**软件架构与中间件-（作业3）**

学号：L170300901 学生：卢兑玧

可扩展的网页架构和分散系统

网页分散系统设计时的考虑

建立和管理具有扩张性的网站或网页应用软件是什么呢？ 在基础意义上，一个具有扩展性的网站是指用户通过网络连接到远程资源时，资源或资源的访问权限分散在多个服务器上时。

在创建刚刚开始做生意的小规模网站时，如果理解设计大规模网站时的考虑事项和各种交易事项，就可以做出更明智的决定。 以下是设计大规模网页系统时需要考虑的主要事项。

- 可用性(Availability):网站的可用性对很多公司的名声和功能至关重要。 例如，在Amazon.com等网上市场网站，若几分钟服务不正常运行，公司将蒙受数千或数万美元的损失。 因此，拥有可用的、对障碍灵活的架构是最基本的商业要求事项。 为了从分散系统中得到高可用性，有必要考虑发生重要编组双重化和失败时的快速修复方法、在发生问题时只允许部分动作，从而避免发生全面障碍的构成(graceful degradation)。

- 性能(Performance):在大多数网站，性能是非常重要的考虑事项。 网站的速度不仅对使用性、满意度，对收益及维持有直接关联的搜索引擎的排名也会产生重要影响。 最终，为了快速的应答时间和较低的回转时间，制定最优化的系统是非常重要的。

- 可靠性:(Reliability)总是以同样的要求,因为同样的结果应该提供。系统正常动作总是要一句话。数据变化或更新,更新的新的数据表明,必须什么样的数据系统中储存的用户相关数据后,搜索结果时,总是被返还的用户知道。

- 扩张性(Scalability):大规模的分布式系统的规模本身就是扩张性,这样中需要考虑的一个侧面而已。重要的是更多的部下处理为增加吞吐量所需的努力。系统的扩展性是指系统的诸多特征和功能/问题说明以及失调。例如,有多少额外流量处理,储存空间,添加有多容易,有多少种处理的事务和。

- 管理性（Manageability）:设计容易操作的系统是另一个重要的考虑事项。 所谓系统的管理性，是指运用（维持和更新）的扩张性。 要想提高管理性，在发生问题时容易分析，容易理解问题。 此外，更新和修改、系统运用本身都要容易。

- 费用(Cost):费用是重要的因素。 费用当然包括硬件和软件费用，但分发和管理系统的费用也要考虑。 这些费用包括系统构建所需要的时间、运行系统所需要的努力量、对所有需要考虑的事项所需要的教育费用。 也就是说，费用是拥有系统所需的所有费用。

这些考虑可以在设计分散网页系统时作为判断标准使用。 但是提高某一个可能会带来另一个下降。 简单的例子，为了提高处理量，增加多个服务器（水平扩张）需要追加管理多个服务器，因此在管理性和费用方面并不好。

无论设计哪种网页应用，考虑所涉及的考虑都很重要。 为了提高一个, 必须牺牲另一个 但即便如此。

基础

关于系统架构，有几点需要考虑。 什么才是正确的"部分"，怎样才能让这些部分更合适，还有什么是正确的贸易等。 不需要扩张性，投资扩张性不是明智的商业。 但是，在系统设计时提前考虑扩张性，最终有助于节约时间和资源。

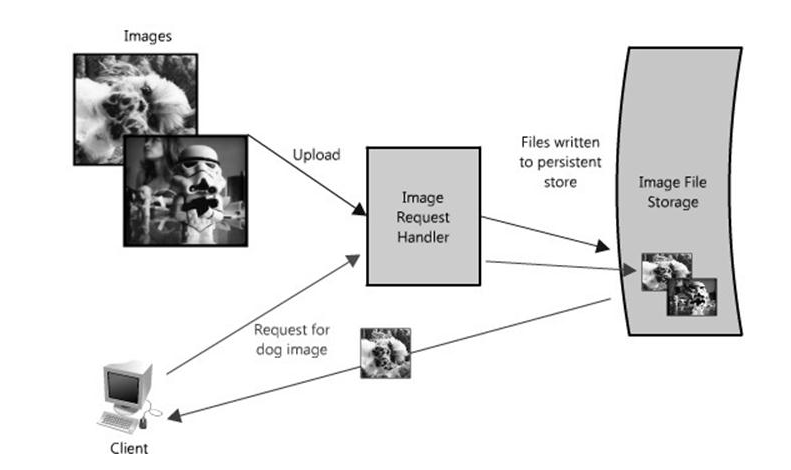
在该节中，几乎所有大规模网络应用所需的核心事项"服务"、"双重化"、"分割"、"例外处理"。 对于这些事项，需要基于前面考虑的事项进行选择和协商。 为了更细致的说明，我想以例题为基础开始讲故事。

如:网络相册应用软件

应该有过在网站上上传图片的经验。 管理和提供众多形象的大规模网站具有挑战性，其课题是构建既能有效费用又能满足高价使用性，具备快速搜索（快速搜索）的架构。

想象一下，像Flickr和Picasa一样，用户可以在中央服务器上传图片，图像可以通过链接或API接收请求。 为了简单，假设该应用程序只有两个功能。 一是在服务器上上传图片，二是在质疑图片。 我们想要有效上传图像的同时，也会考虑让别人在要求查看图像（查看图像的要求通过网页或其他应用程序）时，能够非常快地看到图像。 这与网页服务器或与 CDN 连接的服务器所做的事情类似（CDN 服务器存储信息， 以便在地理位置和物理位置上快速向最近的用户传递信息 ） 。

将下列图像主机应用的功能简单化。



《图1》

举例来说，图像主机应用的实现目标是，在用户立场上，以足够快的体感速度可靠地存储数据，并确保系统的各个部分能够很好地扩张。

以小规模的规模建立这样的系统是非常简单的事情，在一个服务器上也可以工作。 创建这么小的网站不是本文所涉及的。 这里要涉及的是构建像Flickr这样规模增长的系统。

服务(Services)

在设计扩张性系统时，以每个明确的接口为基础，按功能分类思考是很好的方法。 以这种方式设计的系统被称为SOA(Service-Oriented Architecture)。 SOA明确按功能构成服务。 而且，为了与其他服务相互作用，每个服务都使用API形态的抽象界面。

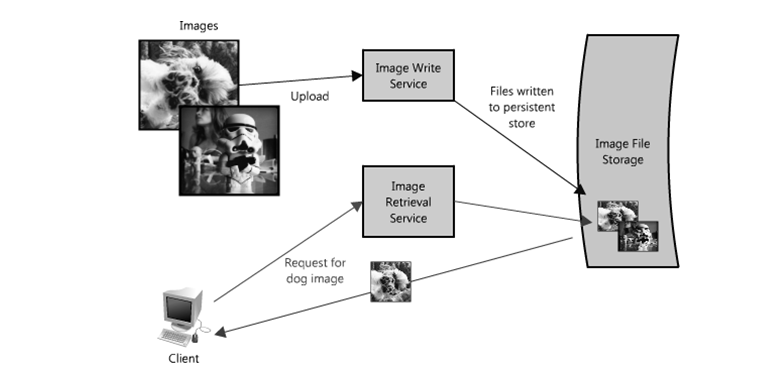
将系统分割成互补的服务，是指将系统以功能为单位分离。 这些抽象化有助于建立服务和服务所处的环境以及服务和服务用户之间的明确关系. 这种明确的技术既有助于分离问题，也有助于各自独立扩张。 从这个意义上讲，系统中的SOA与面向客体的编程非常相似。

在例题应用中，所有上传图像和搜索请求都由同一个服务器处理，但是为了扩展系统，上传和搜索功能分离为各自的服务是合理的。

让我们把表拨快一些,假定现在服务已经用得很多了. 这种假设剧本可以很容易地确认"写"对"读"产生多大影响（因为这两种功能是竞争性地使用共享资源）。 即使假设下载和上传的速度相同（实际下载速度和上传速度比率为3:1左右），虽然"阅读"可以得到Cash的帮助，但"书写"要求最终必须到达磁盘（而且，在eventual consistency的情况下，将发生多次书写）。 甚至所有的数据都存在于存储器或SSD的情况下，数据库的写法也比阅读慢。

这种设计的另一个潜在问题是，Apache或lighthtpd等网络服务器同时连接的数量存在上限（违约价格约为500左右，但也有可能设定更高）。 阅读可能是异步， 可以利用gzip压缩或chunked transfer encoding来优化性能。 网页服务器的每秒阅读请求处理数通常高于最大连接数。 因为网页服务器在处理一个读取请求后，会立即处理另一个请求

另一方面，在上传期间，要保持连接打开的状态。 因此，如果上传1MB需要1秒以上，服务器只能同时处理500个。



《图2》

作为对这种瓶颈现象的应对，如《图2》所示，把阅读和写作分开是很好的办法。

该方法可以独立扩大阅读和写作（普通用户比起写作，阅读更多），也有助于明确确认系统的各个部分是如何运转的。 当出现阅读慢的现象时，很容易解决问题并扩展阅读。

这种方法的优点是能够独立解决各自的问题。 在同一服务领域，不必担心使用和查询。 这两者都是为图像提供系统设计的，但可以各自使用适合的方法来提高各自的性能。 也就是说，用Q管理要求或将人气形象储存在Cashie上。 从维护和费用观点来看，各服务可以按照需要独立扩张。 如果这两者是不独立的、相互结合的形态，那么任何一个都会影响另一个性能。

当然， < 图 2> 所描述的架构即使使用两个不同的endpoint（翻译者注:endpoint， 指的是如何访问服务， 包括协议地址等信息）也动作良好（事实上，这类似于某些云存储提供者和 CDN ） 。 解决这些瓶颈现象有多种方法，每一种解决办法都有不同的交易方式。

例如，Flickr为了解决读写性能问题，将用户分散到不同的Shad。 每个 shad是只处理 shad的用户，当用户增加时，将可处理的 shad添加到集群中。在<图1>中举例的系统(读写在全域的系统)中，根据实际使用量，在硬件层面很容易扩展处理容量，而Flickr这样的架构则必须以Shad为单位扩展（假设所有用户都具有相同的使用量，这样就可以获得额外的空间 ） 。 在《图1》中说明的例子中，发生问题时按功能发生问题，相反，像Flickr这样的情况，只有在特定的用户Shad内发生问题。 在<图1>中说明的例子中，像添加新的元数据或利用所有图像元数据查找一样，对全部数据进行运算是非常容易的。 相反， Flickr使用的架构中，每个Shad都要更新和搜索。

这样的系统架构中没有正确答案。 但是有必要重新思考一下前述考虑事项。 要决定系统所要求的（阅读量多还是写得多，还是两个都多，质疑是关于整个数据库还是部分，如何排列等），需要对各种方案进行排练，理解系统在什么情况下出现问题，并制定明确的计划。

双层鞋(Redundancy)

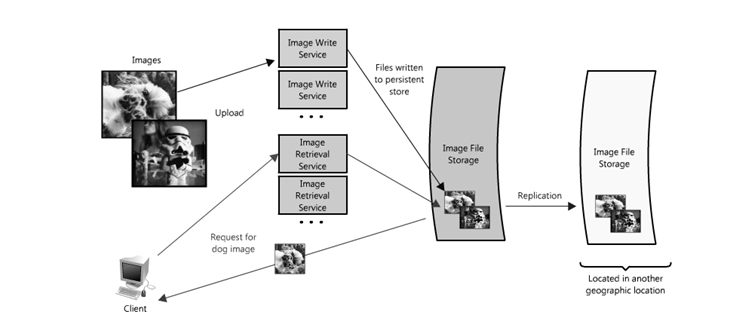
应对障碍的网络架构应考虑数据和服务器的双重化。 例如，如果某个文件只存储在一个服务器上，那么当该服务器出现故障时，该文件就会丢失。 不丢失数据的最普遍的处理方法是在多个服务器上复制数据。

同样的原则也适用于服务。 如果应用软件有重要的核心功能，那么需要让具有相同功能的多个同时运行。

实现系统双重化可以消除单一故障（single point of failure），在发生障碍时也可以备份或系统继续工作。 例如，为了提供相同的服务，当两个即时操作时，如果一个出现障碍，系统只能允许其他即时操作。 这种切断作业可以自动发生，也有需要运营者介入的情况。

在服务双重化时，重要的是建立Shared Nothing架构。 在该架构中，每个节点都可以独立工作，无需管理状态或作业的中央部分。 这种结构在系统想要具有扩张性时很有用。 因为，在没有特殊条件或知识的情况下，可以添加新的节点。 更重要的是，系统将不再具有单一的故障点。 因此，可以更好地应对障碍。

下面以图像服务器应用程序为例，来解释一下这种双重化。 所有图像都将被复制存储在不同的硬件上（理想上像不同的IDC一样地理位置），同时还有多个服务可以执行处理请求的相同动作。 < 图 3> 显示着这样的结构(为了实现这些，需要活塞杆平衡器).



《图3》

分区(Partitions)

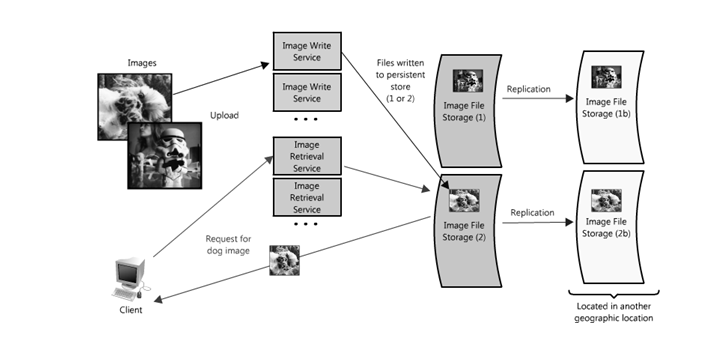
在一个服务器上可能有很多无法承受的数据。 或者为了运算需要大量的计算资源，从而导致性能下降。 解决这些问题有两种选择。 一个是垂直扩张，另一个是横向扩张。

垂直扩张是指向每个服务器添加更多资源。 为了处理大量数据，在服务器上添加硬盘就属于此。 为了更快的计算性能，增加更快的CPU或更大容量的存储器也是一样。 即垂直扩张是指提高各资源的处理能力。

相反，横向扩张是指增加节点。 即，在数据较多的情况下，追加存储部分数据的节点。 在需要大量运算的情况下，应分离运算，在追加的节点上进行作业。 为了兼顾横向扩张的优点，必须遵循系统架构的固有设计原则。 否则，修改或分离功能单位将变得非常不方便。

横向扩张的最普遍的方法是将服务分割成分区或分区单位。 分区可以分散到不同功能的逻辑集合中。 这种分区可以根据特定用户或数据的地理位置制作，也可以根据免费用户还是付费用户的标准制作。 这种形态的好处是以增设为基础，可以提供服务或数据存储。

在图像服务器预制架构中，用于保存图像的文件服务器可以被多个文件服务器取代。 而且每个服务器处理的形象都不一样（图4）。 在这种架构下，当出现磁盘满载的情况时，文件可以存储到其他服务器上。 在该设计中，需要以图像的名称为基础，找到保存该图像的服务器的方法。 图片名称可能是可指服务器的Consistent Hashing形态。 或者图像名称可以是增加的ID，这种情况下，图像查询服务只要知道各服务器存储的ID范围即可



《图4》

当然，将数据或功能分散到多个服务器也存在技术上的挑战。 在诸多核心议题中，有一个是数据本地化是其中之一。 要运算的数据越近，系统的性能就越高。 因此，将所需数据分散到多个服务器上，可能会发生利用成本较高的网络进行阅读的现象，从而产生潜在的性能问题。

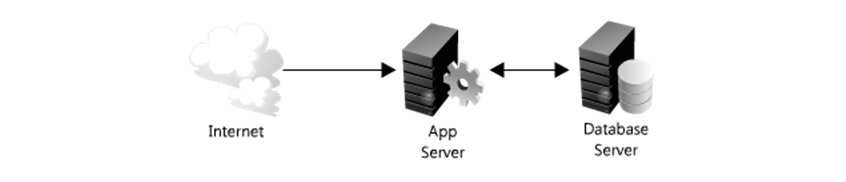
另一个潜在的话题是非合成。 让我们假设一下，共享的资源中存在阅读和书写两种不同的服务。 这里有可能发生竞争条件。 当任何数据要更新时，如果阅读请求比更新请求更早发生，那么相应数据将处于非合成状态。 例如，某客户端发送更新请求，将某图像名称从Dog改为Gizmo，同时，如果其他客户端正在阅读该图像，就会产生竞争条件。 在这种情况下，第二个用户是使用"Dog"还是"Gizmo"来接收，目前尚不清楚。

虽然分管数据有困难，但分管可以使每个问题（数据、路线、使用模式等）以小单位分开。 这一点虽然有助于扩张性和管理性，但也不是没有风险。 有许多减少这些危险因素和解决障碍的方法。 但这篇文章不会涉及。 如果有兴趣，请阅读作者关于故障承受（fault tolerance）和监控的博客文章。

快速、可扩展的数据存取的大厦砌块.

到目前为止，我们了解了设计分散系统的核心考虑事项。 本章要说明困难部分的数据存取扩展。

让我们通过简单的例子来确认为什么会有扩展性的数据存取困难。 像LAMP堆栈制作的应用软件一样，大部分简单的网页应用都和《图5》有着同样的结构。

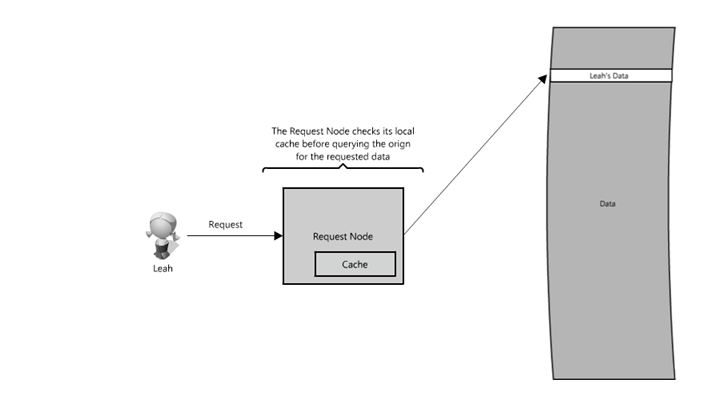


《图5》

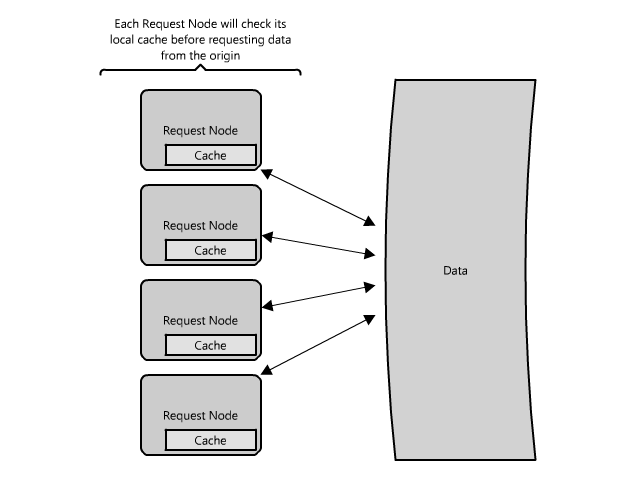
Cash

卡西是基于"地区性原理（locality of reference）"的方法。 Cash在硬件、操作系统、网络浏览器、网络应用等多种地方使用。 缓存就像一个非常短的时间的内存. 缓存容量虽然非常有限，但通常比原来的数据库保存得更快更频繁。 卡西可以位于架构的所有阶段，但很多时候位于靠近前台的地方。 因为通常Cash会使用到服务后台的时间性费用。

在实例的图像服务器架构中， 缓存可供快速数据处理？ 在这种情况下，有两种选择可以增加现金。 三是像《图8》一样，将现金添加到请求节点的方法。



将缓存配置到请求节点可以使响应数据从本地存储空间中获取。 每次请求都是通过服务发送，当请求节点上存在数据时，该节点会快速发送本地开出的数据。 如果缓存没有数据，请求节点将在磁盘上质疑数据。 在这里，Cash可能存在于请求节点的存储器中（会非常快），也可能存在于请求节点的本地磁盘中（比使用网络故事更快）。



如果将这些请求节点扩展到多个节点会发生什么事情？ 正如《图9》所示，如果请求的节点扩展到多个节点，每个节点都可以拥有各自的节点。 但是，如果活塞杆平衡器任意分散请求，那么同样的请求可能会被转移到另一个节点。 即，现金失误会增加。 减少缓存并使用多个缓存的方法是全域缓存和分散缓存。