# 一、高可扩展性和分布式系统常见问题

## 1.1、Web分布式系统设计的原则

可用性、性能、可靠性、可扩展、易管理、成本

## 1.2、构建所有大型Web应用程序的核心

图片托管应用程序例子：

有时，你会在线上传图片，而一些大型网站需要托管和传送大量的图片，这对于构建一个具有成本效益、高可用性并具有低延时（快速检索）的架构是一项挑战。

在一个图片系统中，用户可以上传图片到一个中央服务器里，通过网络连接或API对这些图片进行请求，就像Flickr或者Picasa。简单点，我们就假设这个应用程序只包含两个核心部分：上传（写）图片和检索图片。图片上传时最好能够做到高效，传输速度也是我们最关心的，当有人向图片发出请求时（例如是一个Web页面或其他应用程序）。这是非常相似的功能，提供Web服务或内容分发网络（一个CDN服务器可以在许多地方存储内容，所以无论是在地理上还是物理上都更加接近用户，从而导致更快的性能）边缘服务器。

该系统需要考虑的其他重要方面：

（1）、图片存储的数量是没有限制的，所以存储应具备可伸缩，另外图片计算也需要考虑

（2）、下载/请求需要做到低延迟

（3）、用户上传一张图片，那么图片就应该始终在那里（图片数据的可靠性）

（4）、系统应该易于维护（易管理）

（5）、由于图片托管不会有太高的利润空间，所以系统需要具备成本效益

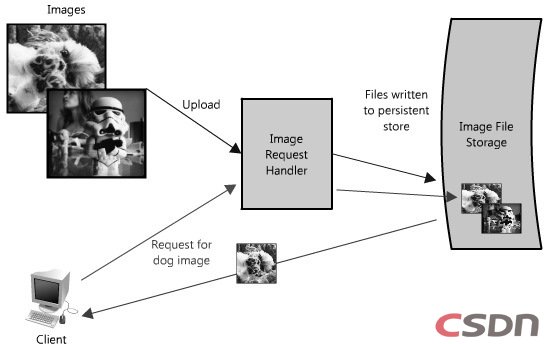


图1 图片托管系统的简化结构图

### 1.2.1、服务

当我们考虑构建可伸缩的系统时，它应有助于解耦功能，系统的每个部分都可以作为自己的服务并且拥有清晰的接口定义。在实践中，这种系统设计被称作面向服务的体系结构（SOA）。对于此类系统，每个服务都有它自己的独特功能，通过一个抽象接口可以与外面的任何内容进行互动，通常是面向公众的另一个服务API。

把系统分解成一组互补性的服务，在互相解耦这些操作块。这种抽象有助于在服务、基本环境和消费者服务之间建立非常清晰的关系。这种分解可以有效地隔离问题，每个块也可以互相伸缩。这种面向服务的系统设计与面向对象设计非常相似。

在我们的例子中，所有上传和检索请求都在同一台服务器上处理。然而，因为系统需要具备可伸缩性，所以把这两个功能打破并集成到自己的服务中是有意义的。

快进并假设服务正在大量使用；在这种情况下，很容易看到写图片的时间对读图片时间会产生多大影响（他们两个功能在彼此竞争共享资源）。根据各自体系，这种影响会是巨大的。即使上传和下载速度相同（这是不可能的，对于大多数的IP网络来说，下载速度：上传速度至少是3:1），通常，文件可以从缓存中读取，而写入，最终是写到磁盘中（也许在最终一致的情况下，可以被多写几次）。即使是从缓存或者磁盘（类似SSD）中读取，数据写入都会比读慢（Pole Position，一个开源DB基准的开源工具和结果）。

这种设计的另一个潜在问题是像Apache或者Lighttpd这些Web服务器通常都会有一个并发连接数上限（默认是500，但也可以更多），这可能会花费高流量，写可能会迅速消掉所有。既然读可以异步或利用其他性能优化，比如gzip压缩或分块传输代码，Web服务可以快速切换读取和客户端来服务于更多的请求，超过每秒的最大连接数（Apache的最大连接数设置为500，这种情况并不常见，每秒可以服务几千个读取请求）。另一方面，写通常倾向于保持一个开放的链接进行持续上传，所以，使用家庭网络上传一个1 MB的文件花费的时间可能会超过1秒，所以，这样的服务器只能同时满足500个写请求。

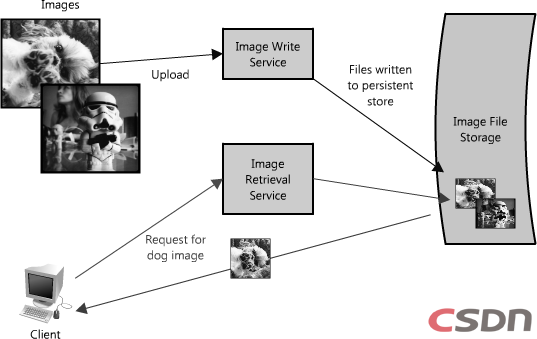


图2：读取分离

规划这种瓶颈的一个非常好的做法是把读和写进行分离，如图2所示。这样我们就可以对它们单独进行扩展（一直以来读都比写多）但也有助于弄明白每个点的意思。这种分离更易于排除故障和解决规模方面问题，如慢读。

这种方法的优点就是我们能够彼此独立解决问题——在同种情况下，无需写入和检索操作。这两种服务仍然利用全球语料库的图像，但是他们可以自由地优化性能和服务方法（例如排队请求或者缓存流行图片——下面会介绍更多）。从维护和成本角度来看，每一个服务都可以根据需要独立进行扩展，但如果把它们进行合并或交织在一起，那么有可能无意中就会对另一个性能产生影响，如上面讨论的情景。

当然，如果你有两个不同的端点，上面的例子可能会运行的很好（事实上，这非常类似于几个云存储供应商之间的实现和内容分发网络）。虽然有很多种方法可以解决这些瓶颈，但每个人都会有不同的权衡，所以采用适合你的方法才是最重要的。

例如，Flickr解决这个读/写问题是通过分发用户跨越不同的碎片，每个碎片只能处理一组用户，但是随着用户数的增加，更多的碎片也会相应的添加到群集里（请参阅Flickr的扩展介绍）。在第一个例子中，它更容易基于硬件的实际用量进行扩展（在整个系统中的读/写数量），而Flickr是基于其用户群进行扩展(but forces the assumption of equal usage across users so there can be extra capacity)。而前面的那个例子，任何一个中断或者问题都会降低整个系统功能（例如任何人都没办法执行写操作），而Flickr的一个中断只会影响到其所在碎片的用户数。在第一个例子中，它更容易通过整个数据集进行操作——例如，更新写服务，包括新的元数据或者通过所有的图片元数据进行搜索——而Flickr架构的每个碎片都需要被更新或搜索（或者需要创建一个搜索服务来收集元数据——事实上，他们就是这样做的）。

当谈到这些系统时，其实并没有非常正确的答案，但有助于我们回到文章开始处的原则上看问题。确定系统需求（大量的读或写或者两个都进行、级别并发、跨数据查询、范围、种类等等），选择不同的基准、理解系统是如何出错的并且对以后的故障发生情况做些扎实的计划。

### 1.2.2、冗余

为了可以正确处理错误，一个Web架构的服务和数据必须具备适当的冗余。例如，如果只有一个副本文件存储在这台单独的服务器上，那么如果这台服务器出现问题或丢失，那么该文件也随即一起丢失。丢失数据并不是什么好事情，避免数据丢失的常用方法就是多创建几个文件或副本或冗余。

同样也适用于服务器。如果一个应用程序有个核心功能，应确保有多个副本或版本在同时运行，这样可以避免单节点失败。

在系统中创建冗余，当系统发生危机时，如果需要，可以消除单点故障并提供备份或备用功能。例如，这里有两个相同的服务示例在生产环境中运行，如果其中一个发生故障或者降低，那么该系统容错转移至那个健康的副本上。容错转移可以自动发生也可以手动干预。

服务冗余的另一重要组成部分是创建一个无共享架构。在这种体系结构中，每个节点都能相互独立运行，并且没有所谓的中央“大脑”管理状态或协调活动其他节点。这对系统的可扩展帮助很大，因为新节点在没有特殊要求或知识的前提下被添加。然而，最重要的是，这些系统是没有单点故障的，所以失败的弹性就更大。

例如在我们的图片服务器应用程序中，所有的图片在另一个硬件上都有冗余副本（理想情况下是在不同的地理位置，避免在数据中心发生一些火灾、地震等自然事故），服务去访问图片将被冗余，所有潜在的服务请求。（参见图3：采用负载均衡是实现这点的最好方法，在下面还会介绍更多方法）

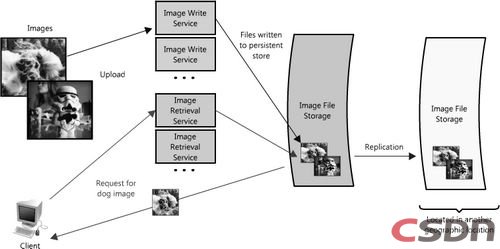


图3 图片托管应用程序冗余

### 1.2.3、分区

数据集有可能非常大，无法安装在一台服务器上。也有可能这样，某操作需要太多的计算资源、性能降低并且有必要增加容量。在这两种情况下，你有两种选择：纵向扩展或横向扩展。

纵向扩展意味着在单个服务器上添加更多的资源。所以，对于一个非常大的数据集来说，这可能意味着添加更多（或更大）的硬件设备，来使一台服务器能容下整个数据集。在计算操作下，这可能意味着移动计算到一个更大的服务器上，拥有更快的CPU或更大的内存。在各种情况下，纵向扩展可以通过提升单个资源的处理能力来完成。

横向扩展在另一方面是添加更多的节点，在大数据集下，这可能会使用第二服务器来存储部分数据集，对于计算资源来说，这意味着分割操作或跨节点加载。为了充分利用横向扩展，它应作为一种内在的系统架构设计原则，否则修改或拆分操作将会非常麻烦。

当谈到横向扩展时，最常见的做法是把服务进行分区或碎片。分区可以被派发，这样每个逻辑组的功能就是独立的。可以通过地理界限或其他标准，如非付费与付费用户来完成分区。这些方案的优点是他们会随着容量的增加提供一个服务或数据存储。

在我们的图片服务器案例中，用来存储图片的单个文件服务器可能被多个文件服务器取代，每个里面都会包含一套自己独特的图像。（见图4）这种架构将允许系统来填充每一个文件/图片服务器，当磁盘填满时会添加额外的服务器。这样的设计需要一个命名方案，用来捆绑图片文件名到其相应的服务器上。图像名字可以形成一个一致的哈希方案并映射到整个服务器上；或者给每张图片分配一个增量ID，当客户端对图片发出请求时，图片检索服务只需要检索映射到每个服务器上（例如索引）的ID。

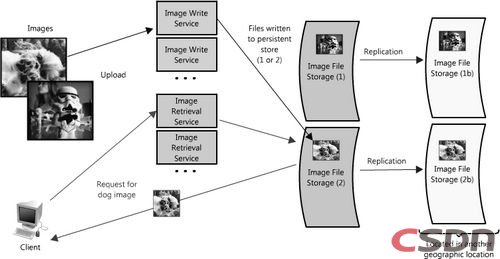


图4 图片托管应用程序冗余和分区

当然，跨越多个服务器对数据或功能进行分区还是有许多挑战的。其中的关键问题是数据本地化。在分布式系统中，数据操作或计算点越接近，系统性能就会越好。因此，它也可能是个潜在问题，当数据分散在多个服务器上时。有时数据不是在本地，那么就要迫使服务器通过网络来获取所需的信息，这个获取的过程就会设计到成本。

另一潜在问题是不一致。当这里有多个服务对一个共享资源执行读写操作时，潜在可能会有另一个服务器或数据存储参与进来，作为竞选条件——一些数据需要更新，但是读的优先级高于更新——在这种情况下，数据就是不一致的。例如在图片托管方案中，有可能出现的不一致是：如果一个客户端发送更新“狗”图片请求，进行重新命名，把“Dog”改成“Gizmo”，但同时，另一个客户端正在读这张图片。在这种情况下，标题就是不清楚的。“Dog”或“Gizmo”应该被第二个客户端接收。

当然，在进行数据分区时会产生一些障碍，但是分区允许把每个问题拆分到管理群里——通过数据、负载、使用模式等。这样对可扩展和易管理都是有帮助的，但也不是没有风险的。还有有很多方式来降低风险和故障处理。

## 1.3、分布式系统核心要素

### 1.3.1、构建快速可伸缩的数据访问块

大多数简单的Web应用程序，例如LAMP堆栈应用程序，看起来如图5所示：

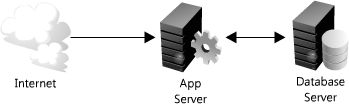


图5：简单的Web应用程序

随着系统渐渐扩大，他们将面临两大主要挑战：构建可扩展的应用程序服务器和数据访问机制。在一个高可扩的应用程序设计中，通常最小化的应用程序（或Web）服务往往能体现一种无共享的架构。这就使得应用程序服务器层要进行横向扩展，这种设计的结果就是把繁重的工作转移到堆栈下层的数据库服务和配置服务上，这才是这一层上真正的可扩展和性能挑战。

本文的剩余部分专门讨论一些常见策略和方法来使这些服务可以快速和可扩展，提升数据的访问速度。

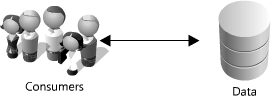


图6 过于简化的的Web应用程序

大多数系统可能会简化成图6那样，这是个非常不错的起点。如果你有大量的数据，想要快速便捷地访问，就好比在你书桌抽屉的最上面有一堆糖果。虽然过于简化，但也暗示了两个难题：可扩展存储和快速的数据访问。

为了这个例子，让我们假设有许多太字节（TB）数据，并且允许用户随机访问一小部分数据（见图7）。这与本文图片应用程序里的在文件服务器上定位一个图片文件非常相似。

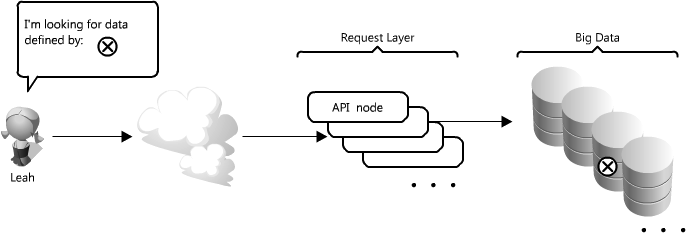


图7 访问特定数据

这也是个非常大的挑战，把TB级数据加载到内存中的成本比较昂贵，这可以直接转化到磁盘上进行IO。从磁盘上读取要比内存慢的多——内存访问和Chuck Norris一样快，而磁盘的访问速度要比在DMV上慢。这个速度不同于大数据集上的合计，实数内存访问大概要比顺序访问快6倍，或者比随机从磁盘上读取快10万倍（参考The Pathologies of Big Data）。此外，即使是unique ID，想要在较少的数据中查找确切的位置也是一项艰巨的任务。

幸运的是，能找到许多方法让这个问题变的简单，这里提供4个非常重要的方法：缓存、代理、索引和负载均衡器。在下面将会详细讨论这4个内容来提升数据访问速度。

### 1.3.2、缓存

缓存就是利用本地参考原则：当CPU要读取一个数据时，首先从缓存中查找，找到就立即读取并送给CPU处理；没有找到，就用相对慢的速率从内存中读取并送给CPU处理，同时把这个数据所在的数据块调入缓存中，可以使得以后对整块数据的读取都从缓存中进行，不必再调用内存。它们几乎被用在每一个计算层上：硬件、操作系统、Web浏览器、Web应用程序等。一个缓存就相当于是一个临时内存：它有一个有限的空间量，但访问它比访问原始数据速度要快。缓存也可以存在于各个层的架构中，但经常在离前端最近的那个层上发现，在那里可以快速实现并返回数据，无需占用下游层数据。

那么如何在我们的API例子里利用缓存使数据访问更快呢？在这种情况下，有许多地方可以插入缓存。一种是在请求层节点上插入缓存，如图8所示。

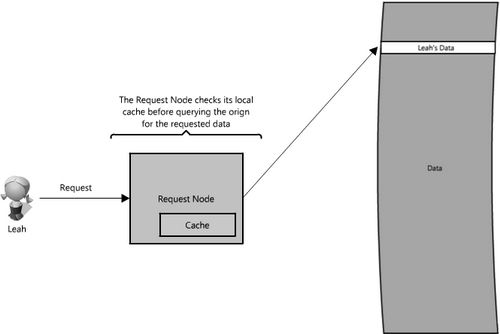


图8 在请求层节点插入缓存

在请求层节点上放置一个缓存，即可响应本地的存储数据。当对服务器发送一个请求时，如果本地存在所请求数据，那么该节点即会快速返回本地缓存数据。如果本地不存在，那么请求节点将会查询磁盘上的数据。请求层节点缓存即可以存在于内存中（这个非常快速）也

可以位于该节点的本地磁盘上（比访问网络存储要快）。

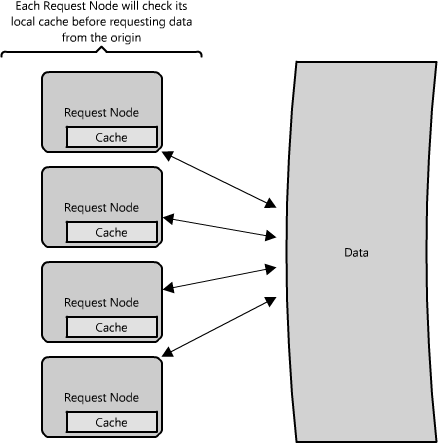


图9 多个缓存

当扩展到许多节点的时候，会发生什么呢？如图9所示，如果请求层被扩展为多个节点，它仍然有可能访问每个节点所在的主机缓存。然而，如果你的负载均衡器随机分布节点之间的请求，那么请求将会访问各个不同的节点，因此缓存遗漏将会增加。这里有两种方法可以克服这个问题：全局缓存和分布式缓存。

### 1.3.3、全局缓存

顾名思义，全局缓存是指所有节点都使用同一个缓存空间。这包含添加一台服务器或某种类型的文件存储，所有请求层节点访问该存储要比原始存储快。每个请求节点会以同种方式查询缓存，这种缓存方案可能有点复杂，随着客户机和请求数量的增加，单个缓存（Cache）很容易溢出，但在某些结构中却是非常有效的（特别是那些特定的硬件，专门用来提升全局缓存速度，或者是需要被缓存的特定数据集）。

在图10中描述了全局缓存常见的两种方式。当一个Cache响应在高速缓存中没有发现时，Cache自己会从底层存储中检索缺少的那块数据。如图11所示，请求节点去检索那些在高速缓存中没有发现的数据。

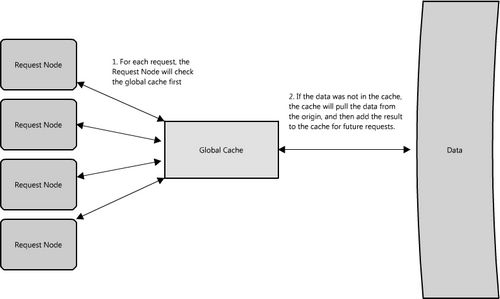


图10 负责检索数据的全局缓存

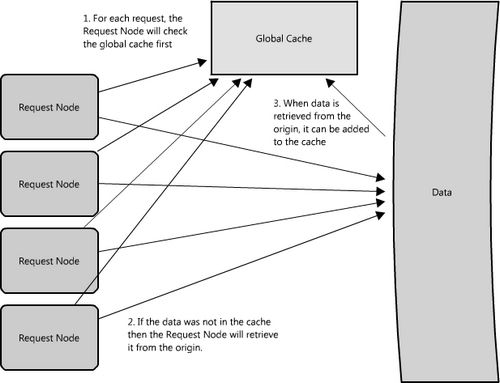


图11 全局缓存里负责检索的请求节点

大多使用全局缓存的应用程序都倾向于使用第一种类型，利用Cache本身来驱逐和获取数据以防止来自客户端的同一个数据区发出大量的请求。然而，在某些情况下，使用第二种实现反而更有意义。例如，如果该缓存用于存储大量的文件，低缓存的命中率会导致高速缓冲区不堪重负和缓存遗漏，在这种情况下， it helps to have a large percentage of the total data set (or hot data set) in the cache.

### 1.3.4、分布式缓存

分布式缓存即缓存在分布式系统各节点内存中的缓存数据。如图12所示，每个节点都有自己的缓存数据，所以如果冰箱扮演着缓存杂货店的角色，那么分布式缓存就是把食物放置在不同的地方——冰箱、橱柜和饭盒——当索取的时候，方便拿哪个就拿哪个，而无需特地往商店跑一趟。通常情况下，会使用一致性哈希函数对缓存进行划分，例如，一个请求节点正在寻找一个特定块的数据，在分布式缓存中，它很快就会知道去哪里找，并确保这些数据是可用的。这种情况下，每个节点都会有一小块缓存，然后在向另一个节点发送数据请求。因此分布式缓存的优点之一就是只需向请求池添加节点即可增加缓存空间，减少对数据库的访问负载量。

当然，分布式缓存也存在缺点，例如单点实效，当该节点出现故障或宕机，那么该节点保存的数据就会丢失。

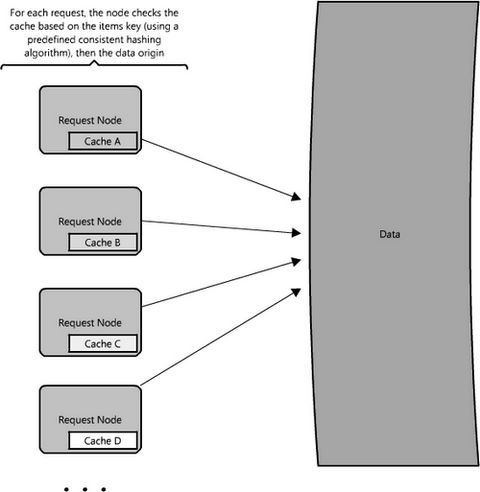


图12 分布式缓存

分布式缓存的突出优点是提高运行速度（前提当然是正确实现）。选择不同的方法也会有不一样的效果，如果方法正确，即使请求数很多，也不会对速度有所影响。然而，缓存的维护需要额外的存储空间，这些通常需要购买存储器实现，但价格都很昂贵。

其中一个非常流行的开源缓存产品：Memcached（即可以在本地缓存上工作也可以在分布式缓存上工作）；然而，这里还有许多其他选项（包括许多语言——或者是框架——特定选项）。

Memcached用于许多大型Web站点，其非常强大。Memcached基于一个存储键/值对的hashmap，优化数据存储和实现快速搜索。

Facebook采用不同类型的缓存技术来提升他们的网站性能（参考“Facebook caching and performance”）。在语言层面上使用$GLOBALS和APC（在PHP里提供函数调用），这有助于使中间函数调用更快（大多数语言都使用这些类型库来提升网站页面性能）。Facebook使用全局缓存并且通过多台服务器对缓存进行分布（参考“Scaling memcached at Facebook”），这就允许他们通过配置用户文件数据来获得更好的性能和吞吐量，并且还可以有一个中心位置更新数据（这是非常重要的，当运行成千上万台服务器时，缓存实效和一致性维护都是非常大的挑战）。

### 1.3.5、代理

简单点讲，代理服务器就是硬件/软件的中间件，接受客户端请求并且将他们转发到后端的源服务器上。通常，代理服务器用于过滤请求、记录请求日志或有时转换请求（通过添加/删除头结点、加密/解密或压缩）。



图13 代理服务器

代理可以优化请求，并且从整个系统的角度来优化请求通信量。一方面，使用代理可以加速数据访问，可以把相同（或相似的）请求重叠压缩成一个请求，然后返回单个结果到请求客户端，这就是压缩转发（collapsed forwarding）。

在一个局域网代理中，例如，客户端无需使用它们自己的IP去连接互联网，对于相同的内容，局域网将压缩来自客户端的请求。它很容易造成混淆，因为很多代理同样也是缓存（它是一个非常合乎逻辑放置缓存的地方），但并非所有缓存都扮演代理这一角色。

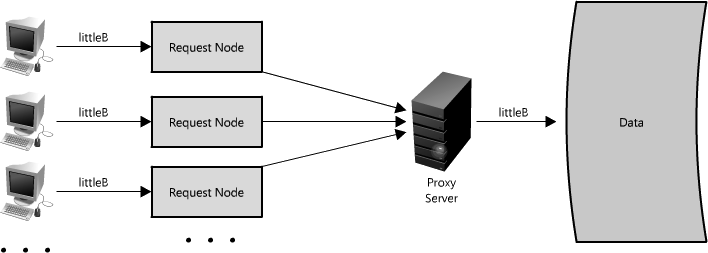


图14 使用一个代理服务器压缩请求

使用代理服务器的另一伟大方式是通过压缩请求对空间数据进行加密。采用这种策略最大化数据本地化的请求会导致减少请求的延迟。例如这里有一大串的节点请求B：partB1、partB2等等。我们可以设置代理来识别个人空间的位置请求，把它们压缩成单一的请求并只返回bigB，大大减少了从数据源处读取数据次数（如图15所示）。在高负载的情况下，代理也特别有用，或者当你具有有限的缓存时，它们基本上可以把多个请求批处理成一个。

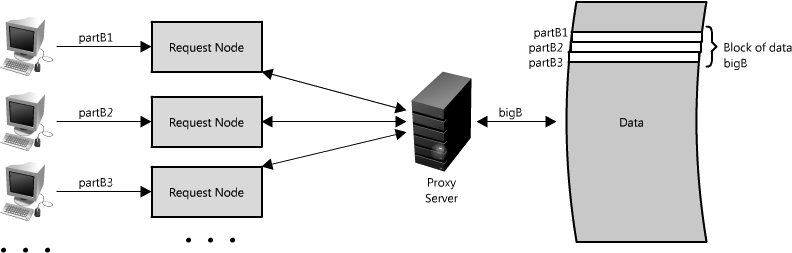


图15 使用代理对空间数据请求进行压缩

如果你正在为系统寻找代理，这里提供几个供你选择：Squid和Varnish，它们都做过非常全面的测试并且被广泛用在许多大型网站上。这些代理解决方案对客户端——服务器端通信提供了许多优化方案。在Web服务器层作为反向代理安装可以大大提高Web服务性能，减少处理传入客户机请求所需的工作量。

### 1.3.6、索引

使用索引来快速访问和优化数据是一个众所周知的策略，最有名的莫过于数据库索引。

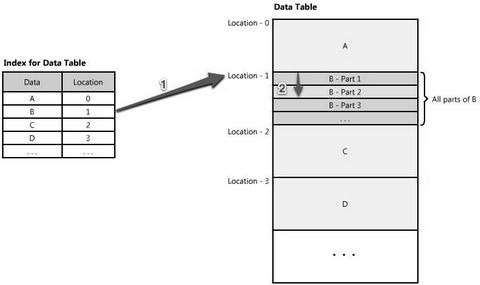


图16 索引

一个索引就是数据库表的目录，表中数据和相应的存储位置的列表。好比是一篇文章的目录，可以加快数据表的。例如让我们来查找一块数据，B中的第二部分——如何发现它的位置？如果你通过数据类型存储了一个索引——例如数据A、B、C——它将告诉你数据B的原始位置。然后你只需去查看B并且根据需要阅读B的数据即可（参考图16）。

这些索引通常存储在内存或者是传入客户端请求的本地中。Berkeley DBs（BDBs）和树数据结构常常被用在有序列表中存储数据，这是访问索引的理想选择。

通常，会把许多层索引作为一个映射，从一个位置移到下一个，以此类推，直到你得到想要的特定块数据（参照图17）。

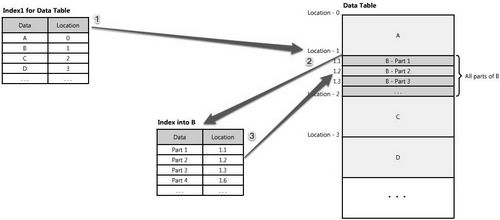
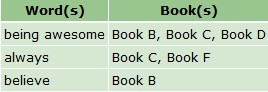


图17 多层索引

索引也可以对相同的数据创建多个不同的视图。对大型数据集来说，这种方法是非常好的，无需创建多个额外的数据副本就可以定义不同的过滤和排序，

例如，早期的图像托管系统实际上是托管图像书本内容，允许客户端查询这些图像中的内容，输入一个主题，就可以把所有相关的内容搜索出来。此外，采用同样的方式，搜索引擎还允许你搜索出HTML内容。在这种情况下，需要很多的服务器来存储这些文件，查找其中一个页面可能会很麻烦。首先，反向索引查询任意个单词或字元祖都需要可以轻松地访问；再有就是导航到正确的页面和位置，检索到正确的图像结果也是项挑战。因此，在这种情况下，反向索引会映射到一个位置（例如书B），然后书B可能会有一个包含所有内容、位置和各个部分出现次数的索引。



这种中间级索引只包含了Words、位置和书B的信息。与所有的信息不得不存储到一个大的反向索引中相比，这种嵌套的索引架构允许每个索引占用较少的空间。在大型系统中，这是非常关键的，即使采用压缩，这些索引也需要占用相当昂贵的存储空间。

例如，让我们假设这个世界上有——100,000,000本书（参考Inside Google Books官方博客）——每本书只有10页，每页只有250个单词，这也就意味着有2500亿个单词。如果每个单词只有5个字节，每个字节占用8 bits（或1个byte，甚至有些字符占用2 bytes），所以5 bytes/单词，那么一个索引所包含的单词就有可能超过一个TB的存储。此外，索引还有可能包含其他信息，例如元祖单词、数据位置等。

能够快速、轻松地找到数据是非常重要的，而使用索引就可以简单高效的实现。

### 1.3.7、负载均衡器

分布式系统的另一个关键部分是负载均衡。负载均衡器几乎是每个架构的主要组成部分，他们的角色是负责把网络请求分散到一个服务器集群中的可用服务器上去，通过管理进入的Web数据流量和增加有效的网络带宽，从而使网络访问者获得尽可能最佳的联网体验的硬件设备。

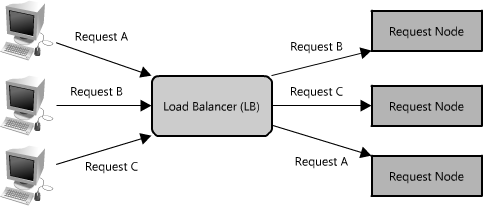


图18 负载均衡器

这里有许多种算法可用于为请求提供服务，包括随机选择一个节点、循环或者甚至是基于某个特定的标准来选择节点，例如内存或CPU利用率。负载均衡器即可以以硬件的方式表现出来，也可以以软件的方式。HAProxy是一个开源的负载均衡器，并且得到了非常广泛的使用。

在一个分布式系统中，负载均衡器通常处于系统的前端位置，所有传入的请求会相应地被路由。在一个复杂的分布式系统中，一个请求被路由到多个负载均衡器上并不常见，如图19所示：

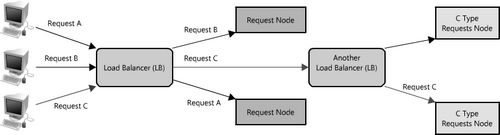


图19 多个负载平衡器

和代理一样，有些负载均衡器也可以基于请求的类型路由到不同的服务器集群上。（技术上来讲，这也被称为反向代理。）

负载均衡器所面临的挑战之一是管理用户特有的会话（user-session-specific）数据。在一个电子商务网站上，当你只有一个客户端时，是很容易让用户把商品放入购物车并且继续访问（这是非常重要的，因为商品很有可能在继续出售，而用户退出时，商品仍然留在购物车里）。然而，如果用户本次会话路由了一个节点，那么当他下次访问的时候会路由一个不同的节点，这样，就很有可能使购物车里的商品不一致，因为新的节点有可能会丢失该用户购物车里原先的商品（当你先放6包Mountain Dew 在购物车里，等到再次登录后发现购物车为空了）。解决这个问题的方法之一是使用sticky sessions，来使用户一直被路由到相同的节点，但它很难利用到可靠性功能，像自动故障转移（automatic failover）。这种情况下，用户的购物车里将会一直有商品，但如果sticky node变的不可用，这就需要特殊情况来处理并且假设购物车里的商品将不再有效（尽管希望这种假设不会被内置于应用程序里）。当然解决这个问题还有许多其他方法，例如本文提到的服务以及不包括（浏览器缓存、cookies和URL重写）。

在一个大型系统里会有各种不同类型的调度和负载均衡算法，包括简单点的像随机选择或循环以及更复杂的机制，例如利用率和容量。所有的这些算法都可以分布流量和请求，并且提供有用的可靠性工具，像自动故障转移或者自动清除一个坏的节点（例如当它无法响应时）。然而，这种高级功能会把问题诊断的复杂。 例如，当遇到高负载情况时，负载均衡器将会移除变慢或超时的节点（因为请求太多，删除节点后会把请求分配到其他节点上），这无疑会加剧其他节点的工作量，即负载加重。这种情况下，大量的监测变的非常重要，因为整个系统流量和吞吐量看起来可能会减少（因为节点服务更少的请求），但可能会累坏个别节点（处理更多的请求）。

负载均衡器也是扩展系统容量的一种简单方式，像文中提到的其他技术，在分布式系统架构中发挥着非常重要的作用。负载均衡器也提供一些重要功能来测试节点的健康状况，例如，如果该节点响应迟钝或过载，它可能就会被删除，然后利用系统中不同的节点冗余。

### 1.3.8、队列

到目前为止，我们已经讨论了许多方法来加快数据读取速度，但扩展数据层的另一个重要组成部分是如何高效的写入数据。在简单的系统中，进程负载等都比较少，并且数据库比较小，毋庸置疑，写的速度肯定不会慢。然而，在大型复杂的系统里，这个速度就很难把握了，可能会花费很长的时间。例如，数据有可能要写到几个不同的地方，不同的服务器或索引、或者系统正处于高负载情况下。在这种情况，该在哪里进行写？或者其他任何任务都有可能花费很长时间，要想在系统实现性能和可用性需要构建异步。处理这种异步的一种常见的方式就是采用队列。

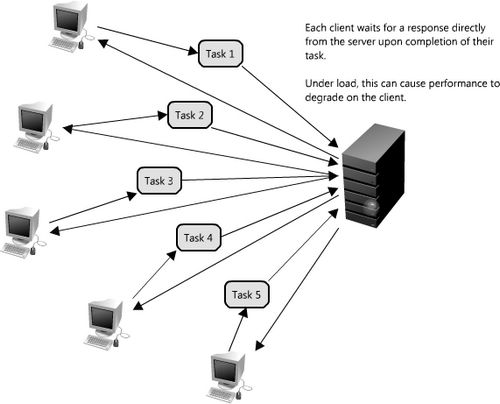


图20 同步请求

想象在一个系统里，每个客户机都要把请求发送至远程服务器，那么服务器应该尽可能快的接收并完成任务，然后把结果返回到相应的客户端。在小型系统中，一台服务器（或逻辑服务器）传入客户端数据会与客户端发出时一样快，这样就比较完美了。然而，当服务器接收到的请求多余它的处理能力时，那么每个客户端必须排队等待服务器处理其他客户端请求，直到轮到你了，服务器才会处理你的请求，直到最终完成。这就是一个同步请求的例子，如图20所示。

这种同步行为会严重降低客户端性能，客户端被迫等待，而通过添加额外的服务器来满足负载并不能解决问题，即使采用最有效的负载均衡也很难保证分配公平，在大客户端下。进一步讲，如果处理请求的服务器不可用或者瘫痪，那么客户端上游也将失败。有效的解决这个问题需要抽象客户端请求以及服务请求的实际工作。

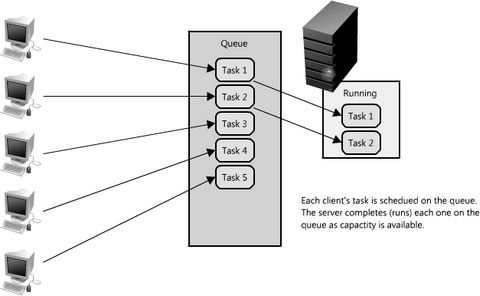


图21 使用队列来管理请求

队列就像听起来那样简单，一个任务进来，就添加到队列里去，然后the workers挑选有能力处理的下一个任务。（参考图21）这些任务有可能仅是简单的写入，也有可能是复杂的，如把文档生成图像预览。当一个客户端把任务请求提交的队列中时，他们不需要被迫等待结果，相反，他们只需确认请求是否被正确接收。

队列使客户端能够以异步的方式工作，对客户端请求和响应提供战略抽象。另一方面，在一个同步系统中，请求和回应是没有分化的，因此他们不能被分开管理。在异步系统中，客户端发出请求任务，服务器对收到的消息进行响应并确认任务被接收，然后客户端可以定期检查任务状态，一旦任务完成，即可看到结果。当客户端在等待异步请求是否完成时，它还可以自由执行其他任务，甚至是向其他服务器发出异步请求。下面要介绍的是消息和队列在分布式系统中的杠杆作用。

队列也对服务中断或失败提供一种保护机制。例如，它很容易创建一个高度健壮的队列，当服务器瞬间失败时，该队列可以把刚刚失败的请求重新发送至服务器。相比直接暴露客户端来间断服务供应，使用队列来保证服务质量更可取，要求必须有复杂且矛盾性的客户端差错处理。

队列是管理分布式通信与任何大规模分布式系统中各个部分之间的基础，并且有许多实现方式。这里有许多开源的队列，如RabbitMQ、ActiveMQ、BeanstalkD，但也有一些当做服务使用，如Zookeeper，甚至是用来数据存储，像Redis。