第1章 概 述

Google、Amazon、Alibaba等互联网公司的成功催生了云计算和大数据两大热门领域。无论是云计算、大数据还是互联网公司的各种应用，其后台基础设施的主要目标都是构建低成本、高性能、可扩展、易用的分布式存储系统。

虽然分布式系统研究了很多年，但是，直到近年来，互联网大数据应用的兴起才使得它大规模地应用到工程实践中。相比传统的分布式系统，互联网公司的分布式系统具有两个特点：一个特点是规模大，另一个特点是成本低。不同的需求造就了不同的设计方案，可以这么说，Google等互联网公司重新定义了大规模分布式系统。本章介绍大规模分布式系统的定义与分类。

1.1 分布式存储概念

大规模分布式存储系统的定义如下：

“分布式存储系统是大量普通PC服务器通过Internet互联，对外作为一个整体提供存储服务。”

分布式存储系统具有如下几个特性：

- 可扩展。分布式存储系统可以扩展到几百台甚至几千台的集群规模，而且，随着集群规模的增长，系统整体性能表现为线性增长。
- 低成本。分布式存储系统的自动容错、自动负载均衡机制使其可以构建在普通PC机之上。另外，线性扩展能力也使得增加、减少机器非常方便，可以实现自动运维。
- 高性能。无论是针对整个集群还是单台服务器，都要求分布式存储系统具备高性能。
- 易用。分布式存储系统需要能够提供易用的对外接口，另外，也要求具备完善的监控、运维工具，并能够方便地与其他系统集成，例如，从Hadoop云计算系统导入数据。

分布式存储系统的挑战主要在于数据、状态信息的持久化，要求在自动迁移、自动容错、并发读写的过程中保证数据的一致性。分布式存储涉及的技术主要来自两个领域：分布式系统以及数据库，如下所示：

- 数据分布：如何将数据分布到多台服务器才能够保证数据分布均匀？数据分布到多台服务器后如何实现跨服务器读写操作？
- 一致性：如何将数据的多个副本复制到多台服务器，即使在异常情况下，也能够保证不同副本之间的数据一致性？
- 容错：如何检测到服务器故障？如何自动将出现故障的服务器上的数据和服务迁移到集群中其他服务器？
- 负载均衡：新增服务器和集群正常运行过程中如何实现自动负载均衡？数据迁移的过程中如何保证不影响已有服务？
- 事务与并发控制：如何实现分布式事务？如何实现多版本并发控制？
- 易用性：如何设计对外接口使得系统容易使用？如何设计监控系统并将系统的内部状态以方便的形式暴露给运维人员？
- 压缩 / 解压缩：如何根据数据的特点设计合理的压缩 / 解压缩算法？如何平衡压缩算法节省的存储空间和消耗的 CPU 计算资源？

分布式存储系统挑战大，研发周期长，涉及的知识面广。一般来讲，工程师如果能够深入理解分布式存储系统，理解其他互联网后台架构不会再有任何困难。

1.2 分布式存储分类

分布式存储面临的数据需求比较复杂，大致可以分为三类：

- 非结构化数据：包括所有格式的办公文档、文本、图片、图像、音频和视频信息等。
- 结构化数据：一般存储在关系数据库中，可以用二维关系表结构来表示。结构化数据的模式（Schema，包括属性、数据类型以及数据之间的联系）和内容是分开的，数据的模式需要预先定义。
- 半结构化数据：介于非结构化数据和结构化数据之间，HTML 文档就属于半结构化数据。它一般是自描述的，与结构化数据最大的区别在于，半结构化数据的模式结构和内容混在一起，没有明显的区分，也不需要预先定义数据的模式结构。

不同的分布式存储系统适合处理不同类型的数据，本书将分布式存储系统分为四类：分布式文件系统、分布式键值（Key-Value）系统、分布式表格系统和分布式数据库。

1. 分布式文件系统

互联网应用需要存储大量的图片、照片、视频等非结构化数据对象，这类数据以对象的形式组织，对象之间没有关联，这样的数据一般称为 Blob（Binary Large Object，二进制大对象）数据。

分布式文件系统用于存储 Blob 对象，典型的系统有 Facebook Haystack 以及 Taobao File System（TFS）。另外，分布式文件系统也常作为分布式表格系统以及分布式数据库的底层存储，如谷歌的 GFS（Google File System，存储大文件）可以作为分布式表格系统 Google Bigtable 的底层存储，Amazon 的 EBS（Elastic Block Store，弹性块存储）系统可以作为分布式数据库（Amazon RDS）的底层存储。

总体上看，分布式文件系统存储三种类型的数据：Blob 对象、定长块以及大文件。在系统实现层面，分布式文件系统内部按照数据块（chunk）来组织数据，每个数据块的大小大致相同，每个数据块可以包含多个 Blob 对象或者定长块，一个大文件也可以拆分为多个数据块，如图 1-1 所示。分布式文件系统将这些数据块分散到存储集群，处理数据复制、一致性、负载均衡、容错等分布式系统难题，并将用户对 Blob 对象、定长块以及大文件的操作映射为对底层数据块的操作。

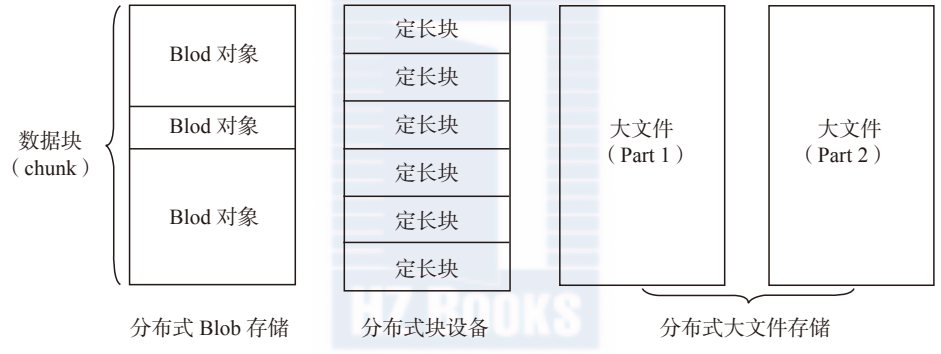

图 1-1　数据块与 Blob 对象、定长块、大文件之间的关系

2. 分布式键值系统

分布式键值系统用于存储关系简单的半结构化数据，它只提供基于主键的 CRUD（Create/Read/Update/Delete）功能，即根据主键创建、读取、更新或者删除一条键值记录。

典型的系统有 Amazon Dynamo 以及 Taobao Tair。从数据结构的角度看，分布式键值系统与传统的哈希表比较类似，不同的是，分布式键值系统支持将数据分布到集群中的多个存储节点。分布式键值系统是分布式表格系统的一种简化实现，一般用作缓存，比如淘宝 Tair 以及 Memcache。一致性哈希是分布式键值系统中常用的数据分布技术，因其被 Amazon DynamoDB 系统使用而变得相当有名。

3. 分布式表格系统

分布式表格系统用于存储关系较为复杂的半结构化数据，与分布式键值系统相比，分布式表格系统不仅仅支持简单的 CRUD 操作，而且支持扫描某个主键范围。分布式

表格系统以表格为单位组织数据，每个表格包括很多行，通过主键标识一行，支持根据主键的 CRUD 功能以及范围查找功能。

分布式表格系统借鉴了很多关系数据库的技术，例如支持某种程度上的事务，比如单行事务，某个实体组（Entity Group，一个用户下的所有数据往往构成一个实体组）下的多行事务。典型的系统包括 Google Bigtable 以及 Megastore，Microsoft Azure Table Storage，Amazon DynamoDB 等。与分布式数据库相比，分布式表格系统主要支持针对单张表格的操作，不支持一些特别复杂的操作，比如多表关联，多表联接，嵌套子查询；另外，在分布式表格系统中，同一个表格的多个数据行也不要求包含相同类型的列，适合半结构化数据。分布式表格系统是一种很好的权衡，这类系统可以做到超大规模，而且支持较多的功能，但实现往往比较复杂，而且有一定的使用门槛。

4. 分布式数据库

分布式数据库一般是从单机关系数据库扩展而来，用于存储结构化数据。分布式数据库采用二维表格组织数据，提供 SQL 关系查询语言，支持多表关联，嵌套子查询等复杂操作，并提供数据库事务以及并发控制。

典型的系统包括 MySQL 数据库分片（MySQL Sharding）集群，Amazon RDS 以及 Microsoft SQL Azure。分布式数据库支持的功能最为丰富，符合用户使用习惯，但可扩展性往往受到限制。当然，这一点并不是绝对的。Google Spanner 系统是一个支持多数据中心的分布式数据库，它不仅支持丰富的关系数据库功能，还能扩展到多个数据中心的成千上万台机器。除此之外，阿里巴巴 OceanBase 系统也是一个支持自动扩展的分布式关系数据库。

关系数据库是目前为止最为成熟的存储技术，它的功能极其丰富，产生了商业的关系数据库软件（例如 Oracle，Microsoft SQL Server，IBM DB2，MySQL）以及上层的工具及应用软件生态链。然而，关系数据库在可扩展性上面临着巨大的挑战。传统关系数据库的事务以及二维关系模型很难高效地扩展到多个存储节点上，另外，关系数据库对于要求高并发的应用在性能上优化空间较大。为了解决关系数据库面临的可扩展性、高并发以及性能方面的问题，各种各样的非关系数据库风起云涌，这类系统成为 NoSQL 系统，可以理解为“Not Only SQL”系统。NoSQL 系统多得让人眼花缭乱，每个系统都有自己的独到之处，适合解决某种特定的问题。这些系统变化很快，本书不会尝试去探寻某种 NoSQL 系统的实现，而是从分布式存储技术的角度探寻大规模存储系统背后的原理。

第一篇
基 础 篇

本篇内容

第 2 章　单机存储系统

第 3 章　分布式系统

第 2 章　单机存储系统

单机存储引擎就是哈希表、B 树等数据结构在机械磁盘、SSD 等持久化介质上的实现。单机存储系统是单机存储引擎的一种封装，对外提供文件、键值、表格或者关系模型。单机存储系统的理论来源于关系数据库。数据库将一个或多个操作组成一组，称作事务，事务必须满足原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）以及持久性（Durability），简称为 ACID 特性。多个事务并发执行时，数据库的并发控制管理器必须能够保证多个事务的执行结果不能破坏某种约定，如不能出现事务执行到一半的情况，不能读取到未提交的事务，等等。为了保证持久性，对于数据库的每一个变化都要在磁盘上记录日志，当数据库系统突然发生故障，重启后能够恢复到之前一致的状态。

本章首先介绍 CPU、IO、网络等硬件基础知识及性能参数，接着介绍主流的单机存储引擎。其中，哈希存储引擎是哈希表的持久化实现，B 树存储引擎是 B 树的持久化实现，而 LSM 树（Log Structure Merge Tree）存储引擎采用批量转储技术来避免磁盘随机写入。最后，介绍关系数据库理论基础，包括事务、并发控制、故障恢复、数据压缩等。

2.1　硬件基础

硬件发展很快，摩尔定律告诉我们：每 18 个月计算机等 IT 产品的性能会翻一番；或者说相同性能的计算机等 IT 产品，每 18 个月价钱会降一半。但是，计算机的硬件体系架构保持相对稳定。架构设计很重要的一点就是合理选择并且能够最大限度地发挥底层硬件的价值。

2.1.1　CPU 架构

早期的 CPU 为单核芯片，工程师们很快意识到，仅仅提高单核的速度会产生过多的热量且无法带来相应的性能改善。因此，现代服务器基本为多核或多个 CPU。经典的多 CPU 架构为对称多处理结构（Symmetric Multi-Processing，SMP），即在一个计算机上汇集了一组处理器，它们之间对称工作，无主次或从属关系，共享相同的物理内存及总线，如图 2-1 所示。

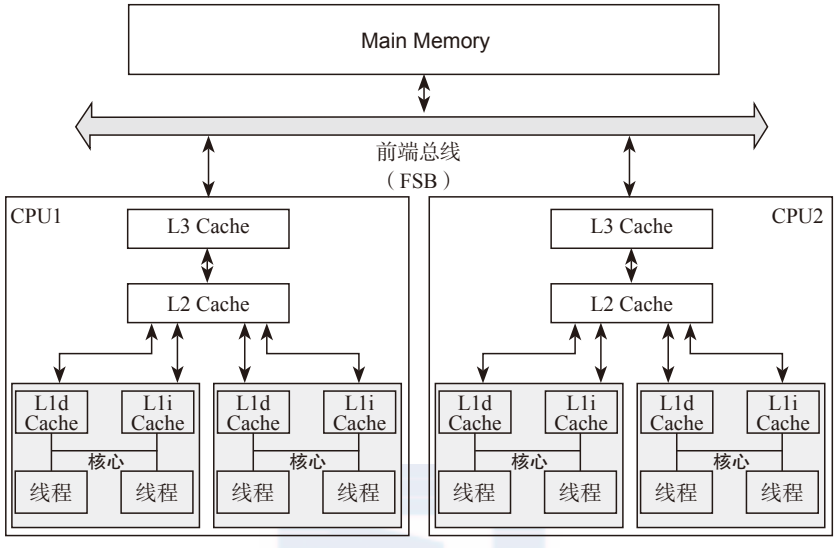

图 2-1　SMP 系统结构

图 2-1 中的 SMP 系统由两个 CPU 组成，每个 CPU 有两个核心（core），CPU 与内存之间通过总线通信。每个核心有各自的 L1d Cache（L1 数据缓存）及 L1i Cache（L1 指令缓存），同一个 CPU 的多个核心共享 L2 以及 L3 缓存，另外，某些 CPU 还可以通过超线程技术（Hyper-Threading Technology）使得一个核心具有同时执行两个线程的能力。

SMP 架构的主要特征是共享，系统中所有资源（CPU、内存、I/O 等）都是共享的，由于多 CPU 对前端总线的竞争，SMP 的扩展能力非常有限。为了提高可扩展性，现在的主流服务器架构一般为 NUMA（Non-Uniform Memory Access，非一致存储访问）架构。它具有多个 NUMA 节点，每个 NUMA 节点是一个 SMP 结构，一般由多个 CPU（如 4 个）组成，并且具有独立的本地内存、IO 槽口等。

图 2-2 为包含 4 个 NUMA 节点的服务器架构图，NUMA 节点可以直接快速访问本地内存，也可以通过 NUMA 互联互通模块访问其他 NUMA 节点的内存，访问本地内存的速度远远高于远程访问的速度。由于这个特点，为了更好地发挥系统性能，开发应用程序时需要尽量减少不同 NUMA 节点之间的信息交互。

2.1.2　IO 总线

存储系统的性能瓶颈一般在于 IO，因此，有必要对 IO 子系统的架构有一个大致的了解。以 Intel x48 主板为例，它是典型的南、北桥架构，如图 2-3 所示。北桥芯片通过前端总线（Front Side Bus，FSB）与 CPU 相连，内存模块以及 PCI-E 设备（如高端的 SSD 设备 Fusion-IO）挂接在北桥上。北桥与南桥之间通过 DMI 连接，DMI 的带宽

为 1GB/s，网卡（包括千兆以及万兆网卡），硬盘以及中低端固态盘（如 Intel 320 系列 SSD）挂接在南桥上。如果采用 SATAZ 接口，那么最大带宽为 300MB/s。

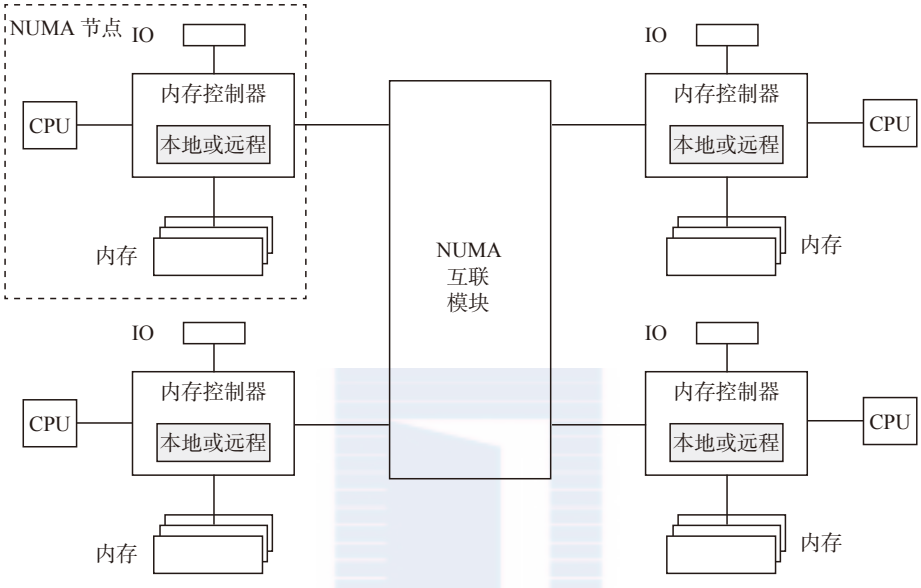

图 2-2 NUMA 架构示例

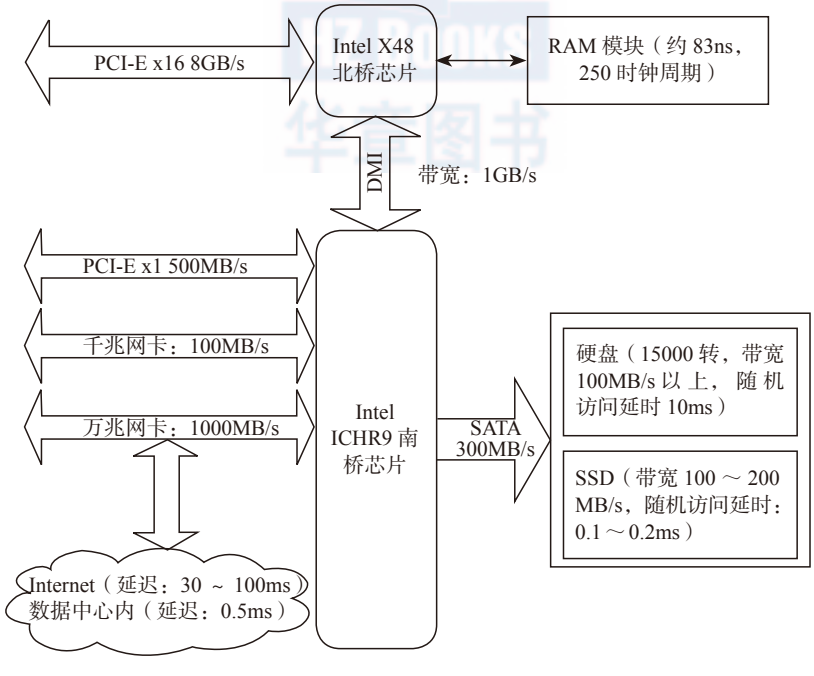

图 2-3 Intel X48 主板南北桥架构

2.1.3　网络拓扑

图 2-4 为传统的数据中心网络拓扑，思科过去一直提倡这样的拓扑，分为三层，最下面是接入层（Edge），中间是汇聚层（Aggregation），上面是核心层（Core）。典型的接入层交换机包含 48 个 1Gb 端口以及 4 个 10Gb 上行端口，汇聚层以及核心层的交换机包含 128 个 10Gb 的端口。传统三层结构的问题在于可能有很多接入层的交换机接到汇聚层，很多的汇聚层交换机接到核心层。同一个接入层下的服务器之间带宽为 1Gb，不同接入层交换机下的服务器之间的带宽小于 1Gb。由于同一个接入层的服务器往往部署在一个机架内，因此，设计系统的时候需要考虑服务器是否在一个机架内，减少跨机架拷贝大量数据。例如，Hadoop HDFS 默认存储三个副本，其中两个副本放在同一个机架，就是这个原因。

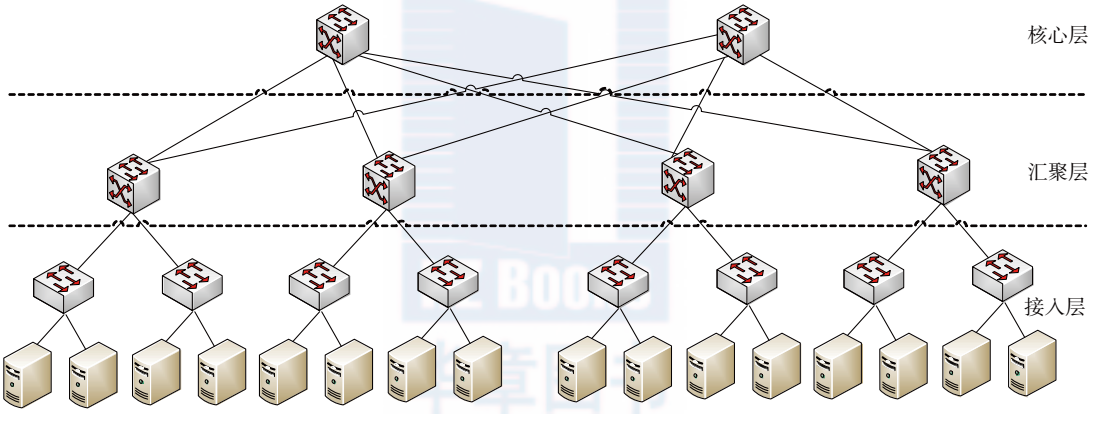

图 2-4　数据中心网络拓扑（三层结构）

为了减少系统对网络拓扑结构的依赖，Google 在 2008 年的时候将网络改造为扁平化拓扑结构，即三级 CLOS 网络，同一个集群内最多支持 20480 台服务器，且任何两台都有 1Gb 带宽。CLOS 网络需要额外投入更多的交换机，带来的好处也是明显的，设计系统时不需要考虑底层网络拓扑，从而很方便地将整个集群做成一个计算资源池。

同一个数据中心内部的传输延时是比较小的，网络一次来回的时间在 1 毫秒之内。数据中心之间的传输延迟是很大的，取决于光在光纤中的传输时间。例如，北京与杭州之间的直线距离大约为 1300 公里，光在信息传输中走折线，假设折线距离为直线距离的 1.5 倍，那么光传输一次网络来回延时的理论值为 1300 × 1.5 ×2 / 300 000= 13 毫秒，实际测试值大约为 40 毫秒。

2.1.4 性能参数

常见硬件的大致性能参数如表 2-1 所示。

表 2-1 常用硬件性能参数

类 别	消耗的时间
访问 L1 Cache	0.5 ns
分支预测失败	5 ns
访问 L2 Cache	7 ns
Mutex 加锁 / 解锁	100 ns
内存访问	100 ns
千兆网络发送 1MB 数据	10 ms
从内存顺序读取 1MB 数据	0.25 ms
机房内网络来回	0.5 ms
异地机房之间网络来回	30~100 ms
SATA 磁盘寻道	10 ms
从 SATA 磁盘顺序读取 1MB 数据	20 ms
固态盘 SSD 访问延迟	0.1~0.2 ms

磁盘读写带宽还是不错的，15000 转的 SATA 盘的顺序读取带宽可以达到 100MB 以上，由于磁盘寻道的时间大约为 10ms，顺序读取 1MB 数据的时间为：磁盘寻道时间 + 数据读取时间，即 10ms + 1MB / 100MB/s×1000 = 20ms。存储系统的性能瓶颈主要在于磁盘随机读写。设计存储引擎的时候会针对磁盘的特性做很多的处理，比如将随机写操作转化为顺序写，通过缓存减少磁盘随机读操作。

固态磁盘（SSD）在最近几年得到越来越多的关注，各大互联网公司都有大量基于 SSD 的应用。SSD 的特点是随机读取延迟小，能够提供很高的 IOPS（每秒读写，Input/Output Per Second）性能。它的主要问题在于容量和价格，设计存储系统的时候一般可以用来做缓存或者性能要求较高的关键业务。

不同的持久化存储介质对比如表 2-2 所示。

表 2-2 存储介质对比

类别	每秒读写（IOPS）次数	每 GB 价格（元）	随机读取	随机写入
内存	千万级	150	友好	友好
SSD 盘	35000	20	友好	写入放大问题
SAS 磁盘	180	3	磁盘寻道	磁盘寻道
SATA 磁盘	90	0.5	磁盘寻道	磁盘寻道

从表 2-2 可以看出，SSD 单位成本提供的 IOPS 比传统的 SAS 或者 SATA 磁盘都要大得多，而且 SSD 功耗低，更加环保，适合小数据量并且对性能要求更高的场景。

2.1.5　存储层次架构

从分布式系统的角度看，整个集群中所有服务器上的存储介质（内存、机械硬盘，SSD）构成一个整体，其他服务器上的存储介质与本机存储介质一样都是可访问的，区别仅仅在于需要额外的网络传输及网络协议栈等访问开销。

如图 2-5 所示，假设集群中有 30 个机架，每个机架接入 40 台服务器，同一个机架的服务器接入到同一个接入交换机，不同机架的服务器接入到不同的接入交换机。每台服务器的内存为 24GB，磁盘为 10×1TB 的 SATA 机械硬盘（15000 转）或者 10×160GB 的 SSD 固态硬盘。那么，对于每台服务器，本地内存大小为 24GB，访问延时为 100ns，本地 SATA 磁盘的大小为 4TB（假设利用率为 40%），随机访问的寻道时间为 10ms，本地 SSD 磁盘的大小为 1TB（假设利用率为 60%），访问延时为 0.1ms，SATA 磁盘和 SSD 的访问带宽受限于 SATA 接口，最大不超过 300MB/s。同一个机架下的服务器的内存总量大致为 1TB，访问延时和带宽受限于网络，访问延时大约为 300μs，带宽为 100MB/s，磁盘总容量为 160TB，访问延时为网络延时加上磁盘寻道时间，大约为 11ms，SSD 容量为 40TB，访问延时为网络延时加上 SSD 访问延时，大约为 2ms。整个集群下所有服务器的内存总量为 30TB，访问延时和带宽受限于网络，跨机架访问需要经过聚合层或者核心层的交换机，访问延时大约为 500μs，带宽大约为 10MB/s，磁盘和 SSD 的访问延时分别为 11ms 以及 2ms，带宽为 10MB/s。

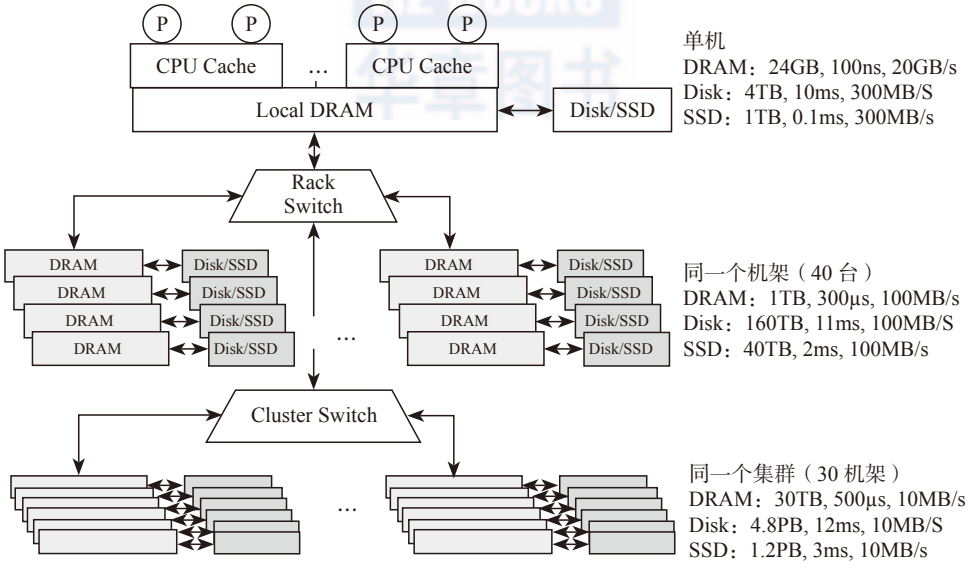

图 2-5　存储层次结构图

存储系统的性能主要包括两个维度：吞吐量以及访问延时，设计系统时要求能够在保证访问延时的基础上，通过最低的成本实现尽可能高的吞吐量。磁盘和 SSD 的访问延时差别很大，但带宽差别不大，因此，磁盘适合大块顺序访问的存储系统，SSD 适合随机访问较多或者对延时比较敏感的关键系统。二者也常常组合在一起进行混合存储，热数据（访问频繁）存储到 SSD 中，冷数据（访问不频繁）存储到磁盘中。

2.2 单机存储引擎

存储引擎是存储系统的发动机，直接决定了存储系统能够提供的性能和功能。存储系统的基本功能包括：增、删、读、改，其中，读取操作又分为随机读取和顺序扫描。哈希存储引擎是哈希表的持久化实现，支持增、删、改，以及随机读取操作，但不支持顺序扫描，对应的存储系统为键值（Key-Value）存储系统；B 树（B-Tree）存储引擎是 B 树的持久化实现，不仅支持单条记录的增、删、读、改操作，还支持顺序扫描，对应的存储系统是关系数据库。当然，键值系统也可以通过 B 树存储引擎实现；LSM 树（Log-Structured Merge Tree）存储引擎和 B 树存储引擎一样，支持增、删、改、随机读取以及顺序扫描。它通过批量转储技术规避磁盘随机写入问题，广泛应用于互联网的后台存储系统，例如 Google Bigtable、Google LevelDB 以及 Facebook 开源的 Cassandra 系统。本节分别以 Bitcask、MySQL InnoDB 以及 Google LevelDB 系统为例介绍这三种存储引擎。

2.2.1 哈希存储引擎

Bitcask 是一个基于哈希表结构的键值存储系统，它仅支持追加操作（Append-only），即所有的写操作只追加而不修改老的数据。在 Bitcask 系统中，每个文件有一定的大小限制，当文件增加到相应的大小时，就会产生一个新的文件，老的文件只读不写。在任意时刻，只有一个文件是可写的，用于数据追加，称为活跃数据文件（active data file）。而其他已经达到大小限制的文件，称为老数据文件（older data file）。

1. 数据结构

如图 2-6 所示，Bitcask 数据文件中的数据是一条一条的写入操作，每一条记录的数据项分别为主键（key）、value 内容（value）、主键长度（key_sz）、value 长度（value_sz）、时间戳（timestamp）以及 crc 校验值。（数据删除操作也不会删除旧的条目，而是将 value 设定为一个特殊的值用作标识）。内存中采用基于哈希表的索引数据结构，哈希表的作用是通过主键快速地定位到 value 的位置。哈希表结构中的每一项包含了三个用于定位数据的信息，分别是文件编号（file id），value 在文件中的位置（value_pos），

value 长度（value_sz），通过读取 file_id 对应文件的 value_pos 开始的 value_sz 个字节，这就得到了最终的 value 值。写入时首先将 Key-Value 记录追加到活跃数据文件的末尾，接着更新内存哈希表，因此，每个写操作总共需要进行一次顺序的磁盘写入和一次内存操作。

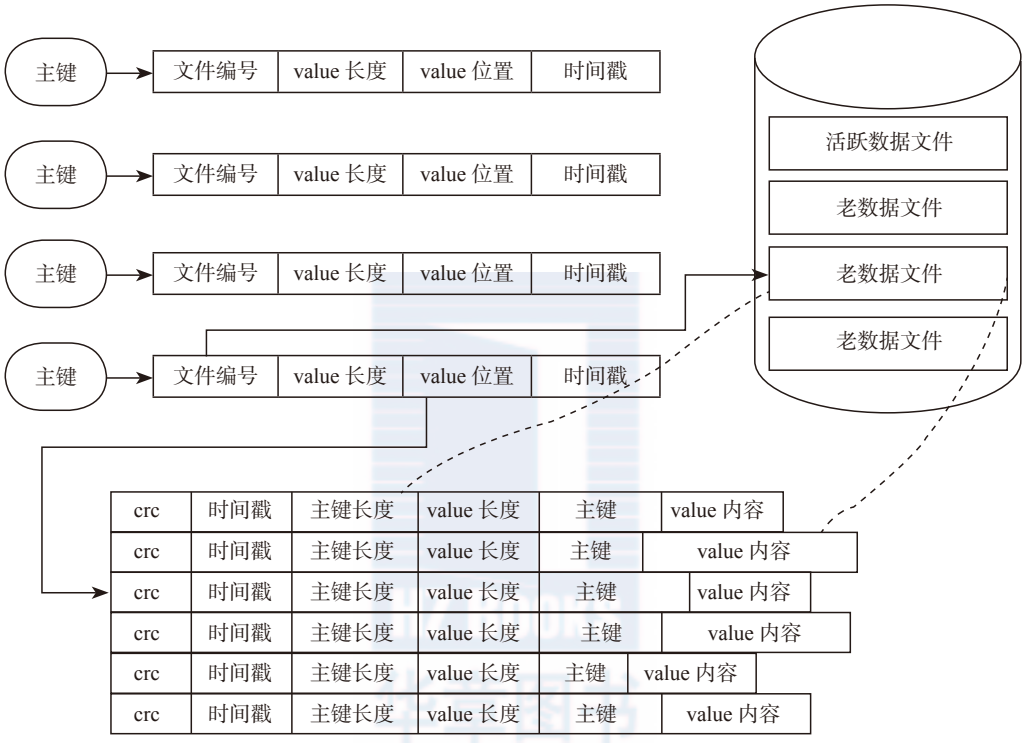

图 2-6　Bitcask 数据结构

Bitcask 在内存中存储了主键和 value 的索引信息，磁盘文件中存储了主键和 value 的实际内容。系统基于一个假设，value 的长度远大于主键的长度。假如 value 的平均长度为 1KB，每条记录在内存中的索引信息为 32 字节，那么，磁盘内存比为 32∶1。这样，32GB 内存索引的数据量为 32GB×32 = 1TB。

2. 定期合并

Bitcask 系统中的记录删除或者更新后，原来的记录成为垃圾数据。如果这些数据一直保存下去，文件会无限膨胀下去，为了解决这个问题，Bitcask 需要定期执行合并（Compaction）操作以实现垃圾回收。所谓合并操作，即将所有老数据文件中的数据扫描一遍并生成新的数据文件，这里的合并其实就是对同一个 key 的多个操作以只保留最新一个的原则进行删除，每次合并后，新生成的数据文件就不再有冗余数据了。

3. 快速恢复

Bitcask 系统中的哈希索引存储在内存中，如果不做额外的工作，服务器断电重启重建哈希表需要扫描一遍数据文件，如果数据文件很大，这是一个非常耗时的过程。Bitcask 通过索引文件（hint file）来提高重建哈希表的速度。

简单来说，索引文件就是将内存中的哈希索引表转储到磁盘生成的结果文件。Bitcask 对老数据文件进行合并操作时，会产生新的数据文件，这个过程中还会产生一个索引文件，这个索引文件记录每一条记录的哈希索引信息。与数据文件不同的是，索引文件并不存储具体的 value 值，只存储 value 的位置（与内存哈希表一样）。这样，在重建哈希表时，就不需要扫描所有数据文件，而仅仅需要将索引文件中的数据一行行读取并重建即可，大大减少了重启后的恢复时间。

2.2.2 B 树存储引擎

相比哈希存储引擎，B 树存储引擎不仅支持随机读取，还支持范围扫描。关系数据库中通过索引访问数据，在 Mysql InnoDB 中，有一个称为聚集索引的特殊索引，行的数据存于其中，组织成 B+ 树（B 树的一种）数据结构。

1. 数据结构

如图 2-7 所示，MySQL InnoDB 按照页面（Page）来组织数据，每个页面对应 B+ 树的一个节点。其中，叶子节点保存每行的完整数据，非叶子节点保存索引信息。数据在每个节点中有序存储，数据库查询时需要从根节点开始二分查找直到叶子节点，每次

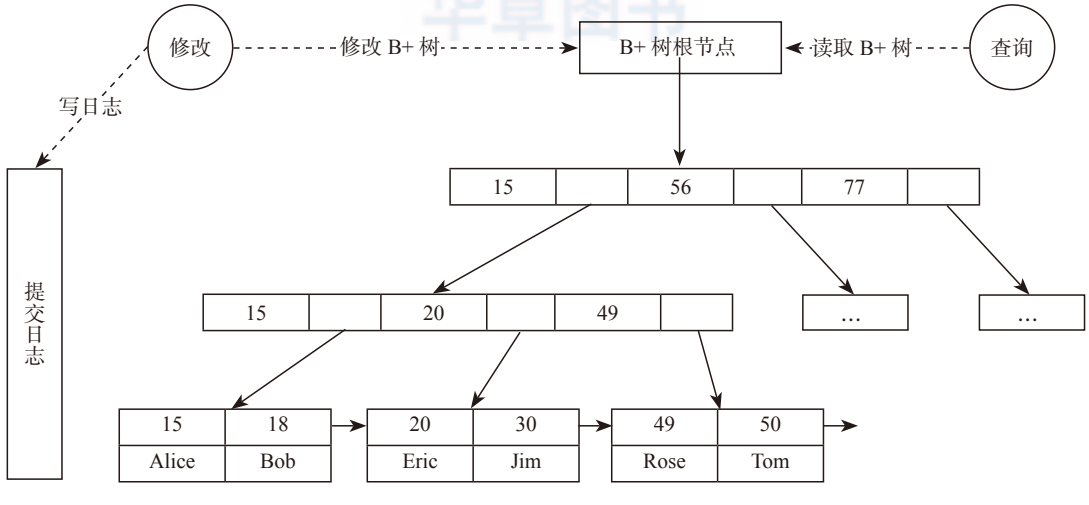

图 2-7 B+ 树存储引擎

读取一个节点，如果对应的页面不在内存中，需要从磁盘中读取并缓存起来。B+ 树的根节点是常驻内存的，因此，B+ 树一次检索最多需要 h−1 次磁盘 IO，复杂度为 $O(h)=O(\log_d N)$（N 为元素个数，d 为每个节点的出度，h 为 B+ 树高度）。修改操作首先需要记录提交日志，接着修改内存中的 B+ 树。如果内存中的被修改过的页面超过一定的比率，后台线程会将这些页面刷到磁盘中持久化。当然，InnoDB 实现时做了大量的优化，这部分内容已经超出了本书的范围。

2. 缓冲区管理

缓冲区管理器负责将可用的内存划分成缓冲区，缓冲区是与页面同等大小的区域，磁盘块的内容可以传送到缓冲区中。缓冲区管理器的关键在于替换策略，即选择将哪些页面淘汰出缓冲池。常见的算法有以下两种。

（1）LRU

LRU 算法淘汰最长时间没有读或者写过的块。这种方法要求缓冲区管理器按照页面最后一次被访问的时间组成一个链表，每次淘汰链表尾部的页面。直觉上，长时间没有读写的页面比那些最近访问过的页面有更小的最近访问的可能性。

（2）LIRS

LRU 算法在大多数情况下表现是不错的，但有一个问题：假如某一个查询做了一次全表扫描，将导致缓冲池中的大量页面（可能包含很多很快被访问的热点页面）被替换，从而污染缓冲池。现代数据库一般采用 LIRS 算法，将缓冲池分为两级，数据首先进入第一级，如果数据在较短的时间内被访问两次或者以上，则成为热点数据进入第二级，每一级内部还是采用 LRU 替换算法。Oracle 数据库中的 Touch Count 算法和 MySQL InnoDB 中的替换算法都采用了类似的分级思想。以 MySQL InnoDB 为例，InnoDB 内部的 LRU 链表分为两部分：新子链表（new sublist）和老子链表（old sublist），默认情况下，前者占 5/8，后者占 3/8。页面首先插入到老子链表，InnoDB 要求页面在老子链表停留时间超过一定值，比如 1 秒，才有可能被转移到新子链表。当出现全表扫描时，InnoDB 将数据页面载入到老子链表，由于数据页面在老子链表中的停留时间不够，不会被转移到新子链表中，这就避免了新子链表中的页面被替换出去的情况。

2.2.3　LSM 树存储引擎

LSM 树（Log Structured Merge Tree）的思想非常朴素，就是将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量写入磁盘，读取时需要合并磁盘中的历史数据和内存中最近的修改操作。LSM 树的优势在于有效地规避了磁盘随机写入问题，但读取时可能需要访问较多的磁盘文件。本节介绍 LevelDB 中的 LSM 树存储引擎。

1. 存储结构

如图 2-8 所示，LevelDB 存储引擎主要包括：内存中的 MemTable 和不可变 MemTable（Immutable MemTable，也称为 Frozen MemTable，即冻结 MemTable）以及磁盘上的几种主要文件：当前（Current）文件、清单（Manifest）文件、操作日志（Commit Log，也称为提交日志）文件以及 SSTable 文件。当应用写入一条记录时，LevelDB 会首先将修改操作写入到操作日志文件，成功后再将修改操作应用到 MemTable，这样就完成了写入操作。

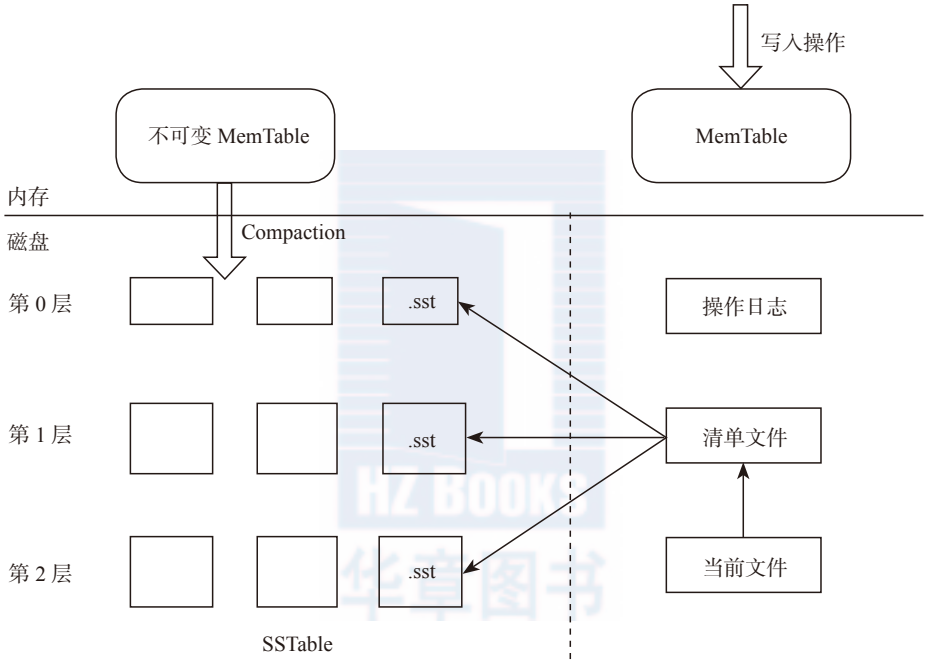

图 2-8　LevelDB 存储引擎

当 MemTable 占用的内存达到一个上限值后，需要将内存的数据转储到外存文件中。LevelDB 会将原先的 MemTable 冻结成为不可变 MemTable，并生成一个新的 MemTable。新到来的数据被记入新的操作日志文件和新生成的 MemTable 中。顾名思义，不可变 MemTable 的内容是不可更改的，只能读取不能写入或者删除。LevelDB 后台线程会将不可变 MemTable 的数据排序后转储到磁盘，形成一个新的 SSTable 文件，这个操作称为 Compaction。SSTable 文件是内存中的数据不断进行 Compaction 操作后形成的，且 SSTable 的所有文件是一种层级结构，第 0 层为 Level 0，第 1 层为 Level 1，以此类推。

SSTable 中的文件是按照记录的主键排序的，每个文件有最小的主键和最大的主键。LevelDB 的清单文件记录了这些元数据，包括属于哪个层级、文件名称、最小主键和

最大主键。当前文件记录了当前使用的清单文件名。在 LevelDB 的运行过程中，随着 Compaction 的进行，SSTable 文件会发生变化，新的文件会产生，老的文件被废弃，此时往往会生成新的清单文件来记载这种变化，而当前文件则用来指出哪个清单文件才是当前有效的。

直观上，LevelDB 每次查询都需要从老到新读取每个层级的 SSTable 文件以及内存中的 MemTable。LevelDB 做了一个优化，由于 LevelDB 对外只支持随机读取单条记录，查询时 LevelDB 首先会去查看内存中的 MemTable，如果 MemTable 包含记录的主键及其对应的值，则返回记录即可；如果 MemTable 没有读到该主键，则接下来到同样处于内存中的不可变 Memtable 中去读取；类似地，如果还是没有读到，只能依次从新到老读取磁盘中的 SSTable 文件。

2. 合并

LevelDB 写入操作很简单，但是读取操作比较复杂，需要在内存以及各个层级文件中按照从新到老依次查找，代价很高。为了加快读取速度，LevelDB 内部会执行 Compaction 操作来对已有的记录进行整理压缩，从而删除一些不再有效的记录，减少数据规模和文件数量。

LevelDB 的 Compaction 操作分为两种：minor compaction 和 major compaction。Minor compaction 是指当内存中的 MemTable 大小到了一定值时，将内存数据转储到 SSTable 文件中。每个层级下有多个 SSTable，当某个层级下的 SSTable 文件数目超过一定设置值后，levelDB 会从这个层级中选择 SSTable 文件，将其和高一层级的 SSTable 文件合并，这就是 major compaction。major compaction 相当于执行一次多路归并：按照主键顺序依次迭代出所有 SSTable 文件中的记录，如果没有保存价值，则直接抛弃；否则，将其写入到新生成的 SSTable 文件中。

2.3　数据模型

如果说存储引擎相当于存储系统的发动机，那么，数据模型就是存储系统的外壳。存储系统的数据模型主要包括三类：文件、关系以及随着 NoSQL 技术流行起来的键值模型。传统的文件系统和关系数据库系统分别采用文件和关系模型。关系模型描述能力强，产业链完整，是存储系统的业界标准。然而，随着应用在可扩展性、高并发以及性能上提出越来越高的要求，大而全的关系数据库有时显得力不从心，因此，产生了一些新的数据模型，比如键值模型，关系弱化的表格模型，等等。

2.3.1　文件模型

文件系统以目录树的形式组织文件，以类 UNIX 操作系统为例，根目录为 /，包

含 /usr、/bin、/home 等子目录，每个子目录又包含其他子目录或者文件。文件系统的操作涉及目录以及文件，例如，打开 / 关闭文件、读写文件、遍历目录、设置文件属性等。POSIX（Portable Operating System Interface）是应用程序访问文件系统的 API 标准，它定义了文件系统存储接口及操作集。POSIX 主要接口如下所示。

- ❑ Open/close：打开 / 关闭一个文件，获取文件描述符；
- ❑ Read/write：读取一个文件或者往文件中写入数据；
- ❑ Opendir/closedir：打开或者关闭一个目录；
- ❑ Readdir：遍历目录。

POSIX 标准不仅定义了文件操作接口，而且还定义了读写操作语义。例如，POSIX 标准要求读写并发时能够保证操作的原子性，即读操作要么读到所有结果，要么什么也读不到；另外，要求读操作能够读到之前所有写操作的结果。POSIX 标准适合单机文件系统，在分布式文件系统中，出于性能考虑，一般不会完全遵守这个标准。NFS（Network File System）文件系统允许客户端缓存文件数据，多个客户端并发修改同一个文件时可能出现不一致的情况。举个例子，NFS 客户端 A 和 B 需要同时修改 NFS 服务器的某个文件，每个客户端都在本地缓存了文件的副本，A 修改后先提交，B 后提交，那么，即使 A 和 B 修改的是文件的不同位置，也会出现 B 的修改覆盖 A 的情况。

对象模型与文件模型比较类似，用于存储图片、视频、文档等二进制数据块，典型的系统包括 Amazon Simple Storage（S3），Taobao File System（TFS）。这些系统弱化了目录树的概念，Amazon S3 只支持一级目录，不支持子目录，Taobao TFS 甚至不支持目录结构。与文件模型不同的是，对象模型要求对象一次性写入到系统，只能删除整个对象，不允许修改其中某个部分。

2.3.2 关系模型

每个关系是一个表格，由多个元组（行）构成，而每个元组又包含多个属性（列）。关系名、属性名以及属性类型称作该关系的模式（schema）。例如，Movie 关系的模式为 Movie（title，year，length），其中，title、year、length 是属性，假设它们的类型分别为字符串、整数、整数。

数据库语言 SQL 用于描述查询以及修改操作。数据库修改包含三条命令：INSERT、DELETE 以及 UPDATE，查询通常通过 select-from-where 语句来表达，它具有图 2-9 所示的一般形式。Select 查询语句计算过程大致如下（不考虑查询优化）：

```
SELECT      <属性表>
FROM        <关系表>
WHERE       <条件>
GROUP BY    <属性表>
HAVING      <条件>
ORDER BY    <属性表>
```

图 2-9 SQL 查询

1）取 FROM 子句中列出的各个关系的元组的所有可能的组合。

2）将不符合 WHERE 子句中给出的条件的元组去掉。

3）如果有 GROUP BY 子句，则将剩下的元组按 GROUP BY 子句中给出的属性的值分组。

4）如果有 HAVING 子句，则按照 HAVING 子句中给出的条件检查每一个组，去掉不符合条件的组。

5）按照 SELECT 子句的说明，对于指定的属性和属性上的聚集（例如求和）计算出结果元组。

6）按照 ORDER BY 子句中的属性列的值对结果元组进行排序。

SQL 查询还有一个强大的特性是允许在 WHERE、FROM 和 HAVING 子句中使用子查询，子查询又是一个完整的 select-from-where 语句。

另外，SQL 还包括两个重要的特性：索引以及事务。其中，数据库索引用于减少 SQL 执行时扫描的数据量，提高读取性能；数据库事务则规定了各个数据库操作的语义，保证了多个操作并发执行时的 ACID 特性（原子性、一致性、隔离性、持久性），后续会专门介绍。

2.3.3　键值模型

大量的 NoSQL 系统采用了键值模型（也称为 Key-Value 模型），每行记录由主键和值两个部分组成，支持基于主键的如下操作：

- Put：保存一个 Key-Value 对。
- Get：读取一个 Key-Value 对。
- Delete：删除一个 Key-Value 对。

Key-Value 模型过于简单，支持的应用场景有限，NoSQL 系统中使用比较广泛的模型是表格模型。表格模型弱化了关系模型中的多表关联，支持基于单表的简单操作，典型的系统是 Google Bigtable 以及其开源 Java 实现 HBase。表格模型除了支持简单的基于主键的操作，还支持范围扫描，另外，也支持基于列的操作。主要操作如下：

- Insert：插入一行数据，每行包括若干列；
- Delete：删除一行数据；
- Update：更新整行或者其中的某些列的数据；
- Get：读取整行或者其中某些列数据；
- Scan：扫描一段范围的数据，根据主键确定扫描的范围，支持扫描部分列，支持按列过滤、排序、分组等。

与关系模型不同的是，表格模型一般不支持多表关联操作，Bigtable 这样的系统也不支持二级索引，事务操作支持也比较弱，各个系统支持的功能差异较大，没有统一的标准。另外，表格模型往往还支持无模式（schema-less）特性，也就是说，不需要预先定义每行包括哪些列以及每个列的类型，多行之间允许包含不同列。

2.3.4 SQL 与 NoSQL

随着互联网的飞速发展，数据规模越来越大，并发量越来越高，传统的关系数据库有时显得力不从心，非关系型数据库（NoSQL，Not Only SQL）应运而生。NoSQL 系统带来了很多新的理念，比如良好的可扩展性，弱化数据库的设计范式，弱化一致性要求，在一定程度上解决了海量数据和高并发的问题，以至于很多人对“NoSQL 是否会取代 SQL”存在疑虑。然而，NoSQL 只是对 SQL 特性的一种取舍和升华，使得 SQL 更加适应海量数据的应用场景，二者的优势将不断融合，不存在谁取代谁的问题。

关系数据库在海量数据场景面临如下挑战：

- **事务** 关系模型要求多个 SQL 操作满足 ACID 特性，所有的 SQL 操作要么全部成功，要么全部失败。在分布式系统中，如果多个操作属于不同的服务器，保证它们的原子性需要用到两阶段提交协议，而这个协议的性能很低，且不能容忍服务器故障，很难应用在海量数据场景。
- **联表** 传统的数据库设计时需要满足范式要求，例如，第三范式要求在一个关系中不能出现在其他关系中已包含的非主键信息。假设存在一个部门信息表，其中每个部门有部门编号、部门名称、部门简介等信息，那么在员工信息表中列出部门编号后就不能加入部门名称、部门简介等部门有关的信息，否则就会有大量的数据冗余。而在海量数据的场景，为了避免数据库多表关联操作，往往会使用数据冗余等违反数据库范式的手段。实践表明，这些手段带来的收益远高于成本。
- **性能** 关系数据库采用 B 树存储引擎，更新操作性能不如 LSM 树这样的存储引擎。另外，如果只有基于主键的增、删、查、改操作，关系数据库的性能也不如专门定制的 Key-Value 存储系统。

随着数据规模越来越大，可扩展性以及性能提升可以带来越来越明显的收益，而 NoSQL 系统要么可扩展性好，要么在特定的应用场景性能很高，广泛应用于互联网业务中。然而，NoSQL 系统也面临如下问题：

- **缺少统一标准**。经过几十年的发展，关系数据库已经形成了 SQL 语言这样的业界标准，并拥有完整的生态链。然而，各个 NoSQL 系统使用方法不同，切换成本高，很难通用。

- **使用以及运维复杂**。NoSQL 系统无论是选型，还是使用方式，都有很大的学问，往往需要理解系统的实现，另外，缺乏专业的运维工具和运维人员。而关系数据库具有完整的生态链和丰富的运维工具，也有大量经验丰富的运维人员。

总而言之，关系数据库很通用，是业界标准，但是在一些特定的应用场景存在可扩展性和性能的问题，NoSQL 系统也有一定的用武之地。从技术学习的角度看，不必纠结 SQL 与 NoSQL 的区别，而是借鉴二者各自不同的优势，着重理解关系数据库的原理以及 NoSQL 系统的高可扩展性。

2.4　事务与并发控制

事务规范了数据库操作的语义，每个事务使得数据库从一个一致的状态原子地转移到另一个一致的状态。数据库事务具有原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）以及持久性（Durability），即 ACID 属性，这些特性使得多个数据库事务并发执行时互不干扰，也不会获取到中间状态的错误结果。

多个事务并发执行时，如果它们的执行结果和按照某种顺序一个接着一个串行执行的效果等同，这种隔离级别称为可串行化。可串行化是比较理想的情况，商业数据库为了性能考虑，往往会定义多种隔离级别。事务的并发控制一般通过锁机制来实现，锁可以有不同的粒度，可以锁住行，也可以锁住数据块甚至锁住整个表格。由于互联网业务中读事务的比例往往远远高于写事务，为了提高读事务性能，可以采用写时复制（Copy-On-Write，COW）或者多版本并发控制（Multi-Version Concurrency Control，MVCC）技术来避免写事务阻塞读事务。

2.4.1　事务

事务是数据库操作的基本单位，它具有原子性、一致性、隔离性和持久性这四个基本属性。

（1）原子性

事务的原子性首先体现在事务对数据的修改，即要么全都执行，要么全都不执行，例如，从银行账户 A 转一笔款项 a 到账户 B，结果必须是从 A 的账户上扣除款项 a 并且在 B 的账户上增加款项 a，不能只是其中一个账户的修改。但是，事务的原子性并不总是能够保证修改一定完成了或者一定没有进行，例如，在 ATM 机器上进行上述转账，转账指令提交后通信中断或者数据库主机异常了，那么转账可能完成了也可能没有进行：如果通信中断发生前数据库主机完整接收到了转账指令且后续执行也正常，那么转账成功完成了；如果转账指令没有到达数据库主机或者虽然到达但后续执行异常（例如

写操作日志失败或者账户余额不足），那么转账就没有进行。要确定转账是否成功，需要待通信恢复或者数据库主机恢复后查询账户交易历史或余额。事务的原子性也体现在事务对数据的读取上，例如，一个事务对同一数据项的多次读取的结果一定是相同的。

（2）一致性

事务需要保持数据库数据的正确性、完整性和一致性，有些时候这种一致性由数据库的内部规则保证，例如数据的类型必须正确，数据值必须在规定的范围内，等等；另外一些时候这种一致性由应用保证，例如一般情况下银行账务余额不能是负数，信用卡消费不能超过该卡的信用额度等。

（3）隔离性

许多时候数据库在并发执行多个事务，每个事务可能需要对多个表项进行修改和查询，与此同时，更多的查询请求可能也在执行中。数据库需要保证每一个事务在它的修改全部完成之前，对其他的事务是不可见的，换句话说，不能让其他事务看到该事务的中间状态，例如，从银行账户 A 转一笔款项 a 到账户 B，不能让其他事务（例如账户查询）看到 A 账户已经扣除款项 a 但 B 账户却还没有增加款项 a 的状态。

（4）持久性

事务完成后，它对于数据库的影响是永久性的，即使系统出现各种异常也是如此。

出于性能考虑，许多数据库允许使用者选择牺牲隔离属性来换取并发度，从而获得性能的提升。SQL 定义了 4 种隔离级别。

- Read Uncommitted (RU)：读取未提交的数据，即其他事务已经修改但还未提交的数据，这是最低的隔离级别；
- Read Committed (RC)：读取已提交的数据，但是，在一个事务中，对同一个项，前后两次读取的结果可能不一样，例如第一次读取时另一个事务的修改还没有提交，第二次读取时已经提交了；
- Repeatable Read (RR)：可重复读取，在一个事务中，对同一个项，确保前后两次读取的结果一样；
- Serializable (S)：可序列化，即数据库的事务是可串行化执行的，就像一个事务执行的时候没有别的事务同时在执行，这是最高的隔离级别。

隔离级别的降低可能导致读到脏数据或者事务执行异常，例如：

- Lost Update (LU)：第一类丢失更新：两个事务同时修改一个数据项，但后一个事务中途失败回滚，则前一个事务已提交的修改都可能丢失；
- Dirty Reads (DR)：一个事务读取了另外一个事务更新却没有提交的数据项；
- Non-Repeatable Reads (NRR)：一个事务对同一数据项的多次读取可能得到不同的结果；

- Second Lost Updates problem (SLU)：第二类丢失更新：两个并发事务同时读取和修改同一数据项，则后面的修改可能使得前面的修改失效；
- Phantom Reads (PR)：事务执行过程中，由于前面的查询和后面的查询的期间有另外一个事务插入数据，后面的查询结果出现了前面查询结果中未出现的数据。

表 2-3 说明了隔离级别与读写异常 (不一致) 的关系。容易发现，所有的隔离级别都保证不会出现第一类丢失更新，另外，在最高隔离级别（Serializable）下，数据不会出现读写的不一致。

表 2-3　隔离级别与读写异常的关系

	LU	DR	NRR	SLU	PR
RU	N	Y	Y	Y	Y
RC	N	N	Y	Y	Y
RR	N	N	N	N	Y
S	N	N	N	N	N

2.4.2　并发控制

1. 数据库锁

事务分为几种类型：读事务，写事务以及读写混合事务。相应地，锁也分为两种类型：读锁以及写锁，允许对同一个元素加多个读锁，但只允许加一个写锁，且写事务将阻塞读事务。这里的元素可以是一行，也可以是一个数据块甚至一个表格。事务如果只操作一行，可以对该行加相应的读锁或者写锁；如果操作多行，需要锁住整个行范围。

表 2-4 中 T1 和 T2 两个事务操作不同行，初始时 A = B = 25，T1 将 A 加 100，T2 将 B 乘以 2，由于 T1 和 T2 操作不同行，两个事务没有锁冲突，可以并行执行而不会破坏系统的一致性。

表 2-4　两个事务操作不同行

T1	T2	A	B
READ(A, t)		25	25
t := t + 100	READ(B, s)		
WRITE(A,t)	s = s * 2;	125	
	WRITE(B, s)		50

表 2-5 中 T1 扫描从 A 到 C 的所有行，将它们的结果相加后更新 A，初始时 A = C = 25，假设在 T1 执行过程中 T2 插入一行 B，那么，事务 T1 和 T2 无法做到可串行化。为了保证数据库一致性，T1 执行范围扫描时需要锁住从 A 到 C 这个范围的所有更新，T2 插入 B 时，由于整个范围被锁住，T2 获取锁失败而等待 T1 先执行完成。

表 2-5 事务读取一段数据范围

T1	T2	A	B	C
SCAN([A=>C], {t1,t3})		25		25
t = t1+t3	INSERT(B, t2)		25	
WRITE(A, t)		50		

多个事务并发执行可能引入死锁。表 2-6 中 T1 读取 A，然后将 A 的值加 100 后更新 B，T2 读取 B，然后将 B 的值乘以 2 更新 A，初始时 A = B = 25。T1 持有 A 的读锁，需要获取 B 的写锁，而 T2 持有 B 的读锁，需要 A 的写锁。T1 和 T2 这两个事务循环依赖，任何一个事务都无法顺利完成。

表 2-6 数据库死锁

T1	T2	A	B
READ(A, t)		25	25
t := t + 100	READ(B, s)		
WRITE(B,t)	s = s * 2;		
	WRITE(A, s)		

解决死锁的思路主要有两种：第一种思路是为每个事务设置一个超时时间，超时后自动回滚，表 2-6 中如果 T1 或 T2 二者之中的某个事务回滚，则另外一个事务可以成功执行。第二种思路是死锁检测。死锁出现的原因在于事务之间互相依赖，T1 依赖 T2，T2 又依赖 T1，依赖关系构成一个环路。检测到死锁后可以通过回滚其中某些事务来消除循环依赖。

2. 写时复制

互联网业务中读事务占的比例往往远远超过写事务，很多应用的读写比例达到 6:1，甚至 10:1。写时复制（Copy-On-Write，COW）读操作不用加锁，极大地提高了读取性能。

图 2-10 中写时复制 B+ 树执行写操作的步骤如下。

1）拷贝：将从叶子到根节点路径上的所有节点拷贝出来。

2）修改：对拷贝的节点执行修改。

3）提交：原子地切换根节点的指针，使之指向新的根节点。

如果读操作发生在第 3 步提交之前，那么，将读取老节点的数据，否则将读取新节点，读操作不需要加锁保护。写时复制技术涉及引用计数，对每个节点维护一个引用计数，表示被多少节点引用，如果引用计数变为 0，说明没有节点引用，可以被垃圾回收。

写时复制技术原理简单，问题是每次写操作都需要拷贝从叶子到根节点路径上的所有节点，写操作成本高，另外，多个写操作之间是互斥的，同一时刻只允许一个写操作。

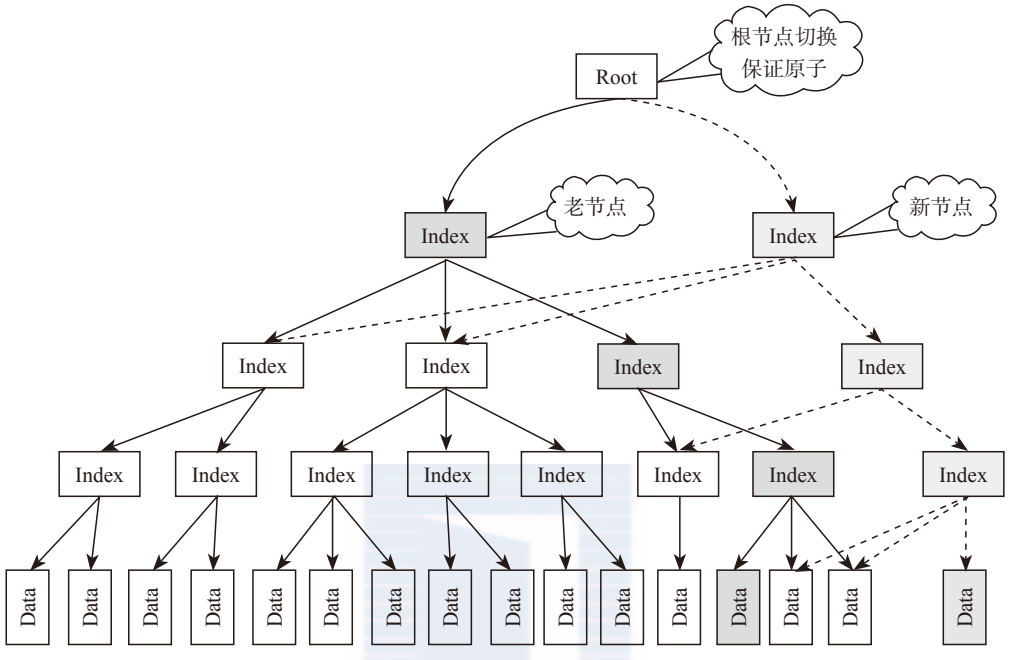

图 2-10　写时复制的 B+ 树

3. 多版本并发控制

除了写时复制技术，多版本并发控制，即 MVCC（Multi-Version Concurrency Control），也能够实现读事务不加锁。MVCC 对每行数据维护多个版本，无论事务的执行时间有多长，MVCC 总是能够提供与事务开始时刻相一致的数据。

以 MySQL InnoDB 存储引擎为例，InnoDB 对每一行维护了两个隐含的列，其中一列存储行被修改的“时间”，另外一列存储行被删除的“时间”，注意，InnoDB 存储的并不是绝对时间，而是与时间对应的数据库系统的版本号，每当一个事务开始时，InnoDB 都会给这个事务分配一个递增的版本号，所以版本号也可以被认为是事务号。对于每一行查询语句，InnoDB 都会把这个查询语句的版本号同这个查询语句遇到的行的版本号进行对比，然后结合不同的事务隔离级别，来决定是否返回改行。

下面分别以 SELECT、DELETE、INSERT、UPDATE 语句来说明。

（1）SELECT

对于 SELECT 语句，只有同时满足了下面两个条件的行，才能被返回：

a）行的修改版本号小于等于该事务号。

b）行的删除版本号要么没有被定义，要么大于事务的版本号。

如果行的修改或者删除版本号大于事务号，说明行是被该事务后面启动的事务修改

或者删除的。在可重复读取隔离级别下，后开始的事务对数据的影响不应该被先开始的事务看见，所以应该忽略后开始的事务的更新或者删除操作。

（2）INSERT

对新插入的行，行的修改版本号更新为该事务的事务号。

（3）DELETE

对于删除，InnoDB 直接把该行的删除版本号设置为当前的事务号，相当于标记为删除，而不是物理删除。

（4）UPDATE

在更新行的时候，InnoDB 会把原来的行复制一份，并把当前的事务号作为该行的修改版本号。

MVCC 读取数据的时候不用加锁，每个查询都通过版本检查，只获得自己需要的数据版本，从而大大提高了系统的并发度。当然，为了实现多版本，必须对每行存储额外的多个版本的数据。另外，MVCC 存储引擎还必须定期删除不再需要的版本，及时回收空间。

2.5 故障恢复

数据库运行过程中可能会发生故障，这个时候某些事务可能执行到一半但没有提交，当系统重启时，需要能够恢复到一致的状态，即要么提交整个事务，要么回滚。数据库系统以及其他的分布式存储系统一般采用操作日志（有时也称为提交日志，即 Commit Log）技术来实现故障恢复。操作日志分为回滚日志（UNDO Log）、重做日志（REDO Log）以及 UNDO/REDO 日志。如果记录事务修改前的状态，则为回滚日志；相应地，如果记录事务修改后的状态，则为重做日志。本节介绍操作日志及故障恢复基础知识。

2.5.1 操作日志

为了保证数据库的一致性，数据库操作需要持久化到磁盘，如果每次操作都随机更新磁盘的某个数据块，系统性能将会很差。因此，通过操作日志顺序记录每个数据库操作并在内存中执行这些操作，内存中的数据定期刷新到磁盘，实现将随机写请求转化为顺序写请求。

操作日志记录了事务的操作。例如，事务 T 对表格中的 X 执行加 10 操作，初始时 X = 5，更新后 X = 15，那么，UNDO 日志记为 <T，X，5>，REDO 日志记为 <T，X，15>，UNDO/REDO 日志记为 <T，X，5，15>。

关系数据库系统一般采用 UNDO/REDO 日志，相关技术可以参考数据库系统实现方面的资料。可以将关系数据库存储模型做一定程度的简化：

1）假设内存足够大，每次事务的修改操作都可以缓存在内存中。

2）数据库的每个事务只包含一个操作，即每个事务都必须立即提交（Auto Commit）。

REDO 日志要求我们将所有未提交事务修改的数据块保留在内存中。简化后的存储模型可以采用单一的 REDO 日志，大大简化了存储系统故障恢复。

2.5.2　重做日志

存储系统如果采用 REDO 日志，其写操作流程如下：

1）将 REDO 日志以追加写的方式写入磁盘的日志文件。

2）将 REDO 日志的修改操作应用到内存中。

3）返回操作成功或者失败。

REDO 日志的约束规则为：在修改内存中的元素 X 之前，要确保与这一修改相关的操作日志必须先刷入到磁盘中。顾名思义，用 REDO 日志进行故障恢复，只需要从头到尾读取日志文件中的修改操作，并将它们逐个应用到内存中，即重做一遍。

为什么需要先写操作日志再修改内存中的数据呢？假如先修改内存中的数据，那么用户就能立刻读到修改后的结果，一旦在完成内存修改与写入日志之间发生故障，那么最近的修改操作无法恢复。然而，之前的用户可能已经读取了修改后的结果，这就会产生不一致的情况。

2.5.3　优化手段

1. 成组提交

存储系统要求先将 REDO 日志刷入磁盘才可以更新内存中的数据，如果每个事务都要求将日志立即刷入磁盘，系统的吞吐量将会很差。因此，存储系统往往有一个是否立即刷入磁盘的选项，对于一致性要求很高的应用，可以设置为立即刷入；相应地，对于一致性要求不太高的应用，可以设置为不要求立即刷入，首先将 REDO 日志缓存到操作系统或者存储系统的内存缓冲区中，定期刷入磁盘。这种做法有一个问题，如果存储系统意外故障，可能丢失最后一部分更新操作。

成组提交（Group Commit）技术是一种有效的优化手段。REDO 日志首先写入到存储系统的日志缓冲区中：

a）日志缓冲区中的数据量超过一定大小，比如 512KB；

b）距离上次刷入磁盘超过一定时间，比如 10ms。

当满足以上两个条件中的某一个时，将日志缓冲区中的多个事务操作一次性刷入磁盘，接着一次性将多个事务的修改操作应用到内存中并逐个返回客户端操作结果。与定期刷入磁盘不同的是，成组提交技术保证 REDO 日志成功刷入磁盘后才返回写操作成功。这种做法可能会牺牲写事务的延时，但大大提高了系统的吞吐量。

2. 检查点

如果所有的数据都保存在内存中，那么可能出现两个问题：

- ❑ 故障恢复时需要回放所有的 REDO 日志，效率较低。如果 REDO 日志较多，比如超过 100GB，那么，故障恢复时间是无法接受的。
- ❑ 内存不足。即使内存足够大，存储系统往往也只能够缓存最近较长一段时间的更新操作，很难缓存所有的数据。

因此，需要将内存中的数据定期转储（Dump）到磁盘，这种技术称为 checkpoint（检查点）技术。系统定期将内存中的操作以某种易于加载的形式（checkpoint 文件）转储到磁盘中，并记录 checkpoint 时刻的日志回放点，以后故障恢复只需要回放 checkpoint 时刻的日志回放点之后的 REDO 日志。

由于将内存数据转储到磁盘需要很长的时间，而这段时间还可能有新的更新操作，checkpoint 必须找到一个一致的状态。checkpoint 流程如下：

1）日志文件中记录“START CKPT”。

2）将内存中的数据以某种易于加载的组织方式转储到磁盘中，形成 checkpoint 文件。checkpoint 文件中往往记录“START CKPT”的日志回放点，用于故障恢复。

3）日志文件中记录“END CKPT”。

故障恢复流程如下：

1）将 checkpoint 文件加载到内存中，这一步操作往往只需要加载索引数据，加载效率很高。

2）读取 checkpoint 文件中记录的“START CKPT”日志回放点，回放之后的 REDO 日志。

上述 checkpoint 故障恢复方式依赖 REDO 日志中记录的都是修改后的结果这一特性，也就是说，即使 checkpoint 文件中已经包含了某些操作的结果，重新回放一次或者多次这些操作的 REDO 日志也不会造成数据错误。如果同一个操作执行一次与重复执行多次的效果相同，这种操作具有“幂等性”。有些操作不具备这种特性，例如，加法操作、追加操作。如果 REDO 日志记录的是这种操作，那么 checkpoint 文件中的数据

一定不能包含“START CKPT”与“END CKPT”之间的操作。为此，主要有两种处理方法：

- checkpoint 过程中停止写服务，所有的修改操作直接失败。这种方法实现简单，但不适合在线业务。
- 内存数据结构支持快照。执行 checkpoint 操作时首先对内存数据结构做一次快照，接着将快照中的数据转储到磁盘生成 checkpoint 文件，并记录此时对应的 REDO 日志回放点。生成 checkpoint 文件的过程中允许写操作，但 checkpoint 文件中的快照数据不会包含这些操作的结果。

2.6　数据压缩

数据压缩分为有损压缩与无损压缩两种，有损压缩算法压缩比率高，但数据可能失真，一般用于压缩图片、音频、视频；而无损压缩算法能够完全还原原始数据，本文只讨论无损压缩算法。早期的数据压缩技术就是基于编码上的优化技术，其中以 Huffman 编码最为知名，它通过统计字符出现的频率计算最优前缀编码。1977 年，以色列人 Jacob Ziv 和 Abraham Lempel 发表论文《顺序数据压缩的一个通用算法》，从此，LZ 系列压缩算法几乎垄断了通用无损压缩领域，常用的 Gzip 算法中使用的 LZ77，GIF 图片格式中使用的 LZW，以及 LZO 等压缩算法都属于这个系列。设计压缩算法时不仅要考虑压缩比，还要考虑压缩算法的执行效率。Google Bigtable 系统中采用 BMDiff 和 Zippy 压缩算法，这两个算法也是 LZ 算法的变种，它们通过牺牲一定的压缩比，换来执行效率的大幅提升。

压缩算法的核心是找重复数据，列式存储技术通过把相同列的数据组织在一起，不仅减少了大数据分析需要查询的数据量，还大大地提高了数据的压缩比。传统的 OLAP（Online Analytical Processing）数据库，如 Sybase IQ、Teradata，以及 Bigtable、HBase 等分布式表格系统都实现了列式存储。本节介绍数据压缩以及列式存储相关的基础知识。

2.6.1　压缩算法

压缩是一个专门的研究课题，没有通用的做法，需要根据数据的特点选择或者自己开发合适的算法。压缩的本质就是找数据的重复或者规律，用尽量少的字节表示。

Huffman 编码是一种基于编码的优化技术，通过统计字符出现的频率来计算最优前缀编码。LZ 系列算法一般有一个窗口的概念，在窗口内部找重复并维护数据字典。常用的压缩算法包括 Gzip、LZW、LZO，这些算法都借鉴或改进了原始的 LZ77 算法，如

Gzip 压缩混合使用了 LZ77 以及 Huffman 编码，LZW 以及 LZO 算法是 LZ77 思想在实现手段的进一步优化。存储系统在选择压缩算法时需要考虑压缩比和效率。读操作需要先读取磁盘中的内容再解压缩，写操作需要先压缩再将压缩结果写入到磁盘，整个操作的延时包括压缩 / 解压缩和磁盘读写的延迟，压缩比越大，磁盘读写的数据量越小，而压缩 / 解压缩的时间也会越长，所以这里需要一个很好的权衡点。Google Bigtable 系统中使用了 BMDiff 以及 Zippy 两种压缩算法，它们通过牺牲一定的压缩比换取算法执行速度的大幅提升，从而获得更好的折衷。

1. Huffman 编码

前缀编码要求一个字符的编码不能是另一个字符的前缀。假设有三个字符 A、B、C，它们的二进制编码分别是 0、1、01，如果我们收到一段信息是 01010，解码时我们如何区分是 CCA 还是 ABABA，或者 ABCA 呢？一种解决方案就是前缀编码，要求一个字符编码不能是另外一个字符编码的前缀。如果使用前缀编码将 A、B、C 编码为：

A：0　B：10　C：110

这样，01010 就只能被翻译成 ABB。Huffman 编码需要解决的问题是，如何找出一种前缀编码方式，使得编码的长度最短。

假设有一个字符串 333444455555666666777777，它是由 3 个 3，4 个 4，5 个 5，6 个 6，7 个 7 组成的。那么，对应的前缀编码可能是：

1）3：000　4：001　5：010　6：011　7：1

2）3：000　4：001　7：01　5：10　6：11

第 1 种编码方式的权值为（3 + 4 + 5 + 6）* 3 + 7 * 1 = 61，而第 2 种编码方式的权值为（3 + 4）* 3 +（5 + 6 + 7）* 2 = 57。可以看出，第 2 种编码方式的长度更短，而且我们还可以知道，第 2 种编码方式是最优的 Huffman 编码。Huffman 编码的构造过程不在本书讨论范围之内，感兴趣的读者可以参考数据结构的相关图书。

2. LZ 系列压缩算法

LZ 系列压缩算法是基于字典的压缩算法。假设需要压缩一篇英文文章，最容易想到的压缩算法是构造一本英文字典，这样，我们只需要保存每个单词在字典中出现的页码和位置就可以了。页码用两个字节，位置用一个字节，那么一个单词需要使用三个字节表示，而我们知道一般的英语单词长度都在三个字节以上。因此，我们实现了对这篇英文文章的压缩。当然，实际的通用压缩算法不能这么做，因为我们在解压时需要一本英文字典，而这部分信息是压缩程序不可预知的，同时也不能保存在压缩信息里面。LZ 系列的算法是一种动态创建字典的方法，压缩过程中动态创建字典并保存在压缩信

息里面。

LZ77 是第一个 LZ 系列的算法，比如字符串 ABCABCDABC 中 ABC 重复出现了三次，压缩信息中只需要保存第一个 ABC，后面两个 ABC 只需要把第一个出现 ABC 的位置和长度存储下来就可以了。这样，保存后面两个 ABC 就只需要一个二元数组 < 匹配串的相对位置，匹配长度 >。解压的时候，根据匹配串的相对位置，向前找到第一个 ABC 的位置，然后根据匹配的长度，直接把第一个 ABC 复制到当前解压缓冲区里面就可以了。

如表 2-7 所示，{S}* 表示字符串 S 的所有子串构成的集合，例如，{ABC}* 是字符串 A、B、C、AB、BC、ABC 构成的集合。每一步执行时如果能够在压缩字典中找到匹配串，则输出匹配信息；否则，输出源信息。执行第 1 步时，压缩字典为空，输出字符‘A’，并将‘A’加入到压缩字典；执行第 2 步时，压缩字典为 {A}*，输出字符‘B’，并将‘B’加入到压缩字典；依次类推。执行到第 4 步和第 6 步时发现字符 ABC 之前已经出现过，输出匹配的位置和长度。

表 2-7　字符串 ABCABCDABC 的 LZ 压缩过程

步　骤	当前输入缓冲	压 缩 字 典	输 出 信 息
1	ABCABCDABC	空	A
2	BCABCDABC	{A}*	B
3	CABCDABC	{AB}*	C
4	ABCDABC	{ABC}*	<0, 3>
5	DABC	{ABCABC}*	D
6	ABC	{ABCABCD}*	<0, 3>
7	空	{ABCABCDABC}*	空

LZ 系列压缩算法有如下几个问题：

1）如何区分匹配信息和源信息？通用的解决方法是额外使用一个位（bit）来区分压缩信息里面的源信息和匹配信息。

2）需要使用多少个字节表示匹配信息？记录重复信息的匹配信息包含两项，一个是匹配串的相对位置，另一个是匹配的长度。例如，可以采用固定的两个字节来表示匹配信息，其中，1 位用来区分源信息和匹配信息，11 位表示匹配位置，4 位表示匹配长度。这样，压缩算法支持的最大数据窗口为 $2^{11} = 2048$ 字节，支持重复串的最大长度为 $2^4 = 16$ 字节。当然，也可以采用变长的方式表示匹配信息。

3）如何快速查找最长匹配串？最容易想到的做法是把字符串的所有子串都存放到一张哈希表中，表 2-7 中第 4 步执行前哈希表中包含 ABC 的所有子串，即 A、AB、

BC、ABC。这种做法的运行效率很低，实际的做法往往会做一些改进。例如，哈希表中只保存所有长度为 3 的子串，如果在数据字典中找到匹配串，即前 3 个字节相同，接着再往后顺序遍历找出最长匹配。

3. BMDiff 与 Zippy

在 Google 的 Bigtable 系统中，设计了 BMDiff 和 Zippy 两种压缩算法。BMDiff 和 Zippy（也称为 Snappy）也属于 LZ 系列，相比传统的 LZW 或者 Gzip，这两种算法的压缩比不算高，但是处理速度非常快。如表 2-8 所示，Zippy 和 BMDiff 的压缩 / 解压缩速度是 Gzip 算法的 5 ～ 10 倍。

表 2-8 各种压缩算法对比

算 法	压 缩 比	压 缩 速 度	解压缩速度
Gzip	13.4%	21MB/s	118MB/s
LZO	20.5%	135MB/s	410MB/s
Zippy	22.2%	172MB/s	409MB/s
BMDiff	数据相关	100MB/s	1000MB/s

相比原始的 LZ77，Zippy 实现时主要做了如下改进：

1）压缩字典中只保存所有长度为 4 的子串，只有重复匹配的长度大于等于 4，才输出匹配信息；否则，输出源信息。另外，Zippy 算法中的压缩字典只保存最后一个长度等于 4 的子串的位置，以 ABCDEABCDABCDE 为例，Zippy 算法的过程参见表 2-9。

表 2-9 Google Zippy 压缩算法

步 骤	当前输入缓冲	Hash[“ABCD”]	输 出
1	ABCDEABCDABCDE	空	A
2	BCDEABCDABCDE	空	B
3	CDEABCDABCDE	空	C
4	DEABCDABCDE	空	D
5	EABCDABCDE	0	E
6	ABCDABCDE	0	<0, 4>
7	ABCDE	5	<5, 4>
8	E	9	E
9	空	9	空

Zippy 算法执行完第 4 步后，发现“ABCD”出现过，于是在压缩字典中记录“ABCD”第一次出现的位置，即位置 0。执行到第 6 步时发现 ABCD 之前出现过，输出匹配信息，同时将数据字典中记录的 ABCD 的位置更新为第二个 ABCD 的位置，即位置 5；执行到第 7 步时，虽然 ABCDE 之前都出现过，但由于数据字典中记录的是第

二个 ABCD 的位置，因此，重复串为 ABCD，而不是理想的 ABCDE。Zippy 的这种实现方式牺牲了压缩比，但是提升了性能。

2）Zippy 内部将数据划分为一个一个长度为 32KB 的数据块，每个数据块分别压缩，多个数据块之间没有联系，因此，只需要两个字节（确切地说，15 个位）就可以表示匹配串的相对位置。另外，Zippy 内部还对匹配信息的表示进行了精心的设计，采用变长的表示方法。如果匹配长度小于 12 个字节（由于前面 4 个字节总是相同，所以 4 <= 匹配长度 < 12，可以通过 3 个位来表示）且匹配位置小于 2048，则使用两个字节表示；否则，使用更多的字节表示。总而言之，Zippy 对匹配信息的编码和实现都非常精妙，感兴趣的读者可以阅读开源的 Snappy 项目的源代码。

相比 Zippy，BMDiff 算法实现显得更为激进。BMDiff 算法将待压缩数据拆分为长度为 b（默认情况下 b = 32）的小段 0 … b−1，b … 2b−1, 2b … 3b−1，以此类推。BMDiff 的字典中保存了每个小段的哈希值，因此，长度为 N 的字符串需要的哈希表大小为 N / b。与 Zippy 算法不同的是，BMDiff 算法并没有保存每个长度为 b 的子串的哈希值，这种方式带来的问题是，某些重复长度超过 b 的子串可能无法被压缩。例如，待压缩字符串为 EABCDABCD，b = 4，字典中保存了 EABC 和 DABC 两个子串，虽然 ABCD 重复出现了两次，但无法被压缩。然而，可以证明，只要重复长度超过 2b – 1，那么一定能够在字典中找到。假如待压缩字符串还是 EABCDABCD，b = 2，那么字典中保存了 EA、BC、DA、BC，压缩程序处理第二个 BC 的时候，发现之前 BC 已经出现过，因此，往前往后找到最长的匹配串，即 ABCD，并输出相应的匹配信息。BMDiff 适合压缩重复度很高的速度，例如网页，Google 的 Bigtable 系统中实现了列存储，相同列的数据存放到一起，重复度很高。

2.6.2　列式存储

传统的行式数据库将一个个完整的数据行存储在数据页中。如果处理查询时需要用到大部分的数据列，这种方式在磁盘 IO 上是比较高效的。一般来说，OLTP（Online Transaction Processing，联机事务处理）应用适合采用这种方式。

一个 OLAP 类型的查询可能需要访问几百万甚至几十亿个数据行，且该查询往往只关心少数几个数据列。例如，查询今年销量最高的前 20 个商品，这个查询只关心三个数据列：时间（date）、商品（item）以及销售量（sales amount）。商品的其他数据列，例如商品 URL、商品描述、商品所属店铺，等等，对这个查询都是没有意义的。

如图 2-11 所示，列式数据库是将同一个数据列的各个值存放在一起。插入某个数据行时，该行的各个数据列的值也会存放到不同的地方。上例中列式数据库只需要读取存储着“时间、商品、销量”的数据列，而行式数据库需要读取所有的数据列。因此，

列式数据库大大地提高了 OLAP 大数据量查询的效率。当然，列式数据库不是万能的，每次读取某个数据行时，需要分别从不同的地方读取各个数据列的值，然后合并在一起形成数据行。因此，如果每次查询涉及的数据量较小或者大部分查询都需要整行的数据，列式数据库并不适用。

很多列式数据库还支持列组（column group，Bigtable 系统中称为 locality group），即将多个经常一起访问的数据列的各个值存放在一起。如果读取的数据列属于相同的列组，列式数据库可以从相同的地方一次性读取多个数据列的值，避免了多个数据列的合并。列组是一种行列混合存储模式，这种模式能够同时满足 OLTP 和 OLAP 的查询需求。

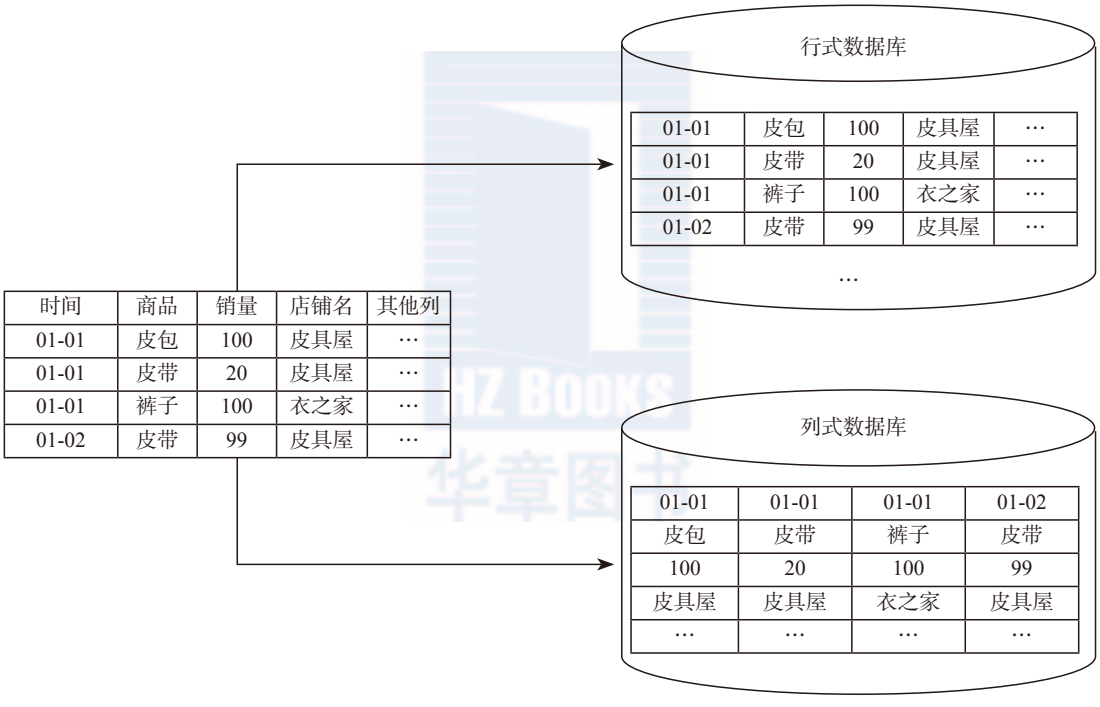

图 2-11　列式数据库示意图

由于同一个数据列的数据重复度很高，因此，列式数据库压缩时有很大的优势。例如，Google Bigtable 列式数据库对网页库压缩可以达到 15 倍以上的压缩率。另外，可以针对列式存储做专门的索引优化。比如，性别列只有两个值，“男”和“女”，可以对这一列建立位图索引：

如图 2-12 所示，“男”对应的位图为 100101，表示第 1、4、6 行值为“男”；“女”对应的位图为 011010，表示第 2、3、5 行值为“女”。如果需要查找男性或者女性的个

数，只需要统计相应的位图中 1 出现的次数即可。另外，建立位图索引后 0 和 1 的重复度高，可以采用专门的编码方式对其进行压缩。

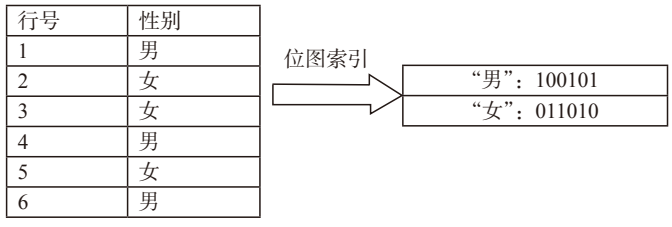

图 2-12　位图索引示意图

第 3 章　分布式系统

水桶无论有多高，其盛水的高度取决于其中最短的那块木板，这就是著名的“木桶效应”。架构设计之初要求我们能够估算系统的性能从而权衡不同的设计方法。本章首先介绍分布式系统相关的基础概念和性能估算方法。接着，介绍分布式系统的基础理论知识，包括数据分布、复制、一致性、容错等。最后，介绍常见的分布式协议。

分布式系统面临的第一个问题就是数据分布，即将数据均匀地分布到多个存储节点。另外，为了保证可靠性和可用性，需要将数据复制多个副本，这就带来了多个副本之间的数据一致性问题。大规模分布式存储系统的重要目标就是节省成本，因而只能采用性价比较高的 PC 服务器。这些服务器性能很好，但是故障率很高，要求系统能够在软件层面实现自动容错。当存储节点出现故障时，系统能够自动检测出来，并将原有的数据和服务迁移到集群中其他正常工作的节点。

分布式系统中有两个重要的协议，包括 Paxos 选举协议以及两阶段提交协议。Paxos 协议用于多个节点之间达成一致，往往用于实现总控节点选举。两阶段提交协议用于保证跨多个节点操作的原子性，这些操作要么全部成功，要么全部失败。理解了这两个分布式协议之后，学习其他分布式协议会变得相当容易。

3.1　基本概念

3.1.1　异常

在分布式存储系统中，往往将一台服务器或者服务器上运行的一个进程称为一个节点，节点与节点之间通过网络互联。大规模分布式存储系统的一个核心问题在于自动容错。然而，服务器节点是不可靠的，网络也是不可靠的，本节介绍系统运行过程中可能会遇到的各种异常。

1. 异常类型

（1）服务器宕机

引发服务器宕机的原因可能是内存错误、服务器停电等。服务器宕机可能随时发生，当发生宕机时，节点无法正常工作，称为“不可用”（unavailable）。服务器重启后，节点将失去所有的内存信息。因此，设计存储系统时需要考虑如何通过读取持久化

介质（如机械硬盘，固态硬盘）中的数据来恢复内存信息，从而恢复到宕机前的某个一致的状态。进程运行过程中也可能随时因为 core dump 等原因退出，和服务器宕机一样，进程重启后也需要恢复内存信息。

（2）网络异常

引发网络异常的原因可能是消息丢失、消息乱序（如采用 UDP 方式通信）或者网络包数据错误。有一种特殊的网络异常称为“网络分区”，即集群的所有节点被划分为多个区域，每个区域内部可以正常通信，但是区域之间无法通信。例如，某分布式系统部署在两个数据中心，由于网络调整，导致数据中心之间无法通信，但是，数据中心内部可以正常通信。

设计容错系统的一个基本原则是：网络永远是不可靠的，任何一个消息只有收到对方的回复后才可以认为发送成功，系统设计时总是假设网络将会出现异常并采取相应的处理措施。

（3）磁盘故障

磁盘故障是一种发生概率很高的异常。磁盘故障分为两种情况：磁盘损坏和磁盘数据错误。磁盘损坏时，将会丢失存储在上面的数据，因而，分布式存储系统需要考虑将数据存储到多台服务器，即使其中一台服务器磁盘出现故障，也能从其他服务器上恢复数据。对于磁盘数据错误，往往可以采用校验和（checksum）机制来解决，这样的机制既可以在操作系统层面实现，又可以在上层的分布式存储系统层面实现。

2. “超时”

由于网络异常的存在，分布式存储系统中请求结果存在“三态”的概念。在单机系统中，只要服务器没有发生异常，每个函数的执行结果是确定的，要么成功，要么失败。然而，在分布式系统中，如果某个节点向另外一个节点发起 RPC（Remote Procedure Call）调用，这个 RPC 执行的结果有三种状态：“成功”、“失败”、“超时”（未知状态），也称为分布式存储系统的三态。

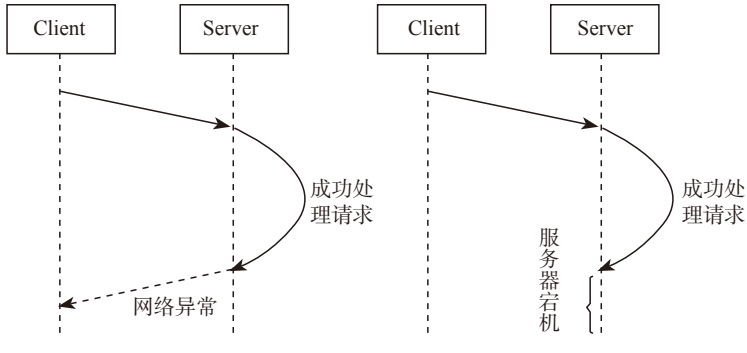

图 3-1　RPC 执行成功但超时

图 3-1 给出了 RPC 执行成功但超时的例子。服务器（Server）收到并成功处理完成客户端（Client）的请求，但是由于网络异常或者服务器宕机，客户端没有收到服务器端的回复。此时，RPC 的执行结果为超时，客户端不能简单地认为服务器端处理失败。

一个更加通俗的例子是 2.4.1 节介绍的 ATM 取款。ATM 取款时 ATM 机有时会提示："无法打印凭条，是否继续取款？"。这是因为 ATM 机需要和银行服务器端通信，二者之间的网络可能出现故障，此时 ATM 机发往银行服务器端的 RPC 请求如果发生超时，ATM 机无法确定 RPC 请求成功还是失败。正常情况下，ATM 机会打印凭条，用于后续与银行服务器端对账。如果无法打印凭条，存在资金安全风险，因此，ATM 机有一个提示。

当出现超时状态时，只能通过不断读取之前操作的状态来验证 RPC 操作是否成功。当然，设计分布式存储系统时可以将操作设计为"幂等"的，也就是说，操作执行一次与执行多次的结果相同，例如，覆盖写就是一种常见的幂等操作。如果采用这种设计，当出现失败和超时时，都可以采用相同的处理方式，即一直重试直到成功。

3.1.2 一致性

由于异常的存在，分布式存储系统设计时往往会将数据冗余存储多份，每一份称为一个副本（replica/copy）。这样，当某一个节点出现故障时，可以从其他副本上读到数据。可以这么认为，副本是分布式存储系统容错技术的唯一手段。由于多个副本的存在，如何保证副本之间的一致性是整个分布式系统的理论核心。

可以从两个角度理解一致性：第一个角度是用户，或者说是客户端，即客户端读写操作是否符合某种特性；第二个角度是存储系统，即存储系统的多个副本之间是否一致，更新的顺序是否相同，等等。

首先定义如下场景，这个场景包含三个组成部分：

- 存储系统：存储系统可以理解为一个黑盒子，它为我们提供了可用性和持久性的保证。
- 客户端 A：客户端 A 主要实现从存储系统 write 和 read 操作。
- 客户端 B 和客户端 C：客户端 B 和 C 是独立于 A，并且 B 和 C 也相互独立的，它们同时也实现对存储系统的 write 和 read 操作。

从客户端的角度来看，一致性包含如下三种情况：

- 强一致性：假如 A 先写入了一个值到存储系统，存储系统保证后续 A，B，C 的读取操作都将返回最新值。当然，如果写入操作"超时"，那么成功或者失败都是可能的，客户端 A 不应该做任何假设。
- 弱一致性：假如 A 先写入了一个值到存储系统，存储系统不能保证后续 A，B，

C 的读取操作是否能够读取到最新值。

- 最终一致性：最终一致性是弱一致性的一种特例。假如 A 首先写入一个值到存储系统，存储系统保证如果后续没有写操作更新同样的值，A，B，C 的读取操作“最终”都会读取到 A 写入的最新值。“最终”一致性有一个“不一致窗口”的概念，它特指从 A 写入值，到后续 A，B，C 读取到最新值的这段时间。“不一致性窗口”的大小依赖于以下的几个因素：交互延迟，系统的负载，以及复制协议要求同步的副本数。

最终一致性描述比较粗略，其他常见的变体如下：

- 读写（Read-your-writes）一致性：如果客户端 A 写入了最新的值，那么 A 的后续操作都会读取到最新值。但是其他用户（比如 B 或者 C）可能要过一会才能看到。
- 会话（Session）一致性：要求客户端和存储系统交互的整个会话期间保证读写一致性。如果原有会话因为某种原因失效而创建了新的会话，原有会话和新会话之间的操作不保证读写一致性。
- 单调读（Monotonic read）一致性：如果客户端 A 已经读取了对象的某个值，那么后续操作将不会读取到更早的值。
- 单调写（Monotonic write）一致性：客户端 A 的写操作按顺序完成，这就意味着，对于同一个客户端的操作，存储系统的多个副本需要按照与客户端相同的顺序完成。

从存储系统的角度看，一致性主要包含如下几个方面：

- 副本一致性：存储系统的多个副本之间的数据是否一致，不一致的时间窗口等；
- 更新顺序一致性：存储系统的多个副本之间是否按照相同的顺序执行更新操作。

一般来说，存储系统可以支持强一致性，也可以为了性能考虑只支持最终一致性。从客户端的角度看，一般要求存储系统能够支持读写一致性，会话一致性，单调读，单调写等特性，否则，使用比较麻烦，适用的场景也比较有限。

3.1.3　衡量指标

评价分布式存储系统有一些常用的指标，下面分别介绍。

（1）性能

常见的性能指标有：系统的吞吐能力以及系统的响应时间。其中，系统的吞吐能力指系统在某一段时间可以处理的请求总数，通常用每秒处理的读操作数（QPS，Query Per Second）或者写操作数（TPS，Transaction Per Second）来衡量；系统的响应延迟，指从某个请求发出到接收到返回结果消耗的时间，通常用平均延时或者 99.9% 以上请求

的最大延时来衡量。这两个指标往往是矛盾的，追求高吞吐的系统，往往很难做到低延迟；追求低延迟的系统，吞吐量也会受到限制。因此，设计系统时需要权衡这两个指标。

（2）可用性

系统的可用性（availability）是指系统在面对各种异常时可以提供正常服务的能力。系统的可用性可以用系统停服务的时间与正常服务的时间的比例来衡量，例如某系统的可用性为 4 个 9（99.99%），相当于系统一年停服务的时间不能超过 365 × 24 × 60 / 10000 = 52.56 分钟。系统可用性往往体现了系统的整体代码质量以及容错能力。

（3）一致性

3.1.2 节说明了系统的一致性。一般来说，越是强的一致性模型，用户使用起来越简单。笔者认为，如果系统部署在同一个数据中心，只要系统设计合理，在保证强一致性的前提下，不会对性能和可用性造成太大的影响。后文中笔者在 Alibaba 参与开发的 OceanBase 系统以及 Google 的分布式存储系统都倾向强一致性。

（4）可扩展性

系统的可扩展性（scalability）指分布式存储系统通过扩展集群服务器规模来提高系统存储容量、计算量和性能的能力。随着业务的发展，对底层存储系统的性能需求不断增加，比较好的方式就是通过自动增加服务器提高系统的能力。理想的分布式存储系统实现了“线性可扩展”，也就是说，随着集群规模的增加，系统的整体性能与服务器数量呈线性关系。

3.2 性能分析

给定一个问题，往往会有多种设计方案，而方案评估的一个重要指标就是性能，如何在系统设计之初估算存储系统的性能是存储工程师的必备技能。性能分析用来判断设计方案是否存在瓶颈点，权衡多种设计方案，另外，性能分析也可作为后续性能优化的依据。性能分析与性能优化是相对的，系统设计之初通过性能分析确定设计目标，防止出现重大的设计失误，等到系统试运行后，需要通过性能优化方法找出系统中的瓶颈点并逐步消除，使得系统达到设计之初确定的设计目标。

性能分析的结果是不精确的，然而，至少可以保证，估算的结果与实际值不会相差一个数量级。设计之初首先分析整体架构，接着重点分析可能成为瓶颈的单机模块。系统中的资源（CPU、内存、磁盘、网络）是有限的，性能分析就是需要找出可能出现的资源瓶颈。本节通过几个实例说明性能分析方法。

1. 生成一张有 30 张缩略图（假设图片原始大小为 256KB）的页面需要多少时间？

❑ 方案 1：顺序操作，每次先从磁盘中读取图片，再执行生成缩略图操作，执行时

间为：30×10ms（磁盘随机读取时间）+ 30×256K / 30MB/s（假设缩略图生成速度为 30MB/s）= 560ms

- 方案 2：并行操作，一次性发送 30 个请求，每个请求读取一张图片并生成缩略图，执行时间为：10ms + 256K / 300MB/s = 18ms

当然，系统实际运行的时候可能有缓存以及其他因素的干扰，这些因素在性能估算阶段可以先不考虑，简单地将估算结果乘以一个系数即为实际值。

2. 1GB 的 4 字节整数，执行一次快速排序需要多少时间？

Google 的 Jeff Dean 提出了一种排序性能分析方法：排序时间 = 比较时间（分支预测错误）+ 内存访问时间。快速排序过程中会发生大量的分支预测错误，所以比较次数为 $2^{28} \times \log(2^{28}) \approx 2^{33}$，其中，约 1/2 的比较会发生分支预测错误，所以比较时间为 $1/2 \times 2^{33} \times 5\text{ns} = 21\text{s}$，另外，快速排序每次分割操作都需要扫描一遍内存，假设内存顺序访问性能为 4GB/s，所以内存访问时间为 28×1GB / 4GB = 7s。因此，单线程排序 1GB 4 字节整数总时间约为 28s。

3. Bigtable 系统性能分析

Bigtable 是 Google 的分布式表格系统，它的优势是可扩展性好，可随时增加或者减少集群中的服务器，但支持的功能有限，支持的操作主要包括：

- 单行操作：基于主键的随机读取，插入，更新，删除（CRUD）操作；
- 多行扫描：扫描一段主键范围内的数据。Bigtable 中每行包括多个列，每一行的某一列对应一个数据单元，每个数据单元包括多个版本，可以按照列名或者版本对扫描结果进行过滤。

假设某类 Bigtable 系统的总体设计中给出的性能指标为：

- 系统配置：同一个机架下 40 台服务器（8 核，24GB 内存，10 路 15000 转 SATA 硬盘）；
- 表格：每行数据 1KB，64KB 一个数据块，不压缩。

a）随机读取（缓存不命中）：1KB/item×300item/s=300KB/s

Bigtable 系统中每次随机读取需要首先从 GFS 中读取一个 64KB 的数据块，经过 CPU 处理后返回用户一行数据（大小为 1KB）。因此，性能受限于 GFS 中 ChunkServer（GFS 系统中的工作节点）的磁盘 IOPS 以及 Bigtable Tablet Server（Bigtable 系统中的工作节点）的网络带宽。先看底层的 GFS，每台机器拥有 10 块 SATA 盘，每块 SATA 盘的 IOPS 约为 100，因此，每台机器的 IOPS 理论值约为 1000，考虑到负载均衡等因素，将随机读取的 QPS 设计目标定为 300，保留一定的余量。另外，每台机器每秒从 GFS 中读取的数据量为 300×64KB = 19.2MB，由于所有的服务器分布在同一个机架下，网络不会成为瓶颈。

b）随机读取（内存表）：1KB/item×20000items/s=20MB/s

Bigtable 中支持内存表，内存表的数据全部加载到内存中，读取时不需要读取底层的 GFS。随机读取内存表的性能受限于 CPU 以及网络，内存型服务的 QPS 一般在 10W，由于网络发送小数据有较多 overhead 且 Bigtable 内存操作有较多的 CPU 开销，保守估计每个节点的 QPS 为 20000，客户端和 Tablet Server 之间的网络流量为 20MB/s。

c）随机写 / 顺序写：1KB/item×8000item/s=8MB/s

Bigtable 中随机写和顺序写的性能是差不多的，写入操作需要首先将操作日志写入到 GFS，接着修改本地内存。为了提高性能，Bigtable 实现了成组提示技术，即将很多写操作凑成一批（比如 512KB ~ 2MB）一次性提交到 GFS 中。Bigtable 每次写一份数据需要在 GFS 系统中重复写入 3 份到 10 份，当写入速度达到 8000 QPS，即 8MB/s 后 Tablet Server 的网络将成为瓶颈。

d）扫描：1KB/item×30000item/s=30MB/s

Bigtable 扫描操作一次性从 GFS 中读取大量的数据（比如 512KB ~ 2MB），GFS 的磁盘 IO 不会成为瓶颈。另外，批量操作减少了 CPU 以及网络收发包的开销，扫描操作的瓶颈在于 Tablet Server 读取底层 GFS 的带宽，估计为 30MB/s，对应 30000 QPS。

如果集群规模超过 40 台，不能保证所有的服务器在同一个机架下，系统设计以及性能分析都会有所不同。性能分析可能会很复杂，因为不同情况下系统的瓶颈点不同，有的时候是网络，有的时候是磁盘，有的时候甚至是机房的交换机或者 CPU，另外，负载均衡以及其他因素的干扰也会使得性能更加难以量化。只有理解存储系统的底层设计和实现，并在实践中不断地练习，性能估算才会越来越准。

3.3 数据分布

分布式系统区别于传统单机系统在于能够将数据分布到多个节点，并在多个节点之间实现负载均衡。数据分布的方式主要有两种，一种是哈希分布，如一致性哈希，代表系统为 Amazon 的 Dynamo 系统；另外一种方法是顺序分布，即每张表格上的数据按照主键整体有序，代表系统为 Google 的 Bigtable 系统。Bigtable 将一张大表根据主键切分为有序的范围，每个有序范围是一个子表。

将数据分散到多台机器后，需要尽量保证多台机器之间的负载是比较均衡的。衡量机器负载涉及的因素很多，如机器 Load 值，CPU，内存，磁盘以及网络等资源使用情况，读写请求数及请求量，等等，分布式存储系统需要能够自动识别负载高的节点，当某台机器的负载较高时，将它服务的部分数据迁移到其他机器，实现自动负载均衡。

分布式存储系统的一个基本要求就是透明性，包括数据分布透明性，数据迁移透

明性，数据复制透明性，故障处理透明性。本节介绍数据分布以及数据迁移相关的基础知识。

3.3.1　哈希分布

哈希取模的方法很常见，其方法是根据数据的某一种特征计算哈希值，并将哈希值与集群中的服务器建立映射关系，从而将不同哈希值的数据分布到不同的服务器上。所谓数据特征可以是 key-value 系统中的主键（key），也可以是其他与业务逻辑相关的值。例如，将集群中的服务器按 0 到 N−1 编号（N 为服务器的数量），根据数据的主键（hash(key) % N）或者数据所属的用户 id（hash(user_id) % N）计算哈希值，来决定将数据映射到哪一台服务器。

如果哈希函数的散列特性很好，哈希方式可以将数据比较均匀地分布到集群中去。而且，哈希方式需要记录的元信息也非常简单，每个节点只需要知道哈希函数的计算方式以及模的服务器的个数就可以计算出处理的数据应该属于哪台机器。然而，找出一个散列特性很好的哈希函数是很难的。这是因为，如果按照主键散列，那么同一个用户 id 下的数据可能被分散到多台服务器，这会使得一次操作同一个用户 id 下的多条记录变得困难；如果按照用户 id 散列，容易出现“数据倾斜”（data skew）问题，即某些大用户的数据量很大，无论集群的规模有多大，这些用户始终由一台服务器处理。

处理大用户问题一般有两种方式，一种方式是手动拆分，即线下标记系统中的大用户（例如运行一次 MapReduce 作业），并根据这些大用户的数据量将其拆分到多台服务器上。这就相当于在哈希分布的基础上针对这些大用户特殊处理；另一种方式是自动拆分，即数据分布算法能够动态调整，自动将大用户的数据拆分到多台服务器上。

传统的哈希分布算法还有一个问题：当服务器上线或者下线时，N 值发生变化，数据映射完全被打乱，几乎所有的数据都需要重新分布，这将带来大量的数据迁移。

一种思路是不再简单地将哈希值和服务器个数做除法取模映射，而是将哈希值与服务器的对应关系作为元数据，交给专门的元数据服务器来管理。访问数据时，首先计算哈希值，再查询元数据服务器，获得该哈希值对应的服务器。这样，集群扩容时，可以将部分哈希值分配给新加入的机器并迁移对应的数据。

另一种思路就是采用一致性哈希（Distributed Hash Table，DHT）算法。算法思想如下：给系统中每个节点分配一个随机 token，这些 token 构成一个哈希环。执行数据存放操作时，先计算 Key（主键）的哈希值，然后存放到顺时针方向第一个大于或者等于该哈希值的 token 所在的节点。一致性哈希的优点在于节点加入 / 删除时只会影响到在哈希环中相邻的节点，而对其他节点没影响。

如图 3-2 所示，假设哈希空间为 0~2^n，一致性哈希算法如下：

- ❑ 首先求出每个服务器的 hash 值，将其配置到一个 0~2^n 的圆环区间上；
- ❑ 其次使用同样的方法求出待存储对象的主键哈希值，也将其配置到这个圆环上；
- ❑ 然后从数据映射的位置开始顺时针查找，将数据分布到找到的第一个服务器节点。

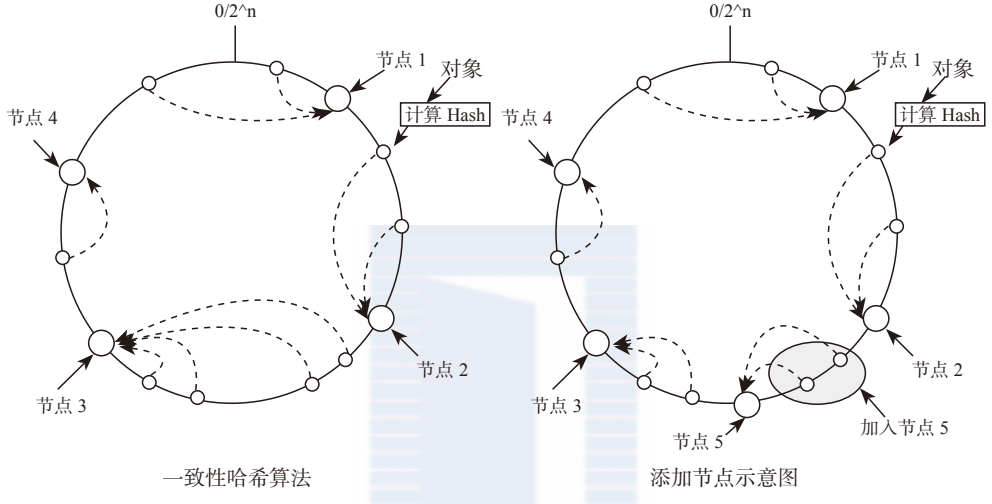

图 3-2 一致性哈希算法

增加服务节点 5 以后，某些原来分布到节点 3 的数据需要迁移到节点 5，其他数据分布均保持不变。可以看出，一致性哈希算法在很大程度上避免了数据迁移。

为了查找集群中的服务器，需要维护每台机器在哈希环中位置信息，常见的做法如下。

（1）O(1) 位置信息

每台服务器记录它的前一个以及后一个节点的位置信息。这种做法的维护的节点位置信息的空间复杂度为 O(1)，然而每一次查找都可能遍历整个哈希环中的所有服务器，即时间复杂度为 O(N)，其中，N 为服务器数量。

（2）O(log$N$) 位置信息

假设哈希空间为 0~2^n（即 N=2^n），以 Chord 系统为例，为了加速查找，它在每台服务器维护了一个大小为 n 的路由表（finger table），$FT_p[i]$=succ(p+2^{i-1})，其中 p 为服务器在哈希环中的编号，路由表中的第 i 个元素记录了编号为 p+2^{i-1} 的后继节点。通过维护 O(log$N$) 的位置信息，查找的时间复杂度改进为 O(log$N$)。

（3）O(N) 位置信息

Dynamo 系统通过牺牲空间换时间，在每台服务器维护整个集群中所有服务器的位

置信息，将查找服务器的时间复杂度降为 O(1)。工程上一般都采用这种做法，Dynamo 这样的 P2P 系统在每个服务器节点上都维护了所有服务器的位置信息，而带有总控节点的存储系统往往由总控节点统一维护。

一致性哈希还需要考虑负载均衡，增加服务节点 node5 后，虽然只影响到 node5 的后继，即 node3 的数据分布，但 node3 节点需要迁移的数据过多，整个集群的负载不均衡。一种自然的想法是将需要迁移的数据分散到整个集群，每台服务器只需要迁移 1/N 的数据量。为此，Dynamo 中引入虚拟节点的概念，5.1 节会详细讨论。

3.3.2　顺序分布

哈希散列破坏了数据的有序性，只支持随机读取操作，不能够支持顺序扫描。某些系统可以在应用层做折衷，比如互联网应用经常按照用户来进行数据拆分，并通过哈希方法进行数据分布，同一个用户的数据分布到相同的存储节点，允许对同一个用户的数据执行顺序扫描，由应用层解决跨多个用户的操作问题。另外，这种方式可能出现某些用户的数据量太大的问题，由于用户的数据限定在一个存储节点，无法发挥分布式存储系统的多机并行处理能力。

顺序分布在分布式表格系统中比较常见，一般的做法是将大表顺序划分为连续的范围，每个范围称为一个子表，总控服务器负责将这些子表按照一定的策略分配到存储节点上。如图 3-3 所示，用户表 (User 表) 的主键范围为 1 ～ 7000，在分布式存储系统中划分为多个子表，分别对应数据范围 1 ～ 1000, 1001 ～ 2000，…6001 ～ 7000。Meta 表是可选的，某些系统只有根表（Root 表）一级索引，在 Root 表中维护用户表的位置信息，即每个 User 子表在哪个存储节点上。为了支持更大的集群规模，Bigtable 这样的系统将索引分为两级 ：根表以及元数据表（Meta 表），由 Meta 表维护 User 表的位置信

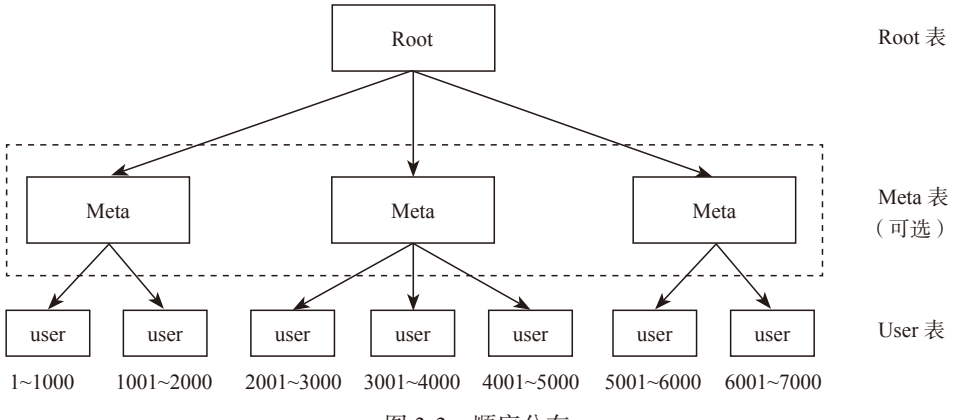

图 3-3　顺序分布

息，而 Root 表用来维护 Meta 表的位置信息。读 User 表时，需要通过 Meta 表查找相应的 User 子表所在的存储节点，而读取 Meta 表又需要通过 Root 表查找相应的 Meta 子表所在的存储节点。

顺序分布与 B+ 树数据结构比较类似，每个子表相当于叶子节点，随着数据的插入和删除，某些子表可能变得很大，某些变得很小，数据分布不均匀。如果采用顺序分布，系统设计时需要考虑子表的分裂与合并，这将极大地增加系统复杂度。子表分裂指当一个子表太大超过一定阀值时需要分裂为两个子表，从而有机会通过系统的负载均衡机制分散到多个存储节点。子表合并一般由数据删除引起，当相邻的两个子表都很小时，可以合并为一个子表。一般来说，单个服务节点能够服务的子表数量是有限的，比如 4000~10000 个，子表合并的目的是为了防止系统中出现过多太小的子表，减少系统中的元数据。

3.3.3 负载均衡

分布式存储系统的每个集群中一般有一个总控节点，其他节点为工作节点，由总控节点根据全局负载信息进行整体调度。工作节点刚上线时，总控节点需要将数据迁移到该节点，另外，系统运行过程中也需要不断地执行迁移任务，将数据从负载较高的工作节点迁移到负载较低的工作节点。

工作节点通过心跳包（Heartbeat，定时发送）将节点负载相关的信息，如 CPU，内存，磁盘，网络等资源使用率，读写次数及读写数据量等发送给主控节点。主控节点计算出工作节点的负载以及需要迁移的数据，生成迁移任务放入迁移队列中等待执行。需要注意的是，负载均衡操作需要控制节奏，比如一台全新的工作节点刚上线的时候，由于负载最低，如果主控节点将大量的数据同时迁移到这台新加入的机器，整个系统在新增机器的过程中服务能力会大幅下降。负载均衡操作需要做到比较平滑，一般来说，从新机器加入到集群负载达到比较均衡的状态需要较长一段时间，比如 30 分钟到一个小时。

负载均衡需要执行数据迁移操作。在分布式存储系统中往往会存储数据的多个副本，其中一个副本为主副本，其他副本为备副本，由主副本对外提供服务。迁移备副本不会对服务造成影响，迁移主副本也可以首先将数据的读写服务切换到其他备副本。整个迁移过程可以做到无缝，对用户完全透明。

假设数据分片 D 有两个副本 D1 和 D2，分别存储在工作节点 A1 和 A2，其中，D1 为主副本，提供读写服务，D2 为备副本。如果需要将 D1 从工作节点 A1 中迁移出去，大致的操作步骤如下：

1）将数据分片 D 的读写服务由工作节点 A1 切换到 A2，D2 变成主副本；

2）增加副本：选择某个节点，例如 B 节点，增加 D 的副本，即 B 节点从 A2 节点获取 D 的副本数据（D2）并与之保持同步；

3）删除工作节点 A1 上的 D1 副本。

3.4　复制

为了保证分布式存储系统的高可靠和高可用，数据在系统中一般存储多个副本。当某个副本所在的存储节点出现故障时，分布式存储系统能够自动将服务切换到其他的副本，从而实现自动容错。分布式存储系统通过复制协议将数据同步到多个存储节点，并确保多个副本之间的数据一致性。

同一份数据的多个副本中往往有一个副本为主副本（Primary），其他副本为备副本（Backup），由主副本将数据复制到备份副本。复制协议分为两种，强同步复制以及异步复制，二者的区别在于用户的写请求是否需要同步到备副本才可以返回成功。假如备份副本不止一个，复制协议还会要求写请求至少需要同步到几个备副本。当主副本出现故障时，分布式存储系统能够将服务自动切换到某个备副本，实现自动容错。

一致性和可用性是矛盾的，强同步复制协议可以保证主备副本之间的一致性，但是当备副本出现故障时，也可能阻塞存储系统的正常写服务，系统的整体可用性受到影响；异步复制协议的可用性相对较好，但是一致性得不到保障，主副本出现故障时还有数据丢失的可能。

本节首先介绍常见的数据复制协议，接着讨论如何在一致性与可用性之间的进行权衡。

3.4.1　复制的概述

分布式存储系统中数据保存多个副本，一般来说，其中一个副本为主副本，其他副本为备副本，常见的做法是数据写入到主副本，由主副本确定操作的顺序并复制到其他副本。

如图 3-4 所示，客户端将写请求发送给主副本，主副本将写请求复制到其他备副本，常见的做法是同步操作日志（Commit Log）。主副本首先将操作日志同步到备副本，备副本回放操作日志，完成后通知主副本。接着，主副本修改本机，等到所有的操作都完成后再通知客户端写成功。图 3-4 中的复制协议要求主备同步成功才可以返回客户端写成功，这种协议称为强同步协议。强同步协议提供了强一致性，但是，如果备副本出现问题将阻塞写操作，系统可用性较差。

假设所有副本的个数为 N，且 N > 2，即备副本个数大于 1。那么，实现强同步协

议时，主副本可以将操作日志并发地发给所有备副本并等待回复，只要至少 1 个备副本返回成功就可以回复客户端操作成功。强同步的好处在于如果主副本出现故障，至少有 1 个备副本拥有完整的数据，分布式存储系统可以自动地将服务切换到最新的备副本而不用担心数据丢失的情况。

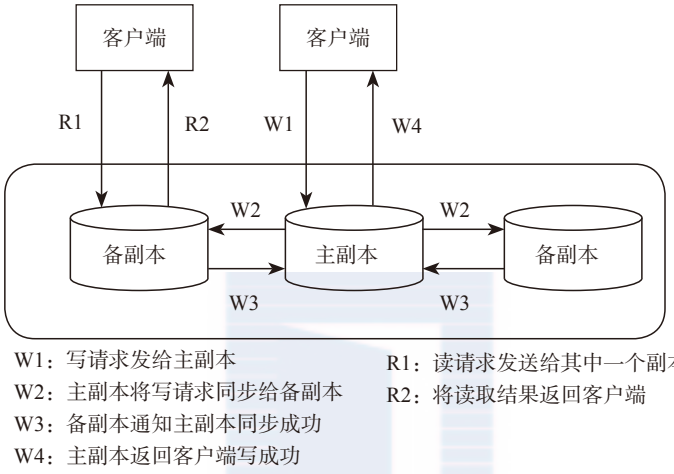

图 3-4　主备复制协议

与强同步对应的复制方式是异步复制。在异步模式下，主副本不需要等待备副本的回应，只需要本地修改成功就可以告知客户端写操作成功。另外，主副本通过异步机制，比如单独的复制线程将客户端修改操作推送到其他副本。异步复制的好处在于系统可用性较好，但是一致性较差，如果主副本发生不可恢复故障，可能丢失最后一部分更新操作。

强同步复制和异步复制都是将主副本的数据以某种形式发送到其他副本，这种复制协议称为基于主副本的复制协议（Primary-based protocol）。这种方法要求在任何时刻只能有一个副本为主副本，由它来确定写操作之间的顺序。如果主副本出现故障，需要选举一个备副本成为新的主副本，这步操作称为选举，经典的选举协议为 Paxos 协议，3.7.2 节将专门进行介绍。

主备副本之间的复制一般通过操作日志来实现。操作日志的原理很简单：为了利用好磁盘的顺序读写特性，将客户端的写操作先顺序写入到磁盘中，然后应用到内存中，由于内存是随机读写设备，可以很容易通过各种数据结构，比如 B+ 树将数据有效地组织起来。当服务器宕机重启时，只需要回放操作日志就可以恢复内存状态。为了提高系统的并发能力，系统会积攒一定的操作日志再批量写入到磁盘中，这种技术一般称为成组提交。

如果每次服务器出现故障都需要回放所有的操作日志，效率是无法忍受的，检查点（checkpoint）正是为了解决这个问题。系统定期将内存状态以检查点文件的形式 dump 到磁盘中，并记录检查点时刻对应的操作日志回放点。检查点文件成功创建后，回放点之前的日志可以被垃圾回收，以后如果服务器出现故障，只需要回放检查点之后的操作日志。

除了基于主副本的复制协议，分布式存储系统中还可能使用基于写多个存储节点的复制协议（Replicated-write protocol）。比如 Dynamo 系统中的 NWR 复制协议，其中，N 为副本数量，W 为写操作的副本数，R 为读操作的副本数。NWR 协议中多个副本不再区分主和备，客户端根据一定的策略往其中的 W 个副本写入数据，读取其中的 R 个副本。只要 W + R > N，可以保证读到的副本中至少有一个包含了最新的更新。然而，这种协议的问题在于不同副本的操作顺序可能不一致，从多个副本读取时可能出现冲突。这种方式在实际系统中比较少见，不建议使用。

3.4.2　一致性与可用性

来自 Berkerly 的 Eric Brewer 教授提出了一个著名的 CAP 理论：一致性（Consistency），可用性（Availability）以及分区可容忍性（Tolerance of network Partition）三者不能同时满足。笔者认为没有必要纠结 CAP 理论最初的定义，在工程实践中，可以将 C、A、P 三者按如下方式理解：

- 一致性：读操作总是能读取到之前完成的写操作结果，满足这个条件的系统称为强一致系统，这里的“之前”一般对同一个客户端而言；
- 可用性：读写操作在单台机器发生故障的情况下仍然能够正常执行，而不需要等待发生故障的机器重启或者其上的服务迁移到其他机器；
- 分区可容忍性：机器故障、网络故障、机房停电等异常情况下仍然能够满足一致性和可用性。

分布式存储系统要求能够自动容错，也就是说，分区可容忍性总是需要满足的，因此，一致性和写操作的可用性不能同时满足。

如果采用强同步复制，保证了存储系统的一致性，然而，当主备副本之间出现网络或者其他故障时，写操作将被阻塞，系统的可用性无法得到满足。如果采用异步复制，保证了存储系统的可用性，但是无法做到强一致性。

存储系统设计时需要在一致性和可用性之间权衡，在某些场景下，不允许丢失数据，在另外一些场景下，极小的概率丢失部分数据时允许的，可用性更加重要。例如，Oracle 数据库的 DataGuard 复制组件包含三种模式：

- 最大保护模式（Maximum Protection）：即强同步复制模式，写操作要求主库先

将操作日志（数据库的 redo/undo 日志）同步到至少一个备库才可以返回客户端成功。这种模式保证即使主库出现无法恢复的故障，比如硬盘损坏，也不会丢失数据。

- 最大性能模式（Maximum Performance）：即异步复制模式，写操作只需要在主库上执行成功就可以返回客户端成功，主库上的后台线程会将重做日志通过异步的方式复制到备库。这种方式保证了性能及可用性，但是可能丢失数据。
- 最大可用性模式（Maximum Availability）：上述两种模式的折衷。正常情况下相当于最大保护模式，如果主备之间的网络出现故障，切换为最大性能模式。这种模式在一致性和可用性之间做了一个很好的权衡，推荐大家在设计存储系统时使用。

3.5 容错

随着集群规模变得越来越大，故障发生的概率也越来越大，大规模集群每天都有故障发生。容错是分布式存储系统设计的重要目标，只有实现了自动化容错，才能减少人工运维成本，实现分布式存储的规模效应。

单台服务器故障的概率是不高的，然而，只要集群的规模足够大，每天都可能有机器故障发生，系统需要能够自动处理。首先，分布式存储系统需要能够检测到机器故障，在分布式系统中，故障检测往往通过租约（Lease）协议实现。接着，需要能够将服务复制或者迁移到集群中的其他正常服务的存储节点。

本节首先介绍 Google 某数据中心发生的故障，接着讨论分布式系统中的故障检测以及恢复方法。

3.5.1 常见故障

来自 Google 的 Jeff Dean 在 LADIS 2009 报告中介绍了 Google 某数据中心第一年运行发生的故障数据，如表 3-1 所示。

表 3-1 Google 某数据中心第一年运行故障

发生频率	故障类型	影响范围
0.5	数据中心过热	5 分钟之内大部分机器断电，一到两天恢复
1	配电装置（PDU）故障	大约 500 到 1000 台机器瞬间下线，6 小时恢复
1	机架调整	大量告警，500~1000 台机器断电，6 小时恢复
1	网络重新布线	大约 5% 机器下线超过两天
20	机架故障	40 到 80 台机器瞬间下线，1 到 6 小时恢复
5	机架不稳定	40 到 80 台机器发生 50% 丢包
12	路由器重启	DNS 和对外虚 IP 服务失效约几分钟

（续）

发生频率	故障类型	影响范围
3	路由器故障	需要立即切换流量，持续约 1 小时
几十	DNS 故障	持续约 30 秒
1000	单机故障	机器无法提供服务
几千	硬盘故障	硬盘数据丢失

从表 3-1 可以看出，单机故障和磁盘故障发生概率最高，几乎每天都有多起事故，系统设计首先需要对单台服务器故障进行容错处理。一般来说，分布式存储系统会保存多份数据，当其中一份数据所在服务器发生故障时，能通过其他副本继续提供服务。另外，机架故障发生的概率相对也是比较高的，需要避免将数据的所有副本都分布在同一个机架内。最后，还可能出现磁盘响应慢，内存错误，机器配置错误，数据中心之间网路连接不稳定，等等。

3.5.2　故障检测

容错处理的第一步是故障检测，心跳是一种很自然的想法。假设总控机 A 需要确认工作机 B 是否发生故障，那么总控机 A 每隔一段时间，比如 1 秒，向工作机 B 发送一个心跳包。如果一切正常，机器 B 将响应机器 A 的心跳包；否则，机器 A 重试一定次数后认为机器 B 发生了故障。然而，机器 A 收不到机器 B 的心跳并不能确保机器 B 发生故障并停止了服务，在系统运行过程中，可能发生各种错误，比如机器 A 与机器 B 之间网络发生问题，机器 B 过于繁忙导致无法响应机器 A 的心跳包。由于机器 B 发生故障后，往往需要将它上面的服务迁移到集群中的其他服务器，为了保证强一致性，需要确保机器 B 不再提供服务，否则将出现多台服务器同时服务同一份数据而导致数据不一致的情况。

这里的问题是机器 A 和机器 B 之间需要对“机器 B 是否应该被认为发生故障且停止服务”达成一致，Fisher 指出，异步网络中的多台机器无法达成一致。当然，在实践中，由于机器之间会进行时钟同步，我们总是假设 A 和 B 两台机器的本地时钟相差不大，比如相差不超过 0.5 秒。这样，我们可以通过租约（Lease）机制进行故障检测。租约机制就是带有超时时间的一种授权。假设机器 A 需要检测机器 B 是否发生故障，机器 A 可以给机器 B 发放租约，机器 B 持有的租约在有效期内才允许提供服务，否则主动停止服务。机器 B 的租约快要到期的时候向机器 A 重新申请租约。正常情况下，机器 B 通过不断申请租约来延长有效期，当机器 B 出现故障或者与机器 A 之间的网络发生故障时，机器 B 的租约将过期，从而机器 A 能够确保机器 B 不再提供服务，机器 B 的服务可以被安全地迁移到其他服务器。

需要注意的是，实现租约机制时需要考虑一个提前量。假设机器 B 的租约有效期为 10 秒，那么机器 A 需要加上一个提前量，比如 11 秒时，才可以认为机器 B 的租约过期。这样，即使机器 A 和机器 B 的时钟不一致，只要相差不会太大，都可以保证机器 B 的租约到期并且已经不再提供服务。

3.5.3 故障恢复

当总控机检测到工作机发生故障时，需要将服务迁移到其他工作机节点。常见的分布式存储系统分为两种结构：单层结构和双层结构。大部分系统为单层结构，在系统中对每个数据分片维护多个副本；只有类 Bigtable 系统为双层结构，将存储和服务分为两层，存储层对每个数据分片维护多个副本，服务层只有一个副本提供服务。单层结构和双层结构的故障恢复机制有所不同。

单层结构的分布式存储系统维护了多个副本，例如副本个数为 3，主备副本之间通过操作日志同步。如图 3-5 所示，某单层结构的分布式存储系统有 3 个数据分片 A、B、C，每个数据分片存储了三个副本。其中，A1，B1，C1 为主副本，分别存储在节点 1，节点 2 以及节点 3。假设节点 1 发生故障，将被总控节点检测到，总控节点选择一个最新的副本，比如 A2 或者 A3 替换 A1 成为新的主副本并提供写服务。节点下线分为两种情况：一种是临时故障，节点过一段时间将重新上线；另一种情况是是永久性故障，比如硬盘损坏。总控节点一般需要等待一段时间，比如 1 个小时，如果之前下线的节点重新上线，可以认为是临时性故障，否则，认为是永久性故障。如果发生永久性故障，需要执行增加副本操作，即选择某个节点拷贝 A 的数据，成为 A 的备副本。

两层结构的分布式存储系统会将所有的数据持久化写入底层的分布式文件系统，每个数据分片同一时刻只有一个提供服务的节点。如图 3-5 所示，某双层结构的分布式存储系统有 3 个数据分片，A、B 和 C。它们分别被节点 1，节点 2 和节点 3 所服务。当节点 1 发生故障时，总控节点将选择一个工作节点，比如节点 2，加载 A 的服务。由于 A 的所有数据都存储在共享的分布式文件系统中，节点 2 只需要从底层分布式文件系统读取 A 的数据并加载到内存中。

节点故障会影响系统服务，在故障检测以及故障恢复的过程中，不能提供写服务及强一致性读服务。停服务时间包含两个部分，故障检测时间以及故障恢复时间。故障检测时间一般在几秒到十几秒，和集群规模密切相关，集群规模越大，故障检测对总控节点造成的压力就越大，故障检测时间就越长。故障恢复时间一般很短，单层结构的备副本和主副本之间保持实时同步，切换为主副本的时间很短；两层结构故障恢复往往实现成只需要将数据的索引，而不是所有的数据，加载到内存中。

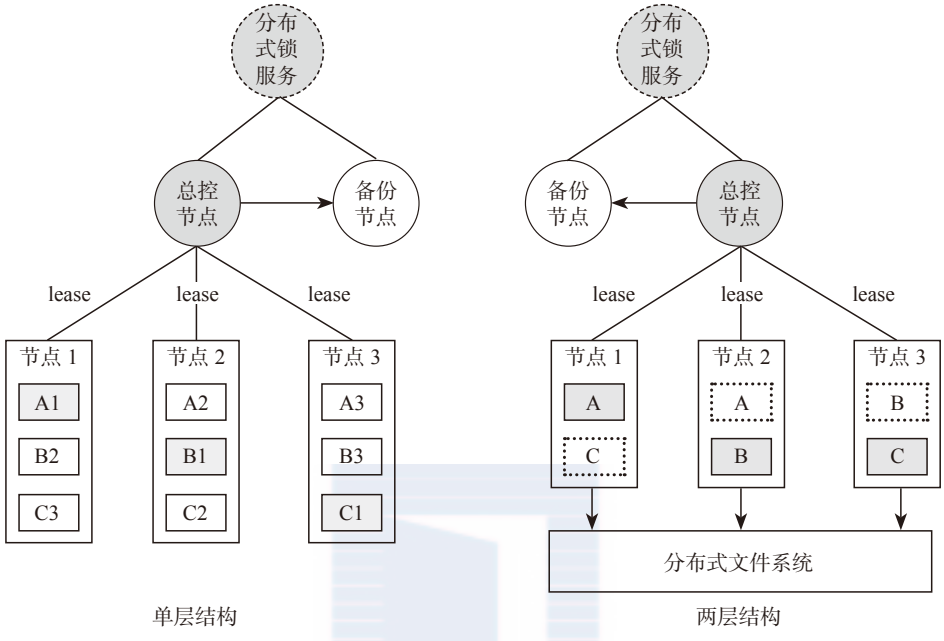

图 3-5　故障恢复

总控节点自身也可能出现故障，为了实现总控节点的高可用性（High Availability），总控节点的状态也将实时同步到备机，当故障发生时，可以通过外部服务选举某个备机作为新的总控节点，而这个外部服务也必须是高可用的。为了进行选主或者维护系统中重要的全局信息，可以维护一套通过 Paxos 协议实现的分布式锁服务，比如 Google Chubby 或者它的开源实现 Apache Zookeeper。

3.6　可扩展性

通过数据分布，复制以及容错等机制，能够将分布式存储系统部署到成千上万台服务器。可扩展性的实现手段很多，如通过增加副本个数或者缓存提高读取能力，将数据分片使得每个分片可以被分配到不同的工作节点以实现分布式处理，把数据复制到多个数据中心，等等。

分布式存储系统大多都带有总控节点，很多人会自然地联想到总控节点的瓶颈问题，认为 P2P 架构更有优势。然而，事实却并非如此，主流的分布式存储系统大多带有总控节点，且能够支持成千上万台的集群规模。

另外，传统的数据库也能够通过分库分表等方式对系统进行水平扩展，当系统处理能力不足时，可以通过增加存储节点来扩容。

那么，如何衡量分布式存储系统的可扩展性，它与传统数据库的可扩展性又有什么区别？可扩展性不能简单地通过系统是否为 P2P 架构或者是否能够将数据分布到多个存储节点来衡量，而应该综合考虑节点故障后的恢复时间，扩容的自动化程度，扩容的灵活性等。

本节首先讨论总控节点是否会成为性能瓶颈，接着介绍传统数据库的可扩展性，最后讨论同构系统与异构系统增加节点时的差别。

3.6.1 总控节点

分布式存储系统中往往有一个总控节点用于维护数据分布信息，执行工作机管理，数据定位，故障检测和恢复，负载均衡等全局调度工作。通过引入总控节点，可以使得系统的设计更加简单，并且更加容易做到强一致性，对用户友好。那么，总控节点是否会成为性能瓶颈呢？

分为两种情况：分布式文件系统的总控节点除了执行全局调度，还需要维护文件系统目录树，内存容量可能会率先成为性能瓶颈；而其他分布式存储系统的总控节点只需要维护数据分片的位置信息，一般不会成为瓶颈。另外，即使是分布式文件系统，只要设计合理，也能够扩展到几千台服务器。例如，Google 的分布式文件系统能够扩展到 8000 台以上的集群，开源的 Hadoop 也能够扩展到 3000 台以上的集群。当然，设计时需要减少总控节点的负载，比如 Google 的 GFS 舍弃了对小文件的支持，并且把对数据的读写控制权下放到工作机 ChunkServer，通过客户端缓存元数据减少对总控节点的访问等。

如果总控节点成为瓶颈，例如需要支持超过一万台的集群规模，或者需要支持海量的小文件，那么，可以采用两级结构，如图 3-6 所示。在总控机与工作机之间增加一层元数据节点，每个元数据节点只维护一部分而不是整个分布式文件系统的元数据。这样，总控机也只需要维护元数据节点的元数据，不可能成为性能瓶颈。假设分布式文件系统（Distributed File System，DFS）中有 100 个元数据节点，每个元数据节点服务 1 亿个文件，系统总共可以服务 100 亿个文件。图 3-6 中的 DFS 客户端定位 DFS 工作机时，需要首先访问 DFS 总控机找到 DFS 元数据服务器，再通过元数据服务器找到 DFS 工作机。虽然看似增加了一次网络请求，但是客户端总是能够缓存 DFS 总控机上的元数据，因此并不会带来额外的开销。

3.6.2 数据库扩容

数据库可扩展性实现的手段包括：通过主从复制提高系统的读取能力，通过垂直拆

分和水平拆分将数据分布到多个存储节点，通过主从复制将系统扩展到多个数据中心。当主节点出现故障时，可以将服务切换到从节点；另外，当数据库整体服务能力不足时，可以根据业务的特点重新拆分数据进行扩容。

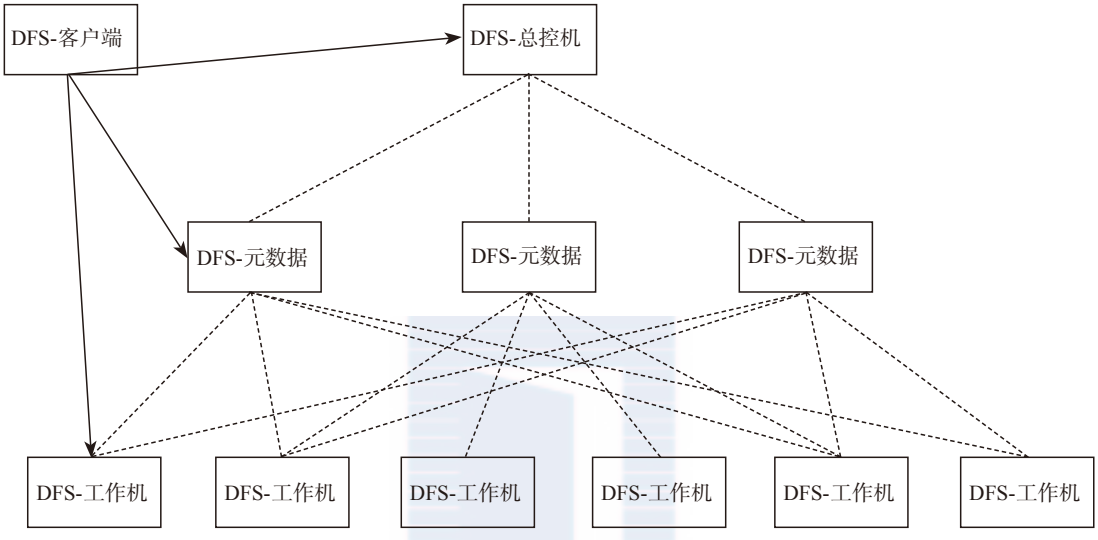

图 3-6　两级元数据架构

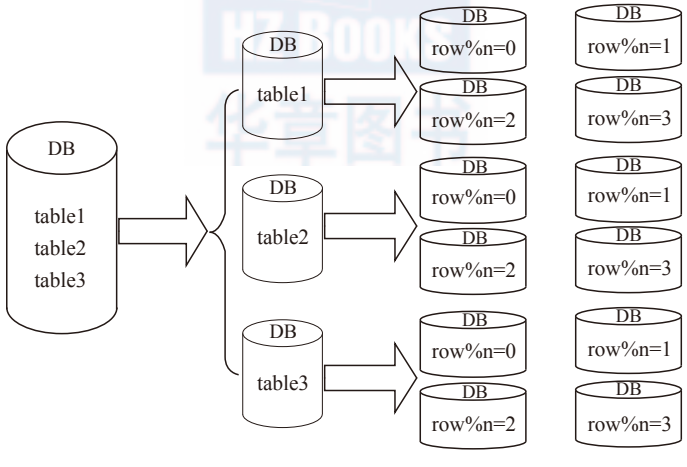

图 3-7　数据库拆分示意图

如图 3-7 所示，假设数据库中有三张表格 table1、table2 以及 table3，先按照业务将三张表垂直拆分到不同的 DB 中，再将每张表通过哈希的方式水平拆分到不同的存储节点。每个拆分后的 DB 通过主从复制维护多个副本，且允许分布到多个数据中心。如果系统的读取能力不足，可以通过增加副本的方式解决，如果系统的写入能力不足，可以

根据业务的特点重新拆分数据，常见的做法为双倍扩容，即将每个分片的数据拆分为两个分片，扩容的过程中需要迁移一半的数据到新加入的存储节点。

传统的数据库架构在可扩展性上面临如下问题：

- 扩容不够灵活。传统数据库架构一般采用双倍扩容的做法，很难做到按需扩容。假设系统中已经有 16 个存储节点，如果希望将系统的服务能力提高 5%，只需要新增 1 个而不是 16 个存储节点。
- 扩容不够自动化。传统数据库架构扩容时需要迁移大量的数据，整个过程时间较长，容易发生异常情况，且数据划分的规则往往和业务相关，很难做到自动化。
- 增加副本时间长。如果某个主节点出现永久性故障，比如硬盘故障，需要增加一个副本，整个过程需要拷贝大量的数据，耗费的时间很长。

3.6.3 异构系统

传统数据库扩容与大规模存储系统的可扩展性有何区别呢？为了说明这一问题，我们首先定义同构系统，如图 3-8 所示。

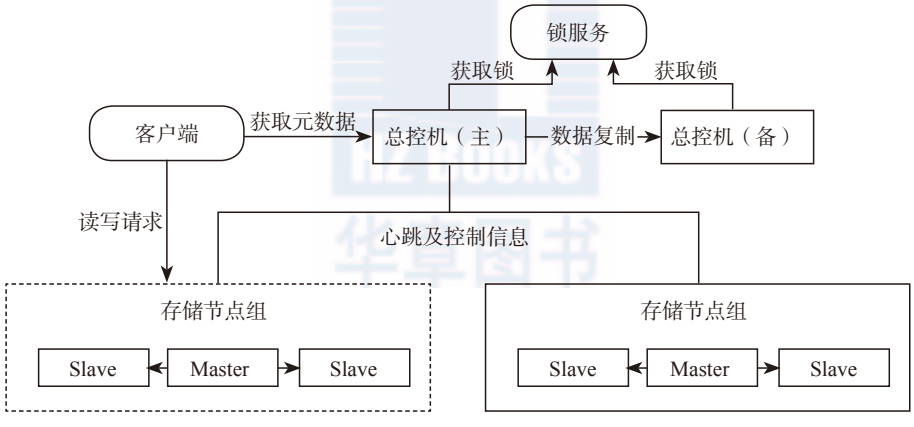

图 3-8 同构系统

将存储节点分为若干组，每个组内的节点服务完全相同的数据，其中有一个节点为主节点，其他节点为备节点。由于同一个组内的节点服务相同的数据，这样的系统称为同构系统。同构系统的问题在于增加副本需要迁移的数据量太大，假设每个存储节点服务的数据量为 1TB，内部传输带宽限制为 20MB/s，那么增加副本拷贝数据需要的时间为 1TB/20MB/s = 50000s ，大约十几个小时，由于拷贝数据的过程中存储节点再次发生故障的概率很高，所以这样的架构很难做到自动化，不适用大规模分布式存储系统。

大规模分布式存储系统要求具有线性可扩展性，即随时加入或者删除一个或者多个

存储节点，系统的处理能力与存储节点的个数成线性关系。为了实现线性可扩展性，存储系统的存储节点之间是异构的。否则，当集群规模达到一定程度后，增加节点将变得特别困难。异构系统将数据划分为很多大小接近的分片，每个分片的多个副本可以分布到集群中的任何一个存储节点。如果某个节点发生故障，原有的服务将由整个集群而不是某几个固定的存储节点来恢复。

如图 3-9 所示，系统中有五个分片（A，B，C，D，E），每个分片包含三个副本，如分片 A 的三个副本分别为 A1，A2 以及 A3。假设节点 1 发生永久性故障，那么可以从剩余的节点中任意选择健康的节点来增加 A，B 以及 E 的副本。由于整个集群都参与到节点 1 的故障恢复过程，故障恢复时间很短，而且集群规模越大，优势就会越明显。

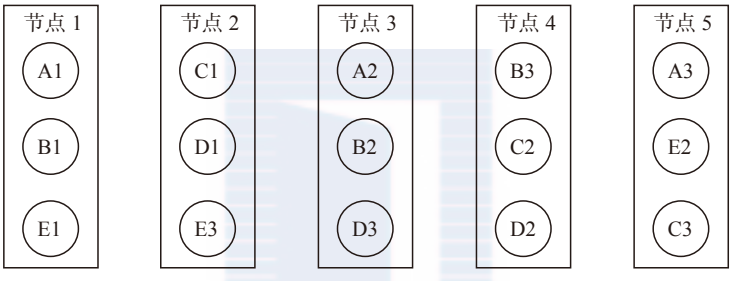

图 3-9　异构系统

3.7　分布式协议

分布式系统涉及的协议很多，例如租约，复制协议，一致性协议，其中以两阶段提交协议和 Paxos 协议最具有代表性。两阶段提交协议用于保证跨多个节点操作的原子性，也就是说，跨多个节点的操作要么在所有节点上全部执行成功，要么全部失败。Paxos 协议用于确保多个节点对某个投票（例如哪个节点为主节点）达成一致。本节介绍这两个分布式协议。

3.7.1　两阶段提交协议

两阶段提交协议（Two-phase Commit，2PC）经常用来实现分布式事务，在两阶段协议中，系统一般包含两类节点：一类为协调者（coordinator），通常一个系统中只有一个；另一类为事务参与者（participants，cohorts 或 workers），一般包含多个。协议中假设每个节点都会记录操作日志并持久化到非易失性存储介质，即使节点发生故障日志也不会丢失。顾名思义，两阶段提交协议由两个阶段组成。在正常的执行过程中，这两个阶段的执行过程如下所述：

- ❑ **阶段 1**：请求阶段（Prepare Phase）。在请求阶段，协调者通知事务参与者准备提交或者取消事务，然后进入表决过程。在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地执行成功）或者取消（事务参与者本地执行失败）。
- ❑ **阶段 2**：提交阶段（Commit Phase）。在提交阶段，协调者将基于第一个阶段的投票结果进行决策：提交或者取消。当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者通知所有的参与者取消事务。参与者在接收到协调者发来的消息后将执行相应的操作。

例如，A 组织 B、C 和 D 三个人去爬长城：如果所有人都同意去爬长城，那么活动将举行；如果有一人不同意去爬长城，那么活动将取消。用 2PC 算法解决该问题的过程如下：

1）首先 A 将成为该活动的协调者，B、C 和 D 将成为该活动的参与者。

2）准备阶段：A 发邮件给 B、C 和 D，提出下周三去爬山，问是否同意。那么此时 A 需要等待 B、C 和 D 的回复。B、C 和 D 分别查看自己的日程安排表。B、C 发现自己在当日没有活动安排，则发邮件告诉 A 他们同意下周三去爬长城。由于某种原因，D 白天没有查看邮件。那么此时 A、B 和 C 均需要等待。到晚上的时候，D 发现了 A 的邮件，然后查看日程安排，发现下周三当天已经有别的安排，那么 D 回复 A 说活动取消吧。

3）此时 A 收到了所有活动参与者的邮件，并且 A 发现 D 下周三不能去爬山。那么 A 将发邮件通知 B、C 和 D，下周三爬长城活动取消。此时 B、C 回复 A“太可惜了”，D 回复 A“不好意思”。至此该事务终止。

通过该例子可以发现，2PC 协议存在明显的问题。假如 D 一直不能回复邮件，那么 A、B 和 C 将不得不处于一直等待的状态。并且 B 和 C 所持有的资源一直不能释放，即下周三不能安排其他活动。当然，A 可以发邮件告诉 D 如果晚上六点之前不回复活动就自动取消，通过引入事务的超时机制防止资源一直不能释放的情况。更为严重的是，假如 A 发完邮件后生病住院了，即使 B、C 和 D 都发邮件告诉 A 同意下周三去爬长城，如果 A 没有备份，事务将被阻塞，B、C 和 D 下周三都不能安排其他活动。

两阶段提交协议可能面临两种故障：

- ❑ 事务参与者发生故障。给每个事务设置一个超时时间，如果某个事务参与者一直不响应，到达超时时间后整个事务失败。
- ❑ 协调者发生故障。协调者需要将事务相关信息记录到操作日志并同步到备用协调者，假如协调者发生故障，备用协调者可以接替它完成后续的工作。如果没有备用协调者，协调者又发生了永久性故障，事务参与者将无法完成事务而一直等待

下去。

总而言之，两阶段提交协议是阻塞协议，执行过程中需要锁住其他更新，且不能容错，大多数分布式存储系统都采用敬而远之的做法，放弃对分布式事务的支持。

3.7.2　Paxos 协议

Paxos 协议用于解决多个节点之间的一致性问题。多个节点之间通过操作日志同步数据，如果只有一个节点为主节点，那么，很容易确保多个节点之间操作日志的一致性。考虑到主节点可能出现故障，系统需要选举出新的主节点。Paxos 协议正是用来实现这个需求。只要保证了多个节点之间操作日志的一致性，就能够在这些节点上构建高可用的全局服务，例如分布式锁服务，全局命名和配置服务等。

为了实现高可用性，主节点往往将数据以操作日志的形式同步到备节点。如果主节点发生故障，备节点会提议自己成为主节点。这里存在的问题是网络分区的时候，可能会存在多个备节点提议（Proposer，提议者）自己成为主节点。Paxos 协议保证，即使同时存在多个 proposer，也能够保证所有节点最终达成一致，即选举出唯一的主节点。

大多数情况下，系统只有一个 proposer，他的提议也总是会很快地被大多数节点接受。Paxos 协议执行步骤如下：

1）批准（accept）：Proposer 发送 accept 消息要求所有其他节点（acceptor，接受者）接受某个提议值，acceptor 可以接受或者拒绝。

2）确认（acknowledge）：如果超过一半的 acceptor 接受，意味着提议值已经生效，proposer 发送 acknowledge 消息通知所有的 acceptor 提议生效。

当出现网络或者其他异常时，系统中可能存在多个 proposer，他们各自发起不同的提议。这里的提议可以是一个修改操作，也可以是提议自己成为主节点。如果 proposer 第一次发起的 accept 请求没有被 acceptor 中的多数派批准（例如与其他 proposer 的提议冲突），那么，需要完整地执行一轮 Paxos 协议。过程如下：

1）准备（prepare）：Proposer 首先选择一个提议序号 n 给其他的 acceptor 节点发送 prepare 消息。Acceptor 收到 prepare 消息后，如果提议的序号大于他已经回复的所有 prepare 消息，则 acceptor 将自己上次接受的提议回复给 proposer，并承诺不再回复小于 n 的提议。

2）批准（accept）：Proposer 收到了 acceptor 中的多数派对 prepare 的回复后，就进入批准阶段。如果在之前的 prepare 阶段 acceptor 回复了上次接受的提议，那么，proposer 选择其中序号最大的提议值发给 acceptor 批准；否则，proposer 生成一个新的提议值发给 acceptor 批准。Acceptor 在不违背他之前在 prepare 阶段的承诺的前提下，接受这个请求。

3）确认（acknowledge）：如果超过一半的 acceptor 接受，提议值生效。Proposer 发送 acknowledge 消息通知所有的 acceptor 提议生效。

Paxos 协议需要考虑两个问题：正确性，即只有一个提议值会生效；可终止性，即最后总会有一个提议值生效。Paxos 协议中要求每个生效的提议被 acceptor 中的多数派接受，并且每个 acceptor 不会接受两个不同的提议，因此可以保证正确性。Paxos 协议并不能够严格保证可终止性。但是，从 Paxos 协议的执行过程可以看出，只要超过一个 acceptor 接受了提议，proposer 很快就会发现，并重新提议其中序号最大的提议值。因此，随着协议不断运行，它会往“某个提议值被多数派接受并生效”这一最终目标靠拢。

3.7.3 Paxos 与 2PC

Paxos 协议和 2PC 协议在分布式系统中所起的作用并不相同。Paxos 协议用于保证同一个数据分片的多个副本之间的数据一致性。当这些副本分布到不同的数据中心时，这个需求尤其强烈。2PC 协议用于保证属于多个数据分片上的操作的原子性。这些数据分片可能分布在不同的服务器上，2PC 协议保证多台服务器上的操作要么全部成功，要么全部失败。

Paxos 协议有两种用法：一种用法是用它来实现全局的锁服务或者命名和配置服务，例如 Google Chubby 以及 Apache Zookeeper。另外一种用法是用它来将用户数据复制到多个数据中心，例如 Google Megastore 以及 Google Spanner。

2PC 协议最大的缺陷在于无法处理协调者宕机问题。如果协调者宕机，那么，2PC 协议中的每个参与者可能都不知道事务应该提交还是回滚，整个协议被阻塞，执行过程中申请的资源都无法释放。因此，常见的做法是将 2PC 和 Paxos 协议结合起来，通过 2PC 保证多个数据分片上的操作的原子性，通过 Paxos 协议实现同一个数据分片的多个副本之间的一致性。另外，通过 Paxos 协议解决 2PC 协议中协调者宕机问题。当 2PC 协议中的协调者出现故障时，通过 Paxos 协议选举出新的协调者继续提供服务。

3.8 跨机房部署

在分布式系统中，跨机房问题一直都是老大难问题。机房之间的网络延时较大，且不稳定。跨机房问题主要包含两个方面：数据同步以及服务切换。跨机房部署方案有三个：集群整体切换、单个集群跨机房、Paxos 选主副本。下面分别介绍。

1. 集群整体切换

集群整体切换是最为常见的方案。如图 3-10 所示，假设某系统部署在两个机房：

机房 1 和机房 2。两个机房保持独立，每个机房部署单独的总控节点，且每个总控节点各有一个备份节点。当总控节点出现故障时，能够自动将机房内的备份节点切换为总控节点继续提供服务。另外，两个机房部署了相同的副本数，例如数据分片 A 在机房 1 存储的副本为 A11 和 A12，在机房 2 存储的副本为 A21 和 A22。在某个时刻，机房 1 为主机房，机房 2 为备机房。

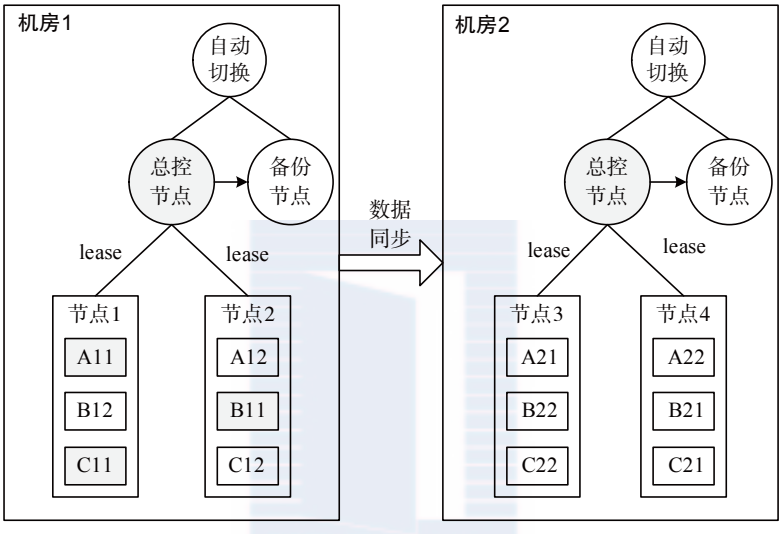

图 3-10　集群整体切换

机房之间的数据同步方式可能为强同步或者异步。如果采用异步模式，那么，备机房的数据总是落后于主机房。当主机房整体出现故障时，有两种选择：要么将服务切换到备机房，忍受数据丢失的风险；要么停止服务，直到主机房恢复为止。因此，如果数据同步为异步，那么，主备机房切换往往是手工的，允许用户根据业务的特点选择“丢失数据”或者“停止服务”。

如果采用强同步模式，那么，备机房的数据和主机房保持一致。当主机房出现故障时，除了手工切换，还可以采用自动切换的方式，即通过分布式锁服务检测主机房的服务，当主机房出现故障时，自动将备机房切换为主机房。

2. 单个集群跨机房

上一种方案的所有主副本只能同时存在于一个机房内，另二种方案是将单个集群部署到多个机房，允许不同数据分片的主副本位于不同的机房，如图 3-11 所示。每个数据分片在机房 1 和机房 2，总共包含 4 个副本，其中 A1、B1、C1 是主副本，A1 和 B1 在机房 1，C1 在机房 2。整个集群只有一个总控节点，它需要同机房 1 和机房 2 的所有

工作节点保持通信。当总控节点出现故障时，分布式锁服务将检测到，并将机房 2 的备份节点切换为总控节点。

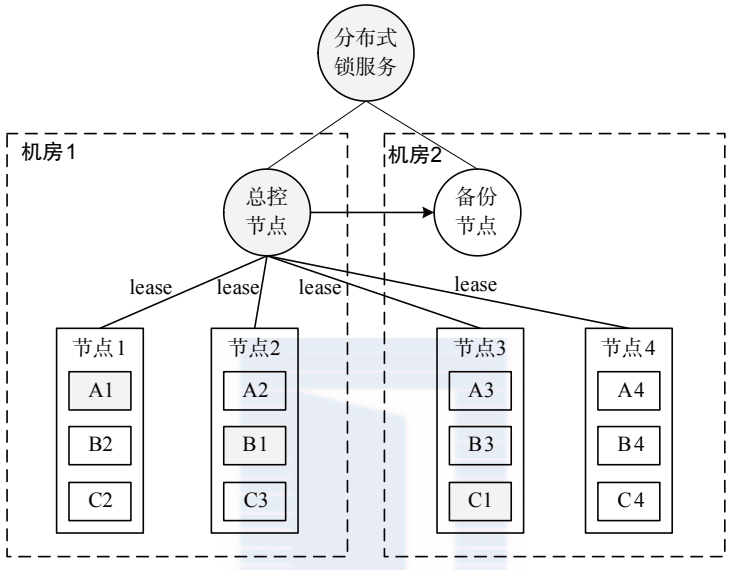

图 3-11 单个集群跨机房

如果采用这种部署方式，总控节点在执行数据分布时，需要考虑机房信息，也就是说，尽量将同一个数据分片的多个副本分布到多个机房，从而防止单个机房出现故障而影响正常服务。

3. Paxos 选主副本

在前两种方案中，总控节点需要和工作节点之间保持租约（lease），当工作节点出现故障时，自动将它上面服务的主副本切换到其他工作节点。

如果采用 Paxos 协议选主副本，那么，每个数据分片的多个副本构成一个 Paxos 复制组。如图 3-12 所示，B1、B2、B3、B4 构成一个复制组，某一时刻 B1 为复制组的主副本，当 B1 出现故障时，其他副本将尝试切换为主副本，Paxos 协议保证只有一个副本会成功。这样，总控节点与工作节点之间不再需要保持租约，总控节点出现故障也不会对工作节点产生影响。

Google 后续开发的系统，包括 Google Megastore 以及 Spanner，都采用了这种方式。它的优点在于能够降低对总控节点的依赖，缺点在于工程复杂度太高，很难在线下模拟所有的异常情况。

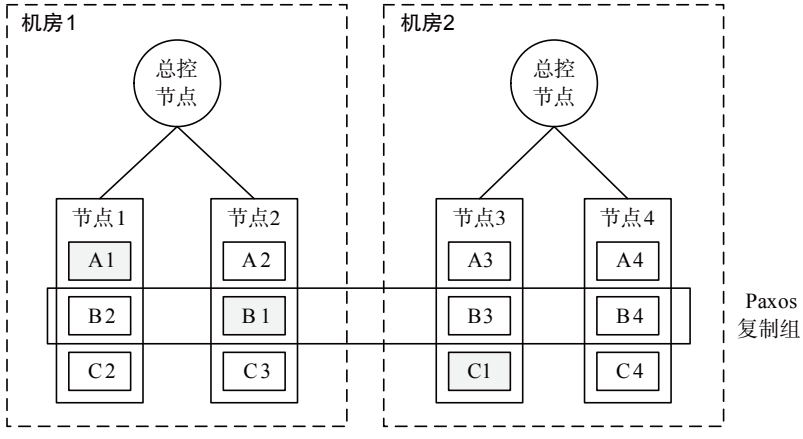

图 3-12　Paxos 选主副本

HZ BOOKS
华章图书