



# 高并发的哲学原理

《Philosophical Principles of High Concurrency》简称 [《PPHC》](#)

stars 3k

followers 2.8k

Words 167765

## 前言

### 写作目标

本书的目标是在作者有限的认知范围内，讨论一下高并发问题背后隐藏的一个哲学原理——找出单点，进行拆分。

### 内容梗概

我们将从动静分离讲起，一步步深入 Apache、Nginx、epoll、虚拟机、k8s、异步非阻塞、协程、应用网关、L4/L7 负载均衡器、路由器(网关)、交换机、LVS、软件定义网络(SDN)、Keepalived、DPDK、ECMP、全冗余架构、用户态网卡、集中式存储、分布式存储、PCIe 5.0、全村的希望 CXL、InnoDB 三级索引、内存缓存、KV 数据库、列存储、内存数据库、Shared-Nothing、计算存储分离、Paxos、微服务架构、削峰、基于地理位置拆分、高可用等等等等。并最终基于地球和人类社会的基本属性，设计出可以服务地球全体人类的高并发架构。

全部共 167765 字。

## 读者评价

会上一谈到架构和 I/O，我都想到你的文章。主讲解答清楚和没解答清楚的，都没你的文章清楚。

—— 秋收，于 RubyConf 2023

像看小说一样把文章都看完了，全程无尿点，作者的脑袋是在哪里开过光，知识储备竟如此扎实

—— 观东山

---

非常棒的技术分享！深入浅出，娓娓道来，让我想起了那本 csapp。

—— drhrchen

---

拜读了！应该算是架构顶级总结！！

—— 雨山前

---

看完了 博主好厉害 学习到了各种骚技巧 和知识 膜拜

—— evanxian

---

写的太好了，不仅充满了理工科的严谨较真，也充满了文科的浪漫

—— 一秒

---

写得很好，视角也是我喜欢的，站在地球表面，述事宏大，思维自信。

—— 纳秒时光

---

全部看完，博主太强了，很受启发

—— Bruce

---

棒

—— JuniaWonter

---

## 作者信息

---

吕文翰

1. GitHub: [johnlui](#)
2. 职位：住范儿创始成员，CTO，监事

## 高并发系统处理经验

1. 2017 年维护的单体 CMS 系统顶住了每日两百万 PV 的压力
2. 2020 年优化一个单机 PHP 商城顶住了 QPS 1000+ 的压力
3. 2021 年设计的分布式电商秒杀系统在实际业务中跑到了最高一分钟 GMV 500 万，QPS 10000+

## 一些微小的成就

1. Laravel 在中国最早的布道者之一
2. Swift 语言贡献者
3. Github 中国区 Swift 语言第 3 (Star 数排序)
4. Github 中国区 PHP 语言第 5 (Star 数排序)

## 开源作品

Go:

1. 开源互联网搜索引擎: <https://github.com/johnlui/DIYSearchEngine>

PHP:

1. 优雅的 Web DSL (框架) : <https://pinatra.github.io>
2. 轻量级 PHP 框架 TinyLara: <https://tinylara.github.io>
3. 阿里云 OSS SDK (16 万下载) : <https://github.com/johnlui/AliyunOSS>

iOS:

1. 再造 “手机QQ” 侧滑菜单: <https://github.com/johnlui/SwiftSideslipLikeQQ>
2. 纯 Swift 编写的提示组件 SwiftNotice: <https://github.com/johnlui/SwiftNotice>
3. 适合大文件上传的 Swift 网络请求库。 <https://github.com/johnlui/Pitaya>
4. 基于 WKWebView 的高性能 Cordova 替代, WebAPP 容器: <https://github.com/Lucky-Orange/BlackHawk>
5. Swift JSON 库: <https://github.com/johnlui/JSONNeverDie>

## 系列文章

PHP:

1. 《利用 Composer 一步一步构建自己的 PHP 框架》
2. 《利用 Composer 完善自己的 PHP 框架》
3. 《Laravel 5 系列入门教程》
4. 《PHP 命名空间 解惑》
5. 《深入理解 Laravel Eloquent》

iOS:

1. [《Auto Layout 使用心得》](#)
2. [《再造“手机QQ”侧滑菜单》](#)
3. [《可视化编程 Tips 系列文章》](#)
4. [《自己动手写一个 iOS 网络请求库》](#)
5. [《自己动手打造基于 WKWebView 的混合开发框架》](#)

获取作者后续作品



# 微信搜一搜

岁寒博客

## 版权声明



本书版权归属于吕文翰，采用署名—非商业性使用—禁止演绎 4.0 协议（CC BY-NC-ND 4.0）协议开源，供互联网用户免费阅读。

纸质图书正在由电子工业出版社出版中，敬请期待。

在遵循许可的前提下，你可以自由地共享，包括在任何媒介上以任何形式复制、发行本作品。但要求你：

### 署名

应在使用本文档的全部或部分内容时候，注明原作者及来源信息，包括：

1. 作者姓名：吕文翰
2. 原始网站地址：<https://pphc.lvwenhan.com>

### 非商业性使用：

不得用于商业出版或其他任何带有商业性质的行为。如需商业使用，请联系作者取得书面授权。

### 禁止演绎

不得发布任何基于本作品创作、复制的演绎作品，只允许原文传播。

# 目录

## 内容动态统计

共有 12 章, 83 篇文章, 总计 167765 字。

## I. 第一部分 通用设计方法

- 1. 高并发问题的通用设计方法 142 字
  - 1.1. 概述 994 字
  - 1.2. 每秒一百万次 HTTPS 请求 602 字
  - 1.3. 动态、静态资源分开部署 573 字
  - 1.4. 数据库独立部署 692 字
  - 1.5. 真实业务经历: CMS 网站 1879 字
  - 1.6. 实战: 虚拟电商平台“静山” 1950 字
  - 1.7. 现实世界中的高并发场景 4149 字
  - 1.8. 面试题 1889 字

## II. 第二部分 计算资源高并发

- 2. 基础设施并发 318 字
  - 2.1. 概述 916 字
  - 2.2. 服务器虚拟化 3347 字
  - 2.3. 虚拟化兼容性 3124 字
  - 2.4. 容器 3468 字
  - 2.5. Kubernetes (K8s) 2867 字
  - 2.6. Spring Cloud 507 字
  - 2.7. 软件架构的本质 677 字
  - 2.8. 实战: Docker 部署静山平台 2250 字
  - 2.9. 面试题 963 字
- 3. 突破编程语言的性能瓶颈 358 字
  - 3.1. 概述 280 字
  - 3.2. 互联网系统规模发展史 1941 字
  - 3.3. 后端语言变迁史 3390 字
  - 3.4. 语言特性如何决定性能 3486 字
  - 3.5. 实战: Go 协程 2544 字
  - 3.6. 面试题 1364 字

## III. 第三部分 网络资源高并发

- 4. 至关重要的 Web Server 311 字
  - 4.1. Apache 1163 字
  - 4.2. Apache、Nginx 的性能差异 837 字

4.3. Nginx 与 epoll 的协同工作	1472 字
4.4. 三种进程模型的技术特点	1847 字
4.5. 笔者的电商秒杀真实经验	893 字
4.6. 面试题	615 字
5. 负载均衡和应用网关	58 字
5.1. 概述	477 字
5.2. 如何支持 50000 QPS	2677 字
5.3. 单机 Kong 的性能极限	1270 字
5.4. 分层的网络	2004 字
5.5. 负载均衡器的工作原理	2094 字
5.6. 面试题	1050 字
6. SDN 搭建负载均衡集群	141 字
6.1. 概述	461 字
6.2. 负载均衡发展史	1982 字
6.3. 交换机	1152 字
6.4. LVS 技术解析	2213 字
6.5. Keepalived 高可用	821 字
6.6. 突破单台服务器的性能极限	2491 字
6.7. 硬件厂商的软件设计	470 字
6.8. 面试题	2080 字
IV. 第四部分 数据库高并发	
7. 数据库及其背后的存储	142 字
7.1. 概述	235 字
7.2. 数据库是个大单点	1607 字
7.3. 存储技术简史	1766 字
7.4. x86 的 I/O 性能劣势	1981 字
7.5. x86 内存技术的演进	718 字
7.6. 面试题	992 字
8. MySQL InnoDB 详解	220 字
8.1. 概述	974 字
8.2. B+ 树	1410 字
8.3. InnoDB 数据插入测试	2681 字
8.4. “2000W 行分表” 问题	1508 字
8.5. 内存缓存: Buffer Pool	1657 字
8.6. 面试题	1197 字
9. 四代分布式数据库的变迁	158 字
9.1. 单机数据库的不可能三角	809 字
9.2. 从读写分离到分布式	958 字
9.3. 第一代分布式: 中间件	788 字

9.4. 第二代分布式: KV	621 字
9.5. 第三代分布式: NewSQL	1783 字
9.6. 第四代分布式: 云上数据库	3736 字
9.7. 番外篇	1224 字
9.8. 面试题	2163 字
10. TiDB 和 OceanBase	263 字
10.1. TiDB 的设计思路	621 字
10.2. OceanBase 设计思路	2412 字
10.3. 如何抉择	421 字
10.4. 面试题	1662 字
11. 缓存与队列	69 字
11.1. 概述	458 字
11.2. 缓存设计实战	1691 字
11.3. 缓存的读写策略	5201 字
11.4. 队列	834 字
11.5. 真实的队列秒杀架构	1217 字
11.6. 缓存和队列的架构意义	587 字
11.7. 面试题	1645 字
V. 第五部分 无限容量架构	
12. 无限容量架构	164 字
12.1. 概述	183 字
12.2. 从业务分库到微服务	852 字
12.3. 削峰	1296 字
12.4. 站在地球表面	2321 字
12.5. 番外篇: 高可用	1270 字
12.6. 面试题	2980 字
VI. 其它系列文章	
. 2023	
. 自己动手开发搜索引擎	13118 字
. 2018	
. 性能之殇	13350 字
. 软件工程师需要的网络知识	10885 字
VII. 第三方技术资料	207 字

# 第一部分 通用设计方法

## 第 1 章 高并发问题的通用设计方法

## 第 1 章 高并发问题的通用设计方法

“高并发”是后端领域的技术圣杯，是一种可遇而不可求的特殊业务需求。

对于每一名后端工程师来说，实现高并发的系统都是一项颇具挑战性的任务。为了应对这一挑战，你需要对网络、缓存、并发、并行、分布式计算以及数据库优化等技术有深入的理解和实际经验。在这个领域，没有虚假的掩饰，只有真实的实力。

### 1.1 概述

我们都知道，互联网服务的核心价值在于流量，流量越大，平台的盈利机会和发展空间就越大，这也是为何大厂总是倾向于招聘拥有高并发经验的研发人员。2007 年，中国电子商务总和交易额突破一万亿人民币，网民数量快速增加，中国互联网正式进入了高并发时代。之后，大厂与创业公司之间的技术壁垒不断加强。

如今，掌握高并发技术已成为大厂技术人员的基本要求。

当然，大厂可以为我们提供高并发系统架构设计的实践机会。然而，如果你没有相关的架构设计经验，大厂的大门又怎会向你敞开呢？这就形成了一个逻辑悖论。

难道我们只能一直陷在这个困境中吗？实际上，我们自己也是这些高并发应用的用户，例如使用微信、使用搜索引擎、刷微博、刷短视频。我们只需要将自己从用户角色转换为设计者的角色，想象自己成为这些大厂的架构师，设身处地思考：如果我来设计这个系统，我该怎么做。

尽管我们都知道高并发的重要性，并学习了许多高并发系统设计的技术资料，但仍然会感到困惑：为什么我们仍然缺乏对设计一个完整的高并发系统的了解？

这是因为你缺乏真实业务场景中的架构设计经验。没有亲自接触过真实的业务需求，就无法感受到面对压力、迎接挑战以及完成任务后的喜悦和轻松。只有通过真实的现场感受，我们学到的各种技术才能真正融会贯通，而不只是一堆零碎知识的杂乱集合。就像那句著名的“我听过很多道理，却仍然过不好这一生”。归根结底，还是缺乏实践。

通过阅读本书，你可以“化身”为那个真的需要解决高并发问题的人，借助笔者的真实高并发处理经验，身临其境地站在架构师的角度，深入理解高并发系统设计的哲学原理。

## 找出单点，进行拆分

开门见山，先说结论，高并发的哲学原理就是——找出单点，进行拆分。要将每一个“大单点”都拆成“一个小单点 + 多个资源并行”的形式。

### “单点”的定义

单点指的是一个 Web 系统中，几乎所有流量都必须经过的特定部分，如果它出现瓶颈或故障，将导致整个系统的性能下降或不可用。单点是一个逻辑概念，在系统架构层面它可能指代各种不同层面的软件、硬件实体。在传统的单体式应用中，可以说遍地都是单点。

在解决高并发问题的过程中，我们会不断地遇到各种单点：Web Server、单个操作系统、虚拟化/容器技术、编程语言运行架构、网络、Unix 进程模型、数据库等。每遇到一个单点，我们都要见招拆招，使用架构工具拆掉它。计算机的虚拟化程度非常高，理论上每个单点都是可以继续往下拆分的。

## 1.2 设定目标：每秒一百万次 HTTPS 请求

---

本书的主题既然是高并发原理与实战，那就要先设定一个高并发的目标：每秒一百万次 HTTPS 请求，即 1,000,000 QPS (Queries Per Second)。某些场景中，也称为：1,000,000 TPS (Transactions Per Second)。

### 性能问题要靠架构解决

在正式展开之前，我们需要明确高并发问题的基本解决思路：系统的性能问题需要通过架构设计来解决。

面对高并发技术需求时，在架构上进行优化是最为简单、对系统稳定性影响最低且最容易获得收益的方法。

即便在我们专注于单个资源的性能优化时，例如 MySQL 单机性能优化（软件优化）或 x86 CPU 多核性能提升（硬件优化），从微观角度来看，这些优化措施实际上也是在进行架构优化——通过调整软件或硬件的运行架构，提升总体性能。

## 没有银弹

“没有银弹”是计算机世界的第一准则，你想获得性能收益，就一定要拿出一些东西，和“信息之神”交换。

架构优化的本质就是拿其它资源或者指标来交换性能，系统总体性能的提升必然需要伴随着某种资源的更多消耗或者某个非关键指标的劣化，像我们常说的“时间换空间”“空间换时间”都是这个道理的实际体现。

## 我们讨论哪个高并发？

本文讨论的是“Web 服务高并发”问题，典型场景为电商秒杀：同一个时刻，数万人抢同一个低价商品，会给系统的每一个层面都造成显著的性能瓶颈，每年的双 11 大促就是这一场景的极致体现。

接下来，请大家跟着笔者一起，沿着 [计算高并发](#) -> [网络高并发](#) -> [数据库（存储）高并发](#) 的道路一步一步将系统性能的上限从单机 100 QPS 提升到 1,000,000（一百万）QPS。

## 1.3 动态、静态资源分开部署

动静分离是高并发架构设计的第一步，部分场景下还是收益最大的那一步，因为我们可以把 99% 的流量压力都转移出去。

### Apache 和 Nginx 的性能差距

十年前，笔者用 4200 元从同学那里买到了一台二手的 Macbook Pro。彼时，Macbook Pro 还是满身接口的厚家伙，也没有细腻的视网膜屏幕，CPU 是 i5 双核，内存只有 4GB，但就是在那台电脑上，笔者使用一张静态图片测试了 Apache 和 Nginx 的性能差距：两万对八万，Nginx 的性能是 Apache 的四倍。

### 加一个 Nginx 软件竟然还能减少 CPU 和内存占用？

如果你的应用现在还在用 Apache 承载全部流量，那只需要在前面加一个 Nginx，承载静态资源的分发，动态请求可以原封不动地使用 http 协议发送给 Apache，只需要这样一个小操作，便可以在服务器配置不变的情况下把系统容量提升一倍以上。此时的系统架构图如图 1-1 所示。

此外，由于 Nginx 针对新图片格式（如 .webp）和视频流式传输等进行了专门的优化，在增加了一个软件后，服务器的 CPU 和内存资源消耗反而会明显下降。在如今流量越来越便宜的时代，静态资源的体积和流量呈指数级增长，Nginx 的性能优势还在持续扩大。

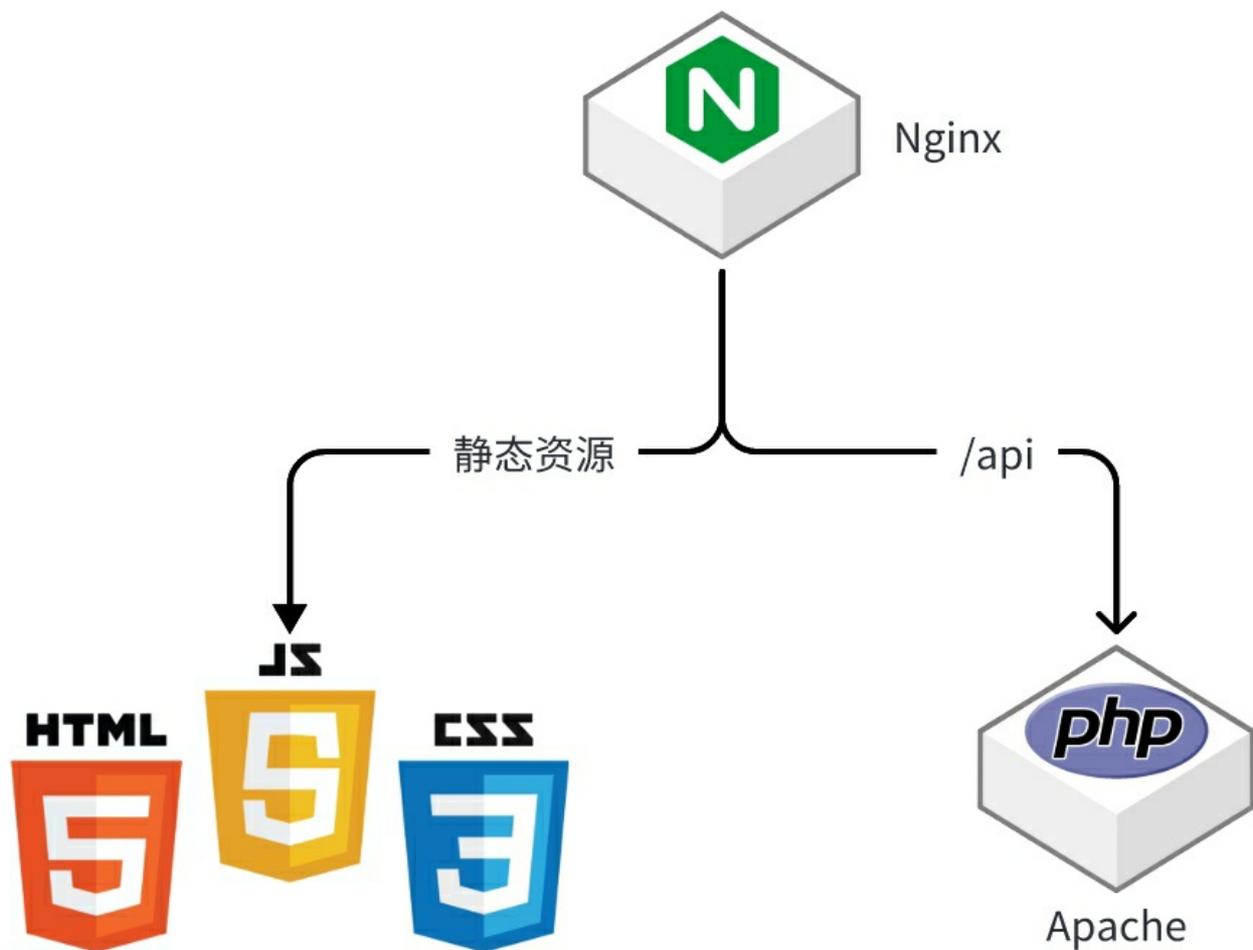


图 1-1 Nginx 承载静态资源分发

## 使用云服务

如果你在用云服务的话，那么把静态资源全部交给云服务商的 CDN 来承载，还可以再获得 90% 的 CPU 节省。同时，CDN 的流量费还比云主机的流量费更便宜，时至今日中国 CDN 市场已经卷的不像样子了。此时的系统架构图如图 1-2 所示。

一个网站

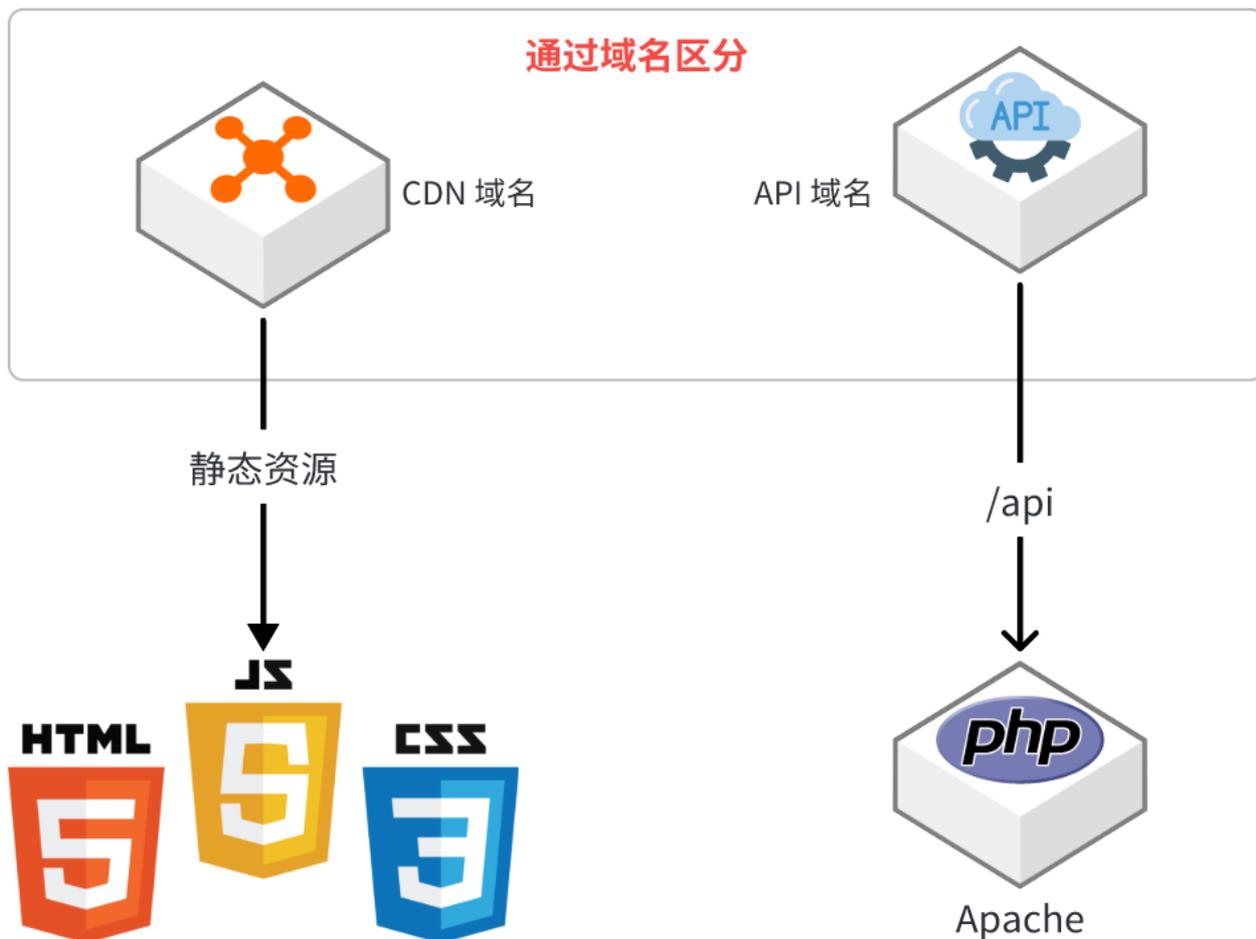


图 1-2 静态资源全部使用云服务商的 CDN 服务

## 1.4 数据库独立部署

在使用 Nginx 承载全部静态资源以后，如果你的低配云主机还是扛不住流量，该怎么办呢？这个时候就需要引入第二台云主机了：专门用来跑数据库。

将后端代码和数据库部署在同一台机器上是“灾难架构”

出于节省成本的需要，把后端代码和数据库部署在同一台机器上确实是一个无奈的选择。但是，我们要清楚的是，这是一种错误的架构设计方案，它只适合超低流量的系统。

一旦系统承受的压力稍大，便会面临“债股双杀”的局面：

1. CPU 被耗尽导致 MySQL 响应变慢；
2. 应用代码需更长时间等待，虽不额外消耗 CPU 资源，却占用大量内存；
3. 系统内存被占用导致 InnoDB 缓存被系统回收，进一步降低了 MySQL 的运行速度；
4. 最终形成“内卷”和“踩踏”，系统性能急剧下降，服务可能完全崩溃。

## MySQL 单独部署时性能非常出色

只要你将 MySQL 与后端代码的 CPU 进行隔离，数据库的极限性能就能达到够用的水平。根据笔者的实测结果，即使是一台1核2G的MySQL服务器，也能够达到200QPS（Query Per Second，每秒执行的SQL语句数）的每秒执行SQL语句数，足以支撑一个每天一百万PV的小网站。

## MySQL 单独部署架构图

应用服务器和 MySQL 数据库服务器分为两台机器部署时，系统架构图如图 1-3 所示。

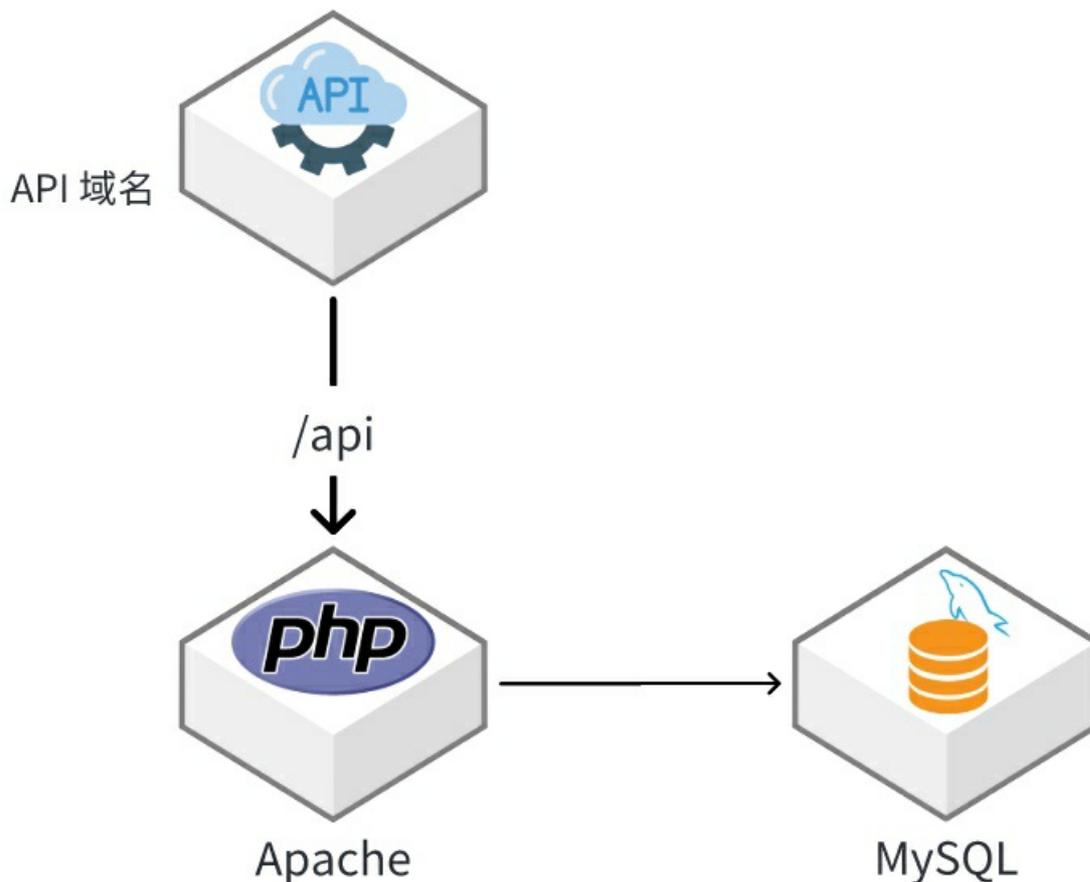


图 1-3 MySQL 单独部署架构图

## 运维哲学初探

在这个小小的例子中，我们已经在不经意间窥探到了过去数十年运维技术圈的基本哲学：从物理机到虚拟机，从虚拟主机到云主机，再从云主机到 Kubernetes，虽然推动运维架构演进的是越来越复杂的软件架构和越来越多的计算资源需求，但是运维的基本哲学是不变的：

运维的核心价值不在于资源的扩充，而在于资源的隔离。

我们将在后面关于云服务和服务器硬件发展的讨论中进一步探讨这一运维哲学。

## 1.5 真实业务经历：CMS 网站

六年前，笔者曾经负责维护过一个面向 SEO 的 CMS（内容管理系统）网站，该网站每日 PV 达到两百万。这类网站往往有很多的“相关内容”需求，需要进行类似于搜索引擎的“相关查询”，导致页面相应十分缓慢。由于内容的独特性，经常受到爬虫和采集机器人的关注，导致频繁被爬取，给运维工作带来了很大压力。

随着网站内容的逐渐丰富，除了被正规蜘蛛访问外，某一天还突然遭遇了每秒 100 次 HTTP 请求的采集机器人袭击，当时的配置是可怜的 1 核 2G 云主机和 1 核 1G 的 MySQL 数据库，网站瞬间就宕机了。后来，在网站恢复以后，笔者测试了一下，由于页面的复杂度非常高，当 QPS 达到 7 以后，数据库就会满载，随后云主机也会满载，网站就挂了。

那么，笔者是如何解决这个问题呢？

### 动静分离

首先，笔者实施了动静分离的策略。将静态资源全部使用 CDN 进行分发，由于 CDN 节点具备强缓存的特点，只需文件名不变，即可维持长时间的缓存效果。这一举措显著降低了静态资源对服务器的负载，使系统的总 CPU 消耗下降了 80%。

### 利用 Elasticsearch 提升网页响应速度

当时，这个网站使用关键词 like 的方法来实现“类似文章推荐”功能。具体来说，我们预设了一个关键词库，然后根据页面上的文章标题和内容进行匹配，匹配到的第一个关键词就会被用于各个表中的 SQL 查询语句，以获取相关内容。然而，这种方法在数据量庞大的情况下会出现显著的性能问题。

为了解决这个问题，笔者采购了一台 8 核 32GB 的云主机，并在其上安装了一个单节点的 Elasticsearch。通过将搜索任务交由 Elasticsearch 处理，不仅成功降低了单页面的响应时间，从 300-500ms 降至约 200ms 左右，同时推荐内容的精准性也得到了显著提升。

### MySQL 索引优化

在绝大多数情况下，为关系型数据库添加索引都是首选的性能优化策略。这种方法通常可以带来数十倍到数万倍的性能提升，尽管需要增加明显的磁盘消耗和略微的内存消耗作为代价。

例如，笔者为一张主要的大表增加了一些简单的单字段索引，结果页面的响应时间就从 200ms 降低到了大约 150ms，效果显著。

### 利用 Redis 缓存

在页面展示的过程中，有很多很难被优化的高耗时操作，是无法利用索引来做性能优化的，例如大表的 `select count(*)`。于是，本着“解决不了问题就解决提问题的人”的思想，笔者将大表的总数统计工作利用后台定时任务来承载，强制压制了业务需求：每个页面无法实时地获取文章总数了，只能获取到一分钟之前的数据。

每分钟，笔者会统计一次那几个内容表的总数，存入 Redis，而页面如果需要使用，直接读 Redis 中的数字即可。经过这一步的优化，笔者把页面的响应时间从 150ms 降低到了 120ms。

## 网络请求并行化

由于每个页面都会获取多个其他表的“类似文章推荐”，所以每个页面对 Elasticsearch 的 HTTP 请求都有多个，而 PHP 默认是一个一个阻塞运行的：上一个不回来，PHP 进程就等待。笔者将这个阻塞操作并行化了：虽然 PHP 是一种阻塞语言，但是一次性发送多个 HTTP 请求的能力还是有的，等到所有请求都返回了数据之后，再继续阻塞地运行 PHP 代码。

这个系统平均每个页面请求了五次 Elasticsearch，每次 15ms。并行化之前，这五个请求总共需要消耗 75 ms。并行化之后，总时间从 75ms 减少到了 25ms。

经过上述的优化措施，笔者将单个页面的平均响应时间压缩到了 70ms，这个数字和之前的 300~500ms 比是一个飞跃，在同样后端计算资源的情况下，系统容量提升了五倍左右，我们 1 核 2G 的小云主机也能顶住 100 QPS 的压力了。

## 为什么不做静态化？

这个时候可能有人会问了，既然是内容网站，为什么不静态化呢？因为数据量太大了，500 万个页面，一个页面 100KB，就是 476GB 的磁盘容量，这个量级太大了。

在这个规模下，和缓慢但可用的数据库相比，这么多静态资源的管理和刷新反而是个更大的问题，不如选择做数据库和架构优化，问题会更少。在百万量级下，数据库绝对是更好的数据存储解决方案，远比自己管理文件要更简单更稳定。如果我们从头观察数据库的发展史，就会发现数据库就是为了处理单靠读写一堆磁盘文件已经无法满足需求的场景而被发明出来的。

## 反爬措施

即便笔者做了那么多，还是不乏有一些爬虫愣头青在学习了 Swoole 和 Go 协程之后，对笔者的网站发动数千 QPS 的死亡冲锋，这个时候再怎么性能优化都是没用的，需要掏出倒数第二个工具：限流。

笔者做了三道限流关卡才最终顶住采集机器人的 DOS：

1. 针对单个 ip 做请求频率限制
2. 针对整个“123.123.123.123/24” ip 段做请求频率限制（很多爬虫采用同一段内的多个 ip 绕过限流）
3. 针对每个 UA 做请求频率限制

在这三板斧使出来以后，天下太平了，网站再也没有被突然发起的死亡冲锋搞挂过。

对了，既然限流是倒数第二个工具，肯定有人好奇最后一个工具是什么？那就是熔断，熔断属于系统鲁棒性工具，是善后用的，在本书的最后一章，我们将对此进行[简要的探讨](#)。

## 1.6 高并发实战：虚拟电商平台“静山”

静山是我国境内最矮的山，它仅仅高出地面 0.6 米，如图 1-4 所示，但却是一座货真价实的山，用来给我们的虚拟电商平台命名正合适。



图 1-4 静山实拍图

假设我们拥有一个电商平台，名为静山，它以 APP 的形式对外服务，没有 Web 端，官网域名为 [js.com](http://js.com)。静山平台的初始架构如图 1-5 所示。

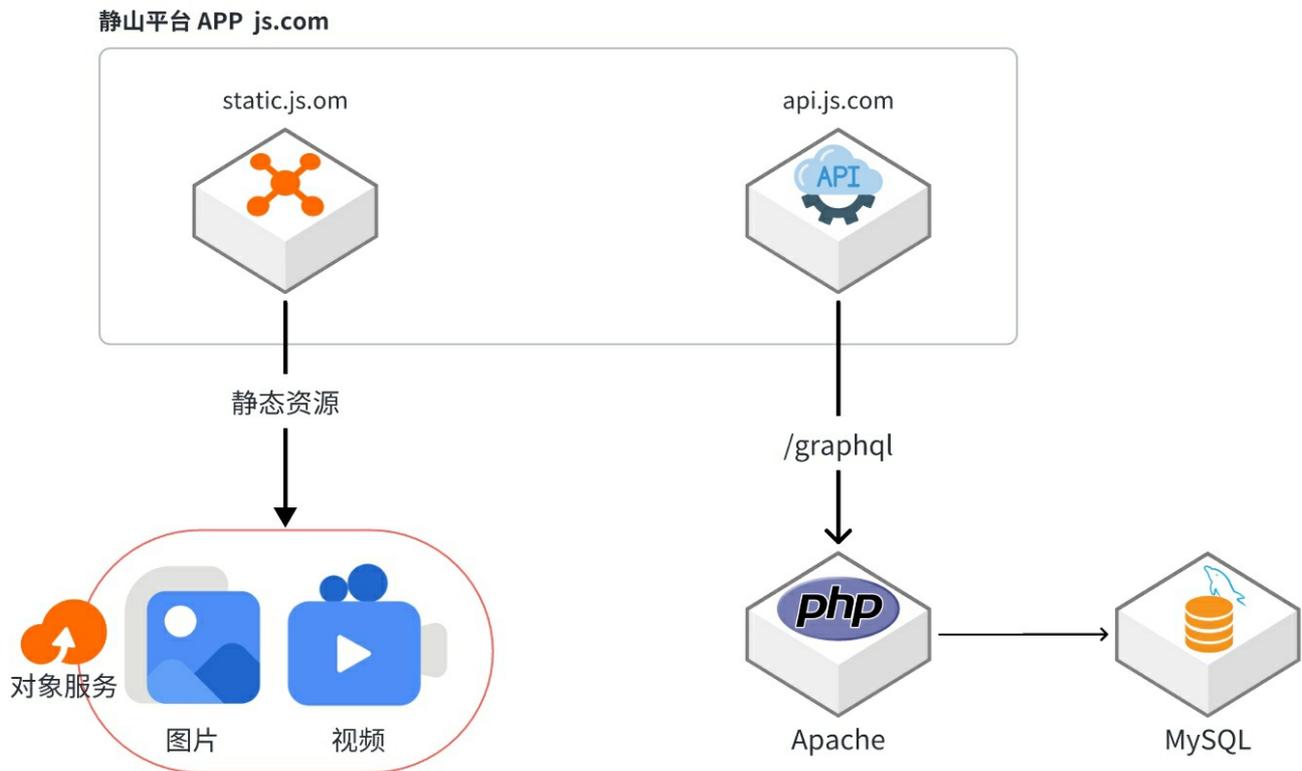


图 1-5 静山平台初始架构

## 静态资源云服务化

作为一个 2023 年的电商平台，静态资源必须全部云服务化，这已经是最新的业界技术标准了。为了实现云服务化，可以采取以下步骤：

1. 存储层：使用对象存储服务（如阿里云 OSS 和亚马逊 S3）来存储静态资源。
2. 前端接入层：通过 CDN（内容分发网络）域名直接对接对象存储服务，实现静态资源的无缝访问和负载均衡。CDN 将静态资源副本分布在全国各个节点，让每个地方的用户都能用最快速度访问静态资源。
3. 减轻后端压力：将静态资源独立出后端系统，减轻后端系统的负担，提高系统的性能和可伸缩性。后端系统可专注于业务逻辑处理，而无需关心静态资源的管理和访问。

## GraphQL 协议

根据笔者过去五年的实践，GraphQL 是最佳的客户端数据交互协议，它能在很大程度上避免后端工程师的低级错误。笔者十分推荐大家尝试这个新技术。

### GraphQL 是最佳的客户端数据交互协议

GraphQL 是一种用于客户端与服务器之间进行数据交互的协议，它是一种灵活且高效的数据传递方式。相比于传统的 RESTful API，GraphQL 具有更好的可扩展性和性能优势。使用 GraphQL，前端可以直接请求所需的数据，而无需经过繁琐的中间件或模板引擎处理。这种直接的数据获取方式可以减少网络传输的数据量，提高应用的性能。

## 避免后端工程师的低级错误

通过使用 GraphQL，后端工程师可以将更多的注意力集中在业务逻辑上，而不是处理底层的数据请求细节。GraphQL 提供了一种类型安全的方式来定义数据模型和查询，减少了潜在的数据一致性问题 and 错误。这使得后端工程师可以更加专注于编写高质量的代码，减少低级错误的发生。

## GraphQL 可以被看作是一种编程语言的“Java 化”

从软件工程的角度来看，GraphQL 可以被看作是一种编程语言的“Java 化”。Java 是一种广泛使用的编程语言，以其严谨的语法规则和强大的生态系统而闻名。类似地，GraphQL 通过增加繁复而严密的规范，使得编程语言可以更加健壮和可靠。这有助于提高软件的整体质量，并减少了由于低级错误而导致的问题。

## 可以只依靠架构师的个人能力来保证软件质量

通过引入 GraphQL，相当于给灵活的编程语言（如 PHP、Go）引入了规范的严格，软件的质量可以得到保证。通过使用这些规范严格的语言和框架，架构师可以更好地管理和控制整个软件开发过程，从而保证软件的质量水平。同时，这也使得非高级工程师能够更容易地写出可用的代码，提高了整个团队的开发效率。

## 数据库分开部署

如果预算有限，我们可以选择基于云主机自建数据库并自己进行数据备份。这种方式的优点是灵活性高，可以根据自己的需求进行定制和优化。但是缺点也很明显，那就是需要投入大量的时间和精力进行运维，而且数据备份和恢复的过程也可能会遇到各种问题。

如果预算充足，我们强烈建议大家直接使用云服务商提供的云数据库服务。云数据库是云服务商最核心的“软件服务”之一，也是最具用户价值和用户粘性的云产品。它的优点在于稳定性高，安全性好，而且可以随着业务的发展进行弹性扩展。此外，云数据库还可以提供丰富的数据分析和处理功能，对于我们未来的高并发系统建设非常重要。

需要注意的是，虽然云主机、对象存储、CDN 都很容易迁移，但云数据库几乎无法在云服务商之间迁移，在选择云数据库服务时，我们需要考虑到这一点。

## 云数据库的价值

云数据库绝不只是“不用自己做运维”那么简单，它拥有很多额外价值：

首先，云数据库拥有完善的数据备份和恢复机制。这不仅可以保证数据库的安全，还能提供自建数据库无法做到的“闪回到任意时刻”的能力。在实际的业务运行过程中，这种需求其实是相当常见的。例如，可能需要从过去的某个时间点恢复数据来进行数据分析或者修复错误。云数据库通常提供了多种备份和恢复策略，可以满足不同的需求。

其次，云数据库的性能更高。一旦你对数据库的需求超过了 1000QPS，自建数据库就很难支撑了。开源的 MySQL 在低配机器上性能很强，但是即便给它 32 核 64GB 内存，它的性能也不会好到哪里去。如果招聘研发团队自建数据库集群，那就太复杂了，小公司完全不值得，每年投入数百万成本养活运维和开发小团队，可能数据库集群还是会经常宕机。相比之下，云数据库可以提供高性能的数据库服务，可以轻松应对高并

发的访问请求。

最后，云数据库具有极佳的弹性。大部分互联网业务的流量都拥有显著的波动性，除了每天每周的业务高峰期之外，电商行业每年还有超级大促日。所以，数据库的弹性非常重要，按需付费可以节约大量金钱。云数据库可以根据实际的业务需求进行资源的动态调整，避免了资源的浪费。

## 1.7 现实世界中的高并发场景

---

本节将介绍现实世界中最常见的高并发场景，以及对应的解决方案。

### 高并发场景之一——电商大促

"大促"是现如今电商平台最常见的营销模式，双 11 和 618 可以说每一个中国人都有所耳闻。

在电商大促期间，由于用户访问流量和交易量在短时间内会快速增加，就导致了一系列高并发问题的出现：

#### 系统响应慢

用户访问量突然增大，系统规模无法短时间内扩大，导致页面加载缓慢，用户体验差。我们可以采取以下技术手段来提升系统的总容量：

- 静态资源缓存：将静态资源如图片、CSS 和 JavaScript 文件缓存到 CDN（内容分发网络）上，减少大促开始时服务器的带宽需求，加快页面加载速度。
- 弹性伸缩：通过自动或手动调整服务器的数量和配置，根据实时负载情况动态分配计算资源，以满足用户访问的需求。
- 负载均衡：将用户请求分发到多个服务器上，避免单个服务器过载，提高系统的吞吐能力。

#### 订单提交失败

订单量突然增加，系统可能无法及时处理订单，导致订单提交失败，甚至重复提交订单等问题。常用解决方案：

- 异步处理：将订单处理过程异步化，即时返回排队中的状态，避免单个网络请求的时间过长影响用户体验和服务器总容量。
- 消息队列：使用消息队列来存储和处理订单，确保订单的顺序和可靠处理。
- 幂等性设计：为订单处理接口设计幂等性，在前端加入订单唯一 UUID，防止重复提交订单和重复支付。

#### 库存超售

在电商大促中，多个用户可能在同一时刻购买同一款商品，极有可能导致超出库存数量的售卖行为，给经营带来损失。为了解决这个严重的问题，可以采取以下措施：

- 预售和抢购限制：提前设置商品的预售数量或限购数量，减少在明确无库存的情况下对库存扣减系统的额外压力。
- 使用队列进行库存扣减：使用消息队列来存储和处理订单，确保库存的顺序扣减。我们假设每次扣减库存需要 20ms 的处理时间，那单线程队列就可以完成每秒 50 单的需求，这个容量其实是不小的，我们假

设客单价为 1000 元，则单个队列处理器可以承担每秒 5 万元，每分钟 300 万元的压力，这个交易额其实已经很大了，全国能达到每分钟 300 万的电商平台应该也没有很多。

- 订单超卖记录：在极限情况下，为了保证系统的总容量能满足大促的要求，我们不会极端严格地限制超卖。如果真的超卖了，需要及时地反映在管理后台，让客服等运营人员介入处理。

## 支付超时

支付系统具有天然的单点性。在电商大促期间，由于用户购买行为集中，支付系统可能面临大量的并发支付请求，难以及时处理所有订单的支付请求，导致用户支付超时，有货卖不出去。这时我们可以做如下处理：

- 多节点部署：由于支付系统具有天然的单点性，单个线程经常需要等待，所以我们要尽量地多节点部署，以利用尽量多的 CPU 核心数来提升系统总容量。
- 异步支付处理：将支付处理过程异步化，把和用户操作相关的前序步骤和后续步骤都进行异步拆分，只在最核心的地方设置同步等待，减少对用户体验的阻滞。
- 合理设置支付超时时间：根据实际情况，合理设置支付超时时间，可以减少单个用户的支付容量以及系统总容量，极端场景下的订单可以在系统自动侦测到了之后进行手动改库修正。

举个例子：用户唤起了支付，但是停留在输入密码的页面一个小时，但是最后依然支付成功了，此时该订单很有可能已经被系统自动取消了。

## 恶意攻击

电商大促期间，由于系统流量集中，使得此时成为了恶意攻击的最佳时机。为了保护系统的安全和稳定，我们可以采取以下措施：

- 防火墙和入侵检测系统：部署硬件防火墙和入侵检测系统，它们可以抵挡住低级的基于网络协议漏洞的低端流量攻击。
- 限制访问频率：通过限制单个 IP 地址、单个 UUID、单个用户 ID 的访问频率和请求次数，防止恶意攻击者通过大量伪造请求或者是利用机器人抢购造成的系统瘫痪。
- 安全加密通信：使用 SSL/TLS 等安全协议对用户和服务器之间的通信进行加密，防止信息泄露和篡改。这一步在 2023 年应该已经全面普及了，如果你的系统还在用 HTTP 协议，需要赶紧升级。

## 高并发场景之一——社交网络

虽然电商大促的问题已被阿里等公司解决。然而，社交网络的高并发问题是目前仍待解决的难点之一。一个典型的案例是微博明星感情突发事件。

2017 年 10 月 8 日中午，微博工程师丁振凯正在举行婚礼，鹿晗突然宣布与关晓彤的恋情，导致微博客户端无法正常刷新、评论以及多个页面无法正常显示等问题。这次事件导致丁振凯中断婚礼，穿着西装戴着大红花坐在电脑前处理问题，成了鹿晗关晓彤事件的余波之一。事后统计，鹿晗公布恋情的微博，被转发了 33 万次，最终覆盖了 8.4 亿人次的微博用户。

在社交网络出现突发热点时，系统需要处理大量的实时数据更新、用户交互和内容传播动作，这些操作同时发生就给系统带来了巨大的压力。我们可以采取如下几个手段：

## 缓存优化

利用缓存技术来减轻数据库的压力，提高读取性能。将热门的用户信息、动态内容等存储在缓存中，加快数据的访问速度。在微博场景下，最大的挑战就是热点数据的及时识别：在某条微博、某个账号突然流量暴增时，能够在多短的时间内将该账号、该条微博缓存化，就成了决定系统是否会崩溃的最重要的因素。

## 异步处理

将耗时的操作异步化，使得系统能够更快地响应用户请求。例如，将评论、点赞等操作先放入消息队列中，然后由后台任务异步处理，避免在高峰期对主数据库和核心缓存形成额外压力。

## 数据库优化

对热点数据和非热点数据分而治之，减少热点数据所在表的长度，可以最大限度地利用数据库本身的性能特定和内存缓存，极大地提升数据库查询和更新的性能。

## 水平扩展

水平扩展是通过增加服务器数量和实施负载均衡来应对高并发访问的策略。通过将系统拆分为多个服务实例，可以同时处理更多请求，提高系统的吞吐量。系统能够以多快的速度进行水平扩展，直接影响普通用户的感受。只要扩容足够快，大多数用户几乎无法察觉到系统曾经宕机过。

## 资源预估和监控

在活动高峰期前，进行系统资源的预估和规划，确保系统能够承受预期的并发流量。同时，实时监控系统的运行状态，及时发现和解决潜在的性能问题。这一条只适用于节假日，明星们不按套路出牌，本条在明星场景下没什么用。

前几年，笔者刷微博的主要乐趣之一，就是见证 @蛋疼的axb 播报他在几年的时间跨度内，每次都被热点打败的故事。

## 高并发场景之一——金融交易

其实，目前狭义的金融系统本身已经没什么高并发问题了，股民数量虽然多，但由于交易的并发量并不大，因此在对比互联网流量时，金融系统的高并发问题相对较小。然而，若干年前的一项创新——“量化交易”，利用算法和网络速度进行套利，使金融交易再次面临了高并发的挑战。

笔者浅尝辄止地开发过量化交易的一个分支——高频交易软件，当时笔者选择了交易所同一个机房的云服务器，使得服务器到交易所服务器公网 ip 的延迟不超过 1ms。从理论上说，这样的设置应该能够轻松实现套利，然而实际上还是盈亏不定。因为笔者发现和自己竞争的就没有真人，全是量化交易机器人。在真金白银面前，机器人是非常疯狂的。

此外，由于金融交易涉及到用户的资产安全，系统的安全性变得至关重要，这使得高并发问题变得更加复杂。在技术上，金融交易高并发设计并没有什么额外的技术问题需要解决，反而是安全问题需要下大力气去解决。

## 高并发场景之一——网络游戏

游戏是一个和 Web、APP 都非常不同的高并发领域。

为什么都 2023 年了，游戏还是要分区？现在的 CPU、内存、磁盘、GPU 已经和 20 年前不可同日而语了，一台双路 EPYC™ 9654 的服务器都已经做到 384 核心 768 线程了，为什么游戏还是要分区呢？

因为游戏中的两个用户需要在极短的时间内针对对方的操作做出反应，不能别人砍了你一刀，你 1 秒之后才感觉到，大家都变成“高 Ping 战士”了，这游戏就没法玩了。

所以，一个游戏区域中，所有用户的数据都需要在内存中进行尽量快地实时计算，包括位置、状态、动作等，宏观上一个游戏分区甚至可以近似地等价于一个单线程应用：我们都知道，单线程应用是最难优化的，基本只能靠单核性能、三级缓存和内存通道数的进步获得。

### 音频、视频高并发

音视频高并发问题涉及到带宽、延迟和音视频处理这三个关键技术难点。

#### 带宽

带宽是指网络能够同时处理和传输的音视频流的数量。高清视频流占用大量的带宽资源，如果网络带宽不足，就会导致音视频传输过程中出现卡顿、延迟等问题，从而影响用户的观看体验。为了解决这个问题，我们可以采取两方面的措施：

1. 砸钱，提高带宽、部署 CDN 节点等。
2. 技术优化：通过 P2P 网络、缓冲策略、流式编码等方式，提升低带宽下用户的体验。

#### 延迟

音视频传输过程中，网络延迟可能导致帧丢失、卡顿等问题，影响用户的观看体验。我们可以采用数据压缩、流媒体缓存等技术手段来降低延迟。

数据压缩可以减小音视频数据的大小，减少传输时间，从而降低延迟。需要注意的是，相比于能压缩到 5% 的 JavaScript 文件的 Gzip 压缩，音视频本身就已经是一种压缩格式了，能够进一步压缩的比率很有限，很难低于 50%，而实际操作中我们一般选择采用新的视频格式来实现更高的压缩比率，而这依赖于客户端硬件、操作系统和浏览器的共同进步。

流媒体缓存可以预先缓存部分音视频数据，其本质是拿“过去的时间”换“现在的时间”。

#### 音视频处理

高清视频流对计算资源有巨大需求，特别是在编解码和图像处理方面，对处理高并发音视频流提出了高要求。为确保音视频处理速度和质量，需要高效利用计算资源。YouTube 和 Bilibili 等知名视频平台通过采用多项先进技术来解决高并发场景的挑战。

首先，这些平台利用先进的编码算法提高视频压缩效率。采用诸如 H.264、H.265 (HEVC) 等编码标准，

在不损失太多质量的情况下，大幅减小视频文件大小，从而降低传输带宽和存储空间需求。

其次，视频平台借助 GPU 和专用芯片等硬件加速技术来提高音视频处理速度，相对于传统的 CPU，在视频转码任务中效率更高。

此外，视频平台还利用云计算优势来解决高并发问题。在低谷期，云服务商通常提供低价计算资源，视频平台可利用此机会进行大规模转码工作。将音视频文件转码为多种格式和分辨率，以更好地适应不同设备和网络环境的播放需求。通过充分利用云计算的弹性和资源可扩展性，视频平台能够灵活应对高并发情况，并同时降低成本。

## 1.8 面试题

---

**No.01**：你遇到过哪些高并发系统？

下面几个案例供读者选择：

1. 电商平台：例如淘宝、京东等，这些平台需要处理用户浏览商品、添加购物车、下单、支付等大量请求，而且需要在同一时间处理大量的并发请求。
2. 社交网络：例如微信、微博等，这些平台需要处理用户发布动态、点赞、评论、聊天等大量请求，而且需要在同一时间处理大量的并发请求。
3. 游戏服务器：游戏需要实时处理玩家的输入指令，同时还需要处理大量的图形渲染和数据计算等任务，因此需要高并发的处理能力。
4. 云计算平台：例如AWS、阿里云等，这些平台需要为用户提供计算、存储、数据库等服务，需要处理大量的并发请求。

**No.02**：高并发系统的通用设计方法是什么？

高并发系统遇到性能瓶颈，一定是某个单点资源达到了极限，例如数据库、后端云服务器、网络带宽等，需要找到这个瓶颈资源，拆分它，提升整个系统的容量。具体来说，有如下几个设计方向：

1. 负载均衡：这是解决高并发问题的最基本和最直接的方式。负载均衡可以将请求分散到多个服务器上，从而使得每个服务器的负载保持在一个相对较低的水平，提高了系统的整体处理能力。
2. 缓存技术：缓存是提高系统性能的一种重要手段。通过将经常访问的数据存储在内存中，可以大大减少对数据库的访问，从而提高系统的响应速度。
3. 异步处理：异步处理是一种将耗时的操作转化为后台任务进行处理的方式，这样主线程就可以立即返回，不需要等待这些操作完成。这种方式可以提高系统的并发处理能力。
4. 消息队列：消息队列是一种用于处理大量并发请求的技术。通过将请求放入消息队列中，然后由专门的服务进程来处理这些请求，可以避免主线程被阻塞，从而提高系统的并发处理能力。
5. 数据库优化：对于数据库来说，优化SQL语句、合理设置索引、使用数据库集群等都可以提高数据库的处理能力，从而提高整个系统的性能。
6. 水平扩展：当系统的负载达到一定程度时，可以通过增加更多的服务器来扩展系统的能力。这通常需要对系统进行微服务化改造，每个服务都可以独立地部署和扩展。
7. 服务隔离：在高并发系统中，通常会有多个服务同时运行。为了保证服务的独立性和稳定性，需要将不同的服务隔离开来，避免一个服务的故障影响到其他服务。

### No.03 : 高并发系统的拆分顺序是什么样的？

高并发系统的拆分顺序应该从静态资源拆分开始，逐步进行到数据库和后端计算的机器分离，然后是设计负载均衡和分布式计算架构，接着是利用数据库集群和分布式数据库提升数据库性能，最后可以考虑基于地域进行数据库拆分。展开如下：

1. 静态资源拆分：首先，将系统中的静态资源（如图片、CSS 文件、JavaScript 文件等）进行拆分。这样可以避免在高并发情况下，静态资源的加载成为系统性能瓶颈。可以将静态资源放在专门的 CDN 上，提高访问速度。
2. 数据库和后端计算的机器分离：将负责处理业务逻辑的服务器与负责存储数据的数据库服务器分离。这样可以降低数据库的压力，提高数据处理速度。同时，可以将不同的业务逻辑部署在不同的服务器上，提高系统的可扩展性和容错能力。
3. 设计负载均衡和分布式计算架构：通过负载均衡技术，将用户请求分发到多台服务器上进行处理。这样可以充分利用多台服务器的资源，提高系统的处理能力。同时，可以采用分布式计算架构，利用多台服务器同时进行服务，进一步提高系统的并发处理能力。
4. 数据库集群和分布式数据库：为了提升数据库的性能，可以采用数据库集群和分布式数据库技术。数据库集群是将多个数据库实例分布在不同的服务器上，通过数据同步和故障转移机制，保证数据库的高可用性。分布式数据库是将数据分散在多个节点上，每个节点只负责部分数据的存储和查询，从而提高整个系统的读写性能。
5. 基于地域进行数据库拆分：如果已经是一个全国性系统，可以考虑基于地域进行数据库拆分。将全国分成几个区域，每个区域的机房只为地理位置在本区的用户提供热数据。这样可以避免跨区域的数据传输，降低网络延迟，提高系统响应速度。对于外区的用户，可以将请求转发回原来的大区，获得终极的系统容量提升。

### No.04 : 静态资源如何加速？

为了加速静态资源的加载，可以采取以下几种方法：

1. 使用高性能的 Web 服务器：如 Nginx，适合处理大量并发请求，提高应用性能。
2. 利用 CDN 服务：将网站内容复制到全球多个服务器，用户从最近的服务器获取信息，减少延迟，提高加载速度。CDN 特别适用于静态资源的分发，可以有效地加速图片、CSS、JavaScript 等静态资源的加载。
3. 进行格式转换、压缩和 HTTP/2 header 重用：可以将 PNG、BMP 图片转换为 WebP 或 JPEG 格式，以减小文件体积。此外，利用压缩算法如 gzip 对文件进行压缩，可以进一步减小文件大小，提高传输效率。另外，HTTP/2 协议支持 header 重用功能，可以减少网络传输时间。

## 第二部分 计算资源高并发

第 2 章 基础设施并发：虚拟机与 Kubernetes (K8s)

第 3 章 突破编程语言的性能瓶颈

### 第 2 章 基础设施并发：虚拟机与 Kubernetes (K8s)

基础设施并发，全称为“基础设施并发管理”，指的是在计算机系统中同时有多个应用程序或任务在共享和访问计算机系统的各种资源（比如 CPU、内存、网络等）的情况下，保证这些资源的正确使用和公平分配的技术能力。笔者想用另一个技术概念来帮助大家理解基础设施并发：

基础设施即服务：IAAS (Infrastructure as a Service) 是一种云计算服务模型，它提供了基础设施资源，如计算、存储和网络等。用户可以根据需要灵活地使用这些资源，而无需关心底层的硬件和软件实现细节。IAAS 通常由云服务提供商提供，用户只需通过互联网访问这些服务即可。

通俗来说，精确地管控每个进程所能使用的 CPU、内存、网络、磁盘等物理计算资源就是基础设施并发管理的目标，它可以让分布式系统更加均衡、更加稳定，减少资源浪费，提升系统的总容量。

## 2.1 概述

---

本节我们以静山平台为例讲述如何在基础设施并发层面增加系统总容量。

静山平台流量持续增大，我们该怎样扩容呢？

假设场景：经过了一段时间的运营，静山电商平台的用户持续增多，在高峰期，同时在线用户来到了 1000 人，每分钟新增订单数 20 个。假设每用户每秒发送 1 个网络请求，则此时后端压力来到了 1000 QPS。

在静山平台中，静态资源完全由云服务商提供，而且由于 CDN 传输的图片之间没有关系，属于纯离散数据，所以理论上一个 CDN 域名就可以拥有无穷大的带宽，这部分的压力我们完全不用担心。而动态 API 请求部分，我们拥有一台跑后端代码的云主机和一台云数据库，这台云主机的单机性能上限为 200 QPS。

### 为何单机只能处理 200 QPS

在一个传统的“Apache + mod\_php”的后端服务器单机架构中，QPS（每秒查询率）很难达到非常高的水平。这是因为 Apache 是一个古老的 Web 服务器软件，而 PHP 也是一种阻塞语言。Apache 无法处理超过 5000 的 TCP 连接数，不论使用多少个核心都一样，一旦超过这个限制，客户端就需要等待其它客户端的 TCP 连接数释放后才能和服务器进行通信，在那之前只能等待。

而当 PHP 在执行比较费时的网络和磁盘 I/O 操作时，它会停下来等待操作完成，此时不消耗 CPU 资源，但内存会被持续占用，而且最重要的 TCP 连接数资源会被持续占用。

因此，在使用 Apache 直接对外提供 HTTP/HTTPS 服务的情况下，单机的极限 QPS 大约在 200 左右。

### 何为基础设施并发

像 Apache 一样，由于计算机体系结构的限制，很多软件无法利用多个 CPU 核心，像 Apache、Redis 甚至是 MySQL 都是无法很好地利用多核心硬件的：最新的 AMD EPYC™ 9654 服务器 CPU 已经来到了 192 核心 384 线程，最常见的双路服务器拥有 768 个超线程，这个核心数恐怕 Go 语言都得败下阵来。所以大部分时候，我们都需要将一台数百个核心的物理服务器拆分成多台虚拟机来使用，以提高计算资源利用率。

对静山平台来说，在面临 1000 QPS 的场景时，我们就需要引入负载均衡器了：一台机器跑 Nginx 充当负载均衡器在前面接收客户端 APP 的 HTTPS 请求，然后转身就把这些请求平均分给 5 台真正的 Apache 后端服务器来进行具体的 PHP 代码执行，之后等 Apache 服务器返回结果之后，再将返回值转发给客户端 APP。

使用负载均衡器将请求分发到多台机器上的处理方式就是基础设施并发，其对应的架构图如图 2-1 所示。

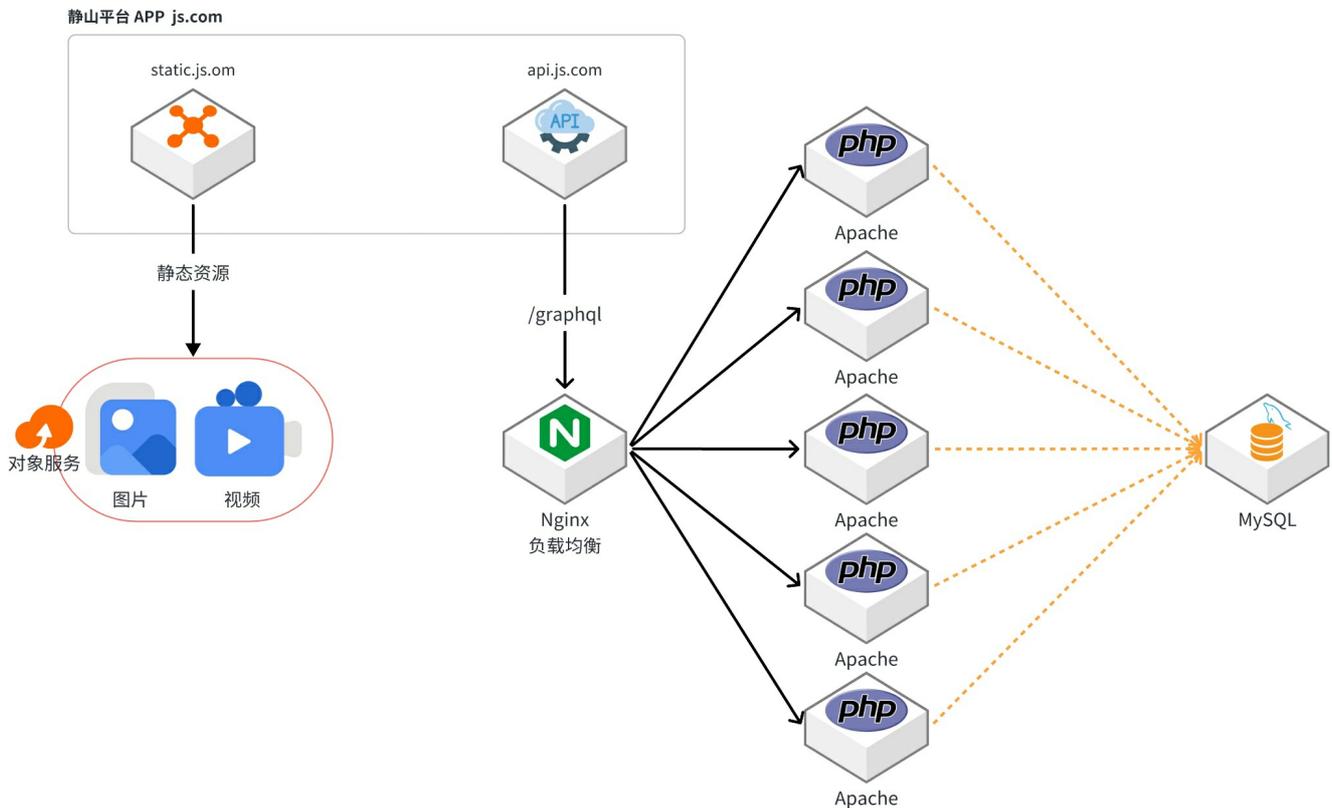


图 2-1 含有负载均衡器的基础设施并发架构图

## 2.2 服务器虚拟化

服务器虚拟化是第一代基础设施并发技术：既然由于软件架构的限制导致单机性能有上限，那我们就把一台多核服务器拆成多台来使用。

服务器虚拟化技术可以在一台物理机上同时运行多台虚拟机，而每一台虚拟机都拥有完整的操作系统和硬件资源，包括 CPU、内存、磁盘、网卡等，虚拟机之间是完全隔离的，它们不知道邻居是谁，而在绝大多数情况下也不需要知道，因为硬件绑定技术已经十分成熟了：邻居哪怕在挖矿，也不会影响你的性能，因为它们的硬件资源已经在底层物理硬件层面被隔离开了。

你是否曾经思考过，在一台裸金属服务器的操作系统内直接运行 Apache、MySQL、Redis、Elasticsearch，与开四台虚拟机分别运行它们之间有什么区别呢？系统设计哲学不是“如无必要勿增实体”吗？虚拟机技术到底有什么价值来让我们新增四个实体呢？

### 资源隔离

首先，就要说到我们第一章中提过的运维哲学了：

运维的核心价值不在于资源的扩充，而在于资源的隔离。

在后端技术领域，有许多成熟的开源软件可供选择，每个都有独特的发展路径，百花齐放、百家争鸣。如果将所有服务都部署在裸金属服务器上，理论上使用内存管道通信的效率会比虚拟机之间通过虚拟以太网通

信更高。然而，在这种情况下实现资源隔离却是一项挑战。

目前，拥有自我资源限制能力的软件并不多见。大部分软件都是“有多少资源就占用多少资源”，即根据请求压力任意获取计算资源，这容易导致其他软件的生存空间被挤占，形成木桶效应，进而导致系统整体性能下降。

虚拟化技术通过将虚拟机与物理核心进行强绑定，可以无限接近将一台 64 核服务器拆分为 32 台 2 核服务器的目标。此外，内存也可以进行完全预分配，从而最大程度地减少虚拟机之间的相互干扰。如图 2-2 所示就是虚拟化技术的示意图。

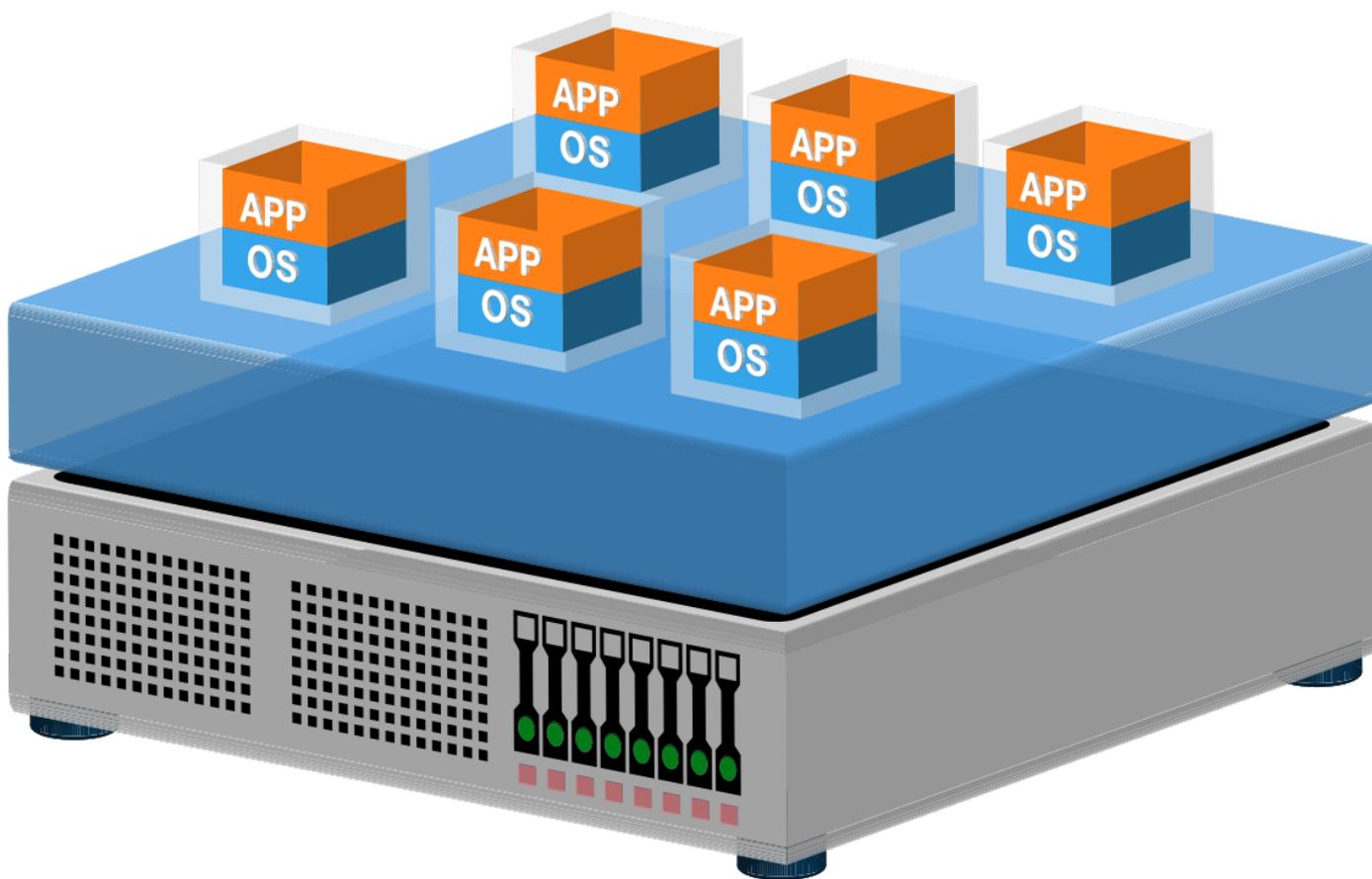


图 2-2 虚拟化技术示意图

### 提高运维效率

在传统的物理服务器时代，操作系统需要运维人员使用光盘安装，软件的安装也需要手动一台一台地操作。当出现软件或硬件故障时，运维人员需要逐一判断和解决，效率非常低下。然而，随着虚拟化技术的发展，物理服务器只需要安装一次操作系统，除非遇到安全性问题，否则几乎不需要更新。而作为业务承载者的虚拟机却可以随着时间的推移不断更新版本，跟上时代的步伐。

虚拟机技术显著降低了对每个重要服务进行资源监测的复杂度，极大地提升了运维效率。目前市场上成熟的监控软件都是读取操作系统指标进行监控的，如果想要对一台机器上的多个进程分别进行监控，技术难度要

大很多。

此外，虚拟机技术还具备部分可编程的特性，这就催生了第一代“自动运维”技术的出现。随着时间的推移，自动运维技术逐渐发展壮大，如今已经创造了一个巨大的市场——以亚马逊云计算为首的云计算市场。

众多中小企业通过购买这些价格低廉的云服务，得以使用过去只有大公司庞大的平台架构部门才能构建起的高级基础服务。这不仅帮助中小企业缩减了成本，还为业务的发展提供了更多的可能性。

### 强大的兼容性和可调试能力

虚拟化技术，如同一条标准化的生产线，为我们提供了一套逻辑上的“不可变基础设施”。这就像是一个魔法，使得当我们的系统规模增大，或者我们采购了新的物理服务器之后，我们仍然可以非常方便地直接运行旧的虚拟机镜像。这种灵活性让我们避免了“新硬件只能安装新版操作系统而无法支持旧版本软件”的尴尬局面。

想象一下，如果有一台虚拟机运行的时间非常久，那它大可以随时迁移到最新的硬件平台之上，不必再费尽心思、小心翼翼地维护一台开机了十年的老旧服务器。这样，我们就避免了因为老旧硬件故障而造成业务的重大损失。

此外，虚拟化技术还赋予了我们无比强大的资源管理能力。我们可以在任何时间对虚拟机系统的关键资源做出任何调整，可以设计各种高级网络架构，可以随时重启虚拟机，不用担心“物理服务器重启之后开机失败”这种处理起来非常麻烦的问题。如果我们对操作系统做出了修改但启动不了，只需要恢复上一版镜像就可以了。

最后，虚拟化技术对操作系统开发的贡献也很大，大幅提升了操作系统开发中涉及到硬件的调试效率。这使得我们发现问题、获取 Dump 信息、测试问题是否已解决的速度都提升了一大截。

### 高可用架构

在裸金属时代，如果我们需要搭建一个高可用集群，我们至少需要 2 台路由器、2 台后端服务器，2 台数据库服务器，架构如图 2-3 所示。

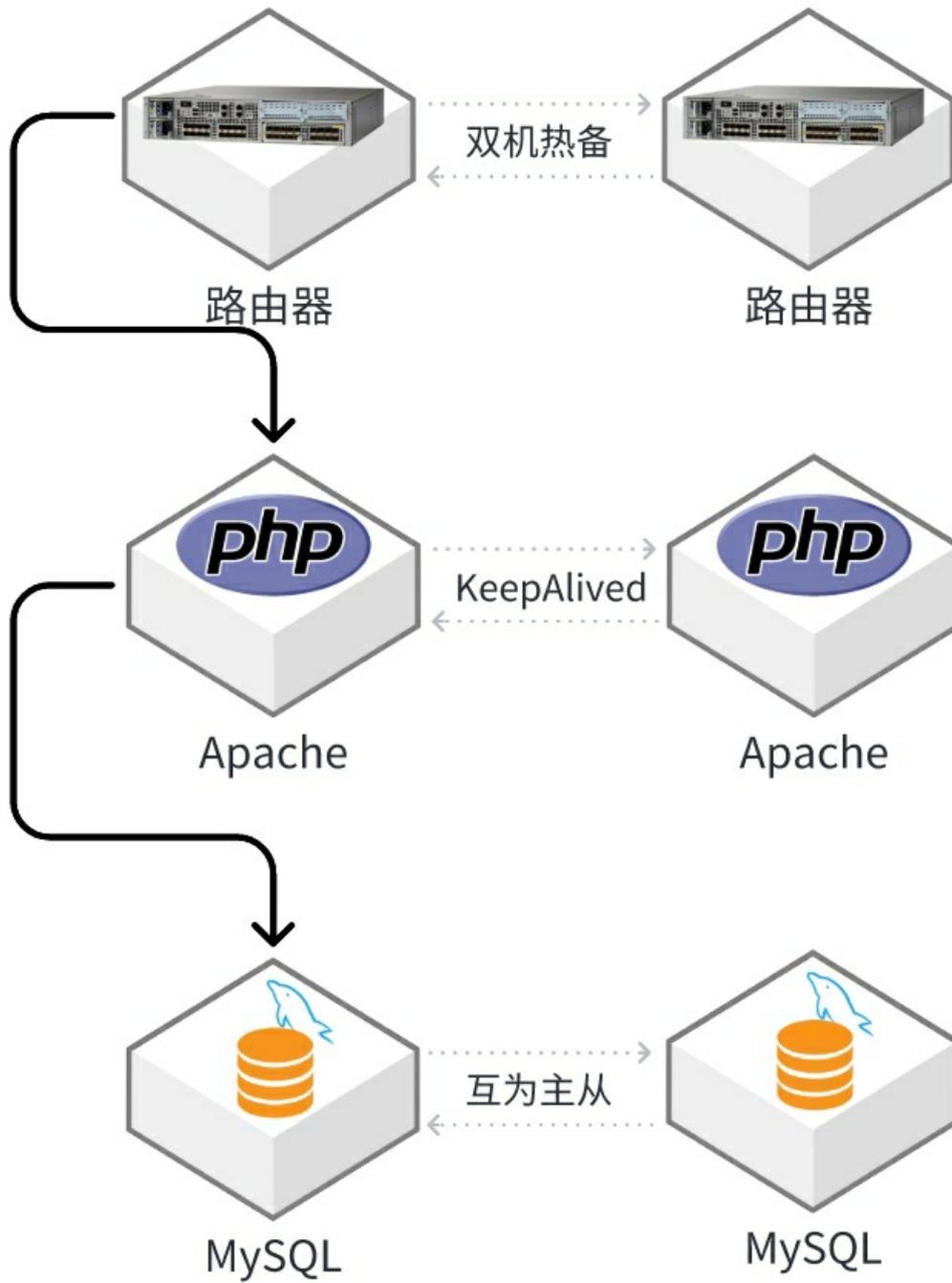


图 2-3 基础高可用集群架构

而使用最新的超融合技术，只需要 2 台物理机便可以把这件事情全干了：因为路由器、后端服务器、数据库服务器全部都可以虚拟化，它们三个完全可以运行在同一台物理服务器之上。两台超融合物理服务器搭建的高可用集群架构如图 2-4 所示。

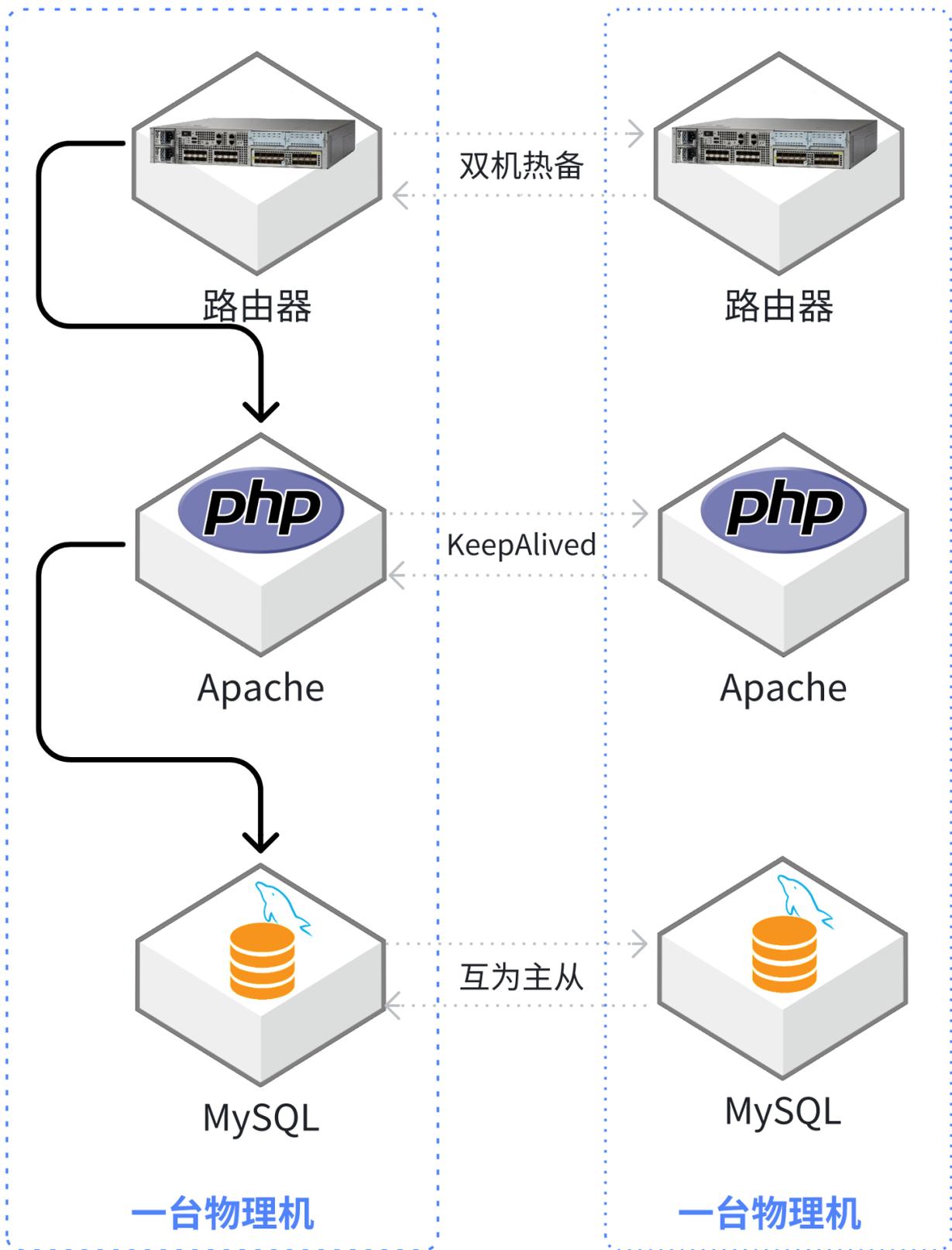


图 2-4 两台物理服务器高可用集群架构

时至今日，VMWare 在服务器超融合领域仍然占据着统治地位。通过将 vSphere 软件安装在三台物理服务器上，形成一个小集群，就能够提供高可用架构的全套基础设施。下面我们将简要介绍一些主要的高可用基础设施。

1. 统一应用网关：就是图 2-4 中的路由器实现的功能，对流量进行转发，单台服务器宕机不影响对外服务。
2. 共享存储：在小集群之内形成一个统一的磁盘存储，单台服务器宕机不影响磁盘的正常使用和数据安全。
3. 虚拟机热漂移：虚拟机无需关机地在物理服务器之间迁移，实现真正的“无缝迁移”，该技术通过共享存储和内存实时拷贝来实现。
4. 自动故障转移：基于上述三个主要的组件和功能，在某一物理机宕机之后，整个集群的对外服务完全不受影响，从网络到计算到数据库都可以在几秒内恢复对外服务。

如果只用一台物理服务器直接安装 Linux，然后把所有应用软件都安装在物理机上，很显然上述这些高可用性特性都做不到：一旦这台物理机的某个部件发生了故障导致宕机的话，全部服务都会挂掉。

## 利用集群架构提升总体性能

由于许多软件都具有类似 Apache 的“单机性能极限”，因此采用虚拟化技术将单个机器拆分成多个机器，可以直接实现性能的提升。假设我们将 Apache、Redis、MySQL 和 Elasticsearch 都安装在裸金属操作系统上，它们可能会异口同声地表示：即使有这么多核心，我也用不上啊。

在后续的第 4 章中，我们将对 Apache 进行非常详细的讨论。现在，让我们先来探讨其他几种技术的“单机性能极限”以及相应的虚拟机集群方案。

### Redis 的单机性能极限

Redis 作为单线程内存数据库（仅限 2020 年 6.0 版本之前），即使只拥有两个处理器核心，其每秒查询数（QPS）也能达到十万次。即便你给它分配了 32 核心处理器，最大性能还是只有十万次查询/秒，只会浪费资源。

在前几天 Redis 回应新兴内存数据库 Dragonfly 挑战的文章中，Redis 维护团队建议大家搭建“单机集群”<sup>④</sup>来实现对机器上所有 CPU 的完全利用，Redis 团队的内心 OS 一定是：宝宝心里苦但宝宝不说。

如果你具备控制物理机的权限，那么基于虚拟化技术搭建多虚拟机上的 Redis 集群是一种比单机集群更加稳定和高性能的解决方案。这是因为虚拟化平台对虚拟环境中操作系统的管理能力要远远高于 Systemd 对进程的控制能力。

### MySQL 的单机性能极限

在当今 NVMe PCIe 5.0 的时代，单个闪存磁盘的读写速度已经达到了惊人的 12.8GB/S。MySQL 的单机 CPU 使用强度也得到了大幅提升。然而，如果使用最新的 AMD EPYC™ 9654 192 核 CPU 搭建的双路服务器（包含两颗 CPU）来运行单机 MySQL，情况会如何呢？显然，单节点 MySQL 无论如何也无法充分利用 384 个物理核心和 768 个线程，其性能极限也会停留在几万 QPS。

实际上，开源标准版 MySQL 的性能瓶颈在于内存容量和磁盘速度，对 CPU 资源的需求并不是很高。在两台虚拟机拥有独立固态硬盘的情况下，将 16 核虚拟机拆分成两台 8 核虚拟机并构建成一个半同步一主一从的 MySQL 集群，可以轻松地将数据库的读性能提升一倍，只需稍微牺牲一点点的写性能。

关于 Web 系统的瓶颈在数据库、数据库的瓶颈在存储的话题，我们将在后面的第四部分专门展开讨论，并使用多章内容进行深入探讨。

## Elasticsearch 的单机性能极限

Elasticsearch 作为一个 Java 应用，显然拥有着明确的单机性能极限：线程切换需要读写内存，性能上限就在那里。所以 Elasticsearch 采用了自选集集群架构来实现水平扩展：官方建议单节点不要超过 32GB 内存。

如果你的物理服务器上面 CPU 和内存资源很多，就应该基于虚拟机技术做多节点 Elasticsearch 集群，Elasticsearch 集群的扩展能力是非常不错的，基本上可以实现线性的集群性能提升。作为一个搜索引擎，大家竟然都拿他当数据库用，从这里就足以看出它优秀的水平扩展能力了。

在后面第 11 章中我们讨论到缓存与队列时，我们还会重点学习一下 Elasticsearch 优秀的内存缓存设计。

## 2.3 常见虚拟化/仿真技术的软件兼容性

“容器编排”无疑是过去十年软件部署领域最火的技术方向，但为了搞懂它到底是什么，我们需要先了解一下软件兼容性和常见的虚拟化/仿真技术。

### 软件的三层兼容性（可移植性）

可移植性：一个可以在某台机器上运行的软件，复制到另一台机器上之后，是否还能运行。

在软件开发中，无论是使用二进制分发（例如 Windows 版微信）还是源代码分发（例如 PHP 网站），都存在三个层次的兼容性（可移植性）。

1. 指令集兼容性（ISA Compatibility）：主要针对二进制软件，指的是针对 x86 编译出的二进制软件无法运行在 ARM 平台上。
2. 二进制库兼容性（Application Binary Interface Compatibility）：指的是 Windows 版软件依赖的 `.Net Framework` 在 Linux 系统上没有，所以 Windows 版微信无法直接在 Linux 上运行。此外，在移动开发领域，不同版本的 Linux 发行版之间的 ABI 兼容性并不好，例如 Apache 这个软件在红帽系发行版上叫 `httpd`，而在 Ubuntu 系发行版上叫 `apache2`。
3. 环境兼容（Environmental Compatibility）：包括配置文件、环境变量、数据库配置、文件系统、用户权限等。在实际运维过程中，大部分长期运营的系统所遇到的兼容性问题都是环境兼容问题，即使是看似最简单的兼容性问题也会让运维人员精疲力尽。

说完了软件兼容性，下面我们还需要结合这三层兼容性了解一下常见的虚拟化/仿真技术，以及它们能解决哪些层面的兼容性。

### 常见的虚拟化/仿真技术

学习过计算机科学的基础课程《计算机组成原理》的小伙伴都知道，计算机从硬件到操作系统的设计其实到处都充满了“模拟”的设计方法，从最底层的使用晶体管模拟逻辑门电路，到最上层的使用 CPU 和操作系

统配合完成的“硬件中断”来模拟“事件驱动”功能，计算机的出现，就是一层又一层虚拟化和仿真思想的应用结果。

针对和容器相关的虚拟化，笔者总结出了如下几种虚拟化/仿真的方法。

## 硬件虚拟化

大家日常使用的云计算平台提供的云主机就属于硬件虚拟化的范畴，对应的在物理机上直接部署的虚拟化宿主操作系统有 VMware 的 ESXi 和微软的 Hyper-V。硬件虚拟化是最常见的虚拟化，你在你的电脑上安装的“虚拟机”软件，在 Windows 或者 macOS 上创建 Linux 虚拟机，就属于硬件虚拟化的范畴。

硬件虚拟化可以给每个虚拟机提供虚拟出来的 CPU、内存、磁盘、网卡甚至是显卡。跟大家想象的可能不太一样，今天的硬件虚拟化其实绝大部分已经由硬件直接提供了，换句话说，今天的 CPU、主板芯片组甚至是硬盘控制器，已经全部内置了用于支撑高性能低损耗虚拟化的集成电路面积，虚拟机早就不是纯软件虚拟机了。Intel 的 VT-d 技术和 AMD 的 AMD-V 技术就是 CPU、内存等进程相关的硬件虚拟化技术。

此外，SR-IOV (Single Root I/O Virtualization) 作为一种针对 PCIe 设备的硬件虚拟化技术，正在大规模普及。其核心优势在于能够突破宿主机操作系统的软件性能瓶颈（例如在《性能之殇（四）-- Unix 进程模型的局限》[一文](#)中所述的上下文切换性能极限）。通过让虚拟机直接在硬件虚拟化层面调用底层硬件，SR-IOV 能够显著提升虚拟机的 I/O 速度，即网络传输速度和磁盘读写速度。

容器和容器编排技术作为一种操作系统内的资源管理方法，理论上和硬件虚拟化没有交集。不过，由于今天的服务器 CPU 的核心数实在是太多了，如今绝大多数的容器其实并不是运行在裸金属服务器上的，而是运行在经过了硬件虚拟化的虚拟机中。

## 指令集仿真

指令集仿真技术通过软件将虚拟机指令集转换为宿主机指令集，提供了无比强大的兼容性，甚至可以在浏览器中运行 Windows 98，如图 2-5 所示。大名鼎鼎的 QEMU 即属于指令集仿真。

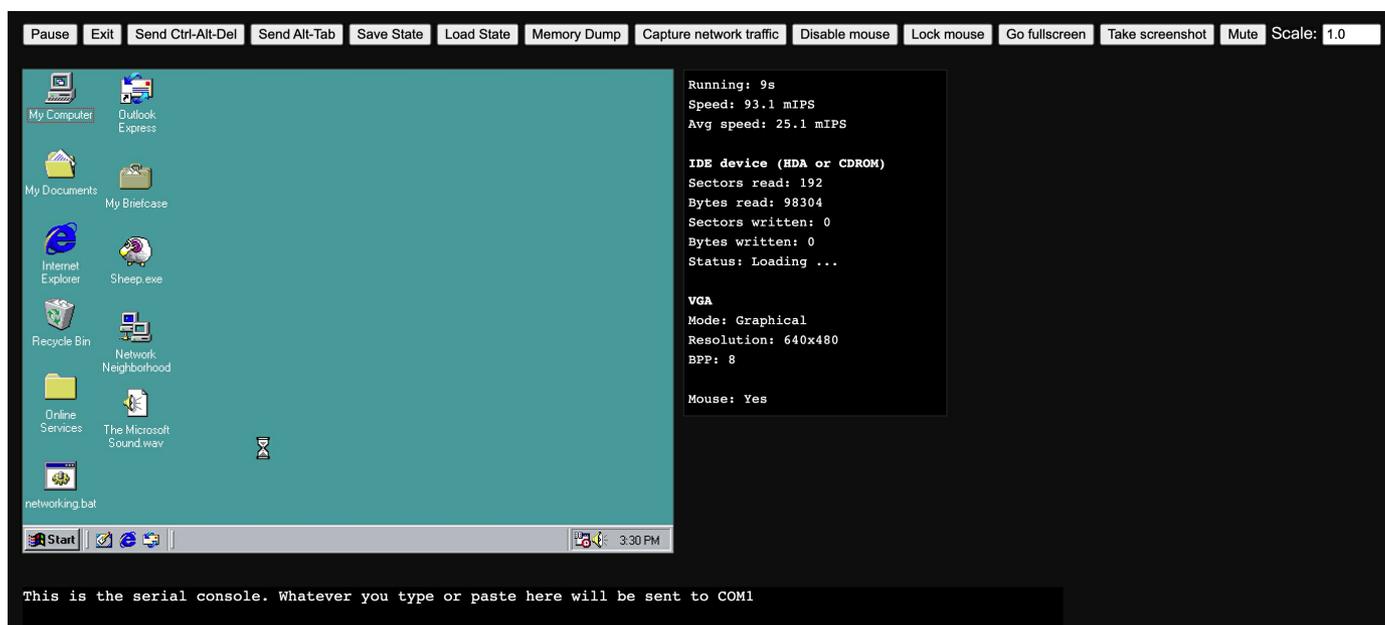


图 2-5 在浏览器中运行 Windows 98

无论是硬件虚拟化，还是指令集仿真，其运行的虚拟机都是一个完整的操作系统，拥有完全独立的系统文件、内存空间、基础库和运行环境，可以实现整个操作系统的整体迁移，理论上讲甚至可以从 x86 宿主机迁移到 ARM 宿主机。

容器和容器编排技术并未用到指令集仿真，它们是一种更高级的资源管理方法，适用于在机器数量很多的环境中进行操作系统内部资源的统一平台化管理。

### 运行库虚拟化

折腾过 Linux 当做软件开发桌面操作系统的读者一定都用过 Wine，它可以让你在 Linux 下运行 Windows 微信，这就是一种运行库虚拟化技术，它的本质是用软件的形式模拟 Windows 操作系统以及它上面的 .Net Framework 之类的二进制库，这个技术看起来非常的具有“邪典”气质，当然，其兼容性也是最差的，往往 QQ 发了一个新版本，Wine 就得赶紧更新，不然就会各种闪退好不热闹。

对了，微软在 Windows 10 中引入的 Windows Subsystem for Linux (WSL) 也是一种运行库虚拟化技术，比虚拟机技术更快，资源浪费更少，有微软的工程师提供支持，兼容性也更好，而且 Linux 自身的 ABI 稳定性一直不错。当下，从后端开发到 AI 社区，WSL 出镜率都很高，微软这一步棋目前看来是走对了。

容器和容器编排就运行在这一层，不过和前面跨操作系统的情况有一个不小的区别：与跨操作系统的情况不同，容器包含的是应用程序及其依赖项的完整运行时环境。容器将应用程序与其所在主机系统隔离开来，提供了一种轻量级的、可移植的方式来部署和管理应用程序。容器编排则负责管理和协调多个容器的运行，以实现自动化部署、扩展和管理等功能。

在容器自己看来，自己就是运行在裸操作系统上的一个普通的进程，可以获取到完全真实的硬件情况。只是它对于系统库的调用会被容器运行时给劫持，容器之间无法相互感受到对方的存在。

### 虚拟机编程语言

一说到虚拟机编程语言，Java 一定是大家第一个想到的语言。早年间，Java 整天号称自己“一次编译，到处运行”，说自己可以统一从服务器到桌面再到嵌入式设备的全部软件运行场景，实际结果大家也看到了：也许 Java 可以，但它在大多数平台上都不是最优解，最终导致它还是只能运行在最重的服务器和大型桌面软件的背后。

Java 的 JVM 提供了一套标准：将 Java 代码编译成字节码，由字节码解释器对其进行解释运行。这些字节码再调用 JVM 专为各种平台各种架构所开发的高度兼容的 API，实现了相当高的“可移植性”，至少在技术上，它做到了。但是，这种兼容性并不属于前面三层的任意一层，因为 Java 技术本身要求你用 Java 语言来编写软件，在逻辑上，这并不是兼容，而是一种 DSL（领域特定语言），只是这种 DSL 能够跨平台运行罢了。

除了 JVM 字节码虚拟机，.Net 的 CLR 也是一种类似的技术，不过 CLR 运行的是二进制程序（Native Code）。

## 文本解释型编程语言

作为一名在 2023 年依然还在某些场景中使用 PHP 的开发者，笔者觉得很有必要向大家展示一下文本解释型编程语言的技术优势。任何一个流行技术，肯定是解决了某一方面甚至是某几方面的大痛点，才得以流行起来的。

笔者知道很多 Java 开发者对 PHP 不屑一顾，但笔者认为 Java 程序员尤其需要“开眼看世界”。笔者不止一次地遇到拿 Java 概念来类比操作系统底层逻辑的程序员了，如果只懂 Java 和 JVM，虽然你也能写出能用的软件，也能赚到钱，但是计算机运行的不是 JVM 操作系统，Java 技术不是计算机的一切。作为软件开发者，我们还是要不断地“向更底层看去”，不断加深对计算机的理解，成为更优秀的自己。

文本解释型编程语言，最典型的的就是 PHP，其本质也是一种可以跨平台运行的软件技术，它采用文本 + 解释器的方式运行，利用专门针对多个操作系统开发的二进制解释器来边解释边运行。PHP 代码文本可以看做一种更加粗糙的字节码，由于 PHP 一直被用在小规模 Web 系统中，所以长期以来，性能低下、API 混乱、代码质量不高就成了 PHP 的知名缺点。但是 PHP 最大的优点其实是他的运行方式：

1. 每个 HTTP 请求都会开启一个 PHP 进程，页面输出完进程就退出，这就在很大程度上让 PHP 完全不用管内存泄露问题，即便质量低劣的应用代码也可以以一种“看起来 OK”的方式长期稳定地运行。
2. 动态载入文本文件使得 PHP 成为了一种“半微服务”架构，如果某个菜鸟提交了一个有语法错误的文件，现有的页面如果对它没有依赖，那现有页面就不会崩溃：每个 URL 都可以看做一个独立的微服务。

相对的，Java 技术虽然常驻内存，自己在内存里就可以解决很多 PHP 必须依赖 Redis 和 MySQL 才能实现的需求，性能更强，但是它更容易因为一段代码的 Bug 就整体崩溃，更容易出现内存溢出（OOM）问题，运维和 Debug 的难度更大。

## 2.4 容器

接下来，我们将深入探讨容器技术的发展历程，并揭示其成功的关键原因。

### 容器技术发展史

容器的发展不是一蹴而就的。作为一种“软件分发与运行标准”，容器在功能上像 Yum 和 Systemd 的结合，这就决定了容器技术只能缓慢地进入到 kernel 和 Linux 生态系统中，随着时间的流淌不断融入，等所有 Linux 发行版、应用软件和管理软件都成熟了以后，才能真正将这个新标准推广给所有 Linux 用户。

容器的发展按照时间顺序经历了文件系统隔离、进程访问隔离、系统资源限制、应用封装四步，下面我们一一展开。

### 文件系统隔离：Chroot

1979 年，美国计算机科学家 Bill Joy 提出并实现了 Chroot（Change Root）命令，内置到了第七版 Unix 系统中，它可以改变某个进程的文件系统根目录：限制这个进程以及所有子进程所能访问的文件系统，提升 Unix 系统安全性。后来人们发现这个工具可以很好地应用在软件的开发和测试过程中：它可以隔离多个软件之间的冲突和干扰，为它们提供相对独立的运行环境。

但是，作为一个安全工具，Chroot 还是不够安全，因为除了文件系统，软件还是可以通过大量的系统调用突破限制，这并不是完美的文件系统隔离方法。为了解决这个问题，2000 年 Linux Kernel 2.3.41 引入了 pivot\_root 技术，它通过直接切换“根文件系统”的方式提高了安全性，今天常见的容器技术也都是优先使用 pivot\_root 来做文件系统隔离的。当然，我们需要知道的是，即便到了 2023 年，文件系统的隔离依然不是完美的，如果你在 Docker 内运行了一个未知应用，你的系统还是有很大的被黑的风险。

## 进程访问隔离：Namespaces

命名空间，后端开发者肯定都不陌生，他就是 Java 和 Go 中的包名：给类安排一个前缀，避免同名的类搞混。但是 Linux 的 Namespaces 却不仅仅是为了防止同名混淆。自 2002 的 Kernel 2.4.19 开始，Namespaces 就被引入了 Linux 内核，而且它在刚刚发布的时候和 Chroot 干的事情是一样的：隔离文件系统。由于 Unix 一切皆文件的思想，可以把初代 Namespaces 看做一种更高级更完善的 Chroot。

后来，随着用户量的增加，大家迫切需要 Namespaces 把其它资源也隔离一下，到了 Kernel 5.6 时代，Namespaces 已经具备了文件系统、主机名、NIS 域名（一种 Unix/Linux 的域控制系统）、进程间通信管道（IPC）、进程编号（PID）、网络、用户和用户组、Cgroup、系统时间一共八种隔离能力。

Namespaces 是容器技术的坚实基础：它不是虚拟机，它没有搞一套虚拟的操作系统接口，而是忠实地把宿主机的信息暴露给了某个 namespace 内的进程，但是不同的 namespace 内的进程之间是相互看不见的。它们都以为是自己独占操作系统的全部资源。

下面问题来了，软件不都是“有多少资源就占用多少资源”的吗？如果某个进程把系统资源吃完了该怎么办呢？轮到系统资源限制功能登场了。

## 系统资源限制：Cgroups

如果我们想抛弃虚拟机技术，用软件的方式直接限制某个进程所能消耗的资源上限，那么就需要内核、基础库、系统调用以及应用软件紧密协同配合。

2008 年，谷歌贡献的 Cgroups 第一次被合并进了 Kernel 2.6.24 中，自此，内核第一次拥有了完善的“进程资源控制”功能，这可以视为容器技术的第一声啼哭。发展到今天，Cgroups 的控制能力已经非常完善了，它可以限制、记录、隔离一个进程组所使用的全部物理资源，功能非常丰富：

1. Block I/O (blkio)：限制块设备（磁盘、SSD、USB 等）的 I/O 速率上限
2. CPU Set(cpuset)：限制任务能运行在哪些 CPU 核心上
3. CPU Accounting(cpuacct)：生成 Cgroup 中任务使用 CPU 的报告
4. CPU (CPU)：限制调度器分配的 CPU 时间
5. Devices (devices)：允许或者拒绝 Cgroup 中任务对设备的访问
6. Freezer (freezer)：挂起或者重启 Cgroup 中的任务
7. Memory (memory)：限制 Cgroup 中任务使用内存的量，并生成任务当前内存的使用情况报告
8. Network Classifier(net\_cls)：为 Cgroup 中的报文设置上特定的 classid 标志，这样 tc 等工具就能根据标记对网络进行配置

9. Network Priority (net\_prio): 对每个网络接口设置报文的优先级

10. perf\_event: 识别任务的 Cgroup 成员, 可以用来做性能分析

登录一台 Linux 系统, 执行 `mount -t Cgroup` 就可以看到本机挂载了哪些 Cgroups 资源, 如代码清单 2-1 所示。

代码清单 2-1 Linux 系统内挂载的 Cgroups 资源

```
root@PPHC ~ # mount -t cgroup
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/usr
b/systemd/systemd-cgroups-agent,name=systemd)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpuacct,cpu)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_prio,net_cl
```

有了 Cgroups, 我们就可以记录并限制一个或者几个进程所能使用的计算资源的上限了, 万事俱备, 只欠封装成应用了。

## 应用封装

### 昙花一现的 LXC

LXC 是第一代容器封装技术, 也是 Docker 最早期采用的技术方案, 但它虽然也把应用封装成了容器, 但却不是直接封装成的 pod, 一个 pod 中包含多个软件, 无法自定义。LXC 就像是新时代的 `yum groupinstall` 命令, 虽然它确实属于容器技术, 但是他的管理思维依然停留在“一次性安装多个软件”的阶段, 价值有限, 在 Docker 抛弃它以后, 逐渐没落。

### Docker : 一夜爆红只需要一个好点子

2010 年, 几个大胡子年轻人创立了一家名为 dotCloud 的 PaaS (平台即服务) 公司, 随后便获得了 YC (Y Combinator 风险投资公司) 的投资, 但是三年过去, 随着谷歌、亚马逊、微软等巨头的入场, dotCloud 的业务每况愈下。终于, 在 2013 年 3 月, 他们决定将核心技术 Docker 开源, 随后的一年, Docker 火爆全球。

Docker 的爆火在于它提出了一套恰逢其时的容器概念: ① 以应用为中心 ② 可以派生版本 ③ 镜像可以上传并共享。

## Docker 的技术优势

Docker 技术相比于 LXC，有了非常大的进步，它们架构上的巨大区别如图 2-6 所示。

## Traditional Linux containers vs. Docker

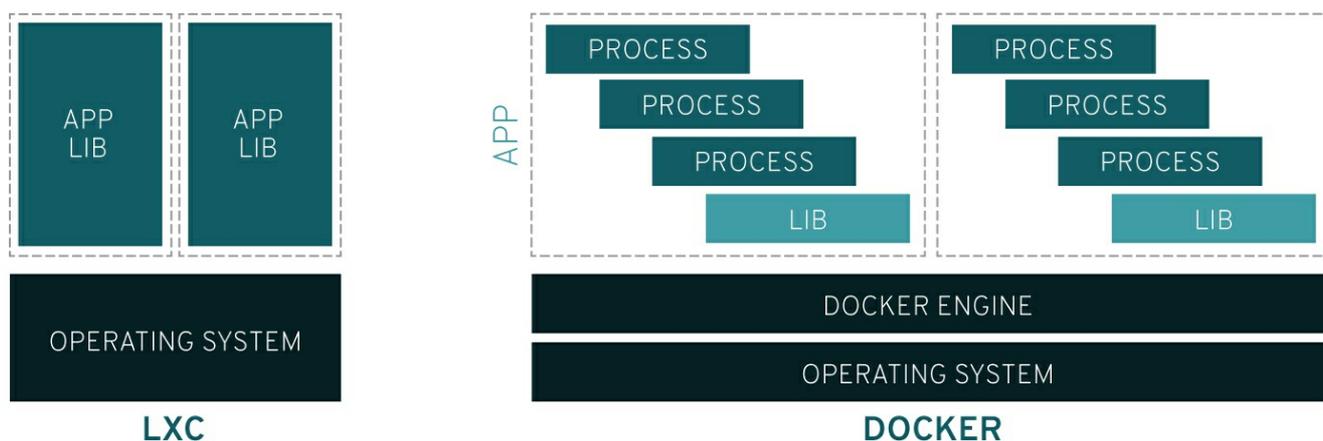


图 2-6 LXC 和 Docker 的技术架构对比

### 镜像技术开天辟地

如果必须找出 Docker 最有价值的创造，那一定是镜像。Docker 通过把整个操作系统的文件系统打包进镜像，真正实现了一次构建、处处运行。而在这之前，只有基于硬件虚拟化技术的虚拟机才能做到这一点，运行库虚拟化、虚拟机编程语言、解释型编程语言都做不到。

除了可以在任意支持 Docker 的 Linux 系统之间完美迁移之外，Docker 镜像还有一个巨大的创新：层。

### “以应用为中心”的思想

Docker 第一次把部署一个软件所需要的环境和依赖放在一起看做了一整个单独的“应用”。而 LXC 容器依然是一种“对系统的封装”，可以视作标准化、轻量化的新时代虚拟机。

1. 应用是独立的，可以随意部署的个体，像经典力学的规律，不随时间和空间的变化而变化。
2. 对系统的封装依然是在某一个时刻在某个操作系统上用标准容器的方式安装了软件，随着时间的流逝，操作系统拥有不同的版本和不同的软件补丁，在不同的时间得到的最终产物并不一致。

### 镜像技术的核心——层

Docker 层是 Docker 镜像的构建块，它们共享相同的文件系统和元数据。每个层都包含对上一层的增量更改，这使得 Docker 镜像更加轻量级和可重用。Docker 使用 UnionFS（联合文件系统）技术将多个层组合在一起，形成一个统一的文件系统。

Docker 层的概念有以下几个特点：

1. 共享：Docker 层可以在多个镜像和容器之间共享，这有助于减少存储空间的浪费。
2. 增量：每一层都是对上一层的增量更改，这意味着你只需要复制发生变化的部分，而不是整个文件系统。
3. 可读性：Docker 层的文件系统是只读的，这意味着你不能直接修改它。但是，你可以在运行时创建新的层来保存更改。

4. 可缓存: Docker 使用缓存机制来提高构建速度。当构建一个新的镜像时, Docker 会检查本地是否存在与新层相同的层, 如果存在, 则可以直接使用缓存的层, 而不需要重新构建。

基于层的概念, 每一个镜像拥有最多 127 个版本, 你不仅可以随时修改、提交、回退版本, 还可以直接基于基础镜像打造自己的软件, 以我们的静山平台为例, 我们可以用一个简单的描述在一个 pod 内搭建好所需要的所有软件:

1. 基于 CentOS 8 基础系统镜像构建新镜像
2. 执行命令: yum 安装 Nginx
3. 执行命令: yum 安装 php-fpm 8
4. 执行命令: yum 安装 MySQL 8
5. 执行命令: 复制 Nginx 和 php-fpm 的配置文件进入镜像
6. 设置启动命令: 启动 php-fpm、Nginx 和 MySQL

我们将上述过程转化为一个名为 Dockerfile 的文本文件, 该文件充当了我们想要构建的新镜像的配置文件。在本例中, 我们基于基础的 CentOS 8 镜像创建了一个新的层, 这一层进行了哪些修改, 后续读者只需阅读 Dockerfile 即可全面了解。这在以前的传统运维过程中几乎是不可能的。这个由 Dockerfile 文本描述的基础运行环境成为了一套“不可变基础设施”。

当然, 我们也可以将这个镜像上传到 DockerHub, 向全球开放, 其他人如果有类似的 LNMP 需求都可以使用我们的镜像。

本章后面的 2.8 节提供了可运行的 Dockerfile, 具备动手能力的读者可以尝试自行编写 Dockerfile, 并与代码清单 2-2 进行对照。

## DockerHub 彻底改变了运维的工作性质

以前的运维, 是坐在小黑屋里, 守着自己的几台机器, 去网上扒各种资料, 试图解决某一台机器遇到的某一个奇葩的问题。可能某一个不小心的 `yum update` 就导致某个系统组件无法启动, 需要折腾好长时间, 服务才能恢复服务。

有了 DockerHub 之后, 运维也成了为开源软件做贡献一样令人激动的事情: 人们可以在镜像社区中相互协作, 逐层搭建出适合自己需求的镜像。你和你的代码将立即参与到了全球运维代码的分发流程中。

除了相互协作, 在面对复杂的运维需求时, 解决问题的过程也是被明文记录的, 一旦你解决了这个问题, 那你的配置文件便可以随意更换机器部署: 一个正常运行的镜像, 可以跨越十年光阴, 从 E5-V4 上运行的 CentOS 7.0 (Kernel 3.10.0) 非常方便地迁移到 AMD EPYC™ 9654 上运行的 Centos Stream 9 (Kernel 6.4.2) 上, 不用担心出现兼容问题。

## 2.5 Kubernetes——软件定义计算

在详细介绍了容器技术的发展历程和技术特性之后, 我们现在正式进入容器编排领域。

容器技术彻底改变了软件的分发方式, 它使得软件不再仅仅分发二进制文件, 而是直接分发整个虚拟化环境

。这种变革使得软件交付更加便捷、精确和可控。而以 Kubernetes 为代表的容器编排工具，则通过软件对分散在世界各地的各种不同硬件配置的物理服务器进行逻辑上的“标准化”和“虚拟化”，进而将跨数据中心甚至跨时区的海量服务器的运维从杂乱无章的状态转变为全局统一协调的状态，让它们百依百顺，如臂指使——容器编排是一种更加优秀的基础设施并发技术。

## Kubernetes 为何会在 2014 年出现

Kubernetes 和 iPhone 出现的原因一样：人类科技的发展恰好到了这个阶段。

Kubernetes 作为软件定义的快速启动、高速伸缩、流量全动态管理的新时代的“虚拟机”技术，它的出现其实是和整个 Web 服务端生态息息相关的，确切地说是 Kernel、硬件和用户需求都发展到了变革的临界点。

## Kernel 的进步

由于 Kernel 中容器能力的进步，一个新时代轻量化的“虚拟机”技术出现了。

容器镜像比虚拟机镜像小很多，这带来了两大优势：快速启动和高速伸缩。

1. 传统虚拟机技术需要每个节点白白浪费 5GB 的磁盘空间，仅仅为了部署 200MB 的后端代码，容器的基础大小可以只有 5MB。
2. 镜像体积的下降，还带来了启动时间的大幅缩短。
3. 传统的服务发现需要应用代码来做，例如 Spring Cloud，而 Kubernetes 在平台层面直接实现了一套服务发现和负载均衡，让一分钟新增数千个节点成为可能。

## 网络的进化

除了容器技术，Linux 网络的进步其实才是技术突变的真正“奇点”。笔者认为容器编排超越虚拟机靠的并不是所谓的标准化可迁移的运行环境，虚拟机也是标准化可迁移的，Kubernetes 真正在“能力”上超越虚拟机，靠的其实是软件定义的容器网络。

传统虚拟机的时代，几乎只有二层 Overlay 网络可用

AWS 在 2009 年 8 月推出了 VPC 网络，第一次把虚拟机用户的虚拟机在网络层面隔离开来，而阿里云直到 2015 年 7 月才发布了 VPC 网络。在那之前，云服务商的租户都是跑在同一个二层网络下的，这十分的不安全，而且虚拟机之间还容易相互影响。

VPC 就是用 Overlay 网络实现的，而直到今天，这还是云虚拟机最新一代的网络技术。虚拟机网络的发展十分缓慢，很多看起来很普通的二层网络应用至今都无法很好地支持，例如 Keepalived 赖以生存的多播技术。

有了二层 Overlay 网络，不同租户的虚拟机之间终于是实现网络隔离了，而服务发现、负载均衡、全三层网络、Sidecar 等复杂后端架构所需要的特性就只能自己用应用代码来实现了。

## Linux 生态中网络的巨大进步

从 iptables 到 Open vSwitch, 从 Netfilter 到 veth, 从 IP in IP 到 VxLAN 再到 BGP 路由, Linux 生态在 21 世纪的前十五年取得了飞跃式的进步, 软件定义网络也从软件领域走向了硬件领域——当下主流的硬件网络设备供应商大多数的商品都已经是基于 Linux Kernel 来运行的了, 这足以证明 Linux 网络生态的性能、稳定性和可扩展性。

最开始 Docker 上面只有简单的 Overlay 网络, 只能在本机的 pod 之间进行标准二层通信, 跨节点只能使用端口映射把端口代理出去。后来 k8s 生态带来了 flannel 网络插件, 使用 IP in IP 技术让所有 pod 进入了同一个二层网络, 可以跨机器通信了。随后有人把用户态的 IP in IP 改成内核态的 VxLAN 来实现了, 进一步提升了网络性能。

在网络基础设施发展的过程中, Istio 横空出世, 以边车模式 (Sidecar) 运行架构改变了半个 Kubernetes 生态, 促成了今天服务网格的流行。Sidecar 能够以应用无感知的形式实现熔断、认证、度量、监控、负载均衡等各种高级功能, 成功地将流量全部管理了起来。

再往后, 就是 calico 带来了基于 BGP 路由协议的全三层网络, 配合 Istio, “流量全动态管理” 这个目标终于实现了, 分布式后端架构的“完全可观测” 畅想终于成为了现实。

## 用户需求的变化

传统机房托管时代, 各家的设备从防火墙、交换机到路由器、存储都是物理隔离的, 各家的服务器之间几乎都是在一个大二层网络之中, 没有任何的流量管控和高级安全措施, 全靠系统里安装的 iptables 防火墙, 而 iptables 甚至当初被发明出来都不是为了当防火墙用的。

最近十年, 规模暴增的头部互联网公司的服务器数量开始爆炸, 必须开发一种软件定义的新时代虚拟机技术了, 于是谷歌便把自己内部了多年的 Borg 容器编排工具使用 Go 语言重写以后开源了, 而这个开源软件就是 Kubernetes。

而在今天这个云计算时代, 中小公司在用惯了云服务之后, 也不喜欢运维岗位了, 软件工程师逐步取代了运维工程师, 以编写代码为手段, 承担起了企业服务器的运维工作, K8s 生逢其时, 扶摇直上, 迅速占领了各种规模的市场, 成为了新时代的服务器标准运维环境。

## Kubernetes 使用软件定义一切的哲学

Kubernetes 平台使用软件接口重新定义了什么是 CPU, 什么是内存, 什么是磁盘, 什么是网卡, 它直接接管了物理机的全部资源, 以软件接口的形式对外暴露资源, 这其实是天马行空的架空软件架构对海量服务器运维领域的一次降维打击: Google 善于开发和软件, 服务器太多管不过来? 一样也可以开发软件来解决啊。

## Kubernetes 相对于虚拟机的优势

Kubernetes 是人类面对暴增的物理服务器和上面数不清的虚拟机的管理难题时想出的一种解决方案, 正如设计模式是面向对象编程思想对现实问题的一种妥协。

以 Kubernetes 为代表的容器编排技术, 就是一种更容易被软件自动化管理的“虚拟机”, 它可以实现虚拟

机一样的功能——资源隔离、降低运维复杂度和性能瓶颈，而且，它还可以在裸金属环境下快速部署出一个数千台服务器的集群。虽然虚拟化技术研发出了很多像 SR-IOV 一样的硬件直通技术，但无论是物理网卡性能还是虚拟网卡性能，都不如容器技术：五年前笔者在一台非常老的服务器上做过超过一天的压力测试，Docker 的反向代理进程又快又稳，可以实现超过万兆的带宽，并持续 24 小时一个包都不丢。

传统虚拟机镜像巨大、启动缓慢、多个操作系统同时运行带来了资源浪费、迁移缓慢、管理困难，而 Kubernetes 镜像很小，启动迅速，资源损耗低，管理 API 丰富，更是可以轻易实现“单机多实例部署”和“目录共享”，可以轻松管理海量服务器，是新时代的优秀基础架构。

Kubernetes 就是云时代的宿主机虚拟化软件和虚拟机内操作系统的集合体。

## “云原生”迷思

在 Kubernetes 发布的前一年，2013 年，EMC 选择和它的子公司 VMware 以及外部投资者通用电气共同成立了 [Pivotal Software, Inc.](#) 公司，这家公司是由 EMC 2012 年收购的 Pivotal Labs 公司一套人马两块牌子摇身一变而来，而这家公司后来成为了云原生第一股，他们致力于“改变世界建造软件的方式”（We are transforming how the world builds software）。

2014 年 6 月，谷歌将其内部使用了数年的容器管理平台 Borg 使用 Go 重写之后开源，命名为 Kubernetes，瞬间成为了当红炸子鸡，不知道是业界苦秦久矣还是觉得谷歌的软件质量更高，随后的一年 Kubernetes 高歌猛进，迅速成为了容器编排领域的事实标准。

2015 年，当时和 Docker 公司合作的并不愉快的红帽调转枪口，迅速和谷歌合作成立了“云原生基金会”即“CNCF 基金会”（Cloud Native Computing Foundation），谷歌为 Kubernetes 发布了 1.0 版，成为了 CNCF 的第一个开源项目。笔者不想再去介绍精彩的容器编排战争，我们只需要知道最终的结果：Kubernetes 生态已经剥离了所有 Docker 专属代码，今天如果你想搭建一个 Kubernetes 集群的话，没有 Docker 的环境才是你的第一选择。Docker 公司的云计算业务早已经倒闭，而面向开发者的业务也变成收取桌面客户端使用费这一种惨淡的营生。

CNCF 成立的这八年来，云原生的定义改过两次，作为学习者我们无需关心今天的云原生到底是什么定义，你只需要知道：Kubernetes 就是云原生，云原生就是 Kubernetes。

## 2.6 Spring Cloud 是微服务的中间态

在后端部署架构从虚拟机向 Kubernetes 进化的过程中，Java 生态诞生了一款功能强大的微服务框架——Spring Cloud。该框架在代码层面提供了微服务架构中最重要注册与发现、负载均衡、容错处理、远程调用和链路追踪等工具和组件，使得在虚拟机时代，我们能够基于传统的虚拟机技术或者云计算虚拟机技术，利用普通的二层以太网，搭建出一套可用的微服务调用框架。

### Spring Cloud 并不是微服务架构的价值所在

需要注意的是，Spring Cloud 和 Kubernetes 所实现的这些价值，并不是微服务相比于单体应用的价值，微服务的价值是

1. 整体系统容量更大
2. 不同的微服务之间可以独立部署和快速迭代，这可以将大团队拆成小团队，提升宏观上的软件质量

Spring Cloud 其实是在弥补微服务架构的弱点。

## Spring Cloud 和 Kubernetes 关于微服务需求的对应关系

在架构上，微服务系统的需求是不会变的，所以 Spring Cloud 和 Kubernetes 有着几乎一一对应的解决方案，具体对应关系如表 2-1 所示。

表 2-1 Spring Cloud 和 Kubernetes 不同组件的对应关系

功能	Kubernetes	Spring Cloud
弹性伸缩	Autoscaling	N/A
服务发现	KubeDNS / CoreDNS	Spring Cloud Eureka
配置中心	ConfigMap / Secret	Spring Cloud Config
服务网关	Ingress Controller	Spring Cloud Zuul
负载均衡	Load Balancer	Spring Cloud Ribbon
服务安全	RBAC API	Spring Cloud Security
跟踪监控	Metrics API / Dashboard	Spring Cloud Turbine
降级熔断	N/A	Spring Cloud Hystrix

## 2.7 软件架构本质上是软件维护团队的组织架构

任何软件架构，本质上都是其背后的软件维护团队的组织架构。

笔者认为，容器、镜像、容器编排的兴起，与其说是一种生产力的创新（技术创新），不如说是一种生产关系的创新：在越来越大的服务器数量面前，以前那套基于虚拟机的管理技术不再适用了，Google 需要一种少数人管理数十万台服务器的新技术，于是 k8s 诞生了。

正如软件架构本质上是软件维护团队的组织架构，运维架构本质上也是运维团队的组织架构：规模大到必须自动化，于是容器编排便自然而然地出现了。写到这里，有一种唯物史观的感觉了。

重要的不是电影拍摄(讲述)的年代，而是拍摄(制作)电影的年代。——戴锦华（北京大学教授，电影评论家）

软件工程研究的对象是人

如果你设计并调整过软件团队的组织形式的话，笔者相信你一定能深刻地体会到这句话。

微服务架构最大的价值，是通过把庞大的软件开发团队进行拆分、解耦、独立组织，进而提升宏观上的软件质量，降低软件的开发成本。

## 宏内核与微内核

在操作系统领域，宏内核与微内核的技术路线之争一直不绝于耳，笔者跟别人不同，不纠结哪种技术路线更加正确，而是主要从组织的角度来看待这个问题。

所谓的微内核宏内核技术路线的差异，背后其实是协作方式的不同造成的：微软可以雇佣并管理庞大的工程师开发一个复杂的、功能完备的内核；而 Kernel 由于属于用爱发电，就只能守住自己的一亩三分地，把大量的系统关键基础设施让给其它的开源软件来实现；而庞大的周边软件 linux 一个人是无法掌控的，于是它们就默认运行在用户态了。

过去十几年，还是有不少组件由于性能和功能的需求从用户态成功进入了内核态的，例如 IPVS、cgroups、eBPF 等。

## 2.8 实战：使用 Docker 部署静山平台

多年以来，笔者一直认为，新技术是“学”不会的，新技术都是“用”会的，所以笔者在过去学习新技术的过程中，从来不专门规划所谓的学习时间，而是找几个入门的文字或者视频教程，直接开始做真实的可以赚钱的需求。读大学时，笔者会通过做外包的方式学习新技术，工作以后笔者会特意选择用新技术去开发一些非关键，但是需要在生产环境运行的软件。

所以，请大家跟着我一起，使用三种不同的容器组织形式，逐步演进，使用 Docker 和 K8s 部署我们的静山平台。

### 传统思维——打造全功能容器

前文我们说过一种制作静山平台容器镜像的方法，它使用的就是打造全功能容器的思想。需求如下：

1. 基于 CentOS 8 基础系统镜像构建新镜像
2. 执行命令：yum 安装 Nginx
3. 执行命令：yum 安装 php-fpm 8
4. 执行命令：yum 安装 MySQL 8
5. 执行命令：复制 Nginx 和 php-fpm 的配置文件进入镜像
6. 设置启动命令：启动 php-fpm、Nginx 和 MySQL

### Dockerfile

我们先新建一个文件夹，然后切换到该目录内：`mkdir -p jingshan && cd jingshan`，然后新建一个名为 Dockerfile 的文件，根据前面的需求写出所需的配置文本，填入其中。可以参考笔者写的代码清单 2-2。

代码清单 2-2 jingshan:0.1 版本镜像的 Dockerfile

```
# 基于 CentOS 基础镜像
FROM centos:latest

# 进入yum.repos.d 目录下
RUN cd /etc/yum.repos.d/

# 修改源链接
RUN sed -i 's/mirrorlist/#mirrorlist/g' /etc/yum.repos.d/CentOS-*

# 要将之前的mirror.centos.org 改成 vault.centos.org
RUN sed -i 's|#baseurl=http://mirror.centos.org|baseurl=http://vault.centos.org|g' /etc/yum.repos.d/Ce
S-*

# 安装 Nginx、PHP8 和 MySQL8
RUN yum -y install epel-release && \
    yum -y install wget && \
    yum -y install nginx && \
    yum -y install php php-fpm php-mysqlnd php-opcache php-mbstring php-json php-gd php-curl php-xml p
zip && \
    yum -y install mysql-server && \
    yum clean all && \
    rm -rf /var/cache/yum/*

# 配置 PHP-FPM
COPY php-fpm.conf /etc/php-fpm.conf

# 配置 Nginx
COPY nginx.conf /etc/nginx/nginx.conf

# 创建 fpm 需要的目录
RUN mkdir -p /run/php-fpm

# 暴露端口
EXPOSE 80

# 启动 PHP-FPM、Nginx 和 MySQL 服务
CMD ["sh", "-c", "php-fpm -F -R && nginx && /usr/sbin/mysqld"]
```

有了 Dockerfile 还不够，我们还需要准备 Nginx 和 PHP 的配置文件，分别如代码清单 2-3 和 2-4 所示。

代码清单 2-3 jingshan:0.1 镜像所需的 Nginx 配置文件 nginx.conf

```

user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;
events {
    worker_connections 1024;
}
http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';
    access_log /var/log/nginx/access.log main;
    sendfile on;
    keepalive_timeout 65;
    server {
        listen 80;
        server_name localhost;
        root /usr/share/nginx/html;
        index index.html index.htm index.php;
        error_page 500 502 503 504 /50x.html;
        location = /50x.html {
            root /usr/share/nginx/html;
        }
        location ~ \.php$ {
            fastcgi_pass 127.0.0.1:9000;
            fastcgi_index index.php;
            fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
            include fastcgi_params;
        }
        location ~ /\.ht {
            deny all;
        }
    }
}

```

代码清单 2-4 jingshan:0.1 镜像所需的 PHP 配置文件 php-fpm.conf

```
[global]
pid = /run/php-fpm.pid
error_log = /var/log/php-fpm.log
daemonize = yes

[www]
user = www-data
group = www-data
listen = 127.0.0.1:9000
listen.owner = www-data
listen.group = www-data
listen.mode = 0660
pm = dynamic
pm.max_children = 5
pm.start_servers = 2
pm.min_spare_servers = 1
pm.max_spare_servers = 3
pm.process_idle_timeout = 10s
include = /etc/php-fpm.d/*.conf
```

## 生成镜像

接下来就可以生成镜像了，执行 `docker build -t jingshan:0.1 .` 命令，稍等片刻，我们就得到了一个镜像，执行 `docker images` 可以看到，如代码清单 2-5 所示。

代码清单 2-5 `docker images` 命令的执行结果

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
jingshan	0.1	f2f0eb3801f1	2 minutes ago	566 MB

## 基于镜像启动容器

可以看到镜像已经在我本地成功生成。接下来，基于这个镜像创建一个容器，服务就可以正式跑起来啦！执行以下命令：

```
docker run -d -p 10086:80 --name first-jingshan jingshan:0.1
```

然后，执行 `docker ps -a` 就可以看到正在运行的容器了，笔者的运行结果如代码清单 2-6 所示（此时该容器已经运行三个小时）。

代码清单 2-6 `docker ps -a` 命令的执行结果

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
33191d4736c9	jingshan:0.1	"sh -c 'ph..."	3 hours ago	Up	3 hours 0.0.0.0:10086->80/tcp	first-gshan

## 检查运行结果

此时，让我们访问服务器的 10086 端口，可以看到如图 2-8 所示的网页。

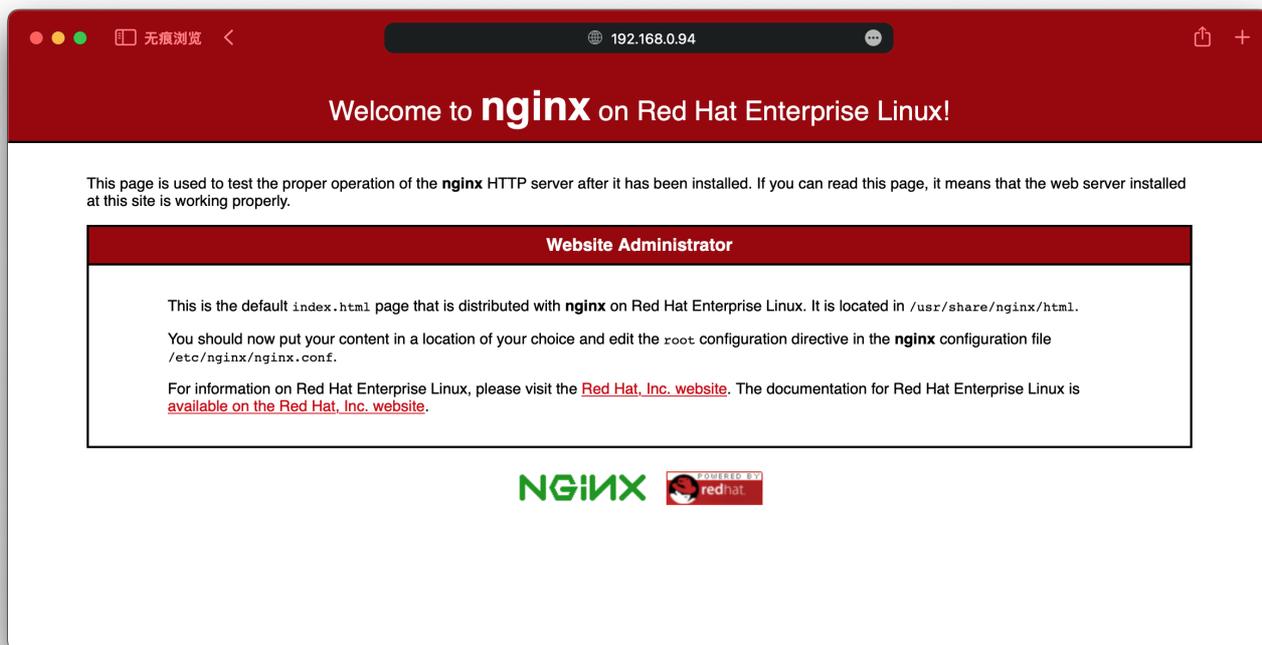


图 2-8 网页访问结果

## 容器思维——多个容器部署在同一个 Pod 内

前面在一个容器内安装所有软件其实不是容器技术的正确使用方式，如果读者需要在生产环境使用 Docker，即便是单机，也推荐使用单台服务器部署 Kubernetes 编排工具，然后使用将多个容器部署在同一个 Pod 内的方式来组织这些容器。

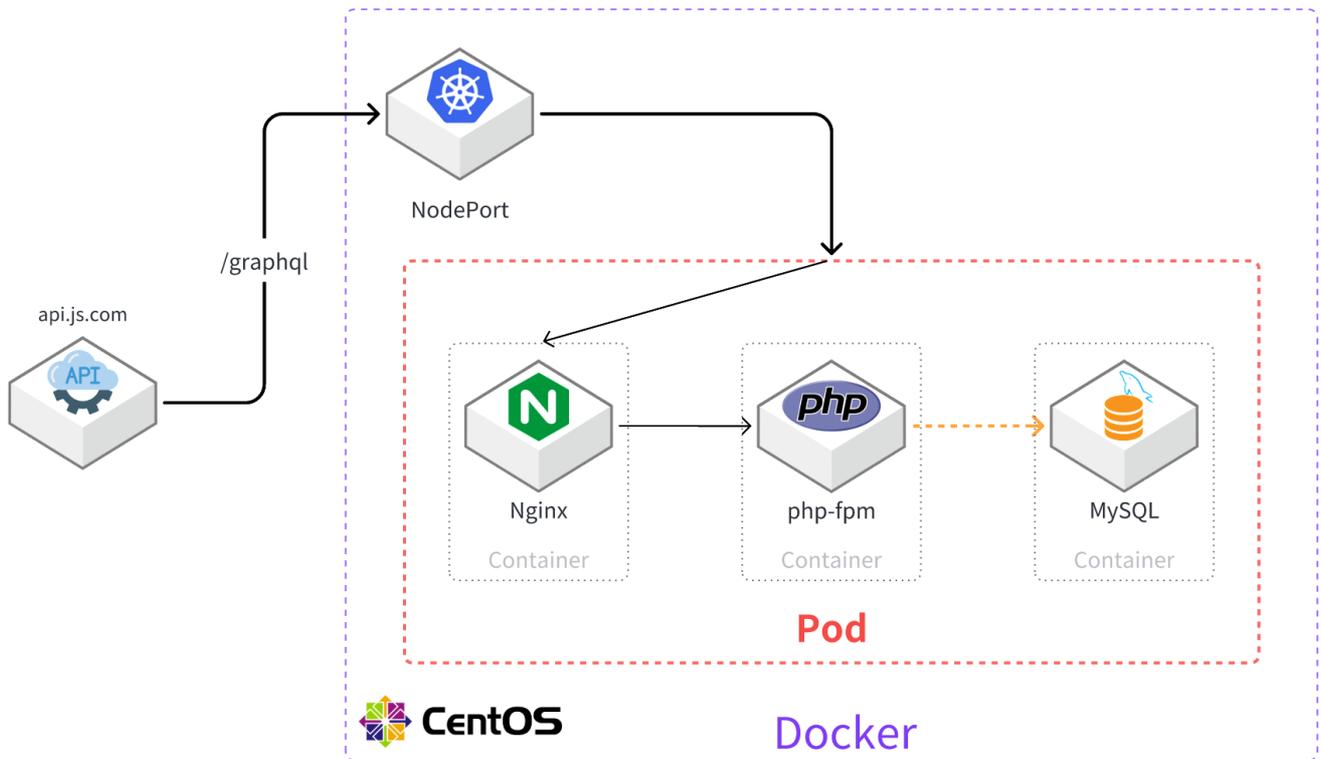


图 2-7 多个容器部署在同一个 Pod 内

在这种组织方式下，系统架构图如图 2-7 所示。

下面笔者陈述一下思路，复杂而枯燥的 Kubernetes 配置文件就不再列出，感兴趣的读者可以自己尝试。

1. 创建一个 pod，在里面添加 container
2. 添加一个基于官方 Nginx 镜像的 container，修改配置文件以配合 php-fpm，对本地（Pod 内）暴露 TCP/80 端口（如果你需要 HTTPS 还需要暴露 443 端口并写入你申请到的 HTTPS 证书和私钥）
3. 添加一个基于官方 php-fpm 镜像的 container，指定根目录到 Nginx 根目录，对本地（Pod 内）暴露 TCP/9000 端口
4. 添加一个基于官方 MySQL 镜像的 container，指定初始用户名与密码，对本地（Pod 内）暴露 TCP/3306 端口
5. 使用 Kubernetes 的 NodePort 服务对外部网络暴露本 pod 的 80 和 3306 端口

在编写完 Kubernetes 配置文件后，部署这个配置文件，Kubernetes 会自己下载镜像文件，按照你的配置搭建出一个 pod：你可以得到一个类似传统虚拟机一样的“模拟虚拟机”，里面运行了三个软件，分别是 Nginx、php-fpm 和 MySQL，这台模拟虚拟机有独立的 ip，并且它已经将自己的 80 和 3306 端口映射到了宿主机的 ip 上。

需要注意的是，在标准的 Docker 环境或者 Kubernetes 环境下，你无法使用宿主机低于 1024 的端口，这些端口只有使用 root 权限启动的进程才可以监听，所以我们还需要在 Kubernetes 集群之外搞一个公网流量入口，才能真正对公网提供服务，这个部分我们在后面的“负载均衡与网关”的章节再予以详细阐述。

平台思维——每个容器一个 Pod

前面是单机 Kubernetes 平台上的部署方法，而如果我们拥有一个多台机器组成的 Kubernetes 集群呢？那就需要按照最新的精细流量管理的方式来组织我们的静山平台了。精细流量管理的基本方法是“层层代理”

:

1. 外部负载均衡集群将公网流量分发到 Kubernetes 集群中的某一台服务器上。这可以通过使用如 Nginx、HAProxy 等负载均衡器来实现，但是更推荐直接使用云服务商的负载均衡服务。
2. 随后，流量进入 Ingress Gateway（入口网关）组件，根据网关的配置，将 HTTPS 请求解包，再把 HTTP 请求发送到某个 Pod 的某个端口上。Ingress Gateway 是 Kubernetes 中的一个组件，用于处理外部流量并将其路由到适当的服务或端口。
3. 之后，流量进入该 Pod 内的边车（Sidecar）组件，又根据边车的配置，将流量发送到 Pod 本地的某个特定的端口上，在这个过程中，记录流量的关键信息。边车模式是一种微服务架构设计模式，其中每个主要服务的 Pod 都有一个额外的 Sidecar 容器，用于处理与主要服务相关的辅助功能，如日志记录、监控、流量控制等。

这是目前最流行的大规模 Kubernetes 集群中流量的管理方式，最后的 Sidecar 就是大名鼎鼎的“服务网格”。具体架构如图 2-8 所示。

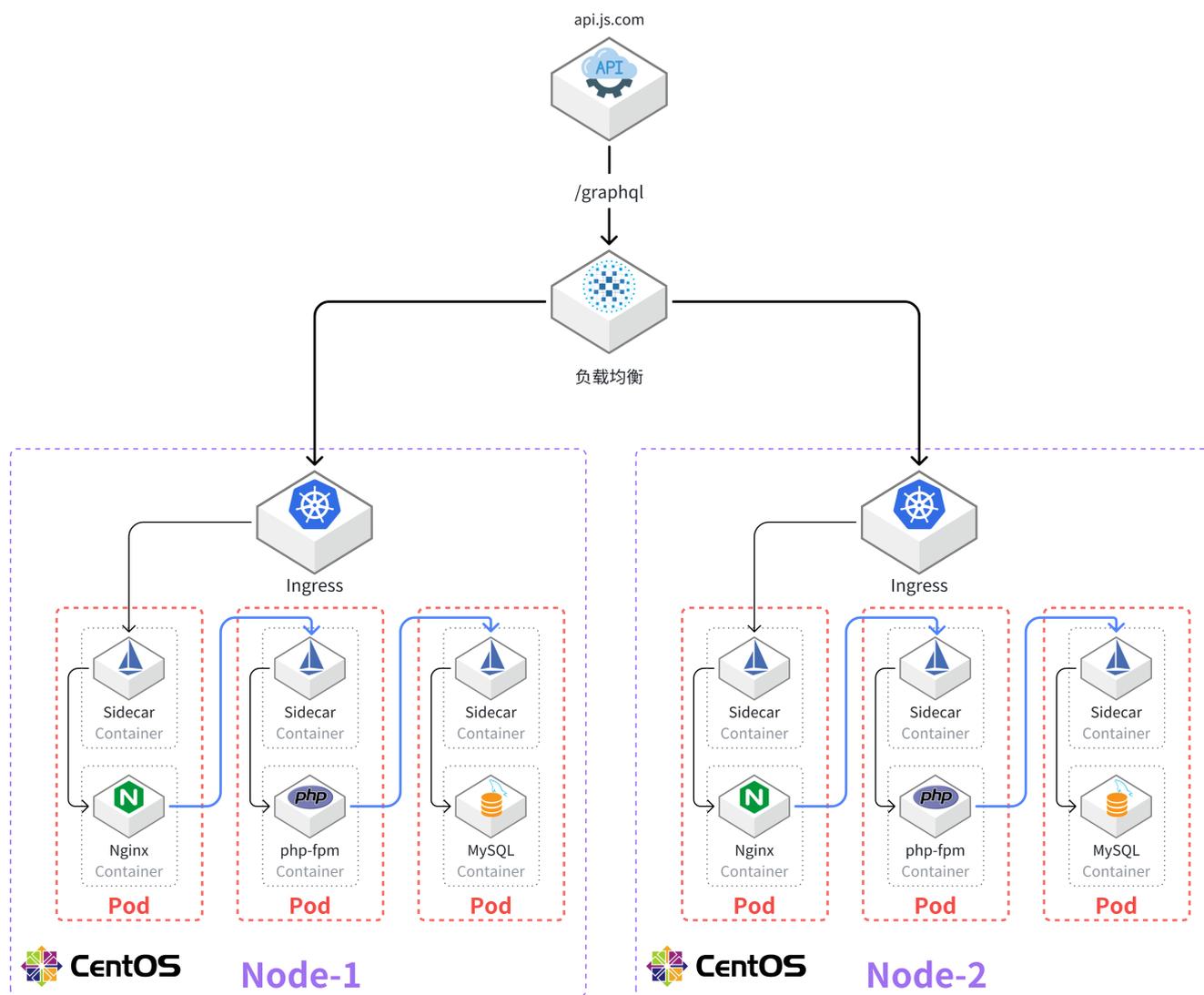


图 2-8 每个容器一个 Pod

## 2.9 面试题

**No.05** : 大型项目架构分层的原因是什么？

由于软件开发的特殊性，导致在一个团队内只有少量的开发可以彻底了解某一个系统，在人数增加以后，沟通复杂度会呈指数型上升。所以，在系统越来越复杂的时候，就需要拆分团队，每个小团队负责某个或者某些服务，服务之间以 API 的形式互相调用，减少沟通损耗，提升研发效率。

亚马逊的创始人贝佐斯有一个披萨定律：如果两张全尺寸的披萨无法喂饱团队，说明团队应该拆分了。

软件架构本质上是软件维护团队的组织架构，组织上是分小组的，那软件架构上自然就是分层的。

**No.06** : 系统如何横向扩张？

一个单体系统进行分布式架构演进的第一步一般就是横向扩张：增加数据库的硬件资源，并将后端代码分散到多台服务器上运行，用一个负载均衡软件对用户流量进行分发。横向扩张通常包括以下步骤：

1. 增加硬件资源：首先需要增加更多的服务器资源，这可能包括CPU、内存、存储等硬件设备。
2. 代码分散运行：然后，需要将后端的代码分散到这些新增的服务器上运行。这样，每个服务器都可以独立地处理一部分工作负载。
3. 引入负载均衡器：为了使用户请求能够均匀地分配到所有的服务器上，需要引入一个负载均衡器。这个负载均衡器可以根据服务器的负载情况，动态地将新的请求分发到空闲的服务器上。

普通的横向扩展使用 Nginx 加上固定数量的后端服务器就可以实现，如果后端压力不定，为了省钱，可以使用 Consul 服务发现，让负载均衡器自己发现并适应不同数量的后端服务器，这样既可以满足业务高峰期的计算资源需求，也可以省钱。

如果业务规模再大，就需要更复杂的 K8s 软件来进行更大规模的计算资源调整了，需要引入外部负载均衡器，设计更大的更复杂的后端集群。而数据库由于拥有单点性，是很难横向扩张的，此时可能需要对后端系统进行拆分，分散数据库的压力，并对高性能需求的模块进行缓存改造，努力顶住压力。

**No.07** : K8s 技术比虚拟机技术先进在哪里？

K8s比虚拟机技术先进的地方主要有：

1. 弹性更强：K8s 能快速启动海量容器，实现更迅速的资源扩展和收缩，摆脱了时间和人工操作的限制，提升了部署速度。
2. 资源利用率更高：K8s 更轻量，能部署更多应用在同一物理服务器上，更精细地控制每个应用的资源占用，节约成本。
3. 管理更方便：K8s 提供全功能 API，可完全自动化运维过程，避免人工错误，适应大规模应用和跨时区数据中心的发展，确保系统稳定性。

## 第 3 章 突破编程语言的性能瓶颈

没有任何编程语言是单纯的“语法集合”，每一种语言都是它背后“运行架构”的外在表现，语言之间的差

异本质上是运行架构设计方向的差异。

运行架构是指编程语言的设计者在设计语言时，为了实现某种功能或者解决某种问题，所采用的一种算法和数据结构的组合。这种组合决定了编程语言的性能、可扩展性和易用性等方面的特征。

例如，C++ 是一种静态类型的编译型语言，它的运行架构就是操作系统暴露出的进程、线程、系统调用等基本能力，包括了内存管理、指针操作等底层机制，这使得 C++ 能够进行高性能计算和系统编程。而 Python 是一种动态类型的解释型语言，它的运行架构是在一个解释器中解释执行字节码，所以他能实现垃圾回收、动态类型检查等高级特性，这使得 Python 非常适合进行快速原型设计和数据分析。

因此，选择一种编程语言时，最重要的其实是考虑它的运行架构是否符合你的需求。

### 3.1 概述

编程语言性能这个话题是本书所有话题之中门槛最低的。关于各个语言性能的帖子长期密集地出现在各种技术论坛中，无非就是 C 比 C++ 强，C++ 比 Java 强，Java 再秒杀 PHP，Python 忽强忽弱，Ruby 比所有技术都慢，本章不讨论这些浅显的性能对比结果，因为这一点都不性感，一点都不哲学。

语言之间的差异本质上是运行架构设计方向的差异。笔者希望读者能够放下你之前所有的关于编程语言性能优劣的认识，跟随笔者的思路，从运行架构的角度再次观察这些常见的编程语言，了解历史的潮流如何选中了这些幸运儿，理解每一个语言为什么这么设计，它这么设计解决了时代面临的哪些棘手的问题，这么设计获得了哪些好处，又同时伴随而来了哪些坏处。

### 3.2 互联网系统规模发展史

编程语言的发展和互联网的发展是密不可分的，本节我们简单了解一下互联网系统规模发展史。

#### WWW 的诞生

1991 年，伯纳斯·李 (Tim Berners-Lee) 在欧洲核子研究中心 (CERN) 工作期间，发明了万维网，也被称为 WWW，这被公认为互联网的起点。目前，我们仍然可以访问这个人类历史上的第一个网页：<http://info.cern.ch/hypertext/WWW/TheProject.html>。



图 3-1 伯纳斯·李使用的 NeXT 品牌计算机

伯纳斯·李使用如图 3-1 所示的 NeXT 品牌的计算机开发了第一个网页，并当做服务器对外提供服务。NeXT 是乔布斯被赶出苹果公司后创办的一家专门生产工作站的企业。经过 11 年的发展，苹果公司收购了 NeXT，然后乔布斯重新回到苹果公司。随后，基于 NeXT 的技术，苹果公司开发出了 Macintosh 操作系统。如今，Macintosh（今天叫 macOS）系统依然是大部分前端开发者每天工作使用的操作系统，这场始于 32 年前的 WWW 故事至今还在继续讲述。

## 毫无压力的静态网页时代

与我们常规的认知有所出入的是，后端技术的诞生实际上比网页的发明要早得多。Perl 语言早在 1987 年就已经面世了。BS（Browser/Server）架构的出现实际上要比 CS（Client/Server）架构要晚很多。

在 1993 年，Rob McCool 首次利用 Perl 语言实现了 CGI，这标志着我们所说的动态网页（通俗来说，就是能够根据用户的输入和交互实时生成内容的网页）的诞生。

1994 年 1 月，斯坦福大学的两位在读硕士杨志远和 David Filo 创立了“Jerry's Guide to the World Wide Web”网站，这个网站收录的都是经过人工筛选的优质网站链接。该网站一经推出便迅速走红，两个月后，它更名为雅虎，人类的互联网时代正式拉开了帷幕。

雅虎网从一开始就采用了 Perl 技术来构建。尽管在 1994 年的点击量就已经超过了一百万，但在那个时期，只要你的服务器带宽足够大，CPU 的计算性能和内存容量都是足够的：因为当时并没有太多复杂的功能需要处理，只需要进行一些简单的数据库存取和静态 HTML 文件生成操作。

互联网开始流行以后，PHP 在 1994 年诞生，ASP 在 1996 年诞生，JSP 在 1998 年诞生。在 Web 2.0 出现之前，互联网上绝大多数的页面依然是静态的、离散的，并没有太多的高并发压力，网站的极限容量几乎完全取决于服务器的互联网带宽大小。

## Web 1.0 时代最具创新力的企业竟然是必胜客

1995 年，必胜客开发了一个网页，消费者可以在网页上下单，等披萨送到以后再付现金。这个创举让必胜客成为了实打实的“最具创新力企业”。

## 小有压力的 Web 2.0 时代

2003 年 9 月，MySpace 上线，随后的 2004 年 2 月，Facebook 也加入了社交网站行列。这两件事成为了 Web 2.0 时代的开端，也是高并发系统的开端。

Web 2.0 可以用一个词来精确定义：用户生成内容（User Generated Content，简称 UGC）。相比于 Web 1.0 时代由编辑们精心筛选的信息，Web 2.0 的核心理念是“用户主动生产的信息更重要”。在这个时代，每时每刻都有数百万用户积极参与到内容的创造与分享中，带来了两个主要的技术挑战：

1. 海量并发写入：相比 Web 1.0 时代中只需读取信息的简单需求，Web 2.0 时代的每个时刻都要求系统能够承受大量并发写入数据的压力。这意味着系统的写入能力需要得到大幅度地提升。
2. 实时分发需求：在 Web 1.0 时代，准备时间可以相对较长，例如花费几个小时生成并存储静态网页。然而，在 Web 2.0 时代，一旦有新的动态信息发布，如用户发布的微博、状态更新等，就希望这些内容能

够迅速被推送给受众，实现实时分发。这就要求系统具备更低的分发延迟和更高的处理速度。

## 同时起飞的电子商务

与 Web 2.0 同时起飞的还有电子商务市场。

2006 年，中国电商市场的总和交易额只有 230 亿元，但到了 2007 年，这个数字直接飙升到了 20000 亿元。这主要得益于互联网用户的快速增长和电子商务平台的兴起。2008 年，中国的网民数量已经超过了 2 亿，这为电子商务的发展提供了巨大的用户基础。

阿里巴巴旗下的淘宝网是中国最大的电子商务平台之一，在 2009 年，淘宝网推出了第一次“双 11”购物节，这是一次为期一天的大型促销活动。当天的交易额达到了平均每日交易额的十倍以上，创造了新的销售纪录。这个活动的成功不仅推动了阿里巴巴的业务增长，也促进了整个电子商务市场的发展。

随着时间的推移，每年的双 11 购物节都成为了一次压力更大的高并发考验，交易额逐年攀升，并在 2020 年达到了顶峰——当日全网交易额超过 5700 亿。

## 异军突起的春晚红包

目前人类历史上最大规模的互联网并发流量既不是 12306 买火车票，也不是双十一大促，而是由春晚红包活动产生的。

近年来，患上流量饥渴症的互联网大厂为了拉新，在营销层面下了不少功夫，而春晚作为一场上亿人观看的晚会，通过红包互动的方式，能够极大程度上满足大厂提高产品曝光率和渗透率的需求。2018 年春节期间，淘宝与春晚达成独家合作，发放超过 10 亿的现金礼包，当时淘宝为春晚准备了三倍于“双 11”的服务器资源。而就在主持人口播活动开始的一瞬间，服务就崩溃了。事后分析，当时的瞬时流量是当年“双 11”的 15 倍。

承包了第二年春晚红包活动的百度，在复盘文章中给出了春晚红包活动的峰值压力：网络请求高达五千万次每秒，这个数字应该也是人类历史上单个信息系统所承受过的最大并发量了。

## 3.3 后端语言变迁史

---

在 Web 1.0 时代，第一个进入互联网技术领域的编程语言是 perl，但在动态网站普及的短短几年之后，Web 2.0 时代就到来了。随后，各种产品形态层出不穷，各种语言、各种技术轮番登场，最终锤炼出了 Java 和 PHP 两种主流技术。

### 纯互联网业务喜欢 PHP

如果一家公司是通过提供免费的在线互联网产品获得用户，再通过卖广告、皮肤、会员等虚拟商品赚钱，那它就属于做纯互联网业务的公司。典型的有腾讯、百度、Facebook、字节、快手等。

### 扎克伯格选择用 PHP 和 MySQL 搭建了 Facebook

2004 年，扎克伯格选择了当时比较流行的 PHP 语言以及不太成熟的 MySQL 数据库来搭建第一版的 Face

book。这个决策在短时间内就取得了巨大的成功，使得 Facebook 战胜了已经拥有巨大先发优势的 MySpace，成为了全美乃至全世界最大的熟人社交网站。

然而，随着用户数量的爆炸式增长，Facebook 不得不背负起巨大的技术债务。为了应对高并发挑战，他们使用 PHP 的语法开发了一种强类型、且编译成二进制文件进行部署的 HACK/HVVM 技术，在 PHP 7 发布之前曾颇为流行。

在接下来的十年里，Facebook 与同样选择了 MySQL 的 Google 以及阿里巴巴一起，将 MySQL 推向了开源数据库的巅峰。这个故事我们在后面的章节还会详细讲述。

和电商业务这种长流程的需求相比，社交网络的数据处理流程都很短，不需要后端代码在内存中做很多的运算，系统的重点更多地落在了数据管理层面，例如高性能的内存缓存和不会丢数据的磁盘数据库。正是由于 PHP 和 MySQL 比较匹配“短周期数据处理”需求，使得它们成为了构建大型社交平台的首选技术，Facebook 的成功也证明了这一点。后来，通过发展这些技术，Facebook 成功地解决了用户数量快速增长带来的巨大压力，为用户提供了稳定、高性能的服务。

新浪、百度等卖网页广告的公司都选择了 PHP

PHP 语言一直以其出色的数据展示和网页生成能力而闻名。首先，PHP 的开发速度非常快，可以迅速实现各种功能需求。其次，PHP 的部署简单便捷，可以轻松搭建和管理网站环境。最重要的是，PHP 非常适合横向扩容，能够应对大规模访问和数据处理的需求。因此，许多卖网页广告的公司都选择了 PHP 作为它们的首选技术。

然而，字节和快手并没有选择使用 PHP。这主要是因为它们在它们成立的时代，PHP 的继任者 Go 语言已经在中国互联网领域开始普及，甚至比 Go 语言在美国的流行更早。Go 语言以其高效、并发性强的特点，成为了新一代互联网公司的首选。因此，字节和快手在技术选型上选择了 Go 语言，相比于 PHP 能够节约不少的服务器采购成本。

电商网站几乎全都选择了 Java

美国亚马逊

1995 年，亚马逊以一个图书销售网站的形式上线，从那之后经历了两次核心技术的换代：首先是从 C 迁移到了 Perl，然后是从 Perl 迁移到了 Java。

美国最大的 C2C 电商 eBay 的核心技术也是很早就从 Perl 迁移到 Java 了。

中国淘宝

淘宝最开始是购买的基于 LAMP 技术的 PHPAution 电商系统，于 2003 年 5 月首次上线。两个月后，基于 like 的搜索顶不住了，于是把自研的阿里巴巴中文站的搜索引擎复制到了淘宝网。到了 2004 年 1 月，MySQL 彻底顶不住了，换成了 Oracle 数据库。又过了一个月，淘宝网开始以 SUN 公司工程师驻场开发的方式，逐步替换掉了全部的 PHP 代码，从此淘宝网成了一个使用 Java + Oracle 开发的网站。

时至今日，阿里巴巴早已经完成了“去 IOE”战略，IBM 小型机、Oracle 数据库和 EMC 高端存储已经从阿里巴巴的机房消失，但是 Java 依然流淌在阿里巴巴集团的血液里。

## 其它电商公司

PHP 语言在纯互联网公司中的核心地位在最近几年已经被 Go 全面替代了，新兴的字节、Bilibili 都开始用 Go 作为主力后端语言。而同样很晚成立的拼多多却起步就用 Java 语言作为主力后端技术，而类电商业务的美团，其后端核心技术也是 Java。

至于京东？在笔者本科还没毕业的时候，就在微博评论里咨询过京东 CTO 李大学：“我看京东加入购物车的接口依然是 .aspx 后缀的，难道京东还在用自己的初代技术 asp？”他回复我“不要只看表面”。实际上京东的核心交易系统一共经历过三代：2003 年购买的第一代 asp 商城，2008 年花了三个多月封闭开发出的基于 .NET 的第二代系统，2012 年开始使用 Java 技术开发的第三代核心交易系统。

## 为什么电商网站都用 Java

最重要的原因：时也运也。

在当时，PHP 技术还不够成熟，主要用于处理网页展示等耗时较短的业务。然而，在面对海量并发的情况下，PHP 无法有效管理和控制与数据库的大量连接。更重要的是，当时的数据库技术也不够成熟，性能较差，功能有限，远远无法与拥有小型机支持的商用解决方案如 Oracle 相媲美，更没有如今各种高性价比的基于 MySQL 的云数据库。

如果我们深入分析技术原因，笔者认为主要有两个。

首先，电商业务是一个长流程业务。如果数据能够在内存中停留足够长的时间，那么系统的整体性能一定会更高。这可以大大缓解 PHP 严重依赖外部数据库（包括内存数据库和磁盘数据库）而导致的性能问题。简单来说，这就是常驻内存技术和每次都需要新开进程技术的核心区别。

其次，Java 是一种对开发者进行强约束的技术。一个优秀的架构师就决定了项目的质量，无论是月薪八千的人还是月薪三万的人来编写业务代码，最终的结果都相差不大。

## 很多其他业务类型的互联网公司也主要使用 Java

除了电商领域，许多其他类型的互联网公司也主要使用 Java 作为主要开发语言。例如 LinkedIn、AT&T、Verizon、IBM 等头部公司都将 Java 作为它们的首选后端开发语言。甚至现在，连 Facebook 的第一后端语言也已经变成了 Java。

## 其它昙花一现的后端技术

在 Web 生态发展的过程中，除了 PHP、Java 双雄之外，也出现了如 Python、Ruby、Scala、Node.js 这样的后端技术之花，但它们都没有达到 PHP 和 Java 那样的高度，下面笔者简单介绍一下它们的兴起和衰落之路。

## Python

Python 作为一种被广泛使用的编程语言，曾经也有一些比较流行的 Web 框架，有不少知名的互联网公司采用 Python 作为主要开发语言，例如 YouTube、Dropbox，国内的豆瓣和知乎等，但是随着 Node.js 和 Go 的兴起，以及 Python 在处理高并发流量和复杂业务时候的工程化能力较弱，现在已经逐渐没有新的公司使用 Python 来开发自己的核心系统了。但是意外的是，最近十年机器学习的崛起又让 Python 重新焕发了生命力。

## Ruby

Ruby 的兴起来源于那段著名的发布于 2005 年 11 月 8 日的 16 分钟视频 [《Ruby on Rails demo》](#)，可以说，正是 Ruby on Rails 拉开了 Web 技术百花齐放的大幕。这段视频由 David Heinemeier Hansson（简称 DHH）录制，DHH 这个名字的知名度一直贯穿到了移动互联网时代。

Github、Twitter、Airbnb 是几个曾经使用 Ruby on Rails (ROR) 技术开发的大型网站。由于 Ruby 这门编程语言对程序员过于友好，导致 ROR 的运行性能相对较弱。因此，随着这些公司的用户量上升，它们纷纷选择将核心系统使用其他技术进行了重写。

## Scala

Scala 第一次被大众所熟知是在 2009 年，当时 Twitter 宣布将核心系统从 Python 语言换成了 Scala 语言，据说减少了 90% 的物理服务器占用，并且在 2012 年美国总统大选期间支撑 Twitter 史无前例地没有宕机。

Scala 可以看做一个运行在 JVM 虚拟机上的、经过了更高级封装的基于 JAVA 的 DSL（领域专用语言），它的能力本质是 JVM 的能力，当然，Java 能力本质上也是来自 JVM。

## Node.js

Node.js、Go 这两种技术能够爆火都是因为出了类似“五分钟开发个人博客”的教程，过去十几年，如果某种技术特别适合拿来开发 Web，那它就能火。根据 2023 年最新的调查，JavaScript 是目前头部科技公司广泛使用的语言之中排名第一的语言，除去前端因素，Node.js 完善的生态，丰富的开源工具库，以及一直在进步的官方维护团队功不可没。

如今，Airbnb、Netflix 等企业主要使用 Node.js 作为后端语言，虽然使用它作为主要后端语言的公司不多，但几乎所有的公司都会使用它。因为它特别适合拿来做“用户接入层”，它可以并发聚合接口，减少流量消耗，提升前端体验。

## 大前端时代

Web 2.0 发展到顶峰的标志，就是乔布斯拿出了人类科技前沿的结晶、科技与人文十字路口的路灯——iPhone 4，移动互联网的时代到来了，手机迅速对所有年龄段的所有人完成了“个人计算机普及”。而前端技术也完成了 JavaScript、jQuery、Angular、React/Vue 的代际更迭，APP 和大前端的时代到来了。

此后，界面由前端完全掌控，后端团队不用再输出网页了，提供稳定且高性能的 API 成了唯一的工作，于是

后端技术也进入了新的时代。

## Go：新时代的 PHP

移动互联网时代，由于网民数量的激增和 24 小时随时都可以使用的手机的普及，APP 的同时在线人数相较于网页时代又提升了一个数量级，因此后端性能压力也上了一个数量级，这推动了 PHP 的接班人——Go 语言的流行。

十一年前笔者第一次写 Go 的时候就意识到了，它就是“C with net”——自带网络库的新时代万能底层语言，写起来像 PHP，跑起来像 C++。Go 还在语言和编译器层面提供了可以实现“超并发”的 Goroutine，普通开发者也可以轻松写出超高性能的接口。

PHP 在中国正在退潮，笔者已经很久没有看到五年以内经验的 PHP 程序员的简历了，目前在笔者公司内部，也正在推动 Go 语言和 Java 语言的落地，在微服务环境下，这也成了比较容易做到的一件事情。

## 3.4 语言特性如何决定性能

通过 3.1 章节的“互联网系统规模发展史”我们可以看出，编程语言的性能从来就不排名第一的因素，不同的业务类型有不同的最佳语言。下面我们逐个分析前面出现过的主流编程语言的优缺点，感受产品形态、业务需求和技术方案的相互追逐，相爱相杀。

### 以 PHP 为代表的全阻塞语言

PHP 是一种单线程全阻塞语言：在每个 HTTP/FastCGI 请求中，PHP 解释器会启动一个进程/线程来运行一段 PHP 代码，在运行的时候，无论是读写磁盘（磁盘 I/O）还是读写数据库（网络 I/O），PHP 线程都会停下来等待：此时并不消耗 CPU 资源，但是 TCP 连接和线程都还在持续等待，所以如果这个请求不结束，那该线程将会一直保持运行，持续消耗着 TCP 连接数资源和内存资源。

由于 PHP 拥有单线程阻塞特性，所以 php-fpm 模式和 Apache 的 mod\_php 模式在解释执行 PHP 代码时的性能是一模一样的。在 2 vCore 4G 内存的情况下，200 QPS 的性能极限是无法通过把 Apache 换成 php-fpm 来解决的。它们的主要区别还是在“海量 HTTP 连接的处理能力”上，我们将在后面第 4 章“至关重要的 Web Server 软件”中做详细的阐述。

那 PHP 这种单线程阻塞语言的性能瓶颈应该怎么突破呢？Node.js 登场了。

### 以 Node.js 为代表的非阻塞 I/O

在 PHP 这样的阻塞式语言中，所有的 I/O 操作都是需要停下来等待的，例如磁盘 I/O，数据库网络 I/O 等，而真正用于计算的 CPU 资源反而大多数时候都在浪费：大部分 API 都不存在复杂的数据转换，时间其实主要花在了和各种数据库的通信上。这个世界上绝大多数语言都是阻塞式运行的，因为这样做虽然性能不高，但却最符合人类大脑的习惯，编码也更加容易。在 Go 语言出现前的时代，高并发问题大多是用多核+多进程/多线程来解决的。

Ryan Dahl 敏锐地发现了 I/O 浪费时间这个问题，并且挑选了一个为浏览器创造的单线程语言 JavaScript

来实现他的抱负：将所有 I/O 操作全部异步化，并利用 js 的单线程排队特性，创造了一种高性能且稳定的后端技术——Node.js。

不过，计算机的世界没有银弹，Node.js 虽然 I/O 性能强，但是代码编写起来却更加地困难：开发者需要额外付出一些异步编程的思考时间，Debug 也更加麻烦。

Node.js 是一种非常神奇的单线程异步非阻塞架构，以 Google V8 引擎作为 JavaScript 解释器，利用事件驱动加非阻塞 I/O 技术，大幅提升了单机能够处理的 QPS 极限，而它“只是完整利用了单核 CPU”而已。

此外，Node.js 还具备一个 Nginx 的优势：可以单机处理海量用户的 TCP 连接。

Node.js 可以完整利用单核 CPU 了，那现在的服务器 CPU 已经做到了单颗 192 核 384 线程，该如何利用这么多的 CPU 核心呢？该 Go 语言登场了。

## 以 Go 语言为代表的协程

为了更好地“直接利用全部 CPU”，Java 诞生了线程池技术，至今还在发光发热。而 Go 选择釜底抽薪：在语言层面打造一个完善的“超并发”工具：Goroutine（协程）。

笔者之所以将 Goroutine 称为“超并发”工具，是因为它是语言层面提供的一个[线程池+协程](#)的综合解决方案，并使用 Channel 管道思想来传递数据，为用户提供了一个无需手动管理的高性能“并发控制运行时”（Concurrency Control Runtime），可以保证榨干所有 CPU 核心的每一个时间片。

Go 的协程从技术原理上讲就是“在一个线程内不断地 goto”，就像 DPDK 通过完全在用户态运行而避免了上下文切换从而大幅提升了网络性能一样，Go 在线程内主动 goto 也可以轻松将 CPU 利用率顶到 100%，实现硬件资源利用的最大化。

当然，“不断地 goto”只是一种形象的类比方法，实际上 Golang 的协程技术经历了好几次迭代，具体实现大家可以看“灯塔” draveness 的书：[《Go 语言设计与实现》](#)。

此外，“吃完多核服务器上的每一个 CPU 核心”也是各种新形态 MySQL 兼容数据库的主要价值，这个我们在后面章节中讨论数据库架构时再进行详细地分析。

## Goroutine 的弱点

就像性能优化的核心是空间换时间、时间换空间一样，Goroutine 也不是银弹，也是牺牲了一些东西的。根据笔者的实践，这个东西就是“极其昂贵的内存同步开销”，而且 Goroutine 引发的这个问题比 Java 的线程池内存同步问题严重的多。

一旦你想在单个 Go 进程内部的海量协程之间做“数据同步”，那你面临的就不只是 CPU 资源浪费那么简单了，你会发现，CPU 依然吃完了，但是并发量还是好低：多线程的内存同步难题已经摧毁了无数 Java 程序员的头发，而 Goroutine “线程 x 协程”数量的内存同步堪称灾难，如果你用过[sync.Map](#)，笔者相信你一定有切身的体会。

那如果我们就是需要在海量协程之间做实时数据同步该怎么办呢？这个时候，高并发哲学思维又要出动了：找出单点，进行拆分！

等等，好像除了唯一的这个 Go 进程找不出单点啊？

没错，这个唯一的 go 进程申请的这段内存就是单点，想解决这个问题需要出大招：找外援。

## Redis 是 Go 协程最亲密的伙伴

就像 MySQL 之于 PHP，MongoDB 之于 Node.js，Redis 就是 Go 协程最亲密的伙伴，是 Go 的最佳拍档。

网络栈是一种贯彻了 Linux 一切皆文件思维的优秀工具，此时可以帮上大忙：找另一个单线程性能之王 Redis 打辅助，就可以帮助海量协程通过排队的方法解决问题：此时一旦某个协程进入网络 I/O 状态，则会立即让出 CPU 时间片，将当前 CPU 核心的指令指针 goto 到下一个协程，不浪费 CPU 资源。

当然，理论上说你也可以选择自己用 Go 写一个类似 Redis 的单线程内存数据库，和你的业务进程进行网络通信，一样可以解决这个问题。

而一旦我们解决了协程之间内存同步的大问题，Go 就可以胡吃海塞，大杀四方，分分钟榨干 192 颗 CPU 核心。

## Node.js 的“多线程”

一些使用 Node.js 的读者可能不同意前面“Node.js 是单线程”的观点，确实，自 V12 开始支持的 worker\_threads 让 Node.js 拥有了一种事实上的“多线程”能力，其运行架构如图 3-2 所示。

但是，需要明确的是，在逻辑上，整个 Node.js 依然是一个线程安全的单线程逻辑处理器，worker\_threads 的出现是为了在 CPU 密集型的业务场景中，利用多核 CPU 来进行并行计算，但不适用于 I/O 密集领域。Node.js 在 I/O 领域的单线程、线程安全及事件驱动特性从未发生任何改变。如果你在 I/O 密集型的业务中使用 worker\_threads，可能会得到总执行时间增加的反向效果。



图 3-2 worker\_threads 运行架构

## Java 在语言设计层面的优势

Java 在软件工程层面的优势我们前面已经说过，其实，Java 能有今天的成功，它在语言设计层面也一定是两把刷子的。

Java 不只是一个编程语言，更是一整套的基于运行时虚拟机技术的解决方案。总体来看，它选择了“空间换时间”：Java 应用对内存的需求量显著超过其它技术，而经过了这么多年的优化，Java 的“时间性能”在绝大多数场景下都已经做到了无限接近 C++ 的水平。

Java 虽然是虚拟机技术，但它是常驻内存的，并且这个技术非常的灵活。对，你没有看错，Java 技术其实非常灵活。Spring 框架对写业务代码的程序员有强约束，这是对使用者的繁琐，Java 语言本身的灵活性是非常高的，他提供了各种各样高级的特性让开发者使用。

这么多年过去，Java 一直都能不断地跟上时代：JDBC、RMI、反射、JIT、数字签名、JWS、断言、链式异常、泛型、注解、lambda、类型推断等等等等。我们知道，传统的 Java 大多采用多线程来实现并行，但是在去年（2022）它甚至发展出了协程 Fiber！

21 世纪的头十年，JVM 在很多公司内都变成了代替虚拟机技术的存在，成为了事实上的“标准服务端运行环境”，以至于诞生了 JPython、JRuby、JPHP 等颇具邪典气质的技术：把动态语言的解释器内置到 JVM 内，再把代码和解释器打包成一个 jar 包或者 war 包，在标准 JVM 中直接部署 Python、Ruby、PHP 语言开发的软件。

这个思想怎么看起来有点眼熟呢？这不就是容器技术吗！

## JVM 优秀的设计哲学

JVM，全称 Java Virtual Machine，中文名为 Java 虚拟机，是 Java 平台的核心组件。它负责执行 Java 字节码，将字节码翻译成底层操作系统可以识别的机器指令。

我们平时说的 Java 语言的特点，绝大多数在本质上都是 JVM 技术的特点。

正是因为 JVM 屏蔽了具体操作系统平台相关的信息，抹平了各种操作系统之间的差异，Java 程序才拥有了“一次编写，到处运行”的能力。

JVM 的设计哲学主要体现在以下几个方面：

1. 平台无关性：JVM 的目标是实现一次编写，到处运行的特性，即编写的 Java 程序可以在任何安装了 JVM 的平台上运行，无需针对特定平台进行修改。这大大提高了软件的可移植性和灵活性，简化了开发和维护的工作量。
2. 内存管理：JVM 负责管理 Java 程序的内存，包括堆内存和栈内存。堆内存用于存储对象实例，栈内存用于存储方法调用。这种内存管理方式使得 Java 程序员无需关心内存分配和回收，可以专注于业务逻辑的开发。同时，JVM 提供了自动垃圾回收机制，能够自动回收不再使用的内存，避免了内存泄漏等问题。
3. 安全性：JVM 提供了一些安全机制，如类加载机制、字节码验证等，以防止恶意代码对系统造成破坏。这些安全机制确保了 Java 程序在运行时的安全性和稳定性。

- 多线程：JVM 支持多线程编程，能够充分利用多核处理器的性能。多线程技术使得 Java 程序能够同时处理多个任务，提高了程序的并发性和响应速度。
- 性能优化：JVM 提供了一些性能优化工具，如即时编译器（JIT），能够将热点代码编译成机器码，提高程序的运行效率。这些性能优化工具使得 Java 程序在运行时能够达到更高的性能水平，满足用户对软件性能的要求。
- 跨平台开发：JVM 的设计使得 Java 程序可以在不同操作系统和硬件平台上运行，实现了真正的跨平台开发。开发者可以在 Windows 和 macOS 系统上开发 Java 应用，并在 Linux 上运行，这大幅提高了开发效率。

## 3.5 实战：利用 Go 语言的协程开发高性能爬虫

Go 语言的协程机制使其成为开发高性能爬虫的理想选择。通过利用外部 Redis 进行“协程间通信”，无论有多少 CPU 核心，Go 都能够充分利用它们。此外，编写 Go 代码也非常简单，无需自己管理进程和线程。然而，由于协程功能强大且代码简洁，调试成本较高：在编写协程代码时，笔者感觉自己像在炼丹，修改一个字符就能让程序从龟速提升到十万倍，简直比操控 ChatGPT 还神奇。

### 遵守法律法规和 robots.txt 业界规范

在编写爬虫之前，我们需要明确了解以下内容：从互联网上爬取内容需要遵守法律法规，并遵循 [robots.txt](#) 业界规范。关于 robots.txt 的具体规范内容，大家可以自行搜索相关资料。

### 笔者的开源项目

笔者开源了一个 Go 语言编写的开源互联网搜索引擎 [DIYSearchEngine](#)，遇到问题的读者可以参考我的代码。

### 爬虫工作流程

我们先设计一个可以落地的爬虫工作流程。

#### 1. 设计一个 User-Agent (UA)

首先，我们需要为我们的爬虫设置一个 User-Agent。为了提高爬虫的成功率，我们可以选择较新的 PC 浏览器的 UA，并对其进行改造，以加入我们自己的项目名称。在笔者的项目中，项目名为“Enterprise Search Engine”，简称 ESE，因此笔者设定的 UA 是 `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/97.0.4280.67 Safari/537.36 ESESpider/1.0`。你可以根据自己项目的需求进行相应的设定。

需要注意的是，有些网站会屏蔽非头部搜索引擎的爬虫，读者需要找到允许普通爬虫爬取的网站。

#### 2. 选择一个爬虫工具库

笔者选择的爬虫工具库是 [PuerkitoBio/goquery](#)。它支持自定义 UA 爬取，并且可以对爬取到的 HTML 页面进行解析，从而获取非常重要的页面标题、页面中包含的超链接等信息。

### 3. 设计数据库

爬虫的数据库设计相对简单，只需要一个表即可。这个表中存储着页面的 URL、爬取到的标题以及网页的文字内容，具体的字段定义可以参考代码清单 3-1。

代码清单 3-1 页面数据表结构

```
CREATE TABLE `pages` (  
  `id` int unsigned NOT NULL AUTO_INCREMENT,  
  `url` varchar(768) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '网页链接',  
  `host` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '域名',  
  `dic_done` tinyint DEFAULT '0' COMMENT '已拆分进词典',  
  `crawl_done` tinyint NOT NULL DEFAULT '0' COMMENT '已爬',  
  `crawl_time` timestamp NOT NULL DEFAULT '2001-01-01 00:00:00' COMMENT '爬取时刻',  
  `origin_title` varchar(2000) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '级页面超链接文字',  
  `referrer_id` int NOT NULL DEFAULT '0' COMMENT '上级页面ID',  
  `scheme` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT 'http/ht',  
  `domain1` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '一级域后缀',  
  `domain2` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '二级域后缀',  
  `path` varchar(2000) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT 'URL 路径',  
  `query` varchar(2000) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT 'URL 查询数',  
  `title` varchar(1000) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '页面标题',  
  `text` longtext CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci COMMENT '页面文字',  
  `created_at` timestamp NOT NULL DEFAULT '2001-01-01 08:00:00' COMMENT '插入时间',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

### 4. 星星之火

爬虫具有一个非常强大的特性：自我增殖。由于每个网页通常都包含其他网页的链接，这使得我们可以通过这种方式实现无限扩展。为了启动爬虫，我们需要选择一个导航网站，并将该网站的链接手动添加到数据库中。读者可以根据个人喜好选择导航网站。笔者选择的是 hao123，此时我们需要将选定网站的首页链接插入 pages 表。

### 5. 运行爬虫

现在，让我们进入实际操作阶段。我们使用递归的思想实现自我增值，基本流程如下：

1. 从数据库中读出一个没有爬过的页面链接
2. 使用 curl 工具类获取网页文本

3. 解析网页文本，提取出标题和页面中含有的超链接
4. 将标题、一级域名后缀、URL 路径、插入时间等信息补充完全，更新到这一行数据上
5. 将页面上的获取到的心得超链接插入 pages 表，第一次自我增殖完成！

爬虫代码可以参考代码清单 3-2，注释中解释了代码的逻辑。

代码清单 3-2 可以循环运行的爬虫伪代码

```

func main() {
    fmt.Println("My name is enterprise-search-engine!")

    // 加载 .env
    initENV() // 该函数的具体实现可以参考项目代码

    // 开始爬
    nextStep(time.Now())

    // 阻塞，不跑爬虫时用于阻塞主线程
    select {}
}

// 循环爬
func nextStep(startTime time.Time) {
    // 初始化 gorm 数据库
    dsn0 := os.Getenv("DB_USERNAME0") + ":" +
        os.Getenv("DB_PASSWORD0") + "@(" +
        os.Getenv("DB_HOST0") + ":" +
        os.Getenv("DB_PORT0") + ")/" +
        os.Getenv("DB_DATABASE0") + "?charset=utf8mb4&parseTime=True&loc=Local"
    gormConfig := gorm.Config{}
    db0, _ := gorm.Open(mysql.Open(dsn0), &gormConfig)

    // 从数据库里取出本轮需要爬的 100 条 URL
    var pagesArray []models.Page
    db0.Table("pages").
        Where("craw_done", 0).
        Order("id").Limit(100).Find(&pagesArray)

    tools.DD(pagesArray) // 打印结果

    // 限于篇幅，下面用文字描述
    1. 循环展开 pagesArray
    2. 针对每一个 page，使用 curl 工具类获取网页文本
    3. 解析网页文本，提取出标题和页面中含有的超链接
    4. 将标题、一级域名后缀、URL 路径、插入时间等信息补充完全，更新到这一行数据上
    5. 将页面上的超链接插入 pages 表，我们的网页库第一次扩充了！

    fmt.Println("跑完一轮", time.Now().Unix()-startTime.Unix(), "秒")

    nextStep(time.Now()) // 紧接着跑下一条
}

```

当我们执行 `go build -o ese *.go && ./ese` 命令之后，得到的输出结果如代码清单 3-3 所示。

代码清单 3-3 爬虫首次运行结果

```
My name is enterprise-search-engine!
```

```
加载.env : /root/enterprise-search-engine/.env
```

```
APP_ENV: local
```

```
[[{1 0 https://www.hao123.com 0 0 2001-01-01 00:00:00 +0800 CST 2001-01-01 08:00:00 +0800 CST 0  
-01-01 00:00:00 +0000 UTC}]]
```

---

## 6. 合法合规：遵循 robots.txt 规范

为了确保爬虫的合法性和合规性，笔者选择了使用 `temoto/robotstxt` 库来检查我们的爬虫是否被允许爬取某个 URL。具体做法是，我们使用一张单独的表来存储每个域名的 robots 规则，并在 Redis 中建立缓存。每次在爬取 URL 之前，我们会先进行一次匹配操作，只有匹配成功的情况下才会进行爬取，以确保我们的爬虫行为符合业界规范的要求。

## 基础知识储备：Goroutine 协程

在开始之前，笔者假设你已经对 Go 协程有一定的了解。Go 协程是一种令人惊叹的技术，写起来特别像魔法。为了更好地理解协程，笔者想分享一个小技巧：当一个协程进入磁盘、网络等需要后台等待的任务时，它会将当前 CPU 核心（可以将其视为一个图灵机）的指令指针跳转到下一个协程的起始指令所在的指针位置。

需要注意的是，协程是一种特殊的并发形式。在并发函数中调用的函数必须都支持并发调用，类似于传统的“线程安全”，笔者称其为“协程安全”代码。如果你不小心编写了“协程不安全”的代码，可能会导致程序卡顿甚至崩溃。

## 使用协程并发爬取网页

为了利用多核 CPU 的全部计算资源，我们会一次取出一批需要爬的 URL，并使用协程并发爬取。

```

// tools.DD(pagesArray) // 打印结果

// 创建 channel 数组
chs := make([]chan int, len(pagesArray))
// 展开 pagesArray 数组
for k, v := range pagesArray {
// 存储 channel 指针
chs[k] = make(chan int)
// 启动协程
go craw(v, chs[k], k)
}

// 注意, 下面的代码不可省略, 否则你上面 go 出来的那些协程会瞬间退出
var results = make(map[int]int)
for _, ch := range chs {
// 神之一手, 收集来自协程的返回数据, 并 hold 主线程不瞬间退出
r := <-ch

_, prs := results[r]
if prs {
results[r] += 1
} else {
results[r] = 1
}
}
// 当代码执行到这里的时候, 说明所有的协程都已经返回数据了

fmt.Println("跑完一轮", time.Now().Unix()-startTime.Unix(), "秒")

```

我们的爬取函数 `craw()` 也需要进行协程化:

```

// 开始爬取, 存储标题, 内容, 以及子链接
func craw(status models.Page, ch chan int, index int) {
// 调用 CURL 工具类爬到网页
doc, chVal := tools.Curl(status, ch)

// 对 doc 的处理在这里省略

// 最重要的一步, 向 channel 发送 int 值, 该动作是协程结束的标志
ch <- chVal
return
}

```

## 真实的爬虫运行架构图

生产环境中的爬虫由于需要爬取数以亿计的网页, 其运行架构是非常复杂的, 笔者的开源项目 [DIYSearchEngine](#) 真实的爬虫架构如图 3-3 所示。

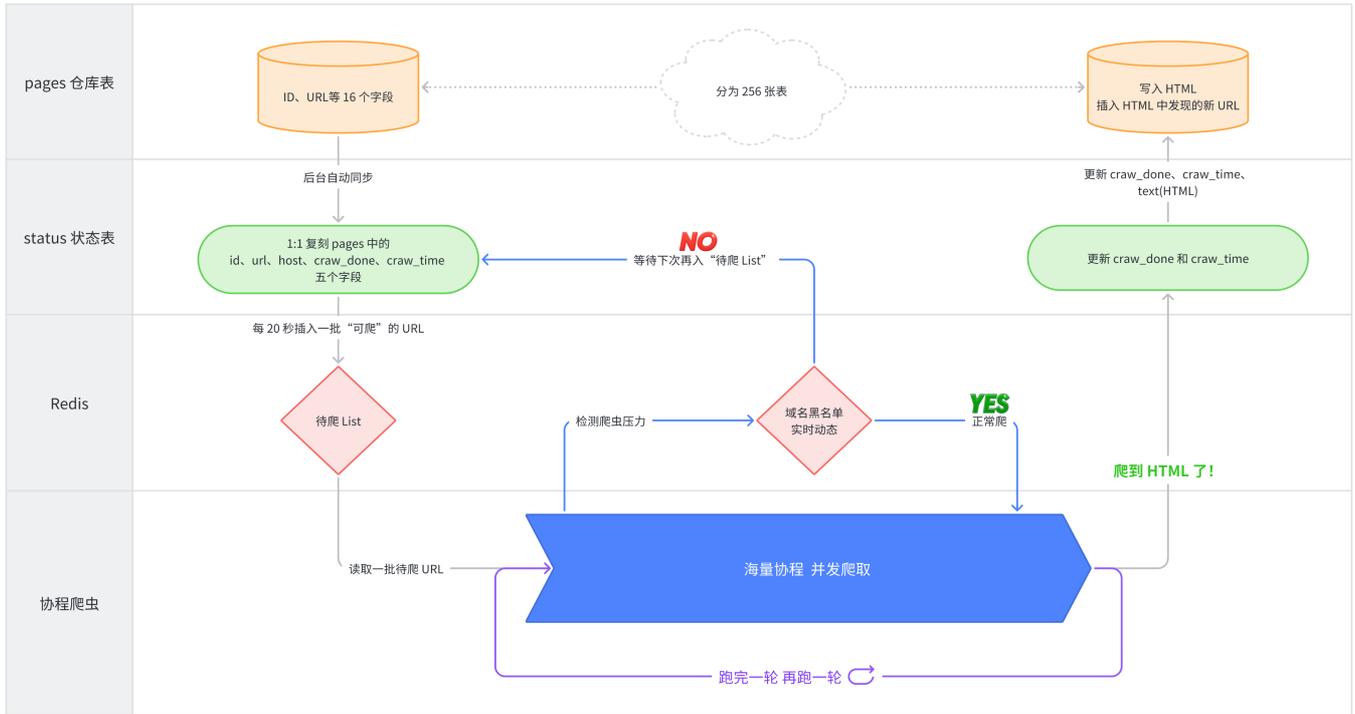


图 3-3 DIYSearchEngine 真实运行架构图

### 3.6 面试题

**No.08 : Node.js 是如何在单线程内实现非阻塞 I/O 的？**

Node.js 借助 libuv 的能力，利用队列和线程池实现了非阻塞 I/O。

首先 Google V8 引擎负责单线程解释执行 JavaScript 代码，在遇到网络或者磁盘 I/O 请求后，运行模式如下：

1. 这个 I/O 请求会被入队列，同时注册一个回调函数，等待 I/O 结果返回以后的回调函数调用
2. 让出主线程，让 V8 继续执行其他 js 代码
3. libuv 开始处理队列任务
  1. windows 系统下，通过 IOCP 组件执行真正的异步 I/O（其实 IOCP 在内核内部依然是线程池实现的）
  2. 由于 Linux 下的 socket 只能阻塞运行，所以是采用的用户态线程池技术 + epoll 实现的非阻塞 I/O
4. 某个队列任务拿到返回结果后，开始在 V8 主线程排队等待执行回调函数
5. 排到自己以后，继续执行该请求的后续代码

Node.js 的核心技术有两个：V8 提供的天然单线程排队执行机制，和 libuv 提供的跨越 Linux、Windows、macOS 三个系统的非阻塞 I/O 能力。

**No.09 : Go 语言为什么快？**

Go 语言之所以快，主要是因为其并发模型、垃圾回收、编译优化、内存管理和标准库等方面都是完全面向 “

高性能网络编程”这个需求设计的。

Go 在语言 runtime 层面实现了完善的用户态协程调度，并且将以 epoll 为代表的 I/O 多路复用贯彻到了网络的方方面面，编码难度低，很容易就能写出高性能的代码。具体来说，Go 语言在设计上有如下几个优势：

1. 并发模型：基于 CSP（Communicating Sequential Processes）理论，使用 goroutines 和 channels 达成了高并发任务的高效率调度和处理。
2. 垃圾回收：并发垃圾回收机制，提高垃圾回收效率，使用写屏障技术减少锁使用，提高程序性能。
3. 编译优化：编译器在编译阶段进行多种优化操作，减少资源消耗，提高运行速度。
4. 内存管理：自动内存管理，减轻程序员负担，减少内存泄漏等问题。
5. 标准库：丰富的标准库提供高度优化的函数和类型，提高开发效率，保证程序性能。

#### No.10 : Goroutine 是如何实现高性能的？

传统的进程/线程之所以慢，是因为当某个 CPU 核心这个图灵机从执行一个线程切换为执行下一个线程的时候，由于会出现用户态->内核态->用户态的切换（即上下文切换），导致了对内存的读写。Go 在语言 runtime 层面实现了一个可以在用户态的单个线程的内部进行自主管理的协程调度机制，让一个协程的代码执行完之后，不需要进行上下文切换，无需读写内存，在操作系统看来，这个线程没有任何的变化，依然在按照顺序执行一个又一个指令。这个机制使得 CPU 时间片得到了最大程度的利用，是 Goroutine 高性能最大的贡献者。

详细地说，Goroutine 拥有如下这些专门的设计来达成超高性能：

1. 并发执行：Goroutine 可以在多个 CPU 核心上并发执行，充分利用多核处理器的性能。通过在操作系统层面进行调度，Goroutine 可以在不同的核上运行，从而实现真正的并行计算。
2. 栈空间：Goroutine 的栈空间非常小，通常只有 2KB 左右。这使得大量的 Goroutine 可以在内存中同时存在，而不会因为栈空间的分配和回收导致性能下降。
3. 上下文切换开销小：Goroutine 的上下文切换开销非常小，因为它们在同一个操作系统线程中运行。当一个 Goroutine 阻塞时，其他 Goroutine 可以继续执行，而不需要等待阻塞的 Goroutine 完成。这避免了传统线程模型中的忙等待问题，提高了程序的执行效率。
4. 简单易用：Goroutine 的使用非常简单，只需要在函数调用前加上关键字 `go` 即可创建一个 Goroutine。这使得开发者可以轻松地编写高并发的代码，而不需要关心复杂的线程同步和互斥问题。
5. 内置调度器：Go 语言内置了一个调度器（scheduler），负责对 Goroutine 进行调度和管理。调度器使用了一种称为 M:N 调度的技术，将多个 Goroutine 分配到多个操作系统线程上执行。这种调度策略可以在保证程序执行顺序的同时，最大限度地利用多核 CPU 的全部计算资源。

## 第三部分 网络资源高并发

第 4 章 至关重要的 Web Server 软件

第 5 章 负载均衡和应用网关

第 6 章 使用软件定义网络 (SDN) 技术搭建大规模负载均衡集群

### 第 4 章 至关重要的 Web Server 软件

Web Server 软件领域，除了独占 Windows 系统的微软 IIS，在主流的服务器操作系统 Linux 市场中，出现过一次非常明显的流行技术更替：从 Apache 切换到 Nginx，如图 4-1 所示。

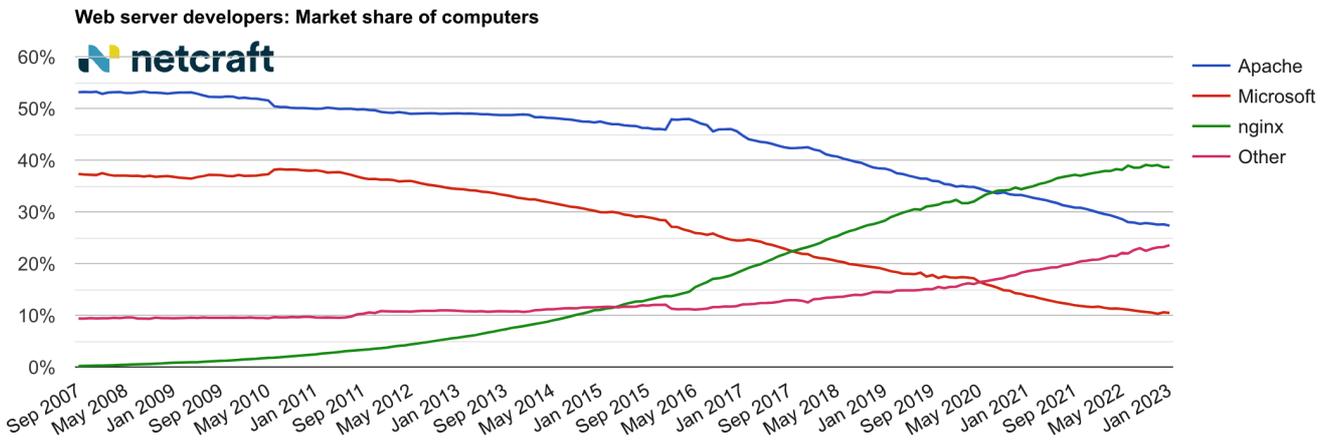


图 4-1 Apache 和 Nginx 的市场占有率变化

Apache 是一个广泛使用的开源 Web Server 软件，它在过去几十年中一直是 Linux 系统上的首选。然而，随着互联网的快速发展和高并发访问需求的增加，Apache 逐渐暴露出了一些问题，如处理大量并发连接时的性能瓶颈、内存占用过高等。

为了解决这些问题，Nginx 应运而生。Nginx 是一个轻量级的高性能 Web Server 和反向代理服务器，它并发处理能力高，内存占用低。Nginx 采用了事件驱动的异步非阻塞模型，可以同时处理大量的并发连接，能够高效地处理静态文件、动态内容和反向代理请求。由于 Nginx 的出色性能和稳定性，它已经成为目前最流行的 Web Server 软件。

## 4.1 Apache——最成功的开源软件之一

Apache HTTP Server 是一个可以和 Kernel、MySQL 并列的最成功的开源软件之一，它的历史就是万维网的历史，就是互联网的历史。

### 互联网的第一声啼哭

1993 年，淮河以南长江以北的丘陵地带有一个名为梨树的生产小队，小队里有一片长条形的池塘，在池塘畔的土坯房里，笔者出生了。同一年，在太平洋的另一头，美国伊利诺伊大学香槟分校内，美国国家超级电脑应用中心（NCSA）相继发布了三个软件：Mosaic 浏览器、CGI 协议、HTTPd Web Server 软件，互联网出生了。

1991 年的 Tim Berners-Lee 创造的那个胚胎，在孕育了两年之后，互联网呱呱坠地了。

### CGI 和 HTTPd 的创造者是一个年仅 20 岁的年轻小伙

当时，Robert McCool 是 NCSA 实验室的一名本科生，他在大二到大三期间成为了 HTTPd 和 CGI 协议的主要作者之一。然而，当他大三结束后，他离开了 NCSA 实验室，导致 HTTPd 失去了维护者。随后，一个由八人组成的小组希望继续开发 NCSA HTTPd，以维护 Web Server 的发展，他们自称为 Apache 小组。随后，该小组发布了从 NCSA HTTPd 衍生而来的 Apache 开源 Web Server。

1995 年 8 月时，几乎所有的网站都使用着 NCSA HTTPd。然而，到了 1996 年 4 月，Apache 就超过

了 NCSA HTTPd，成为市场占有率第一的 Web Server。这一地位保持了整整 20 年，直到 2016 年被 Nginx 超越。

那么，为什么 Robert McCool 要离开实验室呢？原来，他的学长、Mosaic 的主要作者之一 Marc Andreessen，于 1993 年 12 月本科毕业后创办了网景公司，并成功挖走了他。不仅如此，这位学长还挖走了 NCSA 互联网三件套几乎所有的核心开发者。

## Apache：驱动人类历史上大部分流量的开源软件

自浏览器诞生以来，Apache Web Server 与 Perl、PHP 两种主流编程语言紧密合作，传输了互联网上绝大多数的流量。即使在 Java 崭露头角的时代，Java 仍然与 Tomcat 和 Apache 配合多年，直到被 Nginx 超越。Apache 可以与几乎所有常见的后端编程语言配合使用，相较于 Nginx，它拥有更丰富的功能，并且稳定性极佳，远超过 Nginx + php-fpm 的组合。

这一切使得 Apache Web Server 成为了最成功的开源软件之一，为人类创造了巨大的价值。

然而，由于其陈旧的软件架构和对向前兼容性的需求，Apache 最终不得不因为性能较差而让出了第一名的位置。

## Apache 基金会——最成功的开源软件基金会

Apache 邮件小组在早期得到了 IBM 的大力支持，IBM 将 Apache HTTPd 内置到了自家的商用产品中，并为 Apache HTTPd 开源软件提供了资金和人力资源的支持。随后，在 2002 年，Apache Group 成立了一个专门用于孵化新的开源软件的项目 Apache Incubator。同年 11 月，该项目接收了第一个软件：Java 构建工具 ant。在接下来的几年里，Java 生态开源软件相继入驻，直到 2008 年 Hadoop 的到来，正式将 Apache 软件基金会推向了巅峰。

如今，Apache 已经孵化出了多个十分流行的开源软件，涵盖了从 Java 运行容器 Tomcat 到项目全生命周期管理工具 Maven，再到代码管理工具 Subversion (SVN)，以及日志工具类库 Log4j、Web 框架 struts、开源搜索引擎 Lucene、消息中间件 ActiveMQ、RocketMQ、JVM 编程语言 Groovy、多语言软件开发 IDE NetBeans、磁盘大数据处理框架 Hadoop、内存大数据处理框架 Spark、实时大数据处理框架 Flink、分布式消息队列 Kafka、分布式数据库 Cassandra、超大规模磁盘存储引擎 HBase、分布式服务框架 Dubbo、分布式中间件 Zookeeper 等等各种技术领域。这些软件都是其所在领域的绝对领导者，为全球范围内的互联网和软件企业节约了大量的研发成本。

## 4.2 Apache 和 Nginx 性能差异的原因

在深入了解了 Apache HTTP Server 以及 Apache 基金会的发展历程之后，让我们将目光重新聚焦到技术领域。作为一名后端开发者，我们或许都知晓 Nginx 相较于 Apache 具备更出色的性能表现，然而，为何 Nginx 更加强大，其优势究竟体现在何处，我们在网络上只能搜索到一些雷同的文章。

为了解答这个问题，笔者将尝试从互联网的发展、需求的演变以及技术进步的角度来解释这一问题。

## 互联网流量的发展

2000年6月，美国互联网月PV（Page View，页面浏览量）最高的网站是AOL，日均1336万。到了2004年4月，第一名雅虎的日均PV达到了1.886亿，四年时间增长到了14倍。又过了差不多三年，2007年1月的时候，谷歌排名第一，平均日PV为2.37亿，增长不到一倍，此时第二名是雅虎，它的日均PV甚至出现了8.6%的下降。

我们可以明显地观察到，2000年之后的四年中，网民的数量和活跃程度出现了暴增，而2004年之后的三年却只有小幅上涨，可想而知2004年的时候，Web Server软件的压力有多么大：谁能提升一倍的性能，谁就能节约一半的服务器成本，这无疑是一笔巨大的开支。

到了2021年5月，谷歌依然是美国第一大网站，日均PV为5.8亿，只是14年前的2007年的2.45倍，与过去相比，如今的互联网流量增长所带来的压力已经不再那么紧迫了。

## Kernel 的进步

2002年10月4日，Kernel 2.5.46发布，首次引入了事件驱动的异步I/O框架epoll<sup>[6]</sup>。这个功能的发布奠定了Linux在服务器领域的霸主地位，因为它显著提升了Linux在高负载下的I/O性能。那epoll是怎么做到高性能的呢？

当一个系统保持了n个网络连接的时候，传统的select(2)和poll(2)的复杂度都是O(n)，而Apache在对TCP连接和进程进行匹配的时候甚至能搞出O(n^2)的超高复杂度，而epoll的复杂度只有O(log n)。

在Kernel引入epoll几个月之前的2002年春天，在俄罗斯第二大网站Rambler工作的Igor Sysoev在公司的安排下开始开发新一代Web Server，尝试用更少的服务器支撑与日俱增的流量，并提升系统的扩展能力。很显然，他成功了。

## Nginx 横空出世

在epoll发布两周年的那天，Nginx正式开源了，从此HTTP/HTTPS流量的分发效率进入了一个新的时代。Nginx利用了Linux内核新引入的epoll API，大幅降低了海量TCP连接下的CPU负载，显著提升了单台服务器的TCP响应容量。

## 4.3 Nginx 与 epoll 的协同工作机制

众所周知，epoll是一种高性能的事件驱动的I/O多路复用机制。那么，相较于select这种原始的I/O多路复用机制，它具备哪些优势呢？简而言之：实现了从被动到主动的转变。

epoll化被动为主动，以前Apache的select模型需要两次遍历才能实现的网络数据包和线程的匹配，现在通过事件驱动的方式主动献上指针，性能暴增。这就像云原生时代的Prometheus监控：化主动上传为被动查询，大幅提升了单个数据收集节点的性能上限，成功解决了监控领域的高并发性能问题。

在5000个TCP连接的情况下，每收到一个数据包，Nginx找到对应线程的速度比Apache高了两个数量

级，即便是 event 模式下的 Apache，性能依然远低于 Nginx，因为 Nginx 就是专门为“反向代理”设计的，而 Apache 本质是个 Web 应用容器，无法做到纯粹的事件驱动，其性能自然无法与 Nginx 相提并论。

## epoll 的技术原理

笔者将从两个方面讲解 epoll 的技术原理，分别是网络数据包的处理流程，和 I/O 多路复用的实现方式。

### 1. 网络数据包的处理流程

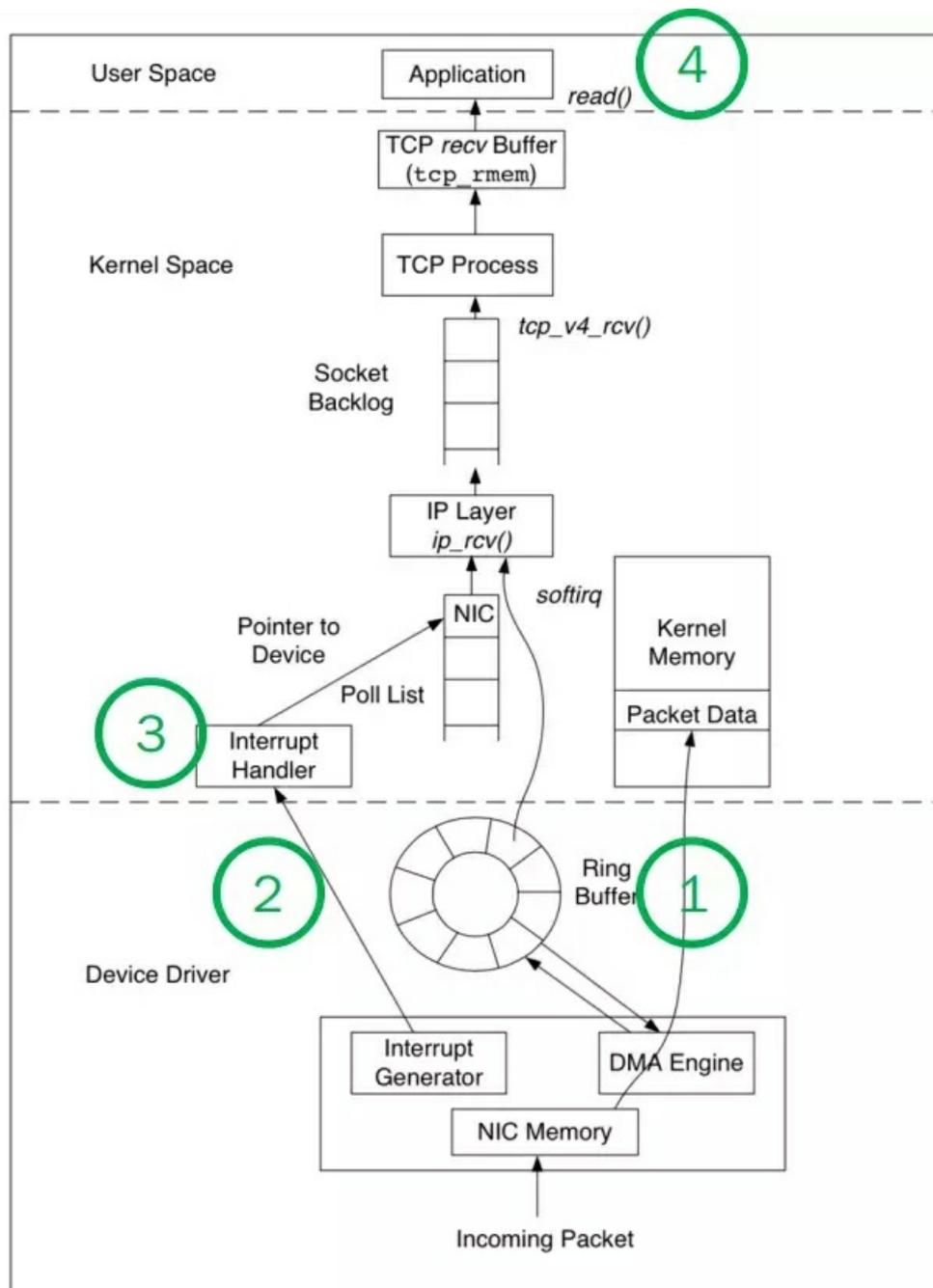


图 4-2 接收到网络数据包以后的处理流程图

图 4-2 是网卡接收到信息后，把信息发送给用户态的进程处理的流程图。这个过程可以分为四步来理解：

1. 网卡接收到一段数据，通过 DMA 方式写入内存
2. NIC 向 CPU 发出硬件中断请求，告诉内核有新的数据过来了
3. Linux 内核响应中断，系统切换为内核态，处理 Interrupt Handler，从 RingBuffer 拿出一个 Packet

，然后解析数据，找到这个端口对应的是哪个 PID，然后包装成 socket 发送给那个进程

4. 系统切换为用户态，用户进程处理内核传递过来的 socket 数据

## 2. I/O 多路复用的实现方式

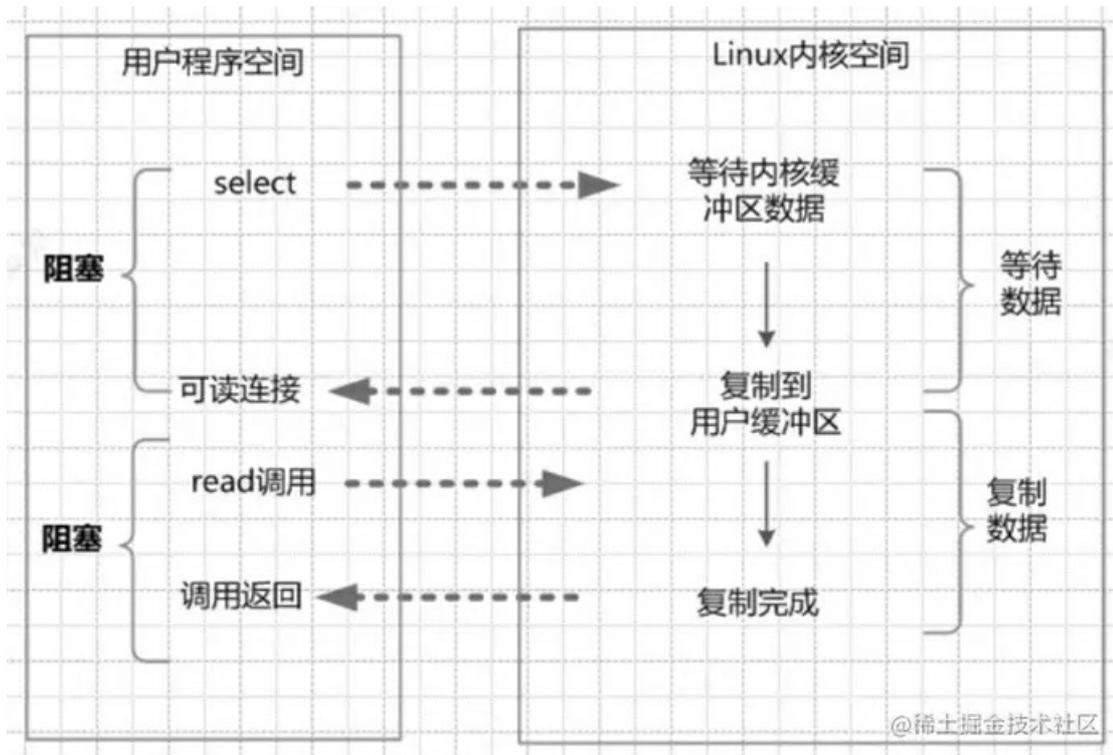


图 4-3 I/O 多路复用原理图

如图 4-3 所示。Nginx 在用户态中创建了一个线程池，池内的每个线程都需要负责处理多个 TCP 连接。通常情况下，这些线程处于休眠状态，不会占用 CPU 资源。当某个 TCP 连接接收到数据时，内核中的事件驱动机制会找到对应的休眠线程，主动调用相应的回调函数。

### epoll 真的是非阻塞的吗？

直接给出结论：不是。

首先，需要明确的是，在 Linux 系统下并不存在真正的非阻塞 I/O。

其次，客观上来说，网络连接是一个需要客户端和服务端共同确认的过程，并且由于光速的限制，信息的传递必然需要一定的时间。因此，单个网络连接的模型一定是“阻塞”的：即数据没有返回之前，需要进行等待。

其实我们的宇宙中根本不存在真正的非阻塞 I/O，因为我们的宇宙遵循因果律。

那 Windows IOCP 为什么号称是非阻塞的呢？IOCP 是 Windows 系统提供的一个系统级的非阻塞 I/O 运行库，这套 API 对于我们的调用者进程来说确实是非阻塞的，但实际上在内核层面它仍然是通过线程池实现的，只是这个线程池并没有留给我们自己来实现，而是由内核帮我们实现了而已。

因此，无论是在 Linux 下的 epoll，还是在 BSD 里的 Kqueue，或者是在 Windows 下的 IOCP，它们都只

是一种“I/O 多路复用”技术。它们只在“连接数为 10000，但同时收发数据的客户端只有 500 个”的场景下才有意义。换句话说，它们解决的是“明明还有 CPU 和内存资源，为什么客户端无法与服务器建立 TCP 连接”的问题。如果同时有 10000 台客户端同时进行数据收发，那么 epoll 和 select 之间并没有太大的性能差异。

## C10K 问题

Nginx 是第一个解决了 C10K 问题的 Web Server。

C10K 指的是单机维持一万个 TCP 连接，首次由 Dan Kegel 在 1999 年提出，最初的那个网页依然可以访问：<http://www.kegel.com/c10k.html>。

这个问题是怎么出现的呢？因为当初设计 Unix 操作系统的时候没有想到计算机软件的规模会发展到如此的大。就像人类已经经历过的千年虫问题，以及从现在开始 15 年后就要经历的 Unix 时间戳耗尽的问题，Unix 的进程 ID (PID) 以及单个进程最大打开文件数都被设定为了“有符号的 16 位整数”，即从 -32767 ~ 32767，所以一台机器在当时的理论极限只有 32767。

Nginx 在用户态利用事件驱动机制，让自己的一个线程可以根据需要绑定多个 TCP 连接，通过这种方式大幅增加了单台服务器所能保持的 TCP 连接数，解决了 C10K 问题。

## 4.4 测试三种 Apache 进程模型的技术特点

在掌握了上述知识的基础上，我们将以 Apache 作为实验对象，探究它的三种运行方式在高并发场景下的差异，从而推导出 Web Server 领域高并发优化的内功心法。

### Apache 的原始并发模型

Apache 支持三种进程模型：`prefork`、`worker` 和 `event`。在此，我们简要分析这三种模式的优缺点。

1. `prefork` 进程模式，内存消耗较大。每当接收到新的数据时，需要使用 `select` 模型遍历 `TCP连接数 × 进程数` 次才能找到匹配的进程。在单机数千个 TCP 连接数的场景中，仅寻找进程操作就消耗了一颗 CPU 核心 100% 的时间片，导致单机性能达到极限，无法充分利用更多的 CPU 资源。
2. `worker` 线程模式，同样使用 `select` 模型来遍历 TCP 请求和线程。其性能上限与 `prefork` 相同，但内存消耗有所降低，初始 TCP 承载能力略好。然而，在请求数突然增加的场景下，`worker` 模式开启新线程的速度反而比 `prefork` 更慢，且基础延迟也比 `prefork` 模式高，在大部分场景下不如 `prefork`。
3. `event` 模式采用与 Nginx 相同的 `epoll` 模型承载，理论上性能与 Nginx 相当。但由于 Apache 通常与 `mod_php`（插件）模式的 PHP 一起部署，再加上 PHP 阻塞运行的特性，其性能与前两种模式并无显著差异。因此，即使在 `event` 模式下运行的 Apache，其性能仍然远低于 Nginx 和 `php-fpm` 的组合。

接下来，我们将使用 `jmeter` 对 `prefork`、`worker`、`event` 三种模式进行性能测试，并额外验证几个关于 Nginx 和 `php-fpm` 的悬而未决的问题。

### 测试环境

#### 客户端

- i5-10400 6 核 12 线程
- 32GB 内存
- 千兆有线网络
- 软件环境
  - macOS
  - Java 19.0.1

## 服务端

- 物理服务器 E5-2682V4 2.5GHz 16 核 32 线程 \* 2 (阿里云 5 代 ECS 同款 CPU) 256GB RAM
- 虚拟机 64 vCPU (将物理机全部的 CPU 资源都赋予了虚拟机)
- 虚拟机内存 32GB
- 软件环境
  - CentOS Stream release 9
  - kernel 5.14.0-200.el9.x86\_64
  - Apache/2.4.53
  - Nginx/1.20.1
  - PHP 8.0.26
- PHP 环境:
  - Laravel 9.19
  - 给默认路由增加 sleep 500ms 的代码, 模拟数据库、Redis、RPC、cURL 微服务等场景
  - 执行 `php artisan optimize` 后测试

## 相关代码及配置

测试代码如代码清单 4-1 所示。

代码清单 4-1 测试用的 PHP 代码

```
Route::get('/', function () {  
    usleep(500000);  
    return view('welcome');  
});
```

Apache prefork 模式的配置文件如代码清单 4-2 所示。

代码清单 4-2 prefork 模式下的 Apache 配置文件

```
<IfModule mpm_prefork_module>
    StartServers      100
    MinSpareServers   5
    MaxSpareServers   100
    MaxRequestWorkers 500
    MaxRequestsPerChild 100000
</IfModule>
```

php-fpm 的配置文件如代码清单 4-3 所示。

代码清单 4-3 prefork 模式下的 php-fpm 配置文件

```
pm = static
pm.max_children = 500
```

## 实验设计

我们将测试三种配置下的性能表现差异：

1. Apache 标准模式：prefork + mod\_php 插件式运行 PHP
2. Nginx + php-fpm 专用解释器
3. Nginx 作为 HTTP 反向代理服务器，将 HTTP 请求转发给 Apache（采用 prefork 模式 + mod\_php 插件式运行 PHP）

## 请求计划

1. 客户端新线程开启后，每隔 5 秒发送一个请求
2. jmeter 用 50 秒开 5000 个线程，持续压测 100 秒，最大请求 QPS 为 1000

## 为什么这么设计？

单独对比 Nginx 和 Apache 性能的文章很多，数据结果也大同小异，无非是 Nginx 的 QPS 更高，但是“为什么 QPS 更高？”却没人回答，本次的实验设计就是要回答这个问题。

## 标准模式：prefork + mod\_php

prefork + mod\_php 模式下的测试结果如图 4-4 和 图 4-5 所示。

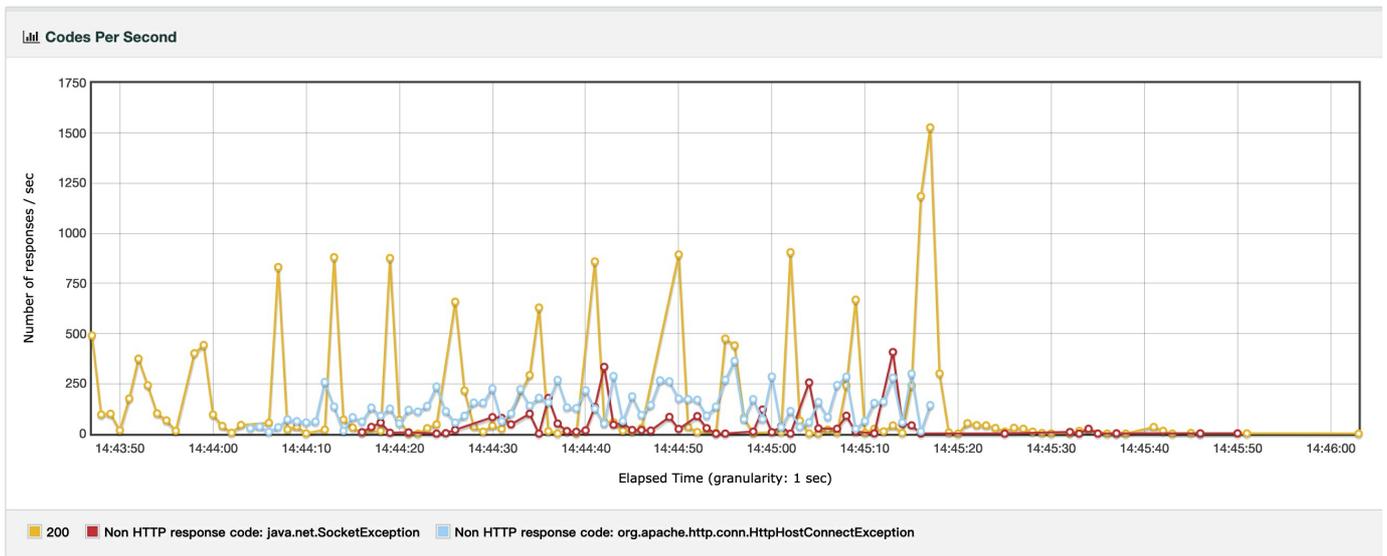


图 4-4 prefork + mod\_php 模式下的 QPS 分布

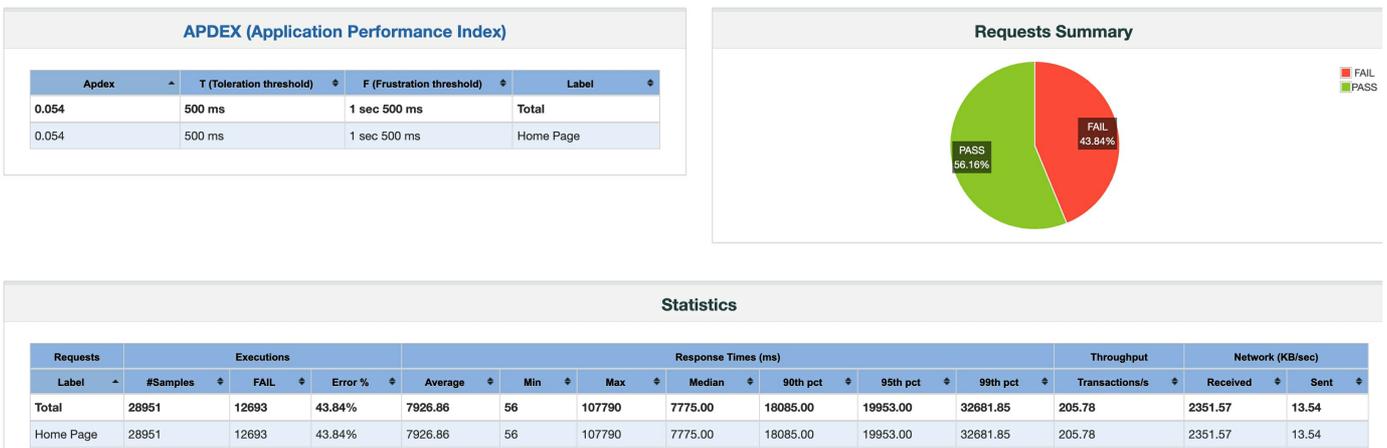


图 4-5 prefork + mod\_php 模式下的统计信息

高性能模式：Nginx + php-fpm

Nginx + php-fpm 模式下的测试结果如图 4-6 和 图 4-7 所示。

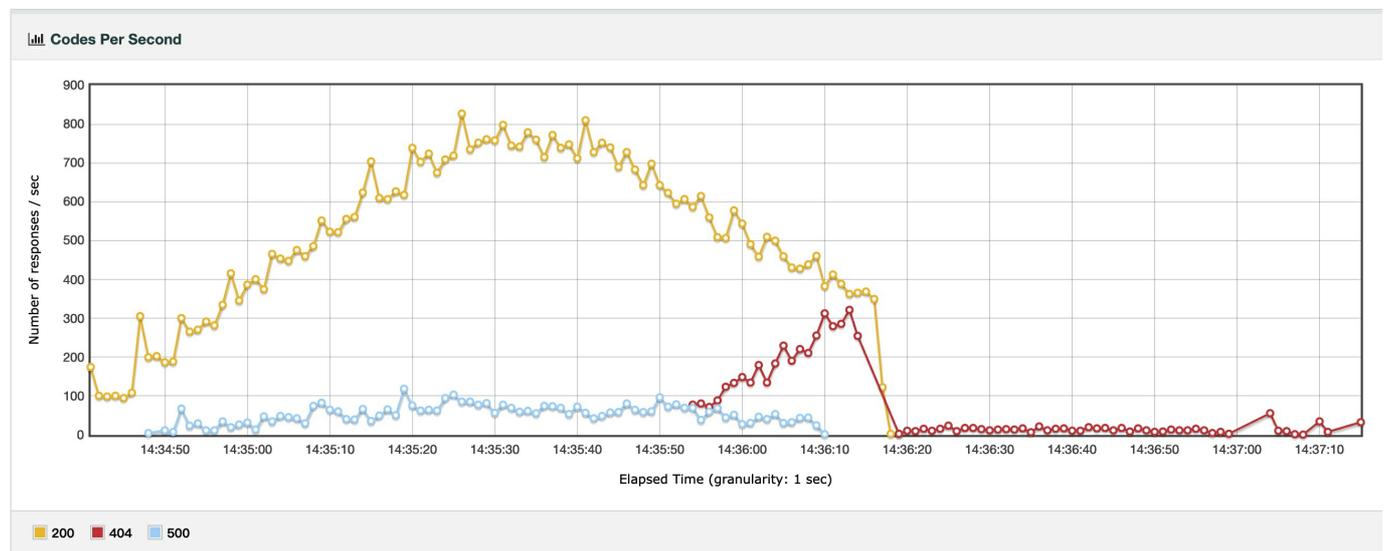


图 4-6 Nginx + php-fpm 模式下的 QPS 分布

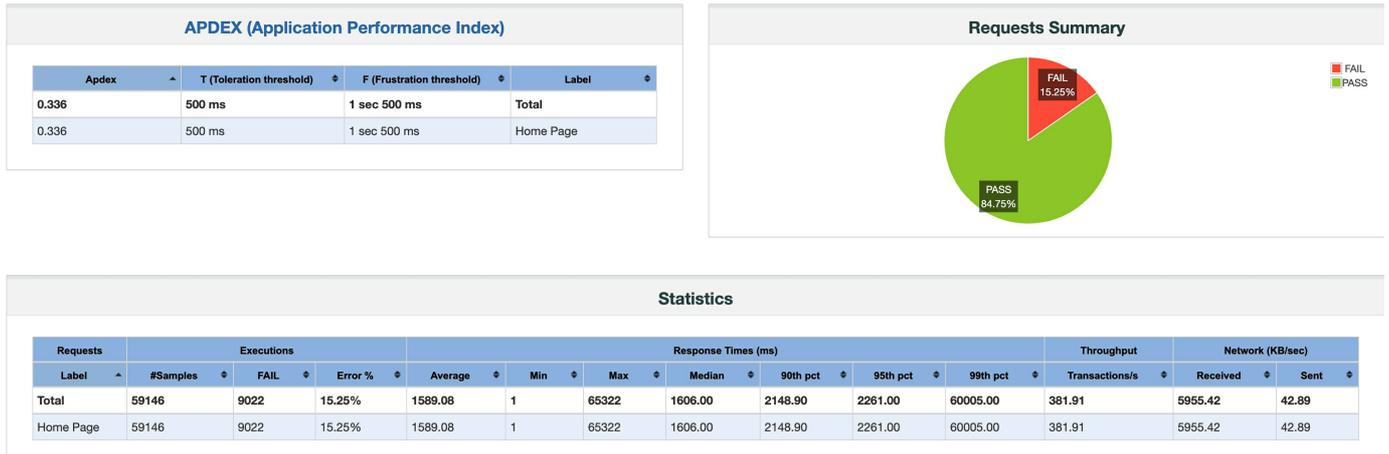


图 4-7 Nginx + php-fpm 模式下的统计信息

## Nginx 反向代理 Apache 模式

Nginx 反向代理 Apache 模式下的测试结果如图 4-8 和 图 4-9 所示。

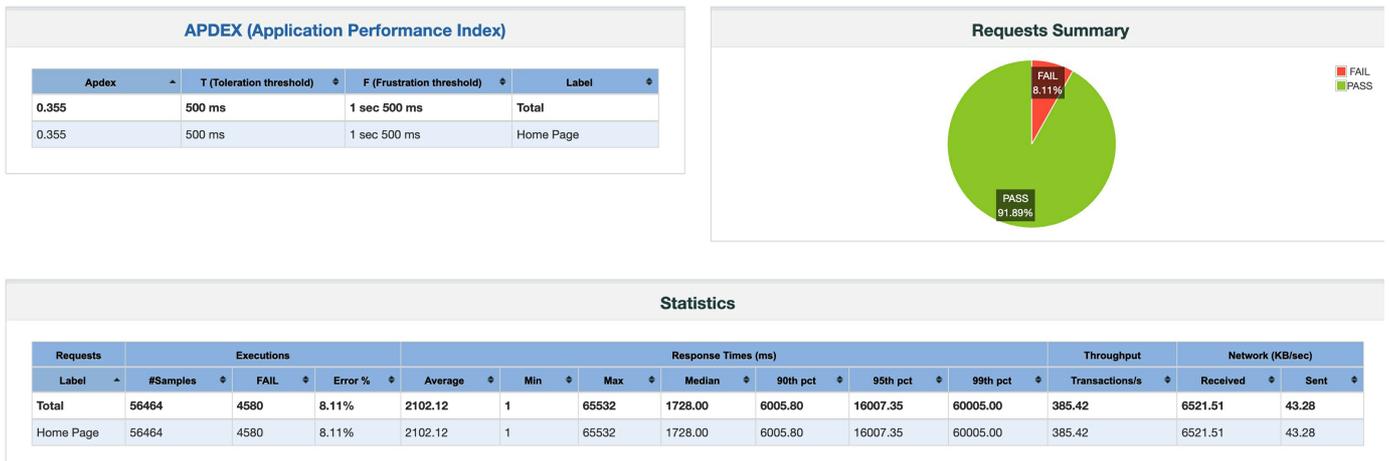


图 4-8 Nginx 反向代理 Apache 模式下的 QPS 分布

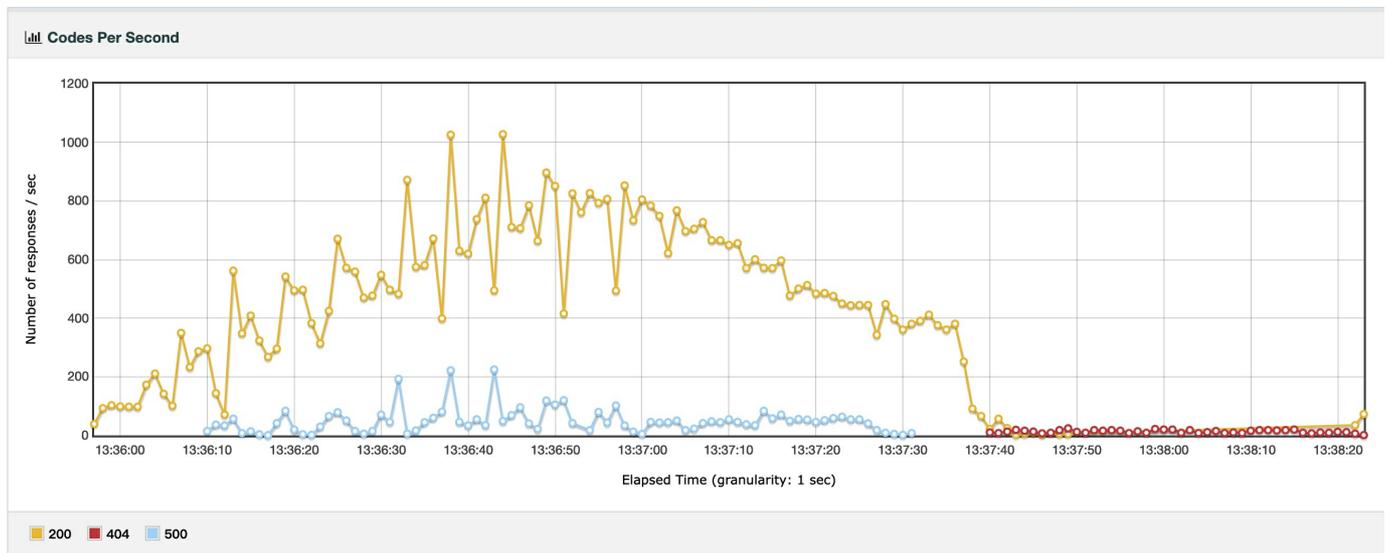


图 4-9 Nginx 反向代理 Apache 模式下的统计信息

## 结果分析

我们可以很明显的看出，Apache + prefork 的问题在于它对数千个 TCP 连接的处理能力不足。

1. Nginx + fpm 一共发出了 59146 个请求，成功了五万个
2. Nginx + Apache 一共发出了 56464 个请求，成功了五万两千个，比 fpm 还多一些
3. fpm 模式最大 QPS 为 800 但比较稳定，Nginx + Apache 最大 QPS 1000 但不够稳定
4. 至于 Apache 标准模式，显然它的技术架构不足以处理 5000 个 TCP 连接下 1000 QPS 的状况，QPS 低且不稳定，错误率高达 43%

## 结论

1. Nginx 处理海量用户的海量 TCP 连接的能力超群
2. 提升 Apache 性能只需要在前面加一个 Nginx 作为 HTTP 反向代理即可，因为此时 Apache 只需要处理好和 Nginx 之间的少量 TCP 连接，性能损耗较小
3. php-fpm 和 mod\_php 在执行 PHP 的时候没有任何性能差异

## epoll 和 prefork 的优劣势对比

### 优势

1. Nginx 每个 worker 进程可以高效地处理上千个 TCP 连接，同时消耗较少的内存和 CPU 资源。这使得单台服务器能够承载比 Apache 多两个数量级的用户量，相较于 Apache 单机 5K 的 TCP 连接数上限（对应于 2000 个在线用户），这是一个巨大的进步。
2. Nginx 对 TCP 的复用使其非常擅长应对海量客户端的直接连接。根据实际测试，在 HTTP 高并发环境下，Nginx 的活跃 TCP 连接数仅为 Apache 的五分之一，并且随着用户量的增加，复用效果更加显著。
3. 在架构上，基于 FastCGI 网络协议进行架构扩展，可以更轻松地利用多台物理服务器的并行计算能力，从而提升整个系统的性能上限。

### 劣势

1. 在低负载下，Nginx 的事件驱动特性导致每个请求的响应时长略长于 Apache prefork 模式（14ms vs 9ms）。

## 4.5 笔者的电商秒杀真实经验

---

笔者在工作中曾经成功地用 Nginx 改造了一个高并发的电商秒杀系统，保证了业务的正常开展，下面是笔者的真实故事。

### 新冠时期的机遇

在新冠疫情初期，住范儿的私域电商业务迅速发展。尽管 3 月中旬才复工，3 月份半个月的 GMV 就超过了前一年的总和，并在 4 月就完成了全年目标。然而，这也导致电商系统的性能压力急剧增加。

当时，我们的电商系统和第一版淘宝一样是购买的一个 PHP 单体系统，天生不具备扩展性。再加上我们采用团购秒杀的业务模式，这给系统带来了巨大的挑战。客户端主要使用微信小程序，而服务端则提供两种主要服务：开团瞬间的海量 HTTP API 请求以及每个页面都非常消耗资源的订单管理后台。

笔者面临的第一个问题是数据库无法承受如此巨大的压力。经过分析，笔者发现请求数最高的接口是商品详情，于是为该接口增加了一层保持时长为一分钟的 Redis 缓存，这个动作显著降低了开团瞬间的数据库压力。

而且幸运的是，当时阿里云刚刚商用 PolarDB 几个月。笔者利用 PolarDB 成功应对了开团三分钟内涌入的大约 4000 名用户带来的巨大的数据库压力。然而，当笔者将后端云主机升级到 16 核 32G 内存时，出现了一个非常奇怪的现象：

1. CPU 和内存占用率分别仅为 8% 和 6%。
2. 大量新用户无法建立 TCP 连接，客户端表现为长时间的等待。
3. 如果运气好，在等待一段时间后成功进入系统，那么访问将会非常丝滑，每个接口的响应时间都非常短。

为什么会出这种现象呢？原因是新用户无法与服务器建立 TCP 连接！

默认情况下，CentOS 7.9 单个进程的最大文件打开数（ulimit）为 1024。由于一切皆文件，每个 TCP 连接也是一个文件，因此也被锁定在了 1024 个。通常情况下，我们会将这个数字设置为 65535。然而，笔者观察到这台虚拟机最大 TCP 连接数只能达到 5-6K 之间，远远无法满足用户需求。无论是采用 prefork、worker 还是 event 模式都是如此。而原因正是我们之前实测过的：此时 Apache 已经忙不过来，花费了一颗 CPU 核心的全部时间片来进行数据包和线程的匹配。

怎么解决的？

为了解决这个问题，笔者在同一台机器上安装了 Nginx 作为反向代理服务器，将所有用户请求转发给 Apache 处理。这个操作立即缓解了请求压力，而且 Nginx 使用的最大活跃 TCP 连接数量也只有 1K，完全满足了三分钟 4000 用户的需求。此时，系统架构和第一章的图 1-1 一样。

## 4.6 面试题

---

**No.11 : Nginx 为什么比 Apache 性能强？**

Nginx 比 Apache 性能强的原因主要有以下几点：

1. Nginx 利用了 Linux Kernel 新引入的名为 epoll 的 I/O 多路复用 API。epoll 是一种高效的 I/O 事件通知机制，它可以同时监听多个文件描述符，当其中任何一个文件描述符就绪时，就会触发相应的事件。这样，Nginx 可以在一个线程中处理多个连接，提高了并发处理能力。
2. Nginx 以流量转发作为设计目标。它采用了事件驱动的异步非阻塞模型，可以高效地处理大量的并发连接。而 Apache 是从标准的磁盘文件和 CGI 协议出发的一个 Web 容器，内部的很多设计都是基于进程和阻塞的，所以在处理大量并发连接时性能较差。

3. Nginx 使用了一些优化策略来提高性能，例如：

- 使用了轻量级的 HTTP 协议解析器，减少了内存消耗。
- 使用了共享内存的方式存储会话信息，避免了频繁的磁盘 I/O。
- 使用了缓存技术，减少了对后端服务器的请求压力。

**No.12 : epoll 为什么能够处理海量的 TCP 连接？**

epoll 能够处理海量 TCP 连接的原因是它采用了事件驱动的方法来优化 TCP 连接和线程的匹配过程。

在传统的 select 方法中，每当收到一个数据包时，都需要遍历现存的所有连接。当连接数达到数千个时，寻找匹配关系的操作就会变得非常慢，甚至需要占据一整个 CPU 核心，导致性能无法继续提升，其他客户端也无法与服务器建立新的 TCP 连接。

而 epoll 通过将正向匹配流程反过来，基于事件驱动实现了高效的 TCP 连接与线程的匹配。具体来说，当接收到数据后，epoll 会通过二叉树快速找到所属的文件，并主动执行该文件对应的回调函数。这个操作大幅提高了处理海量 TCP 连接数时的效率。

## 第 5 章 负载均衡和应用网关

承接前一章中我们对于 Web Server 软件的了解，本章我们将深入地了解负载均衡和应用网关技术的发展，并尝试对它们进行高并发改造。

### 5.1 概述

负载均衡和应用网关是现代 Web 架构中非常重要的组成部分。它们可以帮助我们实现高并发、高可用性和高性能的 Web 服务。

#### 概念介绍

**负载均衡：**负载均衡是一种将请求分发到多个服务器的技术，以平衡服务器的负载并提高整体性能。常见的负载均衡算法有轮询（Round Robin）、加权轮询（Weighted Round Robin）、最小连接数（Least Connections）等。

**应用网关：**应用网关是一个位于客户端和服务器之间的中间层，它可以实现请求路由、过滤、认证、限流等功能。应用网关可以帮助我们更好地管理和维护 Web 服务，灵活、动态地控制流量去往的方向。

#### 真实业务架构图

在上一章中，笔者通过将 Nginx 置于 Apache 之前，成功解决了单体 PHP 网站面临的高并发问题，并暂时支撑了业务。随后，笔者立即着手开始研发分布式电商平台。

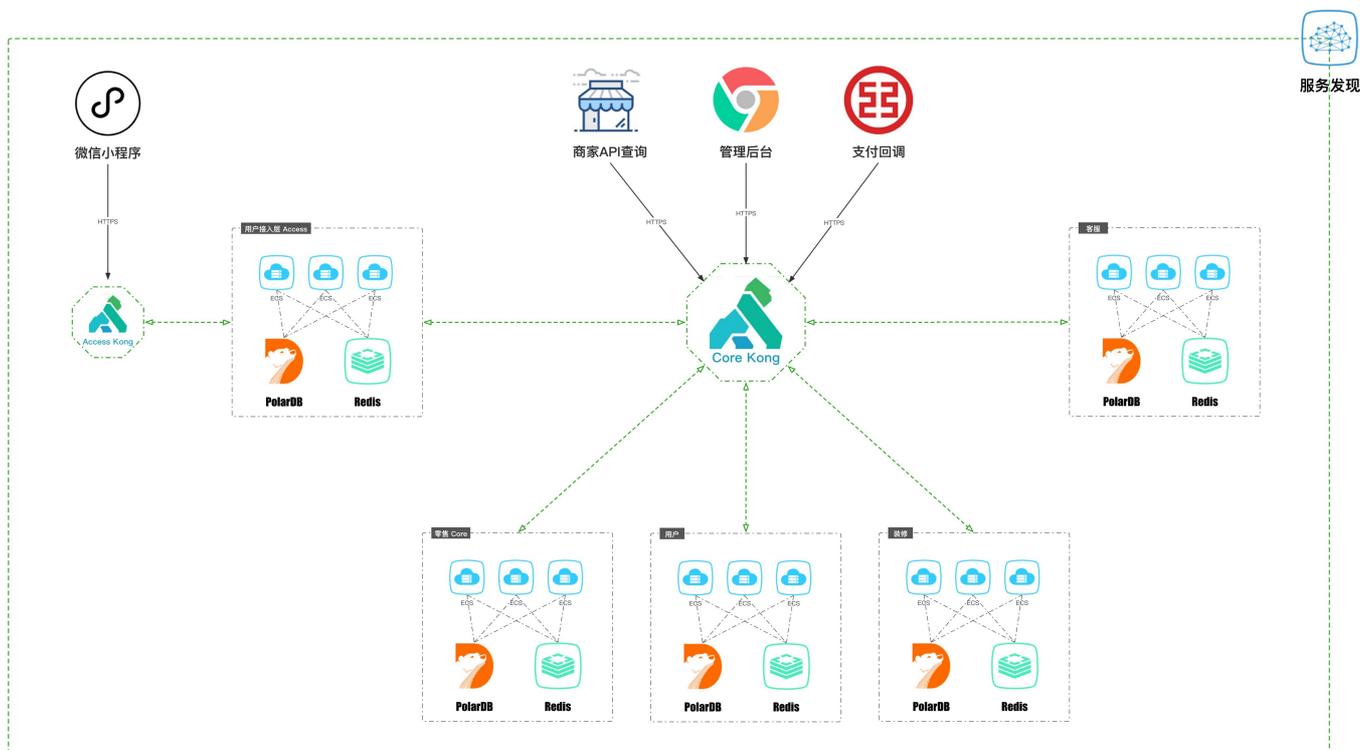


图 5-1 住范儿电商平台的“百亿架构”

如图 5-1 所示是笔者为住范儿电商平台设计的分布式微服务架构，笔者称之为“百亿架构”——在年 GMV 达到百亿之前无需更换架构。经过两年多的运行，这套电商平台距离这个目标已经不到十倍的差距，目前看来十分有望实现最大一百亿 GMV 的设计目标。读者应该能明显地看出，这套架构的核心是 Kong 网关和涵盖所有机器的服务发现。

## 5.2 静山平台如何支持 50000 QPS

下面我们以静山平台为例，从需求出发，深入了解负载均衡和应用网关这两个关键技术。

假设我们的静山电商平台发展迅猛，很快每秒钟的 HTTPS API 请求数量就达到了五万，我们需要着手给静山平台设计高并发架构。

解决问题的第一步，认识问题——让我们先来评估一下五万 QPS 大概是一个什么规模的压力。

### 京东平峰期并发量估算

我们通过京东财报中的数据来评估一下京东工作日上午平峰期的并发量。

根据财报，京东在 2022 年的 GMV 达到了 33155 亿元。我们可以按照以下步骤进行估算：

1. 首先，每年两场大促活动大约占据了全年交易额的 20%。因此，剩下的 80% 的交易额需要分散在 365 天中。
2. 其次，假设每个人每天睡觉 8 小时，那么剩下的 16 小时就会在京东平台购物。我们进一步假设每晚高峰期占据了 4 小时，并且这 4 小时占据了 50% 的交易额。
3. 最后，剩下的 12 小时占据了另外 50% 的交易额。我们假设用户在这 12 小时内下单行为是平均的。

通过以上计算，我们可以得出平峰期每秒交易额为  $331550000 \times 0.8 \div 365 \div 2 \div 12 \div 3600 = 8.4$  万元。以京东的平均订单价值 650 元估计，平峰期每秒订单数约为 130 单。我们假设 QPS 是订单数的 200 倍，则京东在平峰期的 QPS 大约为 2.6 万。这个数字只有我们静山平台的一半。

既然五万 QPS 是一个很大的数字，那我们该如何设计架构满足这个并发量呢？

## Kong 网关需要的硬件规模

目前，在一台拥有 2 vCore 的虚拟机上，Kong 网关在承受 2000 QPS 的压力时对应的 CPU 占用率大约是 20%。经过换算，我们可以得知：如果 Kong 的性能可以随着核心数的增加而线性提升的话，在保持最大 CPU 占用率为 40% 的情况下，我们需要：

$$(50000 / 2000) \times (20\% / 40\%) = 25 \text{ 核}$$

所以，我们至少需要在一台拥有 25 核 CPU 的云服务上安装 Kong 网关，才能应对五万 QPS。实际上这个规模已经超过单机 Kong 的极限了，我们稍后会做详细讨论。现在，让我们先来详细了解一下这个 Kong 网关究竟是什么。

## Kong 网关

在第三章中，我们已经了解到将单台物理服务器拆分成多个虚拟机可以提高系统整体容量。然而，仅仅依靠虚拟机技术无法单独提升系统容量，因为分布式计算架构需要一个流量分发器。通常情况下，我们不会奢侈地为每台后端虚拟机分配一个公网 IP 并使用 DNS 进行流量分发。因此，域名必须解析到某个公网 IP，而该 IP 指向的机器需要承担流量分发器的角色。

为了实现这一目标，笔者选择了 Kong gateway 软件作为反向代理服务器。Kong 能够处理海量 HTTPS 请求，并以 HTTP 协议将其发送到后端的多台应用虚拟机上。

Kong 是基于 OpenResty 技术开发的开源网关。OpenResty 是由国人章亦春创建的开源软件项目，它将非常轻量的 Lua 语言嵌入了 Nginx 对 HTTP 请求处理的整个生命周期。这使得原本只能静态配置的 Nginx 拥有了一种类似于 Perl 和 PHP 的动态脚本语言，从而极大地扩展了 Nginx 的能力。除了具备 HTTP 网关的一整套功能外，Kong 还拥有插件系统和基于数据库的水平扩展能力，理论上可以支持超高并发的系统。

在我们为静山平台引入 Kong 网关以后，其平台架构如图 5-2 所示。

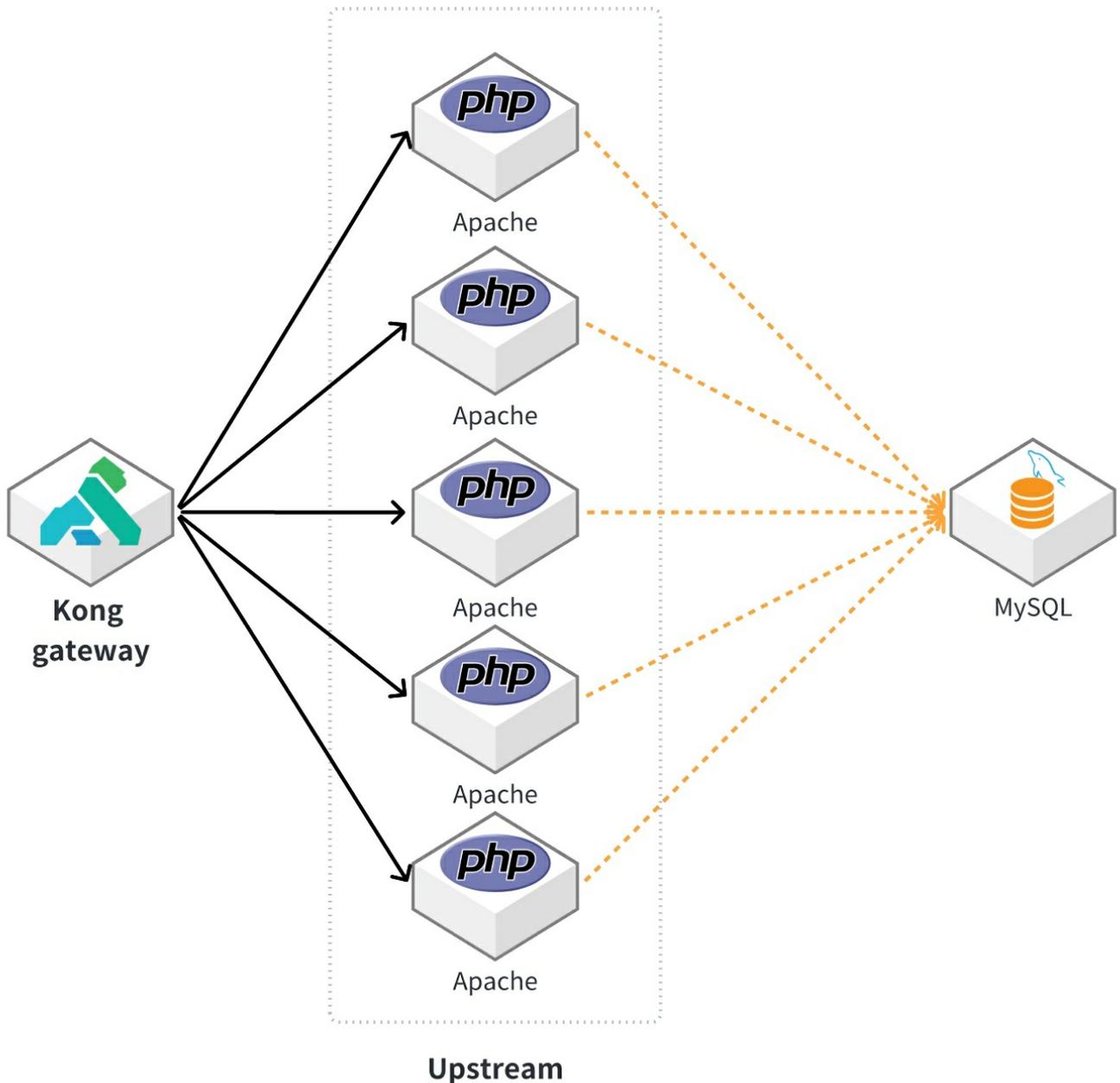


图 5-2 引入 Kong 网关之后的静山平台架构图

### 上游服务器 (Upstream) 的定义

在一个包含 Kong 网关的架构中，上游服务器 (Upstream) 是指执行后端代码的实际机器，也称为应用虚拟机。在接下来的讨论中，我们将频繁使用“上游服务器”这个术语。

### 服务发现的概念

在流量低谷期，笔者使用一台虚拟机运行 Apache 作为上游服务器。如果在开团前需要再启动一台上游服务器，Kong 如何知道新服务器已经启动并获取其 IP 地址呢？这就需要使用服务发现技术。

类似于 Kong，HashiCorp 开源的 Consul 也可以独立于 Kubernetes 环境运行，因此笔者选择使用它来进行服务发现。

服务发现的原理其实非常简单：构建一个共识集群，各个节点之间使用固定的端口进行通信。当新的上游服务器开机后，其上的 Consul 服务会开机启动，并与预先配置好的主节点 IP 进行通信，加入集群并广播自己的 ip。此时，集群内的所有机器都知道有一台新的机器加入集群了。

加入集群后，Consul 会基于本地的配置文件，向整个集群广播自己本机可以提供的服务名称，假设静山平台后端服务的名称为“up”，其对应的默认域名就是 `up.service.consul`。此时，集群内的所有节点都可以通过 Consul 的 DNS 服务，通过查询 `up.service.consul` 域名，获得声明自己可以提供“up”服务的所有机器的 IP 地址。安装 Kong 网关的虚拟机同样位于集群内，所以它也可以使用这个方法获取到新加入集群的服务器的 IP 地址。

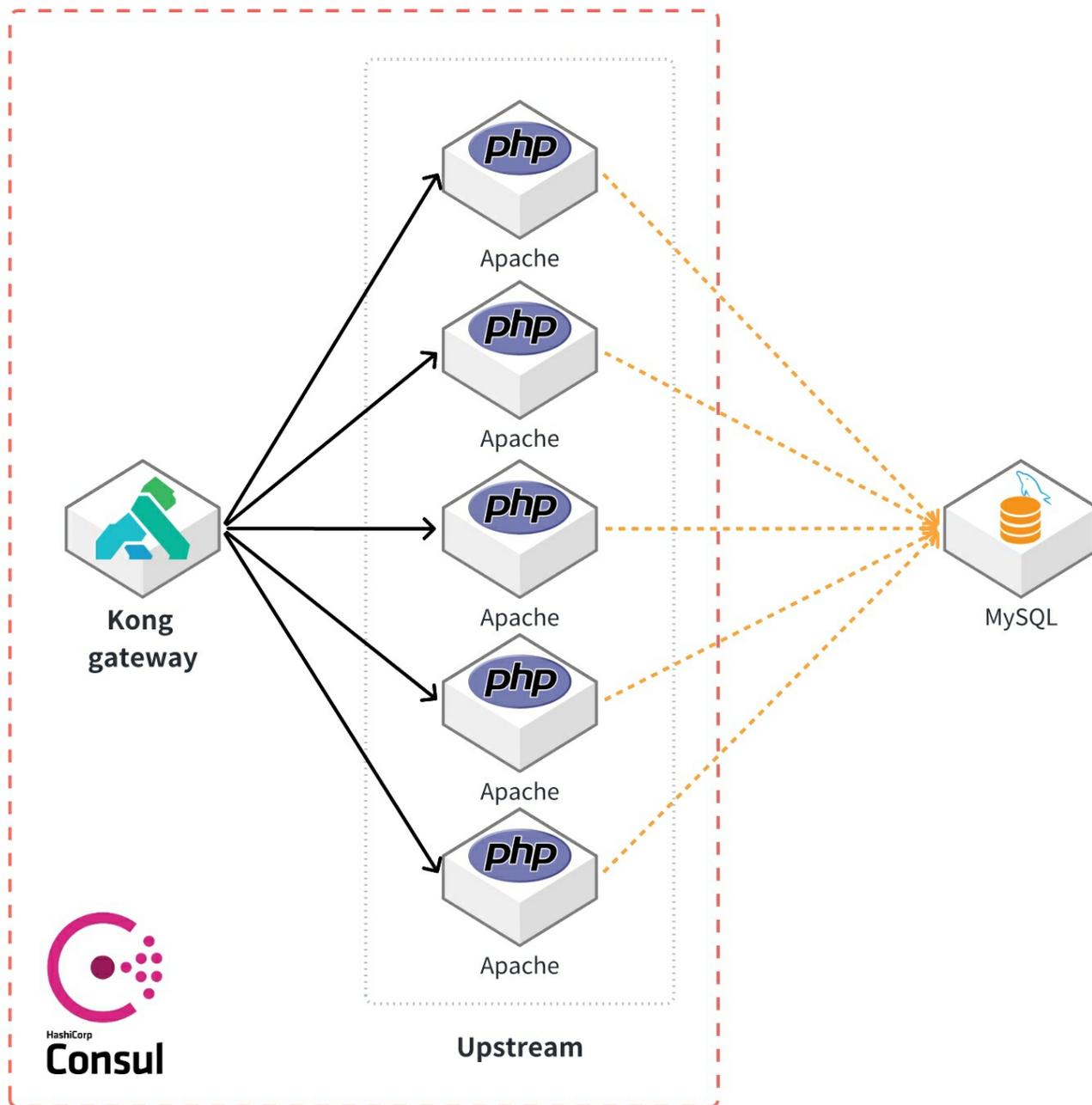


图 5-3 引入服务发现 Consul 之后的静山平台架构

当 Kong 接收到来自客户端的 HTTPS 请求后，需要将请求转换为向 Upstream 发起的 HTTP 请求。此时，Kong 会向本机上安装的 Consul DNS 服务发送查询，以查找 `up.service.consul` 这个域名对应的 IP 地

址。在此之前，该域名只会返回一个 IP 地址，即唯一的上游服务器的 IP 地址。然而，在新机器启动并成功加入集群后，该域名的解析结果将变为两个 IP 地址，Kong 可以将 HTTP 请求均匀地发送给新旧两台上游服务器，从而将系统总容量提升一倍。此时静山平台架构如图 5-3 所示。

## 什么是应用网关

应用网关，也被称为 API 网关，顾名思义，它是所有 API 请求的入口：它接收所有的 HTTP/HTTPS/TCP 请求，并将请求转发给真正的上游服务器。这些上游服务器可以是一堆虚拟机、容器，甚至是多个数据中心各自的应用网关。由于应用网关所承担的任务相对较少，因此它能够使用单台服务器支持很高的并发量。

常见的应用网关软件包括 HAProxy、Nginx、Envoy 等，而 Cisco、Juniper、F5 等一体化设备厂商也提供相关的硬件产品。

除了提升系统容量外，应用网关还具有许多其他优势。

### 1. 解放后端架构

经过对应用网关三年多的使用之后，笔者现在认为所有系统都应该放在应用网关的背后，包括开发环境。应用网关对后端架构的解放作用实在是太大了，可以让你在后端玩出花来：各种语言、各种技术、各种部署形式、甚至全国各地的机房都可以成为某条 URL 的最终真实服务方，让你的后端架构彻底起飞。

### 2. TLS 卸载

终端用户访问应用网关时采用的是 HTTPS 协议，这个协议需要对数据进行加密解密，应用网关非常适合完成这个任务，而背后的业务系统只需提供标准 HTTP 协议即可，从而降低了业务系统的部署复杂度和资源消耗。

### 3. 身份验证和安全性提升

应用网关可以对后端异构系统进行统一的身份验证，无需单独实现。同时也可以统一防火墙白名单，后端系统防火墙只需对网关 IP 开放，极大地提升了后端系统的安全性，降低了海量服务器安全管理的难度。甚至可以针对某条 API 进行单独鉴权，使系统的安全管控能力大幅提升。

### 4. 指标和数据收集

由于所有流量都会经过网关，因此对指标进行收集变得简单了。你甚至可以将双向流量的内容全部记录下来，用于数据统计和安全分析。

### 5. 数据压缩与转换

应用网关还可以统一对流量进行 gzip 压缩，可以将所有业务一次性升级到 HTTP/2 和 HTTP/3，可以对数据进行格式转换（XML 到 JSON）和修改（增加/修改/删除字段），总之就是能够灵活应对各种需求，随心所欲地操控输入和输出的数据。

## 5.3 单机 Kong 的性能极限

承接上文，如果我们真的搭建了一台拥有 25 个核心的虚拟机，并在上面安装了 Kong，它其实是无法承受五万 QPS 压力的——因为此时单机 TCP 连接数已经达到了十万以上，强如 Nginx 也顶不住的。

在经典的 HTTP 反向代理场景下，单机 Nginx 的 QPS 极限大约为一万，一旦超过这个限制，性能将不再增长甚至开始下降，用户体验也会迅速恶化。因此，我们需要对应用网关进行拆分。

### 应用网关如何拆分

从逻辑上讲，应用网关执行的是“反向代理+数据过滤”任务，并没有要求应用网关只能由一台服务器来承担。换句话说，应用网关理论上不是单点，只要多个节点的行为一致，它们就可以共同承担五万个 QPS 的真实用户流量。

我们只需要在多台机器上安装相同版本的应用网关软件，并在它们之间同步配置文件即可。Kong 采用的策略是让多个实例连接到同一个 PostgreSQL 数据库，每五秒钟从数据库获取一次最新的配置。如果数据库出现故障，那么它将保持内存中的现有配置继续运行。

Kong 集群追求的是“最终一致性”，而不是追求五秒钟的得失，这反而让系统格外地容易扩展，格外的健壮。在最后一章中，我们还将看到使用类似思维的“DNS 分布式拆分”。这种简单直接的思维颇具俄罗斯人暴力美学的典范，在后面讨论列存储 ClickHouse（俄罗斯人开发的开源列存储数据库）时还会出现。

应用网关拆分方案已经呼之欲出了——在多台应用网关前面放置一个 TCP 负载均衡器，系统架构如图 5-4 所示。

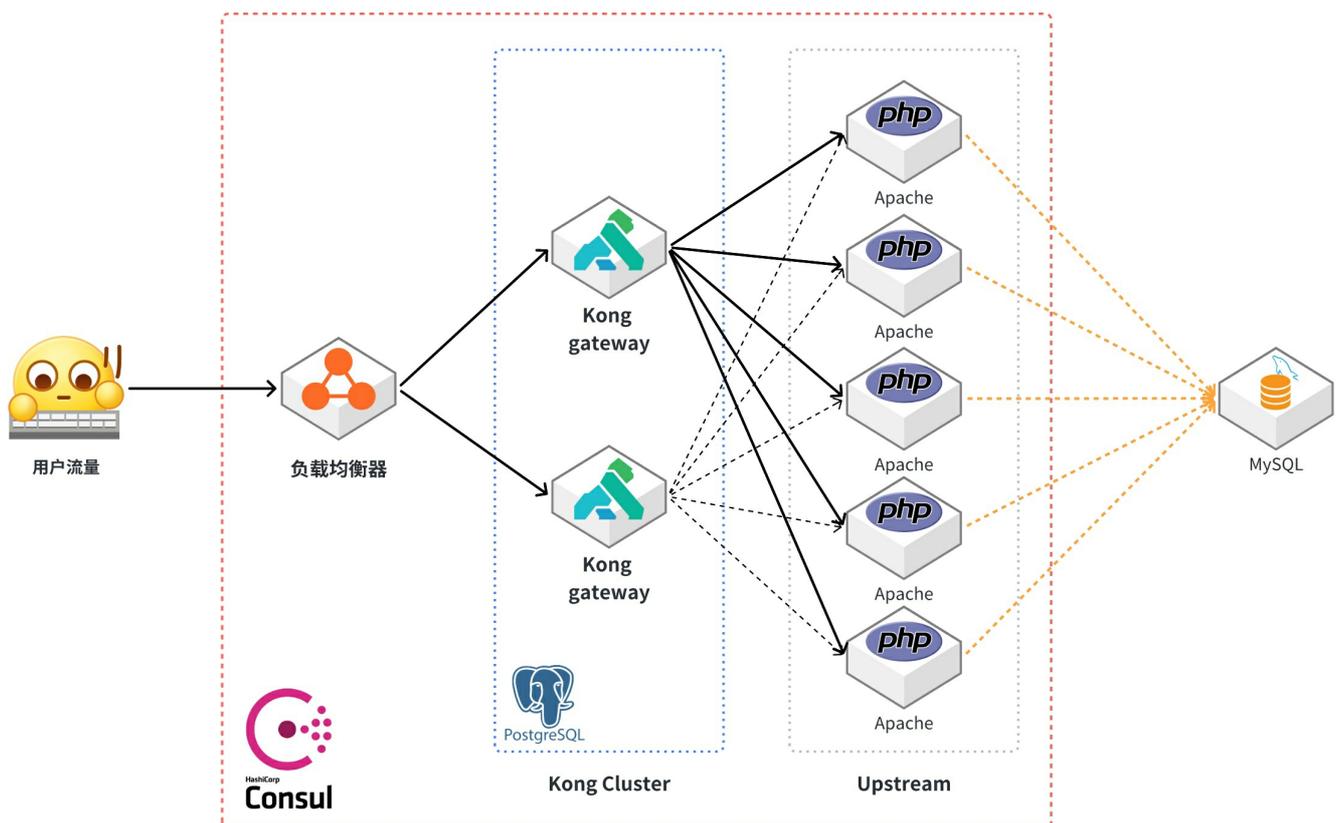


图 5-4 引入负载均衡器之后的静山平台架构图

## 什么是负载均衡器

负载均衡器是一种将网络流量分发到多个计算资源（例如服务器、虚拟机、容器等）进行处理的设备，图 5-4 中的 TCP 负载均衡器就是俗称的“四层负载均衡”。在云计算兴起之前，负载均衡器主要依靠硬件实现，用于将单机无法承受的大流量分散到多台物理机上共同承担。如今，软件定义网络（SDN）已经通过廉价的商用硬件颠覆了这一领域，这是我们下一章的主题，到时详细阐述。

除了提升系统容量外，负载均衡器还具备一些高级功能：

1. 红蓝发布（Blue-Green Deployment）：这是一种发布策略，通过在两个生产环境（蓝色和绿色）之间切换来实现零停机时间部署。在部署新版本时，流量会被切换到新的环境（绿色），一旦新环境稳定运行，流量会完全切换到新环境，旧环境（蓝色）则被废弃。
2. 金丝雀发布（Canary Release）：这是一种渐进式发布策略，通过逐步将新版本的代码推向生产环境来降低风险。只有一小部分用户会看到新版本，一旦出现问题，可以迅速回滚到旧版本。
3. 根据流量特点进行灰度发布：根据用户的流量特点，将新版本的代码推送给特定的用户群体，以便在不影响整体用户体验的情况下进行测试和验证。
4. 主动调节各个后端服务器的压力：负载均衡器可以根据服务器的负载情况动态调整流量分配，以确保每个服务器的负载保持在合理范围内。
5. 屏蔽失效的后端服务器：当某个后端服务器出现故障时，负载均衡器会自动将其从流量分配中移除，确保用户的请求不会被发送到失效的服务器上。

低负载下应用网关和负载均衡器可以是同一个软件

尽管应用网关和负载均衡器是两个不同的概念，但在容量不是特别大（例如不超过 1Gbps）的系统中，它们往往由同一个软件来扮演，例如 Kong 网关就同时具备这两种能力。

## 5.4 分层的网络

读到这里可能有读者会疑惑，既然单机的 Nginx 都顶不住五万 QPS 带来的 TCP 资源开销，那负载均衡器如何顶住呢？因为负载均衡器承载的是比 Nginx 所承载的 TCP 更下面一层的协议：IP 协议。

至此，我们正式进入了网络拆分之路，这条路很难走，但收益也会很大，最终我们将在下一章得到一个 200 Gbps 带宽的“软件定义的负载均衡集群”，让我们正式开始。

网络是分层的

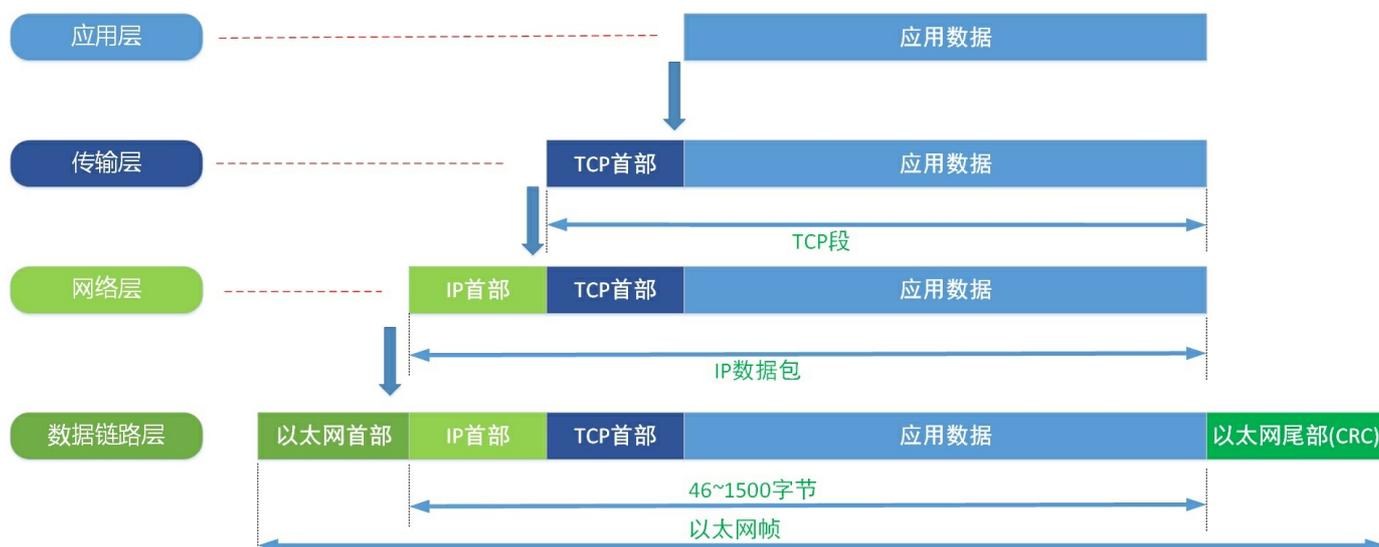


图 5-5 经典 TCP/IP 四层网络协议的首部 (header)

经典 TCP/IP 四层网络协议的首部 (header) 会层层累加, 最后再统一在网络中传输的, 其结构如图 5-5 所示。

如果读者查看过网页的源代码, 肯定知道网页背后是一段 HTML 代码文本, 这段文本是被层层包裹之后, 再在网络中传输的, 这段文本就是图 5-5 中的应用数据的一部分 (包含在 HTTP 协议之中)。以太网之所以拥有如此之强的扩展性和兼容能力, 就是因为它的“分层特性”: 每一层都有专门的硬件设备来对网络进行扩展, 最终组成了这个容纳全球数十亿台网络设备的“互联网”。最近, 这些传统硬件设备的工作越来越多地被软件所定义, 即软件定义网络 (SDN)。

## 什么是软件定义网络

软件定义网络 (Software-Defined Networking, SDN) 是一种将网络资源抽象到虚拟系统中的 IT 基础架构方法。SDN 让 IT 运维团队通过软件抽象出的控制平面来控制复杂网络拓扑中的网络流量, 无需手动处理每个网络设备。SDN 的核心思想是将网络控制与网络转发解耦, 从而实现网络结构的随时更改, 以适应瞬息万变的业务需求。

## 应用数据是什么

应用数据是指网页背后的 HTTP 协议所包含的全部信息。

我们使用 Charles 反向代理软件可以轻松获取 HTTP 协议的详细信息。下面是一个普通的 GET 请求示例。请使用浏览器访问 <http://httpbin.org> (在尝试时, 不要选择 HTTPS 网站) :

## 请求内容

当你的浏览器发起一个网页请求时, 在 TCP 数据流的内部传输的内容就是遵循 HTTP 协议规范的字符串, 如代码清单 5-1 所示。

代码清单 5-1 一个 HTTP 请求的字符串内容

```
GET / HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/Webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6
Cache-Control: max-age=0
Connection: keep-alive
Host: httpbin.org
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36 Edg/108.0.1462.54
```

---

数据解释：

1. 第一行包含三个元素：HTTP 方法、URI（请求路径）、HTTP 版本
2. 之后的每一行以冒号 : 作为分隔符，左边是键，右边是值
3. HTTP 协议中，换行采用的不是 Linux 系统的 `\n`，而是与 Windows 相同的 `\r\n`

响应内容

而浏览器接收到的服务器返回的内容，依然是在 TCP 数据流内部传输的、遵循 HTTP 协议规范的字符串，如代码清单 5-2 所示。

代码清单 5-2 一个 HTTP 响应的字符串内容

```
HTTP/1.1 200 OK
Date: Wed, 04 Jan 2023 12:07:36 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 9593
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>httpbin.org</title>
  <link href="https://fonts.googleapis.com/css?family=Open+Sans:400,700|Source+Code+Pro:300,600|Titium+Web:400,600,700"
    rel="stylesheet">
  <link rel="stylesheet" type="text/css" href="/flasgger_static/swagger-ui.css">
  <link rel="icon" type="image/png" href="/static/favicon.ico" sizes="64x64 32x32 16x16" />
</head>

<body>
  <a href="https://github.com/requests/httpbin" class="github-corner" aria-label="View source on GitHub">
  </a>
  ... .. 此处省略一万个字
</div>
</body>

</html>
```

HTTP 响应的基本规则与 HTTP 请求相同，第一行的三个元素分别是协议版本、状态码和状态码的简短解释。唯一的区别在于，返回值中还包含 HTTP body。

1. 在两个换行 `\r\n\r\n` 之前的内容为 HTTP header
2. 在两个换行之后的内容为 HTTP body
3. HTTP body 就是你在浏览器中执行“查看网页源代码”操作时所看到的内容

## HTTP 层之下是 TCP 层

在 HTTP 数据流的外部，包裹着 TCP 首部的数据，其中包含的字段如图 5-6 所示。

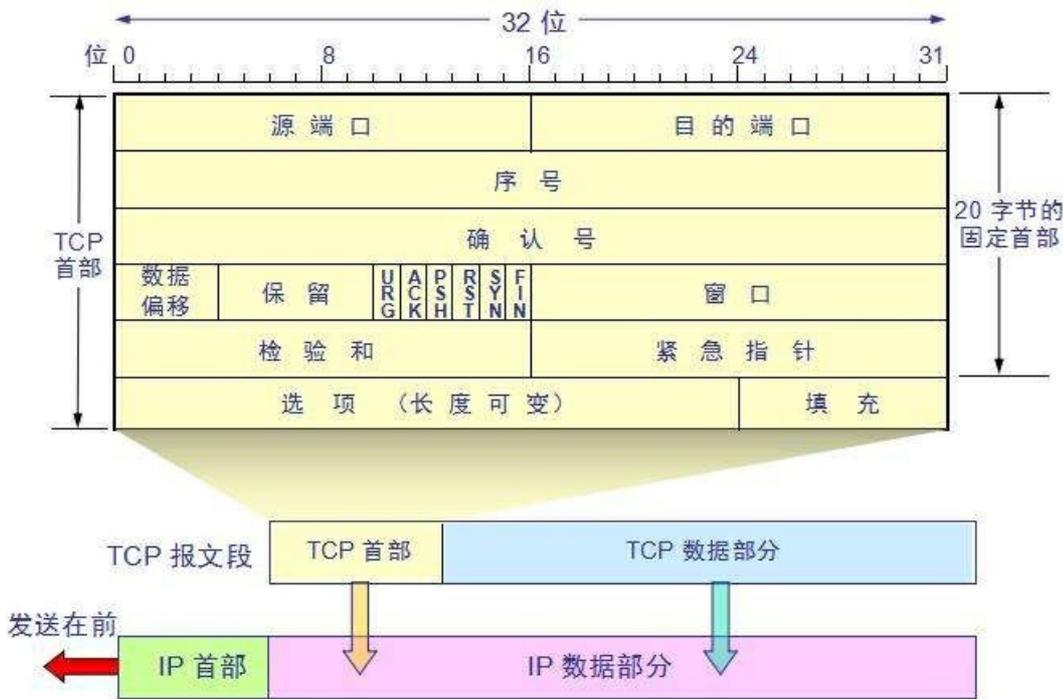


图 5-6 TCP 首部包含的字段

### TCP 首部重要数据描述

1. TCP 首部中最重要的数据是 **源端口** 和 **目的端口**
2. 它们各由 16 位二进制数组成， $2^{16} = 65536$ ，所以网络端口的范围是 0~65535
3. 我们可以注意到，目的端口号这个重要数据是放在 TCP 首部的，与更下层的 IP 首部、以太网帧首部无关

### TCP 层之下是 IP 层

全球所有公网 IPv4 组成了一个大型网络，这个 IP 网络实际上就是互联网的**本体**。IPv6 更加复杂，为了便于读者们理解，本节中的示例均采用 IPv4 协议。

在 IP 层中，每台设备都有一个 IP 地址，形如 **100.100.100.100**：

1. IPv4 地址范围为 0.0.0.0 - 255.255.255.255
2. 255 为 2 的 8 次方减一，所以我们用八位二进制数表示 0-255
3. 四个八位即为 32 位，4 个字节

### IP 首部包含哪些信息

从上图可以看出，IP 首部有固定长度的 20 字节用于存储 IP 数据包的基本信息：

1. 源地址 32 位 (4 个字节)：100.100.100.100
2. 目的地址 32 位 (4 个字节)：110.242.68.3
3. 协议 8 位 (1 个字节)：内部数据包使用的协议，如 TCP、UDP 或 ICMP (Ping 命令使用的协议)
4. "首部检验和" 16 位 (2 个字节)：对 IP 首部的数据进行加工后得到的一串字符串，类似于 MD5，用于验证 IP 首部的数据完整性

IP 首部最重要的数据是“源 IP 地址”和“目的 IP 地址”。

IP 层之下是 MAC 层(物理层)

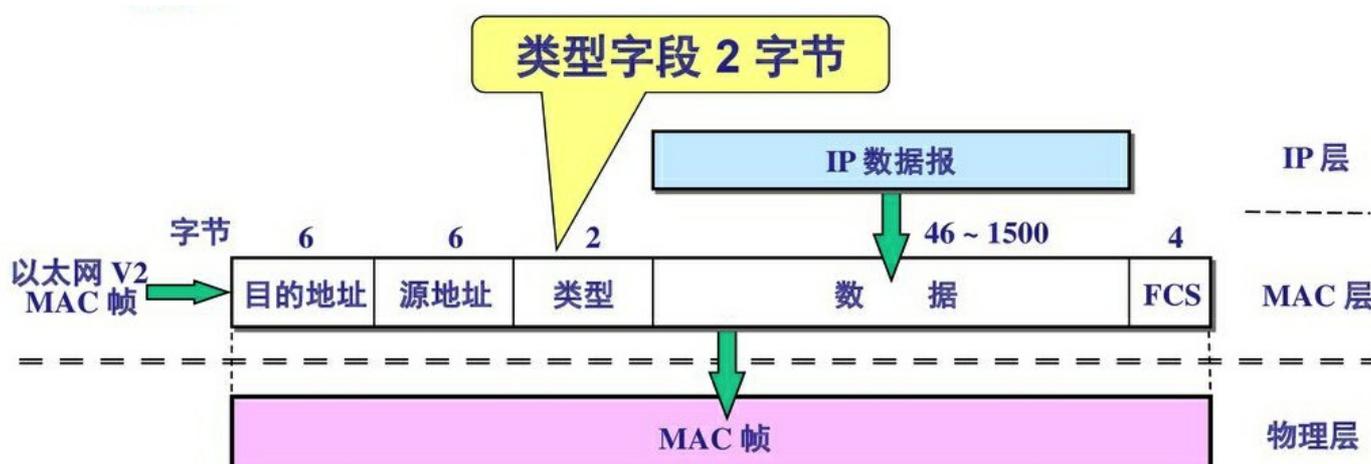


图 5-7 MAC 帧的数据结构

在物理层中，二进制数据以图 5-7 中所示的格式进行组织，其基本单位被称为“MAC 帧”。MAC 层是交换机工作的地方。

不同的网络设备的 MAC 帧长度可能不同，默认为 1500 字节，即 IP 层的数据会按照这个长度进行分包。当局域网速度无法达到协商速率时，需要进行性能优化（例如 iSCSI 网络磁盘），可以使用“巨型帧”技术，将该数字增加到 10000 字节，以提高网络传输性能。然而，根据笔者的实际优化经验，大多数情况下，巨型帧对网络性能的提升小于 5%，属于一种聊胜于无的优化手段。

前两段目的地址和源地址都是 MAC 地址，形式如 AA:BB:CC:DD:EE:FF，共有六段，每段是一个两位的十六进制数。两位十六进制数转换为二进制就是 8 位，因此 MAC 地址的长度为  $8 \times 6 = 48$  位，六个字节。

第三段类型字段使用 16 位二进制表示上一层（IP 层）的网络层数据包的类型：IPv4、IPv6、ARP、iSCSI、RoCE 等等。

## 5.5 负载均衡器的工作原理

本节将详细介绍负载均衡器的工作原理，相信读者在阅读后能够对这一概念有更深入的理解。

### 真实世界中 TCP 连接数和 QPS 的比例

在实际环境中，QPS（每秒查询率）通常低于保持 TCP 连接的客户端数量。也就是说，打开应用的用户并非每秒都会发起一次 API 请求，平均每隔几秒才会有一次操作。假设这个时间为 4 秒一次，那么：如果有 20 万个客户端在线，每个客户端设备平均每 4 秒发送一个 HTTPS 请求，QPS 为  $200000 \div 4 = 50000$ 。

### 单台 Nginx 反向代理的性能极限

Nginx 不仅需要建立 TCP 连接，还需要将 TCP 连接中发送过来的数据包与某个进程/线程进行匹配，同时还需要对 HTTP 协议的信息进行解析、识别、转换、添加。因此，Nginx 也有其 QPS 上限：

在 2015 年主流的服务器 CPU 上，Nginx 官方在进行了极限优化的情况下进行了反向代理性能测试。在完整执行“建立 TCP 连接-发送 HTTPS 请求-断开 TCP 连接”这样一个“网络事务”的情况下，最高性能为 60000 QPS (SSL TPS RSA 2048bit) 。

假设我们使用最新的服务器硬件，当虚拟机 CPU 达到 32 vCore 的时候，未经优化的单机 Nginx 性能就已经达到极限，能承受大约 1 万 HTTPS QPS，对应的连接用户就是 4 万，这就是 Nginx 在单台 x86 物理服务器上的极限。

我们假设单台 Kong 应用网关的极限为 1 万 QPS，于是我们就需要五台 Kong。那么这五台 Kong 前面的 TCP 负载均衡器为何能够扛住呢？因为 TCP 负载均衡器要做的事情比 Kong 少非常多：它只需要在 IP 层做少量的工作即可。

### 使用负载均衡器拆分 TCP 单点

TCP 协议是一种“可靠地传输信息”的方法，它不仅有三握手四次挥手等复杂的控制流程，还会对每一个报文段进行排序、确认、重发等操作来保证最终数据的完整和正确。因此，TCP 本身就是一种需要很多资源处理的单点。现在我们开始拆分这个单点。

### TCP 负载均衡器的工作过程

我们假设一个含有负载均衡器的系统：客户端 IP 为 123.123.123.123，负载均衡器的 IP 为 110.242.68.3 (公网) 和 10.0.0.100 (私网)，五台 Kong 服务器的 IP 为 10.0.0.1 ~ 10.0.0.5，那么这个系统的架构就如图 5-8 所示。

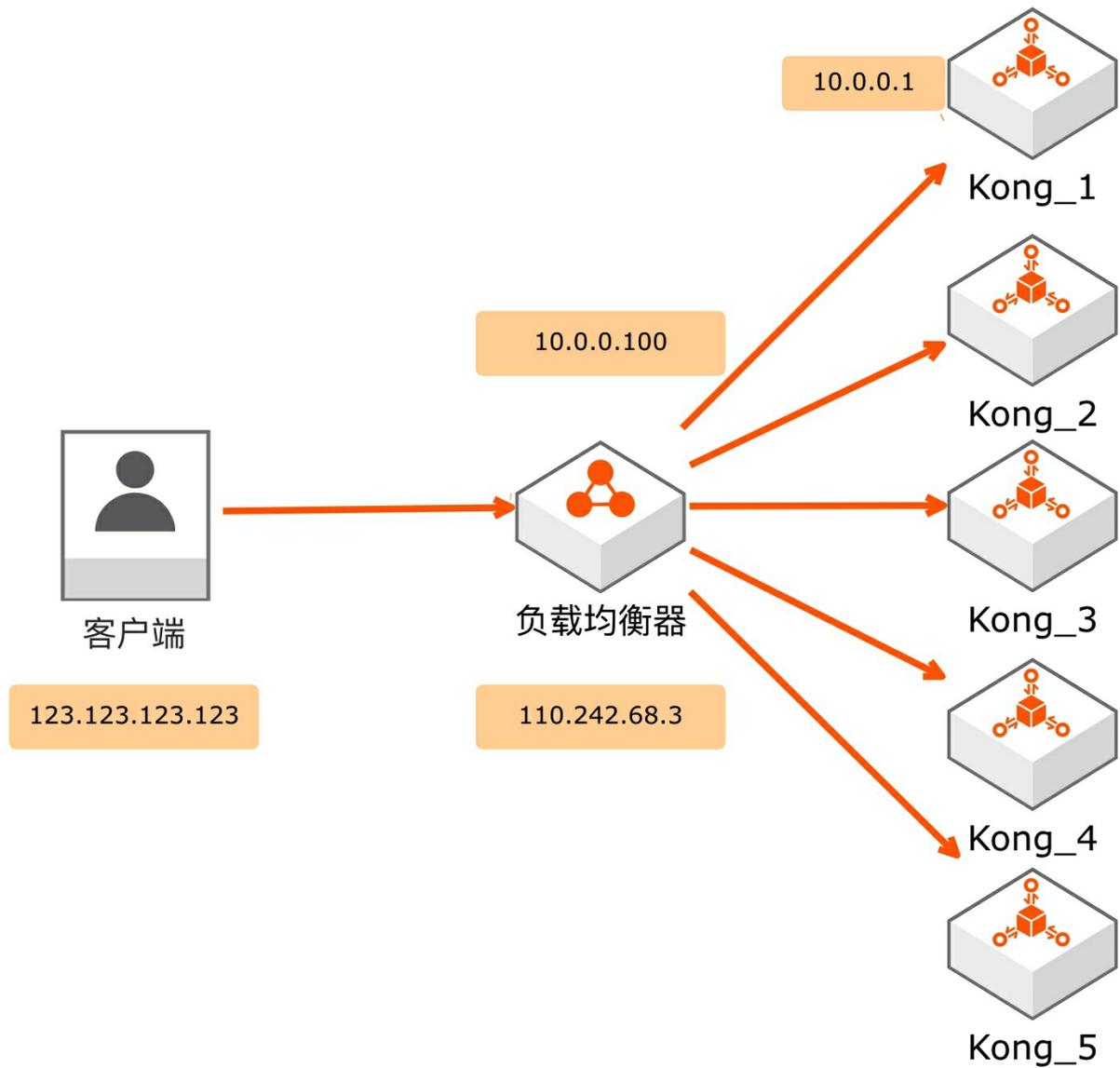


图 5-8 使用 TCP 负载均衡器时的架构图

负载均衡器的工作过程如下：

### 1. 接收数据（左侧）

负载均衡器接收客户端数据包（报文）的过程如下：

1. 负载均衡器收到了一个 IP 报文，其源地址为 123.123.123.123，目的地址为 110.242.68.3。
2. IP 报文内包裹着一个 TCP 报文，具体信息如下：源端口为 52387，目的端口为 443。
3. 需要注意的是，负载均衡器仅接收了一个 IP 报文，并未与客户端进行三次握手，也未与客户端建立“TCP 连接”。

### 2. 发送数据给上游服务器（右侧）

在接收到客户端的 IP 报文后，负载均衡器会寻找一台上游服务器，准备将数据发送过去：

1. 内部 TCP 报文首部：源端口为 45234，目的端口为 443。
2. TCP 报文外层包裹的 IP 首部：源地址为 10.0.0.100，目的地址为 10.0.0.1。

3. 负载均衡器将包裹着 TCP 数据包的 IP 报文发送出去。

### 3. 建立映射关系并进行数据转发

负载均衡器会在内存中创建两个五元组：

左侧五元组

1. 左侧源地址 123.123.123.123
2. 左侧目的地址 110.242.68.3
3. 左侧源端口 52387
4. 左侧目的端口 443
5. 协议 TCP

右侧五元组

1. 右侧源地址 10.0.0.100
2. 右侧目的地址 10.0.0.1
3. 右侧源端口 45234
4. 右侧目的端口 443
5. 协议 TCP

然后，负载均衡器会关联这两个五元组：对两侧发来的数据包（报文）进行拆包和修改（两个地址+两个端口），并从另一侧发送出去。

对于阅读过笔者《软件工程师需要了解的网络知识》系列文章的读者来说，应该能够一眼看出，这就是网关的工作模式，你家几百块的路由器主要执行的就是这个任务。

### 为什么负载均衡器性能开销比 Kong 低

通过前面的内容我们可以发现，负载均衡器/网关只需要执行两个主要任务：建立五元组并关联，以及修改数据包的地址和端口并将其发送出去。这个过程在网络领域被称为 NAT（网络地址转换）。

由于这个操作相对简单，大部分工作可以通过专用硬件来完成。例如，可以开发专门用于存储和关联五元组的芯片，以及专门的 NPU（网络数据包处理器）来进行快速的数据修改。正因为可以使用低性能的专用硬件来实现，家用路由器才能以低于 300 元的终端售价实现超过 1Gbit/S 的 NAT 性能。

### Kong 网关需要建立“真·TCP连接”

与负载均衡器只需执行两个任务相比，Kong 网关需要真正地与客户端建立 TCP 连接。具体来说，它需要进行以下操作：

1. 进行三次握手以建立 TCP 连接；
2. 对数据包进行排序、校验，并在收到心跳包时进行回复；
3. 将该 TCP 连接与一个进程/线程绑定：

- 在收到数据后，找到相应的进程/线程并将数据发送给它；
- 等待进程/线程回复后，再找到对应的 TCP 连接并将数据发送出去。

这一系列操作流程较长，且经常需要进行配置修改。如果设计专用硬件来实现，可能会过于复杂，并且收益可能低于投入。因此，使用通用 CPU 加软件的解决方案更加合适。

#### 四层和七层负载均衡 (L4/L7)

在商业负载均衡公司中，应用网关也被称为七层负载均衡，因为它工作在 OSI 七层网络模型的第七层。而我们讨论的负载均衡工作在 IP 层的称为四层负载均衡，工作在 OSI 七层网络模型的第四层。读者以后看到 L4、L7 这两个词时，应该能一眼看穿它们了，其实一点都不神秘。

#### 五万 QPS 下静山平台的架构图

在一个 TCP 负载均衡器下挂载五个安装了 Kong 应用网关的 6 vCore 虚拟机，再下挂 125 台 8 vCore 的 PHP 虚拟机——静山平台的五万 QPS 终于被系统给支撑住了。此时静山平台的架构如图 5-9 所示。

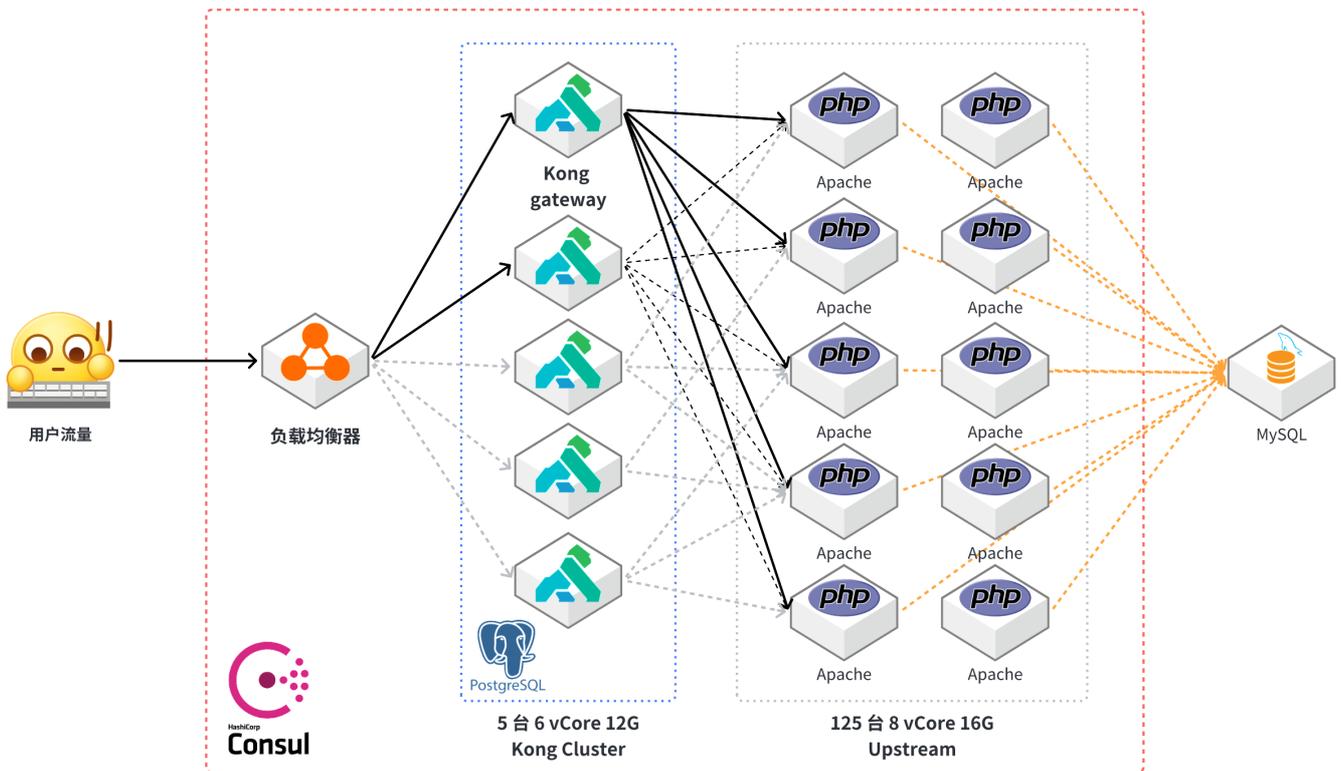


图 5-9 引入 TCP 负载均衡器之后的静山平台架构图

## 5.6 面试题

### No.13 : 为什么需要做负载均衡 ?

网站、移动应用和小程序的流量可能会随着时间不断增长，但单台虚拟机的性能并不能随着配置的增加而线性增长。

这个单机性能的上限与所采用的 Web 服务器和编程语言有关。以 Apache + mod\_php 为例，其单机上限

大约为 200QPS。

当服务器的核心数达到一定规模后，性能将不再增长。如果此时系统压力超过了单台服务器的承受极限，我们就不得不将后端服务器拆分成多台，并设计一个负载均衡架构来处理海量的用户请求。

#### No.14：如何提升系统的横向扩展能力？

要提升系统的横向扩展能力，可以采取以下措施：

1. 在架构设计上去除单机依赖：确保系统的设计不依赖于单个服务器的资源。多台后端服务器可以连接同一个数据库，使用同一台 Redis 缓存服务器，并避免过度依赖本地文件缓存。如果需要使用本地缓存，可以添加自动刷新机制来保证数据的一致性。
2. 集群化改造：对后端应用进行集群化改造，将负载均衡器引入系统中。可以使用软件解决方案如 Nginx、HAProxy、LVS，也可以选择硬件解决方案如 F5。在 TCP 层面进行四层负载均衡，或者使用应用网关来进行七层负载均衡。现代负载均衡架构中，HTTPS 通常也会使用应用网关来进行解包和封装，以减轻后端服务器的 CPU 压力，提高系统的总体容量。

#### No.15：应用网关在架构中的价值在哪里？

应用网关能执行 TLS 卸载、身份验证、指标收集、数据压缩与转换等工作，可以优化后端架构并增强系统容量和鲁棒性。其主要功能包括：

1. 作为 API 统一入口，简化客户端与后端服务的通信；
2. 实现灵活路由配置和负载均衡策略，提高系统性能和可扩展性；
3. 集成认证和授权机制，集中管理和保护后端服务的安全性；
4. 实现请求协议转换，提高系统灵活性和兼容性；
5. 实施缓存机制和限流策略，优化后端服务负载和响应速度；
6. 进行全局请求监控和日志收集，帮助团队发现问题、分析性能瓶颈，优化系统。

#### No.16：常见应用网关有哪些区别？

不同的应用网关在设计重点上存在差异，对比如下：

1. OpenResty 是一个高度自定义的应用网关开发框架，允许用户自行编写代码进行开发。它具有较大的自由度，但基础插件较少，需要投入较多的开发资源才能正常使用。
2. Kong 是一个基于 OpenResty 的商业开源应用网关，拥有丰富的插件和完善的功能。它基于 PostgreSQL 数据库，可以独立部署而无需依赖 Kubernetes 环境，也可以使用云数据库服务。Kong 适合那些开箱即用、流量不是特别高的业务场景。
3. Zuul 具有最强大的流量过滤能力。它可以对请求进行精细化的路由和过滤，确保请求能够正确地转发到相应的后端服务。
4. Spring Cloud Gateway 与 Spring Cloud 生态紧密配合，易于开发和扩展。它适用于 Java 生态系统的微服务系统，可以与其他 Spring Cloud 组件无缝集成。

## 第 6 章 使用软件定义网络（SDN）技术搭建大规模负载均衡集群

软件定义网络（SDN）是一种新兴的网络架构，它将网络控制平面与数据平面分离，使网络管理员能够通过编程的方式对网络进行配置修改和动态管理。在大规模负载均衡集群中，SDN 技术可以实现更高效的资源分配和流量管理。

而我们的目的更加纯粹，降低大规模负载均衡集群的搭建成本，把高端负载均衡器的价格打下来。

## 6.1 概述

---

负载均衡是 SDN 技术的一种十分常见的应用，它可以非常方便、灵活地将网络流量分配到多个服务器或链路上，提高网络性能和可用性。

### SDN 的运行方式

1. 首先，SDN 控制器负责收集整个网络的状态信息，包括交换机、路由器、服务器等设备的配置和性能数据。这些信息可以通过各种协议（如 OpenFlow、BGP-LS 等）从设备上报给控制器。
2. 当有新的流量进入网络时，SDN 控制器会根据预先设定的策略（如最小连接数、最短路径等）为每个数据包选择一个合适的服务器或链路进行处理。这个过程通常称为“路由决策”。
3. SDN 控制器会将选定的服务器或链路的信息发送给交换机或路由器，指导它们将流量转发到相应的目标。这个过程通常称为“流表操作”。
4. 当网络中的某个服务器或链路出现故障时，SDN 控制器可以实时检测到这个问题，并重新调整流量分配策略，把流量转发到健康的服务器或链路上，保证网络的正常运行。

### 设定目标：一百万 QPS

假设我们的静山平台继续迅猛发展，短时间内已经跻身头部平台，将参加今年的双 11 大促，到了活动开始的那一秒，订单数肯定要破万，所以我们需要搭建一个能支持一百万 QPS 的负载均衡架构。

## 6.2 负载均衡发展史

---

接下来，让我们一起回顾负载均衡技术的发展历程。

### F5 创业史

1996 年，华盛顿大学的几个学生共同创建了一家生产负载均衡设备的公司，并以美国人民的老朋友——飓风的最高等级 F5 作为品牌名，以表示他们的设备可以扛住最狂暴的网络流量。当时，互联网的规模每 100 天增长一倍，这也让 F5 在成立三年后迅速上市。2001 年，F5 公司在经历了互联网泡沫后，顺利地把设备卖进了银行等大型机构，因为 F5 比微软和甲骨文都更有前瞻性：他们的 iControl 系统可以提供 API，让大型机构自己开发软件来控制通过负载均衡设备的所有流量。

早在 2001 年，软件的威力就已经开始展现。

### TMOS 软件平台

2003 年非典暴发，电子商务、网络点餐甚至是网络新闻都迎来了爆发式的增长。2004 年，F5 上市了新一代产品：包含 TMOS 软件平台的全套硬件设备，一次性解决了网络访问、数据中心同步访问、远程办公、应

用防火墙等多种需求。此后，市场上 F5 公司的产品越来越具有统治力。

## 迈向应用交付

2006 年以后，传统的 IP 层负载均衡技术开始向应用层发展，F5 等相关设备厂商也紧跟潮流开始推广“应用交付”的概念，开发了很多结合了负载均衡和应用网关的产品，这之后的十年，是传统负载均衡硬件厂商的最后荣光。

2019 年，F5 Networks 以 6.7 亿美金的价格收购了 Nginx，硬件厂商和软件厂商实现了一次梦幻联动，也侧面说明了我们确实已经迎来了软件定义网络的大时代。

## 顺便提一句防火墙

实际上，在今天的企业网络架构中，专业的网关设备都已经消失，取而代之的是一个看起来不是网络设备的网络设备：防火墙。特别是具备应用识别能力的“下一代防火墙”（没错，就是这个中二的名字），更是将防火墙设备的价值推上了巅峰，并一次性让企业级路由器和中低端负载均衡器全部退出市场，这又是一个软件战胜硬件的故事。

一个小八卦，下一代防火墙行业的个中翘楚 Fortinet 公司，由两名出生于北京的清华大学老师的孩子于 2000 年在美国创办。

## 负载均衡一代目：硬件负载均衡

2002 年 2 月，戴尔为 PowerEdge 1650 服务器首次配备了千兆以太网。当时，负载均衡的主流实现是基于硬件的，也可以说是一种“软硬件一体化解决方案”。由于当时的服务器 CPU 单核性能较低，核心数也较少，网卡芯片技术远不如今天的 NPU 强大，因此想要通过运行在标准操作系统（Windows/Linux）内的软件来实现千兆软网关仍然是一个“前沿探索项目”。更不用说万兆负载均衡了：PowerEdge 1650 发布后四个月，万兆以太网的首个技术标准“IEEE 802.3ae 10 Gb/s 光纤以太网标准”才首次发布，距离万兆网卡在服务器端普及还有十多年的时间。

鲜为人知的是，千兆以太网的光纤标准于 1998 年发布，双绞线标准于 1999 年发布，而到了 2002 年，万兆以太网光纤标准已经发布。实际上，千兆以太网在消费端开始普及已经是 2010 年的事情了。

在 21 世纪的第一个十年里，最优秀的超千兆解决方案是利用二层网络的链路聚合协议，使用多个千兆口同时进行负载均衡，实现超过千兆的速度。而且当时能够提供数 G 带宽的负载均衡设备价格昂贵，动辄上百万，对于有需求的终端客户来说是个巨大的负担。然而，在那个移动通信基站都需要完全进口的时代，哪个中国人不是背负着巨大的负担呢？

这些网络硬件厂商实际上也是软件厂商，只不过它们选择将自己的软件安装在这些黑色的铁盒子里出售，这样的产品质量更稳定，更重要的是：这样更贵。相比于购买一个看起来可以随意复制盗版又虚无缥缈的软件，人类社会落后的官僚体系更容易接受购买昂贵的实物。具体的技术分析请继续阅读。

## 负载均衡二代目：软件负载均衡

近年来，随着云计算的兴起，硬件设备和 IT 基础设施发生了翻天覆地的变化。如今，硬件负载均衡逐渐退出舞台，取而代之的是云服务商们利用软件定义网络（SDN）技术构建的低成本高性能的新世界。

今天，一台总价 3 万元的通用 x86 服务器搭配 100G 以太网卡，使用基于 DPDK 开发的用戶态应用程序在 Linux 上发送小包，轻松就能达到 100Gbps 的网卡带宽。（来自 Envoy 作者 Matt Klein 2017 年的一篇英文博客：[Introduction to modern network load balancing and proxying](#)）



图 6-1 Brian Glas 关于软件定义网络的推文

如图 6-1 的推文或许有些夸张，但这正是我们当下的现实：软件正在重新定义网络的一切。

## 价值百万的硬件设备



**F5 VPR-LTM-C2400-AC**

设备类型：负载均衡器

硬件配置：处理器：英特尔四核Xeon处理器（共8个超线程逻辑处理器内核 内存：32GB 硬盘：400GB SSD

性能应用概述：每秒L7请求数：1M 每秒L4连接数：400K 每秒L4 HTTP请求数：7M 最大L4并发连接数：

¥98.5万

2022-12-21

5家商家报价

[查询底价](#)

硬件配置	处理器：英特尔四核Xeon处理器（共8个超线程逻辑处理器内核） 内存：32GB 硬盘：400GB SSD 接口：标配8个万兆/千兆位光纤端口，最大32个 硬件SSL：标配4000/Blade，最大10000 TPS(2K keys)，9Gbps批量加密 硬件DDoS保护：每秒40M SYN cookies 高度：4U 服务器缓存：缺省提供内存仿真cache，提供静态页面加速，降低后台服务器压力 电源：标配两个交流100–240 VAC（800W），50/60Hz自动测距，每插槽电流10A（最大），直流电源（可选）
性能应用概述	每秒L7请求数：1M 每秒L4连接数：400K 每秒L4 HTTP请求数：7M 最大L4并发连接数：24M L4吞吐量：40Gbps L7吞吐量：18Gbps 最大软件压缩：10Gbps 软件架构：64位TMOS 虚拟化支持：每个刀片8个，最大4个刀片 支持的虚拟服务器数量 VIP：无限制 Real Server：无限制 IP Version：支持ipv4 和ipv6 协议层保护：提供HTTP (S),FTP(S), SMTP等协议层安全 安全的管理：可以通过HTTPS、SSH进行安全的远程管理，本地可以通过CONSOLE终端进行管理

图 6-2 F5 VPR-LTM-C2400-AC 的技术参数

如图 6-2 所示是一台 F5 生产的售价百万的硬件负载均衡设备，仅凭一颗 4 核 8 线程的至强 x86 CPU，就实现了 40G 的四层负载均衡能力和 18G 的七层负载均衡（应用网关）能力。

这台设备内部有两个控制器和四个接口插槽，实现了“全双活”，即控制流量的“CPU 内存主板”和收发流量的“网络接口”都是双份。当任意一个资源意外宕机时，另一个备用资源可以无缝接替，实现无丢包的硬件级高可用性。同时，这台设备背后的电源适配器至少有四个，支持运行时热替换，甚至连风扇模组也是冗余、可热替换的。

接下来，我们将利用软件的力量，逐步在标准 x86 服务器上运行的标准 Linux 系统内，构建出一个与这台硬件负载均衡设备具有相同高可用性、且带宽可达 200G 的负载均衡集群。

让我们从交换机技术开始讲起。

## 6.3 交换机

大家还记得上一章中出现过的的负载均衡架构图吗？图 6-3 中就暗藏了一个交换机。

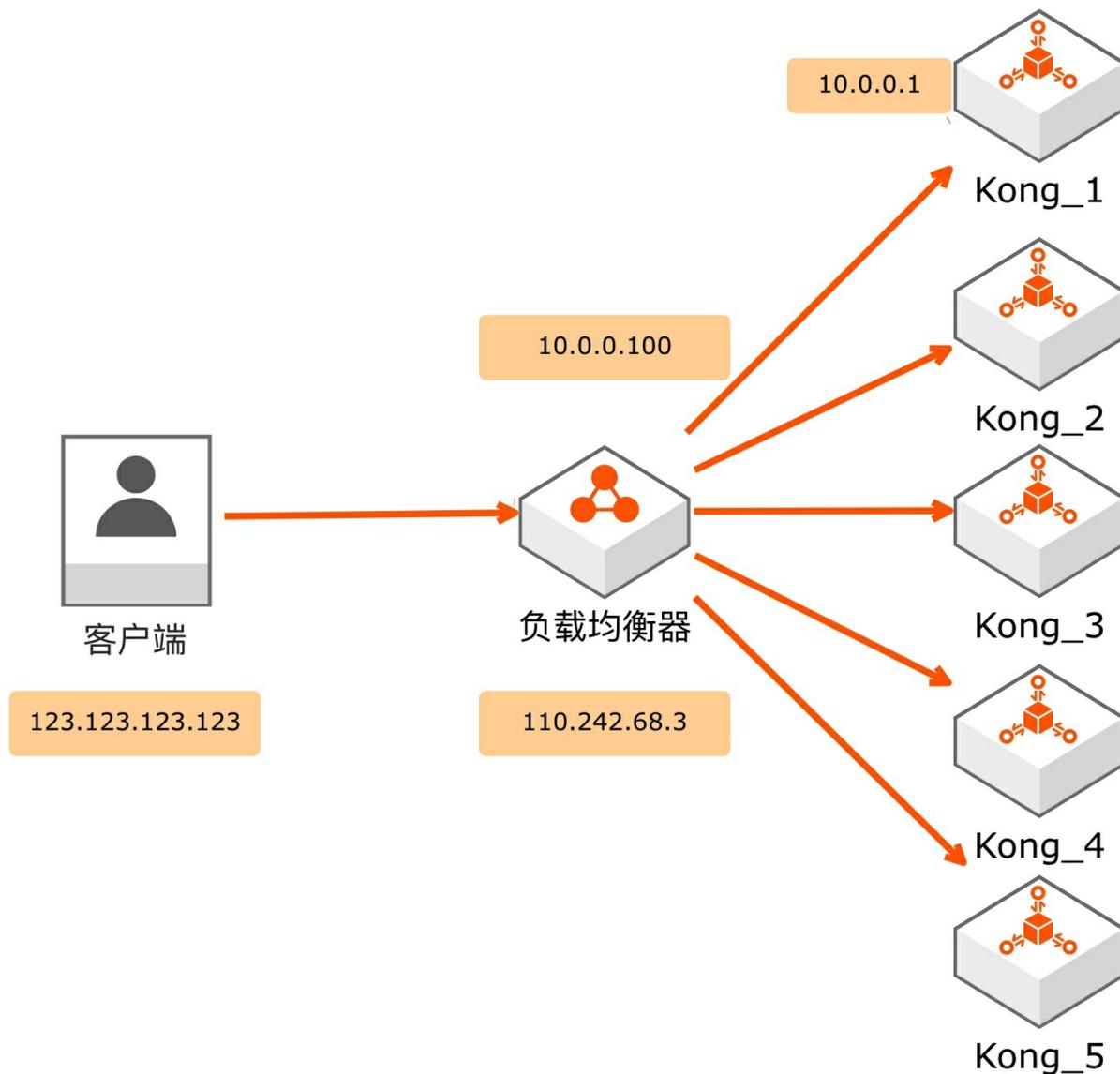


图 6-3 使用 TCP 负载均衡器时的架构图

在左侧，客户端和负载均衡器之间使用公网 IP 进行通信。它们就像是全球互联网中的两台对等设备，只是数据包经过了许多真正的“路由器”的“路由”操作，才能互相接收到对方发送的数据包。

而在右侧，负载均衡器和上游服务器之间的通信则采用了内网 IP：10.0.0.100 和 10.0.0.1。那么，它们是如何进行通信的呢？答案就是通过交换机。

### 路由器和交换机巨大的价格落差

一台能够实现千兆 NAT 功能的路由器，最低价格大约在 300 元。然而，一个所有端口都能同时实现全双工千兆（上下行同时达到千兆）的五口交换机，你知道其价格是多少吗？仅需 39 元，还包邮。

这是因为交换机所承担的任务相对简单，可以使用较低端的芯片来满足需求。

### 交换机的工作原理

网关通过关联两个五元组并对数据进行修改来实现其功能。而交换机则更为简单：它只需维护一个 MAC 地

址与网口之间对应关系的哈希表，无需对数据进行任何修改。该哈希表的键为 48 位长，值为 tinyint：对于四口交换机，可用 2 位表示；对于八口交换机，可用 3 位表示。现在，你是否明白了为什么家用交换机通常都是八口的呢？因为节省一位存储空间，就可以降低一些成本。

交换机的工作模式可以简要描述如下：

1. 当交换机收到一个 MAC 包时，它会查询目的 MAC 地址所在的网口。
2. 如果查询不到，交换机会将该 MAC 包发送到所有端口，包括接收到数据包的端口。
3. 一旦该 MAC 包得到响应，交换机就会学习到该 MAC 地址与端口之间的关系，下次就不需要向所有端口发送了。
4. 如果查询到了，交换机会将数据包单独发送到目标网口。
5. 在整个过程中，交换机不会对数据包进行任何修改。

除了不需要修改数据包之外，交换机也不需要与其他设备通信来建立和更新 MAC 表，只需监听途径交换机的所有数据包的源 MAC 地址即可。

## 交换机技术的优缺点

优点：

1. 由于足够简单，硬件成本可以保持在较低水平。
2. 无需与任何设备通信即可支持新设备的接入，完全自学习。
3. 扩展性极强，交换机可以随意级联，只要 MAC 表容量足够，理论上可以扩展成一颗无限层级的树。

缺点：

1. 存在网络风暴的风险：数据包被无脑地发送到所有端口，可能会被其他交换机再次发送回来，导致正常数据包拥堵。
2. 网络回环非常危险：如果将交换机的两个端口用一根线连接在一起，这台交换机将立即断网，对于这种低层级设备来说，“左右互搏之术”过于困难。

当然，目前主流的商业二层交换机已经具备了许多三层特性，这些缺点通过各种检测技术已经得到了解决。

大家不要认为交换机原理没有用处，SDN 是一种跨越一二三层的技术，物理层（MAC 层）也是我们构建 200G 负载均衡集群的重要技术战场。

准备工作已经完成，下面我们开始搭建负载均衡集群。首先从这一切的基础——LVS 开源软件说起。

## 6.4 LVS 技术解析

本节将深入剖析 LVS 软件的技术原理，探索其设计思想，并追踪数据包的修改和传递路径。

### 章文嵩博士创造 IPVS

1998 年，章文嵩正在读博士二年级，他发现 Cisco 的硬件负载均衡器售价昂贵，于是利用了两周的课余时间

间，创建并开源了 LVS（当时称为 IPVS）。如今，LVS 技术所创造的商业价值已无法估量，互联网上的绝大部分数据包都由 LVS 或承袭 LVS 思想的软件处理。

## 融入 Linux Kernel

2004 年，LVS（IPVS）被纳入了 kernel 2.4，从此之后，所有 Linux 系统都具备了变身为负载均衡器的能力。

## LVS 基本原理

LVS 的基本原理可以用一句话概括：通过修改 MAC 层、IP 层、TCP 层的数据包，实现了一部分交换机和网关的功能，从而指挥流量到达真正的服务器上。

LVS 有三种常用模式：

1. NAT 模式：即网关模式，双向流量均经过网关转发，性能开销最大。
2. TUN 模式：类似于单臂路由，性能高且可跨越机房。
3. DR 模式：仅适用于局域网，但性能惊人，因为回程流量不返回网关，直接在局域网内传输给客户端。

我们以最能体现 LVS 思想的 DR 模式为例，展示 LVS 的基本原理。

## DR 模式数据包推演

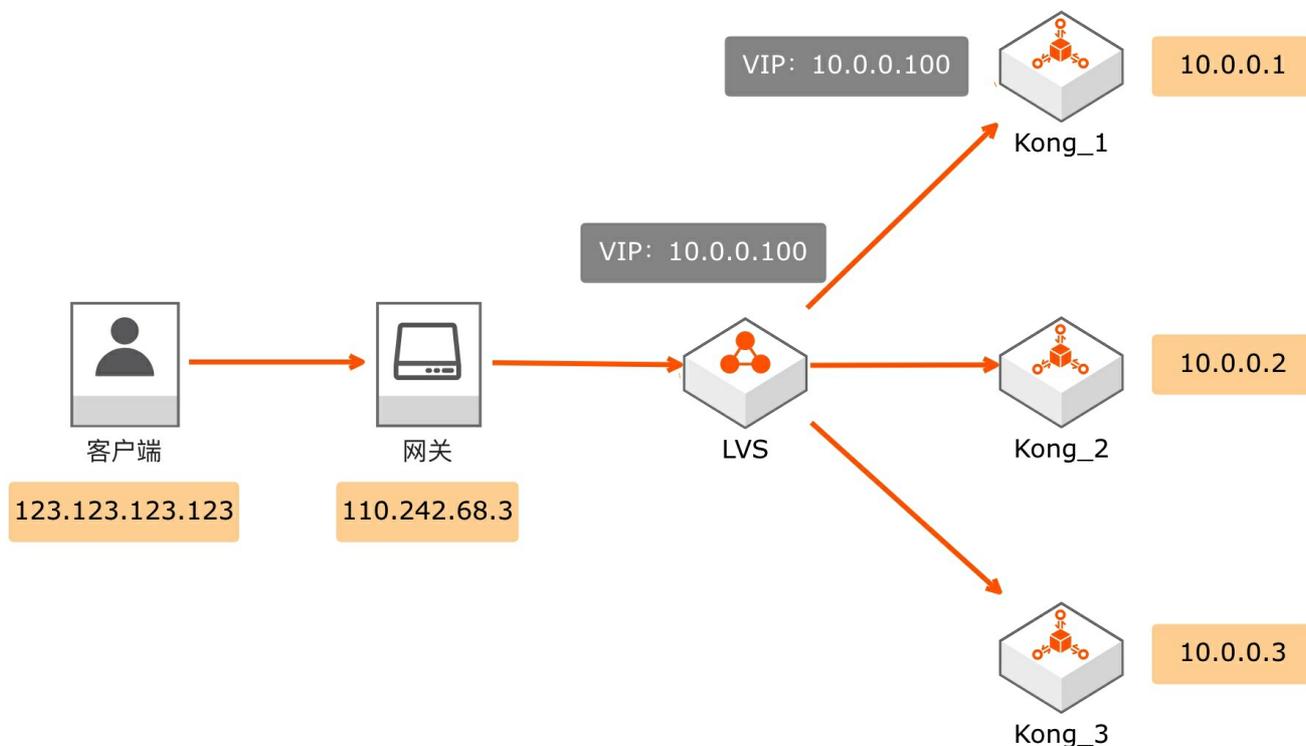


图 6-4 LVS DR 模式架构图

LVS 在 DR 模式时，运行架构如图 6-4 所示。在 DR 模式下，LVS 只负责修改数据包，不充当网关的角色。因此，我们仍然需要一个网关来执行公网 IP 和私网 IP 之间的 NAT 转换。假设客户端 IP 为 123.123.12

3.123, 它向 110.242.68.3 的 80 端口发起了一个 HTTP 请求。

当网关接收到一个发送给 110.242.68.3 的数据包时, 发现协议为 TCP, 目标端口为 80。通过查询自己的 NAT 表, 网关发现内部 IP 为 10.0.0.100 (即虚拟 IP), 内部端口为 80。于是, 网关向局域网发送了一个 IP 包。由于端口和协议保持不变, 本次网络请求中的关键数据有四个: 源 IP 地址、目的 IP 地址、源 MAC 地址和目的 MAC 地址。

1. 客户端发给网关的数据包情况为:

1. 源 ip: 123.123.123.123
2. 目的 ip: 110.242.68.3
3. 源 MAC: 客户端 MAC
4. 目的 MAC: 网关 MAC

2. 网关向局域网发出的数据包情况为:

1. 源 ip: 123.123.123.123
2. 目的 ip: 10.0.0.100 (变了)
3. 源 MAC: 网关 MAC (变了)
4. 目的 MAC: LVS MAC (变了)

3. LVS 接到该数据包后, 会选择一个后端服务器, 假设它选中的是 10.0.0.1 来真正处理请求, 则 LVS 会对数据包进行如下修改后, 再发送给 10.0.0.1:

1. 源 ip: 123.123.123.123
2. 目的 ip: 10.0.0.100
3. 源 MAC: LVS MAC (变了)
4. 目的 MAC: 10.0.0.1 的 MAC (变了)

4. 10.0.0.1 在收到该数据包后, 发现这个包的目的 MAC 地址确实是自己, 而且目的 ip 10.0.0.100 (VIP) 也是自己, 于是对该数据包进行正常的处理, 然后将处理结果发送出去:

1. 源 ip: 10.0.0.100
2. 目的 ip: 123.123.123.123
3. 源 MAC: 10.0.0.1 的 MAC
4. 目的 MAC: 网关 MAC (因为目的 ip 不在“ip+子网掩码”所确定的局域网范围内, 所以该数据包会被发送给网关)

5. 网关收到返回的数据包后, 通过查询“五元组关系表”, 对端口和 ip 信息做出正常的 NAT 修改后, 将数据包发送回 123.123.123.123, 请求结束。

## DR 模式的特点

通过前面的推演过程, 大家可以看出 DR 模式的特点:

1. LVS 只需处理正向数据包, 通常正向数据包 (请求) 远小于反向数据包 (响应), 因此带宽占用较低。
2. 反向数据包通过标准的二层以太网传输, 每台上游服务器都能达到自己的线速 (网络设备接口协商速率, 理论上的最大数据传输速率)。
3. 只能在相同的二层网络下工作 (即同一个局域网), 架构上有局限, 同时安全性较差。

4. 需要在每台上游服务器上将 VIP (10.0.0.100) 配置为 lo (本地回环) 接口的 IP。
5. 需要让每台上游服务器只响应真实 IP (如 10.0.0.1) 的 ARP 查询请求, 如果不小心回复了针对 VIP 的 ARP 请求, 将会引发混乱: 局域网内有多台机器同时声称自己拥有 10.0.0.100 这个 IP, 交换机会崩溃。

在生产环境部署中, 由于 LVS 集群是所有流量的入口, 所以其可用性需要非常高, 一般不会只部署在一个机房里, 因此最常用的是 NAT 模式: 双向流量都经过 LVS 集群, 这样可以实现多地多中心的跨公网多活。

## LVS 设计思想

通过上述过程, 你应该能理解 LVS 的运行原理: 它通过在 LVS 和上游服务器上配置虚拟 IP, 以修改数据包后再发送为手段, 在标准以太网模型下构建了一个可行的负载均衡系统。它不像交换机那样完全不修改数据包, 也不像网关那样维持一个对应关系并修改很多东西, 它修改了数据包, 但不多, 因此可以实现非常高的性能。

## 内核态

LVS 的数据处理组件 IPVS 运行在内核态, 避免了用户态 IPVS 进程和内核态 Linux 网络进程之间频繁的状态切换导致的内存读写开销, 在高并发下这种设计可以带来非常高的性能收益。由于有内核态支持, LVS 比 HAProxy 和 Nginx 的单机性能都要强很多。

2014 年, eBPF 首次被引入了 Kernel 3.18, 它的出现让新时代的 LVS 不再需要合并进内核才能使用内核态, 让普通软件也能直接进入内核态, 为高性能软件打开了一扇大门。eBPF 是目前最热门的内核相关技术, 它为我们自行开发属于自己的内核态软件提供了可能——除了合并进内核之外, Kernel 的用户有了自己“热加载”内核代码的能力。

## 专业的负载均衡协议: OSPF/ECMP

LVS 是运行在标准以太网模型下的负载均衡软件, 配合 Keepalived 可以实现高可用。而 OSPF/ECMP 协议是专业的多链路路由协议, 可以实现不丢包的多活。

OSPF: 开放式最短路径优先协议, 一种基于链路状态的内部网关协议。每个 OSPF 路由器都包含了整个网络的拓扑, 并计算通过网络的最短路径。OSPF 会通过多播的方式自动对外传播检测到的网络变化。

ECMP: 等价多路径协议。当存在多条不同的链路到达同一目的地址时, 利用 ECMP 可以同时使用多条链路, 不仅增加了传输带宽, 还可以无时延、无丢包地备份失效链路的数据传输。如果使用传统的路由算法, 只能利用其中的一条链路进行数据的传输, 还存在丢包的可能性。

## LVS 拆分了网关单点

LVS 是网关型负载均衡继续拆单点的结果: LVS 将网关这个单点拆分成了“重定向”和“转发”, 自己只承担数据包重定向工作, 将转发留给基础网以太网来解决, 在单机上实现了非常高的系统容量。在最新的 x86 服务器上, 单个 LVS 即使在开销更大的 NAT 模式下也可以实现大约 20G 的 TCP 带宽——[用 dperf 测试 LVS 的性能数据](#)。

## 6.5 Keepalived 高可用

LVS 单机 20G 的带宽显然和我们 200G 的目标还相去甚远，但是我们接下来要做的第一件事并不是提升系统容量，而是先提升系统的稳定性：搭建高可用架构。

Keepalived 是一款出色的开源软件，专为与 LVS 配合而设计，主要功能是构建自选举集群。它支持独立部署三台机器用作控制器集群，也可以将控制器部署到应用机器上。与 LVS 一样，基于虚拟 IP 技术，可在任意标准以太网内运行。

### Keepalived 运行原理

Keepalived 的运行原理简单明了：

1. 两台机器配置同一VIP（虚拟 IP）：即两台机器的真实 IP 分别为 10.0.0.101 和 10.0.0.102，但虚拟出一个 IP 10.0.0.100，不属于任何物理设备，可在任何机器间切换绑定。
2. 两机频繁通信，通过分数计算确定哪台机器得分更高，由该机器向局域网发送 VRRP 组播报文宣称 VIP 在我这里。
3. 当得分高的主机宕机、断网或检测到服务进程消失（如 Nginx 挂掉），得分低的机器会在极短时间内接替，宣称 VIP 在自己这里。
4. 发送给 VIP 的数据包会在短时间失效后由新机器承接（实测中断时间小于 1 秒），实现集群高可用。

### 高可用 LVS 集群

LVS 与 Keepalived 结合，基于多台物理机，可实现高可用负载均衡集群，架构如图 6-5 所示。

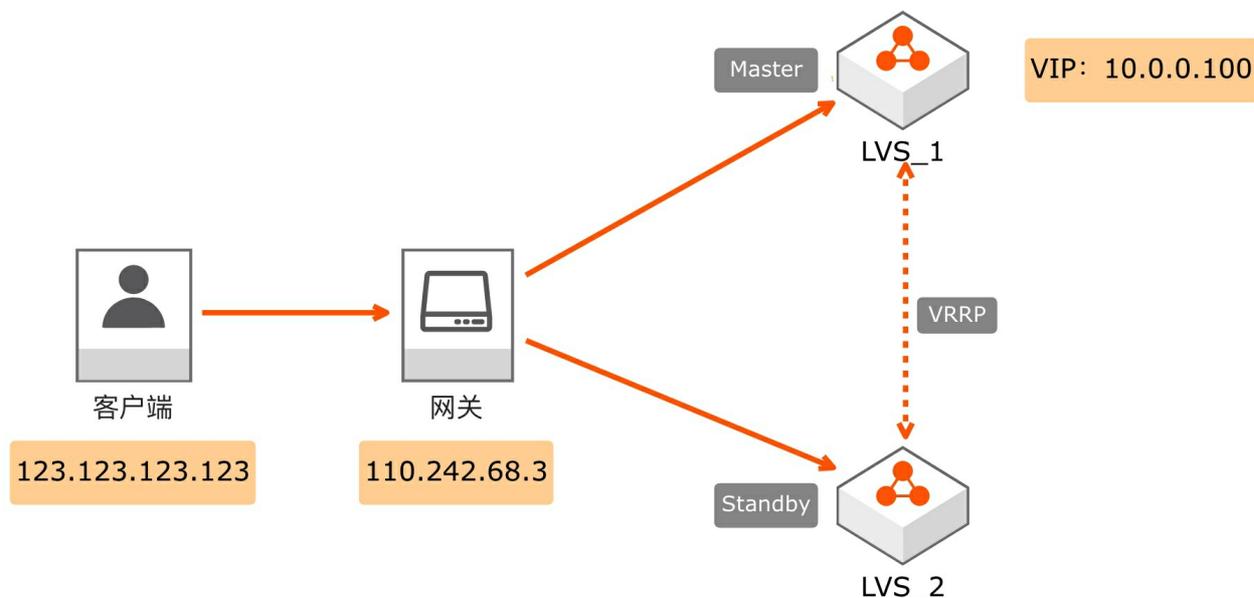


图 6-5 LVS 配合 Keepalived 实现高可用时的架构图

若 Master 因任何原因失效（硬件故障、断电、误操作、光缆挖断、火灾地震等），Standby 机器将迅速接替，接管流量，实现高可用目的。

金融级项目可在一城市跨三个机房做三节点 Keepalived 集群。为何不跨城市？因跨城市属于“两地三中心”高可用技术领域，一般不在二层网络上进行，跨城市专用光纤建设费用高昂，城市内拉光纤更经济实惠。最后一章将探讨终极高可用架构。

## Keepalived 让众多服务高可用

笔者在公司办公区的自建机房部署了一些非关键业务，性能要求不高，故选择单独部署 Keepalived：与 Kong 网关配合提供高可用网关；与 MySQL 配合做双主双活集群；与 Kubernetes 配合做了一个跨越四台物理机的容器集群，并提供高可用服务等。

2023 年 7 月，笔者公司机房发生了空调漏水事故，损坏两台物理服务器。由于提前搭建了三节点 Keepalived 集群，核心服务成功幸存了下来。这是笔者首次经历物理服务器和磁盘全部损毁的事故，Keepalived 经受住了考验！

## 6.6 突破单台服务器的性能极限：从 20G 到 200G

借助 Keepalived，我们成功搭建了高可用的 LVS 集群。现在，让我们来攻克下一个难题：将单台服务器的性能从 20G 提升至 200G。

### LVS 单机性能为何会卡在 20G

最新的 AMD EPYC™ 9654 服务器 CPU 拥有 192 个物理核心，双路平台共有 768 个 vCore，然而 LVS 单机性能却停滞在 20G 附近。为何无法进一步提升性能呢？原因在于 Linux 网络栈已经达到了性能极限。

### Linux 网络栈优化

由于 LVS 基于 Linux 内核中的 Netfilter，依赖于 Linux 网络栈，数据包的发送和接收本身就需要读写内存，而用户态和内核态的转换需要上下文切换，还需要读写内存。在高带宽需求下，相对耗时的内存读写成为了阻止性能进一步提升的主要障碍。DPDK 和 NPU 硬件卸载是解决这一问题的两个方案。

### DPDK

DPDK（Intel 开源的高性能网络数据处理框架）通过申请大页内存和轮询代替中断这两个关键特性，为高速率网卡提供了一种高性能解决方案。其 CPU 亲和性、多核调度架构以及内存 NUMA 优化等基础架构进一步推高了性能，使其成为用户态网络界面框架的首选。

爱奇艺开源的 DPVS<sup>👉</sup>就是 DPDK 技术在负载均衡领域的成功应用。在 10G 网络下进行小包测试，DPVS 的性能是标准 LVS 的五倍。在这里我们保守一点，将其折半为 2.5 倍，那么 DPVS 的单网卡性能极限就是 50G。

### 网卡芯片硬件卸载

最新的 NPU（网络处理器单元）已经支持了许多硬件卸载特性，包括 IP 分片、TCP 分段、小包重组、checksum 校验、硬件 QOS，以及最重要的 VxLAN（虚拟扩展本地局域网）的剥离和插入。这些功能是 DPV

S 的重要组成部分，可以减少数据流之间的干扰，大幅提升系统总容量。

为什么说 VxLAN 最重要？因为它就是当前云服务商给每个租户搭建 VPC 所使用的最主流的技术。

此外，RDMA（远程直接内存访问）技术也是网卡芯片的一个重要发展方向，我们将在后面详细探讨数据库计算和存储分离时再深入介绍。

## 全局锁优化

除了 Linux 网络栈的限制，LVS 本身架构上的全局锁也是一个突破口。全局锁导致了海量的 CPU 核心无法被利用。我们可以借鉴阿里云的处理思路。

## 数据包亲和性优化

阿里云通过 RSS 技术将同一个五元组报文扔到同一个 CPU 上处理，确保入方向的所有相同连接上的报文都能交给相同的 CPU 处理。同时，在每个核转发出去时都使用当前 CPU 上的本地地址，并通过设置一些 fdir 规则，使报文回来时后端服务器访问的目的地址与对应 CPU 上的本地地址匹配。这样就能实现同一连接上左右方向的报文都被同一个 CPU 处理，将存储“五元组对应关系”的内存数据库在不同的 CPU 核心上隔离开，从而实现整体系统容量的线性提升。参考资料：[高性能负载均衡设计与实现](#)。

## 性能问题需靠架构解决

当单网卡性能达到极限 50G 时，如何将集群性能提升至 200G 呢？插入四块网卡可行吗？理论上可行，但现实中如此大流量的系统必须借助架构来提升容量，因为流量不可能凭空一分为四，此外还需要解决高可用问题。

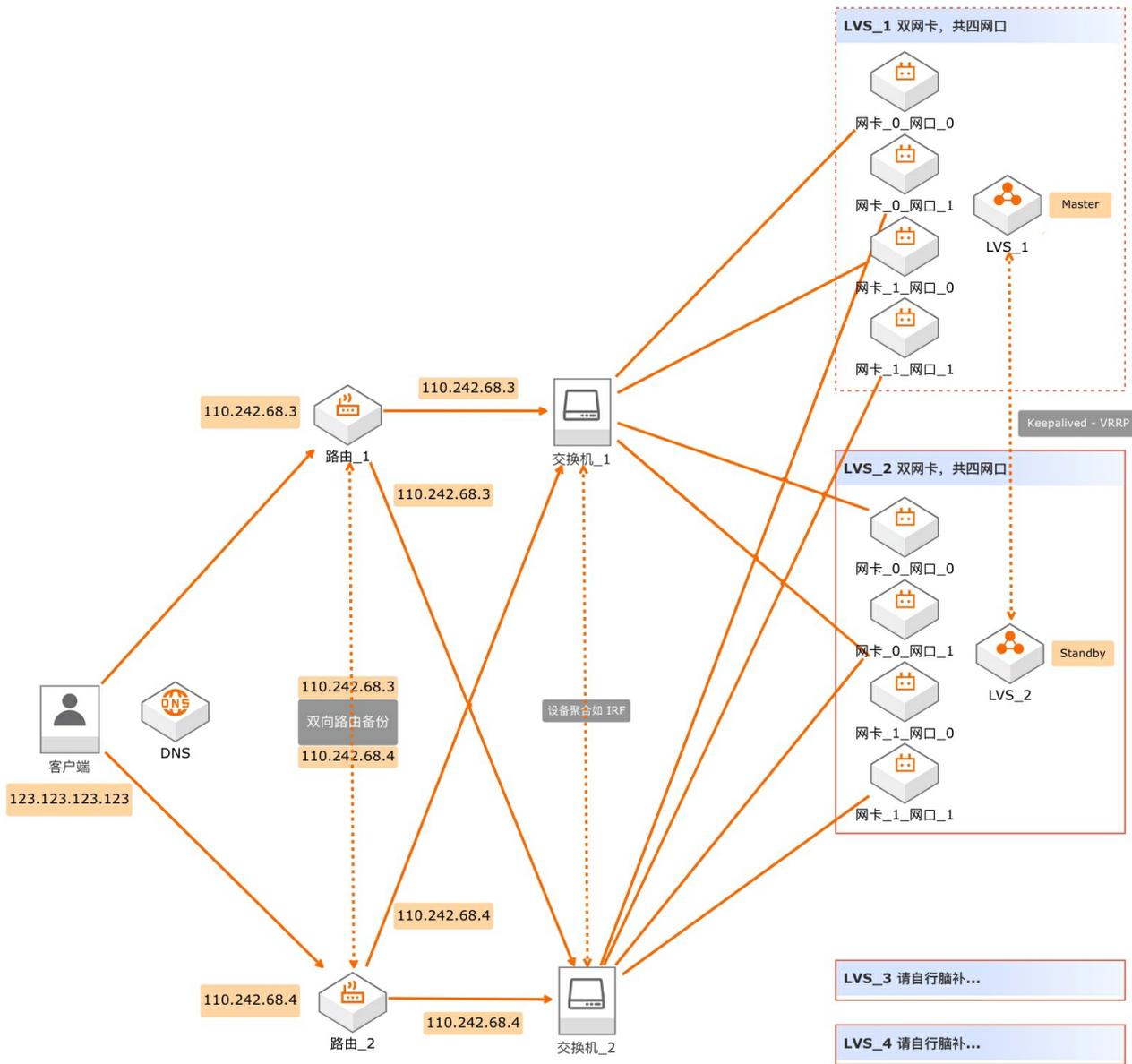


图 6-6 200G 负载均衡集群架构图

如图 6-6 所示就是笔者设计的 200G 负载均衡集群的架构图，下面我们从左到右解析该架构。

### 1. 拆除 IP 单点：朴素的 DNS

基于 DNS 技术的单域名多 IP 扩容技术曾是第一代负载均衡技术：朴素的 DNS 协议可将用户终端流量直接导向全国多个机房，实现真正的性能倍增，尤其适合 Web 1.0 时代的静态网页和搜索引擎等无数据同步需求的业务。

在当年电信、网通、铁通、教育网之间只有 40G 小水管的年代，DNS 技术极大地提升了普通用户获取网络资源的速度，提升了用户体验：引导每个运营商的宽带用户访问放在该运营商机房的服务器，可实现原始的“手搓 CDN”架构。最后一章我们还会谈到 DNS 技术在终极高可用架构中的价值。

经过 DNS 拆分，每个路由拥有一个公网 IP，承载 100G 带宽。需要注意的是，此处的路由真的是路由，并非网关，它只需告诉每个数据包的下一跳（该去何处）是哪个 IP 即可。硬件路由设备的性能非常强大，承受住 200G 带宽甚至不需要高端数据中心核心设备。此架构中的网关由后面的 LVS 充当。

## 2. 拆除交换机单点：OSPF/ECMP 路由技术

上图中的 [路由\\_1](#) 和 [路由\\_2](#) 采用支持 ECMP 的硬件设备或软件路由来充当，它们会将流量平均分配给两个交换机。每台交换机承载 100G 带宽，对交换机来说简直是小菜一碟，2 万人民币的硬件交换机就能实现 24 口 x 100G 的全线速交换。

## 3. 拆除服务器单点：LVS 单机双网卡四网口

每台 LVS 服务器都安装双口 100G 网卡 2 张，共有四个网口，这样单机可以实现 50G x 2 的极限性能。每个网卡上的两个网口分别连接两台交换机，既实现了高可用性，又保证了协议速度（线速）不会成为瓶颈。

## 4. 天下无单点：全冗余架构

大家可以看到，从左至右，整个架构的每一层的每一个节点都与左边一层的所有节点进行连接，这种全冗余架构可满足任意一台设备宕机时整个系统仍可用，甚至系统容量都能保持不变。

如果某台路由宕机，则另一台路由会将其公网 IP 接过来，实现分钟级故障恢复（公网路由表更新较为缓慢）。

如果某台交换机宕机，则左侧路由通过 ECMP 及时调整路由配置，实现秒级切换。

如果某台 LVS 服务器宕机：Keepalived 机制会让 Standby 设备在 1 秒内顶上。

## 系统容量计算

对于整个系统而言，四台安装了双网卡的 LVS 服务器（2 主 2 备）在两个公网 IP 入口的情况下，可以实现总带宽为  $50G \times 2 \times 2 = 200G$ 。即使其中任意一台设备宕机，也不会影响系统的总容量。

然而，200G 并非该系统的极限。如果我们让一组两台 LVS 服务器使用两个 VIP 进行互为主备的双活配置，可以将整个系统的容量提升到理论极限的 400G。接下来，我们将分析各种单项资源的性能极限：

1. 单个公网 IP：最大带宽为系统最大容量的 200G。
2. 单个 VIP：最大带宽为 100G。
3. 单个数据流：最大带宽为单网卡性能极限的 50G。

还记得之前提到的价值 100 万的硬件负载均衡器吗？它的最大 L4 带宽仅为 40G。而我们这套 200/400G 的设备需要多少钱呢？

$$1 \times 2 + 2 \times 2 + 3 \times 4 = 18 \text{万}$$

## 额外的优势

花费 18 万搭建一套图 6-6 中的设备，不仅可以构建一套 200G 的集群，其中的路由和交换机还拥有大量的容量可用于其他业务。

更重要的是：LVS 集群相对于硬件负载均衡设备来说，可以轻松控制且可编程。这对于右侧上游服务器的网

络架构规划非常有利。对于云计算厂商而言，可以开发更丰富的功能来提供更复杂的服务（例如计算和存储分离的数据库），提高硬件资源的利用率，同时可以为云平台用户提供更复杂的功能：

软件带来的无限可能才是这套配置最核心的价值！

## 6.7 殊途共归：硬件厂商的软件设计

图2 Comware V7 模块化体系结构

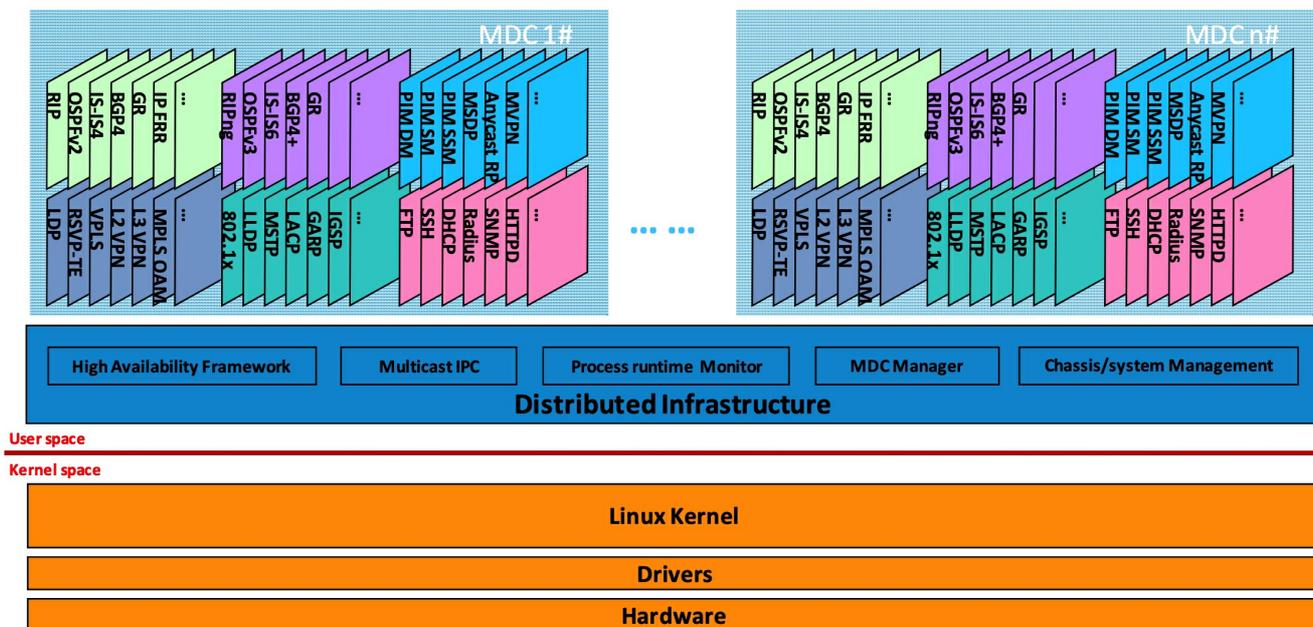


图 6-7 H3C ComwareV7 操作系统架构图

这张图中最重要的就是那根“细细的红线”：网络管理进程完全位于用户态。这正是 DPDK 思想的核心，传统的 UNIX 网络模型已无法应对单机 10G 的速度，因为需要读写内存的上下文切换的速度太慢了。为了实现更高性能，必须从底层网络数据处理流程上进行革命——抛弃操作系统提供的网络栈，直接在用户态接管所有网络流量。

为了提高软件稳定性，H3C 还在系统内开发了一些高可用技术，与我们负载均衡集群高可用的设计思想不谋而合，异曲同工：

1. 支持单播、组播和热迁移的高可靠 IPC (进程间通信) 技术：对应的是双机热备以及 OSPF/ECMP、VRRP 协议等。
2. 多进程主备选举：对应的是 Keepalived 集群选举。
3. 进程级内存备份 (实时热备进程的内存，用于快速故障恢复和主从切换)：对应的是 OSPF/ECMP 不丢包多活技术、服务器的内存镜像技术。

### 负载均衡集群无法单独支撑一百万 QPS

通过本文构建的这个 200/400G 负载均衡集群，配合多个应用网关以及后面海量的物理服务器，我们终于成功实现了一百万 QPS 的目标。然而，这只是 Web 服务层面的百万 QPS，在真实世界中，数据库才是那

个最难解决的单点，接下来我们将会用 5 章的篇幅尝试解决这个问题。

## 6.8 面试题

---

**No.17 : LVS 如何做到高性能？**

要实现高性能的 LVS，可以采取以下措施：

1. 运行在内核态：LVS 运行在内核态，避免了用户态到内核态的上下文切换导致的内存读写开销。这样可以大幅提升大流量下的数据处理能力。
2. 修改目的 IP 地址：LVS 只需要对数据包的目的 IP 地址进行修改，然后利用二层网络做基础的数据转发。这个策略让 LVS 承担了最少的计算和流量转发工作，增加了 CPU 和内存的资源利用率，从而大幅提升了单机性能。
3. 分布式调度：LVS 可以轻松地组成一个逻辑集群，可以进行分布式调度。通过利用多台物理机的资源，可以打造出一个行为一致的负载均衡集群。这样可以实现负载均衡的高可用性和高性能。

**No.18 : 负载均衡器是怎样修改数据包的？**

负载均衡器通过修改数据包的源地址、目标地址、端口号等信息来实现请求的转发和负载均衡。具体的修改方式取决于负载均衡器的实现方式和工作层级。

在网络层（第三层）的负载均衡中，如 LVS（Linux Virtual Server），数据包修改通常是通过网络地址转换（NAT）技术实现的。具体步骤如下：

1. 通过配置负载均衡器，将其设置为对外公开的入口，即公网 IP 地址。
2. 当客户端发送请求时，请求首先到达负载均衡器。
3. 负载均衡器根据设定的负载均衡算法选择一个后端服务器。
4. 负载均衡器将客户端请求中的目标 IP 地址和端口号修改为所选择的后端服务器的 IP 地址和端口号，并将修改后的数据包转发给该后端服务器。
5. 后端服务器接收到转发的数据包后，处理请求并将响应返回给负载均衡器。
6. 负载均衡器再将响应中的源 IP 地址和端口号修改为负载均衡器自身的 IP 地址和端口号，并将响应转发给客户端。

在传输层（第四层）的负载均衡中，如四层负载均衡器，数据包修改一般是通过代理方式实现的。具体步骤如下：

1. 客户端发送请求到负载均衡器。
2. 负载均衡器接收到请求后，建立一个与客户端的连接，并同时建立一个与后端服务器的连接。
3. 负载均衡器将客户端请求中的数据包直接转发给后端服务器。
4. 后端服务器处理请求并将响应返回给负载均衡器。
5. 负载均衡器将响应中的数据包直接转发给客户端。

**No.19 : Keepalived 高可用的运行原理？**

Keepalived 是一个基于 VRRP（Virtual Router Redundancy Protocol，虚拟路由冗余协议）的高可用解

决方案。其运行原理如下：

1. 虚拟路由冗余协议（VRRP）：VRRP 是一种网络协议，允许多个路由设备组成一个逻辑组，并模拟为一个虚拟路由器，提供共享的虚拟 IP 地址。在 Keepalived 中，VRRP 用于将虚拟 IP 地址关联到一个活跃的服务器节点上，其他服务器节点充当备份节点。VRRP 协议周期性地发送 VRRP 消息（包含优先级、状态信息等），通过选择优先级最高的活跃节点来分配虚拟 IP 地址。
2. 主备切换：在 Keepalived 集群中，一个服务器被指定为主服务器（Master），另外一台或多台服务器作为备份服务器（Backup）。主服务器负责处理所有请求并接收来自路由器的数据包，备份服务器则处于待命状态。如果主服务器发生故障，备份服务器中的 VRRP 进程会探测到主服务器不可达的情况，并通过 VRRP 协议选举过程将备份服务器切换为活跃节点，接管主服务器的责任。
3. 健康检查：Keepalived 还提供了健康检查机制，用于监测每个服务器的状态。通过定期发送心跳包或检查特定端口的连通性，Keepalived 可以判断服务器是否正常运行。如果某个服务器无法检测到其他服务器的心跳或检测到服务器故障，它将触发 VRRP 切换过程，将备份服务器切换为活跃服务器。
4. 负载均衡：除了故障转移功能，Keepalived 还可以实现负载均衡。通过配置虚拟 IP 地址关联到多个服务器上，来平衡流量分布。当请求到达负载均衡器时，它会根据一定的算法（如轮询、加权轮询等）将请求转发到不同的服务器节点，以实现负载均衡和提高性能。

#### No.20：如何实现负载均衡的高可用？

在系统架构中，负载均衡作为最明显的单点，其高可用能力至关重要。因此，通常需要对系统的负载均衡进行集群和高可用架构的设计。对于负载均衡来说，集群化相对简单，而实现高可用性则需要借助诸如 Keepalived 等工具来实现。具体来说主要有以下手段：

1. 多节点部署：在负载均衡环境中，运行多个负载均衡器节点。这些节点可以是物理设备、虚拟机或容器等。多节点部署可以提供冗余和故障转移能力。
2. 心跳检测：每个负载均衡器节点都应该定期发送心跳信号来检测其自身和其他节点的状态。可以使用专门的心跳检测工具或实现机制来实现这一功能。如果某个节点无法接收到其他节点的心跳信号，就会判定为故障节点。
3. 故障检测和故障转移：当某个节点被检测出故障后，需要有一个自动化机制将其从活跃节点列表中移除，并将其请求转发给其他正常运行的节点。这种故障检测和故障转移的机制可以通过负载均衡器软件或专用软件（如 Keepalived）来实现。
4. VIP（Virtual IP）漂移：为了实现高可用性，通常会使用虚拟 IP（VIP）来代表负载均衡服务。当一个节点故障时，VIP 应该漂移到一个健康的节点，以确保负载均衡服务的连续性。VIP 漂移可以通过使用虚拟路由冗余协议（VRRP）或其他技术来实现。
5. 数据同步：如果负载均衡器节点之间需要共享会话数据、连接状态等信息，则需要确保这些数据在节点之间进行同步。可以使用共享存储、数据库复制等技术来实现数据的实时同步，以保持高可用性和一致性。
6. 监控和报警：建立有效的监控系统来监测负载均衡器的运行状态和性能。及时发现故障和性能下降，并触发相应的报警机制，以便及时采取措施进行故障排查和修复。

## 第四部分 数据库高并发

第 7 章 最难拆的单点：数据库及其背后的存储

第 8 章 MySQL InnoDB 存储引擎详解

第 9 章 四代分布式数据库的变迁

第 10 章 国产分布式数据库双雄 TiDB 和 OceanBase

第 11 章 秒杀系统的两大利器——缓存与队列

### 第 7 章 最难拆的单点：数据库及其背后的存储

数据库是现代软件系统中必不可少的组成部分，它承担着数据的存储和管理任务。在高性能后端系统中，数据库的设计和性能优化是至关重要的。

前面三个部分，我们用六章的篇幅解决了 Web 服务的百万 QPS 问题，从本章开始，我们将用五章的篇幅来讲述数据库的高并发改造，并尝试构建出能支撑百万 QPS Web 服务的数据库。

#### 7.1 概述

首先要明确，这里的数据库指的是关系型数据库，即满足 ACID 原则并用 SQL 语言进行操作的持久性数据库。持久性指的是服务器突然断电，数据不会丢失。

在高并发和大规模系统中，数据库及其相关的存储设备往往成为整个系统中最难进行水平扩展和拆分的关键组件。

此外，在追求高并发的过程中，我们将不可避免地接触到内存数据库，我们将会用一章的篇幅讲述电商秒杀系统的缓存与队列如何设计。但是需要明确的是，内存数据库只是高并发架构设计的一部分，而且不是最重要的部分，高并发架构的核心永远是磁盘数据库。

## 7.2 数据库是个大单点

---

在 Web 系统中，我们经常面临这样的需求：用户需要逐个注册，ID 不能重复；订单需要逐个处理，两个订单的信息不能混淆。这实际上就是最基本的需求——排队。无论数据库如何拆分，微服务如何设计，总有一个资源必须排队等待处理，即使我们可以使用“拿现在的时间换未来的时间”的方式来优化性能（后续章节将详细介绍），排队仍然是不可或缺的。

在常见的 Web 系统架构中，关系型数据库是最大的不可分割的单点：在一个系统中，多个 API 调用产生的多种行为最终都会作用于同一张表上，只有这样才能保证系统的正常运行。这也是数据库最核心的价值所在——API 可以并发执行，而数据库必须逐个处理请求。

和 Golang 协程使用 Redis 排队、Node.js 内置的单线程队列一样，数据库才是 Web 服务的“根”，是确保最终计算结果符合预期的那个“队列”。

无论进程、线程、协程如何并发执行，数据库的自增、事务和锁都是原子性的。这种原子化能力正是和数据库配合的业务服务器能够多台并发运行的基础，也是多台服务器能够被称为“一个系统”的逻辑基础：如果两个系统的数据库不在一起，那它们就不是同一个系统，就像拼多多有 7.5 亿月活跃用户，淘宝有 8.5 亿，你不能简单地说“拼宝宝”电商系统拥有 16 亿月活跃用户一样。

这种哲学思想我们将在最后一章的终极高可用架构中再次使用。

### 数据库为何成为单点

数据库之所以成为单点，并非是其自身所决定的，而是系统架构需要数据单点的存在，而数据库正是为了满足这一需求而设计的。对于大部分需要记录到数据库中的信息而言，按照先来后到的顺序进行排队入库是不可或缺的硬性要求，否则数据就会出现错误。而对于所谓的 NoSQL 数据库来说，即使是一个简单的 ID 自增操作，也需要扫描整个表才能实现，因此它无法承担数据单点的角色。由于 NoSQL 并非为承担数据单点而设计，它注定只能作为关系型数据库提升性能的辅助工具。

关系型数据库中的“关系”二字，指的是一个表中的数据之间存在着行和列两个方向的关系，这实际上是另一种形式的“时间换空间”、“空间换时间”的思想：通过提前约束这些数据，让它们以一定的规则存储在一起，虽然写入时会稍慢一些，但调用起来却简单高效，下面是几个简单的能力示例：

1. 无需全表扫描即可瞬间找到最大 ID；
2. 即使将海量数据存储于磁盘上，仍能以极快的速度检索满足某个条件的某一行；
3. 在快速定位到某一行后可以迅速提取连续多行的数据。

## 数据库是一个极其复杂的软件

关系型数据库如同空气般充斥在后端技术中，然而可能很少有人意识到一个惊人的事实：99.9% 使用 MySQL 的系统，其业务代码的复杂度远远不及 MySQL 内部的复杂度高。

为了实现关系型数据库的四大原则，MySQL 几乎将计算机的每一种资源都发挥到了极致：进程、线程、多核、网络、寄存器、内存、机械磁盘、固态硬盘。

至此，如果我们开始深入分析 ACID 实现的细节，岂不是陷入了俗套？我们要不走寻常路，不背面试八股文，直接剖析 MySQL 的底层原理。

### 数据库的单点，究竟在哪里？

在探讨持久性和原子性时，许多技术文章都会提到 undo log、redo log、多版本并发和锁等概念。然而，笔者对此持不同看法。让我们从 MySQL 的基础功能出发来思考。

MySQL 是整个 Web 系统中唯一一个能够在意外掉电重启后保持数据不丢失的组件。那么它是如何实现的呢？答案很简单，它依赖于计算机中唯一一个断电不丢数据的部件——磁盘。因此，我们无需过多关注那些 `*do log`，只需将其视为磁盘文件即可。

### 掉电不丢数据是磁盘的重要特性

事实上，MySQL 完全是依靠磁盘不丢数据的特性来实现的：无论你执行了多少次 `update` 语句，无论 MySQL 有多少级缓存和多少种日志，只有当数据成功写入磁盘后，才会向客户端返回成功状态。文件的修改是有队列的，可以确保每次写操作的可靠性。

当然，这里的写入磁盘并不一定指的是真的存储到表对应的数据文件中，也可以是 `*do log`。如果服务器意外重启，MySQL 启动后会将刚才记录的这些 `*do log` 中的信息默默地写入到磁盘上真正的表数据文件中。

“数据库的单点究竟在哪里”的答案已经显而易见：数据库的单点就单在了磁盘上。

## 7.3 存储技术简史

---

既然数据库的单点就是磁盘，那么接下来我们就来了解一下存储技术的发展简史。

### 集中式存储

集中式存储是一种采用独立的控制器（即计算机）控制大量的硬盘，并通过控制器对外提供多种不同层级的接口（硬件层面的 SAS/FC，软件层面的 SCSI/iSCSI/InfiniBand 等），以满足多个客户端、多种不同存储需求的产品。

集中式存储的兴起让 IOE 中的 E（EMC 存储公司）大放异彩：就像前面提到的那台价值百万的负载均衡设备一样，EMC 的集中式存储通过双控制器开机热备+全冗余网络连接，再配合 SAN 交换机和双 HBA 卡，实现了全冗余的存储网络架构。它性能出色，可以支持多台服务器连接使用，并且非常稳定。唯一的缺点就是价格昂贵。光是一个普通的 16 口 16G SAN 交换机，价格就已经超过了 40G 以太网交换机（交换容量

甚至可以达到 50Tbps 以上)。更不用说一年也卖不出去几片的 HBA 卡了。而且，集中式存储设备本身更贵，非常贵，比两台标准 x86 服务器还要贵。

为什么集中式存储这么贵呢？因为它使用了非常高的硬件成本和服务成本，真正解决了大多数企业面临的存储问题：厂家负责上门部署，定期维护，你只需要出高价即可。集中式存储的本质是利用高水平的硬件+硬件级全冗余+保姆式的技术服务，彻底解决了存储这个难题。

## 分布式存储

进入云计算时代，分布式存储大放异彩：既然海量的 x86 服务器已经在机柜里运行着了，为什么不拿出一点点计算资源构建一个省钱的分布式存储呢？而且从需求的角度来看，云计算的规模正在快速增长，集中式存储很难满足如此快速的规模扩张。这时候就需要分布式存储登场了。

分布式存储选择通过海量普通可靠性的硬件+软件+数据冗余的方式，将一群 x86 服务器通过以太网或者 InfiniBand 相互连接，将分散在每一台服务器上的机械磁盘和固态硬盘组织到一起，形成一个庞大的硬盘资源池。这个软件定义的存储集群可以做到与集中式存储一样的三高：高可靠性、高可用率、高性能。

## 两者之间的差异

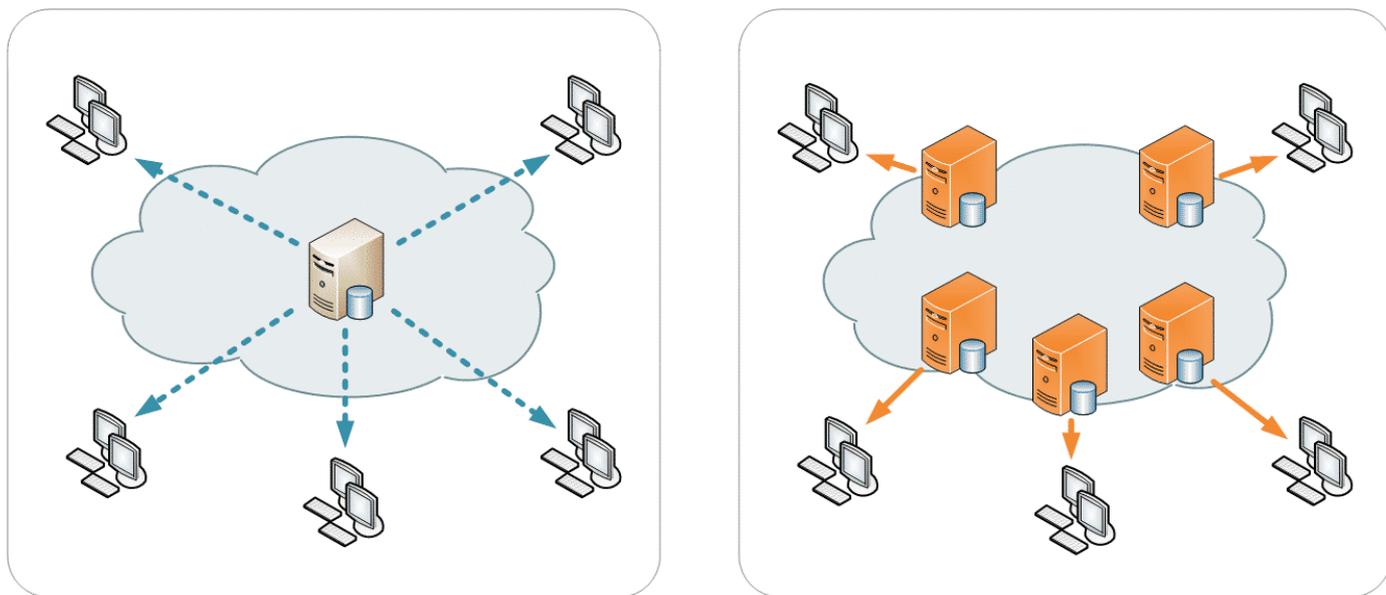


图 7-1 集中式存储和分布式存储的结构差异

如图 7-1 所示就是集中式存储和分布式存储的结构差异。集中式存储将数据集中存储在单一的节点上，而分布式存储则将数据分散存储在多个节点上。这导致了它们各有优劣。

## 集中式存储的优缺点

集中式存储的优点包括：

1. 部署简单、管理方便：所有的数据都存储在单一的节点上，只需要管理一个设备或服务器。
2. 数据一致性高：数据存储在单一的节点上，一致性非常高。
3. 存取速度快：直接通过磁盘硬件接口传输数据，不需要跨越网络传输数据。

集中式存储的缺点包括：

1. 单点故障：如果集中式存储的节点出现故障，整个系统将无法访问数据，导致服务中断。
2. 扩展性差：随着数据量的增加，集中式存储的性能可能会变差，难以实现横向扩展。
3. 高成本：为了提高性能和可靠性，集中式存储通常需要昂贵的硬件设备和专业的维护人员。

分布式存储的优缺点

分布式存储的优点包括：

1. 可扩展性好：由于数据存储在多个节点上，系统可以根据需要动态添加或删除存储节点，从而具有更好的可扩展性。
2. 容错性强：分布式存储通过复制和分散数据存储来提高系统的容错性。如果某个节点发生故障，系统可以从其他节点中恢复丢失的数据。
3. 读性能更好：分布式存储具有更好的性能，因为数据可以并行读取，减少了瓶颈和延迟。但是需要注意的是，由于需要数据校验，写性能更差。

分布式存储的缺点包括：

1. 数据一致性：在分布式存储系统中，数据分布在多个节点上，可能导致数据一致性问题，需要进行额外的一致性协议来保证数据的一致性。
2. 写入性能差：分布式存储的数据需要在写入之后进行跨节点校验，需要消耗额外的时间来达成一致，所以写入性能往往大幅落后于单块磁盘。
3. 复杂性：分布式存储系统的设计和管理相对复杂，需要考虑网络、数据分片、数据复制等多个方面的问题。
4. 延迟：由于数据需要在多个节点之间传输，分布式存储可能会导致较高的延迟，尤其是在跨地域的情况下。

我们正站在 x86 I/O 性能爆发的前夜

虽然我们说分布式存储也有高性能特性，但是，x86 架构下磁盘的性能其实并不出众。近年来，随着分布式存储的市场占有率越来越高，单系统规模也越来越大，从云计算厂商到服务器厂商，都在努力提升分布式存储各个部分的性能。除了“网络性能”和“缓存”在快速进步之外，x86 I/O 系统的“绝对性能”也在迅速提升，如今（2023 年），我们正站在 x86 I/O 性能爆发的前夜。

## 7.4 小型机的优势，x86 的劣势：I/O 性能

《性能之殇》中提到了 [x86 的 I/O 性能被架构锁死的问题](#)。相比之下，小型机的 I/O 性能上限非常高，甚至可以为存储子系统配置专门的 CPU。

如今，海量廉价的 x86 服务器集群在越来越快的网卡速率的支持下，在系统总容量和可用性方面已经超过了小型机，使得小型机只能在特定规模和行业的业务中保持优势。然而，I/O 性能不足一直是 x86 的弱点，直到最近两年才有所改善。

突飞猛进的 PCIe

近年来，已经使用了十年的 PCIe 3.0 突然开始迅速升级，以每年一代的速度推进，每次带宽都能翻倍，看起来进步很大。然而，当我们把目光从惊人的 128GB/s (PCIe 5.0 x32) 的数字上移开，仔细观察一块主板上同时存在的 5.0、4.0 和 3.0 插槽时，会发现一个更令人惊讶的事实：4.0 和 5.0 是在基础电气属性不变、金手指数量完全保持不变的情况下，通过重定时器 and 重驱动器组件实现通信频率两次翻倍，“强上陆地神仙”的结果。此外，5.0 插槽就是 5.0 插槽，不能拆分成两个 4.0 插槽使用。如果想插入 4.0 设备，可以，但只能插入一个：因为从 4.0 到 5.0 并不是水管变粗了，而是水管支持的流速提高了，这对接收端设备的容错能力提出了更高的要求。

为什么英特尔一屁股坐到 PCIe 牙膏上了呢？因为下一代超级 I/O 架构：CXL 需要基于 PCIe 5.0 来实现。

## 全村的希望 CXL

2022 年 11 月，首款支持 CXL 的服务器 CPU (AMD EPYC 9004 系列) 已经上市。尽管目前 CXL 技术的大规模应用尚未展开，但它已成为“全村的希望”。虽然英特尔直到 2019 年才推出了 CXL 标准，但它不仅在短短三年内推动了业界推出兼容 CXL 技术的 CPU (而且是友商开发的)，还吸纳了两个竞争对手——OpenCAPI 联盟和 Gen-Z 联盟。CXL 成为了唯一的“新一代通用 I/O 标准”。

凭什么呢？就凭 CXL 是英特尔向业界投放的一颗重磅炸弹：一次性开放了从 CPU 直接驱动的 DDR 内存到 NVMe SSD 之间广阔的“无人地带”——基于 PCIe 5.0 技术，将 I/O 技术推向了一个新时代。别忘了，PCIe 协议也是由英特尔定义的。

## 内存、磁盘正在双向奔赴

近年来，随着 NVMe SSD 在服务器端的普及，内存和磁盘的性能边界正在变得逐渐模糊。十年前，SATA SSD 刚刚普及时，内存带宽和延迟为 100GB/S、60ns，而 SATA SSD 为 500MB/S、200 $\mu$ s，速度和延迟分别为 [1/200](#) 和 [3000](#)倍；如今，内存和 PCIe 5.0 NVMe 磁盘的对比为 150GB/S、100ns 和 12.8GB/S、9 $\mu$ s，差距缩小至 [1/12](#) 和 [90](#)倍。

**CXL 补上了“存储器山”上 DDR 内存和 NVMe SSD 之间的巨大空隙**

## CXL: A Scalable Solution for Memory

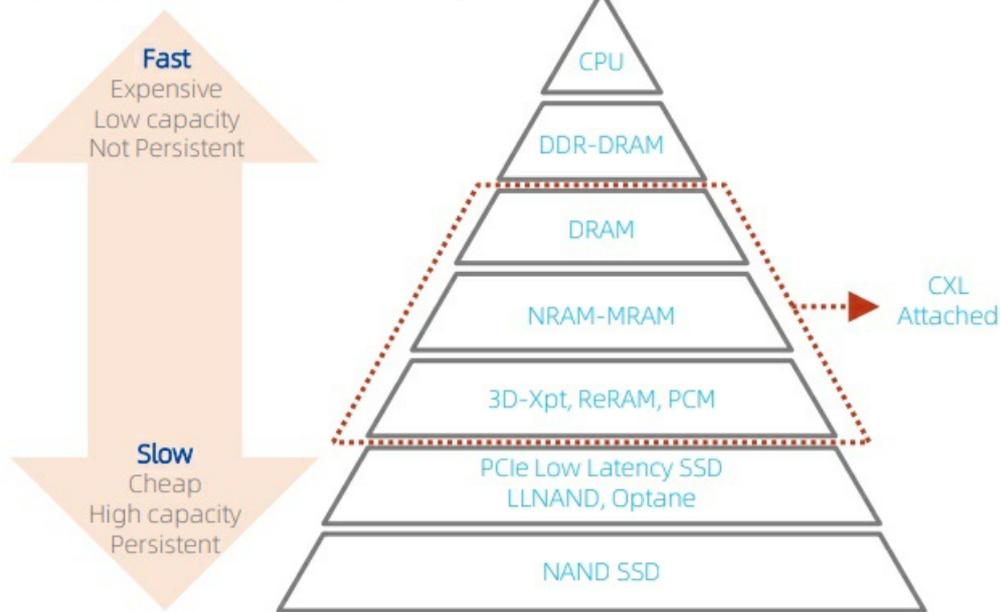


图 7-2 CXL 补上了“存储器山”上 DDR 内存和 NVMe SSD 之间的巨大空隙

《CS:APP》第六章提出了著名的“存储器山”理论<sup>[4]</sup>，如图 7-2 所示：离 CPU 越远的存储器容量更大延迟更高。去年发布的 AMD 9004 系列处理器引入了 12 通道的 DRAM 内存，对电路板设计和制造成本提出了非常高的要求，同时对 CPU 内部内存控制器的驱动能力也带来了挑战。基本可以确定内存条这个将一大块 DRAM 芯片在 Z 轴进行折叠以增加容量的技术已经走到尽头，未来将是 CXL 的时代。

CXL 提供了 I/O（设备发现、初始化、中断等基础功能）、缓存（低延迟数据复制）、内存（统一地址编码）三个模块，一次性解决了海量内存扩展和多级别缓存的需求。目前，支持 CXL 1.1 协议的扩展内存已经上市。

未来，双向奔赴的内存和磁盘将在 CXL 技术中胜利会师：容量和延迟分布将更加均匀，系统宏观性能将进一步提升。

**CXL 三代技术发展的终极目标**

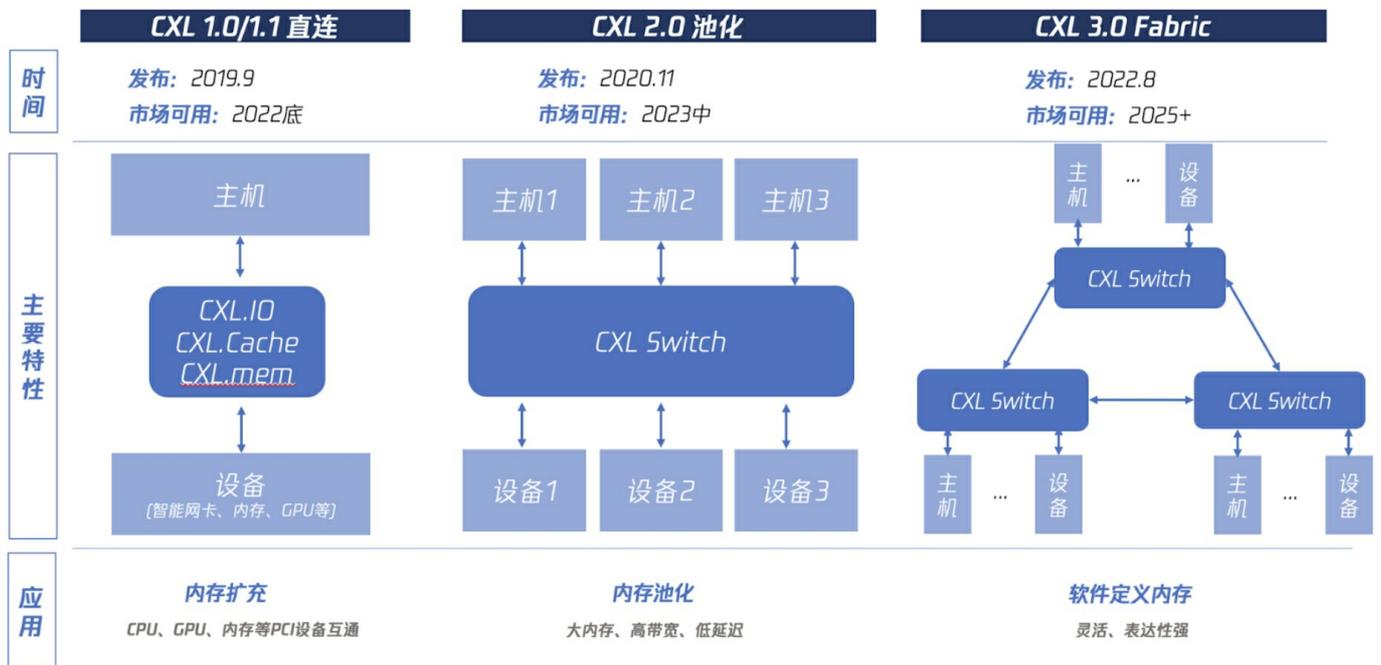


图 7-3 三代 CXL 技术路线图

如图 7-3 所示，CXL 1.0 时代主要实现基本功能，2.0 要实现类似于 SAN 交换机技术的多对多内存池化，3.0 要构建 CXL 互连网络：实现软件定义内存集群。简单来说，就像分布式存储一样，用软件将内存集群化。

内存被软件集群化后，将给数据库架构带来巨大的改变：基于独立的 RDMA 网卡技术的计算与存储分离架构已经涌现出了如 Snowflake、Amazon Aurora、阿里云 PolarDB 等优秀的商用数据库产品。如果内存都能被集群化，那么“计算与存储分离”中的“计算”也将被颠覆。届时，“主从同步”四个字将拥有全新的意义。

### 超高速网卡也需要 CXL 来解决

当前，x86 服务器领域的各大厂商都在不遗余力地努力将 400G 网卡塞进服务器机箱。如果不考虑任何优化措施，仅依靠 Linux 网络栈运行 TCP 时只能达到 5Gbit/s，这主要是因为软件架构的滞后。然而，400G 网卡无法使用则是纯粹的硬件限制：400G 的带宽已经超出了八路内存的理论带宽上限，只要网卡数据仍然需要经过 CPU，并且仍然使用 DDR4/DDR5 内存，那么 400G 网卡就是非常难以实现的。

CXL 技术为解决 400G 网卡的问题提供了一种解决方案：将网卡与 CPU 分离，使它们各自拥有独立的内存空间，并允许 CPU 无障碍地读写网卡芯片的内存空间。从本质上讲，这相当于重新定义了“网卡接收数据后将其存储到内存”这一过程中“内存”的含义。

这种思想其实我们已经有所了解：浮点运算能力远远超过 CPU 的 GPU，通过单独设计内存架构（称为显存），在其内部实现了卓越的性能，然后直接使用 DP 协议输出 8K 画面，而无需让大量数据经过 CPU 和内存，从而避免了性能瓶颈的出现。

### 大语言模型也需要 CXL 技术

最新的 65B 参数的大模型，如果使用 GPU 进行训练，将需要高达 192GB 的显存，而显存容量是目前 GP

U 技术和成本的主要限制因素。借助 CXL 技术，未来的 GPU 将能够充分利用主板上常见的 4TB DDR5 内存，甚至可以使用 PCIe 5.0 12.8GB/s 的 SSD 磁盘进行训练，人均 650 亿参数大模型指日可待！

## 7.5 x86 内存技术的演进

---

最初的 x86 架构只支持 32 位寻址，最大物理内存限制为 4GB。但随着扩展模式（Extended Mode）的引入，x86 架构可以支持 64 位寻址，最大物理内存容量可达到几十 TB。现在容量够了，该怎么继续提升内存的性能呢？

### 内存子系统优化

x86 发展出了两个主要的内存子系统优化方案，显著提升了 x86 体系计算机的内存速度。

1. 多级缓存：现代 x86 架构通常配备多级缓存（L1、L2、L3 等）。这些高速缓存位于处理器核心旁边，可以更快地存取数据，减少对内存的读写操作。
2. NUMA 架构：非一致性存储访问（NUMA）架构是在多处理器系统中常见的一种设计。它通过将内存分为多个区域，并为每个处理器分配特定的内存区域，减少了不必要的远程内存访问，提高了内存访问的性能。

### 最近十年内存其实一直在变慢

众所周知，近十年来服务器 CPU 核心数量激增，内存通道也从最初的双通道逐渐发展至如今的 12 通道。然而，相较于内存带宽，CPU 核心数量的增长更为迅猛：这些年来，每个 CPU 核心所能获得的内存带宽持续下降，导致每台虚拟机的内存读写速度反而变慢。

### 英特尔对 x86 内存体系的限制

几年前，微软统计了其服务器各项部件的总成本，惊讶地发现[超过 50% 的服务器费用都用于购买内存条](#)。内存不仅硬件成本高昂，现有的内存架构还导致内存在虚拟机内部存在巨大的浪费：几乎每台物理服务器都浪费了大约 50% 的内存。而这实际上是英特尔有意为之的。

x86 的内存子系统一直封闭不开放，直到近年来机器学习的兴起使人类社会对服务器内存容量的需求再次攀升。在英伟达股价不断飙升之后，英特尔才被迫开放了 CXL 标准，基于 PCIe 5.0 的高带宽，使得内存能够像硬盘一样自由扩展。与此同时，由于 CXL 的高带宽通信能力，机器内不同内存模块之间（如内存、显存、网卡缓存）的数据传输速度也得到了显著提升。

## 7.6 面试题

---

**No.21**：机械磁盘和固态硬盘分别适合什么数据库算法？

机械硬盘（HDD）和固态硬盘（SSD）在数据库算法的选择上有着不同的特点。

### 1. 机械硬盘（HDD）：

- 特点：机械硬盘使用机械臂来读取和写入数据，它们较慢且寻址时间较长。机械硬盘的性能受限于物

理旋转速度和机械臂的定位速度。

- 适合的数据库算法：机械硬盘适合顺序访问和批量处理的算法。例如，在大规模数据加载、备份和批量分析等场景下，机械硬盘可以更好地处理大量连续的数据读写请求。

## 2. 固态硬盘 (SSD) :

- 特点：固态硬盘使用闪存芯片来存储数据，具有更快的读写速度和更低的访问延迟。与机械硬盘相比，固态硬盘的并发性能更好。
- 适合的数据库算法：固态硬盘适合频繁随机读写和高并发访问的算法。例如，对于在线事务处理 (OLTP) 系统和需要低延迟响应的应用程序，固态硬盘可以提供更好的性能。

目前，固态硬盘正逐渐取代机械硬盘成为存储设备的主流选择。在现代数据库系统中，使用固态硬盘通常能够获得更好的性能和响应时间。随着时间的推移，机械硬盘的用武之地越来越少，正在逐渐变成备份数据专用的技术，和磁带抢夺冷数据备份市场去了。

### No.22 : x86 的磁盘性能经过了什么样的发展历程？

世纪之交时，x86 服务器进入了主流市场，从那时起，它的磁盘技术经历了四次主要的技术迭代。

1. IDE (Integrated Drive Electronics) : IDE 是一种串行接口，需要精确地规划电路的长度，速度较慢，只有 133 MB/S。在高速磁盘的需求出现后，IDE 被 SATA 串行技术所取代。
2. SATA (Serial ATA) : SATA 是一种串行通信协议，相比于 IDE，它具有更高的传输速度和更小的线缆体积。SATA 的第一代产品就已经可以提供 1.5 Gbps 的传输速度，而到了第三代 SATA III，传输速度已经提升到了 6 Gbps。这种接口已经成为现代计算机的主流硬盘接口。
3. NVMe (Non-Volatile Memory Express) : NVMe 是一种高速非易失性存储器接口技术，专为固态硬盘 (SSD) 设计。它通过 PCIe 总线连接，利用并行传输和队列化技术提高了数据传输速率和 I/O 操作效率。NVMe 接口的传输速率可以超过 10 GB/S。
4. CXL (Compute Express Link) : CXL 是一种新兴的高速互连技术，它不仅支持存储设备，还可以连接加速器、图形处理单元 (GPU) 等。CXL 是基于 PCIe 5.0 的新一代高速互连标准，提供了更高的带宽和更低的延迟。它使得服务器可以通过直接内存访问 (DMA) 方式与存储设备进行高速通信，进一步提高存储性能和数据处理能力。

CXL 是一个全新的存储技术发展方向，它统一支持了从内存到 NVMe 磁盘之间巨大的速度鸿沟，支持内存、缓存、磁盘甚至是 GPU 等外围专用加速器的高速连接，对大规模数据中心和高性能计算应用的发展也有很大的帮助。

## 第 8 章 MySQL InnoDB 存储引擎详解

在前几章中，我们通常以架构为武器，对性能问题进行降维打击。然而，本章不同，我们要啃硬骨头：深入理解 MySQL 的 InnoDB 存储引擎，以便读者们能够顺利阅读下一章。

还记得我们的目标吗？一百万 QPS

在经过性能优化的电商业务中，假设每次 API 调用平均执行五条 SQL，那么数据库的 QPS 就是 500 万。稍微接触过一些高并发系统的人都能一眼看出，这是一个多么惊人的数字。对于开源 MySQL 来说，单机一万的 QPS 已经非常优秀，而 500 万简直就像天方夜谭。别着急，慢慢往后看，这真的有可能实现。

## 8.1 概述

InnoDB 是磁盘存储引擎技术的巅峰之作，代表了磁盘数据库最先进的技术水平，给人类创造了巨大的价值。它提供了许多高级功能，如事务支持、行级锁定、崩溃恢复等。

### InnoDB 的发展历程和设计目标

2004 年，扎克伯格选择了 MySQL 来创建 Facebook，不承想，Facebook 迅速火遍了全世界。随后的十年里，Google、亚马逊、阿里巴巴等互联网巨头纷纷选择 MySQL 作为自己的核心数据库，这带来了 MySQL 的高速发展：这些互联网巨头投入大量技术资源将 MySQL 打造成了一款非常优秀的开源数据库软件，而 InnoDB 则是其中最受重视的部分。

这些公司的数据库需求基本相似：表字段较少，但行数众多；对单个查询的时间要求极高；很少存储长字符串和二进制文件。这种需求恰好不是传统数据库（如 Oracle）的强项。而且，互联网公司的发展一日千里，无论是授权费还是实施速度，买 Oracle 都不如自己魔改 MySQL 来的方便。之后的十几年，MySQL 特别是 InnoDB 的发展真的可以说是虎虎生风、一日千里，如果你站在 2014 往回看 2004，也确实称得上恍如隔世。

但 InnoDB 也不是银弹，它的性能也是靠“取舍”得来的。

### InnoDB 拿什么换信息之神换了什么？

和之前的 MyISAM 相比，InnoDB 最大的变化就是将磁盘上数据的基本存储结构从索引+数据这样的分体式，变成了所有数据都挂在索引上的整体式：从“B+ 树索引”加“磁盘连续存储数据”（中间用指针链接）变成了“B+ 树存储全部索引和数据”。

这个操作给 MySQL 带来了翻天覆地的变化。那么，InnoDB 到底做了哪些取舍呢？

#### 代价

1. 插入和更新性能显著下降（100~1000 倍，但绝对耗时仍在毫秒级别）
2. 一次性读取连续多行的复杂度大幅提升（大约提升了  $\text{行数}/10$  那么多倍）
3. 事务隔离导致大表 count(\*) 的返回时间令人崩溃（多少倍已经无法衡量，一千倍到一千亿倍吧）
4. 放弃了数据完全压缩能力（磁盘占用 2-5 倍）
5. 写入放大显著增加，对磁盘特别是固态硬盘 (SSD) 形成了额外的压力

#### 收益

1. 支持事务：事务是极其核心的功能进步，使 MySQL 摆脱了玩具定位，真正实现了 ACID，成为了一个合

格的 OLTP 数据库

2. 大表的单行读取性能暴增：这是数据挂在 B+ 树索引上带来的优势，还可以依靠内存缓存进行加速和局部性优化
3. DML (表结构变更) 操作的安全性大幅提升，数据损坏的概率大幅降低
4. 联表查询性能大幅提升，让 MySQL 在 OLAP 方向也有很大进步
5. 支持全文检索：类似于 ES 的倒排索引
6. 其他小收益：支持了行锁、内存缓存、外键约束等

这些代价和收益，基本都是两个东西带来的：B+ 树和 Buffer Pool。下面我们分别认识一下这两个技术。

## 8.2 B+ 树

B+ 树是 1970 年由 Rudolf Bayer 教授在《Organization and Maintenance of Large Ordered Indices》一文中提出的，此后迅速成为海量数据存储与检索的重要工具。以查询为主要场景的关系型数据库，无一例外地选择了 B+ 树。

### B+ 树的基本思想

B+ 树是一种平衡多路查找树，其灵感源自平衡二叉树和 B 树，但专为速度较慢的“外存”设计，因此具有独特的设计方向：

尽量减少数据不断增长时的磁盘 I/O 数量，包括插入新数据场景和查询场景。

它将经典平衡二叉树的“再平衡”过程颠倒过来了——最底层的叶子节点紧密排列，新增数据时不断添加新的叶子节点，需要再平衡的不是叶子节点，而是上面的索引页。而索引页具有以下特点：

1. 索引页数量少
2. 再平衡时，B+ 树算法调整的索引节点数量也很少
3. 索引容量足够大：3 层索引可承载 2000 万行数据，4 层索引可承载 200 多亿行
4. 索引页少，可以将所有索引全部载入内存，读取索引的磁盘 I/O 趋近于 0

### InnoDB 是如何组织数据的

下面我们来具体认识一下 InnoDB 中的索引和数据是怎么利用 B+ 树思想在磁盘上进行组织的。

#### 1. 页

页是 MySQL 中数据存储的基本单元，整颗 B+ 树就是一个又一个相互使用指针连接在一起的页组成的。由于 InnoDB 出现的时候，SSD 还没有出现，所以它是为了机械磁盘及其 512 字节的扇区而设计的，所以页块的默认大小被设置为了 16KB（32 个连续扇区）。

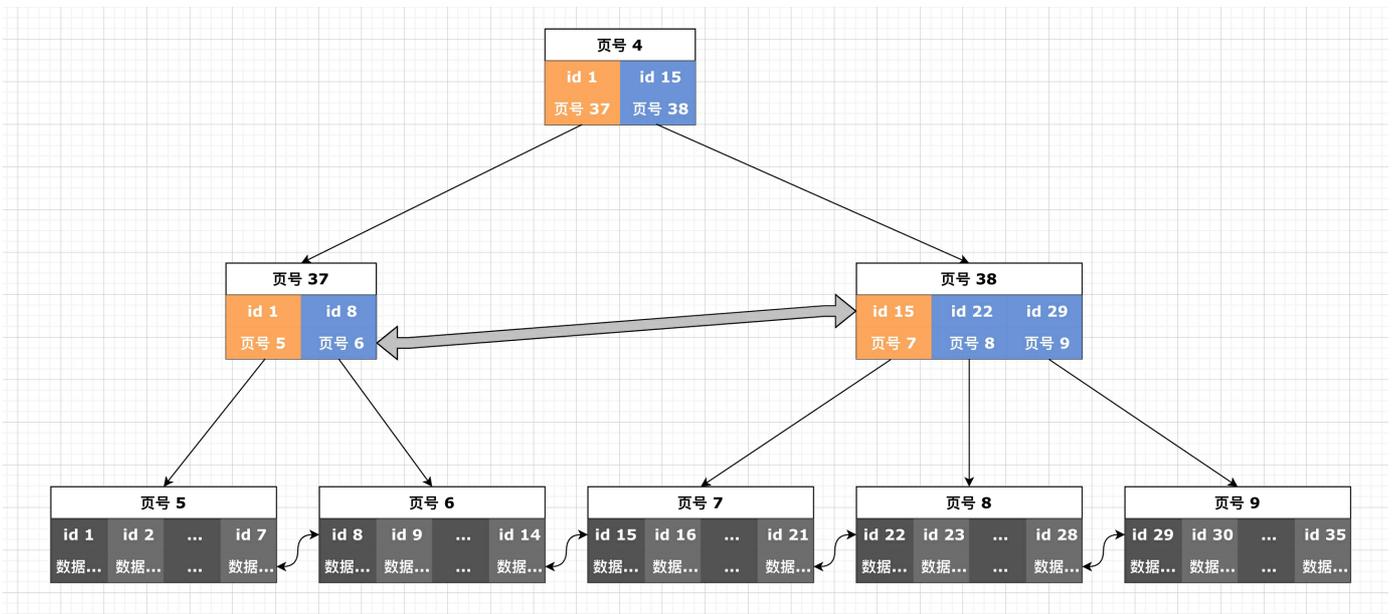


图 8-1 三层索引下的页结构图 (35 行数据)

图 8-1 展示出了页之间的指针关系：

1. 上层页对下层页拥有 **单向** 指针
2. 同一层内相邻的页之间拥有 **双向** 指针，无论是上面的索引页层还是底层的数据页层
3. 最底层数据页层中，每一页可以存储多行数据，每一行数据拥有指向下一行的 **单向** 指针

而在物理层面，每一页的内部结构都如图 8-2 所示。



图 8-2 InnoDB 索引页的内部结构

## 2. 索引页里面有什么

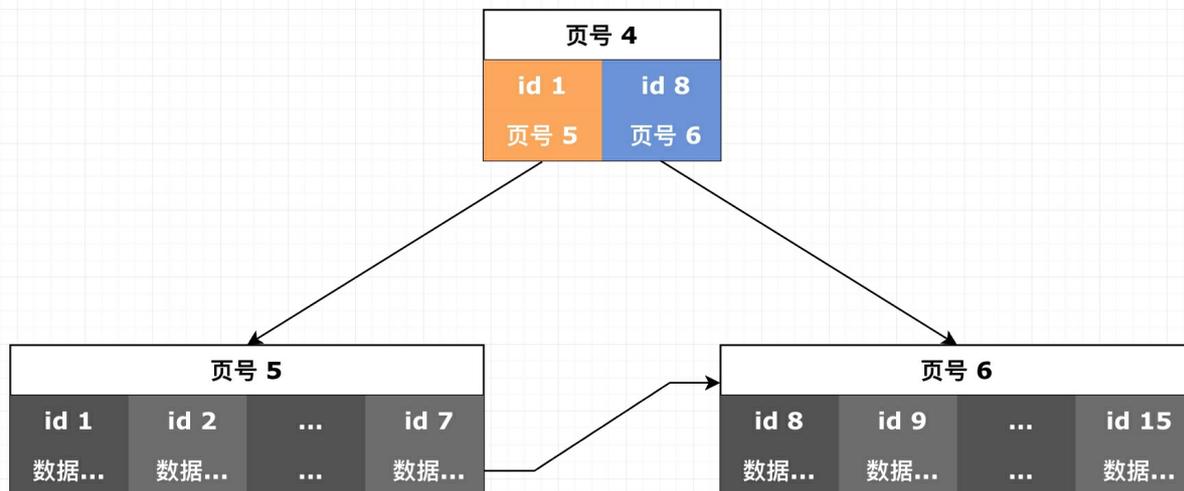


图 8-3 二层索引下的页结构图 (15 行数据)

我们用更清晰的二层索引簇结构来展示索引页包含的关键信息，如图 8-3 所示。顶部那个彩色的页就是索引页。

除了头部和尾部的基础信息字段之外，索引页的“用户存储数据”紧密排列着指向下一层页的指针，结构为：`id|页号`。

这里的 id 就是这张表默认主键的那个 id，一般为 `int` (4 字节) 或者 `bigint` (8 字节)。该数字的含义是：

该页号对应的页，以及下挂的所有页，所拥有的全部数据行中，id 最小的那行数据的 id。

InnoDB 使用这个数字可以快速定位某一行数据所处的页的位置。

页号就是页的编号，在不同版本的 MySQL 上这个页编号的长度是不一样的，下面我们会通过测试来确定 MySQL 8.0 中页编号的长度。

### 3. 数据页里面有什么

B+ 树上层的所有页只存储索引，用最底层的页存储数据。这是 B+ 比 B 树优秀的地方：以一丢丢写入速度为代价，让较少的索引层数内存下了更多的索引指针，可以支撑海量的数据行数。

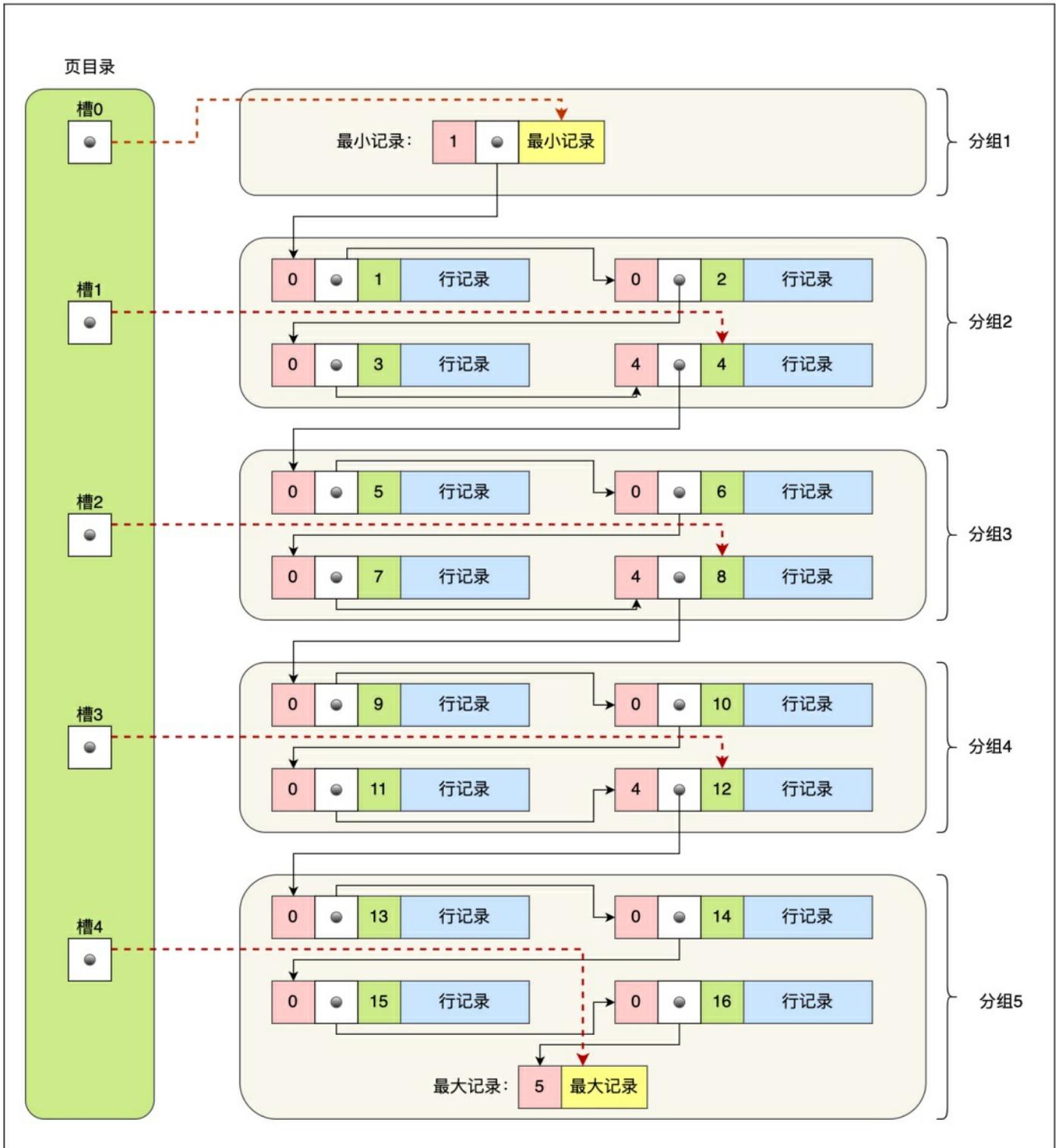


图 8-4 数据页内部结构

如图 8-4 所示，数据页内部是分槽的，相当于自己又加了一层索引。数据页内部拥有最小记录指针和最大记录指针，有没有觉得一花一世界，数据页内部结构和 B+ 树颇有几分神似呢？

### SQL 查询过程

有了前面的铺垫，真正的 SQL 查询过程就呼之欲出了。假设我们要找出 `id=6` 的一行数据，则我们执行的 SQL 为 `select * from users where id=6`。我们还怎么找出这一行数据呢？只需要逐层比对即可：

1. 将顶部页（16KB）的数据读入内存，将 6 与每个 `id|页号` 进行大小对比，找到 6 所在的页：大于等于当

- 前 id 且小于右侧邻居的 id
2. 将上一步找到的那一页数据读入内存，重复上述比对操作，直到找到最底层的数据页
  3. 将数据页读入内存，找出 `id=6` 索引下挂载的全部数据，这就是所需的这行数据

## 8.3 InnoDB 数据插入测试

接下来，我们将对 InnoDB 进行一次数据插入测试，以揭示其索引页“再平衡”的具体操作，追踪索引扩层的具体动作，确定 `id|页号` 中页号的数据长度，并解决“2000w 行是否该分表”这个历史悬案。

### 神奇的“2000W 行分表”历史悬案

相信大家都听说过“单表到了 2000 万行就需要分表了”，甚至有人还看过“京东云开发者”的[那篇著名的文章](#)，但是那篇文章为了硬凑 2000 万搞出了很多不合理的猜想。

下面我们实际测试一下 MySQL 8.0.28 运行在 CentOS Stream release 9 上（文件系统为 ext4），索引层数和数据行数之间的关系，相信测试完以后，你会对这个问题有深刻的理解。

### 测试准备

测试表结构如代码清单 8-1 所示。

代码清单 8-1 InnoDB 插入测试的数据表结构

```
CREATE TABLE `index_tree` (  
  `id` int unsigned NOT NULL AUTO_INCREMENT,  
  `s1` char(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL DEFAULT 's1',  
  `s2` char(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL DEFAULT 's2',  
  `s3` char(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL DEFAULT 's3',  
  `s4` char(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL DEFAULT 's4',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

我们采用不可变长度的 char 来进行测试，根据 MySQL 8.0 [关于 CHAR 和 VARCHAR 类型的官方文档](#)，当我们只保存 `s1` 这种 ASCII 字符时，一行数据的长度很容易就可以计算出来： $4 + 255 + 255 + 255 + 255 = 1024$  字节

### ibd 结构探测工具

我们采用阿里巴巴开源的 [MySQL InnoDB Java Reader](#) 来窥探 ibd 内部所有页的情况，主要是看它们的层级。

### 开始插入数据

id	s1	s2	s3	s4
1	s1	s2	s3	s4

图 8-5 插入一行数据后的表数据

在只插入了一行数据时，表数据如图 8-5 所示，此时的 MySQL InnoDB Java Reader 结果如代码清单 8-2 所示。

代码清单 8-2 一行数据时的 MySQL InnoDB Java Reader 结果

```
====page number, page type, other info=====
0, FILE_SPACE_HEADER, space=1289, numPagesUsed=5, size=7, xdes.size=1
1, IBUF_BITMAP
2, INODE, inode.size=4
3, SDI
4, INDEX, root.page=true, index.id=4605, level=0, numOfRecs=1, num.dir.slot=2, garbage.space=0
5, ALLOCATED
6, ALLOCATED
```

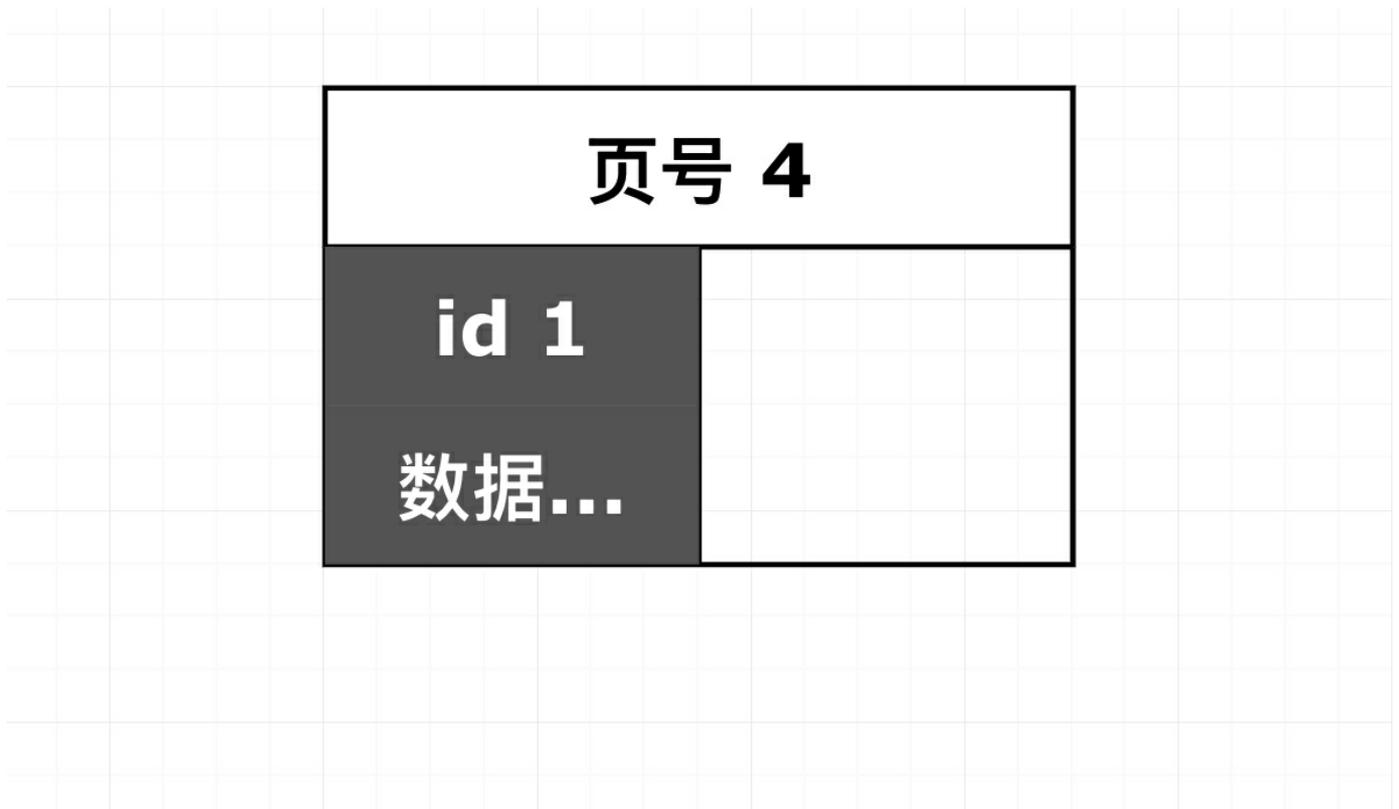


图 8-6 一层索引下的页结构图（1 行数据）

可以看出，只有一行数据时，ibd 文件内部只有一个页，此时的页结构如图 8-6 所示，`index_tree.ibd` 文件的大小为 112KB。

### 1. 首次索引分级

我们继续插入数据，在插入了第 15 行后，这个 idb 文件从 1 页分裂成了 3 页，此时的 MySQL InnoDB J

ava Reader 结果如代码清单 8-3 所示。

代码清单 8-3 15 行数据时的 MySQL InnoDB Java Reader 结果

```
====page number, page type, other info====  
0, FILE_SPACE_HEADER, space=1289, numPagesUsed=7, size=8, xdes.size=1  
1, IBUF_BITMAP  
2, INODE, inode.size=4  
3, SDI  
4, INDEX, root.page=true, index.id=4605, level=1, numOfRecs=2, num.dir.slot=2, garbage.space=0  
5, INDEX, index.id=4605, level=0, numOfRecs=7, num.dir.slot=3, garbage.space=7350  
6, INDEX, index.id=4605, level=0, numOfRecs=8, num.dir.slot=3, garbage.space=0  
7, ALLOCATED
```

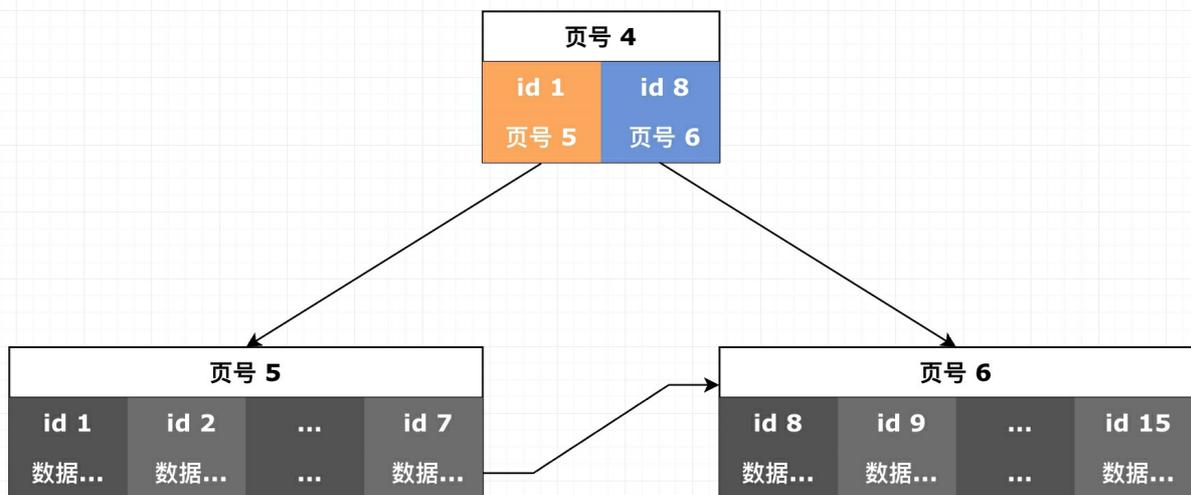


图 8-7 二层索引下的页结构图 (15 行数据)

我们能够看出，本来这 14 条数据都是在初始的那个 4 号页内部储存的，即数据部分至少有  $1024 \times 14 = 14\text{KB}$  的容量，在插入第 15 条数据迈向 15KB 的时候，innodb 发了页的分级：B+ 树分出了两级，顶部为一个索引页 4，底部为两个数据页 5 和 6，5 号页拥有 7 行数据，6 号页拥有 8 行数据。此时的页结构如图 8-7 所示。

这个转变说明每一页可用的数据容量为  $14\text{KB} - 15\text{KB}$  之间。而且，从 `garbage.space` 可以看出，5 号页是之前那个唯一的 4 号页，而新的 4 号页和 6 号页则是本次分级的时候新建的。

下面让我们继续插入数据，看它什么时候能从二层增长为三层。

## 2. 二层转换为三层

以 500 为步长批量插入数据，在 16500 行的时候，还是二层，此时的 MySQL InnoDB Java Reader 结果如代码清单 8-4 所示。

代码清单 8-4 16500 行数据时的 MySQL InnoDB Java Reader 结果

====page number, page type, other info====

```

0, FILE_SPACE_HEADER, space=1292, numPagesUsed=37, size=1664, xdes.size=22
1, IBUF_BITMAP
2, INODE, inode.size=4
3, SDI
4, INDEX, root.page=true, index.id=4608, level=1, numOfRecs=1180, num.dir.slot=296, garbage.space=0
5, INDEX, index.id=4608, level=0, numOfRecs=7, num.dir.slot=3, garbage.space=7350
6, INDEX, index.id=4608, level=0, numOfRecs=14, num.dir.slot=4, garbage.space=0
7, INDEX, index.id=4608, level=0, numOfRecs=14, num.dir.slot=4, garbage.space=0
8, INDEX, index.id=4608, level=0, numOfRecs=14, num.dir.slot=4, garbage.space=0
9, INDEX, index.id=4608, level=0, numOfRecs=14, num.dir.slot=4, garbage.space=0

```

但是当表长度来到 17000 的时候，已经是三层了，此时的 MySQL InnoDB Java Reader 结果如代码清单 8-5 所示。

代码清单 8-5 17000 行数据时的 MySQL InnoDB Java Reader 结果

====page number, page type, other info====

```

0, FILE_SPACE_HEADER, space=1292, numPagesUsed=39, size=1728, xdes.size=22
1, IBUF_BITMAP
2, INODE, inode.size=4
3, SDI
4, INDEX, root.page=true, index.id=4608, level=2, numOfRecs=2, num.dir.slot=2, garbage.space=0
5, INDEX, index.id=4608, level=0, numOfRecs=7, num.dir.slot=3, garbage.space=7350
6, INDEX, index.id=4608, level=0, numOfRecs=14, num.dir.slot=4, garbage.space=0
7, INDEX, index.id=4608, level=0, numOfRecs=14, num.dir.slot=4, garbage.space=0

```

... ..

```

36, INDEX, index.id=4608, level=0, numOfRecs=14, num.dir.slot=4, garbage.space=0
37, INDEX, index.id=4608, level=1, numOfRecs=601, num.dir.slot=152, garbage.space=7826
38, INDEX, index.id=4608, level=1, numOfRecs=614, num.dir.slot=154, garbage.space=0
39, ALLOCATED
40, ALLOCATED
... ..

```

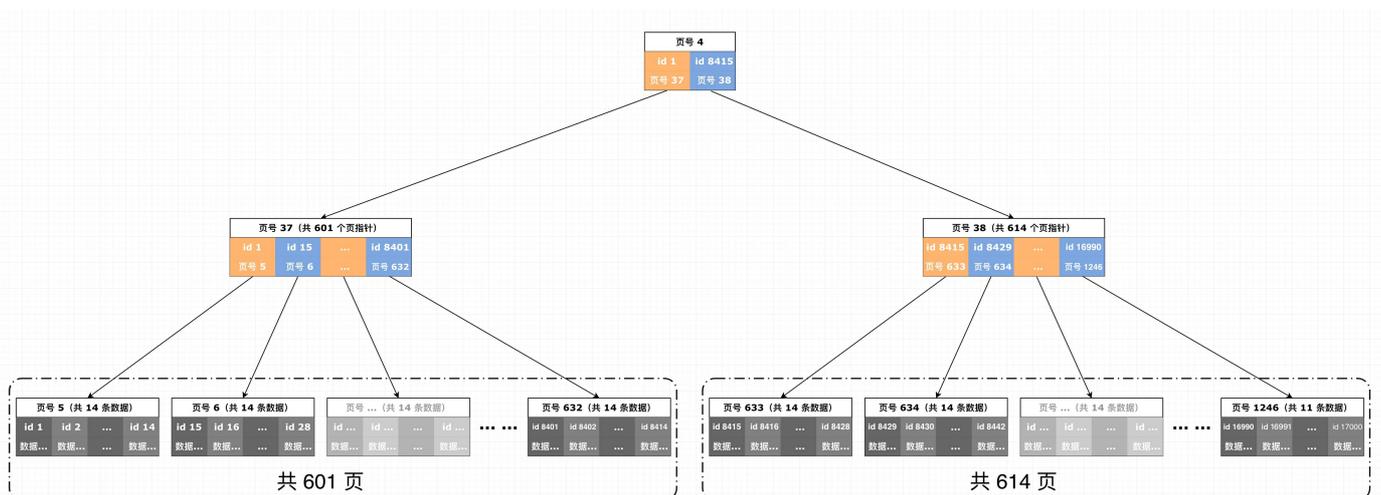


图 8-8 三层索引下的页结构图 (17000 行数据)

在 17000 行时, InnoDB 的索引页变成了 3 层, 此时的页结构如图 8-7 所示, `index_tree.ibd` 文件的尺寸为 27MB。

在整颗 B+ 树从二层转换为三层的过程中, 只修改了三个页:

1. 将目前唯一的索引页 4 号的数据复制到 37 号页中, level 保持不变 (此时 37-63 号已经被提前 `ALLOCATED` 出来用作备用页了)
2. 将 38 号页初始化成一个新的 level=1 的索引页, 并将左侧 37 号页右边一半的页指针转移给 38 号页, 再删除 37 号页中的原指针
3. 重新初始化 4 号页, 设置为顶层(level=2)索引页, 创建两个页指针: 第一个指向 37, 第二个指向 38

为什么是 17000 行呢? 我们来计算一下二层索引的极限容量:

1. 已知一个最底层(level=0)的数据节点可以存储 14 条数据
2. 假设索引页内部的一个页指针的长度是 `4+8=12` 字节, 那 2 层索引的极限就是:

$$(14 * 1024 / 12) * 14 = 16725.33$$

和实测值完美契合!

此时计算可知, 一个索引页至少可以存储 `14 * 1024 / 12=1194.66666666` 个页指针。

### 3. 三层转换为四层

继续向 `index_tree` 表中批量插入数据, 在数据继续分层之前, 整棵树的结构保持不变, 只是会不断增加 level=0 和 level=1 的页的数量。

但当行数来到了 21427000 行时, 索引就从 3 层转换为了 4 层了, 此时磁盘 `ibd` 文件为 24GB, 此时的 MySQL InnoDB Java Reader 结果如代码清单 8-6 所示。

代码清单 8-6 21427000 行数据时的 MySQL InnoDB Java Reader 结果

```
====page number, page type, other info====
```

```
0, FILE_SPACE_HEADER, space=1292, numPagesUsed=4, size=1548032, xdes.size=256
```

```
1, IBUF_BITMAP
```

```
2, INODE, inode.size=4
```

```
3, SDI
```

```
4, INDEX, root.page=true, index.id=4608, level=3, numOfRecs=2, num.dir.slot=2, garbage.space=0
```

```
5, INDEX, index.id=4608, level=0, numOfRecs=7, num.dir.slot=3, garbage.space=7350
```

```
6, INDEX, index.id=4608, level=0, numOfRecs=14, num.dir.slot=4, garbage.space=0
```

```
... ..
```

```
1424021, INDEX, index.id=4608, level=2, numOfRecs=601, num.dir.slot=152, garbage.space=7826
```

```
1424022, INDEX, index.id=4608, level=2, numOfRecs=672, num.dir.slot=169, garbage.space=0
```

```
... ..
```

由此可知，索引结构是这样的：

1. 1 个 4 层(level=3)索引页，含有 2 个 3 层索引页的指针
2. 2 个 3 层(level=2)索引页，其中左侧的 1424021 号页有 601 个底层数据页的指针，右侧的 1424022 号页有 672 个底层数据页的指针
3.  $601+672=1273$  个 2 层索引页，每页含有 1194+ 个底层数据页的指针
4.  $21427000/14=1530500$  个底层数据页，每页含有 14 条数据

转换的过程中，哪些页需要更新数据呢？还是只需要修改三个页：

1. 需要将 4 号页(旧顶层页)的数据拷贝到 1424021 号页(新 level=2 左)中
2. 新生成 1424022 号页(新 level=2 右)，将 1424021 号页(新 level=2 左)内部右侧的 672 个页指针 (`id|页号`)复制到 1424022 号页中，并删除 1424021 号页中的原指针
3. 重新初始化 4 号页，创建两个页指针：第一个指向 1424021，第二个指向 1424022

再增加一层需要再插入 1200 倍的数据，笔者就不测试了，有条件的读者可以自己尝试。

### 计算页指针 `id|页号` 的大小

无论是中文技术文章还是英文技术文章，笔者甚至还查了 MySQL 8.0 InnoDB 的官方文档，并没有说“页号”的大小，甚至对于 `id` 的大小都没有一个统一的说法。下面我们尝试自己算出来：

- 在 14-15 之间一层索引转换成了二层索引，所以页可用容量最大值  $1024 * 15 = 15360$
- 最小值  $1024 * 14 = 14336$
- 在 16500-17000 之间二层索引转换成了三层索引，对应的索引数最大值为  $17000 / 14 = 1214.28$
- 索引数最小值为  $16500 / 14 = 1178.57$

我们拿最大值除以最小值，得到  $15360 / 1178.57 = 13.03$  字节，拿最小值除以最大值，得到  $14336 / 1214.28 = 11.81$  字节，所以我们可以得出结论：



单个 `id|页号` 的大小应该为 12 字节或者 13 字节

接下来怎么确定呢？再拿 `bigint` 做一遍测试就行了。

## 利用 `bigint` 确定页号的大小

我们创建一个名为 `index_tree_bigint` 的表，其结构如代码清单 8-7 所示。

代码清单 8-7 `id` 类型为 `bigint` 时的表结构

```
CREATE TABLE `index_tree_bigint` (  
  `id` bigint unsigned NOT NULL AUTO_INCREMENT,  
  `s1` char(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL DEFAULT 's1',  
  `s2` char(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL DEFAULT 's2',  
  `s3` char(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL DEFAULT 's3',  
  `s4` char(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL DEFAULT 's4',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

单行数据的容量从 1024 字节增加到了 1028 字节。

按照同样的流程进行测试，可以发现还是 14 到 15 条的时候发生的一层转换到两层，从 12500 到 13000 条时从二层转换到了三层，我们使用同样的方法进行计算：

1.  $1028 * 15 / ( 12500 / 14 ) = 17.27$
2.  $1028 * 14 / ( 13000 / 14 ) = 15.50$

主键采用 `bigint` 类型时，单个 `id|页号` 的大小应该为 16 字节或者 17 字节。

得出结论：“页号”为 8 字节

由于页号采用奇数长度的概率非常低，我们可以得出一个十分可信的结论：在 MySQL 8 中，`id` 的长度和类型有关：`int` 为 4 字节，`bigint` 为 8 字节，“页号”的长度为 8 字节。所以，单个 `id|页号` 的大小应该为 12 字节或者 16 字节。

## 8.4 “2000W 行分表”问题

经过上一小节的数据插入测试，我们可以来认真分析一下“2000W 行分表”的问题了。

### 真实世界数据长度及其四层极限

住范儿生产数据库中的电商业务常用大表“订单商品表”，实际单行数据长度为 `0.9KB`，与测试结果的 1KB 相差不大，因此测试结果还是比较符合现实世界的真实情况的。

我们取以下值来计算三层和四层的理论极限：

1. 单页可用数据容量为 14KB。

2. 单行数据为 0.9KB，因此每页最多可以存储  $14/0.9=15.5555$ ，取整为 15 行数据。
3. 主键类型采用 `int`，页指针长度为 12 字节，因此每页最多可以存储  $14*1024/12=1194.6666$ ，取整为 1194 个页指针。

则三层 B+ 树的理论极限为： $1194^2 * 15 = 21384540$ ，大约 2100 万行，与传说中的 2000W 行分表相符。

那么四层到五层呢？

如果需要五层，则行数需要达到  $1194^3 * 15 = 25533140760$ ，即 255 亿行。这个数字已经超过了 `unsigned int` 的上限 42 亿多，需要使用 `bigint` 作为主键。感兴趣的读者可以自行计算 `bigint` 下五层索引甚至六层、七层、八层的行数极限，笔者在此不再赘述。

**2100w** 行以后真的会发生性能劣化吗？

并不会：三层索引和四层索引的性能差异微乎其微，2000W 行分表已经过时了！

实际上，每次 B+ 树增高只会增加两个索引页，修改一个索引页，总共只修改了三个 16KB 的数据页。无论是磁盘 I/O 还是 Buffer Pool 缓存失效，对性能的影响都非常微小：

索引从三层转换到四层，只增加了一次 I/O，绝对性能降低幅度的理论极限仅为  $1/3$ 。而且在有 Buffer Pool 存在的情况下，性能差异几乎可以忽略不计，只增加了  $1\sim 2$  次比大小的计算成本。

那是否意味着不需要再分表了呢？

虽然三层索引和四层索引看起来性能差异不大，但是如果你的单行数据比较大，例如达到了 5KB，仍然建议进行横向分表，这是减少磁盘 I/O 次数的最直接有效的优化方法：

1. 当单行数据为 0.9KB 时，三层树的极限行数是 2100 万。但是当单行数据达到 5KB 时，这个极限将变为仅 285 万行，这可能就不太够了。
2. 数据页具有局部性：每次从磁盘读取都是一整页的数据，因此读取某一行数据后，它 id 附近的数据行也已经在内存缓存中了，读取这一行附近的其他行不会产生磁盘 I/O。减少一行数据的大小，可以提升局部性优势。
3. 在连续读取多行时（例如全表条件查询），巨大的单行数据将迅速丧失“局部性”优势，同时引发磁盘 I/O 次数数量级规模的上升，这就是单行数据较大的表读取速度较慢的原因。

何时进行分表？

2017 年的阿里巴巴 Java 开发手册上说，当单表行数超过 500 万行或单表容量超过 2GB 时，推荐进行分库分表。然而，很多技术博文错误地将其解读为：阿里巴巴建议超过 500 万行的表进行分表。

尽管经过实测，每行数据定长 1024 字节，Buffer Pool 配置为 22GB，在单表体积 24GB 的情况下，四层索引和三层索引之间没有性能差异。但现实世界中的数据表并非如此完美：

1. 为了节省空间和保持扩展性，大多数短字符串类型采用 `varchar` 而非定长的 `char`，导致最底层的每一页包含的数据行数不一致，使平衡多路查找树不平衡。

2. 生产表经常面临数据删除和更新：同层页之间的双向链表和不同层页之间的单向指针需要频繁变化，同样会导致树不平衡。
3. 使用时间越长的表，ibd 文件中的碎片越多，极端情况下（例如新增 10 行删除 9 行）会使数据页的缓存几乎失效。
4. 磁盘上单文件体积过大不仅在读取 IOPS 上不如多文件，还会引发文件系统的高负载：单个文件描述符也是一种“单点”，大文件的读写性能不佳，还容易浪费大量内存。

那么，如何回答“何时进行分表”这个问题呢？很遗憾并没有一个通用的答案，这取决于每个表的读取、新增、更新情况。

虽然在数据库技术层面无法给出具体答案，但从软件工程层面可以得出一个结论：

能不分就不分，不到万不得已不搞分表，如果能通过加索引或加内存解决就不考虑分表，分表会对业务代码产生根本性的影响，并带来长期的技术债务。

B+ 树和分表的问题就讨论到这里，下面我们简单了解一下 Buffer Pool 的设计思想和运行规律。

## 8.5 内存缓存：Buffer Pool

在 MySQL 启动时，Buffer Pool 会向操作系统申请一片连续的内存空间，默认为 128MB，用作内存缓存。笔者强烈建议每台 MySQL 服务器根据自身机器资源情况，增大配置的内存值，这能十分显著地提升 MySQL 的性能。

### 缓存池大小

缓存池的大小由 `innodb_buffer_pool_size` 参数管理。通常建议设置为系统可用内存的 75%，但根据笔者的经验，对于普通的“冷热均衡”数据库，这个比例是合理的。但是对于热数据较少，却需要在短时间内（如几天）读写几乎所有表的全部数据的话，最好将比例设置在 50% 附近，否则运行一段时间后可能会爆内存（OOM 错误），MySQL 进程会被杀掉。

### 缓存池基本结构

缓存池与磁盘数据一样，分为一个个 16KB 的页进行管理。除了缓存索引页和数据页外，缓存池还包含 undo 页、插入缓存、自适应哈希索引和锁信息等。尽管缓存池已经在内存中，但它还需要一个额外的内存索引来保存每一页的表空间、页号、缓存页地址和链表节点信息，这个结构称为控制块。N 个控制块和 N 个 16 KB 数据页连在一起，就构成了 Buffer Pool 占据的那一段连续内存，如图 8-9 所示。

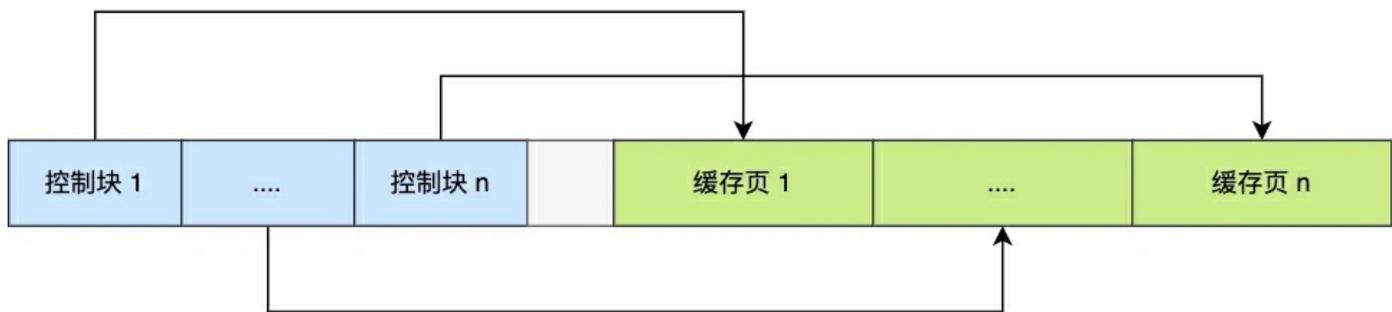


图 8-9 InnoDB 缓存池的结构

## 引入缓存池后的数据读写

缓存池中的 16KB 页与磁盘上的页一一对应，带来了读写两个方向的变化：

1. 读取数据时，如果该页已在内存中，则无需再浪费一次磁盘 I/O。
2. 写入数据时，数据会直接写入缓存中的页（不影响之后的读取），并在成功写入 redo log 后返回成功。同时，该页会被设置为脏页，等待后台进程将其真正写入磁盘。

### 缓存池 LRU 算法

除了控制块和数据页，Buffer Pool 中还存在着管理空闲页的 `free` 链表和管理脏页的 `flush` 链表，我们不再深入了解。

但是，管理缓存生命周期的 LRU 算法我们不能放过，必须狠狠地了解它一下。Buffer Pool 的 LRU 链表的数据结构如图 8-10 所示。

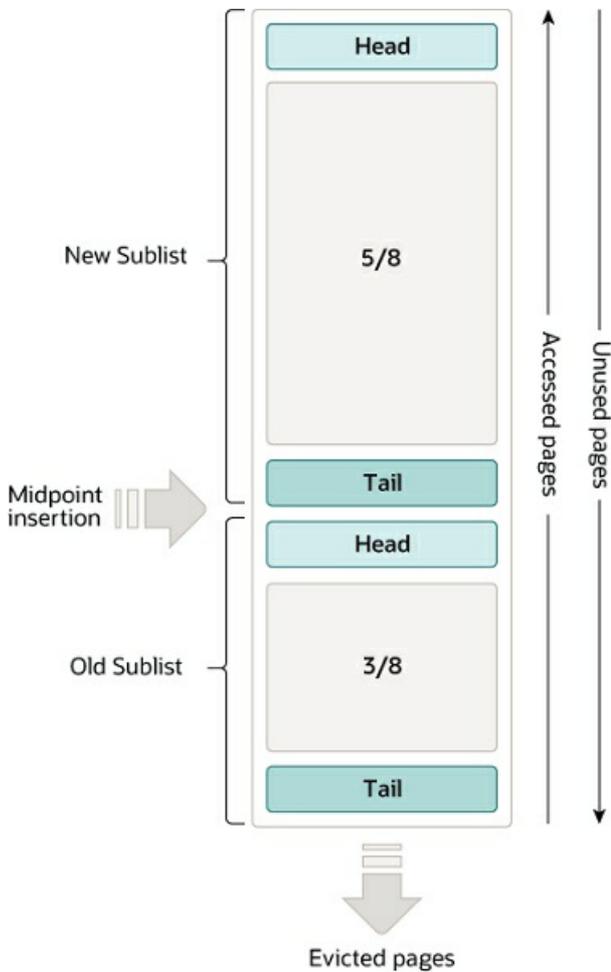


图 8-10 LRU 链表的数据结构

传统的 LRU 算法只用一个链表就实现了“移动至头部”和“淘汰尾部”两个操作，什么 InnoDB 非要搞一个变体呢？还是因为 B+ 树——由于底层数据的非连续性，导致 Buffer Pool 会遇到两个比较严重的问题：预读失效和缓冲池污染。

### 预读失效

预读失效很容易理解，因为预读本质上是基于局部性对需求的一种预估，正常的 SQL 并不能保证每一条取出的数据都是大概集中的，例如取性别为女的用户，在没有索引的情况下，就需要全表遍历，预读失效非常正常。

LRU 链表将数据分为 **新生代** 和 **老生代** 两个区域，分别占据 5/8 和 3/8 的内存空间，预读时只插入老生代的头部，同时老生代尾部元素会被淘汰。当数据真的被读取时，这一页会被立刻转移到新生代的头部，并且会挤出去一个新生代尾部的元素进入老生代的头部，数据还在缓存中。

不知道大家发现了没有，这个操作的本质是给 LRU 算法加了一层 LRU 算法，减小了缓存粒度。

### 缓冲池污染

由于磁盘数据库拥有极大的体量，相比之下内存容量却十分捉襟见肘，所以在用内存来做磁盘缓存时，一旦需求不满足局部性，缓存会被迅速劣化：当一条 SQL 需要扫描海量的数据页时，其它表用得好好的热数据突然就被别人清出内存了，结果就是磁盘 I/O 数量突然增加，系统崩溃。

于是 InnoDB 给“数据被读取时，该页会从老生代转移到新生代的头部”这个操作加了个条件：在老生代里面待的时间要足够久。

这里面有两个点非常巧妙：

1. 改变的是“转移页操作”所需要的条件，而且这个条件（留存时间）的判断非常简单，只需要在加入老生代的时候增加一个时间戳就行，4 个字节，除此之外无需任何维护。
2. 采用时间而不是次数来做限制，更加符合数据库的最终用户——人的真实需求。读取次数可能因为技术原因而增加，时间不会。没有无缘无故的爱，更没有无缘无故的读取。

## Buffer Pool 应该怎么优化

1. 内存配置的越大越好：多一倍的内存，比多一倍的 CPU 更能提高数据库性能。
2. 减少对冷数据的随机调用：优化定时任务和队列的业务代码，避免这种情况。
3. 在大批量执行 update 语句的时候，尽量自己控制事务：InnoDB 底层的数据隔离机制使得它的每一个写动作都是一个事务，所以如果你要在一次会话中写多行数据，最好自己控制事务，可以显著减少对缓冲池的影响以及磁盘 I/O 数量。
4. 避免修改主键：修改主键的值会带来大量的数据移动，磁盘会不堪重负，缓存会疯狂失效。

## 8.6 面试题

No.23 : B+ 树是如何组织数据的？

页是 MySQL 中数据存储的基本单元，整颗 B+ 树就是一个又一个相互使用指针连接在一起的页组成的。

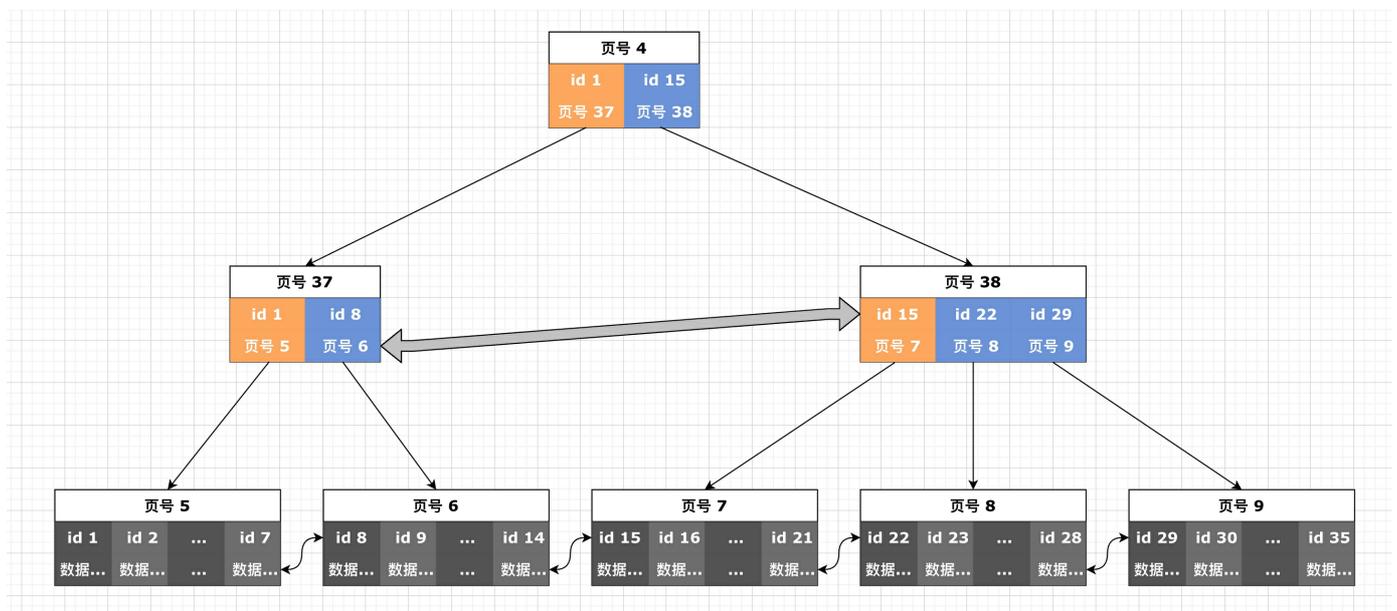


图 8-11 三层索引下的页结构图 (35 行数据)

图 8-11 展示出了页之间的指针关系：

1. 上层页对下层页拥有 **单向** 指针
2. 同一层内相邻的页之间拥有 **双向** 指针，无论是上面的索引页层还是底层的数据页层

3. 最底层数据页层中，每一页可以存储多行数据，每一行数据拥有指向下一行的 **单向** 指针

#### No.24 : InnoDB 为什么适合存储大量数据行？

InnoDB 的多级二叉树把数据进行了非常扁平的多级索引，三层索引就可以存储 2100 万行数据，非常适合在行数特别多的时候进行单个数据的读取和写入。在存储大量数据行时，InnoDB 拥有如下几个优势：

1. 数据访问效率高：InnoDB 使用 B+ 树作为索引结构，这种平衡的多路搜索树具有良好的层次性，在查询大量数据时能提高数据访问效率。
2. 顺序访问性能好：B+ 树的叶子节点使用链表连接，使得范围查询和顺序访问操作非常高效，减少磁盘 IO 次数，提升读取性能。
3. 支持高并发和事务：InnoDB 支持事务，B+ 树在并发访问场景下有较好的性能表现，使用多版本并发控制（MVCC）机制，提供高度并发性并保证数据的一致性和隔离性。
4. 空间利用率高：B+ 树具有较高的空间利用率，非叶子节点只存储关键字信息，叶子节点存储实际数据和指针，减少存储空间的浪费。

#### No.25 : 缓存 LRU 算法是什么，InnoDB 又做了哪些优化？

缓存 LRU（Least Recently Used）算法是一种常用的缓存替换算法，用于确定哪些数据应该从缓存中淘汰。其原理是基于数据的访问时间，最近被使用过的数据将具有较高的访问概率，而较长时间未被使用的数据则可能不再需要。

InnoDB 在缓存 LRU 算法上进行了一些优化：

1. 脏页刷新机制：InnoDB 使用了缓冲池（buffer pool）来缓存数据页，当数据页发生修改时会被标记为脏页。InnoDB 通过脏页刷新机制，将脏页异步写入磁盘，以减少 I/O 操作的等待时间，提高性能。
2. 自适应哈希索引：在 InnoDB 中，数据页可以通过哈希索引进行快速查找。InnoDB 会根据数据页的访问模式自动调整哈希索引，以提高数据访问效率。
3. 随机 I/O 优化：InnoDB 通过顺序扫描和预读取机制，降低了随机 I/O 的频率。它可以通过读取连续的数据页来利用磁盘的预读取特性，减少了访问数据的磁盘 IO 次数。
4. LRU 链长调整机制：InnoDB 会根据系统的访问模式动态调整 LRU 链的长度，以适应不同的工作负载。这可以提高缓存命中率，减少不必要的缓存替换操作。

#### No.26 : 多少行数据以后需要分库分表？

在单行数据为 0.9KB 的情况下，当数据量达到 2100 万行时，InnoDB 的 B+ 树会从三层变为四层。然而，这个转变并不是分库分表的依据。在内存容量足够的情况下，跨越临界点前后的 InnoDB 几乎没有性能差别。

相反，以下策略对于数据库优化来说更有价值：

1. 对于单行数据较大的表进行横向分表。对于行数超过 50 万行的表来说，建议将单行数据限制在 2KB 以内。
2. 对于经常更新和删除数据的表，应该定期执行 B+ 树重整操作。通过执行 `recreate` + `analyze` 操作可以显著提升性能。

3. 从业务角度来看，如果可能的话，尽量避免分表——过早优化是万恶之源。

## 第 9 章 四代分布式数据库的变迁

说到后端系统对于数据库的要求，基本上和你老板一样：既要又要还要。数据库扮演的那个单点角色，在单机上实现已经如此的困难了，换到分布式环境下只会更困难。而分布式数据库的出现也是被迫的：应用规模越来越大，对性能和可用性的要求越来越高，不得不搞分布式数据库了。

接下来请大家坐稳扶好，我们正式开始分布式数据库历史变迁之旅。

### 9.1 单机数据库的不可能三角

正如经济政策的不可能三角“不可能同时实现资本流动自由，货币政策的独立性和汇率的稳定”那样，单机数据库也有一个不可能三角，那就是：①持久化 ②事务隔离 ③高性能，三者不可兼得。

#### 为什么不可能

1. 持久化需要每一次写数据都要落到磁盘上，宕机再启动以后，数据库可以自动修复。如果只要求这一条，很好实现。
2. 事务隔离需要每一次会话（session）的事务都拥有自己的数据库版本：既要多个并行的事务相互之间不会写到对方的虚拟数据库上（读提交），又要不能读到对方的虚拟数据库上（可重复读），还要在一个事务内不能读到别的事务已经提交的新增的数据（幻读），终极需求则是完全串行化——我的读 session 不结束，你就不能读。这个需求和持久化需求结合以后，会大幅增加日志管理的复杂度，但仍然是可控的。
3. 读写都要尽量地快：单独实现也很快，内存数据库嘛（如 Redis），但是加上持久化和事务隔离，就很难做了——需要对前两项进行妥协。

#### MySQL 选择了哪两个？

首先，MySQL 选择了持久化：失去人性，失去很多；失去持久化，失去一切。没有持久化能力的核心数据库就做不了核心数据库，这一条是所有磁盘数据库的刚需，完全无法舍弃。

然后，MySQL 选择了一部分高性能：MyISAM 就是为了快速读写而创造的，早期 MySQL 在低配 PC 机上就有不错的性能。后来更高级的 InnoDB 出现了，小数据量时它的读取性能不如 MyISAM，写性能更是彻底拉胯，但是在面对大数据量场景时，读性能非常强，还能提供很多后端程序员梦寐以求的高级功能（例如丰富的索引），承担了大部分互联网核心数据库的角色。

最后，MySQL 将事务隔离拆成了几个级别，任君挑选：你要强事务隔离，性能就差；你能接受弱事务隔离，性能就强。你说无事务隔离？那你用 MySQL 干什么，Redis 它不香吗。

所以 MySQL 其实选择了 持久化\*1 + 高性能\*0.8 + 事务隔离\*0.5，算下来，还赚了 0.3。

不过，从 MySQL 也可以看出，“数据库的不可能三角”并不是完全互斥的，是可以相互妥协的。

在开始细数分布式数据库之前，我们先看一个非分布式的数据库性能提升方案——读写分离，主从同步。

## 9.2 从读写分离到分布式

由于 Web 系统中读写需求拥有明显的二八分特征——读取流量占 80%，写入流量占 20%，所以如果我们能把读性能拆分到多台机器上，在同样的硬件水平下，数据库总 QPS 也就能提高五倍。

### 各种主从架构

各种主从架构是最简单最直接的读写分离架构。如图 9-1 所示就是阿里云 RDS 的几种主从架构。

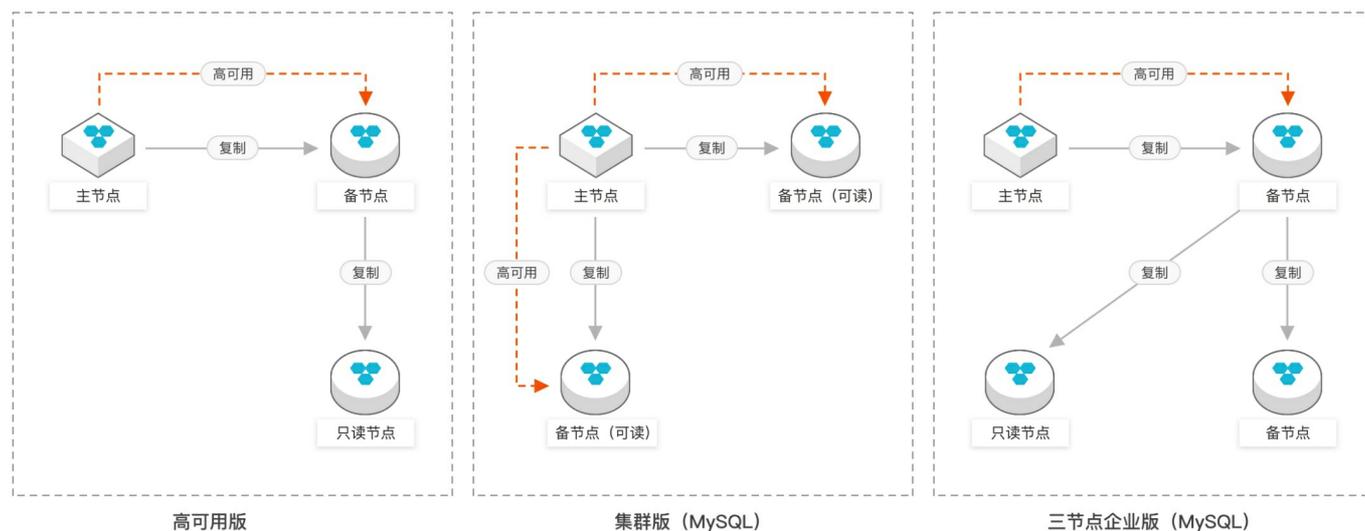


图 9-1 阿里云 RDS 的几种主从架构

无论是远古时代谷歌的 MMM (Multi-Master Replication Manager for MySQL) 还是中古时代的 MySQL 官方的 MGR (MySQL Group Replication)，还是最近刚刚完成开发且收费的官方 InnoDB Cluster，这些主从架构的实现方式都是一致的：基于行同步或者语句同步，[近实时](#)地从主节点向从节点同步新增和修改的数据。

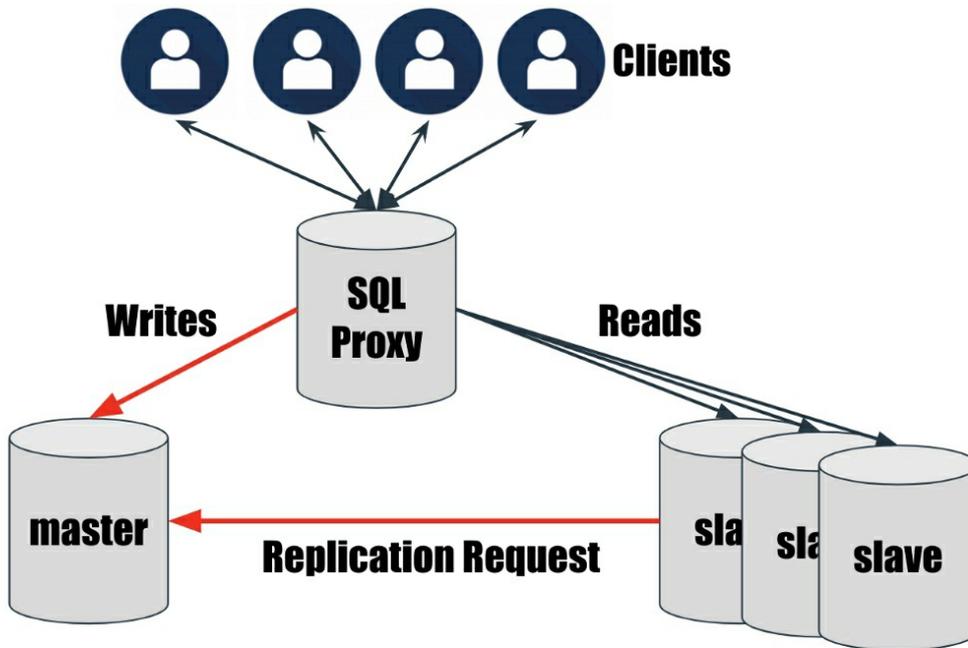


图 9-2 包含网关的主从架构

由于这种方法必然会让主从之间存在一段时间的延迟（数百毫秒到数秒），所以一般在主从节点前面还要加一个网关进行语句分发，其架构如图 9-2 所示。该集群的运行方式如下：

1. `select` 等读语句默认发送到从节点，以尽量降低主节点负载
2. 一旦出现 `update`、`insert` 等些语句，立刻发送到主节点
3. 并且，本次会话（session）内的所有后续语句，必须全部发送给主节点，不然就会出现数据写入了但是读不到的情况

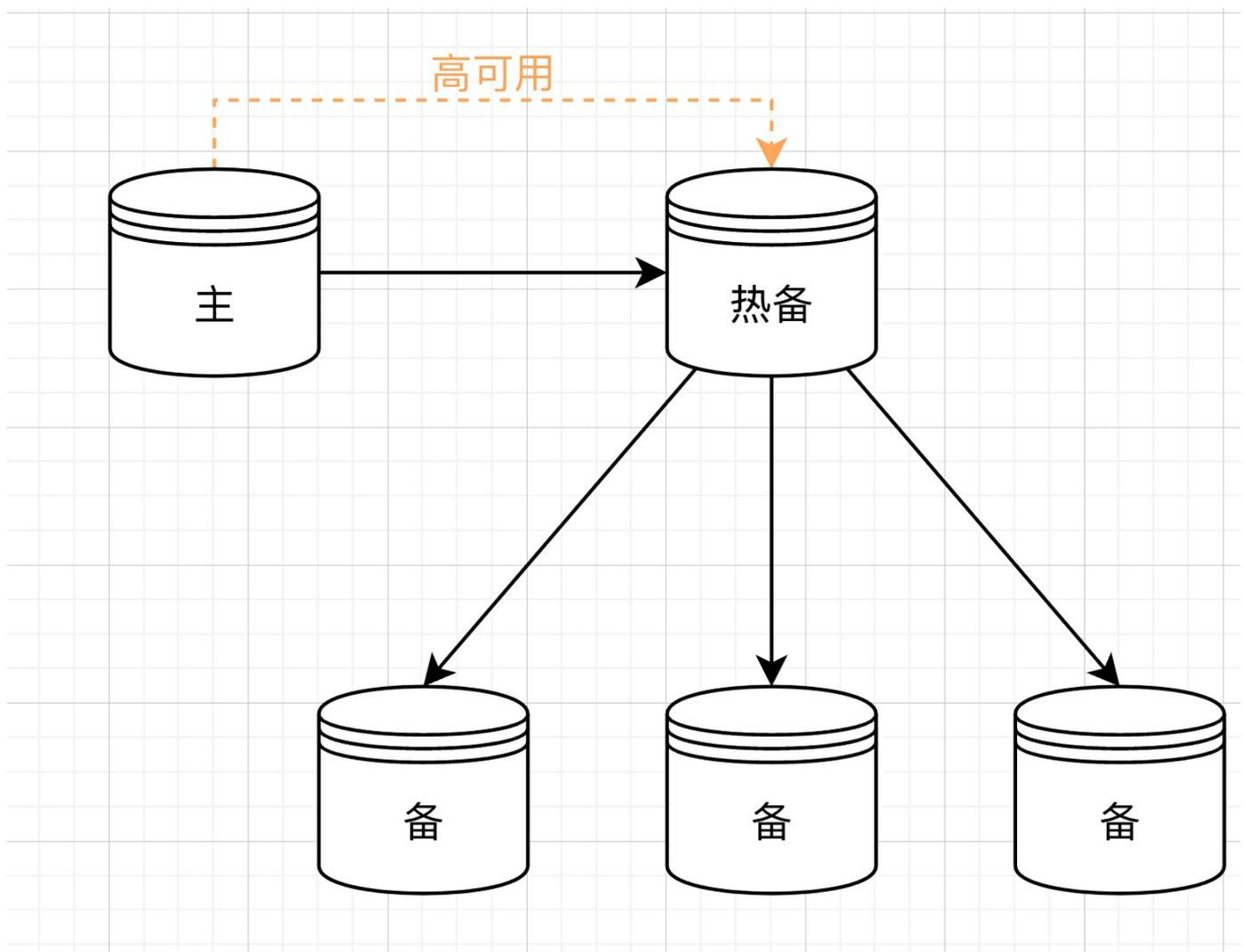


图 9-3 一主四从架构图

像图 9-3 那样搭建一个一主四从的 MySQL 集群，总 QPS 就能从单节点的 1 万提升到 5 万，顺便还能拥有主节点故障后高可用的能力。主从架构比较简单，也没有什么数据冲突问题，就是有一个很大的弱点：

写入性能无法提升：由于数据库承载的单点功能实在是太多了（自增、时间先后、事务），导致哪怕架构玩出了花，能写入数据的节点还是只能有一个，所有这些架构都只能提升读性能。

那怎么提升写性能呢？这个时候就要拿出分布式数据库了。

## 分布式数据库

由于数据库的单点性非常强，所以在谷歌搞出 GFS、MapReduce、Bigtable 三驾马车之前，业界对于高性能数据库的主要解决方案是买 IOE 套件：IBM 小型机 + Oracle 数据库 + EMC 商业存储。而当时的需求也确实更加适合使用商用解决方案。

后来搜索引擎成为了第一代全民网站，而搜索引擎的数据库却“不那么关系型”，所以谷歌搞出了自己的分布式 KV 数据库。后来谷歌发现 SQL 和事务隔离在很多情况下还是刚需，于是在 KV 层之上改了一个强一致支持事务隔离的 Spanner 分布式数据库。而随着云计算的兴起，分布式数据库已经成了云上的“刚需”：业务系统全部上云，总不能还用 Oracle 物理服务器吧？于是云上数据库又开始大踏步发展起来。

下面我们按照时间顺序，逐一梳理分布式数据库的发展史。

### 9.3 第一代分布式数据库：中间件

朴素的中间件其实就是第一代分布式数据库。

#### 在 MySQL 体系内演进

关系型数据库为了解决不可能三角需求，其基本架构 40 年没有变过。

MySQL 自己其实已经是一个非常优秀的满足互联网业务场景的单体数据库了，所以基于 MySQL 的基本逻辑进行架构改进，是最稳妥的方案。

在没有分布式关系型数据库技术出现的时代，后端开发者们往往只能选择唯一的刀耕火种的路：在应用代码里调用多个数据库，以应对单个数据库性能不足的困境。后来，有人把这些调用多个数据的代码抽出来作为单独的一层，称作数据库中间件。

数据库中间件

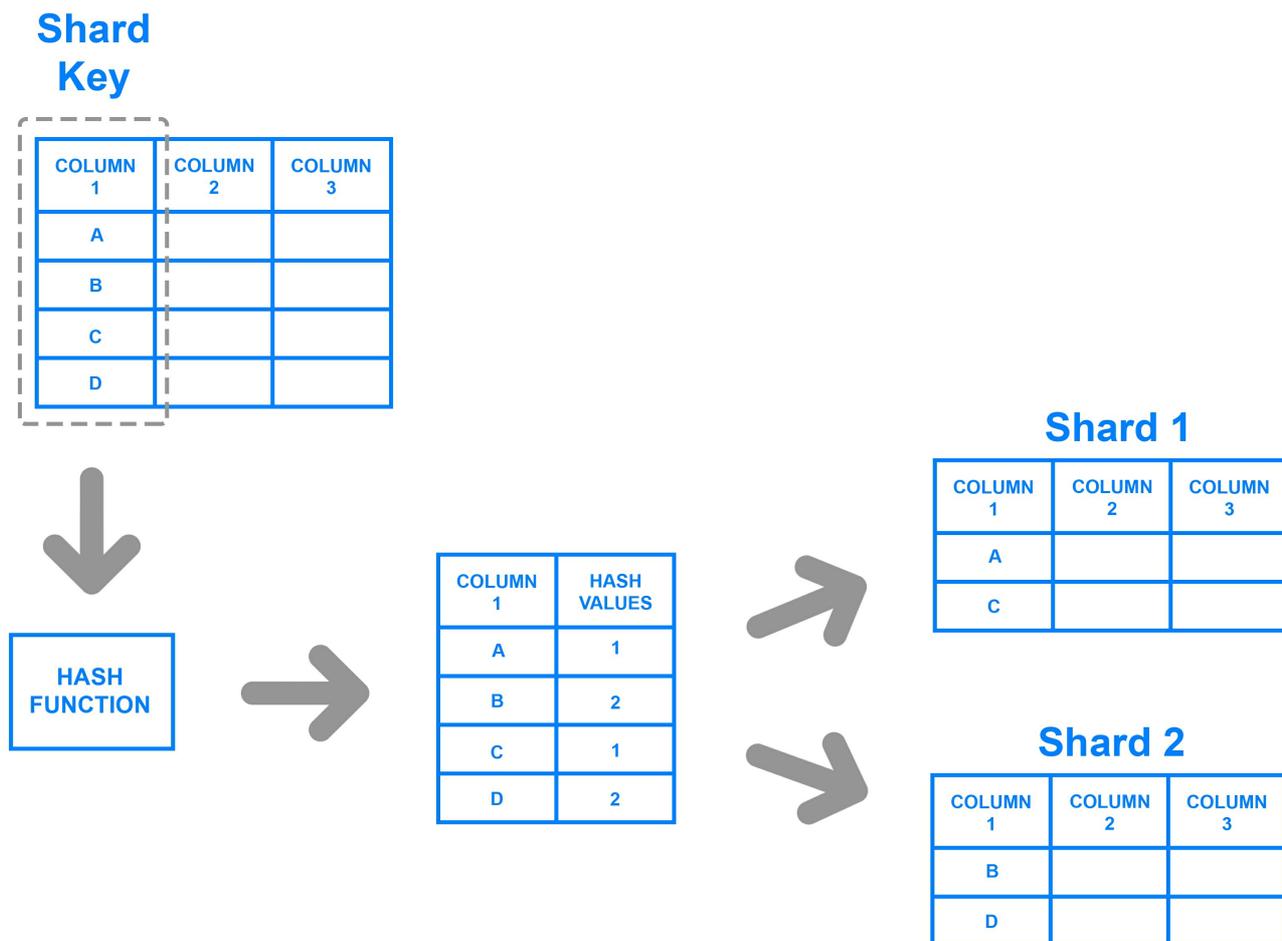


图 9-4 数据库中间件的分表操作

首先，对数据表进行纵向分表：按照一定规则，将一张超多行数的表分散到多个数据库中，如图 9-4 所示。

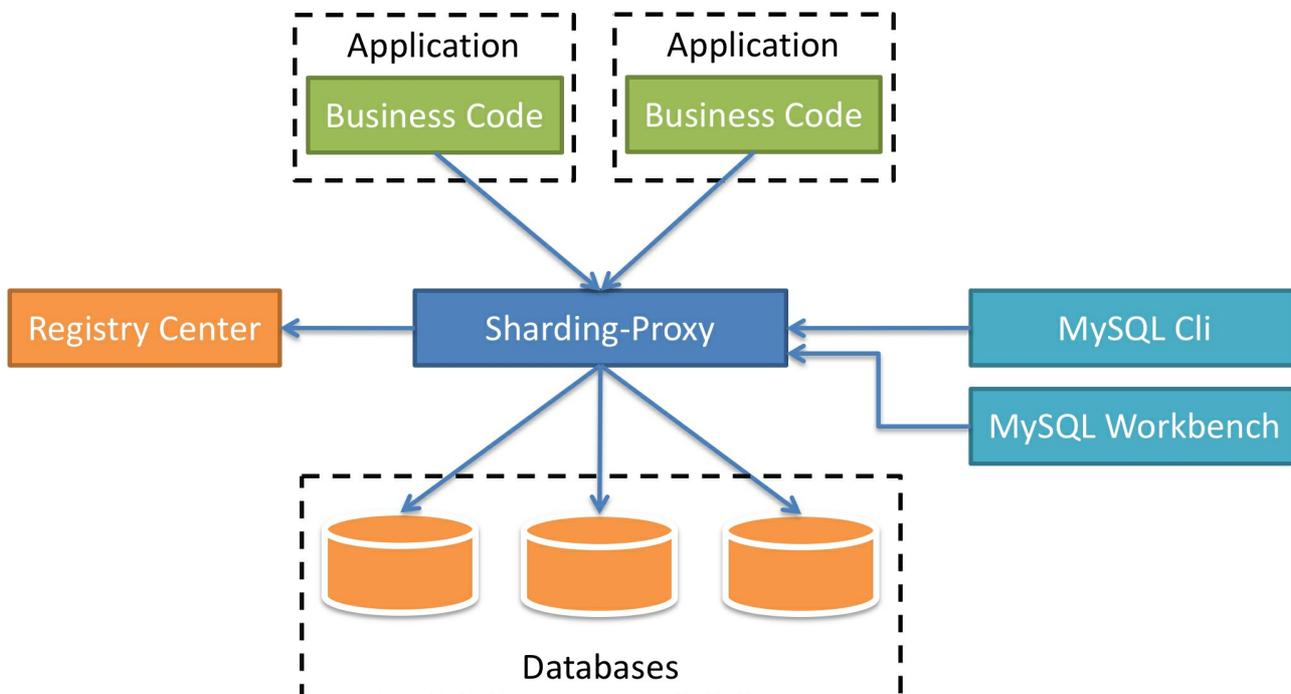


图 9-5 ShardingSphere 中的 Sharding-Proxy 工作方式

然后，无论是插入、更新还是查询，都通过一个代理（Proxy）将 SQL 进行重定向和拆分，发送给多个数据库，再将结果聚合，返回。如图 9-5 所示就是 ShardingSphere 中的 Sharding-Proxy 工作方式。

大名鼎鼎的数据库中间件，其基本原理一句话就能描述：使用一个常驻内存的进程，假装自己是个独立数据库，再提供全局唯一主键、跨分片查询、分布式事务等功能，将背后的多个数据库“包装”成一个逻辑上的单体数据库。

虽然“中间件”这个名字听起来像一个独立组件，但实际上它依然是强业务亲和性的：没有几家公司会自己研发数据库，但每家公司都会研发自己的所谓中间件，因为中间件基本上就代表了其背后的一整套“多数据库分库分表开发规范”。所以，中间件也不属于“通用数据库”范畴，在宏观架构层面，它依然属于应用的一部分。笔者称呼这个时代为刀耕火种时代。

那该怎么脱离刀耕火种呢？人类的大脑是相似的：既然应用代码做数据规划和逻辑判断很容易失控，那我们在数据库层面把这件事接管了行不行呢？当然可以，但是需要拿东西 [找信息之神交换](#)。

历史上，第一个被放弃的是 [事务隔离](#)，而它带来的就是第二代分布式数据库：KV 数据库。

## 9.4 第二代分布式数据库：键值（KV）数据库

第二代分布式数据库的出现是由离散数据需求推动的，诞生了键值对（Key-Value）型分布式数据库。

### 分布式时代的“新·不可能三角”

在分布式数据库时代，持久化已经不是分布式数据库“真正的持久化”了，取而代之的是“数据一致性”：由于数据存在了多台机器上，那机器之间数据的一致性就成了新时代的“持久化”。于是新不可能三角出现

了：①一致性 ②事务隔离 ③高性能。

你是不是在期待 CAP 理论呀？别着急，我们后面会说。

## 分布式 KV 数据库放弃了事务隔离

数据库技术一共获得过四次图灵奖，后面三次都在关系型数据库领域。事务隔离模型是关系型数据库的核心，非常地简洁、优美、逻辑自恰。

Google 是第一个全民搜索引擎，系统规模也达到了史上最大。但是，搜索引擎技术本身却不需要使用关系型数据库来存储：搜索结果中的网页链接之间是离散的，这已经在前面的第 3 章第 4 节“[实战：Go 协程开发高性能爬虫](#)”中有所体现。

由于搜索不需要关系型数据库，自然谷歌搞的分布式数据库就是使用的 KV 模型。谷歌的三驾马车论文发布以后，业界迅速发展出了一个新的数据库门类 NoSQL (Not Only SQL)，专门针对非结构化和半结构化的海量数据。

目前，缓存 (Redis) 和日志 (MongoDB) 大家一般都会用 NoSQL 来承载。在这个领域，最成功的莫过于基于 Hadoop 生态中 HDFS 构建的 HBase 了：它主要提供的是行级数据一致性，即 CAP 理论中的 CP，放弃了事务，可以高性能地存储海量数据。

KV 数据库结构简单，性能优异，扩展性无敌，但是它只能作为核心数据库的高性能补充，绝大多数场景下，核心数据库还是得用关系型。

## 9.5 第三代分布式数据库：NewSQL

---

从 2005 年开始，Google Adwords 开始基于 MySQL 搭建系统，这也推动了 MySQL 在 Google 内部的大规模使用。随着业务的发展，MySQL 集群越来越庞大，其中最痛苦的就是“数据再分片”，据说有一次谷歌对数据库的重新分片持续了 2 年才完成。于是谷歌痛定思痛，搞出了一个支持分布式事务和数据强一致性的分布式关系型数据库：Google Spanner。

2012 年，谷歌发布了 [Spanner 论文](#)，拉开了分布式强一致性关系型数据库的大幕。这个数据库过于厉害，以至于笔者第一次看到它机柜照片 (图 9-6) 的时候直接震惊了。

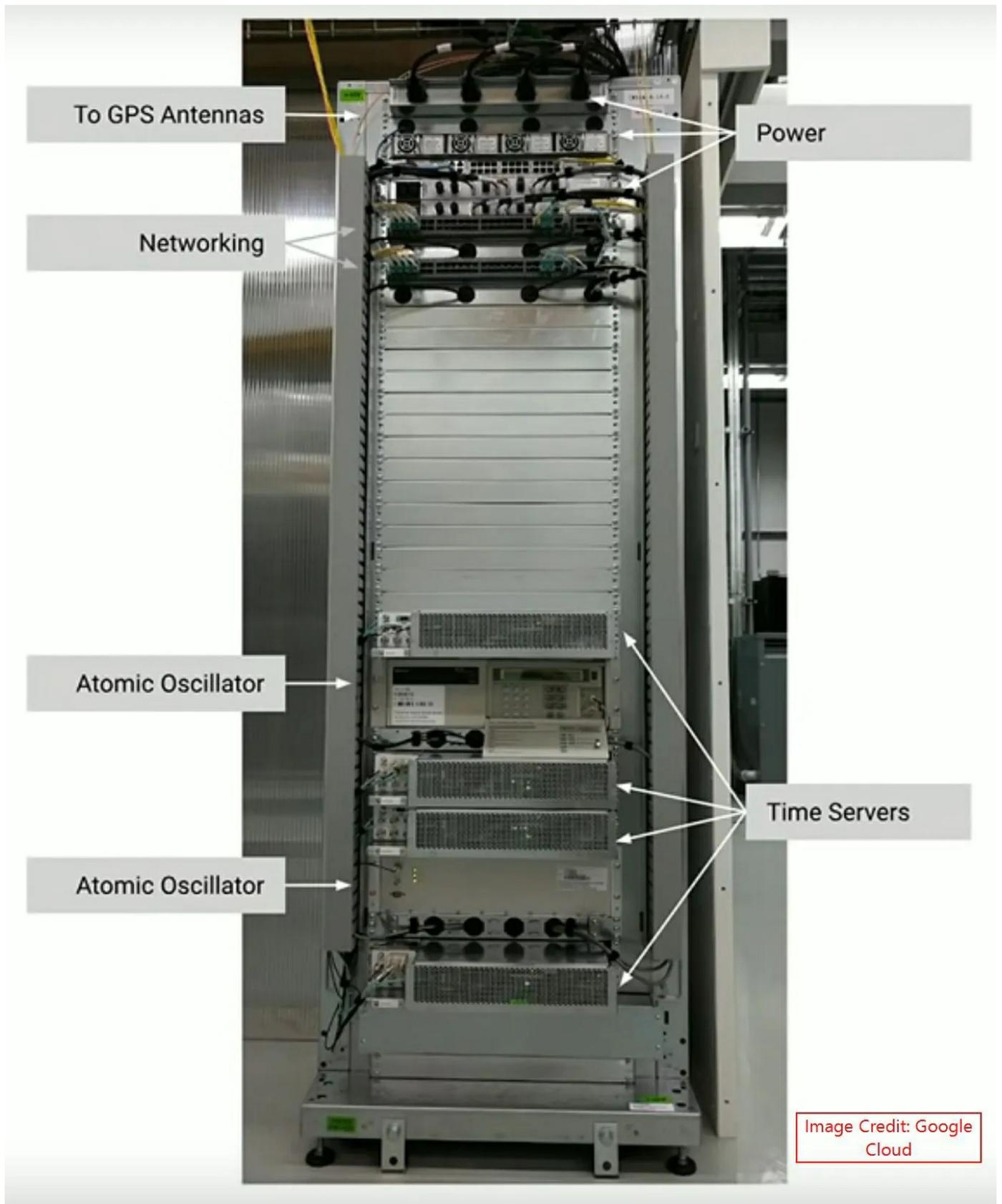


图 9-6 Google Spanner 的机柜照片

这套系统采用了 GPS 授时 + 2 台原子钟 + 4 台时间服务器，让分布在全球多个数据中心的 Spanner 集群进行相当精确的时间同步：基于 TrueTime 服务，时间差可以控制在 10ms 之内。这种真正的全球数据库可以做到即使单个数据中心完全失效，应用也完全无感知。

当然，如此规模的全球数据库全世界也没有几家公司有需求，如果我们只在一个数据中心内署数据库集群，时间同步可以很容易地做到 1ms 之内，原子钟这种高端货还用不到。

## 数据持久性的关键步骤——redo log

上一章我们说过，写入型 SQL 会在写入缓存页 + 写入磁盘 redo log 之后返回成功，此时，真正的 ibd 磁盘文件并未更新。所以，Spanner 使用 Paxos 协议在多个副本之间同步 redo log，只要 redo log 没问题，多副本数据的最终一致性就没有问题。

## 事务的两阶段提交

由于分布式场景下写请求需要所有节点都完成才算完成，所以两阶段提交是必须存在的。单机架构下的事务，也是某一旦一条 SQL 执行出错，整个事务都是要回滚的嘛。多机架构下这个需求所需要的成本又大幅增加了，两阶段提交的流程是这样的：

1. 告诉所有节点更新数据
2. 收集所有节点的执行结果，如果有一台返回了失败，则再通知所有节点，取消执行该事务

这个简单模型拥有非常恐怖的理论故障概率：一旦在第一步执行成功后某台机器宕机，则集群直接卡死：大量节点会被锁住。

Spanner 使用 Paxos 化解了这个问题：只要 leader 节点的事务执行成功了，即向客户端返回成功，而后续数据的一致性则会基于 `prepare timestamp`（启动时间）和 `commit timestamp`（提交时间）加上 Paxos 算法来保证。

## 多版本并发控制（MVCC）

Spanner 使用时间戳来进行事务之间的 MVCC：为每一次数据的变化分配一个全球统一的时间戳。这么做的本质依然是“空间+时间”换时间，而且是拿过去的时间换现在的时间，特别像支持事后对焦的光场相机。

1. 传统的单机 MVCC 是基于单机的原子性实现的事务顺序，再实现的事务隔离，属于即时判断。
2. Spanner 基于 TrueTime 记录下了每行数据的更新时间，增加了每次写入的时间成本，同时也增加了存储空间。
3. 在进行多节点事务同步时，就不需要再和拥有此行数据的所有节点进行网络通信，只依靠 TrueTime 就可以用 Paxos 算法直接进行数据合并——基于时间戳判断谁前谁后，属于事后判断。

## Spanner 放弃了什么

Spanner 是一个强一致的全球数据库，那他放弃了什么呢？这个时候就需要 CAP 理论登场了。

一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

Google Spanner 数据库首先要保证的其实是分区容错性，这是“全球数据库”的基本要求，也最影响他们赚钱；然后是一致性，“强一致”是核心设计目标，也是 Spanner 的核心价值；谷歌放弃的是可用性(A)，只有 majority available。

除此之外，为了“外部一致性”，即客户端看到的全局强一致性，谷歌为每一个事务增加了 2 倍的时钟延迟，换句话说就是增加了写操作的返回时间，这就是分布式系统的代价：目前平均 TrueTime 的延迟为 3.5ms，所以对 Spanner 的每一次写操作都需要增加 7ms 的等待时间。

### Spanner 一致性的根本来源

大家应该都发现了，其实 Spanner 是通过给 Paxos 分布式共识算法加了一个“本地外挂” TrueTime 实现的海量数据的分布式管理，它实现全局强一致性的根本来源是 Paxos 和 TrueTime。而在普通单机房部署的分布式系统中，不需要 GPS 授时和原子钟，直接搞一个时间同步服务就行。

### NewSQL 时代

Google Spanner 的推出代表着一个新时代到来了：基于分布式技术的 SQL 兼容数据库（NewSQL），而兼容到什么地步就看各家的水平了。

NewSQL 最大的特点就是使用非 B 树磁盘存储结构（一般为 LSM-Tree），在上面构筑一个兼容 SQL 常用语句和事务的兼容层，这样既可以享受大规模 LSM-Tree 集群带来的扩展性和高性能，也可以尽量少改动现有应用代码和开发习惯，把悲伤留给自己了属于是。

目前比较常见的 NewSQL 有 ClustrixDB、NuoDB、VoltDB，国内的 TiDB 和 OceanBase 也属于 NewSQL，但它们俩有本质区别，下一章我们会详细讨论。

在 NewSQL 时代之后，随着云计算的兴起，云上数据库突然成为了市场的宠儿，市场占有率迅速上涨。它们其实都是对 MySQL 的改造，并不属于 NewSQL 范畴，下面我们认识一下它们。

## 9.6 第四代分布式数据库：云上数据库

笔者实在是不想用“云原生”这个风口浪尖上的词来形容美丽的云上数据库们，它们就像 TCP/IP，简洁但有用。市场从来不会说谎，它们一定是有过人之处。

### 亚马逊 Aurora 开天辟地

2014 年 10 月，亚马逊发布了 Aurora 云上数据库，开创性地在云环境中将计算节点和存储节点分离：基于云上资源的特点，将计算节点 scale up（增配），将存储节点 scale out（增加节点），实现了极佳的性能/成本平衡。Aurora 将云上关系型数据库产品推向了一个新的高度。

### 计算与存储分离

Aurora 提出的计算与存储分离可以说是目前数据库领域最火的方向，但是它火的原因笔者相信大多数人都认识的不对：不是因为性能强，而是因为便宜。

### 挖掘云计算的价值点

十年前笔者在 SAE 实习的时候，中午大家一起吃饭，组长说云计算就是云安全，这句话当然说的很对。从这句话推开，我们很容易就能找到云计算真正的商业价值在哪里：传统托管式部署，哪些资源浪费的最多，哪

里就是云计算的商业价值所在。

为了满足业务波动而多采购的 CPU 和内存，可能浪费了 50%；网络安全设备，可以说 95% 以上的资源都是浪费；高端存储，这个已经不能用资源浪费来形容了，而是被云计算颠覆了：云厂商用海量的多地域多机房内廉价的 x86 服务器里面的廉价磁盘，基于软件，构建出了超级便宜、多副本、高可用的存储，唯一的问题是性能不是太好。亚马逊 S3 和阿里云 OSS 就是最佳代表，可以说这类对象存储服务，其单价已经低于本地机房的 2.5 寸 SAS 机械磁盘了，更不要说本地机房还需要另外采购昂贵的存储控制器和 SAN 交换机了。

## 云计算与特斯拉

云数据库可以算是云服务厂商最重要的产品：受众足够广，成本足够低，性能足够高。这一点很像特斯拉汽车，时至今日，特斯拉依然在疯狂地想各种办法压低生产成本，虽然在降价，但是单车毛利依然维持在 30% 以上，是 BBA 的 2-3 倍。

Aurora 和 PolarDB 的核心价值是用一种低成本的方式，制造了一个 Oracle 要高成本才能做出来的软件和服务，这才是真的“创造价值”。

## 计算与存储分离是一种“低成本”技术

计算与存储分离并不是什么“高性能”技术，而是一种“低成本”技术：关系型数据的存储引擎 InnoDB 本身就是面向低性能的磁盘而设计的，而 CPU 和内存却是越快越好、越大越好，如果还把磁盘和 MySQL 进程部署在同一台物理机内，一定会造成磁盘性能浪费。计算与存储分离的真正价值在于大幅降低了存储的成本。

## 计算与存储分离的技术优势

虽然说这个架构的主要价值在于便宜，但是在技术上，它也是有优势的：

它显著降低了传统 MySQL 主从同步的延迟。传统架构下，无论是语句同步还是行同步，都要等到事务提交后，才能开始同步，这就必然带来很长时间的延迟，影响应用代码的编写。而计算和存储分离之后，基于 redo log 传递的主从同步就要快得多了，从 1-2s 降低到了 100ms 以下。由于主从延迟降低，集群中从节点的个数可以提升，总体性能可以达到更高。

## Aurora 的主从同步机制

上一章中我们说过，在更新数据时，主节点在完成了 redo log 写入，并对内存缓存 Buffer Pool 中相应的数据页进行修改之后，就会返回成功。这个内存缓存给 Aurora 从节点的数据更新造成了不小的影响：

1. 主从节点之间只有 redo log 传递
2. 从节点在拿到 redo log 之后，会刷新自己 Buffer Pool 中存在的数据库页，其它不存在的页的信息会丢弃
3. 这带来了两个问题：
  1. 从节点的客户端在主从不同步的一段时间内，读到的是旧数据，这个需要网关或者应用代码来处理
  2. 从节点的 Buffer Pool 有效性变差，命中率下降，引发性能下降

## Aurora 的架构局限

Aurora 的出现确实独具慧眼，但是也会被时代所局限。

在 [Aurora 论文](#) 中，开篇就提到 `Instead, the bottleneck moves to the network between the database tier requesting I/Os and the storage tier that performs these I/Os`。Aurora 认为网络速度会成为云数据库的瓶颈，而在它研发的 2012-2013 年也确实如此，当时万兆网卡刚刚普及，CPU 单核性能也不足，软件也没有跟上，可以说速度比千兆网卡也快不了多少，所以亚马逊又搞了神奇的技术：存储节点具有自己将 redo log 写入 ibd 文件的能力。

由于这个神奇能力的存在，Aurora 的多机之间采用传输 redo log 的方式来同步数据，并用一种今天看起来不太靠谱的协议来保证最终一致性：consul 使用的那个 gossip 协议。由于 Aurora 采用六副本技术，所以每次写入都需要发起六次不怎么快的网络 I/O，并且在其中 4 个成功以后才给客户端返回成功。Aurora 确实便宜，但是单节点的性能也确实捉鸡，这代表的就是写入性能差，进而限制了整个集群的性能上限。而且，经过比较长的时间(100ms)才能保证从 [从节点](#) 上读到的数据是最新的，这会让主节点压力增大影响集群性能上限，或者让应用代码做长时间的等待，严重的会引起应用代码的执行逻辑变更，引入持久的技术债务。

那该怎么提升计算存储分离情况下的集群性能呢？我们看阿里云是怎么做的。

## 阿里云 PolarDB 后来居上

阿里云 RDS 集群的成本已经够低了，不需要再用计算存储分离技术降低成本了，而中国市场的用户，普遍需要高性能的 MySQL 数据库：ECS 价格太低了，如果不是运维方便和性能压力过大，谁愿意用你昂贵的数据库服务啊。

2015 年，PolarDB 开始研发，当时 25Gb RDMA 网络已经逐渐普及，所以阿里云将视角放在了网络速度之外：在 I/O 速度没有瓶颈以后，基于内核提供的 syscall 所编写的旧代码将会成为新的性能瓶颈。

站在 2023 年初回头看，阿里云的判断是非常准确的。

### 计算存储分离架构下的整体性能极限

由于所有节点都使用了同一块“逻辑磁盘”，所以 [双主可写](#) 想都不要想，一个计算存储分离的数据库集群的性能上限就是 [主节点的写入性能上限](#)。（Aurora 有多主可写数据库，对 ID 进行自动切分，使用时有一堆限制；PolarDB 也有多主可写数据库，但是更绝：每个库/表只支持绑定到一个可写节点，感情就是对多个数据库做了个逻辑聚合，还不如中间件呢。）

在主节点不接受读的情况下，主节点只承接写入操作，以及和写入操作在同一个会话 session 中的后续的读请求。

那 PolarDB 是如何提升主节点性能的呢？

### 1. 共享的不是 redo log，而是 ibd 文件

主从之间并不是依靠纯属 redo log 来同步数据的，而是直接共享同一个 ibd 文件，即真正的共享磁盘。而且，基于块设备的 Raft 算法也比基于文件的 gossip 协议要快很多。

## 2. 绕过内核和网络栈：大幅提升存储性能，降低延迟，减少 CPU 消耗

虽然对 redo log 的解析这一步在 Aurora 那边是存储做的，PolarDB 这边是主节点做的，看似增加了 CPU 消耗，但是这并不是真正的性能瓶颈所在，真正的瓶颈是网络栈和 UNIX 进程模型。看过笔者《性能之殇》系列文章的人应该都比较熟悉了，这是老生常谈了。那 PolarDB 是怎么优化的呢？

1. 跳过 TCP/IP 网络栈，直接使用 RDMA 网络从存储节点读取数据，延迟暴降
2. 跳过 kernel 的线程调度，自行开发绑定 CPU 核心的状态机，采用非阻塞 I/O，在 CPU 占用下降的情况下，延迟进一步降低

## 3. 提出 ParallelRaft 协议，允许部分乱序提交

ParallelRaft 协议让 Aurora 那边需要执行六次的网络 I/O 变成了一次：只需要向 leader 节点写入成功，剩下的数据同步由 Raft 算法来执行，这和 Google Spanner 的两阶段提交优化是一个思路。

原始的 Raft 协议确实逻辑完备，实现简单，就是一个一个地协商太慢了。ParallelRaft 让收敛协商能够并行起来，加速 redo log 落入 ibd 文件的过程。

## 4. 主从之间基于低延迟的共享存储同步 redo log 数据以刷新 Buffer Pool

基于共享存储的低延迟优势，PolarDB 主从之间使用共享存储来同步 redo log 以刷新缓存，这一点逻辑上和 Aurora 一致，但是实际表现比较好，笔者实测主从同步时间在 20~70ms 范围内。

## 5. 单机性能比标准 MySQL 更强

RDMA 存储比本地存储更快，因为减少了计算和存储争抢中断的问题：I/O 这个 CPU 不擅长的东西完全卸载给了 RDMA 网卡。同配置下 PolarDB 比标准 MySQL 的性能要高几倍。

**PolarDB** 有这么多的优势，那它付出了什么代价呢？

在各种实测里面，PolarDB 在相同规格下对其他的云上数据库都拥有 2 倍的性能优势，但是它基于 RDMA 存储的特点也让它付出了两个代价：1. 硬件成本高昂 2. 扩展性有上限。

是不是感觉很熟悉？Shared-Disk 的代表 Oracle RAC 也有这两个缺点。不知道大家有没有发现，PolarDB 就是云时代的 RAC 数据库：看起来是 Shared-Disk，其实是共享缓存让它们的性能变的超强。

如图 9-7 所示是各代分布式数据库的兼容性和扩展性的对比，我们可以直观地看出 PolarDB、Aurora、Google Spanner 的区别。

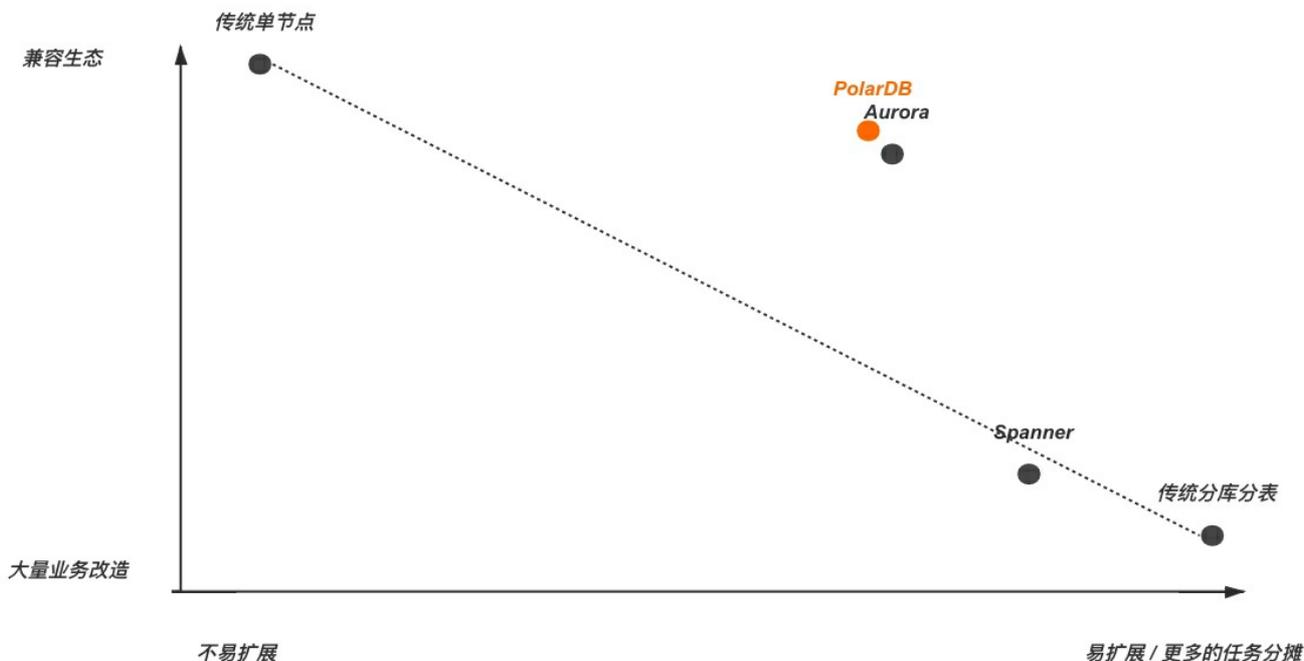


图 9-7 各代分布式数据库的兼容性/扩展性对比

一句话概括 PolarDB：利用了高性能云存储并且做了性能优化的一主多从 MySQL 集群。

### 简单讨论一下 CAP 理论

一个分布式系统中，不可能完全满足①一致性、②可用性、③分区容错性。我们以一个两地三中心的数据库为例：

1. 一致性：同一个时刻发送到三个机房的同一个读请求返回的数据必须一致（强一致读），而且磁盘上的数据也必须在一段时间后变的完全逻辑一致（最终一致）
2. 可用性：一定比例的机器宕机，其它未宕机服务器必须能够响应客户端的请求（必须是正确格式的成功或失败），这个比例的大小就是可用性级别
3. 分区容错性：一个集群由于通信故障分裂成两个集群后，不能变成两个数据不一致的集群（脑裂），对外必须依然表现为一个逻辑集群

在一个分布式数据库系统中，到底什么是可以放弃的呢？笔者觉得可以从分布式系统带来了什么优势这个问题开始思考。

相比于单体系统，一个分布式的数据库，在一致性上进步了吗？完全没有。在可用性上进步了吗？进步了很多。在分区容错性上呢？单体系统没有分区，不需要容错。所以，结论已经一目了然了：

①和③都是分布式系统带来的新问题，只有②是进步，那就取长补短，选择牺牲可用性来解决自己引发的一致性和分区容错性两个新问题。这也正是目前常见分布式数据库系统的标准做法。

## 9.7 番外篇

说完了四代分布式数据库的变迁，我们再来讨论一些周边的重要概念。

## Shared-Nothing、Shared-Memory 和 Shared-Disk

Shared-Nothing 只是一种思想，并不是一种明确的数据库架构，它非常笼统，只是描述了一种状态。在这里我们简单讨论一下 Shared-Nothing。

Shared-Nothing 描述的是一种分布式数据库的运行状态：两台物理机，除了网络通信之外，不进行任何资源共享，CPU、内存、磁盘都是独立的。这样，整个系统的理论性能就可以达到单机的二倍。

怎么理解 Shared-Nothing 思想呢？把它和 Shared-Disk 放到一起就明白了：

Shared-Disk：多台机器通过共享 SAN 磁盘的方式协同工作，让系统整体性能突破单机的极限。Oracle RAC 是这个架构的佼佼者，不过它的成功并不在于磁盘，而在于它的分布式锁（CACHE FUSION）：RAC 利用时间戳和分布式锁实现了分布式事务和多台机器同时可写，大幅提升了集群的性能。注意，时间戳在这里又出现了。CACHE FUSION 其实已经可以被称作 Shared-Memory 了。对这个话题感兴趣的读者可以自己了解，我们不再深入。

21 世纪初，Oracle 推出了 Shared-Disk 的 RAC，IBM 推出了 Shared-Nothing 的 DB2 ICE。十年后，Oracle RAC 发展的如火如荼，而 DB2 ICE 已经消失在了历史的长河中。

但是，2012 年 Google 发布了 Spanner 论文，在非常成熟的世界上最大规模的 KV 数据库之上，构建 SQL 层，实现持久化、事务和多版本并发控制，扛起了 Shared-Nothing 技术方向的大旗，直到今天。

## MongoDB 小故事

十年前笔者在新浪云（SAE）实习的时候，听过一个关于 MongoDB 的技术小故事：当时，SAE 的 KV 服务是使用 MongoDB 实现的，在规模大到一定程度以后，性能会突然下降，SAE 自己解决不了这个问题，就给 MongoDB 开发组的各位大哥买机票请他们到北京理想国际大厦 17 层现场来帮忙，研究了几天，MongoDB 开发组的人说：你们换技术吧，MongoDB 解决不了你们这个规模的问题，然后 SAE 的 KV 就更换技术方案来实现了。

## DBA 晕倒砸烂花盆

也是在 SAE，笔者坐在厕所附近临过道的工位（上厕所很方便），某天早上刚上班，亲眼看到 SAE 的一名 MySQL DBA 从厕所里出来后，晕倒在笔者面前，砸烂了一个大花盆。数据库作为系统架构中最重要的那个单点的残酷，可见一斑。

## 列存储思想

与其将列存储认定为数据库的一种，笔者倒是觉得它更应该被称作一种思想：观察数据到底是被如何读取，并加以针对性地优化。

列存储有点像第一性原理在数据库领域的应用：不被现实世界所束缚，没有屈服于 B 树和它亲戚们的淫威，勇敢地向更底层看去，思考着在我们大量读取数据时，数据怎样组织才能读的更快。

在读取一行数据时，显然 B+ 树的效率无人能及，但是当我们需要读取 100 万行数据中的某一列时，B+ 树就需要把这 100 万行数据全部载入内存：每次将一页 16KB 载入内存，循环这一页内的 14 行数据，把这个特定的字段复制出来；重复执行这个操作 71429 次，才能得到我们想要的结果。这显然是 B+ 树非常不擅长的需求。

而列存储将数据基于行的排布翻转过来了：所有数据基于列，致密地排列在磁盘上，这样对某一列的读取就变成了磁盘顺序读，即便是机械磁盘，顺序读也非常快。

列存储数据库 Clickhouse 堪称俄罗斯人暴力美学的典范，和 Nginx 的气质很像

Clickhouse 推荐使用尽量多的 CPU 核心，对单核性能无要求，笔者使用 Xeon E5-V2 旧服务器测过，速度确实非常惊人，8000 万行的表，查询起来不仅比 MySQL 快，比 Hadoop 也快特别多。

## 9.8 面试题

---

**No.27**：查询请求增加时，如何做数据库的主从分离？

主从分离是一种常见的数据库复制技术，它可以将主数据库的更新操作同步到一个或多个从数据库，以保持数据的一致性并提升系统总容量。下面是实现主从分离的步骤：

1. 配置主数据库：在主数据库上开启二进制日志（binlog），并配置一个唯一标识符（server-id）。
2. 配置从数据库：在从数据库上配置一个唯一标识符（server-id），并指定主数据库的地址、用户名和密码。
3. 启动主数据库：确保主数据库正在运行，并记录当前的位置信息（binlog position）。
4. 启动从数据库：根据配置连接到主数据库，并开始复制主数据库的 binlog。
5. 初始化从数据库：如果是第一次启动从数据库，可能需要执行初始化操作，如通过全量备份从主数据库恢复数据。
6. 执行增量复制：从数据库会按照主数据库的 binlog 顺序执行增量复制，在主数据库进行修改操作时，从数据库将同步执行相同的操作以保持一致性。增量复制会自动进行，无需手动干预。

需要注意的是，主从分离只是实现了数据的复制，使用主从数据库集群需要对应用进行改造，要么就需要和查询请求的负载均衡配合才能发挥最大的价值。

**No.28**：数据库中间件的基本原理是什么？

数据库中间件通过在应用程序与数据库间添加一层代理，提供多种功能以提高数据库性能、可用性和可扩展性。其基本原理包括：

1. 连接池管理：维护连接池以更高效地管理数据库连接。
2. 负载均衡和路由：根据策略将请求分发到多个数据库实例，实现读写分离。
3. 缓存管理：将热门数据保存在内存中以提高读取速度，降低数据库压力。
4. 数据分片和分区：将大规模数据分片或分区存储于不同节点，提高扩展性和并发处理能力。
5. 高可用和容错：监控数据库状态，自动切换到备用节点以保证系统可用性。

## No.29 : 分布式事务如何设计？有哪些关键点？

分布式事务设计需要考虑的关键点：

1. ACID 属性：确保分布式事务的原子性、一致性、隔离性和持久性。
2. 事务协调与管理：需要一个协调者来协调和管理分布式事务中的各个参与者。
3. 分布式锁与并发控制：实现分布式锁以避免对同一资源的冲突操作，保证隔离性。
4. 两阶段提交（2PC）协议：一种经典的分布式事务协议，但需注意其单点故障问题。
5. 异步通信与消息队列：使用异步通信和消息队列可以提高系统的性能和可伸缩性。
6. 容错与恢复：需要考虑容错和异常处理机制，以确保数据的一致性和完整性。

## No.30 : Shared-Nothing、Shared-Memory 和 Shared-Disk 有什么异同？

Shared-Nothing、Shared-Memory 和 Shared-Disk 主要的区别在于数据共享的方式：

1. Shared-Nothing（无共享）：独立拥有计算资源和存储空间，节点间无数据共享，适用于分布式系统和并行计算，具有良好的可伸缩性和容错性。
2. Shared-Memory（共享内存）：所有节点共享物理内存空间，直接读写数据，适用于多线程或多进程应用，但可能面临数据一致性和争用问题。
3. Shared-Disk（共享磁盘）：所有节点连接共享的磁盘或存储设备，直接读写共享存储中的文件或数据，适用于频繁读写共享数据的应用，但可能面临磁盘访问冲突和性能瓶颈。

## No.31 : 列存储的原理是什么？适合哪些场景？

在列存储中，数据按照“列”的方式进行组织和存储，而不是传统 B+ 树等模型所采用的以“行”的方式存储数据。其原理主要包括以下几点：

1. 列式压缩：列存储针对每一列的数据进行压缩，因为同一列通常具有高度的数据相似性，所以这种压缩在大部分时候都非常高效。
2. 去耦合（Decoupling）：在列存储中，数据被划分为各个列存储单元，这样可以将查询只聚焦于所需的列，减少了不必要的 I/O 操作和数据传输，提高了查询效率。
3. 向量化处理（Vectorized Processing）：列存储可以对整个列进行向量化处理，即将多个数据元素一次性加载到寄存器进行操作，充分利用现代 CPU 的 SIMD 指令集，从而减少了循环的开销，加快了数据的处理速度。

列存储适合以下这些场景：

1. 分析型查询：面向大规模数据分析的查询通常是聚合、过滤和计算特定列的子集。列存储可以通过仅加载所需的列来加速此类查询，减少 I/O 操作和 CPU 开销。
2. 宽表数据：当一张表包含大量的列时，行存储会导致较大的存储冗余和 I/O 开销。而列存储可以只加载感兴趣的列，减少存储空间和 I/O 负载。
3. 压缩率高：列存储针对每列进行压缩，对于稀疏或重复数据，可以显著减小存储空间占用，并且在数据传输中节约带宽。
4. 数据压缩和索引建立：由于列存储数据的连续性，数据压缩比较高，能够提高数据的压缩率，减少存储成本。同时，列存储也适合于索引的建立，因为索引通常只需要访问特定的列。

**No.32** : 计算与存储分离的优势和劣势分别有哪些？

计算与存储分离在云计算时代大放异彩，本质是因为云计算厂商首次实现了高性能又稳定的云存储，这种技术产品离不开硬件和 Kernel 的进步。

计算与存储分离的优势在于：

1. 弹性扩展：单独扩展计算和存储资源，满足不同场景需求，更具弹性。
2. 节约成本：单独优化、采购计算和存储资源，降低成本。
3. 简化管理：专注于数据持久性和可靠性的存储资源和专注于数据处理和计算任务的计算资源，降低系统复杂性。
4. 多租户支持：为不同租户提供独立且安全的存储空间，同时共享计算资源，更好地实现资源隔离和多租户支持。

劣势在于：

1. 网络开销：计算和存储分离导致网络开销增加，可能影响系统延迟和吞吐量。
2. 数据一致性：数据更新时需要确保计算节点和存储节点之间的数据保持同步，实现正确的并发控制机制。
3. 系统复杂性增加：计算和存储分离可能增加系统的复杂性，特别是在跨多个计算节点和存储节点进行数据处理和传输时，需要额外的管理和调度机制来协调计算和存储资源。

## 第 10 章 国产分布式数据库双雄 TiDB 和 OceanBase

TiDB 和 OceanBase 是目前中国 NewSQL 数据库的绝代双骄，关于他们的争论一直不绝于耳。

TiDB 是承袭 Spanner 思想的 NewSQL，对 MySQL 的兼容性一般，基于 [key+版本号](#) 的事务控制也比较弱，据说性能比较好，特别是写入性能。

OceanBase 是基于 Shared-Nothing 思想原生开发的分区存储数据库，其每个节点都支持完整的 SQL 查询，相互之间无需频繁通信。OceanBase 还支持多租户隔离，这明显就是为了云服务准备的(无论是公有云还是私有云)，和中小企业无关。另外，OceanBase 对于 MySQL 的兼容性也几乎是 NewSQL 里面最高的，毕竟它需要支持支付宝的真实业务，兼容性是硬性要求，积累了那么多年的老旧业务代码是很难被完全重写的。

下面我们详细对比一下两者的设计思路差异。

### 10.1 TiDB 的设计思路

笔者抽象出的 TiDB 架构图如图 10-1 所示。

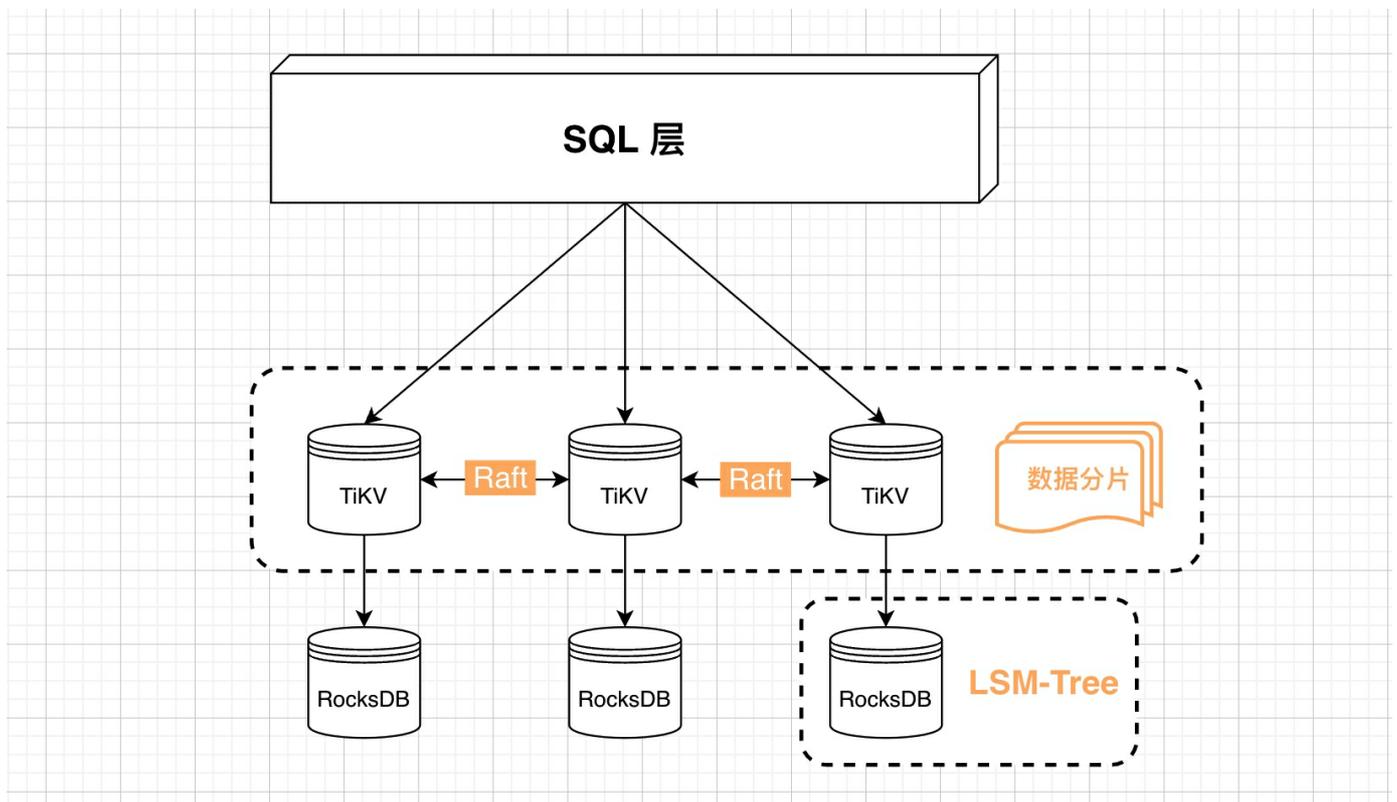


图 10-1 TiDB 系统架构

上图中的“SQL 层”就是解析 SQL 语句并将其转化为 KV 命令的一层，是无状态的，下面的存储层才是核心，它叫 TiKV。

### TiKV 如何存储数据

TiKV 官方原理图如下：



### TiKV Architecture

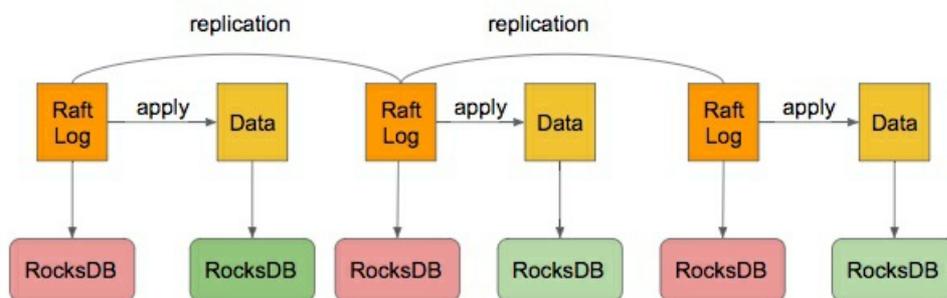


图 10-2 TiKV 官方原理图

TiKV 是 TiDB 的核心组件，一致性和事务隔离都是基于它的能力得以实现的，其官方原理图如图 10-2 所示。每个 TiKV 拥有两个独立的 RocksDB 实例，一个用于存储 Raft Log，另一个用于存储用户数据和多版本隔离数据（基于 [key+版本号](#) 实现），从这里可以看出，TiDB 的存储存在大量冗余，所以 TiDB 的测试性能才会如此的高，符合空间换时间的基本原理。

和 TiKV 并列的还有一个 TiFlash 列存储引擎，是为了 OLAP 在线数据分析用的，我们在此不做详细讨论。

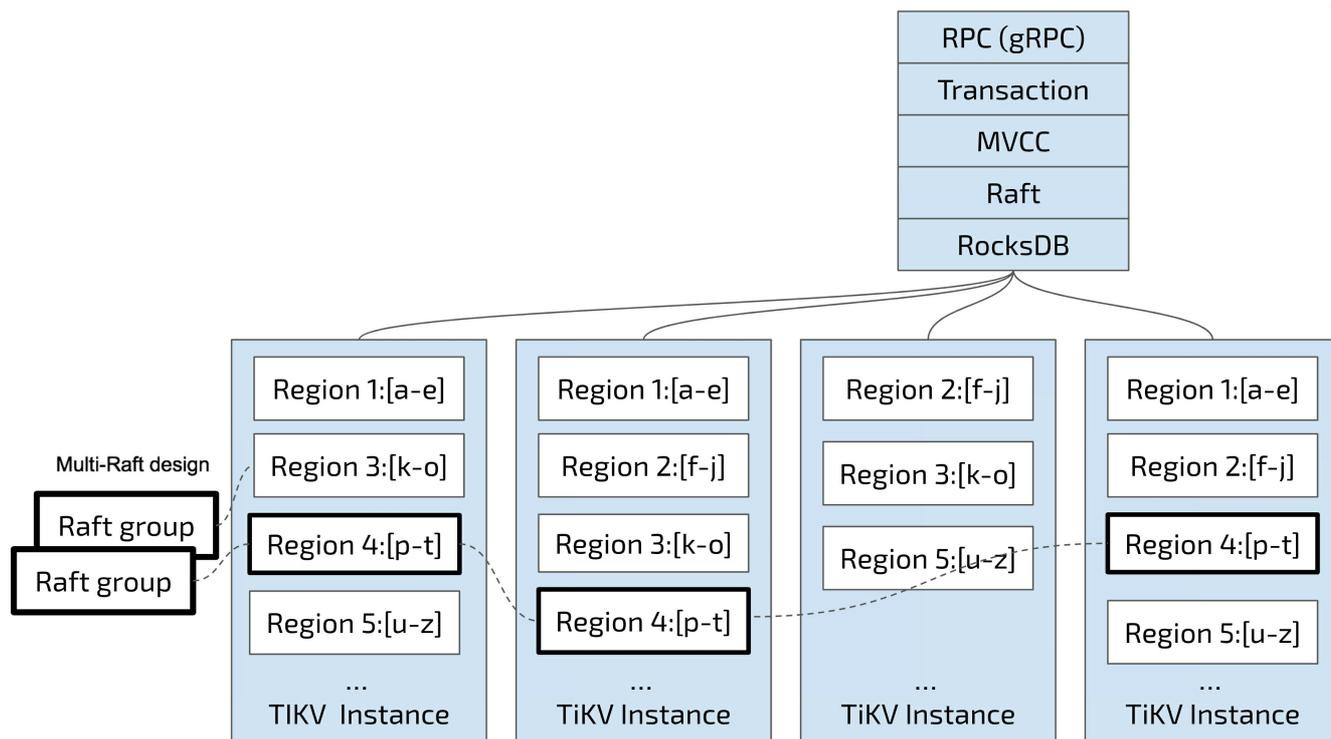


图 10-3 TiKV 数据分片

除此之外，TiKV 还发明了一层虚拟的“分片” (Region)，将数据切分成 96MB~144MB 的多个分片，并且用 Raft 算法将其分散到多个节点上存储。注意，在 TiKV 内部存储用户数据的那个 RocksDB 内部，多个分片是致密存储的，分片之间并没有逻辑关系。TiKV 数据分片的架构如图 10-3 所示。

由于 TiDB 的设计风格比较狂野，所以不兼容的部分比较多，如图 10-4 所示。

## 不支持的功能特性

- 存储过程与函数
- 触发器
- 事件
- 自定义函数
- 外键约束 #18209
- 全文语法与索引 #1793
- 空间类型的函数（即 GIS / GEOMETRY）、数据类型和索引 #6347
- 非 `ascii`、`latin1`、`binary`、`utf8`、`utf8mb4`、`gbk` 的字符集
- SYS schema
- MySQL 追踪优化器
- XML 函数
- X-Protocol #1109
- 列级权限 #9766
- XA 语法（TiDB 内部使用两阶段提交，但并没有通过 SQL 接口公开）
- `CREATE TABLE tblName AS SELECT stmt` 语法 #4754
- `CHECK TABLE` 语法 #4673
- `CHECKSUM TABLE` 语法 #1895
- `REPAIR TABLE` 语法
- `OPTIMIZE TABLE` 语法
- `HANDLER` 语句
- `CREATE TABLESPACE` 语句
- "Session Tracker: 将 GTID 上下文信息添加到 OK 包中"

图 10-4 TiDB 和 MySQL 不兼容的部分

## TiDB 对 CAP 和不可能三角的抉择

TiDB 放弃了新不可能三角中的事务隔离，和 Spanner 一样放弃了 CAP 理论中的“完全可用性”：一旦出现脑裂，就会出现意外的返回结果（如超时），因为 TiDB 选择了保证一致性：如果无法达到数据强一致，就要停止服务。

一句话概括 TiDB：①搭建在 KV 数据库集群之上的，②兼容部分 MySQL 语法的，③有一些事务处理能力的高性能数据库。

## 10.2 OceanBase 设计思路

我们以最新的 OceanBase 4.0 版本的架构为目标进行讨论。

TiDB 底层数据叫分片，那 OceanBase 为什么叫分区呢？因为分片的意思只是数据被分开了（KV 数据库不同的 Key 之间本来也没有关系），但分区表示的是分区内部的数据之间是有联系的：OceanBase 的每个节点本身，依然是一个关系型数据库，拥有自己的 SQL 引擎、存储引擎和事务引擎。

### 简单的分区

OceanBase 在建表时就需要设定数据分区规则，之后每一行数据都属于且仅属于某个分区。在数据插入和查询的时候，需要找到这一行数据所在的区，进行针对性地路由。这和第一代分布式——中间件的思想一致

。这么做相当于简单地并行执行多条 SQL，以数据切分和数据聚合为代价，让数据库并行起来。而这个数据切分和数据聚合的代价，可以很小，也可以很大，需要 OceanBase 进行精细的性能优化，我们下面还会说到。

分区之间，通过 Multi-Paxos 协议来同步数据：每一个逻辑分区都会有多个副本分布在多台机器上，只有其中一个副本会成为 leader，并接受写请求。这里的架构和 PolarDB 一样了，此时，客户端的一致性读需要网关（OBProxy）来判断，主从之间的同步是有可感知的延迟的。

## 节点存储架构

如图所示是 TiDB 的官方存储架构图。

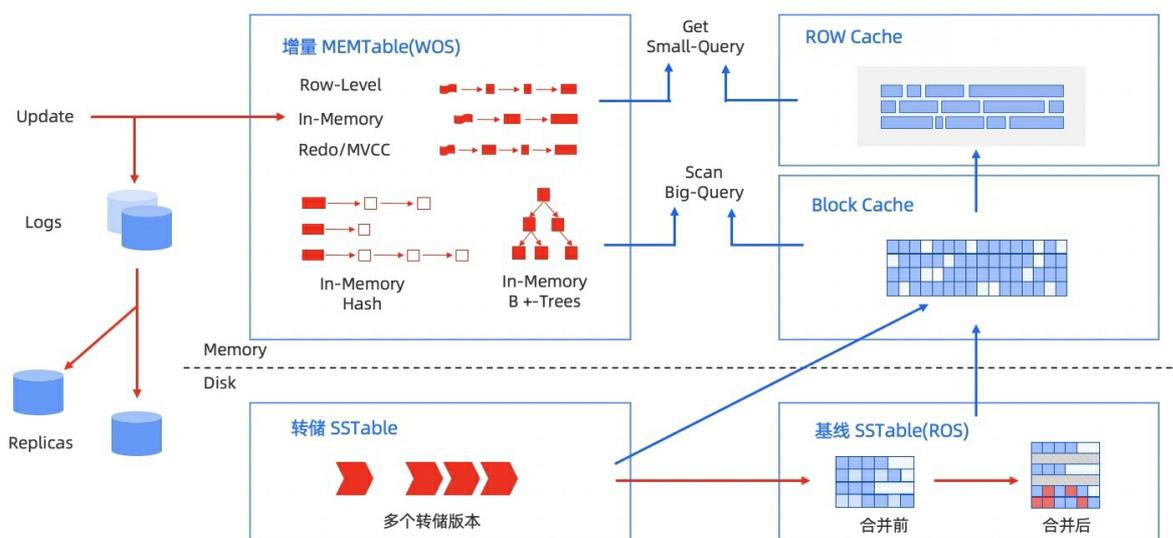


图 10-5 官方存储架构图

OceanBase 数据库的存储引擎基于 LSM Tree 架构，将数据分为静态基线数据（放在 SSTable 中）和动态增量数据（放在 MemTable 中）两部分，其中 SSTable 是只读的，一旦生成就不再被修改，存储于磁盘；MemTable 支持读写，存储于内存。数据库 DML 操作插入、更新、删除等首先写入 MemTable，等到 MemTable 达到一定大小时转储到磁盘成为 SSTable。在进行查询时，需要分别对 SSTable 和 MemTable 进行查询，并将查询结果进行归并，返回给 SQL 层归并后的查询结果。同时在内存实现了 Block Cache 和 Row cache，来避免对基线数据的随机读。

当内存的增量数据达到一定规模的时候，会触发增量数据和基线数据的合并，把增量数据落盘。同时每天晚上的空闲时刻，系统也会自动每日合并。

OceanBase 数据库本质上是一个基线加增量的存储引擎，在保持 LSM-Tree 架构优点的同时也借鉴了部分传统关系数据库存储引擎的优点。

以上是官方描述，笔者本来想简化一下，读了一遍觉得还是直接放上原文吧，原文描述的就非常得清晰精炼了：OceanBase 用内存 B+ 树和磁盘 LSM-Tree 共同构成了数据读写体系，和上一章中的 InnoDB 是多么像啊！只是 OceanBase 做的更细：跟 TiDB 相比，就像是在 TiKV 上面加了一层 Buffer Pool 一样。



当内存的增量数据达到一定规模的时候，会触发增量数据和基线数据的合并，把增量数据落盘。同时每天晚上的空闲时刻，系统也会自动每日合并。

把一整天的新增和修改的数据全部放到内存里，相当于直接变身成了内存数据库（还会用 B 树和 Hashtable 存两份），确实是一种终极的性能优化手段，OceanBase 真有自己的。

## 提升并行查询和数据聚合性能

传统的中间件 Sharding 技术中，也会做一些并行查询，但是它们做的都是纯客户端的查询：proxy 作为标准客户端，分别从多台机器拿到数据之后，用自己的内存进行数据处理，这个逻辑非常清晰，但有两个问题：1. 只能做简单的并行和聚合，复杂的做不了 2. 后端数据库相互之间无通信，没有很好地利用资源，总响应时间很长。

OceanBase 让一切尽可能地并行起来了：在某台机器上的 proxy (OBServer) 接到请求以后，它会担任协调者的角色，将任务并行地分发到多个其他的 OBServer 上执行；同时，将多个子计划划分到各个节点上以后，会在各节点之间建立 M\*N 个网络通信 channel，并行地传输信息；此外，OceanBase 还对传统数据库的执行计划优化做了详细的拆分，对特定的关键词一个一个地做分布式优化，才最终达成了地表最强的成就。

由于本身就是为了兼容 MySQL 而设计的一种新技术实现，所以它拥有非常优秀的兼容性，和 MySQL 不兼容的情况很少，具体如图 10-7 所示。

## 暂不支持的功能

- 暂不支持空间数据类型。
- 不支持 `SELECT ... FOR SHARE ...` 语法。
- 不支持空间分析函数和性能模式函数。
- 对于备份恢复功能，不支持集群级别的备份恢复，不支持冷备份，不支持租户内部部分数据库和表级的备份恢复，不支持备份数据的有效性验证。
- 对于优化器，查看执行计划的命令不支持使用 `SHOW WARNINGS` 显示额外的信息。

图 10-7 OceanBase 和 MySQL 不兼容的部分

## OceanBase 对 CAP 和不可能三角的抉择

由于数据是分区的，所以当脑裂时，两个大脑的数据肯定已经不完整了，相当于两万行的表只剩一万行数据可以进行查询和更新，此时，如果 OceanBase 梗着脖子非要追求数据强一致，也是可以让所有的 OBProxy 拒绝服务的，但是 OceanBase 选择了继续苟着：互联网公司嘛，能实现最终一致性就行了，要啥自行车。

OceanBase 放弃了 CAP 和新不可能三角中的一致性，只能做到最终一致性：为了事务隔离和性能，哥忍了。

其实，不追求强一致和下一章中的终极高并发架构在思想上是一致的，这可能就是经历过大规模生产应用的

数据库，被现实世界毒打后的痕迹吧。

一句话概括 OceanBase: ①世界第一性能的, ②高度兼容 MySQL 的, ③经历过生产系统考验的高性能分布式关系型数据库。

## 10.3 分布式数据库，应该怎么选？

其实，分布式数据库根本就轮不到你来选：应用准备好了吗？有足够的研发资源吗？性能问题已经大到压倒其他需求了吗？

如果你有一个正在成长的业务，影响最小、成本最低的方案就是选择 Aurora/PolarDB 这种高兼容性数据库，等到这类云数据库的主节点达到性能上限了，再对应用做逐步改造，滚动式地替换每个部分的数据库依赖。

如果压力大到必须换分布式数据库技术方案了，再看看你能获得什么样的分布式数据库呢？无非是在哪个云平台就用哪家呗。

没得选，在哪个云平台就只能用哪家。

还记得我们的目标吗？五百万数据库 QPS

在中国，我们现在有下面两种方案可以选择：

1. OceanBase 已经蝉联 TPC-C 数年的全球冠军了，每分钟可以处理 7.07 亿个订单，每秒订单数都已经过千万了，更不要说 QPS 500 万了，所以，如果你用 OceanBase，你的百万 QPS 的高并发系统已经搭建完成了！
2. 如果你用阿里云，那 1 主 4 从，88 vCore 710 GB \* 5 个节点的 PolarDB 集群 [可以跑到大约 200 万 QPS](#)。那离 500 万还有不小的距离呢，不要着急，我们最后一章解决这个问题。

## 10.4 面试题

**No.33**：分布式 CAP 定理讲的是什么？如何推导？

分布式 CAP 定理是关于分布式系统设计中的三个重要属性的不可兼得性的定理。这三个属性分别是：一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）。

1. 一致性指的是在分布式系统中的所有节点上读取到的数据都是一致的。换句话说，当一个节点写入了新的数据后，其他节点必须能够立即读取到最新的数据。
2. 可用性指的是分布式系统在面对部分节点故障或网络分区时仍能保持可用状态，即系统仍然能够接受和处理客户端的请求。
3. 分区容错性是指分布式系统能够在面对网络分区（节点之间无法通信）的情况下依然保持正常运行。

CAP 定理指出，在一个分布式系统中，一致性、可用性和分区容错性这三个属性不可能同时满足。也就是说，在面对网络分区时，我们只能在一致性和可用性之间进行权衡选择。

根据分布式系统设计中的异步通信模型和 FLP 结果（FLP 结果是一个涉及异步系统的定理，表明在异步通信模型中，不存在一致性算法，可以保证在任意节点故障的情况下都能达到一致性）可以推导出 CAP 定理。

#### No.34 : 国产分布式数据库 TiDB 的技术原理是什么？

TiDB 是一种基于高性能键值（KV）磁盘数据库开发的 SQL 兼容分布式数据库，技术原理如下：

1. 分布式架构：TiDB 采用分布式架构，将数据存储和计算分散在多个节点上，实现数据的水平分割和负载均衡。它由三个核心组件组成：TiDB Server、TiKV 和 PD（Placement Driver）。
2. 分布式事务：TiDB 支持分布式事务处理，采用 Google Spanner 的 Percolator 扩展实现。它通过基于多版本并发控制（MVCC）来提供事务的隔离性，使用两阶段提交协议来保证事务的原子性和持久性。
3. 分布式一致性：TiDB 使用 Raft 一致性算法进行数据复制和强一致性保证。Raft 算法确保了数据在集群中各个节点之间的一致性，保证了数据的可靠性和可用性。
4. 分布式存储引擎：TiDB 的底层存储引擎是 TiKV（分布式 Key-Value 存储引擎），它是一个分布式、高可用、自动水平扩展的存储系统。TiKV 使用 Raft 算法进行数据复制，支持强一致性模型和事务操作。
5. 分布式查询优化：TiDB 使用分布式查询优化器来解析和优化 SQL 查询语句，并将查询计划分发给各个节点执行。通过并行处理和智能的数据分片策略，实现了高效的分布式查询操作。
6. 自动水平扩展：TiDB 支持自动水平扩展，可以根据负载情况和数据大小自动添加或删除节点，以提供更好的性能和可伸缩性。

#### No.35 : 国产分布式数据库 OceanBase 的技术原理是什么？

OceanBase 是一种基于数据分区的分布式数据库，和传统的分库分表/中间件技术比较类似，可以看做一种比较完善的高性能分库分表解决方案，技术原理如下：

1. 分布式架构：采用分布式架构，将数据划分为多个区（Partition），每个区可以存储和处理部分数据。不同的区可以分布在不同的服务器上，实现数据的分布存储和并行处理。
2. 高可靠性：通过数据冗余和容错机制来提供高可靠性。它采用了 Multi-Paxos 协议来实现数据的一致性和副本同步，当某个节点发生故障时，可以从其他副本节点中恢复数据。
3. 分布式事务：支持分布式事务，它使用两阶段提交（Two-Phase Commit）协议来实现分布式事务的原子性和一致性。同时，OceanBase 还支持快照隔离级别和多版本并发控制（MVCC）等技术，提供高效的事务处理能力。
4. 水平扩展：可以根据需求进行水平扩展，通过增加节点来实现数据库的容量和性能的扩展。它提供了自动负载均衡和动态迁移数据的功能，保证数据在集群中的均衡存储和访问。
5. 多租户支持：支持多租户架构，可以将不同的租户的数据进行隔离和管理。每个租户可以拥有自己独立的数据库和资源配额，保证不同租户之间的数据安全和性能隔离。

#### No.36 : TiDB 和 OceanBase 分别适合哪些场景？

TiDB 和 OceanBase 都是分布式关系型数据库，但是由于其底层采用了截然不同的两种架构，所以它们适用于不同的场景：

TiDB 适合以下场景：

1. 系统规模高速扩展：TiDB 采用水平扩展的方式，可以通过增加节点来提升系统的性能和容量。

2. 在线分析处理 OLAP：分布式 KV 数据库适合大规模数据读写，尤其是写入性能比读取性能更高。
3. 对一致性有强需求的场景：这是 TiDB 和关系型分布式数据库最大的不同，它可以保证全局强一致，为此甚至牺牲了可用性。

OceanBase 适合以下场景：

1. 分布式事务：OceanBase 支持分布式事务，可以保证多节点之间的数据一致性和事务的 ACID 特性。
2. 超高并发读写：作为世界性能第一的分布式数据库，这一点无需多言。

## 第 11 章 秒杀系统的两大利器——缓存与队列

了解清楚了分布式数据库的底层原理和常用架构之后，本章我们将讨论电商秒杀系统的两大利器，也是秒杀系统高并发能力最大的两个来源：缓存和队列。

### 11.1 概述

电商秒杀业务的特点是短时间内用户量和用户点击量都暴增，最高可能达到百倍之多。虽然电商秒杀业务的流量会在短时间内暴增，但是流量越大，这些请求的“局部性”就越强——被抢购的商品会越集中，这个商品就越适合用缓存技术来承载，缓存的作用就会越大。

#### 电商秒杀业务的特点

电商秒杀业务最大的特点就是请求数会突然暴增，因为用户会停在界面上等待，一旦活动开始，就开始疯狂下单付款，晚了就售罄了。正因为秒杀的突然性，让它在技术上有了如下的特点：

1. 客户端数量会突然暴增，对应的就是 TCP 连接数和 API 请求数的突然暴增
2. 流量集中在少部分商品上
3. 下单压力会变得特别大，对库存管理和数据库写入能力造成了严重威胁

#### 缓存技术的底层原理

缓存技术的底层原理依然是老套的“空间换时间”和“内存比磁盘快”。

宏观上，我们多使用了一些内存空间，提升了系统的响应速度，这就是空间换时间。此外，由于内存必然比磁盘的读写速度快、读写延迟低，所以我们只要把热门数据和计算结果存储在内存中，就可以提高数据的读取速度，进而提升系统总体性能。

### 11.2 缓存设计实战：静山平台

在秒杀场景中，有一些接口的请求频率会明显高于其他接口，我们应该针对这些高频接口做定向缓存优化。

#### 商品详情接口

商品详情接口是整个系统中压力最大的接口，用户在购买之前会访问很多次商品详情。商品详情页面中，包

含介绍视频、头部轮播图、详情页面、多个价格、该商品参加的活动、可以领取的优惠券等信息。

我们可以根据商品被访问的热度，将商品的数据缓存下来。不同的字段对缓存时间的敏感程度是不一样的：商品的基础信息变化的频率较低，我们可以缓存比较长的时间，例如一分钟。而有些信息像价格对时效性的要求很高，需要非常快地更新缓存。

## 库存查询接口

商品库存数量是一个对时间高度敏感的字段，而且这个值和下单接口高度相关，可以说库存查询和下单一起组成了电商秒杀系统中最核心的那个“单点”。

首先，库存及时归零可以在前端就拦截掉海量的对下单接口的无效调用：没有库存的商品无论如何是无法下单成功的，这些发送到后端的压力都是没有收益的浪费资源。

其次，哪怕只出现了一单超卖，平台和商户都会为此承担额外的损失。在最严格的系统架构设计中，这是不可接受的，属于系统设计上的失败。

## 购物车价格计算

和商品详情比，这并不是一个高并发接口，但是随着近两年电商平台的内卷，到手价预估，自动领券并结算，以及自动计算最优惠的折扣方案等需求的提出，给购物车价格计算工作上强度，这个接口的响应时长和占据的 CPU 资源急剧上升，从架构师到 RD，从运营到测试，天下苦购物车久矣...

## 下单接口

下单接口是截止到目前唯一的一个高并发“写入”的接口，可能也是唯一一个需要实时体现出影响的写入型接口。下单接口需要处理好以下几个重要问题。

1. 不要宕机：服务可用是一切的基础，在所有需求之前，需要先保证下单业务的持续运行。
2. 避免超卖：这是秒杀业务最重要的需求，超卖一单就是一单的成本，涉及到钱，要小心处理。
3. 重复点击：在秒杀的时候，用户很容易出现重复点击的情况，这就需要前端后端配合，避免重复下单的情况发生。
4. 错误回滚：在高并发情形下，系统可能会出现各种异常，甚至接口数据前后不一致的情况，这就需要在每一步做好错误判断和状态回滚，避免系统陷入错误沼泽，无法自拔。
5. 避免少卖：如果卖出去的商品数量低于秒杀设置的值也是一种损失。
6. 库存计算的正确性：大部分时候，库存的计算是避免超卖、发生错误、超买等问题的核心步骤，只要库存的计算不出错，其他大部分问题都可以避免。
7. 阻止刷单：很多人会利用机器人来刷单，刷单会对系统造成非常大的负担，需要尽可能地识别出来，屏蔽掉。
8. 防止恶意攻击：秒杀是系统最脆弱的时候，最适合执行恶意攻击，容易出现恶意占用接口资源的攻击行为。我们可以使用严格的数据校验、验证码、请求频率限制等方法来降低恶意攻击行为对系统造成的影响。

## 支付接口

每个订单都需要至少一次支付，在秒杀开始以后，支付接口也会面临非常高的请求压力，需要提前和支付服

务方做好流量预估和技术方案，必要的时候引入多个支付通道进行在线支付，并且要对支付失败做好回滚策略，避免整个系统的“雪崩”。

## 订单状态查询接口

用户在秒杀结束后会频繁查询订单的支付状态和发货状态，此时订单列表和订单详情的压力就变得特别大。这一点我们可以和天猫学习：双 11 刚开始的几分钟，用户无法实时查询到订单的支付状态，在购买系统和支付系统度过了最初的超高压力后，再异步慢慢补上支付状态。此外，在双 11 当天，天猫限制所有用户都只能看到三个月之内的订单。一方面，这可以避免“冷数据”的激活对系统造成过大的压力，另一方面也可以直接把三个月内的订单全部缓存下来，无论是用数据库做磁盘缓存还是直接做内存缓存，都可以有效提升整个系统的响应速度。

## 排行榜接口

排行榜也是一个容易被忽视的高并发场景。由于需要对最高销量的商品进行数据统计，这个行为会给系统带来巨大的压力。此外，许多用户，尤其是卖家以及他们开发的机器人，会频繁地访问排行榜接口。因此我们必须对排行榜接口进行全缓存设计。实时统计将给数据库带来极大的压力，容易导致系统宕机。

## 11.3 缓存的读写策略

在我们设计缓存的技术方案时，最重要的无疑就是缓存的读写策略，而缓存读写策略的核心是两个问题和一个设计。在展开读写策略之前，我们先复习一下大家最常用的缓存使用方法，笔者称之为“原始缓存架构”。

### 原始缓存架构

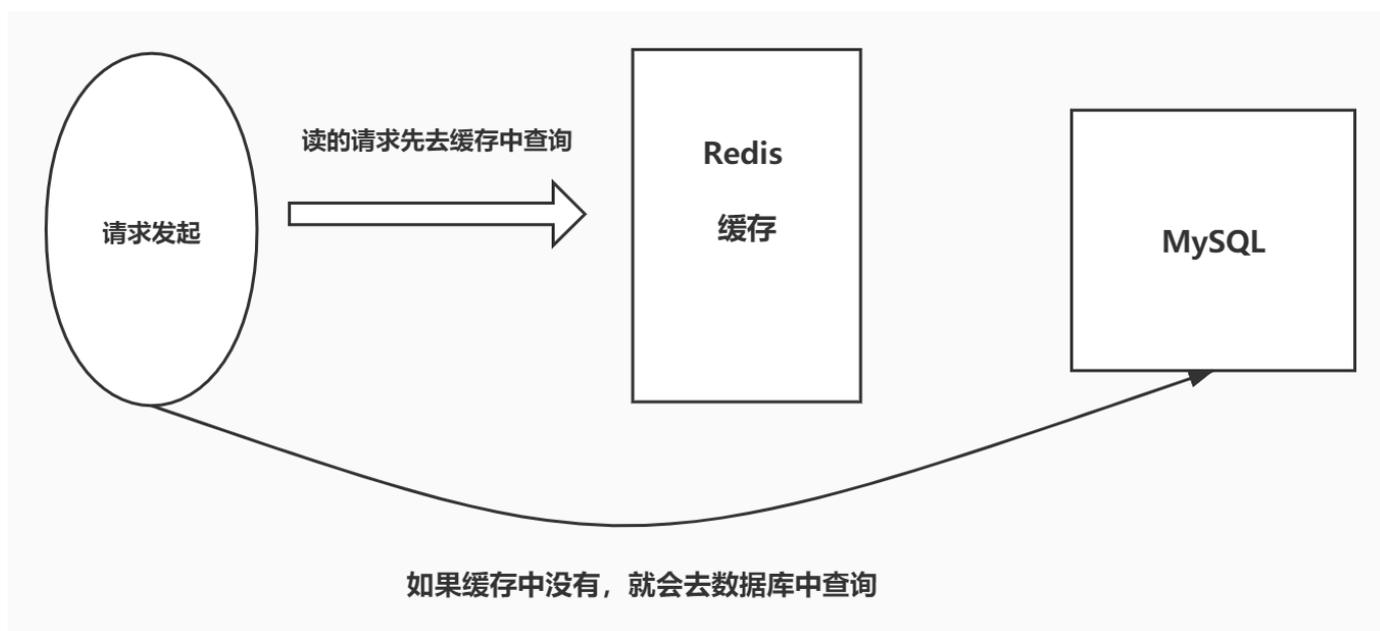


图 11-1 原始缓存架构

如图 11-1 所示，使用 Redis 实现的原始缓存流程如下：

1. 尝试读缓存
2. 若读到，则返回
3. 若没有读到，则去数据库里查询到结果，将结果存入缓存，并返回数据

看完了原始缓存架构，接下来我们展开讨论一下缓存读写策略的两个问题和一个设计。

## 缓存击穿问题

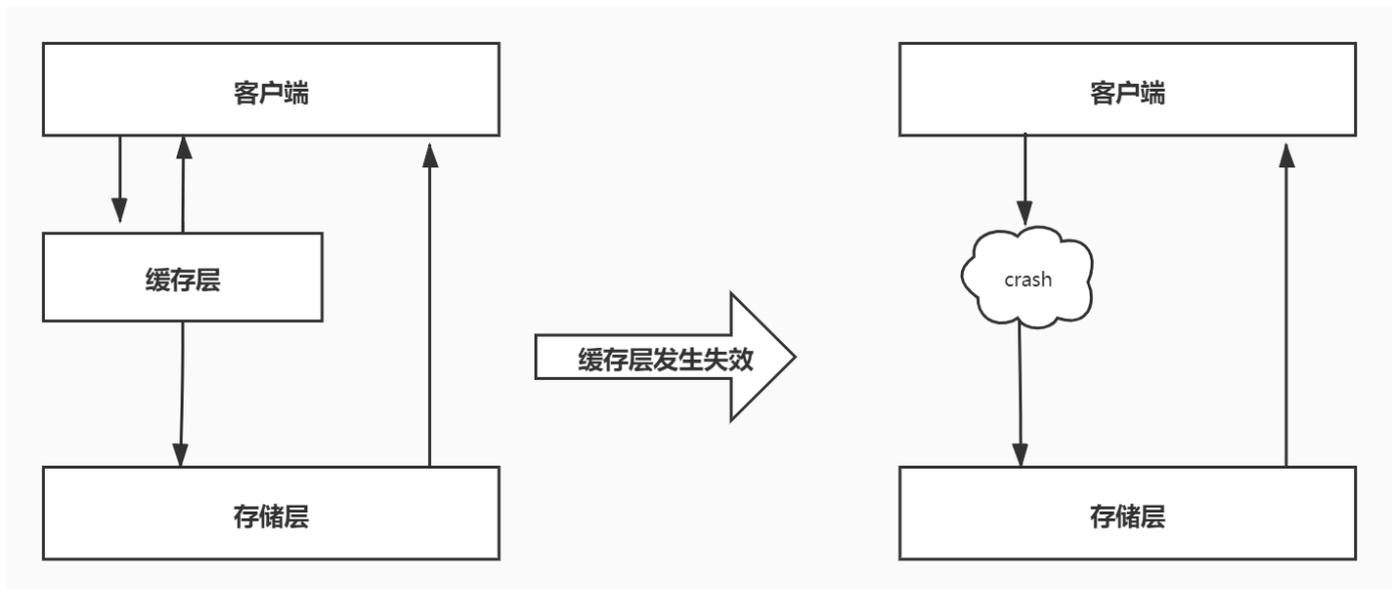


图 11-2 缓存击穿示意图

在秒杀活动期间，对热门商品的读取是非常高频的，这些高频数据称为“热点数据”。而如果在活动的高峰期，缓存失效了的话，瞬间会有多个请求发现读取不到缓存内容，就会选择直接从数据库中读取，进而数据库被瞬间高压击垮。这个过程就称为“缓存击穿”，如图 11-2 所示。

目前我们主要采用两种方法来应对缓存击穿：

1. 互斥锁/分布式锁：保证同一时间只有一个业务线程更新缓存，如果某个请求未能获取互斥锁，则等待或者直接返回空值/默认值。
2. 缓存不过期：可以使用后台进程定期异步更新缓存，也可以定期检查缓存过期时间，在过期前对数据进行刷新，并设置新的过期时间。

## 缓存不过期引发的逻辑悖论

缓存不过期并不意味着永远不做数据淘汰，我们依然需要动态地管理缓存里的数据，因为内存空间不是无限大的。这样就引发了一个逻辑悖论：

如果缓存不过期，那在上游调用者看来，缓存就一定要能够在任何时候查询到任何数据，哪怕部分数据慢一点也不要紧，但是不能查询不到。但是，一旦我们对某些数据进行了淘汰，那它们在上游调用者看来就变得不存在了，这可以视为缓存系统的失效。如果我们再引入一个不存在时候的数据库直连机制，那这个架构不又成了原始缓存架构了。

因为这个悖论的存在，所以缓存不过期必须和其它能够自动过期的缓存配合使用。

## 缓存雪崩问题

缓存雪崩是指在某个时间点，缓存中大量的数据同时失效，导致请求被集中大量发送给了后端的数据库，导致数据库因为瞬间的高负载而崩溃的问题。相比于前面的缓存击穿问题，缓存雪崩的问题更容易发生：因为各种流量都是在秒杀活动开始时同时开始进入高并发状态的，如果缓存超时时间一致，那就很容易出现雪崩问题。

应对这个常见的问题，我们一般按照如下顺序尝试解决。

### ① 错开过期时间

根据资源类型甚至是资源本身的差异将缓存失效时间错开，例如根据商品的主键 ID 设置它们的过期时间。

### ② 缓存不过期

和缓存击穿问题一样，我们也可以使用后台进程定期异步更新缓存来彻底解决这个问题。

### ③ 双 Key 设计

我们对每一条缓存都用两个 key 存两份，一个是主 key，会设置过期时间，一个是备 key，不设置过期时间，它们的 key 不一样，value 值一样。当上游调用者访问不到主 key 时，就直接返回备 key 中的数据，之后只需要等待后台进程生成新的缓存数据即可，在调用者看来，数据没有丢失，甚至连性能都没有下降。

这本质上是空间换时间，用多一倍的内存空间解决缓存失效的问题，虽然确实解决了问题，但是浪费了宝贵的内存空间，只有在小规模的系统或者极其土豪的公司才应该采用。

## 缓存服务器宕机引发的缓存雪崩

笔者不赞成使用所谓的高可用集群来规避缓存服务器宕机，因为缓存服务在绝大多数情况下并不会因为自己的软硬件故障而宕机，而是因为内存爆了，或者被超大读写压力给打挂了。所以，笔者建议，如果缓存服务器宕机，应该直接返回错误，然后让运维来修。

笔者为什么连挣扎都不想挣扎一下？因为笔者看到此时缓存服务器的单点性：当我们的数据读取流程依赖了缓存服务器时，那他对系统的重要性就和背后的数据库一样了，数据库宕机了你还要对外提供服务吗？完全没有必要，出现了数据丢失反而更麻烦。

## 缓存后台更新设计

前面我们说过了缓存不过期的设计方法，也讨论了缓存不过期引发的逻辑悖论，那么，有没有一种比较好的方法规避问题呢？有，虽然不完美，但是可用——使用消息队列对缓存的更新流程进行解耦。

前面的缓存更新中，我们的查询过程和更新过程是必须在同一个线程内先后进行的，这才是最大的束缚。一旦我们能使用消息队列对缓存的工作进行解耦以后，缓存的更新过程一下子就变得自治了。

1. 每次读取缓存，都判断一下是否即将失效
2. 若即将失效，则通过消息队列向后台的缓存更新进程发送一条“某缓存 Key 需要更新”的消息

3. 在高并发下，短时间内后台进程可能会收到多条消息，但这不重要，因为在真的从数据库读取信息更新缓存之前，执行只需要判断一下过期时间就行了，由于队列的排队执行特性，更新数据操作肯定只会被执行一次

这个方法的巧妙之处就是把阻塞的需要立马处理的“数据已经过期”了的噩耗转化成了一条又一条的消息，而消息的处理是顺序的，只要我们的一个队列处理器能够处理得过来某一类的资源就行，哪怕执行不过来也没关系，理论上缓存 key 相互之前没有关系——极限情况下你可以为每个 key 单独设立一个队列处理器来更新它的缓存数据。

## 缓存淘汰策略

在缓存服务器内存不足时，我们需要选择一种方法来淘汰旧数据。常见的替换策略有最近最少使用 (LRU)、先进先出 (FIFO)、最不经常使用 (LFU)、最长时间未使用 (LRU-TTL)、最少经常使用 (MFU)、后进先出 (LIFO)、基于成本的淘汰 (CBA)、随机淘汰等，下面我们简单描述一下它们分别的适用场景。

1. 最少使用 (LRU): LRU 策略认为最近被访问过的缓存数据是最有可能再次被访问的，因此优先保留最近被使用过的数据。适用于访问模式较为集中的场景，商品详情和商品库存适合使用此策略。
2. 先进先出 (FIFO): FIFO 策略按照缓存数据进入缓存的顺序进行淘汰，最早进入缓存的数据最先被淘汰。适用于缓存数据没有特定的访问模式，并且对访问顺序没有特别要求的场景，用户的订单列表等完全个性化的数据适合使用此策略。
3. 最不经常使用 (LFU): LFU 策略认为被访问次数最少的缓存数据在未来的一段时间内仍然可能不被频繁访问，因此优先淘汰访问次数最少的数据。适用于部分数据非常热门，大部分数据较冷门的场景，商品详情和商品库存也非常适合使用此策略。
4. 最长时间未使用 (LRU-TTL): 该策略基于最近最少使用 (LRU)，但还考虑了缓存数据的过期时间。当缓存数据既长时间未被访问又已经过期时，优先淘汰该数据。
5. 最少经常使用 (MFU): 该策略与最不经常使用 (LFU) 相反，优先保留访问次数最多的缓存数据。认为访问次数多的数据在未来仍然可能被频繁访问，适用于访问次数具有较大波动的场景。
6. 后进先出 (LIFO): 该策略与先进先出 (FIFO) 相反，最后进入缓存的数据最先被淘汰。这个策略有一个十分神奇的应用：在秒杀系统从流量高峰下落的过程中，可以应用此策略来缓存商品信息，可以最大限度地保证热门商品保留在缓存中。
7. 基于成本的淘汰 (CBA): 该策略根据缓存数据的成本信息进行淘汰。成本可以包括缓存数据的存储成本、计算成本、更新成本等，换句话说就是先淘汰数据量更大的 key。
8. 随机淘汰 (Random): 该策略在缓存空间不足时，随机选择一个缓存数据进行淘汰，这比较适用于数据离散的场景，电商秒杀业务应用较少。

## 缓存穿透问题

缓存穿透指的是调用者希望查询一个缓存和数据库中都不存在的数据。缓存穿透的原因有两种：

1. 业务误操作，缓存中的数据 and 数据库中的数据都被误删除了，所以导致缓存和数据库中都没有数据。
2. 黑客恶意攻击，故意大量访问不存在的数据。

一般情况下，我们需要对非法请求进行快速的判断，并返回空值。而这个判断可能会出问题——因为这个判断需要读取数据库，这就可能出现一种场景：短时间内海量的 `select exist` 语句被发往数据库，把数据库打

崩。

为了应对这个风险，我们可以在业务代码中做限流，或者使用“布隆过滤器”。

### 布隆过滤器的工作原理

布隆过滤器 (Bloom Filter) 是一种空间效率极高的概率型数据结构，用于判断一个元素是否在一个集合中。它是由一个很长的二进制向量和一组哈希函数组成。当一个元素插入到布隆过滤器中时，会通过多个哈希函数进行映射，将元素的位置标记为 1；当查询一个元素是否存在时，同样会通过多个哈希函数进行映射，如果所有位置都是 1，那么认为该元素可能存在于集合中，但实际上可能不存在。

布隆过滤器的工作原理如下：

1. 初始化：创建一个长度为  $m$  的位数组和一个哈希函数列表。
2. 添加元素：将元素通过哈希函数映射到位数组的  $n$  个位置上，将这些位置的值设置为 1。
3. 查询元素：将元素通过哈希函数映射到位数组的  $n$  个位置上，如果这些位置的值都为 1，那么认为该元素可能存在于集合中；如果存在某个位置的值不为 1，那么认为该元素一定不存在于集合中。
4. 删除元素：由于布隆过滤器不支持删除操作，所以需要重新添加元素来覆盖已删除的元素。

需要注意的是，布隆过滤器存在一定的误判率，即判断一个元素不存在时，实际上可能存在于集合中。我们可以通过调整位数组的长度和哈希函数的数量来降低误判率。

### 缓存预热

在系统规模大到一定程度，例如在活动开始的一瞬间，Redis 缓存的写入流量超过了 1Gbit/S，那么缓存预热将是一种投入小收益大的架构优化方式。通过提前把预期的热数据主动载入到缓存中，可以大大降低秒杀活动开始一瞬间的系统压力：如果业务进程需要等待数据载入缓存，那么就会在一瞬间引发“多级缓存雪崩”，系统在刚开始的 30 秒，压力会非常大。而如果我们提前把热数据载入了内存缓存的话，那活动开始时瞬间的突发流量就会被抹平，等于系统一步就进入了稳定运行状态，对秒杀流程中的所有机器和服务都是一种有效的保护。

### 缓存高可用

经过前面两章分布式数据库的洗礼，想必大家已经了解了“内存缓存同步”是分布式系统的万恶之源了。为什么内存缓存同步这么难搞呢？因为它的性能实在是太高了，内存的读写速度比网络高了两个数量级，延迟更是差了上千倍，所以如果我们需要通过又慢延迟又高的网络来同步内存中的数据，一定要小心翼翼，从架构设计到用户读写的策略，都要保持在分布式缓存系统的“舒适区”内，否则集群的性能和一致性一定会要你好看。

一旦某项数据被放入了缓存，那这项数据大概率首先追求的是高性能，其次是高可用，最后才是节点之间的一致性，所以 Redis 的各种集群方案都是以高性能、高可用为设计目标的。其实直到今天，Redis 集群的依然是遵循最常见的集群设计思路来演进的：主从同步-增量复制-主从切换哨兵-哨兵集群。

基本的设计方法我们在前面的分布式数据库的章节中已经都了解过，下面笔者以 Redis 集群为例，简单阐述

一下缓存高可用目前依然没有被完美解决的一些问题。

1. 全量同步/增量同步悖论：全量同步可以获得最完善的一致性，但是需要消耗大量的资源；增量同步的性能更好，但需要提前规划环形缓冲区的内存空间大小，这就让增量同步在事实上只能用于“计划中”的运维事件，例如运维手动触发的主从切换，无法应用在真正的异常——节点宕机场景下。
2. 脑裂问题：缓存业务天然的高性能需求几乎让任何分布式缓存系统必然选择容忍脑裂问题。其实别说是脑裂了，就是基础的主从强一致性目前都没有一种集群架构能完全做到。
3. 故障切换：虽然哨兵集群在理论上解决了单节点宕机后主节点无法切换的大问题，但是哨兵集群也只是“看起来很美”。现实世界中，单节点宕机只是一个小概率事件，更容易出现的其实是网络故障和集群容量爆满问题，这两个问题哨兵集群都无能为力。

这里笔者想引申解释一下为什么 Kubernetes 集群的高可用看起来那么完美：因为 Kubernetes 只把可以做到集群化高可用的部分留在了自己的系统里，真正的单点——网关和流量洪峰都是无法被 Kubernetes 集群很好地处理的，这两个高可用系统中最关键的要素都需要我们在架构的其他部分自己想办法解决。

## Elasticsearch 的缓存设计方案

作为知名内存使用大户，Elasticsearch 的缓存功能十分丰富，我们以页缓存、分片级请求缓存和查询缓存为例，学习一下 Elasticsearch 优秀的缓存设计思想。

### 页缓存

和 InnoDB 的 Buffer Pool 设计类似，Elasticsearch 会给磁盘上的数据页生成一个内存镜像。当查询数据时，Elasticsearch 首先会检查页缓存中是否已经存在所需的数据，如果存在，则直接返回结果，避免了磁盘 I/O 操作，提高了查询速度。同时，页缓存还可以对写入操作进行缓冲，以提高写入性能。

### 分片级请求缓存

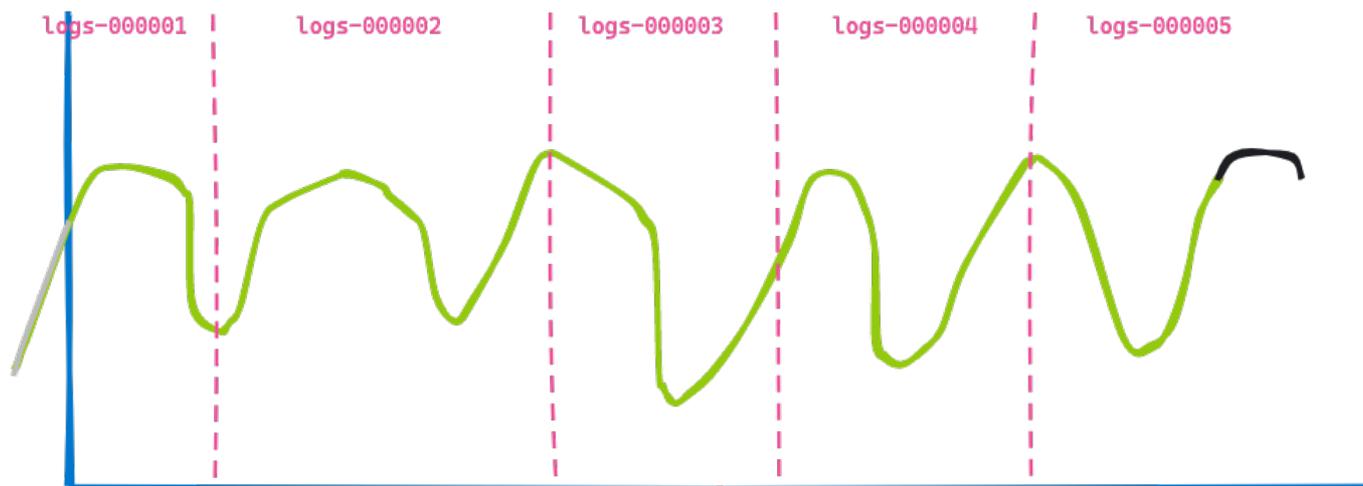


图 11-3 Elasticsearch 分片级请求缓存的适用场景

如图 11-3 所示就是分片级请求缓存的应用场景：在查看日志场景下，旧数据的查询条件是不变的，用户刷新页面只是为了获取新数据，那我们把旧数据的查询结果缓存起来，就可以避免每次都“扫描全盘”。这种缓存主要用于加速 Kibana 的运行速度。它会缓存聚合操作的结果，当相同的查询再次出现时，可以直接返

回结果，而不需要重新进行搜索和聚合操作。

## 查询缓存

查询缓存是针对整个查询结果进行缓存的机制。当一个查询请求到达 Elasticsearch 时，Elasticsearch 会先检查查询缓存中是否已经存在相应的结果，如果存在，则直接返回结果，避免了重复执行查询，其背后的哲学思想和上面的分片级请求缓存一致。

基于局部性原理，这个缓存生效的概率要远大于人们直观的印象。

## 11.4 秒杀系统的核心——队列

有了前面几章数据库部分的铺垫，相信读者现在都认识到了，在库存需要一单一单地减掉的情况下，这个减库存的过程在宏观上必然有一个单点——这个单点需要把所有的需求排成队，一个一个地执行完成。

在一个没有经过任何复杂架构设计的电商系统中，扣库存需要基于 MySQL 中 `update` 语句自带的事务性来实现的排队：每一句 `update` 在执行的时候，MySQL 都会给它自动包裹成一个事务来执行。MySQL 的事务确实非常的严谨，执行不会出错，但是性能实在是不可行，而且如果瞬间并发大量有冲突的事务，还会引发死锁，导致连接数暴涨，数据库服务挂掉。

怎么解决这个问题呢？很简单，用单独设计的队列把这个“单点”的工作接管过来，让 MySQL 能够舒服又高效地运行。

### 队列解决下单性能问题

在团购秒杀活动刚开始的几分钟，下单压力确实很大，但是几分钟后压力就降到了比日常平峰期高不了多少的水平。所以，我们可以用队列来把短期内暴增的数据库压力稍微地均匀分摊一下，让库存的扣减和订单的生成稍微等待一下，我们的数据库就不会挂，秒杀活动就能够正常地进行下去。

### 队列解决超售问题

超售问题的本质是计算机经典的“多线程”问题——在宏观上，同一个时刻有两个线程需要对同一个变量分别执行“减去 1”这个运算，如果我们不对这种并发行为做处理，最终可能两个线程都执行成功了，但是这个变量一共也只被减去了 1，而不是预期的 2。

这个问题用队列也能解决，而且有两种解决方案：

1. 利用队列来降低争抢烈度：用户做秒杀可能一千个人一起抢，但是后端只开了 3 个队列处理器进程，这就变成了 3 个人一起抢，即便还用 MySQL 事务来做原子化操作，死锁的概率就低多了，代价就是大部分用户需要等待前面的任务都处理完才能拿到自己没抢到的结果。
2. 直接利用 Redis 的原子性：将库存值直接存入 Redis，只要 Redis 还是单体架构，那对同一个 key 的操作就是原子化的。这么做的好处是完全不依赖数据库了，理论性能会更高，坏处是 Redis 会承担更大的压力，需要设计更复杂的锁机制来减少对 Redis 的压力，此外 MySQL 和 Redis 之间的数据同步也是一个无法被忽略的大问题。

## 11.5 真实的队列秒杀架构——住范儿电商

在设计住范儿电商秒杀架构时，笔者对开团瞬间的高并发需求做过全链路的分析，经过各种推导、建模，发现高并发下单过程的单点最终只集中到了一个点上——库存。于是为了提升库存的处理能力，笔者把设计出的几种看起来很复杂的架构合并成了一个“略微复杂”的队列扣库存架构，如图 11-4 所示。接下来，我们对每一步进行详细解读。

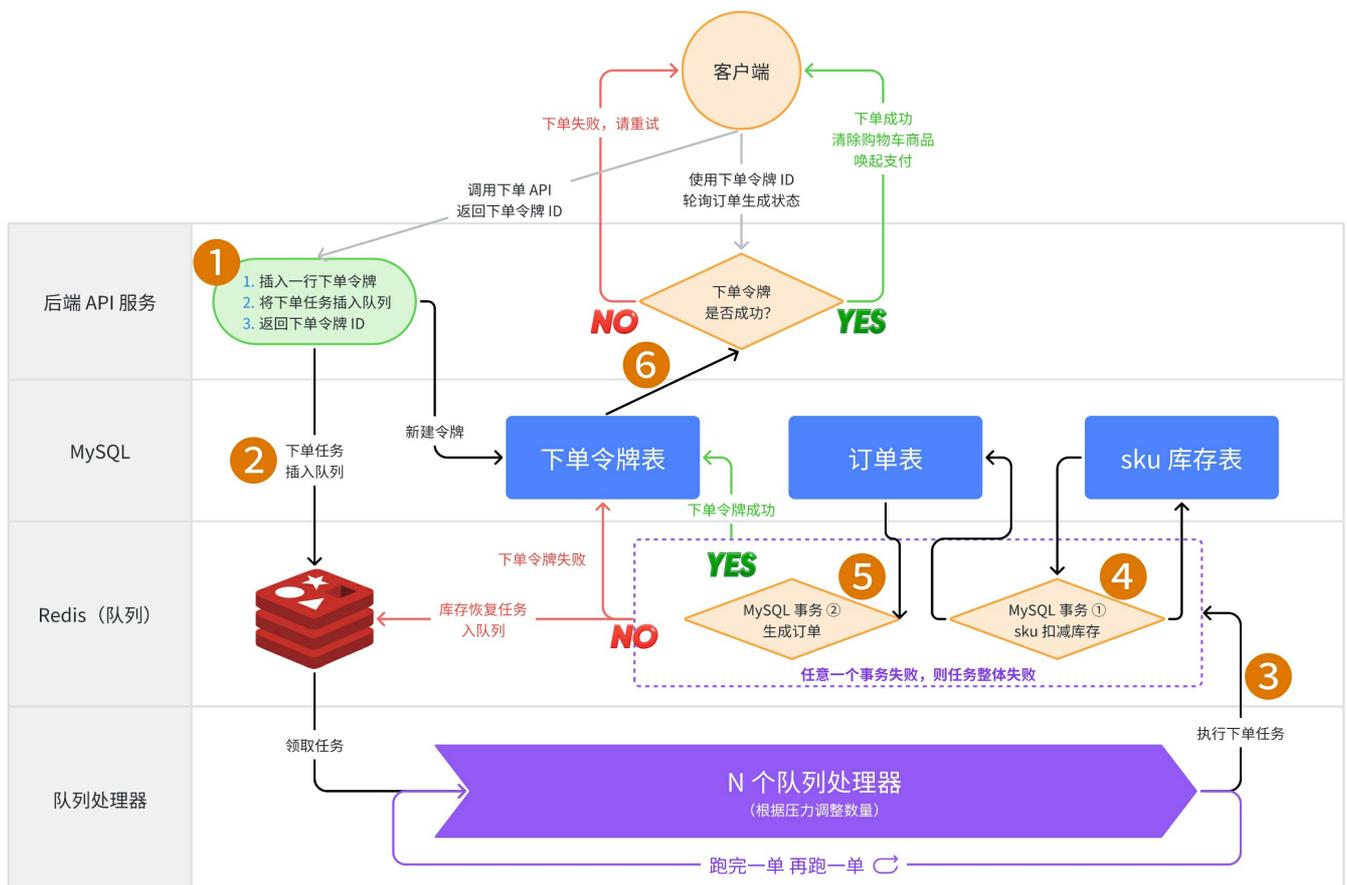


图 11-4 住范儿电商平台真实秒杀架构图

### 1. 获取下单令牌

下单令牌可以看做一个用户在向后端发送了下单请求之后拿到的唯一查询凭证，此处笔者使用 MySQL 来承载，理论上用 Redis 也可以，推荐在性能压力更大的情况下使用 Redis。

### 2. 将下单任务插入进队列

第二步，我们使用下单令牌作为标记，将这个订单的下单任务插入到队列中，等待队列处理器进行处理。

### 3. 队列处理器执行下单任务

第三步，队列处理器取出一个下单任务，开始执行。需要注意的是，此处的下单队列处理器并不只有一个，只要保持一个比较低的并发量就可以实现对 MySQL 的保护。

### 4. 第一个 MySQL 事务：扣减 sku 的库存

随后，在队列处理器进程中，我们会利用 MySQL 事务来进行 sku 库存的真正扣减。

1. 执行一段 SQL `update set store_num = store_num - 1 where store_num >= 1`，然后判断影响的行数，如果大于 0，说明这个 sku 的库存扣减成功了。
2. 如果此时恰好别的队列处理器也在扣减同一个 sku 的库存，那我们的进程会等待，只要最终不超时即可。
3. 我们需要对该订单包含的每一个 sku 都执行一条上述 SQL，任意一个 sku 库存扣减失败，手动将已经扣掉的库存全部回滚。根据住范儿电商系统运行的实际经验，遇到冲突全部回滚的概率非常低，下单成功率是非常高的。

需要说明的是，4、5 步的两个 MySQL 事务是被同一个 Try Catch 包裹住的，所以如果第一个事务成功而第二个事务失败的话，需要再新增一个队列任务，把库存再加回去。

## 5. 第二个 MySQL 事务：生成订单

如果所有 sku 扣库存操作都成功了，那我们就需要执行第二个新建订单的事务了。读者可能会疑惑，生成订单不是一个简单的 Insert 吗，为什么需要事务呢？因为真实世界中，下单的时候需要做的检查和数据的写入还是很多的，例如用户状态检查、收货地址有效性检查、活动和优惠有效性检查、商品有效性检查、店铺有效性检查、购物车对应的商品删除等操作；更极端一点，数据库服务器磁盘满了怎么办。所以下单这么重要的操作加上事务是非常有必要的，我们需要确保订单信息的自治和完整。

在生成订单后，还有各种繁杂的生成快照和日志的需求，笔者在图中没有画，但是依然需要使用队列任务来异步进行。

## 6. 给客户端返回下单结果

下单完成后，就可以将下单令牌标记为下单成功了，客户端的轮询终于可以拿到结果了。之后，我们就可以唤起支付了。

### 为什么要使用两步事务

理论上前面的两个 MySQL 事务完全可以用一个事务来实现，但笔者设计成了两个，这是为了尽量减少多个队列处理线程之间的冲突：在最有可能发生冲突的 sku 扣减库存操作中，一个 MySQL 事务需要执行尽量少的任务，这对我们宏观上的系统总容量来说意义重大。

## 11.6 缓存和队列的架构意义

看了前面的真实流程，想必读者已经感受到了缓存和队列对于一个高并发架构的意义了，下面我们简单讨论一下。

### 缓存的本质是用一致性换取读取性能

缓存好理解，放弃一部分数据一致性，获得一个客观的读取性能提升，这在处理不经常更新的数据时非常有用。

## 队列的本质将关键操作从同步改为异步

队列对于以 HTTP 为基础运行方式的 Web 后端系统意义重大，它的核心价值是把一些耗时的任务从同步执行改为了异步执行。很多任务本身并不需要很多的资源，但是当两个任务连在一起执行的时候就会导致长时间的资源占用。我们使用队列工具，在时间上把任务拆分开，可以很大程度解放系统性能，因为我们学会了“用未来的时间换现在的时间”。

## 消息订阅-架构解耦对高并发系统架构的决定性影响

在业务复杂的大型高并发系统中，各个业务之间的协作是一个大问题：如果用传统的 hook 钩子的方式执行，整个系统中就会出现大量的等待，甚至会显著影响用户体验，代价太大了。而以 Kafka 为代表的消息订阅软件对大型系统的架构设计做出了很大贡献，甚至可以说正是消息订阅带来的架构解耦能力决定了今天的高并发系统的解耦架构。

### Kafka 分布式消息订阅系统

Kafka 把消息订阅-发布系统提升到了一个非常的高度。由于消息之间没有关系，所以数据的离散性让 Kafka 可以放开手脚，设计出了一个近乎完美的高吞吐量、高可靠性、具有持久性的分布式消息订阅系统。

## 11.7 面试题

---

### No.37：如何设计一个每秒一万单的秒杀系统？

使用队列可以实现一个每秒一万单的秒杀系统。

1. 用队列来把短期内暴增的数据库压力稍微地均匀分摊到一段时间中
2. 利用队列来降低争抢烈度：用户做秒杀可能一千个人一起抢，但是后端只开了 3 个队列处理器进程，这就变成了 3 个人一起抢，即便还用 MySQL 事务来做原子化操作，死锁的概率就低了很多，代价就是大部分用户需要等待前面的任务都处理完才能拿到自己没抢到的结果。
3. 直接利用 Redis 的原子性来减库存：将库存值直接存入 Redis，只要 Redis 还是单体架构，那对同一个 key 的操作就是原子化的。这么做的好处是完全不依赖数据库了，理论性能会更高，坏处是 Redis 会承担更大的压力，需要设计更复杂的锁机制来减少对 Redis 的压力，此外 MySQL 和 Redis 之间的数据同步也是一个无法忽略的大问题。

### No.38：分布式缓存如何保证数据一致性？

在分布式缓存系统中，保证数据一致性是最重要的需求。下面是几种常见的保证数据一致性的方法：

1. 缓存失效策略：通过标记失效和重新获取最新数据来避免脏数据的读取。
2. 更新广播：通过消息队列通知变动，使缓存节点及时更新数据。
3. 读写锁：控制对共享资源的访问，读操作并发进行，写操作独占资源。
4. 基于版本号的处理：通过比较版本号来更新缓存。
5. 一致性哈希算法：将数据分散到不同节点，影响部分数据的迁移。
6. 分布式事务：确保操作的原子性和一致性。

### No.39 : 如何解决商品超售问题 ?

商品超售问题可以使用以下策略解决:

1. 库存预留: 在接收订单前, 先将库存进行预留。如果库存不足无法预留, 则不接受新订单。只有当库存足够时才接受订单, 并及时更新库存。
2. 并发控制: 使用锁机制或者队列等方式, 控制对库存的并发访问。当有多个请求同时减少库存时, 只允许一个请求成功, 其他请求需要等待或者进行退款处理。
3. 保证检查库存操作和下单操作的原子性: 在下单过程中, 需要保证检查库存和生成订单的操作是原子执行的, 避免并发操作导致超售问题。

### No.40 : 如何保证消息仅被消费一次 ?

保证消息仅被消费一次, 有以下几种方法:

1. 消费者确认机制 (ACK) : 消费者在处理完消息后向消息队列发送确认信号, 告知队列消息已被正确消费。消息队列收到确认信号后将该消息标记为已处理并删除或移到已消费队列, 确保每条消息只被一个消费者处理。
2. 消息去重: 消费者通过唯一标识符或业务逻辑判断实现消息去重。记录已处理过的消息标识符, 下次消费时先检查该标识符, 若已存在则不再处理。
3. 幂等性处理: 消费者实现幂等性处理, 即使同一条消息被重复消费, 也能保证最终结果一致。通过添加幂等性判断和处理机制, 在多次处理同一条消息时不会产生错误。
4. 消费者分组: 将多个消费者分组成一个消费者组。消息只会被组内某个消费者消费, 其他消费者无法接收相同消息, 确保同一条消息只被消费者组内处理。
5. 基于事务的消费: 使用支持事务的消息队列, 将消息的消费与业务逻辑放在一个事务中进行。如果事务提交成功, 则消息被标记为已处理; 否则回滚事务, 确保消息不被重复消费。

### No.41 : 如何降低分布式消息队列中消息的延迟 ?

要降低分布式消息队列中消息的延迟, 可以尝试以下几种方法:

1. 异步处理: 将操作异步化, 提高处理吞吐量, 减少等待时间。
2. 批量处理: 批量拉取和写入, 减少网络开销和调用次数, 降低延迟。
3. 负载均衡: 合理分配负载, 避免过载导致延迟增加。
4. 预取机制: 引入预取机制, 提前拉取下一批消息, 避免等待时间。
5. 消息索引和缓存: 在内存中维护索引和缓存, 减少访问时间开销。
6. 水平扩展: 增加并行度, 通过分区或增加消费者数量提升处理能力。
7. 性能监控和优化: 定期监控性能指标, 针对瓶颈问题进行优化, 提高整体性能和效率。

### No.42 : Kafka、Hadoop 和 Clickhouse 背后有哪个相同的基本原理 ?

Kafka、Hadoop 和 Clickhouse 都是基于分布式系统的架构设计, 能够处理大规模的数据。它们将数据分散存储在多个节点上, 以实现高可用性和可扩展性。在它们背后有一个相同的基本原理:

把数据之间的关系降到最低, 让数据离散化, 这样就可以尽可能地让数据的处理在多个 CPU 核心甚至是多台机器上并行起来。



# 第五部分 无限容量架构

## 第 12 章 无限容量架构——站在地球表面

## 第 12 章 无限容量架构——站在地球表面

本章我们将从微服务架构讲起，一步一步追根溯源，找寻“分布式数据库”在另一个维度的投影，探寻基建、应用、服务、组织之间的联系，通过观察自然规律和人类社会，引出本文的中心思想，并对“找出单点，进行拆分”做出最后的升华。

最后，我们将得到一个可行的 100 万 API QPS、500 万数据库 QPS 的系统设计方案，并顺便简单地讨论一下“高可用”需求背后的逻辑悖论。

### 12.1 概述

---

如何定义“一个系统”？

前面第七章我们说过：

如果两个系统的数据库不在一起，那它们就不是同一个系统，就像拼多多有 7.5 亿月活跃用户，淘宝有 8.5 亿，你不能简单地说“拼宝宝”电商系统拥有 16 亿月活跃用户一样。

如果两个 API 的数据落到同一张表上，那它们两个就属于同一个系统。

那，在一个电商系统内，用户 API 和商品 API 似乎没有多对多交集？是的，这就是微服务架构的拆分逻辑。

## 12.2 从业务分库到微服务

前面我们说过，微服务的拆分方式，反映的是其背后技术团队的组织方式。

那，技术团队的组织方式是什么决定的呢？

是由系统内各部分天然的内聚性决定的：用户相关的业务和商品相关的业务都有很强的内聚性，它们之间不会主动发生关联，但它们会分别和订单发生关联。

### 数据库调用推演

我们用商品下单处理流程来推演一个电商订单的生命周期内对用户、商品、订单三个部分数据库的读写情况。

1. 搜索：商品数据库-读
2. 点进详情：商品数据库-读，用户数据库-读（地址、会员）
3. 立即购买：商品数据库-读，用户数据库-读，订单数据库-写
4. 支付：订单数据库-读写，用户数据库-读
5. 商家后台查看并处理订单：订单数据库-读写，用户数据库-读，商品数据库-读

我们可以看出，大部分情况下，每一步都只有一个主要的微服务被需要，其它微服务都处于辅助地位：只读，且大部分都是单点读取。这就为我们降低了数据库单点的负载——只要把这三个微服务部署到三个独立的数据库上，就可以通过 API 调用的形式降低单个数据库的极限 QPS。

### 微服务背后的哲学

既然我们不能把拼多多和淘宝的系统称作一个系统，那么，在拼多多和淘宝系统内，肯定还可以基于类似的逻辑继续拆分：

1. 在大量调用的 API 中，一次携带了数据写入的请求一定只会对单个微服务进行写入，但会对多个微服务进行数据读取
2. 如果某个头部 API 请求会对两个微服务系统进行写入，那说明微服务的划分出了问题，需要调整系统结构划分

把几乎不相互写入的数据拆到两个数据库上，这种组织形态在人类社会随处可见：两个国家的人分别在自己

国家申请护照，他们有时也可以到对方国家内的本国领事馆申领本国护照；两个村的人各自在本村的井里打水，有时也可以不怕麻烦地去隔壁村的水井里打水；你每天早上都用滴滴打车上班，万一滴滴打不到车你还可以用高德来补救...

## 微服务照进现实

微服务的拆分思想相信大家理解了，下面我们来解决现实问题。

如果你真的成为了“设计百万 QPS 系统”的架构师，相信我，你第一个想到的，一定是“削峰”。

## 12.3 削峰

---

顾名思义，将突发的流量高峰削平。大促的时候，系统顶不住，其实就是那么一会儿顶不住，只要初期的高热度下去了，系统的整体负载会迅速下降到没那么危险的水平。所以，对付突发流量，削峰是第一要务。

### 缓存——饮鸩止渴

请原谅笔者用饮鸩止渴这个词形容 95% 的 Web 系统的真正性能顶梁柱，实际上缓存的贡献远不止于此。

1. 缓存以降低数据更新频率为代价，极大地提升了读取 API 的 QPS 极限
2. 缓存可以设计成多级，形成一个逻辑上的“存储器山”
3. 缓存可以像事务隔离级别一样划分成好几种“性能+数据一致性”级别

但是，使用缓存确实是在饮鸩止渴：缓存在带来性能的同时，大幅削弱了数据库提供的“单点性”，为系统失效埋下了一堆地雷。

缓存的毒性无法消除，一旦系统的某些部分失效，这杯毒酒就会发作，但是会不会把自己毒死还要看架构水平和基建能力：你的机房别随便断电，你的服务器别随便宕机，你在喝下缓存毒酒之后，就能活的够长。

### 终极缓存方案

我们前面章节中讨论过的 OceanBase 就采用了一种终极缓存方案：

1. 内存不是快吗，那我就把所有热数据全部放到内存里
2. 那对热数据的修改怎么办？redo log 落盘啊
3. 你说数据库重启怎么办？上冗余，一个节点一时半会儿起不来也没问题
4. 整个集群重启呢？都天灾了，集群重启后，停止服务一个小时还是可以接受的吧

12306 每天夜里都要维护一个小时，笔者没有证据地胡乱猜测一下，它在将 redo log 落盘。参考资料：《[12306 互联网售票系统的架构优化及演进](#)》。

### 队列——欢迎来到地球

缓存可以削“读”的峰，队列就是拿来削“写”的峰的。

队列的思想其实早就贯穿在人类社会的每一个角落了：超市结账需要排队，做核酸需要排队，火车站打车需

要排队，网上抢购商品也需要排队。排队的本质是将一拥而上购买变成了“异步”购买：你想买东西？先排队，过段时间轮到你了，看看商品有没有卖完，没卖完你就能买到。

排队的思维特别好理解，而现实中防止超售功能也确实是基于排队功能做的。传统数据库的事务隔离属于强迫用户等待，而现在使用队列系统来处理排队，排队这件事情才真正异步了起来。

### 奇技淫巧

电商下单在普通压力下，一个队列就能解决问题，但是当你面临每秒几十万单的时候，如何让这些订单真正地下单成功，才是最需要解决的问题，这个数字就是“系统容量”。队列是无法提升系统的绝对容量的，那该怎么办呢？

继续找东西和信息之神交换：过去的时间换现在的时间。

在大促之前，先把订单生成好，然后用户下单时直接写入用户信息：不需要执行“检查库存”这个单点操作了，负载低了很多。当初想出这个方法的人实在是太机智了。

### 现实世界中的削峰

现实世界中，一个一百万 QPS 的电商系统，真正需要触达到数据库的 QPS 其实是没有 500 万那么多的，在削峰的操作下，200 万 QPS 的 PolarDB 集群在 Redis 集群的配合下，是可以顶住 100 万 API QPS 的。史上流量最大的那一年双 11，每秒订单创建最大值仅为 58.3 万笔，这已经是这个地球上的最高记录了。

但是，现实世界中，一切都要讲 ROI（收益成本之比）的，搞一套顶配的 PolarDB 集群确实可以顶住巅峰时期一百万 QPS，但是你老板看着账单肯定会肉痛，那，该如何省钱呢？

答：[站在地球表面](#)。

## 12.4 站在地球表面

---



图 12-1 一望无际的华北平原

如图 12-1 所示就是地球表面的代表——一望无际的华北平原。

北京位于美丽的华北平原北端，生活着两千多万人，在巅峰的 2020 年双 11，天猫平台北京地区销售额为 216 亿，全国总额为 4982 亿，占比为 4.33%，略高于北京占全国 3.44% 的 GDP 比例，数据比较可信。使用这些数字我们可以计算得出，北京的两千多万人，给天猫贡献了  $583000 * 0.0433 = 25243.9$  笔/秒的并发。

虽然全国订单数看起来十分惊人，但是北京这一个地方的压力却只有 2.5 万单每秒，这个哪怕不用奇技淫巧，纯靠数据库硬抗，十万数据库 QPS 只用主从架构可能都能抗住。但是，系统能基于地理位置划分吗？系统不是必须全国一盘棋吗？不是的，可以划分。

下面我们讨论一下怎么划分。

### 基于地理位置对应用和数据库分区

为什么非要全国的用户访问同一个数据库呢？我们可以利用微服务思想对业务系统和数据进行拆分：北京的用户和上海的用户，理论上讲都可以只访问“本地天猫”。

接下来我们分析一下，在一个标准的电商业务中，哪些地方会让一个北京的用户和一个上海的用户发生联系。

1. 用户表自增 ID
2. 商品库存检查
3. 商家订单聚合
4. 离线数据分析

实际上，地理上被隔开的两个人，在系统内还真没什么机会需要相互查询对方的数据，这就是我们能基于地理位置对应用和数据库进行分区的现实原因。下面我们一一拆除这几个单点：

1. 用户表自增 ID：可以预分配 ID 段，也可以用算法保证，例如一奇一偶。
2. 商品库存检查：预分配库存，再异步刷新缓存。这个部分能玩出花，甚至有客户端上提前下发抢购结果的骚操作，大家可以自己探索。
3. 商家订单聚合：简单地从两个地方各拉一次即可。
4. 离线数据分析：更不用说了，都离线了，本身也是要做很多数据同步和聚合的，不差这一点。

基于地理位置对应用和数据库进行划分，产生出两个“本地天猫”后，就需要我们老朋友 DNS 出来表演了。

## 进击的 DNS

域名当初可能是为了方便记忆而发明的，但是域名背后的 DNS 服务却几乎是最重要的互联网高并发基础设施：不同地区的人，对同一个域名进行访问，可以获得两个公网 ip，这样“本地天猫”就实现了。

### 类 DNS 哲学思想：Consul 和 Kong

DNS 几乎完全放弃了一致性，但却实现了极高的可用性和分区容错性。其实，gossip 协议也是这个思想：让消息像病毒一样传播，能够实现最终一致性就行了，要啥自行车。

异曲同工的 Kong 集群的设计思想也十分令人震惊：所有节点每 5 秒从数据库读取最新的配置文件，然后，这些节点就成了一个行为完全一致的集群啦。“想那么多干什么，短时间内多个节点的行为不一致，就让它们不一致好了，5 秒之后不就一致了。”

## 高性能计算第一原则

高性能计算第一原则：数据离 CPU 越近，性能越高，容量越小

在我们熟悉的存储器山中，这是一个大家都理解的基本特性，而这个特性引申到分布式系统中，就是：一定不能应用和数据库分离。

和 InnoDB 一样，很多时候其实是“局部性”这个我们宇宙的基本属性在帮助我们提升系统的性能，让应用和数据库分布在同一个地域，也是在利用局部性获得性能增益。

所以，让应用去隔壁区域的数据库读数据是要极力避免的——我们应该用 API 网关直接把请求发给隔壁区域的应用服务器，这显然是在今天这个异地网络传输速度接近光速的时代最佳的选择。

## 向 Clickhouse 学习高并发

Clickhouse 在亿级数据量面前丝毫不怵：MySQL、MongoDB、Hadoop，谁也没有老子快。为什么 Clickhouse 这么快呢？

### 列式存储

首先，它将数据以列为单位组织起来，压缩后存入磁盘上一个又一个的 block，这些 block 就像 InnoDB 的 16KB 页一样，只是它更大（64KB~1MB）。这样，当我们 select 某个 column 的时候，Clickhouse 就能顺序读出磁盘上这个 column 下面所有行的数据。

### 高效压缩

由于同一列通常具有高度的数据相似性，所以列式压缩的效果在大部分情况下都非常好，这也能算作“局部性”在数据库层面的一种应用。

### 充分利用多 CPU 并行计算

除了列存储之外，每个 block 内，Clickhouse 还用“稀疏索引”的方式，将每一列的数据划分为了多个 granularity（颗粒度），然后给每个 granularity 分配一个 CPU 核心进行并行计算，并且它还利用 SSE4.2 指令集，利用 CPU 的 SIMD（Single Instruction Multiple Data）指令，在 CPU 寄存器层面进行并行操作。

### 放弃内存缓存

这是 Clickhouse 整个架构中笔者最喜欢的部分。我们通过前面的章节可以看出，所有的分布式数据库，其本质都是在搞“内存缓存的数据同步”，Clickhouse 直接掀桌子——老子不要内存缓存了。由于所有数据都在磁盘上，而节点的 CPU 又直接和磁盘数据打交道，所以 Clickhouse 实现了真正的并行：增加 CPU 核心数就能提升系统容量，无论在不在同一台机器上都行，反正 CPU 相互之间完全不需要通信。这样，Clickhouse 通过堆核心数就能够实现系统容量的“近线性扩展”。

### 太暴力了，我喜欢

我们可以学习这种思想，打造一个可以线性扩展的系统架构：只要不同地区的本地系统之间完全没有“数据实时同步”需求，那其实它们就是两个系统，就可以实现线性的性能提升。

### 理论无限容量

我们说过，关系型数据库的关系，指的就是两行数据之间的关系。现实世界中，位于异地甚至是异国的两个人之间，几乎是不会发生实时相互数据读取的。

站在地球表面来思考，你会发现人类社会和自然规律都是契合高并发“找出单点，进行拆分”哲学原理的：每一个人类居所，本质上都是散落在整个地球上的一个又一个点。因为这些点的存在，我们发明了国省市县乡村逐级政府，同级政府之间几乎没有相互通信。



将一个大系统拆成不需要实时相互通信的多个小系统，可以获得线性的性能提升。

当你的系统顶不住的时候，按照这个原理来拆就行了，绝对顶得住。别说区区一百万 QPS 了，服务全人类也做得到，毕竟全球 80 亿人都生活在如图 12-2 所示的球体上嘛，这个球体的半径也只有六千多公里，不算是很大的一个球。



图 12-2 我们的地球

价值上完了，我们最后再讨论一下高可用。

## 12.5 番外篇：高可用

笔者相信，很多人都像我一样做过思想实验，希望设计一个“完全高可用”的系统，但是最终可能都败下了阵来，为什么？因为高可用和其它常见的分布式系统需求是互斥的。

数据重要如银行，也只是要求在天灾面前要尽量不丢数据、少丢数据，凭什么你就要求自己的系统永远可用呢？其实，想从架构层面实现高可用是非常困难的，终极高可用就是将数据完整地复制到世界各地的所有节点上，并用超长的时间来达到完全一致，这是什么，这是区块链呀。

高可用和性能、一致性都是冲突的，只能采用策略尽量压制问题。

## 熔断

这个词在技术圈的流行应该有微博一半功劳，压力一大就熔断：主动停止不重要的服务，断尾自救，争取让核心业务不挂。

## 限流

限制一部分地区、一部分用户的访问，以保护整个集群不崩，一般用于限制单个用户对系统造成的压力过大，对面很可能是机器人。

## 笔者对于设备故障的经验

在机房硬件设备中，最容易损坏的肯定是磁盘（硬盘），因为它是在不断磨损的，即便是 SSD 也会随着时间的流逝寿命逐渐丢失，哪怕装在盒子里不通电也一样。但是硬盘之后，容易宕机或者损坏的是什么大家应该就猜不到了。

首先是断电：数据中心因为各种问题断电是最常见的故障，这个故障的概率甚至要高于软件引发的故障和电源适配器的故障。

其次是内存失效：内存以及内存插槽，内存电路，似乎是今天服务器硬件之中除了磁盘之外最容易坏的东西，我们刚刚已经进入了 DDR5 时代，内存功耗又上了一个台阶，恐怕故障率会进一步上升。

然后是网络线材和供电线材接口的问题，时间长了松了或者进灰了，就会丢包或者重启。2011 年震惊世界的中微子超光速事件，就是插头松了导致的。

最后是高温引发的宕机：特别是 GPU 服务器，一旦服务器或者机房的散热系统出现问题，服务器很容易就主动限制性能甚至关机。

其它的，硬件网络设备（路由器交换机）故障、彻底挖断光纤、CPU 损坏、电源转换器寿命耗尽、主板电池故障、地震火灾洪水等，可能在一台服务器的上架寿命之中都完全无法遇到。

## 机房进水导致服务器损坏事件

就在笔者完成第一版书稿的几天后，住范儿公司网络机房内部署的一些非关键业务的服务器遭遇了进水损坏这个致命的问题。这个故事告诉我们，很多看起来很小概率的事故，是真的有可能荒谬地发生的，数据异地备份十分重要。

## Facebook 2021年 10 月 4 号宕机

Facebook 的用户不可谓不多，对高可用的投入不可谓不足，为什么还是会 [整个公司完全宕机 7 小时](#) 呢？

事故的起因是一个错误的命令意外断开了 Facebook 的 DNS 服务，结果问题大了：

1. 所有客户端 API 失效，用户无法获得任何信息
2. 数据中心 VPN 服务失效，无法远程登录到数据中心内的设备上
3. 亲自去机房，发现门禁卡刷不开门，在暴力破拆后才接触到物理设备，插上显示器和键盘才能解决问题
4. 邮件、Google 文档、Zoom 都登不上
5. 办公大楼的门禁卡系统也失效了，无法刷开会议室的门，甚至无法离开办公楼

结合阿里云香港一个数据中心 [因为空调故障导致整个数据中心宕机超过 24 小时](#)，认命吧，商业机构做不了真正的高可用的：资源使用率就是钱呐。

## 12.6 面试题

---

**No.43**：微服务该如何拆分？

微服务的拆分是一个复杂的任务，没有一种通用的方法适用于所有情况。以下是一些常见的方法：

1. 按照业务领域拆分：将不同的业务领域划分为独立的微服务，每个微服务负责处理特定的业务场景和业务逻辑。
2. 按照团队结构拆分：根据团队组织结构和技术能力，将系统拆分为适当规模的微服务，同时确保每个微服务的独立性和自治性。
3. 按照业务边界拆分：确定业务流程、功能模块或子域的边界，以此划分微服务的范围，实现微服务之间的松耦合和高内聚。
4. 按照数据和数据库拆分：根据数据的访问频率、复杂性和关联性，将不同的数据集和数据库划分为不同的微服务，确保每个微服务仅处理其需要的数据。
5. 按照用户界面或外部接口拆分：将不同的用户界面或外部接口划分为不同的微服务，以便更好地满足特定用户需求。
6. 按照功能模块或领域驱动设计拆分：根据系统的功能点或领域概念，将系统划分为独立的微服务，支持团队的自治性和快速迭代。
7. 按照性能和可伸缩性需求拆分：根据系统的负载和流量模式，将不同的功能点拆分为可独立扩展和部署的微服务，实现系统的水平扩展和弹性伸缩。

**No.44**：横跨几十个分布式组件的慢请求要如何排查？

横跨多个分布式组件的慢请求排查方法：

1. 分析请求路径，了解请求流经的组件。
2. 检查日志，特别是错误日志和警告信息，寻找异常或错误。
3. 进行时序分析，绘制性能指标时序图，对比不同组件之间的延迟和性能。
4. 检查资源消耗，包括 CPU、内存、磁盘 I/O 等，查找异常高资源利用率。
5. 检查并发处理能力，确定是否存在并发瓶颈。

6. 检查网络连接，包括延迟和带宽限制，排查网络问题。

我们需要从根本上使用服务治理的方式解决慢请求问题：

1. 并行与串行度：评估请求路径上每个组件的并行和串行处理能力。如果存在串行瓶颈，可以考虑优化请求路径，减少依赖关系，提高并行处理能力。
2. 异常处理：确保每个组件的错误和异常处理机制良好，避免因为异常情况而导致整个请求变慢或失败。
3. 压力测试：模拟高负载场景，持续发送多个请求，并监测系统行为。观察响应时间的变化，并找出性能下降的关键组件。

**No.45**：如何设计全链路压测平台？

全链路压测是指对一个系统或应用的各个环节进行综合性能测试，模拟真实场景下的高负载运行情况，以评估系统在高压下的表现和稳定性。设计全链路压测平台需要考虑以下几点：

1. 全面模拟用户操作流程，涵盖系统各组件和服务。
2. 考虑系统可能面对的不同场景和用户行为，进行并发访问、大规模数据请求、异常输入等模拟。
3. 整合各组件和服务，确保有效协同工作，满足性能目标。
4. 通过全链路压测发现性能瓶颈和薄弱环节，及时进行优化和改进。

在设计全链路压测时，应遵循以下步骤：

1. 确定目标：明确压测的目的和需求，如评估性能极限、发现瓶颈、验证容量等。
2. 制定方案：根据系统架构和功能特点，设计压测方案，包括场景、负载模型、用户数、频率等，并制定测试计划和资源需求。
3. 模拟用户行为：模拟用户操作流程和行为，考虑正常和异常情况。
4. 设置负载：确定并发用户数、请求频率等负载参数，模拟真实用户访问系统的情况，并逐步增加负载直至达到性能极限。
5. 迭代测试：每次测试后分析结果，根据反馈进行调整和改进，优化系统性能和稳定性。

**No.56**：Clickhouse 为何能够在一秒内完成上亿行数据的检索？

ClickHouse 是一种用于大规模数据存储和分析的列式数据库，它针对单字段全表读取的数据分析场景做了特殊的设计：

1. 列式存储：提高查询性能，只加载需要的数据列。
2. 数据压缩：减少磁盘 I/O 和网络传输的数据量，提高查询速度。
3. 并行化处理：充分利用多核和分布式计算资源，加快查询速度。
4. 索引优化：使用多级索引结构，加速数据查找和过滤。
5. 预聚合：提前计算和缓存聚合结果，减少实际查询时的计算量，提高查询速度。

**No.47**：什么时候需要限流？什么时候需要熔断？

限流和熔断都是在分布式系统中用于保护系统资源和提高系统稳定性的策略。

限流是指限制系统的请求流量，以防止系统过载。在以下情况下可能需要限流：

1. 高并发情况：当系统面临大量并发请求时，限流可以控制每秒请求数量，避免系统资源被耗尽。
2. 保护关键资源：对于一些需要保护的关键资源或接口，限流可以防止过度频繁的请求导致资源崩溃或数据不一致。
3. 防止恶意攻击：如果系统遭受到恶意请求或拒绝服务攻击，限流可以起到一定程度的防御作用。

熔断是在系统出现故障或异常情况下，为了保护系统资源和避免级联故障而采取的一种自我保护机制。一般情况下，熔断会在以下情况下启动：

1. 依赖服务故障：当系统依赖的某个服务出现故障或不可用时，为了避免出现雪崩效应，可以通过熔断来快速失败并降低对该服务的依赖。
2. 错误率超过阈值：如果系统的错误率超过预设阈值，说明系统自身出现了故障或异常，熔断可以快速切断请求，避免错误的结果被传播。
3. 主动停止非关键服务：在流量高峰期，对非关键服务进行服务降级甚至是熔断，让出宝贵的计算资源，断尾自救。

**No.48**：如何设计一个跨地区的高可用系统？

设计一个跨地区的高可用系统需要考虑以下几个方面：

1. 多地区部署：在多个地理位置上部署系统组件，以便在发生故障时其他地区可以提供服务，减少单点故障并提高容错性。
2. 负载均衡：利用负载均衡技术将用户请求分配到不同的服务器或数据中心，确保各地区的负载均衡，提高系统性能和稳定性。
3. 数据复制和同步：建立数据复制和同步机制，确保数据一致性和可用性，可采用数据同步技术和数据库复制等方式实现。
4. 异地容灾：将系统组件分布在不同地区，避免同一地区的自然灾害或网络故障对系统造成影响。同时，确保异地之间的网络连接可靠，以保证数据传输和系统协同工作。
5. 监控和自动化：引入监控和自动化机制，及时发现和处理故障。使用监控工具实时监测各地区的系统状态，并设置自动化脚本来进行故障恢复和系统调整。
6. 安全保障：重视跨地区系统的安全性，包括数据传输加密、访问控制和身份认证等安全措施，确保数据传输和存储的安全可靠。

**No.49**：基于地理位置拆分为何成为最流行的高并发系统拆分方案？

基于地理位置的拆分为何成为最流行的高并发系统拆分方案有以下几个原因：

1. 降低延迟：将系统的不同组件分布在多个地理位置上，可以使用户就近访问数据和服务。这样可以显著降低网络延迟，提高用户的访问速度和响应性能。
2. 增加容量：通过在多个地区部署系统的不同组件，可以将用户的请求分散到不同的服务器或数据中心上。这样可以增加整个系统的处理容量，从而提高系统的并发能力和吞吐量。
3. 提高可用性：跨地区部署可以提高系统的可用性和稳定性。如果某一地区发生故障或网络问题，其他地区仍然能够正常提供服务，避免单点故障导致整个系统瘫痪。
4. 大幅提高系统总容量：对于大规模的分布式系统而言，基于地理位置的拆分是一种有效的方式。通过将系

统的不同组件分布在不同地区，可以更好地管理和扩展整个系统，并实现弹性扩展和负载均衡等特性。

**No.50**：存在一种理论上无限容量的分布式系统吗？

理论上讲，目前还没有被证明存在容量无限的分布式系统。根据计算理论和信息论的原理，任何系统都会受到资源限制和通信延迟等因素的制约，因此其容量总是有限的。但是，在目前的实践中，从数据层面进行地理位置的拆分是一种接近无限容量的分布式系统设计方案，这也是目前头部互联网公司都在使用的一种方案。

。

# 其它系列文章

2023 年

2018 年

## 2023 年

### 自己动手开发互联网搜索引擎

---

#### 写在前面

---

本文是一篇教你做“合法搜索引擎”的文章，一切都符合《网络安全法》和 robots 业界规范的要求，如果你被公司要求爬一些上了反扒措施的网站，我个人建议你马上离职，我已经知道了好几起全公司数百人被一锅端的事件。

开源代码地址：<https://github.com/johnlui/DIY-Search-Engine>

## 《爬乙己》

---

搜索引擎圈的格局，是和别处不同的：只需稍作一番查考，便能获取一篇又一篇八股文，篇篇都是爬虫、索引、排序三板斧。可是这三板斧到底该怎么用代码写出来，却被作者们故意保持沉默，大抵可能确实是抄来的罢。

从我年方二十，便开始在新浪云计算店担任一名伙计，老板告诉我，我长相过于天真，无法应对那些难缠的云计算客户。这些客户时刻都要求我们的服务在线，每当出现故障，不到十秒钟电话就会纷至沓来，比我们的监控系统还要迅捷。所以过了几天，掌柜又说我不行了这事。幸亏云商店那边要人，无须辞退，便改为专管云商店运营的一种无聊职务了。

我从此便整天的坐在电话后面，专管我的职务。虽然只需要挨骂道歉，损失一些尊严，但总觉得有些无聊。掌柜是一副凶脸孔，主顾也没有好声气，教人活泼不得；只有在午饭后，众人一起散步时闲谈起搜索引擎，才能感受到几许欢笑，因此至今仍深刻铭记在心。

由于谷歌被戏称为“哥”，本镇居民就为当地的搜索引擎取了一个绰号，叫作度娘。

度娘一出现，所有人都笑了起来，有的叫到，“度娘，你昨天又加法律法规词了！”他不回答，对后台说，“温两个热搜，要一碟文库豆”，说着便排出九枚广告。我们又故意的高声嚷道，“你一定又骗了人家的钱了！”度娘睁大眼睛说，“你怎么这样凭空污人清白……”“什么清白？我前天亲眼见你卖了莆田系广告，第一屏全是。”度娘便涨红了脸，额上的青筋条条绽出，争辩道，“广告不能算偷……流量！……互联网广告的事，能算偷么？”接连便是难懂的话，什么“免费使用”，什么“CPM”之类，引得众人都哄笑起来：店内外充满了快活的空气。

## 本文目标

---

三板斧文章遍地都是，但是真的自己开发出来搜索引擎的人却少之又少，其实，开发一个搜索引擎没那么难，数据量也没有你想象的那么大，倒排索引也没有字面上看着那么炫酷，BM25 算法也没有它的表达式看起来那么夸张，只给几个人用的话也没多少计算压力。

突破自己心灵的枷锁，只靠自己就可以开发一个私有的互联网搜索引擎！

本文是一篇“跟我做”文章，只要你一步一步跟着我做，最后就可以得到一个可以运行的互联网搜索引擎。本文的后端语言采用 Golang，内存数据库采用 Redis，字典存储采用 MySQL，不用费尽心思地研究进程间通信，也不用绞尽脑汁地解决多线程和线程安全问题，也不用自己在磁盘上手搓 B+ 树致密排列，站着就把钱挣了。

## 目录

---

把大象装进冰箱，只需要三步：

1. 编写高性能爬虫，从互联网上爬取网页
2. 使用倒排索引技术，将网页拆分成字典

3. 使用 BM25 算法，返回搜索结果

## 第一步，编写高性能爬虫，从互联网上爬取网页

---

Golang 的协程使得它特别适合拿来开发高性能爬虫，只要利用外部 Redis 做好“协程间通信”，你有多少 CPU 核心 go 都可以吃完，而且代码写起来还特别简单，进程和线程都不需要自己管理。当然，协程功能强大，代码简略，这就导致它的 debug 成本很高：我在写协程代码的时候感觉自己像在炼丹，修改一个字符就可以让程序从龟速提升到十万倍，简直比操控 ChatGPT 还神奇。

在编写爬虫之前，我们需要知道从互联网上爬取内容需要遵纪守法，并遵守 [robots.txt](#)，否则，可能就要进去和前辈们切磋爬虫技术了。robots.txt 的具体规范大家可以自行搜索，下面跟着我开搞。

新建 go 项目我就不演示了，不会的可以问一下 ChatGPT~

### 爬虫工作流程

我们先设计一个可以落地的爬虫工作流程。

#### 1. 设计一个 UA

首先我们要给自己的爬虫设定一个 UA，尽量采用较新的 PC 浏览器的 UA 加以改造，加入我们自己的 spider 名称，我的项目叫“Enterprise Search Engine”简称 ESE，所以我设定的 UA 是 [Mozilla/5.0 \(Windows NT 10.0; Win64; x64\) AppleWebKit/537.36 \(KHTML, like Gecko\) Chrome/97.0.4280.67 Safari/537.36 ESESpider/1.0](#)，你们可以自己设定。

需要注意的是，部分网站会屏蔽非头部搜索引擎的爬虫，这个需要你们转动聪明的小脑袋瓜自己解决哦。

#### 2. 选择一个爬虫工具库

我选择的是 [PuerkitoBio/goquery](#)，它支持自定义 UA 爬取，并可以对爬到的 HTML 页面进行解析，进而得到对我们的搜索引擎十分重要的页面标题、超链接等。

#### 3. 设计数据库

爬虫的数据库倒是特别简单，一个表即可。这个表里面存着页面的 URL 和爬来的标题以及网页文字内容。

```

CREATE TABLE `pages` (
  `id` int unsigned NOT NULL AUTO_INCREMENT,
  `url` varchar(768) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '网页链接',
  `host` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '域名',
  `dic_done` tinyint DEFAULT '0' COMMENT '已拆分进词典',
  `craw_done` tinyint NOT NULL DEFAULT '0' COMMENT '已爬',
  `craw_time` timestamp NOT NULL DEFAULT '2001-01-01 00:00:00' COMMENT '爬取时刻',
  `origin_title` varchar(2000) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '上级页面超链接文字',
  `referrer_id` int NOT NULL DEFAULT '0' COMMENT '上级页面ID',
  `scheme` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT 'http/https',
  `domain1` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '一级域名后缀',
  `domain2` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '二级域名后缀',
  `path` varchar(2000) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT 'URL 路径',
  `query` varchar(2000) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT 'URL 查询参数',
  `title` varchar(1000) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT NULL COMMENT '页面标题',
  `text` longtext CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci COMMENT '页面文字',
  `created_at` timestamp NOT NULL DEFAULT '2001-01-01 08:00:00' COMMENT '插入时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

```

#### 4. 给爷爬！

爬虫有一个极好的特性：自我增殖。每一个网页里，基本都带有其他网页的链接，这样我们就可以道生一，一生二，二生三，三生万物了。

此时，我们只需要找一个导航网站，手动把该网站的链接插入到数据库里，爬虫就可以开始运作了。各位可以自行挑选可口的页面链接服用。

我们正式进入实操阶段，以下都是可以运行的代码片段，代码逻辑在注释里面讲解。

我采用 `joho/godotenv` 来提供 `.env` 配置文件读取的能力，你需要提前准备好一个 `.env` 文件，并在里面填写好可以使用的 MySQL 数据库信息，具体可以参考项目中的 `.env.example` 文件。

```

func main() {
    fmt.Println("My name id enterprise-search-engine!")

    // 加载 .env
    initENV() // 该函数的具体实现可以参考项目代码

    // 开始爬
    nextStep(time.Now())

    // 阻塞，不跑爬虫时用于阻塞主线程
    select {}
}

// 循环爬
func nextStep(startTime time.Time) {
    // 初始化 gorm 数据库
    dsn0 := os.Getenv("DB_USERNAME0") + ":" +
        os.Getenv("DB_PASSWORD0") + "@(" +
        os.Getenv("DB_HOST0") + ":" +
        os.Getenv("DB_PORT0") + ")/" +
        os.Getenv("DB_DATABASE0") + "?charset=utf8mb4&parseTime=True&loc=Local"
    gormConfig := gorm.Config{}
    db0, _ := gorm.Open(mysql.Open(dsn0), &gormConfig)

    // 从数据库里取出本轮需要爬的 100 条 URL
    var pagesArray []models.Page
    db0.Table("pages").
        Where("craw_done", 0).
        Order("id").Limit(100).Find(&pagesArray)

    tools.DD(pagesArray) // 打印结果

    // 限于篇幅，下面用文字描述
    1\ 循环展开 pagesArray
    2\ 针对每一个 page，使用 curl 工具类获取网页文本
    3\ 解析网页文本，提取出标题和页面中含有的超链接
    4\ 将标题、一级域名后缀、URL 路径、插入时间等信息补充完全，更新到这一行数据上
    5\ 将页面上的超链接插入 pages 表，我们的网页库第一次扩充了！

    fmt.Println("跑完一轮", time.Now().Unix()-startTime.Unix(), "秒")

    nextStep(time.Now()) // 紧接着跑下一条
}

```

id	url	host	dic_done	craw_done	craw_time	origin_title	referrer_id	scheme	domain1	domain2	path	query	title	text	created_at
1	https://www.hao123.com	NULL	0	0	2001-01-01 00:00:00	NULL	0	NULL	NULL	NULL	NULL	NULL	NULL	NULL	2001-01-01 08:00:00

我已经事先将 hao123 的链接插入了 pages 表，所以我运行 `go build -o ese *.go && ./ese` 命令之后，得到了如下信息：

```
My name id enterprise-search-engine!
加载.env : /root/enterprise-search-engine/.env
APP_ENV: local
[[[1 0 https://www.hao123.com 0 0 2001-01-01 00:00:00 +0800 CST 2001-01-01 08:00:00 +0800 CST 0001-01-01 00:00:00 +0000 UTC]]]
```



《递龟》

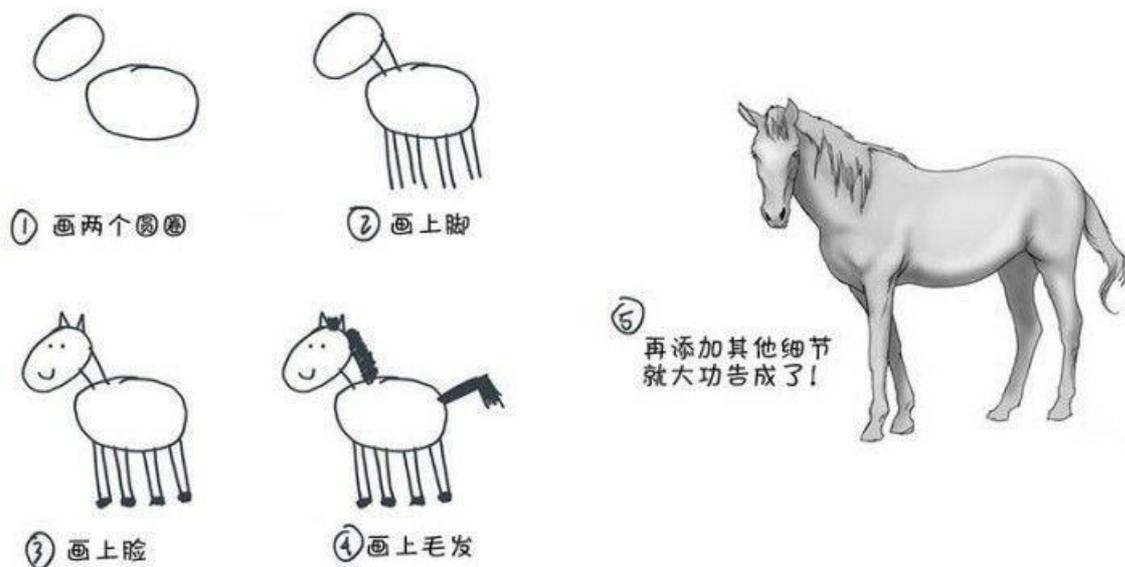
上面的代码中，我们第一次用到了递龟递归：自己调用自己。

## 5. 合法合规：遵守 robots.txt 规范

我选择用 `temoto/robotstxt` 这个库来探查我们的爬虫是否被允许爬取某个 URL，使用一张单独的表来存储每个域名的 robots 规则，并在 Redis 中建立缓存，每次爬取 URL 之前，先进行一次匹配，匹配成功后再爬，保证合法合规。

制造真正的生产级爬虫

# 怎样画马



《怎样画马》

有了前面这个理论上可以运行的简单爬虫，下面我们就要给这匹马补充亿点细节了：生产环境中，爬虫性能优化是最重要的工作。

从某种程度上来说，搜索引擎的优劣并不取决于搜索算法的优劣，因为算法作为一种“特定问题的简便算法”，一家商业公司比别家强的程度很有限，搜索引擎的真正优劣在于哪家能够以最快的速度索引到互联网上层出不穷的新页面和已经更新过内容的旧页面，在于哪家能够识别哪个网页是价值最高的网页。

识别网页价值方面，李彦宏起家的搜索专利，以及谷歌大名鼎鼎的 PageRank 都拥有异曲同工之妙。但本文的重点不在这个领域，而在于技术实现。让我们回到爬虫性能优化，为什么性能优化如此重要呢？我们构建的是互联网搜索引擎，需要爬海量的数据，因此我们的爬虫需要足够高效：中文互联网有 400 万个网站，3500 亿个网页，哪怕只爬千分之一，3.5 亿个网页也不是开玩笑的，如果只是单线程阻塞地爬，消耗的时间恐怕要以年为单位了。

爬虫性能优化，我们首先需要规划一下硬件。

## 硬件要求

首先计算磁盘空间，假设一个页面 20KB，在不进行压缩的情况下，一亿个页面就需要  $20 * 100000000 / 1024 / 1024 / 1024 = 1.86\text{TB}$  的磁盘空间，而我们打算使用 MySQL 来存储页面文本，需要的空间会更大一点。

我的爬虫花费了 2 个月的时间，爬到了大约 1 亿个 URL，其中 3600 万个爬到了页面的 HTML 文本存在了数据库里，共消耗了超过 600GB 的磁盘空间。

除了硬性的磁盘空间，剩下的就是尽量多的 CPU 核心数和内存了：CPU 拿来并发爬网页，内存拿来支撑海量协程的消耗，外加用 Redis 为爬虫提速。爬虫阶段对内存的要求还不大，但在后面第二步拆分字典的时候，大内存容量的 Redis 将成为提速利器。

所以，我们对硬件的需求是这样的：一台核心数尽量多的物理机拿来跑我们的 ese 二进制程序，外加高性能数据库（例如16核64GB内存，NVME磁盘），你能搞到多少台数据库就准备多少台，就算你搞到了 65536 台数据库，也能跑满，理论上我们可以无限分库分表。能这么搞是因为网页数据具有离散性，相互之间的关系在爬虫和字典阶段还不存在，在查询阶段才比较重要。

顺着这个思路，有人可能就会想，我用 KV 数据库例如 MongoDB 来存怎么样呢？当然是很好的，但是 MongoDB 不适合干的事情实在是太多啦，所以你依然需要 Redis 和 MySQL 的支持，如果你需要爬取更大规模的网页，可以把 MongoDB 用起来，利用进一步推高系统复杂度的方式获得一个显著的性能提升。

下面我们开始进行软件优化，我只讲述关键步骤，各位有什么不明白的地方可以参考项目代码。

### 重用 HTTP 客户端以防止内存泄露

这个点看起来很小，但当你瞬间并发数十万协程的时候，每个协程 1MB 的内存浪费累积起来都是巨大的，很容易造成 OOM。

我们在 tools 文件夹下创建 `curl.go` 工具类，专门用来存放全局 `client` 和 `curl` 工具函数：

```
package tools

import ... //省略，具体可以参考项目代码

// 全局重用 client 对象，4 秒超时，不跟随 301 302 跳转
var client = req.C().SetTimeout(time.Second * 4).SetRedirectPolicy(req.NoRedirectPolicy())

// 返回 document 对象和状态码
func Curl(page models.Page, ch chan int) (*goquery.Document, int) {
    ... //省略，具体可以参考项目代码
}
```

### 基础知识储备：Goroutine 协程

我默认你已经了解 go 协程是什么了，它就是一个看起来像魔法的东西。在这里我提供一个理解协程的小诀窍：每个协程在进入磁盘、网络等“只需要后台等待”的任务之后，会把当前 CPU 核心(可以理解成一个图灵机)的指令指针 `goto` 到下一个协程的起始。

需要注意的是，协程是一种特殊的并发形式，你在并发函数内调用的函数必须都支持并发调用，类似于传统的“线程安全”，如果你一不小心写了不安全的代码，轻则卡顿，重则 crash。

一次取出一批需要爬的 URL，使用协程并发爬

协程代码实操来啦！

```

// tools.DD(pagesArray) // 打印结果

// 创建 channel 数组
chs := make([]chan int, len(pagesArray))
// 展开 pagesArray 数组
for k, v := range pagesArray {
// 存储 channel 指针
chs[k] = make(chan int)
// 阿瓦达啃大瓜！！
go crawl(v, chs[k], k)
}

// 注意，下面的代码不可省略，否则你上面 go 出来的那些协程会瞬间退出
var results = make(map[int]int)
for _, ch := range chs {
// 神之一手，收集来自协程的返回数据，并 hold 主线程不瞬间退出
r := <-ch

_, prs := results[r]
if prs {
results[r] += 1
} else {
results[r] = 1
}
}
// 当代码执行到这里的时候，说明所有的协程都已经返回数据了

fmt.Println("跑完一轮", time.Now().Unix()-startTime.Unix(), "秒")

```

`crawl` 函数协程化:

```

// 真的爬，存储标题，内容，以及子链接
func crawl(status models.Page, ch chan int, index int) {
// 调用 CURL 工具类爬到网页
doc, chVal := tools.Curl(status, ch)

// 对 doc 的处理在这里省略

// 最重要的一步，向 channel 发送 int 值，该动作是协程结束的标志
ch <- chVal
return
}

```

协程优化做完了，CPU 被吃满了，接下来数据库要成为瓶颈了。

## MySQL 性能优化

做到这里，在做普通业务逻辑的时候非常快的 MySQL 已经是整个系统中最慢的一环了：pages 表一天就要增加几百万行，MySQL 会以肉眼可见的速度慢下来。我们要对 MySQL 做性能优化。

何以解忧，唯有索引

首先，收益最大的肯定是加索引，这句话适用于 99% 的场景。

在你磁盘容量够用的情况下，加索引通常可以获得数百倍到数万倍的性能提升。我们先给 url 加个索引，因为我们每爬到一个 URL 都要查一下它是否已经在表里面存在了，这个动作的频率是非常高的，如果我们最终爬到了一亿个页面，那这个对比动作至少会做百亿次。

部分场景下很好用的分库分表

非常幸运，爬虫场景和分库分表非常契合：只要我们能根据 URL 将数据均匀地分散开，不同的 URL 之间是没有多少关系的。那我们该怎么将数据分散开呢？使用散列值！

每一个 URL 在 MD5 之后，都会得到一个形如 `698d51a19d8a121ce581499d7b701668` 的 32 位长度的 16 进制数。而这些数字在概率上是均等的，所以理论上我们可以将数亿个 URL 均匀分布在多个库的多个表里。下面问题来了，该怎么分呢？

只有一台数据库，应该分表吗？

如果你看过我的《高并发的哲学原理（八）-- 将 InnoDB 剥的一丝不挂：B+ 树与 Buffer Pool》的话，就会明白，只要你能接受分表的逻辑代价，那在任何大数据量场景下分表都是有明显收益的，因为随着表容量的增加，那棵 16KB 页块组成的 B+ 树的复杂度增加是超线性的，用牛逼的话说就是：二阶导数持续大于 0。此外，缓存也会失效，你的 MySQL 运行速度会降低到一个令人发指的水平。

所以，即便你只有一台数据库，那也应该分表。如果你的磁盘是 NVME，我觉得单机拿出 MD5 的前两位数字，分出来  $16 \times 16 = 256$  个表是比较不错的。

当然，如果你能搞到 16 台数据库服务器，那拿出第一位 16 进制数字选定物理服务器，再用二三位数字给每台机器分 256 个表也是极好的。

我的真实硬件和分表逻辑

由于我司比较节俭贫穷，机房的服务器都是二手的，实在是拿不出高性能的 NVME 服务器，于是我找 IT 借了两台 ThinkBook 14 寸笔记本装上了 CentOS Stream 9：

1. 把内存扩充到最大，形成了 8GB 板载 + 32GB 内存条一共 40GB 的奇葩配置
2. CPU 是 AMD Ryzen 5 5600U，虽然是低压版的 CPU，只有六核十二线程，但是也比 Intel 的渣渣 CPU 快多了（Intel：牙膏真的挤完了，一滴都没有了）
3. 磁盘就用自带的 500GB NVME，实测读写速度能跑到 3GB/2GB，十分够用

由于单台机器只有 6 核，我就各给他们分了 128 个表，在每次要执行 SQL 之前，我会先用 URL 作为参数

获取一下它对应的数据库服务器和表名。表名获取逻辑如下：

1. 计算此 URL 的 MD5 散列值
2. 取前两位十六进制数字
3. 拼接成类似 `pages_0f` 样子的表名

```
tableName := table + "_" + tools.GetMD5Hash(url)[0:2]
```

## 爬虫数据流和架构优化

上面我们已经使用协程把 CPU 全部利用起来了，又使用分库分表技术把数据库硬件全部利用起来了，但是如果你这个时候直接用上面的代码开始跑，会发现速度还是不够快：因为某些工作 MySQL 还是不擅长做。

此时，我们就需要对数据流和架构做出优化了。

### 拆分仓库表和状态表

原始的 `pages` 表有 16 个字段，在我们爬的过程中，只用得到五个：`id url host craw_done craw_time`。而看过我上面的 InnoDB 文章的小伙伴还知道，在页面 HTML 被填充进 `text` 字段之后，`pages` 表的 16 KB 页块会出现频繁的调整和指针的乱飞，对 InnoDB 的“局部性”性能涡轮的施展非常不利，会造成 `buffer pool` 的频繁失效。

所以，为了爬的更快，为 `pages` 表打造一个性能更强的“影子”就十分重要。于是，我为 `pages_0f` 表打造了只包含上面五个字段的 `status_0f` 兄弟表，数据从 `pages` 表里面复制而来，承担一些频繁读写任务：

1. 检查 URL 是否已经在库，即如果以前别的页面上已经出现了这个 URL 了，本次就不需要再入库了
2. 找出下一批需要爬的页面，即 `craw_done=0` 的 URL
3. `craw_time` 承担日志的作用，用于统计过去一段时间的爬虫效率

除了这些高频操作，存储页面 HTML 和标题等信息的低频操作是可以直接入 `pages_0f` 仓库表的。

### 实时读取 URL 改为后台定时读取

随着单表数据量的逐渐提升，每一轮开始时从数据库里面批量读出需要爬的 URL 成了一个相对耗时的操作，即便每张表只需要 500ms，那轮询 256 张表总耗时也达到了 128 秒之多，这是无法接受的，所以这个流程也需要异步化。你问为什么不异步同时读取 256 张表？因为 MySQL 最宝贵的就是连接数，这样会让连接数直接爆掉，大家都别玩了，关于连接数我们下面还会有提及。

我们把流程调整一下：每 20 秒从 `status` 表中搜罗一批需要爬的 URL 放进 Redis 中积累起来，爬的时候直接从 Redis 中读一批。这么做是为了把每一秒的时间都利用起来，尽力填满协程爬虫的胃口。

```

// 在 main() 中注册定时任务
c := cron.New(cron.WithSeconds())
// 每 20 秒执行一次 prepareStatusesBackground 函数
c.AddFunc("* / 20 * * * *", prepareStatusesBackground)
go c.Start()

// prepareStatusesBackground 函数中，使用 LPush 向有序列表的头部插入 URL
for _, v := range _statusArray {
    taskBytes, _ := json.Marshal(v)
    db.Rdb.LPush(db.Ctx, "need_craw_list", taskBytes)
}

// 每一轮都使用 RPop 从有序列表的尾部读取需要爬的 URL
var statusArr []models.Status
maxNumber := 1 // 放大倍数，控制每一批的 URL 数量
for i := 0; i < 256*maxNumber; i++ {
    jsonString := db.Rdb.RPop(db.Ctx, "need_craw_list").Val()
    var _status models.Status
    err := json.Unmarshal([]byte(jsonString), &_status)
    if err != nil {
        continue
    }
    statusArr = append(statusArr, _status)
}

```

## 十分重要的爬虫压力管控

过去十年，中国互联网每次有搜索引擎新秀崛起，我都要被新爬虫 DDOS 一遍，想想就气。这帮大厂的菜鸟程序员，以为随便一个网站都能承受住 2000 QPS，实际上互联网上 99.9% 网站的极限 QPS 到不了 100，超过 10 都够呛。对了，如果有 YisouSpider 的人看到本文，请回去推动一下你们的爬虫优化，虽然你们的爬虫不会持续高速爬取，但是你们在每分钟的第一秒并发 10 个请求的方法更像是 DDOS，对系统的危害更大...

我们要像谷歌那样，做一个压力均匀的文明爬虫，这就需要我们每一个域名的爬虫频率都记录下来，并实时进行调整。我基于 Redis 和每个 URL 的 host 做了一个计数器，在每次真的要爬某个 URL 之前，调用一次检测函数，看是否对单个域名的爬虫压力过大。

此外，由于我们的 `craw` 函数是协程调用的，此时 Redis 就显得更为重要了：它能提供宝贵的“线程安全数据读写”功能，如果你也是 `sync.Map` 的受害者，我相信你一定懂我。

我认为，单线程的 Redis 是 go 协程最佳的伙伴，就像 PHP 和 MySQL 那样。

具体代码我就不放了，有需要的读者可以自己去看项目代码哦。

## 疯狂使用 Redis 加速频繁重复的数据库调用

我们使用协程高速爬到数据了，下一步就是存储这些数据。这个操作看起来很简单，更新一下原来那一行，再插入 N 行新数据不就行了吗，其实不行，还有一个关键步骤需要使用 Redis 来加速：新爬到的 URL 是否已经在数据库里存在了。这个操作看起来简单，但在我们解决了上面这些性能问题以后，庞大的数量就成了这一步最大的问题，每一次查询会越来越慢，查询字数还特别多，这谁顶得住。

如果我们拿 Redis 来存 URL，岂不是需要把所有 URL 都存入 Redis 吗，这内存需求也太大了。这个时候，我们的老朋友，**局部性**又出现了：由于我们的爬虫是按照顺序爬的，那“朋友的朋友也是朋友”的概率是很大的，所以我们只要在 Redis 里记录一下某条 URL 是否存在，那之后一段时间，这个信息被查到的概率也很大：

```
// 我们使用一个 Hash 来存储 URL 是否存在的状态
statusHashMapKey := "ese_spider_status_exist"
statusExist := db.Rdb.HExists(db.Ctx, statusHashMapKey, _url).Val()
// 若 HashMap 中不存在，则查询或插入数据库
if !statusExist {
    ... 代码省略，不存在则创建这行 page，存在则更新信息 ...
    // 无论是否新插入了数据，都将 _url 入 HashMap
    db.Rdb.HSet(db.Ctx, statusHashMapKey, _url, 1).Err()
}
```

这段代码看似简单，实测非常好用，唯一的问题就是不能运行太长时间，隔一段时间得清空一次，因为随着时间的流逝，局部性会越来越差。

细心的小伙伴可能已经发现了，既然爬取状态已经用 Redis 来承载了，那还需要区分 pages 和 status 表吗？需要，因为 Redis 也不是全能的，它的基础数据依然是来自 MySQL 的。目前这个架构类似于复杂的三级火箭，看起来提升没那么大，但这小小的提速可能就能让你爬三亿个网页的时间从 3 个月缩减到 1 个月，是非常值的。

另外，如果通过扫描 256 张表中 `craw_time` 字段的方式来统计“过去 N 分钟爬了多少个 URL、有效页面多少个、因为爬虫压力而略过的页面多少个、网络错误的多少个、多次网络错误后不再重复爬取的多少个”的数据，还是太慢了，也太消耗资源了，这些统计信息也需要使用 Redis 来记录：

```
// 过去一分钟爬到了多少个页面的 HTML
allStatusKey := "ese_spider_all_status_in_minute_" + strconv.Itoa(int(time.Now().Unix())/60)
// 计数器加 1
db.Rdb.IncrBy(db.Ctx, allStatusKey, 1).Err()
// 续命 1 小时
db.Rdb.Expire(db.Ctx, allStatusKey, time.Hour).Err()

// 过去一分钟从新爬到的 HTML 里面提取出了多少个新的待爬 URL
newStatusKey := "ese_spider_new_status_in_minute_" + strconv.Itoa(int(time.Now().Unix())/60)
// 计数器加 1
db.Rdb.IncrBy(db.Ctx, newStatusKey, 1).Err()
// 续命 1 小时
db.Rdb.Expire(db.Ctx, newStatusKey, time.Hour).Err()
```

## 生产爬虫遇到的其他问题

在我们不断提高爬虫速度的过程中，爬虫的复杂度也在持续上升，我们会遇到玩具爬虫遇不到的很多问题，接下来我分享一下我的处理经验。

### 抑制暴增的数据库连接数

在协程这个大杀器的协助之下，我们可以轻易写出超高并行的代码，把 CPU 全部吃完，但是，并行的协程多了以后，数据库的连接数压力也开始暴增。MySQL 默认的最大连接数只有 151，根据我的实际体验，哪怕是一个协程一个连接，我们这个爬虫也可以轻易把连接数干到数万，这个数字太大了，即便是最新的 CPU 加上 DDR5 内存，受制于 MySQL 算法的限制，在连接数达到这个级别以后，处理海量连接数所需要的时间也越来越多。这个情况和《高并发的哲学原理（二）-- Apache 的性能瓶颈与 Nginx 的性能优势》[一文](#)中描述的 Apache 的 prefork 模式比较像。好消息是，最近版本的 MySQL 8 针对连接数匹配算法做了优化，大幅提升了大量连接数下的性能。

除了协程之外，分库分表对连接数的暴增也负有不可推卸的责任。为了提升单条 SQL 的性能，我们给单台数据库服务器分了 256 张表，这种情况下，以前的一个连接+一条 SQL 的状态会突然增加到 256 个连接和 256 条 SQL，如果我们不加以限制的话，可以说协程+分表一启动，你就一定会收到海量的 `Too many connections` 报错。我的解决方法是，在 gorm 初始化的时候，给他设定一个“单线程最大连接数”：

```
dbdb0, _ := _db0.DB()
dbdb0.SetMaxIdleConns(1)
dbdb0.SetMaxOpenConns(100)
dbdb0.SetConnMaxLifetime(time.Hour)
```

根据我的经验，100 个够用了，再大的话，你的 TCP 端口就要不够用了。

### 域名黑名单

我们都知道，内容农场是一种专门钻搜索引擎空子的垃圾内容生产者，爬虫很难判断哪些网站是内容农场，但是人一点进去就能判断出来。而这些域名的内部链接做的又特别好，这就导致我们需要手动给一些恶心的

内容农场域名加黑名单。我们把爬到的每个域名下的 URL 数量统计一下，搞一个动态的排名，就能很容易发现头部的内容农场域名了。

## 复杂的失败处理策略

生产代码和教学代码最大的区别就是成吨的错误处理！—— John · Lui（作者自己）

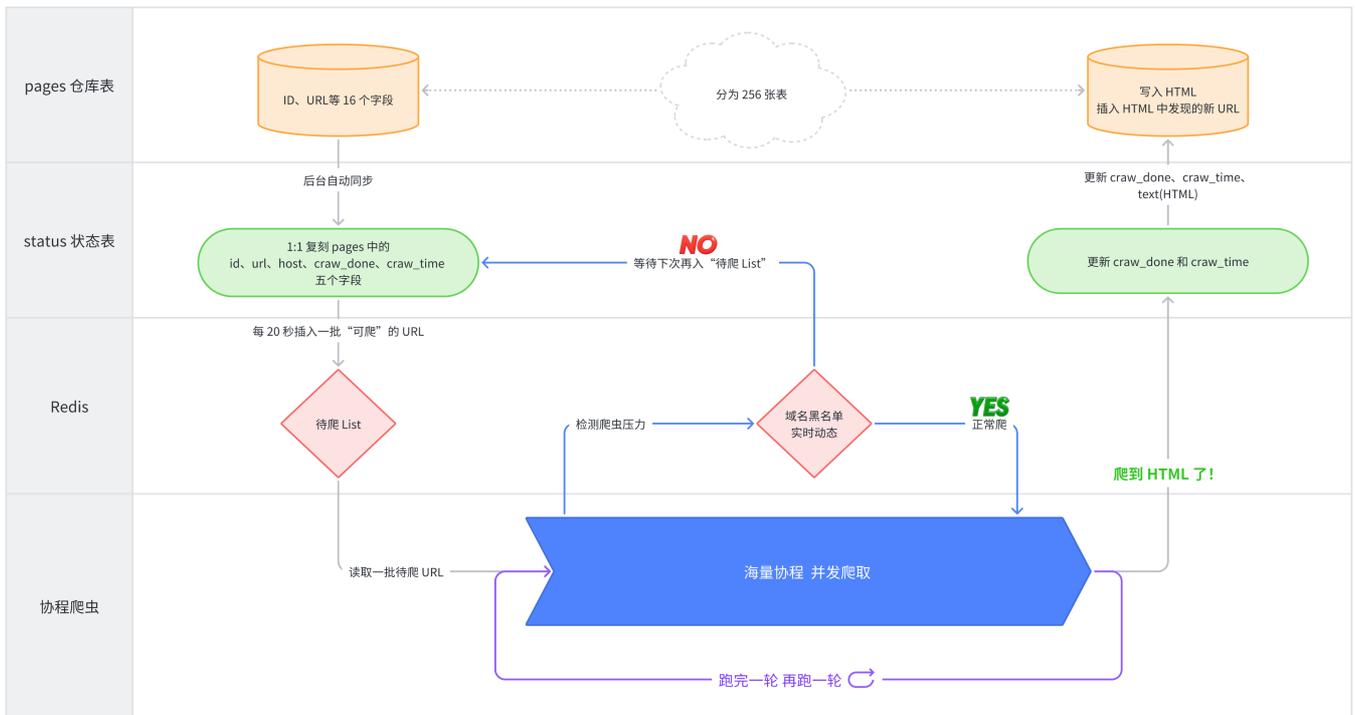
如果你真的要搞一个涵盖数亿页面的可以用的搜索引擎，你会碰到各种各样的奇葩失败，这些失败都需要拿出特别的处理策略，下面我分享一下我遇到过的问题和我的处理策略。

1. 单页面超时非常重要：如果你想尽可能地在一段时间内爬到尽量多的页面的话，缩短你 curl 的超时时间非常重要，经过摸索，我把这个时间设定到了 4 秒，既能爬到绝大多数网页，也不会浪费时间在一些根本就无法响应的 URL 上。
2. 单个 URL 错误达到一定数量以后，需要直接拉黑，不然一段时间后，你的爬虫整天就只爬那些被无数次爬取失败的 URL 上啦，一个新页面也爬不到。这个次数我设定的是 3 次。
3. 如果某个 URL 返回的 HTML 无法被解析，果断放弃，没必要花费额外资源重新爬。
4. 由于我们的数据流已经是三级火箭形态，所以在各种地方加上“动态锁”就很必要，因为很多时候我们需要手动让其他级火箭发动机暂停运行，手动检修某一级发动机。我一般拿 MySQL 来做这件事，创建一个名为 `kvstores` 的表，只有 key value 两个字段，需要的时候我会手动修改特定 key 对应的 value 值，让某一级发动机暂停一下。
5. 由于 curl 的结果具有不确定性，务必需要保证任何情况下，都要给 channel 返回信号量，不然你的整个应用会直接卡死。
6. 一个页面内经常会有同一个超链接重复出现，在内存里保存已经见过的 URL 并跳过重复值可以显著节约时间。
7. 我建了一个 MySQL 表来存储我手动插入的黑名单域名，这个非常好用，可以在爬虫持续运行的时候随时“止损”，停止对黑名单域名的新增爬取。

至此，我们的爬虫终于构建完成了。

## 爬虫运行架构图

现在我们的爬虫运行架构图应该是下面这样的：



爬虫搞完了，让我们进入第二大部分。

## 第二步，使用倒排索引生成字典

那个男人一听就很牛逼的词出现了：倒排索引。

对于没搞过倒排索引的人来说，这个词听起来和“生态化反”一样牛逼，其实它非常简单，简单程度堪比 HTTP 协议。

### 倒排索引到底是什么

下面这个例子可以解释倒排索引是个什么东西：

1. 我们有一个表 titles，含有两个字段，ID 和 text，假设这个表有 100 行数据，其中第一行 text 为“爬虫工作流程”，第二行为“制造真正的生产级爬虫”
2. 我们对这两行文本进行分词，第一行可以得到“爬虫”、“工作”、“流程”三个词，第二行可以得到“制造”、“真正的”、“生产级”、“爬虫”四个词
3. 我们把顺序颠倒过来，以词为 key，以① titles.id ②，③ 这个词在 text 中的位置 这三个元素拼接在一起为一个值，不同 text 生成的值之间以 - 作为间隔，对数据进行“反向索引”，可以得到：
  1. 爬虫: 1,0-2,8
  2. 工作: 1,2
  3. 流程: 1,4
  4. 制造: 2,0
  5. 真正的: 2,2
  6. 生产级: 2,5

倒排索引完成了！就是这么简单。说白了，就是把所有内容都分词出来，再反向给每个词标记出“他出现在

哪个文本的哪个位置”，没了，就是这么简单。下面是我生成的字典中，“辰玺”这个词的字典值：

```
110,85,1,195653,7101-66,111,1,195653,7101-
```

你问为什么我不找个常见的词？因为随便一个常见的词，它的字典长度都是以 MB 为单位的，根本没法放出来...

还有一个牛逼的词，最小完美哈希，可以用来排布字典数据，加快搜索速度，感兴趣的同学可以自行学习

## 生成倒排索引数据

理解了倒排索引是什么以后，我们就可以着手把我们爬到的 HTML 处理成倒排索引了。

我使用 [yanyiwu/gojieba](#) 这个库来调用结巴分词，按照以下步骤对我爬到的每一个 HTML 文本进行分词并归类：

1. 分词，然后循环处理这些词：
2. 统计词频：这个词在该 HTML 中出现的次数
3. 记录下这个词在该 HTML 中每一次出现的位置，从 0 开始算
4. 计算该 HTML 的总长度，搜索算法需要
5. 按照一定格式，组装成倒排索引值，形式如下：

```
// 分表的顺序，例如 0f 转为十进制为 15 strconv.Itoa(i) + "," + // pages.id 该 URL 的主键 ID strconv.Itoa(int(pages.ID)) + "," + // 词频：这个词在该 HTML 中出现的次数 strconv.Itoa(v.count) + "," + // 该 HTML 的总长度，BM25 算法需要 strconv.Itoa(textLength) + "," + // 这个词出现的每一个位置，用逗号隔开，可能有多个 strings.Join(v.positions, ",") + // 不同 page 之间的间隔符 "-"
```

我们按照这个规则，把所有的 HTML 进行倒排索引，并且把生成的索引值拼接在一起，存入 MySQL 即可。

## 使用协程 + Redis 大幅提升词典生成速度

不知道大家感受到了没有，词典的生成是一个比爬虫高几个数量级的 CPU 消耗大户，一个 HTML 动辄几千个词，如果你要对数亿个 HTML 进行倒排索引，需要的计算量是非常惊人的。我爬到了 3600 万个页面，但是只处理了不到 800 万个页面的倒排索引，因为我的计算资源也有限...

并且，把词典的内容存到 MySQL 里难度也很大，因为一些常见词的倒排索引会巨长，例如“没有”这个词，真的是到处都有它。那该怎么做性能优化呢？还是我们的老朋友，协程和 Redis。

## 协程分词

两个 HTML 的分词工作之间完全没有交集，非常适合拿协程来跑。

但是，MySQL 举手了：我顶不住。所以协程的好朋友 Redis 也来了。

使用 **Redis** 做为词典数据的中转站

我们在 Redis 中针对每一个词生成一个 List，把倒排出来的索引插入到尾部：

```
db.Rdb10.RPush(db.Ctx, word, appendString)
```

使用协程从 **Redis** 搬运数据到 **MySQL** 中

你没看错，这个地方也需要使用协程，因为数据量实在是太大了，一个线程循环跑会非常慢。经过我的不断尝试，我发现每次转移 2000 个词，对 Redis 的负载比较能够接受，E5-V4 的 CPU 单核能够跑满，带宽大概 400Mbps。

从 Redis 到 MySQL 的高性能搬运方法如下：

1. 随机获取一个 key
2. 判断该 key 的长度，只有大于等于 2 的进入下一步
3. 把最后一个索引值留下，前面的元素一个一个 **LPop**（弹出头部）出来，拼接在一起
4. 汇集一批 2000 个随机词的结果，append 到数据库该词现有索引值的后面

有了协程和 Redis 的协助，分词加倒排索引的速度快了起来，但是如果你选择一个词一个词地 append 值，你会发现 MySQL 又双叟叟变的超慢，又要优化 MySQL 了！

事务的妙用：**MySQL** 高速批量插入

由于需要往磁盘里写东西，所以只要是一个一个 update，怎么优化都会很慢，那有没有一次性 update 多行数据的方法呢？有！那就是事务：

```
tx.Exec(`START TRANSACTION`)
```

```
// 需要批量执行的 update 语句
```

```
for w, s := range needUpdate {
```

```
    tx.Exec(`UPDATE word_dics SET positions = concat(ifnull(positions,''), ?) where name = ?`, s
```

```
, w)
```

```
}
```

```
tx.Exec(`COMMIT`)
```

这么操作，字典写入速度一下子起来了。但是，每次执行 2000 条 update 语句对磁盘的要求非常高，我进行这个操作的时候，可以把磁盘写入速度瞬间提升到 1.5GB/S，如果你的数据库存储不够快，可以减少语句数量。

世界的参差：无意义的词

这个世界上的东西并不都是有用的，一个 HTML 中的字符也是如此。

首先，一般不建议索引整个 HTML，而是把他用 DOM 库处理一下，提取出文本内容，再进行索引。

其次，即便是你已经过滤掉了所有的 html 标签、css、js 代码等，还是有些词频繁地出现：它们出现的频率如此的高，以至于反而失去了作为搜索词的价值。这个时候，我们就需要把他们狠狠地拉黑，不处理他们的倒排索引。我使用的黑名单词如下，表名为 `word_black_list`，只有两个字段 `id`、`word`，需要的自取：

```
INSERT INTO `word_black_list` (`id`, `word`)
```

```
VALUES
```

```
(1, 'px'),
(2, '20'),
(3, '('),
(4, ')'),
(5, ','),
(6, '.'),
(7, '-'),
(8, '/'),
(9, ':'),
(10, 'var'),
(11, '的'),
(12, 'com'),
(13, ';'),
(14, '['),
(15, ']'),
(16, '{'),
(17, '}'),
(18, '\\'),
(19, '\"'),
(20, '_'),
(21, '?'),
(22, 'function'),
(23, 'document'),
(24, '|'),
(25, '='),
(26, 'html'),
(27, '内容'),
(28, '0'),
(29, '1'),
(30, '3'),
(31, 'https'),
(32, 'http'),
(33, '2'),
(34, '!'),
(35, 'window'),
(36, 'if'),
(37, '\"'),
(38, '\"'),
(39, '。'),
(40, 'src'),
(41, '由\')
```

```
(41, ' '),  
(42, '了'),  
(43, '6'),  
(44, '.'),  
(45, '<'),  
(46, '>'),  
(47, '联系'),  
(48, '号'),  
(49, 'getElementsByTagName'),  
(50, '5'),  
(51, '.'),  
(52, 'script'),  
(53, 'js');
```

至此，字典的处理告一段落，下面让我们一起 Just 搜 it!

### 第三步，使用 BM25 算法给出搜索结果

网上关于 BM25 算法的文章是不是看起来都有点懵？别担心，看完下面这段文字，我保证你能自己写出来这个算法的具体实现，这种有具体文档的工作是最好做的了，比前面的性能优化简单多了。

#### 简单介绍一下 BM25 算法

BM25 算法是现代搜索引擎的基础，它可以很好地反映一个词和一堆文本的相关性。它拥有不少独特的设计思想，我们下面会详细解释。

这个算法第一次被生产系统使用是在 1980 年代的伦敦城市大学，在一个名为 Okapi 的信息检索系统中被实现出来，而原型算法来自 1970 年代 Stephen E. Robertson、Karen Spärck Jones 和他们的同伴开发的概率检索框架。所以这个算法也叫 Okapi BM25，这里的 BM 代表的是 `best matching`（最佳匹配），非常实在，和比亚迪的“美梦成真”有的一拼（Build Your Dreams）

#### 详细讲解 BM25 算法数学表达式的含义

$$Score(Query, D_i) = \sum_i^n W_i \cdot R(Q_i, D_j)$$

我简单描述一下这个算法的含义。

首先，假设我们有 100 个页面，并且已经对他们分词，并全部生成了倒排索引。此时，我们需要搜索这句话“BM25 算法的数学描述”，我们就需要按照以下步骤来计算：

1. 对“BM25 算法的数学描述”进行分词，得到“BM25”、“算法”、“的”、“数学”、“描述”五个词
2. 拿出这五个词的全部字典信息，假设包含这五个词的页面一共有 50 个
3. 逐个计算这五个词和这 50 个页面的 [相关性权重](#) 和 [相关性得分](#) 的乘积（当然，不是每个词都出现在了这 50 个网页中，有多少算多少）
4. 把这 50 页面的分数分别求和，再倒序排列，即可以获得“BM25 算法的数学描述”这句话在这 100 个页面中的搜索结果

[相关性权重](#) 和 [相关性得分](#) 名字相似，别搞混了，它们的具体定义如下：

某个词和包含它的某个页面的“相关性权重”

$$W_i$$

上图中的  $W_i$  指代的就是相关性权重，最常用的是 [TF-IDF](#) 算法中的 [IDF](#) 权重计算法：

$$IDF(q_i) = \log \frac{N - n(Q_i) + 0.5}{n(Q_i) + 0.5}$$

其中：

- $N$  表示待检索文档的数目
- $n(Q_i)$  表示包含  $Q_i$  的文档数

这里的  $N$  指的是页面总数，就是你已经加入字典的页面数量，需要动态扫描 MySQL 字典，对我来说就是 784 万。而  $n(Q_i)$  就是这个词的字典长度，就是含有这个词的页面有多少个，就是我们字典值中  $Q_i$  出现的次数。

这个参数的现实意义是：如果一个词在很多页面里面都出现了，那说明这个词不重要，例如百分百空手接白刃的“的”字，哪个页面都有，说明这个词不准确，进而它就不重要。

词以稀为贵。

我的代码实现如下：

```
// 页面总数
db.DbInstance0.Raw("select count(*) from pages_0f where dic_done = 1").Scan(&N)
N *= 256

// 字典的值中`-`出现的次数
NQi := len(partsArr)

// 得出相关性权重
IDF := math.Log10((float64(N-NQi) + 0.5) / (float64(NQi) + 0.5))
```

某个词和包含它的某个页面的“相关性得分”

$$R(Q_i, D_j) = \frac{f_i \cdot (k_1 + 1)}{f_i + K} \cdot \frac{qf_i(k_2 + 1)}{qf_i + k_2}$$

$$K = k_1 \cdot (1 - b + b \cdot \frac{D_j^l}{avgD^l})$$

这里引入多个变量，其中：

- $k_1, k_2, b$  为调节因子，根据经验通常为  $k_1 = 2, b = 0.75$
- $f_i$  表示  $Q_i$  在  $D_j$  中出现的频率（次数）
- $qf_i$  表示  $Q_i$  在  $Query$  中出现的频率（次数）
- $D_j^l$  表示文档  $D_j$  长度
- $avgD^l$  表示所有文档的平均长度

这个表达式看起来是不是很复杂，但是它的复杂度是为了处理查询语句里面某一个关键词出现了多次的情况，例如“八百标兵奔北坡，炮兵并排北边跑。炮兵怕把标兵碰，标兵怕碰炮兵炮。”，“炮兵”这个词出现了 3 次。为了能快速实现一个能用的搜索引擎，我们放弃支持这种情况，然后这个看起来就刺激的表达式就可以简化成下面这种形式：

$$R(Q_i, D_j) = \frac{f_i \cdot (k_1 + 1)}{f_i + K}$$

需要注意的是，这里面的大写的 K 依然是上面那个略微复杂的样式。我们取  $k_1$  为 2， $b$  为 0.75，页面（文

档) 平均长度我自己跑了一个, 13214, 你们可以用我这个数, 也可以自己跑一个用。

我的代码实现如下:

```
// 使用 - 切分后的值, 为此页面的字典值, 形式为 :  
// 110,85,1,195653,7101  
ints := strings.Split(p, ",")  
  
// 这个词在这个页面中出现总次数  
Fi, err := strconv.Atoi(ints[2])  
// 这个页面的长度  
Dj, _ := strconv.Atoi(ints[3])  
  
k1 := 2.0  
b := 0.75  
  
// 页面平均长度  
avgDocLength := 13214.0  
  
// 得到相关性得分  
RqiDj := (float64(Fi) * (k1 + 1)) / (float64(Fi) + k1*(1-b+b*(float64(Dj)/avgDocLength)))
```

怎么样, 是不是比你想象的简单?

检验搜索结果

我在我搞的“翰哥搜索”页面上搜了一下“BM25 算法的数学描述”, 结果如下:

# 翰哥搜索

查询耗时: 2.761336432s 总已爬页面数: 7841792

2023-07-04 13:09:04

BM25 算法的数学描述

翰哥一下

## TF-IDF与余弦相似性的应用（一）：自动提取关键词---阮一峰的网络日志

本站显示不正常，可能因为您使用了广告拦截器。您使用了广告拦截器，导致本站内容无法显示。请将加入白名单，解除广告屏蔽后，刷新页面。谢谢。与余弦相似性的应用（一）：自动提取关键词阮一峰的网络日志阮一峰的

## 算法\_算法资讯\_算法最新信息\_雷峰网

算法算法资讯算法最新信息雷峰网百度统计开始百度统计结束英鹏账号登录回调您正在使用低版浏览器，为了您的雷峰网账号安全和更好的产品体验，强烈建议使用更快更安全的浏览器研习社雷峰网公开课活动中心专题爱搞机

## 构造二叉树系列-I-lucifer的网络博客

构造二叉树系列的网络博客力扣加加主页官网友链联系我的网络博客回到主页分类标签归档近期文章文章归档关于小站构造二叉树系列二叉树字数统计字阅读时长≈分构造二叉树是一个常见的二叉树考点，相比于直接考察二叉

我搜索“住范儿”，结果如下：

住范儿

翰哥一下

## 上海|成都|北京装修网-住范儿

上海成都北京装修网住范儿首页装修产品整装系整装系环保辅材升级老房翻新装修案例设计师装修攻略  
装修前收房,预算,设计,合同,装修中拆改,水电,防水,木工,油漆,泥瓦,装修后软装,入住,家具,电器,装修学

## 装修美图欣赏\_家装美图大全\_住范儿

装修美图欣赏家装美图大全住范儿首页装修产品整装系整装系环保辅材升级老房翻新装修案例设计师装  
修攻略装修前收房,预算,设计,合同,装修中拆改,水电,防水,木工,油漆,泥瓦,装修后软装,入住,家具,电器,

## 打造“中国版Home-Depot”，住范儿新开2万m<sup>2</sup>超级家居MALL-|投中网

打造“中国版”，住范儿新开万m<sup>2</sup>超级家居投中网首页研究院榜单会议投中社区搜索公众号矩阵超越曲线  
燃点新消费氢元子象三一智识研习社东四十条资本投中网登录注册个人中心退出登录快捷登录密码登录  
新用户注册验证即

## 试水家居新零售，住范儿要打造“人货场”的零售新模式\_腾讯家居·贝壳

试水家居新零售，住范儿要打造“人货场”的零售新模式腾讯家居·贝壳首页新闻中心家居案例灵感设计  
潮互动超级优居装修优居研究院中国人的家全国热门搜索新闻中心潮互动新闻中心家居案例灵感设计优  
居研究院超级搜索

第一个就是我们官网，可以说相当精准了。

看起来效果还不错，要知道这只是在 784 万的网页中搜索的结果哦，如果你有足够的服务器资源，能搞定三  
亿个页面的爬取、索引和查询的话，效果肯定更加的好。

如何继续提升搜索准确性？

目前我们的简化版 BM25 算法的搜索结果已经达到能用的水平了，还能继续提升搜索准确性吗？还可以：

1. 本文全部是基于分词做的字典，你可以再做一份基于单字的，然后把单字的搜索结果排序和分词的搜索结果进行结合，搜索结果可以更准。
2. 相似的原理，打造更加合理、更加丰富的分词方式，构造不同倾向的词典，可以提升特定领域的搜索结果表现，例如医学领域、代码领域等。
3. 打造你自己的 PageRank 技术，从 URL 之间关系的角度，给单个 URL 的价值进行打分，并将这个价值分数放进搜索结果的排序参数之中。
4. 引入 proximity 相似性计算，不仅考虑精确匹配的关键词，还要考虑到含义相近的关键词的搜索结果。
5. 特殊查询的处理：修正用户可能的输入错误，处理中文独特的“拼音匹配”需求等。

参考资料

1. 【NLP】非监督文本匹配算法——BM25 <https://zhuatlan.zhihu.com/p/499906089>

## 2018 年

### 性能之殇

#### (一) 天才冯·诺依曼与冯·诺依曼瓶颈

电子计算机与信息技术是最近几十年人类科技发展最快的领域，无可争议地改变了每个人的生活：从生活方式到战争方式，从烹饪方式到国家治理方式，都被计算机和信息技术彻底地改变了。如果说核武器彻底改变了国与国之间相处的模式，那么计算机与信息技术则彻底改变了人类这个物种本身，人类的进化也进入了一个新的阶段。

简单地说，生物进化之前还有化学进化。然而细胞一经诞生，中心法则的分子进化就趋于停滞了：38亿年来，中心法则再没有新的变动，所有的蛋白质都由 20 种标准氨基酸连成，连碱基与氨基酸对应关系也沿袭至今，所有现代生物共用一套标准遗传密码。正如中心法则是化学进化的产物，却因为开创了生物进化而停止了化学进化，人类是生物进化的产物，也因为开创了文化进化和技术进化而停止了生物进化——进化已经走上了更高的维度。

-- [《进化的阶次 | 混乱博物馆》](#)

#### 本文目标

上面的只是我的喃喃私语，下面我们进入正题。

本文的目标是在我有限的认知范围内，讨论一下人们为了提高性能做出的种种努力，这里面包含硬件层面的 CPU、RAM、磁盘，操作系统层面的并发、并行、事件驱动，软件层面的多进程、多线程，网络层面的分布式，等等等等。事实上，上述名词并不局限于某一个层面，计算机从 CPU 内的门电路到显示器上浏览器中的某行字，是层层协作才得以实现的；计算机科学中的许多概念，都跨越了层级：事件驱动就是 CPU 和操作系统协作完成的。

#### 天才 冯·诺依曼

冯·诺依曼 1903 年 12 月 28 日出生于奥匈帝国布达佩斯，1957 年 2 月 8 日卒于美国，终年 53 岁。在他短暂的一生中，他取得了巨大的成就，远不止于世人熟知的“冯·诺依曼架构”。

约翰·冯·诺伊曼，出生于匈牙利的美国籍犹太人数学家，现代电子计算机与博弈论的重要创始人，在泛函分析、遍历理论、几何学、拓扑学和数值分析等众多数学领域及计算机学、量子力学和经济学中都有重大贡献。

-- [约翰·冯·诺伊曼的维基百科](#)

除了对计算机科学的贡献，他还有一个称号不被大众所熟知：“博弈论之父”。博弈论被认为是20世纪经济学最伟大的成果之一。(说到博弈论，我相信大多数人第一个想到的肯定跟我一样，那就是“纳什均衡”)

## 冯·诺依曼架构

冯·诺依曼由于在曼哈顿工程中需要大量的运算，从而使用了当时最先进的两台计算机 Mark I 和 ENIAC，在使用 Mark I 和 ENIAC 的过程中，他意识到了存储程序的重要性，从而提出了存储程序逻辑架构。

“冯·诺依曼架构”定义如下：

1. 以运算单元为中心
2. 采用存储程序原理
3. 存储器是按地址访问、线性编址的空间
4. 控制流由指令流产生
5. 指令由操作码和地址码组成
6. 数据以二进制编码

## 优势

冯·诺依曼架构第一次将存储器和运算器分开，指令和数据均放置于存储器中，为计算机的通用性奠定了基础。虽然在规范中计算单元依然是核心，但冯·诺依曼架构事实上导致了以存储器为核心的现代计算机的诞生。

注：请各位在心里明确一件事情：存储器指的是内存，即 **RAM**。磁盘理论上属于输入输出设备。

该架构的另一项重要贡献是用二进制取代十进制，大幅降低了运算电路的复杂度。这为晶体管时代超大规模集成电路的诞生提供了最重要的基础，让我们实现了今天手腕上的 Apple Watch 运算性能远超早期大型计算机的壮举，这也是摩尔定律得以实现的基础。

## 冯·诺伊曼瓶颈

冯·诺依曼架构为计算机大提速铺平了道路，却也埋下了一个隐患：在内存容量指数级提升以后，CPU 和内存之间的数据传输带宽成为了瓶颈。

	Read	Write	Copy	Latency
Memory	89146 MB/s	77873 MB/s	76870 MB/s	68.1 ns
L1 Cache	3754.3 GB/s	1895.8 GB/s	3758.5 GB/s	1.0 ns
L2 Cache	1648.0 GB/s	1059.4 GB/s	1559.7 GB/s	5.0 ns
L3 Cache	199.09 GB/s	95259 MB/s	127.67 GB/s	21.3 ns
CPU Type	18-Core Intel Core i9-7980XE (Skylake-X, LGA2066)			
CPU Stepping	H0/M0/U0			
CPU Clock	4000.1 MHz (original: 2600 MHz, overclock: 53%)			
CPU FSB	100.0 MHz (original: 100 MHz)			
CPU Multiplier	40x	North Bridge Clock	1000.0 MHz	
Memory Bus	1600.0 MHz	DRAM:FSB Ratio	48:3	
Memory Type	Quad Channel DDR4-3200 SDRAM (14-14-14-34 CR2)			
Chipset	Intel Union Point X299, Intel Skylake-X			
Motherboard	Gigabyte GA-X299 Aorus Gaming 7			
BIOS Version	F7			

AIDA64 v5.92.4376 Beta / BenchDLL 4.3.759-x64 (c) 1995-2017 FinalWire Ltd.

上图是 i9-7980XE 18 核 36 线程的民用最强 CPU，其配合超频过的 DDR4 3200MHz 的内存，测试出的内存读取速度为 90GB/S。看起来很快了是不是？看看图中的 L1 Cache，3.7TB/S。

我们再来算算时间。这颗 CPU 最大睿频 4.4GHz，就是说 CPU 执行一个指令需要的时间是 0.000000000 227273 秒，即 0.22ns（纳秒），而内存的延迟是 68.1ns。换句话说，只要去内存里取一个字节，就需要 CPU 等待 300 个周期，何其的浪费 CPU 的时间啊。

CPU L1 L2 L3 三级缓存是使用和 CPU 同样的 14 纳米工艺制造的硅半导体，每一个 bit 都使用六个场效应管（通俗解释成三极管）构成，成本高昂且非常占用 CPU 核心面积，故不能做成很大容量。

除此之外，L1 L2 L3 三级缓存对计算机速度的提升来源于计算机内存的“局部性”，相关内容我们之后会专门讨论。

## （二）分支预测、流水线与多核 CPU

CPU 硬件为了提高性能，逐步发展出了指令流水线（分支预测）和多核 CPU，本文我们就将简单地探讨一下它们的原理和效果。

### 指令流水线

在一台纯粹的图灵机中，指令是一个一个顺序执行的。而现实世界的通用计算机所用的很多基础算法都是可以并行的，例如加法器和乘法器，它们可以很容易地被切分成可以同时运行的多个指令，这样就可以大幅提升性能。

指令流水线，说白了就是 CPU 电路层面的并发。

Intel Core i7 自 Sandy Bridge (2010) 架构以来一直都是 14 级流水线设计。基于 Cedar Mill 架构的最后一代奔腾4，在 2006 年就拥有 3.8GHz 的超高频率，却因为其长达 31 级的流水线而成了样子货，被 AMD 1GHz 的芯片按在地上摩擦。

### RISC机器的五层流水线示意图

下图形象的展示了流水线如何提高性能的。



### 缺点

指令流水线通过硬件层面的并发来提高性能，却也带来了一些无法避免的缺点。

1. 设计难度高，一不小心就成为了高频低能的奔四
2. 并发导致每一条指令的执行时间变长
3. 优化难度大，有时候两行代码的顺序变动就可能导致数倍的性能差异，这对编译器提出了更高的要求
4. 如果多次分支预测失败，会导致严重的性能损失

### 分支预测

指令形成流水线以后，就需要一种高效的调控来保证硬件层面并发的效果：最佳情况是每条流水线里的十几个指令都是正确的，这样完全不浪费时钟周期。而分支预测就是干这个的：

分支预测器猜测条件表达式两路分支中哪一路最可能发生，然后推测执行这一路的指令，来避免流水线停顿造成的时间浪费。但是，如果后来发现分支预测错误，那么流水线中推测执行的那些中间结果全部放弃，重新获取正确的分支路线上的指令开始执行，这就带来了十几个时钟周期的延迟，这个时候，这个 CPU 核心就是完全在浪费时间。

幸运的是，当下的主流 CPU 在现代编译器的配合下，把这项工作做得越来越好了。

还记得那个让 Intel CPU 性能跌 30% 的漏洞补丁吗，那个漏洞就是 CPU 设计的时候，分支预测设计的不完善导致的。

### 多核 CPU

多核 CPU 的每一个核心拥有自己独立的运算单元、寄存器、一级缓存、二级缓存，所有核心共用同一条内存总线，同一段内存。

多核 CPU 的出现，标志着人类的集成电路工艺遇到了一个严酷的瓶颈，没法再大规模提升单核性能，只能使用多个核心来聊以自慰。实际上，多核 CPU 性能的提升极其有限，远不如增加一点点单核频率提升的性能多。

## 优势

多核 CPU 的优势很明显，就是可以并行地执行多个图灵机，可以显而易见地提升性能。只不过由于使用同一条内存总线，实际带来的效果有限，并且需要操作系统和编译器的密切配合才行。

题外话：AMD64 技术可以运行 32 位的操作系统和应用程序，所用的方法是依旧使用 32 位宽的内存总线，每计算一次要取两次内存，性能提升也非常有限，不过好处就是可以使用大于 4GB 的内存了。大家应该都没忘记第一篇文章中提到的冯·诺依曼架构拥有 CPU 和内存通信带宽不足的弱点。（注：AMD64 技术是和 Intel 交叉授权的专利，i7 也是这么设计的）

## 劣势

多核 CPU 劣势其实更加明显，但是人类也没有办法，谁不想用 20GHz 的 CPU 呢，谁想用这八核的 i7 呀。

1. 内存读写效率不变，甚至有降低的风险
2. 操作系统复杂度提升很多倍，计算资源的管理复杂了太多了
3. 依赖操作系统的进步：微软以肉眼可见的速度，在这十几年间大幅提升了 Windows 的多核效率和安全性：XP 只是能利用，7 可以自动调配一个进程在多个核心上游走，2008R2 解决了依赖 CPU0 调度导致死机的 bug（中国的银行提的 bug 哦），8 可以利用多核心启动，10 优化了杀进程依赖 CPU0 的问题。

## 超线程技术

Intel 的超线程技术是将 CPU 核心内部再分出两个逻辑核心，只增加了 5% 的裸面积，就带来了 15%~30% 的性能提升。

## 怀念过去

Intel 肯定怀念摩尔定律提出时候的黄金年代，只依靠工艺的进步，就能一两年就性能翻番。AMD 肯定怀念 K8 的黄金一代，1G 战 4G，靠的就是把内存控制器从北桥芯片移到 CPU 内部，提升了 CPU 和内存的通信效率，自然性能倍增。而今天，人类的技术已经到达了一个瓶颈，只能通过不断的提升 CPU 和操作系统的复杂度来获得微弱的性能提升，呜呼哀哉。

不过我们也不能放弃希望，AMD RX VAGA64 显卡拥有 2048 位的显存位宽，理论极限还是很恐怖的，这可能就是未来内存的发展方向。

## （三）通用电子计算机的胎记：事件驱动

Event-Driven（事件驱动）这个词这几年随着 Node.js® 的大热也成了一个热词，似乎已经成了“高性能”的代名词，殊不知事件驱动其实是通用计算机的胎记，是一种与生俱来的能力。本文我们就要一起了解一下事件驱动的价值和本质。

## 通用电子计算机中的事件驱动

首先我们定义当下最火的 x86 PC 机为典型的通用电子计算机：可以写文章，可以打游戏，可以上网聊天，可以读U盘，可以打印，可以设计三维模型，可以编辑渲染视频，可以作路由器，还可以控制巨大的工业机器。那么，这种计算机的事件驱动能力就很容易理解了：

1. 假设 Chrome 正在播放 Youtube 视频，你按下了键盘上的空格键，视频暂停了。这个操作就是事件驱动：计算机获得了你单击空格的事件，于是把视频暂停了。
2. 假设你正在跟人聊 QQ，别人发了一段话给你，计算机获得了网络传输的事件，于是将信息提取出来显示到了屏幕上，这也是事件驱动。

## 事件驱动的实现方式

事件驱动本质是由 CPU 提供的，因为 CPU 作为 控制器 + 运算器，他需要随时响应意外事件，例如上面例子中的键盘和网络。

CPU 对于意外事件的响应是依靠 Exception Control Flow（异常控制流）来实现的。

## 强大的异常控制流

异常控制流是 CPU 的核心功能，它是以下听起来就很牛批的功能的基础：

### 时间片

CPU 时间片的分配也是利用异常控制流来实现的，它让多个进程在宏观上在同一个 CPU 核心上同时运行，而我们都知在微观上在任一个时刻，每一个 CPU 核心都只能运行一条指令。

### 虚拟内存

这里的虚拟内存不是 Windows 虚拟内存，是 Linux 虚拟内存，即逻辑内存。

逻辑内存是用一段内存和一段磁盘上的存储空间放在一起组成一个逻辑内存空间，对外依然表现为“线性数组内存空间”。逻辑内存引出了现代计算机的一个重要的性能观念：

内存局部性天然的让相邻指令需要读写的内存空间也相邻，于是可以把一个进程的内存放到磁盘上，再把一小部分的“热数据”放到内存中，让其作为磁盘的缓存，这样可以在降低很少性能的情况下，大幅提升计算机能同时运行的进程的数量，大幅提升性能。

虚拟内存的本质其实是使用 缓存 + 乐观 的手段提升计算机的性能。

### 系统调用

系统调用是进程向操作系统索取资源的通道，这也是利用异常控制流实现的。

## 硬件中断

键盘点击、鼠标移动、网络接收到数据、麦克风有声音输入、插入 U 盘这些操作全部需要 CPU 暂时停下手头的工作，来做出响应。

## 进程、线程

进程的创建、管理和销毁全部都是基于异常控制流实现的，其生命周期的钩子函数也是操作系统依赖异常控制流实现的。线程在 Linux 上和进程几乎没有功能上的区别。

## 编程语言中的 try catch

C++ 编译成的二进制程序，其异常控制语句是直接基于异常控制流的。Java 这种硬虚拟机语言，PHP 这种软虚拟机语言，其异常控制流的一部分也是有最底层的异常控制流提供的，另一部分可以由逻辑判断来实现。

## 基于异常控制流的事件驱动

其实现在人们在谈论的事件驱动，是 Linux kernel 提供的 epoll，是 2002 年 10 月 18 号伴随着 kernel 2.5.44 发布的，是 Linux 首次将操作系统中的 I/O 事件的异常控制流暴露给了进程，实现了本文开头提到的 Event-Driven（事件驱动）。

## Kqueue

FreeBSD 4.1 版本于 2000 年发布，起携带的 Kqueue 是 BSD 系统中事件驱动的 API 提供者。BSD 系统如今已经遍地开花，从 macOS 到 iOS，从 watchOS 到 PS4 游戏机，都受到了 Kqueue 的荫蔽。

## epoll 是什么

操作系统本身就是事件驱动的，所以 epoll 并不是什么新发明，而只是把本来不给用户空间用的 api 暴露在了用户空间而已。

## epoll 做了什么

网络 I/O 是一种纯异步的 I/O 模型，所以 Nginx 和 Node.js® 都基于 epoll 实现了完全的事件驱动，获得了相比于 select/poll 巨量的性能提升。而磁盘 I/O 就没有这么幸运了，因为磁盘本身也是单体阻塞资源：即有进程在写磁盘的时候，其他写入请求只能等待，就是天王老子来了也不行，磁盘做不到呀。所以磁盘 I/O 是基于 epoll 实现的非阻塞 I/O，但是其底层依旧是异步阻塞，即便这样，性能也已经爆棚了。Node.js 的磁盘 I/O 性能远超其他解释型语言，过去几年在 Web 后端霸占了一些对磁盘 I/O 要求高的领域。

## （四）Unix 进程模型的局限

---

Unix 系统 1969 年诞生于 AT&T 旗下的贝尔实验室。1971 年，Ken Thompson（Unix 之父）和 Dennis Ritchie（C 语言之父）共同发明了 C 语言，并在 1973 年用 C 语言重写了 Unix。

Unix 自诞生起就是多用户、多任务的分时操作系统，其引入的“进程”概念是计算机科学中最成功的概念之

一，几乎所有现代操作系统都是这一概念的受益者。但是进程也有局限，由于 AT&T 是做电话交换起家，所以 Unix 进程在设计之初就是延续的电话交换这个业务需求：保证电话交换的效率，就够了。

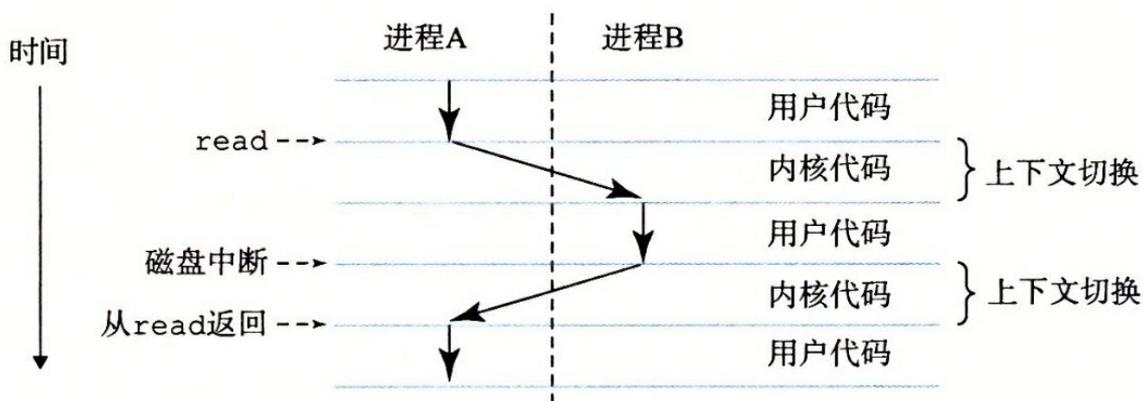
1984年，Richard Stallman 发起了 GNU 项目，目标是创建一个完全自由且向下兼容 Unix 的操作系统。之后 Linus Torvalds 与 1991 年发布了 Linux 内核，和 GNU 结合在了一起形成了 GNU/Linux 这个当下最成功的开源操作系统。所以 Redhat、CentOS、Ubuntu 这些如雷贯耳的 Linux 服务器操作系统，他们的内存模型也是高度类似 Unix 的。

## Unix 进程模型介绍

进程是操作系统提供的一种抽象，每个进程在自己看来都是一个独立的图灵机：独占 CPU 核心，一个一个地运行指令，读写内存。进程是计算机科学中最重要的概念之一，是进程使多用户、多任务成为了可能。

### 上下文切换

操作系统使用上下文切换让一个 CPU 核心上可以同时运行多个进程：在宏观时间尺度，例如 5 秒内，一台电脑的用户会认为他的桌面进程、音乐播放进程、鼠标响应进程、浏览器进程是在同时运行的。



- 图片来自《CS:APP》

### 上下文切换的过程

以下就是 Linux 上下文切换的过程：

假设正在运行网易云音乐进程，你突然想搜歌，假设焦点已经位于搜索框内。

1. 当前进程是网易云音乐，它正在优哉游哉的播放着音乐
2. 你突然打字，CPU 接到键盘发起的中断信号（异常控制流中的一个异常），准备调起键盘处理进程
3. 将网易云音乐进程的寄存器、栈指针、程序计数器保存到内存中
4. 将键盘处理进程的寄存器、栈指针、程序计数器从内存中读出来，写入到 CPU 内部相应的模块中
5. 执行程序计数器的指令，键盘处理程序开始处理键盘输入
6. 完成了一次上下文切换

### 名词解释

■

- 寄存器：CPU 核心里的用于暂时存储指令、地址和数据的电路，和内核频率一样，速度极快
- 栈指针：该进程所拥有的栈的指针
- 程序计数器：简称 PC，它存储着内核将要执行的下一个指令的内存地址。程序计数器是图灵机的核心组成部分。还记得冯·诺依曼架构吗，它的一大创造就是把指令和数据都存在内存里，让计算机获得了极大的自由度。

## Unix 进程模型的局限

Unix 进程模型十分的清晰，上下文切换使用了一个非常简单的操作就实现了多个进程的宏观同时运行，是一个伟大的杰作。但是它却存在着一个潜在的缺陷，这个缺陷在 Unix 诞生数十年之后才渐渐浮出了水面。

### 致命的内存

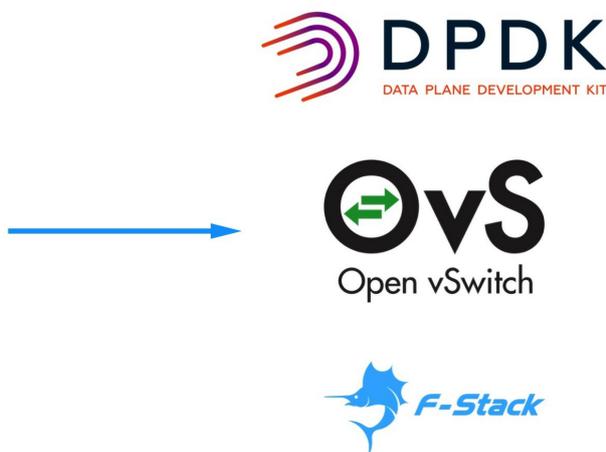
进程切换过程中需要分别写、读一次内存，这个操作在 Unix 刚发明的时候没有发现有什么性能问题，但是 CPU 裹挟着摩尔定律一路狂奔，2000 年，AMD 领先 Intel 两天发布了第一款 1GHz 的微处理器“AMD Athlon 1GHz”，此时一个指令的执行时间已经低到了 1ns，而其内存延迟高达 60ns，这导致了一个以前不曾出现的问题：

上下文切换读写内存的时间成了整个系统的性能瓶颈。

### 软件定义一切

我们将在下一篇文章探讨 SDN（软件定义网络），在这里我们先来看一下“软件定义一切”这个概念。当下，不仅有软件定义网络，还有软件定义存储，甚至出现了软件定义基础架构（这不就是云计算嘛）。是什么导致了软件越来越强势，开始倾入过去只有专业的硬件设备才能提供的高性能高稳定性服务呢？我认为，就是通用计算机的发展导致的，确切地说，是 CPU 和网络的发展导致的。

当前的民用顶级 CPU 的性能已经爆表，因为规模巨大，所以其价格也要显著低于同性能的专用处理器：自建 40G 软路由的价格大约是 40G 专用路由价格的二十分之一。



## (五) DPDK、SDN 与大页内存

---

上文我们说到，当今的 x86 通用微处理器已经拥有了十分强大的性能，得益于其庞大的销量，让它的价格和专用 CPU 比也有着巨大的优势，于是，软件定义一切诞生了！

### 软路由

说到软路由，很多人都露出了会心的微笑，因为其拥有低廉的价格、超多的功能、够用的性能和科学上网能力。现在网上能买到的软路由，其本质就是一个 x86 PC 加上多个网口，大多是基于 Linux 或 BSD 内核，使用 Intel 低端被动散热 CPU 打造出的千兆路由器，几百块就能实现千兆的性能，最重要的是拥有 QOS、多路拨号、负载均衡、防火墙、VPN 组网、科学上网等强大功能，传统路由器抛开科学上网不谈，其他功能也不是几百块就搞得定的。

### 软路由的弱点

软路由便宜，功能强大，但是也有弱点。它最大的弱点其实是性能：传统 \*UNIX 网络栈的性能实在是不高。

软路由的 NAT 延迟比硬路由明显更大，而且几百块的软路由 NAT 性能也不够，跑到千兆都难，而几百块的硬路由跑到千兆很容易。那怎么办呢？改操作系统啊。

### SDN

软件定义网络，其本质就是使用计算机科学中最常用的“虚拟机”构想，将传统由硬件实现的交换、网关、路由、NAT 等网络流量控制流程交由软件来统一管理：可以实现硬件不动，网络结构瞬间变化，避免了传统的停机维护调试的烦恼，也为大规模公有云计算铺平了道路。

### 虚拟机

虚拟机的思想自底向上完整地贯穿了计算机的每一个部分，硬件层有三个场效应管虚拟出的 SRAM、多个内存芯片虚拟出的一个“线性数组内存”，软件层有 jvm 虚拟机，PHP 虚拟机（解释器）。自然而然的，当网络成为了更大规模计算的瓶颈的时候，人们就会想，为什么网络不能虚拟呢？

### OpenFlow

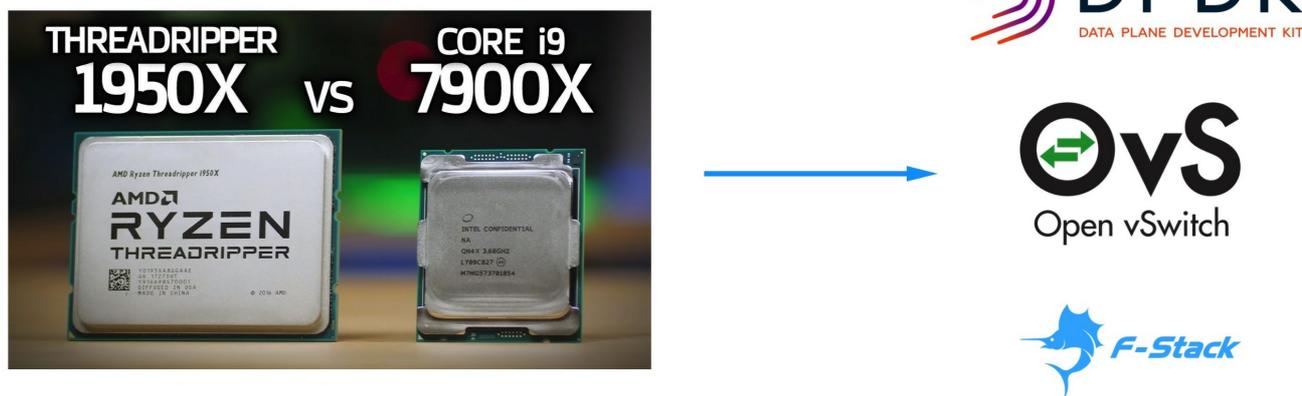
最开始，SDN 还是基于硬件来实施的。Facebook 和 Google 使用的都是 OpenFlow 协议，作用在数据链路层（使用 MAC 地址通信的那一层，也就是普通交换机工作的那一层），它可以统一管理所有网关、交换等设备，让网络架构实时地做出改变，这对这种规模的公司所拥有的巨大的数据中心非常重要。

### DPDK

DPDK 是 SDN 更前沿的方向：使用 x86 通用 CPU 实现 10Gbps 甚至 40Gbps 的超高速网关（路由器）。

### DPDK 是什么

Intel DPDK 全称为 Intel Data Plane Development Kit，直译为“英特尔数据平面开发工具集”，它可以摆脱 \*UNIX 网络数据包处理机制的局限，实现超高速的网络包处理。



## DPDK 的价值

当下，一台 40G 核心网管路由器动辄数十万，而 40G 网卡也不会超过一万块，而一颗性能足够的 Intel CPU 也只需要几万块，软路由的性价比优势是巨大的。

实际上，阿里云和腾讯云也已经基于 DPDK 研发出了自用的 SDN，已经创造了很大的经济价值。

[【DPDK峰会回顾】支撑双十一的高性能负载均衡是如何炼成的](#) [阿里云携领先SDN能力，亮相全球网络技术盛会ONS](#) [腾讯云超高网络性能云主机揭秘](#) [F-Stack 全用户态 \(Kernel Bypass\) 服务开发套件](#)

## 怎么做到的？

DPDK 使用自研的数据链路层（MAC地址）和网络层（ip地址）处理功能（协议栈），抛弃操作系统（Linux, BSD 等）提供的网络处理功能（协议栈），直接接管物理网卡，在用户态处理数据包，并且配合大页内存和 NUMA 等技术，大幅提升了网络性能。有论文做过实测，10G 网卡使用 Linux 网络协议栈只能跑到 2G 多，而 DPDK 分分钟跑满。

## 用户态网络栈

上篇文章我们已经说到，Unix 进程在网络数据包过来的时候，要进行一次上下文切换，需要分别读写一次内存，当系统网络栈处理完数据把数据交给用户态的进程如 Nginx 去处理还会出现一次上下文切换，还要分别读写一次内存。夭寿啦，一共 1200 个 CPU 周期呀，太浪费了。

而用户态协议栈的意思就是把这块网卡完全交给一个位于用户态的进程去处理，CPU 看待这个网卡就像一个假肢一样，这个网卡数据包过来的时候也不会引发系统中断了，不会有上下文切换，一切都如丝般顺滑。当然，实现起来难度不小，因为 Linux 还是分时系统，一不小心就把 CPU 时间占完了，所以需要小心地处理阻塞和缓存问题。

## NUMA

NUMA 来源于 AMD Opteron 微架构，其特点是将 CPU 直接和某几根内存使用总线电路连接在一起，这样 CPU 在读取自己拥有的内存的时候就会很快，代价就是读取别 U 的内存的时候就会比较慢。这个技术伴随着服务器 CPU 核心数越来越多，内存总量越来越大的趋势下诞生的，因为传统的模型中不仅带宽不足，而且极易被抢占，效率下降的厉害。

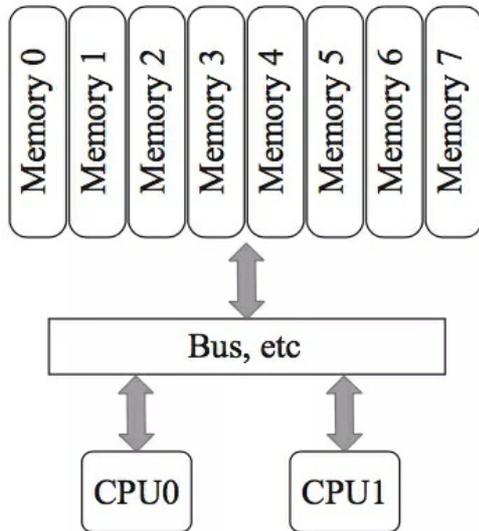


图 2-14 SMP 系统示意图

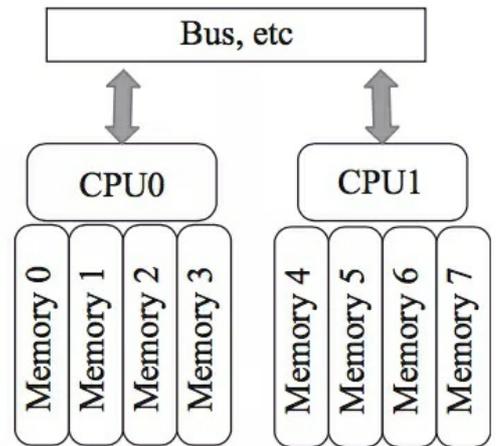


图 2-15 NUMA 系统示意图

NUMA 利用的就是电子计算机（图灵机 + 冯·诺依曼架构）天生就带的涡轮：局部性。 [涡轮：汽车发动机加上涡轮，可以让动力大增油耗降低](#)

## 细说大页内存

### 内存分页

为了实现虚拟内存管理机制，前人们发明了内存分页机制。这个技术诞生的时候，内存分页的默认大小是 4KB，而到了今天，绝大多数操作系统还是用的这个数字，但是内存的容量已经增长了不知道多少倍了。

### TLB miss

TLB (Translation Lookaside Buffers) 转换检测缓冲区，是内存控制器中为增虚拟地址到物理地址的翻译速度而设立的一组电子元件，最近十几年已经随着内存控制器被集成到了 CPU 内部，每颗 CPU 的 TLB 都有固定的长度。

如果缓存未命中 (TLB miss)，则要付出 20-30 个 CPU 周期的带价。假设应用程序需要 2MB 的内存，如果操作系统以 4KB 作为分页的单位，则需要 512 个页面，进而在 TLB 中需要 512 个表项，同时也需要 512 个页表项，操作系统需要经历至少 512 次 TLB Miss 和 512 次缺页中断才能将 2MB 应用程序空间全部映射到物理内存；然而，当操作系统采用 2MB 作为分页的基本单位时，只需要一次 TLB Miss 和一次缺页中断，就可以为 2MB 的应用程序空间建立虚实映射，并在运行过程中无需再经历 TLB Miss 和缺页中断。

## 大页内存

大页内存 HugePage 是一种非常有效的减少 TLB miss 的方式，让我们来进行一个简单的计算。

2013 年发布的 Intel Haswell i7-4770 是当年的民用旗舰 CPU，其在使用 64 位 Windows 系统时，可以提供 1024 长度的 TLB，如果内存页的大小是 4KB，那么总缓存内存容量为 4MB，如果内存页的大小是 2MB，那么总缓存内存容量为 2GB。显然后者的 TLB miss 概率会低得多。

DPDK 支持 1G 的内存分页配置，这种模式下，一次性缓存的内存容量高达 1TB，绝对够用了。

不过大页内存的效果没有理论上那么惊人，DPDK 实测有 10%~15% 的性能提升，原因依旧是那个天生就带的涡轮：局部性。

## (六) 现代计算机最亲密的伙伴：局部性与乐观

冯·诺依曼架构中，指令和数据均存储在内存中，彻底打开了计算机“通用”的大门。这个结构中，“线性数组”内存天生携带了一个涡轮：局部性。

### 局部性分类

#### 空间局部性

空间局部性是最容易理解的局部性：如果一段内存被使用，那么之后，离他最近的内存也最容易被使用，无论是数据还是指令都是这样。举一个浅显易懂的例子：

循环处理一个 Array，当处理完了 [2] 之后，下一个访问的就是 [3]，他们在内存里是相邻的。

#### 时间局部性

如果一个变量所在的内存被访问过，那么接下来这一段内存很可能被再次访问，例子也非常简单：

```
$a = [];  
if ( !$b ) {  
    $a[] = $b;  
}
```

在一个 function 内，一个内存地址很可能被访问、修改多次。

### 乐观

“乐观”作为一种思考问题的方式广泛存在于计算机中，从硬件设计、内存管理、应用软件到数据库均广泛运用了这种思考方式，并给我们带来了十分可观的性能收益。

#### 乐观的 CPU

第一篇文章中的 L1 L2 L3 三级缓存和第二篇文章中的分支预测与流水线，均是乐观思想的代表。

## 乐观的虚拟内存

虚拟内存依据计算机内存的局部性，将磁盘作为内存的本体，将内存作为磁盘的缓存，用很小的性能代价带来了数十倍并发进程数，是乐观思想的集大成者。

## 乐观的缓存

Java 经典面试题 LRU 缓存实现，也是乐观思想的一种表达。

同样，鸟哥的 [yac](#) 也是这一思想的强烈体现。

### 设计 Yac 的经验假设

1. 对于一个应用来说，同名的Cache键，对应的Value，大小几乎相当。
2. 不同的键名的个数是有限的。
3. Cache的读的次数，远远大于写的次数。
4. Cache不是数据库，即使Cache失效也不会带来致命错误。

### Yac 的限制

1. key的长度最大不能超过48个字符。(我想这个应该是能满足大家的需求的，如果你非要用长Key，可以MD5以后再存)
2. Value的最大长度不能超过64M，压缩后的长度不能超过1M。
3. 当内存不够的时候，Yac会有比较明显的踢出率，所以如果要使用Yac，那么尽量多给点内存吧。

## 乐观锁

乐观锁在并发控制和数据库设计里都拥有重要地位，其本质就是在特定的需求下，假定不会冲突，冲突之后再浪费较长时间处理，比直接每次请求都浪费较短时间检测，总体的性能高。乐观锁在算法领域有着非常丰富而成熟的应用。

## 乐观的分布式计算

分布式计算的核心思想就是乐观，由 95% 可靠的 PC 机组成的分布式系统，起可靠性也不会达到 99.99%，但是绝大多数场景下，99% 的可靠性就够了，毕竟拿 PC 机做分布式比小型机便宜得多嘛。下一篇文章我会详细介绍分布式计算的性能之殇，此处不再赘述。

## 乐观的代价

出来混，早晚是要还的。

乐观给了我们很多的好处，总结起来就是一句话：以微小的性能损失换来大幅的性能提升。但是，人在河边走，哪有不湿鞋。每一个 2015 年 6 月入 A 股的散户，都觉得大盘还能再翻一番，岂不知一周之后，就是股灾了。

乐观的代价来自于“微小的性能损失”，就跟房贷市场中“微小的风险”一样，当大环境小幅波动的时候，他确实能承担压力，稳住系统，但是怕就怕突然雪崩：

1. 虚拟内存中的内存的局部性突然大幅失效，磁盘读写速度成了内存读写速度，系统卡死
2. 分布式数据库的六台机器中的 master 挂了，系统在一秒内选举出了新的 master，你以为系统会稳定运行？master 挂掉的原因就是压力过大，这样就会导致新的 master 瞬间又被打挂，然后一台一台地继续，服务彻底失效。例如：[「故障说明」对六月六日 LeanCloud 多项服务发生中断的说明](#)

## (七) 分布式计算、超级计算机与神经网络共同的瓶颈

分布式计算是这些年的热门话题，各种大数据框架层出不穷，容器技术也奋起直追，各类数据库（Redis、Elasticsearch、MongoDB）也大搞分布式，可以说是好不热闹。分布式计算在大热的同时，也存在着两台机器也要硬上 Hadoop 的“面向简历编程”，接下来我就剖析一下分布式计算的本质，以及我的理解和体会。

### 分布式计算的本质

分布式计算来源于人们日益增长的性能需求与落后的 x86 基础架构之间的矛盾。恰似设计模式是面向对象对现实问题的一种妥协。

#### x86 服务器

x86 服务器，俗称 PC 服务器、微机服务器，近二十年以迅雷不及掩耳盗铃之势全面抢占了绝大部分的服务器市场，它和小型机比只有一个优势，其他的全是缺点，性能、可靠性、可扩展性、占地面积都不如小型机，但是一个优势就决定了每年 2000 多亿美元的 IDC 市场被 x86 服务器占领了 90%，这个优势就是价格。毕竟有钱能使磨推鬼嘛。

现有的分布式计算，无论是 Hadoop 之类的大数据平台，还是 HBase 这样的分布式数据库，无论是 Docker 这种容器排布，还是 Redis 这种朴素分布式数据库，其本质都是因为 x86 的扩展性不够好，导致大家只能自己想办法利用网络来自己构建一个宏观上更强性能更高负载能力的计算机。

**x86 分布式计算**，是一种新的计算机结构。

基于网络的 x86 服务器分布式计算，其本质是把网络当做总线，设计了一套新的计算机体系结构：

- 每一台机器就等于一个运算器加一个存储器
- master 节点就是控制器加输入设备、输出设备

#### x86 分布式计算的弱点

上古时代，小型机的扩展能力是非常变态的，到今天，基于小型机的 Oracle 数据库系统依旧能做到惊人的性能和可靠性。实际上单颗 x86 CPU 的性能已经远超 IBM 小型机用的 PowerPC，但是当数量来到几百颗，x86 服务器集群就败下阵来，原因也非常简单：

1. 小型机是专门设计的硬件和专门设计的软件，只面向这种规模（例如几百颗 CPU）的计算

2. 小型机是完全闭源的，不需要考虑扩展性，特定的几种硬件在稳定性上前进了一大步
3. x86 的 I/O 性能被架构锁死了，各种总线、PCI、PCIe、USB、SATA、以太网，为了个人计算机的便利性，牺牲了很多的性能和可靠性
4. 小型机使用总线通信，可以实现极高的信息传递效率，极其有效的监控以及极高的故障隔离速度
5. x86 服务器基于网络的分布式具有天然的缺陷：
  1. 操作系统决定了网络性能不足
  2. 网络需要使用事件驱动处理，比总线电路的延迟高几个数量级
  3. PC 机的硬件不够可靠，故障率高
  4. 很难有效监控，隔离故障速度慢

## x86 分布式计算的基本套路

### Google 系大数据处理框架

2003 年到 2004 年间，Google 发表了 MapReduce、GFS (Google File System) 和 BigTable 三篇技术论文，提出了一套全新的分布式计算理论。MapReduce 是分布式计算框架，GFS (Google File System) 是分布式文件系统，BigTable 是基于 Google File System 的数据存储系统，这三大组件组成了 Google 的分布式计算模型。

MapReduce 的基本思想其实很容易理解：由于数据流实在是太大，所以需要基于磁盘，这样数据就被天然地分布到了多台机器上，然后“就地计算”即可，这被称作“计算向数据靠拢”。除此之外，MapReduce 还提供了全套的数据分片、信息交换、任务分发和聚合功能，使用者只需要使用自己的编程语言，分发任务即可，使用者甚至不需要知道具体的并发模型是怎么设计的，用就完了。

Hadoop、Spark、Storm 是目前最重要的三大分布式计算系统，他们都是承袭 Google 的思路实现并且一步一步发展到今天的。

### Redis、MongoDB 的分布式

Redis 有两个不同的分布式方案。Redis Cluster 是官方提供的工具，它通过特殊的协议，实现了每台机器都拥有数据存储和分布式调节功能，性能没有损失。缺点就是缺乏统一管理，运维不友好。Codis 是一个非常火的 Redis 集群搭建方案，其基本原理可以简单地描述如下：通过一个 proxy 层，完全隔离掉了分布式调节功能，底层的多台机器可以任意水平扩展，运维十分友好。

MongoDB 官方提供了一套完整的分布式部署的方案，提供了 mongos 控制中心，config server 配置存储，以及众多的 shard (其底层一般依然有两台互为主从强数据一致性的 mongod)。这三个组件可以任意部署在任意的机器上，MongoDB 提供了 master 选举功能，在检测到 master 异常后会自动选举出新的 master 节点。

## 问题和瓶颈

人们费这么大的劲研究基于网络的 x86 服务器分布式计算，目的是什么？还不是为了省钱，想用一大票便宜的 PC 机替换掉昂贵的小型机、大型机。虽然人们已经想尽了办法，但还是有一些顽固问题无法彻底解决。

## master 失效问题

无论怎样设计，master 失效必然会导致服务异常，因为网络本身不够可靠，所以监控系统的容错要做的比较高，所以基于网络的分布式系统的故障恢复时间一般在秒级。而小型机的单 CPU 故障对外是完全无感的。

现行的选举机制主要以节点上的数据以及节点数据之间的关系为依据，通过一顿猛如虎的数学操作，选举出一个新的 master。逻辑上，选举没有任何问题，如果 master 因为硬件故障而失效，新的 master 会自动顶替上，并在短时间内恢复工作。

而自然界总是狠狠地打人类的脸：

1. 硬件故障概率极低，大部分 master 失效都不是因为硬件故障
2. 如果是流量过大导致的 master 失效，那么选举出新的 master 也无济于事：提升集群规模才是解决之道
3. 即使能够及时地在一分钟之内顶替上 master 的工作，那这一分钟的异常也可能导致雪崩式的 cache miss，从磁盘缓存到虚拟内存，从 TLB 到三级缓存，再到二级缓存和一级缓存，全部失效。如果每一层的失效会让系统响应时间增加五倍的话，那最终的总响应时长将是惊人的。

## 系统规模问题

无论是 Master-Slave 模式还是 Proxy 模式，整个系统的流量最终还是要落到一个特定的资源上。当然这个资源可能是多台机器，但是依旧无法解决一个严重的问题：系统规模越大，其本底性能损失就越大。

这其实是我们所在的这个宇宙空间的一个基本规律。我一直认为，这个宇宙里只有一个自然规律：熵增。既然我们这个宇宙是一个熵增宇宙，那么这个问题就无法解决。

## 超级计算机

超级计算机可以看成是一个规模特别巨大的分布式计算系统，他的性能瓶颈从目前的眼光来看，是超多计算核心（数百万）的调节效率问题。其本质是通信速率不够快，信息传递的太慢，让数百万核心一起工作，传递命令和数据的工作占据了绝大多数的运行时间。

## 神经网络

深度学习这几年大火，其原因就是卷积神经网络（CNN）造就的 AlphaGo 打败了人类，计算机在这个无法穷举的游戏里彻底赢了。伴随着 Google 帝国的强大推力，深度学习，机器学习，乃至人工智能，这几个词在过去的两年大火，特别是在中美两国。现在拿手机拍张照背后都有机器学习你敢信？

机器学习的瓶颈，本质也是数据交换：机器学习需要极多的计算，而计算速度的瓶颈现在就在运算器和存储器的通信上，这也是显卡搞深度学习比 CPU 快数十倍的原因：显存和 GPU 信息交换的速度极快。

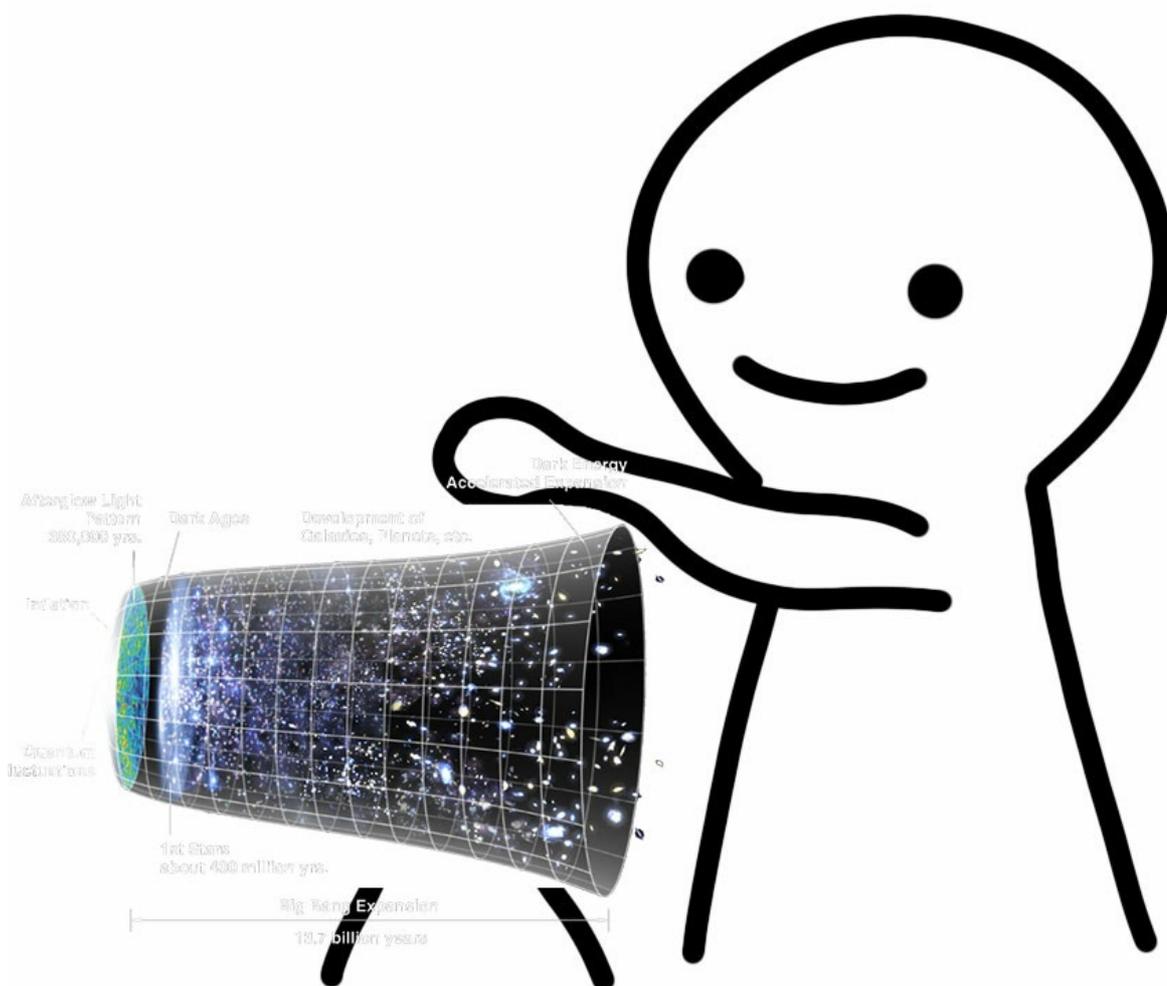
## 九九归一

分布式系统的性能问题，表现为多个方面，但是归根到底，其原因只是一个非常单纯的矛盾：人们日益增长的性能需求和数据一致性之间的矛盾。一旦需要强数据一致性，那就必然存在一个限制性能的瓶颈，这个瓶颈就是信息传递的速度。

同样，超级计算机和神经网络的瓶颈也都是信息传递的速度。

那么，信息传递速度的瓶颈在哪里呢？

我个人认为，信息传递的瓶颈最表层是人类的硬件制造水平决定的，再往底层去是冯·诺依曼架构决定的，再往底层去是图灵机的逻辑模型决定的。可是图灵机是计算机可行的理论基础呀，所以，还是怪这个熵增宇宙吧，为什么规模越大维护成本越高呢，你也是个成熟的宇宙了，该学会自己把自己变成熵减宇宙了。



# 你也是个成熟的宇宙了，该学会自己把自己变成熵减宇宙了

软件工程师需要了解的网络知识：从铜线到 HTTP

起步

写作目标

本文面向中国互联网界众多的“应用软件工程师”，确切地说，面向 Web 后端工程师（Java、PHP），We

b 前端工程师，移动开发工程师（iOS、Android）。本文将从铜线讲起，一路讲到 HTTP，为大家剖析出一个真实的“网络”。

## 写作由来

### 内容来源

前两天我给一个要跳槽的做 iOS 的哥们儿讲了几个小时的网络，给他的面试铺路，在讲之前，我就意识到了这次的内容如果能够整理一下将会是一套丰富的面向软件工程师的网络教程。

### 为什么是我

我装系统起家，从自己搭建网站开始了解 PHP 技术，大学期间通过做外包成了一名 Web 全栈，毕业后创业因为招不到满意的 iOS 于是自己动手写 Swift——当时 Swift 刚刚发布半年，基础库匮乏而且还有缺陷，于是我自己造了一堆轮子，其中就有网络库，于是我对 HTTP 有了深入的理解（实际上超级简单没啥好深入的），又搞了 HTTPS pinning，加上我自己做后端和运维，自己申请、购买、部署证书，对 HTTPS 有了一些了解；后来再次创业自己搭建办公室网络，对路由器、交换机、网关、DHCP、DNS 等的概念有了亲身体会，加上我偶尔也会从我买了不看的书堆中找出《图解 HTTP/TCP,IP/网络硬件》啥的看一看，加上这次讲解的机会，终于融会贯通了。

### 说说融会贯通

大学时候我对计算机的理解融会贯通的点在“编译”，最近的融会贯通在于我看了神书《CS:APP》，对计算机系统又有了更深入的理解：硬件和操作系统是密切配合的；当前 x86\_64 PC 的性能大部分来自于“缓存” + “乐观”的概念。以后有机会的话我会写文章分享一下我的理解。

## 本文约定

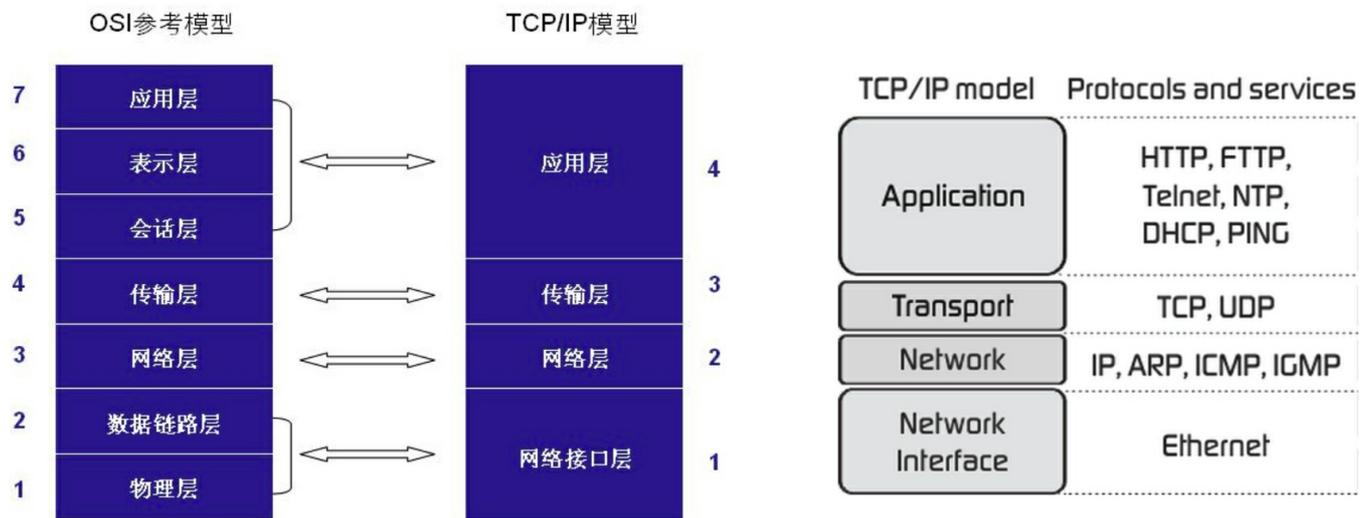
本文中，我们将进行如下几个重要约定：

1. 本文目标是让软件工程师了解网络各层的本质，而不是事无巨细什么都讲
2. 有些部分我将会用打比方的形式讲解，目的是让读者更容易理解，而不是每个细节都完全是真实情形
3. 如果有我理解的不对的地方欢迎在评论里指出

## 以太网与交换机

---

### 网络七层、四层模型



四层模型是 TCP/IP 技术的实际模型，七层模型是标准化组织制订的理论规范，两者有如上图的对应关系。人们很少用到七层模型，一般常见的地方在负载均衡时：四层负载均衡和七层负载均衡，分别指的是在 TCP 和 HTTP 层面进行负载均衡。

## 以太网

### 历史沿革

Robert Metcalfe 在施乐帕洛阿尔托研究中心时发表了一篇名为《以太网：局域计算机网络的分布式包交换技术》的文章，随后获得了“具有冲突检测的多点数据通信系统”的专利，并于 1979 年创办了 3COM 公司，对，就是 H3C 的美国母亲。

### 以太网是什么

以太网是一系列标准，其显著特点是构造简单，可以多台计算机组成一个网络，虽然 IBM 的令牌环网理论上要比以太网优越，但是以太网胜在简单、便宜。实践证明，真正的计算机网络并不需要令牌环网的高吞吐量优势，而是更低价格更强扩展能力。以太网早期支持不同规格的同轴电缆和双绞线，最终双绞线技术胜出：可以实现更高速率。什么是同轴电缆呢？有线电视网络用的就是同轴电缆，起定义就是字面意思：两根线的轴是同一条线。

本文中我们只讨论最新的八芯双绞线实现的千兆网络，这也是当下最常见的局域网技术。

### 双绞线是什么

双绞线也是字面意思：八根铜线两两绞在一起，目的是尽量减少电磁干扰：电磁干扰是阻碍双绞线获得 10G、40G、100G 等更高带宽最大的阻力，当下更高速度的网络一般用光纤技术来实现。

八芯双绞线就是八根铜线绞在一起，双绞线里的铜线和 USB 线里的铜线没有任何区别，都是铜线，只有电导率、粗细、电阻 等基础电气属性的不同。

### 以太网传递的是什么

以太网传递的是电信号：一股又一股的电流而已。跟供电铜线不同的是，以太网协议确定了一系列的约定，让一股又一股的电流能够传递信息：0 或者 1。

### 一个简化模型

六类双绞线实现千兆以太网时的工作频率是 250MHz，即每根铜线中的电流一秒钟都会改变 250 000 000 次，每一次都会携带一个电压：我们假设为 -5V 和 0V，代表 0 和 1，那么只需要四根铜线就能够实现 1Gbps 的单向带宽，另外四根用于实现反向 1Gbps 的带宽。

超五类双绞线实现千兆以太网时只有 125MHz 的频率，它采用一个电压表示两个位的方法来实现千兆网络：即一个电压代表 00 或 01 或 10 或 11。我们假设这四个电压分别是：-5V -3.5V -2V 0V。这样每两根铜线合在一起恰好可以实现 250Mbps 的双向传输（还需要两端设备支持串扰消除技术），八根铜线恰好实现 1Gbps 双向带宽。

### 一个关于计算机的常识

计算机内部许多电路的电压都不是标准的，例如主板标准电压为 12V，一般情况下实际电压都会稍高于这个数字。所以才有了各种纠错方法来保证信息传递的正确。

### 小结论

超五类双绞线和六类双绞线虽然都能实现千兆网络，但是我们可以看出两者是截然不同的两种技术：六类双绞线由于本身电气性能有很大优势，所以需要的设备更简单，抗干扰能力更强，而超五类可以说是“勉强”实现了千兆网络，抗干扰能力差；如果电压因为电路问题或者受到外部干扰而产生了一些变化，很可能导致数据传输错误导致校验失败而引发重新传输，结果就是实测带宽无法达到千兆。我就遇到过超五类双绞线因为线材受损导致网络连接速度在 1G 和 100M 之间跳跃，最终导致网卡自动关闭的事情。

### 结论

全双工千兆以太网，实现的是每秒钟在网线的两个方向分别传递 1G 个的 0 或 1。

### 交换机

交换机是一种十分重要的设备，可以让多台计算机连入同一个局域网。以太网技术胜出的一大原因就是基于交换机技术可以实现低成本和高可靠性的网络扩展：对于早期计算机网络来说，没有什么比低成本地将新的计算机接入网络更激动人心的事情了。

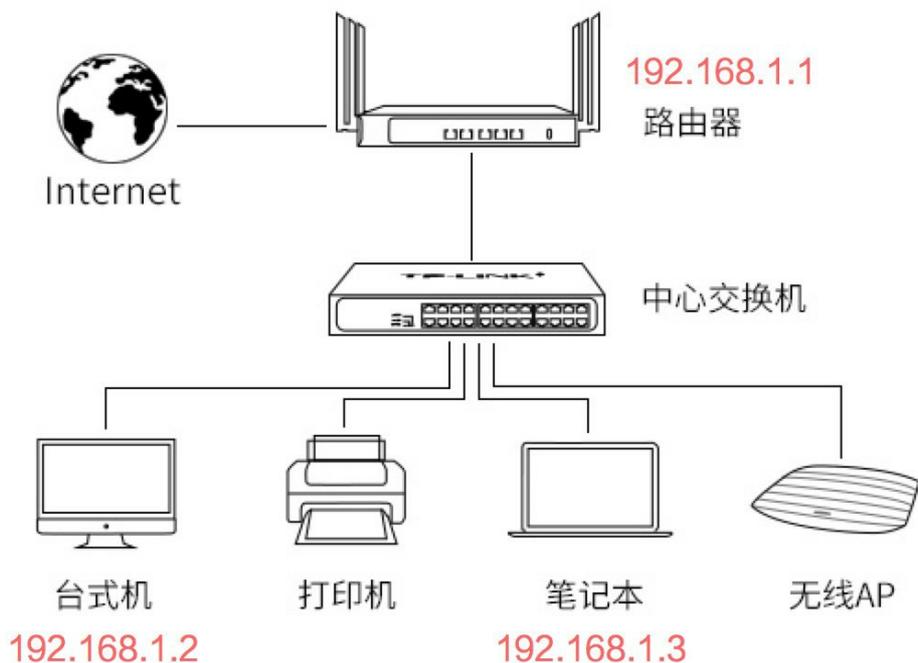
以太网交换机典型的网络拓扑为树状：每一台交换机都可以下接 N 台计算机或交换机。

交换机工作在 OSI 模型的第二层（数据链路层），TCP/IP 模型的第一层。

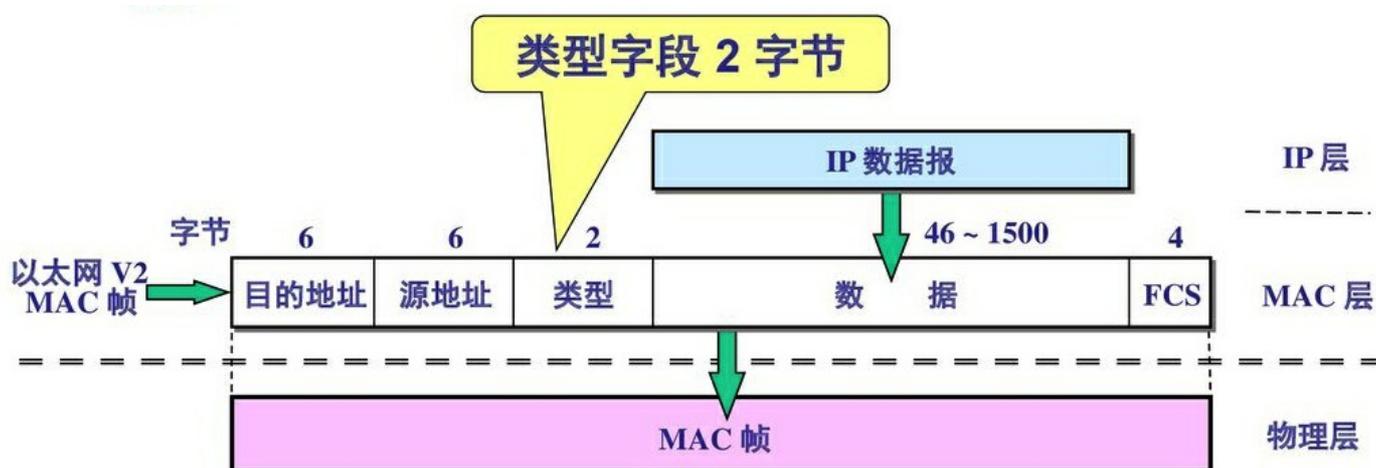
### 局域网典型拓扑图

## 02 标准交换

作为基本型中心交换机使用，连接多个有线设备



以太网帧



解读

物理层中的二进制数据会以上图中的格式进行组织，其基本单元被称为 MAC 帧。

1. 每一台交换机的 MAC 帧 的长度都是一个固定的数值，多台交换机不一定一致
2. 目的地址和源地址均为 MAC 地址，典型代表为 AA:BB:CC:DD:EE:FF，共有六段，每一段是一个两位的 16 进制数，十进制表示为 0-255，恰好可以用八位二进制数表示。所以 MAC 地址的长度为  $8 \times 6 = 48$  位。
3. 类型字段采用 16 位二进制表示更上层的网络层数据包的类型：IP、ARP、ICMP 等。

在理解网络的任何时候都要用位来表示数据，字节在这里毫无意义，只会干扰我们的大脑。

## 交换机工作原理

MAC帧 中没有 ip 的概念，只有 MAC 地址的概念。

假设局域网中一台 ip 为 192.168.1.2 的电脑（插在交换机接口 1 上）希望打开 192.168.1.3 这台服务器（插在交换机接口 2 上）上的网页，就会发生如下事情：

1. 192.168.1.2 向局域网发出一个 ARP 包，询问拥有 192.168.1.3 这个 ip 的计算机的 MAC 地址，假设为 AA:BB:CC:DD:EE:FF
2. 将 TCP 数据包放在 IP 数据包的内部，再将 IP 数据包放在 MAC 帧 内部，通过 1 口将 MAC 帧 发给了交换机
3. 交换机拿到数据后，并不知道 AA:BB:CC:DD:EE:FF 这个 MAC 地址的设备插在自己的哪一个网口上，于是将这个 MAC 帧 发送到所有口上，包括 1 口
4. 2 口回应了，这时交换机就完成了第一次的自学习：AA:BB:CC:DD:EE:FF 这个 MAC 地址的设备插在自己的 2 口上，下次再转发就只发给 2 口就行了
5. 交换机会自己维护一个 MAC 地址 - 物理接口 对应关系的缓存表，并在一定时间内刷新这张表，重新缓存

## MAC 地址

世界上每一块网卡都有一个全世界唯一的 MAC 地址，由厂商向 IEEE 购买，再预先烧录进芯片里：每个有线网卡、每台路由器、每个无线网卡甚至每个蓝牙芯片都有自己的唯一 MAC 地址。

## 三层交换机

现在的网络组建特别是大型网络组建时，三层交换机会会有很多应用，其本质是一种拥有部分路由器功能的高级交换机，感兴趣的同学可以自己了解。

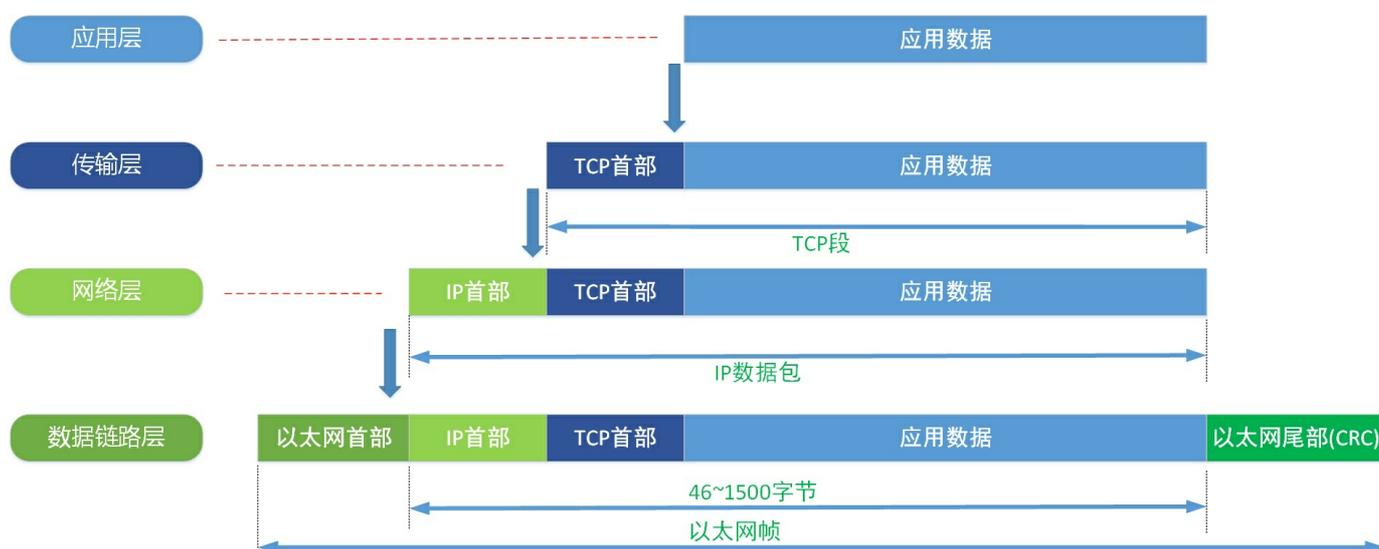
## 结论

交换机第一次规范了双绞线中传递的 0、1 所代表的含义，这个就叫做“协议”。至此，交换机已经实现了局域网内数据的传递，可以由任意一台设备向任意一台设备传递二进制数据。

## TCP/IP

---

### 那些首部



一个 HTTP 请求发送到服务器上，需要在头部按顺序加上 TCP首部、IP首部、以太网首部，这样才能保证这个 HTTP 请求的二进制数据能够在复杂的网络环境中得到可靠的传输：这三个首部在经过各种网络设备时会被大量修改以实现正确传输：以太网首部在经过交换机时并不会被修改，但是 IP首部 和 TCP首部 在经过路由器时会被修改。

## TCP/IP 概述

当我们通过书本、博文等了解 TCP/IP 技术时，均将这两者作为两层来讲，似乎他们就是网络标准中的两个不同的虚拟化层级，但事实并没有这么简单。

### 我的虚拟化观

当初我学习《计算机组成原理》这本书的时候，我印象最深刻的就是里面对于计算机“虚拟化”的描述：只看计算机的硬件部分，其本质就是多层虚拟化：用逻辑电路、加法电路、积分电路、微分电路等模拟出算数逻辑单元、再和寄存器一起虚拟出运算器和控制器，配合由六个三极管组成一位的 SRAM 虚拟出的 L1、L2、L3 缓存，再配合由电容构成的 DRAM 虚拟出的“线性数组”内存，再由各种总线相互连接，实现了一个可以被操作系统软件控制的基础硬件平台。之后，这个基础硬件平台和操作系统一起虚拟出提供基本计算和逻辑判断服务的 CPU 以及可以被 C 指针读取的一个又一个内存存储单元。

计算机本身就是一层又一层虚拟化的产物，其复杂性远超我们日常生活中的复杂机械设备如汽车。计算机是一个纯粹的“人造”产物，是人类智能的集中体现。计算机从最底层的电路到最上层浏览器中显示出一个字，这背后正是虚拟化思想降低了复杂度，让人脑这个超低内存的计算机能够建造出如此纷繁复杂的计算机大厦。

### TCP/IP 的关系

IP 技术是从 TCP 技术中拆分得来的，如此多层的虚拟化显然不是一开始就这么设计的，人脑是趋利避害的，不会随意提高复杂度。

TCP/IP 技术源于美国军方自 1969 年起开始建设的控制点分散的网络系统 ARPAnet。1975 年，ARPAnet 的规模达到一百多台计算机，只靠传统的硬件协议（如以太网协议）已经很难再支撑更大规模的网络了，

于是开始研发 TCP/IP 技术，并在 1980-1983 年间完成了内部转换。1983 年，支持异构网络的完全架空的 TCP/IP 协议正式发布。此处的异构网络指的是 ARPAnet、以太网、令牌环网、光纤网络（如 FDDI 和现在常说的 FTTH）、PPP 网络（如电话拨号 PPPoE）等 OSI 1-2 层的网络。

就在 1983 年，TCP/IP 被 Unix 4.2 BSD 系统采用。随着 Unix 的成功，TCP/IP 逐步成为 Internet 的标准网络协议，兼容多种物理实现。

## TCP/IP 和以太网

以太网诞生后，提供了一种简单的容易扩展的多台计算机相连接的电信号传输系统，能够一次性传输特定长度的 0、1 信息。以太网和 TCP/IP 是独立发展的，以太网是当前最成功的局域网技术，TCP/IP 则是整个 Internet 的标准协议：无论是局域网内常用的以太网，还是 FDDI 光纤网络，甚至是 LTE 4G 网络，都支持 TCP/IP 协议在其之上运行。从这个角度来看，TCP/IP 才是 Internet 的本体。

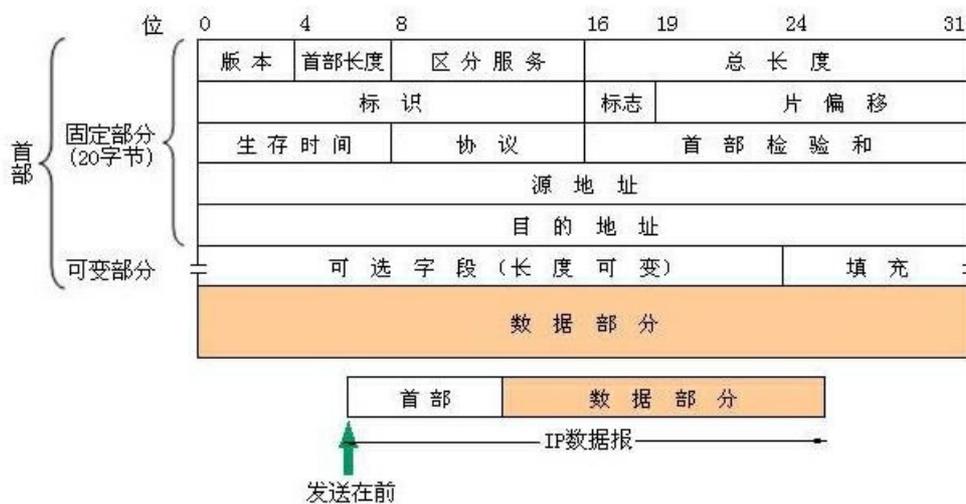
TCP/IP 协议簇是先有实现后有协议的，是从一个已经商用的成熟的网络架构中拆分出来的。

## IP 首部

像以太网帧拥有首部一样，IP 报文也是用首部来描述属性信息的。

### 详解图

以下为 ip 首部详解图 (IPv4)：



### 简单解释

上图中每一行表示 32 位二进制数据。拥有 ip 首部特征的数据会被交换机、路由器、电脑网卡等以太网设备当做有效的 IP 报文（也称“IP 数据报”）。

### 重要数据描述

1. 总长度：界定了本次 ip 报文的长度，便于读取有效数据
2. 源地址：本次 ip 报文是由那个 ip 地址发出的

3. 目的地址：本次 ip 报文需要发给哪一台设备

源地址和目的地址均为 32 位（4 个字节）。我们常见的一个 IPv4 地址为：192.168.1.1，IPv4 地址范围为 0.0.0.0 - 255.255.255.255，255 为 2 的 8 次方减一，也就是说用八位二进制可以表示 0-255，四个八位即为 32 位。

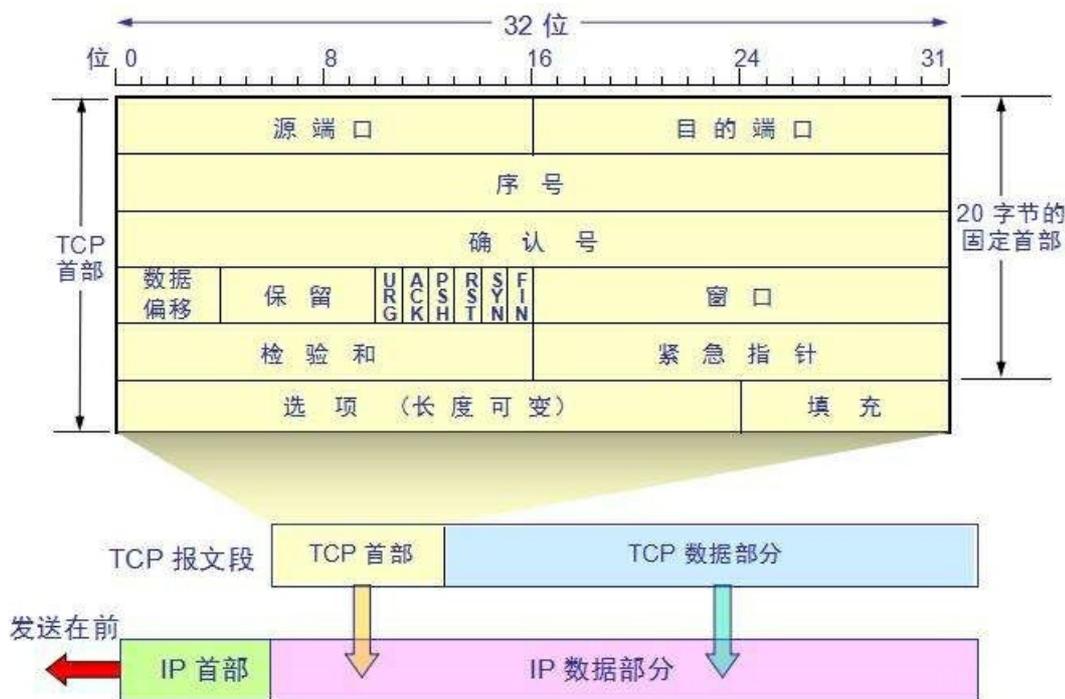
### 小 tip

1 字节等于八位，字节这个单位的出现是因为一个内存地址对应的数据长度为八位，是一个内存相关概念，在理解网络时我们最好抛弃这个概念，全部使用位。

### 结论

ip 层实际上就是规定了一个首部，里面最重要的数据是源 ip 地址和目的 ip 地址。

### TCP 首部



### 重要数据描述

1. TCP 首部中最重要的数据是源端口和目的端口
2. 他们各由 16 位二进制数组成， $2^{16} = 65536$ ，即端口范围为 0-65535
3. 我们可以注意到，目的端口号这个重要数据是放在 TCP 首部的，和 IP 首部、以太网帧首部毫无关系

## TCP 和路由器

### 基础梳理

1. 截至目前，我们已经得到了三个首部：以太网首部、IP 首部、TCP 首部。

2. 这三者看起来类似，实际却完全不同：

1. 以太网首部是以太网技术提供的基础数据 package 功能：其数据包的长度是以太网独有的，和其他技术如光纤环网 FDDI 是完全不同的。实际上其他技术都不一定用的是帧这种基础逻辑单元，也不一定用的是这种简单的交换方式。
2. IP 是网络层协议，其存在的意义是规定一种跨物理实现的“虚拟网络”，让这个网络在上层看来是一致的。从这个意义上来讲，IP 层是 Internet 的本体。IP 网络是真正的全球统一的网络。每一台接入网络的计算机都有一个 ip 地址。
3. TCP 是传输层协议，其目的是在统一的 IP 层上实现“可靠”的信息传递。

## 可靠的 TCP

以太网数据帧和 IP 数据包都只是简单地规定了头部应该如何携带信息，而以太网帧并不保证能够送达，也不能保证按照顺序送达，出现了可靠性问题。

### 一个假设

假设我们需要从 192.168.1.2 向 192.168.1.3 发送一首歌，这首歌是真正的二进制数据，全部采用 0、1 组成，这样会便于我们理解，因为人脑处理文本信息的时候总是有一种障碍。

在没有 TCP 协议的情况下，我们知道了本机以及目标计算机的 ip 地址，我们将歌曲的二进制信息按照 1500 字节（12000位）一块，分别包裹上 IP 首部和以太网首部，通过网口将这段 0、1 发送了出去。假设一共 2000 个以太网帧。接着就会出现下面几种情况：

1. 前 1000 个以太网帧被交换机完美地转发过去了，但是后 1000 个因为交换机受到干扰而没有发到 192.168.1.3 那里，歌曲放到一半就放不出来了。
2. 2000 个都发过去了，但是顺序错乱了：我们会得到一个放不出来的奇怪二进制文件。
3. 2000 个都发过去了，但是部分数据遭到了破坏，0 变成了 1，结果歌曲放一半播放器崩溃了。

这时候我们就会发现只靠 IP 协议是无法满足所有通信要求的。

### 如何实现可靠传输？

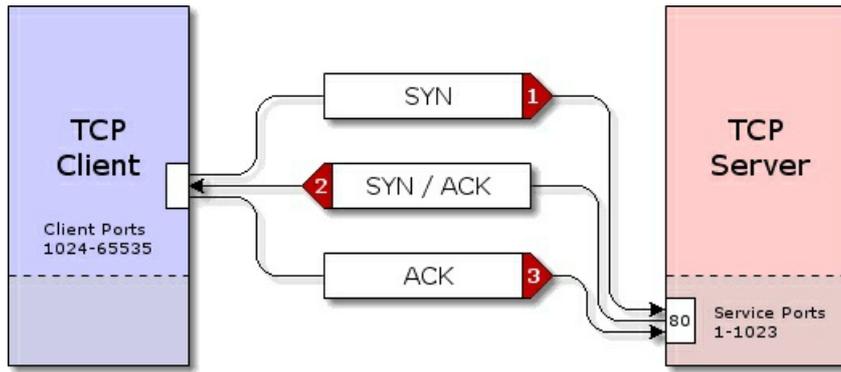
TCP 通过校验、序列号机制、确认应答机制、重发控制、连接管理等特性实现了可靠传输。具体的特性不再展开叙述，因为 TCP 实在是太复杂，展开讲还能再写五个本文这么长的系列文章。下面我重点介绍 TCP 实现可靠传输的几个重点功能：

1. TCP 以段为基本单位发送数据，段的长度是在首次建立连接的时候双方约定好的。
2. 序列号和确认应答机制：每个段的发送都会携带一个整数序列号：当前段第一位在完整数据中的字节顺序，每次接收到一个段，远程计算机都要回复一个带序列号的“确认收到”。
3. 重发机制：首个段发送的时候使用一个比较大的 timeout 值，之后每次的 timeout 的值都是实时计算的，因为 TCP 希望在网络情况变化时也能够尽可能地提供高性能的传输。timeout 时间过了还没有收到携带本段序列号的“确认收到”，那就重发。

TCP 还有基于窗口的发送速度优化、流量控制、拥塞控制等内容不再赘述。

## 三次握手和四次挥手

### 三次握手

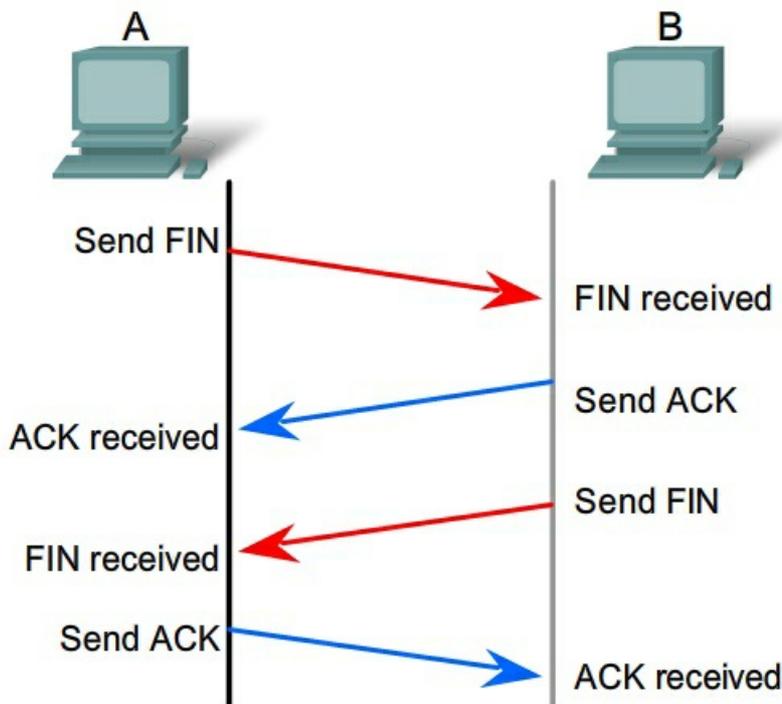


为了建立一个可靠的 TCP 连接，客户端和服务端之间需要进行三次数据发送：

- 客户端：我要建立连接。
- 服务端：收到。我也要建立连接。
- 客户端：收到。

经历过这三个 IP 包的来往，这个可靠的 TCP 连接才算建立成功了。

### 四次挥手



同上，经过这四次 IP 包的往来，双方都认为这个 TCP 连接已经断开了，相应的内存资源就可以释放了。

为什么握手是三次而挥手是四次？

原因很简单：TCP 是全双工协议，即可以同时发送和接收数据，两条通道是完全独立的。

1. 尝试建立连接时，两者之间什么关系都没有，而“收到。我也要建立连接。”这两个动作是有顺序的，直

接用一个 IP 包发送就可以了，节省时间。

2. 尝试断开连接时，“收到。我也要断开连接。”这两个动作之间还有其他事情要发生：客户端这边是不会再发送数据了，但是服务器发给客户端的 IP 包可能还在路上，所以两个方向的 TCP 流是并行的，要单独分别断开连接。

## 路由器

严格意义的路由器是工作在网络层的设备，即 IP 层。但是现代以太网路由器都是 路由器、网关、NAT 服务器、DNS 服务器、DHCP 服务器 的结合体，不少路由器还有防火墙功能。下面我将分开解释这几种功能。

## 路由器

Router 本质是一种“智能”设备，其会为经过它的每一个数据帧寻找一条最佳传输路径，以将该数据最有效地传送到目的站点。Internet 是网状的，而且是动态的，所以路由器是一种非常重要的设备，可以说是它保持了 Internet 的高性能。

## 网关

Gateway 是用于两个不同类型的网络之间通信的设备，例如实现两个以太网的相互通信，实现家庭以太网中的所有设备和光纤背后的服务器的通信。在实际场景中，网关将实现一个重要功能：沟通局域网中的设备和公网设备，例如让 192.168.1.2 这台计算机能够从 baidu.com 的服务器下载网页显示到浏览器上。

## 子网掩码

一台计算机所拥有的 ip 地址和子网掩码配合，让这台计算机认识到了自己的局域网的范围在哪儿。我们采用家用网络中最常见的 192.168.1.2/255.255.255.0 的配合来解释它的作用：

1. 192.168.1.2 是本机的 ip，也是判定某个 ip 是不是局域网 ip 的基础
2. 255.255.255.0 的意思是前三段保持一致，最后一段 0-255 都是局域网 ip
3. 换句话说就是 192.168.1.0 - 192.168.1.255 都是和本机一个局域网的计算机

那判断目标 ip 是不是同一个局域网有什么用呢？

1. 如果要连接的是局域网内的 192.168.1.3，那么连接方式将是：
  1. 发送 ARP 请求得到该 ip 地址对应的 MAC 地址
  2. 将数据包上 TCP 首部、IP 首部（目标 ip 地址 192.168.1.3）、以太网首部（目标地址是 192.168.1.3 的 MAC 地址），发送给交换机
  3. 交换机会直接将这个包发到目标计算机所在的网口上
2. 如果要连接的是 114.114.114.114 这个 ip，那么本机就会将这个包发给网关 192.168.1.1：
  1. 发送 ARP 请求得到网关的 MAC 地址
  2. 将数据包上 TCP 首部（目标端口 80，源端口 20000）、IP 首部（目标 ip 地址 114.114.114.114，源 ip 地址 192.168.1.2）、以太网首部（目标地址是 192.168.1.1 的 MAC 地址），发送给交换机

3. 交换机会直接将这个包发到网关所在的网口上
4. 网关收到了这个以太网帧，进行层层解包：
  1. 目标 MAC 地址是自己，说明这个包是合法的
  2. IP 首部中的目标 ip 地址不是自己，说明这是一个需要网关进行转发的数据包，接着进入转发流程：
    1. TCP 首部中，目标端口 80 不会变，但是源端口 20000 指的是 192.168.1.2 这台计算机的源端口，在网关 192.168.1.1 上这个端口已经被别的局域网机器用过了，该怎么办？修改端口
    2. 将 TCP 首部中的源端口改为 50000，将 IP 首部中的源 ip 地址改为本路由器的 WAN 口 ip，即公网 ip，发送出去。我们假设本路由器的公网 ip 为 106.0.0.1。
    3. 此时网关设备的内存里已经建立起了一个映射：50000 端口对应的是内网的 192.168.1.2 的 20000 端口，当来自 114.114.114.114 80 端口的 IP 包到达时，同样将目标 ip 地址从 106.0.0.1 改为 192.168.1.2，目标端口从 50000 改为 20000，发送到局域网交换机上，再由交换机进行以太网帧转发。

## WAN 口、LAN 口

现代家用路由器一般有一个 WAN 口（广域网端口）和 4-8 个 LAN 口（局域网端口）。有多个 LAN 口本质上是集成了一个交换机。

## NAT 服务器

网络地址转换：将一个公网 ip 直接映射到一个内网 ip 上，当 IP 包经过路由器时，路由器会修改里面的来源 IP 地址和目的 IP 地址，让双方都以为自己真的是和对方直接连接的。

## DNS 服务器

IP 网络中每一个网络终端都有一个 ip 地址，但是一串数字十分难记，于是域名便诞生了。路由器充当 DNS 服务器的目的是提升连接速度，节省局域网内设备的 DNS 查询时间。

## DHCP 服务器

提供自动分配 IP 服务，免去手动设置 ip 地址、子网掩码、网关、DNS 服务器的烦恼，让网络实现真正的即开即用。

# HTTP 和 HTTPS

---

## HTTP

在以前的文章中，我大力推荐过《图解 HTTP》这本书。这是一本好书，但是 HTTP 协议本身是一个静态协议：跟 HTML 一样是一堆标记的集合，十分简单。

我们首先明确一个简单的事实：TCP 首部后面的部分，依然是一堆二进制数据，但是此时，采用 HTTP 协议解析这堆数据之后，其内容终于可读了。

## HTTP 是什么

HTTP 是 WWW（万维网）拥有的标准协议，用于在客户端和服务端之间传递信息：服务器给客户端传递网页，客户端给服务端传递需要的页面的 URL，上传文件等。

### 前提

在讨论 HTTP 协议之前，我们必须首先认识到 HTTP 协议是站在巨人的肩膀上的：

1. HTTP 往下看，是 TCP 协议保证了可靠传输，再往下是 IP 协议保证了 Internet 的大和谐，再往下是以太网协议在局域网内传递信息，再向底层追究，是双绞线中的电压变化将 0、1 一步步向下传递的。
2. HTTP 协议很简单，但却提供一个体验良好的应用标准，到今天依然生命力旺盛。为什么？因为 TCP/IP 协议簇将复杂度消化了。

### 一个普通的 GET 例子

我们使用 Charles 反向代理软件可以轻易地得到 HTTP 协议的细节。下面我们展示一个普通的 GET 例子。使用浏览器访问 <http://killtyz.com>（自己尝试的时候不要选择 HTTPS 网站）：

#### 请求内容

```
GET / HTTP/1.1
Host: killtyz.com
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.119 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9
```

#### 解释：

1. 第一行有三个元素：HTTP 方法、uri、HTTP 版本
2. 之后的每一行均以 : 作为间隔符，左边是 key，右边是 value（当然都是在 trim 之后）
3. HTTP 协议中，换行采用的不是 Linux 系统采用的 \n，而是跟 Windows 一样的 \r\n。

#### 响应内容

```
HTTP/1.1 200 OK
Date: Thu, 25 Jan 2018 10:36:10 GMT
Server: Apache
Content-Length: 1321
Content-Type: text/html; charset=UTF-8
Proxy-Connection: Close
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>KillTYZ 干掉拖延症</title>
  <link href="http://libs.baidu.com/bootstrap/3.2.0/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="/css/main.css">

  <script src="http://libs.baidu.com/jquery/1.11.1/jquery.min.js"></script>
  <script src="http://libs.baidu.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
  <script src="/js/main.js"></script>
</head>
<body>
  <div id="wrap">
    <div id="head">
      <div class="logo">
        
      </div>
      <div class="title">
        <h2>干掉拖延症</h2>
      </div>
    </div>
    <div class="content">
      <div id="box">
        <div id="add">
          <input id="add-input" type="text" class="form-control" placeholder="添加任务">
        </div>
        <div id="list" class="list-group">
        </div>
      </div>
    </div>
    <div id="foot">
      &copy;2018 <a href="http://killtyz.com">KillTYZ</a> | <a href="https://github.com/johnlu
i/KillTYZ">Github Repo</a>
      <br>Powered by <a href="http://tinylara.com">TinyLara</a>
    </div>
  </div>
</body>
</html>
```

响应的基本套路和请求一样，第一行的三个元素分别是 协议版本、状态码、状态码的简短解释。需要注意的只有一点：

### HTTP header 和 HTTP body

1. 两个换行即 `\r\n\r\n` 之前的内容成为 HTTP header
2. 两个换行之后的内容称为 HTTP body
3. HTTP body 就是你在浏览器查看源代码看到的内容

### POST 例子

一下均为 Request 的 HTTP 内容。

### Content-Type: application/x-www-form-urlencoded

```
POST /api/app HTTP/1.1
Host: killtlyz.com
Content-Length: 18
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.119 Safari/537.36
Cache-Control: no-cache
Content-Type: application/x-www-form-urlencoded
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9

post=man&key=value
```

### Content-Type: multipart/form-data

```
POST /api/app HTTP/1.1
Host: killtlyz.com
Content-Length: 5195
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.119 Safari/537.36
Cache-Control: no-cache
Origin: chrome-extension://fhhbjgbiflinjbdggehcdcbncdddomop
Postman-Token: 45479b21-15fa-9232-ab8b-52c7dde8523d
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary1s68Wb5ccTHj384y
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9

-----WebKitFormBoundary1s68Wb5ccTHj384y
Content-Disposition: form-data; name="image"; filename="QQ20141011-2.jpg"
Content-Type: image/jpeg

***二进制文件内容***
-----WebKitFormBoundary1s68Wb5ccTHj384y
Content-Disposition: form-data; name="post"

man
-----WebKitFormBoundary1s68Wb5ccTHj384y
Content-Disposition: form-data; name="oo"

xx
-----WebKitFormBoundary1s68Wb5ccTHj384y--
```

## Cookie 相关 Header

1. Set-Cookie: 响应的 HTTP header 中如果有这个字段, 那么浏览器会把这个字段的 value 设定到本地的 Cookie 里, 配合服务端的 Session 可以实现登录状态临时记录的功能。
2. Cookie: 会出现在请求的 HTTP header 中。Set-Cookie 设置的 Cookie 会有一个作用域名, 一般为 www.baidu.com 或者 .baidu.com, 在浏览器本地记录的众多 Cookie 中, 只要有满足这个作用域名要求的, 下一次 HTTP 请求发出的时候会把这些条 Cookie 全部带上。

更多详细解释需要的时候可以自己查, 都是明码实价, 童叟无欺的。

## HTTPS

HTTPS 这个名字取得不好, 让很多人都误解了, 以为他是和 HTTP 类似的协议, 这是不对的。

HTTPS 全称为 HTTP Over TLS。(SSL/TLS 是一系列承前启后的加密协议族, 此处统称为 TLS。)

## 什么是 TLS

TLS 中文名称为安全传输层协议, 其目的是在客户端与服务端之间建立一个防窃听、防篡改的可信信息传

递通道。

技术细节不再阐述

TLS 真正的组成非常的复杂，本文不想讨论那些技术细节，更不想讨论客户端和服务端建立连接的繁琐方式，什么先非对称加密再对称加密，一概不讨论；本文只进行概述、阐述特点。

技术特点

TLS 采用非对称加密和对称加密结合的方式，在客户端和服务端之间建立起一个防窃听、防篡改的通信通道。TLS 具有以下特点：

1. 基于 TCP 协议，不需要再苦苦解决网络可靠性问题
2. 采用服务端部署证书的形式提供对称加密基础
3. TLS 连接首次建立的过程十分消耗资源，不仅费 CPU 还非常耗时（和简单不可靠的 HTTP 比）
4. TLS 没有完美实现防窃听、防篡改功能：中间人攻击依然存在相当大的可能性

**TLS 实现原理**

我从证书的两种签名方式来讲解 TLS 实现原理的简单描述，并分别阐述当前 HTTPS 证书的两大层面的功能。

自签名 TLS 证书

任何一台安装了 OpenSSL 开源软件的计算机均可以生成 TLS 证书并签名。制造自签名证书分为以下几步：

1. 使用 RSA 算法生成私钥。私钥为绝密，由于证书公开，所以拥有私钥的人将实质上拥有证书的所有权。
2. 生成与私钥一对一的根证书，并填入 TLS 证书需要的必要信息（common name 等）。
3. 采用严格配对的私钥+证书部署 Apache 或 Nginx。

第三方签名的 TLS 证书

分为以下几步：

1. 使用 RSA 算法生成私钥。私钥为绝密，由于证书公开，所以拥有私钥的人将实质上拥有证书的所有权。
2. 生成与私钥一对一的 csr 文件。
3. 将此文件上传到证书颁发商的网站，他们将用他们的根证书或者从根证书派发出的二级证书为我们的 csr 文件签名，得到 TLS 证书。期间会填入经过他们实际验证的必要信息，如 common name 和公司名称。
4. 采用严格配对的私钥+证书部署 Apache 或 Nginx。

最重要的一点

RSA 公钥加密算法从数学上决定了无法从公开的信息（证书）反推出私钥。所以说，只要私钥不泄露，哪怕有人自己写代码强制使用公开的 TLS 证书和他自己伪造的私钥也是不可能的：数学上无法成立，根本就无法

和客户端正常交互建立 TLS 连接。

## 防窃听、防篡改

- 防窃听：TLS 内部携带的数据就是完整的 HTTP 协议，request 和 response 都会被加密，完全无法破解，除非 RSA 算法被破解。TLS 还会在表面上加上少许 HTTP header，只有极少数必要信息如域名等。所以，就算 Twitter 用了 HTTPS，功夫网还是能够侦测出你在访问 Twitter。
- 防篡改：TLS 会对每一次数据交互进行严格的校验。HTTP 时代大家饱受运营商劫持的困扰，本质就是对 HTTP 数据的篡改，明文的嘛，正常。TLS 层中的数据被修改一位，这个 TLS 连接就会崩塌，两边瞬间就都知道了。

## 当前 HTTPS 技术的两层功能

1. 加密。无论证书是自签名还是服务商签名，只要证书没有过期，就可以实现加密，保证信息传递的防窃听、防篡改。
2. 可信。全球数十家服务商的根证书是预置在操作系统内部的：iOS、macOS、Windows 都是这样。因为我们的计算机信任根证书，所以才信任由其派生出的二级及三级证书。可信虽然会在 Chrome 上被标红，但却可以“仍然继续”跳过；而证书过期则不行：TLS 连接都无法建立，Chrome 想继续都继续不了呀。

## TLS 的局限

TLS 虽然很强大，还是有一些问题。我们先说小问题，再说大问题。

### 安全要求不完善

TLS 证书是标准证书，和 HTTP 业务无关，这就导致我们必须采用现成的字段来保存这个证书可以用于那些域名：common name 字段。

我们知道，协议 + 域名 + 端口 组成了一个“域”，域是浏览器中的基本安全单位，用在很多地方。TLS 证书等于说放开了端口这个要求，这样一来一个证书就可以被部署到任意的 N 个端口上。

### 中间人攻击风险依旧

中间人攻击指的是中间有一个人伪造是你想连接的那台服务器，窃取你的信息：在咖啡厅开一个假 wifi，就可以通过修改 DNS 的方式假装你的笔记本是百度的服务器，这样就可以获取想要的信息了。TLS 防止的是传输过程中的防窃听、防篡改，无法解决服务端伪造问题。

中间人攻击分为三个方式：

1. HTTP 转 HTTPS。早期网银攻击经常采用这种方式：用户访问网银网站，浏览器默认发出的是 HTTP 请求，本来该网站会将用户跳转到 HTTPS，但是中间人从中作梗：跟客户交流时采用 HTTP，跟银行交流时采用 HTTPS，这样你的银行卡和密码就全暴露了。
2. 合法证书方式。采用特殊手段签出一张合法的证书，正大光明地做中间人。
3. 终端自残式。破坏自己的终端，强制让自己的操作系统信任一张全域名证书：这样任何网络请求都是明文了。所以 HTTPS 防不了 APP 破解者。

除非客户端和服务端预先进行信息约定，不然从理论上讲是不可能建立一个完全可信的加密数据通道的。

防止中间人攻击的方法

只有一个：在 APP 中内置证书，每次建立连接时都进行比对。感兴趣的可以到我的网站搜 SSL，我专门阐述了如何设置 SSL pinning（钢钉）。

总结

HTTP 真的是一种十分简单的协议，HTTPS 只要了解了 TLS 的基本概念也不复杂，复杂度都在 TCP/IP 那里被消化了。

# 第三方技术资料

📖 高并发的哲学原理

《Philosophical Principles of High Concurrency》

Copyright © 2023 吕文翰