My Philosophy on Alerting

原文：My Philosophy on Alerting

知乎：My Philosophy on Alerting

译者：姚钢强

翻译：

审核和编写报警规则时，需要考虑以下的这些原则

- 报警的（电话，短信）触达应当是紧急的，重要的，可行动的，真实的。
- 规则应当表示你的服务正在或者即将发生问题。
- 过度监控比监控不足更加棘手，应移除告警噪音，以保持报警项的精确和有效。
- 你应该总是能够将问题分为以下几种：基本功能的可用性问题；延迟；正确性（数据的完整性、时效性和持久性）；以及特定功能问题。
- 规则描述症状是更好的方法，可以更轻松，更全面，更可靠地捕获更多的问题。
- 在基于症状的页面或仪表板中包含基于原因的信息，但要避免直接针对原因发出警报。
- 报警越往上层的服务走，在一个报警规则中可以抓住的明显问题就越多。但如果走得太远，就无法清晰地了解现状了。
- 如果你想在值班时，报警系统保持安静，那么需要有一套系统和标准化的流程能够自动处理那些需要被尽快处理，但不至于让你半夜三点钟爬起来上线的事件。

导论

在为各种不同的服务（包括大型和小型，快速发展的产品以及核心基础架构的多个部分）服务了七年之后，我已经形成了一套监控和报警的理念。它反映了我对监控和报警的基本看法。

- 每次收到实时报警（传呼机响起），我都应该能做出紧急反应。但是我每天只能这样做几次，否则会感到疲劳。
- 每个报警项都应该是可行动的，「报警又来了」这种信息不能指导行动。
- 每个报警项都应该需要人的智慧来处理，而不是机器人、脚本式的自动回应。

当我审核或者编写新的报警规则时，下面这些是我经常思考的一些问题：

- 它是否检测到了原本未被检测到的状况，这些状况是紧急的、可操作的、正在发生的或即将被用户看到的。请注意，零冗余情况（本来是多活三个节点，现在只剩一个节点存活了）或者接近满载（例如存储）的情况也算迫在眉睫。
- 是否发生过无视这条报警的情况，并且知道他是良性报警？什么时候，为什么，我能改进规则来避免这种情况吗？
- 它是否识别出肯定是（将会）伤害用户的情况？ 假如面对测试流量进入服务器集群之类的情况。是否能够识别出不会对用户造成伤害的情况并将其过滤掉？
- 我可以针对这个报警采取行动吗？该行动是紧急的，还是可以等到我醒来后，周末或者下一季度结束后？
- 是否有其他人需要同时被通知到？他们会不会解决问题？或者说我要替别人解决问题？我们能把这些事情联系起来吗？我的新添加的报警能不能等他们尝试去解决？

下面的这些想法当然是有些理想化的；在不断发展和变化的服务中，报警系统是不会完美的--但也有一些技巧可以让你更接近这个目标。

为用户进行监控

我称之为“基于症状的监测”，而不是“基于原因的监测”。你的用户关心你的 MySQL 是否宕机了吗？不，他们关心的是他们的查询是否失败。（也许你已经感到尴尬不安了，爱上你的MySQL服务器的 Nagios 规则了？但是你的用户甚至不知道你的MySQL服务器的存在！）你的用户关心非服务路径的程序是否在循环重启吗？不，他们关心的是他们的功能是否失效。他们关心你的数据推送是否失败吗？不，他们关心的是他们拿到的是不是最新的结果。

通常情况下用户只关心很少的一部分事情

- 基本的可用性和正确性。没有 "Oops!"，没有 500，没有挂起的请求、半加载的页面；丢失的 Javascript，CSS，图像，视频。任何以某种方式破坏核心服务的事情都应该被认为是不可用的。
- 系统响应要足够快。
- 完整性/新鲜度/持久性。你的用户数据应该是安全的，应该在你要求的时候回来，搜索索引应该是最新的。即使暂时不可用，用户也应该完全相信它会回来。
- 功能。你的用户关心的是服务的所有功能是否正常工作--你应该监控任何对你的服务很重要的方面，即使它不是核心功能/可用性（例如，计算器和股票行情显示在搜索结果中）。

差不多就是这样了。数据库服务器不可用和用户数据不可用之间有一个微妙但重要的区别。前者是一个可能的原因，后者是一个症状。你不可能总是干净利落地区分这些事情，特别是当你没有办法模拟用户的视角时候。但如果你能做到模拟，你就应该去做。

基于原因的报警是糟糕的（有时是必要的）

你可能会说，"但是，我知道数据库服务器无法访问会导致用户数据不可用。" 这很好。警惕数据不可用的问题。出现以下症状时发出警报：500，the Oops！但是一些白盒指标表示不是所有的服务都是从数据库获取数据然后到达客户端的。基于原因报警是不够好的，为什么呢？（"But," you might say, "I know database servers that are unreachable results in user data unavailability." That's fine. Alert on the data unavailability. Alert on the symptom: the 500, the Oops!, the whitebox metric that indicates that not all servers were reached from the database's client. Why?）

- 无论如何，症状都是必须进行告警的。产生症状的原因可能是网络断开、CPU 争用，或其他无数你还没有想到的问题。所以你必须抓住症状，而不是原因。
- 如果同时关注症状和原因，就会有冗余的报警；这些警报需要单独调整和管理，并导致症状和原因报警的重复或复杂的依赖关系树。
- 所谓不可避免的结果并不总是不可避免的：也许你的数据库服务器不可用了，因为你要开启一个新的实例或者关闭一个旧的实例。又或者是添加了一个功能来做请求的快速 failover，所以你不需要关心单个服务器的可用性。当然，可以用越来越复杂的规则来捕捉所有这些详细的原因，但为什么要这么做呢？这样会导致更多的虚假报警，更多的混乱，更多的调整，而且没有任何收获，这回导致你花在修复重要的警报上的时间变少。

但有时它们（基于原因的报警）是必要的。对于 "几乎 "用完配额的情况，如内存，磁盘即将用满时没有（通常）任何症状，所以你需要这些报警知道系统即将出现问题。但是尽量少用这些，一定不要为你能捕捉到的症状写基于原因的报警规则

从出口（或更远处）发出警报

在 C\S 系统中，最好的报警来自于客户端。

- 客户端可以看到重试的结果，客户端与服务器之间的网络延迟，并且可以比服务端更好地知晓面向用户的延迟和错误

- 在许多情况下，客户端（例如，混频器或应用程序服务器）聚合来自许多后端的响应，例如缓存服务，数据库，帐户管理/授权服务，查询分片等。当你在做基础设施的变更时，如果你使用客户端指标做监控，这个监控规则会更加健壮，不用频繁地更改。
- 许多情况下，客户端可以比后端呈现出更简单的整体视图。例如，如果一个请求打到数百个查询服务器上，那么从每个查询服务器的来看的话，视角就太有限了，是无法成为一个有用的警报来源的。

对于很多服务来说，这意味着要对最前端的负载均衡器所看到的延迟、错误等情况进行报警。这样如果 server 宕机了，只有他们真正对于用户产生影响时你才会看到报警。但是看到的问题比你从 server 看到的更大，覆盖的角度更广：如果 server 都宕机了，或者服务出了不计其数的500，或者 10% 的连接断开了，你的负载均衡器知道，但你的服务器可能不知道。

不过请注意使用特别外层的监控，可能会引入你无法控制和负责的代理。如果你能可靠地捕捉到你的用户到底看到了什么（例如，通过浏览器端工具），那就太好了！但请记住，这样的信息充满了噪音（ISP、浏览器、客户端负载和性能）；因此，最外层的监控不应该是唯一的方法。如果你的外部监控不能总是与你保持畅通，它可能也是有损的。如果走到这种极端，这仍然是一个有用的信息，但可能并不是你想要的。

原因仍然有用

基于原因的规则仍然是有用的。特别是，它们可以帮助你快速跳到生产系统中的已知缺陷。如果你在自动将症状与原因联系起来的过程中获得了很多价值，也许是因为有一些原因是你无法控制的，无法消除的，我提倡这样做：

- 当你写下（或发现）一个代表原因的规则时，检查相关的症状的规则是否也在报警系统中。如果没有，就将症状规则添加进系统。
- 在报警信息里写一个简短的摘要，列出所有基于原因的规则。一个人快速浏览一下，就可以确定他们刚刚被报警的症状是否有已经确定的原因。这可能看起来像：
    - TooMany500StatusCodes
    - Served 10.7% 5xx results in the last 3 minutes!
    - Also firing:
        - JanitorProcessNotKeepingUp
        - UserDatabaseShardDown
        - FreshnessIndexBehind

在这种情况下，很明显最有可能的 500 来源是数据库问题。 相反，如果引发的症状是磁盘已满，或者结果页面返回为空或数据陈旧，就可能是其他两个原因了。

- 删除有噪声的，持续的，低价值的基于原因的报警规则。

使用这种方法，那些不协调、嘈杂的规则所带来的精神负担已经从寻呼机的哔哔声（以及调查、跟进等等）变成了一行要浏览的文本。最后，由于您无论如何都需要清晰的调试仪表板（对于不以警报开头的问题），这是另一个公开基于原因的规则的好地方。

就是说，如果您的调试仪表板使您能够从症状足够快地转移到引起原因的地方，那么您根本不需要花时间在基于原因的规则上。

工单、报告和电子邮件

你通常有一些警报需要尽快注意，但不需要马上关注处理。我称之为「次关键报警」。

- 工单跟踪系统会很有用。只要能将同一警报的多次触发正确地放入单个故障单/错误中，让报警系统在工单系统建立一个 bug 类型的工单，这样用户跟踪效果就会很好。如果没有后续负责分类和关闭 bug 工单的责任，此系统将失败。如果报警系统打开的 bug 工单在数周内不被看到处理，那么这显然无法作为在次关键警报警得严重之前处理报警的方式，系统会失败。如果你的团队只是超负荷工作或没有分配足够的人员来跟进，此系统也会失败；你需要诚实地知道这需要花费多少时间，否则你会越来越落后。
- 每日（或更频繁）的报表也是可行的。一种可行的方法是编写长期存在的次关键规则（例如，「数据库容量已超过 90%」或「我们在过去一天处理了 1000 个非常慢的请求」，然后定期发出报表，用来显示所有当前触发的规则。 但是同样，如果没有问责制，这意味着电子邮件警报会更加垃圾邮件，因此请确保指定值班人员（或其他人）每天（或每次交班或其他有效方式）进有效处理。
- 每个报警都应通过工作流系统进行跟踪。不要只将它们转存到电子邮件列表或 IRC 频道中。通常，这很快就会变成专门的垃圾邮件列表或渠道，以便可以将其忽略。除了要短暂（通常几天，最多几周）来审核新报警规会不会频繁出发，这么做几乎总是一个坏主意。这么做也容易忽略这些警报的数量，突然间，数千个应用程序服务器每分钟都会触发一些旧的，调整不当的规则，导致打爆了邮箱；这种情况十分槽糕。

根本的一点是要建立一个系统和标准化流程，这仍然需要有响应的问责制，但不会高成本地叫醒某人，打断他们的晚餐，或阻止其与孩子的拥抱。

操作手册

操作手册是报警系统的重要组成部分；对于捕捉到的基于症状的每个报警，最好有一个条目可以进一步解释警报的含义以及如何处理警报。

通常，如果您的操作手册有很长的详细流程图，就有可能花较多时间来记录可能出现的问题，而花很少的时间来修复它，除非根本原因完全不在您的控制范围内，或者根本上不需要人工干预（例如 致电供应商）。我所见过的最好的操作手册都对警报的含义做了一些说明，以及有关警报应该注意的点（我们从 VendorX 的小部件中发现了一系列断电现象；如果你发现了这一点，请添加它到错误12345，我们将在对其进行跟踪处理）。大多数类似说明应该是短暂，不能长期使用的，因此 Wiki 或类似的产品的是很好的工具。

跟踪和问责

追踪所有收到的报警。如果收到了报警，而人们只是说 "我看了，没有什么问题"，这是一个相当强烈的信号，你需要删除报警规则，或者降级，或者以其他方式收集数据。准确率低于 50% 报警是不能使用的；即使是那些 10% 的假阳性警报，也值得多加考虑是否对齐进行修改。
拥有一个适当的系统（例如，每周对所有页面进行检查，每季度进行一次统计）可以帮掌握正在发生的事情，并弄清将报警从一个人转移到另一个人时出现的问题。

你太天真了！

尽管我认为这些使用报警的原则是极其好的，但是如果遇到了下面这些情况是可以破坏这些原则的

- 在基于症状的报警噪音基础上，有明确的导致症状的原因。例如，如果您的服务有 99.99%的可用性，但您有一个导致 0.001% 请求失败的事件，则不能将其作为一种症状进行警报（因为它处于基础症状的报警噪音中），但可以捕获导致症状的原因。也许值得尝试在堆栈中传递这些信息，但也许最简单的方法就是基于原因上发出报警。
- 您无法监控出口的数据，因为丢失了数据精度。例如，也许可以容忍某些 handlers/endpoints/backends/URLs 响应慢（例如信用卡验证相比与浏览待售的商品）

或可用性低(例如收件箱的后台刷新)。在负载均衡上，并不能将此类特殊的接口区别出来。需要从链路进行跟踪，从能发现数据区别的地方发出报警。

- 症状要到很晚才会出现，比如你的配额用完了。当然，你需要今早报警，有时这意味着要找到一个基于原因的报警(比如使用率 > 80%，按照最近1h的增长速度，将在 <4h 内用完)。但如果你能做到这一点，你应该也能找到一个不那么紧急的类似原因(例如配额> 90%，按最近 1d 的增长速度将在< 4d内用完)，可以涵盖大多数情况，并以工单，邮件提醒或每日问题报告的方式处理，而不是报警时已经面临最紧急的状况了。
- 设置的报警规则比他们要检测的问题更复杂。 有时候报警确实会这样。 我们的目标应该是趋向于简单，健壮，自我保护的系统(您怎么没注意到自己用完了配额？为什么这些数据不能转移到其他地方？)从长远来看，它们应该趋向于简单化 。但为了保持报警的精确，千万不要执着于此，因为这种局部最优对应的是相对复杂的规则。