Notes on Distributed Systems for young blood

原文地址：<https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/>

原作者：Jeff Hodoges，原Twitter架构师

作者博客：<https://www.somethingsimilar.com/>

作为系统工程师，我已经考虑很久去分享一些在实际工作中所学到的知识，这些都是我们在生产环境中趟过一个又一个坑所换来的经验教训。它们有非常好的警醒作用，当然，如果有人能自己全部经历一遍会有更好的效果。

很多新手会将[Fallacies of Distributed Computing](http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing)（介绍连接：https://en.wikipedia.org/wiki/Fallacies\_of\_distributed\_computing）和CAP理论作为他们的座右铭。但是这些单纯的理论上并不能有效的帮助他们迈出分布式系统领域的第一步。当他们迈出第一步的时候，才会惊讶于自己的无知。

下面是我在成为一个分布式系统工程师所必须的知识清单。对于一个准备踏入这一领域的新手来说，会是一个不错的向导。这其中有一些很琐碎，有一些很令人惊讶，但是他们都是不可置疑的。这份清单简明的解释了他们即将踏入的领域，虽然不够全面，但会让他们迈好第一步。

这份清单最有意思的是在聚焦技术问题的同时，也列举了一些一个工程师可能陷入的社交问题，自从分布式系统变得越来越庞大，工程师们也不得已的面向更多更庞大的团队和组织。这种社交问题通常是很多开发者工作中最难处理的问题，尤其是大型分布式系统中。

我们所受的教育、经历和职业背景更趋向于培养我们成专业技术人才，良好的社交环境会使我们更高效、更愉悦～所以我们必须去正确面对它，人远没有程序那么严苛，尽管他们连一个标准的抽象接口都没有～

Ok～ 开始进入正题

## 一．分布式系统千奇百状，因为他们经常出错：

当问及是什么把分布式系统工程师和其他软件领域的工程师割裂开时，新手常常会回答是延迟，并且深信它就是分布式系统困难的罪魁祸首。

他们错了，区分出分布式系统工程师的大概是“错误”，或者“一部分错误”。例如一个互斥锁unlock失败，可以假定unlock的进程不稳定并且让他退出。但是一个分布式的互斥锁的unlock操作，必须要内置到他的锁协议里。

那些没做过分布式系统的工程师经常会有这样的想法：“这样会向两台机器同时执行写操作”或者“如果写入失败了，就要不断的重试直到成功”。这些人一定没有意识到网络系统的异常要远多于单机操作，并且这些错误只是影响了系统的一小部分。多个写操作可能只有一个成功，如何保证数据的一致性？这些局部错误往往更难找到失败的原因。

交换机或者全GC让服务仿佛消失一般；socket写操作看起来成功了，实际在另一台机器上失败了；一台机器上的磁盘驱动过慢导致整体集群的通信像龟速一样，等等。在本地内存读写操作要远比在距离几个交换机的另一台机器上可靠的多。

所以，要为容错而设计

注：分布式系统-容错性 <http://wrran.com/blog/2017/12/28/review/distributed-system/8-fault-tolerance/>

## 二．构建一个高可用的分布式系统的费用，要远大于构建一个单机高可用统

构建一个高可用的分布式系统需要的钱远比一个单机系统多的多，就是因为机器越多，带来的错误就越多。虚拟机和云技术使得分布式系统构建成本变低，但这可不代表着你可以像单机系统一样去设计和测试。分布式系统的错误情景在单机系统很难复现出来。无论是远大于单机系统的数据量，还是数据中心的网络环境，分布式系统都需要真正的分布（模拟远不够）才能够暴漏出他的bug。

## 三．开源的分布式系统的健壮性，远小于单机系统

长时间运行分布式系统的集群对开源社区来说是一个巨大的负担。爱好者和外行们是开源软件的引擎但是他们没有足有的经济基础去测试、解决分布式系统的诸多问题。爱好者通常是利用空闲时间去，在他们仅有的机器上写开源的代码，仅仅是为了乐趣。这就意味着很难找到，愿意花费大量金钱去维持、运行大量的机器。

这种松懈的态度甚至影响了很多企业工程师的工作态度。但是开源社区的的需求优先级和你所在的公司的需求优先级是不同的！

当然很多开源团体已经意识到这个问题，但是仍然没有彻底的解决。

## 四．协调通信非常困难

无论何时尽可能避免机器之间通信。通常这被解释为“水平扩展性”。水平扩展的真正秘密就是独立：机器之间如果需要通过通讯以达成共识，就需要将数据传递到每一台机器上，这样这批机器就变成最小单元。每当两台机器对第三方服务形成依赖的时候，这个服务就变得很难实现。信息是有传递速度的上限，并且网络通信要比你想象的脆弱，所以你大部分关于一致性的想法都是错的。学习一下“两将军问题”和“拜占庭将军问题”对你会很有帮助。（Paxos算法确实非常难，以至于那些技术大牛们也并不比你懂得更多）

注：paxos算法简介：<http://www.solinx.co/archives/403>

## 五．能在内存解决的问题，就都不是大问题

对一个分布式系统工程师而言，单机上的本地问题非常简单。相比于在指针之间，在交换机上的数据传输更难被梳理清楚。任何已经落于纸上的提高效率的小技巧在计算机科学伊始就不再奏效，更别提在分布式系统中了。大部分论文、可实现的算法，都是基于单机系统的，这一数量在分布式系统中大幅减少。

## 六．“太慢了”将会是你最难解决的问题

“太慢了”一般指一个用户的请求在通过一系列的服务后响应太慢了。这也许意味着一台或多台机器之间的传输通道非常慢。“太慢了”很难解决，因为他并没有准确指出到底是哪里出了问题。那些导致不符合你预期的异常很多时候都潜伏在黑暗的角落里。并且直到这些问题爆发出来的时候，你都不会有足够的资源（时间、资源、工具）去解决它。Dapper 和Zipkin就是为了解决这个而诞生。

# 注：以Dapper、Zipkin和LightStep [x]PM为例阐述分布式跟踪的过去、现在和未来：<http://www.infoq.com/cn/articles/distributed-tracing-microservices>

## 七．背压机制要贯穿你的系统

背压是一个提供服务方，向一个发起请求方返回的一种失败信号，并且发起请求要处理这些失败以防止对自身和服务方造成过载。背压设计，代表着在过载和系统故障时限制资源的使用。这是设计一个高可用分布式系统的基础模块之一。

背压的实现，通常意味着在资源发生限制或者系统故障的时候，需要将新请求数据落地，或者将错误信息返回给调用方（或者两者都实现）。对其他系统的连接和请求能够快速的回馈超时和错误非常重要。

如果没有背压机制，可能会发生多层级的故障或者无意义的信息丢失。当一个系统不能很好的处理其他系统的错误信息，那就导致依赖它的系统接收到大量的错误信息。

注： 背压机制与flow control： <https://www.cnblogs.com/iceTing/p/6238207.html>

## 八．要找到局部高可用的方法

局部高可用就是要在系统部分功能挂掉的时候，能够返回一些信息。

搜索系统就是探索这个问题的绝佳案例。搜索系统一直在结果质量和响应时间之间做平衡。典型的搜做系统都会在设置一个搜索限制时间，如果在搜索全部信息之前达到了限制时间，无论结果是否完整都要返回结果。这就保证搜索系统在面对访问量间歇性波动的时候更容器扩缩，并且和那些导致无法搜索完整数据集的错误，保持统一的处理方式。系统既返回结果集的部分数据给用户，又提升了系统的弹性。

同理，我们考虑下web应用中个人信息的问题。在某种情况下，无论你做了什么，用于存储用户个人信息的设备都会在同一时间宕机并且用户会有所感知。那么在这样的系统中，我们可以允许什么样的局部失败？

这引发了很多的思考。相比于所有信息不可用来说，用户通常更能接受部分数据的丢失（最好是其他用户的数据）。如果服务过载了，或者集群中某些机器宕机，相对于全体用户集来说，更小的用户群数据丢失更可取。而且除此之外，我们不希望一些无关的功能，例如公共图片下载模块，因为个人信息有问题而受到影响。想想我们要做多少工作才能将这些模块拆分开并加以维护。

能够提供系统局部高可用性的设计方案，会是你个人技术栈的重要组成部分。

## 九．唯一能够衡量你工作完成与否的是各项指标

各项指标（例如特定action的延迟、 增长率和变化率）是唯一能说明系统预期的表现，和它实际表现的差异。能够了解系统在运行20天和15天的区别，就是一个优秀工程师和一个新手的重要区别。然而，指标虽然是了解问题和行为的必要条件，但并不能告诉你接下来要怎么做。

就日志来说，日志文件非常有必要，但是他们更容易产生误导。例如，通常记录几个简单错误就占据了日志的大部分空间，但实际上只有在很少的请求中触发这个问题。因为在大多是情况下成功的日志是多余的（因为记录成功的日志会导致磁盘问题）。并且因为工程师经常猜测哪种情况下的错误信息才是有用的，导致日志经常被奇怪的小段落填满。所以在记录日志的时候请假设下，会有一个从来没看过代码的人来看日志。

我亲眼见过很多工程师过度的依赖于在日志中看到的信息，而没有第一时间查看是否和各项指标相悖。我还经常看到有工程师在一整段的日志中，像福尔摩斯一样分析错误的行为和原因。但是请记住两点：1. 要记住通过日志就判断正确的情况比较少见， 2. 你并不是福尔摩斯，除非各项指标或者事实证明了你的观点，

## 十．使用百分比而不是平均值

在大多数的分布式系统中，百分比（50%， 90%， 99.99%）比平均值更能提供精确的有用的信息。使用均值就假定了系统的各项指标是符合正态分布的，事实上，这不是工程师关注的重点。“平均响应时间”是一项常用的指标，但是我从没有见过哪一个分布式系统的响应时间是符合正态分布的。如果某指标不符合正态分布，那么平均值就没有什么意义，甚至会导致不正确的判断和误解。避免这种陷阱应该使用百分比来记录。这样你就能更改清楚用户是怎样看待你的系统的。

## 十一. 学会预估系统承载能力

因为这点你需要了解一天又多少秒。了解在一个持续运行系统中，与运行三个月后才能更换机器时， 你要运行一个任务需要的机器数量的区别。甚至在开发完成之前再更换。

设想下推文服务。在一台普通的机器上你能处理多少的推文id？那么，在2012年底一台典型的24GB内存的机器，你需要减去系统所占用的4-5GB内存，同时至少需要处理一对请求，并且推文id为 8 位，能处理多少推文id呢？这是除去你正在做的事情，背后需要进行的计算。Jeff Dean的 Numbers Everyone Should Know 是一个很好的教程。

注：Numbers Everyone Should Know： <http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf>

## 十二. 基础设施功能是逐步提供的

Feature flags是产品工程师在现有系统中推出新功能的常用方式。他通常与前端A/B测试相关联，用于向特定人群显示新的设计或者功能。但是他们同样适用于基础设计建设。

很多项目的失败是因为他们追求“大而全”或者一系列的“大而全”，这通常导致他们太晚的发现bug而不断的回滚。但是通过“feature flag”方式，你会对你的项目产生信心，并且降低失败的成本。

假设你正在将单个数据库，转变成屏蔽细节的服务，通过feature flag，你可以写入旧的数据库的同时，缓慢的写入新得服务中。以此作为过渡来保证写入的准确性和写入速度。当写入进度达到100%并且全部落地到新的数据存储区完成的时候，你可以使用单独的feature flag从新的服务读取数据来测试性能问题，而不用通过用户请求响应的数据。另一个feature flag可以用来读取新老系统的数据以校验数据正确性和完整性。最后的feature flag可以用来缓慢的提升新系统的读写数据的性能。

通过将新功能拆分成多个步骤，并且通过feature flags来保证快速的部分功能实现，你能够在bug出现的时候，更轻松的发现和解决他们，而不是面对宇宙大爆炸似的bug喷发。当问题出现的时候，你可以快速的将相关的feature flag回滚。通过步骤拆分，您可以在互不影响的流量下进行debug和测试，因为你知道任何你造成的问题，都不是毁灭性的。通过feature flags你也能够选择其他的迁移策略，例如基于某个用户的请求进行迁移，这样能够更好的洞悉新系统的一切。当你的新服务只是一个雏形的时候，你也能够通过feature flags来降低新系统占用的资源。

现在，feature flags对于传统程序员或者新的工程师，听起来会造成一定的混乱。使用feature flags意味着要处理基础设施的多版本控制，并且版本要成为一种规范。这是一门很深的学问。谁让单机系统上的运行方式在分布式系统之中玩不转呢？

Feature flags最好理解为，为实现全局的简单和弹性，与本地复杂性的一种权衡。

## 十三. 明智的选择id命名空间

你选择的id的命名空间会限制你的系统。

获取一份数据所需的ID越多，数据分区的选择就越多。获取一份数据所需的ID越少，就更容易消耗系统的性能。

参考Twitter API的第一版。所有每一条推文的查询、创建、删除都是根据简单的数字类型的id。推文的id是64位的数字没有引入其他的数据类型。随着推文数量的增加，很明显的意识到同一个用户的数据都集中在一台机器上的时候，创建用户推文和其订阅其他用户的timeline会更加快速。

但是Public API 要求每个推文都能够根据推文id进行寻址访问。为了以用户维度进行拆分推文，就必须要构建一个新的查询服务。这就意味着必须要知道每一个用户拥有哪些推文id。如果必要的话，是可以做到，但是需要花一点成本。

一种替代的API是需要每条推文查询的时候都提供用户的id，起初推文的id只是用与简单的存储，直到以用户为维度进行数据分区。另一种替代方案是推文id本身就包含用户的id信息，这就是意味着要以id不再是k排序和数字类型为代价。

要小心在id中包含信息，无论是明确的还是隐藏的。客户端也许会利用你id的结构去解析用户隐私数据，爬取你的系统数据（自增id就是典型的例子），或者是其他各种方式的攻击。

## 十四. 利用数据的局部性

数据的处理和缓存越接近持久化存储，处理的速度就更有效，并且会更统一保证缓存数据的正确性和访问速度。网络相比指针引用，文件读取又更高的出错率和延迟。

数据的局部性不仅是空间上的，同样也是时间上的。如果大量的用户在相近的时间发起了相同的请求，也许他们的请求可以合并成一个。又或者大量的请求访问邻近的数据，他们可以合并成一个更大的请求。这样做通常会降低数据的交互频率，并且更方便做故障管理。

## 十五. 先写缓存数据，再写持久化数据是错误的

这种情况要比你想象的多。尤其分布式系统的初始设计人员是一个缺乏经验的工程师。很多你接手的系统都会有这个问题。如果系统的构建者在谈论“Russian-doll caching”,那么你就有很大的几率面对各种问题。这个问题可能已经不复存在，但我仍然对它深恶痛绝。这个问题的常见表现就是用户信息神秘更新到系统的旧值当中（例如，昵称，邮件地址，密码等）

注：缓存更新策略总结: <https://coolshell.cn/articles/17416.html>。

## 十六. 计算机比你想象的更强大

今天在这个领域，会有很多没有足够经验的从业者，提供了很多关于计算机可用性的错误信息。在2012年低，一个六七个人开发的轻量级的web服务，24GB内存和一定的磁盘空间就足够你使用。一个复杂的CURD（Crud是指在做计算处理时的增加(Create)、读取查询(Retrieve)、更新(Update)和删除(Delete)几个单词的首字母简写。crud主要被用在描述软件系统中数据库或者[持久层](https://baike.baidu.com/item/%E6%8C%81%E4%B9%85%E5%B1%82" \t "_blank)的基本操作功能）应用，在单机上基于当今编程语言的runtime，可以轻松处理每秒几千的请求。而且这只是一个下线。所以大多数情况下单机处理每秒几百个请求根本不值得吹嘘。

更高的性能并不难，只要你愿意根据你针对系统的分析结果进行优化。

## 十七. 用CAP理论来评判系统

CAP理论是你构建一个系统必不可少的一条。他不是你可以作为基础理论并且推导出一个可用系统的原理。它的范围太笼统，可用的结局方案也太多了。

然而它很适用于评判一个分布式系统的好坏，并且使人明白要做出什么样的权衡。设计一个系统、并且在每一个子系统的迭代中贯穿CAP理论最终一定帮你完成一个更好的设计。作为家庭作业，实现一个满足CAP理论的Russian-doll caching。

最后一点建议：排除C，A，P的一个，你不要选择CA。

注：从ACID到CAP/BASE: <https://www.jianshu.com/p/68c7c16b3fbd> .

## 十八. 拆分服务

服务意味着“一个层次高于存储系统，并且典型的是提供请求和响应API的分布式系统“。如果代码存在于单独的服务而不是整体的系统中，更容易注意到代码的修改。

拆分的服务通常提供了封装好的lib库的好处。拆分出的服务更快更简单的部署和更改其提供的lib库，而不是升级其客户端的所有lib库。（当然如果拆分出的系统不能够轻易的部署升级，那么就需要依赖他的系统能够做到这点）。这种简单性，是因为拆分出的服务有更少的code和操作依赖，并且它创建了严格的系统边界，使得库的使用没有什么捷径。捷径通常导致很难做系统内部的迁移或者客户端升级新版本。

协调一个库的使用成本也通常要低于维护分布在各个子系统中的lib库。升级一个lib库，即使API没有任何变化，都需要所有的子系统重新部署一遍。当数据发生错误的时候要保证系统按照顺序进行部署就变得异常苦难（通常你根本无法意识到这个问题）。如果各个系统的拥有不同的负责人，那么升级所有客户端带来的社交成本就更加恐怖。通常他们和你的优先级不一致，所以了解升级的必要性并且愿意去升级就更难了。

典型的是一个屏蔽存储层信息的服务要进行变更的时候，拆分出的服务比起其面向的存储服务，对上层的切面更小。通过拆分服务，客户端系统不需要知道，底层系统迁移到新系统的数据格式和复杂性，只需要新的服务自己测试自己本身的各种可能出现的错误。