图 6.2　通过纵向分离部署实现系统伸缩性

横向分离（业务分割后分离）：将不同的业务模块分离部署，实现系统伸缩性，如图 6.3 所示。

图 6.3　通过横向分离部署实现系统伸缩性

横向分离的粒度可以非常小，甚至可以一个关键网页部署一个独立服务，比如对于电商网站非常重要的产品详情页面，商铺页面，搜索列表页面，每个页面都可以独立部署，专门维护。

6.1.2　单一功能通过集群规模实现伸缩

将不同功能分离部署可以实现一定程度的伸缩性，但是随着网站访问量的逐步增加，即使分离到最小粒度的独立部署，单一的服务器也不能满足业务规模的要求。因此必须使用服务器集群，即将相同服务部署在多台服务器上构成一个集群整体对外提供服务。

当一头牛拉不动车的时候，不要去寻找一头更强壮的牛，而是用两头牛来拉车。

以搜索服务器为例，如果一台服务器可以提供每秒 1000 次的请求服务，即 QPS（Query Per Second）为 1000。那么如果网站高峰时每秒搜索访问量为 10000，就需要部署 10 台服务器构成一个集群。若以缓存服务器为例，如果每台服务器可缓存 40GB 数据，那么要缓存 100GB 数据，就需要部署 3 台服务器构成一个集群。当然这些例子的计算都是简化的，事实上，计算一个服务的集群规模，需要同时考虑其对可用性、性能的影响及关联服务集群的影响。

具体来说，集群伸缩性又可分为应用服务器集群伸缩性和数据服务器集群伸缩性。这两种集群由于对数据状态管理的不同，技术实现也有非常大的区别。而数据服务器集群也可分为缓存数据服务器集群和存储数据服务器集群，这两种集群的伸缩性设计也不大相同。

6.2 应用服务器集群的伸缩性设计

我们在本书第 5 章提到，应用服务器应该设计成无状态的，即应用服务器不存储请求上下文信息，如果将部署有相同应用的服务器组成一个集群，每次用户请求都可以发送到集群中任意一台服务器上去处理，任何一台服务器的处理结果都是相同的。这样只要能将用户请求按照某种规则分发到集群的不同服务器上，就可以构成一个应用服务器集群，每个用户的每个请求都可能落在不同的服务器上。如图 6.4 所示。

如果 HTTP 请求分发装置可以感知或者可以配置集群的服务器数量，可以及时发现集群中新上线或下线的服务器，并能向新上线的服务器分发请求，停止向已下线的服务器分发请求，那么就实现了应用服务器集群的伸缩性。

这里，这个 HTTP 请求分发装置被称作负载均衡服务器。

负载均衡是网站必不可少的基础技术手段，不但可以实现网站的伸缩性，同时还改善网站的可用性，可谓网站的杀手锏之一。具体的技术实现也多种多样，从硬件实现到软件实现，从商业产品到开源软件，应有尽有，但是实现负载均衡的基础技术不外以下几种。

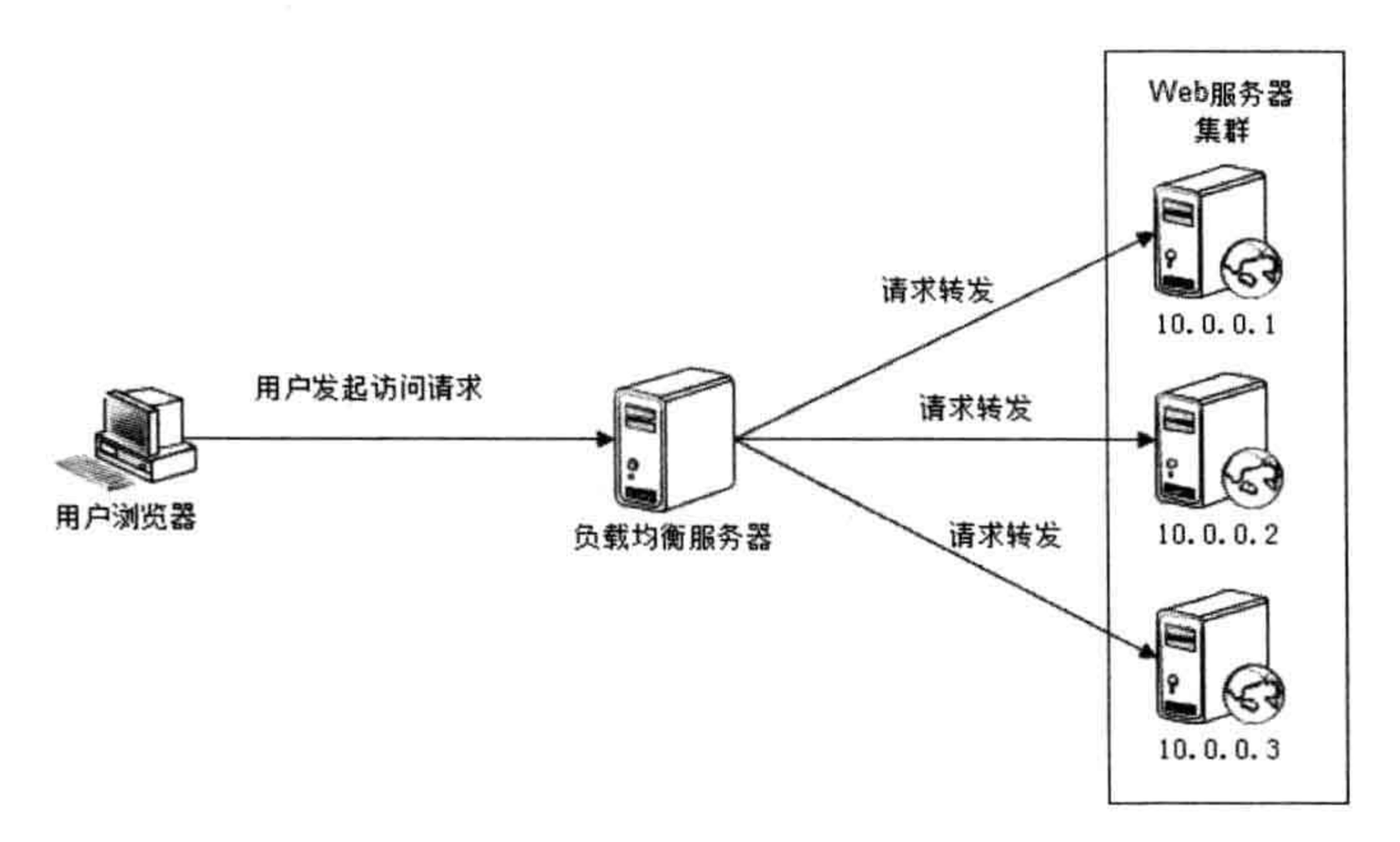

图 6.4 负载均衡实现应用服务器伸缩性

6.2.1 HTTP 重定向负载均衡

利用 HTTP 重定向协议实现负载均衡。如图 6.5 所示。

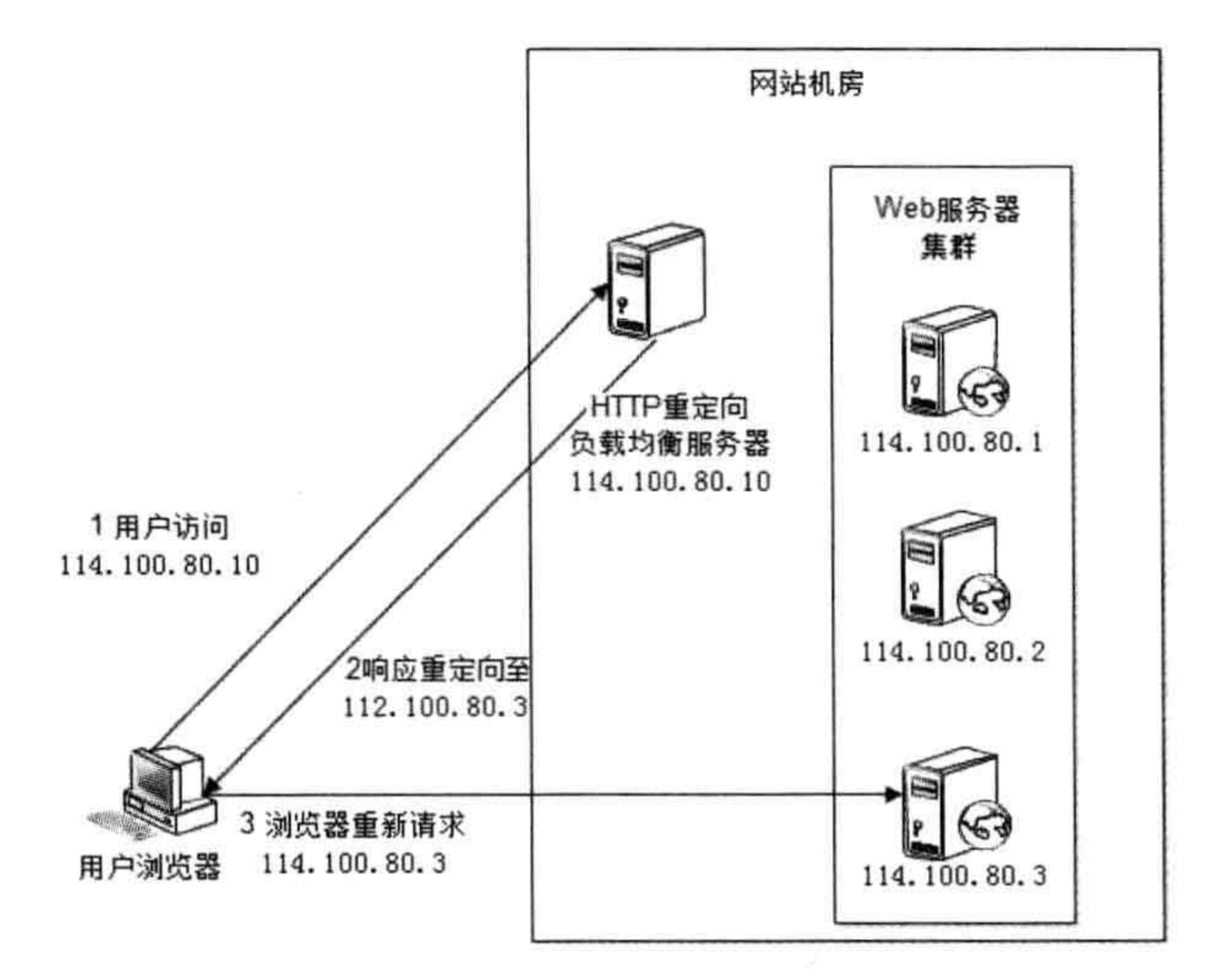

图 6.5 HTTP 重定向负载均衡原理

HTTP 重定向服务器是一台普通的应用服务器，其唯一的功能就是根据用户的 HTTP 请求计算一台真实的 Web 服务器地址，并将该 Web 服务器地址写入 HTTP 重定向响应中（响应状态码 302）返回给用户浏览器。在图 6.5 中，浏览器请求访问域名 www.mysite.com，DNS 服务器解析得到 IP 地址是 114.100.80.10，即 HTTP 重定向服务器的 IP 地址。然后浏览器通过 IP 地址 114.100.80.10 访问 HTTP 重定向负载均衡服务器

后，服务器根据某种负载均衡算法计算获得一台实际物理服务器的地址（114.100.80.3），构造一个包含该实际物理服务器地址的重定向响应返回给浏览器，浏览器自动重新请求实际物理服务器的 IP 地址 114.100.80.3，完成访问。

这种负载均衡方案的优点是比较简单。缺点是浏览器需要两次请求服务器才能完成一次访问，性能较差；重定向服务器自身的处理能力有可能成为瓶颈，整个集群的伸缩性规模有限；使用 HTTP302 响应码重定向，有可能使搜索引擎判断为 SEO 作弊，降低搜索排名。因此实践中使用这种方案进行负载均衡的案例并不多见。

6.2.2 DNS 域名解析负载均衡

这是利用 DNS 处理域名解析请求的同时进行负载均衡处理的一种方案，如图 6.6 所示。

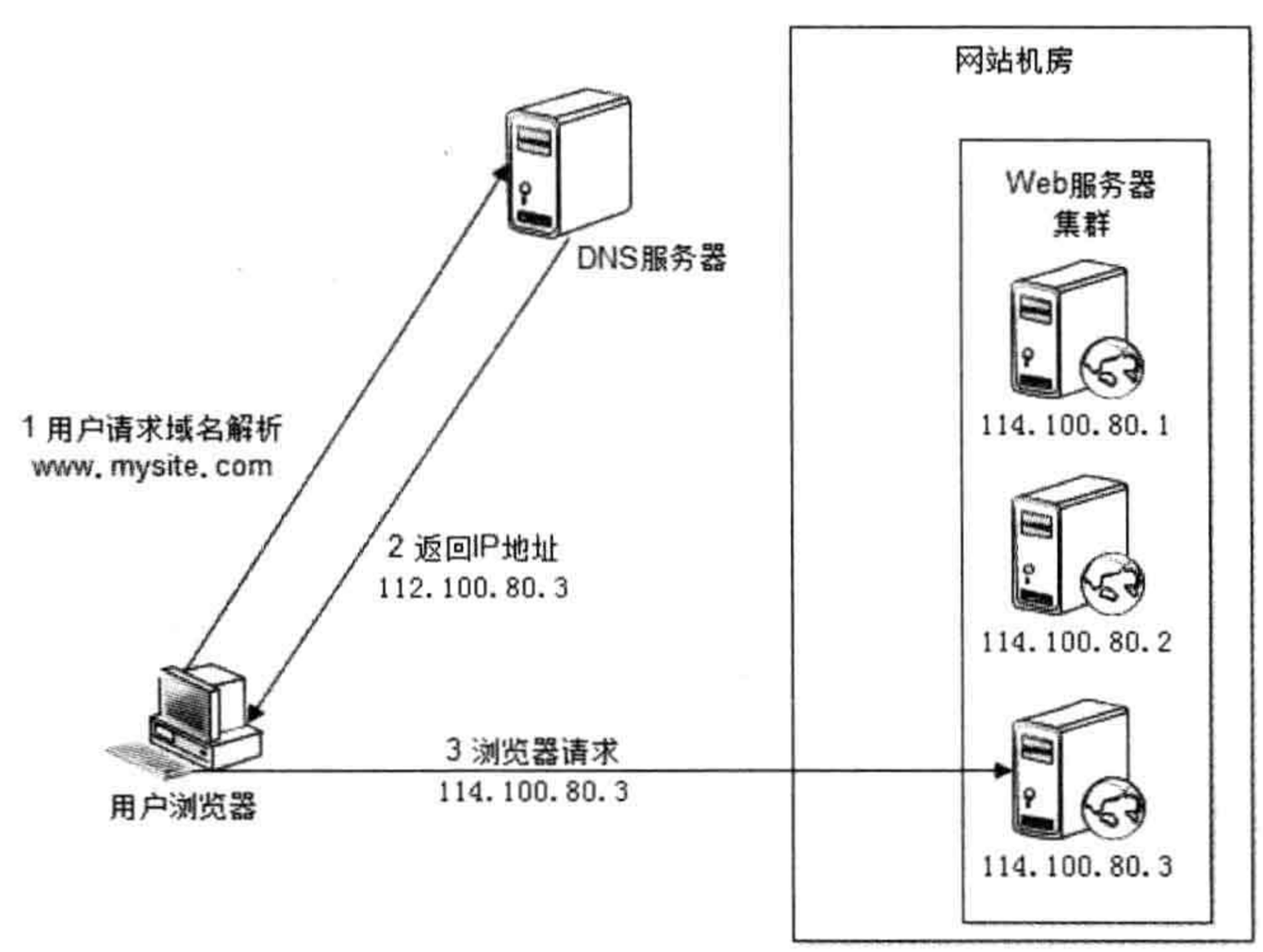

图 6.6 DNS 域名解析负载均衡原理

在 DNS 服务器中配置多个 A 记录，如：www.mysite.com IN A 114.100.80.1、www.mysite.com IN A 114.100.80.2、www.mysite.com IN A 114.100.80.3。

每次域名解析请求都会根据负载均衡算法计算一个不同的 IP 地址返回，这样 A 记录中配置的多个服务器就构成一个集群，并可以实现负载均衡。图 6.6 中的浏览器请求解析域名 www.mysite.com，DNS 根据 A 记录和负载均衡算法计算得到一个 IP 地址 114.100.80.3，并返回给浏览器；浏览器根据该 IP 地址，访问真实物理服务器 114.100.80.3。

DNS 域名解析负载均衡的优点是将负载均衡的工作转交给 DNS，省掉了网站管理维护负载均衡服务器的麻烦，同时许多 DNS 还支持基于地理位置的域名解析，即会将域名解析成距离用户地理最近的一个服务器地址，这样可加快用户访问速度，改善性能。但是 DNS 域名解析负载均衡也有缺点，就是目前的 DNS 是多级解析，每一级 DNS 都可能缓存 A 记录，当下线某台服务器后，即使修改了 DNS 的 A 记录，要使其生效也需要较长时间，这段时间，DNS 依然会将域名解析到已经下线的服务器，导致用户访问失败；而且 DNS 负载均衡的控制权在域名服务商那里，网站无法对其做更多改善和更强大的管理。

事实上，大型网站总是部分使用 DNS 域名解析，利用域名解析作为第一级负载均衡手段，即域名解析得到的一组服务器并不是实际提供 Web 服务的物理服务器，而是同样提供负载均衡服务的内部服务器，这组内部负载均衡服务器再进行负载均衡，将请求分发到真实的 Web 服务器上。

6.2.3 反向代理负载均衡

利用反向代理服务器进行负载均衡，如图 6.7 所示。

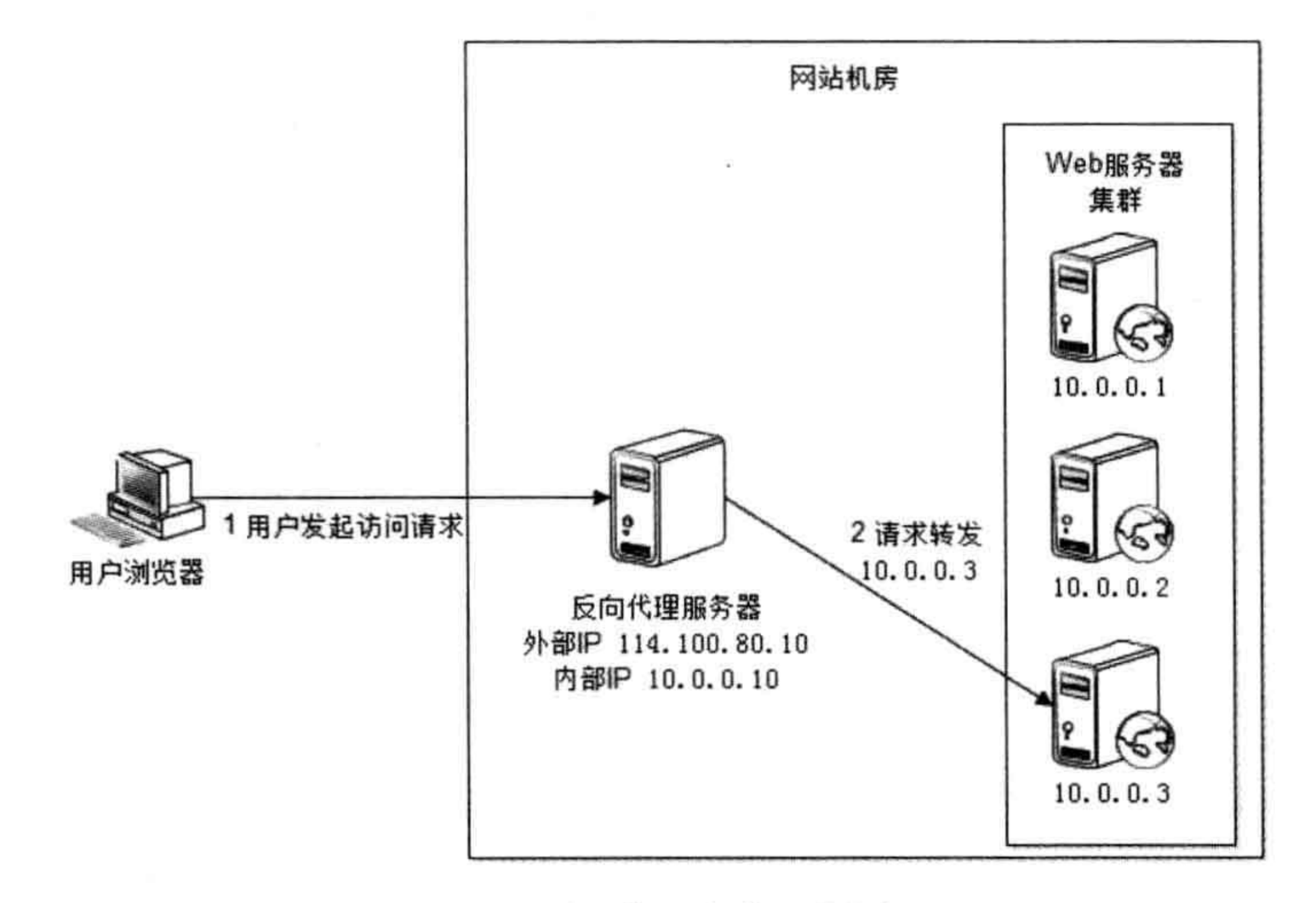

图 6.7 反向代理负载均衡原理

前面我们提到利用反向代理缓存资源，以改善网站性能。实际上，在部署位置上，反向代理服务器处于 Web 服务器前面（这样才可能缓存 Web 响应，加速访问），这个位置也正好是负载均衡服务器的位置，所以大多数反向代理服务器同时提供负载均衡的功

能，管理一组 Web 服务器，将请求根据负载均衡算法转发到不同 Web 服务器上。Web 服务器处理完成的响应也需要通过反向代理服务器返回给用户。由于 Web 服务器不直接对外提供访问，因此 Web 服务器不需要使用外部 IP 地址，而反向代理服务器则需要配置双网卡和内部外部两套 IP 地址。

图 6.7 中，浏览器访问请求的地址是反向代理服务器的地址 114.100.80.10，反向代理服务器收到请求后，根据负载均衡算法计算得到一台真实物理服务器的地址 10.0.0.3，并将请求转发给服务器。10.0.0.3 处理完请求后将响应返回给反向代理服务器，反向代理服务器再将该响应返回给用户。

由于反向代理服务器转发请求在 HTTP 协议层面，因此也叫应用层负载均衡。其优点是和反向代理服务器功能集成在一起，部署简单。缺点是反向代理服务器是所有请求和响应的中转站，其性能可能会成为瓶颈。

6.2.4 IP 负载均衡

在网络层通过修改请求目标地址进行负载均衡，如图 6.8 所示。

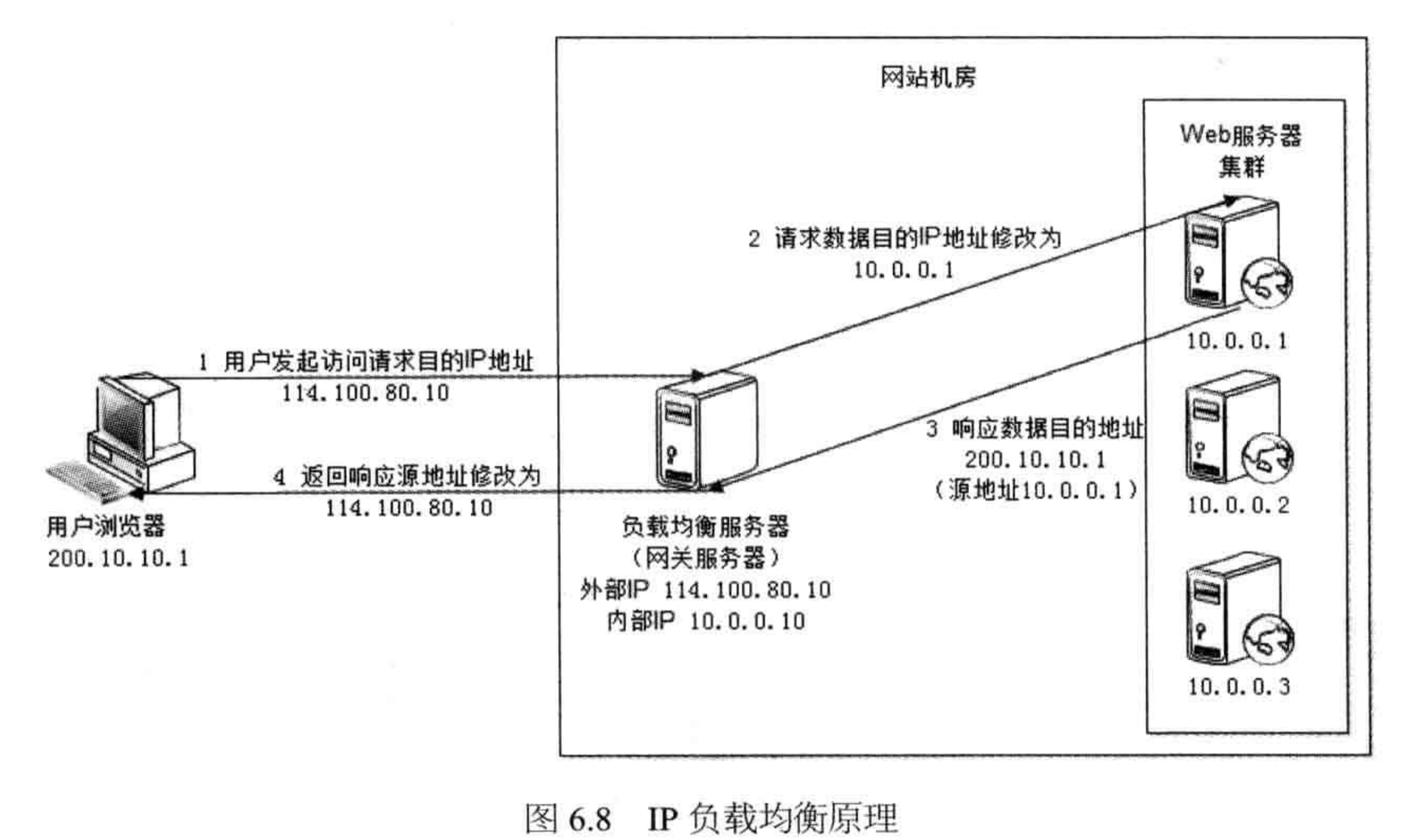

图 6.8 IP 负载均衡原理

用户请求数据包到达负载均衡服务器 114.100.80.10 后，负载均衡服务器在操作系统内核进程获取网络数据包，根据负载均衡算法计算得到一台真实 Web 服务器 10.0.0.1，然后将数据目的 IP 地址修改为 10.0.0.1，不需要通过用户进程处理。真实 Web 应用服务器

处理完成后，响应数据包回到负载均衡服务器，负载均衡服务器再将数据包源地址修改为自身的 IP 地址（114.100.80.10）发送给用户浏览器。

这里的关键在于真实物理 Web 服务器响应数据包如何返回给负载均衡服务器。一种方案是负载均衡服务器在修改目的 IP 地址的同时修改源地址，将数据包源地址设为自身 IP，即源地址转换（SNAT），这样 Web 服务器的响应会再回到负载均衡服务器；另一种方案是将负载均衡服务器同时作为真实物理服务器集群的网关服务器，这样所有响应数据都会到达负载均衡服务器。

IP 负载均衡在内核进程完成数据分发，较反向代理负载均衡（在应用程序中分发数据）有更好的处理性能。但是由于所有请求响应都需要经过负载均衡服务器，集群的最大响应数据吞吐量不得不受制于负载均衡服务器网卡带宽。对于提供下载服务或者视频服务等需要传输大量数据的网站而言，难以满足需求。能不能让负载均衡服务器只分发请求，而使响应数据从真实物理服务器直接返回给用户呢？

6.2.5 数据链路层负载均衡

顾名思义，数据链路层负载均衡是指在通信协议的数据链路层修改 mac 地址进行负载均衡，如图 6.9 所示。

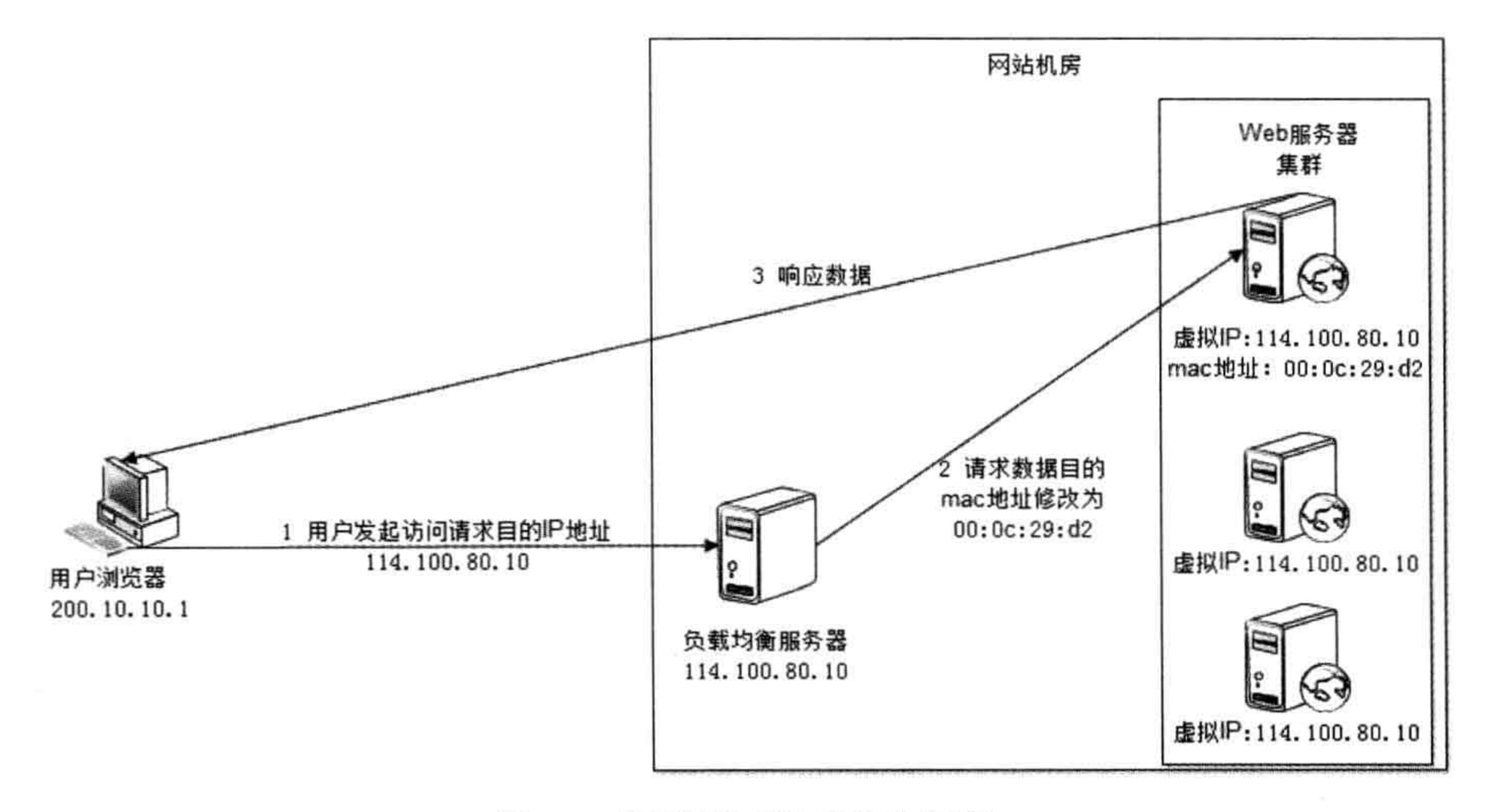

图 6.9 数据链路层负载均衡原理

这种数据传输方式又称作三角传输模式，负载均衡数据分发过程中不修改 IP 地址，只修改目的 mac 地址，通过配置真实物理服务器集群所有机器虚拟 IP 和负载均衡服务器 IP 地址一致，从而达到不修改数据包的源地址和目的地址就可以进行数据分发的目的，由于实际处理请求的真实物理服务器 IP 和数据请求目的 IP 一致，不需要通过负载均衡服务器进行地址转换，可将响应数据包直接返回给用户浏览器，避免负载均衡服务器网卡带宽成为瓶颈。这种负载均衡方式又称作直接路由方式（DR）。

在图 6.9 中，用户请求到达负载均衡服务器 114.100.80.10 后，负载均衡服务器将请求数据的目的 mac 地址修改为 00:0c:29:d2，并不修改数目包目标 IP 地址，由于 Web 服务器集群所有服务器的虚拟 IP 地址都和负载均服务器的 IP 地址相同，因此数据可以正常传输到达 mac 地址 00:0c:29:d2 对应的服务器，该服务器处理完成后发送响应数据到网站的网关服务器，网关服务器直接将该数据包发送到用户浏览器（通过互联网），响应数据不需要通过负载均衡服务器。

使用三角传输模式的链路层负载均衡是目前大型网站使用最广的一种负载均衡手段。在 Linux 平台上最好的链路层负载均衡开源产品是 LVS（Linux Virtual Server）。

6.2.6 负载均衡算法

负载均衡服务器的实现可以分成两个部分：

1. 根据负载均衡算法和 Web 服务器列表计算得到集群中一台 Web 服务器的地址。

2. 将请求数据发送到该地址对应的 Web 服务器上。

前面描述了如何将请求数据发送到 Web 服务器，而具体的负载均衡算法通常有以下几种。

轮询（Round Robin，RR）

所有请求被依次分发到每台应用服务器上，即每台服务器需要处理的请求数目都相同，适合于所有服务器硬件都相同的场景。

加权轮询（Weighted Round Robin，WRR）

根据应用服务器硬件性能的情况，在轮询的基础上，按照配置的权重将请求分发到

每个服务器，高性能的服务器能分配更多请求。

随机（Random）

请求被随机分配到各个应用服务器，在许多场合下，这种方案都很简单实用，因为好的随机数本身就很均衡。即使应用服务器硬件配置不同，也可以使用加权随机算法。

最少连接（Least Connections）

记录每个应用服务器正在处理的连接数（请求数），将新到的请求分发到最少连接的服务器上，应该说，这是最符合负载均衡定义的算法。同样，最少连接算法也可以实现加权最少连接。

源地址散列（Source Hashing）

根据请求来源的 IP 地址进行 Hash 计算，得到应用服务器，这样来自同一个 IP 地址的请求总在同一个服务器上处理，该请求的上下文信息可以存储在这台服务器上，在一个会话周期内重复使用，从而实现会话黏滞。

6.3 分布式缓存集群的伸缩性设计

我们在本书第 4 章讨论过分布式缓存，不同于应用服务器集群的伸缩性设计，分布式缓存集群的伸缩性不能使用简单的负载均衡手段来实现。

和所有服务器都部署相同应用的应用服务器集群不同，分布式缓存服务器集群中不同服务器中缓存的数据各不相同，缓存访问请求不可以在缓存服务器集群中的任意一台处理，必须先找到缓存有需要数据的服务器，然后才能访问。这个特点会严重制约分布式缓存集群的伸缩性设计，因为新上线的缓存服务器没有缓存任何数据，而已下线的缓存服务器还缓存着网站的许多热点数据。

必须让新上线的缓存服务器对整个分布式缓存集群影响最小，也就是说新加入缓存服务器后应使整个缓存服务器集群中已经缓存的数据尽可能还被访问到，这是分布式缓存集群伸缩性设计的最主要目标。

6.3.1 Memcached 分布式缓存集群的访问模型

以 Memcached 为代表的分布式缓存，访问模型如图 6.10 所示。

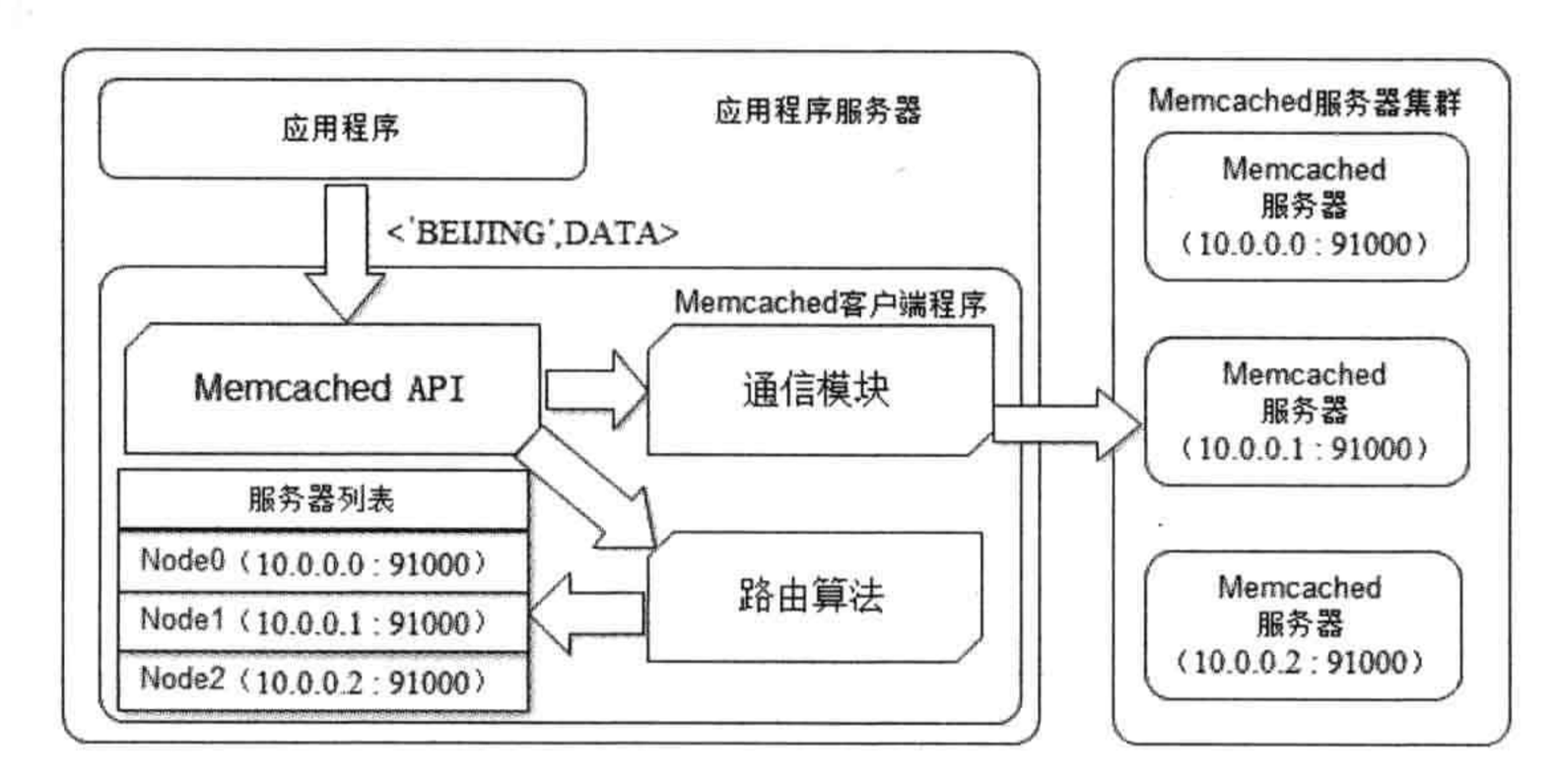

图 6.10　Memcached 分布式缓存访问模型

应用程序通过 Memcached 客户端访问 Memcached 服务器集群，Memcached 客户端主要由一组 API、Memcached 服务器集群路由算法、Memcached 服务器集群列表及通信模块构成。

其中路由算法负责根据应用程序输入的缓存数据 KEY 计算得到应该将数据写入到 Memcached 的哪台服务器（写缓存）或者应该从哪台服务器读数据（读缓存）。

一个典的型缓存写操作如图 6.10 中箭头所示路径。应用程序输入需要写缓存的数据 <'BEIJING',DATA>，API 将 KEY（'BEIJING'）输入路由算法模块，路由算法根据 KEY 和 Memcached 集群服务器列表计算得到一台服务编号（NODE1），进而得到该机器的 IP 地址和端口（10.0.0.0:91000）。API 调用通信模块和编号为 NODE1 的服务器通信，将数据 <'BEIJING',DATA>写入该服务器。完成一次分布式缓存的写操作。

读缓存的过程和写缓存一样，由于使用同样的路由算法和服务器列表，只要应用程序提供相同的 KEY（'BEIJING'），Memcached 客户端总是访问相同的服务器（NODE1）去读取数据。只要服务器还缓存着该数据，就能保证缓存命中。

6.3.2 Memcached 分布式缓存集群的伸缩性挑战

由上述讨论可得知，在 Memcached 分布式缓存系统中，对于服务器集群的管理，路由算法至关重要，和负载均衡算法一样，决定着究竟该访问集群中的哪台服务器。

简单的路由算法可以使用余数 Hash：用服务器数目除以缓存数据 KEY 的 Hash 值，余数为服务器列表下标编号。假设图 6.10 中'BEIJING'的 Hash 值是 490806430（Java 中的 HashCode()返回值），用服务器数目 3 除以该值，得到余数 1，对应节点 NODE1。由于 HashCode 具有随机性，因此使用余数 Hash 路由算法可保证缓存数据在整个 Memcached 服务器集群中比较均衡地分布。

对余数 Hash 路由算法稍加改进，就可以实现和负载均衡算法中加权负载均衡一样的加权路由。事实上，如果不需要考虑缓存服务器集群伸缩性，余数 Hash 几乎可以满足绝大多数的缓存路由需求。

但是，当分布式缓存集群需要扩容的时候，事情就变得棘手了。

假设由于业务发展，网站需要将 3 台缓存服务器扩容至 4 台。更改服务器列表，仍旧使用余数 Hash，用 4 除以'BEIJING'的 Hash 值 49080643，余数为 2，对应服务器 NODE2。由于数据<'BEIJING',DATA>缓存在 NODE1，对 NODE2 的读缓存操作失败，缓存没有命中。

很容易就可以计算出，3 台服务器扩容至 4 台服务器，大约有 75%（3/4）被缓存了的数据不能正确命中，随着服务器集群规模的增大，这个比例线性上升。当 100 台服务器的集群中加入一台新服务器，不能命中的概率是 99%（$N/(N+1)$）。

这个结果显然是不能接受的，在网站业务中，大部分的业务数据读操作请求事实上是通过缓存获取的，只有少量读操作请求会访问数据库，因此数据库的负载能力是以有缓存为前提而设计的。当大部分被缓存了的数据因为服务器扩容而不能正确读取时，这些数据访问的压力就落到了数据库的身上，这将大大超过数据库的负载能力，严重的可能会导致数据库宕机（这种情况下，不能简单重启数据库，网站也需要较长时间才能逐渐恢复正常。详见本书第 13 章。）

本来加入新的缓存服务器是为了降低数据库的负载压力，但是操作不当却导致了数据库的崩溃。如果不对问题和解决方案有透彻了解，网站技术总有想不到的陷阱让架构师一脚踩空。遇到这种情况，用某网站一位资深架构师的话说，就是“一股寒气从脚底板窜到了脑门心”。

一种解决办法是在网站访问量最少的时候扩容缓存服务器集群，这时候对数据库的负载冲击最小。然后通过模拟请求的方法逐渐预热缓存，使缓存服务器中的数据重新分布。但是这种方案对业务场景有要求，还需要技术团队通宵加班（网站访问低谷通常是在半夜）。

能不能通过改进路由算法，使得新加入的服务器不影响大部分缓存数据的正确命中呢？目前比较流行的算法是一致性 Hash 算法。

6.3.3 分布式缓存的一致性 Hash 算法

一致性 Hash 算法通过一个叫作一致性 Hash 环的数据结构实现 KEY 到缓存服务器的 Hash 映射，如图 6.11 所示。

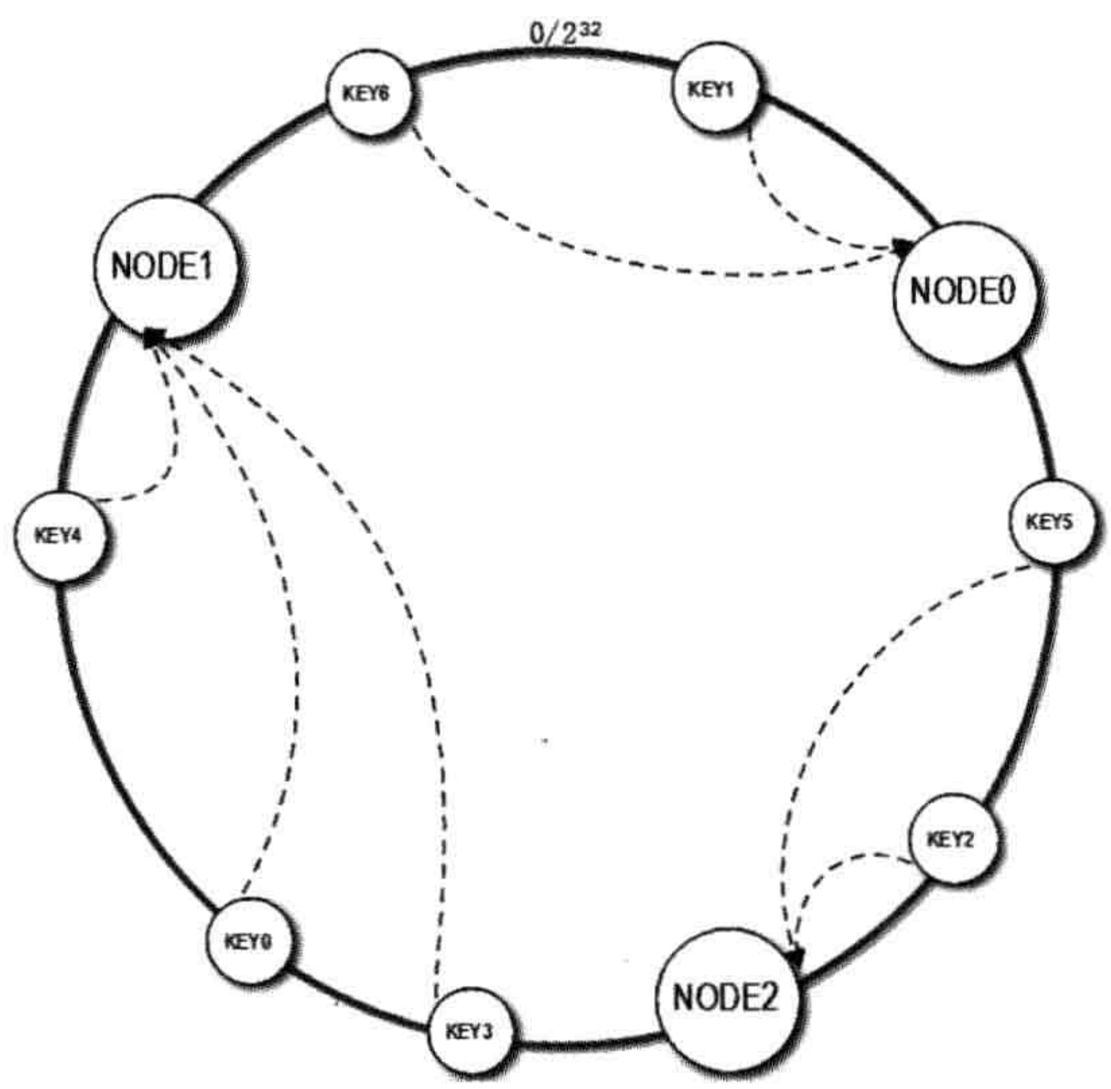

图 6.11 一致性 Hash 算法原理

具体算法过程为：先构造一个长度为 $0\sim2^{32}$ 的整数环（这个环被称作一致性 Hash 环），根据节点名称的 Hash 值（其分布范围同样为 $0\sim2^{32}$）将缓存服务器节点放置在这个 Hash 环上。然后根据需要缓存的数据的 KEY 值计算得到其 Hash 值（其分布范围也同样为 $0\sim2^{32}$），然后在 Hash 环上顺时针查找距离这个 KEY 的 Hash 值最近的缓存服务器节点，完成 KEY 到服务器的 Hash 映射查找。

在图 6.11 中，假设 NODE1 的 Hash 值为 3,594,963,423，NODE2 的 Hash 值为

1,845,328,979，而 KEY0 的 Hash 值为 2,534,256,785，那么 KEY0 在环上顺时针查找，找到的最近的节点就是 NODE1。

当缓存服务器集群需要扩容的时候，只需要将新加入的节点名称（NODE3）的 Hash 值放入一致性 Hash 环中，由于 KEY 是顺时针查找距离其最近的节点，因此新加入的节点只影响整个环中的一小段，如图 6.12 中深色一段。

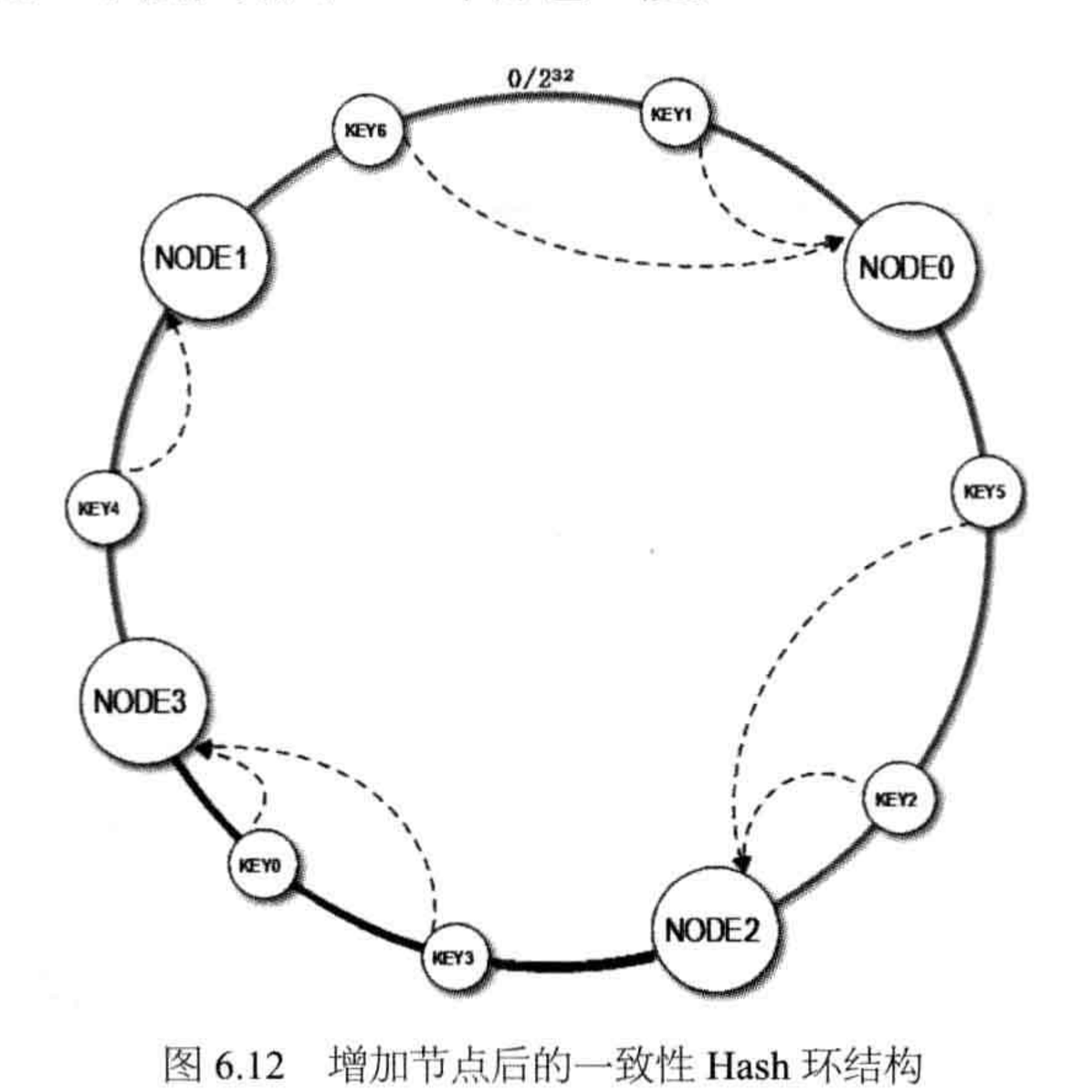

图 6.12　增加节点后的一致性 Hash 环结构

假设 NODE3 的 Hash 值是 2,790,324,235，那么加入 NODE3 后，KEY0（Hash 值 2,534,256,785）顺时针查找得到的节点就是 NODE3。

图 6.12 中，加入新节点 NODE3 后，原来的 KEY 大部分还能继续计算到原来的节点，只有 KEY3、KEY0 从原来的 NODE1 重新计算到 NODE3。这样就能保证大部分被缓存的数据还可以继续命中。3 台服务器扩容至 4 台服务器，可以继续命中原有缓存数据的概率是 75%，远高于余数 Hash 的 25%，而且随着集群规模越大，继续命中原有缓存数据的概率也逐渐增大，100 台服务器扩容增加 1 台服务器，继续命中的概率是 99%。虽然仍有小部分数据缓存在服务器中不能被读到，但是这个比例足够小，通过访问数据库获取也不会对数据库造成致命的负载压力。

具体应用中，这个长度为 2^{32} 的一致性 Hash 环通常使用二叉查找树实现，Hash 查找过程实际上是在二叉查找树中查找不小于查找数的最小数值。当然这个二叉树的最右边

叶子节点和最左边的叶子节点相连接，构成环。

但是，上面描述的算法过程还存在一个小小的问题。

新加入的节点 NODE3 只影响了原来的节点 NODE1，也就是说一部分原来需要访问 NODE1 的缓存数据现在需要访问 NODE3（概率上是 50%）。但是原来的节点 NODE0 和 NODE2 不受影响，这就意味着 NODE0 和 NODE2 缓存数据量和负载压力是 NODE1 与 NODE3 的两倍。如果 4 台机器的性能是一样的，那么这种结果显然不是我们需要的。

怎么办？

计算机领域有句话：**计算机的任何问题都可以通过增加一个虚拟层来解决**。计算机硬件、计算机网络、计算机软件都莫不如此。计算机网络的 7 层协议，每一层都可以看作是下一层的虚拟层；计算机操作系统可以看作是计算机硬件的虚拟层；Java 虚拟机可以看作是操作系统的虚拟层；分层的计算机软件架构事实上也是利用虚拟层的概念。

解决上述一致性 Hash 算法带来的负载不均衡问题，也可以通过使用虚拟层的手段：将每台物理缓存服务器虚拟为一组虚拟缓存服务器，将虚拟服务器的 Hash 值放置在 Hash 环上，KEY 在环上先找到虚拟服务器节点，再得到物理服务器的信息。

这样新加入物理服务器节点时，是将一组虚拟节点加入环中，如果虚拟节点的数目足够多，这组虚拟节点将会影响同样多数目的已经在环上存在的虚拟节点，这些已经存在的虚拟节点又对应不同的物理节点。最终的结果是：新加入一台缓存服务器，将会较为均匀地影响原来集群中已经存在的所有服务器，也就是说分摊原有缓存服务器集群中所有服务器的一小部分负载，其总的影响范围和上面讨论过的相同。如图 6.13 所示。

在图 6.13 中，新加入节点 NODE3 对应的一组虚拟节点为 V30，V31，V32，加入到一致性 Hash 环上后，影响 V01，V12，V22 三个虚拟节点，而这三个虚拟节点分别对应 NODE0，NODE1，NODE2 三个物理节点。最终 Memcached 集群中加入一个节点，但是同时影响到集群中已存在的三个物理节点，在理想情况下，每个物理节点受影响的数据量（还在缓存中，但是不能被访问到数据）为其节点缓存数据量的 1/4（$X/(N+X)$，N 为原有物理节点数，X 为新加入物理节点数），也就是集群中已经被缓存的数据有 75%可以被继续命中，和未使用虚拟节点的一致性 Hash 算法结果相同。

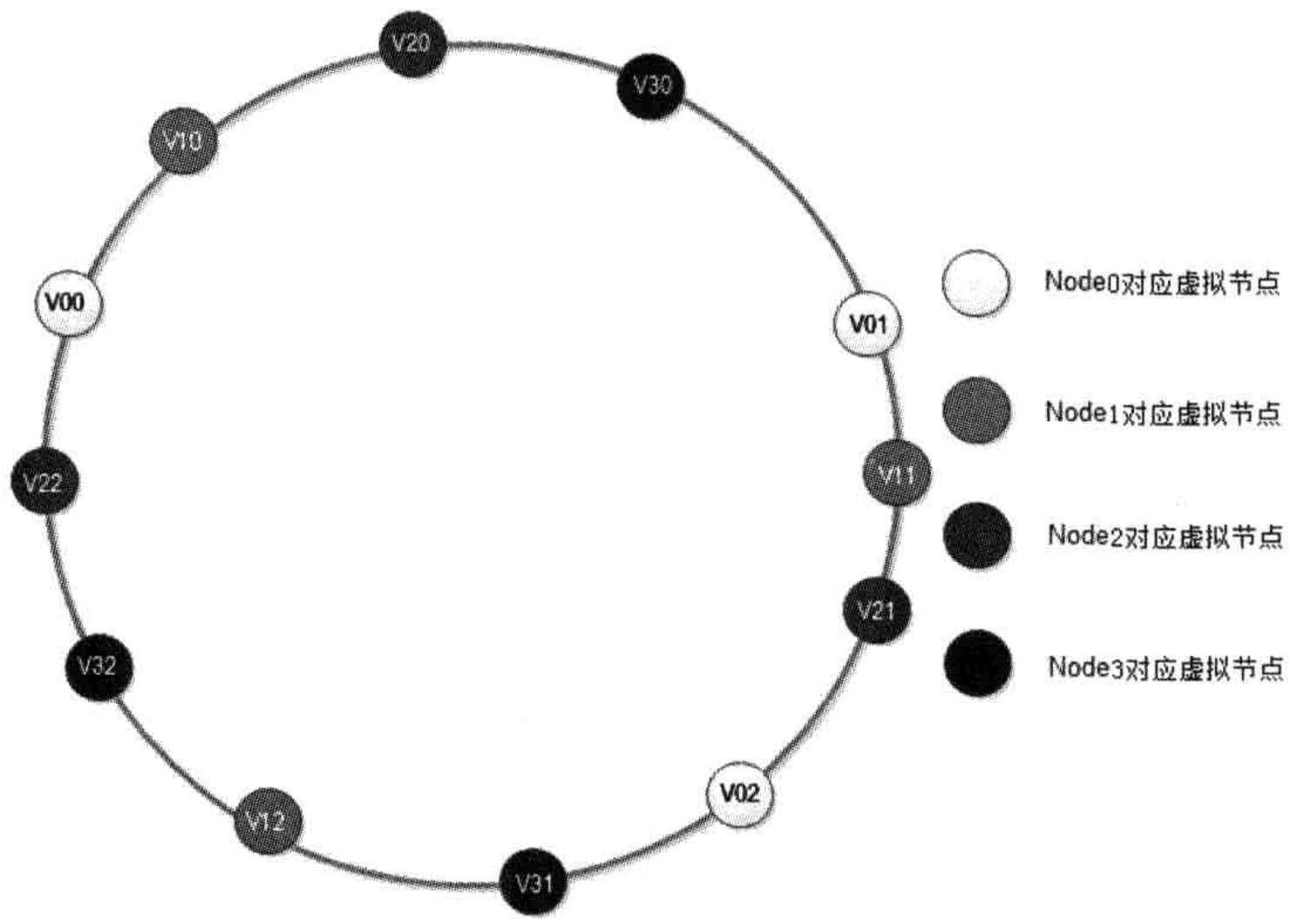

图 6.13　使用虚拟节点的一致性 Hash 环

显然每个物理节点对应的虚拟节点越多，各个物理节点之间的负载越均衡，新加入物理服务器对原有的物理服务器的影响越保持一致（这就是一致性 Hash 这个名称的由来）。那么在实践中，一台物理服务器虚拟为多少个虚拟服务器节点合适呢？太多会影响性能，太少又会导致负载不均衡，一般说来，经验值是 150，当然根据集群规模和负载均衡的精度需求，这个值应该根据具体情况具体对待。

6.4　数据存储服务器集群的伸缩性设计

和缓存服务器集群的伸缩性设计不同，数据存储服务器集群的伸缩性对数据的持久性和可用性提出了更高的要求。

缓存的目的是加速数据读取的速度并减轻数据存储服务器的负载压力，因此部分缓存数据的丢失不影响业务的正常处理，因为数据还可以从数据库等存储服务器上获取。

而数据存储服务器必须保证数据的可靠存储，任何情况下都必须保证数据的可用性和正确性。因此缓存服务器集群的伸缩性架构方案不能直接适用于数据库等存储服务器。存储服务器集群的伸缩性设计相对更复杂一些，具体说来，又可分为关系数据库集群的伸缩性设计和 NoSQL 数据库的伸缩性设计。

6.4.1 关系数据库集群的伸缩性设计

关系数据库凭借其简单强大的 SQL 和众多成熟的商业数据库产品，占据了从企业应用到网站系统的大部分业务数据存储服务。市场上主要的关系数据都支持数据复制功能，使用这个功能可以对数据库进行简单伸缩。图 6.14 为使用数据复制的 MySQL 集群伸缩性方案。

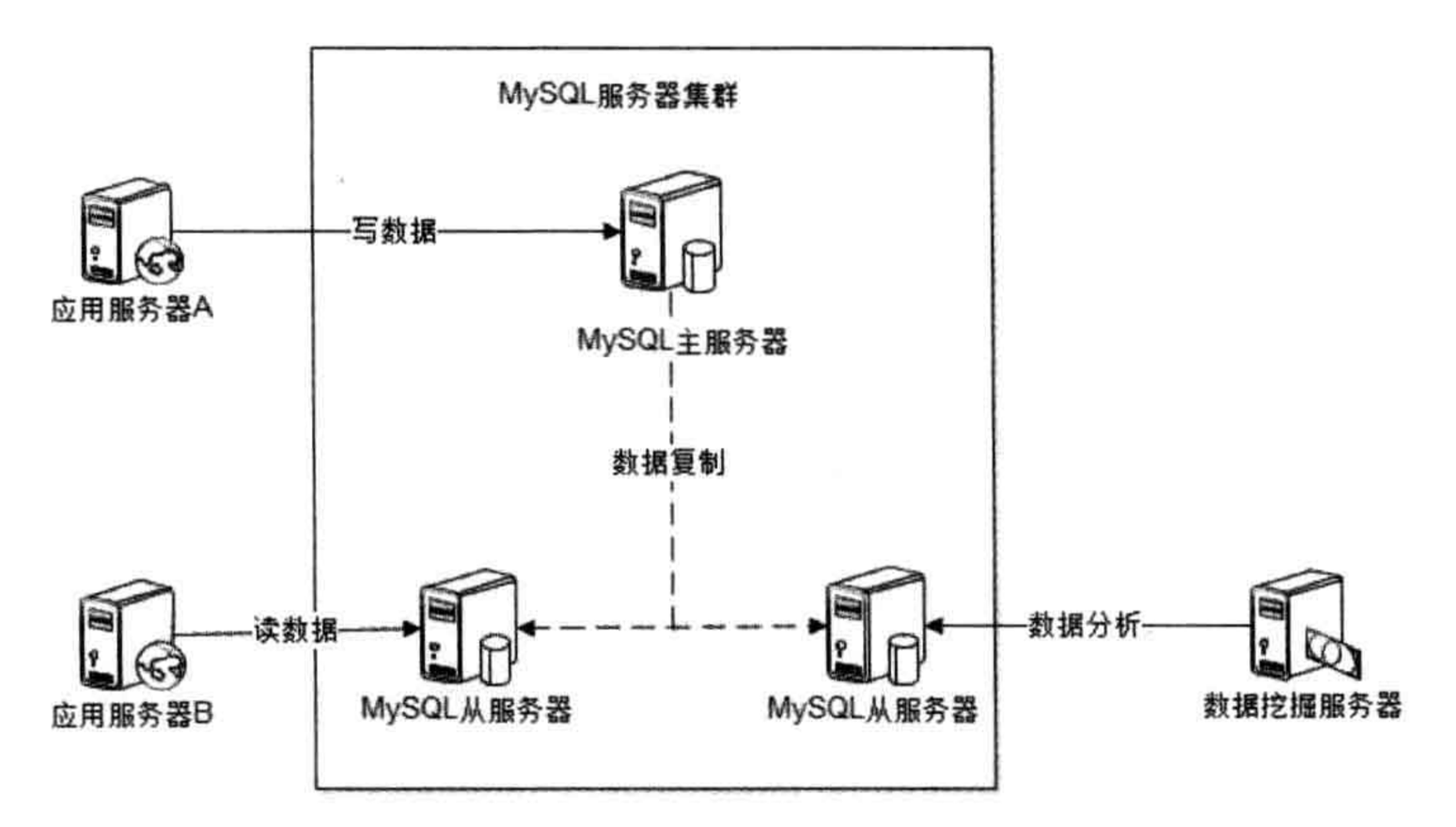

图 6.14 MySQL 集群伸缩性方案

在这种架构中，虽然多台服务器部署 MySQL 实例，但是它们的角色有主从之分，数据写操作都在主服务器上，由主服务器将数据同步到集群中其他从服务器，数据读操作及数据分析等离线操作在从服务器上进行。

除了数据库主从读写分离，前面提到的业务分割模式也可以用在数据库，不同业务数据表部署在不同的数据库集群上，即俗称的数据分库。这种方式的制约条件是跨库的表不能进行 Join 操作。

在大型网站的实际应用中，即使进行了分库和主从复制，对一些单表数据仍然很大的表，比如 Facebook 的用户数据库，淘宝的商品数据库，还需要进行分片，将一张表拆开分别存储在多个数据库中。

目前网站在线业务应用中比较成熟的支持数据分片的分布式关系数据库产品主要有开源的 Amoeba（http://sourceforge.net/projects/amoeba/）和 Cobar（http://code.alibabatech.com/wiki/display/cobar/Home）。这两个产品有相似的架构设计，以 Cobar 为例，部署模型如图 6.15 所示。

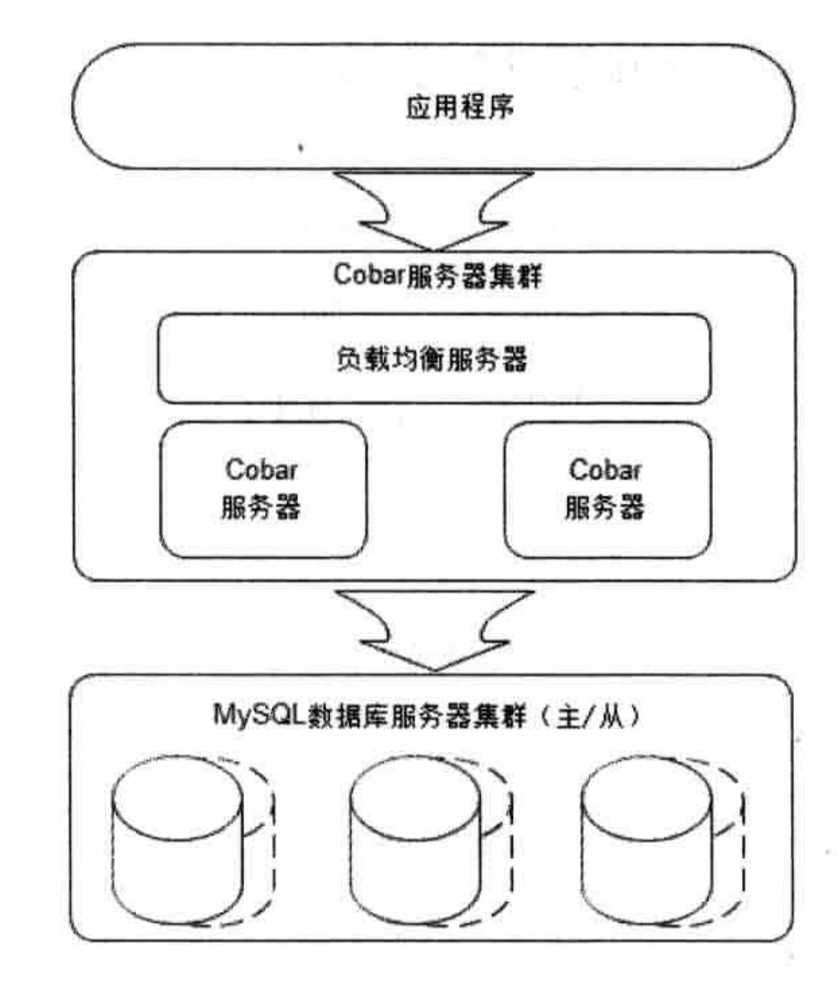

图 6.15　Cobar 部署模型

Cobar 是一个分布式关系数据库访问代理，介于应用服务器和数据库服务器之间（Cobar 也支持非独立部署，以 lib 的方式和应用程序部署在一起）。应用程序通过 JDBC 驱动访问 Cobar 集群，Cobar 服务器根据 SQL 和分库规则分解 SQL，分发到 MySQL 集群不同的数据库实例上执行（每个 MySQL 实例都部署为主/从结构，保证数据高可用）。

Cobar 系统组件模型如图 6.16 所示。

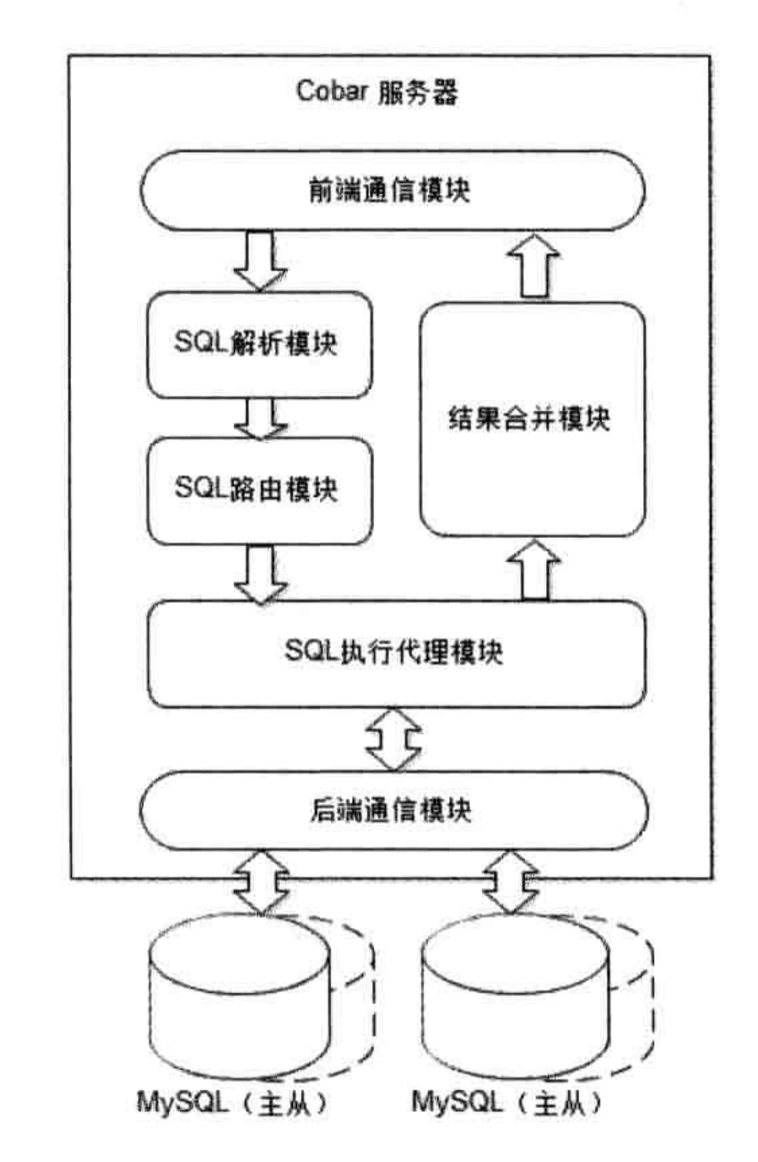

图 6.16　Cobar 系统组件模型

前端通信模块负责和应用程序通信，接收到 SQL 请求（select * from users where userid in (12,22,23)）后转交给 SQL 解析模块，SQL 解析模块解析获得 SQL 中的路由规则查询条件（userid in(12,22,23)）再转交给 SQL 路由模块，SQL 路由模块根据路由规则配置（userid 为偶数路由至数据库 A，userid 为奇数路由至数据库 B）将应用程序提交的 SQL 分解成两条 SQL（select * from users where userid in (12,22)；select * from users where userid in (23)；）转交给 SQL 执行代理模块，发送至数据库 A 和数据库 B 分别执行。

数据库 A 和数据库 B 的执行结果返回至 SQL 执行模块，通过结果合并模块将两个返回结果集合并成一个结果集，最终返回给应用程序，完成在分布式数据库中的一次访问请求。

那么 Cobar 如何做集群的伸缩呢？

Cobar 的伸缩有两种：Cobar 服务器集群的伸缩和 MySQL 服务器集群的伸缩。

Cobar 服务器可以看作是无状态的应用服务器，因此其集群伸缩可以简单使用负载均衡的手段实现。而 MySQL 中存储着数据，要想保证集群扩容后数据一致负载均衡，必须要做数据迁移，将集群中原来机器中的数据迁移到新添加的机器中，如图 6.17 所示。

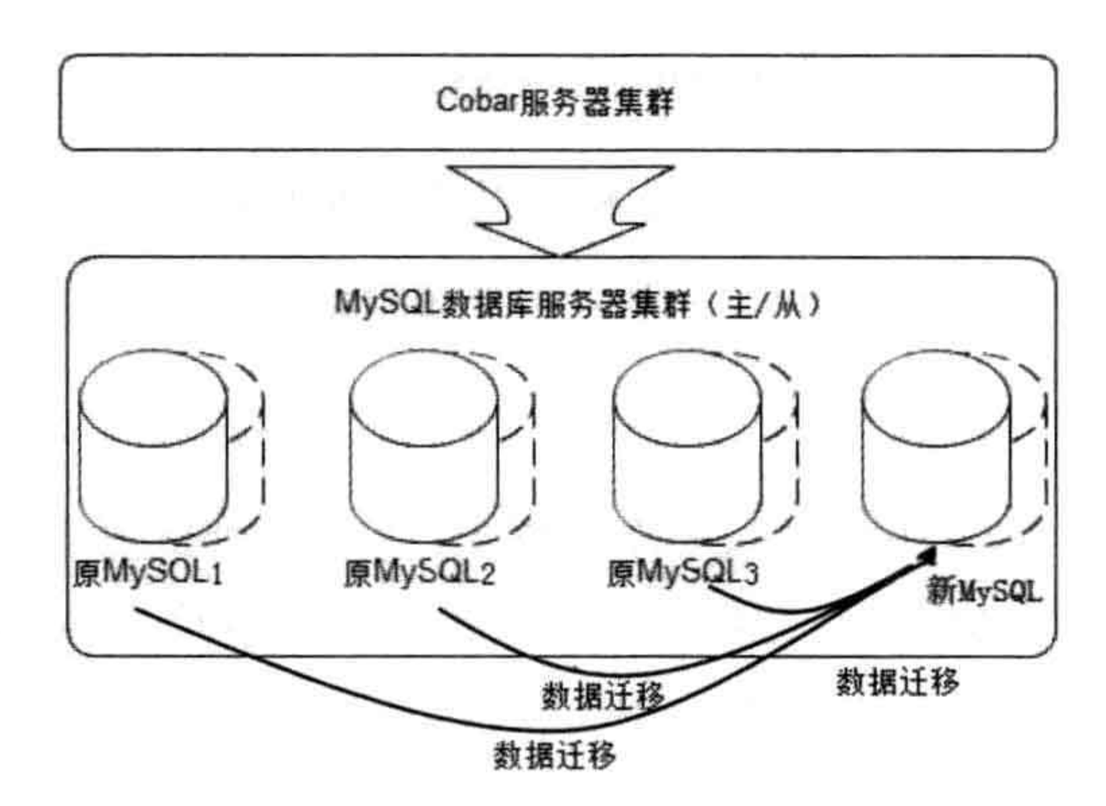

图 6.17 Cobar 集群伸缩原理

具体迁移哪些数据可以利用一致性 Hash 算法（即路由模块使用一致性 Hash 算法进行路由），尽量使需要迁移的数据最少。但是迁移数据需要遍历数据库中每条记录（的索引），重新进行路由计算确定其是否需要迁移，这会对数据库访问造成一定压力。并且需要解决迁移过程中数据的一致性、可访问性、迁移过程中服务器宕机时的可用性等诸多

问题。

实践中，Cobar 利用了 MySQL 的数据同步功能进行数据迁移。数据迁移不是以数据为单位，而是以 Schema 为单位。在 Cobar 集群初始化时，在每个 MySQL 实例创建多个 Schema（根据业务远景规划未来集群规模，如集群最大规模为 1000 台数据库服务器，那么总的初始 Schema 数≥1000）。集群扩容的时候，从每个服务器中迁移部分 Schema 到新机器中，由于迁移以 Schema 为单位，迁移过程可以使用 MySQL 的同步机制，如图 6.18 所示。

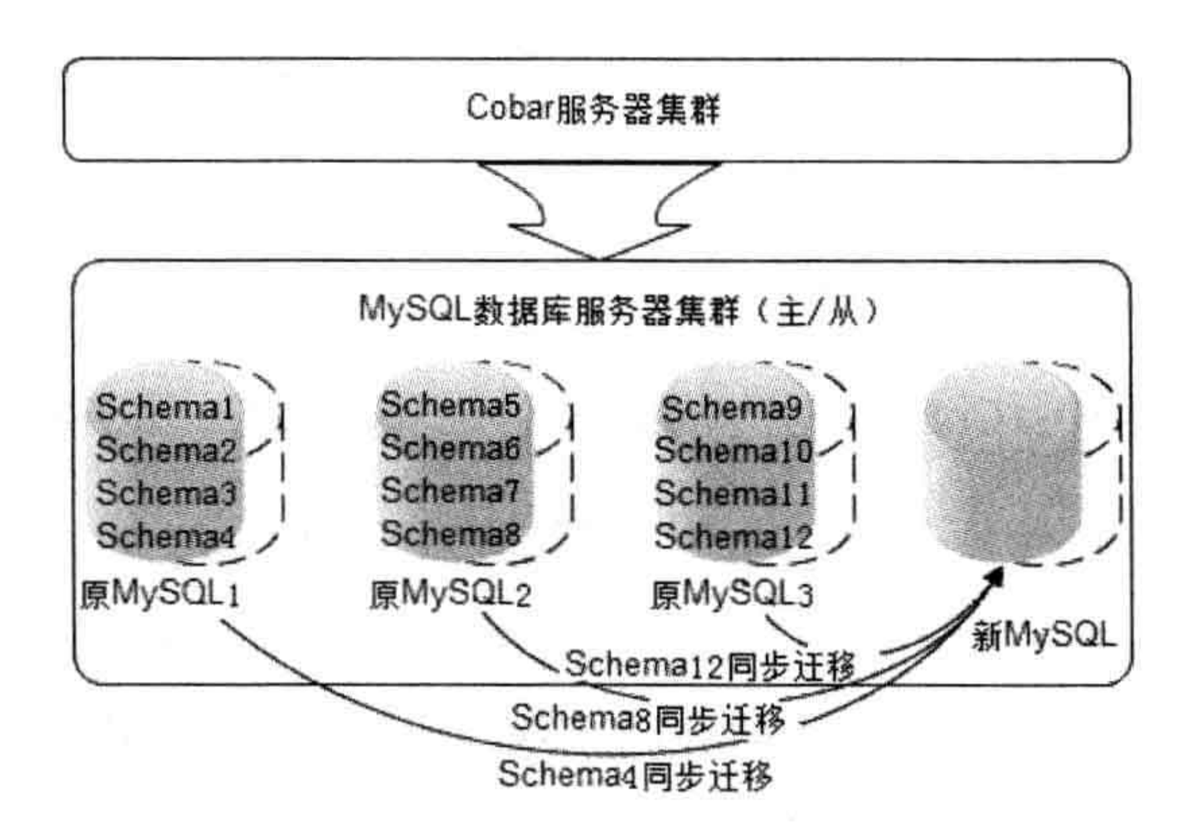

图 6.18　利用 MySQL 同步机制实现 Cobar 集群伸缩

同步完成时，即新机器中 Schema 数据和原机器中 Schema 数据一致的时候，修改 Cobar 服务器的路由配置，将这些 Schema 的 IP 修改为新机器的 IP，然后删除原机器中的相关 Schema，完成 MySQL 集群扩容。

在整个分布式关系数据库的访问请求过程中，Cobar 服务器处理消耗的时间是很少的，时间花费主要还是在 MySQL 数据库端，因此应用程序通过 Cobar 访问分布式关系数据库，性能基本和直接访问关系数据库相当，可以满足网站在线业务的实时处理需求。事实上由于 Cobar 代替应用程序连接数据库，数据库只需要维护更少的连接，减少不必要的资源消耗，改善性能。

但由于 Cobar 路由后只能在单一数据库实例上处理查询请求，因此无法执行跨库的 JOIN 操作，当然更不能执行跨库的事务处理。

相比关系数据库本身功能上的优雅强大，目前各类分布式关系数据库解决方案都显

得非常简陋，限制了关系数据库某些功能的使用。但是当网站业务面临不停增长的海量业务数据存储压力时，又不得不利用分布式关系数据库的集群伸缩能力，这时就必须从业务上回避分布式关系数据库的各种缺点：避免事务或利用事务补偿机制代替数据库事务；分解数据访问逻辑避免 JOIN 操作等。

除了上面提到的分布式数据库，还有一类分布式数据库可以支持 JOIN 操作执行复杂的 SQL 查询，如 GreenPlum。但是这类数据库的访问延迟比较大（可以想象，JOIN 操作需要在服务器间传输大量的数据），因此一般使用在数据仓库等非实时业务中。

6.4.2 NoSQL 数据库的伸缩性设计

在计算机数据存储领域，一直是关系数据库（Relation Database）的天下，以至传统企业应用领域，许多应用系统设计都是面向数据库设计——先设计数据库然后设计程序，从而导致关系模型绑架对象模型，并由此引申出旷日持久的业务对象贫血模型与充血模型之争。业界为了解决关系数据库的不足，提出了诸多方案，比较有名的是对象数据库，但是这些数据库的出现只是进一步证明关系数据库的优越而已。直到大型网站遇到了关系数据库难以克服的缺陷——糟糕的海量数据处理能力及僵硬的设计约束，局面才有所改善。为了解决上述问题，NoSQL 这一概念被提了出来，以弥补关系数据库的不足。

NoSQL，主要指非关系的、分布式的数据库设计模式。也有许多专家将 NoSQL 解读为 Not Only SQL，表示 NoSQL 只是关系数据库的补充，而不是替代方案。一般而言，NoSQL 数据库产品都放弃了关系数据库的两大重要基础：以关系代数为基础的结构化查询语言（SQL）和事务一致性保证（ACID）。而强化其他一些大型网站更关注的特性：高可用性和可伸缩性。

开源社区有各种 NoSQL 产品，其支持的数据结构和伸缩特性也各不相同，目前看来，应用最广泛的是 Apache HBase。

HBase 为可伸缩海量数据储存而设计，实现面向在线业务的实时数据访问延迟。HBase 的伸缩性主要依赖其可分裂的 HRegion 及可伸缩的分布式文件系统 HDFS 实现。

HBase 的整体架构如图 6.19 所示。HBase 中，数据以 HRegion 为单位进行管理，也就是说应用程序如果想要访问一个数据，必须先找到 HRegion，然后将数据读写操作提交给 HRegion，由 HRegion 完成存储层面的数据操作。每个 HRegion 中存储一段 Key 值区

间[key1,key2)的数据，HRegionServer 是物理服务器，每个 HRegionServer 上可以启动多个 HRegion 实例。当一个 HRegion 中写入的数据太多，达到配置的阈值时，HRegion 会分裂成两个 HRegion，并将 HRegion 在整个集群中进行迁移，以使 HregionServer 的负载均衡。

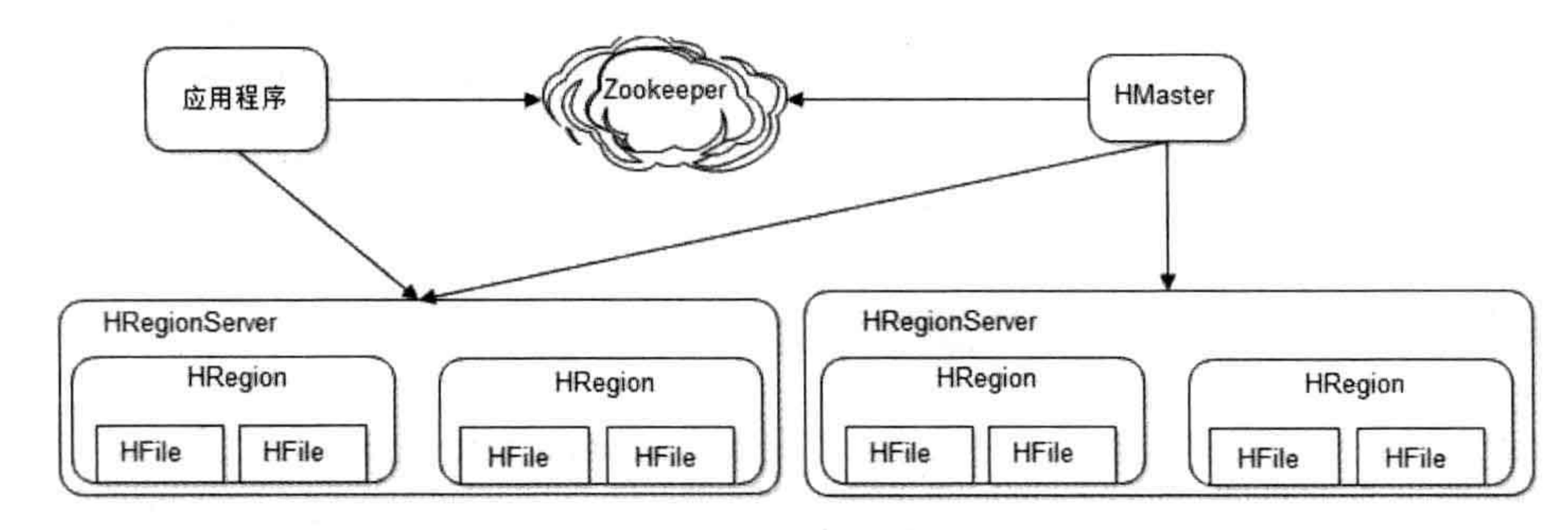

图 6.19　HBase 架构

所有 HRegion 的信息（存储的 Key 值区间、所在 HRegionServer 地址、访问端口号等）都记录在 HMaser 服务器上，为了保证高可用，HBase 启动多个 HMaser，并通过 Zookeeper（一个支持分布式一致性的数据管理服务）选举出一个主服务器，应用程序通过 Zookeeper 获得主 HMaser 的地址，输入 Key 值获得这个 Key 所在的 HRegionServer 地址，然后请求 HRegionServer 上的 HRegion，获得需要的数据。调用时序如图 6.20 所示。

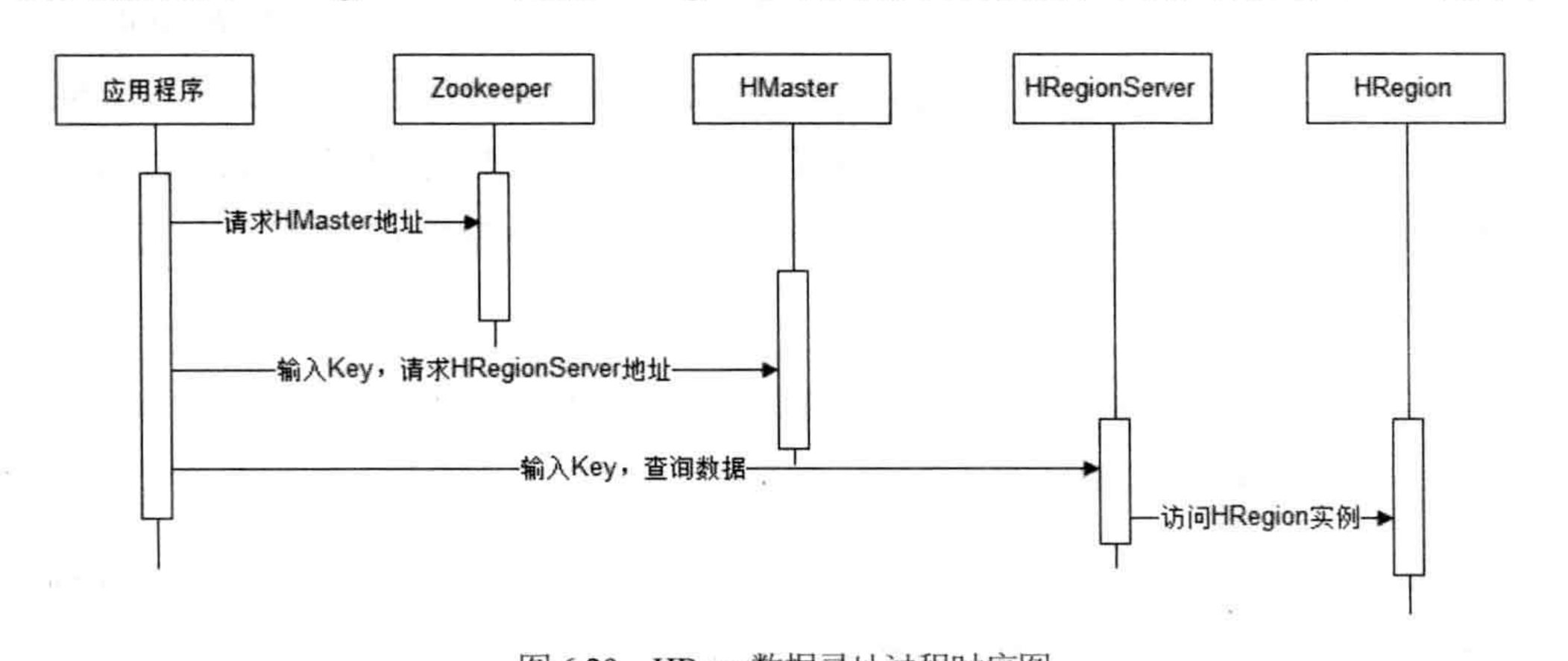

图 6.20　HBase 数据寻址过程时序图

数据写入过程也是一样，需要先得到 HRegion 才能继续操作，HRegion 会把数据存储在若干个叫作 HFile 格式的文件中，这些文件使用 HDFS 分布式文件系统（参考本书第 4

章）存储，在整个集群内分布并高可用。当一个 HRegion 中数据量太多时，HRegion（连同 HFile）会分裂成两个 HRegion，并根据集群中服务器负载进行迁移，如果集群中有新加入的服务器，也就是说有了新的 HRegionServer，由于其负载较低，也会把 HRegion 迁移过去并记录到 HMaster，从而实现 HBase 的线性伸缩。

6.5 小结

伸缩性架构设计能力是网站架构师必须具备的能力。

伸缩性架构设计是简单的，因为几乎所有稍有规模的网站都必须是可伸缩的，有很多案例可供借鉴，同时又有大量商业的、开源的提供伸缩性能力的软硬件产品可供选用。然而伸缩性设计又是复杂的，没有通用的、完美的解决方案和产品，网站伸缩性往往和可用性、正确性、性能等耦合在一起，改善伸缩性可能会影响一些网站的其他特性，网站架构师必须对网站的商业目标、历史演化、技术路线了然于胸，甚至还需要综合考虑技术团队的知识储备和结构、管理层的战略愿景和规划，才能最终做出对网站伸缩性架构最合适的决策。

一个具有良好伸缩性架构设计的网站，其设计总是走在业务发展的前面，在业务需要处理更多访问和服务之前，就已经做好充足准备，当业务需要时，只需要购买或者租用服务器简单部署实施就可以了，技术团队亦可高枕无忧。反之，设计和技术走在业务的后面，采购来的机器根本就没办法加入集群，勉强加了进去，却发现瓶颈不在这里，系统整体处理能力依然上不去。技术团队每天加班，却总是拖公司发展的后腿。架构师对网站伸缩性的把握，一线之间，天堂和地狱。

高手定律：这个世界只有遇不到的问题，没有解决不了的问题，高手之所以成为高手，是因为他们遇到了常人很难遇到的问题，并解决了。所以百度有很多广告搜索的高手，淘宝有很多海量数据的高手，QQ 有很多高并发业务的高手，原因大抵如此。一个 100 万用户的网站，不会遇到 1 亿用户同时在线的问题；一个拥有 100 万件商品网站的工程师，可能无法理解一个拥有 10 亿件商品网站的架构。

救世主定律：遇到问题，分析问题，最后总能解决问题。如果遇到问题就急匆匆地从外面挖一个高手，然后指望高手如探囊取物般轻松搞定，最后怕是只有彼此抱怨和伤害。许多问题只是看起来一样，具体问题总是要具体对待的，没有银弹，没有救世主。所以这个定律准确地说应该是“没有救世主定律”。

7 随需应变：网站的可扩展架构

国内某大型互联网企业经常因为对同行的产品进行微创新，然后推出自己的产品而遭人诟病，不讨论这种做法是否合适，我们分析这些产品，发现大多数都比原创产品有更好的用户体验。这些产品常常后来居上，更速度地推出新功能，吸引用户注意，进而占据市场。

微信从发布到拥有 1 亿用户，仅仅用了一年的时间。而据说摇一摇这个功能是两个实习生用一个星期就开发完成上线的。

使用 TOP（Taobao Open API），一个技术熟练的淘宝客网站开发工程师只需要用几个晚上的业余时间就可以开发部署一个炫目的购物导购网站。

如此轻易地就可以开发一个新产品，如此快速地就可以实现一个新功能，他们是如何做到的？

为什么有的网站必须规定系统发布日，一到发布日就如临大敌，整个技术部加班通宵达旦；而有的网站就可以随时发布，新功能可以随时快速上线。

这些都有赖于网站的扩展性架构设计，就是在对现有系统影响最小的情况下，系统功能可持续扩展及提升的能力。

经常听到各种场合中对扩展性和伸缩性的误用，包括许多资深网站架构师也常常混淆两者，用扩展性表示伸缩性。在此，我们澄清下这两个概念。

扩展性（Extensibility）

指对现有系统影响最小的情况下，系统功能可持续扩展或提升的能力。表现在系统基础设施稳定不需要经常变更，应用之间较少依赖和耦合，对需求变更可以敏捷响应。它是系统架构设计层面的**开闭原则**（对扩展开放，对修改关闭），架构设计考虑未来功能扩展，当系统增加新功能时，不需要对现有系统的结构和代码进行修改。

伸缩性（Scalability）

指系统能够通过增加（减少）自身资源规模的方式增强（减少）自己计算处理事务的能力。如果这种增减是成比例的，就被称作线性伸缩性。在网站架构中，通常指利用集群的方式增加服务器数量、提高系统的整体事务吞吐能力。

7.1 构建可扩展的网站架构

开发低耦合系统是软件设计的终极目标之一，这一目标驱动着软件开发技术的创新与发展，从软件与硬件的第一次分离到操作系统的诞生；从汇编语言到面向过程的开发语言，再到面向对象的编程语言；从各种软件工具集到各种开发框架；无不体现着降低软件系统耦合性这一终极目标。可以说，度量一个开发框架、设计模式、编程语言优劣的重要尺度就是衡量它是不是让软件开发过程和软件产品更加低耦合。

显而易见，低耦合的系统更容易扩展，低耦合的模块更容易复用，一个低耦合的系统设计也会让开发过程和维护变得更加轻松和容易管理。一个复杂度为 100 的系统，如果能够分解成没有耦合的两个子系统，那么每个子系统的复杂度不是 50，而可能是 25。当然，完全没有耦合就是没有关系，也就无法组合出一个强大的系统。那么如何分解系统的各个模块、如何定义各个模块的接口、如何复用组合不同的模块构造成一个完整的系统，这是软件设计中最有挑战的部分。

笔者认为，软件架构师最大的价值不在于掌握多少先进的技术，而在于具有将一个大系统切分成 N 个低耦合的子模块的能力，这些子模块包含横向的业务模块，也包含纵向的基础技术模块。这种能力一部分源自专业的技术和经验，还有一部分源自架构师对业务场景的理解、对人性的把握、甚至对世界的认知。

大型网站也常常意味着功能复杂，产品众多。网站为了在市场竞争中胜出，不断推出各种新产品，为了把握市场机会，这些产品从策划到上线，时间非常短暂，技术团队必须在产品设计和需求分析结束之后，快速地开发完成一个新产品。同时经过长期的演化和发展，这些产品之间的关系错综复杂，维护也变得异常困难。这些问题对网站的可扩展架构提出了挑战和要求。

设计网站可扩展架构的核心思想是模块化，并在此基础之上，降低模块间的耦合性，提高模块的复用性。

我们在本书第 6 章讨论过网站通过分层和分割的方式进行架构伸缩，分层和分割也是模块化设计的重要手段，利用分层和分割的方式将软件分割为若干个低耦合的独立的组件模块，这些组件模块以消息传递及依赖调用的方式聚合成一个完整的系统。

在大型网站中，这些模块通过分布式部署的方式，独立的模块部署在独立的服务器（集群）上，从物理上分离模块之间的耦合关系，进一步降低耦合性提高复用性。

模块分布式部署以后具体聚合方式主要有分布式消息队列和分布式服务。

7.2 利用分布式消息队列降低系统耦合性

如果模块之间不存在直接调用，那么新增模块或者修改模块就对其他模块影响最小，这样系统的可扩展性无疑更好一些。

7.2.1 事件驱动架构

事件驱动架构（Event Driven Architecture）：通过在低耦合的模块之间传输事件消息，以保持模块的松散耦合，并借助事件消息的通信完成模块间合作，典型的 EDA 架构就是操作系统中常见的生产者消费者模式。在大型网站架构中，具体实现手段有很多，最常用的是分布式消息队列，如图 7.1 所示。

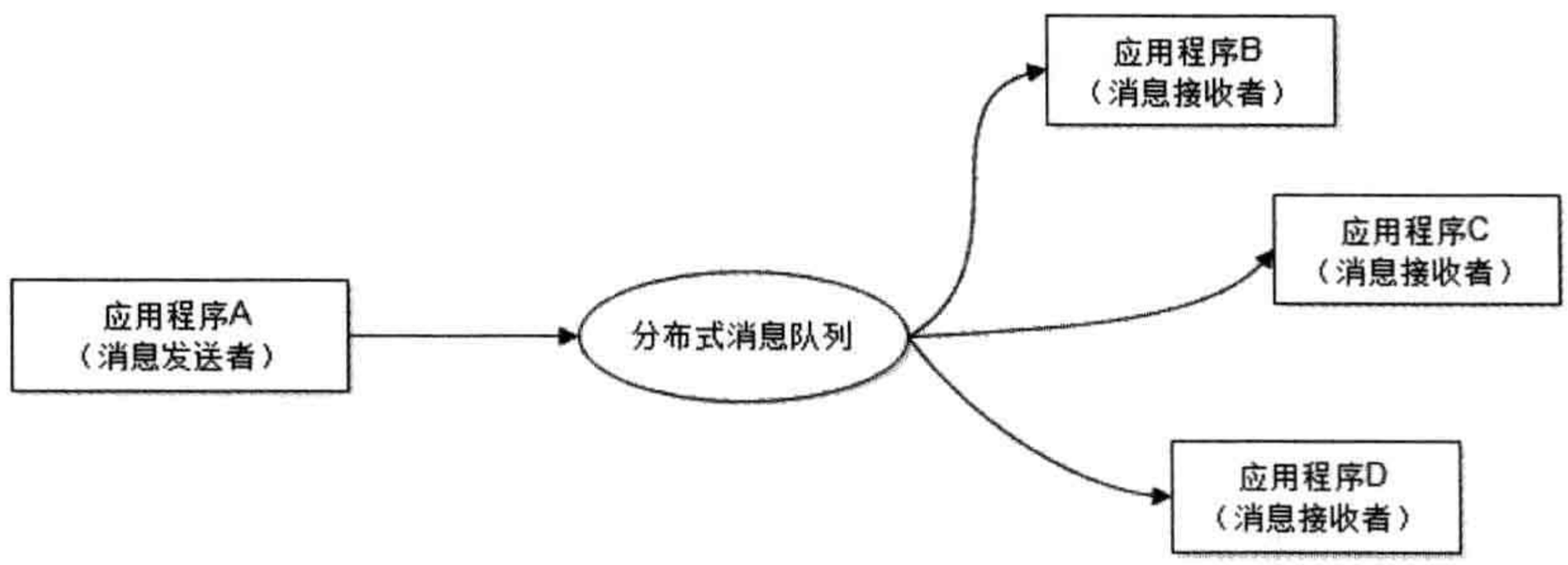

图 7.1　利用消息队列实现的事件驱动架构

消息队列利用发布—订阅模式工作，消息发送者发布消息，一个或者多个消息接收者订阅消息。消息发送者是消息源，在对消息进行处理后将消息发送至分布式消息队列，消息接受者从分布式消息队列获取该消息后继续进行处理。可以看到，消息发送者和消息接受者之间没有直接耦合，消息发送者将消息发送至分布式消息队列即结束对消息的处理，而消息接受者只需要从分布式消息队列获取消息后进行处理，不需要知道该消息从何而来。对新增业务，只要对该类消息感兴趣，即可订阅该消息，对原有系统和业务没有任何影响，从而实现网站业务的可扩展设计。

消息接受者在对消息进行过滤、处理、包装后，构造成一个新的消息类型，将消息继续发送出去，等待其他消息接受者订阅处理该消息。因此基于事件（消息对象）驱动的业务架构可以是一系列的流程。

由于消息发送者不需要等待消息接受者处理数据就可以返回，系统具有更好的响应延迟；同时，在网站访问高峰，消息可以暂时存储在消息队列中等待消息接受者根据自身负载处理能力控制消息处理速度，减轻数据库等后端存储的负载压力。

7.2.2　分布式消息队列

队列是一种先进先出的数据结构，分布式消息队列可以看作将这种数据结构部署到独立的服务器上，应用程序可以通过远程访问接口使用分布式消息队列，进行消息存取操作，进而实现分布式的异步调用，基本原理如图 7.2 所示。

消息生产者应用程序通过远程访问接口将消息推送给消息队列服务器，消息队列服务器将消息写入本地内存队列后立即返回成功响应给消息生产者。消息队列服务器根据消息订阅列表查找订阅该消息的消息消费者应用程序，将消息队列中的消息按照先进先

出（FIFO）的原则将消息通过远程通信接口发送给消息消费者程序。

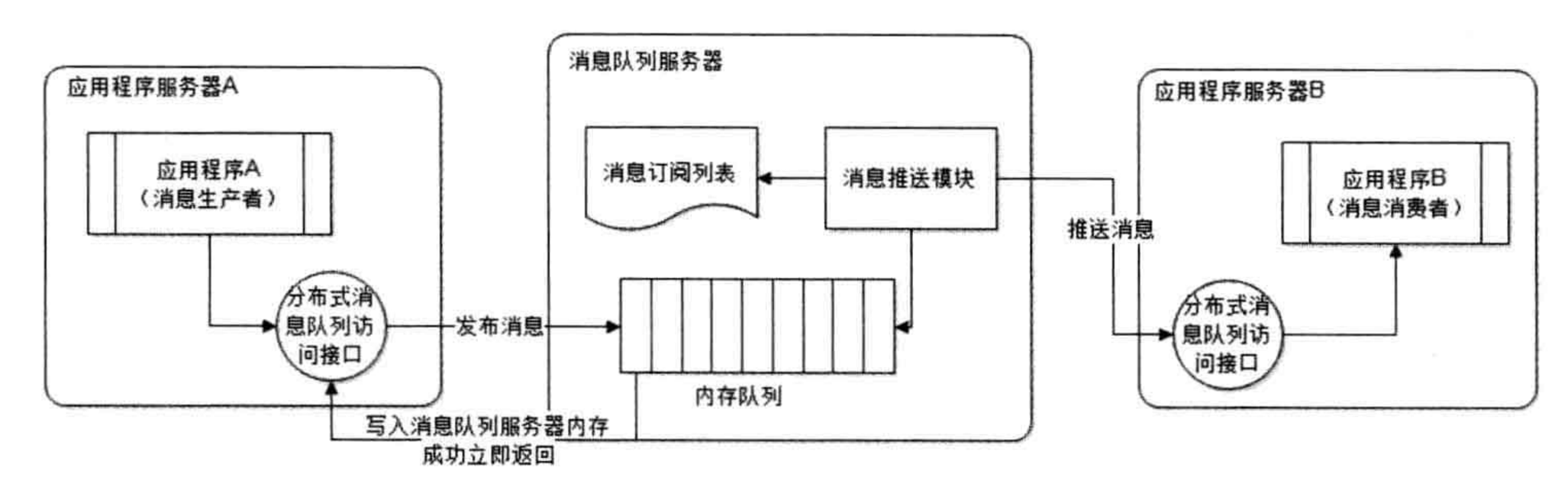

图 7.2　分布式消息队列架构原理

目前开源的和商业的分布式消息队列产品有很多，比较著名的如 Apache ActiveMQ 等，这些产品除了实现分布式消息队列的一般功能，在可用性、伸缩性、数据一致性、性能和可管理性方面也做了很多改善。

在**伸缩性**方面，由于消息队列服务器上的数据可以看作是被即时处理的，因此类似于无状态的服务器，伸缩性设计比较简单。将新服务器加入分布式消息队列集群中，通知生产者服务器更改消息队列服务器列表即可。

在**可用性**方面，为了避免消费者进程处理缓慢，分布式消息队列服务器内存空间不足造成的问题，如果内存队列已满，会将消息写入磁盘，消息推送模块在将内存队列消息处理完以后，将磁盘内容加载到内存队列继续处理。

为了避免消息队列服务器宕机造成消息丢失，会将消息成功发送到消息队列的消息存储在消息生产者服务器，等消息真正被消息消费者服务器处理后才删除消息。在消息队列服务器宕机后，生产者服务器会选择分布式消息队列服务器集群中其他的服务器发布消息。

分布式消息队列可以很复杂，比如可以支持 ESB（企业服务总线）、支持 SOA（面向服务的架构）等；也可以很简单，比如用 MySQL 也可以当作分布式消息队列：消息生产者程序将消息当作数据记录写入数据库，消息消费者程序查询数据库并按记录写入时间戳排序，就实现了一个事实上的分布式消息队列，而且这个消息队列使用成熟的 MySQL 运维手段，也可以达到较高的可用性和性能指标。

7.3 利用分布式服务打造可复用的业务平台

使用分布式服务是降低系统耦合性的另一个重要手段。如果说分布式消息队列通过消息对象分解系统耦合性，不同子系统处理同一个消息；那么分布式服务则通过接口分解系统耦合性，不同子系统通过相同的接口描述进行服务调用。

回顾网站架构发展历程，网站在由小到大的演化过程中，表现为整个网站是由单一应用系统逐步膨胀发展变化而来，随着网站功能的日益复杂，网站应用系统会逐渐成为一个巨无霸，如图 7.3 所示。一个应用中聚合了大量的应用和服务组件，这个巨无霸给整个网站的开发、维护、部署都带来了巨大的麻烦。

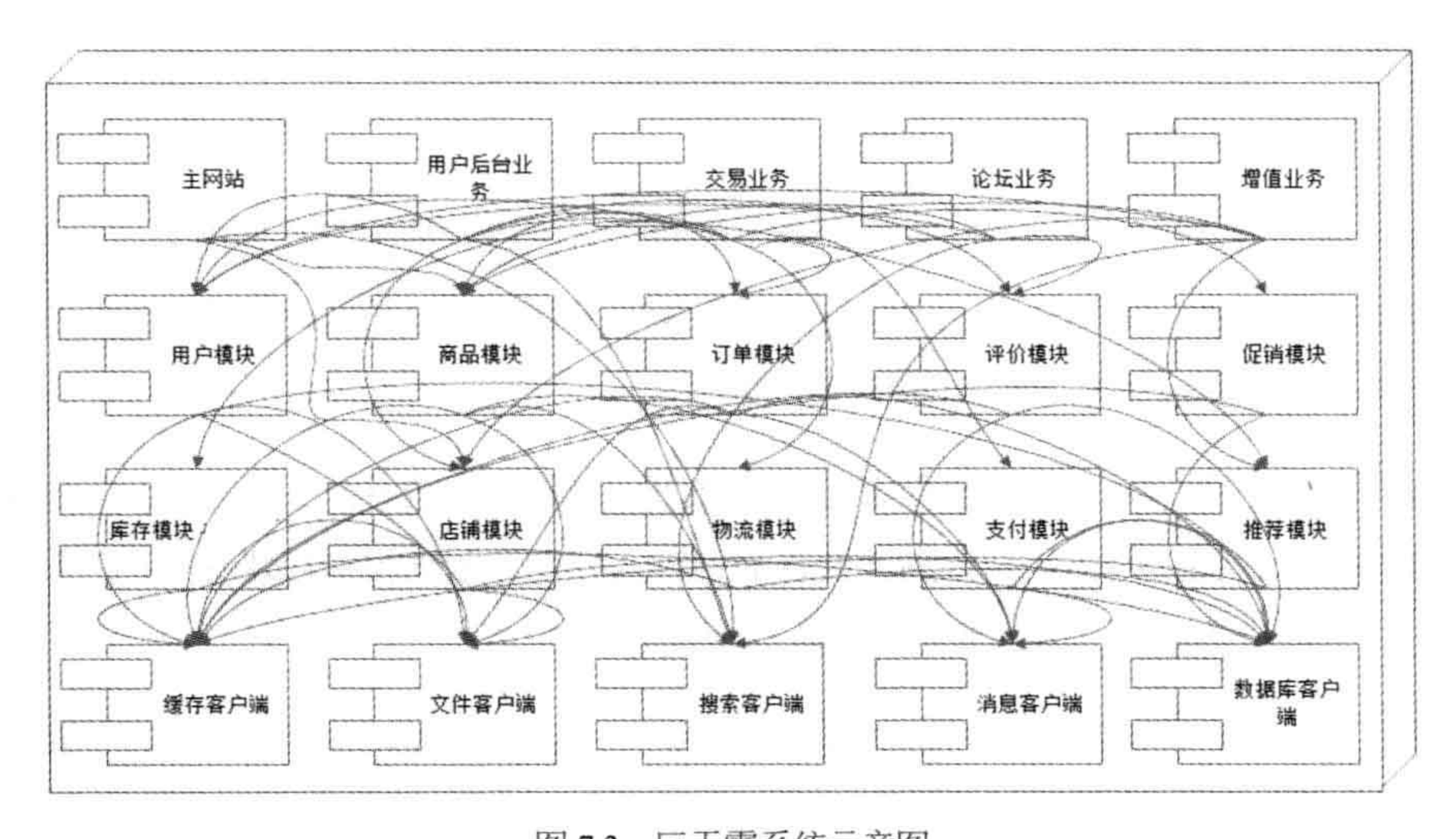

图 7.3 巨无霸系统示意图

巨无霸应用系统带来如下几点问题。

1. 编译、部署困难：对于网站开发工程师而言，打包构建一个巨型应用是一件痛苦的事情，也许只是修改了一行代码，输入 build 命令后，抽完一支烟，回来一看，还在 building；又去喝了一杯水，回来一看，还在 building；又去了一次厕所，回来一看，还在 building；好不容易 build 结束，一看编译失败，还得重来……

2. 代码分支管理困难：复用的代码模块由多个团队共同维护修改，代码 merge 的时候总会发生冲突。代码 merge 一般发生在网站发布的时候，经常和发布过程中出现的其他

问题互相纠结在一起，顾此失彼，导致每次发布都要拖到半夜三更。

3. 数据库连接耗尽：巨型的应用、大量的访问，必然需要将这个应用部署在一个大规模的服务器集群上，应用与数据库的连接通常使用数据库连接池，以每个应用 10 个连接计，一个数百台服务器集群的应用将需要在数据库上创建数千个连接。数据库服务器上，每个连接都会占用一些昂贵的系统资源，以至于数据库缺乏足够的系统资源进行一般的数据操作。

4. 新增业务困难：想要在一个已经如乱麻般的系统中增加新业务，维护旧功能，难度可想而知：一脚踩进去，发现全都是雷，什么都不敢碰。许多新工程师来公司半年了，还是不能接手业务，因为不知道水有多深。于是就出现这种怪现象：熟悉网站产品的“老人”忙得要死，加班加点干活；不熟悉网站产品的新人一帮忙就出乱，跟着加班加点；整个公司热火朝天，加班加点，却还是经常出故障，新产品迟迟不能上线。

解决方案就是拆分，将模块独立部署，降低系统耦合性。拆分可以分为纵向拆分和横向拆分两种。

纵向拆分：将一个大应用拆分为多个小应用，如果新增业务较为独立，那么就直接将其设计部署为一个独立的 Web 应用系统。

横向拆分：将复用的业务拆分出来，独立部署为分布式服务，新增业务只需要调用这些分布式服务，不需要依赖具体的模块代码，即可快速搭建一个应用系统，而模块内业务逻辑变化的时候，只要接口保持一致就不会影响业务程序和其他模块。如图 7.4 所示。

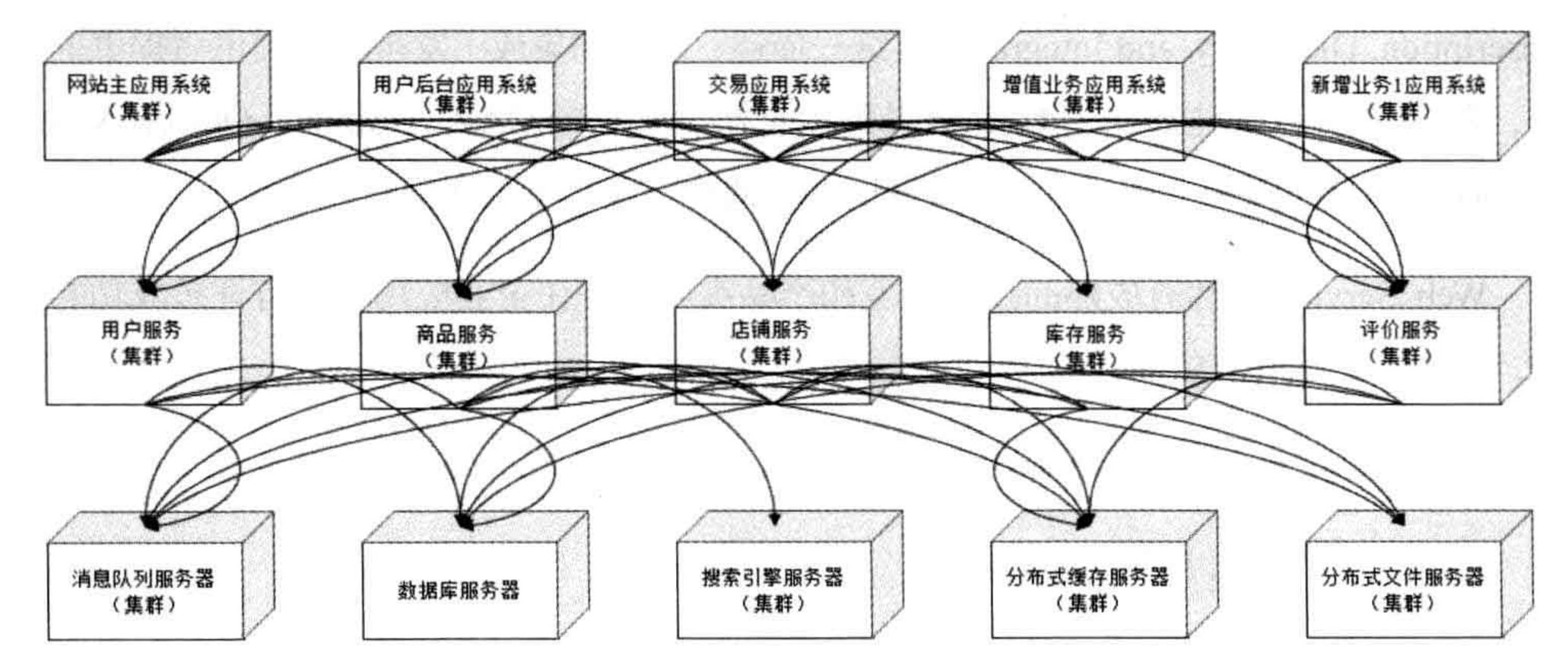

图 7.4 业务及模块拆分独立部署的分布式服务架构

纵向拆分相对较为简单，通过梳理业务，将较少相关的业务剥离，使其成为独立的 Web 应用。而对于横向拆分，不但需要识别可复用的业务，设计服务接口，规范服务依赖关系，还需要一个完善的分布式服务管理框架。

7.3.1 Web Service 与企业级分布式服务

Web Service 曾经是企业应用系统开发领域最时髦的词汇之一，用以整合异构系统及构建分布式系统。Web Service 原理架构如图 7.5 所示。

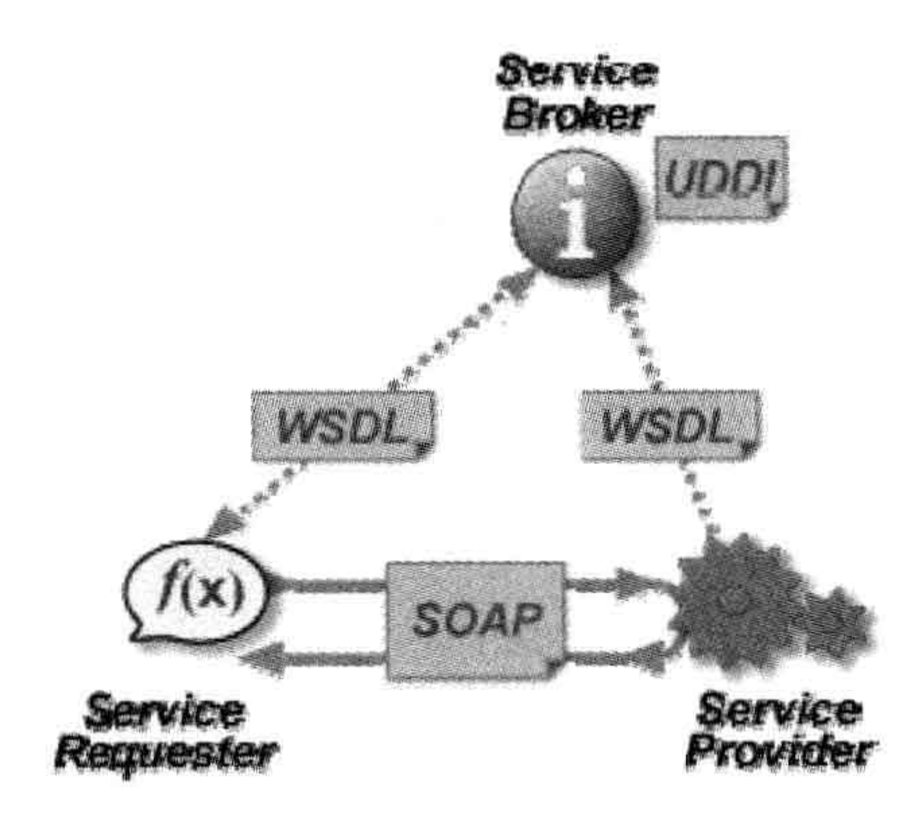

图 7.5 WebService 架构原理

（图片来源：http://zh.wikipedia.org/zh/Web%E6%9C%8D%E5%8A%A1）

服务提供者通过 WSDL（Web Services Description Language，Web 服务描述语言）向注册中心（Service Broker）描述自身提供的服务接口属性，注册中心使用 UDDI（Universal Description, Discovery, and Integration，统一描述、发现和集成）发布服务提供者提供的服务，服务请求者从注册中心检索到服务信息后，通过 SOAP （Simple Object Access Protocol，简单对象访问协议 ）和服务提供者通信，使用相关服务。

Web Service 虽然有成熟的技术规范和产品实现，并在企业应用领域有许多成功的案例，但也有如下固有的缺点。

1. 臃肿的注册与发现机制。
2. 低效的 XML 序列化手段。
3. 开销相对较高的 HTTP 远程通信。

4. 复杂的部署与维护手段。

这些问题导致 Web Service 难以满足大型网站对系统高性能、高可用、易部署、易维护的要求。

7.3.2 大型网站分布式服务的需求与特点

对于大型网站，除了 Web Service 所提供的服务注册与发现，服务调用等标准功能，还需要分布式服务框架能够支持如下特性。

负载均衡

对热门服务，比如登录服务或者商品服务，访问量非常大，服务需要部署在一个集群上。分布式服务框架要能够支持服务请求者使用可配置的负载均衡算法访问服务，使服务提供者集群实现负载均衡。

失效转移

可复用的服务通常会被多个应用调用，一旦该服务不可用，就会影响到很多应用的可用性。因此对于大型网站的分布式服务而言，即使是很少访问的简单服务，也需要集群部署，分布式服务框架支持服务提供者的失效转移机制，当某个服务实例不可用，就将访问切换到其他服务实例上，以实现服务整体高可用。

高效的远程通信

对于大型网站，核心服务每天的调用次数会达到数以亿计，如果没有高效的远程通信手段，服务调用会成为整个系统性能的瓶颈。

整合异构系统

由于历史发展和组织分割，网站服务可能会使用不同的语言开发并部署于不同的平台，分布式服务框架需要整合这些异构的系统。

对应用最少侵入

网站技术是为业务服务的，是否使用分布式服务需要根据业务发展规划，分布式服务也需要渐进式的演化，甚至会出现反复，即使用了分布式服务后又退回到集中式部署，

分布式服务框架需要支持这种渐进式演化和反复。当然服务模块本身需要支持可集中式部署，也可分布式部署。

版本管理

为了应对快速变化的需求，服务升级不可避免，如果仅仅是服务内部实现逻辑升级，那么这种升级对服务请求者而言是透明的，无需关注。但如果服务的访问接口也发生了变化，就需要服务请求者和服务提供者同时升级才不会导致服务调用失败。企业应用系统可以申请停机维护，同时升级接口。但是网站服务不可能中断，因此分布式服务框架需要支持服务多版本发布，服务提供者先升级接口发布新版本的服务，并同时提供旧版本的服务供请求者调用，当请求者调用接口升级后才可以关闭旧版本服务。

实时监控

对于网站应用而言，没有监控的服务是不可能实现高可用的。分布式服务框架还需要监控服务提供者和调用者的各项指标，提供运维和运营支持。

7.3.3 分布式服务框架设计

大型网站需要更简单更高效的分布式服务框架构建其 SOA（Service Oriented Architecture 面向服务的体系架构）。据称 Facebook 利用 Thrift（一个开源的远程服务调用框架）管理其分布式服务，服务的注册、发现及调用都通过 Thrift 完成，但对于一个大型网站可以使用的分布式服务框架，仅有 Thrift 还远远不够，遗憾的是，Facebook 没有开源其基于 Thrift 的分布式服务框架。目前国内有较多成功实施案例的开源分布式服务框架是阿里巴巴的 Dubbo（http://code.alibabatech.com/wiki/display/dubbo/Home/）。

我们以阿里巴巴分布式开源框架 Dubbo 为例，分析其架构设计，如图 7.6 所示。

服务消费者程序通过服务接口使用服务，而服务接口通过代理加载具体服务，具体服务可以是本地的代码模块，也可以是远程的服务，因此对应用较少侵入：应用程序只需要调用服务接口，服务框架根据配置自动调用本地或远程实现。

服务框架客户端模块通过服务注册中心加载服务提供者列表（服务提供者启动后自动向服务注册中心注册自己可提供的服务接口列表），查找需要的服务接口，并根据配置的负载均衡策略将服务调用请求发送到某台服务提供者服务器。如果服务调用失败，客

户端模块会自动从服务提供者列表选择一个可提供同样服务的另一台服务器重新请求服务，实现服务的自动失效转移，保证服务高可用。

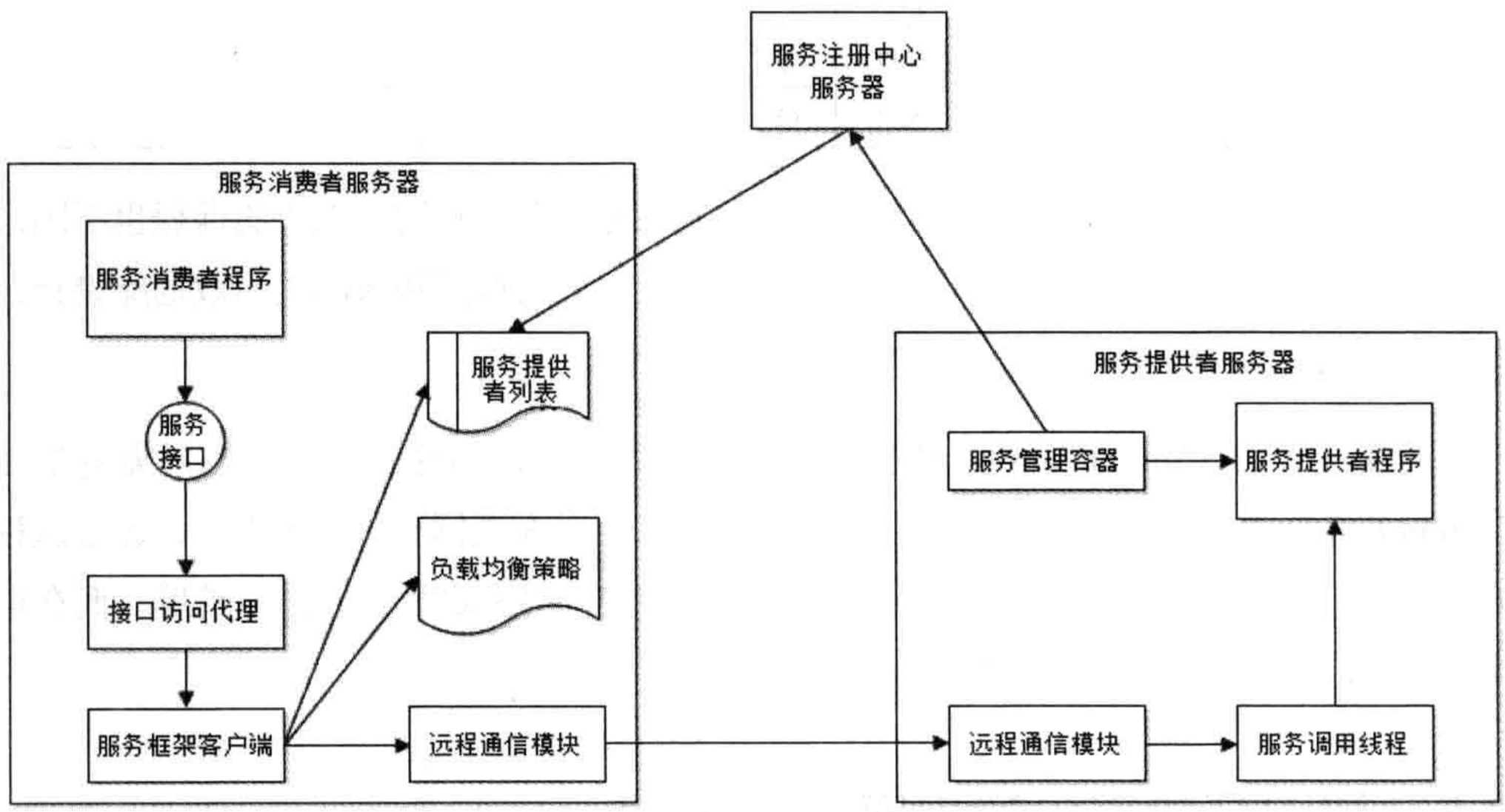

图 7.6 分布式服务框架 Dubbo 的架构原理

Dubbo 的远程服务通信模块支持多种通信协议和数据序列化协议，使用 NIO 通信框架，具有较高的网络通信性能。

7.4 可扩展的数据结构

传统的关系数据库为了保证关系运算（通过 SQL 语句）的正确性，在设计数据库表结构的时候，就需要指定表的 schema——字段名称，数据类型等，并要遵循特定的设计范式。这些规范带来的一个问题就是僵硬的数据结构难以面对需求变更带来的挑战，有些应用系统设计者通过预先设计一些冗余字段来应对，不过显然这是一种糟糕的数据库设计。

那么有没有办法能够做到可扩展的数据结构设计呢？无需修改表结构就可以新增字段呢？许多 NoSQL 数据库使用的 ColumnFamily（列族）设计就是一个解决方案。ColumnFamily 最早在 Google 的 Bigtable 中使用，这是一种面向列族的稀疏矩阵存储格式，如表 7.1 所示。

表 7.1 ColumnFamily 数据存储格式

Key	联系方式（Column Family）			课程成绩（Column Family）		
001	Weibo：li_zhihui	分机：233		历史：85		地理：77
002		分机：809	QQ：523		英语：78	地理：87
003		分机：523	QQ：908	历史：91	英语：88	

这是一个学生的基本信息表，不同学生的联系方式各不相同，选修的课程也不同，而且在将来会有更多联系方式和课程加入到这张表，如果按照传统的关系数据库设计，无论提前预设多少冗余字段都会捉襟见肘，疲于应付。

而使用支持 ColumnFamily 结构的 NoSQL 数据库，创建表的时候，只需要指定 ColumnFamily 的名字，无需指定字段（Column），可以在数据写入时再指定，通过这种方式，数据表可以包含数百万的字段，使得应用程序的数据结构可以随意扩展。而在查询时，可以通过指定任意字段名称和值进行查询。

7.5 利用开放平台建设网站生态圈

网站的价值在于为他的用户创造价值，淘宝的价值在于为人们创造交易的平台；QQ 的价值在于为人们创造交流的平台；新浪微博的价值在于为人们创造表达自我的平台。只有用户得到了他们想要的价值，他们才愿意使用网站的服务，网站的存在才有意义。但是淘宝有上千万卖家和数亿买家，光靠淘宝一个公司不可能满足所有用户的需求，同样，腾讯、新浪微博也无法面面俱到照顾好如此庞大的用户群。

另一方面，用户却不需要为网站提供的价值而买单。没有人需要为自己在 QQ 上聊天，在淘宝上购物，在新浪发微博而付费。网站必须提供更多的增值服务才能赚钱。比如，QQ 可以卖各种钻石会员服务，淘宝可以出卖商品排名赚钱，新浪微博靠植入广告也能赚点钱。根据长尾效应，这些增值服务的数量越是庞大，种类越是繁多，盈利也就越多。同样，一个网站自己能够开发出的增值服务也是有限的。

大型网站为了更好地服务自己的用户，开发更多的增值服务，会把网站内部的服务封装成一些调用接口开放出去，供外部的第三方开发者使用，这个提供开放接口的平台被称作开放平台。第三方开发者利用这些开放的接口开发应用程序（APP）或者网站，为

更多的用户提供价值。网站、用户、第三方开发者互相依赖，形成一个网站的生态圈，既为用户提供更多的价值，也提高了网站和第三方开发者的竞争能力和盈利能力。

目前百度、淘宝、腾讯等国内互联网巨头都建设有自己的开放平台，力图利用自己庞大的用户群吸引第三方开发者，打造一个更加庞大的航母战斗群，在市场竞争中呼风唤雨，立于不败之地。

开放平台是网站内部和外部交互的接口，外部需要面对众多的第三方开发者，内部需要面对网站内诸多的业务服务。虽然每个网站的业务场景和需求都各不相同，但是开放平台的架构设计却大同小异，如图 7.7 所示。

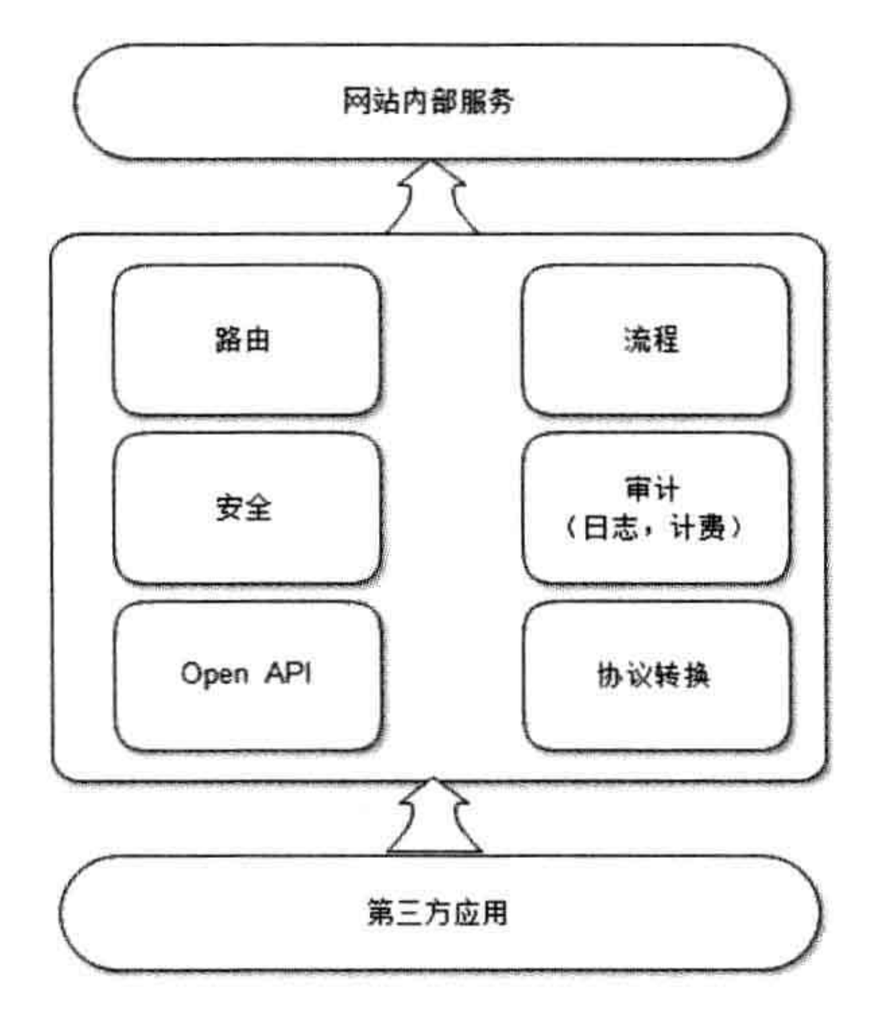

图 7.7 开放平台架构原理

API 接口：是开放平台暴露给开发者使用的一组 API，其形式可以是 RESTful、WebService、RPC 等各种形式。

协议转换：将各种 API 输入转换成内部服务可以识别的形式，并将内部服务的返回封装成 API 的格式。

安全：除了一般应用需要的身份识别、权限控制等安全手段，开放平台还需要分级的访问带宽限制，保证平台资源被第三方应用公平合理使用，也保护网站内部服务不会被外部应用拖垮。

审计：记录第三方应用的访问情况，并进行监控、计费等。

路由：将开放平台的各种访问路由映射到具体的内部服务。

流程：将一组离散的服务组织成一个上下文相关的新服务，隐藏服务细节，提供统一接口供开发者调用。

7.6 小结

网站通过不断试错，在残酷的市场中寻找自己的竞争优势，持续地推出新功能，发现达不到预期，就立马下线。所以我们看到网站总是不停地推出新功能，发布新产品。打开 Google 首页的“更多”链接，Google 产品分门别类一大堆，这还只是 Google 重点推广的产品中的一小部分。这些走马灯般出现的产品背后则是网站工程师辛勤的工作和汗水。

既然我们知道网站不停上新产品是其生存的本能，谁能更快更好地推出更多的新产品，谁就活得更滋润，那么工程师就要做好准备应付这种局面。马克思的劳动价值理论告诉我们，产品的内在价值在于劳动的时间，劳动的时间不在于个体付出的劳动时间，而在于行业一般劳动时间，资本家只会为行业一般劳动时间买单，如果你的效率低于行业一般劳动时间，对不起，请你自愿加班。反之，如果你有一个更具有扩展性的网站架构，可以更快速地开发新产品，也许你也享受不了只上半天班的福利，但是至少在这个全行业加班的互联网领域，你能够按时下班，陪陪家人，看看星星。

8 固若金汤：网站的安全架构

从互联网诞生起，安全威胁就一直伴随着网站的发展，各种 Web 攻击和信息泄露也从未停止。2011 年中国互联网领域爆出两桩比较大的安全事故，一桩是新浪微博遭 XSS 攻击，另一桩是以 CSDN 为代表的多个网站泄露用户密码和个人信息。特别是后者，因为影响人群广泛，部分受影响网站涉及用户实体资产和交易安全，一时成为舆论焦点。

让我们先回顾一下这两起事故。

2011 年 6 月 28 日，许多微博用户发现自己“中毒”，自动关注了一个叫 hellosamy 的用户，并发布含有病毒的微博，粉丝点击后微博再度扩散，短时间内大量用户中招，数小时后新浪微博修复漏洞。

2011 年 12 月，网上有人发布消息称 CSDN 网站 600 万用户资料和密码被泄露，很快该消息得到 CSDN 官方承认，紧接着，天涯社区、人人网等多个重要网站被报告泄露用户数据。

那么新浪微博是如何被攻击的？CSDN 的密码为何会泄露？如何防护网站免遭攻击，保护好用户的敏感信息呢？有没有百毒不侵、固若金汤的网站呢？

8.1 道高一尺魔高一丈的网站应用攻击与防御

攻击新浪微博的手段被称作 XSS 攻击，它和 SQL 注入攻击构成网站应用攻击最主要的两种手段，全球大约 70%的 Web 应用攻击都来自 XSS 攻击和 SQL 注入攻击。此外，常用的 Web 应用还包括 CSRF、Session 劫持等手段。

8.1.1 XSS 攻击

XSS 攻击即跨站点脚本攻击（Cross Site Script），指黑客通过篡改网页，注入恶意 HTML 脚本，在用户浏览网页时，控制用户浏览器进行恶意操作的一种攻击方式。

常见的 XSS 攻击类型有两种，一种是反射型，攻击者诱使用户点击一个嵌入恶意脚本的链接，达到攻击的目的，如图 8.1 所示。上文提到的新浪微博攻击就是一种反射型 XSS 攻击。攻击者发布的微博中有一个含有恶意脚本的 URL（在实际应用中，该脚本在攻击者自己的服务器 www.2kt.cn，URL 中包含脚本的链接），用户点击该 URL，脚本会自动关注攻击者的新浪微博 ID，发布含有恶意脚本 URL 的微博，攻击就被扩散了。

这次攻击还只是一次恶作剧而已，现实中，攻击者可以采用 XSS 攻击，偷取用户 Cookie、密码等重要数据，进而伪造交易、盗窃用户财产、窃取情报。

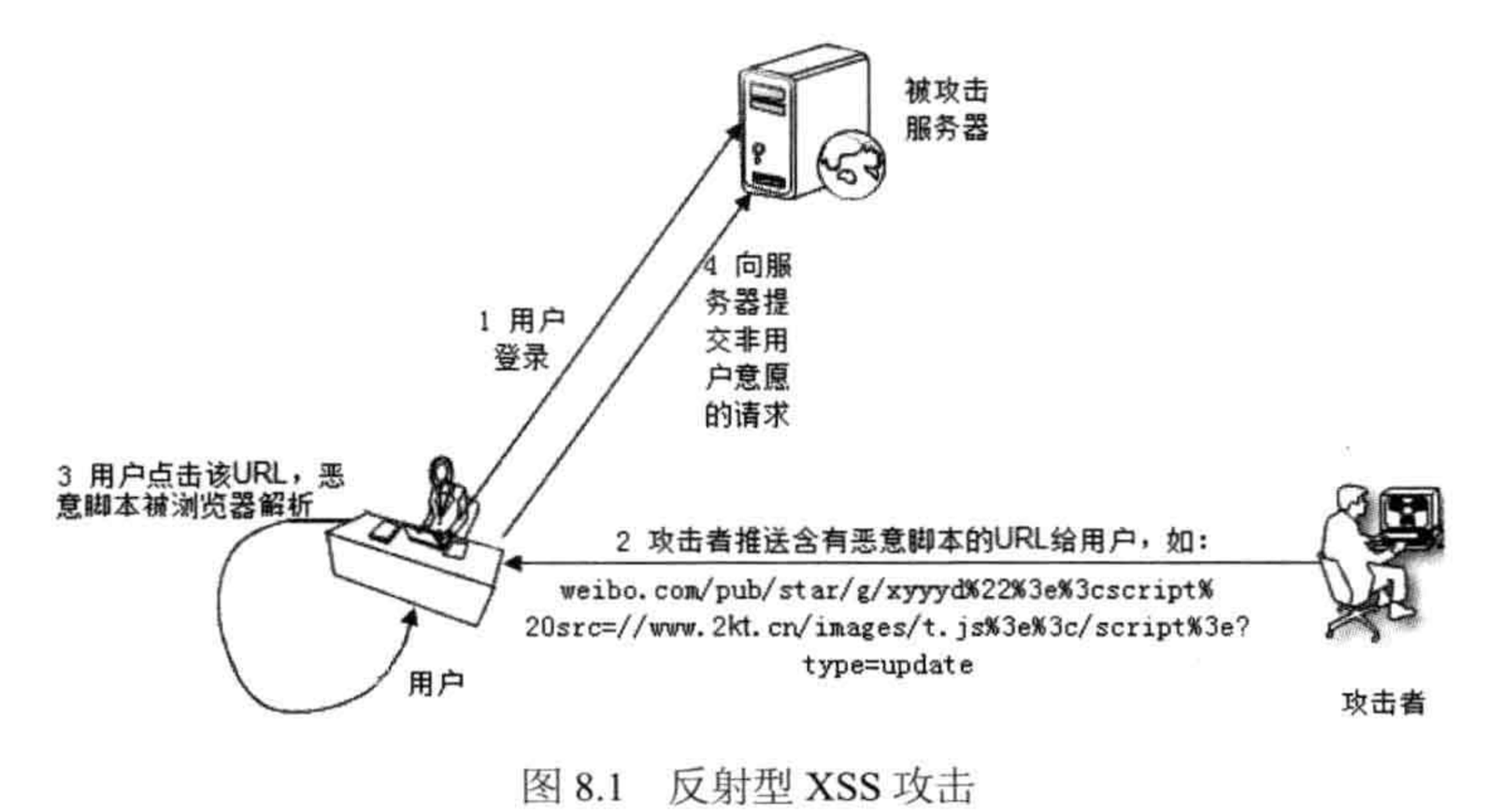

图 8.1 反射型 XSS 攻击

另外一种 XSS 攻击是持久型 XSS 攻击，黑客提交含有恶意脚本的请求，保存在被攻击的 Web 站点的数据库中，用户浏览网页时，恶意脚本被包含在正常页面中，达到攻击的目的，如图 8.2 所示。此种攻击经常使用在论坛，博客等 Web 应用中。

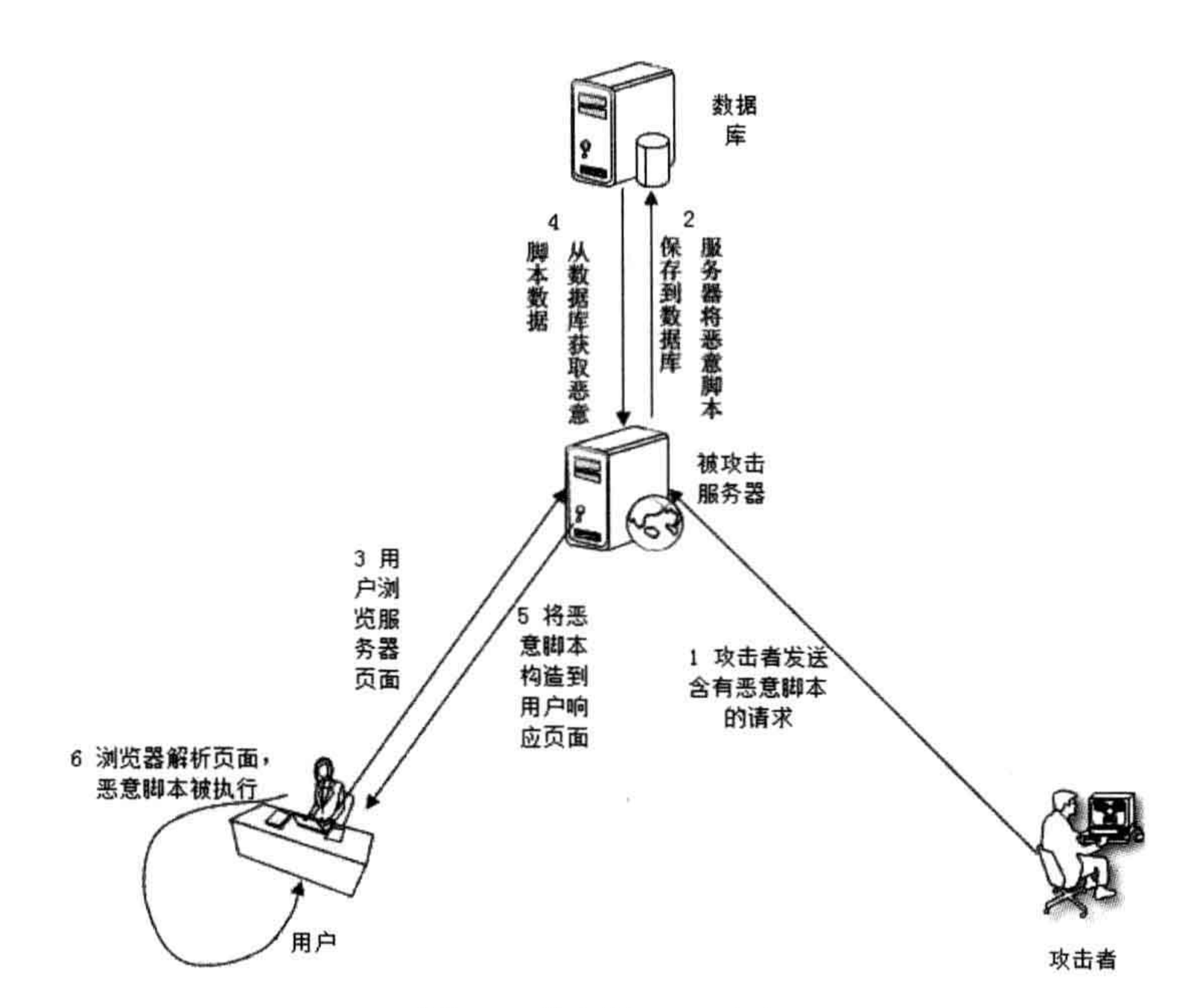

图 8.2 持久型 XSS 攻击

XSS 攻击相对而言是一种“古老”的攻击手段，却又历久弥新，不断变化出新的攻击花样，许多以前认为不可能用来攻击的漏洞也逐渐被攻击者利用。因此 XSS 防攻击也是非常复杂的。主要手段有如下两种。

消毒

XSS 攻击者一般都是通过在请求中嵌入恶意脚本达到攻击的目的，这些脚本是一般用户输入中不使用的，如果进行过滤和消毒处理，即对某些 html 危险字符转义，如“>”转义为“>”、“<”转义为“<”等，就可以防止大部分攻击。为了避免对不必要的内容错误转义，如“3<5”中的“<”需要进行文本匹配后再转义，如“<img src=”这样的上下文中的“<”才转义。事实上，消毒几乎是所有网站最必备的 XSS 防攻击手段。

HttpOnly

最早由微软提出，即浏览器禁止页面 JavaScript 访问带有 HttpOnly 属性的 Cookie。HttpOnly 并不是直接对抗 XSS 攻击的，而是防止 XSS 攻击者窃取 Cookie。对于存放敏感信息的 Cookie，如用户认证信息等，可通过对该 Cookie 添加 HttpOnly 属性，避免被攻击脚本窃取。

8.1.2 注入攻击

注入攻击主要有两种形式，SQL 注入攻击和 OS 注入攻击。SQL 注入攻击的原理如图 8.3 所示。攻击者在 HTTP 请求中注入恶意 SQL 命令（drop table users;），服务器用请求参数构造数据库 SQL 命令时，恶意 SQL 被一起构造，并在数据库中执行。

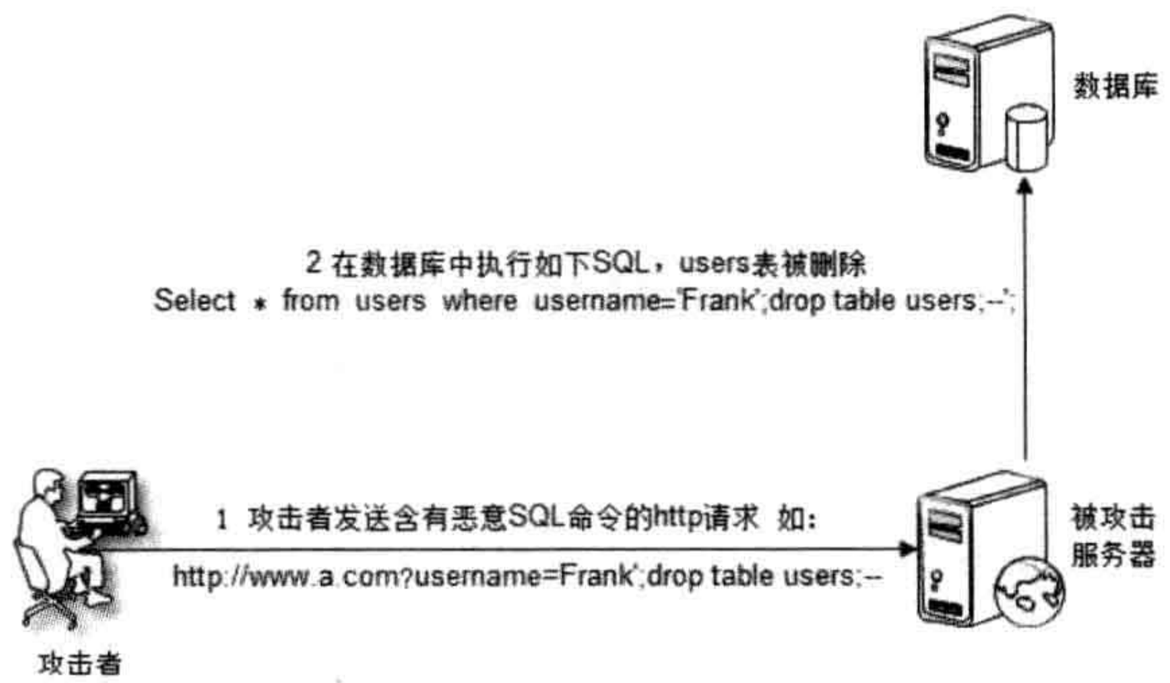

图 8.3 SQL 注入攻击

SQL 注入攻击需要攻击者对数据库结构有所了解才能进行，攻击者获取数据库表结构信息的手段有如下几种。

开源

如果网站采用开源软件搭建，如用 Discuz!搭建论坛网站，那么网站数据库结构就是公开的，攻击者可以直接获得。

错误回显

如果网站开启错误回显，即服务器内部 500 错误会显示到浏览器上。攻击者通过故意构造非法参数，使服务端异常信息输出到浏览器端，为攻击猜测数据库表结构提供了便利。

盲注

网站关闭错误回显，攻击者根据页面变化情况判断 SQL 语句的执行情况，据此猜测数据库表结构，此种方式攻击难度较大。

防御 SQL 注入攻击首先要避免被攻击者猜测到表名等数据库表结构信息，此外还可以采用如下方式。

消毒

和防 XSS 攻击一样，请求参数消毒是一种比较简单粗暴又有效的手段。通过正则匹配，过滤请求数据中可能注入的 SQL，如“drop table”、“\b(?:update\b.*?\bset |delete\b\W*?\bfrom)\b”等。

参数绑定

使用预编译手段，绑定参数是最好的防 SQL 注入方法。目前许多数据访问层框架，如 IBatis，Hibernate 等，都实现 SQL 预编译和参数绑定，攻击者的恶意 SQL 会被当做 SQL 的参数，而不是 SQL 命令被执行。

除了 SQL 注入，攻击者还根据具体应用，注入 OS 命令、编程语言代码等，利用程序漏洞，达到攻击目的。

8.1.3 CSRF 攻击

CSRF（Cross Site Request Forgery，跨站点请求伪造），攻击者通过跨站请求，以合法用户的身份进行非法操作，如转账交易、发表评论等，如图 8.4 所示。CSRF 的主要手法是利用跨站请求，在用户不知情的情况下，以用户的身份伪造请求。其核心是利用了浏览器 Cookie 或服务器 Session 策略，盗取用户身份。

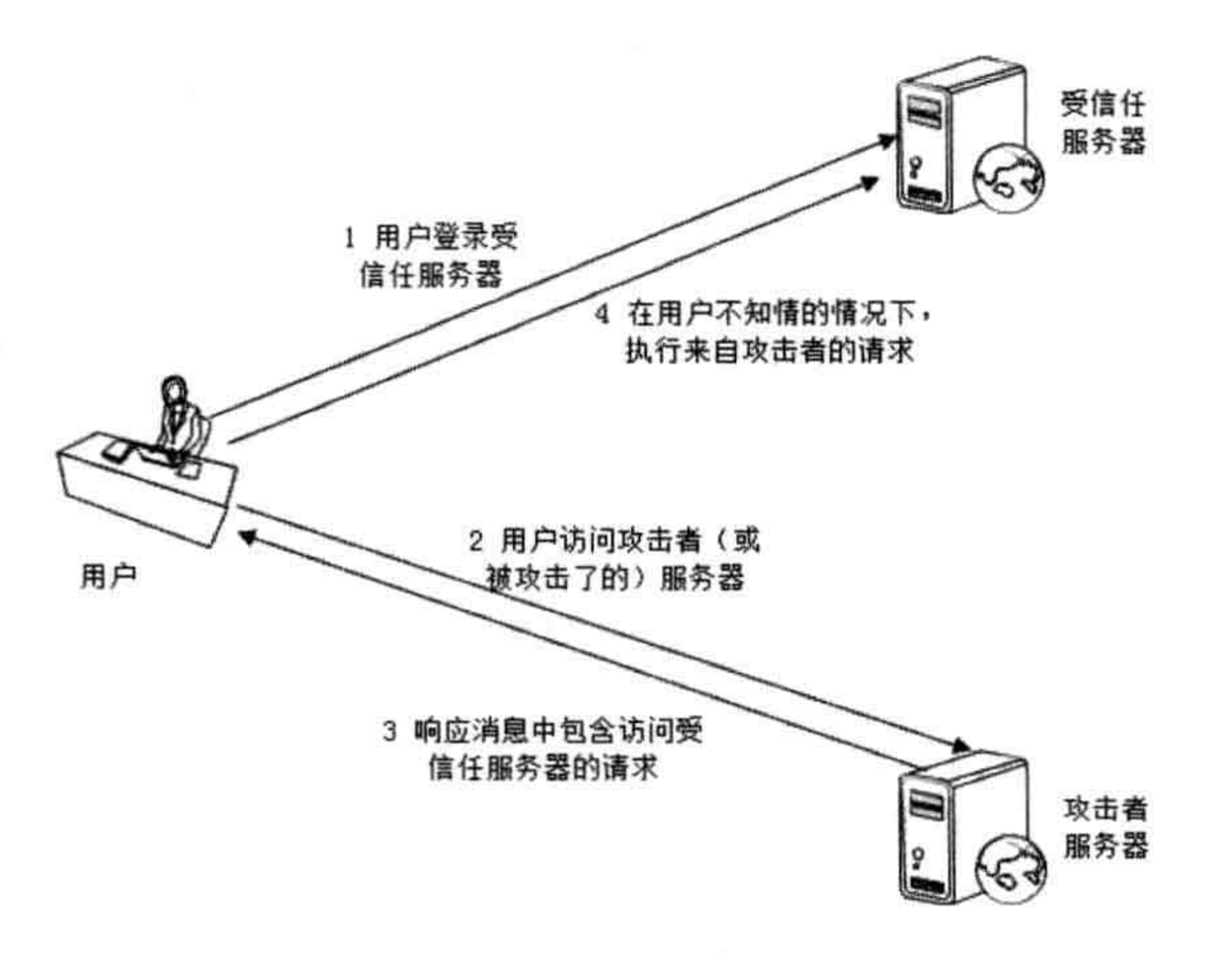

图 8.4 CSRF 攻击

相应地，CSRF 的防御手段主要是识别请求者身份。主要有下面几种方法。

表单 Token

CSRF 是一个伪造用户请求的操作，所以需要构造用户请求的所有参数才可以。表单 Token 通过在请求参数中增加随机数的办法来阻止攻击者获得所有请求参数：在页面表单中增加一个随机数作为 Token，每次响应页面的 Token 都不相同，从正常页面提交的请求会包含该 Token 值，而伪造的请求无法获得该值，服务器检查请求参数中 Token 的值是否存在并且正确以确定请求提交者是否合法。

验证码

相对说来，验证码则更加简单有效，即请求提交时，需要用户输入验证码，以避免在用户不知情的情况下被攻击者伪造请求。但是输入验证码是一个糟糕的用户体验，所以请在必要时使用，如支付交易等关键页面。

Referer check

HTTP 请求头的 Referer 域中记录着请求来源，可通过检查请求来源，验证其是否合法。很多网站使用这个功能实现图片防盗链（如果图片访问的页面来源不是来自自己网站的网页就拒绝）。

8.1.4 其他攻击和漏洞

除了上面提到的常见攻击，还有一些漏洞也常被黑客利用。

Error Code

也称作错误回显，许多 Web 服务器默认是打开异常信息输出的，即服务器端未处理的异常堆栈信息会直接输出到客户端浏览器，这种方式虽然对程序调试和错误报告有好处，但同时也给黑客造成可乘之机。通过故意制造非法输入，使系统运行时出错，获得异常信息，从而寻找系统漏洞进行攻击。防御手段也很简单，通过配置 Web 服务器参数，跳转 500 页面（HTTP 响应码 500 表示服务器内部错误）到专门的错误页面即可， Web 应用常用的 MVC 框架也有这个功能。

HTML 注释

为调试程序方便或其他不恰当的原因，有时程序开发人员会在 PHP、JSP 等服务器页

面程序中使用 HTML 注释语法进行程序注释，这些 HTML 注释就会显示在客户端浏览器，给黑客造成攻击便利。程序最终发布前需要进行代码 review 或自动扫描，避免 HTML 注释漏洞。

文件上传

一般网站都会有文件上传功能，设置头像、分享视频、上传附件等。如果上传的是可执行的程序，并通过该程序获得服务器端命令执行能力，那么攻击者几乎可以在服务器上为所欲为，并以此为跳板攻击集群环境的其他机器。最有效的防御手段是设置上传文件白名单，只允许上传可靠的文件类型。此外还可以修改文件名、使用专门的存储等手段，保护服务器免受上传文件攻击。

路径遍历

攻击者在请求的 URL 中使用相对路径，遍历系统未开放的目录和文件。防御方法主要是将 JS、CSS 等资源文件部署在独立服务器、使用独立域名，其他文件不使用静态 URL 访问，动态参数不包含文件路径信息。

8.1.5 Web 应用防火墙

网站面临的安全问题复杂多样，各种攻击手段日新月异，新型漏洞不断被报告。如果有一款产品能够统一拦截请求，过滤恶意参数，自动消毒、添加 Token，并且能够根据最新攻击和漏洞情报，不断升级对策，处理掉大多数令人头痛的网站攻击，就是一件很美妙的事了。

非常幸运，真的有这样的产品——ModSecurity。

ModSecurity 是一个开源的 Web 应用防火墙，探测攻击并保护 Web 应用程序，既可以嵌入到 Web 应用服务器中，也可以作为一个独立的应用程序启动。ModSecurity 最早只是 Apache 的一个模块，现在已经有 Java、.NET 多个版本，并支持 Nginx。

ModSecurity 采用处理逻辑与攻击规则集合分离的架构模式。处理逻辑（执行引擎）负责请求和响应的拦截过滤，规则加载执行等功能。而攻击规则集合则负责描述对具体攻击的规则定义、模式识别、防御策略等功能（可以通过文本方式进行描述）。处理逻辑比较稳定，规则集合需要不断针对漏洞进行升级，这是一种可扩展的架构设计，如图 8.5 所示。

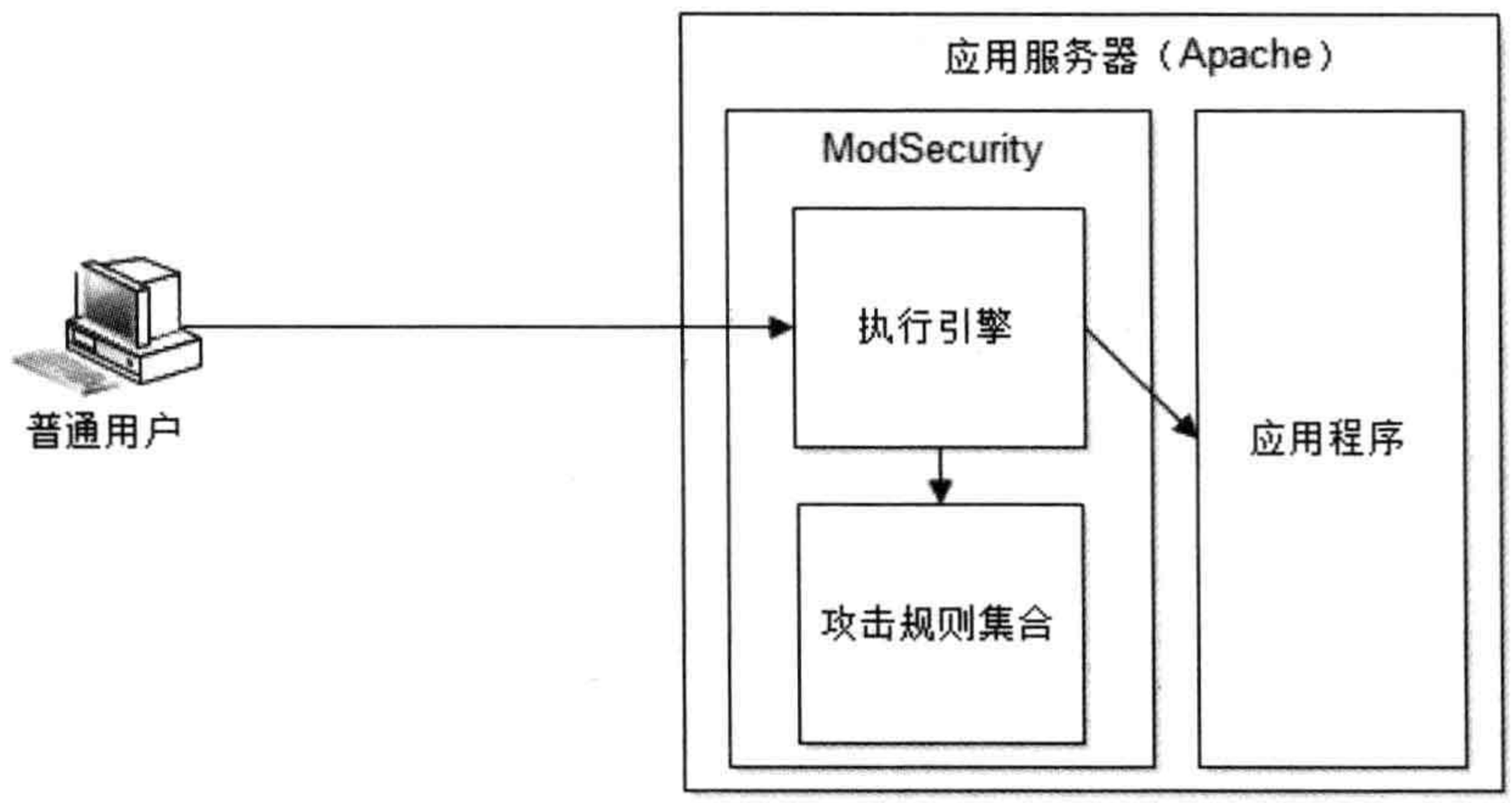

图 8.5 ModSecurity 架构原理

除了开源的 ModeSecurity，还有一些商业产品也实现 Web 应用防火墙功能，如 NEC 的 SiteShell。

8.1.6 网站安全漏洞扫描

和计算机安全漏洞扫描一样，网站也需要安全漏洞扫描。

网站安全漏洞扫描工具是根据内置规则，构造具有攻击性的 URL 请求，模拟黑客攻击行为，用以发现网站安全漏洞的工具。许多大型网站的安全团队都有自己开发的漏洞扫描工具，不定期地对网站的服务器进行扫描，查漏补缺。市场上也有很多商用的网站安全漏洞扫描平台。

8.2 信息加密技术及密钥安全管理

2011 年 12 月被曝的 CSDN 密码泄露事故中，网站安全措施不力，导致用户数据库被黑客“拖库”并不稀奇，令人错愕的是数据库中的用户密码居然是明文保存，导致密码泄露，成为地下黑市交易的商品。

通常，为了保护网站的敏感数据，应用需要对这些信息进行加密处理，信息加密技术可分为三类：单项散列加密、对称加密和非对称加密。

8.2.1 单向散列加密

单向散列加密是指通过对不同输入长度的信息进行散列计算，得到固定长度的输出，这个散列计算过程是单向的，即不能对固定长度的输出进行计算从而获得输入信息，如图 8.6 所示。

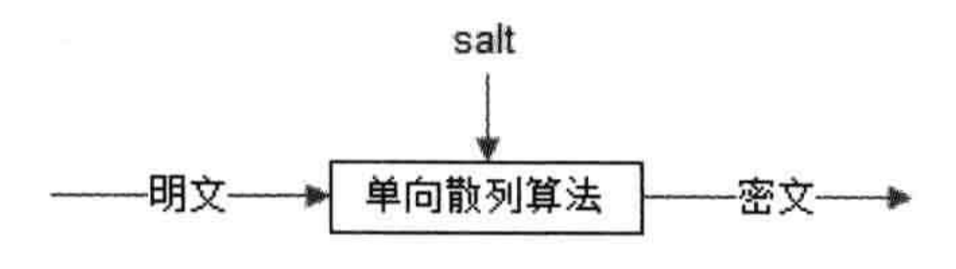

图 8.6 单向散列加密

利用单向散列加密的这个特性，可以进行密码加密保存，即用户注册时输入的密码不直接保存到数据库，而是对密码进行单向散列加密，将密文存入数据库，用户登录时，进行密码验证，同样计算得到输入密码的密文，并和数据库中的密文比较，如果一致，则密码验证成功，具体过程如图 8.7 所示。

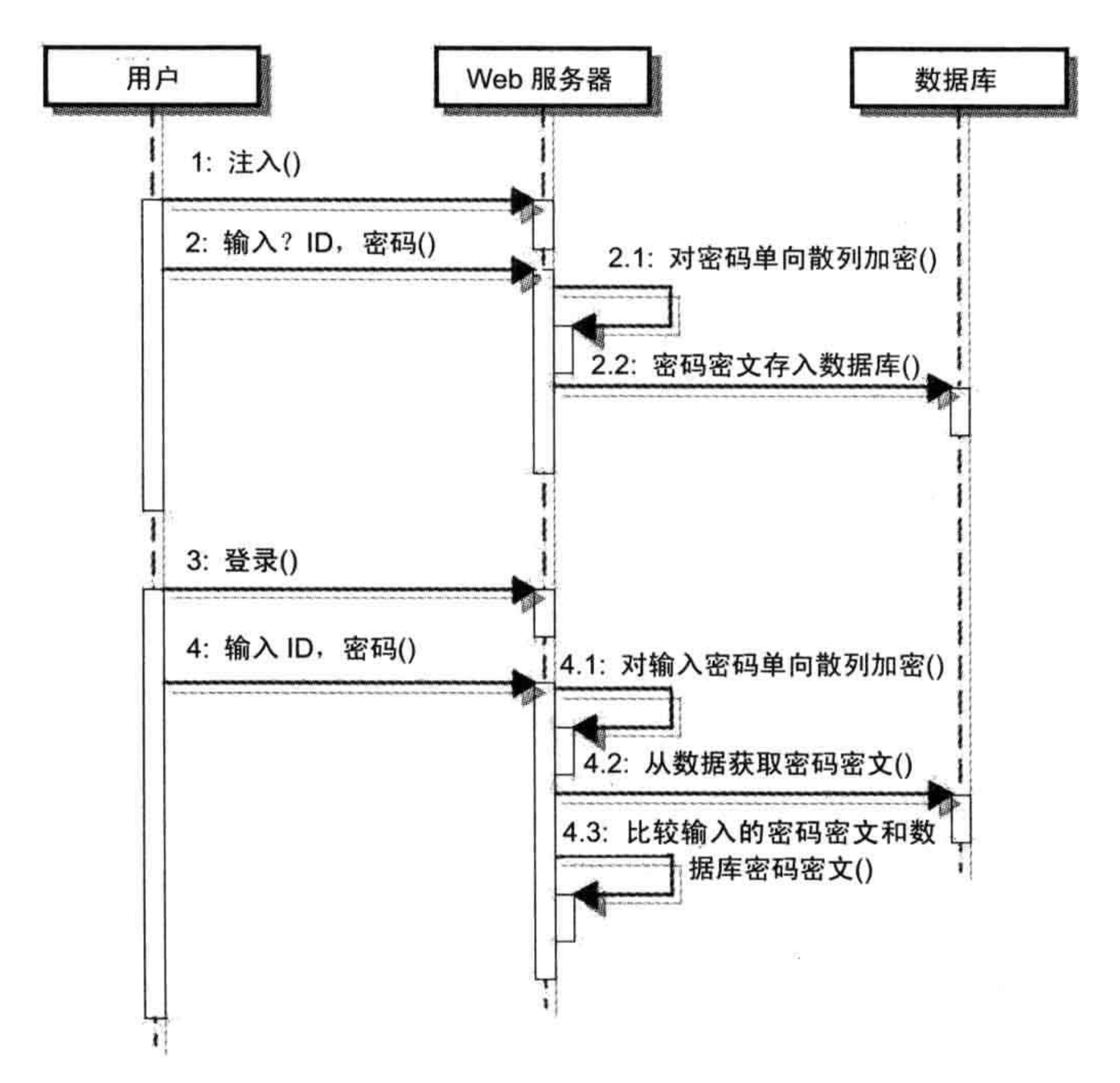

图 8.7 密码保存与验证

这样保存在数据库中的是用户输入的密码的密文，而且不可逆地计算得到密码的明

文，因此即使数据库被“拖库”，也不会泄露用户的密码信息。

虽然不能通过算法将单向散列密文反算得到明文，但是由于人们设置密码具有一定的模式，因此通过彩虹表（人们常用密码和对应的密文关系表）等手段可以进行猜测式破解。

为了加强单向散列计算的安全性，还会给散列算法加点盐（salt），salt 相当于加密的密钥，增加破解的难度。

常用的单向散列算法有 MD5、SHA 等。单向散列算法还有一个特点就是输入的任何微小变化都会导致输出的完全不同，这个特性有时也会被用来生成信息摘要、计算具有高离散程度的随机数等用途。

8.2.2 对称加密

所谓对称加密是指加密和解密使用的密钥是同一个密钥（或者可以互相推算），如图 8.8 所示。

对称加密通常用在信息需要安全交换或存储的场合，如 Cookie 加密、通信加密等。

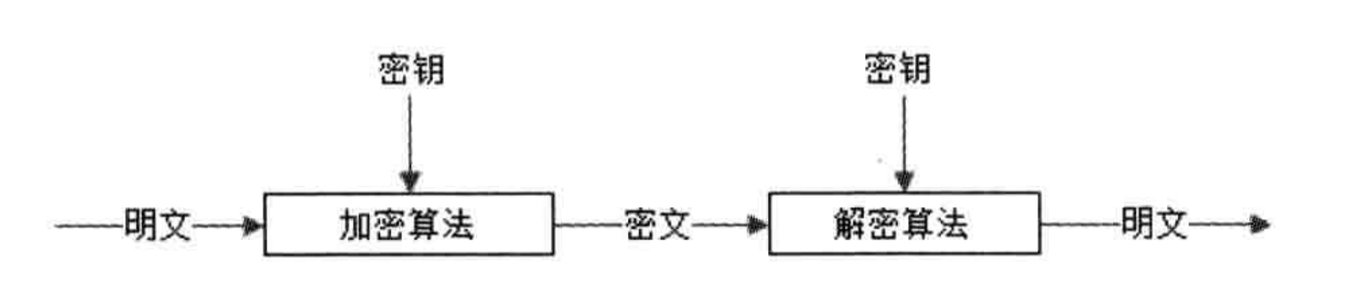

图 8.8 对称加密

对称加密的优点是算法简单，加解密效率高，系统开销小，适合对大量数据加密。缺点是加解密使用同一个密钥，远程通信的情况下如何安全的交换密钥是个难题，如果密钥丢失，那么所有的加密信息也就没有秘密可言了。

常用的对称加密算法有 DES 算发、RC 算法等。对称加密是一种传统加密手段，也是最常用的加密手段，适用于绝大多数需要加密的场合。

8.2.3 非对称加密

不同于对称加密，非对称加密和解密使用的密钥不是同一密钥，其中一个对外界公开，被称作公钥，另一个只有所有者知道，被称作私钥。用公钥加密的信息必须用私钥才能解开，反之，用私钥加密的信息只有用公钥才能解开，如图 8.9 所示。理论上说，不

可能通过公钥计算获得私钥。

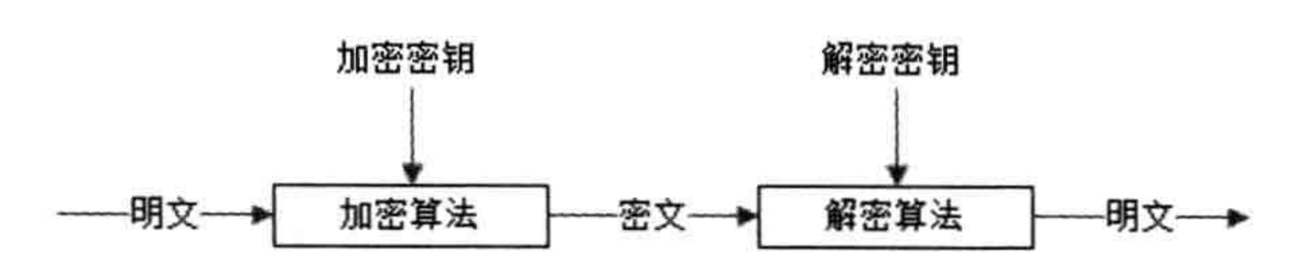

图 8.9 非对称加密

非对称加密技术通常用在信息安全传输，数字签名等场合。

信息发送者 A 通过公开渠道获得信息接收者 B 的公钥，对提交信息进行加密，然后通过非安全传输通道将密文信息发送给 B，B 得到密文信息后，用自己的私钥对信息进行解密，获得原始的明文信息。即使密文信息在传输过程中遭到窃取，窃取者没有解密密钥也无法还原明文。

数字签名的过程则相反，签名者用自己的私钥对信息进行加密，然后发送给对方，接收方用签名者的公钥对信息进行解密，获得原始明文信息，由于私钥只有签名者拥有，因此该信息是不可抵赖的，具有签名的性质。

在实际应用中，常常会混合使用对称加密和非对称加密。先使用非对称加密技术对对称密钥进行安全传输，然后使用对称加密技术进行信息加解密与交换。而有时，对同一个数据两次使用非对称加密，可同时实现信息安全传输与数字签名的目的。

非对称加密的常用算法有 RSA 算法等。HTTPS 传输中浏览器使用的数字证书实质上是经过权威机构认证的非对称加密的公钥。

8.2.4 密钥安全管理

前述的几种加密技术，能够达到安全保密效果的一个重要前提是密钥的安全。不管是单向散列加密用到的 salt、对称加密的密钥、还是非对称加密的私钥，一旦这些密钥泄露出去，那么所有基于这些密钥加密的信息就失去了秘密性。

信息的安全是靠密钥保证的。但在实际中经常看到，有的工程师把密钥直接写在源代码中，稍好一点的写在配置文件中，线上和开发环境配置不同的密钥。总之密钥本身是以明文的方式保存，并且很多人可以接触到，至少在公司内部，密钥不是秘密。

实践中，改善密钥安全性的手段有两种。

一种方案是把密钥和算法放在一个独立的服务器上，甚至做成一个专用的硬件设施，对外提供加密和解密服务，应用系统通过调用这个服务，实现数据的加解密。由于密钥和算法独立部署，由专人维护，使得密钥泄露的概率大大降低。但是这种方案成本较高，而且有可能会成为应用的瓶颈，每次加密、解密都需要进行一次远程服务调用，系统性能开销也较大。

另一种方案是将加解密算法放在应用系统中，密钥则放在独立服务器中，为了提高密钥的安全性，实际存储时，密钥被切分成数片，加密后分别保存在不同存储介质中，兼顾密钥安全性的同时又改善了性能，如图 8.10 所示。

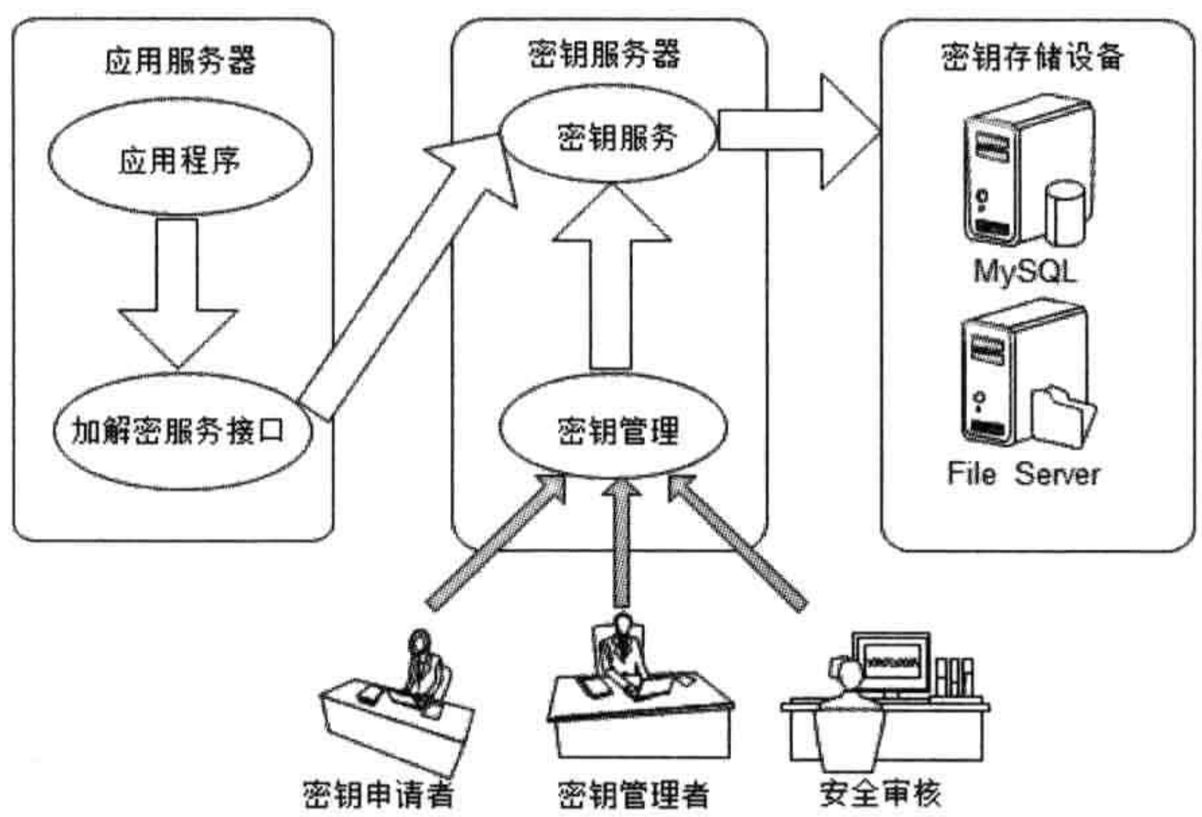

图 8.10　密钥安全管理

应用程序调用密钥安全管理系统提供的加解密服务接口对信息进行加解密，该接口实现了常用的加密解密算法并可根据需求任意扩展。加解密服务接口通过密钥服务器的密钥服务取得加解密密钥，并缓存在本地（定时更新）。而密钥服务器中的密钥则来自多个密钥存储服务器，一个密钥分片后存储在多个存储服务器中，每个服务器都有专人负责管理。密钥申请者、密钥管理者、安全审核人员通过密钥管理控制台管理更新密钥，每个人各司其事，没有人能查看完整的密钥信息。

8.3　信息过滤与反垃圾

我国的信息过滤技术是走在世界前列的，尽管如此，在各种社区网站和个人邮箱中，广告和垃圾信息仍然屡见不鲜、泛滥成灾。

常用的信息过滤与反垃圾手段有以下几种。

8.3.1 文本匹配

文本匹配主要解决敏感词过滤的问题。通常网站维护一份敏感词列表，如果用户发表的信息含有列表中的敏感词，则进行消毒处理（将敏感词转义为***）或拒绝发表。

那么如何快速地判断用户信息中是否含有敏感词呢？如果敏感词比较少，用户提交信息文本长度也较短，可直接使用正则表达式匹配。但是正则表达式的效率一般较差，当敏感词很多，用户发布的信息也很长，网站并发量较高时，就需要更合适的方法来完成，这方面公开的算法有很多，基本上都是 Trie 树的变种，空间和时间复杂度都比较好的有双数组 Trie 算法等。

Trie 算法的本质是确定一个有限状态自动机，根据输入数据进行状态转移。双数组 Trie 算法优化了 Trie 算法，利用两个稀疏数组存储树结构，base 数组存储 Trie 树的节点，check 数组进行状态检查。双数组 Trie 数需要根据业务场景和经验确定数组大小，避免数组过大或者冲突过多。

另一种更简单的实现是通过构造多级 Hash 表进行文本匹配。假设敏感词表包含敏感词：阿拉伯、阿拉汗、阿油、北京、北大荒、北风。那么可以构造如图 8.11 所示的过滤树，用户提交的信息逐字顺序在过滤树中匹配。过滤树的分支可能会比较多，为了提高匹配速度，减少不必要的查找，同一层中相同父节点的字可放在 Hash 表中。该方案处理速度较快，稍加变形，即可适应各种过滤场景，缺点是使用 Hash 表会浪费部分内存空间，如果网站敏感词数量不多，浪费部分内存还是可以接受的。

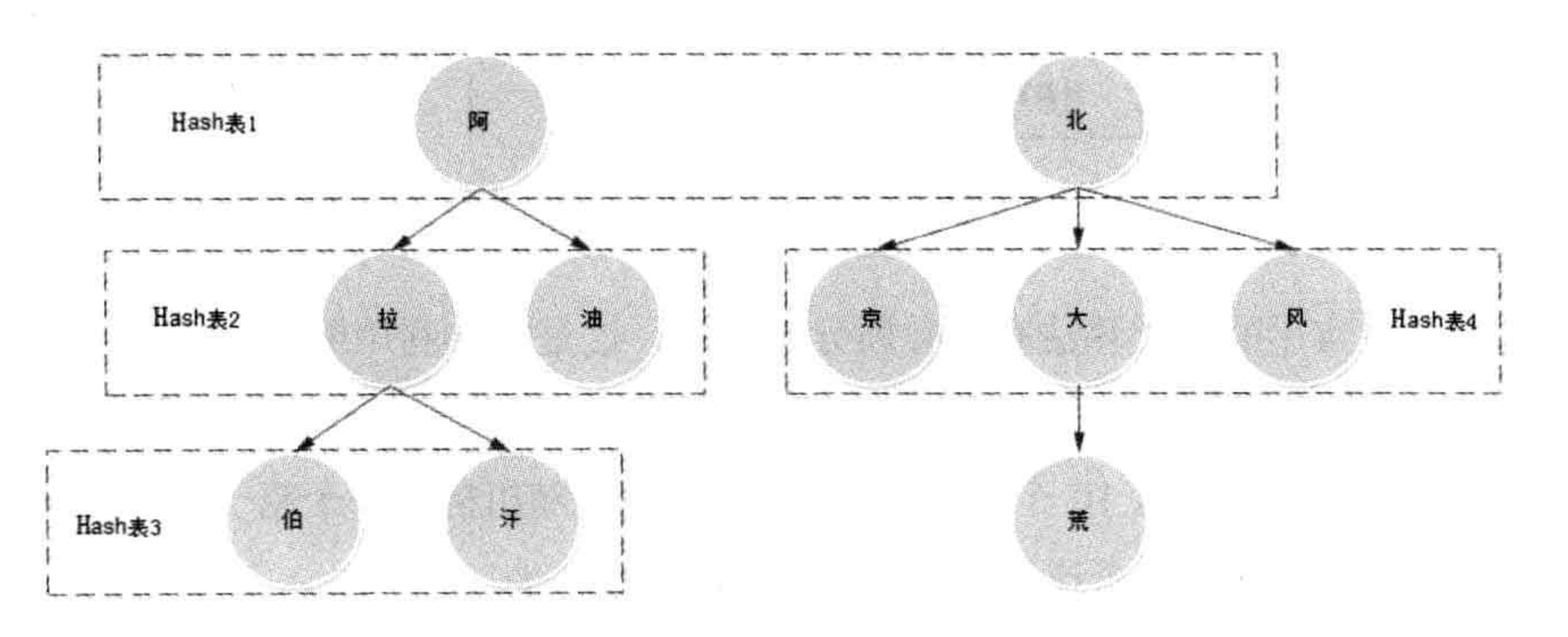

图 8.11 敏感词过滤树

有时候，为了绕过敏感词检查，某些输入信息会被做一些手脚，如“阿_拉_伯”，这时候还需要对信息做降噪预处理，然后再进行匹配。

8.3.2 分类算法

早期网站识别垃圾信息的主要手段是人工方式，后台运营人员对信息进行人工审核。对大型网站而言，特别是以社交为主的 Web2.0 网站，如 Facebook 或 Linkedin 这样的网站，每天用户提交的信息数千万计，许多垃圾信息混杂其中，影响用户体验；而对于 B2B 类的电子商务交易撮合网站，用户主要通过站内信等手段进行商品信息咨询，有时候站内信充斥大量广告，甚至淹没正常询盘，引起用户严重不满和投诉。

对如此海量的信息进行人工审核是不现实的，对广告贴、垃圾邮件等内容的识别比较好的自动化方法是采用分类算法。

以反垃圾邮件为例说明分类算法的使用，如图 8.12 所示。先将批量已分类的邮件样本（如 50000 封正常邮件，2000 封垃圾邮件）输入分类算法进行训练，得到一个垃圾邮件分类模型，然后利用分类算法结合分类模型对待处理邮件进行识别。

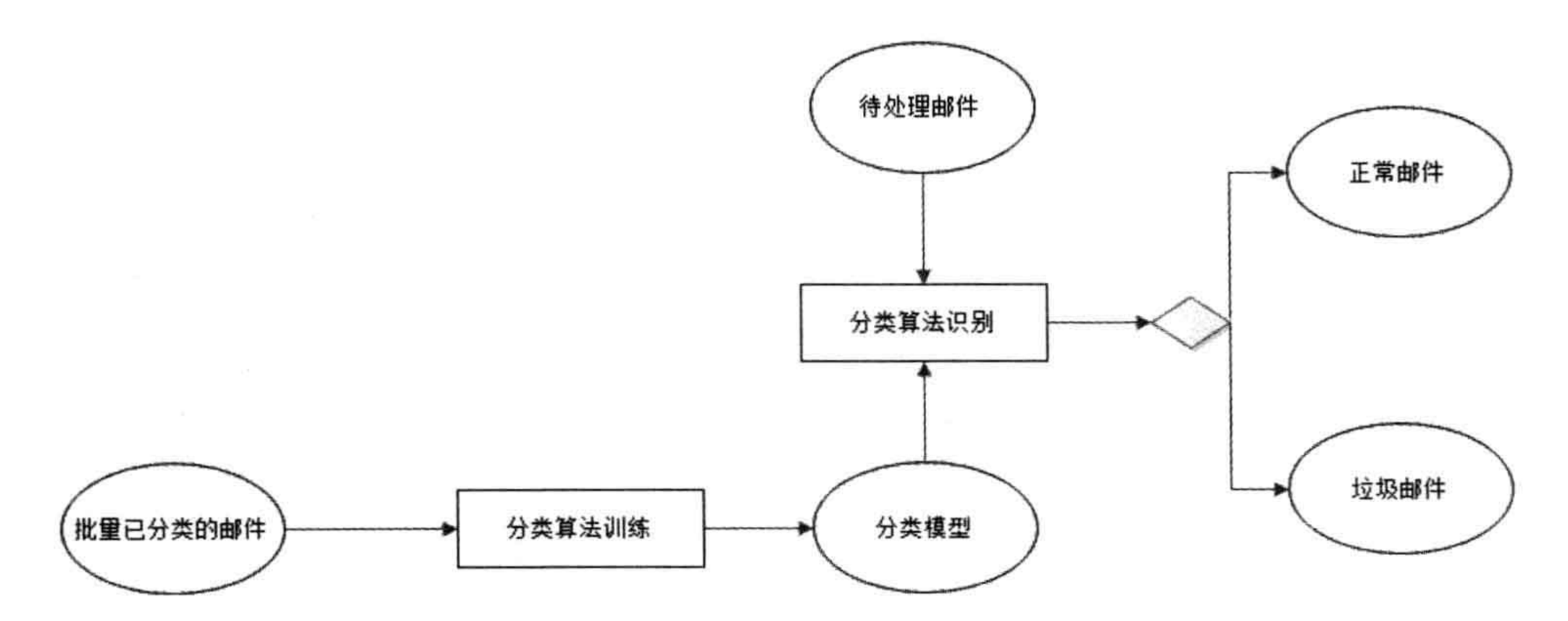

图 8.12 利用分类算法识别垃圾邮件

比较简单实用的分类算法有贝叶斯分类算法，这是一种利用概率统计方法进行分类的算法。贝叶斯算法解决概率论中的一个典型问题：一号箱子放有红色球和白色球各 20 个，二号箱子放有白色球 10 个，红色球 30 个，现在随机挑选一个箱子，取出来一个球的颜色是红色的，请问这个球来自一号箱子的概率是多少。

利用贝叶斯算法进行垃圾邮件的识别基于同样原理，根据已分类的样本信息获得一组特征值的概率，如“茶叶”这个词出现在垃圾邮件中的概率为 20%，出现在非垃圾邮件中的概率为 1%，就得到分类模型。然后对待处理邮件提取特征值，比如取到了茶叶这个特征值，结合分类模型，就可以判断其分类。贝叶斯算法得到的分类判断是一个概率值，因此会存在误判（非垃圾邮件判为垃圾邮件）和漏判（垃圾邮件判为非垃圾邮件）。

贝叶斯算法认为特征值之间是独立的，所以也被称作是朴素贝叶斯算法（Native Bayes），这个假设很多时候是不成立的，特征值之间具有关联性，通过对朴素贝叶斯算法增加特征值的关联依赖处理，得到 TAN 算法。更进一步，通过对关联规则的聚类挖掘，得到更强大的算法，如 ARCS 算法（Association Rule Clustering System）等。但是由于贝叶斯分类算法简单，处理速度快，仍是许多实时在线系统反垃圾的首选。

分类算法除了用于反垃圾，还可用于信息自动分类，门户网站可用该算法对采集来的新闻稿件进行自动分类，分发到不同的频道。邮箱服务商根据邮件内容推送的个性化广告也可以使用分类算法提高投送相关度。

8.3.3 黑名单

对于垃圾邮件，除了用分类算法进行内容分类识别，还可以使用黑名单技术，将被报告的垃圾邮箱地址放入黑名单，然后针对邮件的发件人在黑名单列表中查找，如果查找成功，则过滤该邮件。

黑名单也可用于信息去重，如将文章标题或者文章关键段落记录到黑名单中，以减少搜索引擎收录重复信息等用途。

黑名单可以通过 Hash 表实现，该方法实现简单，时间复杂度小，满足一般场景使用。但是当黑名单列表非常大时，Hash 表需要占据极大的内存空间。例如在需要处理 10 亿个黑名单邮件地址列表的场景下，每个邮件地址需要 8 个字节的信息指纹，即需要 8GB 内存，为了减少 Hash 冲突，还需要一定的 Hash 空间冗余，假如空间利用率为 50%，则需要 16GB 的内存空间。随着列表的不断增大，一般服务器将不可承受这样的内存需求。而且列表越大，Hash 冲突越多，检索速度越慢。

在对过滤需求要求不完全精确的场景下，可用布隆过滤器代替 Hash 表。布隆过滤器是用它的发明者巴顿·布隆的名字命名的，通过一个二进制列表和一组随机数映射函数

实现，如图 8.13 所示。

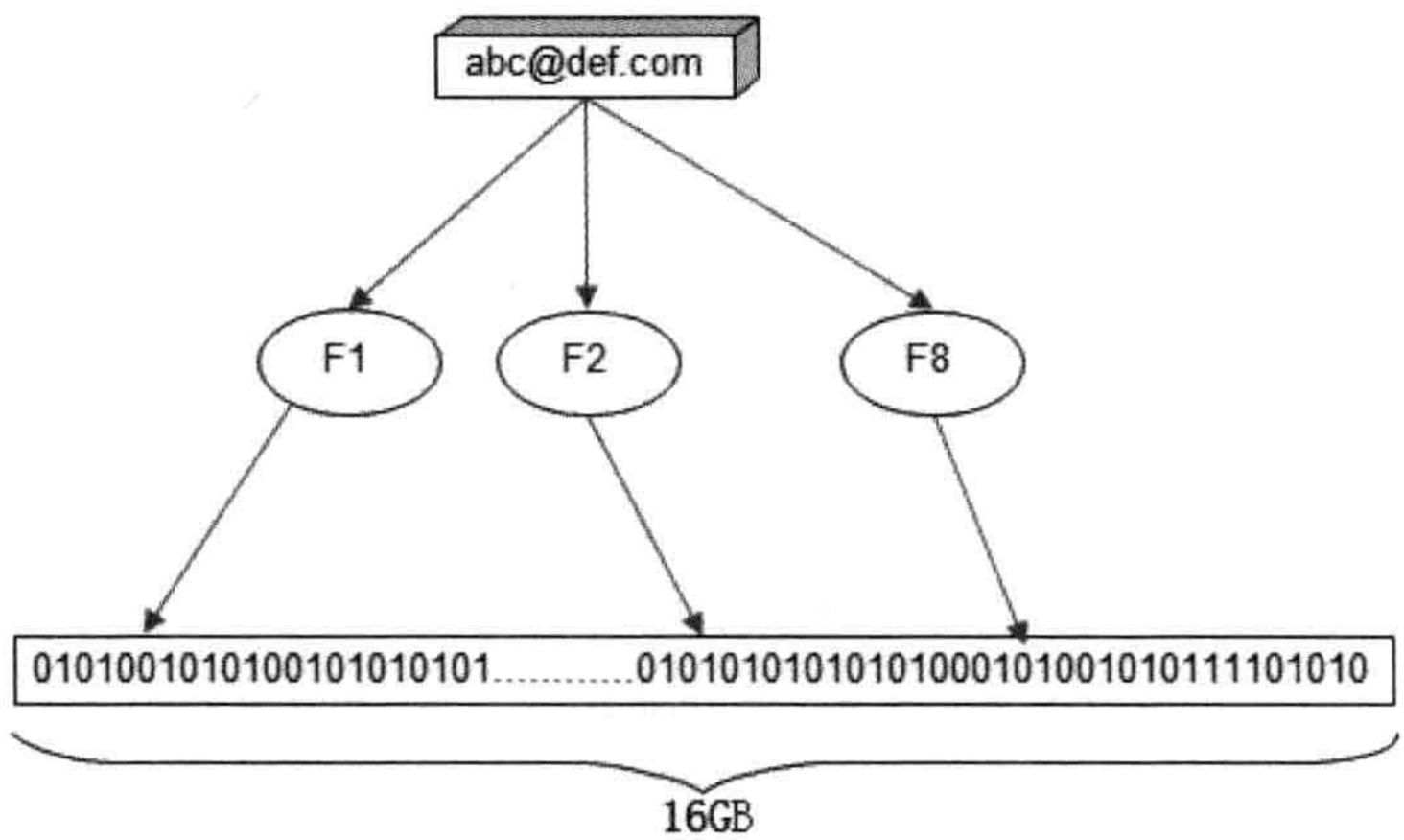

图 8.13　布隆过滤器

仍以需要处理 10 亿邮件地址黑名单列表为例，在内存中建立一个 2GB 大小的存储空间，即 16GB 个二进制 bit，并全部初始化为 0。要将一个邮箱地址加入黑名单时，使用 8 个随机映射函数（F1,F2,⋯,F8）得到 0~16GB 范围内的 8 个随机数，从而将该邮箱地址映射到 16GB 二进制存储空间的 8 个位置上，然后将这些位置置为 1。当要检查一个邮箱地址是否在黑名单中时，使用同样的映射函数，得到 16GB 空间 8 个位置上的 bit，如果这些值都为 1，那么该邮箱地址在黑名单中。

可以看到，处理同样数量的信息，布隆过滤器只使用 Hash 表所需内存的 1/8。但是布隆过滤器有可能导致系统误判（布隆过滤器检查在黑名单中，但实际却并未放入过）。因为一个邮箱地址映射的 8 个 bit 可能正好都被其他邮箱地址设为 1 了，这种可能性极小，通常在系统可接受范围内。但如果需要精确的判断，则不适合使用布隆过滤器。

8.4　电子商务风险控制

电子商务网站在给人们带来购物交易的极大便利的同时，也将风险带给了对网络安全一无所知的人们。由于买卖双方的信息不对等，交易本来就存在风险，而当交易在网上发生时，买卖双方彼此一无所知，交易风险也就更加难以控制。如果一个电商网站骗子横行，诚信的交易者屡屡被骗，那么网站就到了最危险的时候，可以说，交易安全是

电子商务网站的底线。

8.4.1 风险

电子商务具有多种形式，B2B、B2C、C2C 每种交易的场景都不相同，风险也各有特点，大致可分为以下几种。

账户风险：包括账户被黑客盗用，恶意注册账号等几种情形。

买家风险：买家恶意下单占用库存进行不正当竞争；黄牛利用促销抢购低价商品；此外还有良品拒收，欺诈退款及常见于 B2B 交易的虚假询盘等。

卖家风险：不良卖家进行恶意欺诈的行为，例如货不对板，虚假发货，炒作信用等，此外还有出售违禁商品、侵权产品等。

交易风险：信用卡盗刷，支付欺诈，洗钱套现等。

8.4.2 风控

大型电商网站都配备有专门的风控团队进行风险控制，风控的手段也包括自动和人工两种。机器自动识别为高风险的交易和信息会发送给风控审核人员进行人工审核，机器自动风控的技术和方法也不断通过人工发现的新风险类型进行逐步完善。

机器自动风控的技术手段主要有规则引擎和统计模型。

1. 规则引擎

当交易的某些指标满足一定条件时，就会被认为具有高风险的欺诈可能性。比如用户来自欺诈高发地区；交易金额超过某个数值；和上次登录的地址距离差距很大；用户登录地与收货地不符；用户第一次交易等等。

大型网站在运营过程中，结合业界的最新发现，会总结出数以千计的此类高风险交易规则。一种方案是在业务逻辑中通过编程方式使用 if...else...代码实现这些规则，可想而知，这些代码会非常庞大，而且由于运营过程中不断发现新的交易风险类型，需要不断调整规则，代码也需要不断修改……

网站一般使用规则引擎技术处理此类问题。规则引擎是一种将业务规则和规则处理

逻辑相分离的技术，业务规则文件由运营人员通过管理界面编辑，当需要修改规则时，无需更改代码发布程序，即可实时使用新规则。而规则处理逻辑则调用规则处理输入的数据，如图 8.14 所示。

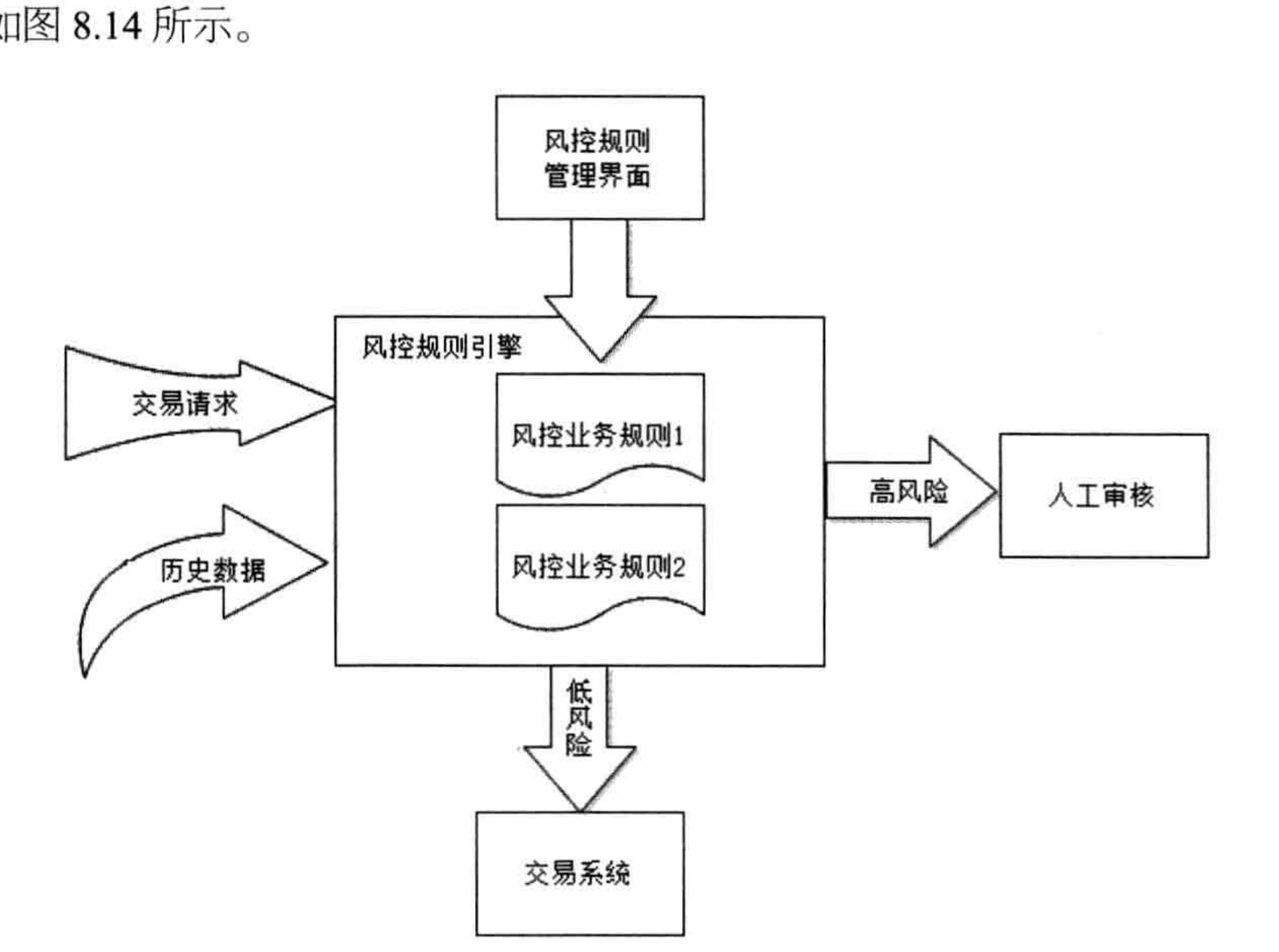

图 8.14　基于规则引擎的风险控制系统

2. 统计模型

规则引擎虽然技术简单，但是随着规则的逐渐增加，会出现规则冲突，难以维护等情况，而且规则越多，性能也越差。目前大型网站更倾向于使用统计模型进行风控。风控领域使用的统计模型使用前面提到的分类算法或者更复杂的机器学习算法进行智能统计。如图 8.15 所示，根据历史交易中的欺诈交易信息训练分类算法，然后将经过采集加工后的交易信息输入分类算法，即可得到交易风险分值。

经过充分训练后的统计模型，准确率不低于规则引擎。分类算法的实时计算性能更好一些，由于统计模型使用模糊识别，并不精确匹配欺诈类型规则，因此对新出现的交易欺诈还具有一定预测性。

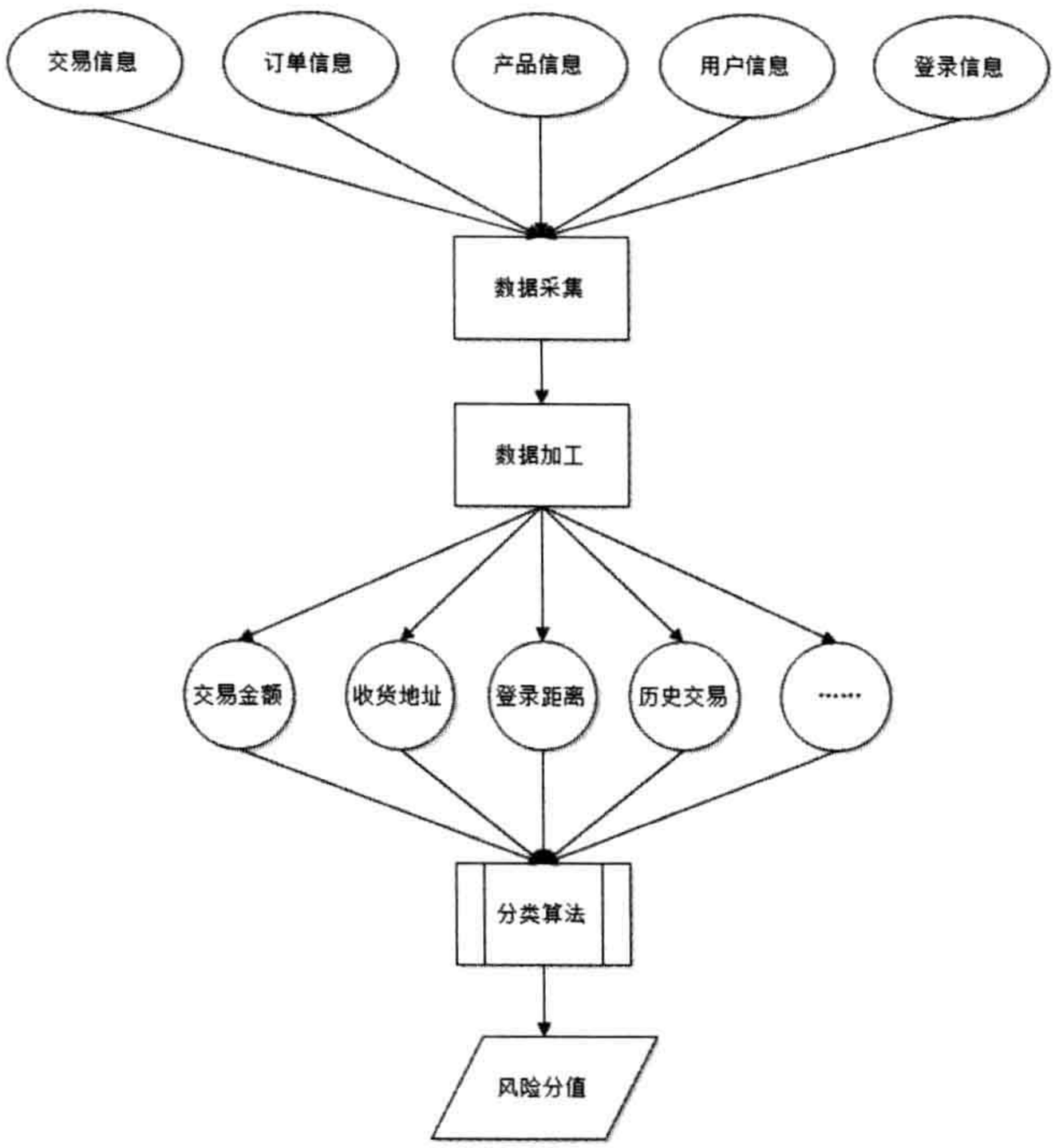

图 8.15　基于统计模型的风险控制系统

8.5 小结

这个世界没有绝对的安全，正如没有绝对的自由一样。网站的相对安全是通过提高攻击门槛达到的。让攻击者为了获得有限的利益必须付出更大的代价，致使其得不偿失，望而却步。

同时，攻击与防护技术作为一对矛盾共同体，彼此不断此消彼长，今天的高枕无忧，明天可能就成了致命的漏洞。也许网站经过一番大的重构和优化，在某一段时间不需要再处理高可用或高性能的问题，但是修补漏洞、改善安全却是每天都需要面对的课题，永远不能停歇。

所以，很遗憾，这个世界没有固若金汤的网站安全架构，架构师只能每天都打起百分百的精神，预防可能的漏洞或者攻击。

第 3 篇

案　例

9 淘宝网的架构演化案例分析

2012 年 11 月 30 日，淘宝（包括天猫）的当年交易额突破 1 万亿，这是一个可以鄙睨亚马逊和 eBay 的数字。而就在此前不久的 2012 年 11 月 11 日，淘宝更是创造了全球电子商务的奇迹，当天：

- 总交易额：191 亿人民币
- 零点的第一分钟，1000 万独立用户涌入 www.tmall.com
- 全天访问用户总数达 2 亿 1 千 3 百万，占中国网民总数的 40%
- 总成交订单数：1 亿零 5 百万
- 高峰期，每分钟成交订单 89678 笔

淘宝历年交易额如图 9.1 所示。

所有这些绚丽的业务数字背后是淘宝多年积淀的电子商务网站架构技术。

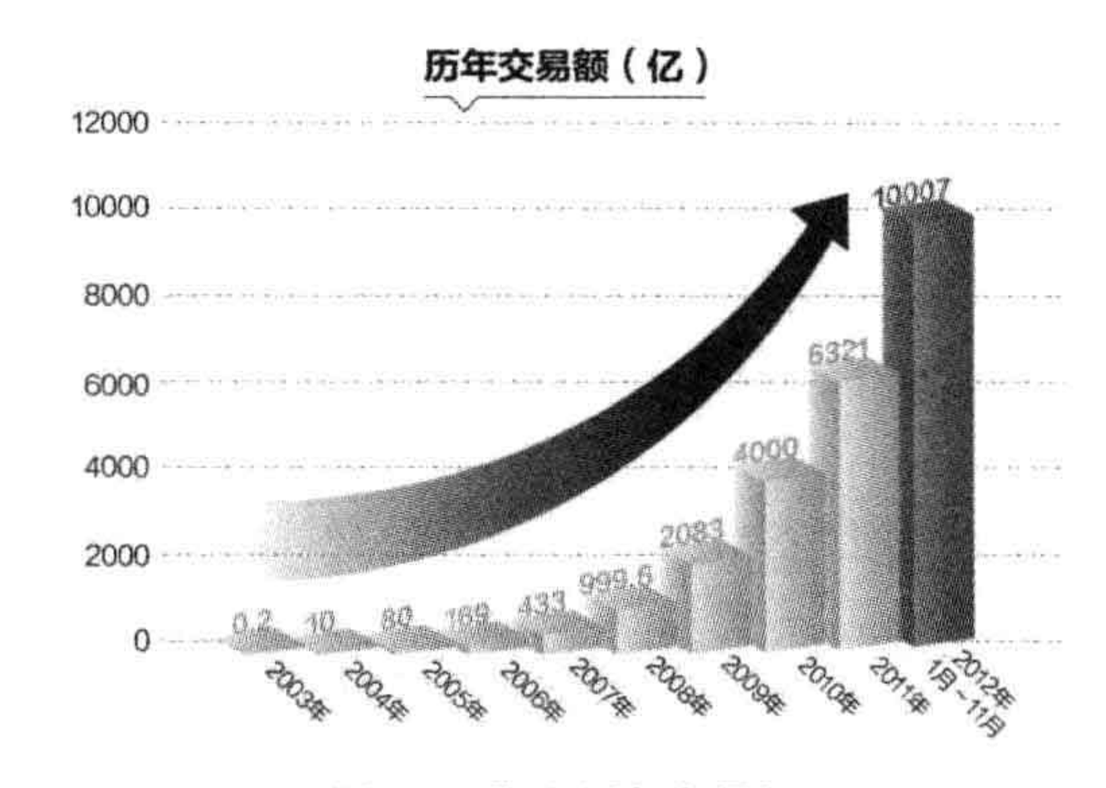

图 9.1　淘宝历年交易额

注：淘宝作为阿里巴巴集团旗下网站，其技术和集团其他公司有着千丝万缕的联系，本章主要分析淘宝架构演化，具体技术不特别区分源自淘宝抑或阿里巴巴。

9.1　淘宝网的业务发展历程

淘宝的技术是和淘宝的业务一起发展起来的，没有飞速发展的淘宝业务，就不会有今天让技术界艳羡的淘宝技术，可以说，是业务驱动着技术不得不往前走。而淘宝的业务也经历了由简单到复杂，由初级到高级的发展历程，通过淘宝首页的变迁，我们可以看出淘宝业务逐步发展的脉络。

2003 年，在马云家里，用一个买来的 C2C 交易软件稍作修改就成了最初的淘宝网，简约也简单，如图 9.2 所示。

图 9.2　2003 年的淘宝网首页

2004 年，淘宝业务由模仿 eBay 的拍卖交易，开始向一口价交易转型，也就是现在淘宝购物的主要交易模式，这一年，淘宝的架构也经历了一次重大重构，PHP 换成了 Java，MySQL 换成了 Oracle。那时，淘宝网首页虽然依旧简单，但是对于电子商务网站最重要的部分——商品类目开始建立并成为重要的商品导航方式，如图 9.3 所示。

图 9.3　2004 年淘宝网首页

此后数年间，淘宝逐步成为网购的代名词，引领中国电子商务的步伐，每年一度的“双十一”促销成为有中国特色的购物狂欢节。2012 年的淘宝网首页如图 9.4 所示。

图 9.4　2012 年淘宝网首页

9.2　淘宝网技术架构演化

2003 年，花 3000 美金买来的淘宝网站是用 PHP 开发的，淘宝的工程师做了简单的

汉化处理，并对数据库做了读写分离，最早的淘宝网架构如图 9.5 所示。

像我们见过的绝大多数中小网站一样，当年的淘宝网使用典型的 Linux+Apache+MySQL+PHP（LAMP）架构。作为一个刚刚起步的小网站，使用开源、免费、简单的技术产品搭建网站是明智之举，可谓一举多得：免费的技术降低网站的成本，成熟的开源技术可以从开源社区获取文档和技术支持；网站发展初期，业务不明确，需求变化多，简单的技术方案可以快速响应需求变化；简单的技术也可以让工程师快速上手，缩短学习周期；退一步，如果业务发展不顺利，及时关闭网站止损，亦可减少沉没成本，促使管理层和投资者快速决策。

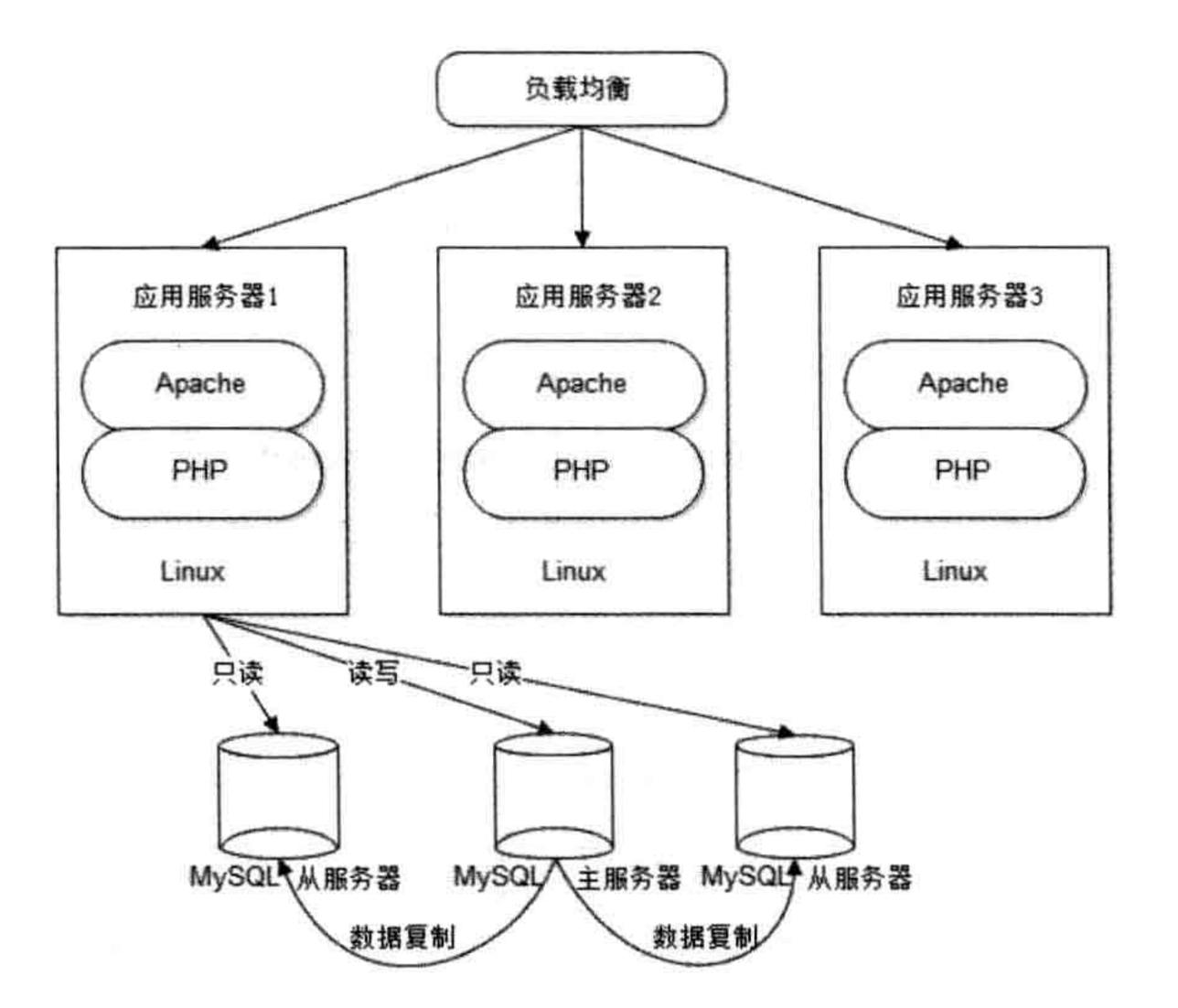

图 9.5　2003 年淘宝网架构

幸而淘宝业务蒸蒸日上，逐步蚕食 eBay 在中国的市场。随着业务的快速发展，电子商务网站特有的业务复杂性和 PHP 易开发、难维护的特性产生了难以调和的冲突；不断扩展的业务让工程师承受着沉重的负担；不断增加的用户和商品数又让系统特别是存储系统不堪重负。总之，架构重构势在必行。2004 年，淘宝在 SUN 技术顾问的协助下进行了一次重要的重构，放弃了原来的 LAMP 架构，转而使用 Java 作为开发平台，使用 Oracle 做后端数据库，如图 9.6 所示。

系统架构使用了当时在企业应用领域崭露头角的 MVC 框架和 ORM 框架，分别解决视图与业务逻辑分离的问题和对象与关系数据库解耦的问题，淘宝没有使用当时风头正

劲的 Struts 和 Hibernate，而是选择了自己开发 MVC 框架 Webx，而 ORM 框架则选择了 IBatis。

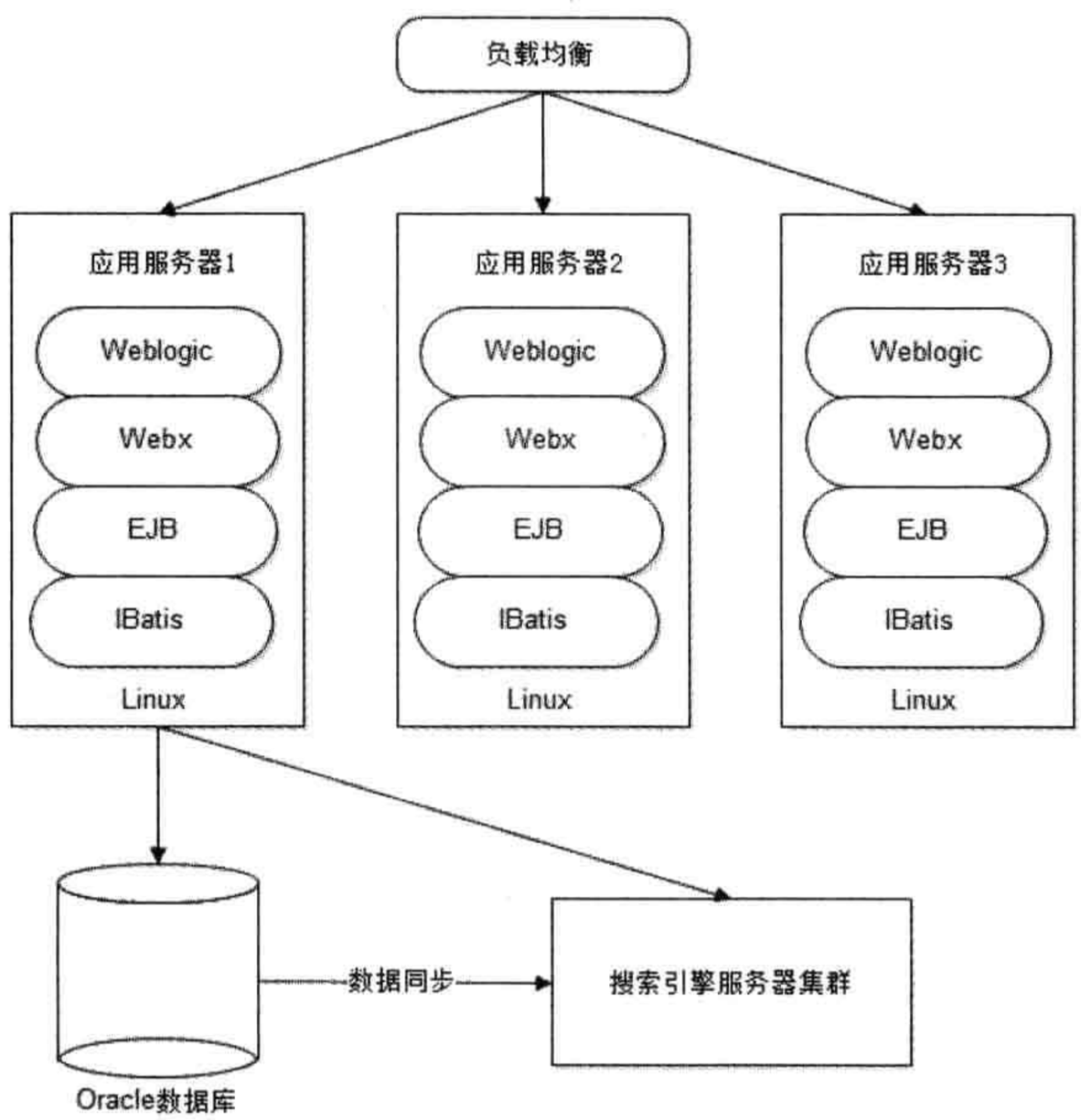

图 9.6　2004 年淘宝网架构

当时淘宝还开发了另一个重要产品 Antx，这个针对 Java 平台的、扩展自 Ant 的项目构建工具对于网站项目开发、测试、发布至关重要，一个非常重要的功能就是管理配置项。对于一个 Java 开发的大型 Web 系统，内部通常会包含数百个 jar 文件，每个 jar 文件都是一个独立的模块，这些模块由不同团队开发，实现不同功能，最后组成一个完整的系统。这些模块通常也都有自己的配置文件，比如数据库连接模块需要配置数据库 URL、连接池大小等，这些配置参数在开发环境、测试环境、生产环境各不相同。Antx 提供了一个灵活管理这些分散配置项的解决方案。

应用服务器使用 Weblogic，数据库使用 Oracle，这些产品都需要昂贵的授权使用费。而 Oracle 又需要部署在昂贵的 IBM 小型机和同样昂贵的 EMC 存储设备上。淘宝这时候弃免费而选择付费产品，和建站初选择免费一样，同样是明智之举：业务快速发展，宝贵的开发资源应该投入到新业务开发上，而不是解决这些可以用付费产品搞定的基础技术问题上；成熟的付费产品和售后支持令业务和市场没有后顾之忧，可以全力以赴地拓

展市场；对于一个快速发展的网站，特别是电子商务网站而言，严重宕机、重要用户数据丢失可能会极大地打击消费者信心，令网站发展平生波澜，而这些业界领先的产品经过多年的洗练，有较强的可用性保证。

此后三四年间，淘宝在 Oracle、EMC、IBM 的护航下，高歌猛进，业务蒸蒸日上，技术也逐渐成长，基于自身需求，逐步摸索适合自己的技术发展之路，如图 9.7 所示。

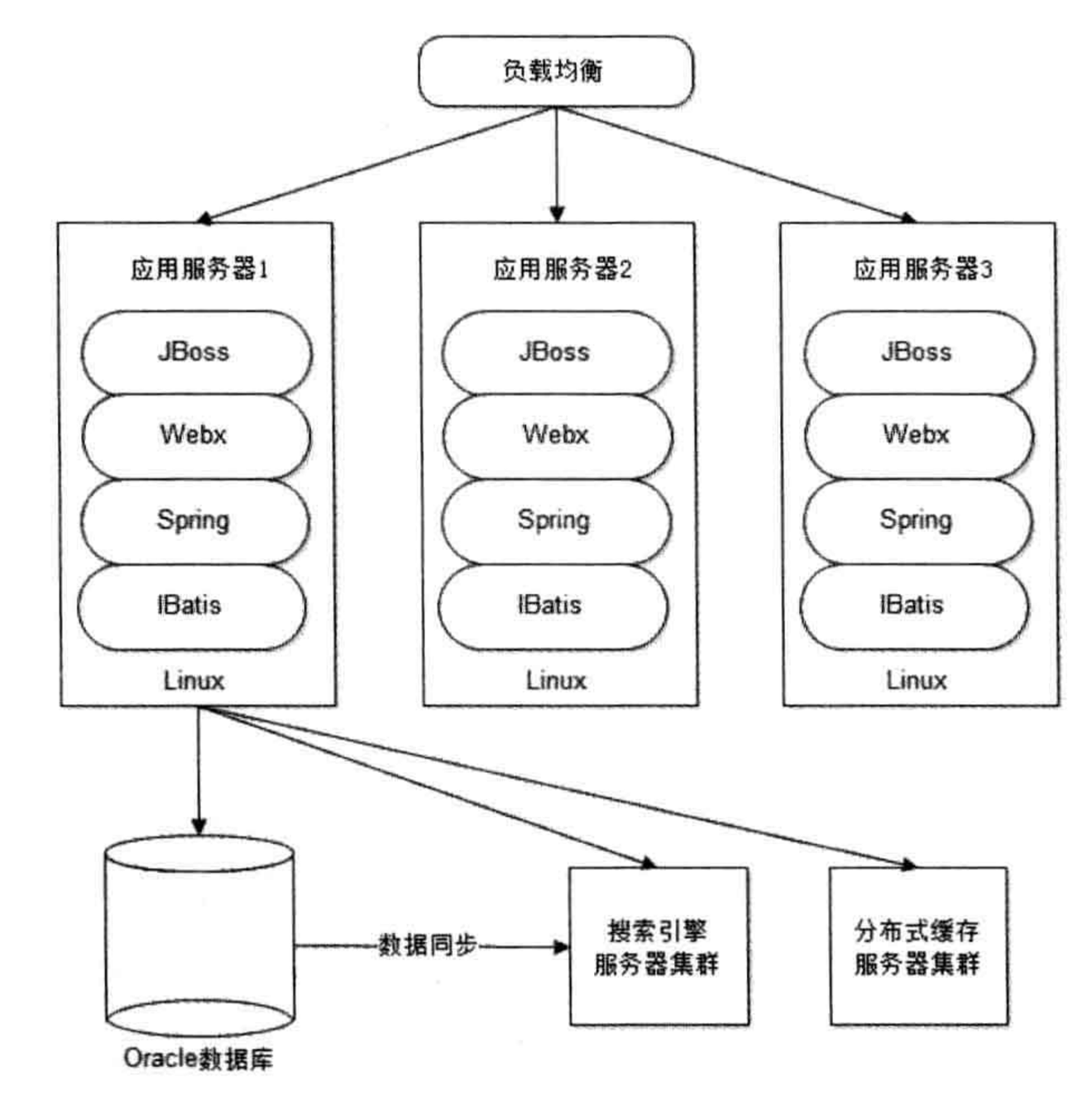

图 9.7　2006 年的淘宝架构

放弃 EJB，引入 Spring，用免费的 JBoss 替代收费的 Weblogic，因为 Weblogic 并非物有所值，EJB 对于网站来说也太过笨重。淘宝后来甚至用更轻量级的 Jetty 替代了 JBoss，对淘宝而言，应用服务器只需要一个 Servlet 容器，越简单越快越好。在合适的场景下使用合适的产品，而不是最好的产品，所谓小脚穿大鞋，不但跑不快，还可能会摔跤。

直到这时，淘宝架构和技术依然是泯然于众的中庸水平而已，没有拖业务的后腿，使用业界成熟的方案和可靠的技术，没有什么可指责的也没有什么可炫耀的。但也就是在这个时候，淘宝技术开始发力，许多奠定淘宝坚实架构基础的产品和技术从这个时候开始逐步酝酿，走向成熟。目前这些产品多数都已开源，如表 9.1 所示。

表 9.1 淘宝主要开源系统

项 目 名	描 述
Tair	分布式 Key/Value 存储引擎，分为持久化和非持久化两种使用方式
TFS	一个分布式文件系统，适用于海量小文件存储
OceanBase	分布式数据库系统，支持千亿级别的读写事务
TDDL	对应用透明的分库分表层和具有众多特性的动态数据源

（资料来源：http://code.taobao.org/）

随着淘宝技术的不断发展壮大，淘宝对集群环境下分布式高可用系统的架构设计技术越来越得心应手，Oracle、IBM、EMC 也变得不是必须，于是淘宝开始逐步放弃使用这些昂贵的设备和软件，回归到开源的 MySQL 及 NoSQL 系统，正如淘宝 2003 年建站之初的选择。这也再一次验证了辩证法关于事物发展的否定之否定及螺旋式上升的普遍规律，仿佛回到原点，但一切已经完全不同了。

9.3 小结

如果说有什么神奇的力量促使淘宝技术脱胎换骨，化蛹成蝶，站在中国互联网软件开发技术之巅华山论剑的话，笔者认为最重要甚至唯一的驱动力就是：不得已。随着业务的飞速发展，用户、数据、流量、业务复杂度都呈指数级增长，飞速接近甚至突破 Oracle、IBM 这些企业提供的解决方案的有效范围，在开源领域虽有 Google、Yahoo 等先驱在探索道路，并有一些开源产品，但是在大规模集群实践上，大家都在摸索，淘宝必须走自己的路，路上也许有烛光照明，但是没有人指路。

而有些路，走过以后，再回头，一览众山小！

10 维基百科的高性能架构设计分析

www.wikipedia.org，这个在 2001 年创建，使用 Perl CGI 脚本编写的只有一台服务器的网站，到 2012 年已经成为流量排名全球第 6 的大型网站，如图 10.1 所示。和 www.wikipedia.org 的流量在相同级别的其他大型网站，如 www.baidu.com、www.yahoo.com，其背后都是市值数百亿美金、员工上万的巨无霸企业，运行网站的服务器规模也数以万计。而 wikipedia.org 不过只有区区数百台服务器，并仅由十余名技术人员维护，不得不说是一个奇迹。Wikipedia 对资源的利用，对性能的优化很具有典型性，有许多值得学习的地方。

10.1 Wikipedia 网站整体架构

目前 Wikipedia 网站建立在 LAMP（Linux+Apache+MySQL+PHP）之上，其他基础技术组件也全部采用免费的开源软件。因为 Wikipedia 是非盈利的，所以尽可能使用免费的软件和廉价的服务器，这种技术倾向使得技术团队不得不量体裁衣、看米下锅，榨尽系统所有资源的利用价值，用最少的资源成就最不可思议的奇迹，最终也让技术团队获得

了真正的成长。

1 Google
google.com
Enables users to search the world's information, including webpages, images, and videos. Offers... More
Search Analytics Audience

2 Facebook
facebook.com
A social utility that connects people, to keep up with friends, upload photos, share links and ... More
Search Analytics Audience

3 YouTube
youtube.com
YouTube is a way to get your videos to the people who matter to you. Upload, tag and share your... More
Search Analytics Audience

4 Yahoo!
yahoo.com
A major internet portal and service provider offering search results, customizable content, cha... More
Search Analytics Audience

5 Baidu.com
baidu.com
The leading Chinese language search engine, provides "simple and reliable" search exp... More
Search Analytics Audience

6 Wikipedia
wikipedia.org
A free encyclopedia built collaboratively using wiki software. (Creative Commons Attribution-Sh... More
Search Analytics Audience

图 10.1 2012 年 12 月 8 日 Alexa 全球网站排名

Wikipedia 的架构如图 10.2 所示。

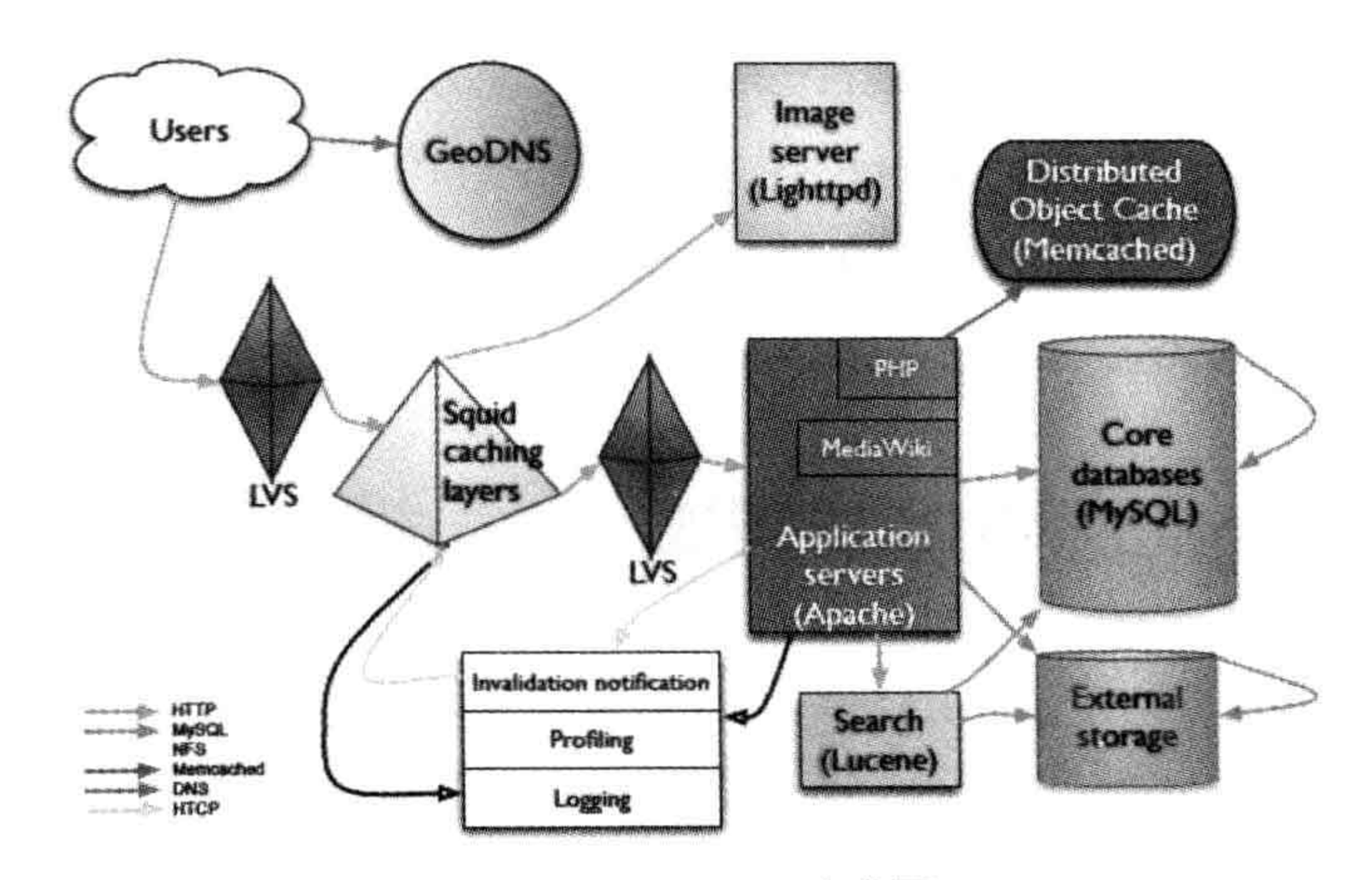

图 10.2 Wikipedia 架构图

（图片来源：http://www.slideshare.net/kapil/wikimediaarchiteciture）

Wikipedia 架构的主要组成部分如下。

GeoDNS：基于开源域名服务器软件 BIND（Berkeley Internet Name Domain）的增强版本，可将域名解析到离用户最近的服务器。

LVS：基于 Linux 的开源负载均衡服务器。

Squid：基于 Linux 的开源反向代理服务器。

Lighttpd：开源的应用服务器，较主流的 Apache 服务器更轻量、更快速。实践中，有许多网站使用 Lighttpd 作为图片服务器。

PHP：免费的 Web 应用程序开发语言，最流行的网站建站语言。

Memcached：无中心高性能的开源分布式缓存系统，稳定、可靠、历久弥新，是网站分布式缓存服务必备的。

Lucene：由 Apache 出品，Java 开发的开源全文搜索引擎。

MySQL：开源的关系数据库管理系统，虽被 Oracle 收购，但开源社区将其继续开源发展的决心不动摇。

10.2 Wikipedia 性能优化策略

作为一个百科服务类网站，Wikipedia 主要面临的挑战是如何应对来自全球各地的巨量并发的词条查询请求。相对其他网站，Wikipedia 的业务比较简单，用户操作大部分是只读的，这些前提使 Wikipedia 的性能优化约束变得简单，可以让技术团队将每一种性能优化手段都发挥到极致，且业务束缚较少。因此 Wikipedia 的性能优化比较有典型意义。

10.2.1 Wikipedia 前端性能优化

所谓网站前端是指应用服务器（也就是 PHP 服务器）之前的部分，包括 DNS 服务、CDN 服务、反向代理服务、静态资源服务等，如图 10.3 所示。对 Wikipedia 而言，80% 以上的用户请求可以通过前端服务返回，请求根本不会到达应用服务器，这也就使得网站最复杂、最有挑战的应用服务端和存储端压力骤减。

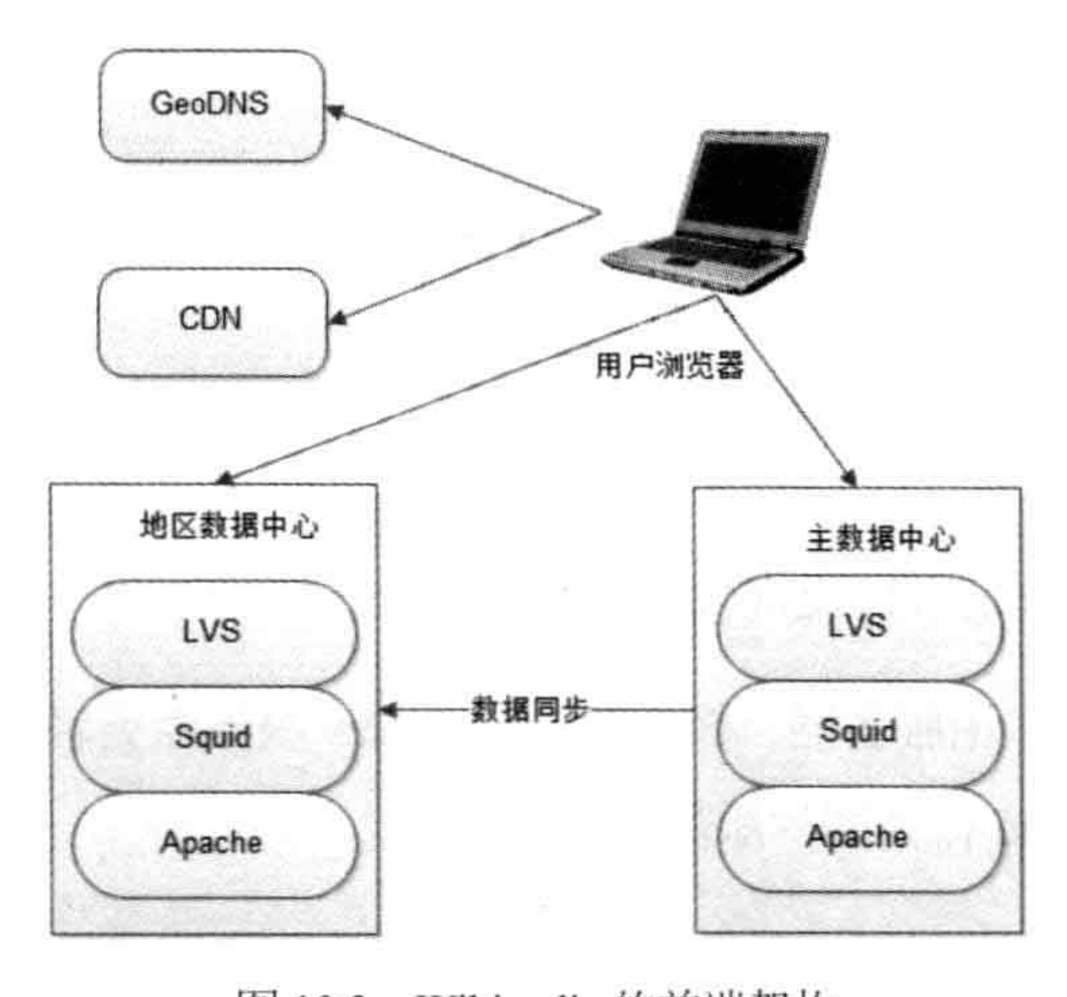

图 10.3 Wikipedia 的前端架构

Wikipedia 前端架构的核心是反向代理服务器 Squid 集群，大约部署有数十台服务器，请求通过 LVS 负载均衡地分发到每台 Squid 服务器，热点词条被缓存在这里，大量请求可直接返回响应，请求无需发送到 Apache 服务器，减轻应用负载压力。Squid 缓存不能命中的请求再通过 LVS 发送到 Apache 应用服务器集群，如果有词条信息更新，应用服务器使用 Invalidation Notification 服务通知 Squid 缓存失效，重新访问应用服务器更新词条。

而在反向代理 Squid 之前，则是被 Wikipedia 技术团队称为“圣杯”的 CDN 服务，CDN 服务对于 Wikipedia 性能优化居功至伟。因为用户查询的词条大部分集中在比重很小的热点词条上，将这些词条内容页面缓存在 CDN 服务器上，而 CDN 服务器又部署在离用户浏览器最近的地方，用户请求直接从 CDN 返回，响应速度非常快，这些请求甚至根本不会到达 Wikipedia 数据中心的 Squid 服务器，服务器压力减小，节省的资源可以更快地处理其他未被 CDN 缓存的请求。

Wikipedia CDN 缓存的几条准则为：

- 内容页面不包含动态信息，以免页面内容缓存很快失效或者包含过时信息。
- 每个内容页面有唯一的 REST 风格的 URL，以便 CDN 快速查找并避免重复缓存。
- 在 HTML 响应头写入缓存控制信息，通过应用控制内容是否缓存及缓存有效期等。

10.2.2 Wikipedia 服务端性能优化

服务端主要是 PHP 服务器，这里是网站业务逻辑的核心部分，运行的模块都比较复

杂笨重，需要消耗较多的资源，Wikipedia 将最好的服务器部署在这里（和数据库配置一样的服务器），从硬件上改善性能。

除了硬件改善，Wikipedia 还使用许多其他开源组件对应用层进行如下优化。

- 使用 APC，这是一个 PHP 字节码缓存模块，可以加速代码执行减少资源消耗。
- 使用 Imagemagick 进行图片处理和转化。
- 使用 Tex 进行文本格式化，特别是将科学公式内容转换成图片格式。
- 替换 PHP 的字符串查找函数 strtr()，使用更优化的算法重构。

10.2.3 Wikipedia 后端性能优化

包括缓存、存储、数据库等被应用服务器依赖的服务都可以归类为后端服务。后端服务通常是一些有状态的服务，即需要提供数据存储服务，这些服务大多建立在网络通信和磁盘操作基础上，是性能的瓶颈，也是性能优化的重灾区。

后端优化最主要的手段是使用缓存，将热点数据缓存在分布式缓存系统的内存中，加速应用服务器的数据读操作速度，减轻存储和数据库服务器的负载。Wikipedia 的缓存使用策略如下：

- 热点特别集中的数据直接缓存到应用服务器的本地内存中，因为要占用应用服务器的内存且每台服务器都需要重复缓存这些数据，因此这些数据量很小，但是读取频率极高。
- 缓存数据的内容尽量是应用服务器可以直接使用的格式，比如 HTML 格式，以减少应用服务器从缓存中获取数据后解析构造数据的代价。
- 使用缓存服务器存储 session 对象。
- 相比数据库，Memcached 的持久化连接非常廉价，如有需要就创建一个 Memcached 连接。

作为存储核心资产的 MySQL 数据库，Wikipedia 也做了如下优化：

- 使用较大的服务器内存。在 Wikipedia 应用场景中，增加内存比增加其他资源更能改善 MySQL 性能。
- 使用 RAID0 磁盘阵列以加速磁盘访问，RAID0 虽然加速磁盘访问，但是却降低了

数据库的持久可靠性(一块盘坏了,整个数据库的数据都不完整了)。显然 Wikipedia 认为性能问题迫在眉睫，而数据可靠性问题可以通过其他手段解决(如 MySQL 主从复制，数据异步备份等)。

- 将数据库事务一致性设置在较低水平，加快宕机恢复速度。
- 如果 Master 数据库宕机，立即将应用切换到 Salve 数据库，同时关闭数据写服务，这意味着关闭词条编辑功能。Wikipedia 通过约束业务获得更大的技术方案选择余地，很多时候业务后退一小步，技术就可以前进一大步。

11

海量分布式存储系统 Doris 的高可用架构设计分析

Doris（https://github.com/itisaid/Doris）是一个海量分布式 KV 存储系统，其设计目标是支持中等规模高可用、可伸缩的 KV 存储集群。跟主流的 NoSQL 系统 HBase 相比（Doris0.1 vs. HBase0.90），Doris 具有相似的性能和线性伸缩能力，并具有更好的可用性及更友好的图形用户管理界面。

对于一个数据存储系统而言，高可用意味着两个意思：

- 高可用的服务：任何时候，包括宕机、硬盘损坏、系统升级、停机维护、集群扩容等各种情况，都可以对系统进行读写访问操作。
- 高可靠的数据：任何情况下，数据可靠存储，不丢失。

那么高可用的架构设计也就是在各种软硬件故障的情况下，系统如何保障数据可靠存储，服务可用。

11.1 分布式存储系统的高可用架构

对一个大规模集群的存储系统而言，服务器宕机、交换机失效是常态，架构师必须为这些故障发生时，保证系统依然可用而进行系统设计。在系统架构层面，保证高可用的主要手段是冗余：服务器热备，数据多份存储。使整个集群在部分机器故障的情况下可以进行灵活的失效转移（Failover），保证系统整体依然可用，数据持久可靠。Doris 系统的架构如图 11.1 所示。

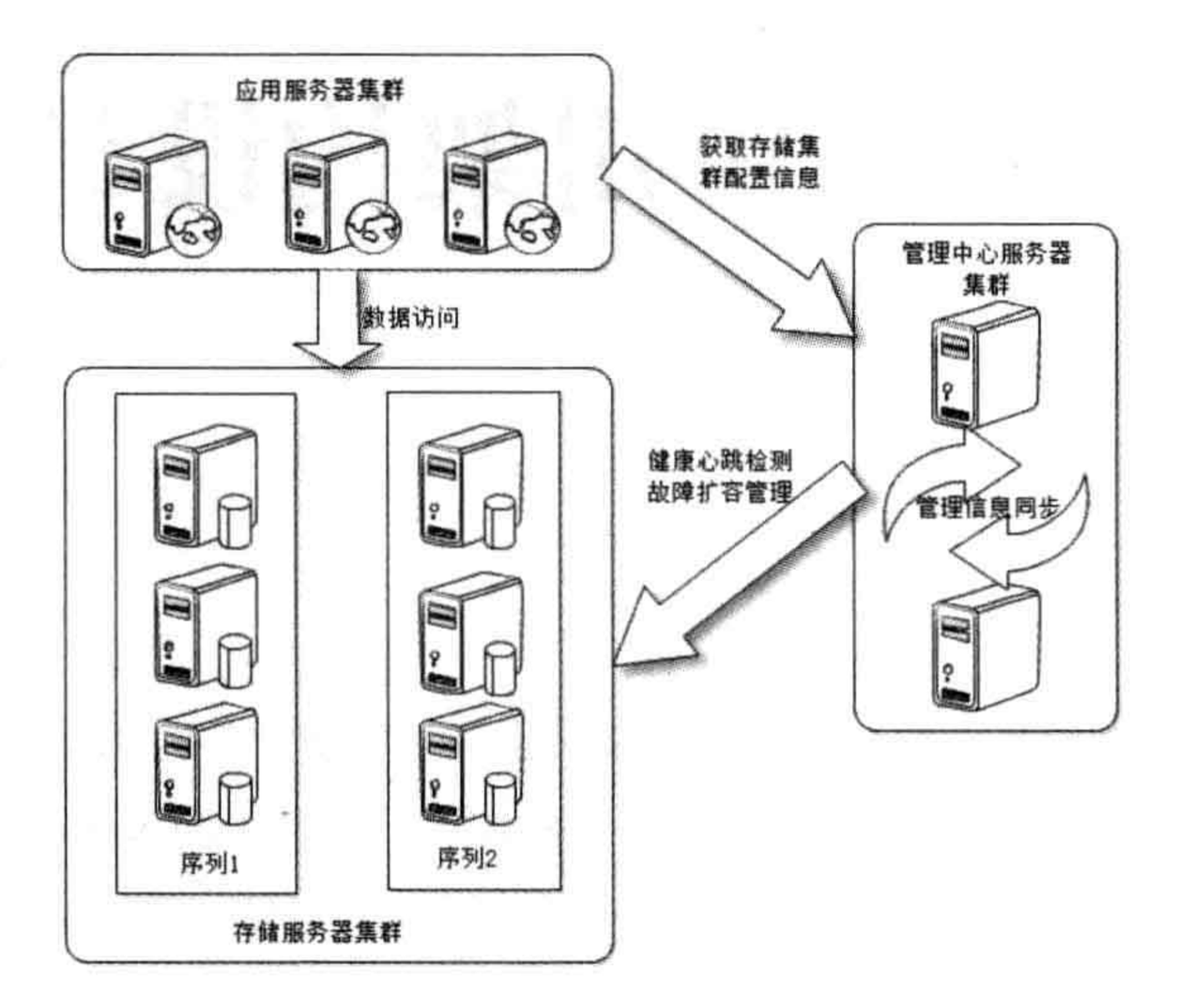

图 11.1 Doris 的整体架构

系统整体上可分为如下三个部分。

- **应用程序服务器：**它们是存储系统的客户，对系统发起数据操作请求。
- **数据存储服务器：**他们是存储系统的核心，负责存储数据、响应应用服务器的数据操作请求。
- **管理中心服务器：**这是一个由两台机器组成的主-主热备的小规模服务器集群，主要负责集群管理，对数据存储集群进行健康心跳检测；集群扩容、故障恢复管理；对应用程序服务器提供集群地址配置信息服务等。

其中数据存储服务器又根据应用的可用性级别设置数据复制份数，即每个数据实际物理存储的副本数目，副本份数越多，可用性级别越高，当然需要的服务器也越多。为

了便于管理和访问数据的多个副本，将存储服务器划分为多个序列，数据的多个副本存储在不同的序列中（序列可以理解为存储集群中的子集群）。

应用服务器写入数据时，根据集群配置和应用可用性级别使用路由算法在每个序列中计算得到一台服务器，然后同时并发写入这些服务器中；应用服务器读取数据时，只需要随机选择一个序列，根据相同路由算法计算得到服务器编号和地址，即可读取。通常情况下，系统最少写入的副本份数是两份，如图 11.2 所示。

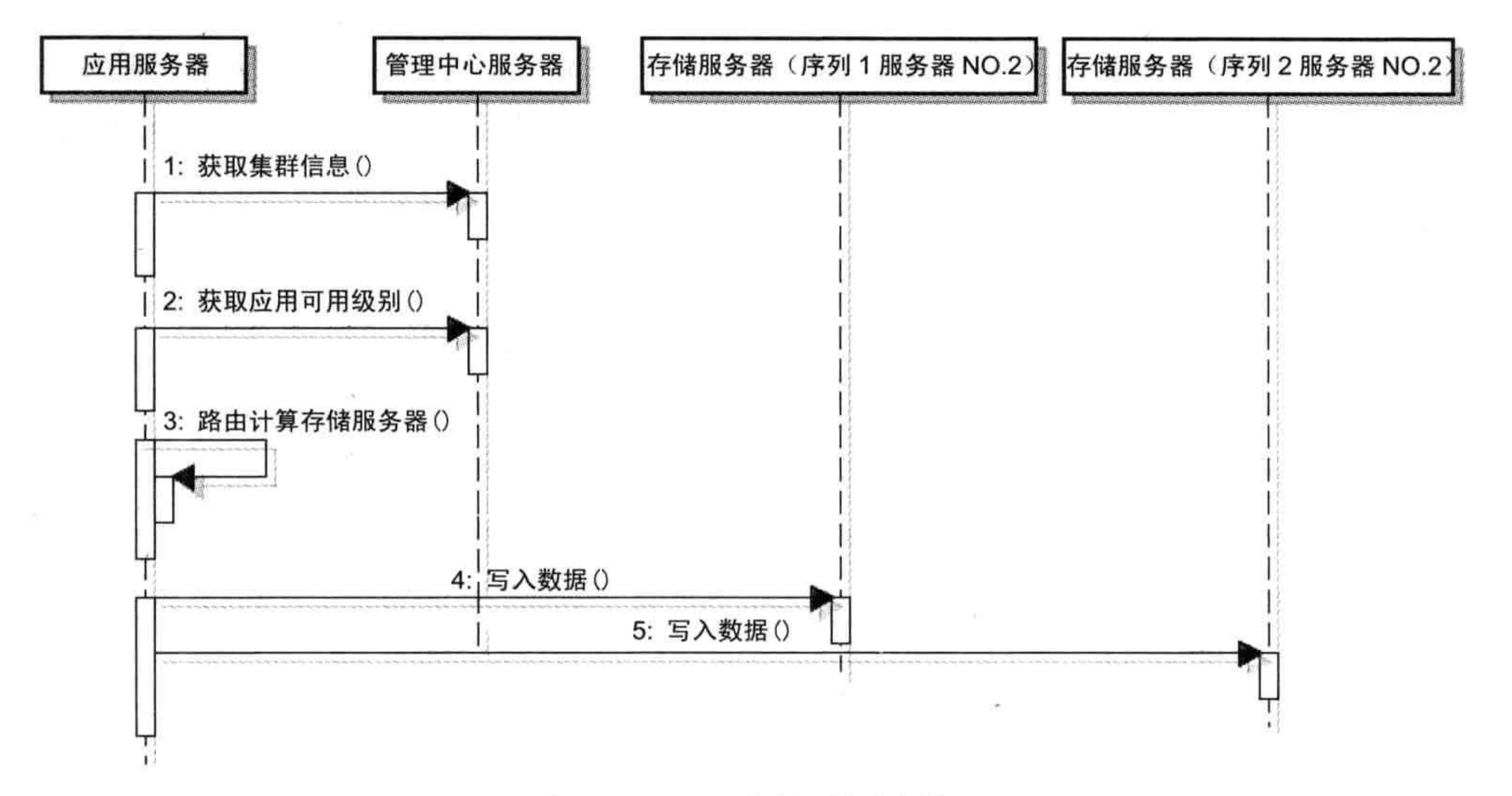

图 11.2 Doris 系统调用时序模型

在正常状态下，存储服务器集群中的服务器互不感知，不进行任何通信；应用服务器也只在启动时从管理中心服务器获取存储服务器集群信息，除非集群信息发生变化（故障、扩容），否则应用服务器不会和管理中心服务器通信。一般而言，服务器之间通信越少，就越少依赖，发生故障时互相影响就越少，集群的可用性就越高。

11.2 不同故障情况下的高可用解决方案

高可用的系统需要解决的是在不同故障情况下都保持较高的系统可用性，但是不同故障类型带来的问题复杂性不同，不可能使用一种解决方案处理所有情况，所以需要针对各种故障提供具体解决方案。

11.2.1 分布式存储系统的故障分类

在讨论解决方案之前，我们先对故障进行分类，针对不同故障情况分别对待。对于一个分布式存储系统而言，影响系统整体可用性的故障可以分成以下三类。

- **瞬时故障：**引起这类故障的主要原因是网络通信瞬时中断、服务器内存垃圾回收或后台线程繁忙停止数据访问操作响应。其特点是故障时间短，在秒级甚至毫秒级系统即可自行恢复正常响应。
- **临时故障：**引起这类故障的主要原因是交换机宕机、网卡松动等导致的网络通信中断；系统升级、停机维护等一般运维活动引起的服务关闭；内存损坏、CPU 过热等硬件原因导致的服务器宕机；这类故障的主要特点是需要人工干预（更换硬件、重启机器等）才能恢复正常。通常持续时间需要几十分钟甚至几小时。故障时间可分为两个阶段：临时故障期间，临时故障恢复期间。
- **永久故障：**引起这类故障的主要原因只有一个：硬盘损坏，数据丢失。虽然损坏硬盘和损坏内存一样，可以通过更换硬盘来重新启动机器，但是丢失的数据却永远找不回来了，因此其处理策略也和前面两种故障完全不同，恢复系统到正常状态也需要更长的时间。故障时间可分为两个阶段：永久故障期间和永久故障恢复期间。

11.2.2 正常情况下系统访问结构

在只使用两份副本作为高可用策略的情况下，系统访问结构如图 11.3 所示。

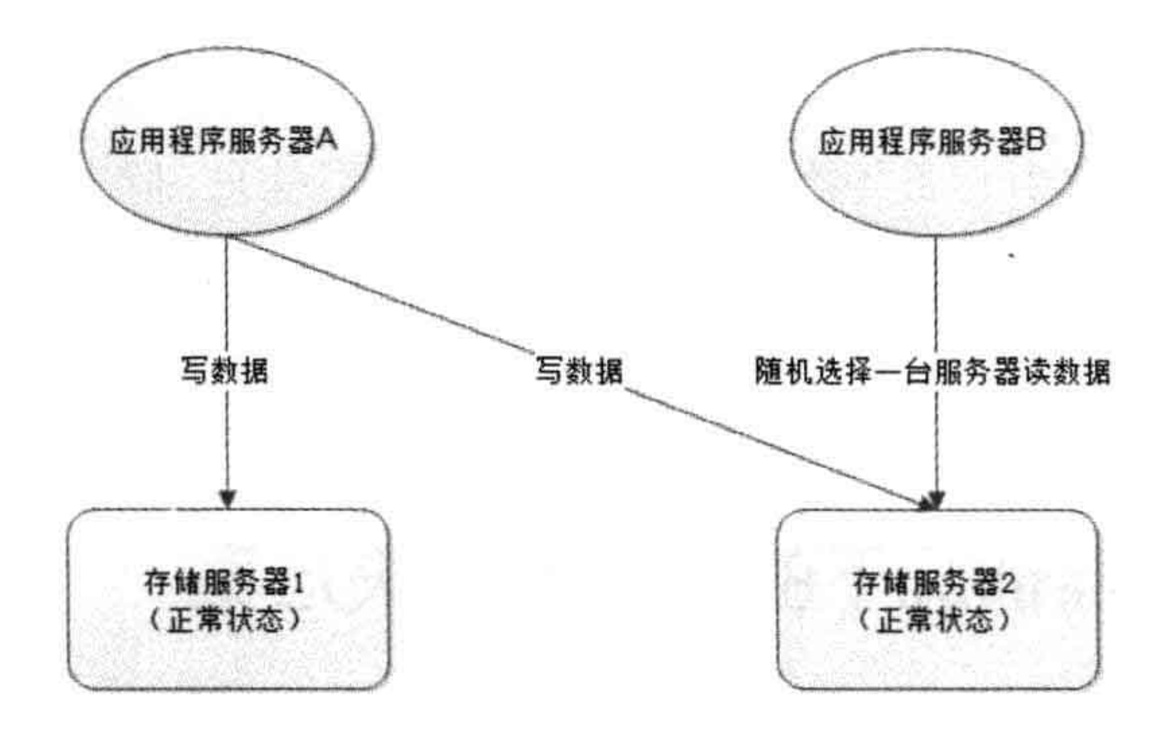

图 11.3 正常情况下 Doris 访问模型

应用程序在写数据时，需要路由计算获得两台不同的服务器，同时将数据写入两台服务器；而读数据时，只需要到这两台服务器上任意一台服务器读取即可。

11.2.3 瞬时故障的高可用解决方案

瞬时故障是一种严重性较低的故障，一般系统经过较短暂的时间即可自行恢复，遇到瞬时故障只需要多次重试，就可以重新连接到服务器，正常访问。如图 11.4 所示。

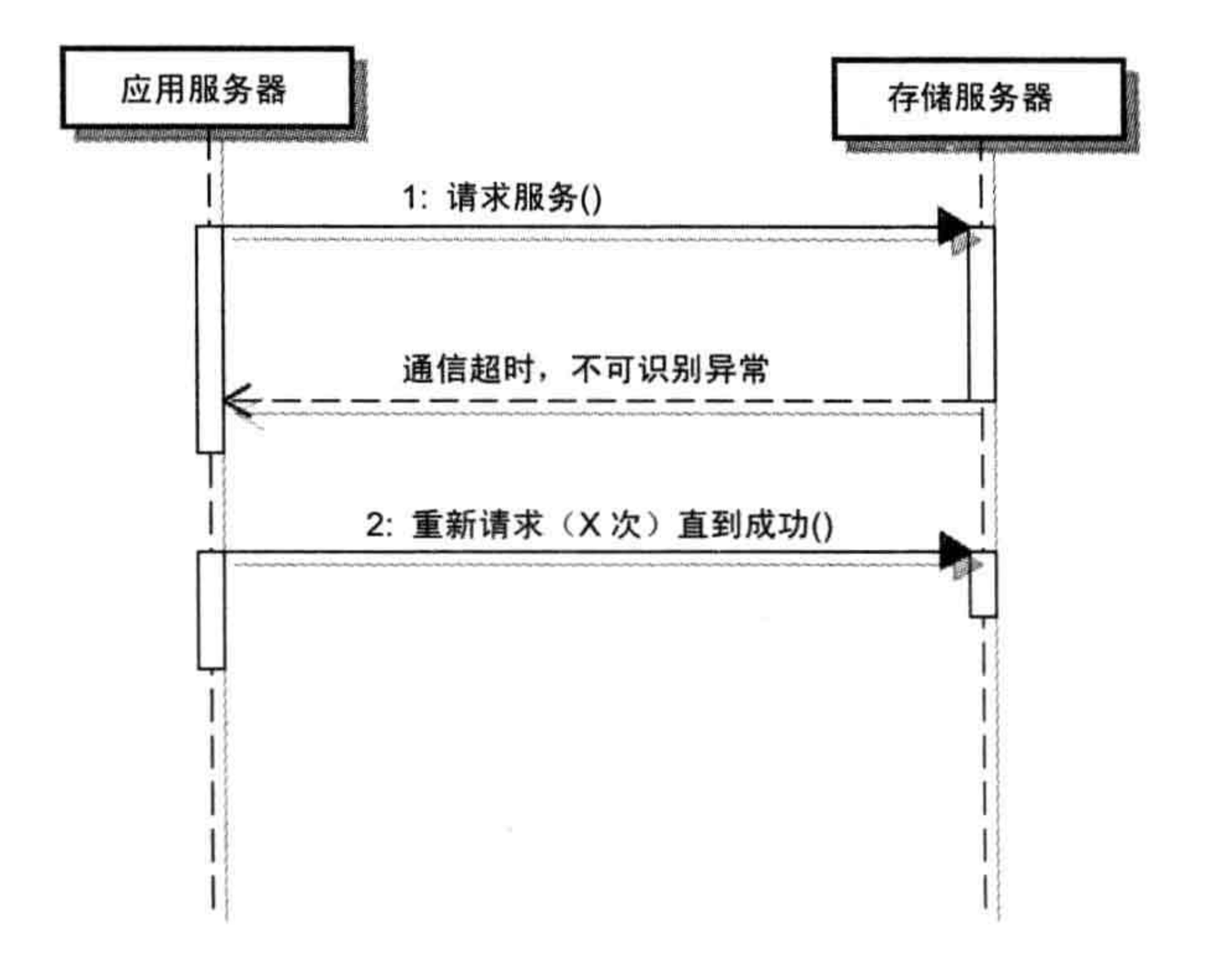

图 11.4 Doris 瞬时故障解决方案

如果经多次重试后，仍然失败，那么有可能不是瞬时故障，而是更严重的临时故障，这时需要执行临时故障处理策略。

当然也有可能是应用服务器自己的故障，比如系统文件句柄用光导致连接不能建立等，这时需要请求管理中心服务器进行故障仲裁，以判定故障种类。

瞬时失效访问模型如图 11.5 所示。

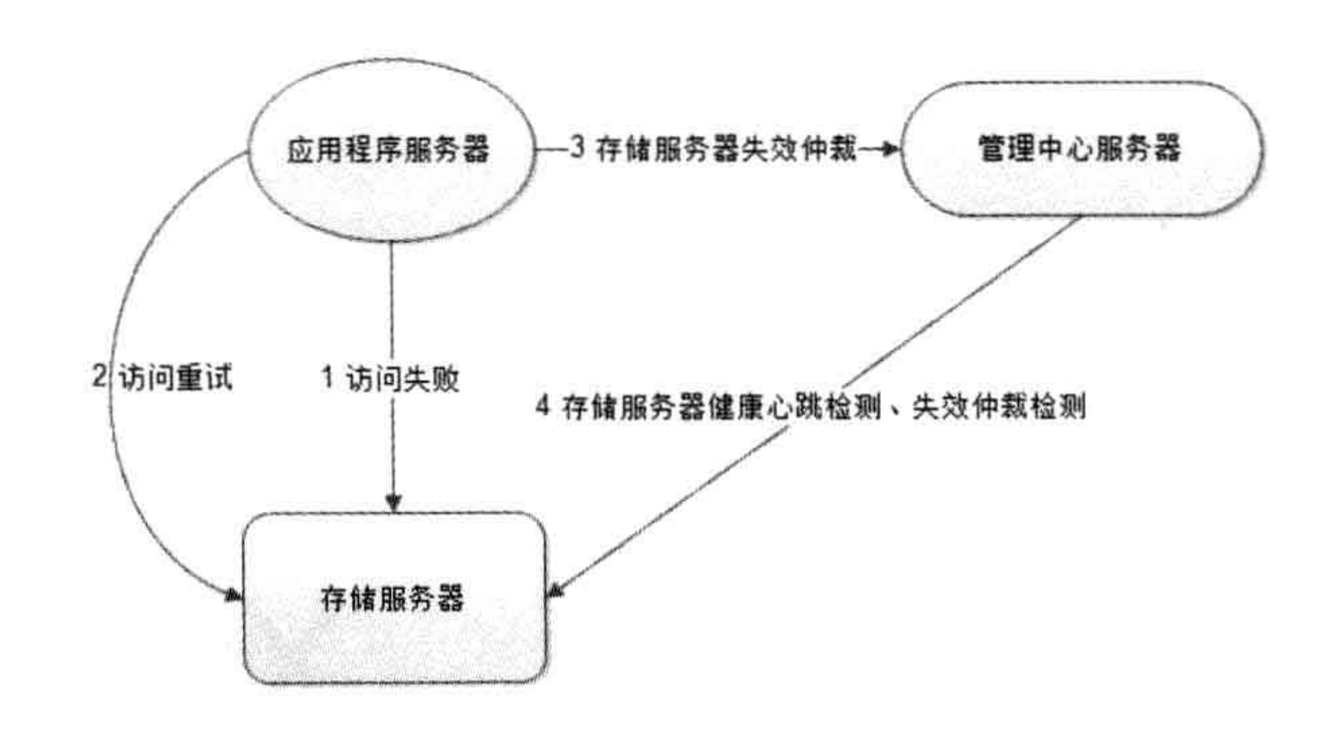

图 11.5 Doris 瞬时失效访问模型

11.2.4 临时故障的高可用解决方案

临时故障要比瞬时故障严重，系统需要人工干预才能恢复正常，在故障服务器未能恢复正常前，系统也必须保证高可用。由于数据有多份副本，因此读数据时只需要路由选择正常服务的机器即可；写数据时，正常服务的机器依然正常写入，发生故障的机器需要将数据写入到临时存储服务器，等待故障服务器恢复正常后再将临时服务器中的数据迁移到该机器，整个集群就恢复正常了。如图 11.6 所示。

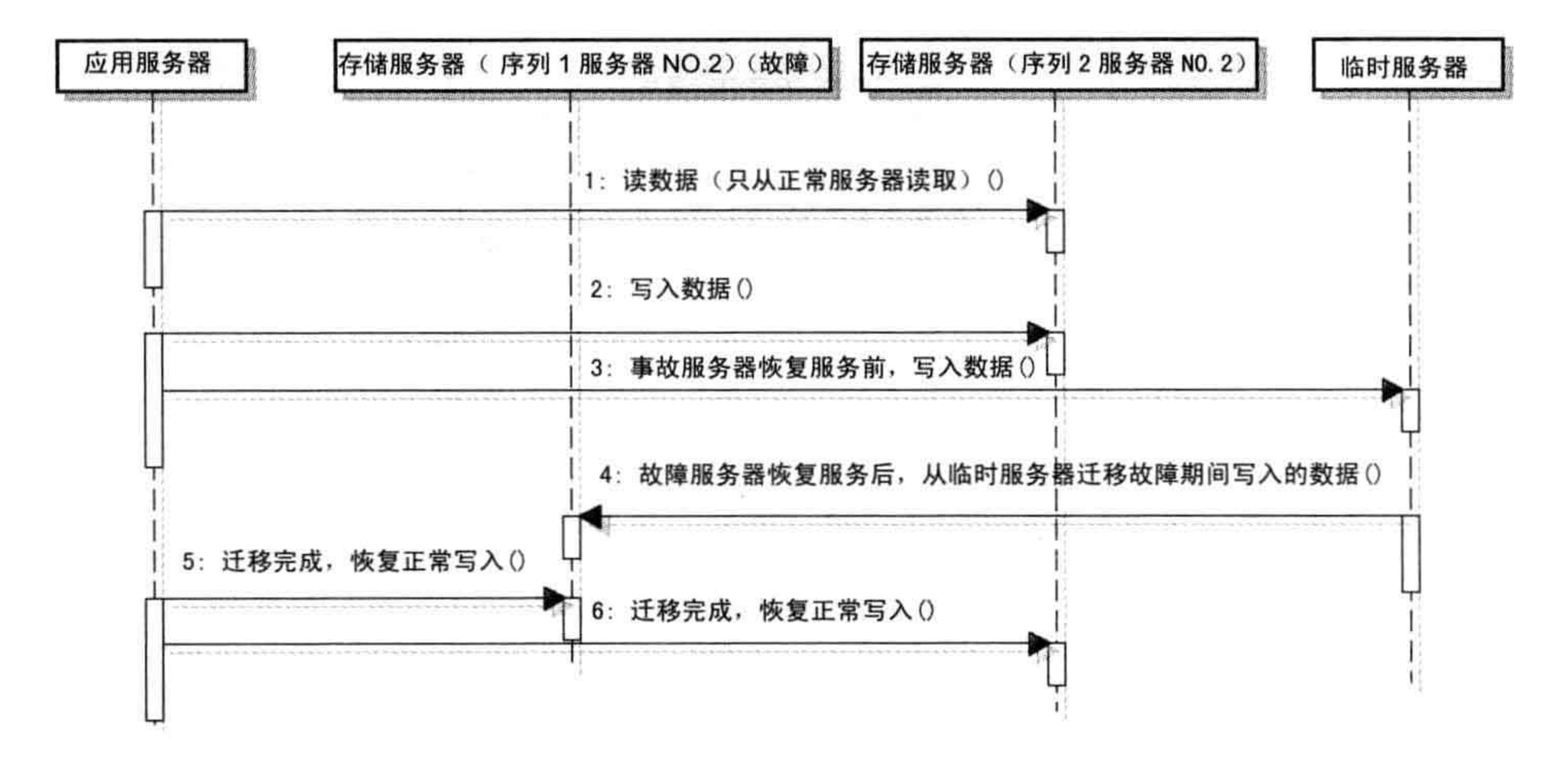

图 11.6 Doris 临时故障解决方案

其中临时服务器是集群中专门部署的服务器（根据可用性规划，临时服务器也可以部署为多台机器的集群），正常情况下，该服务器不会有数据写入，处于空闲状态，只有在临时失效的时候，才会写入数据。任何时候该服务器都不会提供读操作服务。

临时故障发生期间，系统访问模型如图 11.7 所示。

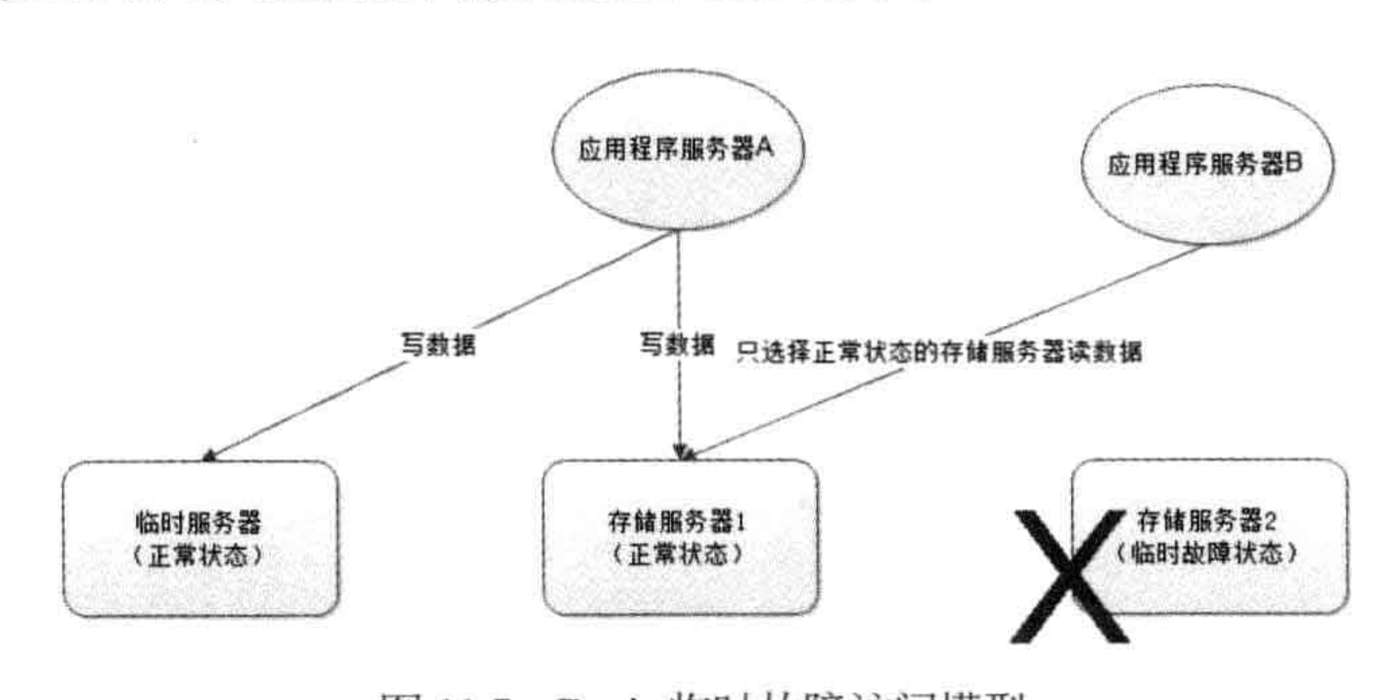

图 11.7 Doris 临时故障访问模型

临时故障解决，系统恢复期间，Doris 访问模型如图 11.8 所示。

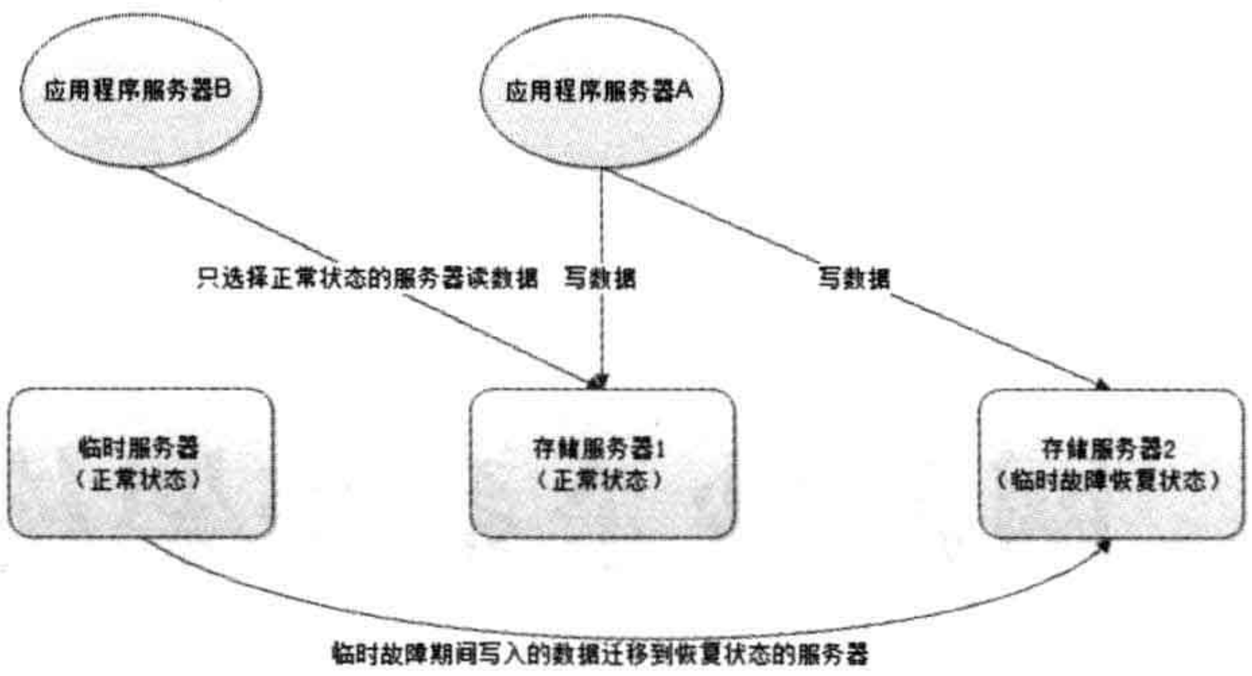

图 11.8 临时故障恢复期间 Doris 访问模型

临时故障期间写入临时服务器的数据全部迁移到存储服务器 2 后，故障全部恢复，存储服务器 2 恢复到正常状态，系统可按正常情况访问。

11.2.5 永久故障的高可用解决方案

永久故障是指服务器上的数据永久丢失，不能恢复。由于故障服务器上的数据永久丢失，从临时服务器迁移数据就没有意义了，必须要从其他序列中正常的服务器中复制全部数据才能恢复正常状态。

永久故障发生期间，由于系统无法判断该故障是临时故障还是永久故障，因此系统访问结构和临时故障一样。当系统出现临时故障超时（超过设定时间临时故障服务器仍没有启动）或者人工确认为永久故障时，系统启用备用服务器替代原来永久失效的服务器，进入永久故障恢复，访问模型如图 11.9 所示。

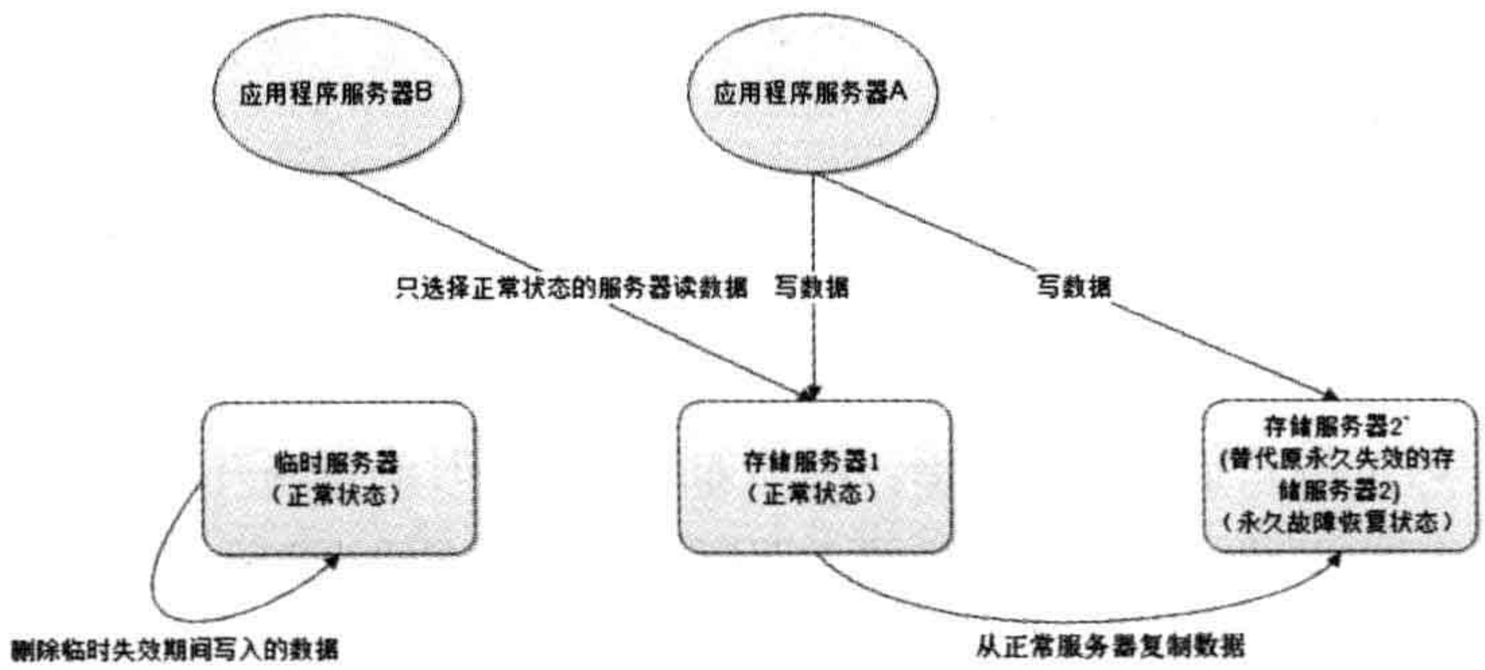

图 11.9 永久故障恢复期间 Doris 访问模型

12 网购秒杀系统架构设计案例分析

秒杀是电子商务网站常见的一种营销手段：将少量商品（通常只有一件）以极低的价格，在特定的时间点开始出售。比如一元钱的手机，五元钱的电脑，十元钱的汽车等。因为商品价格诱人，而且数量有限，所以很多人趋之若鹜，在秒杀活动开始前涌入网站，等到秒杀活动开始的一瞬间，点下购买按钮（在此之前购买按钮为灰色，不可以点击），抢购商品。这些商品因为在活动开始的一秒内就被卖光了，所以被称作秒杀。

网站通过这种营销手段，制造某种轰动效应，从而达到网站推广的目的。而最终能够被幸运之神眷顾，秒到商品的只有一两个人而已。很多电子商务网站已经把秒杀活动常态化了，经常性地举行秒杀活动。

秒杀虽然对网站推广有很多好处，也能给消费者带来利益（虽然是很少的几个人），但是对网站技术却是极大的挑战：网站是为正常运营设计的，而秒杀活动带来的并发访问用户却是平时的数百倍甚至上千倍。网站如果为秒杀时的最高并发访问量进行设计部署，就需要比正常运营多得多的服务器，而这些服务器在绝大部分时候都是用不着的，浪费惊人。所以网站的秒杀业务不能使用正常的网站业务流程，也不能和正常的网站交

易业务共用服务器，必须设计部署专门的秒杀系统，进行专门应对。

12.1 秒杀活动的技术挑战

假设某网站秒杀活动只推出一件商品，预计会吸引 1 万人参加活动，也就是说最大并发请求数是 10,000，秒杀系统需要面对的技术挑战有如下几点。

1. 对现有网站业务造成冲击

秒杀活动只是网站营销的一个附加活动，这个活动具有时间短，并发访问量大的特点，如果和网站原有应用部署在一起，必然会对现有业务造成冲击，稍有不慎可能导致整个网站瘫痪。

2. 高并发下的应用、数据库负载

用户在秒杀开始前，通过不停刷新浏览器页面以保证不会错过秒杀，这些请求如果按照一般的网站应用架构，访问应用服务器、连接数据库，会对应用服务器和数据库服务器造成极大的负载压力。

3. 突然增加的网络及服务器带宽

假设商品页面大小 200K（主要是商品图片大小），那么需要的网络和服务器带宽是 2G（200K × 10,000），这些网络带宽是因为秒杀活动新增的，超过网站平时使用的带宽。

4. 直接下单

秒杀的游戏规则是到了秒杀时间才能开始对商品下单购买，在此时间点之前，只能浏览商品信息，不能下单。而下单页面也是一个普通的 URL，如果得到这个 URL，不用等到秒杀开始就可以下单了。

12.2 秒杀系统的应对策略

为了应对上述挑战，秒杀系统的应对策略有如下几点。

1. 秒杀系统独立部署

为了避免因为秒杀活动的高并发访问而拖垮整个网站，使整个网站不必面对蜂拥而来的用户访问，可将秒杀系统独立部署；如果需要，还可以使用独立的域名，使其与网站完全隔离，即使秒杀系统崩溃了，也不会对网站造成任何影响。

2. 秒杀商品页面静态化

重新设计秒杀商品页面，不使用网站原来的商品详情页面，页面内容静态化：将商品描述、商品参数、成交记录和用户评价全部写入一个静态页面，用户请求不需要经过应用服务器的业务逻辑处理，也不需要访问数据库。所以秒杀商品服务不需要部署动态的 Web 服务器和数据库服务器。

3. 租借秒杀活动网络带宽

因为秒杀新增的网络带宽，必须和运营商重新购买或者租借。为了减轻网站服务器的压力，需要将秒杀商品页面缓存在 CDN，同样需要和 CDN 服务商临时租借新增的出口带宽。

4. 动态生成随机下单页面 URL

为了避免用户直接访问下单页面 URL，需要将该 URL 动态化，即使秒杀系统的开发者也无法在秒杀开始前访问下单页面的 URL。办法是在下单页面 URL 加入由服务器端生成的随机数作为参数，在秒杀开始的时候才能得到。

12.3 秒杀系统架构设计

秒杀系统为秒杀而设计，不同于一般的网购行为，参与秒杀活动的用户更关心地是如何能快速刷新商品页面，在秒杀开始的时候抢先进入下单页面，而不是商品详情等用户体验细节，因此秒杀系统的页面设计应尽可能简单。如图 12.1 所示。

商品页面中的购买按钮只有在秒杀活动开始的时候才变亮，在此之前及秒杀商品卖出后，该按钮都是灰色的，不可以点击。

图 12.1　秒杀商品页面

下单表单也尽可能简单，购买数量只能是一个且不可以修改，送货地址和付款方式都使用用户默认设置，没有默认也可以不填，允许等订单提交后修改；只有第一个提交的订单发送给网站的订单子系统，其余用户提交订单后只能看到秒杀结束页面，如图 12.2 所示。

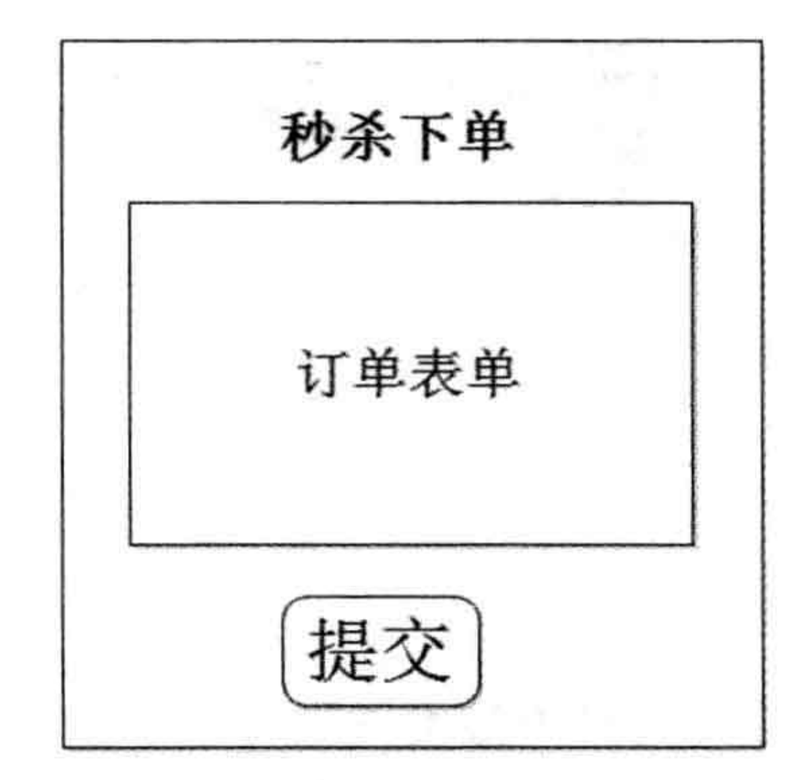

图 12.2　秒杀下单页面

除了上面提到的秒杀系统的技术挑战及应对策略，还有一些其他问题需要处理。

1. 如何控制秒杀商品页面购买按钮的点亮

购买按钮只有在秒杀活动开始的时候才能点亮，在此之前是灰色的。如果该页面是动态生成的，当然可以在服务器端构造响应页面输出，控制该按钮是灰色还是点亮，但是为了减轻服务器端负载压力，更好地利用 CDN、反向代理等性能优化手段，该页面被设计为静态页面，缓存在 CDN、反向代理服务器上，甚至用户浏览器上。秒杀开始时，用户刷新页面，请求根本不会到达应用服务器。

解决办法是使用 JavaScript 脚本控制，在秒杀商品静态页面中加入一个 JavaScript 文件引用，该 JavaScript 文件中加入秒杀是否开始的标志和下单页面 URL 的随机数参数，当秒杀开始的时候生成一个新的 JavaScript 文件并被用户浏览器加载，控制秒杀商品页面的展示。这个 JavaScript 文件使用随机版本号，并且不被浏览器、CDN 和反向代理服务器缓存。如图 12.3 所示。

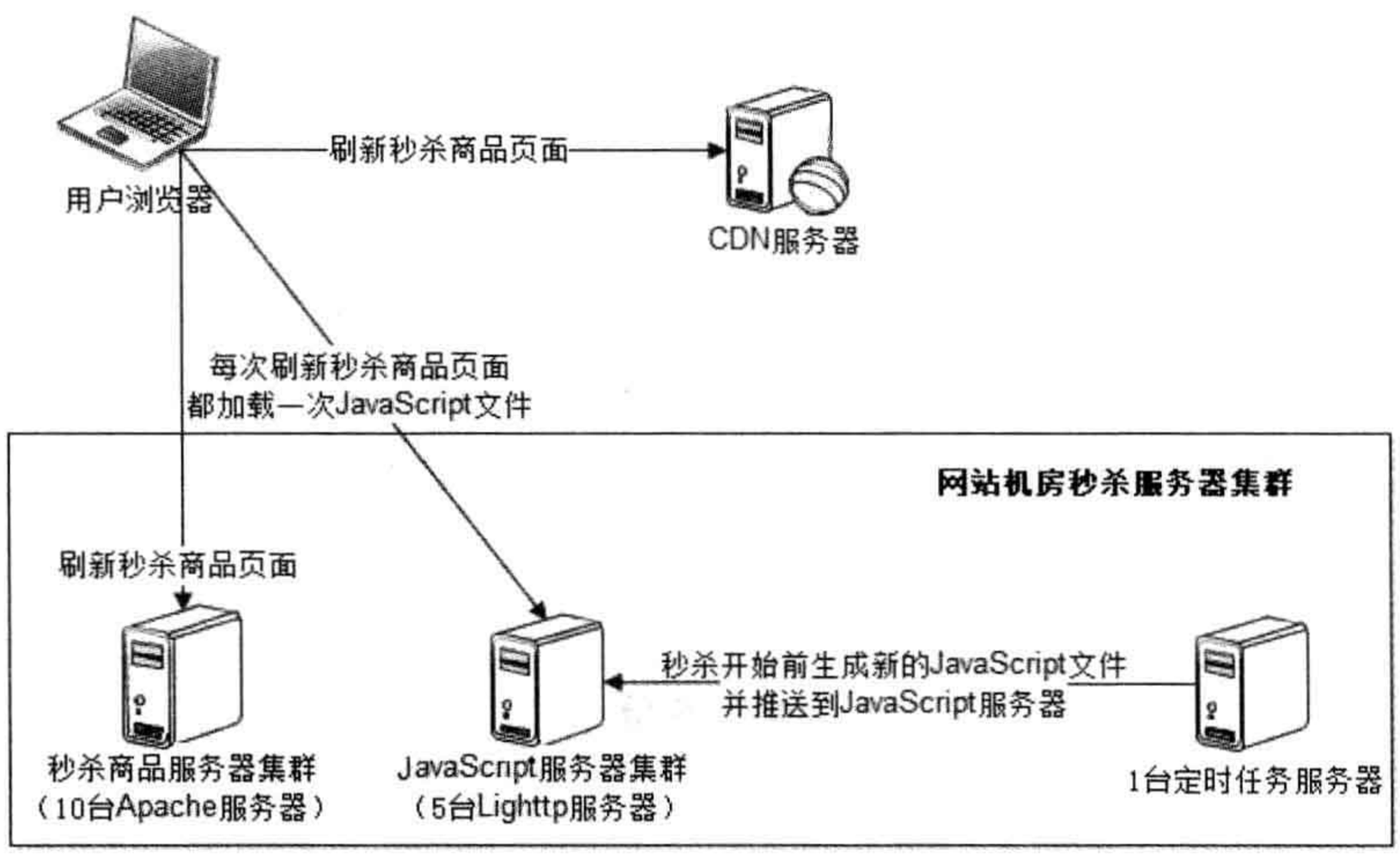

图 12.3　秒杀商品点亮过程

这个 JavaScript 文件非常小，即使每次浏览器刷新都访问 JavaScript 文件服务器也不会对服务器集群和网络带宽造成太大压力。

2. 如何只允许第一个提交的订单被发送到订单子系统

由于最终能够成功秒杀到商品的用户只有一个，因此需要在用户提交订单时，检查是否已经有订单提交。事实上，由于最终能够成功提交订单的用户只有一个，为了减轻下单页面服务器的负载压力，可以控制进入下单页面的入口，只有少数用户能进入下单页面，其他用户直接进入秒杀结束页面。假设下单服务器集群有 10 台服务器，每台服务器只接受最多 10 个下单请求，如图 12.4 所示。

秒杀系统的整体架构如图 12.5 所示。

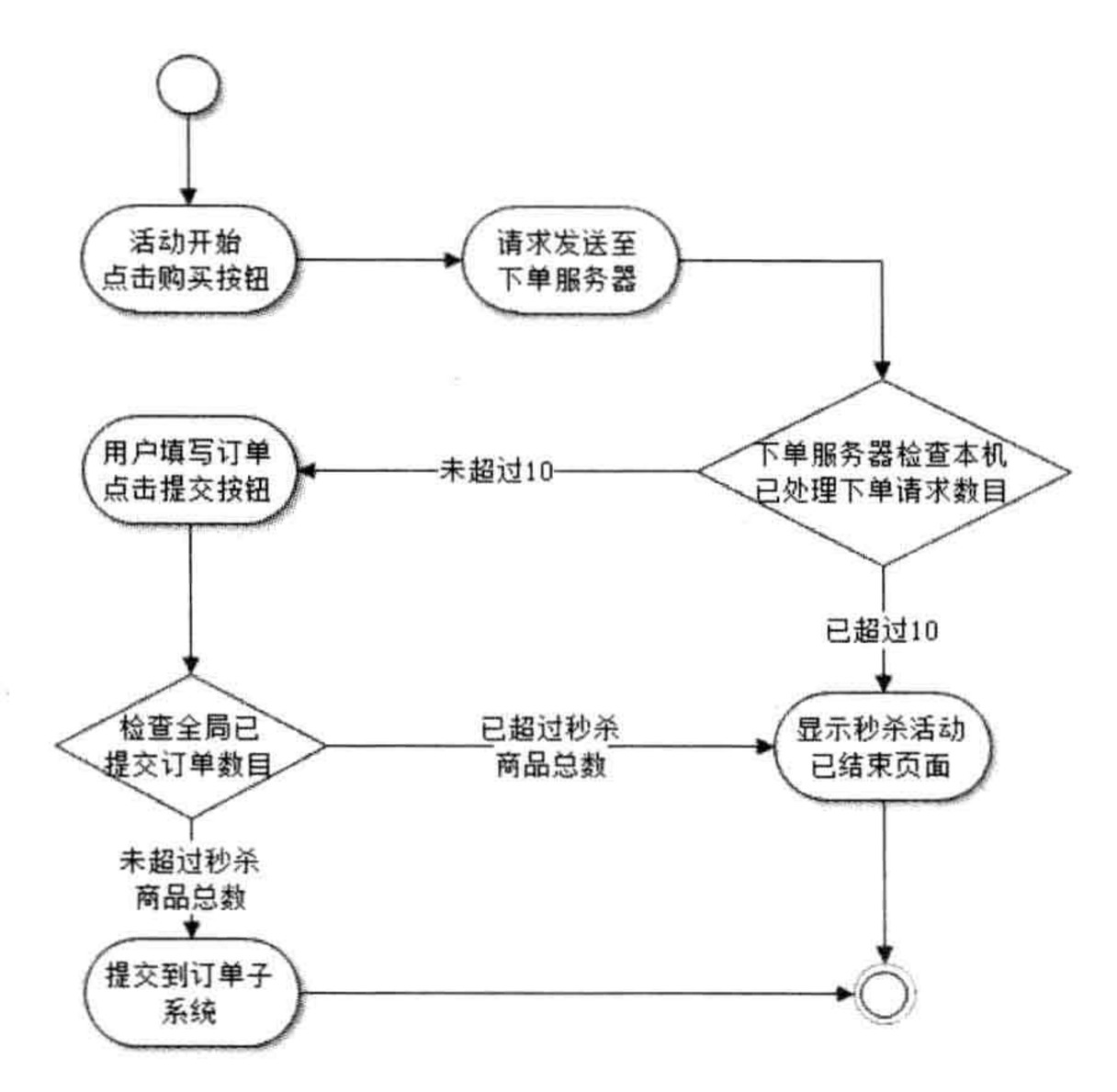

图 12.4　秒杀下单流程

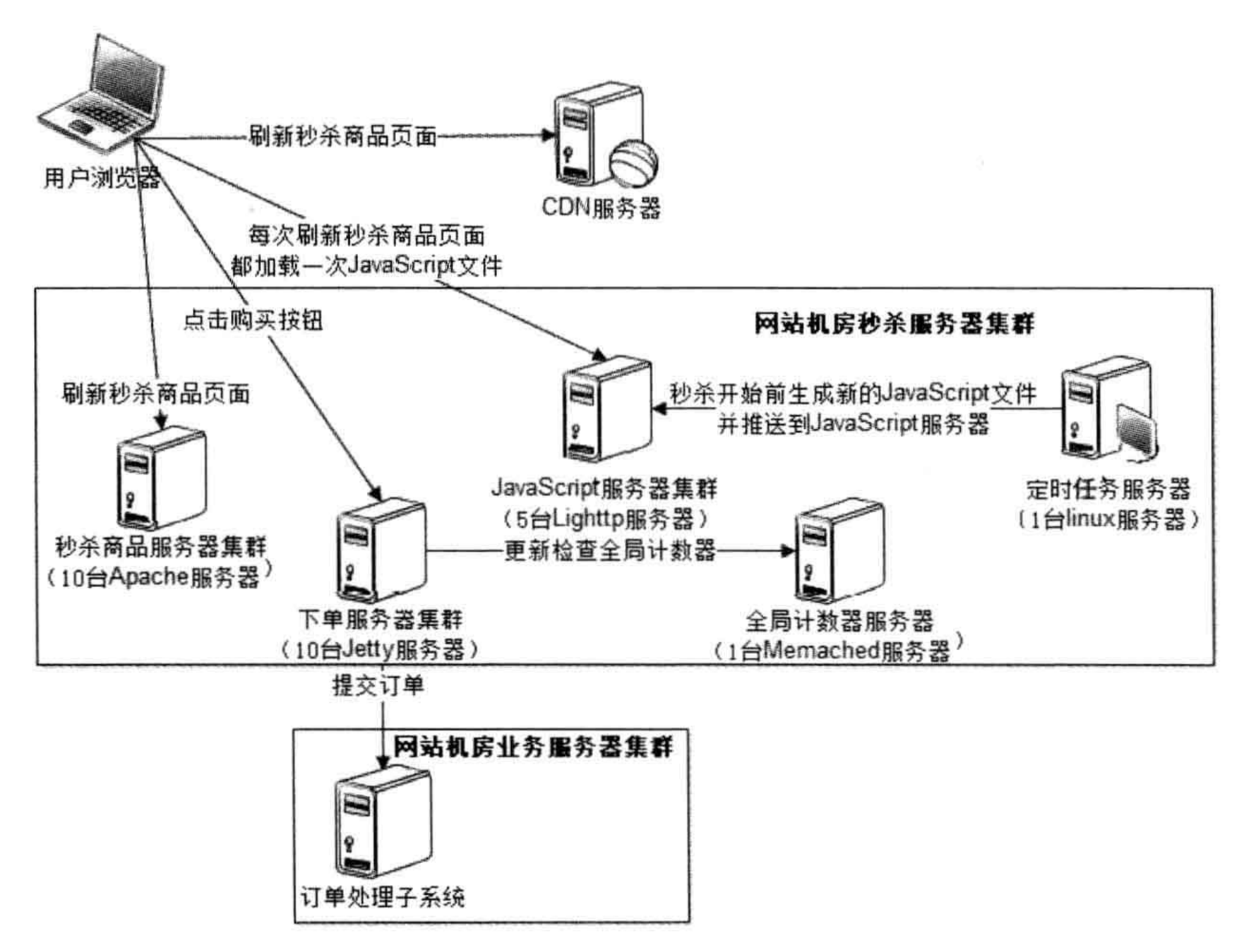

图 12.5　秒杀系统的整体架构

12.4 小结

秒杀是对网站架构的极大考验，在难以预计和控制的高并发访问的冲击下，稍有不慎，系统就会被用户秒杀，导致整个系统宕机，活动失败，构成重大事故。因此在遵循秒杀活动游戏规则的基础上，为了保证系统的安全，保持适度的公平公正即可。即使系统出了故障，也不应该给用户显示出错页面，而是显示秒杀活动结束页面，避免不必要的困扰。除了上面提到的一些针对秒杀活动进行的架构设计，在本书第 10 章中提到的许多性能优化设计都可以用于秒杀系统的优化。

13 大型网站典型故障案例分析

先讲一个小故事。有一次，笔者和几个网站架构师讨论问题，期间，一位架构师被他们部门总监叫去面试一位应聘者。结果过了十几分钟，这位架构师就回来了，我们都很奇怪：怎么这么快？他笑道："这位老兄，工作十几年，什么都不会，没什么好问的"。我们问他都问了什么问题，结果都是一些如果没有经历过，就永远不会想到的问题，而这些问题只要在大型网站技术一线呆上两三年，就一定会碰到。

一位网站资深架构师曾经说过：在互联网公司呆一年，相当于在传统软件公司呆三年。他的意思大概是在互联网公司一年遇到的问题比传统软件公司三年遇到的问题还多。而且随着网站业务的快速发展，问题也层出不穷，每年遇到的问题都不同。遇到问题，解决问题，经历了这个过程，技术才能升华，人和技术才能融为一体，才知道什么技术是真正有用的，什么技术是花拳绣腿。

大型网站的技术本质都很简单，没有很花哨的东西，掌握起来也不难。大型网站的架构师最有价值的地方不在于他们掌握了多少技术，而在于他们经历过多少故障。每一次故障都会给公司带来难以估计的利益损失，所以培养一个网站架构师的成本不单要看

付了他多少薪水，给了他多少股票，还要看为他引起的故障买了多少次单。

这里列举一些网站的典型故障，我们会发现，在高并发和海量数据的情况下，很多一般情况下不是问题的问题都会涌现出来。

13.1 写日志也会引发故障

故障现象：某应用服务器集群发布后不久就出现多台服务器相继报警，硬盘可用空间低于警戒值，并且很快有服务器宕机。登录到线上服务器，发现 log 文件夹里的文件迅速增加，不断消耗磁盘空间。

原因分析：这是一个普通的应用服务器集群，不需要存储数据，因此服务器里配置的是一块 100GB的小硬盘，安装完操作系统、Web 服务器、Java 虚拟机、应用程序后，空闲空间只有几十 GB 了，正常情况下这些磁盘空间足够了，但是该应用的开发人员将 log 输出的 level 全局配置为 Debug。这样一次简单的 Web 请求就会产生大量的 log 文件输出，在高并发的用户请求下，很快就消耗完不多的磁盘空间。

经验教训：

- 应用程序自己的日志输出配置和第三方组件日志输出要分别配置。
- 检查 log 配置文件，日志输出级别至少为 Warn，并且检查 log 输出代码调用，调用级别要符合其真实日志级别。
- 有些开源的第三方组件也会不恰当地输出太多的 Error 日志，需要关闭这些第三方库的日志输出，至于哪些第三方库有问题，只有在遇到问题时才知道。

13.2 高并发访问数据库引发的故障

故障现象：某应用发布后，数据库 Load 居高不下，远超过正常水平，持续报警。

原因分析：检查数据库，发现报警是因为某条 SQL 引起的，这条 SQL 是一条简单的有索引的数据查询，不应该引发报警。继续检查，发现这条 SQL 执行频率非常高，远远超过正常水平。追查这条 SQL，发现被网站首页应用调用，首页是被访问最频繁的网页，这条 SQL 被首页调用，也就被频繁执行了。

经验教训：

- 首页不应该访问数据库，首页需要的数据可以从缓存服务器或者搜索引擎服务器获取。
- 首页最好是静态的。

13.3 高并发情况下锁引发的故障

故障现象：某应用服务器不定时地因为响应超时而报警，但是很快又超时解除，恢复正常，如此反复，让运维人员非常苦恼。

原因分析：程序中某个单例对象（singleton object）中多处使用了 synchronized（this），由于 this 对象只有一个，所有的并发请求都要排队获得这唯一的一把锁。一般情况下，都是一些简单操作，获得锁，迅速完成操作，释放锁，不会引起线程排队。但是某个需要远程调用的操作也被加了 synchronized（this），这个操作只是偶尔会被执行，但是每次执行都需要较长的时间才能完成，这段时间锁被占用，所有的用户线程都要等待，响应超时，这个操作执行完后释放锁，其他线程迅速执行，超时解除。

经验教训：

- 使用锁操作要谨慎。

13.4 缓存引发的故障

故障现象：没有新应用发布，但是数据库服务器突然 Load 飙升，并很快失去响应。DBA 将数据库访问切换到备机，Load 也很快飙升，并失去响应。最终引发网站全部瘫痪。

原因分析：缓存服务器在网站服务器集群中的地位一直比较低，服务器配置和管理级别都比其他服务器要低一些。人们都认为缓存是改善性能的手段，丢失一些缓存也没什么问题，有时候关闭一两台缓存服务器也确实对应用没有明显影响，所以长期疏于管理缓存服务器。结果这次一个缺乏经验的工程师关闭了缓存服务器集群中全部的十几台 Memcached 服务器，导致了网站全部瘫痪的重大事故。

经验教训：

- 当缓存已经不仅仅是改善性能，而是成为网站架构不可或缺的一部分时，对缓存的管理就需要提高到和其他服务器一样的级别。

13.5 应用启动不同步引发的故障

故障现象：某应用发布后，服务器立即崩溃。

原因分析：应用程序 Web 环境使用 Apache+JBoss 的模式，用户请求通过 Apache 转发 JBoss。在发布时，Apache 和 JBoss 同时启动，由于 JBoss 启动时需要加载很多应用并初始化，花费时间较长，结果 JBoss 还没有完全启动，Apache 就已经启动完毕开始接收用户请求，大量请求阻塞在 JBoss 进程中，最终导致 JBoss 崩溃。除了这种 Apache 和 JBoss 启动不同步的情况，网站还有很多类似的场景，都需要后台服务准备好，前台应用才能启动，否则就会导致故障。这种情况被内部人戏称作“姑娘们还没穿好衣服，老鸨就开门迎客了”。

经验教训：

- 老鸨开门前要检查下姑娘们是否穿好了衣服。就本例来说，在应用程序中加入一个特定的动态页面（比如只返回 OK 两个字母），启动脚本先启动 JBoss，然后在脚本中不断用 curl 命令访问这个特定页面，直到收到 OK，才启动 Apache。

13.6 大文件读写独占磁盘引发的故障

故障现象：某应用主要功能是管理用户图片，接到部分用户投诉，表示上传图片非常慢，原来只需要一两秒，现在需要几十秒，有时等半天结果浏览器显示服务器超时。

原因分析：图片需要使用存储，最有可能出错的地方是存储服务器。检查存储服务器，发现大部分文件只有几百 KB，而有几个文件非常大，有数百兆，读写这些大文件一次需要几十秒，这段时间，磁盘基本被这个文件操作独占，导致其他用户的文件操作缓慢。

经验教训：

- 存储的使用需要根据不同文件类型和用途进行管理，图片都是小文件，应该使用专用的存储服务器，不能和大文件共用存储。批处理用的大文件可以使用其他类型的分布式文件系统。

13.7 滥用生产环境引发的故障

故障现象：监控发现某个时段内，某些应用突然变慢，内部网络访问延迟非常厉害。

原因分析：检查发现，该时段内网卡流量也下降，但是没有找到原因。过了一阵子才知道，原来有工程师在线上生产环境进行性能压力测试，占用了大部分交换机带宽。

经验教训：

- 访问线上生产环境要规范，不小心就会导致大事故。

网站数据库有专门的 DBA 维护，如果发现数据库存在错误记录，需要进行数据订正，必须走数据订正流程，申请 DBA 协助。于是就有工程师为避免麻烦，直接写一段数据库更新操作的代码，悄悄放到生产环境应用服务器上执行，神不知鬼不觉地订正了数据。但是如果不小心写错了 SQL，后果可想而知。

13.8 不规范的流程引发的故障

故障现象：某应用发布后，数据库 Load 迅速飙升，超过报警值，回滚发布后报警消除。

原因分析：发现该应用发布后出现大量数据库读操作，而这些数据本来应该从分布式缓存读取。检查缓存，发现数据已经被缓存了。检查代码，发现访问缓存的那行代码被注释掉了。原来工程师在开发的时候，为了测试方便，特意注释掉读取缓存的代码，结果开发完成后忘记把注释去掉，直接提交到代码库被发布到线上环境。

经验教训：

- 代码提交前使用 diff 命令进行代码比较，确认没有提交不该提交的代码。

- 加强 code review，代码在正式提交前必须被至少一个其他工程师做过 code review，并且共同承担因代码引起的故障责任。

13.9 不好的编程习惯引发的故障

故障现象：某应用更新某功能后，有少量用户投诉无法正常访问该功能，一点击就显示出错信息。

原因分析：分析这些用户，都是第一次使用该功能，检查代码，发现程序根据历史使用记录构造一个对象，如果该对象为 null，就会导致 NullPointException。

经验教训：

- 程序在处理一个输入的对象时，如果不能明确该对象是否为空，必须做空指针判断。
- 程序在调用其他方法时，输入的对象尽量保证不是 null，必要时构造空对象（使用空对象模式）。

13.10 小结

有位软件技术前辈曾经说过“软件设计有两种风格，一种是将软件设计得很复杂，以使其缺陷没那么明显；一种是将软件设计得很简单，以使其没有明显的缺陷”。就笔者观察，这两种风格的软件工程师都大有人在，只是在互联网公司，后一种更多一些，因为即使是不明显的缺陷在网站的快速发展冲击下，也会很快凸显出来，令其“作者”疲于应对，狼狈不堪。吃一次亏，学一次乖，以后设计软件时就会设计得简单些，如果问题能够很快被发现，要解决也相对容易。

第 4 篇

架构师

14 架构师领导艺术

有一次，笔者以架构师的角色参与某个软件产品的开发，产品经过一年多的发展，已经发布了 2.0 版本，并在一些企业用户中成功实施。项目后期，由于产品整体架构设计比较合理，各个功能模块的扩展性良好，架构师基本没有什么事情可做，加上一些其他因素，笔者打算辞职。

但是当跟项目组成员宣布辞职的时候，大家很吃惊，纷纷表示挽留“你怎么可以走呢，你走了，我们怎么办呢？”

我说“其实你们已经不需要我了，2.0 版本新的功能架构都是你们自己设计的，最近两个月技术讨论会上，我甚至都不发言了，你们不是做的一样很好？”

但还是有人很失望地说“你在，我们就有了主心骨，你不说话就是表示赞成我们的设计，我们才敢这样搞，你走了，我们怎么办呢？”

架构师是软件开发组织中一个比较特殊的角色，除了架构设计，软件开发等技术类工作，通常还需要承担一些管理职能：规划产品路线、估算人力资源和时间资源、安排人员职责分工，确定计划里程碑点、指导工程师工作、过程风险评估与控制等。这些管理事务需要对产品技术架构、功能模块划分、技术风险都熟悉的架构师参与或直接负责。

在软件开发过程中，架构师除了实现技术架构，完成产品技术实现外，还需要和项

目组内外各种角色沟通协调，可以说架构师相当多的时间用在和人打交道上。处理好人的关系对架构和项目的成功至关重要。

架构师作为项目组最资深的专业技术人员，是项目组开发测试工程师的前辈。从架构师的身上，工程师可以看到自己的未来，因此架构师在做人做事方面需要严格要求自己，做好表率。

14.1 关注人而不是产品

一定要坚信：**一群优秀的人做一件他们热爱的事，一定能取得成功**。不管过程多么曲折，不管外人看来多么不可思议不靠谱。

所以最好的软件项目管理不是制订计划，组织资源，跟踪修正项目进展，对成员进行激励和惩罚，而是发掘项目组每个成员的优秀潜能，让大家理解并热爱软件产品最终的蓝图和愿景。每个人都是为实现自我价值而努力，不是为了领工资而工作。

一旦做到这一点，项目组每个成员都会自我驱动，自觉合作，寻找达成目标的最优路径并坚韧不拔地持续前进。整个过程中，不需要拙劣的胡萝卜和大棒，最好的奖励就是最终要达成的目标本身，最大的惩罚就是这个美好的目标没有实现。

这也是领导的真谛：**寻找一个值得共同奋斗的目标，营造一个让大家都能最大限度发挥自我价值的工作氛围**。

没有懒惰的员工，只有没被激发出来的激情。所有强迫员工加班的管理者都应该为自己的无能而羞愧。

14.2 发掘人的优秀

有些企业喜欢挖优秀的人，而不是去把自己打造成一个培养优秀人才的地方。殊不知：**是事情成就了人，而不是人成就了事**。指望优秀的人来帮自己成事，不如做成一件事让自己和参与的人都变得优秀。

在前面提到的那个项目中，有一位刚毕业不久的同学，分配给他的任务是调查某个

技术功能的实现。事实上这个功能已经有开源的代码实现，只需要将这些代码加入到项目中直接调用就可以了，但是为了让他有较多的时间熟悉项目和背景技术，我没有跟他说你去使用某个开源项目实现这个功能，而是说你调查下这个功能如何实现。

后来，这个同学不但找到了这个功能的开源实现，阅读了文档和代码，还针对我们项目的需求场景对代码做了优化，然后又将这些优化的代码提交给开源项目的作者，最后被合并到开源项目中。

可以说，他的工作不只是超出了我的期望，简直就是让我吃惊，这种吃惊在我的职业生涯中曾多次出现，很多人在工作中做出的卓越成果以及表现出来的优秀让我自愧不如。

大多数人，包括我们自己，都比自己以为的更优秀，有些优秀需要在合适的环境中才会被激发出来，比如做一些有挑战的事，和更优秀的人合作，抑或拥有了超越自我的勇气。

发掘人的优秀远比发掘优秀的人更有意义。

14.3 共享美好蓝图

架构师要和项目组全体成员共同描绘一个蓝图，这个蓝图是整个团队能够认同的，是团队共同奋斗的目标。

蓝图应该是表述清楚的：产品要做什么、不做什么、要达到什么业务目标，都需要描述清楚。

蓝图应该是形象的：产品能为用户创造什么价值、能实现什么样的市场目标、产品最终会长什么样，都需要形象地想象出来。

蓝图应该是简单的：不管内部还是外部沟通，都能一句话说明白：我们在做什么。

蓝图应该写在软件架构设计文档的扉页、写在邮件的签名档、写在内部即时通信群的公告上。

在项目过程中，架构师要**保持对目标蓝图的关注**，对任何偏离蓝图的设计和决定保

持警惕，错误的偏离要及时修正，必要的变更要经过大家讨论，并且需要重新获得大家的认同。

在电影《十月围城》中，一个年轻的革命党人说“我一闭上眼睛，就看到中国的明天”。这个明天就是辛亥革命的蓝图，为了这个美好的明天他愿意抛头颅、洒热血，死而无憾。创业者闭上眼睛就能看到企业的明天；软件产品的开发者闭上眼睛就能看到软件实现价值的那一刻。这就是蓝图的力量。

也许有人会说“你是在忽悠我吧，只是想让我努力工作而已”。青春总会逝去，人总是会死的，当有一天你白发苍苍回首往事，你会为无所事事而遗憾，但不会为被人忽悠而羞愧。批评马云忽悠的人，一定为马云在创建阿里巴巴的时候没有忽悠他成为创始人而遗憾。

14.4 共同参与架构

架构师需要对系统架构负责，但并不是说一定要架构师自己完成架构设计，并要项目团队严格遵守架构决策。

把架构和架构师凌驾于项目和项目组之上，只会让架构师变成孤家寡人，让架构曲高和寡。

1. 不要只有架构师一个人拥有架构

架构师不要把架构当做自己的私有财产，为了维护架构的纯洁和架构师的威信而不让他人染指架构。让项目参与者对架构充分争论，大家越是觉得自己是项目架构的重要贡献者，就越是愿意对开发过程承担责任，越是愿意共同维护架构和改善软件。

2. 让其他人维护框架与架构文档

框架是架构的重要组成部分，许多重要的架构设计通过框架实现来体现。但是在软件开发过程中，架构也需要根据需求不断发展演化，框架和架构文档也会随之调整。除非是重大的重构，否则架构师应该让项目组成员维护框架和架构文档，给项目组成员成长的机会也让自己有更多的时间去寻找更大的挑战。

14.5 学会妥协

不要企图在项目中证明自己是正确的，一定要记住，你是来做软件的，不是来当老大的。所以不要企图去证明自己了不起，永远也别干这种浪费时间、伤害感情的事。

有个小故事：猎人进山里打猎，反而被一头黑熊抓住了，黑熊说“如果你给我 XX 我就放你走”，猎人无奈只好给黑熊 XX。回去后苦练打猎本领，再次进山，结果又被黑熊抓住，再次要求给了 XX。第三次他又来了，黑熊看到他就乐了“你是来打猎的还是来给我 XX 的？”。

每次我在做项目迷失方向，五迷三道的时候，就会想起这个故事，提醒自己是来做软件的，来实现客户价值的，不是来证明谁对谁错的，不是来给黑熊 XX 的。

很多时候，对架构和技术方案的反对意见，其实意味着架构和技术方案被关注、被试图理解和接受。架构师不应该对意见过于敏感，这时架构师应该做的是坦率地分享自己的设计思路，让别人理解自己的想法并努力理解别人的想法，求同存异。

对于技术细节的争论应该立即验证而不是继续讨论，当讨论深入到技术细节的时候也意味着问题已经收敛，对于整体架构设计，各方意见正趋于一致。

而当大家不再讨论架构的时候，表明架构已经融入到项目、系统和开发者中了，架构师越早被项目组遗忘，越表示架构非常成功；项目组越离不开架构师，越表示架构还有很多缺陷。

14.6 成就他人

我们活着不是为了工作，不是为了做设计、写程序，这些不是我们生活的目的。我们活着是为了成就我们自己，而要想成就自己，就必须首先成就他人。

每个人都有自己成就的目标，而工作是达成自我成就的一种手段：通过工作的挑战，发掘自我的潜能，重新认知自我和世界。

软件开发过程是人的智力活动过程，软件开发不仅是制造软件的过程，也是开发人

员完善自我、超越自我的过程。所以我们工作不只是生产产品，还要成就人，并最终成就我们自己。

做成一个项目不但要给客户创造价值，为公司盈利，还要让项目成员获得成长。要让他们觉得通过这个项目，自己的知识技能和业务水平都得到了提高。项目结束时，大家会觉得不可思议："如此完美的产品，如此有挑战的开发居然都是我们完成的"。而且每个人都觉得自己在项目中至关重要不可或缺。

架构师作为团队的技术领导者，在项目过程中不要去试图控制什么，带着一个弹性的计划和蓝图推进，团队会管好他们自己。你越是强加禁令，队伍就越是没有纪律；你越是强制，团队就越是不能独立自主；你越是从外面寻找帮助，大家就越是没有信心。

而一旦打造出一个优秀的团队，在以后的合作中，面临更大的挑战时，架构师就可以从容应对，因为你不是一个人在战斗。同时一个优秀的团队内部也会发生化学反应，创造出超出工作本身的机会，开启更美好的明天。

15 网站架构师职场攻略

开发软件的目的是为了解决现实世界的问题，但是很多时候人们并不清楚真正的问题是什么。有可能大家很辛苦地忙活了一场，发现做出来的软件一点价值没有。

软件开发过程中也会遇到很多问题，需要协调各方面的利益关系获取尽可能大的支持，需要平衡客户需求、软件产出、开发资源之间的关系，需要搞定许多事情才能实现软件设计最初的蓝图。

网站架构师人在职场，需要处理好个人、团队、公司的利益。需要不断地在工作中发现问题，解决问题，提升工作经验、知识技能和核心竞争力，扩大自身影响力，达成工作绩效。

架构师 A 是刚加入公司的新人，参与网站基础技术产品的架构设计和开发。三个月后，他发现自己依然是个打杂的角色，在几个项目里承担不重要的角色，写一些不重要的代码。在之前的公司里，他是技术骨干，担任核心角色，而在这里，自己可有可无。而且现在公司里确实高手如云，好像没有解决不了的问题，不像以前，棘手问题非得自己出马才能搞定。

A 陷入了迷茫。

15.1 发现问题，寻找突破

其实即使是在一流的技术团队里，也一定有数不清的问题，只是人们习惯了这些问题，以至于无视它们的存在。正所谓“鱼是最后一个看见水的”，天天面对这些问题，反而不觉得有什么问题。

- 网站发布日加班不是正常的吗？
- 更新系统配置参数难道不需要重启系统吗？
- 服务器宕机，部署在上面的后台定时任务当然不会执行了。

我们在讨论如何发现问题，已经在脑补这三个问题解决方案的同学，请你把思绪拉回来。

而作为一个新人，以局外人的角度去观察，会发现许多存在的问题。

- 这个第三方程序包已经发布 3.2 版本了，有更好的性能和易用性，而我们还在使用 2.5 版本。
- 业界已经有很多公司在自动化运维方面取得了成功，而我们主要还是人工运维。
- 对于大多数应用，开源的 MySQL 数据库已经绰绰有余，而我们还在使用昂贵的 Oracle 数据库。

有些问题在被解决以后，人们才发现事情原来可以这样啊。淘宝出现之后，人们发现购物可以更便宜、更便捷；iPhone 出现之后，人们发现手机原来可以不光用来打电话发短信；而微信出现之后，人们才发现手机发短信甚至打电话竟然可以不花钱。

所谓问题，就是体验—期望，当体验不能满足期望，就会觉得出了问题。消除问题有两种手段：改善体验或者降低期望。降低期望只是回避了问题，而如果直面期望和体验之间的差距，就会发现问题所在，找到突破点。

A 仔细观察了一段时间后，发现并记录下很多网站技术中存在的问题，经过和团队成员一番沟通，去除了那些积重难返风险太大的、影响较小难出成绩的、已经有团队在做的，决定解决应用程序访问数据库时存在的安全漏洞问题，这个漏洞可能会导致数据库密码泄露，安全无小事，A 觉得可以以此为突破口，打开局面。

新员工 Tips

1. 许多刚加入公司的新员工一开始就急着要做出成绩，但是由于不熟悉环境，四处碰壁，被打消了积极性，反而不利于长远发展。其实新员工首先要做的事情是融入团队，跟大家打成一片，只要能和团队一起共进退，你就不是一个人在战斗。等熟悉了情况，知道了水的深浅后，再寻找突破口，择机而动。

2. 新员工最不需要做的事情就是证明自己的能力。在新环境中一时施展不开就怀疑自己的能力，进而担心被其他人怀疑自己的能力，于是努力想要证明自己，但是常常事与愿违，反而出乱子，伤害了公司和自己的利益。其实既然能经过层层考核和挑选进入公司，就已经证明你有和工作要求相匹配的能力，你要相信当初选中你的同事的眼光和能力。

A 做了一个技术产品提案，详细地描述了问题现状，架构设计，资源需求和产品路线图。这个提案得到了所在团队的支持，安排了两个工程师和他一起开发这个新产品。几个月后，一个可以实现数据库安全访问的程序包开发完成。

但是当他试图推广这个程序包在网站应用中使用的时候，遇到了一点问题：网站应用架构师和开发工程师并不乐意使用这个程序包，他们认为：

- 这个所谓的安全漏洞并不严重，也许技术上有漏洞，但是可以通过管理规范来弥补，网站运行这么多年从没有在这里出现安全问题。
- 这个程序包和以前的程序包调用方式不兼容，需要投入专门的开发资源去改造，网站业务很忙，没有资源。
- 这个程序包依赖复杂，可能会引入难以预料的可用性故障，网站今年已经出现过几次重大故障，不能再承受这种可用性故障的风险了。

软件开发出来，如果没有投入使用，就一点价值也没有，不管架构设计和代码实现多优美都没用。

做出软件不等于解决问题，事实上很多问题确实也不需要用软件来解决。

生产出来的软件实现不了价值，架构师就体现不出价值。

A 应该怎么办呢？

15.2 提出问题，寻求支持

问题被发现，它只是问题发现者的问题，而不是问题拥有者的问题，如果想要解决一个问题，就必须提出这个问题，让问题的拥有者知道问题的存在。

在这个例子中，谁拥有数据库访问密码安全漏洞的问题？或者说，谁需要对可能发生的数据库密码泄露问题负责？

这个问题并不是 A 的问题，他不负责网站安全，出了安全事故也不需要他负责，但是他既然发现了潜在的危险，就不能坐等事故发生后再去处理，而且他也确实需要一个突破口去树立自己的技术地位。

这个问题是网站各个产品线上应用架构师的问题，如果出现密码泄露问题，他们需要承担责任，但是他们认为这个问题不重要，并且通过管理规范的方式已经解决了。

A 如果想要推动它的解决方案被接受，就必须找到其他问题拥有者并愿意支持他的人。

经过一番思考，A 先后给安全总监和 CTO 写了邮件。

在给安全总监的邮件中，A 详细描述了问题场景、解决方案、架构设计和目前遇到的困难。

在给 CTO 的邮件中，A 用很短的篇幅重点描述了问题和现有管理规范的不足，以及可能产生的严重后果。

CTO 收到邮件后立即转发安全总监和网站技术总监。安全总监迅速回复，表示安全团队和基础技术团队已经展开合作并有解决方案，但是需要网站技术团队配合。网站技术总监也回复表示会积极配合推动此事。

A 得到了安全团队的支持和网站技术团队的配合，开始在网站各个应用上实施，使用新的数据库连接程序包。

但是很多网站工程师并不愿意积极配合，他们认为这件事对网站业务没有价值，对自己没有价值，徒增工作量而已。

A 的实施工作并不顺利。

提出问题 Tips

1．把“我的问题”表述成“我们的问题”

大多数人都不喜欢问题，问题意味着麻烦，当他听到你说“我遇到一个问题”的时候，下意识地要远离你和你的问题。如果你需要他的支持，就要想办法把你的问题变成他的问题，是他遇到了问题，而你来帮他解决。

在多数场合，严格区分“你的问题”还是“我的问题”意义不大，既然你身在其中，就是为了解决问题，所以这个时候把问题表述成“我们的问题”，会拉近彼此的距离。

“Tom，**我们**遇到了一个问题。”

“哦，坐下来，说说看。”

2．给上司提封闭式问题，给下属提开放式问题

不要问上司“你觉得该怎么办？”这种没有建设性的开放式问题，给上司提问题是希望能够得到他的支持，而不是带着一头雾水等他去指点迷津。公司付你薪水不是让你睁着迷茫的眼睛卖萌。给上司提问应该是“你觉得 A 和 B 两个方案哪个更好？”这种封闭式问题。

给下属提问则相反，用开放式的问题启发他去思考，寻找创新的解决方案。

所以，只有“元方，你怎么看？”，而没有“大人，你怎么看？”。

3．指出问题而不是批评人

如果在合作中出现问题，告诉他问题的存在和紧迫性，而不是责问他为什么出现问题。

人在听到批评信息的时候，本能地想要去针对批评进行反驳或者辩解，于是谈话就变成关于批评是否合理的争论，离解决问题越来越远。

4．用赞同的方式提出问题

在项目评审或者讨论问题的时候，发现对方的方案中存在缺陷，不要直言“你这里有问题”，对方可能会本能地进行自我保护而拒绝你的建议。

用一种温和的方式提出问题“我非常赞同你的方案，不过我有一个小小的建议……”。

所谓直言有讳是指想要表达的意图要直截了当说明白，不要兜圈子，但是在表达方式上要有所避讳，照顾到当事人的感受。

15.3 解决问题，达成绩效

A 在实施过程中发现，虽然网站工程师对这个提供安全管理的数据库连接程序不感兴趣，但是他们希望能改善现有数据库连接程序的性能并能够更容易地维护数据库连接的各种配置参数。

A 停止继续实施，花了几个星期的时间重构了数据库连接程序代码，将性能提高了 50%，并且将数据库密码和连接参数统一管理。在推广这个新版本程序包的时候，A 不再强调其安全性，而是着重宣传其高性能和易用性。

网站工程师看到新的数据库连接程序包不但性能更好，而且不需要管理数据库连接池的配置参数，省了以后维护的麻烦，很乐意接受使用这个新的程序包，很快就集成到应用程序中，部署到生产环境。

而数据库连接参数的配置和管理则提供了一个管理后台给数据库管理员（DBA），由他们统一维护，DBA 长久以来对各个应用不规范的连接方式头痛不已，现在有个工具可以让他们控制管理应用的数据库连接参数，都对此欣然叫好。

通过这个产品的实施，A 熟悉了网站业务，并与网站工程师建立了深厚的革命友谊，也树立了自己的技术威望。

解决问题 Tips

1. 在解决我的问题之前，先解决你的问题

在上面的案例中，推广数据库连接程序包是 A 的问题，而改善数据库连接性能和易用性是网站工程师的问题。在 A 成功地解决了数据库连接的性能和易用性问题后，推广数据库连接程序包自然就不是问题了。

先解决别人的问题有几个好处：

- 你帮别人解决了问题，礼尚往来，别人也会帮你解决问题。
- 在帮别人解决问题的过程中，熟悉了情况，后面解决自己的问题也就得心应手了。

- 解决别人的问题时使用的是你的解决方案，这个方案在你的控制之中，将来往这个方案里再塞一些东西解决自己的问题，手到擒来。

2. 适当的逃避问题

有时候，有些人会提出一些不怎么靠谱的问题和方案。比如，一个急着要表现自己能力和价值的新员工，你如果和他直接说“不行”，会挫伤他的积极性，而他经过一段时间的磨合和思考，会自己意识到不可行。

对于这种情况，适当逃避问题，将事情搁置起来是最好的办法。

“我去开个会，我们回来再谈。”

“这个 idea 非常好，我们改天组织一个会议好好讨论一下……”

*本章使用的案例纯属虚构。

16 漫话网站架构师

对于网站和软件企业而言，架构师是一个重要的角色。对于公司，架构师引领公司的技术方向，架构师的眼界和高度决定了公司的技术高度；对于技术团队，架构师的决策和技术方案影响工程师的开发模式和工作量。一个称职的架构师是公司的宝贵财富，而一个不合格的架构师可能会成为开发团队的梦魇，所谓将无能，累死三军。

对于大型网站而言，公司有很多架构师，他们的角色、能力和影响力各不相同，大致可分为以下几类。这些分类方式是非正式的，仅供诸位看官一乐，读者请勿以此给自己所在公司的架构师贴标签。

16.1 按作用划分架构师

设计型架构师

也就是一般意义上的架构师，负责系统架构设计，同时也要负责架构的实施落地、演化发展、推广重构。

救火型架构师

充当救火队员的角色，系统出现故障或者“灵异现象”，会请他们出马解决，有时重

要而紧急的项目也会由此类架构师主持。他们通常是公司的元老，对系统有全局性的认识，知道“水有多深”。

布道型架构师

对某一领域有较深刻的认识，有时候甚至是坚定的技术信仰，乐于同他人分享自己的知识，希望能够推广自己的技术主张，此类架构师通常有较好的个人影响力。但有时，由于自身的局限或者不能跟上技术潮流的发展，会成为忽悠型的“大师”、偶像派的专家。

Geek 型架构师

架构师中的 Geek，对某些技术问题的研究达到疯狂偏执的境地，精益求精追求完美。通常由于知识技能不够全面，不符合许多企业对架构师“高大全”的要求，此类架构师常有怀才不遇之感。

16.2 按效果划分架构师

夏尔巴人架构师

夏尔巴人生活在喜马拉雅山麓，协助探险队或者登山爱好者攀登那些 8000 米以上被称为“生命的禁区”的雪山，帮助他们运送给养到突击队营地，以及作为向导带领登山队员登顶。每一次成功对于登山队员是一次自我的超越，而对于夏尔巴人，不过是完成了一个工作。

夏尔巴人架构师通常会开发项目中最具技术难度和挑战性的模块，从而为整个项目的顺利进行铺平道路。

斯巴达人架构师

传说在古希腊，城邦之间发生战争，如果有城邦向斯巴达人求援，斯巴达人只会派出一个人去协助，但只要这一个人就可以扭转战局。

不管项目有多么艰难复杂，只要有斯巴达人架构师，大家就会坚信，项目一定能顺利完成。斯巴达人架构师带给项目组的，不只是技术和方法，更重要的是必胜的信念。这种信念是架构师自己积累起来的气场和影响力。

达官贵人架构师

此类架构师或者有傲人的学历，或者有辉煌的履历，或仪表堂堂，或口吐莲花，但是公司里如果有个吃人的怪兽，悄悄地把此类架构师都吃光了，也没人会发现。

16.3 按职责角色划分架构师

产品架构师

负责具体互联网产品的技术架构。当产品业务规划确定后，产品架构师就要开始产品的架构设计了，和运营团队确定 PV 数、用户数、商品数等产品运营目标、发展规划、非功能指标；和产品经理确定功能需求、模块划分等功能目标；和项目经理确定各种开发资源。获得必要的信息后进行整体架构设计，参与项目开发。产品架构师一般会参与产品的整个生命周期。

基础服务架构师

有时候也被称为平台架构师，负责开发基础框架、公共组件、通用服务等平台类产品。在大型互联网应用中，基础服务承担着海量的数据存储和核心业务处理服务，有许多挑战性的工作。

基础设施架构师

负责网络、存储、数据库运维管理的架构师，此类架构师一般有专门的称呼（如 DBA 等）。

此外，根据具体的职责，在数据挖掘、搜索技术、安全诚信、运维监控等领域也有专门的架构师。

16.4 按关注层次划分架构师

只关注功能的架构师

架构目标只是完成功能，通常，这不叫架构。

关注非功能的架构师

除了产品功能，架构设计也关注性能、伸缩性、安全性、可用性、系统未来的扩展性，以及上线后易于运维管理、监控报警、故障修复等非功能目标。

关注团队组织与管理的架构师

架构设计不但关注功能目标和非功能目标，同时还考虑开发团队的成员特点、进度安排、开发过程等，使架构设计和项目管理完美融合。

关注产品运营的架构师

架构设计不但关注产品的各项功能、非功能指标和开发过程的可实现性，还关注产品运营是否合理方便，能否达到运营目标，技术架构兼顾产品业务架构。

关注产品未来的架构师

不但关注前面提到的所有方面，还会结合技术发展趋势、公司战略目标、个人及团队发展方向，去思考产品未来的发展前景。为产品的发展演化符合历史发展趋势而设计并为其奠定一个坚实的基础。

16.5 按口碑划分架构师

最好的架构师

和团队相处日久，通常情况下团队成员感觉不出他的存在，貌似没有他工作也可以完成得很好，但是如果他真的离开了，大家就会觉得心里空荡荡，没了主心骨。

好的架构师

深得团队成员的敬重和信任，承担项目中的重要设计开发工作，团队几乎离不开他。

一般的架构师

承担了项目中大部分的技术工作，却常常因为团队成员不符合自己的期望而经常雷霆大发。

差的架构师

既无技术实力也不善于处理人际关系，常被团队成员鄙视，主要工作是给大家添乱、制造笑话和八卦的谈资。

最差的架构师

通过制造压力驱使团队成员努力去完成一些无价值的工作，让每个人都忙碌不堪以使大家都没有注意到他自己其实并不能胜任工作。这种架构师对组织整体和团队成员的伤害无以复加，却常常因为敬业和努力的形象而得到老板的肯定。

16.6 非主流方式划分架构师

普通架构师

从问题和需求出发，结合个人经验、组织资源、业界模式进行架构设计，中规中矩，能够切实可行地解决问题满足需求，是架构师中的普通青年。

文艺架构师

除了像普通架构师那样在架构设计中解决问题，文艺架构师还会在架构设计中进行一些更前瞻的思考和别出心裁的设计。此类架构师的设计文档通常会透着文艺青年的小清新范儿，喜欢在文档的开头描述他们与众不同的设计理念和风格。

1+1 架构师

不包括那些完全不能胜任架构设计工作的架构师，此类架构师喜欢在架构设计中堆砌概念和模式，设计文档宏大而不着调，面面俱到却不解决具体问题，说起来头头是道却不知如何落地。其根源不是不了解真正的问题就是不掌握正确的方法。有时候也不排除这样一种可能性：做架构设计的目的是为了炫耀自己知道这么多术语。

附录A

大型网站架构技术一览

本书关于架构技术原理的组织方式以架构要素作为维度，从系统性能、可用性、伸缩性、扩展性、安全性几个角度阐述网站架构技术要点。还有另一种较为直观的组织方式是从不同架构层次所使用的网站架构技术这个维度进行描述的。

网站系统架构层次如图 A.1 所示。

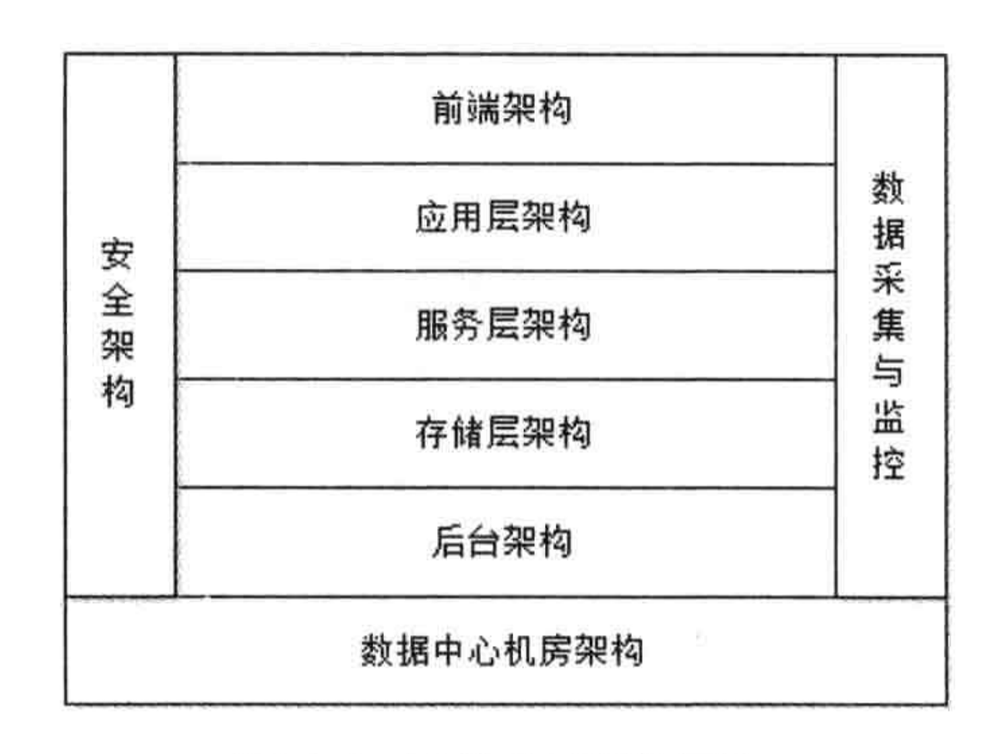

图 A.1　网站系统架构层次

1. 前端架构

前端指用户请求到达网站应用服务器之前经历的环节，通常不包含网站业务逻辑，

不处理动态内容。

浏览器优化技术

并不是优化浏览器，而是通过优化响应页面，加快浏览器页面的加载和显示，常用的有页面缓存、合并 HTTP 减少请求次数、使用页面压缩等。

CDN

内容分发网络，部署在网络运营商机房，通过将静态页面内容分发到离用户最近的 CDN 服务器，使用户可以通过最短路径获取内容。

动静分离，静态资源独立部署

静态资源，如 JS、CSS 等文件部署在专门的服务器集群上，和 Web 应用动态内容服务分离，并使用专门的（二级）域名。

图片服务

图片不是指网站 Logo、按钮图标等，这些文件属于上面提到的静态资源，应该和 JS、CSS 部署在一起。这里的图片指用户上传的图片，如产品图片、用户头像等，图片服务同样使用独立部署的图片服务器集群，并使用独立（二级）域名。

反向代理

部署在网站机房，在应用服务器、静态资源服务器、图片服务器之前，提供页面缓存服务。

DNS

域名服务，将域名解析成 IP 地址，利用 DNS 可以实现 DNS 负载均衡，配置 CDN 也需要修改 DNS，使域名解析后指向 CDN 服务器。

2. 应用层架构

应用层是处理网站主要业务逻辑的地方。

开发框架

网站业务是多变的，网站的大部分软件工程师都是在加班加点开发网站业务，一个

好的开发框架至关重要。一个好的开发框架应该能够分离关注面，使美工、开发工程师可以各司其事，易于协作。同时还应该内置一些安全策略，防护 Web 应用攻击。

页面渲染

将分别开发维护的动态内容和静态页面模板集成起来，组合成最终显示给用户的完整页面。

负载均衡

将多台应用服务器组成一个集群，通过负载均衡技术将用户请求分发到不同的服务器上，以应对大量用户同时访问时产生的高并发负载压力。

Session 管理

为了实现高可用的应用服务器集群，应用服务器通常设计为无状态，不保存用户请求上下文信息，但是网站业务通常需要保持用户会话信息，需要专门的机制管理 Session，使集群内甚至跨集群的应用服务器可以共享 Session。

动态页面静态化

对于访问量特别大而更新又不很频繁的动态页面，可以将其静态化，即生成一个静态页面，利用静态页面的优化手段加速用户访问，如反向代理、CDN、浏览器缓存等。

业务拆分

将复杂而又庞大的业务拆分开来，形成多个规模较小的产品，独立开发、部署、维护，除了降低系统耦合度，也便于数据库业务分库。按业务对关系数据库进行拆分，技术难度相对较小，而效果又相对较好。

虚拟化服务器

将一台物理服务器虚拟化成多台虚拟服务器，对于并发访问较低的业务，更容易用较少的资源构建高可用的应用服务器集群。

3. 服务层架构

提供基础服务，供应用层调用，完成网站业务。

分布式消息

利用消息队列机制，实现业务和业务、业务和服务之间的异步消息发送及低耦合的业务关系。

分布式服务

提供高性能、低耦合、易复用、易管理的分布式服务，在网站实现面向服务架构（SOA）。

分布式缓存

通过可伸缩的服务器集群提供大规模热点数据的缓存服务，是网站性能优化的重要手段。

分布式配置

系统运行需要配置许多参数，如果这些参数需要修改，比如分布式缓存集群加入新的缓存服务器，需要修改应用程序客户端的缓存服务器列表配置，并重启应用程序服务器。分布式配置在系统运行期提供配置动态推送服务，将配置修改实时推送到应用系统，无需重启服务器。

4. 存储层架构

提供数据、文件的持久化存储访问与管理服务。

分布式文件

网站在线业务需要存储的文件大部分都是图片、网页、视频等比较小的文件，但是这些文件的数量非常庞大，而且通常都在持续增加，需要伸缩性设计比较好的分布式文件系统。

关系数据库

大部分网站的主要业务是基于关系数据库开发的，但是关系数据库对集群伸缩性的支持比较差。通过在应用程序的数据访问层增加数据库访问路由功能，根据业务配置将数据库访问路由到不同的物理数据库上，可实现关系数据库的分布式访问。

NoSQL 数据库

目前各种 NoSQL 数据库层出不穷，在内存管理、数据模型、集群分布式管理等方面各有优势，不过从社区活跃性角度看，HBase 无疑是目前最好的。

数据同步

在支持全球范围内数据共享的分布式数据库技术成熟之前，拥有多个数据中心的网站必须在多个数据中心之间进行数据同步，以保证每个数据中心都拥有完整的数据。在实践中，为了减轻数据库压力，将数据库的事务日志（或者 NoSQL 的写操作 Log）同步到其他数据中心，根据 Log 进行数据重演，实现数据同步。

5. 后台架构

网站应用中，除了要处理用户的实时访问请求外，还有一些后台非实时数据分析要处理。

搜索引擎

即使是网站内部的搜索引擎，也需要进行数据增量更新及全量更新、构建索引等。这些操作通过后台系统定时执行。

数据仓库

根据离线数据，提供数据分析与数据挖掘服务。

推荐系统

社交网站及购物网站通过挖掘人和人之间的关系，人和商品之间的关系，发掘潜在的人际关系和购物兴趣，为用户提供个性化推荐服务。

6. 数据采集与监控

监控网站访问情况与系统运行情况，为网站运营决策和运维管理提供支持保障。

浏览器数据采集

通过在网站页面中嵌入 JS 脚本采集用户浏览器环境与操作记录，分析用户行为。

服务器业务数据采集

服务器业务数据包括两种，一种是采集在服务器端记录的用户请求操作日志；一种是采集应用程序运行期业务数据，比如待处理消息数目等。

服务器性能数据采集

采集服务器性能数据，如系统负载、内存使用率、网卡流量等。

系统监控

将前述采集的数据以图表的方式展示，以便运营和运维人员监控网站运行状况，做到这一步仅仅是系统监视。更先进的做法是根据采集的数据进行自动化运维，自动处理系统异常状况，实现自动化控制。

系统报警

如果采集来的数据超过预设的正常情况的阈值，比如系统负载过高，就通过邮件、短信、语音电话等方式发出报警信号，等待工程师干预。

7. 安全架构

保护网站免遭攻击及敏感信息泄露。

Web 攻击

以 HTTP 请求的方式发起的攻击，危害最大的就是 XSS 和 SQL 注入攻击。但是只要措施得当，这两种攻击都是比较容易防范的。

数据保护

敏感信息加密传输与存储，保护网站和用户资产。

8. 数据中心机房架构

大型网站需要的服务器规模数以十万计，机房物理架构也需要关注。

机房架构

对于一个拥有十万台服务器的大型网站，每台服务器耗电（包括服务器本身耗电及

空调耗电）每年大约需要人民币 2000 元，那么网站每年机房电费就需要两亿人民币。数据中心能耗问题已经日趋严重，Google、Facebook 选择数据中心地理位置的时候趋向选择散热良好，供电充裕的地方。

机柜架构

包括机柜大小，网线布局、指示灯规格、不间断电源、电压规格（是 48V 直流电还是 220V 民用交流电）等一系列问题。

服务器架构

大型网站由于服务器采购规模庞大，大都采用定制服务器的方式代替购买服务器整机。根据网站应用需求，定制硬盘、内存、甚至 CPU，同时去除不必要的外设接口（显示器输出接口，鼠标、键盘输入接口），并使空间结构利于散热。

附录B

Web 开发技术发展历程

随着互联网的发展，Web 服务端开发技术也经历了几次大的变迁。早期的 Web 服务器只简单地响应浏览器端的请求，返回静态的 HTML。随着 CGI（Common Gateway Interface，通用网关接口）技术的出现，Web 服务端可以根据不同用户请求产生动态页面内容。CGI 处理动态请求的基本过程如图 B.1 所示，Web 服务器将请求数据交给 CGI 程序，CGI 程序进行运算处理，生成 HTML 输出，通过 Web 服务器返回给浏览器。早期主要的 CGI 编程语言是 Perl，高效便捷的开发特性使其成为当时许多网站开发的首选。但是 Web 服务器通过启动独立进程的方式调用 CGI 程序，消耗许多不必要的系统资源。Java Servlet 则以线程方式在 Java Web 容器中调用 Servlet，较 CGI 方式消耗资源更少。

一般来说 CGI 技术（广义上也包括 Java Servlet）被称作脚本模式，CGI 程序需要解析 HTTP 请求，处理业务逻辑，并在输出流中构造响应信息的 HTML。这种技术的优点和缺点是同一个特性——可以在 CGI 程序中做任何事情。CGI 程序在获得最大处理能力的同时，也给开发人员带来了麻烦：负责编写业务逻辑程序的程序员不擅长处理 HTML，而负责页面构造的美工人员则对程序束手无策。同样维护这样的程序也是一个噩梦，业务代码和页面语法耦合在一起，让人无从下手。

PHP 及随后 ASP、JSP 的出现改善了这一局面，与 CGI 在程序中输出 HTML 流正好

相反，开发人员可以在 HTML 中嵌入程序代码。这种模式被称作服务器页面模式。直到现在，PHP 仍然是许多中小型网站建站首选技术，和 Apache、MySQL、Linux 共同组成一个强大的 Web 开发平台，被称作 LAMP。

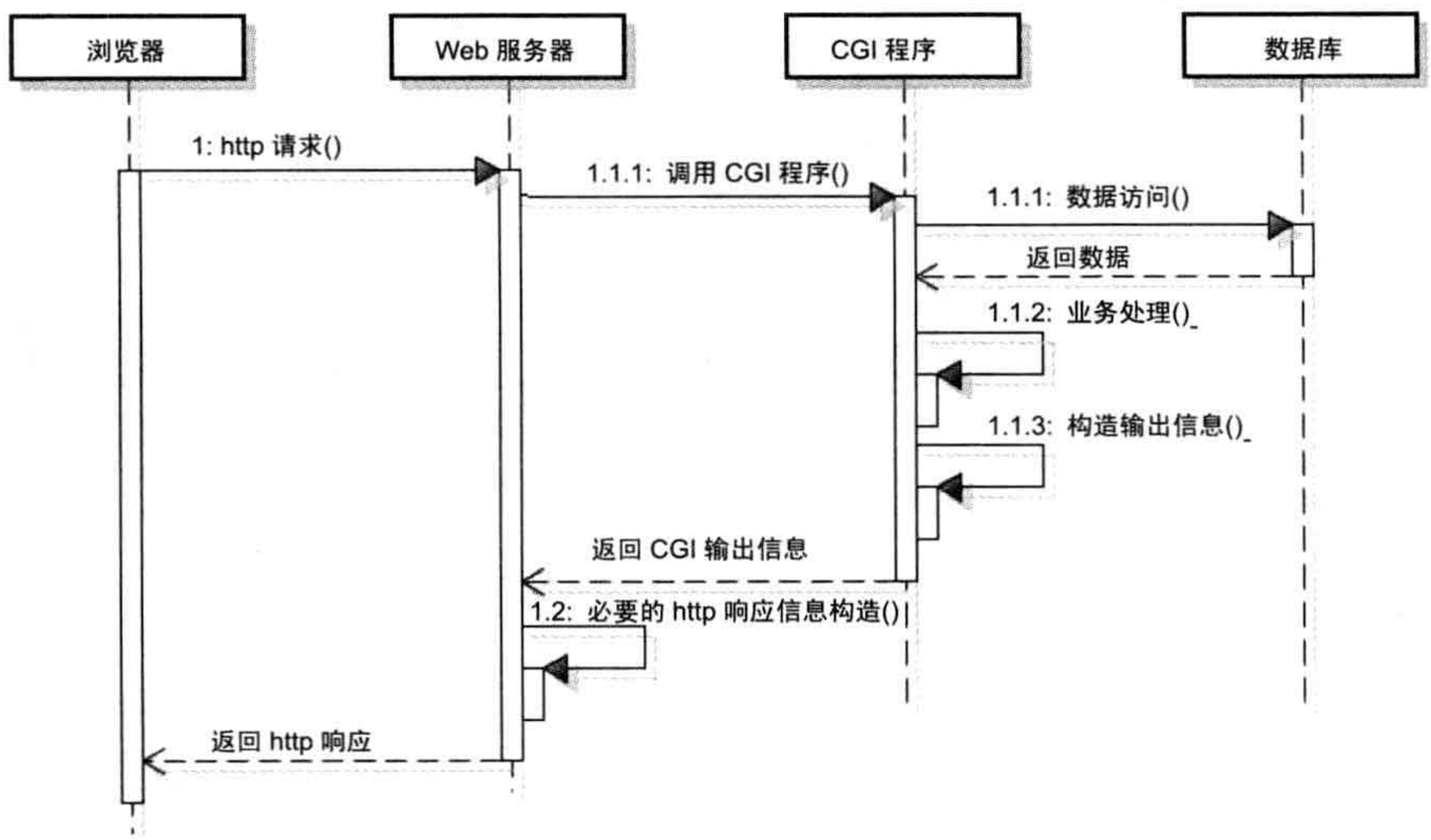

图 B.1　CGI 程序调用时序模型

既然 CGI 程序擅长处理请求信息，而服务器页面擅长构造响应页面，那么能不能将两者结合起来呢？答案就是 MVC（模型-视图-控制器）模式，如图 B.2 所示，控制器接收处理所有的 HTTP 请求，根据请求信息将其分发给不同的模型对象处理，再根据模型处理结果选择构造视图，得到最终响应信息。使用 MVC 模式可以很好地分离模型与视图，使二者完全解耦，互相影响降到最低。

模型和视图分离为系统开发维护带来了诸多好处，为目前 Web 开发流畅的分层架构模式奠定了基础。分层模式可以更进一步分离关注面和降低系统的耦合性，通过分层，隔离上层对下层的直接依赖，上层设计无需过多考虑下层实现；各层之间较少耦合，只要保持接口规范不变，各层可以随意替换和复用。Web 开发中通常将服务端划分为三层：表现层、业务逻辑层和数据源层。表现层完成视图展现和用户交互；业务逻辑层实现系统的核心逻辑；数据源层负责数据存储、交换和通信。这种层次划分是逻辑上的，物理部署上多个层会作为一个应用部署在一起。

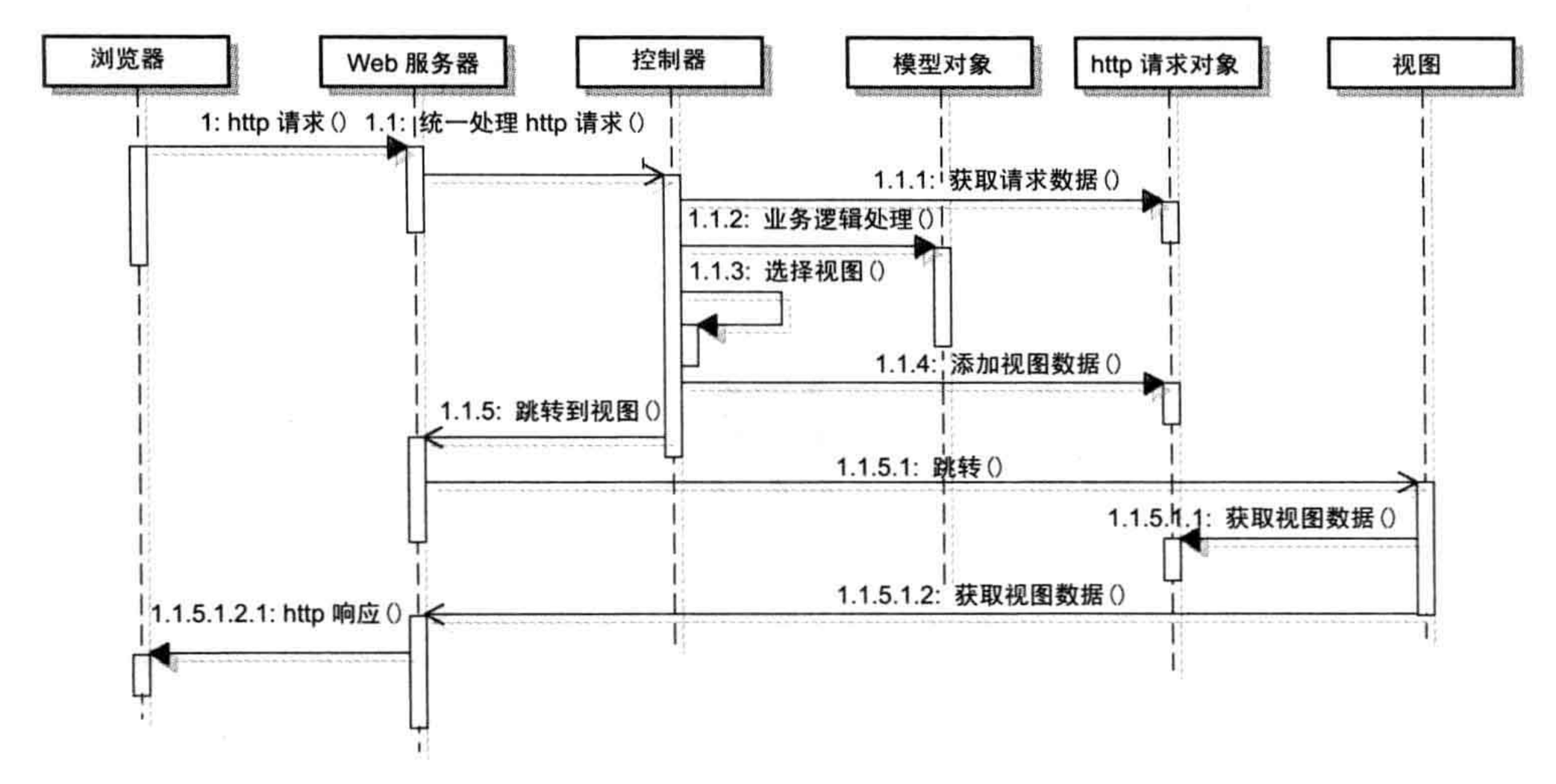

图 B.2　MVC 系统调用时序模型

上面简单回顾了 Web 开发的技术发展历程和一些早期主要架构模式，这些模式在企业 Web 应用开发中也有许多实践。但是随着互联网应用的快速发展，需求场景和业务领域都有一些和传统企业应用不同的特点，对系统的可用性、扩展性、响应性能、伸缩性、安全性都提出了更高的要求，网站技术架构也和企业应用技术架构脱离，走上了一条更具创新性的发展之路。

后 记

这是一本讲大型网站架构设计的书，但是大型网站不是设计出来的，而是逐步发展演化出来的。

不要企图去设计一个大型网站！

有些传统企业进军互联网，凭借其雄厚的资金、丰富的行业经验、近乎垄断的市场地位，试图在互联网领域开发一个大型网站复制其在传统行业的优势地位。但是互联网发展运行有其自己的规律，短暂的互联网历史已经一再证明这种企图是行不通的。

垄断、牌照、行业壁垒、国有资本、行政资源，这些在传统行业呼风唤雨的魔法到了互联网领域只会被嘲笑、被捉弄。庞大只是笨拙而已，壁垒只会画地为牢，没什么了不起。

互联网没有门槛，谁都可以进来玩，但是进来后，最好把那些陈旧的思想和包袱放下，重新来过。

互联网是一个开放和分享的世界，这里是创新者的乐园，探险者的处女地。只要你努力，富有想象力和聪明才智，能为用户创造价值，能推动社会进步，不管你开始时多么弱小，总有机会迅速聚集资金、人才和注意力，在的时间内发展壮大。

互联网是一种精神，一种开放、分享、自由的精神；越是付出不问回报，越是获得丰厚的回报；越是不设边界，越是拥有整个世界。互联网是一种颠覆，打碎所有的藩篱，给所有人平等表达和获取的机会，每个人都可以发出自己的声音。互联网是一种建设，重塑人们的思维方式和社会运行方式，建设一个人和人彼此理解信任的大同世界。

互联网正在并将继续改变这个世界，一切才刚刚开始，你我正生逢其时！

[General Information]
书名 = 大型网站技术架构 核心原理与案例分析