BIG DATA

# 3.分布式系统设计的CAP理论

——分布式数据库的由来和原理

# 更强的算力

- 阶段一：提高主频
  - 散热限制
- 阶段二：优化计算法
  - 精简指令集RISC和复杂指令集CISC
  - 指令流水
- 阶段三：提高核心数
  - 双核、四核
  - GPU

# 内存一致性

- 内存一致性模型描述的是程序在执行过程中内存操作正确性的问题。
  - 内存操作包括读操作和写操作，每一操作又可以用两个时间点界定：发出和响应）。
  - 在假定没有流水线的情况下（即单个处理器内指令的执行是按顺序执行的），设系统内共有 ***N*** 个处理器，每个处理器可发出 $S_n(0<n\leq N)$ 个内存操作（读或写），那么可能的执行顺序总共有：

$$\frac{(\sum_{n=1}^{n=N} s_n)!}{\prod_{n=1}^{n=N} s_n!}$$

  - 内存一致性模型描述的就是这些操作可能的执行顺序中那些是正确的。

# 典型的内存一致性模型

- 线性一致性（Linearizability）：
  - 或严格一致性（Strict consistency），任何对一个内存位置X的读操作，将返回最近一次对该内存位置的写操作所写入的值。
- 原子一致性（Atomic consistency）：
  - 读操作未能立即读到此前最近一次写操作的结果，但多读几次还是获得了正确结果。所有对数据的修改操作都是原子的，不会产生竞态冲突。
- 顺序一致性（Sequential consistency ）：
  - （多处理器上并发程序）任何一次执行结果都相同，就像所有处理器的操作按照某个顺序执行，各个微处理器的操作按照其程序指定的顺序进行。换句话说，所有的处理器以相同的顺序看到所有的修改。读操作未必能及时得到此前其他处理器对同一数据的写更新。但是各处理器读到的该数据的不同值的顺序是一致的。

# 更强的数据库?

- 3.1 分布式系统的伸缩性
- 3.2 横向扩展方案
- 3.3 CAP理论
- 3.4 BASE模型
- 3.5 Web分布式系统设计

**问题**

**哪些指标能用来衡量数据库的“更强”？**

BIGGER

FASTER

convenience

# 数据库事务可靠性基本要求：ACID

- 事务：
  - 由一系列数据库操作组成的一个完整的逻辑过程。
- Atomicity（原子性）：
  - 一个事务（transaction）中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。
- Consistency（一致性）：
  - 事务对数据库的作用使数据库从一个一致状态转换到另一个一致状态。在事务开始之前和事务结束以后，数据库的完整性没有被破坏。
- Isolation（隔离性）：
  - 多个事务并发执行时，应互不影响，其结果要和这些事务独立执行的结果一样（并发控制）。每个事务的更新在它被提交之前，对其他事务都是不可见的。
- Durability（持久性）：
  - 事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

# 实例

- 数据表test有两列A和B，在这两列中要求：
  - 约束A值和B值必须相加得100

- 创建该表的SQL语句如下：

```
CREATE TABLE test (
        A INTEGER,
        B INTEGER CHECK(A + B = 100)
        );
```

# 原子性失败

- 一个事务：*从A减10并且加10到B*
- 如果成功，它将有效，因为数据继续满足约束。
- 然而，假设在从A移走10后，这个事务不能去修改B。
  - 如果这个数据库保持A的新值，原子性和约束将都被违反。
  - 原子性要求这两部分事务都完成或两者都不完成。
- 如果这些操作有秩序的运行，则必须要进行隔离。

- 0级隔离性：一个事务不会覆盖更高级别的事务的脏读；
- 1级隔离性：事务不会发生更新丢失；
- 2级隔离性：事务不会发生更新丢失和脏读；
- 3级隔离性（真隔离性）：除具备2级隔离性性质外，还支持重复读
- 提问：要避免原子性失败，至少需要几级隔离性？
  - 2级

# 一致性失败

- 一致性要求数据符合所有的验证规则。

- 假设一个事务尝试从A减10而不改变B。
- 因为一致性在每个事务后被检查，众所周知在事务开始之前A+B=100。
- 如果这个事务从A转移10成功，满足原子性。
- 然而，一致性验证将得到A+B=90。

- 提问：请列举一些其它不满足一致性的操作
  - A或B加减小数
  - A或B加减>100的数

# 隔离故障

- 考虑这两个事务（只有写任务）：
  - T1从A转移10到B；T2从B转移10到A。
- 有四种行动:
  - A−10、B+10、B−10、A+10
- 在没有并发控制下，实际行动顺序可能是:
  - A−10、B−10、B+10、A+10
- 如果T1在一半的时候失败
  - A失败->B失败
- 假设两个事务在同一时间执行，每个都是尝试修改同一个数据。这两个中的一个必须为保证隔离等待直到另一个完成。

# 持久性失败

- 假设一个事务从A转移10到B。
- 该事务将A-10和B+10的修改信号发给磁盘，之后给用户发送了成功信号（web）。
- 然而，这些变化仍在磁盘缓冲区中排队等待被提交到磁盘。
- 因为不可抗力，磁盘缓冲区被清空了，但此时用户已经收到成功信号。

# 实现ACID的手段

- 预写式日志（Write ahead logging）：
  - log文件中通常包括redo和undo信息。
  - 假设一个程序在执行某些操作的过程中机器掉电了。在重新启动时，程序可能需要知道当时执行的操作是成功了还是部分成功或者是失败了。
  - 如果使用了WAL，程序就可以检查log文件，并对突然掉电时计划执行的操作内容跟实际上执行的操作内容进行比较。在这个比较的基础上，程序就可以决定是撤销已做的操作还是继续完成已做的操作，或者是保持原样。
  - HBase
- 影子分页（Shadow paging）：
  - 影子分页是一种写时复制技术，当修改一个页面时会分配一个影子页面。由于影子页面没有被别的地方引用，可以自由修改，不必顾虑一致性。
  - 修改完成后，需要被持久化时，所有引用原页面的地方都被修改为引用影子页面。

# CAP理论

在计算机科学中，CAP 理论又称为布鲁尔定理(Brewer's theorem)，是由柏克莱加州大学计算机科学家埃里布鲁尔(Eric Brewer)在 1998 年提出一个假说，并在 2000 年的分布式计算原则研讨会上发表。这个假说是 Brewer 及同事在横向可伸缩性分布系统设计方面的多年辛勤劳动的结晶。在 2002 年，麻省理工学院的赛斯·吉尔伯特(Seth Gilbert)和南希·林奇(Nancy Lynch)又完成了布鲁尔假说的证明，使之脱离了唯像学说而成为一个定理。但应说明的是吉尔伯特和林奇证明的布鲁尔定理比布鲁尔提出的假说更为狭义。目前，CAP 理论已成为分布式系统设计与构建的重要理论基石。

# 3.1 分布式系统的伸缩性

(1)横向扩展(Horizontal Scaling)是指向逻辑单元之外的扩展，增加更多逻辑单元的资源，并使它们像是一个单元一样工作。大多数集群方案、分布式文件系统、负载平衡等都可以提高横向的可伸缩性。

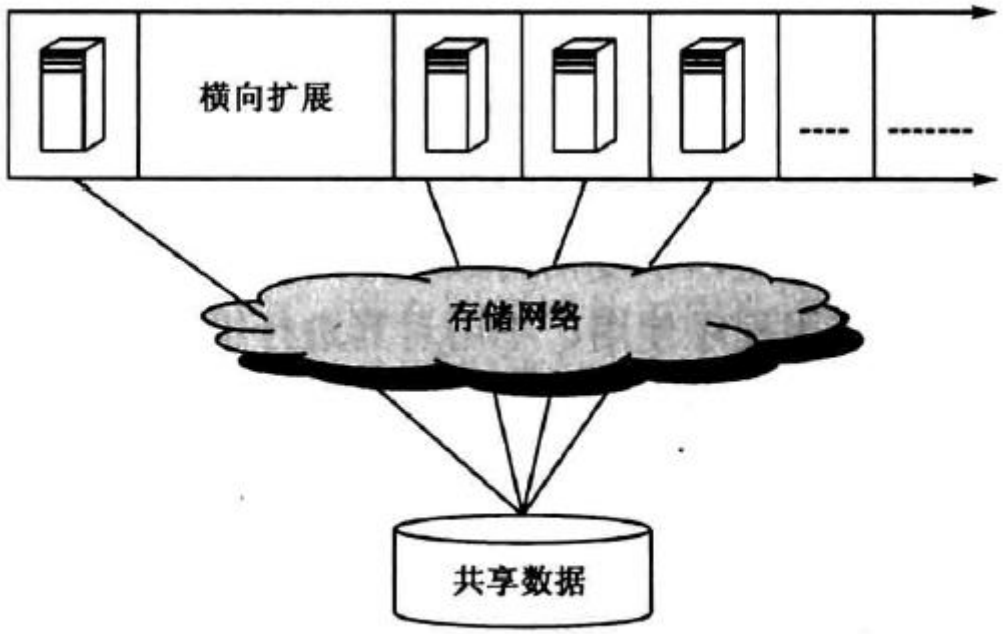
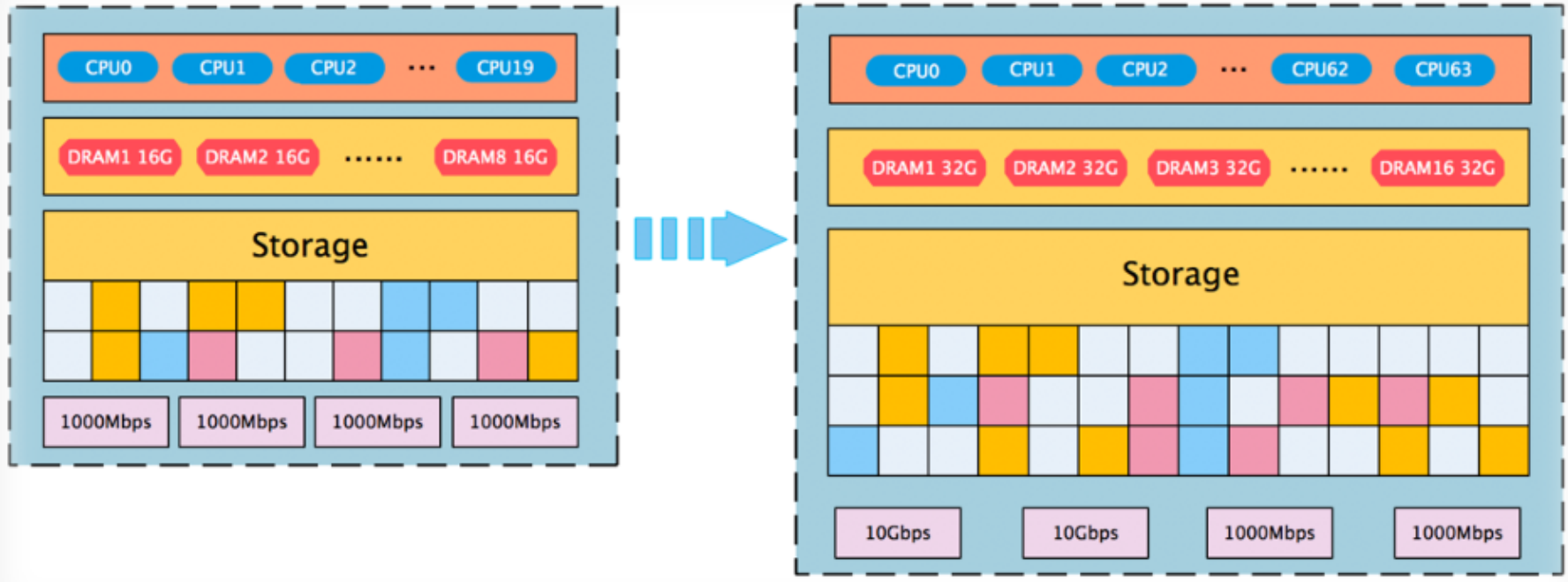



图 3-2　横向扩展示意图

# 3.1 分布式系统的伸缩性

(2)纵向扩展(Vertical Scaling)是指在同一个逻辑单元内增加资源来提高处理能力。例如在现有服务器上增加 CPU 计算能力，或者在现有的 RAID/SAN 存储中增加硬盘来提高存储能力。



# 3.1 分布式系统的伸缩性

除了横向扩展与纵向扩展概念之外，还有下述几种较常用的扩展概念。

(1)线性扩展性：在扩大规模的时候，扩展因子保持为常数。

(2)次线性扩展性：小于 1.0 的扩展性因子。

(3)超线性扩展性：因为增加更多组件而获得更佳的性能，在 RAID 系统中跨多个磁盘的 I/O，当磁盘越多，系统性能越好。

(4)负扩展性：当规模扩大的时候，系统性能变坏。

# 3.1.2 影响横向扩展的主要因素

问题

哪些因素会影响数据库横向扩展？

**数据规模**

**数据应用方式**

**数据应用效率**

**数据内外部结构**

# 1、引用数据

某些数据更新非常频繁。例如，Web 服务器的日志数据，以及某个车间的计算机的仪表读数等。但是，季度销售额这样的历史数据却更新不频繁，甚至不进行更新。引用数据是仅提供给应用程序使用、不用对其进行维护的数据。引用数据的主要特点是不但相对稳定，而且在设定时间内有效。例如，定单输入系统使用的产品目录、航班时刻表，以及金融系统中使用的账目表，这种数据相对稳定，由于它可以提供其他应用程序使用，因此频繁的更改会导致混乱。如果一个价格列表中的价格在一天内更改多次，则客户会感觉很混乱而且不满意。引用数据通常以固定的时间间隔进行更改。例如，价格可按天或按周更改，而账号可只是按月更改。引用数据还有一个版本标签，该标签包含在引用该数据的事务中。例如，采购订单可能会引用，用于创建订单的目录版本，以便在定价时不引起混乱。业务可能选择接受若干个引用数据的版本，以避免客户使用过期的引用数据。

# 2、活动数据

活动数据是与特定活动或业务事务相关联的数据。例如，采购订单或股票下跌将生成一些与该事务相关联的数据。该数据仅在特定业务活动范围内相关，除了一些历史原因外，当该活动完成之后，这些数据并不是非常有用，活动数据就转变为引用数据。

活动数据的更新率也很低。例如，在创建一个采购订单后，仅当发生状态更改或发货日期更改时，它才进行更改，但这些事件的发生相对不太频繁，每天仅有几个更新。一般而言，活动数据易于识别，并且通常不在活动范围之外访问。这表明，如果活动数据因横向扩展而在若干个数据库间拆分，将很容易查找。需要访问特定采购订单的业务事务通常需要知道编号，因此，如果采购订单以编号范围分区，则很容易找到数据库以访问所需的采购订单。活动数据通常由另一个数据对象限定范围。例如，将特定客户的所有订单存储在与该客户的客户信息所在的同一数据库中，这是最常见的订单存储方式。同样地，也可将特定供应商的所有采购订单存储在该供应商所在的供应商数据库中。这表明，在必要时，复制活动数据相对比较容易，而且在很多情况下，在需要横向扩展时，可以将活动数据分散到若干个数据库中。采用哪个方法来横向扩展活动数据主要取决于数据使用率因素。

# 3、资源数据

资源数据是与业务相关的核心数据，如库存清单、账户数据、客户档案等都属于资源数据。如果资源数据丢失，基本上无法进行工作。因此，资源数据库通常使用大量的数据集成和高可用性功能来确保资源数据始终可用。资源数据通常具有非常高的并发要求，因为很多应用程序和用户都需要访问同一数据，需要较高的更新率。库存清单项的可用数量或账户余额在一天内可能发生多次更改。在满足数据集成和高可用性需要方面，纵向扩展对资源数据则更适用。

资源数据通常只是当前活动数据。非活动账户、废止的部分等通常在一个相对静态的历史表中维护，变成了引用数据。资源数据的快照可能由于历史或报告原因才会用到，因此它们也是引用数据。当资源数据变成引用数据后，对数据集成和高可用性的要求就不重要。从资源数据到引用数据的转换保留了资源数据的相关性，同时也减少了资源数据的大小和增加了横向扩展的需要。

# 4、数据分区

横向扩展数据最有效的方法之一是将数据分散到多个数据库中，以便每个数据库服务器可以处理该数据的一部分。虽然这是一个横向扩展数据很直接的方式，但并不是所有数据都可以有效地进行分区，而且即使它可以分区，其分区方式也将对性能产生较大的影响。为了说明分区的影响，来考虑一个订单数据库进行分区几种方案。

(1)根据定购的内容对订单进行分区，图书订单存储在一个数据库中，服装订单存储在另一个数据库中等。

(2)通过订单号范围来拆分订单，即通过订单号来访问订单。但如果单号与客户表有大量链接，则该方案也需要分布式链接。解决该链接问题的方法是，按客户号对订单数据库进行分区，以便对于给定客户而言，要使用的订单数据库始终是已知的。如果客户数据库进行了分区，这将特别有效，而且每个客户的订单与客户位于同一个数据库中。其他数据可能必须链接到订单数据，如果可能，该数据应该对相同的方案进行分区以避免分布式链接。该数据的一部分可以是引用数据(例如，项描述)，而且可以复制到所有订单数据库中，以消除到清单数据库的分布式链接。如果应用程序数据可以划分到多个数据库中，并且多个服务器提供的额外处理功能优于汇集结果的通信成本，则可以对该数据进行分区。并不是所有的应用程序数据都可以有效地进行分区，而选择正确的分区方案对于分区数据的有效横向扩展而言是必要的。

# 5、数据相互依赖与耦合

1)分布数据的依赖

如果数据库的各个部分由不同的应用程序使用，则以应用程序边界对数据库进行拆分，这样每个应用程序都可处理自己专用的数据。

如果不同应用程序使用的数据可以进行分段以便提供专用的数据库处理，而不会导致由于数据相互依赖而引起的过多网络通信量所带来的影响。

2)分布数据的耦合

对于数据耦合需要考虑的最后一个因素是共享表的处理方式。将有一定数量的表由多个应用程序访问；因此，当拆分数据时，必须确定在何处放置共享表。在某些情况下，一个表可能由多个应用程序读取但只由一个应用程序更新，因此使用更新应用程序进行定位需要是合理选择。如果该表相对较小而且广泛用于多个应用程序，则将其复制到多个数据库很有意义。如果该表由单个应用程序更新，这是最容易的，这样来自主控副本的事务复制可用于使其他副本保持最新状态。

3)高耦合资源数据

资源数据由于具有大量完整性约束，所以是高度相互依赖，这意味着高度耦合。在某些情况下，不能按应用程序拆分资源数据。解决这个问题的方法是：

(1)对应用程序进行某些更改，可以横向扩展引用数据和活动数据；

(2)对资源数据进行分区，以便在资源数据高度耦合时仍然可以横向扩展，但是要求拆分数据的一些横向扩展选项可能要求应用程序重构，以适应数据重构。

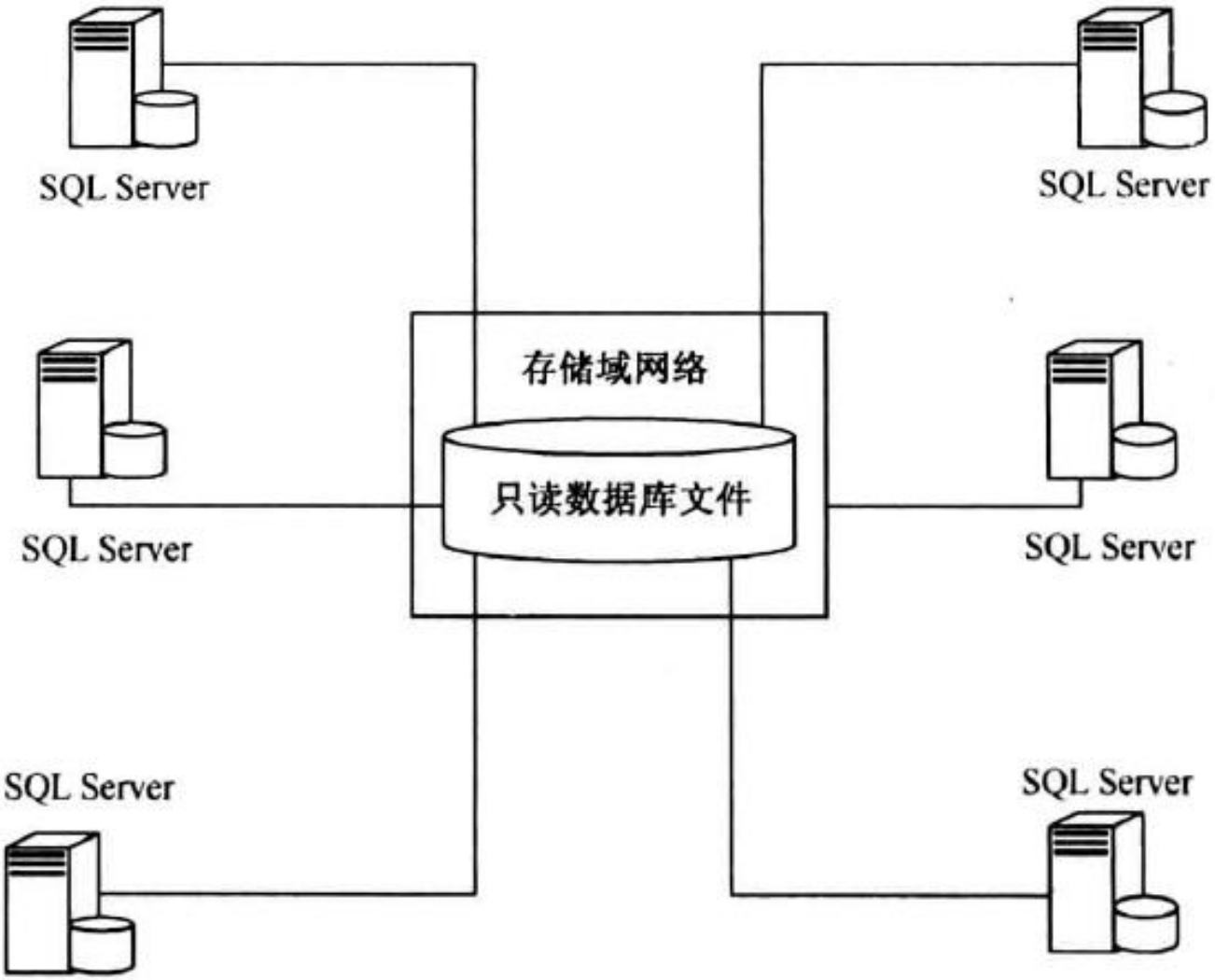
# 6、更改应用程序的能力

设计应用程序时，需要考虑横向扩展策略，也就是说，横向扩展策略不需要更改应用程序。在早些的应用程序，横向扩展时，可能需要对查询和存储过程进行少量的更改，或者可能需要重新考虑应用程序的工作方式。显然，如果使用的应用程序，对横向扩展应具有最大的灵活性。因此，当设计一个新应用程序时，应该考虑横向扩展，因为如果当应用程序在生产时耗尽资源后再进行更改，比一开始就设计横向扩展要难得多。任何横向扩展策略对于应用程序代码必须透明。

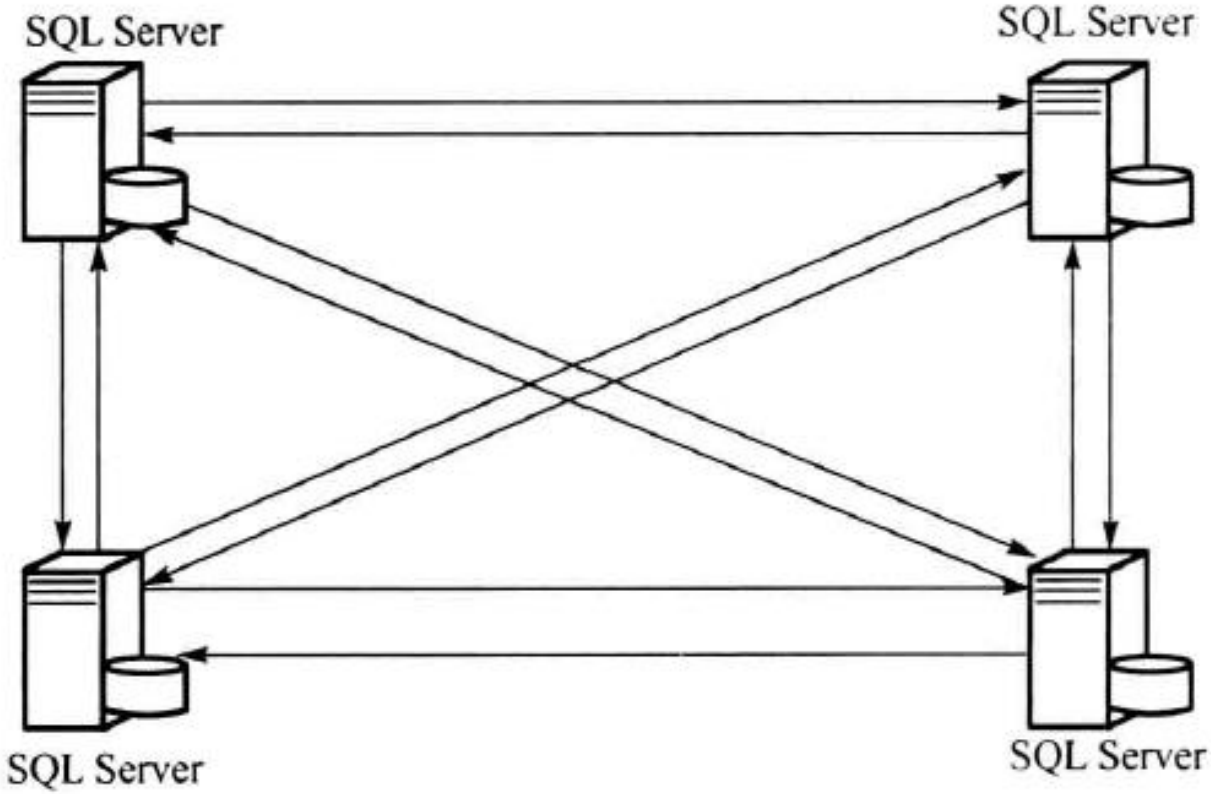
# 3.2 横向扩展方案

- 3.2.1 可伸缩共享数据库
- 3.2.2 对等复制的横向扩展方案
- 3.2.3 链接服务器和分布式查询
- 3.2.4 分布式区视图
- 3.2.5 数据依赖型路由的横向扩展

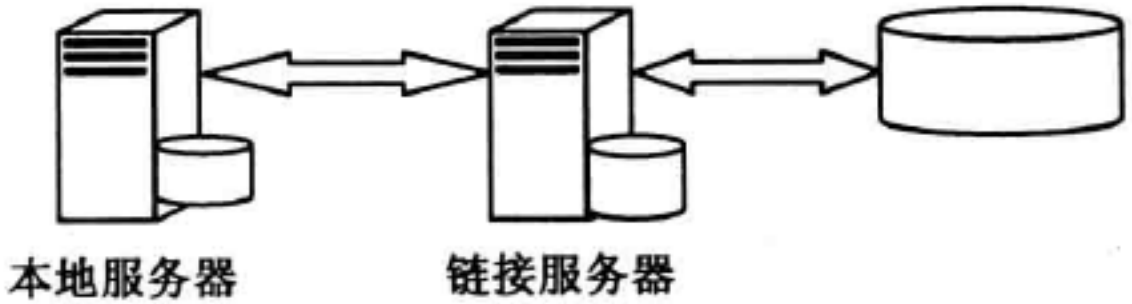
# 3.2.1 可伸缩共享数据库



# 3.2.2 对等复制的横向扩展方案



# 3.2.3 链接服务器和分布式查询



# 3.2.4 分布式区视图

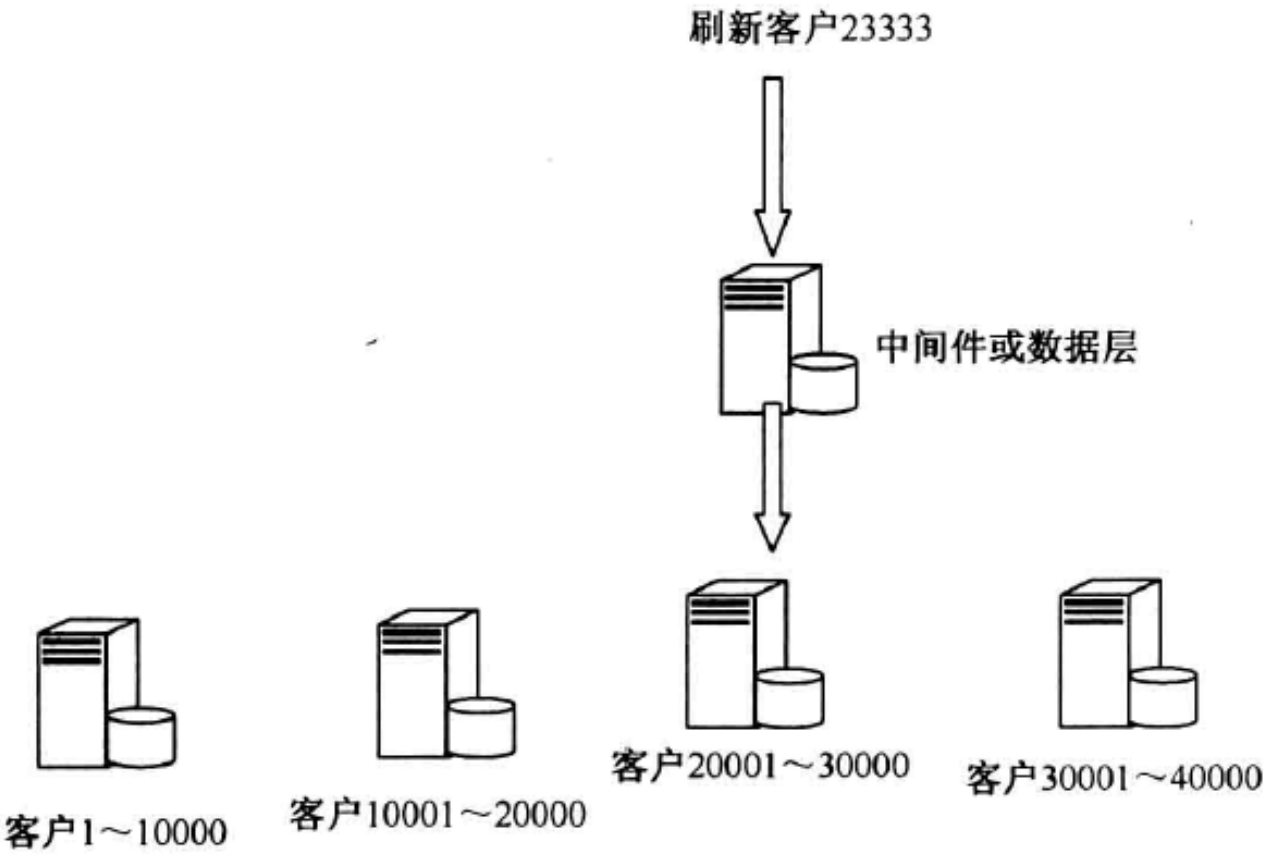



图 3-6　数据依赖型路由

# 影响横向扩展解决方案选择的因素

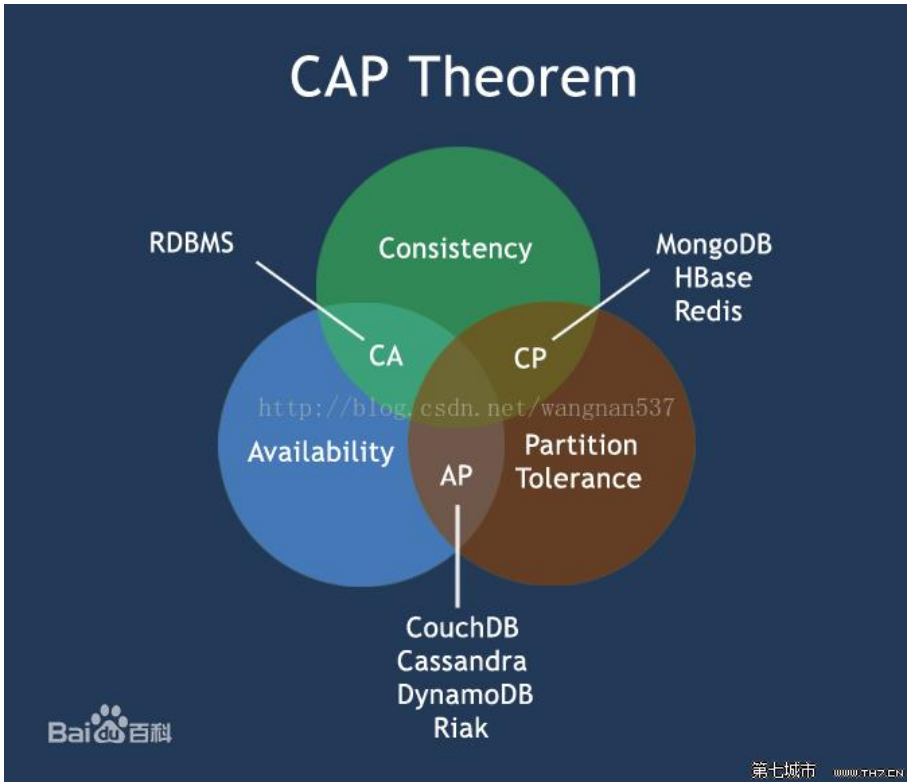
| | 更新频率 | 更改应用程序的能力 | 数据可分区性 | 数据耦合 |
|---|---|---|---|---|
| 可伸缩共享数据库 | 只读 | 需要少量更改或无需更改 | 无要求 | 无要求 |
| 对等复制 | 通常为读取，无冲突 | 需要少量更改或无需更改 | 无要求 | 无要求 |
| 链接服务器 | 最小化跨数据库更新 | 最少更改 | 通常不需要 | 具有低耦合性非常重要 |
| 分布式分区视图 | 可以频繁更新 | 可能需要一些更改 | 非常重要 | 影响极小 |
| 数据依赖型路由 | 可以频繁更新 | 可能会进行重大更改 | 非常重要 | 低耦合可能有助于某些应用程序 |
| 面向服务的数据体系结构 | 可以频繁更新 | 需要大量更改 | 通常不需要，除非与 DDR 合并 | 要求服务之间具备低耦合性 |

# 3.3 CAP理论

- 3.3.1 分布系统设计的核心系统需求
- 3.3.2 CAP 定理

# 3.3.1 分布系统设计的核心系统需求

- C: Consistency
  - 数据一致更新，所有数据变动都是同步的
- A: Availability
  - 好的响应性能，稳定性（可用性）
- P: Tolerance of network Partition
  - 分区容忍性(规模)



# 3.3.2 CAP 定理

- **定理：任何分布式系统只可同时满足二点，没法三者兼顾。**

图中的 $N_1$，$N_2$ 为网络中的两个节点。它们共享同一数据 $V$，其值为 $V_0$。$N_1$ 上有一个算法 A，可以认为 A 中无 bug，并且是可预测和可靠的。$N_2$ 上有一个类似的算法 B。在这个例子中，利用 A 算法将新值写入 $V$，而利用 B 算法读取 $V$ 的值。

正常情况下的过程如下：

(1) A 写入新的 $V$ 值，称作 $V_1$。

(2) $N_1$ 发送信息给 $N_2$，更新 $V$ 的值。

(3) 现在 B 读取的 $V$ 值将会是 $V_1$ (图 3-8)。

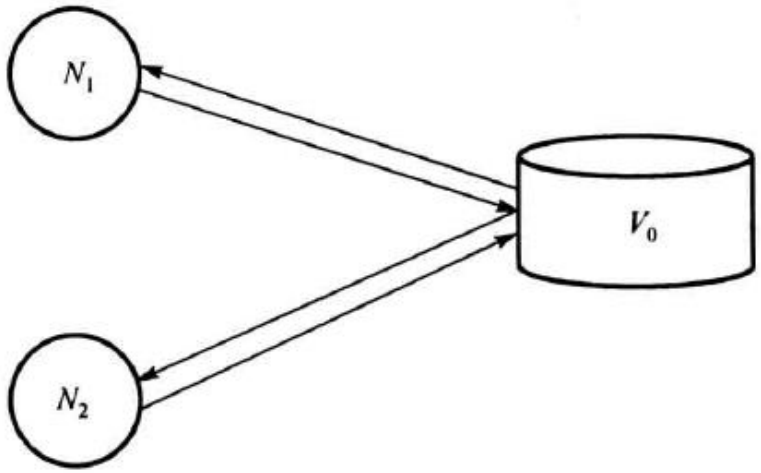



图 3-7　网络中的两个节点

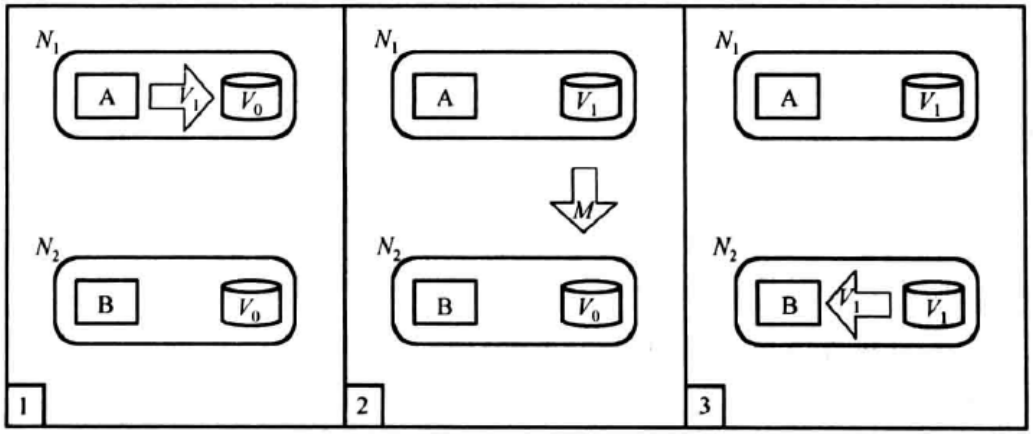
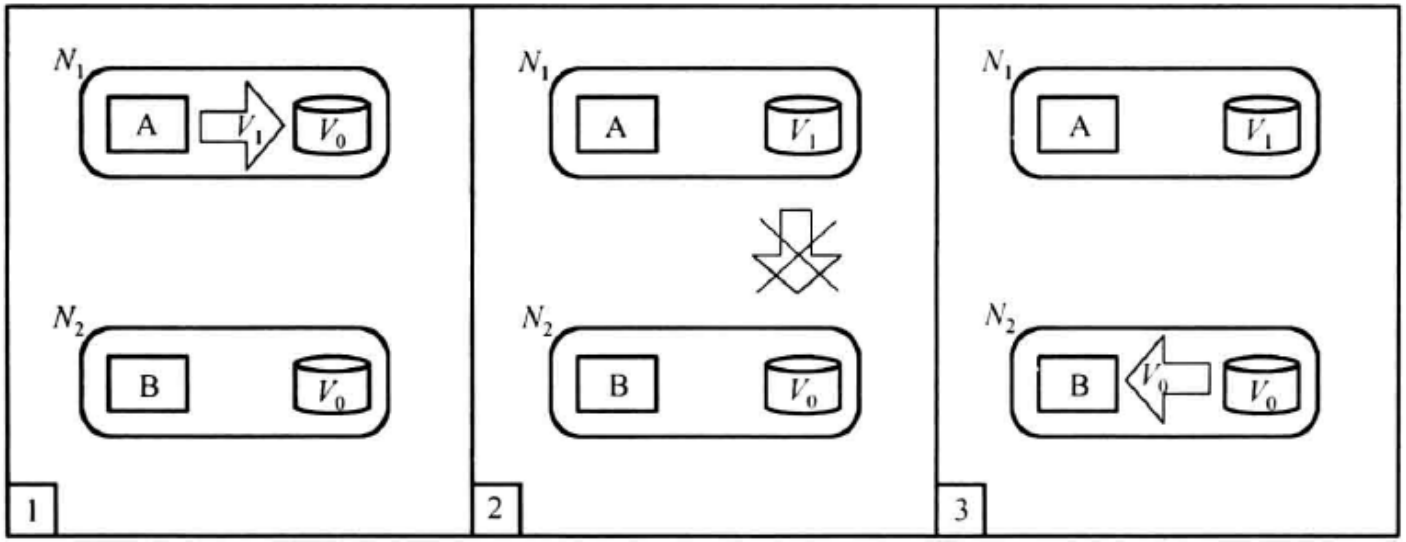



图 3-8　正常情况下的过程

# 分区情况下

如果网络断开(分区)表明从 $N_1$ 无法发送信息到 $N_2$，那么在第 3 步的时候，$N_2$ 就会具有一个与 $N_1$ 不一致的 $V$ 值，其中在 $N_1$ 中的 $V=V_1$，在 $N_2$ 中的 $V=V_0$。

如果规模达到几百个事务，这将成为一个大问题。如果 M 是一个异步消息，那么 $N_1$ 无法知道 $N_2$ 是否收到了消息。即使 M 是保证能发送的(保证传送)，$N_1$ 也无法知道是否消息由于分区事件的发生而延迟，或 $N_2$ 上的其他故障而延迟。如果 M 是一个同步消息，那将使得 $N_1$ 上 A 的写操作和 $N_1$ 到 $N_2$ 的更新事件成为一个原子操作，而这将导致等待问题。已经说明使用其他的变种方式，即使是部分同步模型也无法保证原子性。因此，CAP 定理表明如果使 A 和 B 高可用(以最小的延迟提供服务)并且使所有的 $N_1$ 到 $N_n$($n$ 的值可以是数百甚至是上千)的节点能够网络分区，那么有时就可能出现某些节点 $V$ 的值是 $V_0$，而其他节点 $V$ 的值是 $V_1$。也就是说，选择了可用性和分区容错性，牺牲了一致性(图 3-9)。



# 放弃分区容错性

- Postgres,MySQL, etc（relational）数据库；
- Vertica（column-oriented）数据库；
- Aster Data（relational）数据库；
- Greenplum（relational）数据库。

# 放弃可用性

- BigTable（column-oriented/tabular）；
- Hypertable（column-oriented/tabular）；
- HBase（column-oriented/tabular）；
- MongoDB（document-oriented）；
- Terrastore（document-oriented）；
- Redis（key-value）；
- Scalaris（key-value）；
- MemcacheDB（key-value）；
- Berkeley DB（key-value）。

# 放弃一致性

- Dynamo（key-value）；
- Voldemort（key-value）；
- Tokyo Cabinet（key-value）；
- KAI（key-value）；
- Cassandra（column-oriented/tabular）；
- CouchDB（document-oriented）；
- SimpleDB（document-oriented）；
- Riak（document-oriented）。

# 3.4 BASE模型

- 3.4.1 三个核心需求分析
- 3.4.2 ACID、BASE 与CAP 的关系
- 3.4.3 CAP 与延迟
- 3.4.4 CAP 理论的进一步研究

最终一致(Basically Available, Soft-state,Eventually consistent，BASE)是 ACID 的反面，但任何架构都不完全基于 BASE 或完全基于 ACID。BASE 理论是 ACID 理论与实际相结合的产物。BASE 英文中有碱的意思，这个正好和 ACID 的酸的意义相对。BASE 恰好和 ACID 是相对的，BASE 要求牺牲高一致性，获得可用性或可靠性。

# 3.4.1 三个核心需求分析

随着互联网应用的飞速发展，数据量与日俱增，传统的 ACID 数据库已经不能满足如此大的数据存储了。这个时候需要设计出好的分布式数据存储方式。而这些分布式数据存储方式受到 CAP 理论的约束，不可能同时达到高一致性、高可用性、高分区容错性的完美设计。所以在设计的时候要适当取舍，重点关注对应用需求来说比较重要的，而放弃不次要的，在 CAP 的三个核心系统需求之间进行取舍，设计出实际应用的存储方案。目前众多的分布式数据系统通过降低一致性来换取可用性。下面是一个简单的例子，可以从事务的角度分析如下，参考图 3-10 所示。

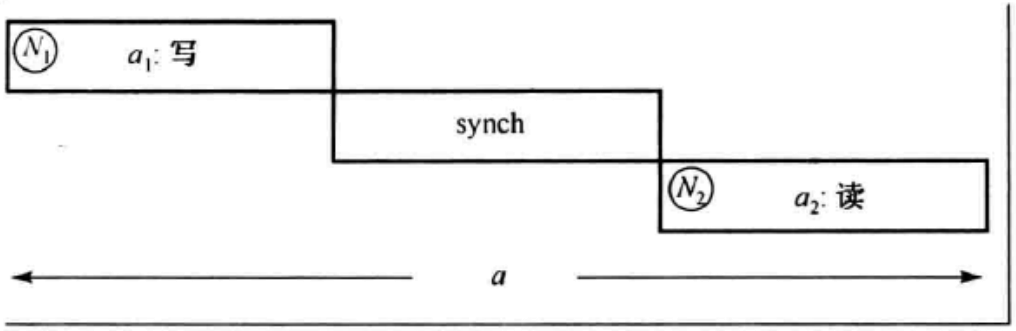



图 3-10　基于事务的分析

两个节点数据冗余，第一个节点先有一个写操作，第二个节点后有一个读操作。$a$ 是整个过程，要具有一致性，则需要等待 $a_1$ 进行写，然后同步到 $a_2$，然后 $a_2$ 再进行写，只有整个事务完成以后，$a_2$ 才能够进行读。致使得整个系统的可用性下降。$a_2$ 一直阻塞在那里等待 $a_1$ 同步到 $a_2$。这个时候如果对一致性要求不高，$a_2$ 可以不等待 $a_1$ 数据对于 $a_2$ 的写同步，直接读取，这样虽然此时的读写不具有一致性，但是在后面可以通过异步的方式使得 $a_1$ 和 $a_2$ 的数据最终一致，达到最终一致性。

# 3.4.2 ACID、BASE 与CAP 的关系

ACID 和 BASE 代表了两种截然相反的设计思想， ACID 注重一致性，是数据库的传统设计思路。BASE 在 20 世纪 90 年代后期提出，抓住了当时正逐渐成形的高可用性的设计思路，并且把不同性质之间的取舍摆上台面。大规模跨区域分布的系统，包括云在内，同时运用了这两种思路。出现较晚的 BASEE 是基本可用、软状态、最终一致性的英文缩写。其中的软状态和最终一致性这两种技巧善于分区的场合，并提高了可用性。

CAP 与 ACID 的关系复杂，也更易引起误解。其中一个原因是 ACID 的 C 和 A 字母所代表的概念不同于 CAP 的 C 和 A。还有一个原因是选择可用性只部分地影响 ACID 约束。

- 原子性A：复合操作单一化
- 一致性C：事务不能破坏数据库规则
- 隔离性I：事务
- 持久性D：

# 3.4.3 CAP 与延迟

虽然 CAP 理论的经典解释忽略网络延迟，但在实际中延迟和分区密切相关。CAP 理论实际应用的场景发生在操作的间歇，系统需要在这段时间内作出关于分区的选择。

(1) 如取消操作，则降低系统的可用性；

(2) 如继续操作，则可能损失系统一致性。

依靠多次尝试通信的方法来达到一致性，比如，两阶段事务提交，仅仅是推迟了决策的时间，系统终究要作出一个决定，无限期地尝试下去，本身就是选择一致性而牺牲可用性的表现。因此以实际效果而言，分区相当于对通信的时限要求。如果系统不能在时限内达到数据一致性，就表明发生了分区的情况，必须就当前操作在 C 和 A 之间作出选择。从这个实用的观察角度出发可以导出下述的结论。

(1) 分区并不是全体节点的一致见解，因为有些节点检测到了分区，有些可能没有。

(2) 检测到分区的节点，即进入分区模式，这是优化 C 和 A 的核心环节。

# 3.4.4 CAP理论的进一步研究

- 强一致性
  - 读写均一致
- 弱一致性
  - 保证写，不保证读一致
- 最终一致性
  - 保证写，仅保证总有一个时刻可读（弱一致性的特例）

| 对比项 | 强一致性 | 最终一致性 | 弱一致性 |
| --- | --- | --- | --- |
| 场景定义 | 假设三个进程 A、B、C 相互独立，且都对存储系统进行读写操作 | | |
| 数据一致性表现 | A 写入数据到存储系统之后，存储系统能够保证后续任何时刻发起读操作的 B、C 可以读到 A 写入的数据 | A 写入数据到存储系统之后，经过一定的时间，或者在某个特定的操作之后，BC 最终会读到 A 写入的数据 | A 写入数据到存储系统之后，存储系统不能够保证后续任何时刻发起读操作的 B、C 可以读到 A 写入的数据 |
| 示例 | OLTP 需要强一致性 | OLAP 需要最终一致性 | 绝大多数应用不能够容忍弱一致性 |

# 其它一致性

- 因果一致性
  - 在最终一致性的基础上加上“通知”事件
- Read-your-writes一致性
  - 即顺序一致性，在最终一致性的基础上保证顺序性事务的强一致
- 会话一致性
  - 在会话阶段保证Read-your-writes一致
- 单调读一致
  - 读取操作的顺序一致性
- 单调写一致
  - 写入操作的顺序一致性

# 3.5 Web分布式系统设计

设计大型 Web 系统时，需要注意的一些核心原则如下。

(1) 可用性；

(2) 性能；

(3) 可靠性；

(4) 可扩展；

(5) 易管理；

(6) 成本。

上面的这些原则为设计分布式 Web 架构提供了理论基础指导。但是，这些规则之间也可能彼此相矛盾，例如，为了扩充存储容量，可以通过添加更多的服务器(可伸缩性)，这个方案是易管理，但成本高。因此，构建所有大型 Web 应用程序都要考虑服务、冗余、分区和故障处理能力，并对的每个因素都会涉及选择和折中。

一些大型网站需要管理和传送大量的图片，需要构建一个具有低成本、高可用性和低延时(快速检索)的架构。

# 3.5.1 系统核心需求

- 图片系统主要考虑的因素：

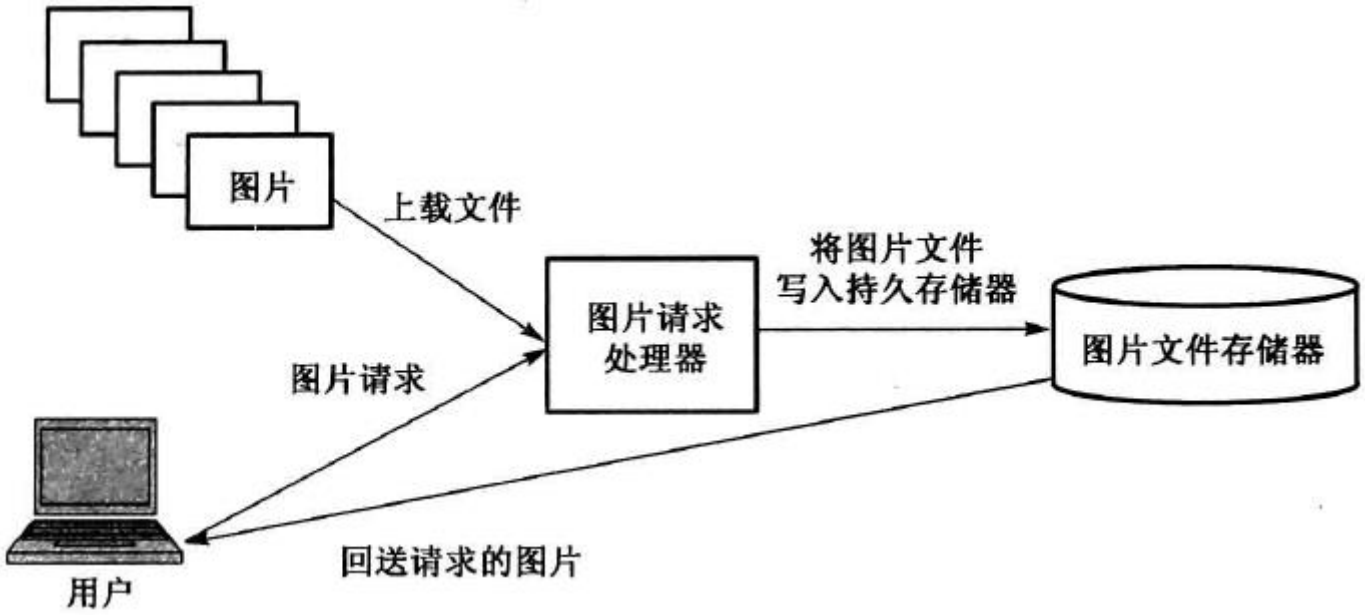
(1) 存储图片的数量无限制，所以存储系统应具备可伸缩性；

(2) 下载/请求需要做到低延迟；

(3) 用户上传一张图片，那么图片就应该始终存储在那里（图片数据的可靠性）

(4) 系统应该易于维护与易管理；

(5) 由于图片托管没有太高的利润空间，所以系统需要较高的效益。



# 3.5.2 系统服务

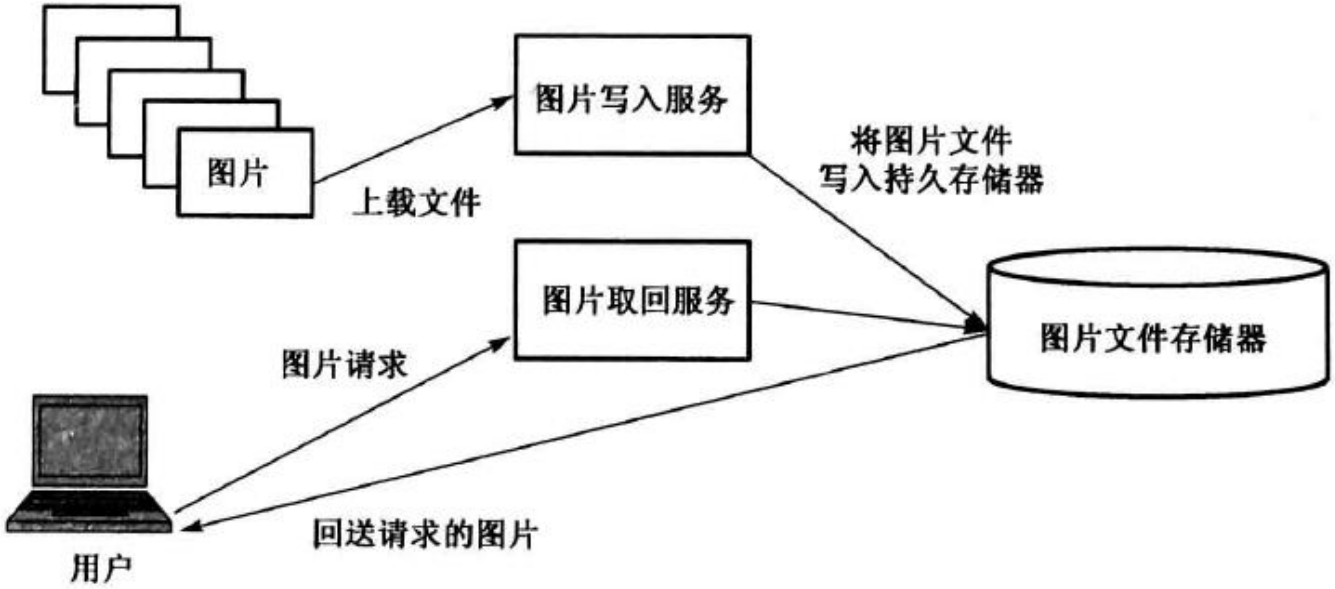
- 构建可伸缩分布式系统时，所有上传和检索都在一台服务器，会造成问题：

(1) 写图片和读图片两个功能共享同一台服务器资源。一般来说，下载速度是上传速度的3倍，通常文件可以从缓存中读取，而最终写入到磁盘中(也许在最终一致的情况下，可以被多写几次)。即使是从缓存或者磁盘中读取，数据写入都比读慢。这种情况导致了对共享同一台服务器资源存取速度的不匹配。

(2) Web 服务器通常都有一个并发连接数上限值，读可以异步或利用其他性能优化方法，Web 服务可以快速切换读取客户端的更多的请求，超过每秒的最大连接数(Apache 的最大连接数设为 500)。此外，写通常倾向于保持一个开放的链接进行持续上传，上传一个 1MB 的文件花费的时间可能会超过 1 秒，所以，服务器只能同时满足 500 个写请求。

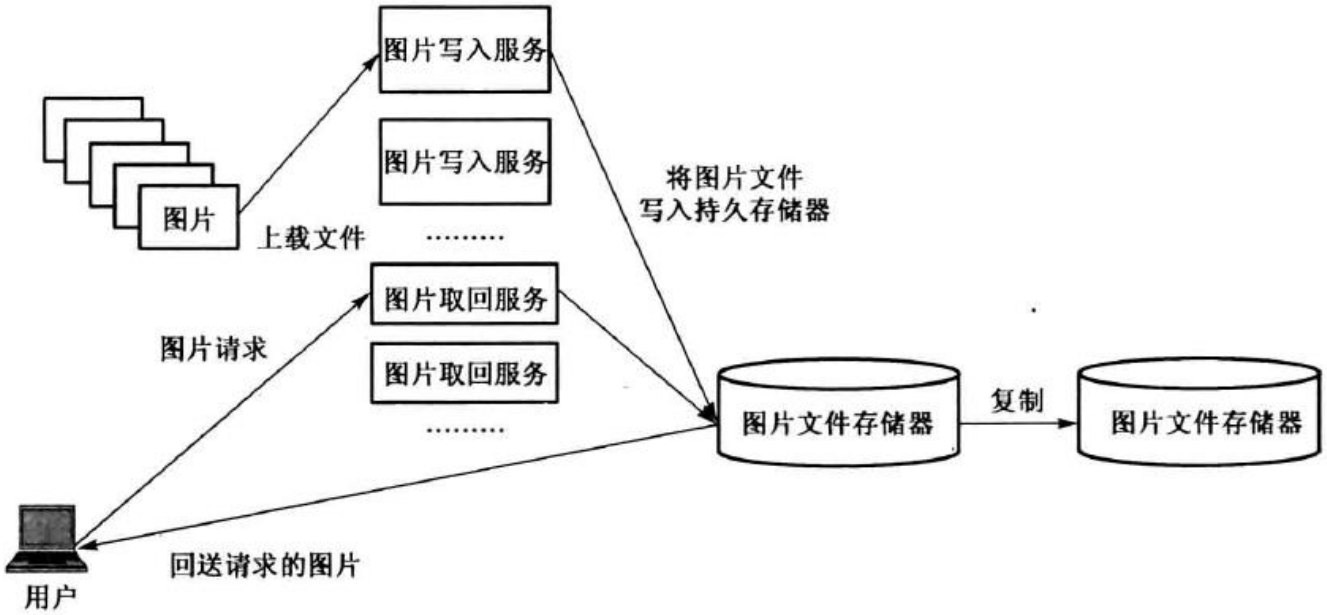
# 读写分离

克服这种瓶颈的一种常用方法是将读过程和写过程分离，如图 3-12 所示。读过程和写过程分离之后就可以对它们单独进行扩展。由于读操作比写操作多，更易于排除故障和解决规模方面问题，进而能够彼此独立解决问题。不必同时考虑写入和检索操作。



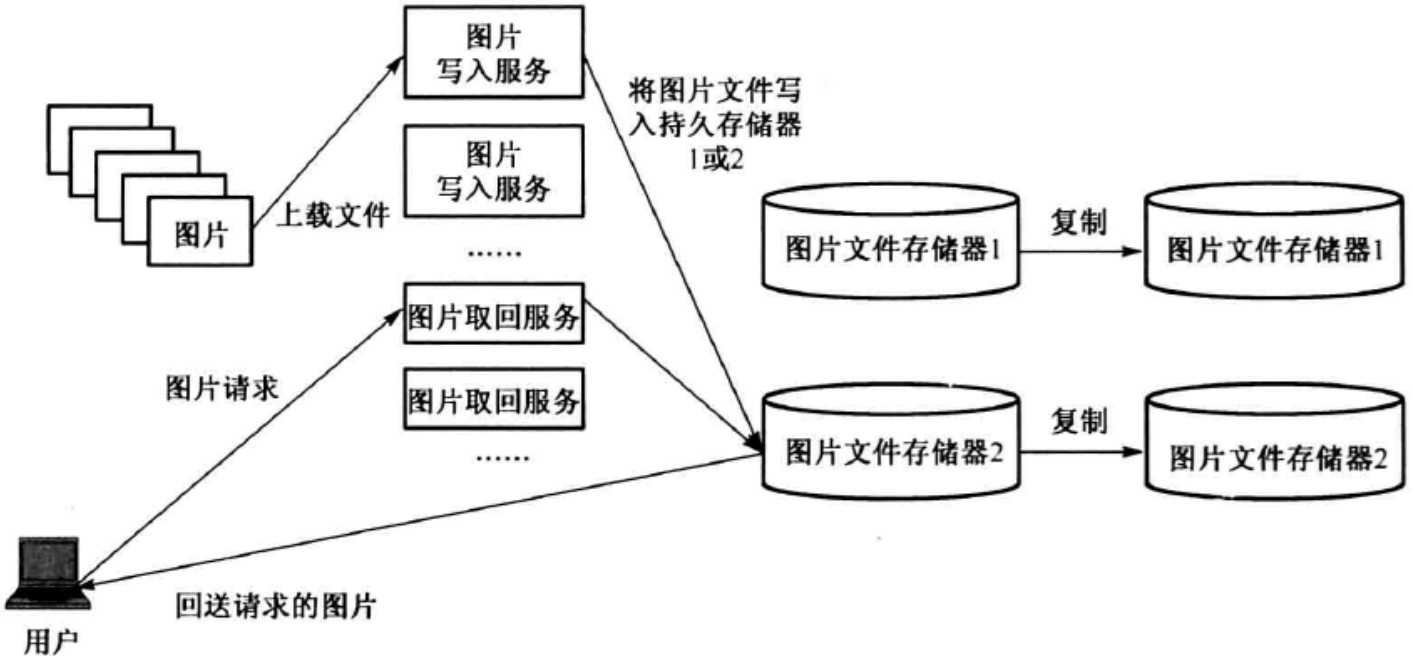
# 3.5.3 冗余

为了正确处理错误，一个 Web 架构的服务和数据必须具备适当的冗余。例如，如果只有一个副本文件存储在这台单独的服务器上，那么这台服务器出现问题或丢失，那么该文件也随即一起丢失。避免数据丢失的常用方法就是多创建几个文件或副本。这样的方法也同样也适用于服务器。如果一个应用程序的核心功能是应确保有多个副本或版本在同时运行，这样可以避免单节点失败。在系统中创建冗余后，当系统发生危机时，如果需要，可以消除单点故障并提供备份或备用功能。例如，这里有两个相同的服务示例在生产环境中运行，如果其中一个发生故障或者降低，那么该系统容错转移至那个健康的副本上。容错转移可以自动发生也可以手动干预。



# 3.5.4 分区

当横向扩展时，最常见的做法是把服务进行分区或碎片。分区可以被派发，这样每个逻辑组的功能就是独立的，也可以通过地理界限或其他标准，如非付费与付费用户来完成分区。这些方案的优点是他们会随着容量的增加提供一个服务或数据存储。在图片服务器案例中，用来存储图片的单个文件服务器可能被多个文件服务器取代，每个里面都会包含一套自己独特的图像，如图 3-14 所示。这种架构将允许系统来填充每一个文件/图片服务器，当磁盘填满时会添加额外的服务器。这样的设计需要一个命名方案，用来捆绑图片文件名到其相应的服务器上。图像名字可以形成一个一致的哈希方案并映射到整个服务器上；或者给每张图片分配一个增量 ID，当客户端对图片发出请求时，图片检索服务只需要检索映射到每个服务器上(例如，索引)的 ID。



# 参考资料


- 陈明，大数据概论，科学出版社，2015
- Abraham Silberschatz，数据库系统概念，机械工业出版社，2012
- 周一豪，基于弱内存一致性的多线程确定性执行技术研究，哈尔滨工业大学，2018
- 李彬，姜建国，少数决：更安全的分布式一致性算法选举机制，计算机科学与探索，2020