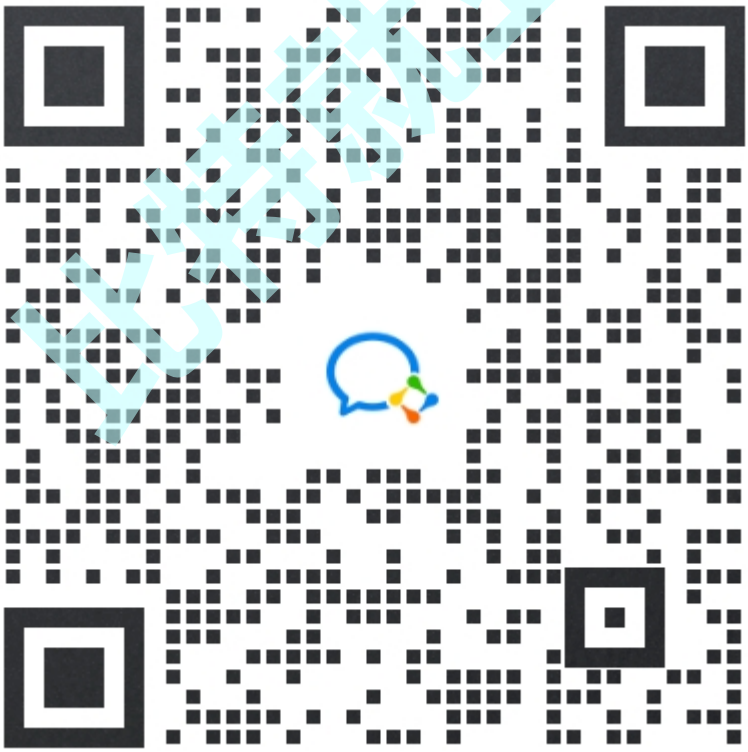
服务端高并发分布式结构演进之路

版权说明

对比特课程感兴趣，可以联系这个微信。



代码 & 板书链接

https://gitee.com/HGtz2222/bitproject/tree/master/redis

概述

在进行技术学习过程中，由于大部分读者没有经历过一些中大型系统的实际经验，导致无法从全局理解一些概念，所以本文以一个 "电子商务" 应用为例，介绍从一百个到千万级并发情况下服务端的架构的演进过程，同时列举出每个演进阶段会遇到的相关技术，让大家对架构的演进有一个整体的认知，方便大家对后续知识做深入学习时有一定的整体视野。

常见概念

在正式引入架构演进之前，为避免读者对架构中的概念完全不了解导致低效沟通，优先对其中一些比较重要的概念做前置介绍：

基本概念

应用（Application）/ 系统（System）

为了完成一整套服务的一个程序或者一组相互配合的程序群。生活例子类比：为了完成一项任务，而搭建的由一个人或者一群相互配的人组成的团队。

模块（Module）/ 组件（Component）

当应用较复杂时，为了分离职责，将其中具有清晰职责的、内聚性强的部分，抽象出概念，便于理解。生活例子类比：军队中为了进行某据点的攻克，将人员分为突击小组、爆破小组、掩护小组、通信小组等。

分布式（Distributed）

系统中的多个模块被部署于不同服务器之上，即可以将该系统称为分布式系统。如 Web 服务器与数据库分别工作在不同的服务器上，或者多台 Web 服务器被分别部署在不同服务器上。生活例子类比：为了更好的满足现实需要，一个在同一个办公场地的工作小组被分散到多个城市的不同工作场地中进行远程配合工作完成目标。跨主机之间的模块之间的通信基本要借助网络支撑完成。

集群（Cluster）

被部署于多台服务器上的、为了实现特定目标的一个/组特定的组件，整个整体被称为集群。比如多个 MySQL 工作在不同服务器上，共同提供数据库服务目标，可以被称为一组数据库集群。生活例子类比：为了解决军队攻克防守坚固的大城市的作战目标，指挥部将大批炮兵部队集中起来形成一个炮兵打击集群。

分布式 vs 集群。通常不用太严格区分两者的细微概念，细究的话，分布式强调的是物理形态，即工作在不同服务器上并且通过网络通信配合完成任务；而集群更在意逻辑形态，即是否为了完成特定服务目标。

主（Master）/ 从（Slave）

集群中，通常有一个程序需要承担更多的职责，被称为主；其他承担附属职责的被称为从。比如 MySQL 集群中，只有其中一台服务器上数据库允许进行数据的写入（增/删/改），其他数据库的数据修改全部要从这台数据库同步而来，则把那台数据库称为主库，其他数据库称为从库。

中间件（Middleware）

一类提供不同应用程序用于相互通信的软件，即处于不同技术、工具和数据库之间的桥梁。生活例子类比：一家饭店开始时，会每天去市场挑选买菜，但随着饭店业务量变大，成立一个采购部，由采购部专职于采买业务，称为厨房和菜市场之间的桥梁。

评价指标（Metric）

可用性（Availability）

考察单位时间段内，系统可以正常提供服务的概率/期望。例如：年化系统可用性 = 系统正常提供服务时长 / 一年总时长。这里暗含着一个指标，即如何评价系统提供无法是否正常，我们就不深入了。平时我们常说的 4 个 9 即系统可以提供 99.99% 的可用性，5 个 9 是 99.999% 的可用性，以此类推。我们平时只是用高可用（High Availability HA）这个非量化目标简要表达我们系统的追求。

响应时长（Response Time RT）

指用户完成输入到系统给出用户反应的时长。例如点外卖业务的响应时长 = 拿到外卖的时刻 - 完成点单的时刻。通常我们需要衡量的是最长响应时长、平均响应时长和中位数响应时长。这个指标原则上是越小越好，但很多情况下由于实现的限制，需要根据实际情况具体判断

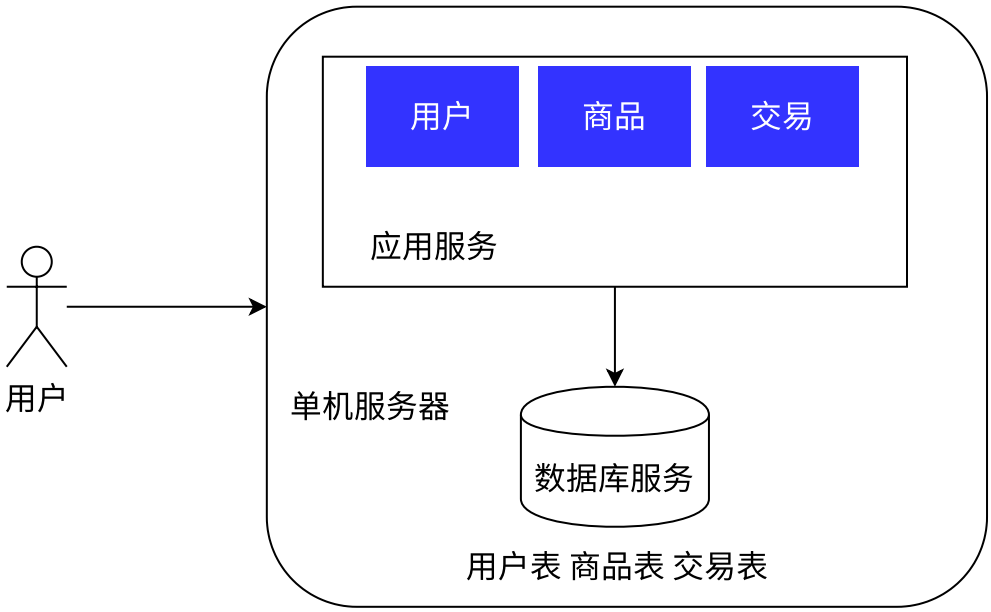
吞吐（Throughput）vs 并发（Concurrent）

吞吐考察单位时间段内，系统可以成功处理的请求的数量。并发指系统同一时刻支持的请求最高量。例如一条辆车道高速公路，一分钟可以通过 20 辆车，则并发是 2，一分钟的吞吐量是 20。实践中，并发量往往无法直接获取，很多时候都是用极短的时间段（比如 1 秒）的吞吐量做代替。我们平时用高并发（Hight Concurrnet）这个非量化目标简要表达系统的追求。

架构演进

单机架构

初期，我们需要利用我们精干的技术团队，快速将业务系统投入市场进行检验，并且可以迅速响应变化要求。但好在前期用户访问量很少，没有对我们的性能、安全等提出很高的要求，而且系统架构简单，无需专业的运维团队，所以选择单机架构是合适的。



用户在浏览器中输入 www.bit.com，首先经过 DNS 服务将域名解析成 IP 地址 10.102.41.1，随后浏览器访问该 IP 对应的应用服务。

相关软件

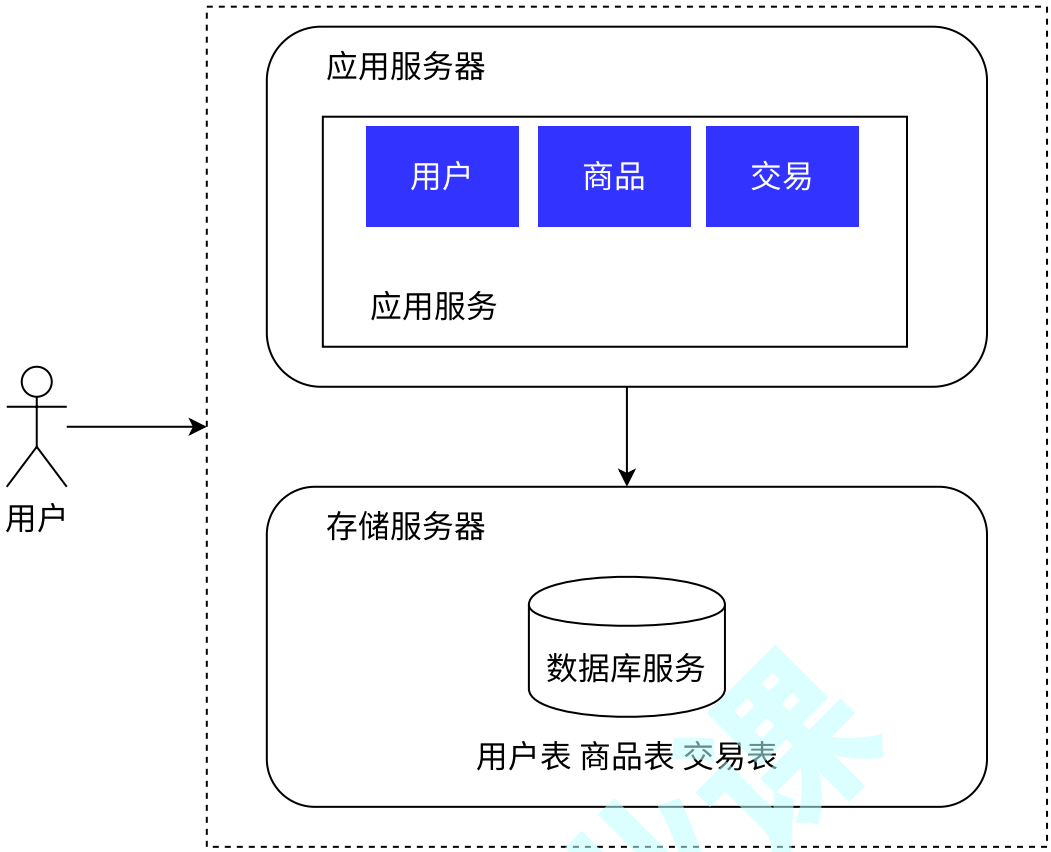
Web 服务器软件：Tomcat、Netty、Nginx、Apache 等

数据库软件：MySQL、Oracle、PostgreSQL、SQL Server 等

> 同学们目前的训练大多针对该阶段的业务系统，包括本科毕业设计的系统实现。

应用数据分离架构

随着系统的上线，我们不出意外地获得了成功。市场上出现了一批忠实于我们的用户，使得系统的访问量逐步上升，逐渐逼近了硬件资源的极限，同时团队也在此期间积累了对业务流程的一批经验。面对当前的性能压力，我们需要未雨绸缪去进行系统重构、架构挑战，以提升系统的承载能力。但由于预算仍然很紧张，我们选择了将应用和数据分离的做法，可以最小代价的提升系统的承载能力。



和之前架构的主要区别在于将数据库服务独立部署在同一个数据中心的其他服务器上，应用服务通过网络访问数据。

应用服务集群架构

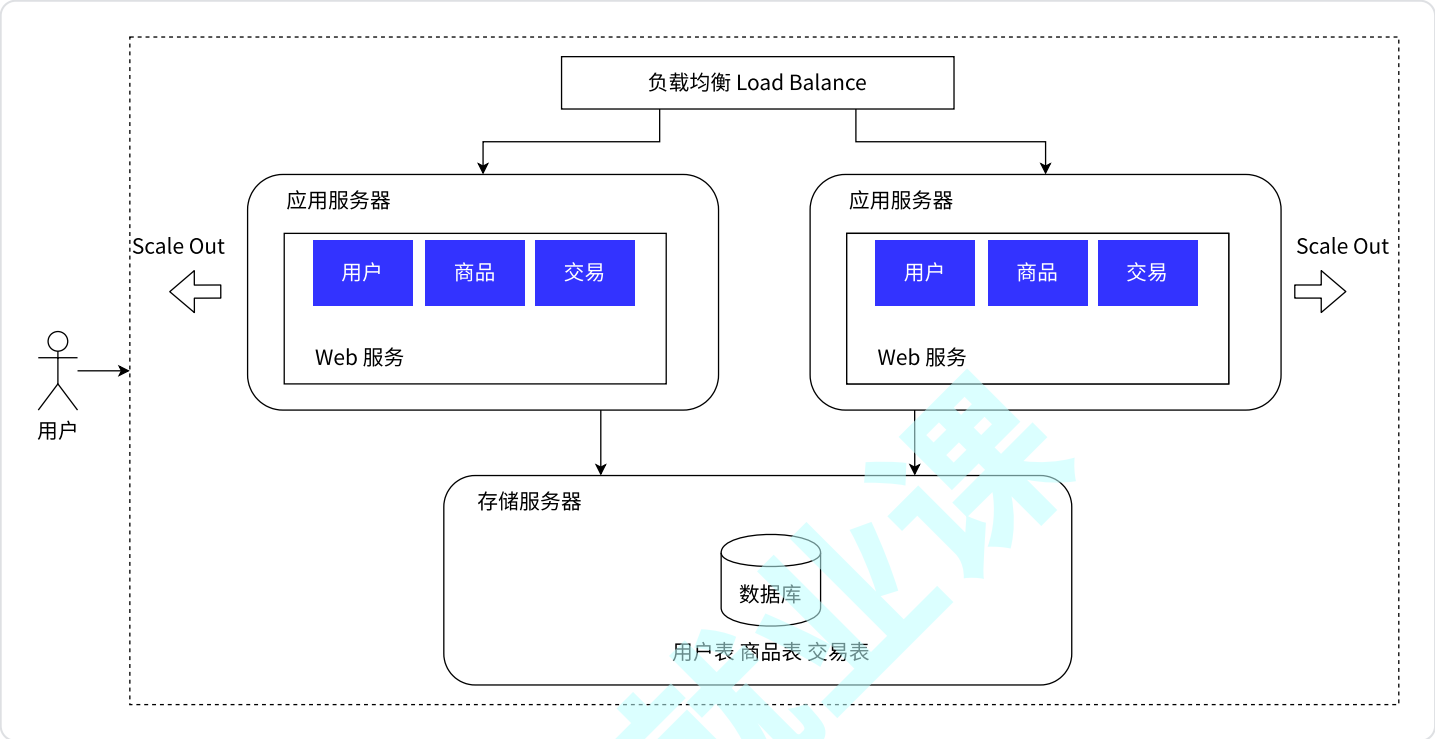
我们的系统受到了用户的欢迎，并且出现了爆款，单台应用服务器已经无法满足需求了。我们的单机应用服务器首先遇到了瓶颈，摆在我们技术团队面前的有两种方案，大家针对方案的优劣展示了热烈的讨论：

- 垂直扩展 / 纵向扩展 Scale Up。通过购买性能更优、价格更高的应用服务器来应对更多的流量。这种方案的优势在于完全不需要对系统软件做任何的调整；但劣势也很明显：硬件性能和价格的增长关系是非线性的，意味着选择性能 2 倍的硬件可能需要花费超过 4 倍的价格，其次硬件性能提升是有明显上限的。
- 水平扩展 / 横向扩展 Scale Out。通过调整软件架构，增加应用层硬件，将用户流量分担到不同的应用层服务器上，来提升系统的承载能力。这种方案的优势在于成本相对较低，并且提升的上限空间也很大。但劣势是带给系统更多的复杂性，需要技术团队有更丰富的经验。

经过团队的学习、调研和讨论，最终选择了水平扩展的方案，来解决该问题，但这需要引入一个新的组件 —— 负载均衡：为了解决用户流量向哪台应用服务器分发的问题，需要一个专门的系统组件做流量分发。实际中负载均衡不仅仅指的是工作在应用层的，甚至可能是其他的网络层之中。同时流量调度算法也有很多种，这里简单介绍几种较为常见的：

- Round-Robin 轮询算法。即非常公平地将请求依次分给不同的应用服务器。

- Weight-Round-Robin 轮询算法。为不同的服务器（比如性能不同）赋予不同的权重（weight），能者多劳。
- 一致哈希散列算法。通过计算用户的特征值（比如 IP 地址）得到哈希值，根据哈希结果做分发，优点是确保来自相同用户的请求总是被分给指定的服务器。也就是我们平时遇到的专项客户经理服务。



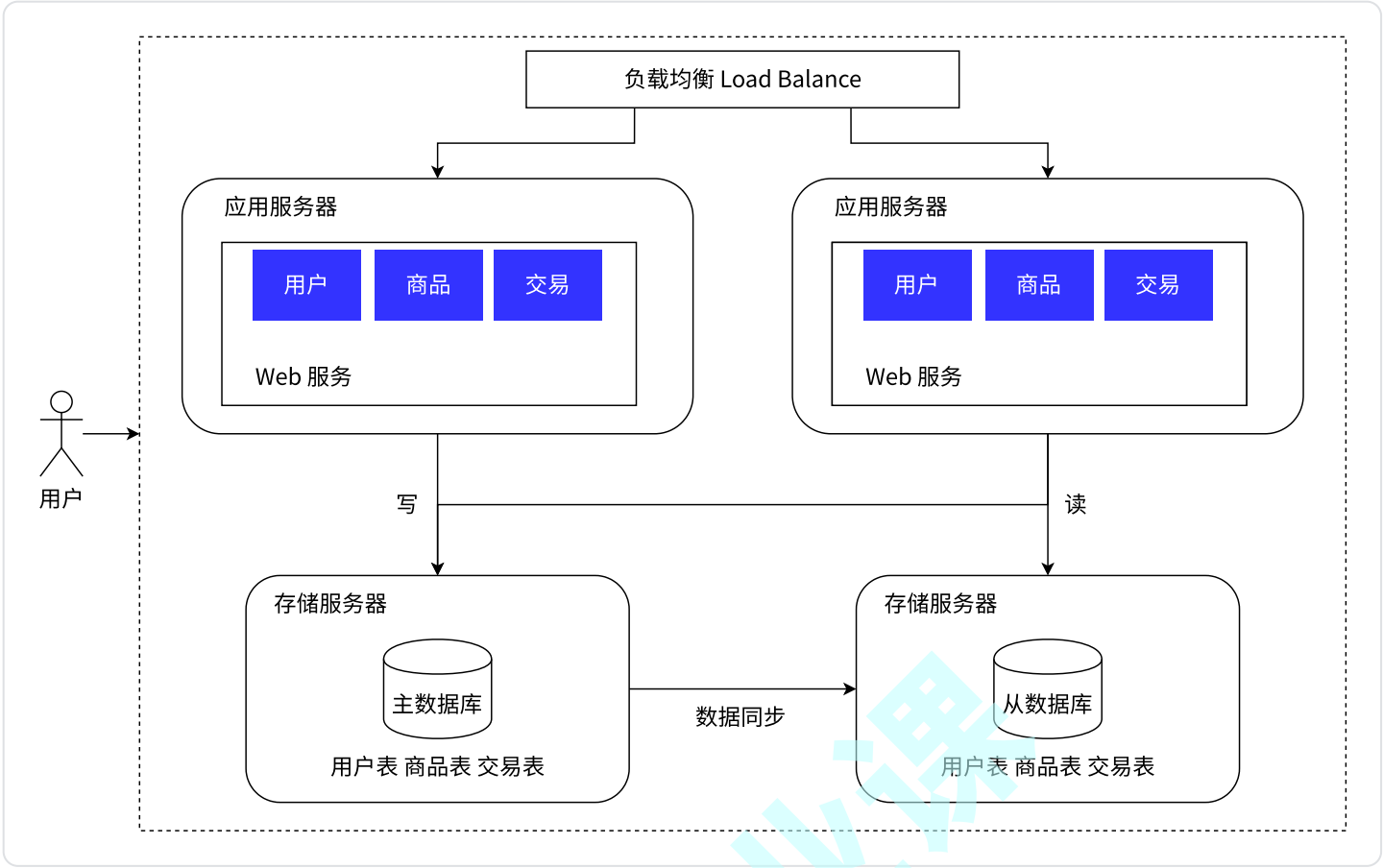
相关软件

负载均衡软件：Nginx、HAProxy、LVS、F5 等

读写分离 / 主从分离架构

上一节提到，我们把用户的请求通过负载均衡分发到不同的应用服务器之后，可以并行处理了，并且可以随着业务的增长，可以动态扩张服务器的数量来缓解压力。但是现在的架构里，无论扩展多少台服务器，这些请求最终都会从数据库读写数据，到一定程度之后，数据的压力称为系统承载能力的瓶颈点。我们可以像扩展应用服务器一样扩展数据库服务器么？答案是否定的，因为数据库服务有其特殊性：如果将数据分散到各台服务器之后，数据的一致性将无法得到保障。所谓数据的一致性，此处是指：针对同一个系统，无论何时何地，我们都应该看到一个始终维持统一的数据。想象一下，银行管理的账户金额，如果收到一笔转账之后，一份数据库的数据修改了，但另外的数据库没有修改，则用户得到的存款金额将是错误的。

我们采用的解决办法是这样的，保留一个主要的数据库作为写入数据库，其他的数据库作为从属数据库。从库的所有数据全部来自主库的数据，经过同步后，从库可以维护着与主库一致的数据。然后为了分担数据库的压力，我们可以将写数据请求全部交给主库处理，但读请求分散到各个从库中。由于大部分的系统中，读写请求都是不成比例的，例如 100 次读 1 次写，所以只要将读请求由各个从库分担之后，数据库的压力就没有那么大了。当然这个过程不是无代价的，主库到从库的数据同步其实是由时间成本的，但这个问题我们暂时不做进一步探讨。



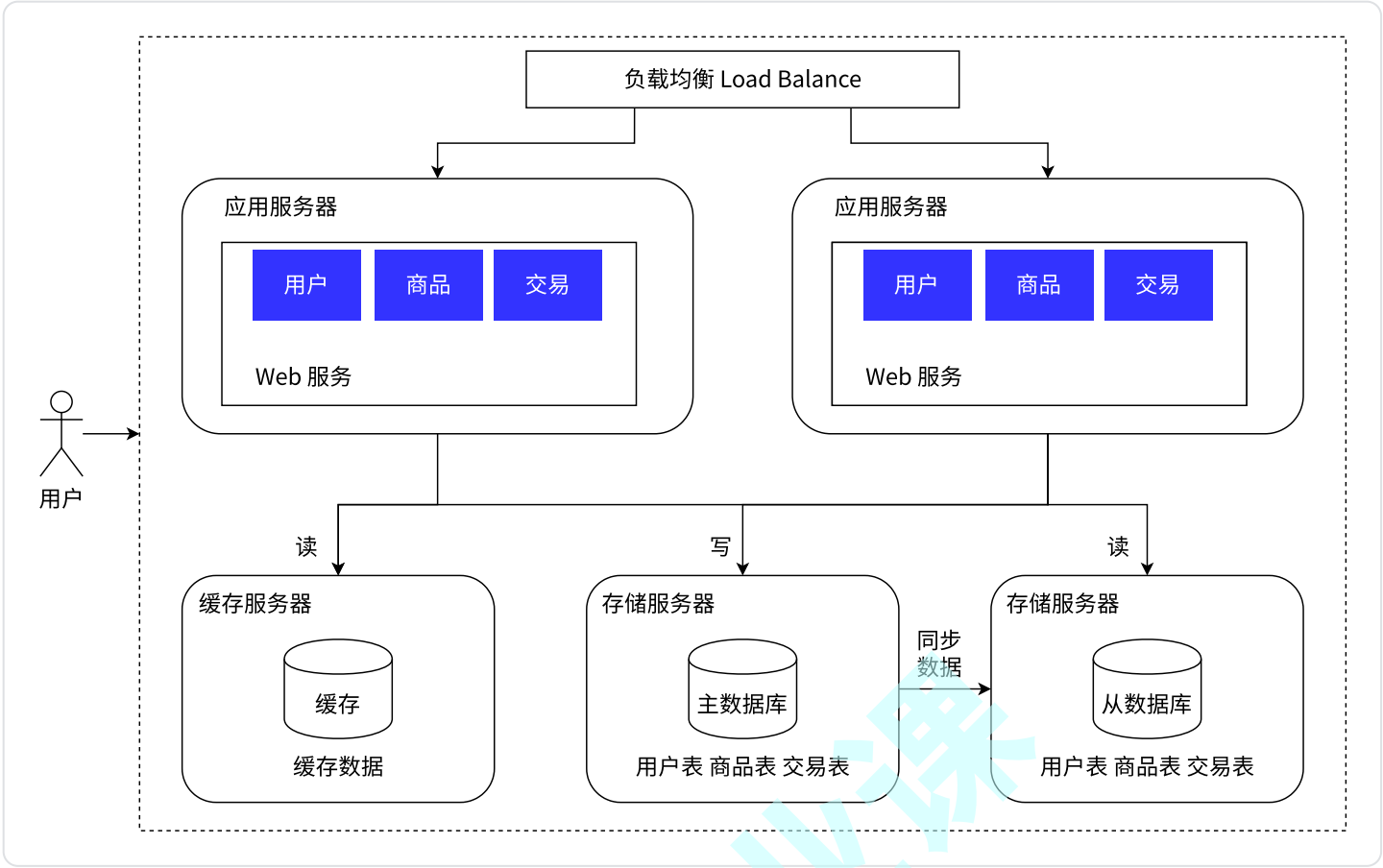
应用中需要对读写请求做分离处理，所以可以利用一些数据库中间件，将请求分离的职责托管出去。

相关软件

MyCat、TDDL、Amoeba、Cobar 等类似数据库中间件等

引入缓存 —— 冷热分离架构

随着访问量继续增加，发现业务中一些数据的读取频率远大于其他数据的读取频率。我们把这部分数据称为热点数据，与之相对应的是冷数据。针对热数据，为了提升其读取的响应时间，可以增加本地缓存，并在外部增加分布式缓存，缓存热门商品信息或热门商品的 html 页面等。通过缓存能把绝大多数请求在读写数据库前拦截掉，大大降低数据库压力。其中涉及的技术包括：使用memcached作为本地缓存，使用 Redis 作为分布式缓存，还会涉及缓存一致性、缓存穿透/击穿、缓存雪崩、热点数据集中失效等问题。

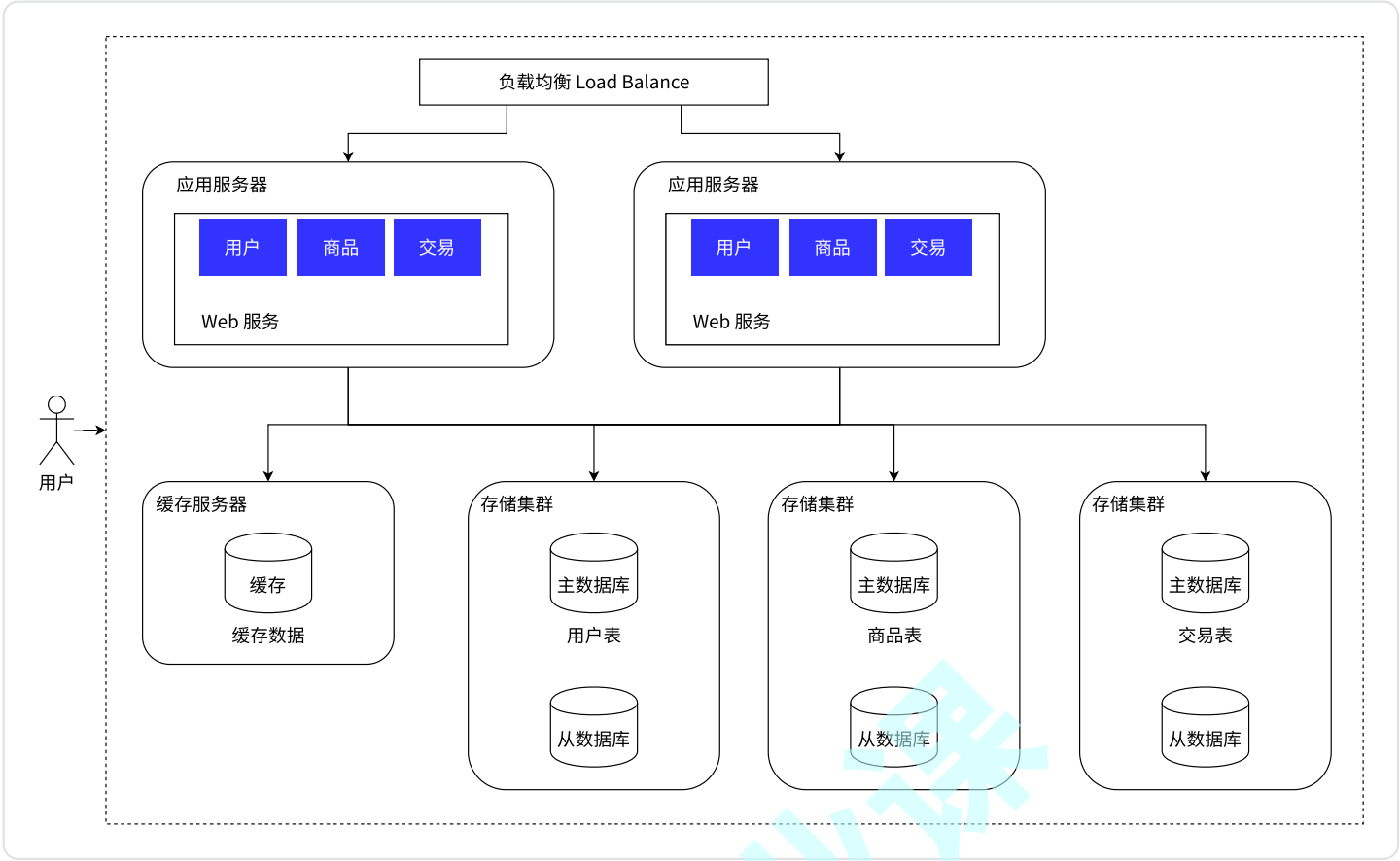


相关软件：

Memcached、Redis 等缓存软件

垂直分库

随着业务的数据量增大，大量的数据存储在同一个库中已经显得有些力不从心了，所以可以按照业务，将数据分别存储。比如针对评论数据，可按照商品ID进行hash，路由到对应的表中存储；针对支付记录，可按照小时创建表，每个小时表继续拆分为小表，使用用户ID或记录编号来路由数据。只要实时操作的表数据量足够小，请求能够足够均匀的分发到多台服务器上的小表，那数据库就能通过水平扩展的方式来提高性能。其中前面提到的Mycat也支持在大表拆分为小表情况下的访问控制。这种做法显著的增加了数据库运维的难度，对DBA的要求较高。数据库设计到这种结构时，已经可以称为分布式数据库，但是这只是一个逻辑的数据库整体，数据库里不同的组成部分是由不同的组件单独来实现的，如分库分表的管理和请求分发，由Mycat实现，SQL的解析由单机的数据库实现，读写分离可能由网关和消息队列来实现，查询结果的汇总可能由数据库接口层来实现等等，这种架构其实是MPP（大规模并行处理）架构的一类实现。

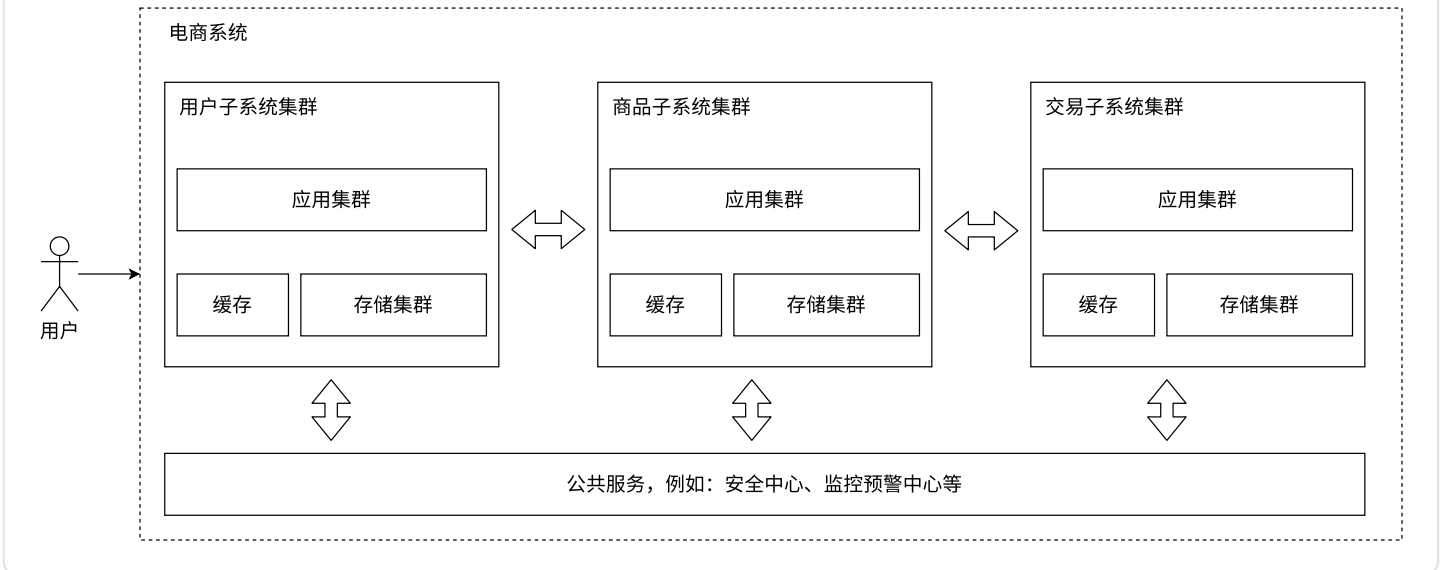


相关软件

Greenplum、TiDB、Postgresql XC、HAWQ等，商用的如南大通用的GBase、睿帆科技的雪球DB、华为的LibrA 等

业务拆分 —— 微服务

随着人员增加，业务发展，我们将业务分给不同的开发团队去维护，每个团队独立实现自己的微服务，然后互相之间对数据的直接访问进行隔离，可以利用 Gateway、消息总线等技术，实现相互之间的调用关联。甚至可以把一些类似用户管理、安全管理、数据采集等业务提成公共服务。



尾声

至此，一个还算合理的高可用、高并发系统的基本雏形已显。注意，以上所说的架构演变顺序只是针对某个侧面进行单独的改进，在实际场景中，可能同一时间会有几个问题需要解决，或者可能先达到瓶颈的是另外的方面，这时候就应该按照实际问题实际解决。如在政府类的并发量可能不大，但业务可能很丰富的场景，高并发就不是重点解决的问题，此时优先需要的可能会是丰富需求的解决方案。

对于单次实施并且性能指标明确的系统，架构设计到能够支持系统的性能指标要求就足够了，但要留有扩展架构的接口以便不备之需。对于不断发展的系统，如电商平台，应设计到能满足下一阶段用户量和性能指标要求的程度，并根据业务的增长不断的迭代升级架构，以支持更高的并发和更丰富的业务。

所谓的“大数据”其实是海量数据采集清洗转换、数据存储、数据分析、数据服务等场景解决方案的一个统称，在每一个场景都包含了多种可选的技术，如数据采集有Flume、Sqoop、Kettle等，数据存储有分布式文件系统HDFS、FastDFS，NoSQL数据库HBase、MongoDB等，数据分析有Spark技术栈、机器学习算法等。总的来说大数据架构就是根据业务的需求，整合各种大数据组件组合而成的架构，一般会提供分布式存储、分布式计算、多维分析、数据仓库、机器学习算法等能力。而服务端架构更多指的是应用组织层面的架构，底层能力往往是由大数据架构来提供。