技术团队如何为金融系统保驾护航

**介绍**

大家好，我是姚磊，来自瀚银，目前和团队从事金融科技方向的产品研发。2015年和团队一起完成了宜信支付账户平台v3.0版本从无到有的搭建，2016年和团队一起做瀚银聚合支付产品。目前我们和中小民营银行及互联网银行合作，帮助刚成立的一些中小银行搭建聚合支付+账户+清结算系统，同时输出基于资金存管、消费金融、供应链金融等场景的解决方案到合作银行。

金融系统属于一类系统，无论是基础的支付、账户、结算和资金存管，还是偏业务的消费金融、供应链金融以及行业金融解决方案，都具有一定的类似性，因此一些经验也可以在这些场景下复制。金融系统的金融属性在我理解是“正好一次”，即多一次不行，少一次也不行，因此作为一个资金交易系统，以我个人填过的坑来看，面临的挑战有一下几点：比如以一个清结算系统T+1日的结算功能举例，第一严重是资金重复结算，第二严重的是结算时间延迟了，第三严重的是系统不能提供稳定的服务。

为了保证系统高可用和资金安全，瀚银为此也买过很多单，因此，这些经验都是通过真金白银换来的。本文通过真实案例和切身实战经验来描述瀚银走过的这些坑，希望正在经历像瀚银（创业团队或者中小初创团队）曾经一样面临这些问题的团队能够提早避免。

任何系统建设从0到1相对比较简单，但是要做到从有到好或者更好那就比较不容易，比如经常发现开发同事开发一个新功能，自己单元测试通过就提测了，到了测试那里一堆问题，主要原因是好多时候经验一般的开发人员只保证了代码主分支业务没有问题，但是异常分支业务都没有考虑就提测了，代码的异常分支处理才是代码逻辑最需要考虑的点。本文会从三个角度展开描述，一个是针对已经存在的系统如何主动发现问题潜在问题，另外一个基于系统建设过程中不同项目阶段的暴露的问题不一样来描述，最后讲瀚银如何通过一些最佳实践来规避问题和提高质量。

**开发常见问题**

1. 开发人员如何提高代码质量，减少频繁迭代产生的bug？
2. 线上环境突发事故，第一时间如何决策减少事故影响范围？
3. 开发人员排查问题速度过慢？
4. 随着业务的增长，问题越来越多，第一优先级需要解决什么？
5. 系统突然CPU、内存利用率暴增，如何定位代码？
6. 数据库连接数被耗尽，怎么办？
7. 各种OOM如何预防？
8. 随着系统交易量的增加，高可用系统的设计点很多，如何快速抓住建设要点？

......

通过本文的分享，会对这些问题在不同的地方给出答案。

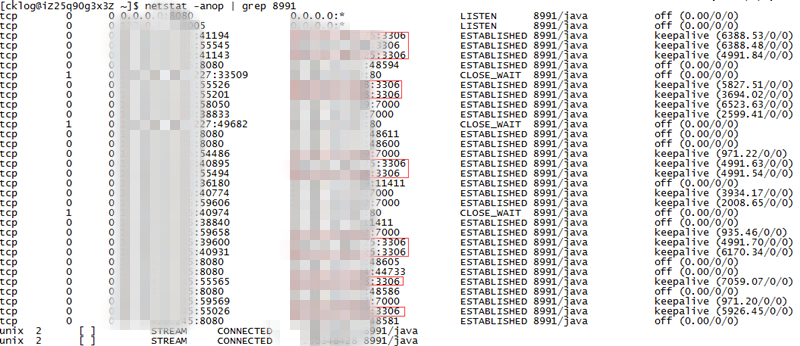
**从操作系统层面发现业务系统问题**

墨菲定律中提到会出错的总会出错，代码的bug也一样，会发生的一定会发生，只是看在哪种场景下会触发。未运行的代码是静止的，只有运行中的代码才能能真实地体现业务系统的状况，如何不阅读代码从操作系统层面去发现正在运行的代码问题是非常必要的。

无论多复杂的系统运行在linux之上无非就是一个进程，进程下面又由多个线程去执行每一行代码的业务逻辑，所以从操作系统层面只需要关注进程和线程即可。对于java编写的应用主要性能关注点是操作系统CPU和内存两个指标，在运行中的代码内存比CPU更加容易出问题。为什么java编写的代码内存比较容易出问题呢？如今硬件效率的提升导致（CPU的速度和支持多任务）处理器运算速度和和存储设备不是同一个量级，计算机都不得不加入一层高速缓存来作为内存与处理器之间的缓冲，缓存的引入提升了效率，但是带来一致性问题，程序的复杂度也增加了，比如java主内存和工作内存同步问题。如何从操作系统级别发现一些问题？

1. 了解应用和外部接口有哪些交互？是否有异常现象？

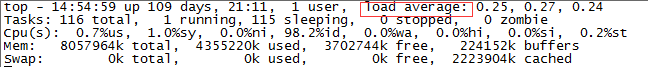
应用除了处理本身代码逻辑以外，往往还需要和外部服务或者接口通信。通过 netstat -anop | grep pid 就可以发现系统正在和哪些外部接口通信，比如下图中3306的端口显示系统有连接mysql数据库，并且有11个连接。系统有时候报错“has already more than 'max\_user\_connections' active connections”，这时候通过此命令就可能发现有比较多的连接数，当连接数非常多的时候最大的可能是代码在不停的操作数据库或者有慢SQL导致的连接不释放，也就是说当你发现应用连接数大于日常正常数量的时候，系统的代码可能就有问题。同理，下面第二张图通过netstat -anop | grep 6379得知应用有连接redis.



bbbb_副本

1. 线程数异常？

任何资源都是有限的，应用设计的时候需要考虑资源限制才能避免应用在某些时候因为资源过度使用而奔溃，线程数的控制就非常重要。线程的无限制创建，最终导致其不可控，特别是隐藏在代码中的创建线程方法。当系统的SY值过高时，表示linux需要花费更多的时间进行线程切换，Java造成这种现象的主要原因是创建的线程比较多，且这些线程都处于不断的阻塞（锁等待，IO等待）和执行状态的变化过程中，这就产生了大量的上下文切换，这时候体现在操作系统层面一种现象是下图中load average过高，这个值多少合适一般和cpu个数有关系，假如有两个cpu，这个值若大于4，系统建议报警。



除此之外，Java应用在创建线程时会操作JVM堆外的物理内存，太多的线程也会使用过多的物理内存，到了一定情况下系统应用就会报错“Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread”，这个错误是告诉我们系统创建了太多的线程导致的。这种错误的常见场景是写一个定时器去处理任务，定时器的频率1分钟一次，但是任务处理完需要10分钟，定时器浪打浪就会不停的产生新的线程，最终导致触发native OOM.

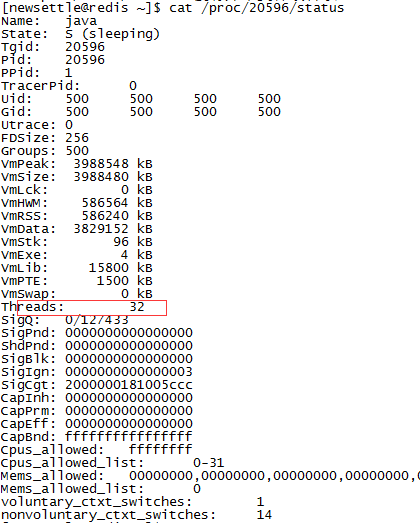
线程数过多会导致应用异常，那么如何快速统计一个进程的线程数量，有以下命令都可以使用：

cat /proc/pid/status

top -bH -d 3 -p pid

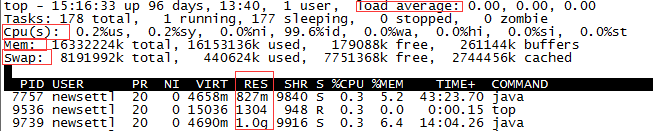
pstree -p pid | wc -l

pstack pid | head -1



1. 检查系统瓶颈

系统资源瓶颈一般关注CPU、内存、硬盘、硬盘IO、网络IO，Linux操作系统总体性能可以通过top命令如下图，一般系统应用只需要关注load average、Cpu、Mem、Swap、Res这些指标。



Load average: 说明CPU的拥堵情况，线程过多会导致这个值升高；

Cpu(s): us表示应用空间占比，sy表示内核空间占比，此值过高表示系统应用异常；

Mem: used过高需要关注，buffers表示用作内核缓存的内存量；

Swap: 系统的物理内存不够用的时候，把硬盘空间中的一部分空间释放出来，以供当前运行的程序使用；

RES: 进程使用的，未被换出的常驻/物理内存，一般这个值和JVM的参数配置有关，如果%MEM使用过高则需要关注；

VIRT: 进程使用的虚拟内存，VIRT=SWAP+RES.

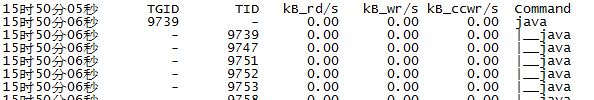
以上都是通过top命令总体检查，还有以下方法可以也可以查找系统瓶颈：

Iostat: 查看系统总体IO情况，不可以通过进程查询；

Pidstat：如果需要查看某个进程占用的物理内存和虚拟内存，直接使用 pidstat -r -p pid 1 2查看：



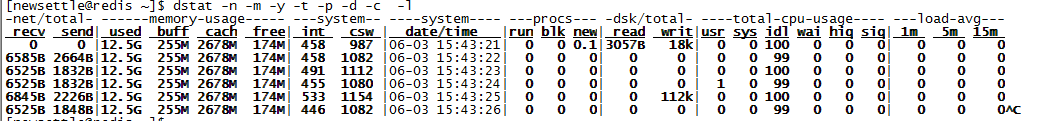
如果要查看某个进程的文件IO情况，可以通过命令pidstat -d -t -p pid 1 5查看：



Vmstat: swpd过高通常是物理内存不够用了,swap的消耗主要关注swap IO,swap IO 发生比较频繁，那么严重影响系统的性能:



Dstat: dstat -n -m -y -t -p -d -c -l

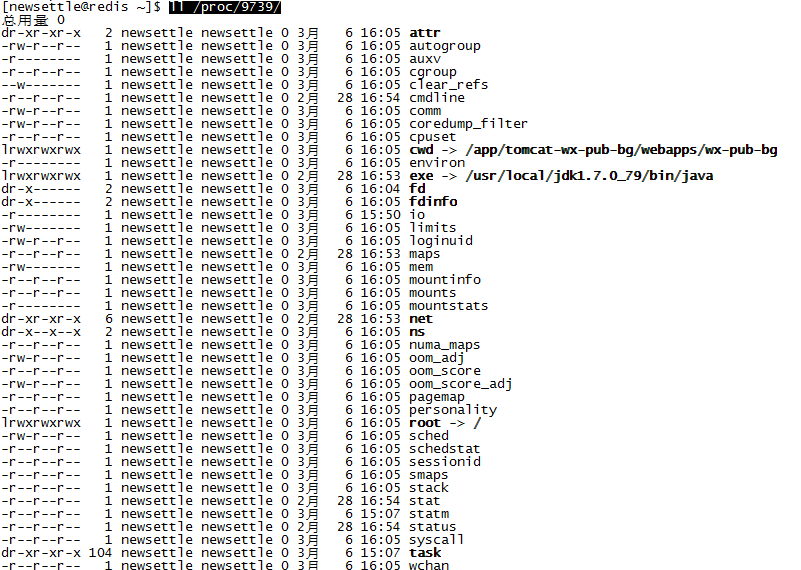


Strace：此命令用来跟踪进程执行时的系统调用和所接受的信号，linux中进程不能直接访问硬件设备，必须由用户态模式切换为内核模式通过系统调用访问硬件设备。通过strace -T -t -f -p pid可以显示具体进程的系统调用情况。

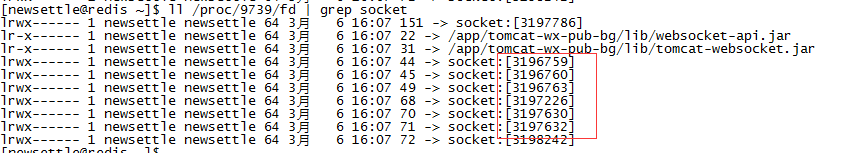
网络IO可以通过cat /proc/interrupts查看，网络IO消耗需要关注网卡中断是否不是均衡的分配到各个CPU，Java应用本身一般不会造成网络IO消耗严重。一般java最常用的网络命令是抓包分析请求内容，比如说需要抓取指定IP的网络包tcpdump -i eth0 |grep "110.3.224.253".

1. 检查系统应用的进程

对于java应用从操作系统层面观察，就只有进程和线程两个指标，任何东西在操作系统层面都是以文件的形式存储的，进程也不例外。Linux上部署一个tomcat程序产生一个进程，这个进程所有的东西都在这个目录下ll /proc/pid/



通过ll -tla /proc/pid/fd可以查看当前进程运行情况，比如想查看当前进程的socket通信，可以通过 /proc/pid/fd | grep socket,进而通过lsof | grep 3198242可以判断出这个socket通信连接的是redis服务。



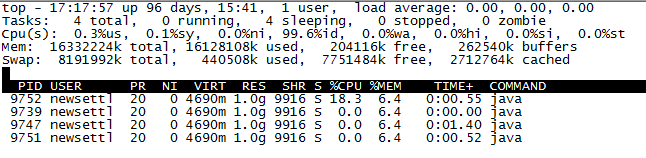


1. 检查系统应用的线程

Java应用出现异常最希望的就是通过线程或者进程定位问题，然后定位到具体的代码位置，最终解决问题。对于java程序，如果内存发生OOM，做法是通过jmap -dump:format=b,file=/tmp/dump.dat pid导出内存栈，然后通过可视化工具MAT定位出问题的代码，但是需要知道的是dump会导致full GC一次；如果是想通过线程栈定位正在执行的线程在做什么事情，可以通过jstack -l pid导出线程栈进行分析。假如通过上面提到的top命令或pidstat等命令发现某个进程消耗的内存、CPU或者IO过高，这时候需要定位到具体代码在做什么，定位方法如下：

步骤一：首先通过top或者pidstat命令定位的具体的进程号，比如9739；

步骤二：top -p 9739后，然后按住shift+H健开启Show threads on, 通过观察定位到具体的线程号，比如9751：



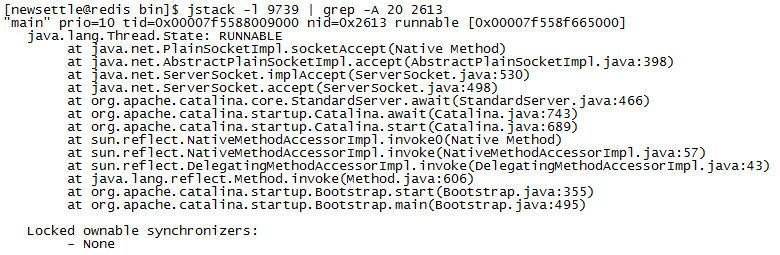
步骤三：把9751转换为十六进制2617，因为线程栈中线程的序号是以十六进制存储的：



步骤四：jstack -l 9739 | grep 2617 来分析这个线程正在做的事情，下图显示是gc线程：



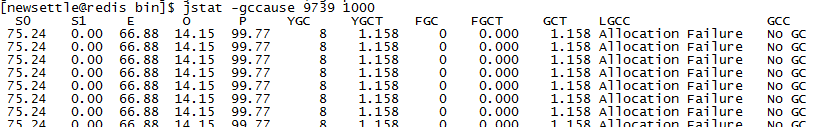
同理换一个其他线程9747，也可以发现代码正在执行的事情：



所以如果CPU或者内存消耗过高，通过以上步骤就可以准确地定位具体的业务代码。

1. 观察应用GC情况

GC会导致系统应用暂时的停顿，如果频繁的GC就会产生延迟响应，对于互联网面向用户的实时交易延迟是不可接受的。开启GC日志可以收集GC的统计信息，可以方便定位观察应用长时间停顿原因，命令如下jstat -gccause pid 1000.



JVM调优的方法都是为了尽量降低GC所导致的应用停顿时间，但是目前内存管理方面JVM已经做得非常不错了，因此如果不是有确切的理由证明GC造成性能低下，就没有必要做过多的JVM方面的调优，因为有可能反而调出问题。

1. 检查应用日志

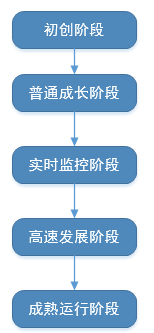
一个应用做的好不好，从应用打印日志就可以看出来，好的日志打印会打的和表格一样，可以通过AWK命令快速定位问题；除了打印规范，就是检测日志中的Exception和Error，业务日志是运行期产生的，每天产生不一样的异常，一个成熟的应用发生Exception和Erro的情况非常少，但是往往好多初建的项目源于赶进度只要业务功能不影响就忽略了日志异常，从而隐藏了很多问题。对于日志部分，建议是通过工具或者脚本实时监控日志中的带有Exception或者Error的关键字，有问题就报出来，类似下图：



以上内容简单总结了如何从操作系统层面发现应用可能存在的问题，这种些方法完全不需要通过阅读业务代码，也不限定于某个具体的业务系统，对于任何应用都能简单地做一个体检。本文的第二个视角来自项目周期，项目从一开始立项开发再到持续迭代发布新功能，最后到成熟期运营，每个阶段特点不一样，需要解决的问题不一样，根据瀚银的项目经验，不同的阶段会存在哪些问题呢？

**从项目角度看业务系统问题**

经历过一个项目完整开发周期的同学肯定清楚每个不同的阶段都会遇到不同问题，解决问题就感觉像游戏打怪升级，并且比较有意思的现象是高级的bug一定出现在低级的bug之后，只有解决了低级的bug，才会出现更加高级的bug.根据经验本文把项目周期定义为这五个阶段，初创阶段（协调）->普通成长阶段（质量）->实时监控阶段（预警）->高速发展阶段（高速扩张）->成熟运行阶段（稳定），分别来分析这几个阶段可能遇到的问题，希望帮助到正在创业的或者中小项目团队提前避免，因为团队初期一定会遇到这些问题，成熟的团队都已经历过了这个阵痛期。



1. 初创阶段：项目初创阶段，人员刚刚组成，团队属于磨合期，初期沟通低效，时间成本高。这个阶段作为技术团队首先要提高对外和对内的协作效率，对外主要沟通对象是产品和测试，对内就是技术团队内部合作问题。因为每个人考虑问题的角度不一样，这个阶段很有可能花费大量时间沟通，这个阶段各个团队的leader一定要精选，这样统一的入口和出口对接人会解决很多问题。
2. 普通成长阶段：项目/系统开始迭代开发，业务正常发展，线上有商户/用户使用，但是偶发发现有商户或用户投诉系统产品功能不完善，开发人员开发的功能有bug，其实这个时候进入了质量问题暴漏阶段，问题主要体现在产品设计有漏掉、开发代码质量不高、测试场景不充分等，所以这个时候建议有一定项目管理流程和机制，并不是一味地提倡快速迭代，因为步子迈得太大一定会容易摔跤。同时，另外一个问题发生了，如果每次都是用户先发现问题，这个场景很尴尬，并且会导致用户流失，所以系统迫切地需要一个实时监控系统，所有的问题早于用户发现，早发现早解决。
3. 实时监控阶段：如果系统没有监控和裸奔没有什么区别，就好比你驾驶着一辆没有仪表盘的车辆，每天都提心吊胆，担心系统出问题后用户投诉。实时监控系统是所有项目人员的眼睛，有了实时监控相当于给系统撒了一张网，让系统所有的点都是可控的、预期的，给人安全感。实时监控主要监控分为业务监控和非业务监控，具体实现这里不做阐述。
4. 高速发展阶段：随着业务的前期的发展和积累，监控系统也能帮助我们看住系统，这时候业务发展稳定，可能就悄悄来到了高速发展阶段。业务量骤增导致系统并发响应慢、数据库数据量增长过快、单点故障等现象，这个时候其实遇到的问题是性能问题。对于一个初创团队一开始一定是业务核心，不可能过度设计性能问题，好的架构也不是一次设计出来的，一定是演变过来的。这个阶段最需要解决的就是高可用问题，如何建设高可用系统有很多这方面的书籍，但是依据瀚银的经验，优先需要做到的几点是：分布式部署、灰度发布和实时预警。分布式部署可以横向扩展避免单点故障，灰度发布可以解决上线带来的事故并且能够秒级回滚，实时预警让系统处于看管之中。
5. 成熟运行阶段：经历了以上阶段之后，就是成熟运行阶段了。从系统来说，成熟运行阶段只关心一个问题就是线上偶发事故的响应机制。任务系统都一定会发生事故，像最近的发生的Gitlab和AWS事件，事故发生了不可怕，如何快速恢复是最重要。瀚银的生产事故响应机制是如果事故发生了第一时间不是去解决问题、定位问题，而是如何快速恢复交易、减少影响范围、通知商户/用户。为什么这样说？因为技术人员的特点是发生事故后不自然地就去想分析问题和定位问题，这个思路是不对的，增加了事故影响时间。成熟运行阶段，从项目来说，这个时候可能开发任务不那么紧急了，这个时候还需要注重如何提高团队的积极性、如何做好技术分享、如何归纳总结历史问题和知识等。

总体来看这五个阶段，对于创业团队和初创团队基本是这个顺序，当然不排除很有经验的团队会一开始把这些问题都解决了，所以这些经验给有需要的初创团队。如果一个系统建设解决了协调阶段、监控阶段、高速发展阶段所面临的痛点，那么这个团队无论做什么项目，一定可以事半功倍。

本文至此通过两个视角描述了如何发现系统应用的一些问题，最后附录一些瀚银的业务代码开发的自查、审查规范，瀚银内部称为checklist，这些规范都来自项目实际经验。

**瀚银开发规范**

1. 编程规范

（1）防御式编程，根据有限枚举先处理断言，再处理错误，最后处理正常逻辑，正常逻辑外尽可能多的处理异常分支；

（2）开发自测代码必须有单元测试用例；

（3）新方案引进或者重大变更需要上级审核，比如表结构修改增加、中间件的引进、新项目的搭建等；

（4）上线任何功能都必须考虑实时监控埋点数据；

（5）上线任何功能都必须满足标准日志轨迹输出，把业务日志打印成表格的形式，方便通过实时日志解析展示可视化日志轨迹，方便快速问题订单跟踪；

（6）业务代码中所有SQL耗时打印耗时；

（7）业务代码中关键方法打印耗时；

（8）业务代码中异常栈禁止吃掉，需要打印到输出日志中，方便实时异常字抓取报警；

（9）和第三方接口交互，发生网络异常的地方需要抓取IOException处理，同时打印发生网络异常的URL，便实时异常字抓取报警；

（10）和第三方接口交互，需要设置连接超时和读取超时时间，避免同步线程阻塞；

（11）和第三方接口交互，需要考虑是否需要通过代理出网；

（12）和第三方接口交互，需要考虑是否要相互添加白名单；

（13）和第三方接口交互，需要考虑设置合适的work线程符合第三方并发数量限制；

1. 安全规范
2. 页面请求参数严格限制或者校验处理，防止SQL注入；
3. 页面URL请求做细粒度的权限拦截，防止访问权限过大；
4. 部署在公网的应用做好防止XSS攻击的防范措施；
5. 和第三方系统交互需要互加白名单确保安全；
6. 系统全站提供HTTPS服务；
7. 和第三方系统交互报文需要加密传输；
8. 用户敏感数据做数据脱敏；
9. 预防页面被频繁请求，占用系统资源；
10. 预防API被频繁请求，占用系统资源；
11. 性能规范
12. 常见OOM预防
13. 禁止应用中显示创建线程，避免不可控出现unable to create new native thread；
14. 控制select/update/delete/insert的数据级和可变集合的size，避免随着业务增加内存数据量不可控；
15. 页面查询不推荐全表查询，查询通过查询条件限制查询条数；
16. 页面下载条数和下载次数做限制，避免请求过多导致OOM;
17. SQL优化目标必须满足range、ref或者consts，不可以是all类型，避免慢SQL导致连接数耗尽影响业务功能；
18. 代码书写中考虑MySQL中共享锁和排它锁场景，预防产生死锁；
19. 代码中使用@Transactional要考虑数据库死锁场景；
20. 数据库单表达到一定数据量级需要做分库分表或者冷热数据隔离，避免业务增加带来的性能问题；
21. 尽量避免使用全局变量防止并发出现线程安全问题，从而影响业务；
22. 定时器问题预防
23. 定时器浪打浪情况下，任务重复处理会导致资金风险，建议使用redis避免；
24. 定时器浪打浪情况下，启动多个定时器即默认启动多个线程，影响系统性能;
25. 定时器浪打浪情况下，如果定时任务处理过慢会导致内存耗尽；
26. 避免系统中出现单点故障，包括中间件和应用程序等所有的节点；
27. 能异步处理的别同步处理，异步可以释放线程资源，避免阻塞，提高响应效率；
28. 随着业务量的增加，考虑功能拆分和数据库表拆分，除此支付系统建议按照通道拆分，不同的通道指定独立的work线程，分而治之，避免相互之间影响；提高并发的一个思路就是拆分，拆分后通过异步提高并发效率；
29. 事务中有http调用，严重降低并发能力。事务的原则是事务代码少，快进快出；
30. 当 for update 语句和 gap lock 和 next-key lock 锁相混合使用，又没有注意用法的时候，就非常容易出现死锁的情况。通过业务分析，大量使用锁的目的是用来防重，但是完全没有必要通过悲观锁的方式来解决，可用方法有：redis做分布式锁，主键防重，版本号（乐观锁）防重;
31. 日志打印问题-日志的打印必须是以 logger.error 或者 logger.warn 的方式打印出来，大量大量的日志影响磁盘IO,也会造成线程阻塞；还有Log4j1.2.14 版本使用了 synchronized 锁，大量压测会发现线程处于block状态，需要修改log4j配置，提高并发能力。
32. 操作规范
33. 上线功能模块必须进行灰度发布环境，以确保上线不会影响线上交易；
34. 上线功能模块判断是否需要进行额外的压力测试；
35. 上线功能模块判断是否需要考虑模块之间的先后上线顺序；
36. 上线功能模块判断是否需要运营提供额外支持，比如运营后台参数配置等事项；
37. 上线功能模块判断是否需要运维提供额外支持，比如配置网络环境、添加证书秘钥、创建文件目录、添加和删除jar包等事项；
38. 上线功能模块判断是否需要DBA提供额外支持，比如新增模块添加数据库访问白名单、增加数据库连接数等；
39. 上线功能模块判断是否添加了报警功能，包括业务监控、关键字监控、响应码监控等；
40. 上线功能模块判断是否需要执行额外SQL和有执行顺序;
41. 上线后产品是否需要进行业务点验收；
42. 线上功能模块发生事故或者bug，第一响应动作是通过灰度环境恢复系统而不是定位问题；
43. 交易相关的注意点
44. 对于第三方查询接口查询订单不存在的情况需要设置单独响应码，做特殊报警提醒处理，付款类的交易不可以设置为失败状态，这样可以避免资金重付支付；查询操作本身失败代码异常处理中不可以设置订单失败；
45. 资金类交易订单状态设置是根据第三方响应码设置的，订单状态的设置采用保守策略，对于不确定的状态不可以直接设置失败，这样可以避免资金重付支付；
46. 资金类交易订单需要有额外的主动查询和核对查询，以避免第三方接口异常情况；
47. 资金类交易需要有自动路由切换功能，当第三方接口发生宕机的时候，确保后续交易可以引流到成功的通道；
48. 资金类订单需要考虑并发情况下的重复提交，收款类的交易可以重复提交订单提高成功率然后后续通过补偿机制处理，付款的类交易绝对不可以；
49. 资金类订单需要通过数据库乐观锁库和更新条数来避免并发修改同一条记录，进而避免重复支付产生的资金风险；
50. 核心交易系统需要做到7\*24监控室轮流值班，确保交易系统第一时间发现问题修复问题；
51. 核心交易系统需要每天做日志巡检，这种弥补机制可以二次帮助确认系统问题，同时让每个同学更加了解系统；

前车之鉴，后车之师，希望以上经验的和实践能够帮到像瀚银一样在创业路上的同学们。作为技术团队，遇到的问题建议通过工具、技术化、自动化手段来控制和避免，因为团队初期依赖的是靠谱的人，然后团队再进化就是靠谱的人+一定的机制流程，最后所有的就变成自动化工具来完成。

很高兴能够分享，希望和大家共勉！！！