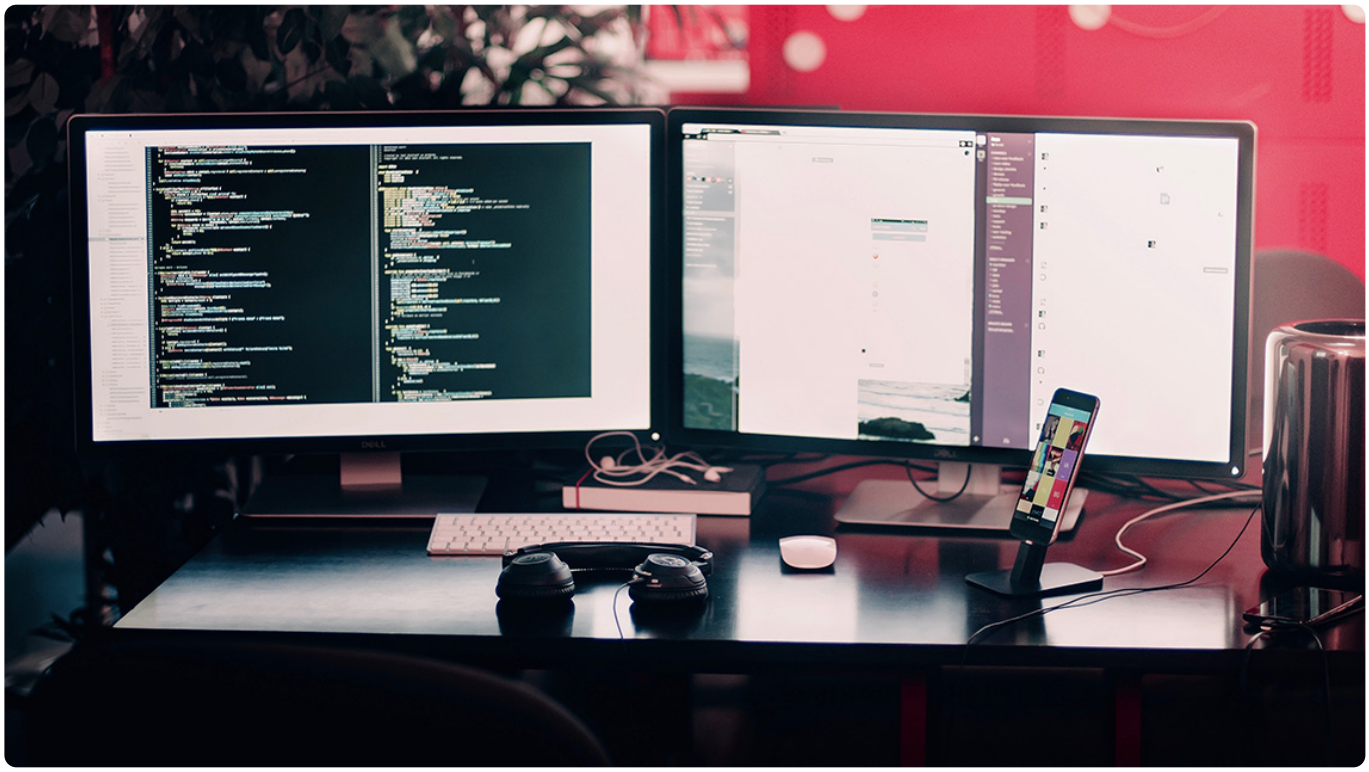
52 | 故障排查与根因分析

2019-10-29 许式伟

许式伟的架构课 进入课程 >



讲述：姚迪迈
时长 15:25　大小 14.13M

你好，我是七牛云许式伟。

写在故障排查之前

一旦 SRE 发现了我们线上业务出现故障，要如何解决呢？

正确解决问题的逻辑不是当场把问题修复好，而是用最快的方式让问题得以缓解或消失。考核 SRE 的核心指标中，通常会有季度故障率或年故障率指标。一次磨蹭的线上故障恢复，可能直接导致团队的年终奖取消。

及时故障恢复的手段应该怎么样，首先要看故障原因的初判。如果是在软硬件环境升级中发现的故障，通常的做法是版本回滚。如果是因为用户请求导致负载过高，则考虑扩容。如果

扩容不能让问题消失，则考虑服务降级，关闭一些不重要的功能以降低负载，或者主动抛弃一定比例的请求。如果是软硬件环境本身故障导致的业务故障，则通常是进行流量切换，把用户流量导向其他没有问题的任务实例。

解决了故障，我们就要开始定位引发故障的根本原因。

需要理解的一点是，定位故障原因的细节必然是和具体业务服务息息相关。但是有效地应对紧急问题，及时恢复故障的方法论，对每个业务团队来说却往往都是完全通用的。

另外，并不一定要线上出现了故障才需要去排查原因。

我们做事不能这么 “故障驱动”，要居安思危，主动出击。一个 SRE 例行的工作应该包含服务入口的指标检查，一旦发现存在非预期的指标波动，虽然有可能还没有触发警告线水位，就应该去排查问题。这就如同医生需要对病人定期进行诊断一样，这样的工作不一定要病人真生病了才去进行。

例行的排查过程需要有结论，发现问题风险要有应对方案。要避免团队陷在例行指标检查中，把它看做一项纯事务的工作来做。

SRE 的确看起来会有很多事务性的工作。但其实对工程师来说，这是一份极好的工作。

为什么我这么说?

我想从我自己的工作经历谈起。我的第一份工作是在金山，做的是 WPS 办公软件。我还没有毕业就去实习了，第一项工作任务做的是存盘和读盘，我们简称 IO。

在大部分工程师看来，IO 是一个非常事务性的编码工作，否则也轮不到我一个新手实习生来做。IO 工作干的最多的事情就是把一个个变量写到磁盘中，再把它从磁盘读出来恢复变量内容。听起来很无聊，对吧?

但实际上它非常核心。

首先，用户文件对办公软件来说，是最重要的资产。它的位置就如同用户关系在微信和 QQ 中的地位类似。当时 WPS 存盘读盘用的是 MFC 序列化框架。用这套框架最大的问题是版

本不兼容。新版本保存的文件，老版本读不出来。从需求分析角度来说，这是极不应该发生的，这根本就是在抛弃用户。

其次，数据是软件的灵魂。要了解一个软件的设计思想，最快的方法不是读代码，是研究它的数据结构。所以我当时虽然没有在做具体的某个功能，但是每每分析到微软的文件数据格式中出人意表的精巧设计，就会和团队中做对应功能的成员沟通，把微软设计思想告诉他们，而这往往能让他们受益匪浅，甚至改变自己的架构设计。

最后，业务功能是点，IO 是面。通过做存盘和读盘工作，我就成为了第一个对办公软件整体的架构设计思路有了全局理解的人。这很可能也是我仅仅毕业两年，就得以能够主导办公软件这样庞大的业务系统的整体架构的原因。

可能会有人觉得我太幸运了，第一个工作任务就是 IO 这样的核心工作。

但是我想说的是，遇到这种幸运的人，我显然不是第一个。对 WPS 团队来说，我只是第一个抓住这个幸运机会的人。

其实对于服务端工程师来说，SRE 工作也是一样的幸运机会。

对于一个服务端软件，最大的挑战并不是把业务逻辑写出来，而是写出来后如何保障 7x24 小时持续不间断的运行。

如果你不碰线上环境，你就也无法成为最理解系统的专家。SRE 工作是最有机会成为专家的，它和架构师一样，需要整体理解业务架构，并且通过线上环境的理解，去反向影响业务架构，以提高系统的稳定性。

从另一个角度来说，SRE 有越多的事务工作，就意味着这里面有越多的价值洼地还没有被挖掘。这也是为什么近年来服务治理的迭代如此日新月异的原因。

今天的 SRE 是幸运的，我们正处在历史变革的浪潮之中。

故障排查的方法论

回到正题。

今天我们想聊一聊的，是我们如何建立一种系统性或者说结构化的方案，来进行故障排查，找到问题的根因。

故障排查是 SRE 的一项关键技能。这项技能有时候会被认为是与生俱来的，有些人有，有些人没有。造成这种误解的原因通常是：对一个经常需要进行排除故障的人来说，让他去解释 “如何” 去进行故障排查是一件很难的事情。

但是我想说的是，故障排查是一个可以自我学习，也是一个可以传授的技能。

新手们常常不能有效地进行故障排查，是因为要做到高效排查的门槛比较高。理想情况下我们需要同时具备两个条件：

对通用的故障排查过程的理解（不依靠任何特定系统）。

对发生故障的系统的足够了解。虽然只依靠通用性的流程和手段也可以处理一些系统中的问题，但这样做通常是很低效的。对系统内部运行的了解往往限制了 SRE 处理系统问题的有效性，对系统设计方式和构建原理的知识是不可或缺的。

让我们一起来看一下基本的故障排查流程。从理论上讲，我们将故障排查过程定义为反复采用假设 - 验证排除手段的过程：针对某系统的一些观察结果和对该系统运行机制的理论认知，我们不断提出一个造成系统问题的假设，进而针对这些假设进行测试和排除。

我们从收到系统问题报告开始处理问题。通过观察监控系统的监测指标和日志信息了解系统目前的状态。再结合我们对系统构建原理、运行机制，以及失败模型的了解，提出一些可能的失败原因。

接下来，我们可以用以下两种方式测试假设是否成立。第一种方式，可以将我们的假设与观察到的系统状态进行对比，从中找出支持假设或者不支持假设的证据。另一种方式是，我们可以主动尝试 “治疗” 该系统，也就是对系统进行可控的调整，然后再观察操作的结果。

第二种方式务必要谨慎，以避免带来更大的故障。但它的确可以让我们更好地理解系统目前的状态，排查造成系统问题的可能原因。

无论用上述两种方式中的哪一种，都可以重复测试我们的假设，直到找到根本原因。

真正意义上的 “线上调试” 是很少发生的，毕竟我们遇到故障的时候，首先不是排查故障而是去恢复它，这有可能会破坏掉部分的现场。所以，服务端软件的 “线上调试” 往往在事后发生，我们主要依赖的就是日志。这里的日志是广义的，它包括监控系统背后的各类观测指标的时序数据，以及应用程序的程序日志。

为了排查故障，我们平常就需要准备。如果缺乏足够的日志信息，我们很有可能就无法定位到问题的原因。

首先，我们必须能够检查系统中每个组件的工作状态，以便了解整个系统是不是在正常工作。

在理想情况下，监控系统完整记录了整个系统的监控指标。这些监控指标是我们找到问题所在的开始。查看基于时间序列的监控项的报表，是理解某个系统组件工作情况的好办法，可以通过几个图表的相关性来初步进行问题根源的判定。

但是要记住，相关性（Correlation）不等于因果关系（Causation）。一些同时出现的现象，例如集群中的网络丢包现象和硬盘不可访问的现象可能是由同一个原因造成的，比如说断电。但是网络丢包现象并不是造成硬盘损坏现象的原因，反之亦然。况且，随着系统部署规模的不断增加，复杂性也在不断增加，监控指标越来越多。不可避免的，一些现象会恰巧和另外一些现象几乎同时发生。所以相关性（Correlation）只能找到问题的怀疑对象，但是，它是否是问题根源需要进一步的分析。

问题分解（Divide & Conquer）也是一个非常有用的通用解决方案。在一个多层系统中，整套系统需要多层组件共同协作完成。最好的办法通常是从系统的一端开始，逐个检查每一个组件，直到系统最底层。

但真正协助我们找根因的关键无疑是日志。在日志中记录每个操作的信息和对应的系统状态，可以让你了解在某一时刻整个组件究竟在做什么。一些跟踪工具，例如 Google Dapper 提供了非常有用的了解分布式系统工作情况的一种方式。类似 Hadoop HDFS 之于 Google GFS，当前开源领域中也有 Open Tracing 项目对应于 Google Dapper，其项目主页如下：

https://opentracing.io/

文本日志对实时调试非常有用。将日志记录为结构化的二进制文件通常可以保存更多信息，有助于利用一些工具进行事后分析。

在日志中支持多级是很重要的，尤其是可以在线动态调整日志级别。平常如果我们记录太多日志会导致成本过高，但是如果需要及时通过调试来定位问题，我们不需要通过重新发布新版本的软件来追加跟踪日志。这项功能可以让你在不重启进程的情况下详细检查某些或者全部操作，同时这项功能还允许当系统正常运行时，将系统日志级别还原。

根据服务的流量大小，有时可能采用采样记录的方式会更好，例如每 1000 次操作记录一次。

不同服务之间怎么把请求给串起来的?

答案是用 Request ID。在整个 Tracing 的业务链中，我们给每个用户的 API 请求分配一个 Request ID，该 API 请求的响应过程中所有相关的日志都会记录 Request ID，以便我们可以便捷地通过它检索到与该 API 请求相关的所有日志。

有时，我们会通过 HTTP 回复包（Respnose）来快捷地得到 API 请求链的概要版本，比如通过 X-RequestTrace 这样的 HTTP 头来得到。这是一种更轻盈的 Tracing 实现，实现非常容易。当然缺点也比较明显，它只能真正去在线调试，而无法像日志那样去查看历史。

仅仅通过 Request ID 来查看请求链，对于故障排查是不够的。定位问题本身就是 “假设 - 验证排除 - 再假设 - 再验证排除” 这样的循环，直至最后定位到问题。所以基于时序数据的日志系统，往往查询支持非常多样化的过滤条件，功能非常强大。如果你对这方面的详细内容感兴趣，欢迎体验七牛云的 Pandora 日志管理系统。

第三种重要的排查问题手段，是提供服务状态查询 API 和监控页面来暴露当前业务系统的状态。通过监控页面可以显示最近的 RPC 请求采样信息。这样我们可以直接了解该软件服务器正在运行的状态，包括与哪些机器在通信等等，而不必去查阅具体的架构文档。另外，这些监控页面可能同时也显示了每种类型的 RPC 错误率和延迟的直方图，这样可以快速查看哪些 RPC 存在问题。

这种方法很像 X-RequestTrace 机制，它非常轻便。如果故障的现场还在，或者故障过去的时间还不长，那么它将非常有助于问题的排查。但是如果问题的现场已经被破坏，那我们

就只能另寻他途。

现实的实操建议

在现实中，要想让业务系统的故障排查更简单，我们可能最基本要做的是：

- 增加系统的可观察性。不要等狼来了羊丢了才想着要补牢。在实现之初就给每个组件增加白盒监控指标和结构化日志。
- 使用成熟的、观察性好的 RPC 框架。保证用户 API 请求信息用一个一致的方法在整个系统内传递。例如，使用 Request ID 这样的唯一标识标记所有组件产生的所有相关 RPC。这有效地降低了需要对应上游某条日志记录与下游组件某条日志记录的要求，加速了故障排查的效率。

代码中对现存错误的假设，以及环境改变也经常会导致需要故障排查。简化、控制，以及记录这些改变可以降低实际故障排查的需要，也能让故障排查更简单。

对于故障排查，我们需要建立系统化的对故障排查的手段，而不是依靠运气和经验。这将有助于限定你的服务的故障恢复时间（MTTR）。同时，就算是团队里的新手也可以更加快捷有效地解决问题。

造成故障排查过程低效的原因，通常集中在 “假设 - 验证排除” 环节上。这主要还是由于对业务系统不够了解而导致的。它通常表现在：

- 在错误的方向上浪费时间。关注了错误的系统现象，或者错误地理解了系统现象的含义。
- 试图解决与当前系统问题相关的一些问题，却没有认识到这些其实只是巧合，或者这些问题其实是由于当前系统的问题造成的。比如，数据库压力大的情况下可能导致机房局部环境温度也有所上升，于是试图解决环境温度问题。
- 将问题过早地归结为极不可能的因素，或者念念不忘之前曾经发生过的系统问题，认为还是由之前的问题造成。
- 不正确地修改了系统的配置信息、输入信息或者系统运行环境，造成不能安全和有效地测试假设，导致验证的逻辑本身存在问题。

理解我们逻辑推理过程中的错误是避免这些问题发生的第一步，这样才能更有效地解决问题。区分我们知道什么，我们不知道什么，我们还需要知道什么可以让查找问题原因和修复问题更容易。

另外，尽管系统能够帮上很多忙，但是故障排查有赖于很多背景知识，需要你更详细地学习业务系统的运行原理，了解分布式系统运行的基本模式。

结语

今天我们聊的是线上故障的排查与根因分析。

从理论上讲，我们将故障排查过程定义为反复采用“假设 - 验证排除”手段的过程：针对某系统的一些观察结果和对该系统运行机制的理论认知，我们不断提出一个造成系统问题的假设，进而针对这些假设进行测试和排除。

为了有效排查故障，日志系统在里面起到了关键作用。定位问题本身就是 “假设 - 验证排除 - 再假设 - 再验证排除” 这样的循环，直至最后定位到问题。所以基于时序数据的日志系统，往往查询支持非常多样化的过滤条件，功能非常强大。如果你对这方面的详细内容感兴趣，欢迎体验七牛云的 Pandora 日志管理系统。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。下一讲我们聊聊“过载保护与容量规划”。

如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。

上一篇　51 | 故障域与故障预案

下一篇　53 | 过载保护与容量规划

精选留言 (2)

写留言

Aaron Cheung
2019-10-29

如果你不碰线上环境，你就也无法成为最理解系统的专家。SRE 工作是最有机会成为专家的，它和架构师一样，需要整体理解业务架构，并且通过线上环境的理解，去反向影响业务架构，以提高系统的稳定性。

...
展开

2

leslie
2019-10-29

老师说的其实正是我在考虑的：关于"时序日志系统"会去了解体验一下看看是否是对现有监控良好的补充；好的监控才能更及时的发现问题，然后处理问题。运维的本质其实就

是"发现问题->分析问题->核实问题->分析/分解问题->解决问题"的过程。

展开