突破技术领导力

来自硅谷CTO实战秘籍

Martin Abbott

软件架构大师

eBay前CTO，《架构即未来》作者

硅谷顶级公司技术管理实战心得

十年精品课程内容独家集萃

技术领导者成长宝典

突破技术领导力

来自硅谷CTO实战秘籍

Boolan
高端IT互联网教育平台

目录

第一部分 架构

第二部分 领导力

第一部分

架构

从HARVEY和IRMA飓风中得到的教训

我们每个人都对这场风暴在休斯顿和佛罗里达造成的可怕破坏感到悲伤，包括人员和财产的损失。飓风的影响可以通过提前在高危地区进行通知和储备物资准备，以及对后续重建做分级响应，来被有效的限制。

数据中心的日常运营也应该有类似的准备思路：使用柴油发电机来保证能源供应不会中断，电池来保证服务器在过渡过程中的能保持运行，有可能的话，储备水或水井来替换公共用水用来为服务器冷却，以及事务和饮用水来供应无法离开的保障人员。

当一个准备妥当的地点，例如数据中心，遭受飓风带来的强风，大雨和洪水时，会发生什么？在某些情况下，数据中心不会受到影响，尽管仍然会有一些服务中断的情况。

数据中心可能存在的故障点

让我们来看看如何使数据中心的功能不受影响。在数据中心选址时，进行大规模的风险调查。数据中心的建造要花费巨大的经费，其失败所造成的潜在业务影响，可能因为后续的合同违约而及其昂贵。这些因素使得数据中心应该坐落于不会受到洪水影响的地方，远离有害物质的运输路线，以及足够坚固以承受该地区可能发生的风暴。

对于数据中心失去电力，应该用“何时”而不是“如果”的方式来评估（无论是停电还是计划中的维护活动），柴油发电机是一种非常普遍的解决方法，且要准备24小时以上的燃料以及多个备用补充方案。数据中心应该能够在没有电力支持的情况下运行数日后数周，在一些情况下甚至是数月。洪水是如何影响数据中心的？数据中心的服务器入口通常是电力接入口，且被埋在地下。电力供应在洪水中极有可能中断，或被当地政府因为安全原因关闭。数据中心如果没有被洪水影响，应该能够通过发电机运作，尽管这种情况下燃料补充会变得很难。如果主电力室积水超过两英尺，数据中心将会停止运行。

许多大型数据中心通过蒸馏水来冷却服务器。蒸馏水冷却大部分情况下是一种极为有效的冷却方式，但却由于供水原因存在潜在的风险。在许多地方，

公共供水会因为电力短缺而中断。而数据中心可以通过使用水箱储存水或水井来缓解这类风险。就像发电机的情况一样，数据中心应该能够在没有公共供水的情况下运行数小时到数天，在结构上要足够坚固以应付风暴。还有其他的风险吗？网络连接和带宽。

大多数数据中心需要与其他数据中心进行通信，以实现其OLAP或OLTP用途。没有网络连接，服务器将不可用。数据应该是无碍的，但它会变得越来越陈旧。交易和流量也会完蛋。就像电力供应一样，网络连接也通常埋藏于地下。由于距离和地理限制，洪水可能会影响网络路线，数据集中和传输设施。电信设施通常会有发电机等其他保障设施，但可能会被迫转移到不太有利的地方，运营时间也会缩短。

对于可用性十分谨慎的数据中心，往往通过负载多样性和物理连接多样性来减轻载体中断和“backhoe fades”。 这种方式在在类似大规模洪水的情况下会很有帮助。事实情况是，一个无法连接到的数据中心是没有用处的。所有关于建筑设计，能源短缺，冷却问题和消防保护的防灾措施，在链接中断时都是无意义的。

为必然做准备

减轻以上风险最好的方式就是不要依赖于一个单一地点的数据中心。当一个数据中心故障时，另一个可以顶上。托管或云服务能够在单个地点损失的情况下幸存。飓风会造成区域性的影响，例如Irma扰乱了佛罗里达的大部分地区。过去几年，许多公司决定建造两个相聚20英里以内的数据中心来支持同步数据备份。例如纽约地区的一个主要站点和其他地区的一个灾害恢复站点。复制选项和数据库管理技术进步到今天已经足以支持更广的分散。通过选择不同地区的数据中心来避免区域性的影响。

从3个地点开始运营可以比2个更便宜，还可以通过从最近的位置服务客户，以更少的响应时间来提高客户满意度。从多个位置进行操作还可以方便的调整各个位置的冗余度。不同位置的组合能够带来更经济的运营。

读书笔记：

AI机器人是如何击败世界顶级游戏玩家的？

上个月，由Elon Musk一手创办的OpenAI公司开发的AI机器人击败了世界最顶级的Dota 2前职业选手。对于AI和机器学习领域来说，这无疑又是一个新的里程碑，同时对于当下热门AI的讨论又添了一把火。而在开始讨论前，我想先介绍一下这次讨论所需要的背景知识。

传统编程的进化

大多数电脑编程可以被概括为三个步骤：1.读取数据 2.操作数据 3.输出结果。

想象一下，比如你周末要飞去某个地方。你可能会打开你的旅游App，然后输入一些信息，如时间、人数、机场等等。然后App会利用这些数据查询数据库中满足条件的航班，最后在你的App终端上显示你需要的航班信息。

这种软件设计之道自编程创立开始，就一直是不变的基准。而人工智能，特别

是机器学习已经另辟蹊径。但在第一步（读取数据）和第三步:（输出结果）上是一致的。

人工智能在第二步（处理数据）上与之前已不再一样。例如还是找航班，开发者很容易通过读代码来理解计算机被编程去执行的一系列操作。如果开发者想要改变或者提升程序性能，那么只需要添加一些代码或者改一下代码即可。例如，你想在终端上在附件时间段比较不同航班之间的价格，开发者只需要改几行代码就可以做到。可以说代码指引计算机一步一步直到输出最后的结果。换而言之，程序只会做代码告诉它做的事，因此不会自己多做也不会少做。

与之相比，机器学习的输出结果不再依赖于编程代码所编写的结构。当有新的修改需求时，开发者无法读取到对应的代码并且修改。输出的结果只取决于程序中的神经网络。

运行中的神经网络

神经网络是什么？它的核心是神经元。与传统编程一样，神经元需要对它输入数据，然后针对数据进行数学运算，最后输出结果。比较典型的神经网络的神经元会收到成百上千个数字作为输入，通常会是由0或1的数字组成。单个神经元会将这些数字乘以权重再求和，得到结果。许多神经元会将这些结果再转化为一个0或1的数

字。然后将结果依次发送给下一个神经元，直到到达最终的输出神经元为止。

这里以一个神经元的数学计算为例，假设x1、x2、x3表示输入数据，w1、w2、w3表示存储在神经元中的权重，那么神经网络中的神经元的计算就会诸如：x1*w1+x2*w2+x3*w3....

你也可以换种角度去理解神经元内部的计算过程：神经元会读取一批次的数据，而神经元中的权重决定了输入数据中的重要性。如果输入不重要，这个输入的权重也将接近于0，因此不会传递到下一个神经元。所以说权重有效地决定哪些输入是有价值的，哪些输入是可以摒弃的。

在神经网络中，像我之前所提到的神经元会以平行并排的方式组合成神经元矩阵模型。这样向神经网络输入的数据会并行地进入成百上千个神经元，每一个神经元都具有不同的权重。这些神经元的输出会被传入到下一层的神经元，通常会有多层的深度。因此会被叫做深度神经网络。换种方式看待神经网络，它们是以行和列的方式组合的神经元矩阵，神经元之间相互连接着。而神经网络的最终层就是输出层。所以最后的输出结果是由网络中的神经元经过成百上千万的计算得出的。

开发者在软件中编写神经网络时，每个神经元的权重会以随机值来进行初始化。也就是说，此时会随机地减少、增加数据的权重或者直接抛弃某些输入数据，神

经网络的输出值也会变成一种随机值。然后，通过一个名为“训练”的过程，权重会从初始随机值一点点变化为可以产生可用输出结果的可用值。

训练是一个耗时而又复杂的数学计算过程。这就像我们不断训练，然后可以做得更好。例如，我们想学习用弓箭来进行射击和瞄准，我们会先捡起弓箭，然后对准目标，拉满弓弦然后放箭。以我来说，通常会射不中，因此我会不断地练习，通过我与靶子的距离和方向来不断纠正我的射击成果。

在神经网络中的训练中，每个神经元的权重都会轻微地变化来提高输出的质量，就如提升射击一样。最普遍的产生这种纠正变化的方法叫做反向传播法。反向传播法是一种数学方法，适用于任何神经元的权重。在训练中，从神经网络输入到输出，然后这个输出值会和期望中正确的输出值进行比较之间的误差，通过这种误差，反向传播法会改变每个神经元的权重来减少这个误差。如果训练过程和反向传播的过程一切顺利，最后输出的误差会不断减小直到它的结果到达专家级或更好。

AI vs 人类

以最近OpenAI的机器人击败世界顶级Dota2玩家为例，它的输出，就是一系列的操作步骤，策略和决定。它也是从一开始的随机操作步骤进步为后来足以轻松击败世界顶级前职业玩家的出色机器人。胜利的关键就是存储在神经元中的权重和神经网

络架构自身。

有趣的是创造Dota2机器人的神经网络的程序员是否能理解机器击败人类玩家的一步步的操作？答案是不行。程序员只能看到关乎输出结果的神经网络的那一部分，但是无法解释为什么机器人会采取那些特定的步骤来制定它的行为和策略。所有的程序员只能通过那个巨大的权重矩阵来解释这些。

另外一个有意思的问题是程序员用传统的编程方式写的程序也可以一步一步击败Dota2职业玩家？答案也是不行的。由程序员专门编写指引的计算机程序，很容易就会被专业玩家打败。但神经网络可以通过训练来学习那些开发者也没有的知识，把学习成果存储在神经元中，然后利用它来击败人类。

让这款Dota2机器人变得与众不同的是它通过与自己对战来学习如何击败职业玩家，而不像大多数其他机器学习程序那样通过程序员给予的数据来训练。对于机器学习，好的数据就像黄金一样，非常的稀有和昂贵。(标注:这也是为什么Google和大型技术公司会收集大量数据)。数据被用于训练神经网络来做到很多有意思的事情比如说在图片中识别人物和地点或者是从你们一家人的声音中识别出你的声音。OpenAI的机器人除了自身团队提供的一些教练外几乎都是与自己在对抗，OpenAI已经很清楚地展示了学习的过程不会耗费成吨的数据，就像自己可以生产黄金一样。

那么OpenAI机器人的这次发展，是否意味着其它机器人以后可以通过与自己对抗来变成更强大的机器人呢？答案是否定的。但是这确实说明人可以通过编写两个不同的机器人彼此对抗来变得更加强大。这其中关键性的推动者是我们自身。大家可以想象什么样的机器人可以通过这种方式来进行发展，有些机器也许会有用也许也会产生危害。显而易见的是，游戏机器人并不会产生什么危险，除了意外地给一些游戏中的玩家造成不好的体验。但是，不难想象有些超级机器人仍然很危险，也许你可以想象训练一个机器人与自己练习美式足球，直到它的技术超越了职业选手并且在策略上超越所有的NFL（美国国家足球联盟）教练。接着会发生什么？答案是毁灭般的影响。你能准备好接受这样的事实吗？

AKF(ABBOTT的公司)合伙人们建议那些董事会和高管，在AI技术即将出现他们各自市场的情况下，带着团队去发现创新源和将毁灭的模式。沃尔玛已经在店中使用面部识别技术来判断在付款时消费者是否对购物满意。这也许比起亚马逊会带来潜在的优势？那机器学习和AI如何为支付系统防御诈骗或者是预防洗钱呢？

读书笔记：

何时选择切分你的服务

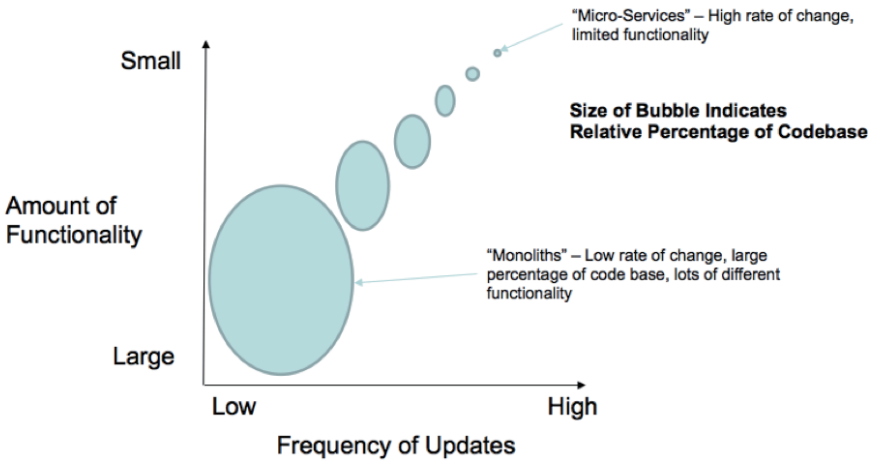
根据AKF伸缩性立方的Y轴数据显示，发展中的公司应该考虑将产品以服务或资源为边界进行模块切分。很多人会问我们：服务最好应该切分到什么粒度？相类似的问题还有：我们的程序应该切分为多少个才合适？为了更好地帮助回答这些问题，我们会基于吞吐量、可用性、可伸缩性和成本，列出一系列的考虑因素。只有在这些因素下，你才能决定程序是否应该被组合成一个大而整的代码库，还是切分为小而单独的服务和泳道。但你得牢记：过分的切分可能会带来高额的成本和低微的回报。而那些发展停滞的公司更应该聚焦于如何开发市场化的产品而不是根据下面这些考虑去调整他们的服务规模。

开发者的吞吐量

修改频率：如果某个服务需要在大型代码库上进行频繁修改，通常这会引起代码资源的竞争，并且会浪费一部分时间去解决不同团队之间的产品合并的冲突。这样高频变化的服务应该被切分成细粒度的服务，并且被安置在自己独立的错误泳道，这

样的话即便频繁更新也不会影响彼此的服务。而低频变化的服务应该被合并起来，因为即便分离这种服务也没有任何价值，而且在更新这些服务也几乎没有什么风险。

下面的图表表示了我们所推荐的关于功能、更新频率、代码库相关百分比之间的关系。你的高风险、商业关键服务应该驻留在右上部分，这部分应该由小而专的团队来更新。而低风险并且很少更新的功能应该被整合成大而整的服务，如下图：



复用程度：如果库或者服务在整个产品中有很高程度的复用，应该考虑去将它们剥离开，将其中专门针对某个功能或服务的代码分离。这样服务就可以在编译时链接，部署时作为一个共享的动态可加载库或者直接作为一个独立的运行时服务。

团队规模：小而精的团队可以胜任处理功能独立且高频变化的微服务，或者是

非常多功能但低频变化的服务。这会给他们更好的归属感，增加专业化程度，让他们
非常多功能但低频变化的服务。这会给他们更好的归属感，增加专业化程度，让他们
自主工作。另外团队的规模也决定了是否应该切分服务。越大，团队的协调成本也会更高，也就越需要切分团队来减少代码库的冲突。在这种情况下，我们会基于缩小团队规模这个因素来进行切分产品，以减少冲突。而理想的切分是基于增加可用性、可扩展性、减少需求的开发时间来拆分。

专业化的技能：一些服务可能需要一些和其他团队不一样的特殊的开发技能。例如当需要使你的部分产品运行速度更快，那通常需要掌握编程语言并且对算法和渐近分析有很深入的了解。擅长这部分工作的工程师比起代码库的其他成员会有完全不同的技能树，这完全可以解释因为他们更聚焦于用户交互和体验上。再比如，你的产品的一部分代码需要很深入的领域知识比如支付领域。这些例子都需要考虑去切分服务并且标明该服务的规模。

关于可用性和容错性的思考

理想的可靠性：如果服务产生故障，且某些功能可以承受被其影响，那你可以将其组合为一个大服务。实际上，当某个功能产生故障，有些功能也不应该工作（例如，如果无法了解有多少可供交易的股票，则不应该在股票交易平台进行交易）。但如果你需要每个功能相对其他功能是独立的，那么你就需要把他们切分为单独的服

务。

业务的关键性：这取决于这项服务对于你的商业价值的创造有多大的重要性，同时也要考虑到服务的可用性。一种了解重要性的方式是衡量一天停机所产生的一天总失。如果企业无法承受服务的故障，那么就切分服务直到这个影响是可以被接受的。

故障风险：确定服务的不同故障模式（例如计费服务收费错误）每种故障模式发生的可能性和严重程度是多少，发生故障的可能性有多大？风险越高，就越应该分割。

可扩展性的思考

数据可扩展性：某个服务也许是代码库的一小部分，但是随着服务需要处理的数据量的提升，再做一次切分也是有意义的。

服务可扩展性：这项服务相对于其他服务的用户量是多大？例如某个服务可能需要在高峰时间段满足短时间的用户爆发，而另外一个服务需要满足稳定、渐近的增长。如果你切分它们，你就可以分别满足各自的需求，而不需要设计一个解决方案来同时满足两者。

对其他服务的数据依赖：如果对于另外一个服务的数据依赖无法抹除或是无法用异步调用的方式处理，那么拆分服务所得到的好处可能还不如拆分所付出的劳力。

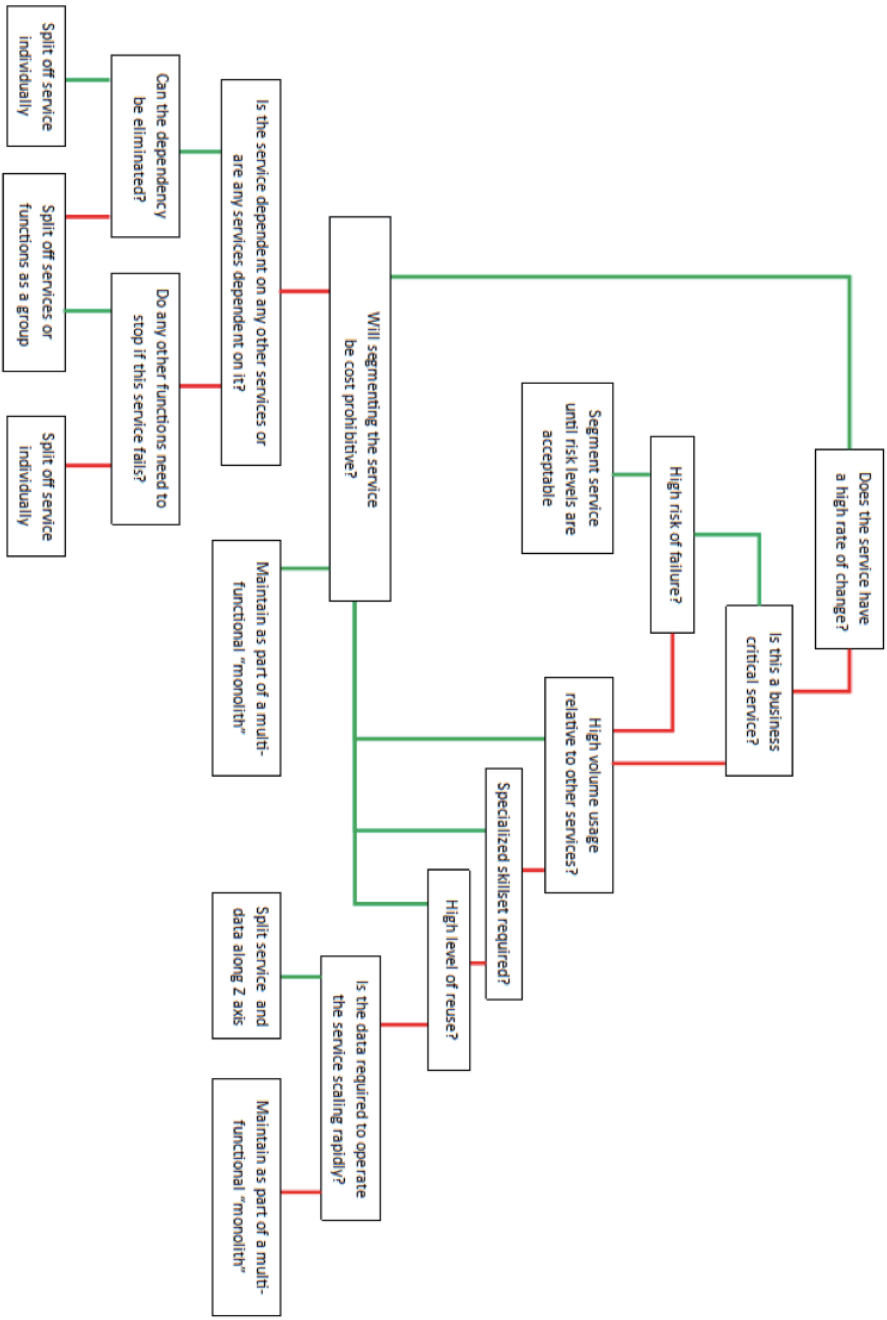
成本的思考

分割代码的劳力：如果服务的耦合度很高，可能需要几个月去拆分它们，那么你就需要决定是否这样的时间成本足够合适？你还需要考虑开发部署新服务所付出的劳力成本。

共享可持久存储层：如果在分割新服务时，仍需要依赖一个共享的数据库，那也许你无法充分得到切分服务的好处。你也可以在新的服务泳道放置一个只读数据库副本来提升可用性和性能，但它也会提高劳力和成本。

网络配置：是否这项服务需要它自己的子域名？你是否需要修改负载均衡路由或是防火墙规则？根据我们团队的专业知识来看，网络配置的变更所付出的劳力比其他因素都大。确保在考虑切分成本时算入网络配置的变化。

下面的例子可以用于快速决定是否一个服务或功能应该被切分为更小的微服务，或是将相似或独立的服务组合在一起，还是保持一个多功能、低频变化的整体项目。


Does the service have a high rate of change?
Is this a business critical service?
High risk of failure?
Segment service until risk levels are acceptable
High volume usage relative to other services?
Specialized skillset required?
High level of reuse?
Is the data required to operate the service scaling rapidly?
Split service and data along Z axis
Maintain as part of a multi-functional "monolith"
Will segmenting the service be cost prohibitive?
Maintain as part of a multi-functional "monolith"
Is the service dependent on any other services or are any services dependent on it?
Can the dependency be eliminated?
Split off service individually
Split off services or functions as a group
Do any other functions need to stop if this service fails?
Split off service individually

读书笔记：

拆库扩展

实践中我们看到的最常见的阻碍系统扩展的瓶颈就是数据库。回顾我们之前的文章“拆分应用程序或服务进行扩展”，对工程师来说，在多维数据集的X轴上创建扩展应用，即在单个大型数据库中持久化数据，并使用多个“克隆”应用程序服务器从该数据库上检索和存储数据，是很常见的手段。尤其是对于初创企业而言，这是一个很适合的方案，似乎只要实现得足够好就能消除对应用程序服务器稳定及友好方面的需求，从而提升客户使用体验。

然而，这个单一的大型数据结构有三个方面的问题：

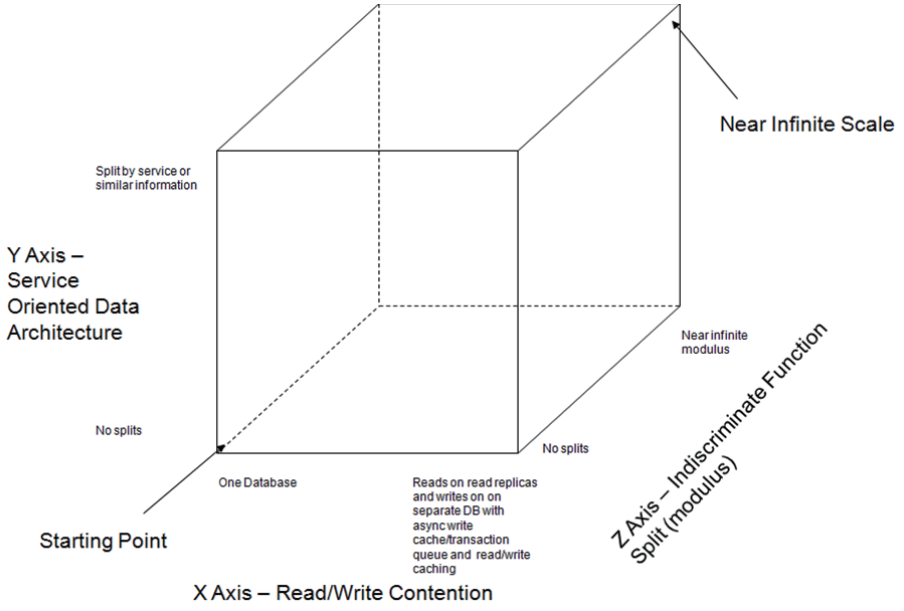
1. 即使使用集群技术（存在第二个物理系统或数据库，在第一个物理系统或数据库发生故障的情况下可以接替承担负载），主数据库的故障也将导致所有用户社区的短期服务中断。

2. 这种方法最终依赖于CPU速度、内存访问速度和带宽、大容量存储访问速度和带宽等方面的技术提升，来满足企业的业务需求。

3. 极端情况下，依靠上述的两点措施不是最经济的解决方案，因为最新和最快

的技术相对于上一代的技术来说是有溢价的，并且与同等金额的旧的或更小的（如较少的cpu数量）的系统相比，新技术不一定有相同的处理能力。

正如我们在过去的文章中所提到的，一个优秀的工程团队会考虑如何在完全依靠技术进步之前，更好地扩展平台。通过对我们以前提出的"拓展性立方"（Scale Cube）进行小小的修改，应用于分割扩展服务的问题的相同概念，同样可以应用在解决如何拆分数据库的问题上面。与AKF服务缩放立方体一样，AKF数据库缩放立方体由X，Y和Z轴组成，每个轴都采用不同的方法来扩展应用于数据库的事务。立方体的最左下角（坐标X = 0，Y = 0 和 Z = 0）表示最糟糕的单一数据库 - 所有数据位于同一数据库并且所有请求都直接访问该数据库的情况。



立方体的X轴表示通过复制多个数据库实例进行扩展负载的方法。这是大多数公司在扩展数据库中使用的首选方式，并且通常是最简单的实现方式，而且在工程时间和硬件上都是最低成本的。许多第三方和开源数据库系统具有特有属性的或功能，可以将数据接近实时地复制到多个“读取数据库”。这种方法的工程成本很低，因为通常只需要把数据库调用区分为“读取”或“写入”，并发送到对应的写入数据库或读取数据库。每个读取数据库的负载应尽可能平均分配，许多公司使用简单的第三方负载均衡器来执行此分发。

包括在我们的X轴中的解决方案为第三方和开源的缓存技术，即可以将请求在“缓存”与数据库间进行分割，防止在缓存未命中之前直接从数据库读取数据。缓存是减少数据库负载的另一种简单方法，但是在我们的经验中，这对于快速增长的SaaS站点是不够的。如果正确实施，则X轴上的拆分还可以提高可用性，由于是接近实时地复制数据库，在“写入服务器”发生故障的情况下，读取服务器可以被提升为单一的“写入服务器”。缓存和读/写拆分（我们的X轴）的组合对于许多公司来说已经足够了，但对于具有极高增长速度和大量数据保留需求的公司来说通常是不够的。

就像服务多维数据集一样，我们的数据库多维数据集的Y轴表示按功能、服务或资源的拆分。服务可能代表一组用例，通常可以考虑用动词或动作（例如“登录”）来设想，而以资源为导向的拆分是最容易想到的，比如“帐户信息”等。这些拆分不仅可以有助于处理像X轴一样的跨系统的事务分割，还可以通过将特定请求的信息保留在

内存中而不是进行磁盘访问来加速数据库调用。正如我们在扩展服务方面的建议一样，确定这些拆分的完成顺序是依据哪些是用最少的代价带给你最大收益的。这些拆分通常会给工程团队带来更高的代价，因为他们需要将应用程序也拆分开。一种可行的办法是将一体化的应用程序，进行物理上的拆分,然后通过面向服务或面向资源池的URL/URI来调用。虽然这种方法可以加速跨多个系统扩展事务处理，类似于我们X轴的实现，但它没有减少各类服务、回收池、资源、应用程序所需的系统内存等额外好处。在大型团队中考虑这样拆分的另一个原因是，专业的团队专注于特定的服务或资源，可以减少的应用学习曲线，提高质量，缩短上市时间（较小的代码库）等。这种拆分通常被称为"泳道"应用程序和数据集，特别是当数据库和应用程序都被拆分为表示"故障域"或故障隔离基础架构时。

Z轴表示通过执行查找、模数或其他不加区分的函数（例如散列）来拆分事务的方式。最常见的做法是在实体关系允许的情况下切分你的资源。在传媒业务中，你可以考虑用article_id或media_id字段进行切分，而在交易业务中，可能会用product_id字段切分。在您将客户从产品中分离出来并在客户和产品中执行拆分的情形中，你将使用到Y轴拆分（按资源或称为 - 客户和产品划分）和Z轴拆分（客户和产品的模数）。

Z轴拆分往往是工程团队执行代价最高的，因为原来在数据库中执行的许多功能（例如连接），现在需要在应用程序中实现。也就是说，可能的话，它们代表了大多

数公司最大的扩展潜力。

读书笔记：

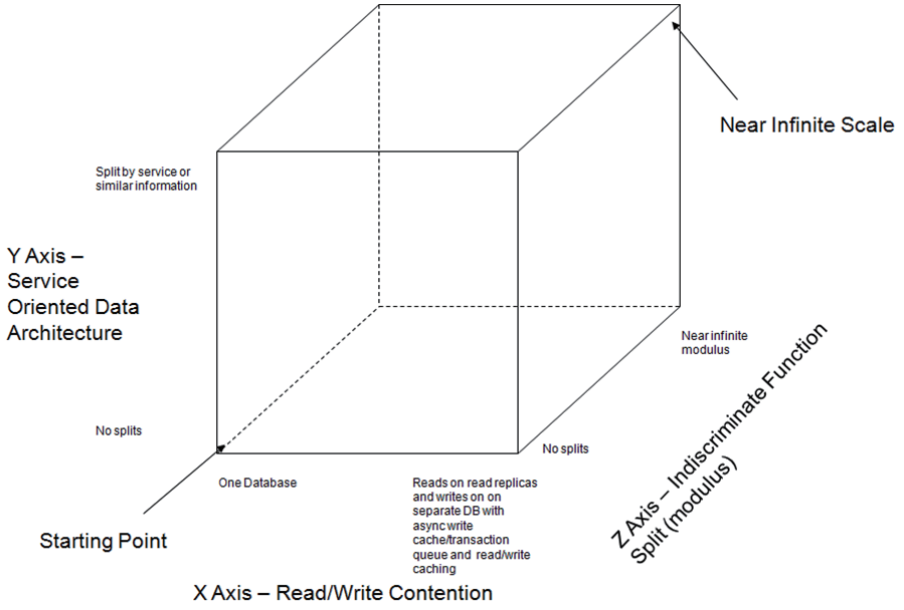
拆分应用程序或服务进行扩展

大多数互联网产品一开始，都是作为单个应用程序，运行于应用服务器或应用/网络混合的服务器上的，有的还连接着数据库。绝大多数功能都基于同一个仓库，共享物理和虚拟服务器资源：内存、处理器、磁盘和网络接口等等。或许有的工程师前瞻性地将另一个应用服务器混在其中，这样当一个服务器发生故障时，可以使用它来维持系统运行。

这种一体化设计很可能适用于许多低流量站点。然而，如果产品非常成功并且受到广泛和快速的追捧使用，则可能导致用户感知到的响应时间过长到产品几乎完全不可用的程度。在某些时候，由于入站请求速率明显高于系统的处理能力，导致对大量请求的响应不及时，系统甚至发生宕机。

一个优秀的工程团队会考虑如何在发生这样的灾难性事故之前尽可能地扩展平台。扩展平台的方法有很多，我们使用一个以三个方面衡量程度的三维立方体来表达，我们称之为AKF拓展立方体。

AKF缩放立方体由X，Y和Z轴组成，每个轴上都采用不同的方法来缩放服务。立方体的最左下角（坐标X = 0，Y = 0和Z = 0）表示上述最糟糕情形下的一体化服务或产品：其中所有功能都基于使用单个服务器的单个代码库，它使用的服务器具有有限的内存、CPU速度、网络端口和大容量存储等。



立方体的X轴表示在同一应用程序和数据集的多个实例上均衡负载的方法。这是大多数公司用于扩展其服务的首选方案，并且从每秒钟扩展处理的请求来看是非常有效的。通常，处理中等规模企业的业务需求是足够了。与许多其他选项相比，这种方法的工程成本较低，因为不需要重新构建代码库，除非当应用程序处于维护状态时，工程团队需要消除其对特定服务器的耦合度。该方法很简单：克隆系统和服务，并允

许它存在于N个服务器上，每个服务器处理总请求的1/N。理想情况下，分发方法是依靠以高度可用的方式配置的负载均衡器，如果活动单位体由于硬件或软件问题而退出服务，则会有其他单位体变为活动状态保持服务不中断。我们不建议使用轮换DNS作为平衡负载的方式。如果应用程序处于维护状态，可以通过多种方法来解决，包括集中式状态服务，无状态的重新设计，或作为最后手段使用负载平衡器提供持久连接。虽然X轴的方法对于许多公司来说是足够的，并且在多个主机之间分配请求的处理，但它并没有解决其他潜在的瓶颈，如内存限制，内存被用于缓存信息或结果。我们不建议使用轮询调度　DNS作为负载平衡的方法。如果应用程序维护状态，则可以通过各种方法来解决这个问题，包括集中式状态服务，重新设计无状态，或作为使用负载平衡器提供持久连接的最后手段。虽然Xaxis方法对于许多公司来说是足够的，并且在多个主机之间分配请求的处理，但它并没有解决其他潜在的瓶颈，如内存限制、内存被用于缓存信息或结果。

立方体的Y轴表示按功能，服务或资源进行拆分。服务可能代表一组用例，通常可以考虑用动词或动作（例如"登录"）来设想，而以资源为导向的拆分是最容易想到的，比如"帐户信息"等。这些拆分不仅可以有助于处理像X轴一样的跨系统的事务分割，还可以有助于减少或分配专用于多个系统中任何给定应用程序的内存量。确定这些拆分顺序的建议方法是确定哪些是用最少的代价带给你最大收益的。这些分裂通常会对工程团队带来更高的成本，因为他们经常会要求将应用程序分开。作为一个快速的第一步，单个应用程序可以放置在多个服务器上，并将某些服务器专用于特定的

“服务”或URI。虽然这种方法将有助于跨多个系统扩展事务处理，类似于我们的X轴实现，但它可能不会提供减少服务/池/资源/应用程序所需的系统内存量的额外好处。在大型团队中考虑这样拆分的另一个原因是，专业的团队专注于特定的服务或资源，可以减少的应用学习曲线，提高质量，缩短上市时间（较小的代码库）等。 这种拆分通常被称为“泳道”应用程序和数据集，特别是当数据库和应用程序都被拆分为表示“故障域”或故障隔离基础架构时。

Z轴表示通过执行查找，模数或其他不加区分的函数（例如散列）来拆分事务的方式。与Y轴分割一样，这种拆分不仅有助于故障隔离，而且显着减少了所需的内存量（缓存等），并减少设备/服务所需的稳定存储空间。在这种情况下，您可以尝试使用内容id（文章）、列表ID或来自接收到的IP地址的哈希值等的模数。Z轴拆分通常是所有拆分中代价最高的，我们只推荐给具有超快增速和超高交易频率的客户。只有在公司沿着Y轴执行了颗粒化的分割之后，才应该使用它。也就是说，它可以提供最大程度的可扩展性，因为它创建的“泳道”中的数量几乎是无限的。例如，如果公司使用一些事务ID的模数实施Z轴拆分，并且实现是一个可配置的数字“N”，则N的值可以是10,100,1000等，并且N中每个数量级的增长都会为公司创造同等数量级的规模增长。

读书笔记：

故障隔离
或“泳道”架构

我们之前的两篇文章“拆库扩展”和“拆分应用程序或服务进行扩展”已经引用了一个我们称之为“泳道架构（Swimlaning Architectures）”的概念。

我们之前的两篇文章已经涵盖了这一概念的基础知识，但是我们还没有花太多时间来讨论在技术架构中采用这种拆分方式的原因。

在我们的定义中，“泳道”是指一个故障域。故障域表示一组服务存在于一个被边界包围的环境中，使得其中的任何故障都发生在边界内，并且故障不会传播或影响该边界之外的服务。这种故障隔离域的好处有两个方面：

1）故障检测：在检测粒度足够的情况下，组件识别故障的时间显著地降低了。这是因为查找原因或故障所在都被限定在了与之相关的故障隔离域中。

2）故障隔离：如前所述，故障不会波及平台内的其他业务。因此，只有一部分用户或产品的功能的一部分受到影响，这取决于服务的具体实现方式。

“泳道”架构是一种每个故障隔离域完全隔离的体系架构。为了实现这一点，理想情况下，泳道或故障域之间相互不发生交集。在这种类型的架构中同步调用时绝对被禁止的，因为在跨域同步调用时，即使是合理的超时和检测机制应用，也可能触发一系列的错误。严格来说，如果一个故障隔离域通过调用连接到另一个域中的任何其他服务，或者接收到来自其他域或服务的调用，那这个域并不是故障隔离的。

在隔离域之间进行异步调用是可被接受的，但并不可取。如果这样的通信是必要的，那么在异步调用时的故障检测和超时控制，以确保在任何服务上的端口调用不会过载是非常重要的。

正如我们之前指出的，泳道应该将所有的服务都放置在故障域内。例如，如果需要数据库访问，那么包含该泳道所有适当信息的数据库应与执行泳道功能或功能必需的所有应用程序和网络服务器处在同一故障域内。此外，该数据库不应用于处理其他泳道的其他服务的请求。我们的原则是一个主机上的一个产品数据库。

像过去我们在拓展立方体中所指出的那样，可以有许多方式来设想泳道架构。你可以根据分离的服务例如“登录”和“购物车”（两个单独的泳道）展开设想，每个泳道都有各自的网络和应用服务器以及位于泳道内的全部数据存储区，并且只响应该泳道中的系统。对应于我们以前介绍的缩放立方体，这将是一个“Y”轴泳道。

另一种方案是对你的客户群或你的订单号或产品目录进行拆分。

假设是进行这种强行的拆分（如对id取模），那就是按客户、订单号或产品id隔离的Z轴泳道。

将服务和数据库分离的概念与故障隔离域相结合，就创建了一个可扩展和高可用的平台。

读书笔记：

制造还是购买？

在我们的许多工作中，我们发现自己要帮助我们的客户了解什么时候适合制造，何时应该购买。

如果您简单的在网上搜索"制造还是购买"，您会发现数以百计的文章、流程和决策树来描述何时制造何时购买。其中许多都是以成本为核心决策因素，有些侧重于维护内部发展的贴现现金流，有些则侧重于战略。另一些文章融合了两者。

这是一组我们帮助客户决定应该制造还是购买时所提出的简单的问题：

1. 这个“事物”（产品/架构组件/功能）是否在我们的业务中创造战略上的差异？

我们探讨您是否正在创建转换成本，降低退出障碍，增加进入门槛等，这将为您提供相对于竞争对手的优势。有关此主题的更多信息，请查阅波特五力模型。如果

这个问题的答案是“不 - 它不会产生竞争优势”，那么99%的时间你应该停下来，试图找到一个打包的产品，开源的解决方案，或外包供应商来构建你所需要的。如果答案是“是”，请转到问题2。

2. 我们是否是创造这个“事物”最合适的公司？

这个问题有助于通知您是否可以有效地构建它并实现所需的价值。这是一个“核心还是环境”的问题；它会询问您的商业模式是否支持制造问题中的项目，以及如果您具有比其他任何人更好地制造它的对应技能。例如，如果您是社交网站，您可能没有任何业务需要建立关系数据库来供您自己使用。如果您可以回答“是”这个问题，请转到问题3，并在此处停下来，如果答案为“否”，请找到外部解决方案。请不要愚弄自己 - 如果你回答“是”，因为你相信你有世界上最聪明的人（你确实可能有），你真的需要稀释他们的努力，来关注在那些能使你成功之外其他的事情吗？

3. 你想要制造的“事物”，是不是只有很少的，甚至没有竞品？

我们知道这个问题的措辞有些尴尬 –但目的是为了在“制造”决策过程中的每一处回答“是”，才能退出这四个问题。如果有很多供应商能够提供这种“事物”，这是一个潜在的迹象，表明这种空间可能成为一种商品。随着时间的推移，商品产品的功能差异会变得很小，最终竞争价格也会随着时间的推移降低。因此，由于功能收敛和定价

下降，今天的“健在”决策将在明天看起来很糟糕。如果您回答“是”（即“是，有少量或不存在竞品”），请继续问题4。

4. 我们能够有效的控制成本了制造这个“事物”？

考虑到这个“事物”的整个生命周期（开始启用直到生命周期的终点），制造它是否比购买它更便宜？许多公司使用成本作为判断标准，但他们常常忽略了一个关键点，那就是维护自有“事物”，“小部件”，“功能”等的成本究竟是多少。如果您的业务真的在成长并且非常成功，您真的想在你产品的生命周期中，继续支持内部开发负载均衡器，数据库等？不要只因为你想做一件整齐的事情就肯定的回答这个问题。你的工作就是为股东创造价值 - 不要在“整齐的事情”上花费功夫- 除非这件“整洁的事情”能创造股东的价值。

还有许多更复杂的问题可以可以用来探讨制造事物，而不是购买，的正当性，但是我们认为这四个问题在大多数情况下都是足够的。

当所有4个问题的答案为“是”时，会得出“制造”决定。

我们建议您认真考虑购买或外包，只要你在任何时候对上述任何问题回答了“否”。

读书笔记：

忘掉朝鲜的核武器，他们是利用网络攻击来对付你的

朝鲜近期在弹道导弹和核武器上的举动是令人担心的。虽然我们似乎越来越接近核战争，或许比自古巴导弹危机以来的任何时候都更接近，但这种情况发生的可能性相对较低。即使是像金正恩这样一个明显的非理性国家元首，也必须明白，使用核装置将导致全世界的反对。对任何一个国家使用核装置都将在较短时间内终结他的统治，甚至毁灭我们今天所知的朝鲜民主主义人民共和国。这就引出了问题，为什么金正恩会采取这种冒险的外交政策呢？许多政治家和战略家似乎认为，这是一个强制其他国家认同朝鲜，并减少联合国目前对其施加的严厉制裁的战略。也许吧，但是或许除此之外，或者反过来说，金正恩是想让我们从他已经进行了多年的战争上转移视线：一场针对许多国家的网络战争。

一个国家发动的网络战争和无国籍团体发起的网络恐怖主义都是为了攻击我们的经济基础设施。朝鲜和恐怖分子都明白，攻击我们的经济、贸易和个人财富是对我们国家和人民造成伤害的最有效的方法。朝鲜有可能是藏在多次针对金融机构的袭击事件背后的、与WannaCry勒索软件的爆发有关的和索尼影业遭到袭击的幕后黑手，

涉嫌从孟加拉国中央银行强占8100万美元。所有这些事件都可能是由强大的朝鲜人民战组织"Unit 180"实施的。

当无法直接通过袭击窃取钱财或破坏商业贸易，恐怖分子和民族国家都谋求利用公司的产品作恶。最近的例子包括伊斯兰国（ISIS）使用eBay的交易市场向美国的一位操作者汇钱，俄罗斯在Facebook上购买广告，试图影响美国的总统选举。网络战和恐怖主义不仅仅是威胁——它们每天都在发生。前面的例子说明了事情的变化。你面临的问题是——你的公司是否已经变得足够优秀，可以保护自己免受这种不断增长和进化的威胁？

我们服务过的大多数公司的答案是"不"。安全组织似乎无视网络威胁的变化。他们继续专注于屏障保护系统和网络响应程序。很少有金融部门以外的公司开发了分析系统来帮助识别新出现的威胁和恶意活动。极少的公司会坚持进行积极的"巡逻"，以确定数字业务外围的威胁。这里有几个问题可以帮助你，评估你的公司是否有在网络战和恐怖主义活动下取得成功所必需的思维方式：

谁会对你造成伤害，他们打算怎么做？

退伍军人都知道，一场成功的防御不仅仅需要在墙壁后面探头和下蹲。你还必须进行巡逻和侦察周边，了解敌人将从何处来、以何种数量和能力。如果您的安全团

队没有积极尝试识别组织之外的威胁，那么我的意思就是这将超出你的意料——你肯定会感到惊讶。

你如何在你的产品和运作中发现新兴的行为？

鉴于存在利用你的产品作恶的威胁，你如何确定何时出现新的行为或趋势？你有怎样的分析系统来识别现有的角色或用户在用新的或奇怪的方式进行操作？您如何关注现有用户和新用户的新的模式或使用趋势？在高频的交易环境中，如何识别在99.99%的有效交易以外的，可能存在的少于1个基点的恶意操作？这些问题不太可能由"传统"安全团队来解答——他们需要具有深厚的分析能力和专门用于分析和机器学习的系统。同样的，传统的分析团队可能没有正确的思维去找出恶意的交易。

你有合适的人吗？

这是其中最重要的问题。你不需要解雇你的CSSP人员——你的安全团队中仍然需要他们。但是，你也希望有被证明能够像网络犯罪分子、恐怖分子和以战争为重点的民族国家一样思考和使用他们工具的人们。这些人不太愿意穿西装打领带去上班，而是喜欢穿短裤和凉鞋。传统的企业理念和工具将阻碍他们取得成功。他们需要使用洋葱浏览器，并且可以访问您不太可能希望其余的员工前往的网站。对大多数公司来说，其中最大的障碍是适应一个公司的文化——但我可以保证的是，如果你没有这样

的雇员，那你将不可能在网络战中存活下来。

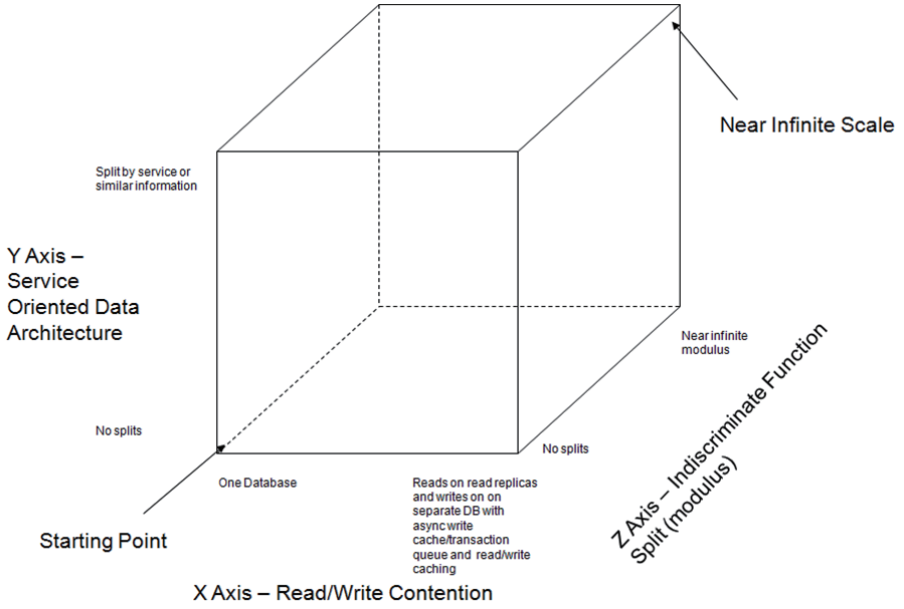
读书笔记：

怎样去扩展只读的子系统

许多SasS（ Software as a Service，软件即服务）系统有大量的部分专用于读取或搜索信息。读取或搜索在电子商务网站中可能在基于关键字的产品 / 库存搜索中被实现，或者它在内容站点中用一个非结构化字符串搜索或用正则表达式搜索所有的索引内容来实现。

在高频交易站点里，这个活动非常的繁重，即使在主数据库上成本高昂，也应考虑一个伟大的目标，即将它沿着规模立方体的Y轴进行分解，正如我们对数据库扩展后的描述。

AKF规模立方体可以被应用于读/搜索子系统来创建多维的分割以接近无限的规模。下面是我们的立方体图，它从描绘了一个读/搜子系统的三个维度。



正如我们过去的立方体一样， X轴是对事物负载和多个克隆之间的平衡。它允许系统在事务方面进行扩展，但不一定是数据的大小。

Y轴是沿着独立功能或服务的分割。当分割读/搜子系统时，就是对原始数据库进行Y抽拆分，我们创建特定地产品目录、用户特定信息、订单历史、存档内容、当前内容、推荐内容等相关的只读子系统来做递归式分割。

Z轴是模数，它进行功能查找或不加区分的进行功能拆分。

让我们来看看X和Z轴怎样描述一个物理系统，它可以很容易为读取做扩展。这个物理结构应该是有聚合器、负载平衡器、一个Nxm矩阵组成， N个系统持有1/N的数据及M个存储各自沿着N个维度得到一个月的交易。这个存储子系统不一定要是关系数据库，它们可以被存储在内存数据区中，如memcached或 Berkeley DB的实例。

读和搜的请求对于一个X聚合（ X随着请求数量而扩展）是负载均衡的。每一个对N个唯一数据集的N个请求通过负载平衡器到达M个系统中的一个，并且它有N个数据层的。这N个唯一的请求是 被汇总并排序的，然后返回给客户。

这样一个系统的好处是可以很容易的扩容出N个被列出的项目数量及M个被请求的事务数量。如果N的大小适当，则这些项目可以从内从中被返回，从而增加事务处理速度。

如果你想要更高的速度和更大的容错能力，你可以沿着本文档所描述的Y轴对只读子系统进行更多的拆分。

读书笔记：

如何减少大城市犯罪

我是Malcolm Gladwell的超级粉丝。Gladwell善于将那些复杂概念和难以理解的学术性研究转化为容读的文章，这种才华即便在他所在新闻领域也是首屈一指的。以在《引爆流行》（英文名 The Tipping Point）一书中展示的完美技巧为例，书中他征服了复杂性理论（又叫混沌理论）这个复杂的主题，使得其在读者面前易于接受。同时，在书中他也介绍了破窗理论。

破窗理论这个名字来源于1982年的大西洋月刊文章上。这篇文章要求读者想象一幢有几扇破窗户的建筑，其作者认为正是这些破窗子的存在招致更多破坏者来破坏其它更多的窗户，而随着破坏效应不断得循环扩散，下面会吸引流浪汉加入，因而造成附近建筑也会接连破坏。之后作者不断扩展理论，他还认为肆意破坏的效应会吸引犯罪，在不严肃对待破坏公物行为的社区，犯罪率会飙升。破窗理论的推论是城市只有靠加强对犯罪的法律制约才能减少犯罪率。有几个知名的例子似乎也论证了这个理论的正确性，比如纽约市长朱利亚尼的"零容忍计划"，该计划重点关注于破坏公物、公共场合饮酒、随地大小便和地铁逃票这些行为。该计划推出的十年内，犯罪率降低

了，这显然与计划的初衷是一致的。而其他城市和它们的实验也产生了类似的效果，这些证据似乎都证明了这个理论的假设是正确的。

但不要着急. . .

下面让我们走进自称为"流氓经济学家"的Stephen Levitt和他的合著者Stephen Dubner，两位都是《魔鬼经济学》一书的作者。虽然两位作者并不否认破窗理论可以解释犯罪率的下降，但他们对这一方法对于下降犯罪率的主要解释产生质疑。在纽约实施零容忍计划的这个十年期间，全国的犯罪率都在下降。犯罪率的全国性下降同时发生在那些破窗的城市也发生在没有破窗的城市。此外，不论警力开支的增加或减少，犯罪率都在下降。因此作者认为，破窗不能成为这个现象的主要解释。最有可能的解释和高度相关的变量因素应该是潜在罪犯的数量减少。 期间Roe v. Wade让流产变得合法化，这个举措直接使弃婴的数量大幅减少，而这些弃婴中有很大一部分人会有很大概率成为罪犯。但Gladwell因此误以为是破窗理论解释了犯罪率的降低，可这种解释显然不是最好的解释，该理论可能处于误导群众（最坏的情况）和解释不完整（最好的情况）之间的某种地位。

这中间发生了什么？

公正的说，不应该来让Gladwell来为这次失误负责。他并非科学家，因此没有

被训练过如何科学地推导自己的理论。更何况，即便在受训的科学家中，这种错误也时常发生。那么究竟错在哪里？这中间的错误其实可以从破窗理论的研究者们和《魔鬼经济学》作者的研究方法的差异性来得出。破窗理论的研究者初研之时是问"破坏行为是否会招致更多的破坏和不断升级的犯罪？"。而Levitt和Dubner 提出的问题是"什么变量可以解释犯罪率的问题？"

破窗理论始于一个以推论分析为中心的问题，其一开始就是从假设出发——即"破坏公物和其他小罪的犯罪现象会招致类似或更令人发指的罪行"，之后的过程就是接着尝试去证实或反驳假设。推论就是始于对数据的宽泛而抽象的认知，这种认知可以理解为对关系的概括或假设，它在试图去推测出不同数据元素之间的特定关系。破窗理论的研究者就是都从假设开始，他们开发了一系列的实验来检验这个假设，然后最终去评估各种面向城市治安的破窗理论方法所产生的时间序列数据。因此他们所缺乏的是去提出一个问题来包含一系列可能因素的范围选项。

而《魔鬼经济学》作者是从一个归纳性问题开始。归纳即是将对数据的具体观察变成总结的过程。这些归纳通常是以数据如何互相影响的假设或模型的形式呈现。其有助于了解应该向数据去询问哪些问题。归纳就是去问"什么自变量产生的变化与某些因变量产生的结果变化相一致？"然而推论是从自变量到因变量，但归纳是从因变量中倒推去识别与自变量的关系。

这又能说明什么？

直接就跳到推论阶段，而不是通过归纳阶段形成正确的问题和假设，正是我们在开发大数据程序和实现大数据解决方案时所见到的最大错误。我们都用独特的经验和独特的偏见看待问题，这些因素结合常常导致我们竞相假设并想测试它们。这里的问题是双重的。最好的情况也就是开发出一个不完整的结果（部分或大部分是不正确的），就像破窗理论研究人员一样。而最糟糕的情况是，我们会面临统计学家所谓的第一类型错误——即得出了完全错误结果。当我们没有办法为数据集中的输出结果寻找替代或更好的解释时，我们最后得到第一类型错误的概率就会增加。但归纳有助于发现那些可供选择或可支撑性的解释，通过探索数据来发现其中的潜在关系有助于我们提出正确的问题并形成更好的假设和更好的模型。跳过归纳阶段极有可能使我们得到一个错误的、误导性或不合标准的答案。

但是仅仅同时操作归纳和推论还是不够的。我们还必须意识到，面向归纳的解决方案不同于面向推论的解决方案。此外，我们也得知道在同一系统上执行的这两个过程虽然相互补充，但实际上也可以互相干扰。归纳必然面向宽泛，因而这必然是缓慢而乏味的过程。但反之推论只需要少量重要数据，并且实现推论相对更快。因而归纳系统最好的解决方案是在观察数据上只强加很少的关系或结构。而推论系统为了实现更快的响应速度，比起归纳系统它应该加上更多的结构。虽然探索事物的两个阶段（归纳和演绎）相互支持，但由于各自的差异性注定它们在各自的解决方案上拥有不

同的特殊需求。

同样，并非每个人都有能力操作归纳和推论。从我们的经验来看，那些善于决定如何证明变量之间关系的人往往不善于模式识别，反之亦然。

有两个现象值得注意，第一个是归纳和推论系统应该被隔离开，其次是执行任务的人也需要区分开，这对如何开发我们的分析系统和组织我们的大数据团队至关重要。

读书笔记：

第二部分

领导力

谷歌在解雇JAMES DAMORE过程中的对与错

AFK Partners 常说"对负面事件的浪费是非常可怕的"。当企业的某些事情变糟时，与之相关者（利益相关者，合伙人和雇员）都会付出对应的代价。所以，在已经付出代价的情况下，应该从冲突事件中最大化学习的机会。谷歌在解雇James Damore（"反多样性宣言"的作者）的事件中，浪费了一次极佳的从此次事件中学习的机会。尽管谷歌解雇James Damore看上去是在做一件"正确的事情"，但是这件事是否有"正确的理由"却仍然有待商榷。这里"正确的理由"应该是：多样性对于一家企业是十分有价值的，因为多样性会激发企业创新，帮助企业取得的成功。但是更进一步讲，多样性是很难实现且极为宝贵的，在这一过程中需要付出极大的努力，并且极易因为一些细微偏差而脱离轨道。企业不能允许她的员工与宝贵的多样性失之交臂。

多样性驱动创新与成功

我博士论文的刊物曾向我阐述过多样性，及其对创新，上市时间以及技术产品成功的益处。简单来说，通过有目的的组织安排，使团队在先验（生来具有的）和经

验（通过经验获得的）两个方面的均具备更高的多样性时，团队在创新性往往也能取得上更好的成就。刊登在哈佛商业评论上的研究也正式了这一观点，多样团队在创新上相比一般团队更为突出。多样团队更容易理解市场和客户普遍需求。具备多样性管理团队的公司，在财务收益上，有35%的几率领先其所处行业的平均水平。而董事会中有女性的企业，相比于没有女性的企业，拥有更高的回报率和净利润。

我们应该努力使我们的团队在洞察力和技能上具备差异化。正如我们在《架构及未来》这本书中提到的，这样的差异能够计划积极的冲突。而积极冲突的增多，能够为企业取得更广泛的战略可能性，版主企业取得更多，更高层次的成功。

长久以来，我们希望在追求多样性的过程中保证公平性。但公平性的问题在于，对一个人的公平很可能天然的导致对另一个人的不公。公平是主观的，公平也往往是政治性的。但是成功却是客观而可衡量的。所以，让我们把这样的冲突抛开，来拥抱多样性，因为它能够驱动创新与成功。毕竟，一个无法为成功提供支持的团队成员，不配待在一支成功的团队里。

取得多样性是困难的

尽管多样性是如此有价值，但获取多样性的代价也同样很高，特别是在软件团队中。正如我的同事Robin McGlothin最近写道，最近25年来，计算机科学领域内

授予女性的学位在不断下降。美国人口中大多数其他的少数群体也在他们相关的领域中缺乏应有的代表性。

在每一个供不应求的市场中，企业都应该尽可能的去追求一创新性的方式去吸引，培养和保留人才。这样的措施包括特别的指导项目，培训项目，或者在当地高校设立奖学金来吸引相关的问题组。这对于某些人来说或许是不公平的，但这就资本的真相与优势，通过市场的力量来解决供求问题。当某种技能或特征有巨大需求却缺乏供应时，这种技能的价值便会上涨。上面提到的各种举措，不过是我们吸引我们认为有价值的技能时，所需付出的成本。

渴望通过创新来取得成功的公司，必须坚定而专注的去追求多样性。任何关于这个目标的分歧都会对成功的可能性产生消极的影响。多少具备多样性背景的人会因为James Damore信件而离开，或者已经离开谷歌？哪怕是损失一个优秀的求职者，都是在艰难追求成功的过程中不可接受的损失。

底线

构建组织和创建企业文化的过程中，有意的发掘和培养多样性，会为企业在创新，上市时间，收益率和净利润等方面带来巨大的益处。尽管回报丰厚，但是追求这样成果的代价也很高。成功需要坚定而专注的去追求多样的卓越。

成功的企业在这类议题中不允许有任何的异议，因为这样的异议会有损企业对理想求职者的吸引力。在巨大的需求下，表现拘和限制，只会使求职者流向更舒适的环境中去。

简而言之，谷歌在解雇James Damore这件事上做的很对。但是他们没能很好的利用这次不幸的事件。当他们被问及为何解雇James Damore时，正确的回答应该是：我们认为在经验，背景，性别和种族上的多样性，能够驱动更高水平的创新和更伟大的成功。我们的企业文化不容忍不认可这样价值观的员工。

读书笔记：

达克效应[1]：一种最近流行的认知偏倚在技术世界里也存在吗？

我们都饱受各种认知偏倚之苦。这些认知偏倚犹如精神过滤器或滤镜，改变或包装着我们周围的真实世界。2016年一种独特的偏倚引起了广泛关注-达克效应（the Dunning-Kruger Effect）。维基百科将其定义为：

“...一种认知偏倚，其中低能力的人在错误地评估自己的认知能力时会感受到虚幻的优越性。”

1999年，康奈尔大学教授David Dunning和Justin Kruger共同撰写了一篇名为“欠缺和无知：辨析由于自身的不足而导致高估自我会有多困难（Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments）”的论文。该论文指出在某一领域能力欠缺的人通常无知到无法认识到自身的欠缺。或者简单地说，我们通常处于这样一个场景中，即我们不知道我们不知道什么，以至于在某一特定领域无法评判我们的专业水平。

这种效应或偏倚，也被称为“乌比冈湖效应（Lake Wobegon effect）”或“虚幻

优势（illusory superiority）”，并且与彼得原则[2]（the Peter Principle）有紧密联系，Donald Rumsfeld如是说：

有已知的已知；我们知道我们知道的东西。我们也知道有已知的未知；也就是说，我们知道有些东西我们不知道。但同样存在未知的未知——我们不知道我们其实不知道这些东西。如果纵观我们国家以及其他自由国度的历史，我们就知道后一类更为困难。

拿John Cleese的话说，愚蠢的人没有能力意识到自己有多愚蠢。

Dunning参与提出D-K效应有一段趣事。他读到一则发生在匹兹堡的不同寻常的银行抢劫案。不同寻常之处在于该案的抢劫者，McArthur Wheeler，直接笑对监控摄像头，丝毫不掩饰自己的抢劫过程。然而当他迅速被捕后非常惊讶，并告诉当局说：“...但我用了果汁啊！”

果汁？

Wheeler告诉警察，凭借监控探头他们不可能抓到他，因为他涂了柠檬汁后可以隐形。据说在脸上涂柠檬汁可以让摄像头看不到你！也许类似于使用柠檬汁可隐去墨水痕迹。Wheeler甚至想出一个测试该理论的方法，在用柠檬汁涂抹他的脸后，拍

摄了一幅宝丽来照片，果然，他的脸没有出现在照片上！警察对此无法解释，但Wheeler对摄影一无所知，就像他对入室盗窃一无所知一样。显然Wheeler对于如何盗窃知之甚少以至于不知道自己的无知。

所以，在技术世界中是否也存在Dunning-Kruger效应呢？绝对有...

就像司机一般总是相信自己的驾驶技术够得上一级方程式赛车水准，直到遇见制动和转弯技术远超自己人开着一辆破烂不堪的车从自己旁边呼啸而过时才知道人外有人。我们都低估了凡事的可能性。我们生活在我们自己的小世界中，并且仅将我们的能力与和我们活在同一小世界中的人比较。所以对于无知我们不得而知——我们不知道在我们的世界之外有更加厉害的司机。

你可能认为贵公司的效率已达到了巅峰，直到你从Google，Facebook，Amazon等处找来一些人，他们向你揭示出什么叫真正的巅峰——完备的敏捷过程和文化可以做些什么来缩短投入市场的时间，SRE[3]和DevOps[4]有多高效，如何保持创新，连续交付可以做什么等等。

AKF深信在不同团队中需要交叉展开“体验多样性”，从而将新的DNA注入到一个在不同领域生长的公司或其他领域中。我们见过不计其数的公司，其员工总是一尘不变，平均员工雇用期超过15年。在技术领域15年时间已产生巨大变化，阅读一本

书或参加一次会议可以为了解一项新的技术进程带来最新、最好的接触，但远远不够。持续不断地向公司输送新鲜血液，并有意推动伴随这些新鲜血液一起输入的流程、技术、组织架构的多样性交叉就显得尤其重要。

减轻D-K效应在技术上的影响，消除个人和组织偏见的其他方法包括：

360度考评（360 degree reviews） - Dunning自己曾说："洞察自己的途径是通过他人"。有什么能比定期360考评更好地获得反馈呢？

代码审核（Code reviews） - 如果你的部分开发人员受D-K效应困扰，那么就意味着你需要依赖代码审核来消除他们的代码的缺陷。只是要确保你不把两个D-K开发人员配在一起来做代码审核。

规划扑克[5]（Planning Poker） - 要求团队以真正的敏捷方式评估一项任务或项目，减少由于D-K效应而造成项目被鱼雷搬摧毁的机会。

征求意见（Soliciting advice） - 开源软件使用的日益增多意味着没有任何一家供应商可以提供理想中的专家以供咨询。适当利用OSS社区，而不是完全依赖于自己的那些只知道如何拼写Cassandra的开发人员。只是请注意，你可能不知道你搜罗的那些意见是好是坏。

正式面试（Proper interviewing）- 确保你的面试过程可以淘汰掉"自信的白痴"。考虑用一些虚假的问题来衡量候选人的反应，就像Jimmy Kimmel的"谎言目击者新闻[6]（Lie Witness News）"。至少要求对新候选人进行团队面试和团队共识。

简而言之，Dunning-Kruger在技术领域如同其他地方一样猖獗，甚至更是如此。不要指望它不存在于你的团队中，并且需做好防范。同时看看你自己。当我们发觉栽在一场我们本以为稳过的测试中时，我们中间有谁不会感到震惊？我们都在这样或那样的时候经历过达克效应。

---

[1]. Dunning-Kruger，达克效应（D-K Effect），全称为邓宁-克鲁格效应效应，由来自康奈尔大学（Cornell University）的Justin Kruger和David Dunning提出。它是一种认知偏差现象，指的是能力欠缺的人在自己认识不足的基础上得出错误结论，但是无法正确认识到自身的不足，察觉到自己的错误行为。这些能力欠缺者们沉浸在自我营造的虚幻的优势之中，常常高估自己的能力水平，却无法客观评价他人的能力。

[2]. The Peter Principle，彼得原则，是指："在一个等级制度中，每个职工趋向于上升到他所不能胜任的地位"，出自劳伦斯·彼得(en:Laurence J. Peter)和雷蒙德·霍尔(en:Raymond hull)所写的一本书，名为《彼得原则-何以事情时常会出错》。彼得原则解释了人力资源中的级际竞争。

[3]. SRE，网站可靠性工程，起源于google，在国内被称为运维工程师（OPS，operations），直接掌管着互联网公司的机器和服务，保证网站不宕机是他们的使命。当然，国内的运维工程师离Google SRE有较大的差距，Google SRE基本是从软件研发工程师转型，有很强的编程算法能力，同时具备系统管理员的技能，熟悉网络架构等，是一个要求非常高的职业。

[4]. DevOps（英文 Development 和 Operations 的组合）是一组过程、方法与系统的统称，用于促进软件开发、技术运营和质量保障（QA）部门之间的沟通、协作与整合。它的出现是由于软件行业日益清晰地认识到：为了按时交付软件产品和服务，开发和运营工作必须紧密合作。

[5]. Planning Poker, 计划扑克，是一个促使达成团队一致意见的团队构建活动。它由敏捷软件开发（agile software development）团队用来评估一定量的工作需要花多长时间完成。计划扑克的目的是确保开发团队中的每个人都积极地参与到评估过程并贡献其知识。该活动在可能有很多未知变量且为了得到精确的估计，需要很多领域的专业知识。

[6]. 美国著名脱口秀主持人Jimmy Kimmel的节目有一整人单元叫Lie Witness News（谎言目击者新闻），记者在大街上向路人说些假新闻或展示些假产品后问路人们的看法，看他们会应景说出什么荒唐的答案。

读书笔记：

工程指标

一个经常引起激烈辩论的话题是"怎么衡量工程师"？我是一个相当追求数据驱动的人，所以我是一个指标的粉丝，只要他们是： 1）正确测量的 2）正确使用的 3）不被隔离的。我最喜欢的三个是：速率、效率及成本。

速率 - 这是一个来自敏捷开发方式的度量。速率是（或你使用任何其它度量单位，如理想时间）一个团队中的工程师冲刺完成故事点的总和。 正如我之后讨论的那样，这个指标没有标准的好或坏，并且不应该有意的使用它来比较一个工程师与另一个工程师。这个指标将用来帮助更好的进行预估，就是那样。不要推断更多故事点或将一个团队和另一个团队比较，只要将它反馈给工程师和团队，这样它们在工作能获得更多的预见性。

效率 - 一个软件开发人员花费在开发相关活动（如：编码、设计、和产品经理进行讨论等）上的时间除以他们可用时间的总和（假设每天10个小时），这就提供了其工程的效率。这个指标是被设计来了解软件开发人员实际花费在开发软件上的时

间。这个指标经常令人惊讶。达到60%或以上是杰出的。我们经常看到开发团队是低于40%的。这个用于识别工程师把时间花费在哪里是很有帮助的。是不是有太多的公司会以没有直接关联到产品的推出？你是不是正在做许多的人力资源培训等？这个指标真的是为管理团队的，用来确定什么是没有花费在开发时间上的，然后摆脱它。

成本 - 技术成本占收入的百分比是一个很好的成本基础指标，它用来了解你在技术上花费了多少。这是非常有用的，它能用来与另一种技术（ Saas或基于互联网的公司）比较，并且你可以看到指标随着时间而改变。大多数创业者在开始的时候技术成本（工程师，托管等）都在整体收入的50%之上，但是随着收入的增长和业务的增加会快速减少。是的，扩大一个业务涉及有效的增长花费。已成立的收入范围在数千万的公司这个百分比通常在10%以下。在非常大的亿万收入的公司往往会驱使它降到57%。

现在我们知道了一些最常见的指标，那么它们应该怎么被使用？管理者和主管最常见的使用指标的方式是随着时间来对工程师进行相互比较或比较一个团队。 顺便说一下这项工作的效率和成本指标是管理效能的主要指标。管理者做出的大部分成本决策包括人力、合约等。所以他们应该来改进这些指标。在产品方面推出方面，对每次故事点的冲刺完成的衡量又称速率，如上所诉，它是用来更好的进行评估，而不是对开发者进行加速。不正确的使用这个指标将会导致过大的估量，而不是更快的发展。

一个有趣的开发者比较来自于1967年的格兰特（ Grant）和沙文（ Sackman）一篇文章，他们阐明最快的程序员和最慢的程序员完成一个任务的时间对比率是28:1。这是一个广泛被引用的比率，但是2000年的一篇文章将这个数字改进为最多为4:1，但是更像2:1。虽然2倍多的速度相差仍然使人印象深刻，但是它并没有优化产品的整体质量。一个非常快速并且高质量完成的一个工程师不和产品经理进行相互交流必然不是整体最有效的。我的观点是当比较工程师时，有许多别的因素要被考虑，而不仅仅是每个版本的故事点。

读书笔记：

产品开发生命周期或软件开发生命周期

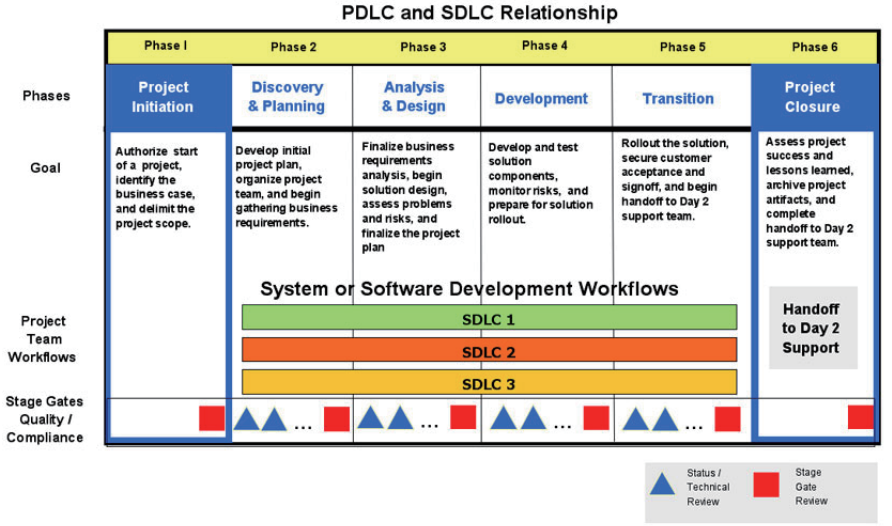
作为一个频繁的技术写作者，我经常发现我自己提到的方法或程序都是团队用来生产软件的。通常给出的两个术语是软件开发生命周期（SDLC）和产品开发生命周期（PDLC）。我的问题是它们真的可以互换吗？我不认为是这样，这里将给出为什么。

维基百科，我们集体的智慧，没有给出关于产品开发生命周期的相关条目，但是解释了产品生命周期与一个产品在市场上的寿命有关，并且涉及到许多专业的学术。根据这个定义，阶段包括引入市场，发展，成熟及饱和。这真的不是我感兴趣的产品开发生命周期。在新的产品开发下，我们找到了一个更加贴切的产品开发生命周期定义，它包含引入一个新产品进入市场的完整过程及以下步骤：产生创意、筛选创意、概念开发、业务分析、beta测试、技术实现、商业化及定价。

下面的软件开发生命周期，维基百科没有让我们失望，并把它解释为软件产品开发的一种加强结构。在这篇文章中引用了多个不同的模型，包括经典瀑布以及敏

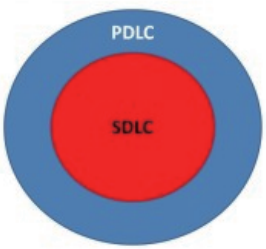
捷、 RAD（ Rapid Application Development，快速应用开发） 、 Scrum和其它。

在我看来产品开发生命周期是的产品研发的首要过程，包括业务部门在内。软件开发生命周期是产品开发生命周期中的特殊步骤，它是被技术部门完成的（包括产品经理）。在HBSC站点的一张图片，似乎没有任何随附的解释来很好的描述这个图表。



另一种解释方式，我认为它们是对于我来说的，即所有的专业软件项目都是产品，但不是所有的产品开发都包括软件开发。看下面的文氏图。前期工作（总线分析，竞争分析等）和后期工作（基础设施、支持、折旧等）是产品开发生命周期的一

部分，并且对于在软件开发生命周期中成功的创建出软件产品至关重要。有一些非软件产品仍然需要通过产品开发生命周期来进行研发。



你互换的使用它们吗？你认为它们的不同是什么？

读书笔记：

早期故障检测监控

AKF经常向我们的客户推荐采用业务度量监控（它使用高级用户活动或交易形成的图案，通常用来提供事件的早期警告）。业务度量监控不会告诉你问题是什么及在哪里，而是告诉你某些可能的异常，并且它应该进行调查。早期预警方面能减少检测时间，从而缩短整体的MTTR。

在eBay，我们有接近实时的用户指标图表，如：卖价、买价、登入和新用户注册。数据是每周进行绘制的。每天使用的图案按照一个容易识别的方式呈现 ，即峰值和谷值。这些绘制图显示在24*7小时有人值班的网络运营中心。这种与前一周图案的偏差进行比较的方式被证实是有用的，可以发现问题，如欧盟ISP不稳定影响了尝试访问eBay的客户。

例如一个星期三晚上，一切似乎很正常 - 直到卖价和买价突然急剧下降。 NOC快速启动了SEV1流程和技术资源来检查他们的领域。该站点没有可识别的错误，服务被确认为正常工作，但用户活动仍然明显较低。 SEV1流程进行了大约20分钟，这

个根本原因被确定。美国偶像的最后一集正在播出。我们的站点是正常的，只是我们客户还有其它的事情。 业务度量监控一直在工作 - 它们用一个使用模式的异常图案发出了警告。

世界杯是世界上最受欢迎的足球赛事，甚至可以说是全世界最受欢迎的体育赛事。在英国播出的赛事吸引了大量的观众，而播出到半场前一般是没有广告的。当时有一部关于英国的电力系统如何准备应对世界杯比赛直播的纪录片。直播进行到接近一半时，很大一部分的观众会去洗手间，打开电茶壶的开关。值得庆幸的是，记录片是关于电力公共事业的，而不是污水。这会导致电力负载的增加，将使电力系统产生严重的问题，需要对应的措施来维持电压和频率。电力公司已经通过标记"高峰"为这种情况做准备 – 利用柴油发电机联机来帮助处理增加的负载。公用电网的稳定性类似于居住区电网- 太多是坏的，太少是坏的，刚好合适是最好的。电力公司的运营不想将发电机带入的太早或太晚。他们需要客户的实时信息。这个方案是在运营中心放一个电视转播世界杯，使工程师能够在半场时间前启动发电机，并在负载增长减弱时关闭它。领着报酬看世界杯当然是一件意想不到的好事！

你的公司如何像英国电力那样做出反应？ 赞助商的活动或病毒营销可能导致你的系统超载。考虑在云中使用弹性计算来满足峰值需求 - 相当于为世界杯使用了柴油发电机。根据需求的峰值来扩大，然后关闭它。拥有基础设施，对于高峰采用租赁。使用业务度量监控来检测工作负荷的变化。

读书笔记：

定义MVP

我们经常把"最小可行产品"（注：英文为Minimum Viable Product，MVP全称）或MVP这类术语挂在嘴边，但大家是否都认可它的含义呢？我相信关于MVP的定义在团队中是值得树立的。我们快速梳理一下关于MVP三个相似但不同的定义:

1. 最小可行产品（MVP）是一种开发技术。使新产品或网站先开发足够满足早期使用的功能，而在后续早期用户反馈中完善并完成功能集。

2. 在产品开发方式中，最小可行产品（MVP）是投资回报率最高的产品。这种产品只有足够用于发布的核心功能，仅此而已。

3. Eric Ries第一次用这个术语时，曾这样描述：最小可行产品是新产品的一种版本，他可以让团队以最小的精力来最大化有效地了解用户。

我个人倾向将这些定义组合起来，我会从这中间挑出一些内容：

> 最小可行产品（MVP）：提供足够合适的功能来解决客户存在的问题，通过

降低在不了解客户情况下的风险，来拥有最大的投资回报。

就像不存在两个团队以一模一样的方式来实现敏捷一样，我们不可能有一模一样的关于MVP的定义，但是你的团队成员必须有完全一致的理解，否则有些人会把MVP当做一个完整功能的项目。所以，花些时间跟你们的全能敏捷团队来弄清关于MVP的定义吧！

读书笔记：

Boolan

高端IT互联网教育平台