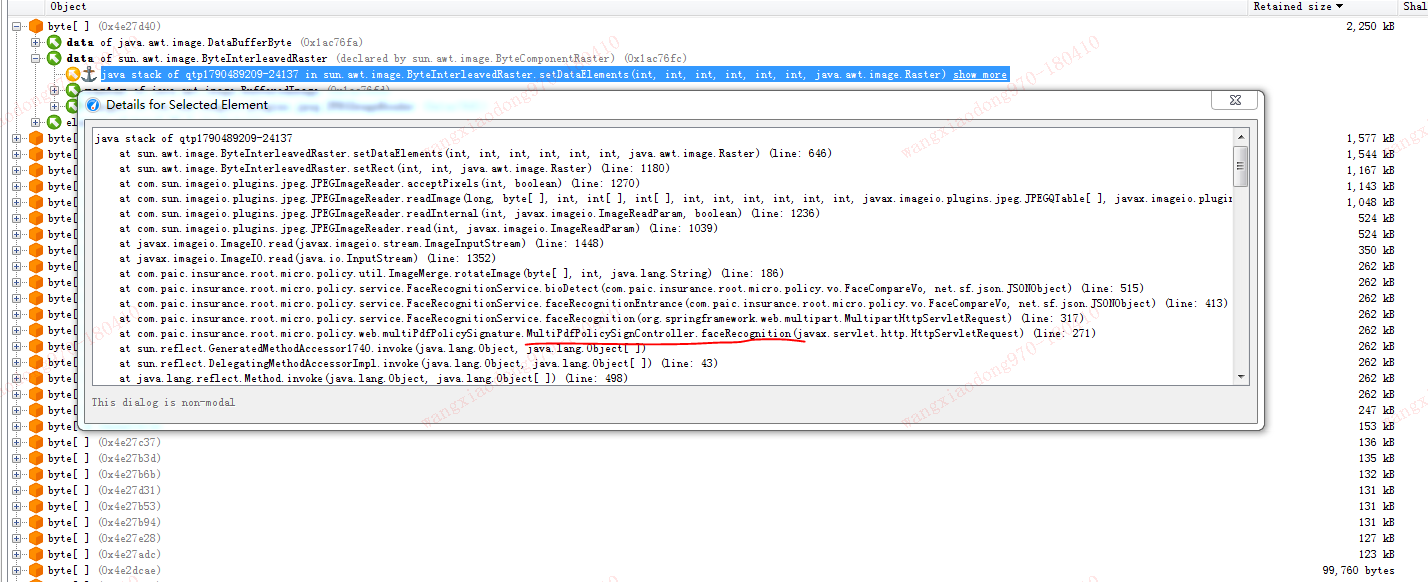


批改提核初始化接口



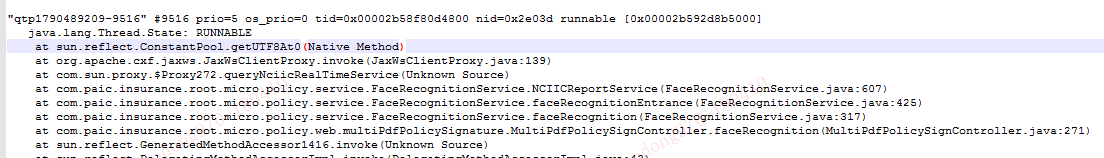
此时定位这块代码存在问题，但是这块代码已经很久没有修改过。

另外再分析了byte的信息，发现此时人脸识别上传的图片也比较大，很多为系统允许通过的2m，也就是图片上传还依旧存在大图片的情况，其中也有很多几百kb的图片。

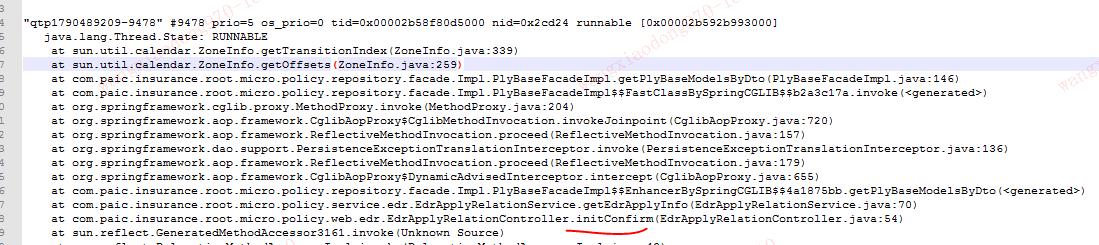


再次检查jstack信息，发现高负载时也伴随着图片上传。（高负载时主要留意RUNNABLE的线程）

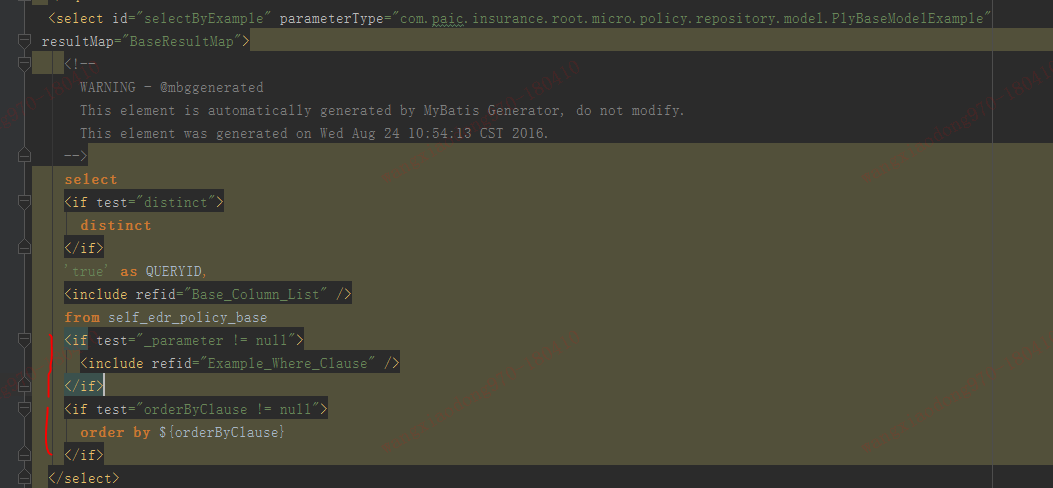
人脸识别



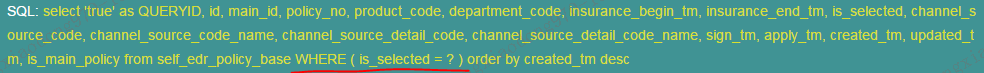
批改提核申请



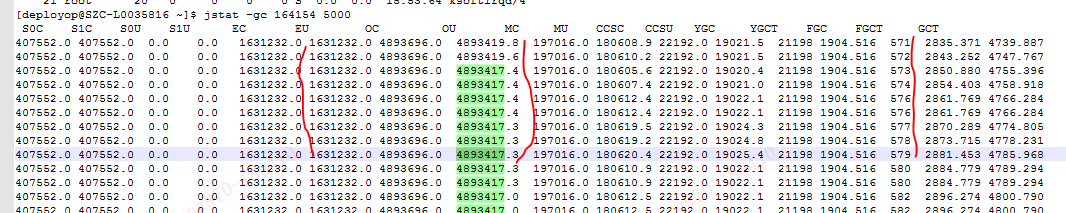
随后检查代码，发现提核确实存在bug，查询数据库时，通过判断参数是否为空，如果为空则不加入该条件。是一种常见的写法。



但在业务代码中，并没有判断参数为空的情况，依旧让查询了数据库，因此产生了如下的sql，这个表需要依赖于main\_id，在main\_id存在的情况下，最多就6条记录，但不存在就会拉取上百万的数据量到系统里。



导致了新生代被沾满，老年代也就跟着升了上去，触发了FGC，且由于大量数据处理时间长，导致了，效益低，也就报了GC overhead limit exceeded的问题。



后面再检查也发现了还存在一些类似这种情况的bug，一样的sql判断参数，业务代码并没有考虑参数为空的情况，导致大量数据被拉取。

因此，这次高负载的问题，是由于原本系统就存在一定bug，但当时的并发量和业务情况并没有足以让问题明显暴露出来，这也是为什么保单中心之前有时会有高负载的问题，但持续并不久，后面随着业务的增涨，如数据异构，异步增加任务、人脸识别图片大小前端没做上传限制、保单全量异构以及批改业务的增长(从之前的2千多增长到目前的3千多)，导致了之前的严重bug暴露了出来。并且只有在查询条件为空时，才会明显暴露这个问题。

## 高负载诊断步骤

1. 首要留下案发现场

1） 堆快照

jmap -dump:live,format=b,file=./dump.bin pid

dump下堆的当前情况，如果没有反应，可以使用-F参数强制dump出，但此时live参数会无效。

一般启动脚本也有配置-XX:+HeapDumpOnOutOfMemoryError在内存溢出时，dump出堆的快照。

2）实例情况

jmap -histo:live pid 留下当时的实例情况，会统计实例个数和字节大小。同样，如果命令无效，可以使用-F参数。

3) 线程运行情况

首先留下高负载的子线程id

top -Hp pid

再留下当时的线程运行情况

jstack -l pid

如果无效，同样可以加入-F 参数强制输出。

3）gc情况

再检查gc的情况

jmap -heap pid

检查当前堆中老年代和新生代的情况

再定时打印gc 情况

jstat -gc pid 5000

主要检查gc回收情况

1. 分析问题

优先检查jvm的配置情况，也可以通过jinfo查看当前jvm的配置参数，特别注意某些情况下SurvivorRatio参数是没有效果的，比如使用吞吐量的垃圾收集器，采用了自适应策略等。

再检查实例情况，也就是jmap -histo:live pid 输出的实例情况信息。检查是否存在大对象，或者大量实例的情况，如果存在再找相关代码。如果不确定业务代码；则可以分析dump文件，定位具体的业务代码；

jstack文件分析：

如果对于高负载情况，重点关注RUNNABLE线程

如果cpu很闲，则重点关注waiting for monitor entry线程

如果在定位高负载子线程没法定位到业务代码，则分析jstack会变得比较耗时，每个RUNNABLE都需要检查对应的业务代码。

1. 注意

dump文件，超4G需要拆分下载下来，下载后最好用git提供的终端来进行合并，用windows命令可能会有问题。

dump文件分析有eclipse的MAT，但是如果文件太大，会报堆溢出；此时可以使用jprofile;