大规模系统中的故障*

作者：本·毛雷尔(Ben Maurer)
译者：杨任宇　欧阳晋　胡春明

关键词：大规模分布式系统　故障诊断　事故复审

脸谱最佳实践：面对快速变化的系统可靠性。

故障已经成为任何大规模系统工程中的一部分。脸谱(Facebook)的价值观之一就是拥抱故障。门洛帕克市(Menlo Park)脸谱总部的墙上，悬挂着这样的海报：“假如没有恐惧，你会做什么？”和“天佑勇者”。

为了让脸谱在快速变化中保持系统的高可靠性，我们研究了通用的故障模式，并构建了用于处理故障的各种抽象。这使我们能够采取最好的措施确保在脸谱的整个基础设施中提高系统可靠性。为了指导我们抽象出可靠性概念，就必须理解大规模系统中的故障。为此我们构建了故障诊断工具和事故复审(incident review)机制，来推动我们不断改进系统，并预防未来的故障。

虽然每一种故障都有自己特有的情形，但是，很多故障都能归结到为数有限的几个根本原因上。

单机故障

一台单独的机器经常会发生孤立的故障(例如，某台机器的硬盘驱动器出现了故障，或部署在某台机器上的软件服务出现了诸如内存崩溃或死锁等代码缺陷)。这种故障不会影响到基础设施的其他部分。

避免此类单机故障的关键是实现自动化。自动化可以将已知的故障模式（如产生S.M.A.R.T.[1]错误的硬盘驱动器）和搜索未知问题的异常征兆（如替换那些响应时间异常慢的服务器）相结合，达到自动检测与处理故障的最佳效果。当自动化方法找出了某个未知问题的征兆时，就可以通过人工检查帮助我们开发出更好的工具来检测并解决未来可能出现的问题。

合法负载发生变化

脸谱用户的行为方式变化有时会给计算基础设施带来挑战。例如，发生世界范围的重大事件时，独特的负载类型会以异乎寻常的方式增加计算基础设施的压力。贝拉克·奥巴马

* 本文译自*Communications of the ACM*，“Fail at Scale”，2015, 58(11):44~49一文。

[1] S.M.A.R.T. (Self-Monitoring, Analysis and Reporting Technology)：自动监测、分析和报告技术。——译者注

(Barack Obama) 赢得 2008 年美国总统大选时，他的脸谱主页经历了创纪录的活跃。全美橄榄球超级杯 (Super Bowl) 或世界杯等大型体育赛事中的关键比赛也会导致极高的用户访问量及更新量。为应对这种负载变化，需要一些负载测试手段，如采用灰度上线 (dark launches)[2]，以对用户透明的方式激活软件的功能点，从而确保新的软件功能上线时能够适应并处理负载压力。

在这些事件发生期间收集到的统计数据通常能为系统设计提供一种独特的视角。重大事件常常会引发用户行为的改变（例如围绕某个特定对象所发生的比较集中的用户行为）。与这些变化相关的数据通常会引领系统设计决策的方向，使系统能在后续事件中运行得更加顺畅。

人为错误

脸谱鼓励工程师们“快速行动，破除陈规”（这是悬挂在脸谱总部墙上的另一幅海报），但在快速行动的同时，也会引入许多人为因素导致的错误。数据表明，人为错误是造成系统故障的因素之一。图 1 是对严重程度达到违反用户服务等级协议 (Service-Level Agreement, SLA) 的故障事件按时间进行的统计分析。每违反一次服务等级协议对应着一个没有达到内部可靠性目标并触发了一次系统报警的实例。由于我们的目标非常严格，所以这些故障中的大多数都属于小故障，不会被用户察觉。图 1(a) 表明，虽然一周内每天的网站流量基本保持不变，但周六和周日的故障发生率明显低于工作日。图 1(b) 展示了连续 6 个月每周发生的故障数，其中只有两周没有发生故障：圣诞节假期以及员工同行评议的那一周。

这两组数据似乎意味着，当脸谱员工忙于其他事务（如周末、假期，甚至绩效评估）而无法活跃地更新基础设施时，系统的可靠性显得相对更高。我们认为，这一现象并不说明问题应归咎于员工的粗心大意，而是证明了基础设施在面对诸如单机故

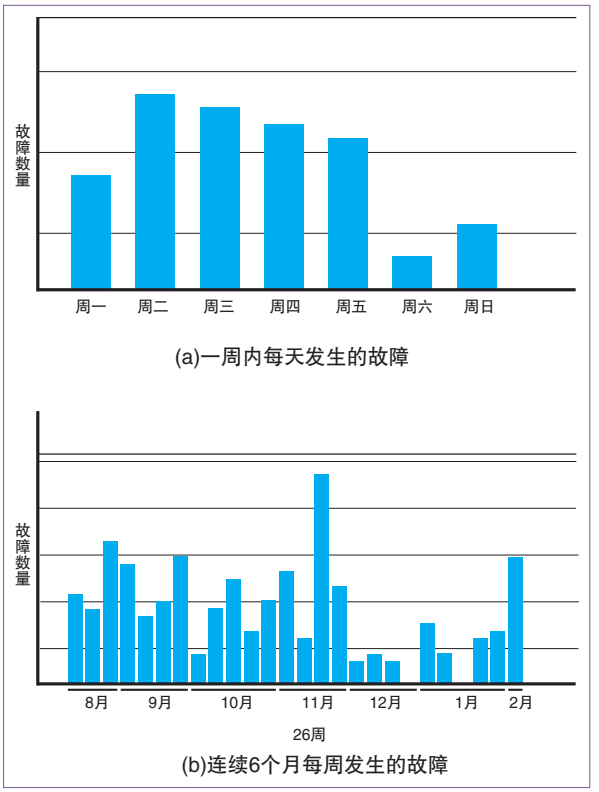

图1　违反服务等级协议的故障事件数

[2] 按产品需求优先级，抽出核心需求，在满足用户基本要求的情况下让产品快速上线，并通过限制流量、白名单等机制进行产品试用，以此收集用户的意见，从而萃取出用户潜在的需求，形成后续更有针对性的设计方案。——译者注

障等非人为因素的错误时，大都能够自我修复。

容易引发故障的三条途径

虽然每个故障产生的根本原因各不相同，但我们还是总结出了放大故障并导致故障大范围扩散的三种常见原因。对于每一种原因，我们都开发了防御性措施来减缓故障的大面积扩散。

快速部署的配置变更

配置系统的设计倾向于能够在全局范围内快速复制系统变更。快速的配置变更是一种强大的工具，它可以使工程师快速管理新发布的产品或调整设置。但是，快速的配置变更也意味着，如果部署了错误的配置，将快速引发故障。针对这一问题，为防止因配置变更引发故障，我们在实践中采用了一系列措施：

让所有人使用同一个公共的配置系统 使用一个公共配置系统可以确保流程和工具在各种类型的配置上均适用。在脸谱，我们发现团队有时会倾向于采用一次性的方式来处理配置问题。避免这种倾向，并采用统一的方式进行系统配置管理是一种事半功倍的提高站点可靠性的方法。

静态验证 (statically validate) 配置变更 许多配置系统允许采用一些弱类型 (loosely typed) 的配置方式，如 JSON[3] 结构。这些配置类型使工程师容易犯一些低级错误，如输错字段名称，在该使用整型的地方使用了字符串类型等。静态验证是捕捉这种低级错误最有效的方法。一些结构化的格式(例如脸谱使用的 Thrift [4]) 可以提供最基础的配置验证能力。此外，通过编写程序性的验证方法来进行更深入细节的需求验证也不失为一种方法。

运行小规模测试 (canary) 先将配置部署到小范围的服务中，可以防止出现一些灾难性的变更故障。小规模测试可以采取多种形式。最为常见的是 A/B 测试，例如只向 1% 的用户发布新配置。多个 A/B 测试可以同时运行，从而在一段时间内收集测试数据，跟踪各种性能指标。

然而从可靠性的角度看，A/B 测试并不能满足我们所有的需求。面向小部分用户所发布的配置变更，也可能导致相关服务器出现崩溃或内存溢出，很明显这类故障的影响将超出参与测试的有限用户范围。此外 A/B 测试也非常耗时。在进行微小变更时，工程师们往往不愿意使用 A/B 测试。因此，脸谱基础设施在小规模的服务器集群中自动测试新的配置。例如，如果我们希望向 1% 的用户部署一个新的 A/B 测试，我们会优先选择与访问量较少的服务器相关的 1% 的用户。我们在短时间内监控这些服务器，确保它们不发生崩溃或其他高可见性的问题。这种机制为所有的配置变更提供了一种基本的“合理性检测”(sanity check)，确保不会引发大范围的故障。

保留完好配置 脸谱的配置系统设计成在更新配置过程中发生故障时可以保留原有正确的配置。通常如果不提出特别要求，开发者所创造的配置系统往往会在收到无效配置更新时导致系统崩溃。我们则倾向于构建一种能在这种情形下保留原有配置的系统，并在配置更新失败的情况下向系统操作员发出报警信息。使用旧的配置来运行系统一般要好过向用户返回错误信息。

让回滚变得容易 有时，尽管付出了最大努力，但仍然部署了错误的配置。快速发现错误并回滚到变更之前的配置是解决这种问题的关键。我们的配置系统由版本控制来支持，所以回滚变得容易。

对核心服务的严格依赖

开发者通常假定诸如配置管理、服务发现、存储系统等核心服务永远不会出现故障。然而，这些核心服务中一旦出现哪怕是很短暂的故障，也会造成大规模的事故。

对核心服务数据的缓存 对这类服务的严格依赖通常是没有必要的。这些服务所返回的数据可以以一种方式被缓存下来，使得其中的某一个服务出现短暂的运行中断时，仍可支持大部分服务继续运行。

[3] JSON: JavaScript Object Notation，是一种轻量级的数据交换格式。——译者注

提供强化的 API 来使用核心服务 当使用核心服务时，遵循最佳实践而开发的共享库是对这些核心服务的最好补充。例如，一些库可以提供设计良好的应用编程接口，以管理缓存或提供良好的故障处理机制。

执行故障演习 开发者也许会觉得自己开发的应用能够承受核心服务的中断，但在尝试之前，谁也没有把握。对于这些中断，我们必须开发一组系统来进行故障演习，范围从对单机的故障注入，到人工触发的整个数据中心的服务中断。

延迟增加与资源耗尽

某些故障导致服务对客户端的响应时间延迟。这种延迟的增加可能微乎其微（例如人工配置错误造成 CPU 利用率增加，但仍在服务能接受的范围之内），或者可能趋于无穷大（如线程的服务响应出现死锁）。在脸谱，基础设施可以毫无困难地应对少量的访问延迟增加，但大量的访问延迟则会引发连锁故障。几乎所有的服务对等待处理的请求数都有一个上限。这种限制可能是因为在“一个线程处理一个请求”的服务模型中，系统仅能够提供有限的服务线程；或者可能是因为在一个基于事件的服务中，用于处理事件请求的内存是有限的。如果一个服务出现了大量额外的访问延迟，那么调用该服务的其他服务将会面临资源耗尽的问题。这种故障将会在多个服务层级中传播和扩散，从而导致大范围的系统故障。

资源耗尽是一种危害极大的故障模式，因为对一部分请求的处理所引发的故障将会导致所有请求无法被处理。例如，假定一个服务调用了一个新的只发布给 1% 用户的测试性服务。通常情况下，调用该测试性服务的请求需要花费 1 毫秒，但由于该服务出现故障，请求的处理时间变为 1 秒。新服务为了处理这 1% 用户的请求，要占用更多的处理线程，从而导致其他 99% 的用户请求无法正常处理。

为了避免这种类型的故障出现，同时确保较低的虚警率，我们采用了一系列技术手段：

受控的延迟 在分析以前发生的与延迟有关的故障时，我们发现，在很多最糟糕的故障中，都有大量在请求队列中排队等待处理的请求。出问题的服务因受资源限制（如有限的可用线程数量或内存等），会将请求放入待处理队列暂存下来，以保证将服务的资源使用量控制在限定范围之内。由于服务处理请求的速度赶不上请求到达的速度，待处理的请求队列会越来越长，直到超出应用所设置的上限。为了解决这一问题，我们试图在不影响正常操作期间的可靠性的前提下，对等待队列的长度加以限制。我们经过研究发现过度缓存 (bufferbloat) 与这一问题较为相似——都需要为了确保系统可靠性而进行排队，但又要确保不会在队列拥塞时造成过长的请求延迟。我们试验了一种名为 CoDel 的受控延迟算法 [1] 的变种：

```
onNewRequest(req, queue):
if (queue.lastEmptyTime()<(now-
N seconds)) {
    timeout = M ms
} else {
    timeout = N seconds;}
queue.enqueue(req, timeout)
```

在这一算法中，如果队列在过去的 N 毫秒 [4] 内没有被清空，那么请求在等待队列中的等待时间将被限制在 M 毫秒内；如果队列在过去的 N 毫秒内被清空，则请求在队列中的等待时间将被限制在 N 毫秒内。这一算法将防止请求在队列中长时间等待（长时间等待时，最近一次清空时间 lastEmptyTime 距离当前时间，会超过 M 毫秒的上限而超时），同时，也允许请求队列在短时间内激增以确保系统的可靠性。把请求的超时时间设置得这么短，似乎与我们的直觉相悖，但这样可让系统在无法应对快速到达的访问请求时快速丢弃来不及处理的请求，而不是让等待的请求堆积。设置短暂的超时能够确保服务器总是接受略微超过其实际处理能力的工作量，这样服务器也永远不会空闲。

[4] 原文如此，从伪代码看，疑应为N秒（或者伪代码中“N second”应为“N ms”），下文同。——编者注

该算法的另一个有吸引力的好处是 M 和 N 的设置并不需要人工调节。其他解决等待队列堆积问题的方法（如设置队列元素个数，或设置队列的等待超时时间）都需要基于每个服务进行调节。我们发现，M 为 5 毫秒、N 为 100 毫秒的设置在许多应用场景中效果都良好。脸谱的开源 Wangle 库 [5] 给出了该算法的一种实现，它也被用于我们的 Thrift [4] 框架。

自适应的后进先出 (Last-In First-Out, LIFO) 队列 大多数服务都按照先进先出 (First-In First-Out, FIFO) 的顺序处理请求队列。然而，当请求队列长度出现峰值时，先进入队列的请求已经在队列中等待了相当长的时间，以至于用户可能已经终止了产生请求的操作。与处理刚刚到达队列的请求相比，优先处理先进入队列的请求可能既耗费了资源又不会产生用户期望的价值。因此，我们采用自适应的后进先出的方式来处理请求。在正常运行情况下，服务器按照先进先出的顺序处理请求，但当请求队列开始形成时，服务器则切换到后进先出模式。自适应后进先出模式与 CoDel 算法可以完美地结合在一起，如图 2 所示。CoDel 算法设置较短的等待超时，阻止队列中请求的堆积，自适应后进先出模式则将新到来的请求置于队列前端，使新请求在 CoDel 所设置的等待超时截止前获得处理的可能性最大化。脸谱的 PHP 运行时环境 HHVM[3] 就包含了自适应后进先出算法的实现。

并发控制 CoDel 算法和“自适应后进先出”都运行在服务器端。服务器通常是实施延迟预防措施的最佳位置，因为一个服务器节点往往服务于大量的客户端，且相比于客户端拥有更多的信息。然而，有些故障非常严重，以至于服务器端的控制也无法奏效。这时，我们需要在客户端采取一种临时的应急措施。每个客户端会追踪基于每个服务的未被及时处理的请求个数。在发起新的请求时，如果这个服务中未被及时处理的请求个数超过了预设的上限，那么该请求会被立刻标记为错误。这种机制能够避免单个服务独占所有的客户端资源。

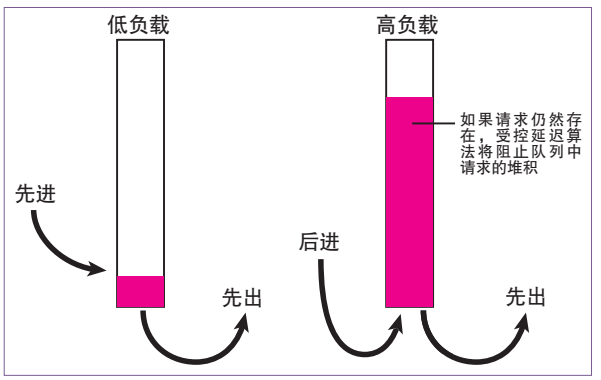

图2 先进先出（左）以及使用Codel算法的自适应后进先出

故障诊断工具

尽管采取了最佳的预防性措施，但总有一些故障仍可能发生。在服务中断故障发生时，正确的工具能够帮你快速找到问题的根本原因，尽可能缩短故障持续时间。

利用Cubism实现高密度仪表盘

处理事故时，快速获取信息相当重要。好的仪表盘 (dashboard) 可使工程师快速检查可能存在异常的性能指标，并利用这些信息推断故障的根本原因。然而，我们发现仪表盘变得太大，很难对它们进行快速浏览和定位，同时仪表盘的图表上曲线过多，很难一眼扫清并发现问题。

为解决这一问题，我们利用 Cubism[2] 构建了一个顶层仪表盘。Cubism 是一个用于绘制水平图的软件框架。水平图是一种曲线图，利用不同颜色以更高的密度对信息进行编码，使用户容易对多个相似的数据序列进行比较。例如，我们利用 Cubism 比较不同数据中心的性能指标。基于 Cubism 构建的工具使工程师能轻松地用键盘操作和浏览，从而快速查看多个性能指标。图 3 显示了利用分区图 (area chart) 和水平图对同一数据集在不同高度层进行呈现的效果。在分区图中，在 30 个像素的高度层已经很难辨认数据，但在水平图中，仍能非常容易地识别出数据的峰值。

关注最新的变更

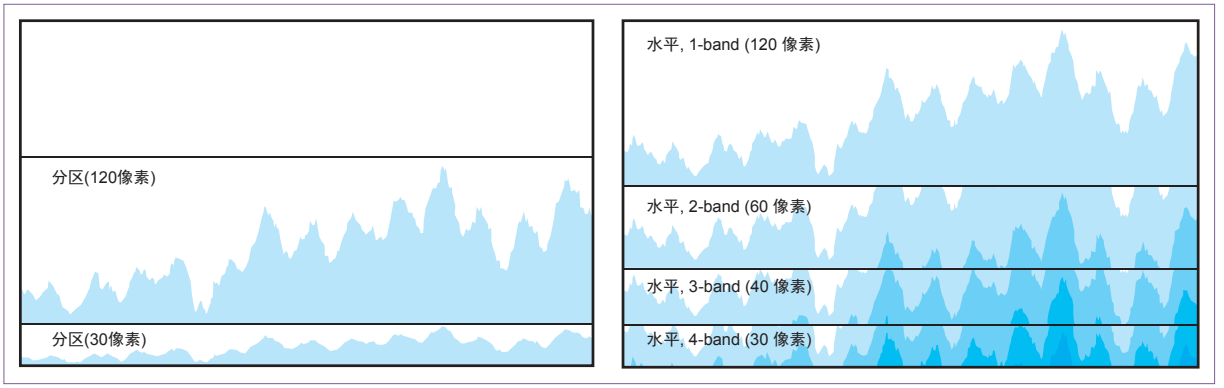

图3　分区图 (左)和水平图（右）

引发故障的头号原因是人为错误，因此，排除故障最有效的办法就是找出人们最近变更了什么。我们使用名为“OpsStream”的工具收集最近发生的从配置变更到软件部署等各类变更信息。然而，我们发现，随着时间的推移，这些数据源变得极其混乱嘈杂，成千上万的工程师进行不同的变更，当故障发生时，变更数据太多以至于无法对其进行精确的评估。

为解决这一问题，我们的工具尝试寻找故障与相关变更之间的关联关系。例如，当一个异常出现时，除了输出堆栈调试信息外，我们还输出请求所读取并最近修改的任意配置项。通常导致一个产生多个堆栈踪迹的异常的原因就存在于这些配置项数值之中。然后，我们就能快速应对这一异常，例如回滚配置项，并联系提交该配置变更的工程师进行后续处理。

从故障中学习

故障发生之后，事故复审过程会帮助我们从这些事故中学习并总结教训。

事故复审的目的不是问责。在事故复审中，迄今还没有人因为其造成的事故而被解雇。评审的目的是深入了解到底发生了什么，修复造成事故的缺陷，同时设置必要的安全机制来减少未来此类事故的发生及其可能产生的影响。

事故复审的方法论

脸谱开发了名为“DERP”（检测、扩展检讨、补救、预防）的事故复审方法论，来指导事故复审使之更为有效。

检测　故障是如何被发现的？告警、仪表盘，还是用户报告？

扩展检讨　与该故障关联的各类人员是否迅速介入到故障的分析和处理？这些人员是否是通过告警信息自动加入，而不是依赖于人工通知？

补救　修复故障时，采取了哪些步骤？这些步骤能否实现自动化？

预防　哪些改进能够避免此类故障再次发生？在故障再次发生时，如何使其更“温和”一些？如何进一步缩短故障恢复时间，以降低此类故障带来的负面影响？

DERP 帮助我们明确了分析故障的步骤。在这种分析方法的帮助下，即使无法避免此类故障的再次发生，至少在下次也能够更快地修复故障。

快速行动的同时避免二次破坏

“快速行动”与高可靠性之间并没有冲突。为了协调这两个要求，脸谱的基础设施提供了很多安全措施：配置系统能够防止错误配置的快速部署；核心服务向客户端提供强化的应用编程接口以防止故障的发生；核心库避免请求延迟发生时耗尽系统资源。为了处理其他不可避免的可能导致系统崩溃的故障，我们构建了简单易用的仪表盘和诊断工具，帮助工程师通过分析问题，找出可能引发故障的最近变更。最为重

要的是，在故障发生后，通过事故复审吸取教训，帮助我们构建更加可靠的计算基础设施。■

参考文献

[1] CoDel (controlled delay) algorithm http://queue.acm.org/detail.cfm?id=2209336.

[2] Cubism; https://square.github.io/cubism/.

[3] HipHop Virtual Machine (HHVM); http://bit.ly/1Qw68bz.

[4] Thrift framework; https://github.com/facebook/fbthrift.

[5] Wangle library; https://github.com/facebook/wangle/blob/master/wangle/concurrent/Codel.cpp.

作　者：

本·毛雷尔(Ben Maurer)：脸谱万维网基础团队技术主管。负责脸谱面向用户产品的整体性能与可靠性。在加入脸谱之前，与路易斯·梵阿纳(Luis von Ahn)共同创立了reCAPTCHA公司。

译　者：

杨任宇

CCF学生会员。北京航空航天大学博士生。主要研究方向为大规模分布式系统、系统资源调度与容错技术等。

yangry@act.buaa.edu.cn

欧阳晋

阿里巴巴云计算公司高级研发工程师。主要研究方向为大规模分布式计算系统、资源调度与管理技术。

jin.oyj@alibaba-inc.com

胡春明

CCF专业会员、本刊编委。北京航空航天大学副教授。主要研究方向为计算系统虚拟化、分布式系统等。

hucm@buaa.edu.cn

CCF@U: CCF走进高校（2015年）

序号	演讲人	时间	高　校	演讲题目
345	罗　训	11月24日	河北工程大学	看得见的计算之美
346	罗　训	11月24日	河北师范大学	看得见的计算之美
347	罗　训	11月25日	河北经贸大学	计算机工作者的开源小时代
348	孟小峰	11月26日	河北农业大学	大数据管理及在线聚集分析
349	郑　宇	12月10日	同济大学	Trajectory Data Mining
350	姚期智、杨士强	12月10日	上海交通大学	Quantum Computing--A Great Science in the Making
351	陈益强	11月17日	湖南理工学院	面向 E-health 的可穿戴计算
352	陈振宇	11月30日	南通大学	工业驱动的软件测试研究
353	公茂果 苗启广	12月5日	华南理工大学	多目标深度神经网络与稀疏特征学习 基于非凸抗噪声损失函数的鲁棒 Boosting 算法
354	杨士强	12月24日	河北大学	互联网时代，如何做到时随心不随
355	何万青	12月26日	西北师范大学	青年学子核心竞争力培养
356	吴文峻	12月26日	湖南大学	基于 MOOC 的实验教学和行为数据分析
357	苗启广	12月28日	太原师范学院	社交网络大数据分析及应用
358	杨士强 李国良	12月29日	北京理工大学	漫谈论文质量与学术评价 做科研的一点体会——打通任督二脉