如何应对在线故障

在线故障？

- 意料之外的错误、无响应或者响应缓慢
- 服务中，影响用户体验
- 不能停机或者大面积停机
- 需要尽快恢复

应对思路

1. 根据经验来分析。如果应急团队中有人对相应的问题有经验，并能确定能够通过某种手段恢复系统的正常运行，那么应该第一时间恢复（回滚等），同时务必要保留现场，以备后续对问题的定位和修复；如果没有人有经验，则需要使用比较粗暴的办法保证服务可用，如定时重启、限流、降级等。
2. 业务负责人、技术负责人、核心研发人员、架构师、运维工程师以及运营人员对问题的原因进行快速分析。分析的过程需要首先考虑系统近期的变化，包括以下几方面：
   - 系统最近是否进行了发布上线工作?
   - 服务的使用方是否有运营活动?
   - 网络是否有流量的波动?
   - 最近的业务量是否上升?
   - 运营人员是否在系统了做了变动?
   - 依赖的基础平台和资源是否进行了发布上线?
   - 依赖的其他系统是否进行了发布上线?

可能的原因

- **代码Bug**：逻辑不严谨、连接未释放
- **代码性能**： 循环外部调用、未使用批量读取、正则循环等
- **内存泄漏**： 本地缓存
- **异常流量/攻击**： DDOS
- **业务量提升**： 容量预估失误
- **外部系统问题**：数据库、搜索引擎、分布式缓存、消息队列等中间件的性能问题

> CPU、内存、IO指标异常

三步走

- 监控：“我并不知道我要做什么”。需要监控机制来发现、暴露系统的性能问题。这里一般依赖于系统级别或者业务级别的监控工具。
- 分析：“我知道我要做什么”。需要计算机基础知识和分析工具。
- 解决：“我知道我需要知道什么了”。系统、程序参数的调整、代码的重构优化。

> 理解一个系统应该如何工作并不能使人成为专家，只能靠调查系统为何不能正常工作才行。

准备工作

> 故障分析的准备工作、需要掌握的知识，based on CentOS 6.5 && JDK 1.8.0_121

- **计算机基础知识**：计算机网络、操作系统、计算机组成原理
- **Java内存管理**：垃圾回收算法、垃圾回收器、关键GC参数、JVM内存模型等
- **Java代码基准性能测试**：可以使用JMH(微基准测试框架)来进行，能够去除JIT热点代码编译对性能的影响。
- **HotSpot虚拟机体系结构**
- **系统参数调优**
- **掌握系统常用诊断工具、JDK自带诊断工具以及其他诊断工具的使用**
- **了解业务系统**：总体架构、压力方向、容量预估、系统相关软件的版本、模式以及参数

常用系统诊断工具-CentOS自带

- uptime： 系统的运行时间、平均负荷，包括1分钟、5分钟、15分钟内可以运行的任务平均数量，包括正在运行的任务、虽然可以运行但正在等待某个处理器空闲以及阻塞在不可中断休眠状态的进程（等待IO，状态为D）的任务。
- dmesg | tail：该命令会输出系统日志的最后10行。常见的OOM kill和TCP丢包在这里都会有记录。
- vmstat 1：实时性能检测工具,可以展现给定时间间隔的服务器的状态值,包括服务器的CPU使用率、内存使用、虚拟内存交换情况、IO读写情况等系统核心指标。r，等待CPU资源的进程数，这个比平均负载load更能体现CPU的繁忙状况；b，阻塞在不可中断休眠状态的进程数；si、so，swap区的使用情况，如果不为0说明已经开始使用swap区；us、sy、id、wa、st，CPU使用状况，id + us + sy = 100。
- mpstat -P ALL 1： 该命令用来显示每个CPU的使用情况。如果有一个CPU占用率特别高，说明有可能是一个单线程应用程序引起的。
- free -m： 该命令可以查看系统内存的使用情况，-m参数表示按照兆字节展示。Buffer和Cache都被计算在了used里面，真正反映内存使用状况的是第二行。如果可用内存较少，会使用swap区，增加IO开销，降低性能。
- top： 包含了系统全局的很多指标信息，包括系统负载情况、系统内存使用情况、系统CPU使用情况等等，基本涵盖了上述几条命令的功能。
- netstat -tanp：查看Tcp网络连接状况。

> iproute工具集：ss、ip，可以替代netstat

常用系统诊断工具-Sysstat

- sar -n DEV 1 ： sar命令主要用来查看网络设备的吞吐率。可以通过网络设备的吞吐量，判断网络设备是否已经饱和。
- sar -n TCP,ETCP 1：查看TCP连接状态。active/s，每秒主动发起的连接数目(connect)；passive/s，每秒被动发起的连接数目（accept）;retrans/s，每秒重传的数量，能够反映网络状况和是否发生了丢包。
- iostat -xz 1：查看机器磁盘IO情况。%iowait,;await(ms)，IO操作的平均等待时间，是应用程序在和磁盘交互时，需要消耗的时间，包括IO等待和实际操作的耗时；avgqu-s，向设备发出的平均请求数量；%util,设备利用率。

> sar、iostat、pidstat属于sysstat软件套件

JDK诊断工具

- jstack：Java堆栈跟踪工具，主要用于打印指定Java进程、核心文件或远程调试服务器的Java线程的堆栈跟踪信息。
- jmap：Java内存映射工具（Java Memory Map），主要用于打印指定Java进程、核心文件或远程调试服务器的共享对象内存映射或堆内存细节。
- jhat：Java堆分析工具（Java Heap Analysis Tool），用于分析Java堆内存中的对象信息。
- jinfo：Java配置信息工具（Java Configuration Information），用于打印指定Java进程、核心文件或远程调试服务器的配置信息，也可以动态修改JVM参数配置。
- jstat：JVM统计监测工具（JVM Statistics Monitoring Tool），主要用于监测并显示JVM的性能统计信息，包括gc统计信息。
- jcmd：Java 命令行（Java Command），用于向正在运行的JVM发送诊断命令请求。由于jmap官方标注的是unsupported，jcmd可以作为其替代工具。
- visualvm：通过JMX接口连接JVM进程，从而能够看到JVM上的线程、内存、类等信息。可以安装各种插件。(通过CATALINA_OPTS开启Tomcat jmx接口)
- jconsole：功能类似于visualvm，可以显示具体的线程堆栈信息以及内存中各个年代的占用情况，并支持直接远程执行MBEAN。

其他工具

- jmc：Java Mission Control，是一款采样型的集诊断、分析和监控与一体的非常强大的工具。由于收费的原因，用的不是太多。
- greys-atonomy：在线诊断工具，通过动态修改字节码能够达到无须重启jvm添加日志、监测方法耗时等动态增强代码的目的。
- arthas：阿里开源的Java诊断工具箱，基于greys-atonomy而来，包括在线诊断、反编译字节码、查看最耗费资源的Java线程等。
- jwebap：JavaEE性能检测框架，基于ASM增强字节码实现。支持：HTTP请求、JDBC连接、method的调用轨迹跟踪以及次数、耗时的统计。二次开发的suishen-webap，加入了对Java8的支持以及Redis连接的监控。
- awesome-scripts：封装了很多常用诊断工具、脚本等，包括greys-atonomy、sjk、VJTools以及获取最耗资源的线程堆栈信息、统计TCP连接数目等脚本。

分析思路

- 根据日志输出的异常信息定位问题，需要区分Tomcat中的catalina.out（标准输出和错误）和localhost.xx.log（应用初始化的日志，错误则无法启动）
- 磁盘是否已满(`df -h`)？->删除多余日志
- 流量是否有异常？->限流、降级、扩展服务结点、架构优化
- 外部系统问题->数据库、搜索引擎、分布式缓存、消息队列的故障解决、性能优化、分区设计等
- 应用的CPU、内存、IO!!!

CPU分析

- 使用top、vmstat、ps等命令定位CPU使用率高的线程：`top -p [processId] -H`。
- `jstack [pid]`打印繁忙进程的堆栈信息
- 通过`printf %0x [processId]`转换进程id为16进制，在堆栈信息中查找对应的堆栈信息
- `jstat -gcutil [pid]`，查看GC的情况是否正常，是否GC引起了CPU飚高
- JVM加入`-XX:+PrintCompilation`参数，查看是否是JIT编译引起了CPU飚高

> CPU繁忙：线程中有无限空循环、无阻塞、正则匹配或者单纯的计算；发生了频繁的GC；多线程的上下文切换；JIT编译

CPU分析Tips

- 一个进程的CPU使用率是其所有线程之和（线程对应LWP），CPU使用率高可以配合mpstat具体分析，是否是单线程应用程序引起的。
- top的CPU使用率近似实时，ps则是平均使用率
- top的CPU使用率默认是Iris Mode，为单CPU衡量的一个值，最大值为100%。可切换为Solaris Mode，此值在多处理器环境下，为占的总的CPU的使用率，例如，4核CPU中%CPU最高值是400%。
- jstack查看线程栈时需要注意：由于jstack dump实现机制每次只能转储出一个线程的栈信息，因此输出信息中可能会看到一些冲突的信息，如一个线程正在等待的锁并没有被其他线程持有，多个线程持有同一个锁等。

内存分析

> 内存使用不当：频繁GC，响应缓慢；OOM，堆内存、永久代内存、本地线程内存

- **堆外内存**：JNI、Deflater/Inflater、DirectByteBuffer。通过vmstat、top、pidstat等查看swap和物理内存的消耗状况。通过Google-preftools来追踪JNI、Deflater这种调用的资源使用状况。
- **堆内存**：创建的对象、全局集合、缓存、ClassLoader、多线程
  - 查看JVM内存使用状况：`jmap -heap <pid>`。
  - 查看JVM内存存活的对象：`jmap -histo:live <pid>`。
  - 把heap里所有对象都dump下来，无论对象是死是活：`jmap -dump:format=b,file=xxx.hprof <pid>`
  - 先做一次Full GC，再dump，只包含仍然存活的对象信息：`jmap -dump:format=b,live,file=xxx.hprof <pid>`
  - 使用 Eclipse MAT 或者 jhat 打开堆dump的文件，根据内存中的具体对象使用情况分析
  - VJTools中的 vjmap 可以分代打印出堆内存的对象实例占用信息

磁盘IO分析

- `iostat -xz 1` 查看磁盘IO情况。
- r/s, w/s, rkB/s, wkB/s 等指标过大，可能会引起性能问题。
- await 过大，可能是硬件设备遇到了瓶颈或者出现故障。一次IO操作一般超过20ms就说明磁盘压力过大。
- avgqu-sz 大于1，可能是硬件设备已经饱和。
- %util 越大表示磁盘越繁忙，100%表示已经饱和。
- 通过使用 strace 工具定位对文件IO的系统调用

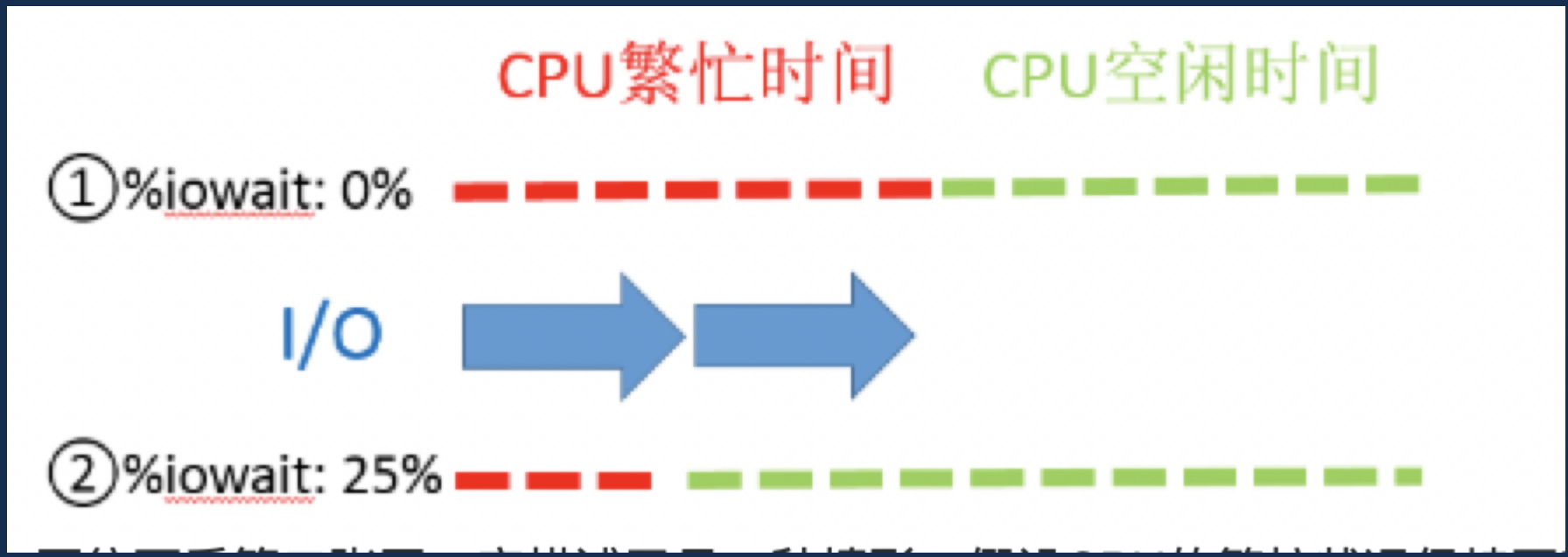
> IO性能差：大量的随机读写，设备慢，文件太大

网络IO分析

- `netstat -anpt` 查看网络连接状况。当TIME_WAIT或者CLOSE_WAIT连接过多时，会影响应用的响应速度。前者需要优化内核参数，后者一般是代码Bug没有释放网络连接。
- 使用 tcpdump 来具体分析网络IO的数据。tcpdump出的文件直接打开是一堆二进制的数据，可以使用Wireshark查看具体的连接以及其中数据的内容。`tcpdump -i eth0 -w tmp.cap -tnn dst port 8080`
- `sar -n DEV`，查看吞吐率和吞吐数据包数，判断是否超过网卡限制。

IO分析Tips

- %iowait 在Linux的计算为CPU空闲、并且有仍未完成的IO请求的时间占总时间的比例。
- %iowait 升高并不一定代表IO设备有瓶颈。需要结合其他指标来判断，如 await (IO操作等待耗时)、 svctm (IO操作服务耗时)等。
- avgqu-sz 是按照单位时间的平均值，所以不能反映瞬间的IO洪水



在线代码分析

- 远程Debug：Tomcat远程调试
- 在线Trace： BTrace、HouseMD、Greys-atonomy、arthas

故障解决

- 代码Bug： fix
- 性能问题： CPU、内存、IO使用优化
- JVM配置

CPU使用优化

- **不要存在一直运行的线程**（无限循环），可以使用sleep休眠一段时间。这种情况普遍存在于一些pull方式消费数据的场景下，当一次pull没有拿到数据的时候建议sleep一下，再做下一次pull。
- 轮询的时候可以使用 wait/notify 机制代替循环。
- **避免正则表达式匹配、过多的计算**。例如，避免使用String的format、split、replace方法；避免使用正则去判断邮箱格式（有时候会造成死循环）；避免序列/反序列化。
- **使用线程池**，减少线程数以及线程的切换。
- 多线程对于锁的竞争可以考虑 **减小锁的粒度**（使用ReetrantLock）、 **拆分锁**（类似ConcurrentHashMap分bucket上锁），或者使用CAS、ThreadLocal、不可变对象等 **无锁技术**。此外，多线程代码的编写最好使用JDK提供的并发包、Executors框架以及ForkJoin等，此外Disruptor和Actor在合适的场景也可以使用。
- 结合JVM和代码一起进行分析， **避免产生频繁的GC**，尤其是Full GC。

内存使用优化

- 使用 **基本数据类型** 而不是其包装类型能够节省内存。
- **尽量避免分配大对象** 。大对象分配的代价以及初始化的代价很大，不同大小的大对象可能导致Java堆碎片，尤其是CMS;
- **避免改变数据结构大小** 。如避免改变数组或array backed collections / containers的大小;对象构建（初始化）时最好显式批量定数组大小;改变大小导致不必要的对象分配，可能导致Java堆碎片。
- 避免保存重复的String对象，同时也需要小心String.subString()与String.intern()的使用，中间过程会生成不少字符串。
- 尽量不要使用finalizer。
- **释放不必要的引用** ：ThreadLocal使用完记得释放以防止内存泄漏，各种stream使用完也记得close。
- **使用对象池避免无节制创建对象，造成频繁GC** 。但也不要随便使用对象池，除非像连接池、线程池这种初始化/创建资源消耗较大的场景。
- 缓存失效算法，可以考虑使用SoftReference、WeakReference保存缓存对象。
- 谨慎热部署/加载的使用，尤其是动态加载类等。
- 打印日志时 **不要输出文件名、行号** ，因为日志框架一般都是通过打印线程堆栈实现，生成大量String。此外，打印日志时， **先判断对应级别的日志是否打开再做操作** ，否则也会生成大量String。

IO使用优化

- 考虑使用异步写入代替同步写入，可以借鉴Redis的AOF机制。
- 利用预读取或者缓存，减少随机读。
- 尽量批量写入，减少IO次数和寻址。
- 使用数据库代替文件存储。
- 使用异步IO、多路复用IO/事件驱动IO代替同步阻塞IO。
- 使用协程提高网络IO性能: Quasar。

JVM配置

- 合理设置各个代的大小。新生代尽量设置的大，但不能过大（会产生碎片），同样也要避免Survivor设置过大和过小。
- 选择合适的GC策略。需要根据不同的场景选择合适的GC策略。这里需要说的是，CMS并非全能的。除非特别需要再设置，毕竟CMS的新生代回收策略ParNew并非最快的，且会产生碎片。此外，G1直到JDK8的出现也并没有得到广泛应用，并不建议使用。
- 老年代优先使用Parallel GC（-XX:+UseParallel[Old]GC），可以保证最大的吞吐量。由于CMS会产生碎片，确实有必要才改成CMS或G1。
- 注意内存墙（严重阻碍处理器性能发挥的内存瓶颈），一般讲单点应用堆内存设置为4G到5G即可，依靠可扩展性提高并发能力。
- 设置JVM的内存大小有一个经验法则：完成Full GC后，应该释放出70%的内存。
- 配置堆内存和永久代/元空间内存之和小于32GB，从而可以使用压缩指针节省对象指针的占用。
- 打开GC日志并读懂GC日志，以便于排查问题。GC日志文件可以使用GC Histogram（gchisto）生成图表和表格。

代码性能建议

- 算法、逻辑上是程序性能的首要，遇到性能问题，应该首先 **优化程序的逻辑处理** 。
- **优先考虑使用返回值而不是异常表示错误** 。虽然现代JVM已经做了大量优化工作，但毕竟异常是有代价的，需要在合适的地方使用。一般用错误码返回值处理可能会发生的事情，用异常捕捉处理不期望发生的事情。如果使用异常并且比较关注性能，可以通过覆盖掉异常类的fillInStackTrace()方法为空方法，使其不拷贝栈信息。
- **查看自己的代码是否对内联是友好的** 。内联友好指的方法的大小不超过35字节(默认的内联阈值，不建议修改）、非虚方法（虚方法指的是在运行期才能确定执行对象的方法，最新的JVM对非虚方法会通过CHA类层次分析来判断是否可以内联）。

附录-技术选型规范

http://git.etouch.cn/gitbucket/BTC/btc_doc/wiki/技术选型规范

Q&A