WEB Based 系统可靠性及优化

2016年06月

（北京邮电大学吴国仕）

第一部分：概述

WEB BASED 系统概述

- WEB BASED信息系统，企业关注什么？
  - ✓ 功能，简单易用（功能设计）
  - ✓ 24*7？（硬件）（集群）
  - ✓客户增多了，系统是否可以满足其要求？（硬件）（集群）
  - ✓在不改变系统代码的情况下，增加机器？（硬件）（集群）
  - ✓系统响应时间？（软件编程实现）

WEB BASED 系统概述

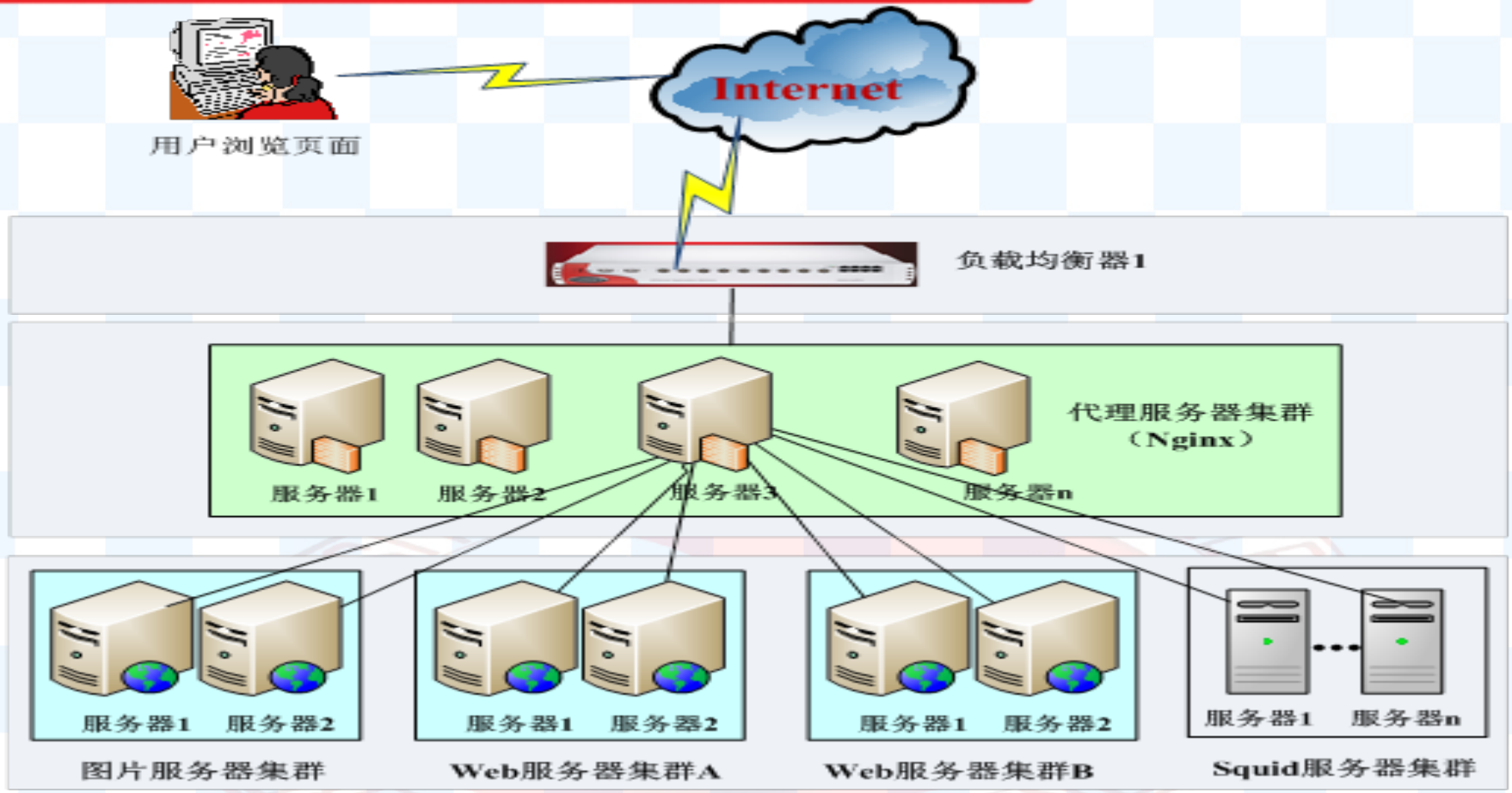
硬件保证系统

- ✓24*7 需要集群来保证
- ✓数据的可靠性，需要数据库的方面的保证，比如数据的备份、数据库的集群，异地容灾等
- ✓系统的性能，需要从系统的客户端及服务器端来考虑。

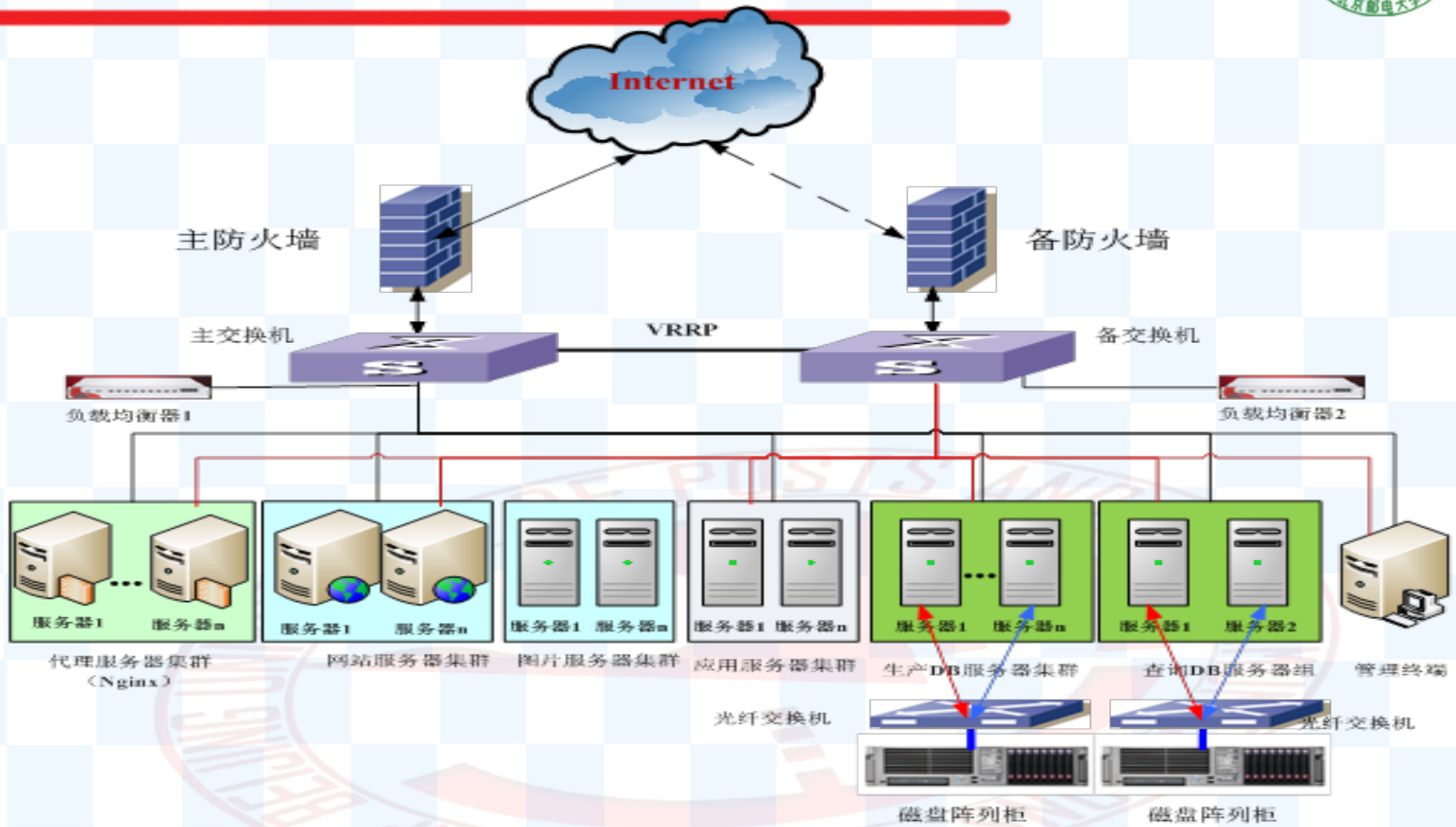
WEB BASED 系统概述

- ✓ 在硬件确定的情况下，系统响应时间？（软件编程实现）
  - 服务器端的系统优化
  - 客户端的系统优化

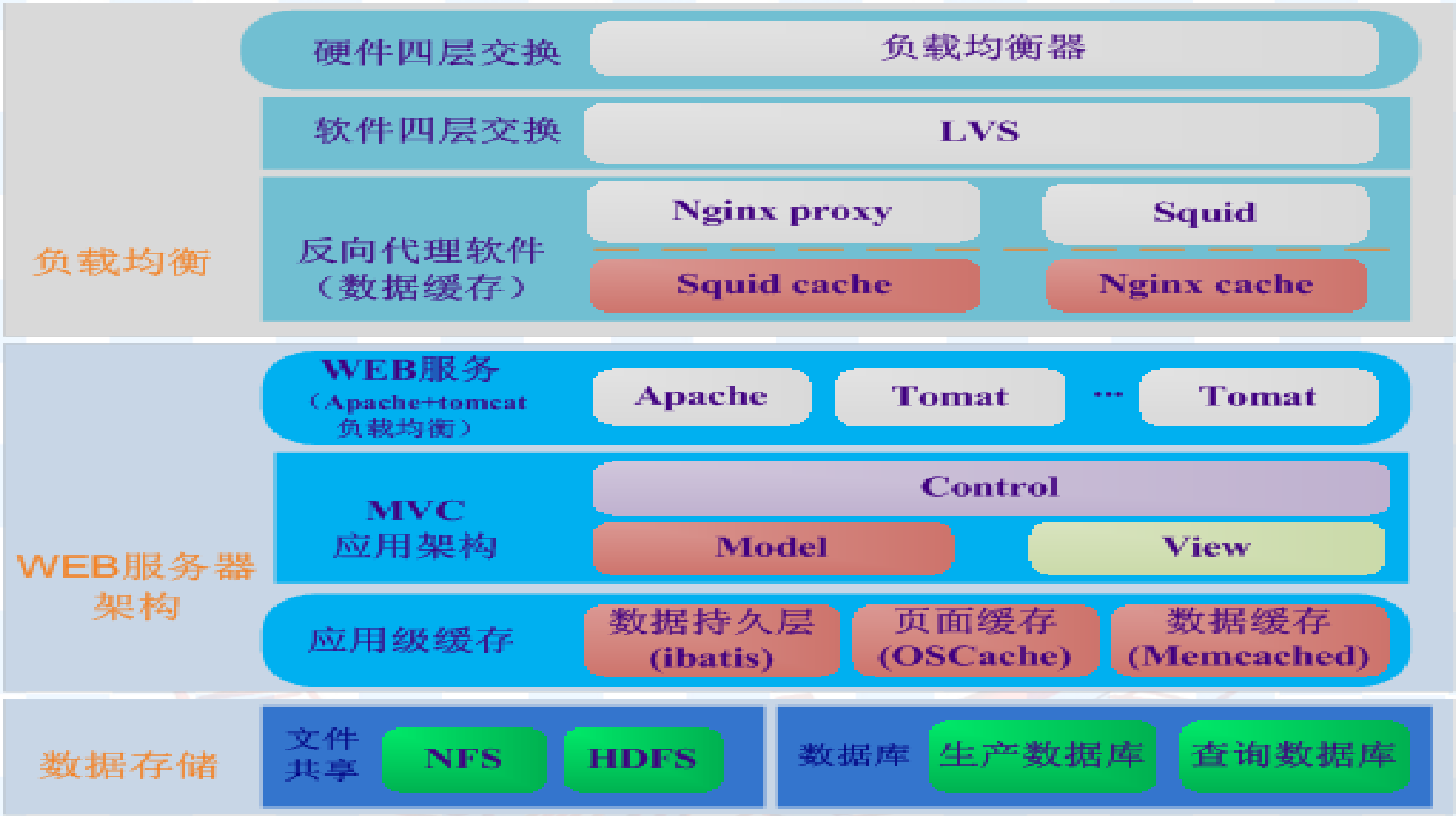
网站的物理架构实例


用户浏览页面
Internet
负载均衡器1
服务器1
服务器2
服务器3
服务器n
代理服务器集群
（Nginx）
服务器1
服务器2
图片服务器集群
服务器1
服务器2
Web服务器集群A
服务器1
服务器2
Web服务器集群B
服务器1
服务器n
Squid服务器集群

网络拓扑结构实例


Internet
主防火墙
备防火墙
主交换机
VRRP
备交换机
负载均衡器1
负载均衡器2
服务器1
服务器m
代理服务器集群
（Nginx）
服务器1
服务器n
网站服务器集群
服务器1
服务器m
图片服务器集群
服务器1
服务器n
应用服务器集群
服务器1
服务器n
生产DB服务器集群
服务器1
服务器2
查询DB服务器组
管理终端
光纤交换机
光纤交换机
磁盘阵列柜
磁盘阵列柜

网站的系统分层架构实例

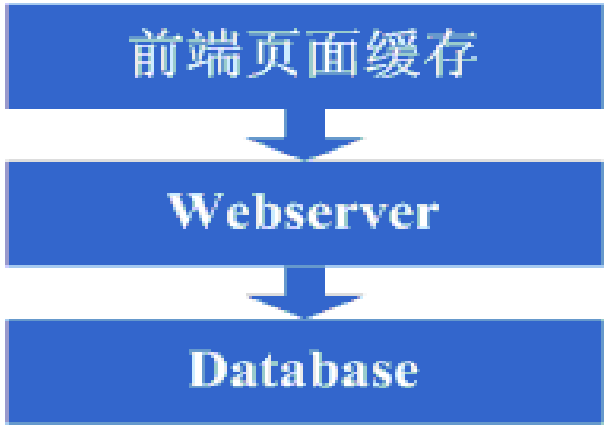


服务器端系统优化-服务器结构发展

- 构架演变第一步：web应用和数据库分离。

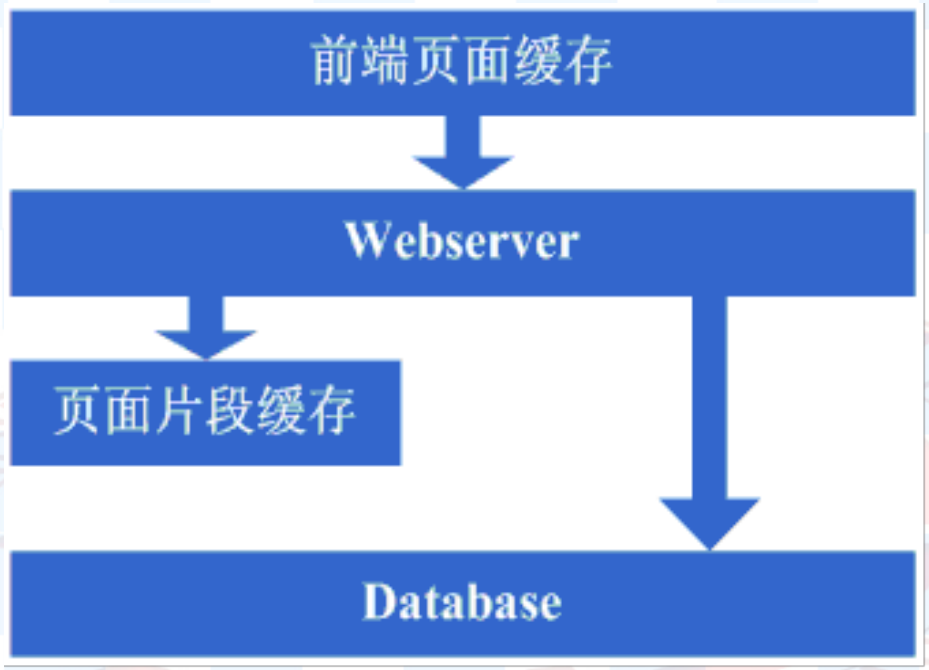
服务器端系统优化-服务器结构发展

- 构架演变第二步：增加页面缓存



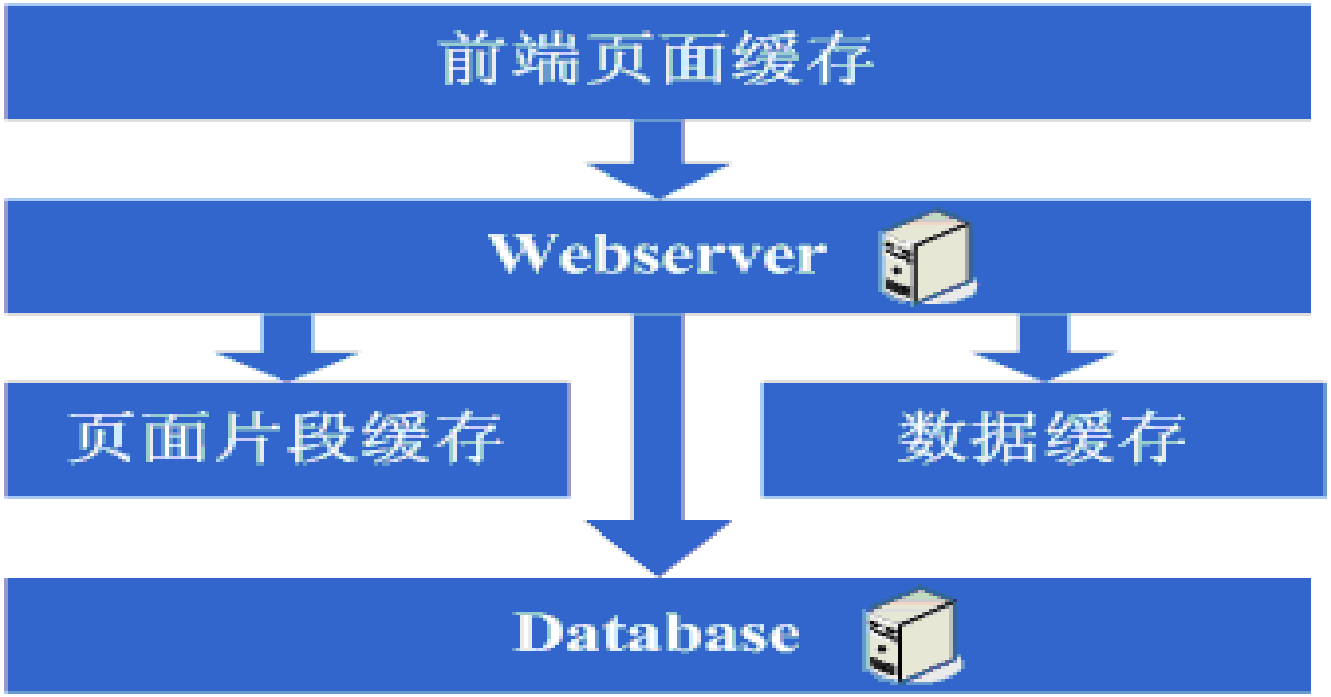
服务器端系统优化-服务器结构发展

- 构架演变第三步：增加页面片段缓存



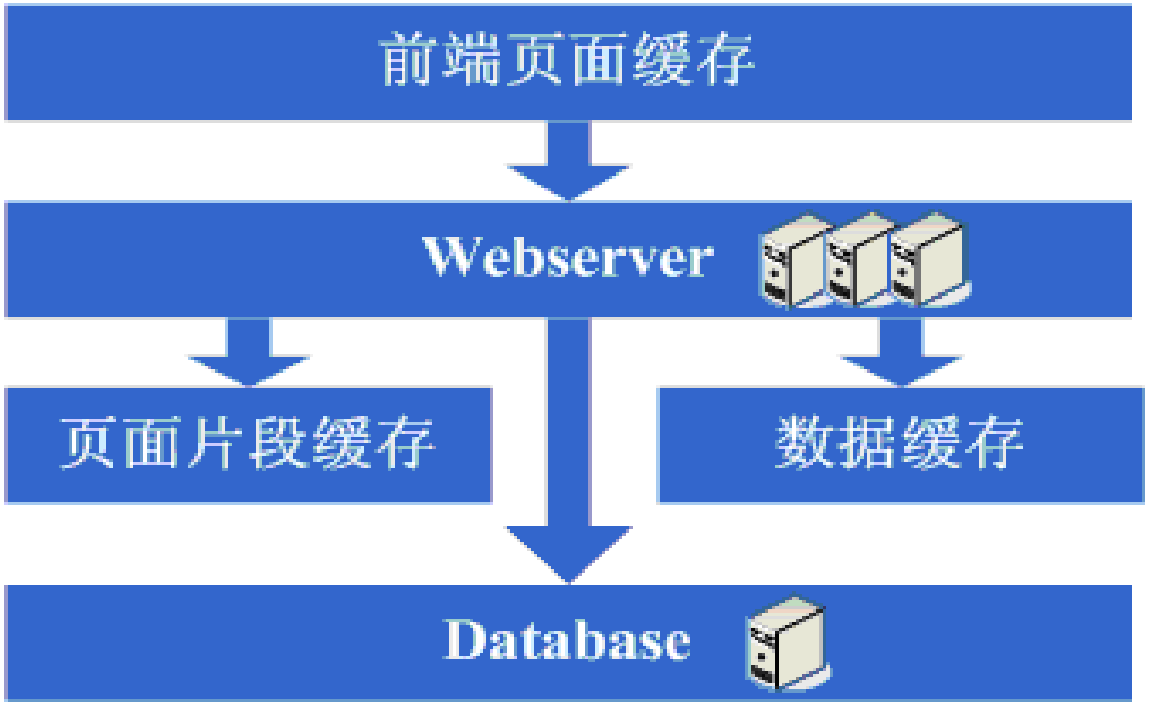
服务器端系统优化-服务器结构发展

- 构架演变第四步：数据缓存



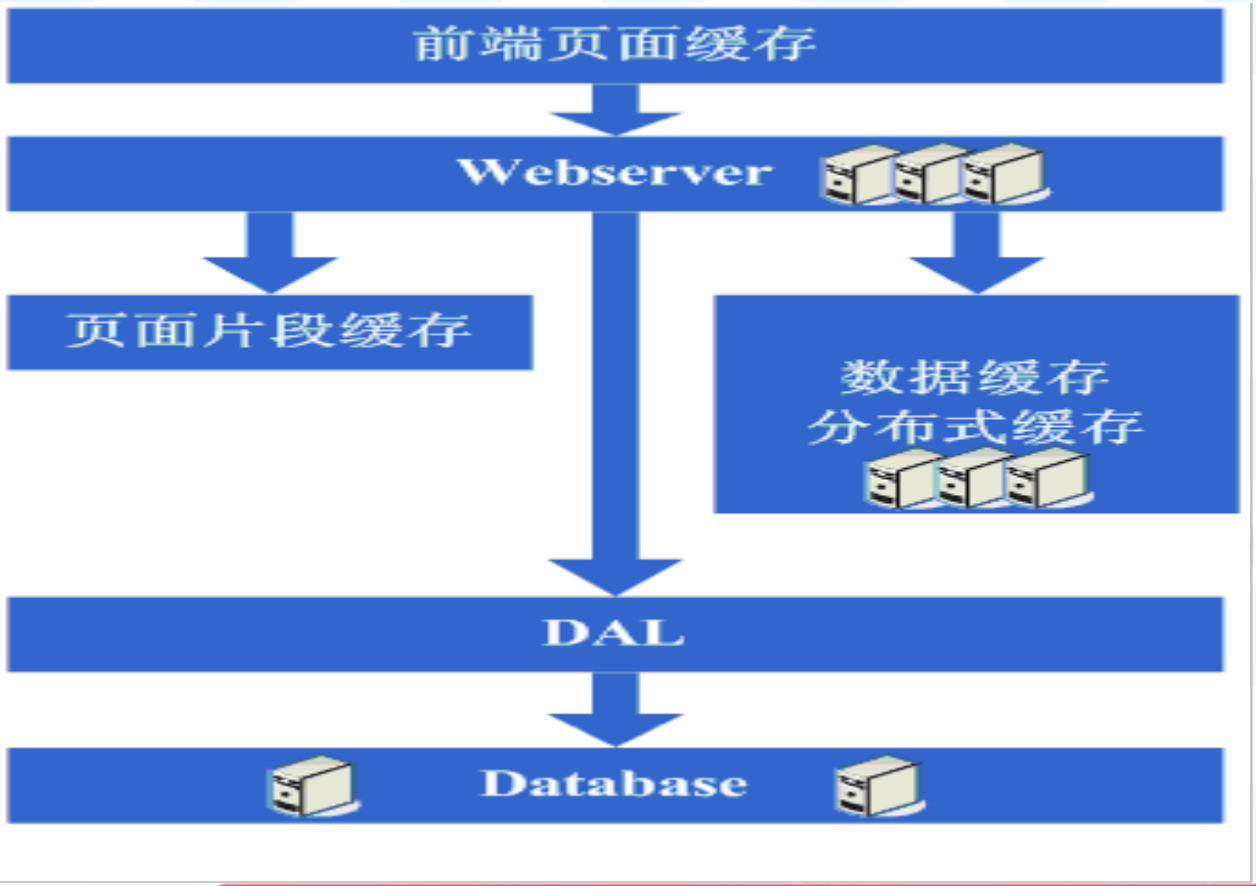
服务器端系统优化-服务器结构发展

- 构架演变第五步：增加webserver



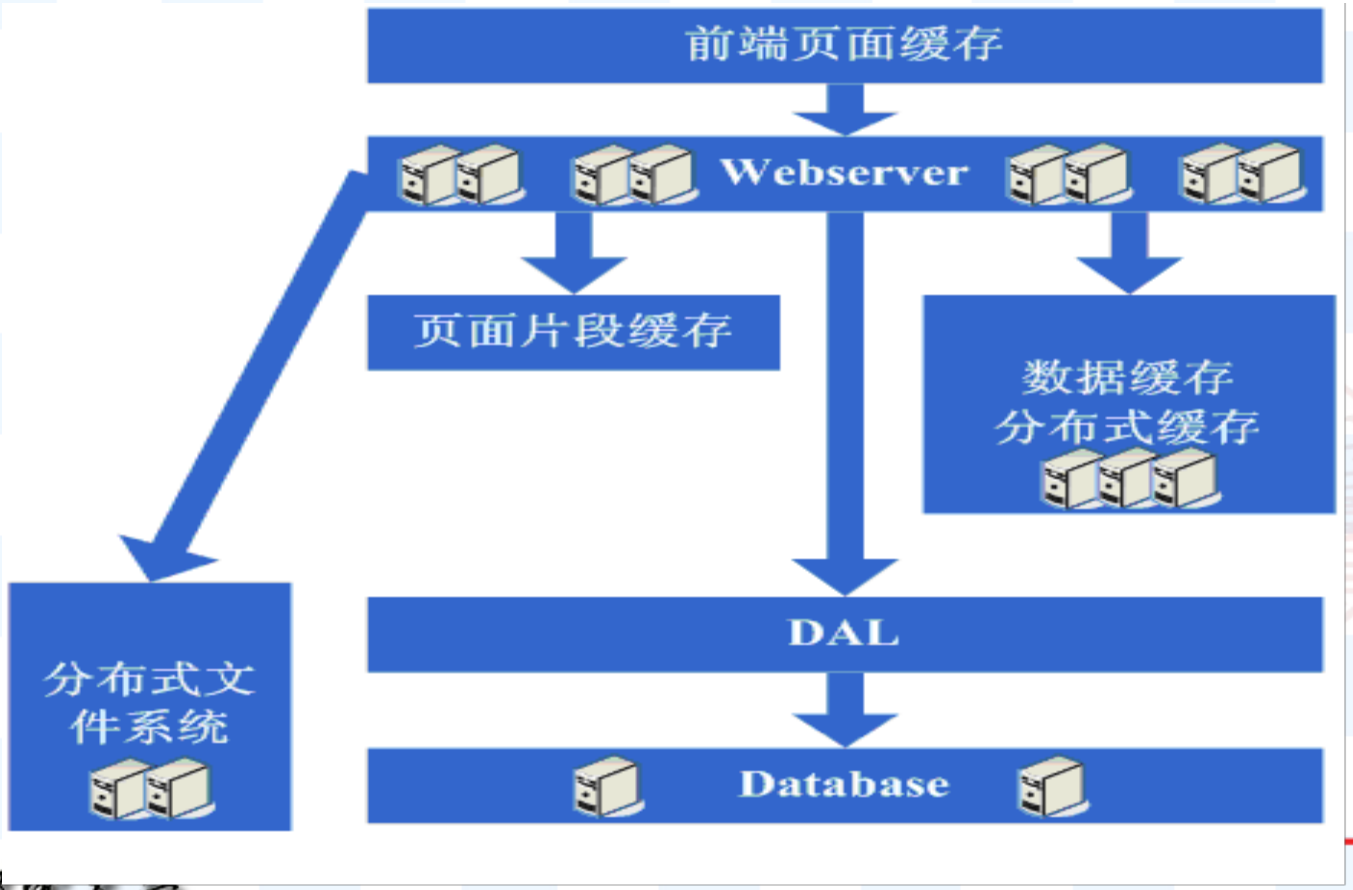
服务器端系统优化-服务器结构发展

- 构架演变第六步：分库



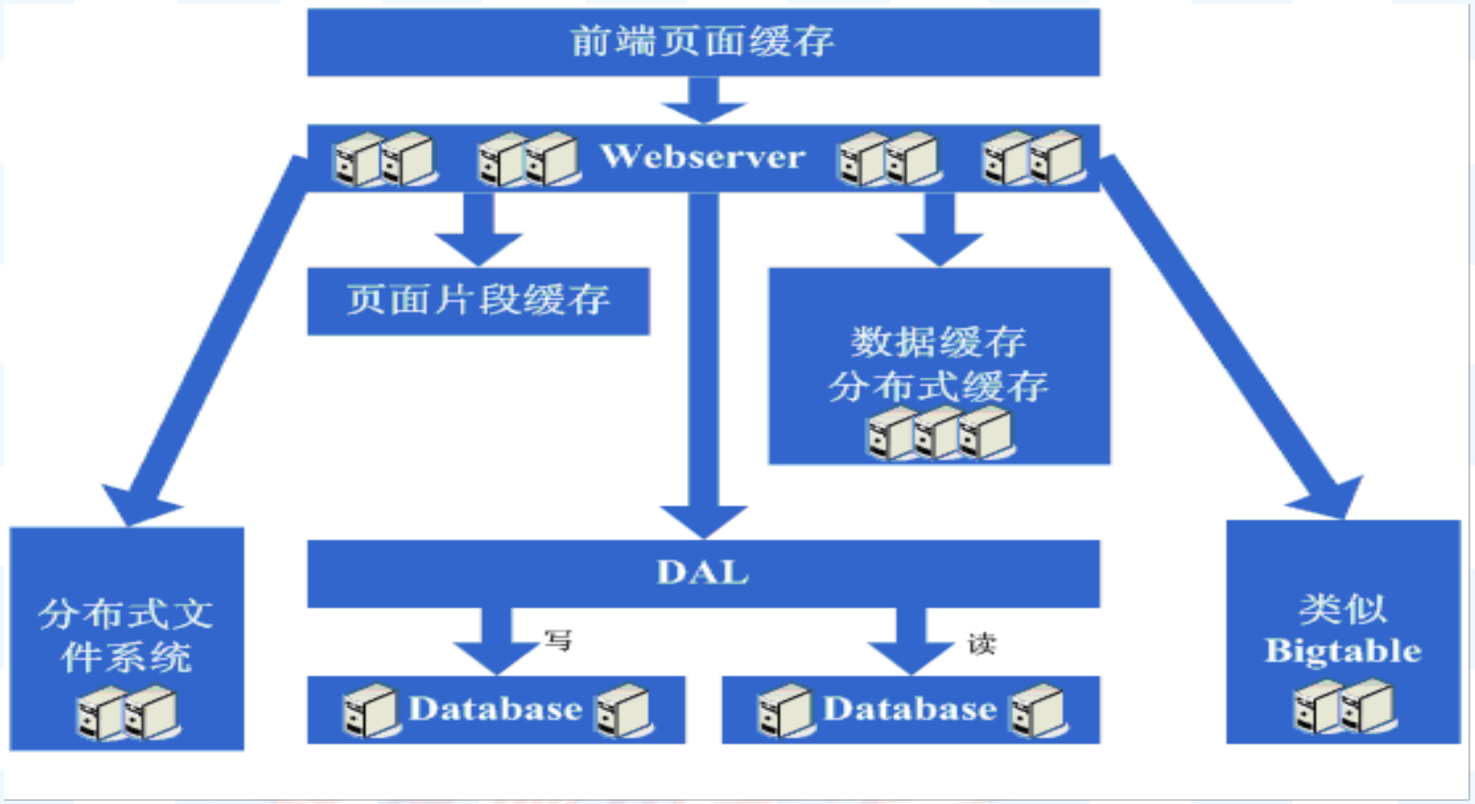
服务器端系统优化-服务器结构发展

- 构架演变第七步：增加更多的web server



服务器端系统优化-服务器结构发展

- 构架演变第八步：数据读写分离和廉价存储方案



第二部分：系统集群概念

背 景

集群技术的出现和IA架构服务器的快速发展为社会的需求提供了新的选择。它价格低廉，易于使用和维护，而且采用集群技术可以构造超级计算机，其超强的处理能力可以取代价格昂贵的中大型机，为行业的高端应用开辟了新的方向。

集群技术是一种相对较新的技术，通过集群技术，可以在付出较低成本的情况下获得在性能、可靠性、灵活性方面的相对较高的收益。

目前，在世界各地正在运行的超级计算机中，有许多都是采用集群技术来实现的。

什么是集群？

集群是由一些互相连接在一起的计算机构成的一个并行或分布式系统。这些计算机一起工作并运行一系列共同的应用程序，同时，为用户和应用程序提供单一的系统映射。从外部来看，它们仅仅是一个系统，对外提供统一的服务。集群内的计算机物理上通过电缆连接，程序上则通过集群软件连接。这些连接允许计算机使用故障应急与负载平衡功能，而故障应急与负载平衡功能在单机上是不可能实现的。

集群的目的

服务器集群系统通俗地讲就是把多台服务器通过快速通信链路连接起来

- ✓从外部看来，这些服务器就像一台服务器在工作
- ✓而对内来说，外面来的负载通过一定的机制动态地分配到这些节点机中去，从而达到超级服务器才有的高性能、高可用。

集群的优点

- 高可伸缩性：服务器集群具有很强的可伸缩性。随着需求和负荷的增长，可以向集群系统添加更多的服务器。在这样的配置中，可以有多台服务器执行相同的应用和数据库操作。
- 高可用性：高可用性是指，在不需要操作者干预的情况下，防止系统发生故障或从故障中自动恢复的能力。通过把故障服务器上的应用程序转移到备份服务器上运行，集群系统能够把正常运行时间提高到大于99.9%，大大减少服务器和应用程序的停机时间。
- 高可管理性：系统管理员可以从远程管理一个、甚至一组集群，就好象在单机系统中一样。

集群与管理

构建集群系统必须包含对系统及网络管理的两方面的考虑。服务器集群十分复杂，而复杂的技术又往往会引入许多人为的错误，因此系统应有网络资源管理、系统监测管理，并具有可以简化管理过程的工具。

如果仅仅把集群视为单一系统或把它视为分立的服务器，那么这种管理软件是不能胜任集群管理工作的。当我们观察集群上运行的一个应用程序时，需要站在单一系统角度；当我们试图区分、定位一个出错部件时，又需要站在分立服务器角度。如果管理系统不能提供必需的监测及管理能力，那么该集群是不能在重要的应用环境中投入使用的。

集群与成本

并不是所有的服务器都需要采用系统级的冗余。因此，我们需要对比一下是系统发生故障所造成的损失大，还是购买及管理一个集群系统的费用高，从中找出一个较好的选择。

集群的类型

集群系统（Cluster），有时也称为机群或群集系统。

集群技术本身有很多种分类，市场上的产品很多，也没有很标准的定义，较为常见的主要分为三种类型。

- 高可用性集群(High Availability Cluster)/容错集群(Fail-over Cluster)
- 负载均衡集群(Load balancing Cluster)
- 高性能计算集群(High Performance Computing Cluster)高性能计算集群具有响应海量计算的性能，主要应用于科学计算、大任务量的计算等。

高可用性集群

当集群中的一个系统发生故障时，集群软件迅速做出反应，将该系统的任务切换到集群中其它正在工作的系统上执行。

考虑到计算机硬件和软件的易错性，高可用性集群的目的主要是为了使集群的整体服务尽可能可用。如果高可用性集群中的主节点发生了故障，那么这段时间内将由次节点代替它。次节点通常是主节点的镜像，所以当它代替主节点时，它可以完全接管其身份。

高可用性集群

高可用性(HA)集群致力于使服务器系统的运行速度和响应速度尽可能快。它们通常利用在多台机器上运行的冗余节点和服务进行相互跟踪。如果某个节点失败，它的替补将在几秒钟或更短时间内接管它的职责。因此，对于用户而言，群集永远不会停机。

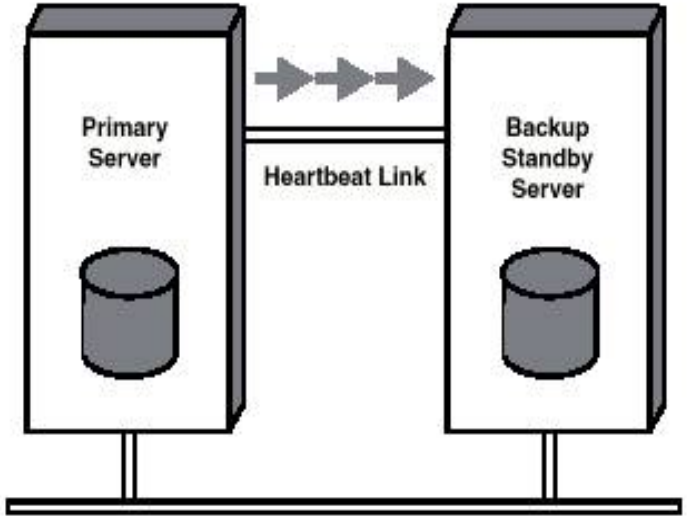
有些HA集群还可以实现节点间冗余应用程序。即使用户使用的节点出了故障，他所打开的应用程序仍将继续运行，该程序会在几秒之内迁移到另一个节点，而用户只会感觉到响应稍微慢了一点。但是，这种应用程序级冗余要求将软件设计成具有集群意识的，并且知道节点失败时应该做什么。

高可用性集群拓扑结构

有两种典型的拓扑结构可以实现高可用性：主从服务器和活动第二服务器。

主从服务器

通常把一个服务器安排为“主”服务器，一个服务器为“第二”服务器；由主服务器为用户提供服务，第二服务器除了在主服务器出错时接管工作外，没有其它用处。



两台服务器通过一种被称为“心跳”（heartbeat）的机制进行连接，用于监控主服务器的状态，一旦发现主服务器宕机或出现不能正常工作的情况，心跳会通知第二服务器，接替出问题的主服务器。

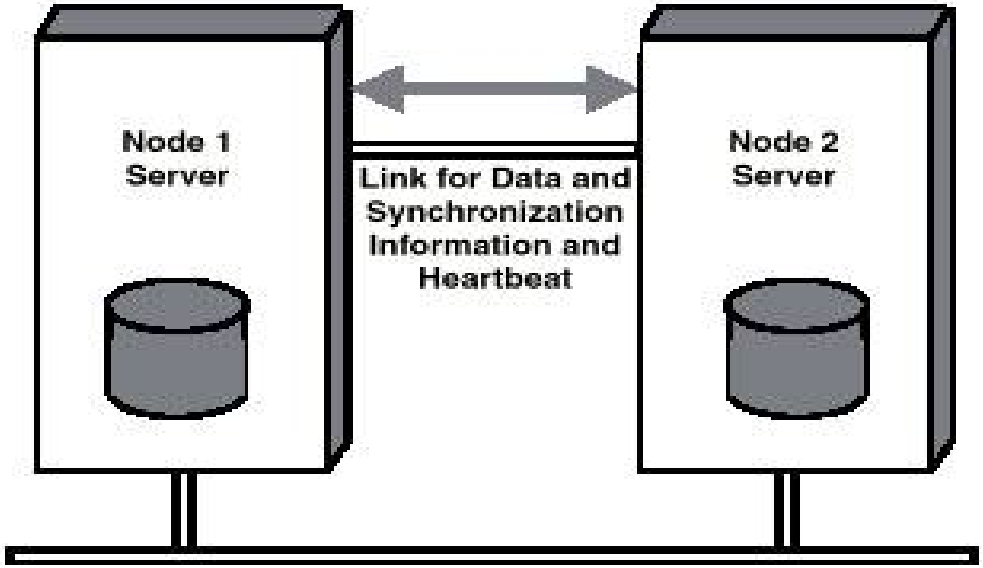
“心跳” 可以通过专用线缆、网络链接等方式实现。

活动第二服务器

一个功能基本一致但成本低得多的方式是使第二服务器可以处理其它的应用程序，当主服务器发生故障时，第二服务器能够接管主服务器的工作。这种被称为“活动第二服务器”方法的主要优点是在保持使用第二服务器的同时，获得服务器冗余，而不是仅仅把第二服务器作为备份使用。这种方法可以降低集群系统的运行费用。

活动第二服务器有三种实现形式：“全部复制”、“0共享”和“全部共享”。

“全部复制”方式



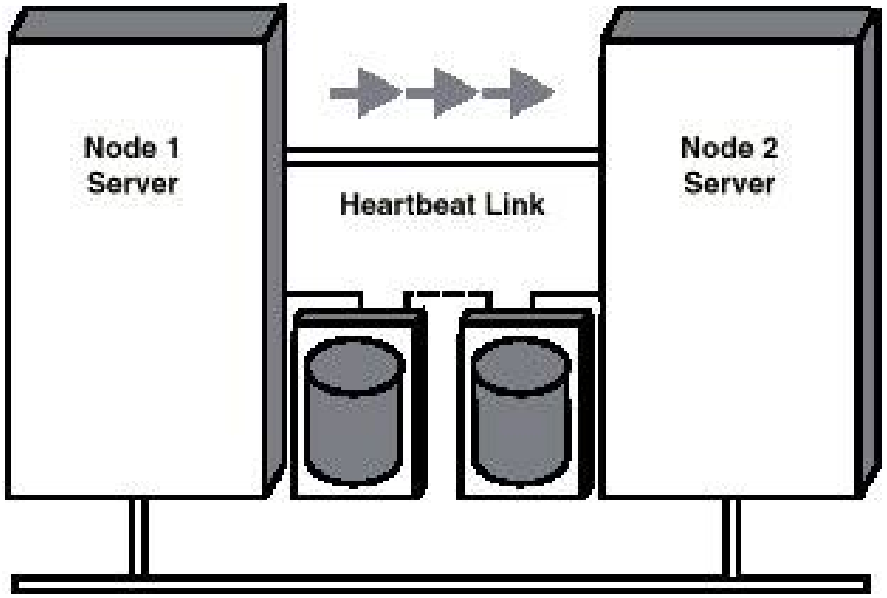
第一种方式称作“全部复制”，就是指彻底的服务器冗余。每个服务器都有自己的磁盘。数据不断地被拷贝到第二服务器的磁盘上，以保证故障发生时，第二服务器可以使用当前的数据。

“全部复制”方式的优缺点

这种方法增加了服务器及网络的负荷，可能会严重影响系统性能。另一个缺点是当一个服务器发生故障时，可能会有主、从服务器的不一致现象：某个磁盘上的事务处理可能并没有完全在另一个磁盘上得到继续。因为即便以最快的网络相连，两个服务器间信息传送仍会有一定的延迟。

由于数据被完全复制，所以应用程序可以在任一服务器上运行，从而可以更好地平衡负载。另外，节点在空间上可以是分散的，节点间可通过广域网互连，物理上可以距离很远。

“0共享”方式

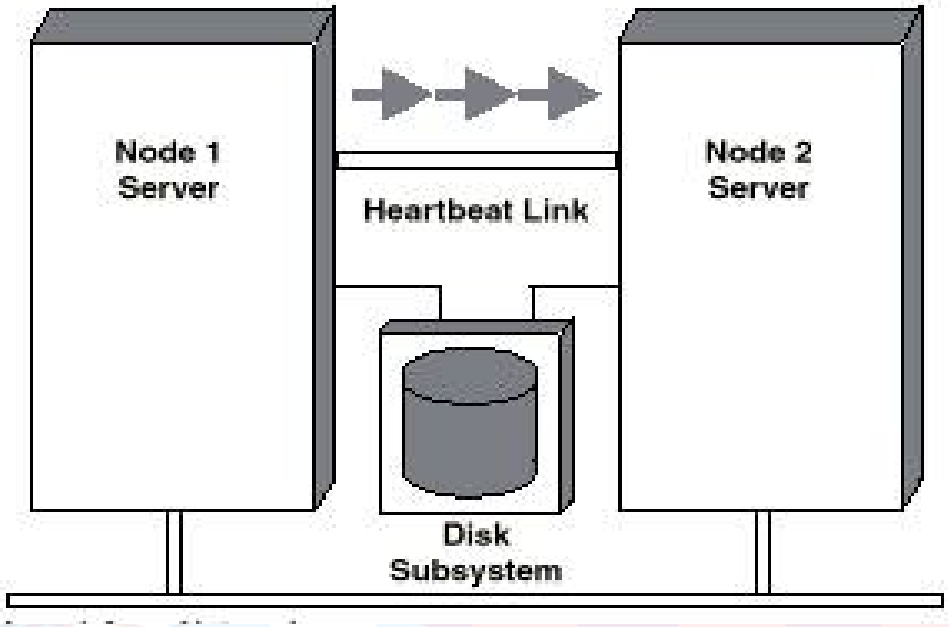


第二种方式是“0共享”，它是指两个服务器物理上连接到同一个磁盘组上，每个服务器都“拥有”自己的磁盘。在正常情况下，各服务器只能存取各自的数据。当一台服务器发生故障时，另一台服务器自动获得对方磁盘的读写权限，并对之进行操作。

“0共享”方式的优缺点

这种方式不必不停地在服务器间拷贝数据，从而大大降低了网络负荷。在这种方式下，磁盘是唯一可能产生导致长时间停机故障的地方。所以，在这种体系结构中，一般在磁盘子系统中采用RAID（Redundant Arrays of Inexpensive Disks，RAID）技术或者采用冗余镜象磁盘，以保证在磁盘出错时，应用程序和数据的可用性。

“全部共享”方式



第三种方式是“共享一切”，也就是说让多个服务器在同一时间共享同一磁盘。在这种方式中，所有与磁盘相连的服务器在正常运行时可在相同时刻共享磁盘存取通道。这种方式要求开发一个复杂的锁定管理软件，保证在一个时刻只有一个服务器在读写数据。

“全部共享”方式优缺点

“共享一切”方法也需要采用RAID技术或磁盘镜象来保障数据的安全。

由于每个服务器的数据被送到相同的磁盘上，所以不必在服务器间复制数据。这样在正常运作时，可以在不影响系统性能的前提下，提供高可用性。

高可用性集群对比一览表

集群方法	描述	优点	局限性
主从服务器	只是在主服务器发生故障时，第二服务器才能投入运行，接管一切。	易于实现。	成本高。因为第二服务器不能处理其它任务。
活动第二服务器	第二服务器也被用来运行任务处理。	成本低。因为第二服务器也能运行。	复杂性增加。
“全部复制”	每个服务器都有自己的磁盘。主、次服务器之间不停地进行数据拷贝。	高可用性和容错。适合于对可用性敏感的环境。	拷贝操作使网络及服务器负荷很大。可能会有发生不同步的风险。有故障发生时，可能会有丢失事件。应用程序需要全面的修改。
“0共享”	服务器连到相同的磁盘系上，但每个服务器都拥有属于自己的磁盘，如果某个服务器出错，它的磁盘将由另一服务器接管。	因为无需拷贝数据，所以降低了网络及服务器的一般运行开销。	通常需要磁盘镜象或RAID技术来补偿磁盘故障给系统带来的灾害。
“全部共享”	多服务器可同时共享磁盘存取。	低网络及服务器运行开销。由于磁盘故障而引发系统停机的风险被降低。	需要锁定管理软件；需要磁盘镜象或RAID技术。

负载均衡集群

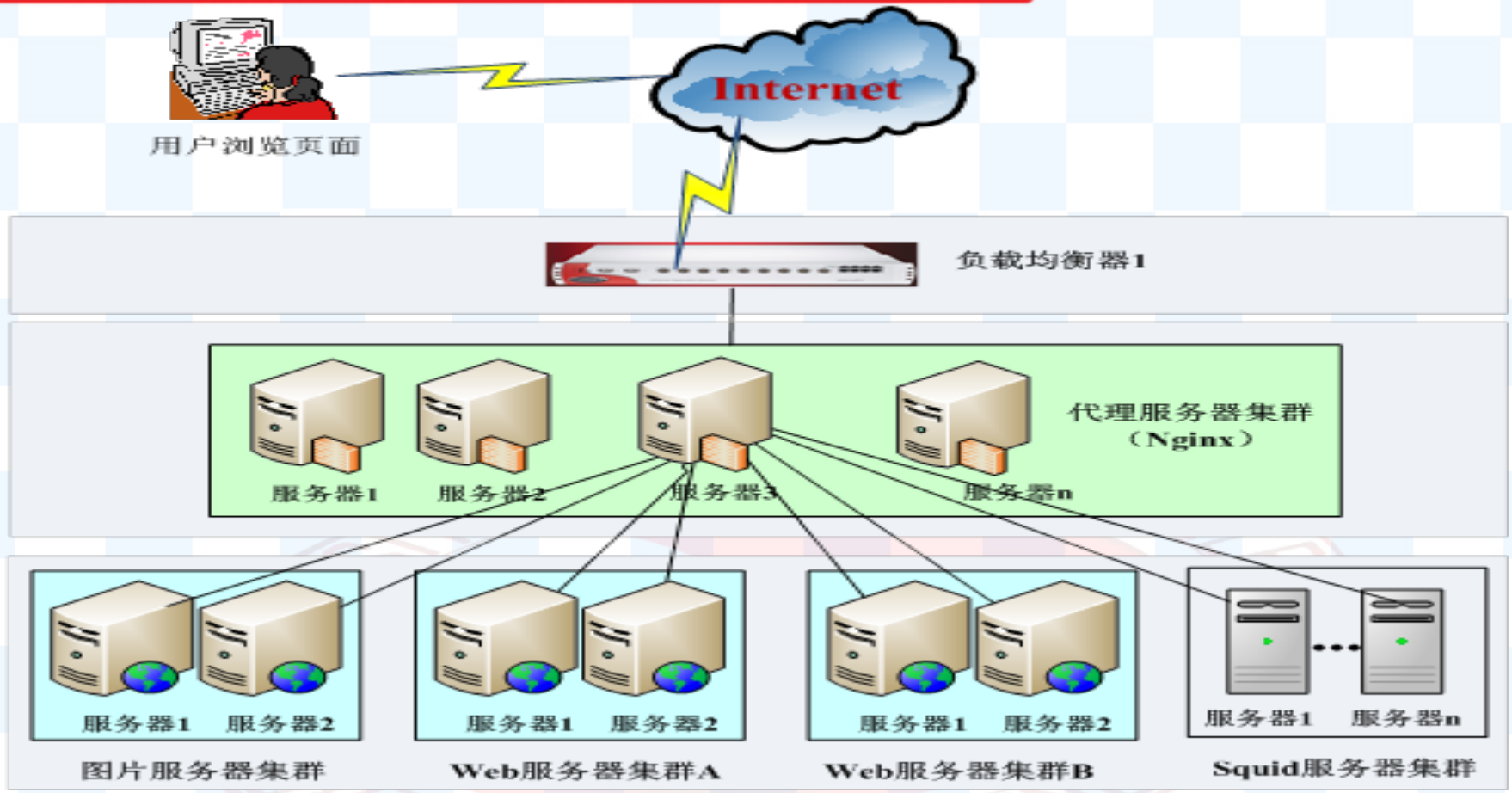
对于企业来说，负载均衡集群是应用面最广、最有发展潜力的集群应用形式。

随着信息化的发展，用户需求成几何方式增长，从而导致网站的访问量不断增长。由于企业资源中很多是客户信息，多媒体素材，包括音频、视频、动画等，会产生极大的网络流量负载和处理负载。在这种情况下，采用负载均衡集群是一个极好的选择。

负载均衡集群一般用于WEB服务器、代理服务器等。这种集群可以在接到请求时，检查接受请求较少，不繁忙的服务器，并把请求转到这些服务器上。

网络负载均衡功能增强了Web服务器、流媒体服务器和终端服务等Internet服务器程序的可用性和可伸缩性。

网站的物理架构实例


用户浏览页面
Internet
负载均衡器1
服务器1
服务器2
服务器3
服务器n
代理服务器集群
（Nginx）
服务器1
服务器2
图片服务器集群
服务器1
服务器2
Web服务器集群A
服务器1
服务器2
Web服务器集群B
服务器1
服务器n
Squid服务器集群

代理服务器与反向代理服务器

- 代理服务器称为转发代理服务器

普通的转发代理服务器是客户端与原始服务器之间的一个中间服务器。为了从原始服务器获取内容，客户端发送请求到代理服务器，然后代理服务器从原始服务器中获取内容再返回给客户端。客户端必须专门地配置转发代理来访问其他站点，如在浏览器中配置代理服务器地址及端口号等。

- ✓ 比如校园网用户通过代理访问国外网站
- ✓ 公司内网用户通过公司的统一代理访问外部Internet网站等

代理服务器与反向代理服务器

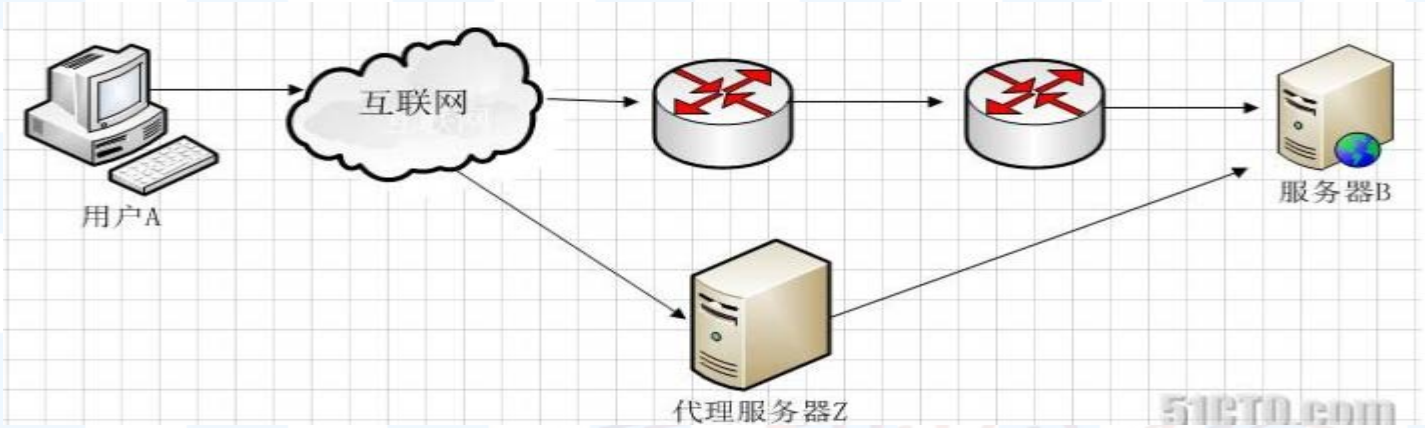
- 反向代理服务器

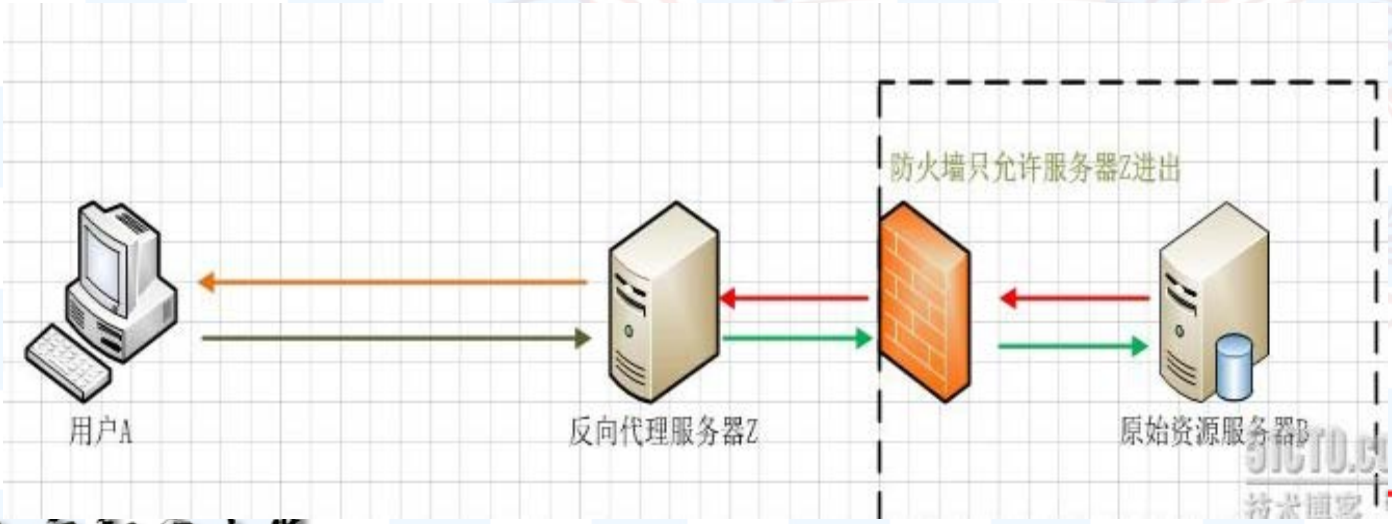
而反向代理服务器则相反，在客户端来看它就像一个普通的Web服务器。客户端不要做任何特殊的配置。客户端发送普通的请求来获取反向代理所属空间的内容。反向代理决定将这些请求发往何处，然后就好像它本身就是原始服务器一样将请求内容返回。

- ✓ 典型应用就是为处于防火墙后的服务器提供外部Internet用户的访问
- ✓ 反向代理能够用于在多个后端服务器提供负载均衡，或者为较慢的后端服务器提供缓存。
- ✓ 反向代理还能够简单地将多个服务器映射到同一个URL空间。

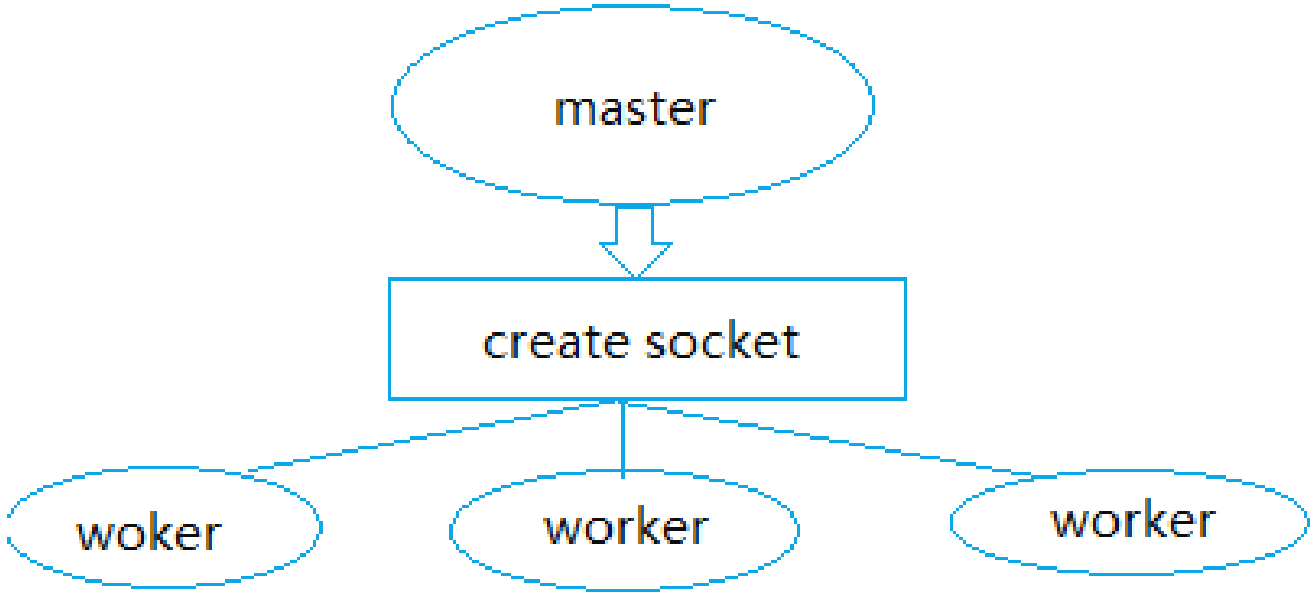
代理服务器与反向代理服务器

- 两者区别





NginX架构



代理服务器与反向代理服务器

- 两者区别
- 相同点：两者的相同点在于都是用户和服务器之间的中介，完成用户请求和结果的转发。
- 不同点：
  - ✓转发代理的内部是客户端，而反向代理的内部是服务器。即内网的客户端通过转发代理服务器访问外部网络，而外部的用户通过反向代理访问内部的服务器。
  - ✓转发代理通常接受客户端发送的任何请求，而反向代理通常只接受到指定服务器的请求。如校园网内部用户可以通过转发代理访问国外的任何站点(如果不加限制的话)，而只有特定的请求才发往反向代理，然后又反向代理发往内部服务器。

负载均衡集群的特点

所有节点对外提供相同的服务，这样可以实现对单个应用程序的负载均衡，而且同时提供了高可用性，性能价格比极高。

网络流量负载均衡是一个过程，它检查集群的入网流量，然后将流量分发到各个节点以进行适当处理。负载均衡网络应用服务要求群集软件检查每个节点的当前负载，并确定哪些节点可以接受新的作业。因此，集群中的节点(包括硬件和操作系统等)没有必要是一致的。

负载均衡-基于DNS的负载均衡--一个域名绑定多个IP

在DNS服务器中，可以为多个不同的地址配置同一个名字，而最终查询这个名字的客户机将在解析这个名字时得到其中的一个地址。因此，对于同一个名字，不同的客户机会得到不同的地址，它们也就访问不同地址上的Web 服务器，从而达到负载均衡的目的。

缺点：

- DNS 服务器将Http请求平均地分配到后台的Web服务器上， 最慢的 Web服务器将成为系统的瓶颈，处理能力强的服务器不能充分发挥作用；
- 未考虑容错，如果后台的某台Web服务器出现故障，DNS服务器仍然会把DNS 请求分配到这台故障服务器上，导致不能响应客户端。
- 最后一点是致命的，有可能造成相当一部分客户不能享受Web服务，并且由于DNS缓存的原因，所造成的后果要持续相当长一段时间(一般DNS的刷新周期约为24小时)。所以在国外最新的建设中心Web站点方案中，已经很少采用这种方案了。

负载均衡-基于通过硬件四层交换实现负载均衡

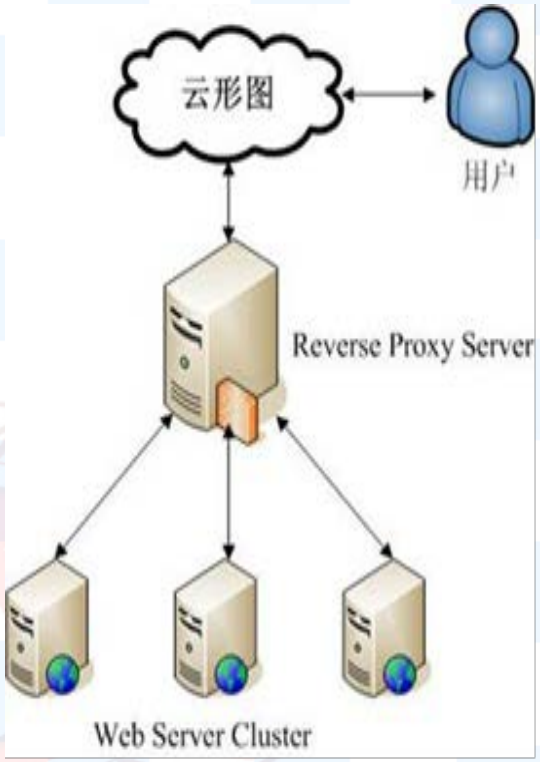
- 在硬件四层交换产品领域，有一些知名的产品可以选择，比如Alteon、F5等，这些产品很昂贵，但是物有所值，能够提供非常优秀的性能和很灵活的管理能力。Yahoo中国当初接近2000台服务器使用了三四台Alteon就搞定了

负载均衡-通过软件四层交换实现负载均衡

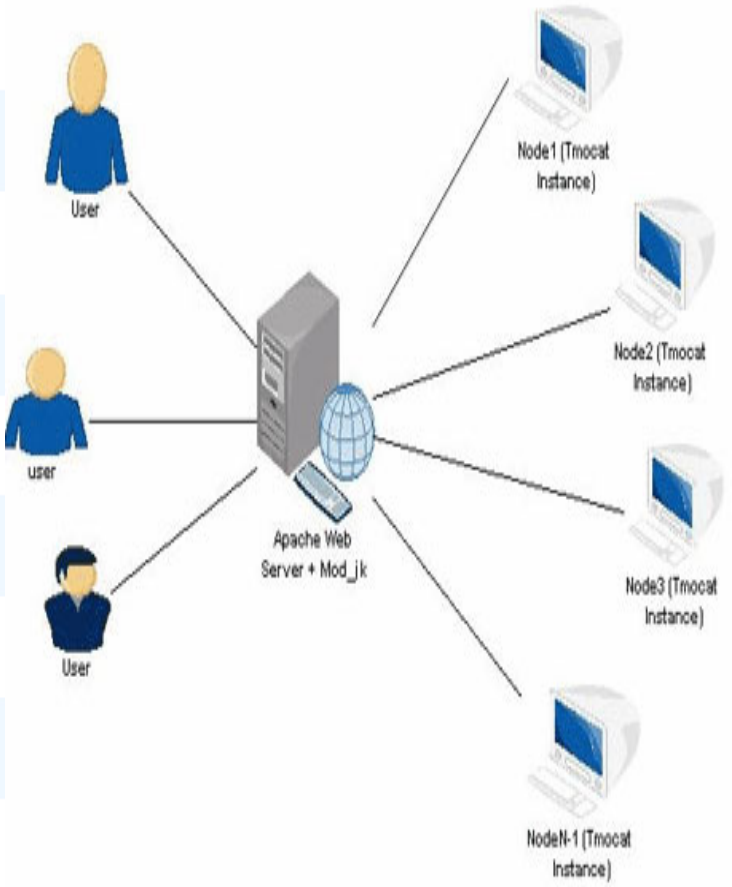
- 软件四层交换我们可以使用Linux上常用的LVS来解决，LVS就是Linux Virtual Server，他提供了基于心跳线heartbeat的实时灾难应对解决方案，提高系统的鲁棒性，同时可供了灵活的虚拟VIP配置和管理功能，可以同时满足多种应用需求，这对于分布式的系统来说必不可少。
- 一个典型的使用负载均衡的策略就是，在软件或者硬件四层交换的基础上搭建squid集群，这种思路在很多大型网站包括搜索引擎上被采用，这样的架构低成本、高性能还有很强的扩张性。

负载均衡-通过反向代理服务器实现负载均衡

- 当互联网用户请求 WEB 服务时，DNS 将请求的域名解析为反向代理服务器的 IP 地址，这样 URL 请求将被发送到反向代理服务器，由反向代理服务器负责处理用户的请求与应答、与后台 WEB 服务器交互。利用反向代理服务器减轻了后台 WEB 服务器的负载，提高了访问速度，同时避免了因用户直接与 WEB 服务器通信带来的安全隐患。
- 常见的组件有：**Squid（Opensource)、Nginx**（“engine x”）



Apache +tomcat集群实现负载均衡



- 使用 apache和多个tomcat 配置一个可以应用的web网站，用Apache进行分流，把请求按照权重以及当时负荷分tomcat1,tomcat2...去处理，要达到以下要求：
  - ✓ Apache 做为HttpServer，通过mod_jk连接器连接多个 tomcat 应用实例，并进行负载均衡。
  - ✓ 同时还要配置session复制，也就是说其中任何一个tomcat的添加的session，是要同步复制到其它tomcat， 集群内的tomcat都有相同的session，并为系统（包括 Apache 和 tomcat）设定 Session 超时时间。

高性能计算集群

高性能计算集群具有响应海量计算的性能，主要应用于科学计算、大任务量的计算等。有并行编译、进程通讯、任务分发等多种实现方法。

高性能计算集群涉及为解决特定的问题而设计的应用程序，针对性较强，一般信息系统采用的比较少。

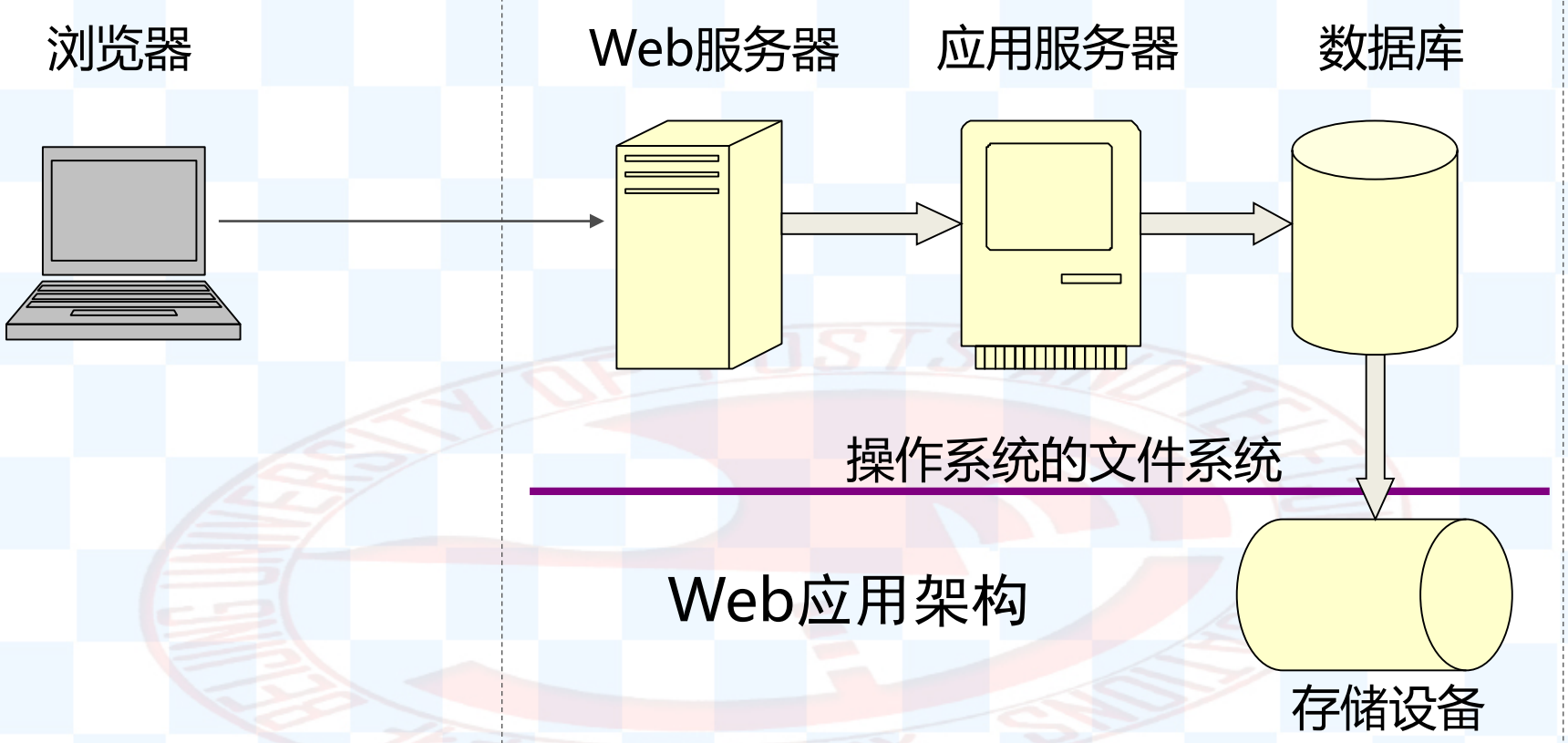
在集群的这三种基本类型之间，经常会发生混合。高可用性集群可以在其节点之间均衡用户负载。同样，也可以从要编写应用程序的集群中找到一个并行集群，使得它可以在节点之间执行负载均衡。从这个意义上讲，这种集群类别的划分只是一个相对的概念，而不是绝对的。

第三部分：服务器端优化技术

缓存是什么，解决什么问题？

- Cache是高速缓冲存储器 一种特殊的存储器子系统，其中复制了频繁使用的数据以利于快速访问
- 凡是位于速度相差较大的两种硬件/软件之间的，用于协调两者数据传输速度差异的结构，均可称之为 Cache

基于Web应用的系统架构图



Web应用系统存在哪些速度差异？

- 读取文件系统 → 读取磁盘
- 读取数据库内存 → 读取文件系统
- 读取应用内存 → 访问数据库服务器
- 读取静态文件 → 访问应用服务器
- 读取浏览器缓存 → 访问网站

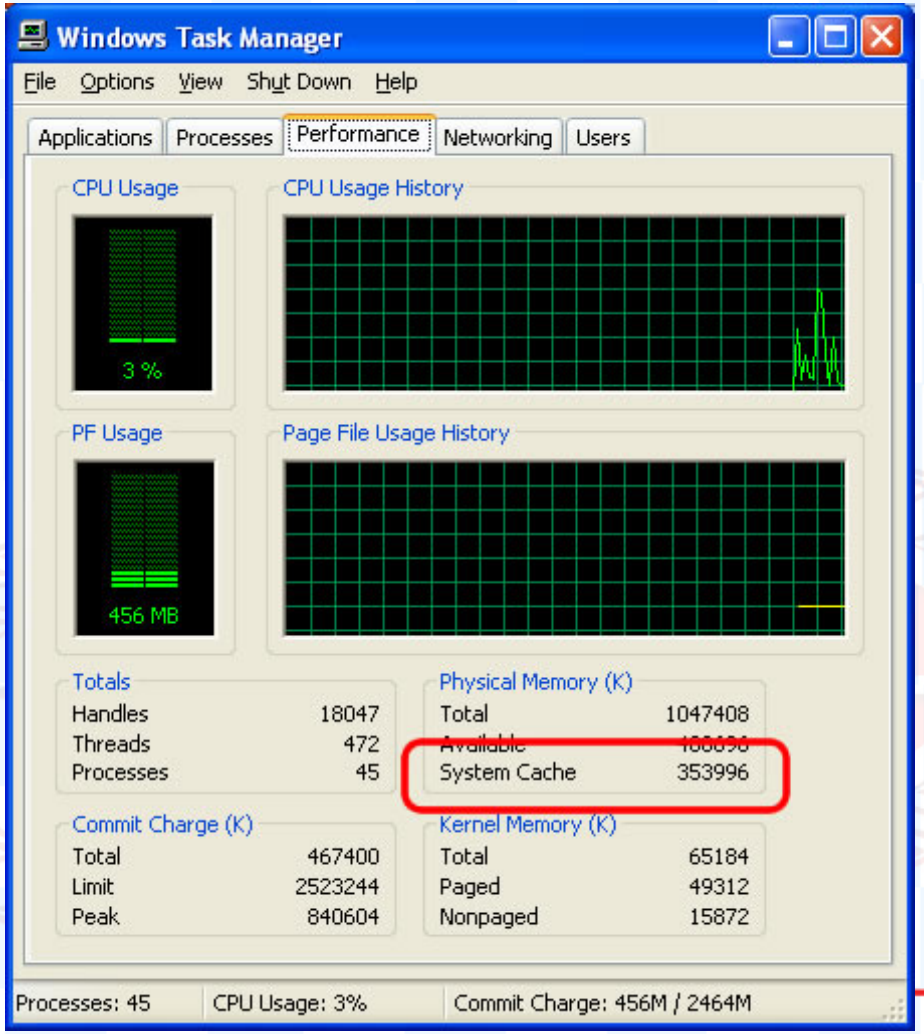
缓存技术分类

- 操作系统磁盘缓存 → 减少磁盘机械操作
- 数据库缓存 → 减少文件系统I/O
- 应用程序缓存 → 减少对数据库的查询
- Web服务器缓存 → 减少应用服务器请求
- 客户端浏览器缓存 → 减少对网站的访问

操作系统缓存概述

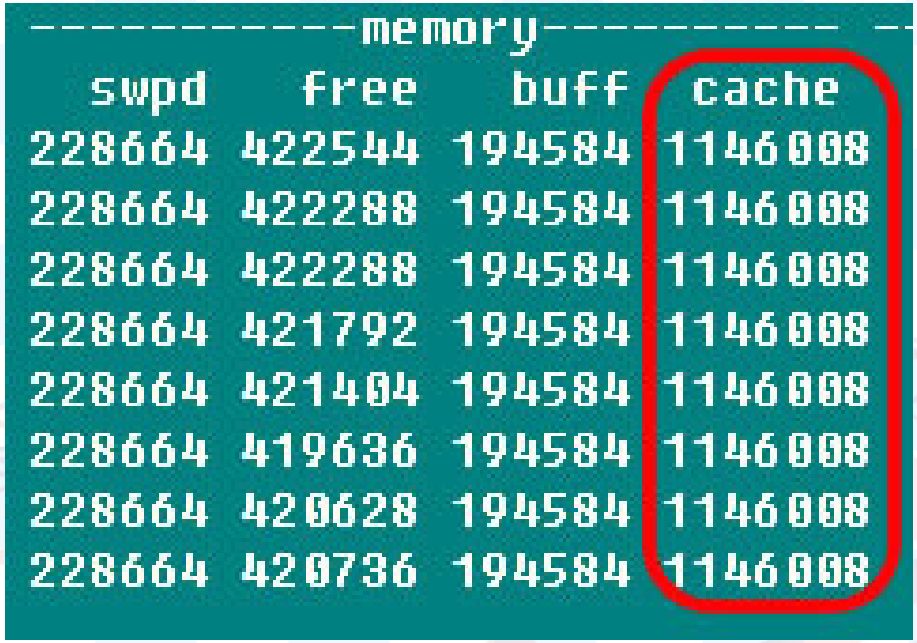
- 文件系统提供的Disk Cache：操作系统会把经常访问到的文件内容放入到内存当中，由文件系统来管理
- 当应用程序通过文件系统访问磁盘文件的时候，操作系统从Disk Cache当中读取文件内容，加速了文件读取速度
- Disk Cache由操作系统来自动管理，一般不用人工干预，但应当保证物理内存充足，以便于操作系统可以使用尽量多的内存充当Disk Cache，加速文件读取速度
- 特殊的应用程序对文件系统Disk Cache有很高的要求，会绕开文件系统Disk Cache，直接访问磁盘分区，自己实现Disk Cache策略
  - ✓ Oracle的raw device(裸设备) – 直接抛弃文件系统
  - ✓ MySQL的InnoDB： innodb_flush_method = O_DIRECT

Windows的Disk Cache



北京郵電大學

Linux的Disk Cache



数据库缓存的重要性

- 为什么数据库非常依赖缓存？
  - ✓ 数据库通常是企业应用系统最核心的部分
  - ✓ 数据库保存的数据量通常非常庞大
  - ✓ 数据库查询操作通常很频繁，有时还很复杂
  - ✓ 以上原因造成数据库查询会引起非常频繁的磁盘I/O读取操作，迫使CPU挂起等待，数据库性能极度低下
- 数据库有哪些缓存策略？
  - ✓ Query Cache
  - ✓ Data Buffer
  - ✓ Connection Pool

Query Cache

- 以SQL作为key值缓存查询结果集
- 一旦查询涉及的表记录被修改，缓存就会被自动删除
- 设置合适的Query Cache会极大提高数据库性能
- Query Cache并非越大越好，过大的Qquery Cache会浪费内存。
- MySQL: query_cache_size= 128M

MySQL Query Cache监控工具

- show status like 'Qcache%';
- mysqlreport脚本
- MySQL Administrator

Query Cache状态示例

__ Query Cache _____

Memory usage 25.77M of 64.00M **%Used: 40.26**

Block Fragmnt 24.73%

Hits 6.98M 3.2/s

Inserts 100.87M 46.9/s

Insrt:Prune 34.15:1 45.5/s

Hit:Insert 0.07:1

show status like 'Qcache%'

show status like 'Qcache%';

```
mysql> show status like 'Qcache%';
+-------------------------+-----------+
| Variable_name           | Value     |
+-------------------------+-----------+
| Qcache_free_blocks      | 1         |
| Qcache_free_memory      | 190827616 |
| Qcache_hits             | 0         |
| Qcache_inserts          | 4         |
| Qcache_lowmem_prunes    | 0         |
| Qcache_not_cached       | 32        |
| Qcache_queries_in_cache | 3         |
| Qcache_total_blocks     | 9         |
+-------------------------+-----------+
8 rows in set (0.00 sec)
```

Data Buffer

- data buffer是数据库数据在内存中的容器
- data buffer的命中率直接决定了数据库的性能
- data buffer越大越好，多多益善
- MySQL的InnoDB buffer：
  innodb_buffer_pool_size = 2G
- MySQL建议buffer pool开大到服务器物理内存60-80%

MySQL buffer 监控工具

- show status like 'innodb%';
- mysqlreport脚本
- innotop

MySQL buffer 监控工具

- show status like 'innodb%';

```
mysql> show status like 'innodb%';
+---------------------------------+---------+
| Variable_name                   | Value   |
+---------------------------------+---------+
| Innodb_buffer_pool_pages_data   | 73      |
| Innodb_buffer_pool_pages_dirty  | 0       |
| Innodb_buffer_pool_pages_flushed | 13      |
| Innodb_buffer_pool_pages_free   | 35446   |
| Innodb_buffer_pool_pages_misc   | 1       |
| Innodb_buffer_pool_pages_total  | 35520   |
| Innodb_buffer_pool_read_ahead_rnd | 1       |
| Innodb_buffer_pool_read_ahead_seq | 0       |
| Innodb_buffer_pool_read_requests | 1327    |
| Innodb_buffer_pool_reads        | 21      |
| Innodb_buffer_pool_wait_free    | 0       |
| Innodb_buffer_pool_write_requests | 36      |
| Innodb_data_fsyncs              | 15      |
| Innodb_data_pending_fsyncs      | 0       |
| Innodb_data_pending_reads       | 0       |
| Innodb_data_pending_writes      | 0       |
| Innodb_data_read                | 3362816 |
| Innodb_data_reads               | 82      |
| Innodb_data_writes              | 25      |
| Innodb_data_written             | 432640  |
| Innodb_dblwr_pages_written      | 13      |
| Innodb_dblwr_writes             | 3       |
| Innodb_log_waits                | 0       |
| Innodb_log_write_requests       | 5       |
| Innodb_log_writes               | 4       |
| Innodb_os_log_fsyncs            | 9       |
| Innodb_os_log_pending_fsyncs    | 0       |
| Innodb_os_log_pending_writes    | 0       |
| Innodb_os_log_written           | 4096    |
| Innodb_page_size                | 16384   |
| Innodb_pages_created            | 1       |
| Innodb_pages_read               | 72      |
| Innodb_pages_written            | 13      |
| Innodb_row_lock_current_waits   | 0       |
| Innodb_row_lock_time            | 0       |
| Innodb_row_lock_time_avg        | 0       |
| Innodb_row_lock_time_max        | 0       |
```

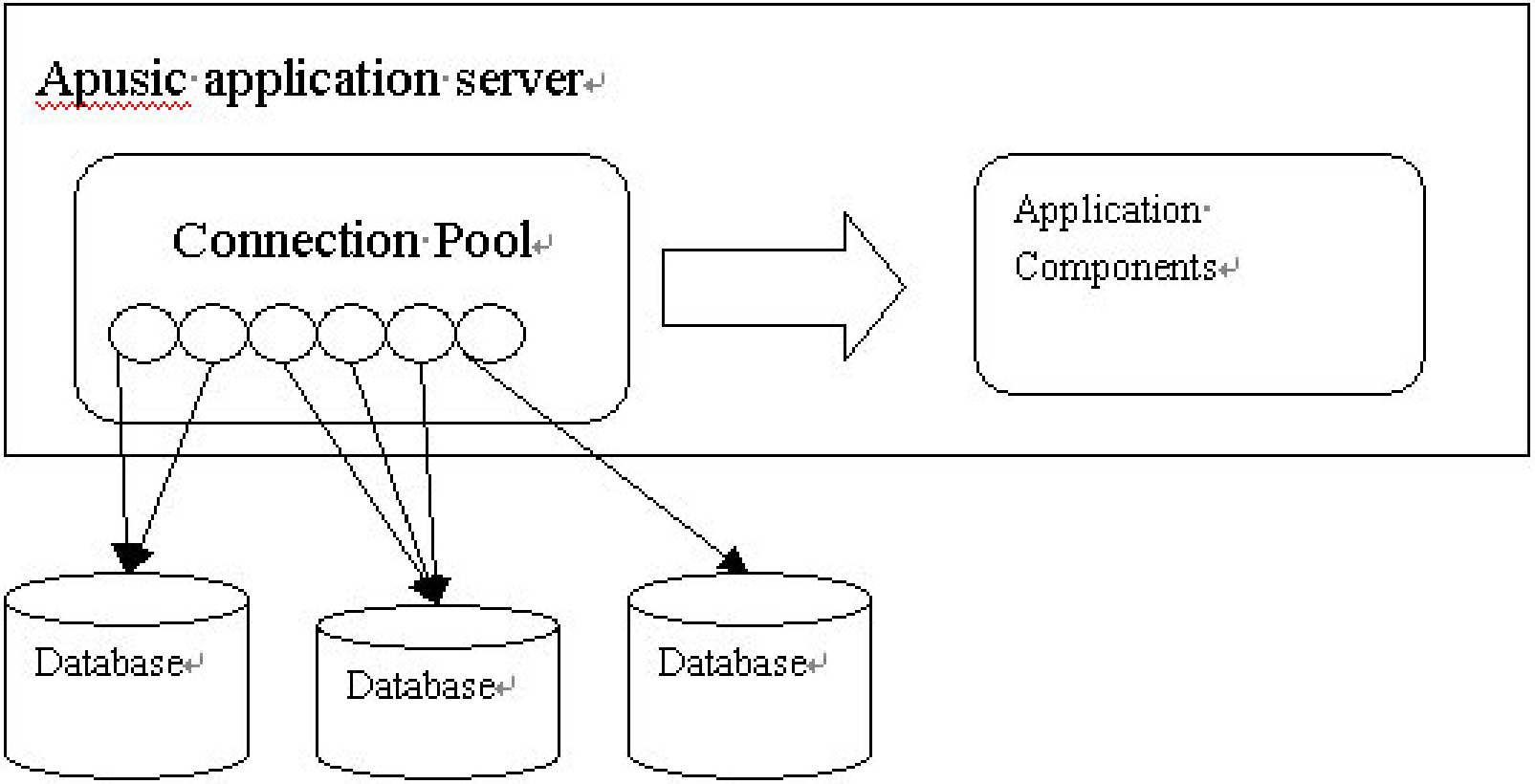
数据库连接池

- 创建连接需要耗费时间
  - ✓创建一个连接大概需要1-2秒的时间。
- 在某一时刻连接必须服务于一个用户，以免造成事务冲突
  - ✓来自不同用户的请求（都使用了同一个连接）对相同的事务进行操作，如果一个请求试图回滚，那么所有使用相同连接的数据库操作都要被回滚。
- 保持连接打开状态的代价很大
  - ✓尤其是在系统资源（例如内存）方面。
  - ✓数据库产品的许可证都按照同时打开的连接数目来收费。

数据库连接池

- 连接池中保存了一些Connection对象，这些对象被所有Servlet和JSP页面所共享。对于每个请求都会分配给它一个连接，使用完后再收回这个连接。
- 创建连接需要时间
  - ✓放入池中的连接只被创建一次，以后一直重用这个连接
- 共享连接会造成多线程问题
  - ✓每个请求将得到它自己的Connection对象，所以在某一时刻它只被一个线程使用，从而避免了潜在的多线程问题
- 连接的资源有限,每个连接都会得到有效的使用

数据库连接池


Apusic application server
Connection Pool
Application Components
Database
Database
Database

应用程序缓存概述

- 对象缓存
- 查询缓存
- 页面缓存
  - ✓ 动态页面静态化
  - ✓ Servlet缓存
  - ✓ 页面内部缓存

对象缓存

- 由O/R Mapping框架例如Hibernate提供，透明性访问，细颗粒度缓存数据库查询结果，无需业务代码显式编程，是最省事的缓存策略
- 当软件结构按照O/R Mapping框架的要求进行针对性设计，使用对象缓存将会极大降低Web系统对于数据库的访问请求
- 良好的设计数据库结构和利用对象缓存，能够提供极高的性能，对象缓存适合OLTP（联机事务处理）应用

对象缓存分类

- 对映射数据库表记录的entity对象进行缓存
- 对1对n关系的集合进行缓存
- 对n对1关系的关联对象进行缓存

Hibernate对象缓存配置

配置entity对象缓存

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

配置关联集合的缓存

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() { return tickets; }
```

仅仅添加Annotation就可以了，无须编码，即可自动享受对象缓存。Hibernate会拦截对象的CRUD操作，针对对象读取操作进行缓存，针对对象修改操作自动清理缓存

Hibernate二级缓存是提升web应用性能的法宝

- OLTP类型的web应用，由于应用服务器端可以进行群集水平扩展，最终的系统瓶颈总是逃不开数据库访问；
- 哪个框架能够最大限度减少数据库访问，降低数据库访问压力，哪个框架提供的性能就更高；
- 针对数据库的缓存策略：
  - ✓ 对象缓存：细颗粒度，针对表的记录级别，透明化访问，在不改变程序代码的情况下可以极大提升web应用的性能。对象缓存是ORM的制胜法宝。
  - ✓ 对象缓存的优劣取决于框架实现的水平，Hibernate是目前已知对象缓存最强大的开源ORM
  - ✓ 查询缓存：粗颗粒度，针对查询结果集，应用于数据实时化要求不高的场合

查询缓存

- 对数据库查询结果集进行缓存，类似数据库的Query Cache
- 适用于一些耗时，但是时效性要求比较低的场景。查询缓存和对象缓存适用的场景不一样，是互为补充的
- 当查询结果集涉及的表记录被修改以后，需要注意清理缓存

Hibernate查询缓存

- 在配置文件中打开Query Cache
  - ✓hibernate.cache.use_query_cache true
- 在查询的时候显式编码使用Cache

```
List blogs = sess.createQuery("from Blog blog where blog.blogger =
    :blogger") .setEntity("blogger",blogger) .
setMaxResults(15) .
setCacheable(true) .
setCacheRegion("frontpages") .
list();
```

Hibernate查询缓存特征

- 并非缓存整个查询结果集，而是缓存查询结果集entity对象的id集合
  - ✓ [blogId1, blogId2, blogId3,...]
  - ✓ 在遍历结果集的时候，再按照blogId去查询blog对象，例如select blog.* from blog where id=?
  - ✓ 如果此时blog配置了对象缓存，则自动读取对象缓存
- Hibernate查询缓存会自动清理过期缓存
  - ✓ 一旦结果集涉及的entity被修改，查询缓存就被自动清理

页面缓存

- 页面缓存的作用是什么？
  - ✓ 针对页面的缓存技术不但可以减轻数据库服务器压力，还可以减轻应用服务器压力
  - ✓ 好的页面缓存可以极大提高页面渲染速度
  - ✓ 页面缓存的难点在于如何清理过期的缓存
- 页面缓存技术有哪些？
  - ✓ 动态页面静态化
  - ✓ Servlet缓存
  - ✓ 页面局部缓存

动态页面静态化

- 利用模板技术将访问过一次的动态页面生成静态html，同时修改页面链接，下一次请求直接访问静态链接页面
- 动态页面静态化技术的广泛应用于互联网CMS/新闻类Web应用，但也有BBS应用使用该技术，例如Discuz!
- 无法进行权限验证，无法显示个性化信息
- 可以使用AJAX请求弥补动态页面静态化的某些缺点

Servlet缓存

- 针对URL访问返回的页面结果进行缓存，适用于粗粒度的页面缓存，例如新闻发布
- 可以进行权限的检查
- OScache提供了简单的Servlet缓存(通过web.xml中的配置)
- 也可以自己编程实现Servlet缓存

OSCache Servlet缓存示例

```
<filter>
    <filter-name>CacheFilter</filter-name>
    <filter-class>com.opensymphony.oscache.web.filter.CacheFilter</filte
    <init-param>
        <param-name>time</param-name>
        <param-value>600</param-value>
    </init-param>
    <init-param>
        <param-name>scope</param-name>
        <param-value>session</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>CacheFilter</filter-name>
    <url-pattern>/news/*</url-pattern>
</filter-mapping>
```

页面局部缓存

- 针对动态页面的局部片断内容进行缓存，适用于一些个性化但不经常更新的页面(例如博客)
- OSCache提供了简单的页面缓存
- 可以自行扩展JSP Tag实现页面局部缓存

OSCache的页面局部缓存

```
<%@ taglib uri="http://www.opensymphony.com/oscache" prefix="cache" %>
<cache:cache>
    ... some jsp content ...
</cache:cache>

<cache:cache key="foobar" scope="session">
    ... some jsp content ...
</cache:cache>

<cache:cache key="<%= product.getId() %>" time="1800" refresh="<%= needRefresh
%>">
    ... some jsp content ...
</cache:cache>

<cache:cache key="<%= product.getId() %>" cron="0 2 * * *" refresh="<%=
needRefresh %>">
    ... some jsp content ...
</cache:cache>
```

应用缓存的缓存服务器

- EHCache
  - ✓ 适合充当对象缓存和Hibernate集成效果很好，Gavin King也是EHCache作者之一
- OSCache
  - ✓ 充当Servlet和页面缓存
  - ✓ 在一个Web应用当中可以同时使用OSCache和EHCache
- JBossCache
  - ✓ 在Java群集环境下使用
  - ✓ 支持缓存在节点之间的复制，在JBoss AS上被用来实现HTTP Session的内存复制功能

非Java实现的通用缓存产品

- **Memcached**
  - ✓ 在大规模互联网应用下使用
  - ✓ 每秒支撑1.5万～2万次请求
- **Tokyo Tyrant**
  - ✓ 兼容memcached协议，可以持久化存储
  - ✓ 支持故障切换，对缓存服务器有高可靠性要求可以使用
  - ✓ 每秒支撑0.5万～0.8万次请求

Web服务器端缓存技术

- 基于代理服务器模式的Web服务器端缓存
  - ✓ squid/nginx
- Web服务器缓存技术被用来实现CDN（内容分发网络 content delivery network）
- 被国内主流门户网站大量采用
- 不需要编程，但仅限于新闻发布类网站，页面实时性要求不高

基于AJAX技术的浏览器缓存

- 使用AJAX调用的时候，将数据库在浏览器端缓存
- 只要不离开当前页面，不刷新当前页面，就可以直接读取缓存数据
- 只适用于使用AJAX技术的页面

未使用AJAX缓存代码

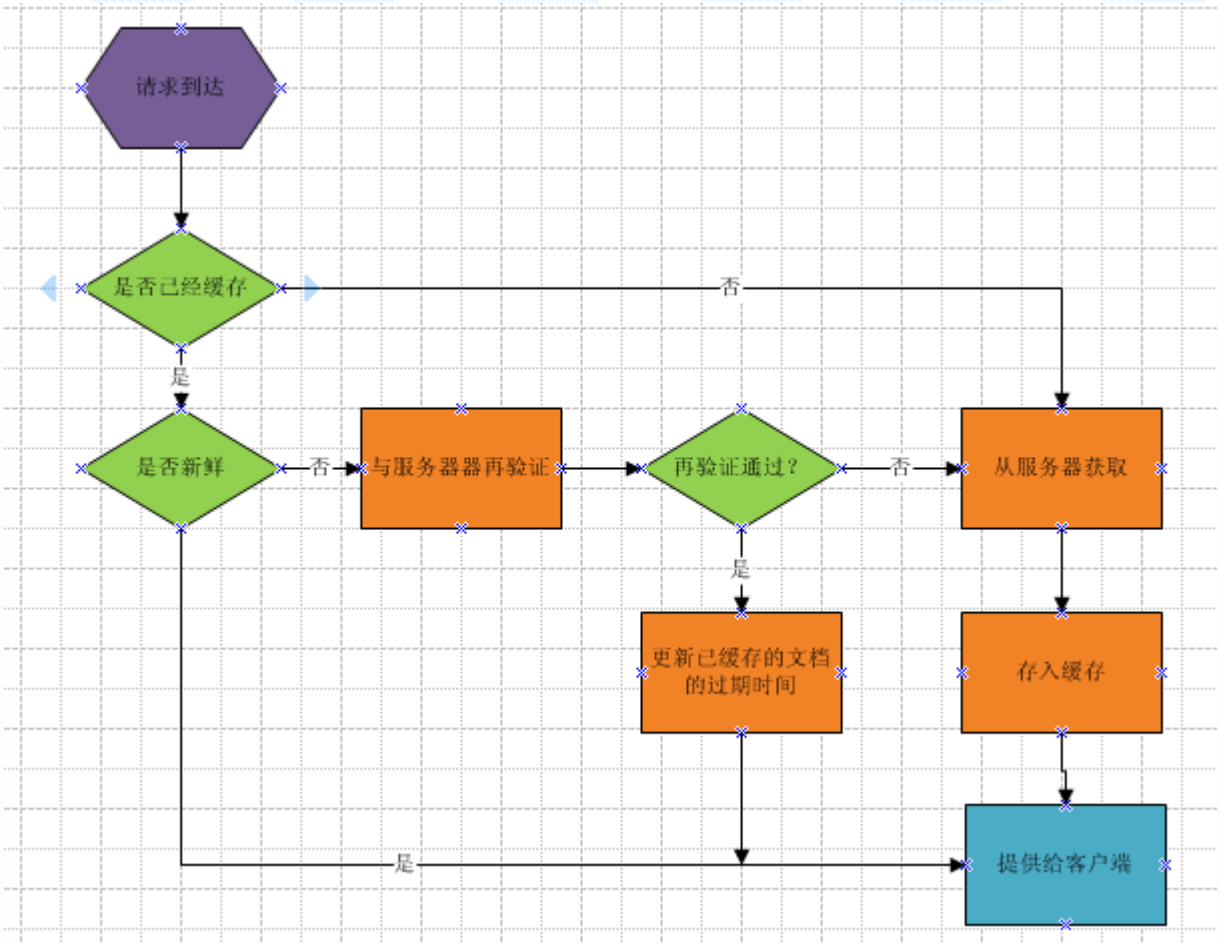
```
1. <script src="jQuery.js"></script>
 2.   <script language="JavaScript">
 3.   <!--
 4.    function gopage(option,obj){
 5.    var page = $("#page").html() * 1;
 6.    obj.disabled = true;
 7.    if(option == '+'){
 8.       page ++ ;
 9.       var url = "data.php?page="+page + "&r="+Math.random();
10.     //获取ajax url，后面的random是为了防止浏览器缓存xml内容
11.    }else{
12.       page --;
13.       if(page < 1)page =1;
14.       var url = "data.php?page="+page + "&r="+Math.random();
15.
16.    }
17.    $("#data").html("loading...");//设置loading状态
18.    $("#page").html(page);
19. //下面是ajax处理
20.    $("#data").load(url,
21.       {limit: 25},
22.            function() { obj.disabled = false; }
23.       );
24.
25.    }
26.   //-->
27.   </script>
28. <input value="<" onclick="gopage('-',this)" type="button">
29. <input value=">" onclick="gopage('+',this)" type="button">
30.   page:<span id="page">1</span>
31. <div style="border: 2px solid red; padding: 2px;" id="data">data area</div>
```

北京郵電大學

使用AJAX缓存代码

```
# <script language="JavaScript">
#   <!--
# var cache_data = new Array();//定义全局变量用来保存缓存数据
#     function gopageCache(option,obj){
#     var page = $("#page2").html() * 1;
#     if(option == '+'){
#         page ++ ;
#         var url = "data.php?page="+page + "&r="+Math.random();
#     }else{
#         page --;
#         if(page < 1)page =1;
#         var url = "data.php?page="+page + "&r="+Math.random();
#
#     }
# /*下面是缓存增加的部分*/
#     if( (cache_data[page] !=null) && (cache_data[page].length > 1) ) {
# //如果缓存存在,则直接调用缓存数据,不用再去服务器进行数据请求
#         alert('cache hit');
#         $("#data2").html(cache_data[page]);
#         $("#page2").html(page);
#         return true;
#     }
#     $("#page2").html(page);
#     obj.disabled = true;
#     $("#data2").html("loading(cache enabled)...");
#     $("#data2").load(url,
#         {limit: 25},
#             function(responseText) { obj.disabled = false;
#                         cache_data[page] = responseText;//将当前的数据存入到内存(缓存变量)中 }
#         );
#
#     }
#
#   //-->
#   </script>
# <input value="<" onclick="gopageCache('-',this)" type="button">
# <input value=">" onclick="gopageCache('+',this)" type="button">
#   page:<span id="page2">1</span>
# <div style="border: 2px solid red; padding: 2px;" id="data2">data area</div>
```

基于HTTP协议的资源缓存



第四部分：前端优化技术

主要内容

- 前端优化的重要性
- 前端优化的主要方法
- 推荐前端优化工具

注：主要是来自于《高性能网站建设指南》一书部分内容

一、前端优化的重要性

- 优化分类
- 用户响应时间分布(二八原则)

一、前端优化的重要性

- 优化分类
  - 前端优化：通常只要较少的时间和资源，例如修改Web服务器配置文件、将脚本和样式表放在特定位置、合并图片、合并脚本等；
  - 后端优化：通常代价比较大，例如重新设计应用程序架构和代码、查找和优化临界代码路径、添加或改动硬件、对数据库进行分布化等。

一、前端优化的重要性

- 用户响应时间分布
  - 10%～20%——从服务器端获取HTML文档上（后端优化内容）
  - 80%～90%——下载页面中的所有组件上（前端优化内容）
- 根据二八原则，我们应该更关注前端优化。

注：这数据不是猜测的，可以查阅十大网站的具体数据。

二、前端优化的主要方法

- 减少http请求次数
- 充分利用并行下载
- 减小元素的大小
- 其他方法

二、前端优化的主要方法——减少http请求次数

- 优化图片，有三个具体方案：图片地图、CSS Sprites、内联图片三种，最值得关注的是CSS Sprites。
- 合并脚本和样式表，虽然合并有悖于模块化开发的原则，但非常有利于性能。

二、前端优化的主要方法——减少http请求次数

- **关注Expires头的设置。**缓存可以使一些http请求转为调用客户端的已有资源。另外，HTTP1.1引入了“Cache-Control”头，可以用“max-age”来设置缓存的时间长度。

二、前端优化的主要方法——减少http请求次数

- 移除ETag或者对ETag进行专门配置，以免影响缓存调取
  - 因为默认配置的ETag，和原始服务器的属性相关，当多台服务器时，会导致缓存调取失败。（N台服务器，缓存调取成功的概率是1/N）

二、前端优化的主要方法——减少http请求次数

- 加载后下载（Post-Onload Download）
  脚本和样式表内联，但在页面加载完成后通过onload事件，动态下载外部脚本和样式表。

二、前端优化的主要方法

- 减少http请求次数
- 充分利用并行下载
- 减小元素的大小
- 其他方法

二、前端优化的主要方法——充分利用并行下载

- 使用两个主机名。HTTP1.1建议浏览器从每个主机名并行下载两个元素，使用多个主机名能进行更多的并行下载，但由于带宽和CPU速度，过多的并行下载也会降低性能。Yahoo！的研究表明，使用两个主机名性能最优。

二、前端优化的主要方法——充分利用并行下载

- 将脚本放在底部，以避免对并行下载的影响。因为下载脚本时，并行下载是禁止的。

三、前端优化的主要方法

- 减少http请求次数
- 充分利用并行下载
- 减小元素的大小
- 其他方法

三、前端优化的主要方法——减小元素的大小

- **精简脚本。** 用JSMin精简外置脚本，这能移除所有的注释以及不必要的空白字符，使脚本减小20%。（内联的脚本也应该尽量精简）
- **优化CSS。** 合并相同的类、移除不使用的类。同时，应避免使用CSS表达式，改用一次性表达式和使用事件处理器。

三、前端优化的主要方法——减小元素的尺寸

- 务必使用Gzip对脚本、样式表、html文档进行压缩，这通常能减小60%的数据量。

三、前端优化的主要方法

- 减少http请求次数
- 充分利用并行下载
- 减小元素的大小
- 其他方法

三、前端优化的主要方法——其他方法

- 避免没必要的重定向
  - 在重定向完毕并且html文档下载完毕之前，没有任何东西展示给用户；
  - 对于为了跟踪流量而使用的重定向，建议改用“referer”或“beacon + XMLHttpRequest”；

三、前端优化的主要方法——其他

- 使用CDN（Content Delivery Network），缩小内容和用户的距离
- 将样式表放在顶部，使内容逐步呈现。
  - 尽管整个页面的加载总耗时可能并无变化，但逐步呈现内容，能使用户感觉更快。
- 适当减少主机名，以减少DNS查找
- Ajax下的优化问题

四、前端优化工具推荐

- HTTP请求图表使用IBM Page Detailer
- 响应时间的测量使用Gomez的Web监视服务
- 页面中的脚本及CSS分析使用Firebug
- 分析页面性能使用Yslow
- 抓包工具HTTP Analyzer