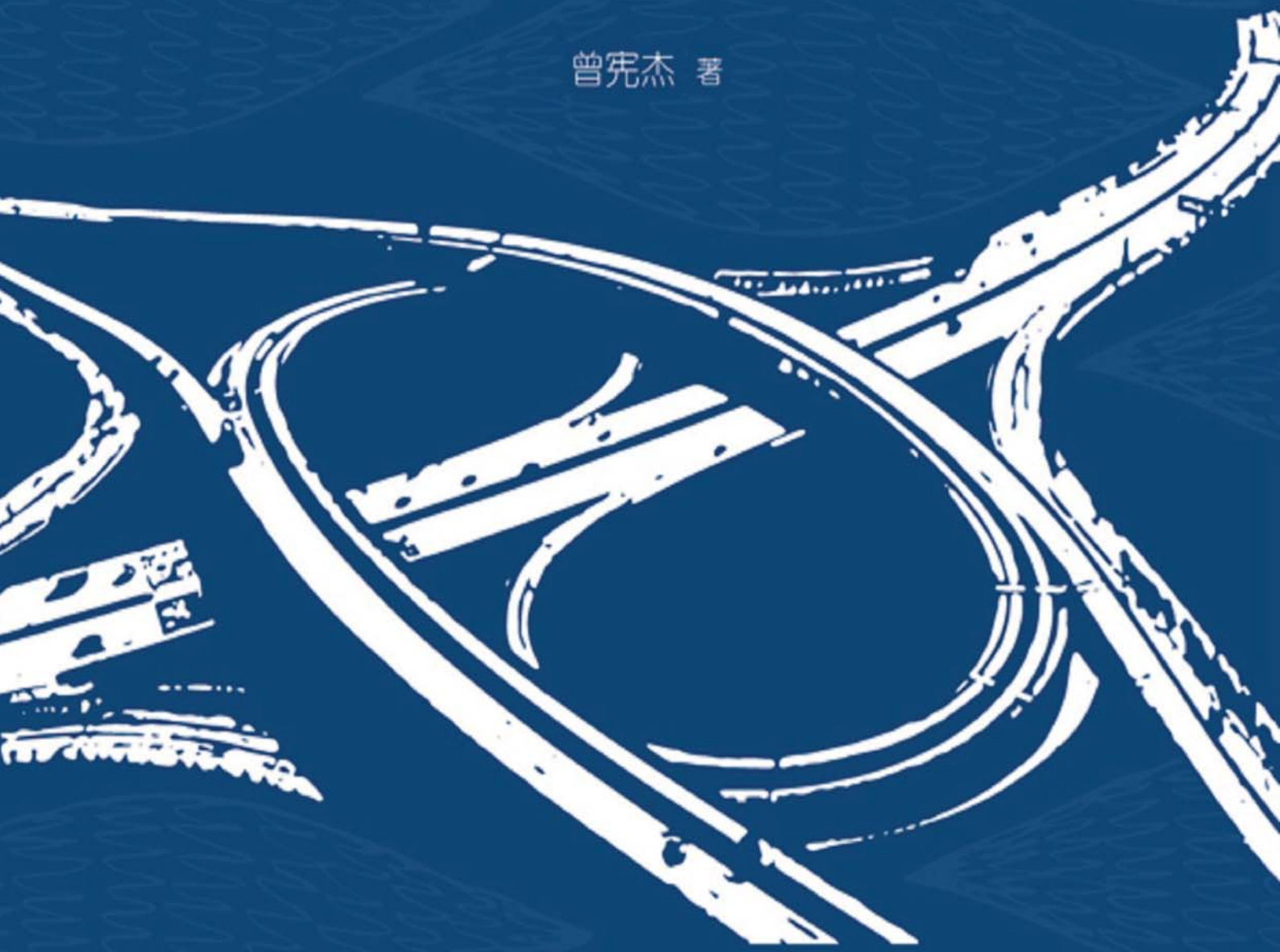
大型网站的访问负载前所未有，高性能、高可靠成为现实挑战。本书凝聚了作者在淘宝高速发展和架构变迁中积累的宝贵经验，网站开发、架构人员不可不读。

——章文嵩 阿里巴巴集团高级研究员/核心系统研发部负责人

Broadview
www.broadview.com.cn

大型网站系统
与Java中间件实践

曾宪杰 著



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

本书围绕大型网站和支撑大型网站架构的 Java 中间件的实践展开介绍。从分布式系统的知识切入，让读者对分布式系统有基本的了解；然后介绍大型网站随着数据量、访问量增长而发生的架构变迁；接着讲述构建 Java 中间件的相关知识；之后的几章都是根据笔者的经验来介绍支撑大型网站架构的 Java 中间件系统的设计和实践。希望读者通过本书可以了解大型网站架构变迁过程中的较为通用的问题和解法，并了解构建支撑大型网站的 Java 中间件的实践经验。

对于有一定网站开发、设计经验，并想了解大型网站架构和支撑这种架构的系统的开发、测试等的相关工程人员，本书有很大的参考意义；对于没有网站开发设计经验的人员，通过本书也能宏观了解大型网站的架构及相关问题的解决思路和方案。

图书在版编目（CIP）数据

大型网站系统与 Java 中间件实践 / 曾宪杰著. —北京：电子工业出版社，2014.4
ISBN 978-7-121-22761-5

Ⅰ. ①大… Ⅱ. ①曾… Ⅲ. ①网站－建设②JAVA 语言－程序设计 Ⅳ. ①TP393.092②TP312

中国版本图书馆 CIP 数据核字(2014)第 059272 号

策划编辑：张春雨
责任编辑：徐津平
印　　刷：北京丰源印刷厂
装　　订：三河市鹏成印业有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：21　字数：338.7 千字
印　　次：2014 年 4 月第 1 次印刷
印　　数：4000 册　　定价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

2 第 2 章 大型网站及其架构演进过程

2.1 什么是大型网站

通过第 1 章我们了解了分布式系统的相关基础知识，大型网站是一种很常见的分布式系统，而本书重点要介绍的中间件系统也是在大型网站的架构变化中出现并发展的，那么我们很有必要从大型网站的架构演进过程入手，先从整体上了解这个变化过程。首先，我们来看一下怎样的网站被称为大型网站。

关于大型网站的定义，在学术上并没有精确地定义，下面是将笔者自己的理解介绍给大家。

网站是用来访问的，访问量大就应该是大型网站。这个说法不全对，从 www.alexa.com 上可以看到不同网站的大概访问量，排在前面的都是比较出名且大型的网站。不过我在这里举一个反例，在我写这段内容的时候，下面这个网站在中国的 Traffic Rank 的排名是第 179 位，在整个互联网的排名是第 1301 位，这个网站怎么说也不算小了。来看看我说的是哪个网站吧，如图 2-1 所示。

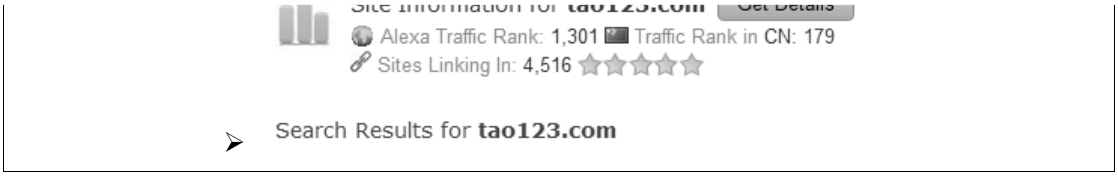


图 2-1　Alexa 上 tao123.com 的排名信息

再看看这个站点，如图 2-2 所示。

图 2-2　tao123.com 的界面

看到这里，我想大家不会认为上面这个网站本身是一个大型网站了。这么看来访问量很大不是成为大型网站的一个充分条件。

我们再看看数据量，数据量应该是我们关注的另一个维度的条件。一个大型网站应该有大量的数据，或者说是海量数据才行。

在我看来，访问量和数据量二者缺一不可。仅有访问量的例子前面已经给大家看了；对于仅有数据量的情况，我们可以简单想象一下，一个网站有非常多的数据，可是每天访问量很低，页面浏览量（PV）也很低，那这个网站肯定也不算是大型网站。此外，除了海量数据和高并发的访问量，本身业务和系统的复杂度也是考察的方面。

大型网站要支撑海量的数据和非常高并发的访问量，那么它肯定是一个分布式系统。即便你用小型机而不是 PC Server，你也需要用集群来支撑而不是靠单机。我没有见过也没有用过大型机，关于用大型机来实现的情况，这里暂不讨论。

2.2　大型网站的架构演进

我们现在常用的大型网站都是从小网站一步一步发展起来的，这个过程中会有一些通用的问题要解决，而这些也是我们构建中间件系统的基础，那么我们就从最简单的网站结构开始，看看随着网站从小到大的变化，网站架构发生了哪些变化。

2.2.1 用 Java 技术和单机来构建的网站

我们先从最简单的开始吧。说到做网站，不管大家是自己动手实践过，还是听说过，肯定能反应出很多技术名词，例如 LAMP、MVC 框架、JSP、Spring、Struts、Hibernate、HTML、CSS、JavaScript、Python，等等。

笔者最早接触网站是在 1998 年，那个时候主要是上网看新闻、收发电子邮件，当时只了解 HTML。到了 2000 年的时候，接触了一些 ASP，才弄明白怎么做动态的内容。当时，大家都是在自己的机器上搭建一个环境来学习相关技术，或者做开发和测试。我们可以看一下采用 Java 技术、使用单机构建的网站的样子。

我们基本上会选择一个开源的 Server 作为容器，直接使用 JSP/Servlet 等技术或者使用一些开源框架来构建我们的应用；选择一个数据库管理系统来存储数据，通过 JDBC 进行数据库的连接和操作——这样，一个最基础的环境就可以工作了（如图 2-3 所示）。相信很多在学校接触网站开发的同学都是从这样的做法开始的。

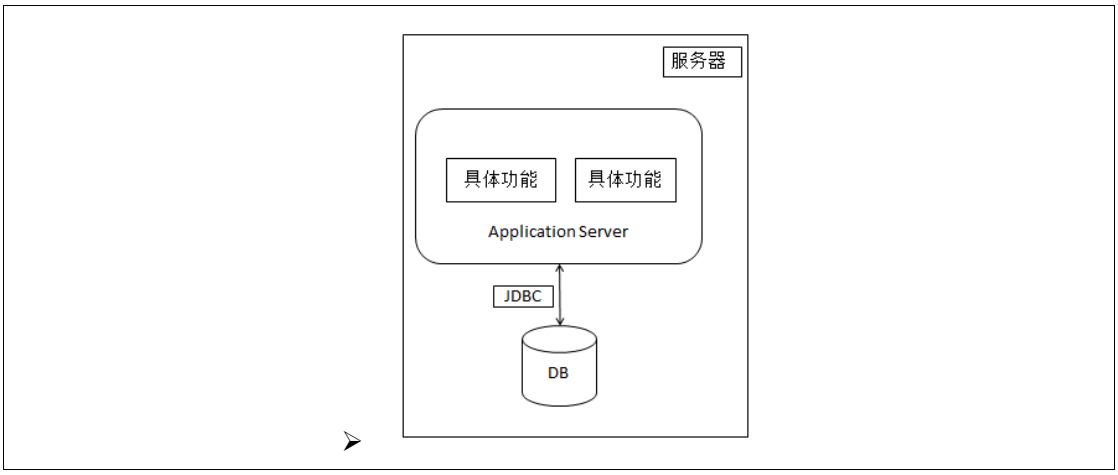


图 2-3 技术单机构建的网站

对于实际的大型网站来说，情况远比我们看到的这个例子要复杂。我们回顾一下在前面章节中讲到的计算机的组成部分，在大型网站中，其实最核心的功能就是计算和存储。在图 2-3 中，DB 就是用来存储数据的，而 Application Server 完成了业务功能和逻辑，是用于计算的。一个网站从小到大的演进可以说都是在围绕着这两个方面进行处理。因此图 2-3 所示的网站可以作为我们的一个起点。

2.2.2 从一个单机的交易网站说起

为了更好地说明网站从小到大的演进过程，我们下面举一个交易类网站的例子。当然，这个例子和架构的变化过程并不是某一个具体交易类网站的真实过程，而是糅合了通用的架构变化方式的一个示例，这样能更好地说明问题。

此外，还有两点要说明。首先，我们下面重点关注的是随着数据量、访问量提升，网站结构发生了什么变化，而不关注具体的业务功能点。其次，下面的网站演进过程是为了让大家更好地了解网站演进过程中的一些问题和应对策略，并不是指导大家按照下面的过程来改进网站。

作为一个交易网站，需要具备的最基本功能有三部分：

用户
- 用户注册
- 用户管理
- 信息维护
- ……

商品
- 商品展示
- 商品管理
- ……

交易
- 创建交易
- 交易管理
- ……

现在的交易网站有很多，并且功能非常丰富。我们从简单的开始，假设我们只支持这三部分的功能，那么，基于 Java 技术用单机来构建这个交易网站的话，大概会是图 2-4 的样子。

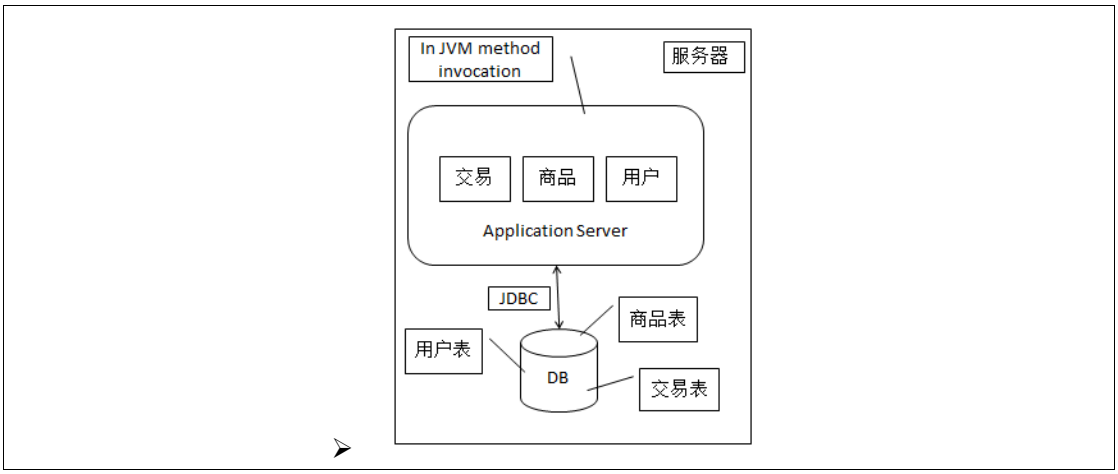


图 2-4　基于 Java 技术用单机构建的交易网站

与图 2-3 的结构是一样的，在这里有两个地方需要注意，即各个功能模块之间是通过 JVM 内部的方法调用来进行交互的，而应用和数据库之间是通过 JDBC 进行访问的。后面

围绕着这两个部分会有不同的变化。

2.2.3 单机负载告警，数据库与应用分离

我们很幸运，网站对外服务后，访问量不断增大，我们这个服务器的负载持续升高，必须要采取一些办法来应对了。这里我们先不考虑更换机器或各种软件层面的优化。我们来看一下结构上的变化。首先我们可以做的就是把数据库与应用从一台机器分到两台机器，那么，我们的网站结构会变成图 2-5 所示的样子。

网站的机器从一台变成了两台，这个变化对于我们来说影响很小。单机的情况下，我们的应用也是采用 JDBC 的方式同数据库进行连接，现在数据库与应用分开了，我们只是在应用的配置中把数据库的地址从本机改到了另外一台机器上而已，对开发、测试、部署都没有什么影响。

调整以后我们能够缓解当前的系统压力，不过随着时间的推移，访问量继续增大，我们的系统还是需要继续演进的。

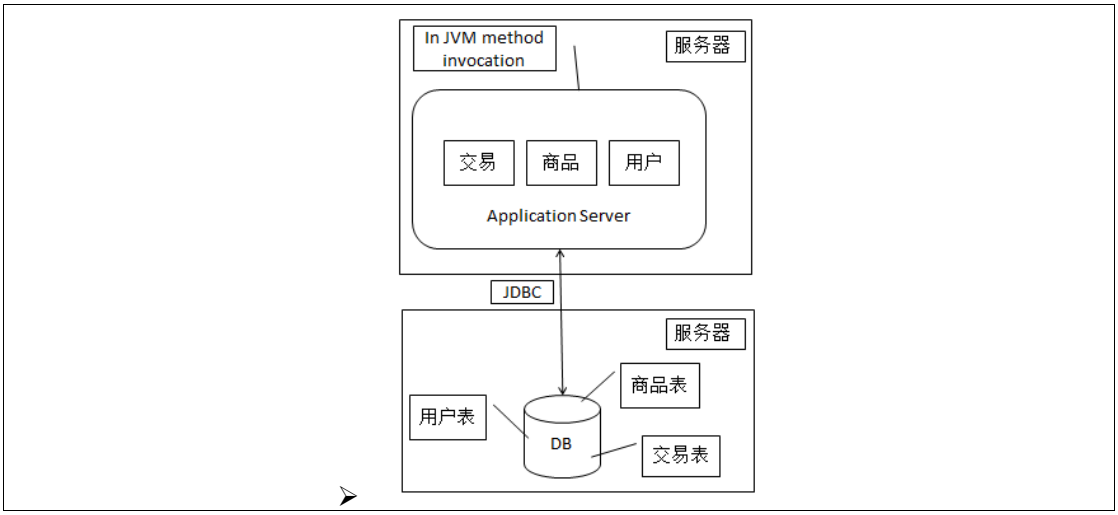


图 2-5 应用与数据库分开的结构

2.2.4 应用服务器负载告警，如何让应用服务器走向集群

我们接着看一下应用服务器压力变大的情况。

应用服务器压力变大时，根据对应用的监测结果，可以有针对性地进行优化。我们这里要介绍的是把应用从单机变为集群的优化方式。

先来看一下这个变化，如图 2-6 所示。

在图 2-6 中，应用服务器从一台变为了两台。这两个应用服务器之间没有直接的交互，

它们都是依赖数据库对外提供服务的。在增加了一台应用服务器后，我们有下面两个问题需要解决：

最终用户对两个应用服务器访问的选择问题。这在前面的章节提到过，可以通过 DNS 来解决，也可以通过在应用服务器集群前增加负载均衡设备来解决。我们这里选择第二种方案。

Session 的问题。接下来就会详细讲述。

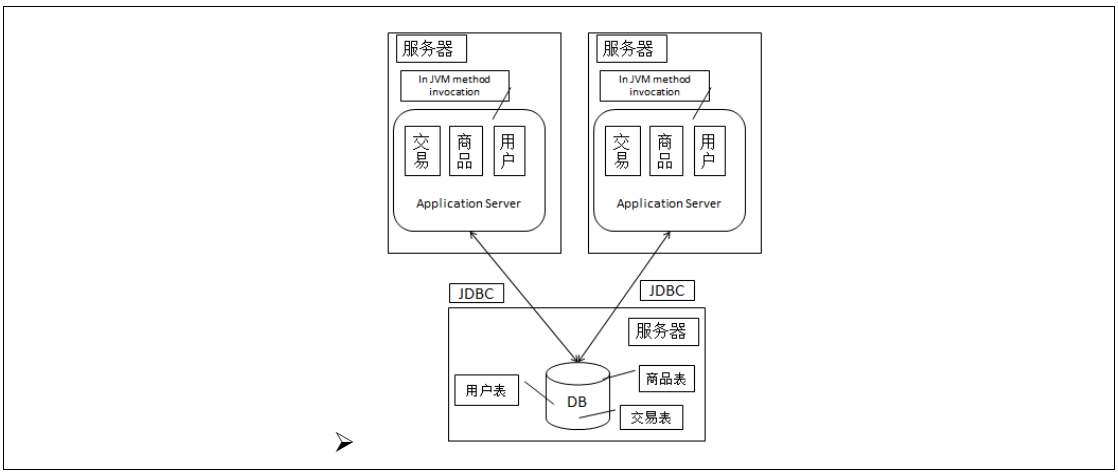


图 2-6　应用服务器集群

2.2.4.1　引入负载均衡设备

采用了负载均衡设备后，系统的结构看起来如图 2-7 所示。

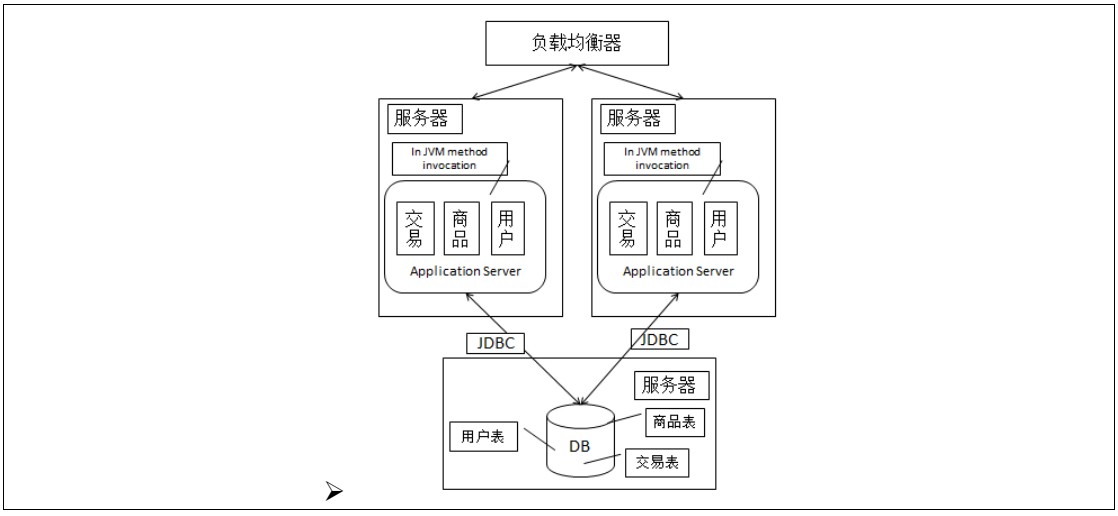


图 2-7　引入负载均衡设备的结构

这时，我们会遇到一个与 Session 相关的问题，下面介绍什么是 Session 问题，以及如

何解决它。

2.2.4.2 解决应用服务器变为集群后的Session问题

先来看一下什么是 Session。

用户使用网站的服务，基本上需要浏览器与 Web 服务器的多次交互。HTTP 协议本身是无状态的，需要基于 HTTP 协议支持会话状态（Session State）的机制。而这样的机制应该可以使 Web 服务器从多次单独的 HTTP 请求中看到“会话”，也就是知道哪些请求是来自哪个会话的。具体实现方式为：在会话开始时，分配一个唯一的会话标识（SessionId），通过 Cookie 把这个标识告诉浏览器，以后每次请求的时候，浏览器都会带上这个会话标识来告诉 Web 服务器请求是属于哪个会话的。在 Web 服务器上，各个会话有独立的存储，保存不同会话的信息。如果遇到禁用 Cookie 的情况，一般的做法就是把这个会话标识放到 URL 的参数中。我们可以通过图 2-8 来看一下上述过程。

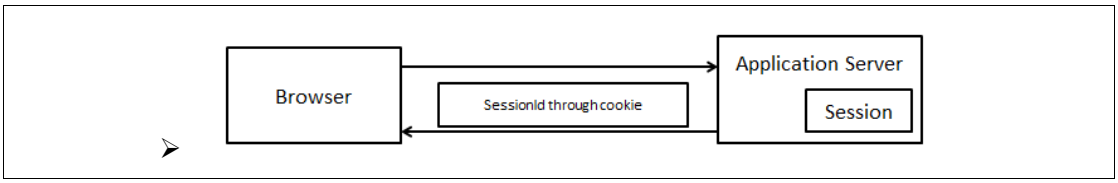


图 2-8 Session

当我们的应用服务器从一台变到两台后，如同图 2-7 中的结构，我们就会遇到 Session 的问题了。具体是指什么问题呢？

我们来看图 2-9，当一个带有会话标识的 HTTP 请求到了 Web 服务器后，需要在 HTTP 请求的处理过程中找到对应的会话数据（Session）。而问题就在于，会话数据是需要保存在单机上的。

在图 2-9 所示的网站中，如果我第一次访问网站时请求落到了左边的服务器，那么我的 Session 就创建在左边的服务器上了，如果我们不做处理，就不能保证接下来的请求每次都落在同一边的服务器上了，这就是 Session 问题。

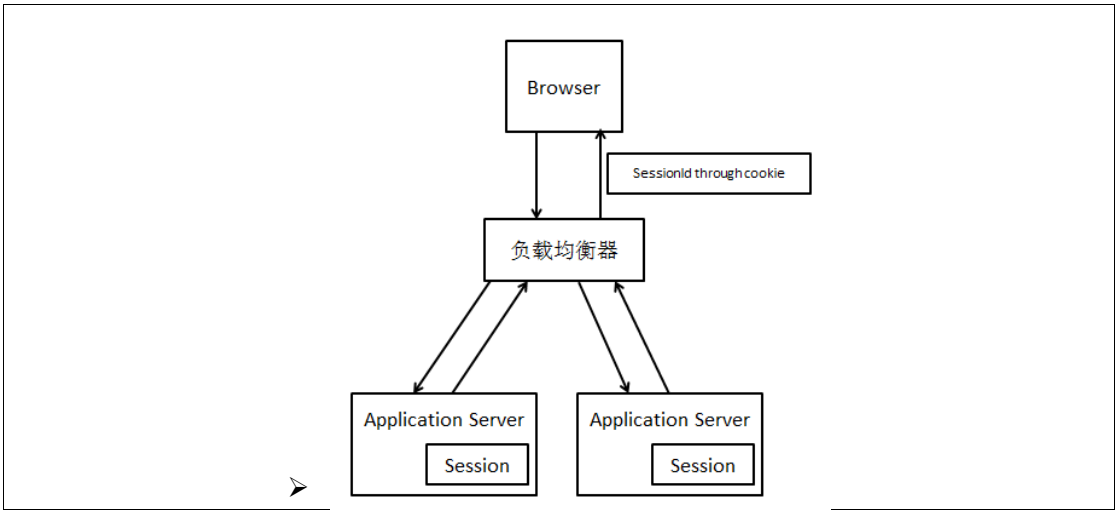


图 2-9　负载均衡、应用集群与 Session

我们看看这个问题的几种解决方案。

1．Session Sticky

在单机的情况下，会话保存在单机上，请求也都是由这个机器处理，所以不会有问题。Web 服务器变成多台以后，如果保证同一个会话的请求都在同一个 Web 服务器上处理，那么对这个会话的个体来说，与之前单机的情况是一样的。

如果要做到这样，就需要负载均衡器能够根据每次请求的会话标识来进行请求转发，如图 2-10 所示，称为 Session Sticky 方式。

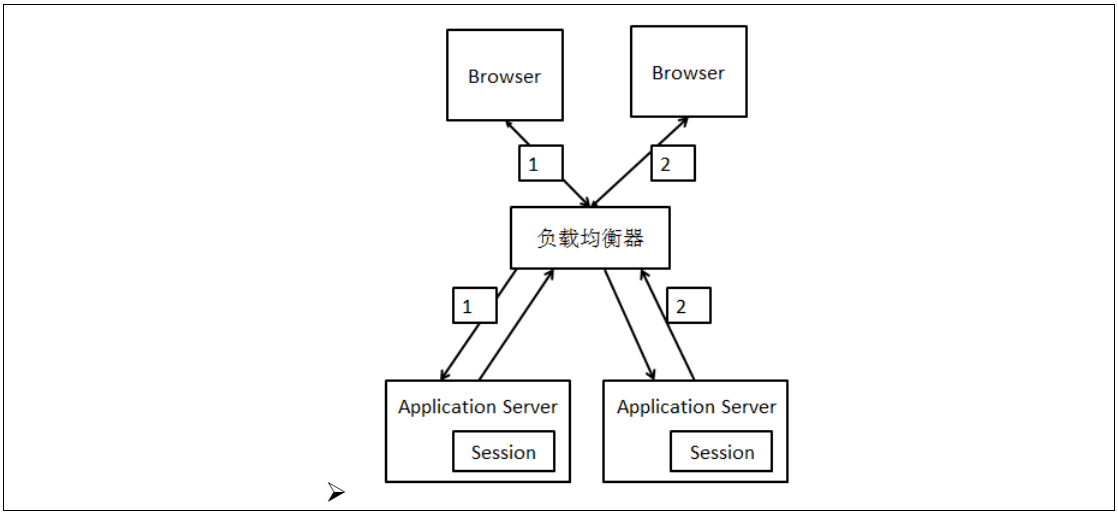


图 2-10　Session Sticky 方式

这个方案本身非常简单，对于 Web 服务器来说，该方案和单机的情况是一样的，只是我们在负载均衡器上做了“手脚”。这个方案可以让同样 Session 的请求每次都发送到同一个服

务器端处理，非常利于针对 Session 进行服务器端本地的缓存。不过也带来了如下几个问题：

如果有一台 Web 服务器宕机或者重启，那么这台机器上的会话数据会丢失。如如果会话中有登录状态数据，那么用户就要重新登录了。

会话标识是应用层的信息，那么负载均衡器要将同一个会话的请求都保存到同一个 Web 服务器上的话，就需要进行应用层（第 7 层）的解析，这个开销比第 4 层的交换要大。

负载均衡器变为了一个有状态的节点，要将会话保存到具体 Web 服务器的映射。和无状态的节点相比，内存消耗会更大，容灾方面会更麻烦。

这种方式我们称为 Session Sticky。打个比方来说，如果说 Web 服务器是我们每次吃饭的饭店，会话数据就是我们吃饭用的碗筷。要保证每次吃饭都用自己的碗筷的话，我就把餐具存在某一家，并且每次都去这家店吃，是个不错的主意。

2．Session Replication

如果我们继续以去饭店吃饭类比，那么除了前面的方式之外，如果我在每个店里都存放一套自己的餐具，不就可以更加自由地选择饭店了吗？Session Replication 就是这样的一种方式，这一点从字面上也很容易看出来。

先看一下图 2-11，如下。

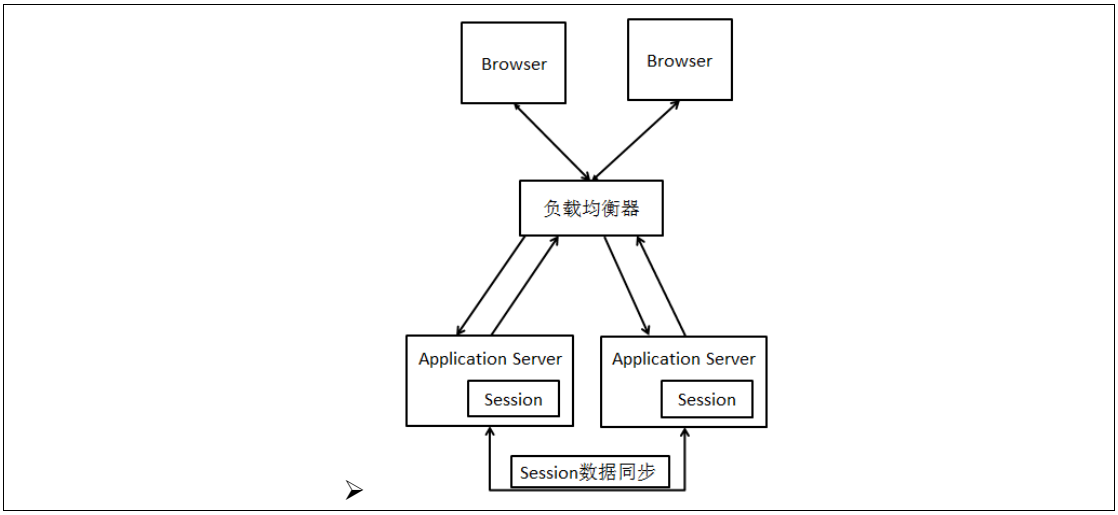


图 2-11　SessionReplication 方式

可以看到，在 Session Replication 方式中，不再要求负载均衡器来保证同一个会话的多次请求必须到同一个 Web 服务器上了。而我们的 Web 服务器之间则增加了会话数据的同步。通过同步就保证了不同 Web 服务器之间的 Session 数据的一致。就如同每家饭店都有我的碗筷，我就能随便选择去哪家吃饭了。

一般的应用容器都支持（包括了商业的和开源的）Session Replication 方式，与 Session

Sticky 方案相比，Session Replication 方式对负载均衡器没有那么多的要求。不过这个方案本身也有问题，而且在一些场景下，问题非常严重。我们来看一下这些问题。

同步 Session 数据造成了网络带宽的开销。只要 Session 数据有变化，就需要将数据同步到所有其他机器上，机器数越多，同步带来的网络带宽开销就越大。

每台 Web 服务器都要保存所有的 Session 数据，如果整个集群的 Session 数很多（很多人在同时访问网站）的话，每台机器用于保存 Session 数据的内容占用会很严重。

这就是 Session Replication 方案。这个方案是靠应用容器来完成 Session 的复制从而使得应用解决 Session 问题的，应用本身并不用关心这个事情。不过，这个方案不适合集群机器数多的场景。如果只有几台机器，用这个方案是可以的。

3．Session 数据集中存储

同样是希望同一个会话的请求可以发到不同的 Web 服务器上，刚才的 Session Replication 是一种方案，还有另一种方案就是把 Session 数据集中存储起来，然后不同 Web 服务器从同样的地方来获取 Session。大概的结构如图 2-12 所示。

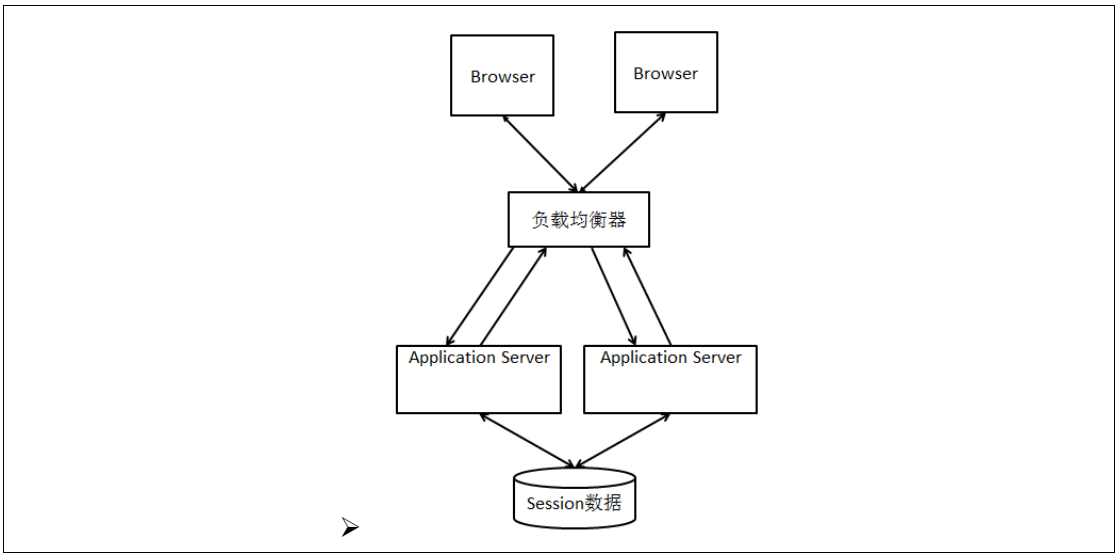


图 2-12　集中存储 Session 方式

可以看到，与 Session Replication 方案一样的部分是，会话请求经过负载均衡器后，不会被固定在同样的 Web 服务器上。不同的地方是，Web 服务器之间没有了 Session 数据复制，并且 Session 数据也不是保存在本机了，而是放在了另一个集中存储的地方。这样，不论是哪台 Web 服务器，也不论修改的是哪个 Session 数据，最终的修改都发生在这个集中存储的地方，而 Web 服务器使用 Session 数据时，也是从这个集中存储 Session 数据的地方来读取。这样的方式保证了不同 Web 服务器上读到的 Session 数据都是一样的。而存储 Session 数据的具体方式，可以使用数据库，也可以使用其他分布式存储系统。这个方案解决了 Session Replication 方案中内存的问题，而对于网络带宽，这个方案也比 Session Replication 要好。

该方案存在的问题是什么呢？

读写 Session 数据引入了网络操作，这相对于本机的数据读取来说，问题就在于存在时延和不稳定性，不过我们的通信基本都是发生在内网，问题不大。

如果集中存储 Session 的机器或者集群有问题，就会影响我们的应用。

相对于 Session Replication，当 Web 服务器数量比较大、Session 数比较多的时候，这个集中存储方案的优势是非常明显的。

4．Cookie Based

Cookie Based 方案是要介绍的最后一个解决 Session 问题的方案。这个方案对于同一个会话的不同请求也是不限制具体处理机器的。和 Session Replication 以及 Session 数据集中管理的方案不同，这个方案是通过 Cookie 来传递 Session 数据的。还是先看看下面的图 2-13 吧。

从图 2-13 可以看到，我们的 Session 数据放在 Cookie 中，然后在 Web 服务器上从 Cookie 中生成对应的 Session 数据。这就好比我每次都把自己的碗筷带在身上，这样我去哪家饭店吃饭就可以随意选择了。相对于前面的集中存储，这个方案不会依赖外部的一个存储系统，也就不存在从外部系统获取、写入 Session 数据的网络时延、不稳定性了。不过，这个方案依然存在不足：

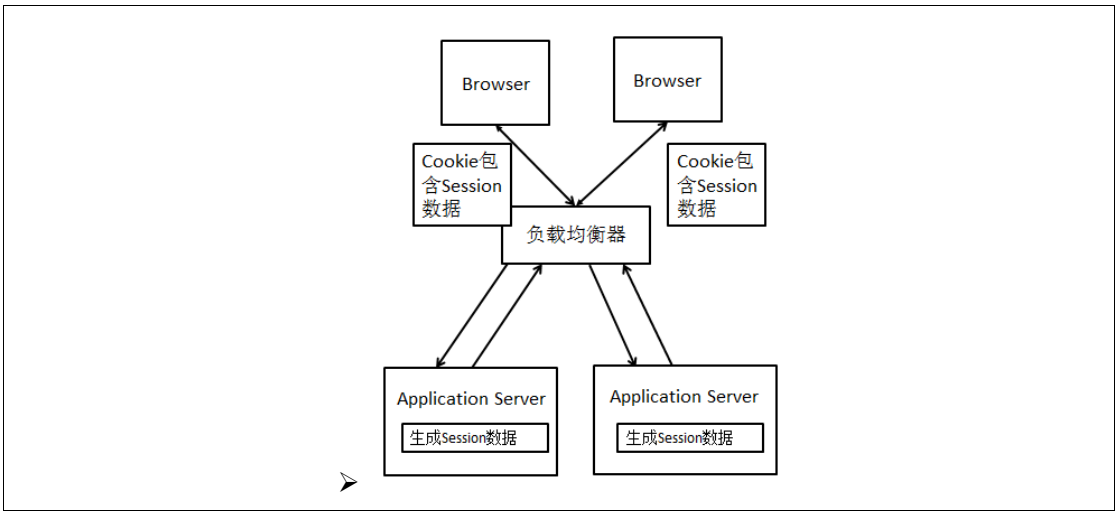


图 2-13　CookieBased 的方式

Cookie 长度的限制。我们知道 Cookie 是有长度限制的，而这也会限制 Session 数据的长度。

安全性。Session 数据本来都是服务端数据，而这个方案是让这些服务端数据到了外部网络及客户端，因此存在安全性上的问题。我们可以对写入 Cookie 的 Session 数据做加密，不过对于安全来说，物理上不能接触才是安全的。

带宽消耗。这里指的不是内部 Web 服务器之间的带宽消耗，而是我们数据中心的整体外部带宽的消耗。

性能影响。每次 HTTP 请求和响应都带有 Session 数据，对 Web 服务器来说，在同样的处理情况下，响应的结果输出越少，支持的并发请求就会越多。

2.2.4.3 小结

前面介绍了 Web 服务器从单机到多机情况下的 Session 问题的解决方案。这 4 个方案都是可用的方案，不过对于大型网站来说，Session Sticky 和 Session 数据集中存储是比较好的方案，而这两个方案又各有优劣，需要在具体的场景中做出选择和权衡。

不管采用上述何种方案，我们都可以在一定程度上通过增加 Web 服务器的方式来提升应用的处理能力了。接下来，我们来看一下数据库方面的变化。

2.2.5 数据读压力变大，读写分离吧

2.2.5.1 采用数据库作为读库

随着业务的发展，我们的数据量和访问量都在增长。对于大型网站来说，有不少业务是读多写少的，这个状况也会直接反应到数据库上。那么对于这样的情况，我们可以考虑使用读写分离的方式。

从图 2-14 中可以看到，我们在前面的结构上增加了一个读库，这个库不承担写的工作，只提供读服务。

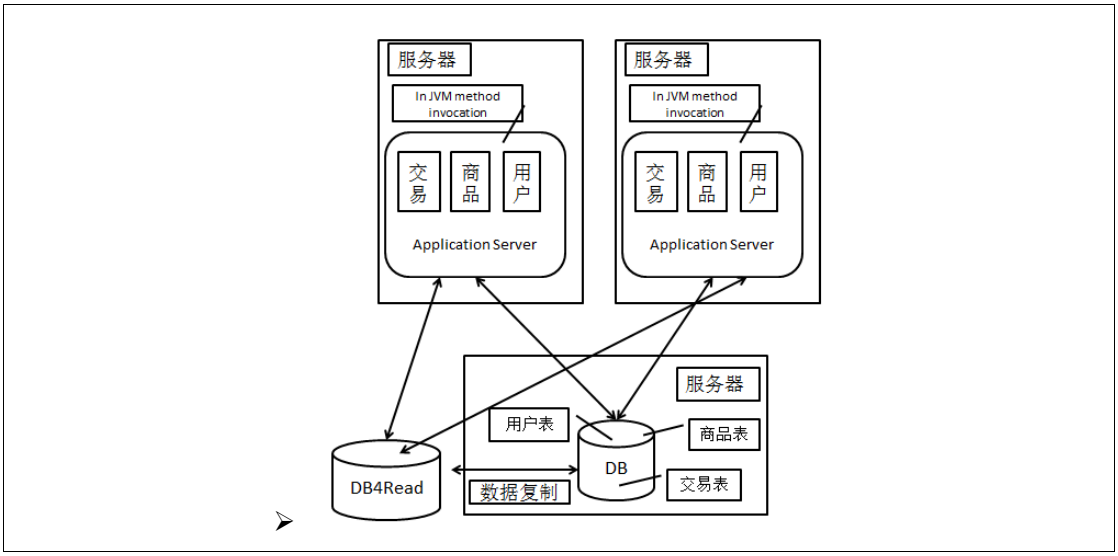


图 2-14 加入读库后的架构

这个结构的变化会带来两个问题：

数据复制问题。
应用对于数据源的选择问题。

我们希望通过读库来分担主库上读的压力，那么首先就需要解决数据怎么复制到读库的问题。数据库系统一般都提供了数据复制的功能，我们可以直接使用数据库系统的自身机制。但对于数据复制，我们还需要考虑数据复制时延问题，以及复制过程中数据的源和目标之间的映射关系及过滤条件的支持问题。数据复制延迟带来的就是短期的数据不一致。例如我们修改了用户信息，在这个信息还没有复制到读库时（因为延迟），我们从读库上读出来的信息就不是最新的，如果把这个信息给进行修改的人看，就会让他觉得没有修改成功。

不同的数据库系统有不同的支持。例如 MySQL 支持 Master（主库）+Slave（备库）的结构，提供了数据复制的机制。在 MySQL 5.5 之前的版本支持的都是异步的数据复制，会有延迟，并且提供的是完全镜像方式的复制，保证了备库和主库的数据一致性（这里是指不考虑延迟的影响时）。而在 MySQL 5.5 中加入了对 semi-sync 的支持，从数据安全性上来说，它比异步复制要好，不过从我们做读写分离的角度来看，还是存在着复制延迟的可能。再例如 Oracle，之前接触的主要是 Data Guard 方案，这个方案主要用于容灾、数据库保护以及故障恢复等场景，该方案在实施中又分为物理备库（物理 StandBy）和逻辑备库（逻辑 StandBy）。在 Oracle 10g 以前，物理备库是不可读的，但是物理备库保证了日志与主库的一致，是很强的数据保证的做法；而逻辑备库可以提供读服务，不过在有大量更新操作时，它会有非常明显的延迟。

前面描述了这么多，一方面是为了让大家简单了解数据库系统自身的一些支持，另一方面也是为了说明数据库系统层面提供的对数据复制的支持是相对有限的。后面在分布式数据访问层的章节会展开介绍这部分，并详细介绍笔者自己的一些经历。

对于应用来说，增加一个读库对结构变化有一个影响，即我们的应用需要根据不同情况来选择不同的数据库源。写操作要走主库，事务中的读也要走主库，而我们也要考虑到备库数据相对于主库数据的延迟。就是说即便是不在事务中的读，考虑到备库的数据延迟，不同业务下的选择也会有差异。

提到读写分离，我们更多地是想到数据库层面。事实上，广义的读写分离可以扩展到更多的场景。我们看一下读写分离的特点。简单来说就是在原有读写设施的基础上增加了读“库”，更合适的说法应该是增加了读“源”，因为它不一定是数据库，而只是提供读服务的，分担原来的读写库中读的压力。因为我们增加的是一个读“源”，所以需要解决向这个“源”复制数据的问题。

2.2.5.2 搜索引擎其实是一个读库

把搜索引擎列在这里可能出乎很多读者的意料。这里列出搜索引擎不是要讲与搜索相关的技术或方案，而更多的是要介绍大型网站的站内搜索功能。

以我们所举的交易网站为例，商品存储在数据库中，我们需要实现让用户查找商品的功能，尤其是根据商品的标题来查找的功能。对于这样的情况，可能有读者会想到数据库中的 like 功能，这确实是一种实现方式，不过这种方式的代价也很大。还可以使用搜索引擎的倒排表方式，它能够大大提升检索速度。不论是通过数据库还是搜索引擎，根据输入的内容找到符合条件的记录之后，如何对记录进行排序都是很重要的。

搜索引擎要工作，首要的一点是需要根据被搜索的数据来构建索引。随着被搜索数据的变化，索引也要进行改变。这里所说的索引可以理解为前面例子中读库的数据，只不过索引的是真实的数据而不是镜像关系。而引入了搜索引擎之后，我们的应用也需要知道什么数据应该走搜索，什么数据应该走数据库。构建搜索用的索引的过程就是一个数据复制的过程，只不过不是简单复制对应的数据。我们还是看一下引入搜索引擎之后的结构，如图 2-15 所示。

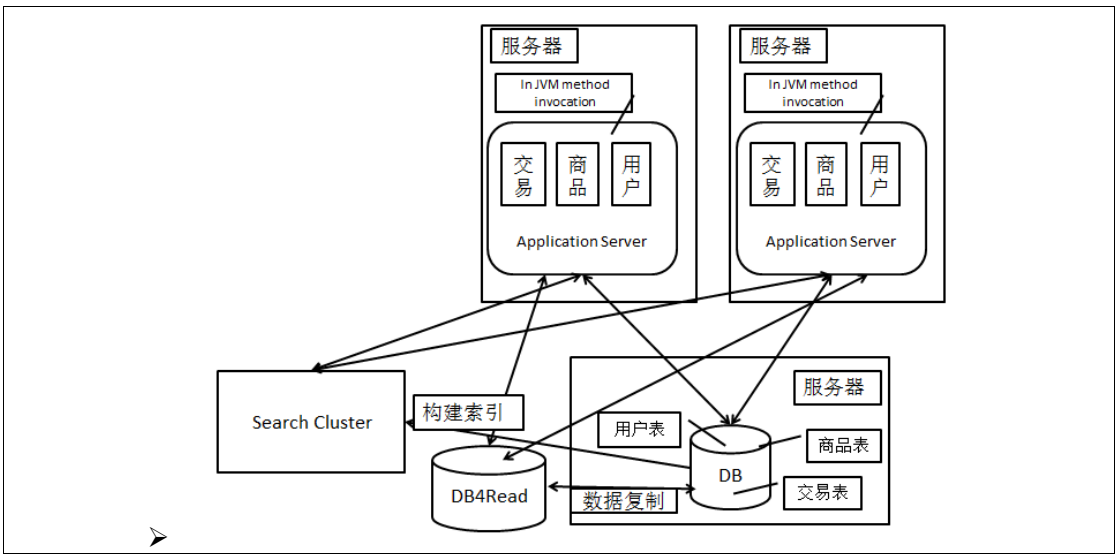


图 2-15 引入搜索引擎的结构

可以看到，搜索集群（Search Cluster）的使用方式和读库的使用方式是一样的。只是构建索引的过程基本都是需要我们自己来实现的。可以从两个维度对于搜索系统构建索引的方式进行划分，一种是按照全量/增量划分，一种是按照实时/非实时划分。全量方式用于第一次建立索引（可能是新建，也可能是重建），而增量方式用于在全量的基础上持续更新索引。当然，增量构建索引的挑战非常大，一般会加入每日的全量作为补充。实时/非实时的划分方式则体现在索引更新的时间上了。我们当然更倾向于实时的方式，之所以有非实时方式，主要是考虑到对数据源头的保护。

总体来说，搜索引擎的技术解决了站内搜索时某些场景下读的问题，提供了更好的查询效率。并且我们看到的站内搜索的结构和使用读库是非常类似的，我们可以把搜索引擎当成一个读库。

2.2.5.3 加速数据读取的利器——缓存

缓存，也就是我们常说的 Cache，来源于 1967 年的一篇电子工程期刊论文。缓存的概念在早期主要用于计算机硬件中，例如 CPU 的高速缓存、硬盘中的缓存等。

我们在这里不讨论这些硬件，而要讨论在大型网站里面起到缓存作用的一些系统，而且我们不是要介绍具体的系统，而是介绍缓存的一些用法，并看看缓存系统是否可以看做一个读库。

1．数据缓存

在大型网站中，有许多地方都会用到缓存机制。首先我们看一下网站内部的数据缓存。大型系统中的数据缓存主要用于分担数据库的读的压力，从目的上看，类似于我们前面提到的分库和搜索引擎。

如图 2-16 所示，可以看到缓存系统和搜索引擎、读库的定位是很类似的，缓存系统一般是用来保存和查询键值（Key-Value）对的。同样的，业务系统需要了解什么数据会在缓存中。缓存中数据的填充方式会有不同，一般我们在缓存中放的是“热”数据而不是全部数据，那么填充方式就是通过应用完成的，即应用访问缓存，如果数据不存在，则从数据库读出数据后放入缓存。随着时间的推移，当缓存容量不够需要清除数据时，最近不被访问的数据就被清除了。这种使用方式与前面分库的数据复制以及搜索引擎的构建索引的方式是不同的。不过还有一种做法与前面两种方式是类似的，那就是在数据库的数据发生变化后，主动把数据放入缓存系统中。这样的好处（相对于前面使用缓存的方式）是，在数据变化时能够及时更新缓存中数据，不会造成读取失效。这种方式一般会用于全数据缓存的情况。使用这种方式有一个要求，即根据数据库记录的变化去更新缓存的代码要能够理解业务逻辑。

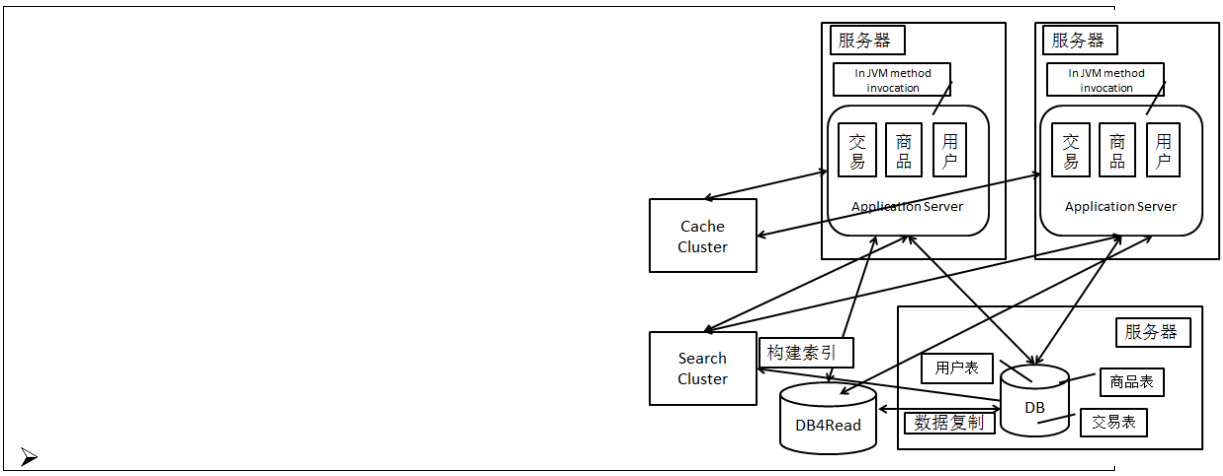


图 2-16　加入缓存后的结构

2．页面缓存

除了数据缓存外，我们还有页面缓存。数据缓存可以加速应用在响应请求时的数据读取速度，但是最终应用返回给用户的主要还是页面，有些动态产生的页面或页面的一部分特别热，我们就可以对这些内容进行缓存。ESI 就是针对这种情况的一个规范。从具体的实现上来说，可以采用 ESI 或者类似的思路来做，也可以把页面缓存与页面渲染放在一起处理，下面举个具体的例子来说明一下这两种做法的差别。

我们的系统使用 Java 技术构建 Web 服务，而在 Web 服务前端有 Apache/Nginx 服务器。

图 2-17 表示对于 ESI 的处理是在 Apache 中进行。Web 服务器产生的请求响应结果返回给 Apache，Apache 中的模块会对响应结果做处理，找到 ESI 标签，然后去缓存中获取这些 ESI 标签对应的内容，如果这些内容不存在（可能没有生成或者已经过期），那么 Apache 中的模板会通过 Web 服务器去渲染这些内容，并且把结果放入缓存中，用内容替换掉 ESI 标签，返回给客户的浏览器。这种方式的职责分工比较清楚。不过 Apache 的 ESI 模块总是要对响应结果做分析，然后进行 ESI 相关的操作。如果在 Web 服务器处理时就能够直接把 ESI 相关的工作做完会是一个更好的选择。

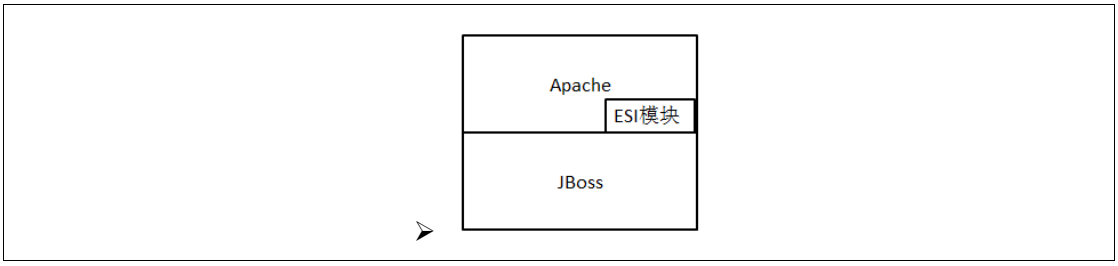


图 2-17　Apache 中的 ESI 模块

图 2-18 就是改进后的样子。Apache 中不再有 ESI 相关的功能了，而是在 Web 服务器中完成渲染及缓存相关的操作。这样的做法更高效，它把渲染与缓存的工作结合在了一起，而且这种做法只是看起来没有前一种方式分工清晰而已。

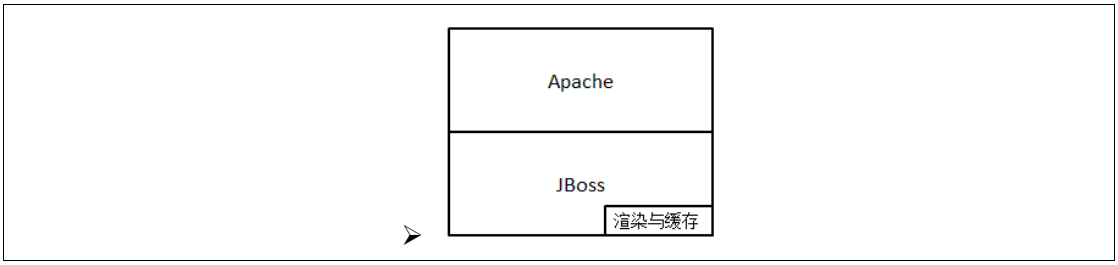


图 2-18　JBoss 中的 ESI 功能

对于使用缓存来加速数据读取的情况，一个很关键的指标是缓存命中率，因为如果缓存命中率比较低的话，就意味着还有不少的读请求要回到数据库中。此外，数据的分布与更新策略也需要结合具体的场景来考虑。从分布上来说，我们主要考虑的问题是需要有机制去避免局部的热点，并且缓存服务器扩容或者缩容要尽量平滑（一致性 Hash 会是不错的选择）。而在缓存的数据的更新上，会有定时失效、数据变更时失效和数据变更时更新的不同选择。

2.2.6 弥补关系型数据库的不足，引入分布式存储系统

在之前的介绍中用于数据存储的主要是数据库，但是在有些场景下，数据库并不是很合适。我们平时使用的多为单机数据库，并且提供了强的单机事务的支持。除了数据库之外，还有其他用于存储的系统，也就是我们常说的分布式存储系统。分布式存储系统在大型网站中有非常广泛的使用。

常见的分布式存储系统有分布式文件系统、分布式 Key-Value 系统和分布式数据库。文件系统是大家所熟知的，分布式文件系统就是在分布式环境中由多个节点组成的功能与单机文件系统一样的文件系统，它是弱格式的，内容的格式需要使用者自己来组织；而分布式 Key-Value 系统相对分布式文件系统会更加格式化一些；分布式数据库则是最格式化的方式了。具体到分布式存储的实现，我们将在后续的章节探讨。

分布式存储系统自身起到了存储的作用，也就是提供数据的读写支持。相对于读写分离中的读“源”，分布式存储系统更多的是直接代替了主库。是否引入分布式系统则需要根据具体场景来选择。分布式存储系统通过集群提供了一个高容量、高并发访问、数据冗余容灾的支持。具体到前文提到的三个常见类，则是通过分布式文件系统来解决小文件和大文件的存储问题，通过分布式 Key-Value 系统提供高性能的半结构化的支持，通过分布式数据库提供一个支持大数据、高并发的数据库系统。分布式存储系统可以帮助我们较好地解决大型网站中的大数据量和高并发访问的问题。引入分布式存储系统后，我们的系统大概会是图 2-19 的样子。

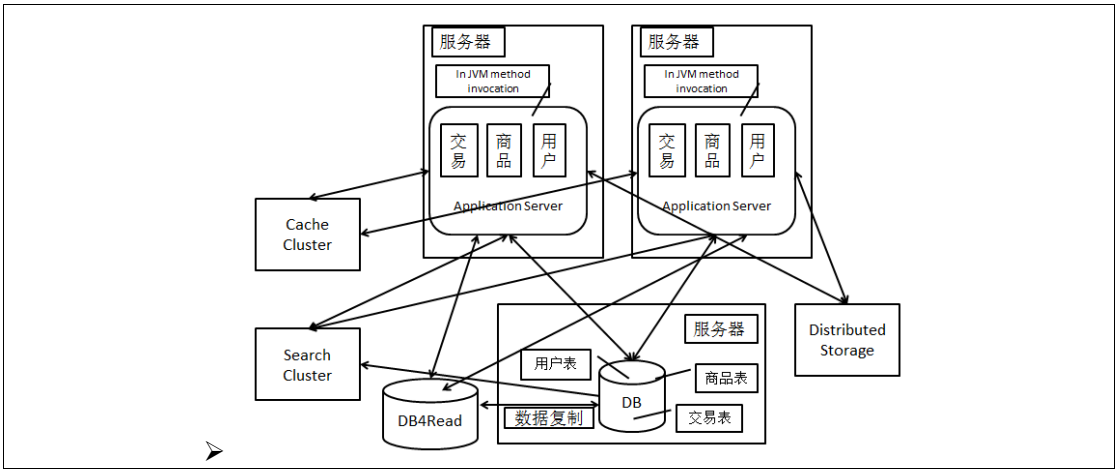


图 2-19　引入分布式存储系统的结构

2.2.7　读写分离后，数据库又遇到瓶颈

通过读写分离以及在某些场景用分布式存储系统替换关系型数据库的方式，能够降低主库的压力，解决数据存储方面的问题。不过随着业务的发展，我们的主库也会遇到瓶颈。我们的网站演进到现在，交易、商品、用户的数据还都在一个数据库中。尽管采取了增加缓存、读写分离的方式，这个数据库的压力还是在继续增加，因此我们需要去解决这个问题，我们有数据垂直拆分和水平拆分两种选择。

2.2.7.1　专库专用，数据垂直拆分

垂直拆分的意思是把数据库中不同的业务数据拆分到不同的数据库中。结合现在的例子，就是把交易、商品、用户的数据分开，如图 2-20 所示。

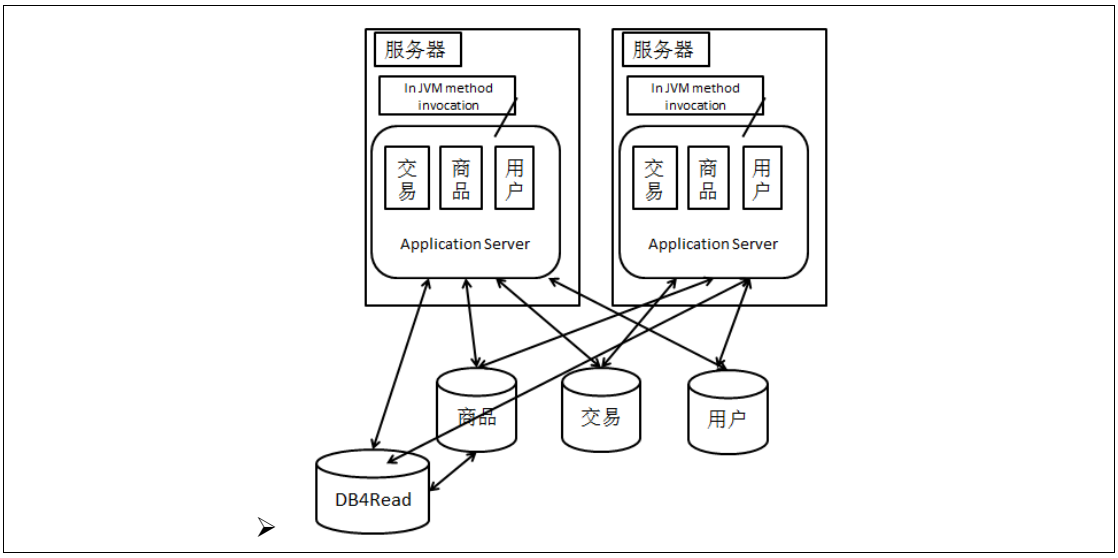


图 2-20　数据库垂直拆分后的结构

这样的变化给我们带来的影响是什么呢？应用需要配置多个数据源，这就增加了所需的配置，不过带来的是每个数据库连接池的隔离。不同业务的数据从原来的一个数据库中拆分到了多个数据库中，那么就需要考虑如何处理原来单机中跨业务的事务。一种办法是使用分布式事务，其性能要明显低于之前的单机事务；而另一种办法就是去掉事务或者不去追求强事务支持，则原来在单库中可以使用的表关联的查询也就需要改变实现了。

对数据进行垂直拆分之后，解决了把所有业务数据放在一个数据库中的压力问题。并且也可以根据不同业务的特点进行更多优化。

2.2.7.2 垂直拆分后的单机遇到瓶颈，数据水平拆分

与数据垂直拆分对应的还有数据水平拆分。数据水平拆分就是把同一个表的数据拆到两个数据库中。产生数据水平拆分的原因是某个业务的数据表的数据量或者更新量达到了单个数据库的瓶颈，这时就可以把这个表拆到两个或者多个数据库中。数据水平拆分与读写分离的区别是，读写分离解决的是读压力大的问题，对于数据量大或者更新量的情况并不起作用。数据水平拆分与数据垂直拆分的区别是，垂直拆分是把不同的表拆到不同的数据库中，而水平拆分是把同一个表拆到不同的数据库中。例如，经过垂直拆分后，用户表与交易表、商品表不在一个数据库中了，如果数据量或者更新量太大，我们可以进一步把用户表拆分到两个数据库中，它们拥有结构一模一样的用户表，而且每个库中的用户表都只涵盖了一部分的用户，两个数据库的用户合在一起就相当于没有拆分之前的用户表。我们先来简单看一下引入数据水平拆分后的结构，如图 2-21 所示。

我们来分析一下水平拆分后给业务应用带来的影响。

首先，访问用户信息的应用系统需要解决 SQL 路由的问题，因为现在用户信息分在了两个数据库中，需要在进行数据库操作时了解需要操作的数据在哪里。

此外，主键的处理也会变得不同。原来依赖单个数据库的一些机制需要变化，例如原来使用 Oracle 的 Sequence 或者 MySQL 表上的自增字段的，现在不能简单地继续使用了。并且在不同的数据库中也不能直接使用一些数据库的限制来保证主键不重复了。

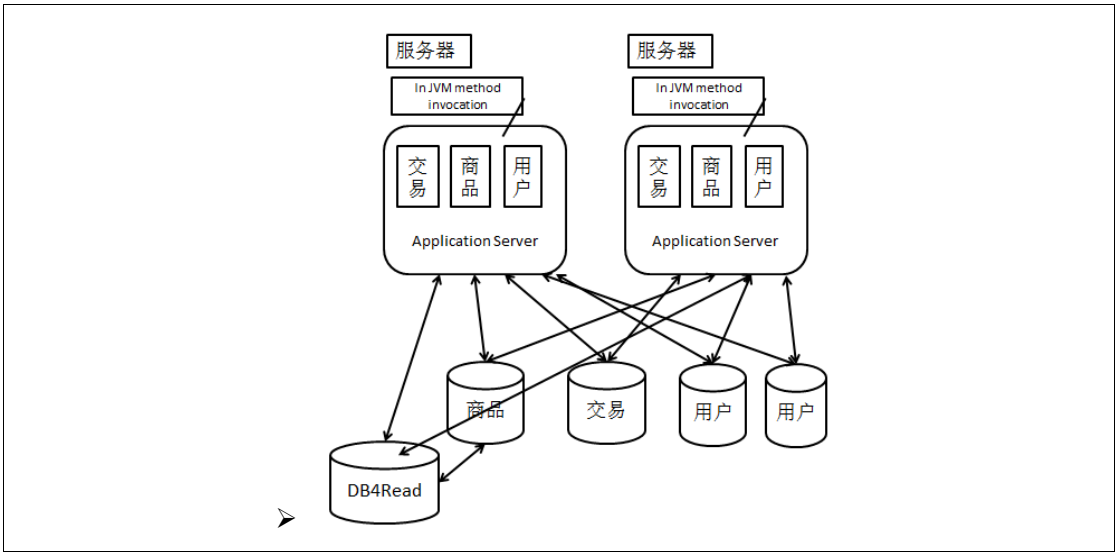


图 2-21 数据水平拆分后的结构

最后，由于同一个业务的数据被拆分到了不同的数据库中，因此一些查询需要从两个数据库中取数据，如果数据量太大而需要分页，就会比较难处理了。

不过，一旦我们能够完成数据的水平拆分，我们将能够很好地应对数据量及写入量增长

的情况。具体如何完成数据水平拆分，在后面分布式数据访问层的章节中我们将进行更加详细的介绍。

2.2.8 数据库问题解决后，应用面对的新挑战

2.2.8.1 拆分应用

前面所讲的读写分离、分布式存储、数据垂直拆分和数据水平拆分都是在解决数据方面的问题。下面我们来看看应用方面的变化。

之前解决了应用服务器从单机到多机的扩展，应用就可以在一定范围内水平扩展了。随着业务的发展，应用的功能会越来越多，应用也会越来越大。我们需要考虑如何不让应用持续变大，这就需要把应用拆开，从一个应用变为两个甚至多个应用。我们来看两种方式。

第一种方式，根据业务的特性把应用拆开。在我们的例子中，主要的业务功能分为三大部分：交易、商品和用户。我们可以把原来的一个应用拆成分别以交易和商品为主的两个应用，对于交易和商品都会有涉及用户的地方，我们让这两个系统自己完成涉及用户的工作，而类似用户注册、登录等基础的用户工作，可以暂时交给两系统之一来完成（注意，我们在这里主要是通过例子说明拆分的做法），如图 2-22 所示，这样的拆分可以使大的应用变小。

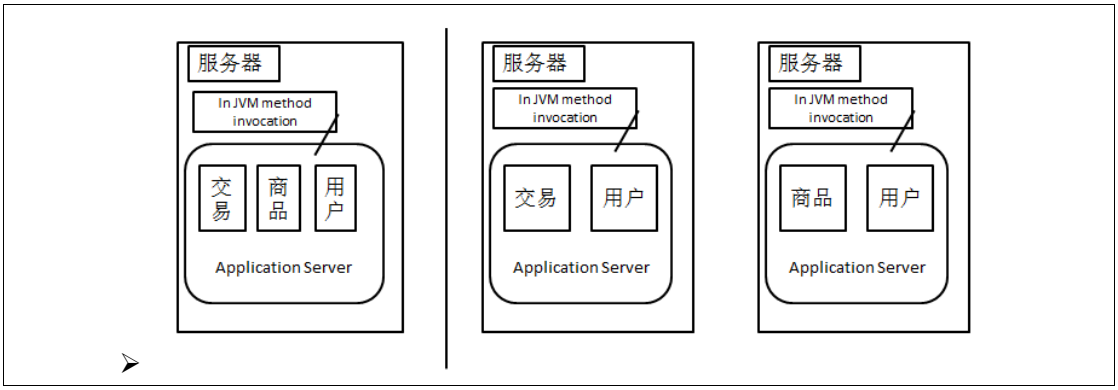


图 2-22 根据功能拆分应用

我们还可以按照用户注册、用户登录、用户信息维护等再拆分，使之变成三个系统。不过，这样拆分后在不同系统中会有一些相似的代码，例如用户相关的代码。如何能够保证这部分代码的一致以及如何对其复用是需要解决的问题。此外，按这样的方式拆分出来的新系统之间一般没有直接的相互调用。而且，新拆出来的应用可能会连接同样的数据库。

来看一个具体的例子，如图 2-23 所示。

我们根据业务的不同功能拆分了几个业务应用，而且这些业务应用之间不存在直接的调用，它们都依赖底层的数据库、缓存、文件系统、搜索等。这样的应用拆分确实能够解决当

下的一些问题，不过也有一些缺点。

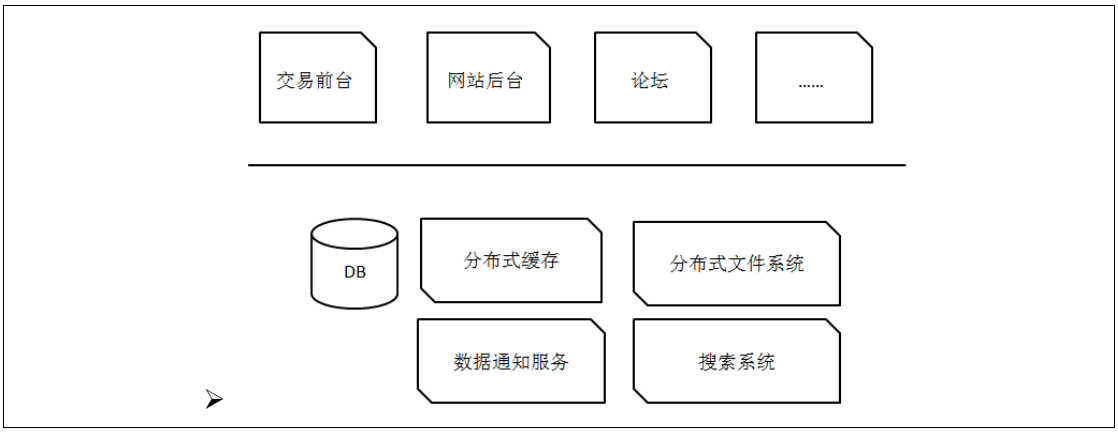


图 2-23　按功能拆分后的结构

2.2.8.2　走服务化的路

我们再来看一下服务化的做法。图 2-24 是一个示意图。从中可以看到我们把应用分为了三层，处于最上端的是 Web 系统，用于完成不同的业务功能；处于中间的是一些服务中心，不同的服务中心提供不同的业务服务；处于下层的则是业务的数据库。当然，我们在这个图中省去了缓存等基础的系统，因此可以说是服务化系统结构的一个简图。

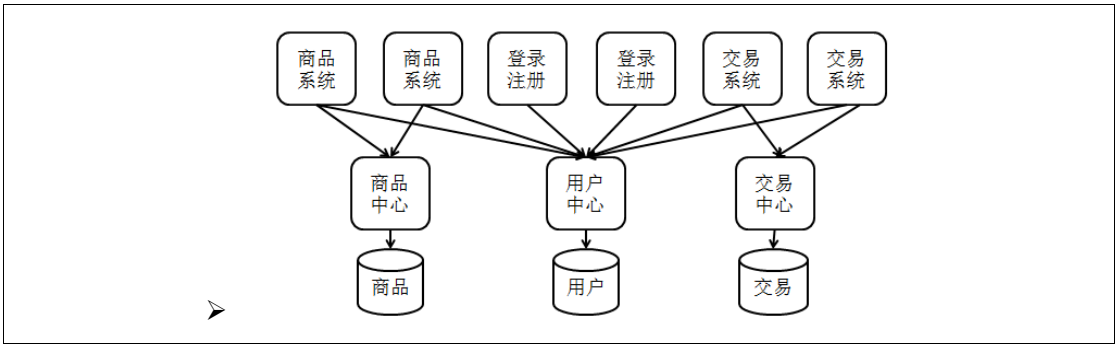


图 2-24　服务化结构

图 2-24 与之前的图相比有几个很重要的变化。首先，业务功能之间的访问不仅是单机内部的方法调用了，还引入了远程的服务调用。其次，共享的代码不再是散落在不同的应用中了，这些实现被放在了各个服务中心。第三，数据库的连接也发生了一些变化，我们把与数据库的交互工作放到了服务中心，让前端的 Web 应用更加注重与浏览器交互的工作，而不必过多关注业务逻辑的事情。连接数据库的任务交给相应的业务服务中心了，这样可以降低数据库的连接数。而服务中心不仅把一些可以共用的之前散落在各个业务的代码集中了起来，并且能够使这些代码得到更好的维护。第四，通过服务化，无论是前端 Web 应用还是服务中心，都可以是由固定小团队来维护的系统，这样能够更好地保持稳定性，并能更好地

控制系统本身的发展，况且稳定的服务中心日常发布的次数也远小于前端 Web 应用，因此这个方式也减小了不稳定的风险。

要做到服务化还需要一些基础组件的支撑，在后面服务框架的章节我们会具体介绍。

2.2.9 初识消息中间件

最后我们来看一下消息中间件。维基百科上对消息中间件的定义为“**Message-oriented middleware**（**MOM**） is software infrastructure focused on sending and receiving messages between distributed systems.”意思就是面向消息的系统（消息中间件）是在分布式系统中完成消息的发送和接收的基础软件。图 2-25 更直观地表示了消息中间件。

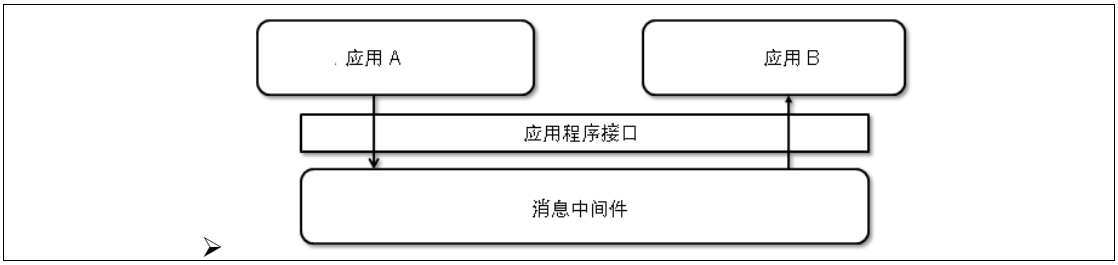


图 2-25 消息中间件

消息中间件有两个常被提及的好处，即异步和解耦。从图 2-25 中可以看到，应用 A 和应用 B 都和消息中间件打交道，而这两个应用之间并不直接联系。这样就完成了解耦，目的是希望收发消息的双方彼此不知道对方的存在，也不受对方影响，所以将消息投递给接收者实际上都采用了异步的方式。在后面消息中间件的章节（第 6 章）中会展开来讲相关内容。

2.2.10 总结

至此，我们通过一个例子来讲解了交易网站的架构演进。这里我必须要再强调一下，这只是一个例子，不是某个网站真实的演进过程，实际的网站演进过程与自身的业务和不同时间遇到的问题有密切关系，没有固定的模式。我们是希望通过这个例子向大家讲述可能遇到的问题类型和基本的解决思路。这些思路在具体的实现过程中都有更多的工作和选择要做。后面关于 Java 中间件的实践部分（第 3 章）会继续讲解一些更细节的内容。

最后，我们通过一张图来看看经过演进之后，我们的网站变成什么样子了，如图 2-26 所示。

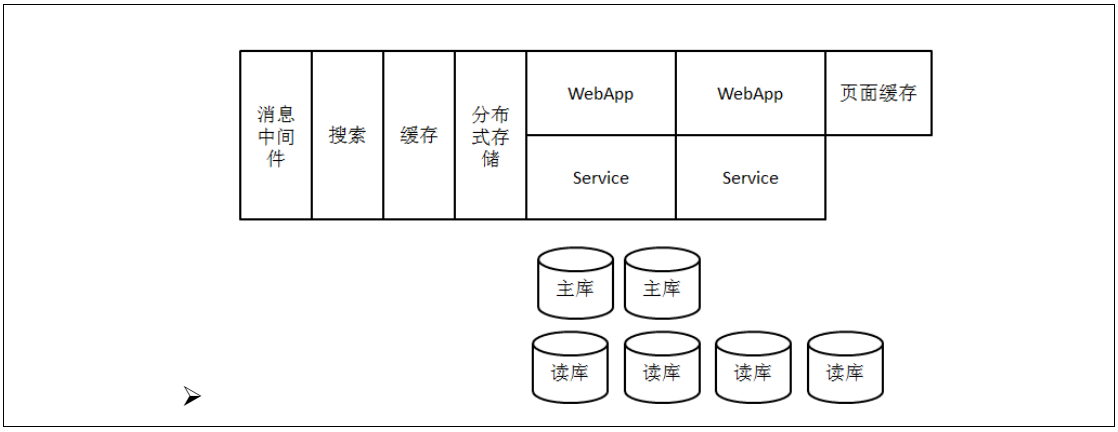


图 2-26　整体结构图

这比起最初的图 2-4 已经丰满了很多。在后面介绍 Java 中间件实践时，我们可以看到 Java 中间件在这个图上的位置。而在介绍完 Java 中间件实践以及构建大型网站的其他要素后，将会给出一张更完整的图。