**可伸缩性最佳实践：来自eBay的经验**

在eBay，可伸缩性是我们每天奋力抵抗的一大架构压力。我们所做的每一项架构及设计决策，身前身后都能看到它的踪影。当我们面对的是全世界数以亿计的用户，每天的页面浏览量超过10亿，系统中的数据量要用皮字节（1015或250）来计算——可伸缩性是生死交关的问题。  
  
  
  
在一个可伸缩的架构中，资源的消耗应该随负载线性（或更佳）上升，负载可由用户流量、数据量等测量。如果说性能衡量的是每一工作单元所需的资源消 耗，可伸缩性则是衡量当工作单元的数量或尺寸增加时，资源消耗的变化情况。换句话说，可伸缩性是整个价格-性能曲线的形状，而不是曲线上某一点的取值。  
  
可伸缩性有很多侧面——事务的方面、运营的方面、还有开发的方面。我们在改善一个Web系统的事务吞吐量的过程中学到了很多经验，本文总结了其中若 干关键的最佳实践。可能很多最佳实践你会觉得似曾相识，也可能有素未谋面的。这些都是开发和运营eBay网站的众人的集体经验结晶。  
  
最佳实践 #1：按功能分割  
相关的功能部分应该合在一起，不相关的功能部分应该分割开来——不管你把它叫做SOA、功能分解还是工程秘诀。而且，不相关的功能之间耦合程度越松散，就越能灵活地独立伸缩其中的一部分。  
  
在编码层次，我们无时不刻都在运用这条原则。JAR文件、包、Bundle等等，都是用来隔离和抽象功能的机制。  
  
在应用层次，eBay将不同的功能划分成几个应用程序池。销售功能由一组应用服务器运行，投标功能由另一组负责，搜索又是另外一组服务器。我们把总 共约16,000台应用服务器分成220个池。这样就可以根据某项功能的资源消耗，单独地伸缩其中一个池。我们也因此得以进一步隔离及合理化资源依赖关系 ——比如销售池只需要访问后台资源的一个相对较小的子集。  
  
在数据库层次，我们也采取同样的做法。eBay没有无所不包的单一数据库，相反我们有一组数据库主机存放用户数据、一组存放商品数据、一组存放购买数据……总共1000个逻辑数据库分布在400台物理主机上。同样，这种做法让我们得以单独为某一类数据伸缩其数据库设施。  
  
最佳实践 #2：水平切分  
按功能分割对我们的帮助很大，但单凭它还不足以得到完全可伸缩的架构。即使将功能一一解耦，单项功能的资源需求随着时间增长，仍然有可能超出单一系 统的能力。我们常常提醒自己，“没有分割就没有伸缩”。在单项功能内部，我们需要能把工作负载分解成许多我们有能力驾驭的小单元，让每个单元都能维持良好 的性能价格比。这就是水平分割出场的时候了。  
  
在应用层次，由于eBay将各种交互都设计成无状态的，所以水平分割是轻而易举之事。用标准的负载均衡服务器来路由进入的流量。所有应用服务器都是 均等的，而且任何服务器都不会维持事务性的状态，因此负载均衡可以任意选择应用服务器。如果需要更多处理能力，只需要简单地增加新的应用服务器。  
  
数据库层次的问题比较有挑战性，原因是数据天生就是有状态的。我们会按照主要的访问路径对数据作水平分割（或称为“sharding”）。例如用户 数据目前被分割到20台主机上，每台主机存放1/20的用户。随着用户数量的增长，以及每个用户的数据量增长，我们会增加更多的主机，将用户分散到更多的 机器上去。商品数据、购买数据、帐户数据等等也都用同样的方式处理。用例不同，我们分割数据的方案也不同：有些是对主键简单取模（ID尾数为1的放到第一 台主机，尾数为二的放到下一台，以此类推），有些是按照ID的区间分割（1-1M、1-2M等等），有些用一个查找表，还有些是综合以上的策略。不过具体 的分割方案如何，总的思想是支持数据分割及重分割的基础设施在可伸缩性上远比不支持的优越。  
  
最佳实践 #3：避免分布式事务  
看到这里，你可能在疑惑按功能划分数据和水平划分数据的实践如何满足事务要求。毕竟，几乎任何有意义的操作都要更新一个以上的实体——立即就可以举 出用户和商品的例子。正统的广为人知的答案是：建立跨资源的分布式事务，用两段式提交来保证要么所有资源全都更新，要么全都不更新。很不幸，这种悲观方案 的成本很可观。伸缩、性能和响应延迟都受到协调成本的反面影响，随着依赖的资源数量和客户数量的上升，这些指标都会以几何级数恶化。可用性亦受到限制，因 为所有依赖的资源都必须就位。实用主义的答案是，对于不相关的系统，放宽对它们的跨系统事务的保证。  
  
左右逢源是办不到的。保证跨多个系统或分区之间的即时的一致性，通常既无必要，也不现实。Inktomi的Eric Brewer十年前提出的CAP公理是这样说的：分布式系统的三项重要指标——一致性（Consistency）、可用性（Availability）和 分区耐受性（Partition-tolerance）——在任意时刻，只有两项能同时成立。对于高流量的网站来说，我们必须选择分区耐受性，因为它是实 现可伸缩的根本。对于24x7运行的网站，选择可用性也是理所当然的。于是只好放弃即时一致性（immediate consistency）。  
  
在eBay，我们绝对不允许任何形式的客户端或者分布式事务——因此绝不需要两段式提交。在某些经过仔细定义的情形下，我们会将作用于同一个数据库 的若干语句捆绑成单个事务性的操作。而对于绝大部分操作，单条语句是自动提交的。虽然我们故意放宽正统的ACID属性，以致不能在所有地方保证即时一致 性，但现实的结果是大部分系统在绝大部分时间都是可用的。当然我们也采用了一些技术来帮助系统达到最终的一致性（eventual consistency）：周密调整数据库操作的次序、异步恢复事件，以及数据核对（reconciliation）或者集中决算（settlement batches）。具体选择哪种技术要根据特定用例对一致性的需求来决定。  
  
对于架构师和系统的设计者来说，关键是要明白一致性并非“有”和“没有”的单选题。现实中大多数的用例都不要求即时一致性。正如我们经常根据成本和其他压力因素来权衡可用性的高低，一致性也同样可以量体裁衣，根据特定操作的需要而保证适当程度的一致性。   
  
最佳实践 #4：用异步策略解耦程序  
提高可伸缩性的另一项关键措施是积极地采取异步策略。如果组件A同步调用组件B，那么A和B就是紧密耦合的，而紧耦合的系统其可伸缩性特征是各部分 必须共同进退——要伸缩A必须同时伸缩B。同步调用的组件在可用性方面也面临着同样的问题。我们回到最基本的逻辑：如果A推出B，那么非B推出非A。也就 是说，若B不可用，则A也不可用。如果反过来A和B的联系是异步的，不管是通过队列、多播消息、批处理还是什么其他手段，它们就可以分别地伸缩。而且，此 时A和B的可用性特征是相互独立的——即使B受困或者死掉，A仍然能够继续前进。  
  
整个基础设施从上到下都应该贯彻这项原则。即使在单个组件内部也可通过SEDA（分阶段的事件驱动架构，Staged Event-Driven Architecture）等技术实现异步性，同时保持一个易于理解的编程模型。组件之间也遵守同样的原则——尽可能避免同步带来的耦合。在多数情况下， 两个组件在任何事件中都不会有直接的业务联系。在所有的层次，把过程分解为阶段（stages or phases），然后将它们异步地连接起来，这是伸缩的关键。  
  
最佳实践 #5：将过程转变为异步的流  
用异步的原则解耦程序，尽可能将过程变为异步的。对于要求快速响应的系统，这样做可以从根本上减少请求者所经历的响应延迟。对于网站或者交易系统， 牺牲数据或执行的延迟时间（完成全部工作的实践）来换取用户的延迟时间（用户得到响应的时间）是值得的。活动跟踪、单据开付、决算和报表等处理过程显然都 应该属于后台活动。主要用例过程中常常有很多步骤可以进一部分解成异步运行。任何可以晚点再做的事情都应该晚点再做。  
  
还有一个同等重要的方面认识到的人不多：异步性可以从根本上降低基础设施的成本。同步地执行操作迫使你必须按照负载的峰值来配备基础设施——即使在 任务最重的那一天里任务最重的那一秒，设施也必须有能力立即完成处理。而将昂贵的处理过程转变为异步的流，基础设施就不需要按照峰值来配备，只需要满足平 均负载。而且也不需要立即处理所有的请求，异步队列可以将处理任务分摊到较长的时间里，因而起到削峰的作用。系统的负载变化越大，曲线越多尖峰，就越能从 异步处理中得益。  
  
最佳实践 #6：虚拟化所有层次  
虚拟化和抽象化无所不在，计算机科学里有一句老话：所有问题都可以通过增加一个间接层次来解决。操作系统是对硬件的抽象，而许多现代语言所用的虚拟 机又是对操作系统的抽象。对象-关系映射层抽象了数据库。负载均衡器和虚拟IP抽象了网络终端。当我们通过分割数据和程序来提高基础设施的可伸缩性，为各 种分割增加额外的虚拟层次就成为重中之重。  
  
在eBay，我们虚拟化了数据库。应用与逻辑数据库交互，逻辑数据库再按照配置映射到某个特定的物理机器和数据库实例。应用也抽象于执行数据分割的 路由逻辑，路由逻辑会把特定的记录（如用户XYZ）分配到指定的分区。这两类抽象都是在我们自己开发的O/R层上实现的。这样虚拟化之后，我们的运营团队 可以按需要在物理主机群上重新分配逻辑主机——分离、合并、移动——而完全不需要接触应用程序代码。  
  
搜索引擎同样是虚拟化的。为了得到搜索结果，一个聚合器组件会在多个分区上执行并行的查询，但这个高度分割的搜索网格在客户看来只是单一的逻辑索引。  
  
以上种种措施并不只是为了程序员的方便，运营上的灵活性也是一大动机。硬件和软件系统都会故障，请求需要重新路由。组件、机器、分区都会不时增减、 移动。明智地运用虚拟化，可使高层的设施对以上变化难得糊涂，你也就有了腾挪的余地。虚拟化使基础设施的伸缩成为可能，因为它使伸缩变成可管理的。  
  
最佳实践 #7：适当地使用缓存  
最后要适当地使用缓存。这里给出的建议不一定普遍适用，因为缓存是否高效极大地依赖于用例的细节。说到底，要在存储约束、对可用性的需求、对陈旧数 据的容忍程度等条件下最大化缓存的命中率，这才是一个高效的缓存系统的最终目标。经验证明，要平衡众多因素是极其困难的，即使暂时达到目标，情况也极可能 随着时间而改变。  
  
最适合缓存的是很少改变、以读为主的数据——比如元数据、配置信息和静态数据。在eBay，我们积极地缓存这种类型的数据，并且结合使用“推”和“ 拉”两种方法保持系统在一定程度上的更新同步。减少对相同数据的重复请求能达到非常显著的效果。频繁变更、读写兼有的数据很难有效地缓存。在eBay，我 们大多有意识地回避这样的难题。我们一直不对请求间短暂存在的会话数据作任何缓存。也不在应用层缓存共享的业务对象，比如商品和用户数据。我们有意地牺牲 缓存这些数据的潜在利益，换取可用性和正确性。在此必须指出，其他网站采取了不同的途径，作了不同的取舍，也同样取得了成功。  
  
好东西也会过犹不及。为缓存分配的内存越多，能用来服务单个请求的内存就越少。应用层常常有内存不足的压力，因此这是非常现实的权衡。更重要的一 点，当你开始依赖于缓存，那么主要系统就只需要满足缓存未命中时的处理要求，自然而然你就会想到可以削减主要系统。但当你这样做之后，系统就完全离不开缓 存了。现在主要系统没办法直接应付全部流量，也就是说网站的可用性取决于缓存能否100%正常运行——潜在的危局。哪怕是例行的操作，比如重新配置缓存资 源、把缓存移动到别的机器、冷启动缓存服务器，都有可能引发严重的问题。  
  
做得好，缓存系统能让可伸缩性的曲线向下弯曲，也就是比线性增长还要好——后续请求从缓存中取数据比从主存储取数据成本低廉。反过来，缓存做得不好 会引入相当多额外的经常耗费，也会妨碍到可用性。我还没见过哪个系统没机会让缓存大展拳脚的，关键是要根据具体情况找到适当缓存策略。  
  
总结  
可伸缩性有时候被叫做“非功能性需求”，言下之意是它与功能无关，也就比较不重要。这么说简直错到了极点。我的观点是，可伸缩性是功能的先决条件——优先级为0的需求，比一切需求的优先级都高。  
  
  
希望以上最佳实践能对你有用，希望能帮助你从新的角度审视你的系统，无论其规模如何。