**至简大道：**

**编程范式：**

**资料1.函数式编程和声明式语言**

<https://wenku.baidu.com/view/83c863cfe309581b6bd97f19227916888486b9e5.html>

编程语⾔可以分成两类：

命令式

声明式

**资料2.java编程范式**

<https://blog.csdn.net/zhan107876/article/details/123736469>

java理论知识之命令式编程，声明式编程和函数式编程的区别

要的编程范式有三种：

命令式编程(Imperative)

声明式编程(Declarative)

函数式编程(Functional)【严格说其实也属于声明式编程的子类型】

**通信协议：TCP & UDP**

[用的两个网络协议——TCP和UDP](https://baijiahao.baidu.com/s?id=1748334531859727810&wfr=spider&for=pc)

[面试突击69：TCP 可靠吗？为什么？ (baidu.com)](https://baijiahao.baidu.com/s?id=1739878370441810484&wfr=spider&for=pc)

[面试官：TCP真的可靠吗？ - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/522055020)

**TCP可靠性总结：**

**总结1**

TCP 保证可靠性的主要手段有 6 个：校验和、确认应答、超时重传、流量控制、拥塞控制、丢弃重复数据。其中流量控制和拥塞控制很容易搞混，我们要清楚的知道，流量控制是针对接收端接收能力的控制机制，而拥塞控制是针对当前网络的控制机制，所以千万不要搞混了。

**总结2**

1.TCP通过序号和超时重传保证了端到端的可靠。

2.TCP并不能保证应用层的可靠。

3.异常的情况分为，网络故障，主机崩溃和进程崩溃。网络故障和主机故障可以看作是一类故障，当然是指除了主机崩溃并在TCP放弃连接之前，就重启了的情况。

**数据模型及数据库核心知识：**



核心提炼-->**关系模型：（行式数据库/列式数据库）**

sql关系数据库模型（行列转换思想，OLTP/OLAP，OLAP-->星型模型&雪花模型）

--> 常用关系数据库、分布式关系数据库

文档模型（no-sql数据库模型，内存数据库）

图数据关系模型

地理空间数据库模型（R树，TDegine、geomesa、postgres—也支持关系数据模型）

**数据模型性能优化：**以空间换时间（PS：物化视图也是以空间换时间的思想—预先定时/实时计算更新存入“缓存”）

**索引：**牺牲写的性能，在写入的时候保证有序，以“写的耗时换取查询性能的提升”，以确保后续查询有序的性能需求,索引创建通常用于低频率变更或者不会变更的字段，否则会导致重建索引引起性能问题。

**LSM：**提升写的性能(某些数据库会牺牲查询的性能)。

**数据库优化方案选择及优先级：**

如果我们的目的在于提升数据库高并发访问的效率，那么**首先**考虑的是如何**`优化SQL和索引`**，这种方式简单有效;**其次**才是采用**`缓存的策略`**，比如使用 Redis将热点数据保存在内存数据库中，提升读取的效率;**最后**才是对数据库采用**`主从架构`**，进行读写分离。

按照上面的方式进行优化，使用和维护的成本是由低到高的。

**核心基础及技术点（关键字带超链接）：**

# [HBase、LevelDB、RocksDB等NoSQL背后的存储原理：LSM-tree日志结构合并树](http://www.python88.com/topic/149594 )

**[布隆过滤器优化LSM性能](https://www.bilibili.com/read/cv18223855)**

**【数据密集型应用--中文版.pdf—-->page 81 “**布隆过滤器**”优化LSM SSTable性能】**

[经典论文解读——布隆过滤器 (qq.com)](https://mp.weixin.qq.com/s?__biz=MjM5ODYwMjI2MA==&mid=2649776882&idx=1&sn=48b75e9461db82c684e8a906ab058aea&chksm=beccf18989bb789f87e5b2eefffd9320d8019d64b701341ac7358921ae3121b1d4f445dc72f6&mpshare=1&scene=1&srcid=0414ExLPpBNpQQyAoFeBUX9O&sharer_sharetime=1681406180909&sharer_shareid=1caebd111771a010a32c5c4cb318177d&key=036cfd63d4ddf8e71b6a8835c077938c7d6972d6dfccb37638cb369af5710b9a0f7893e5287420a08348404469adb228eb92d4eb3ad3f4b4d46232083764312e0230464fafc5766b891d2970dea390c7b2dd68fdd8a5d05c7c17f15844a7a22446346bb054a496d75a072924870496df87d978556b4bba565745cd89e2cce418&ascene=1&uin=MTMwOTY3MTQ4Mg==&devicetype=Windows+10&version=62060833&lang=en&countrycode=AD&exportkey=n_ChQIAhIQdznWs3ztiRAWIlonq+KyaRLsAQIE97dBBAEAAAAAAKVIFmUpco4AAAAOpnltbLcz9gKNyK89dVj0cJhVXsQSYHdymmeLo0Pvw34HuMBjNFR97ykiX1t/44ennngI+C7iw8rYsBN3yxShrdwJ0GzFxz8E95mzRdRmCsHBeZvKXzqYWme4opsgmCVRazCMbd0YQ6T4qFmSm+w7/U/K2DfEvxPgDEvyKBtJoG7y0SAPYKLVzU5NNo7VatgvpMmrHub8yn4QNC4S1wyxXHuX0YPdnqo8ySivm8MNfEYRNCOHwSlzmu4RzuzXg9HT4m9Pt4mOA+8cdlJ6HXTgjFE+sd7I&acctmode=0&pass_ticket=19CvL457ppviU4mei0QDV5z3OW9qoRjY/gHN8wZqgvDjWVGHGiZKkTgIGlDlpVD1H5VmRgH5/FRvpnshW8xfXw==&wx_header=0)

**全文索引（Lucene）**

**LSM原理应用场景（关系数据库和NoSql数据库）：**

**【分布式存储】**

·leveldb/rocksdb

<https://blog.csdn.net/muyimo/article/details/111697286>

**·[Hbase使跳跃表来保证内存中key的有序原理](https://baijiahao.baidu.com/s?id=1730553634179202266&wfr=spider&for=pc)**

<https://baijiahao.baidu.com/s?id=1730553634179202266&wfr=spider&for=pc>

# [跳跃表的原理以及实现](https://blog.csdn.net/qq_39033181/article/details/116560794)

[深入理解跳表及其在Redis中的应用 (baidu.com)](https://baijiahao.baidu.com/s?id=1758591784472523517&wfr=spider&for=pc)

[03 Redis 网络IO模型简介\_redis的io模型\_天秤座的架构师的博客-CSDN博客](https://blog.csdn.net/u012921921/article/details/122872607)

[(51条消息) 分布式锁-Redis红锁解决方案\_redis 红锁\_苍煜的博客-CSDN博客](https://blog.csdn.net/qq_41694906/article/details/124906428)

[Redis 分布式锁的正确实现原理演化历程与 Redisson 实战总结 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/465904846)

[Redis分布式锁的基本原理和Redis到底是否真正适用于分布式锁 (baidu.com)](https://baijiahao.baidu.com/s?id=1668195144545480668&wfr=spider&for=pc)

Tips:跳表

# 

1. LSM思想

**【Mysql-LSM树详解】**

<https://blog.csdn.net/weixin_39519454/article/details/122511748>

[LSM树详解 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/181498475)【同上】

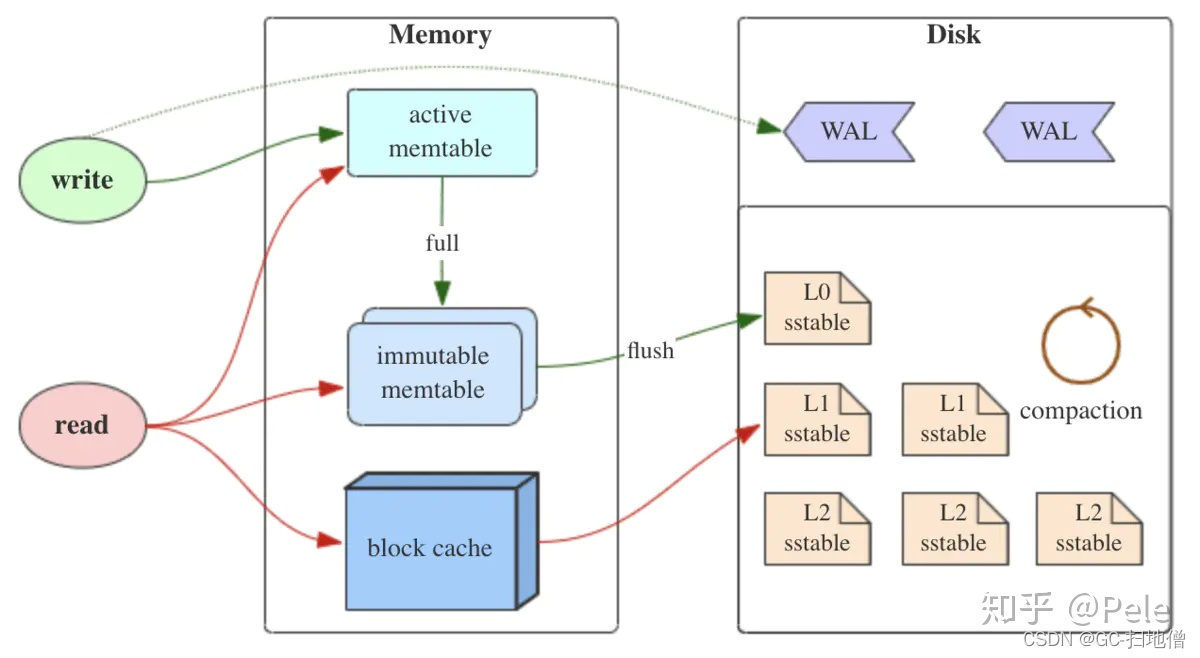
# [基于LSM-Tree存储的数据库性能改进｜DB·洞见#2 - 哔哩哔哩 (bilibili.com)](https://www.bilibili.com/read/cv18223855)

# [LSM树详解](https://blog.csdn.net/weixin_39519454/article/details/122511748 )

[LSM树详解 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/181498475)

**核心摘要如下：**

LSM树的**核心特点**是**利用顺序写来提高写性能**，但因为分层(此处分层是指的分为内存和文件两部分)的设计会稍微降低读性能，但是通过牺牲小部分读性能换来高性能写，使得LSM树成为非常流行的存储结构。



如上图所示，LSM树有以下三个重要组成部分：

***1) MemTable***

MemTable是在***内存***中的数据结构，用于保存最近更新的数据，会按照Key有序地组织这些数据，LSM树对于具体如何组织有序地组织数据并没有明确的数据结构定义，例如Hbase使跳跃表来保证内存中key的有序。

因为数据暂时保存在内存中，内存并不是可靠存储，如果断电会丢失数据，因此通常会通过WAL(Write-ahead logging，预写式日志)的方式来保证数据的可靠性。

***2) Immutable MemTable***

当 MemTable达到一定大小后，会转化成Immutable MemTable。Immutable MemTable是将转MemTable变为SSTable的一种中间状态。写操作由新的MemTable处理，在转存过程中不阻塞数据更新操作。

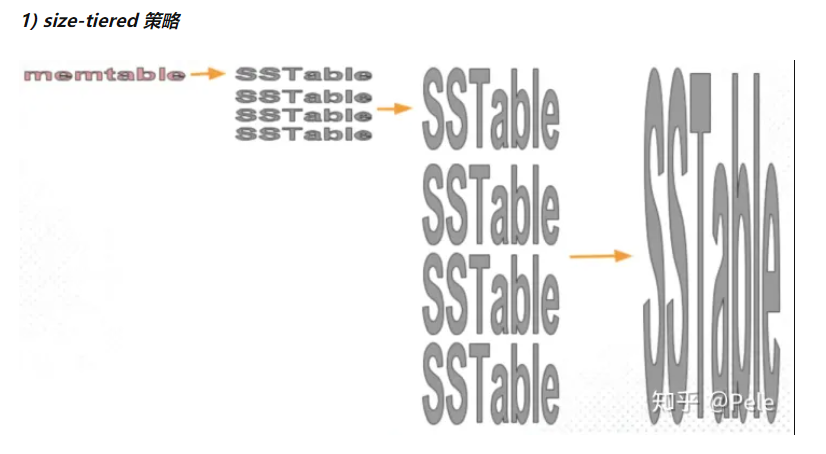
***3) SSTable(Sorted String Table)***

***有序键值对***集合，是LSM树组在***磁盘***中的数据结构。为了加快SSTable的读取，可以通过建立key的索引以及布隆过滤器来加快key的查找。

这里需要关注一个重点，LSM树(Log-Structured-Merge-Tree)正如它的名字一样，LSM树会将所有的数据插入、修改、删除等操作记录(注意是操作记录)保存在内存之中，当此类操作达到一定的数据量后，再批量地顺序写入到磁盘当中。这与B+树不同，B+树数据的更新会直接在原数据所在处修改对应的值，但是LSM数的数据更新是日志式的，当一条数据更新是直接append一条更新记录完成的。这样设计的目的就是为了顺序写，不断地将Immutable MemTable flush到持久化存储即可，而不用去修改之前的SSTable中的key，保证了顺序写。

因此当MemTable达到一定大小flush到持久化存储变成SSTable后，在不同的SSTable中，可能存在相同Key的记录，当然最新的那条记录才是准确的。这样设计的虽然大大提高了写性能，但同时也会带来一些问题：

1）冗余存储，对于某个key，实际上除了最新的那条记录外，其他的记录都是冗余无用的，但是仍然占用了存储空间。因此需要进行Compact操作(合并多个SSTable)来清除冗余的记录。  
2）读取时需要从最新的倒着查询，直到找到某个key的记录。最坏情况需要查询完所有的SSTable，这里可以通过前面提到的索引/布隆过滤器来优化查找速度。



·SSTable排序字符串表（Sorted String Table）

·WAL（write-ahead-log）预写式⽇志（也称为重做⽇志（redo log））

· **SSTable & memtable**

<https://blog.csdn.net/muyimo/article/details/111697286>

·HBase,LevelDB,RocksDB这些NoSQL存储都是采用的LSM树

**核心对比：**

**LSM和B树对比**

⽇志结构索引将数据库分解为可变⼤⼩的段，通常是⼏兆字节或更⼤的⼤⼩，并且总是按顺序编写段。相⽐之下，B树将数据库分解成固定⼤⼩的**块或⻚⾯**，传统上⼤⼩为4KB（有时会更 ⼤），并且⼀次只能读取或写⼊⼀个⻚⾯。B树这种设计**更接近于底层硬件**，因为磁盘也被安排在固定⼤⼩的块中。每个⻚⾯都可以使⽤地址或位置来标识，这允许⼀个⻚⾯引⽤另⼀个⻚⾯ —— 类似于指针，但在磁盘⽽不是在内存中。

B树的基本底层写操作是⽤新数据覆盖磁盘上的⻚⾯。假定覆盖不改变⻚⾯的位置;即，当⻚⾯被覆盖时，对该⻚⾯的所有引⽤保持完整。这与⽇志结构索引（如LSM树）形成鲜明对⽐，后者只附加到⽂件 （并最终删除过时的⽂件），但从不修改⽂件。

通常LSM树的写⼊速度更快，⽽B树的读取速度更快。 LSM树上的读取通常⽐较慢，因为它们必须在压缩的不同阶段检查⼏个不同的数据结构和SSTables。

1. **从机械硬盘到固态硬盘， b+树还有用么？**

<https://www.zhihu.com/question/304196656>

[从机械硬盘到固态硬盘， b+树还有用么？ - 知乎 (zhihu.com)](https://www.zhihu.com/question/304196656/answer/647303416)

1. 机械硬盘和ssd固态硬盘的原理对比分析

<https://www.cnblogs.com/ricklz/p/16415763.html>

1. B-Tree、B+ Tree索引总结：



**关系数据库：**

**关系数据库的数据存储（非关系数据库也借用了这个思路存储）都用了LSM这个思想的存储数据（SSTable、WAL—Write Ahead Log）。**

如果你给它的表某个字段创建了B/B+索引,就会以B/B+树去存储索引（通过写的顺序——分割生长树等操作，保证后续查询有序：以空间换时间）， 存储引擎默认会以主键作为索引， innoDB存储引擎（mysql默认8.0+以它为存储引擎）如果表没有创建主键会给你生成rowid以便存储索引。

Todo：如下结论待核实【结合“数据密集型应用--中文版.pdf” page 81 B树相关知识】

**几种常用关系数据库对比：**

# [Postgresql 与 MySQL 比较](https://blog.csdn.net/Wang_Dong_Liang/article/details/127569792 )

**[MySQL与PostgreSQL的对比 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/435829273?utm_id=0)【有很多结论不完全正确，其对比的mysql版本估计是比较老的，低于8.0】**

[PostgreSQL 创建B-Tree索引的过程 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/351302989)

[postgresql和mysql哪个好\_w3cschool](https://www.w3cschool.cn/mysql/mysql-m7kw2oyj.html)

堆组织表和索引组织表时空对比：

[达梦索引组织表和堆表 - ly\_nye - 博客园 (cnblogs.com)](https://www.cnblogs.com/ly-nye/p/16552819.html)

[堆表（HOT）和索引组织表（IOT）优缺点 – 运维那点事 (ywnds.com)](http://www.ywnds.com/?p=7702)

**[mysql分区表](https://www.cnblogs.com/wy123/p/9778590.html)：**

分区的字段，必须是表上所有的唯一索引（或者主键索引）包含的字段的子集  
 换句话说就是：（所有的）字段必须出现在（所有的）唯一索引或者主键索引的字段中，  
 或者更通俗讲就是，一个表上有一个或者多个唯一索引的情况下，分区的字段必须被包含在所有的主键或者唯一索引字段中。

**Oracle没有这限制**

# [堆表和索引组织表(深入思考：oracle、innoDB和mylsam)](https://blog.csdn.net/qq_44793993/article/details/120284793)

****

**总结与结论：**

1. Oracle和Mysql默认索引使用的是B+树存储，postgres SQL默认索引使用的是B树存储
2. Oracle和Postgres都是用的堆存储，不像mysql一样有存储引擎一说，mysql默认使用的是InnoDB存储引擎至少会有一个“主键索引/聚簇索引”（内节点中目录项记录的内容是`索引列 +页号`的搭配，数据会存储在叶子结点），非聚簇索引叶子结点存的是指针“索引列的值+ 主键值+页号”（参考“206\_索引的数据结构.md”）
3. Oracle B+索引的叶子节点存储的是rowid，而Mysql B+索引叶子节点存储的是用户数据（聚簇索引）
4. 非聚簇索引oracle、postgres、mysql叶子结点都是存储的指针而不是用户数据（Oracle和Postgres都是非聚集索引，堆表不存在聚簇索引）
5. MySql和Postgers索引是以页的形式存储，oracle索引是以块的形式存储（布隆过滤）
6. 可以通过业务代码，联合主键将数据库某几个字段字符串合并加密（比如MD5\_KEY），构造一个新的字段作为“非聚簇索引”进行性能优化

F.Oracle默认都是用的堆表，Postgres完全面向堆表的，mysql默认用的是索引表（InnoDB存储引擎）。

oracle还支持索引组织表、聚集表等，需要额外使用organization进行声明。

**【 Ref ： [深入思考：oracle、innoDB和mylsam](https://blog.csdn.net/qq_44793993/article/details/120284793)**

**堆组织表的**最大特点就是无序，数据是以堆的方式管理的，增加数据的时候，会使用堆中找到第一个能够放下此数据的自由空间。从表中删除数据的时候，允许以后的新插入的数据或者修改完毕的数据复用这块空间。当进行全表搜索的时候，数据获取的顺序和“命中”顺序有关（物理层面），和插入顺序无关。

**堆组织表的主键和唯一索引区别不大，和普通索引的区别也基本是在唯一约束上的区别。创建索引可以优化查询性能，避免全表扫描。**

**索引组织表**一定会有一个主键，即使不声明也会隐式分配一个主键。虽然从数据库层面来看，数据行还是无序的，但是索引组织表创建后，数据文件和索引文件是集中存放的，因此至少会存在一个主键索引。当数据进行查询的时候，优化器会看到表中存在索引，于是数据返回的顺序是按照主键排好序的。

以上说了这么多，就是想说：集合数据库底层的数据最原始的状态是无序的、堆放式存储的。而引入基础索引的最大意义是将数据行按照某些字段组织起来。

这种**堆表+索引的方式，索引文件和数据文件是独立存放的，索引文件中保存了某个字段在数据文件的物理地址rowid。（不存在什么回表的说法，因为是都是根据rowid去数据文件中查询数据行）**

**索引组织表的应用场景**:不适合频繁删改、完全由主键组成的表、频繁需要查询主键的表、对排序有需求

**】**

**postgres binlog实现事务回滚和重写【todo】**

# [深入思考：oracle、innoDB和mylsam](https://blog.csdn.net/qq_44793993/article/details/120284793)

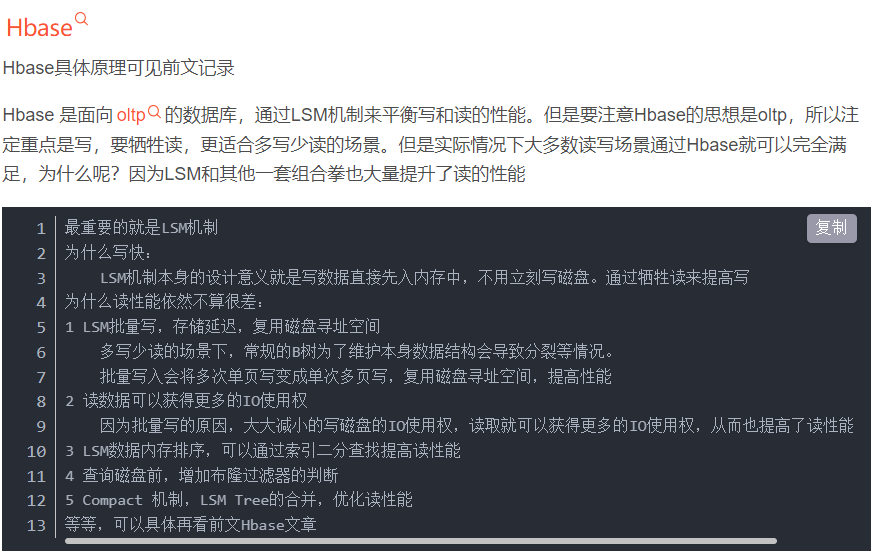
**[聚簇索引和非聚簇索引的区别\_glenshappy的博客-CSDN博客](https://blog.csdn.net/glenshappy/article/details/127414513)**

**[聚簇索引和非聚簇索引 (qq.com)](https://mp.weixin.qq.com/s?__biz=MzAxMjY5NDU2Ng==&mid=2651861539&idx=1&sn=77e036f27b300dfddc979842f3255b30&chksm=8049776ab73efe7cdc3c777e6c499bfcd92538d97210e8b745e7402d2e1068367a9877f8dca4&scene=27)**

**[一文详解InnoDB最核心组件Buffer Pool](https://zhuanlan.zhihu.com/p/424152851)**

**[mysql的数据加载到内存的过程和机制](https://blog.csdn.net/sinat_38852244/article/details/123816051?ydreferer=aHR0cHM6Ly93d3cuYmFpZHUuY29tL2xpbms/dXJsPUlNWV9QUjNIc0F2emVJaU5iWmZBTU41LVBrck1GTTFCWnhUc2l5aHUzc0NtU0xVOXlfdVU1aGU1TlBxZ1Q3RU9OczFEbUE3dWJVV2M1a2RyYTVTR3FmZGNFWjFWX1N3bmtDMjlxVU9FVnp1JndkPSZlcWlkPTk5YmZkOWM4MDAwMjA5NTAwMDAwMDAwNDY0MzYyNzM1)**

[(50条消息) OLTP和OLAP问题的个人总结\_我爱肉肉的博客-CSDN博客](https://blog.csdn.net/weixin_40503364/article/details/126214403?ydreferer=aHR0cHM6Ly93d3cuYmFpZHUuY29tL2xpbms/dXJsPW40RHMwNVV2dS1WM0NyM3N2N1RHd2MzdTc3NHRDcFFoYUh2c21NZkc4WjNGYU9xbzJvQk5PWjlJNHBlRlEwSXhiVzExcnpnOVBhOXplcWdYV3ItVU5MMjA3c2IyUmxMa0l6X2QzMG0zYTJpJndkPSZlcWlkPThhNGM2OTNhMDAwMDllMWQwMDAwMDAwNDY0MzY3M2Rk)



**详细对比：**

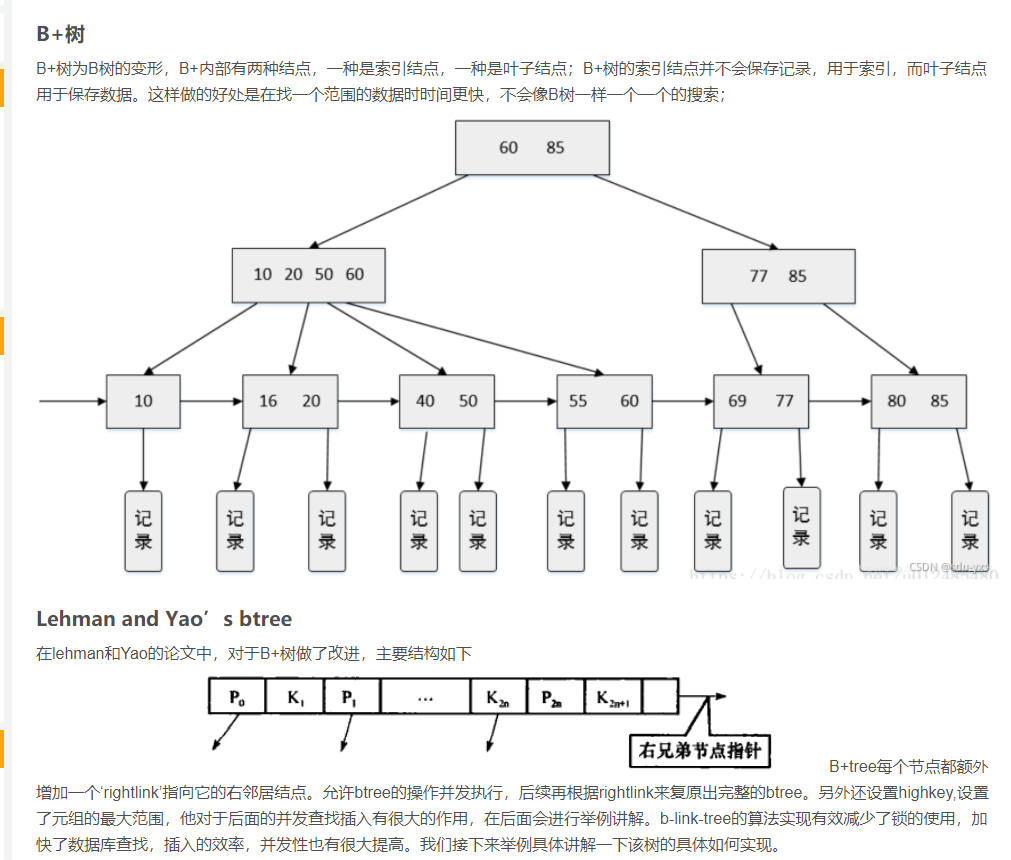
**注意区分：**

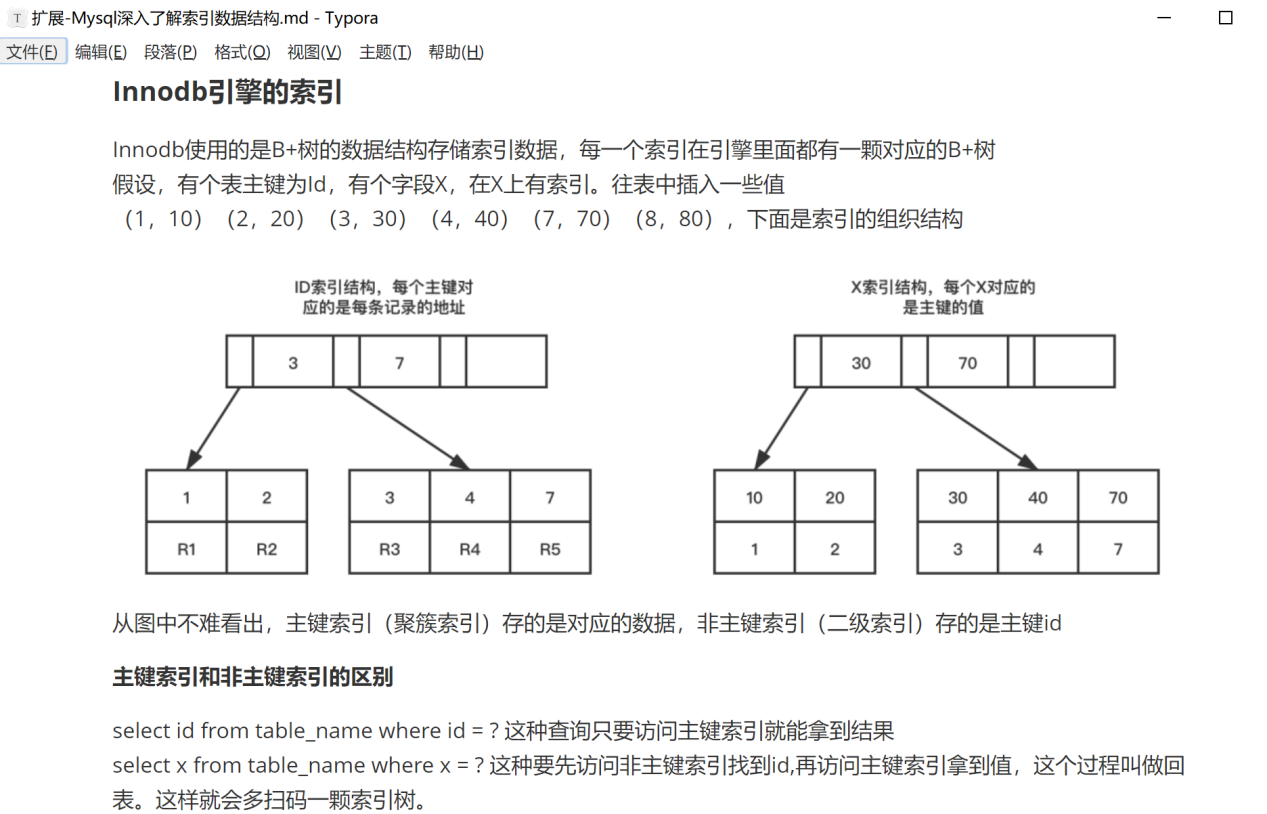
**根节点（Root Pages/MataPage）& 内节点（Internal Pages） & 叶子结点（Leaf Pages）**

**mysql B+索引：**

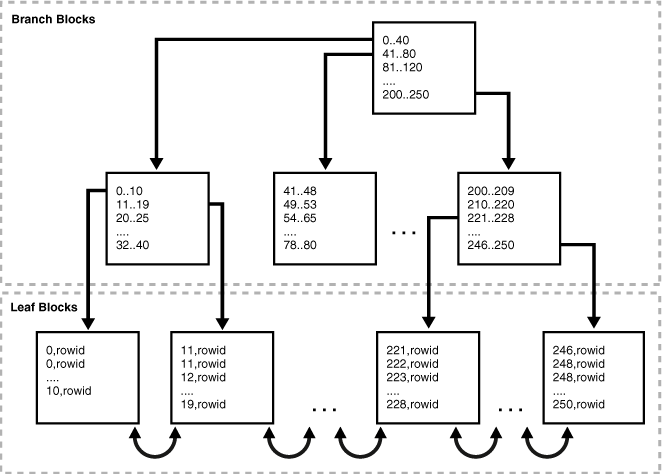
以页的形式存储，只有叶子节点才存储详细数据（含双向链表：以便范围查找），中间的那些层级的只存index

## Optimization and Indexes: <https://dev.mysql.com/doc/refman/8.0/en/optimization-indexes.html>

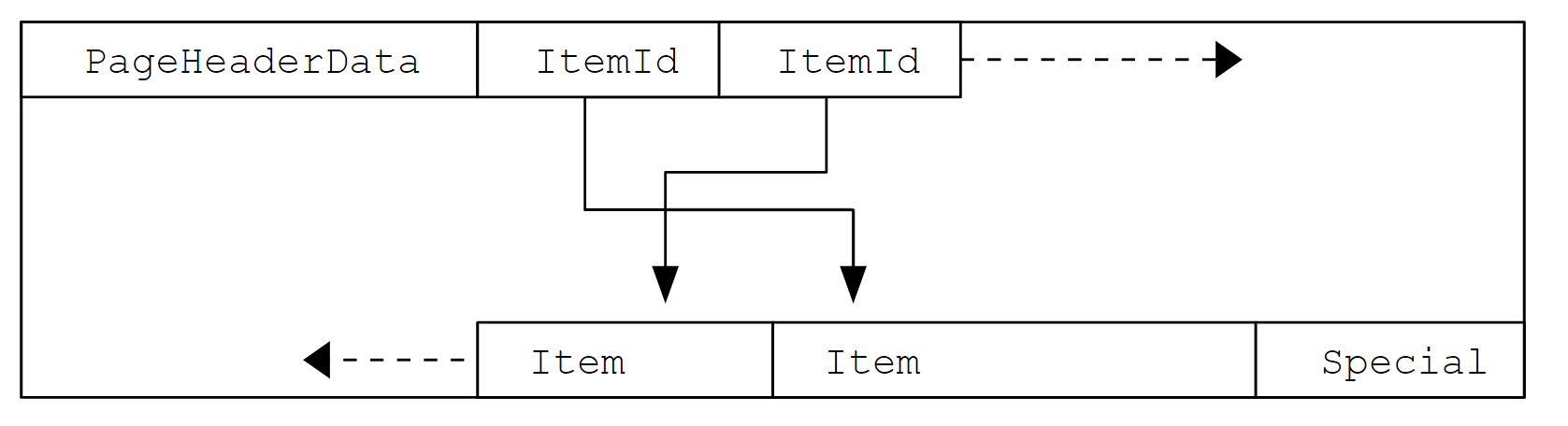




**[oracle B+索引](https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/indexes-and-index-organized-tables.html" \l "GUID-FC93A85B-C237-4249-AD1E-FF54576ED050)：**以块的形式存储，所有节点（含叶子节点）只存储了index & rowid没有详细数据，



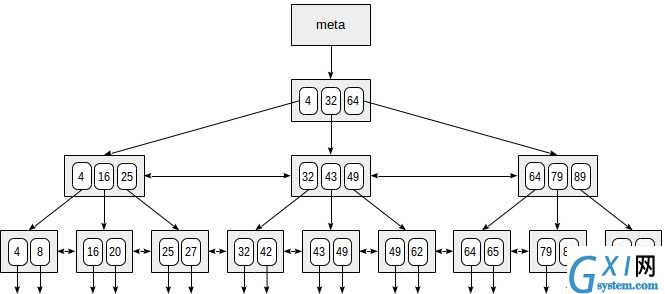
**[postgres B索引](https://www.postgresql.org/docs/15/btree-implementation.html)：**[以页的形式存储](https://www.postgresql.org/docs/15/storage-page-layout.html)，



# 

# [postgreSQL源码分析——索引的建立与使用——B-Tree索引](https://blog.csdn.net/m0_53446118/article/details/120914806)

**[PostgreSQL中的B-TREE索引](https://www.gxlsystem.com/shujuku-150130.html)**



该索引最顶层的页是元数据页，该数据页存储索引root页的相关信息。内部节点位于root下面，叶子页位于最下面一层。向下的箭头表示由叶子节点指向表记录（TIDs）。 B-tree 中一个节点有多个分支，即每页（通常 8KB ）具有许多 TIDs 。

1、B-tree是平衡树，即每个叶子页到root页中间有相同个数的内部页。因此查询任何一个值的时间是相同的。

2、B-tree中一个节点有多个分支，即每页（通常8KB）具有许多TIDs。因此B-tree的高度比较低，通常4到5层就可以存储大量行记录。

3、索引中的数据以非递减的顺序存储（页之间以及页内都是这种顺序），同级的数据页由双向链表连接。因此不需要每次都返回root，通过遍历链表就可以获取一个有序的数据集。

b+tree会将行存储在索引页中，所以一页能存下的记录数会大大减少，从而导致b+tree的层级比单纯的b-tree深一些。 特别是行宽较宽的表。

例如行宽为几百字节，16K的页可能就只能存储十几条记录，一千万记录的表，索引深度达到7级，加上metapage，命中一条记录需要扫描8个数据块。

而使用PostgreSQL堆表+PK的方式，索引页通常能存几百条记录（以16K为例，约存储800条记录），索引深度为3时能支撑5亿记录，所以命中一条记录实际上只需要扫描5个块(meta+2 branch+leaf+heap)。

# 

# [PostgreSQL B-Tree论文解读](https://zhuanlan.zhihu.com/p/166398779)

pg中文社区

[pg社区](https://www.postgresql.org/ )：<https://www.postgresql.org/>

**关系数据库索引深入扩展**

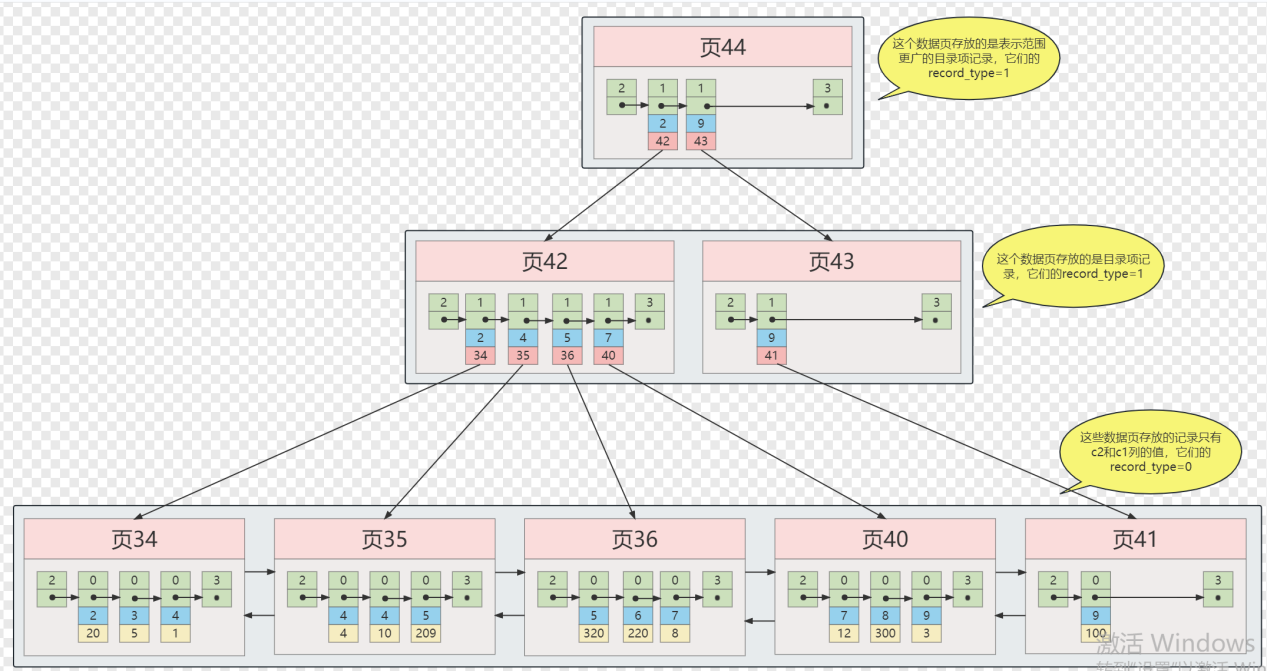
[聚簇索引和非聚簇索引的区别\_glenshappy的博客-CSDN博客](https://blog.csdn.net/glenshappy/article/details/127414513)

[聚簇索引和非聚簇索引 (qq.com)](https://mp.weixin.qq.com/s?__biz=MzAxMjY5NDU2Ng==&mid=2651861539&idx=1&sn=77e036f27b300dfddc979842f3255b30&chksm=8049776ab73efe7cdc3c777e6c499bfcd92538d97210e8b745e7402d2e1068367a9877f8dca4&scene=27)

# [深入思考：oracle、innoDB和mylsam](https://blog.csdn.net/qq_44793993/article/details/120284793)

1. **聚集索引：**聚簇索引并不是一种单独的索引类型，而是一种数据存储方式(所有的用户记录都存储在了**叶子节点即在索引中存储所有⾏数据**)，也就是所谓的索引即数据，数据即索引。
2. **非聚集索引（仅在索引中存储对数据的引⽤）：** 上边介绍的聚簇索引只能在搜索条件是主键值时才能发挥作用，因为B+树中的数据都是按照主键进行排序的。那如果我们想以别的列作为搜索条件该怎么办呢? 肯定不能是从头到尾沿着链表依次遍历记录一遍。

**答案:**我们可以多建几棵B+树，不同的B+树中的数据采用不同的排序规则。比方说我们用c2列的大小作为数据页、页中记录的排序规则，再建一棵B+树



**小结: 聚簇索引与非聚簇索引的原理不同，在使用上也有一些区别:**

A. 聚簇索引的`叶子节点存储的就是我们的数据记录`，`非聚簇索引的叶子节点存储的是数据位置(oracle是这样，)`。非聚族索引不会影响数据表的物理存储顺序。

B. 一个表`只能有一个聚簇索引`，因为只能有一种排序存储的方式，但可以有`多个非聚族索引`，也就是多个索引目录提供数据检索。

C. 使用聚簇索引的时候，数据的查询效率高，但如果对数据进行插入，删除，更新等操作，效率会比非聚簇索引低。

**参见文档：**

****

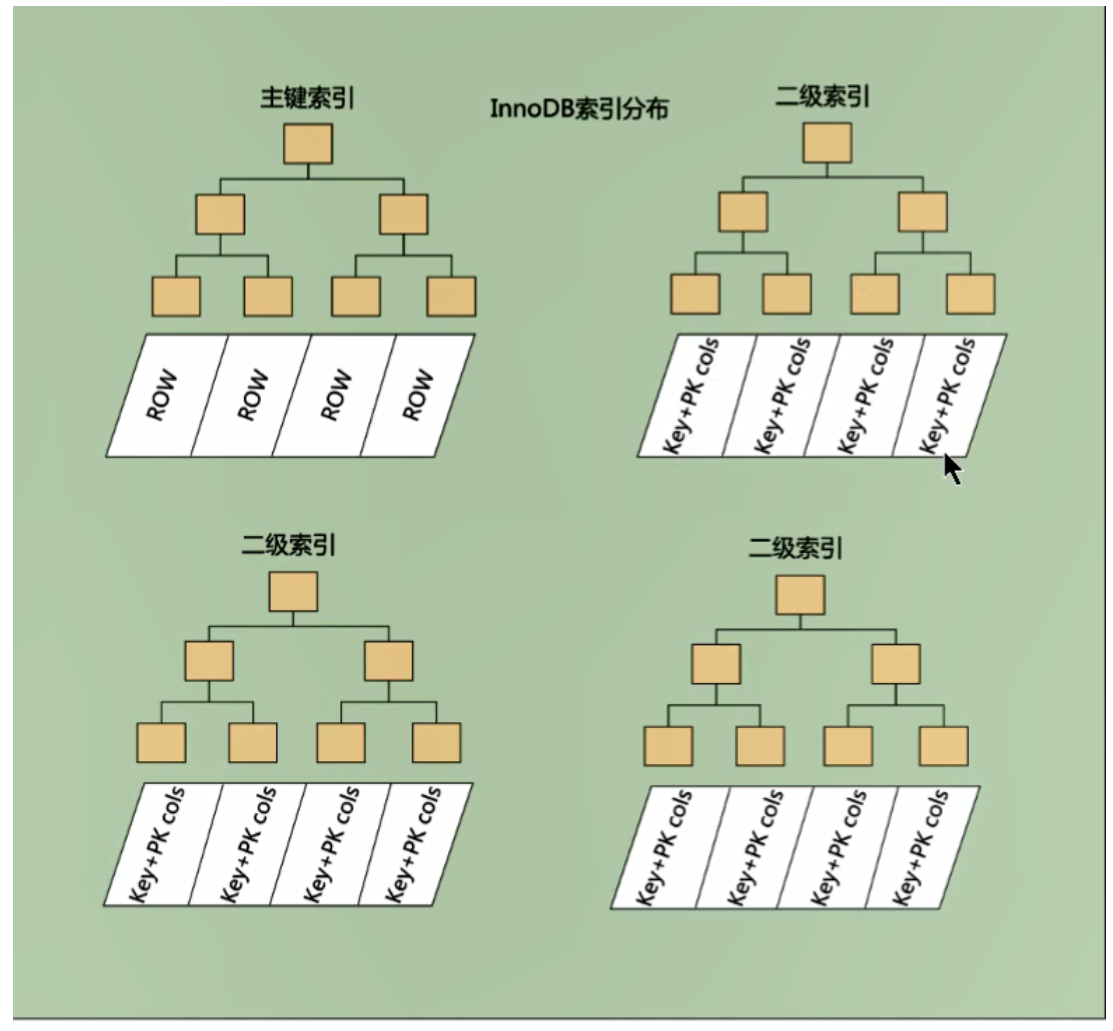
## 第“3.3 常见索引概念”--“3.4 InnoDB的B+树索引的注意事项”

**列的索引（index with included columns）或覆盖索引（covering index）**

其存储表的⼀部分在索引内。这允许通过单独使⽤索引来回答⼀些查询（这种情况叫做：索引覆盖（cover）了查询）。与任何类型的数据重复⼀样，**聚簇和覆盖索引可以加快读取速度**，**但**是它们**需要额外的存储空间**，并且会**增加写⼊开销**。**数据库还需要额外的努⼒来执⾏事务保证**，因为应⽤程序不应该因为重复⽽导致不⼀致。

**概念:回表**

我们根据这个以c2列大小排序的B+树只能确定我们要查找记录的主键值，所以如果我们想根据c2列的值查找到完整的用户记录的话，仍然需要到聚簇索引中再查一遍，这个过程称为回表。也就是根据2列的值查询一条完整的用户记录需要使用到2棵B+树!

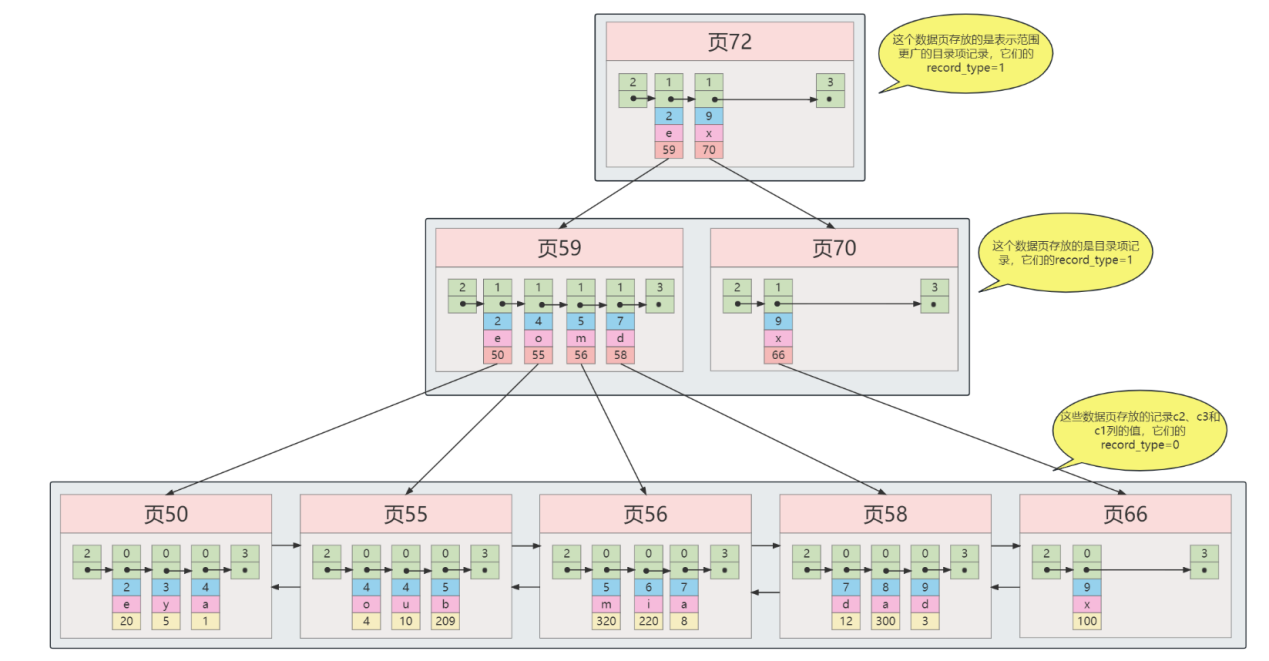


**联合索引**

我们也可以同时以多个列的大小作为排序规则，也就是同时为多个列建立索引，比方说我们想让B+树按照 `c2和c3列`的大小进行排序，这个包含两层含义:

先把各个记录和页按照c2列进行排序

在记录的c2列相同的情况下，采用c3列进行排序



1. **数据库文件后缀名**

MySQL：

**InnoDB存储引擎模式**

**表结构**

为了保存表结构，InnoDB在数据目录下对应的数据库子目录下

创建了一个专门用于描述表结构的文件 **表名.frm**

ps: **MySQL8.0**中不存在表名.frm(表结构文件)，并到了表名**.ibd**中

**表中数据和索引**

含MySQL 5.6.6之后版本中，InnoDB新建表不会默认存到系统表空间，**默认使用独立表空间**

系统表空间（system tablespace）

**默认情况下**，存放在**ibdata1**文件

独立表空间(**file-per-table tablespace**)

会在数据库子目录下创建表名**.ibd的文件**

**配置**

[server]

# **配置系统表空间**

对应文件路径以及大小，这个文件是自扩展文件。

innodb\_data\_file\_path=data1:512M;data2:512M:autoextend

# **配置默认表空间**

0：代表使用系统表空间； 1：代表使用独立表空间

innodb\_file\_per\_table=0

**MyISAM存储引擎模式**

**默认是使用InnoDB，需要创表时通过指定引擎**

CREATE TABLE `user` (

`id` INT NOT NULL AUTO\_INCREMENT,

`username` VARCHAR (255) COLLATE utf8mb4\_german2\_ci DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE = MYISAM DEFAULT CHARSET = utf8mb4 COLLATE = utf8mb4\_german2\_ci;

**表结构**

**MySQL5.7**中和InnoDB一样，存储在**表名.frm**中

**MySQL8.0**中存储在**表名.xxx.sdi**中

**表中数据和索引**

MyISAM中索引都是二级索引， 数据和索引分开存放

**表名.MYD**(MYData)

​ **数据信息文件，存储数据信息**

**表名.MYI**(MYIndex)

​ **存放索引信息文件**

# [MYSQL中INNODB，MYISAM，MEMORY索引区别](https://www.cnblogs.com/king0207/p/14542191.html)

SQLServer： .mdf

Oracle：.dbf 和 .ora

PostgreSQL：无后缀名

Access：早期是 .mdb ，后来是 .accdb

1. 数据库架构及容灾

**[Mysql 三大日志（binlog、redolog和undolog） - 雷雷提 - 博客园 (cnblogs.com)](https://www.cnblogs.com/Alei777/p/16439794.html)**

**[Mysql 数据恢复逻辑 基于binlog redolog undolog (baidu.com)](https://baijiahao.baidu.com/s?id=1739504407945420445&wfr=spider&for=pc)**

**[Mysql 核心日志（redolog、undolog、binlog） - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/213770128)**

[面试系列-mysql主从复制 - 腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/2142283)

[Oracle专家高级编程 第五章 重做和回滚 - BigBender - 博客园 (cnblogs.com)](https://www.cnblogs.com/BigBender/p/14663800.html)【讲解不够清晰】

[oracle中数据是怎样前滚和回滚的\_百度知道 (baidu.com)](https://zhidao.baidu.com/question/692562557733811844.html)【讲解不够清晰】

[mysql主从复制 - 腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/2142283)

1. **谷歌三驾马车 【todo】**
2. 事务的实现及源码 【锁粒度控制：todo】--》by wang









**a.**数据库的块或者页，是一个小文件，这个文件会尽量与操作系统的块或者页对齐，以便读取的时候，尽可能的利用操作系统的io性能

**b.**数据库的在实现事务的时候，会上锁，其根本就是对这些块或者页的文件上锁，这样可以避免对全表上锁

**c.**我们假设每个块文件有40条数据，我们通过mvcc的方式，允许每一行可以复制出一个块，允许他只对其中一行进行修改，这样一来，这个块儿可以最多允许有40个副本，也就是同时允许有40个请求对这个块进行修改, 如此，就在满足事务的情况下，保证了性能。

而redis hive等数据库，其文件分布方式为了吞吐量，就不会拆分成一个个小块儿，所以他们实现事务的成本就及其高昂，通常都会支持事务

【**todo：**关系数据库B/B+搜索过程中，树如何加载的？ **注意点：**按照子块还是整个树？】

PS：**mvcc** 一般不会讲到文件锁，这里最关键的就是这个点，具体的锁，其实是锁到一个个文件上，这个点很重要，因为连续的小文件的写入和读取，性能一定远远弱于单一大文件，而且多个小文件，也无法做到磁盘的连续读写，这就意味着关系型数据库从最基本的原理来讲，他在大数据的情况下就不可能达到kafka或者hbase这样的高性能，这也意味着，事务和高性能，需要有取舍，除非存储设备产生重大变化。【措辞有些问题，忽略了索引的存在以及】

【**todo：** kafka事务以及关系数据库】

可参考[【转：分布式存储】-leveldb/rocksdb](https://blog.csdn.net/muyimo/article/details/111697286)

1. [分布式事务的挑战及常见解法 (qq.com)](https://mp.weixin.qq.com/s?__biz=Mzg5Mjc3MjIyMA==&mid=2247561421&idx=1&sn=a2e444ee24f1aded96699a4279fe218b&chksm=c03abadef74d33c8cb73a4495468088549b6f355e7b9dba77be74c1ce6c4e2782fa63193fb53&mpshare=1&scene=1&srcid=0417svcGocxMtbYCcQspDrgb&sharer_sharetime=1681741715683&sharer_shareid=1caebd111771a010a32c5c4cb318177d&key=4beaf6b02602a277bc20b88b9b5c1ddfc896f0ea781ff40fa3de11cd2b6ec272dc4e43e97d5a2c72434b9835f36a1fef0ebaee33bfca3bd7dcbf3eb2485074029c3924cead50831887d5827058ad850b6f75675d4c24d8844ad6cb3f7de487c7c0a104cb0640bd26d9eba3e98f29e36a9b1fba7638b6bbf44baa363d2cde3eb9&ascene=1&uin=MTMwOTY3MTQ4Mg==&devicetype=Windows+10&version=62060833&lang=en&countrycode=AD&exportkey=n_ChQIAhIQiuu344dwmZp1zq1XRtiwAxLsAQIE97dBBAEAAAAAALdEFp+C5H4AAAAOpnltbLcz9gKNyK89dVj0plA1xg5+lpjYfx9YtVsLL5OnASVXFIkDflpZOJrV85kLpvneT+c7z2NvM+PjFrff84MG0PjaF7TKMTPybsXOq0OYLE4BWJz0TXByoeEm26uhra1dC2y/q0yC2ZzFhKYKUTEfNN/kQQ+O3lXyK6T2HocqVzwdmFlsGTuSkov3UymRU0mPT7l9tKhuHIWHMv4yMfGOm2/YCA82iGT3+WDvc7jNgDj3OmTUNadK5fIpyzDK3p35VFxYQv9LBYij5pUr6GTkwGNr&acctmode=0&pass_ticket=19CvL457ppviU4mei0QDV5z3OW9qoRjY/gHN8wZqgvBsj/3S2zVHnL8L1TkNFOu7DAilbpEL2ssQBR3UWqmt1w==&wx_header=0)

【强一致性方案：CockroachDB 或者mysql 组复制技术，简称 MGR (MySQL Group Replication)】

[分布式应用框架Akka详解 - java的涟漪 - ITeye博客](https://www.iteye.com/blog/luo-yifan-2035833)

# 1.[Actor——Actor模型原理的通俗理解 (转) - 会飞的斧头 - 博客园 (cnblogs.com)](https://www.cnblogs.com/xingchong/p/16202290.html)

# 2.[Akka 架构简介](https://www.jianshu.com/p/dc9e39a90d5b)

3.[分布式应用框架Akka详解 - java的涟漪 - ITeye博客](https://www.iteye.com/blog/luo-yifan-2035833)

4. [深入解析actor 模型（一)： actor 介绍及在游戏行业应用 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/427806717)

1. **Kafka分布式存储消息队列相关**

[大白话带你认识Kafka (qq.com)](https://mp.weixin.qq.com/s?__biz=Mzg5Mjc3MjIyMA==&mid=2247544111&idx=1&sn=b748fb7721983343877b393b1fcba74d&source=41&ascene=1&devicetype=Windows+10&version=62060833&nettype=WIFI&abtest_cookie=AAACAA%3D%3D&lang=en&countrycode=AD&exportkey=n_ChQIAhIQMF86yj4mQMc6OnJECoIj%2FRLsAQIE97dBBAEAAAAAAI38DWuWd7UAAAAOpnltbLcz9gKNyK89dVj0vJFjsT70RIlNmPlAAxBykPdKNOGtx5okdiYGuGL%2B%2Bidi6oLkZ1QZb7spcEKHoFZaRjImu1YiNM34i6vEJr3%2B7QgmTtV4cA5oFJFJVMz4Bi5fmrmJjlCo%2FN4U1URK%2Be21HjNpSkzW14LA0x4JreqaNGiMj55ynxJ2oNgUTtPii%2BkOwBtJEDH%2BevNdJWDs8dKn6HOG6FK5RGaTsgTn4ognPKYuo4ewYEJSB5MRb7%2B2npCB6V93ahK2r57S2iGElL%2F6PnGB8kBd&pass_ticket=pe%2B%2Fk65Jk6LO2rZVpTyHt94IQAteml13QFfvsQj1JQtPnCHdXudSoLyB6pMO3RJMvuXyPNJebfTKFflH%2BA0Oqw%3D%3D&wx_header=0&key=0525c3553a228e148abdf99aad049ba5c8e7e30d75d1f5546024b26f45101f4af9730a21ec6e85b1a843999134bf6cd39d375ba1d9fd3520c8edb9ee94f6d78cbfe94e9b0bcf817f898aa2f01f97e9d25216187cdae9cf40778aafb4e472ff7baed1263859d1ba335fe8ac4892c17357276afa2c83e28ee08a31b90c218aa71d&uin=MTMwOTY3MTQ4Mg%3D%3D&acctmode=0)

#### [为什么Kafka速度那么快 - 邴越 - 博客园 (cnblogs.com)](https://www.cnblogs.com/binyue/p/10308754.html)【磁盘顺序读写RAL/WAL、LSM Tree思想、Memory Mapped Files(后面简称mmap)——基于sendfile实现Zero Copy】

[如何保证消息不被重复消费（如何保证消息消费时的幂等性） - \_\_Meng - 博客园 (cnblogs.com)](https://www.cnblogs.com/mengchunchen/p/10007537.html)

[(57条消息) kafka怎么保证消息顺序？\_kafka如何保证消息顺序消费\_普通网友的博客-CSDN博客](https://blog.csdn.net/m0_67400972/article/details/126074716)

[(57条消息) kafka消息丢失原因及解决方案\_kafka消息丢失解决方案\_YonJ的博客-CSDN博客](https://blog.csdn.net/YoungJ_Zhou/article/details/125605128)

[kafka入门介绍 - OrcHome](https://www.orchome.com/5)

[kafka日志存储以及清理机制\_kafak\_泽睿\_InfoQ写作社区](https://xie.infoq.cn/article/34bd37dc6554a6bf63570de90)

[(50条消息) 如何确定Kafka的分区数、key和consumer线程数\_利白的博客-CSDN博客](https://blog.csdn.net/libaineu2004/article/details/80168583)

[消息中间件你选对了吗？Kafka与RabbitMQ谁更胜一筹？ (qq.com)](https://mp.weixin.qq.com/s?__biz=MzkwOTIxNDQ3OA==&mid=2247533060&idx=1&sn=458f932dbca7c66a8c5bacc2365f55cd&chksm=c13c1465f64b9d736abc16824f2617010e66643e3362df7f6a69dc04455358ce4687db3722c0&mpshare=1&scene=1&srcid=0417AGwk4H9Y0HpEYek35vLw&sharer_sharetime=1681661458695&sharer_shareid=1caebd111771a010a32c5c4cb318177d&key=0604132a6ad20822d772b8100a6111587328bbbd4e1855e35da7ba9617536df33870fcb662a0271f0ccb17008f8dc92f6a18dbc074bfce17affa7a01bd0ccffa3e454ba0cfcc24647398767e59ef0f501faf1b24725e9aaebc8cfdde7c84cfe13d50a1e7eaf6ddf20410803ad9f9b1e1b23d34f3c0c6176604762d8a469488e4&ascene=1&uin=MTMwOTY3MTQ4Mg%3D%3D&devicetype=Windows+10&version=62060833&lang=en&countrycode=AD&exportkey=n_ChQIAhIQSW1Y%2Ba9RlDaEpf2cjV%2BfPRLsAQIE97dBBAEAAAAAAKQAExspBvoAAAAOpnltbLcz9gKNyK89dVj03aNaV%2FPLY28aXnIgobQZRYSTLgKVnDRwgeEEqQza7D1S4f3l%2Ftd1%2FIx%2BNGgWFEFvk7b0m3EhLWNd16N8cuKZjFjtr84C8D%2BC%2FeS9sF6VqdqwApF71dscOhlbpzDUNmzNiLZjnHqH%2FAvwKtI2MxD6xwOaRgl2ZAGVleoTYZEnwTVG6z1%2BLeveFPtbFwpit8HDs%2F4Yvf0KqcuqzoTbvGWiFakcBBjp29J19SRNoWzNooGgYvKaT%2FRfOaxGXS91gRK%2BB8024geC&acctmode=0&pass_ticket=pe%2B%2Fk65Jk6LO2rZVpTyHt94IQAteml13QFfvsQj1JQvDnLtagfwipHGKJjSk0AznJxciGmrcHpwd%2B4ez5wlORQ%3D%3D&wx_header=0)【场景对比选型】

**扩展：[kafka满足的是CAP定律当中的CA](https://cloud.tencent.com/developer/article/2231973)**

[谈一谈 Kafka 在 CAP 三大特性之间做的妥协和改进 - 腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/2231973)

[一文搞懂HBase的基本原理-hbase的工作原理 (51cto.com)](https://www.51cto.com/article/660015.html) 【CAP原理及常见应用所属类型】

【摘取】

## CAP理论

2000年，Berkerly大学有位Eric Brewer教授提出了一个CAP理论，在2002年，麻省理工学院的Seth Gilbert(赛斯·吉尔伯特)和Nancy Lynch(南希·林奇)发表了布鲁尔猜想的证明，证明了CAP理论的正确性。所谓CAP理论，是指对于一个分布式计算系统来说，不可能同时满足以下三点：

* 一致性(Consistency)

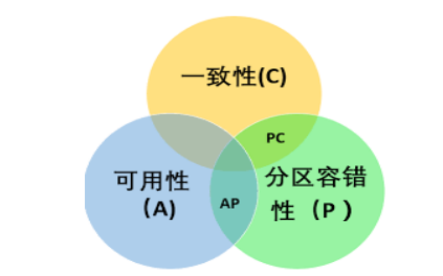
等同于所有节点访问同一份最新的数据副本。即任何一个读操作总是能够读到之前完成的写操作的结果，也就是说，在分布式环境中，不同节点访问的数据是一致的。

* 可用性(Availability)

每次请求都能获取到非错的响应——但是不保证获取的数据为最新数据。即快速获取数据，可以在确定的时间内返回操作结果。

* 分区容错性(Partition tolerance)

以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。即指当出现网络分区时(系统中的一部分节点无法与其他的节点进行通信)，分离的系统也能够正常运行，即可靠性。



如上图所示：一个分布式的系统不可能同时满足一致性、可用性和分区容错性，最多同时满足两个。当处理CAP的问题时，可以有一下几个选择：

* 满足CA，不满足P。将所有与事务相关的内容都放在同一个机器上，这样会影响系统的可扩展性。传统的关系型数据库。如MySQL、SQL Server 、PostgresSQL等都采用了此种设计原则。
* 满足AP，不满足C。不满足一致性(C)，即允许系统返回不一致的数据。其实，对于WEB2.0的网站而言，更加关注的是服务是否可用，而不是一致性。比如你发了一篇博客或者写一篇微博，你的一部分朋友立马看到了这篇文章或者微博，另一部分朋友却要等一段时间之后才能刷出这篇文章或者微博。虽然有延时，但是对于一个娱乐性质的Web 2.0网站而言，这几分钟的延时并不重要，不会影响用户体验。相反，当发布一篇文章或微博时，不能够立即发布(不满足可用性)，用户对此肯定不爽。所以呢，对于WEB2.0的网站而言，可用性和分区容错性的优先级要高于数据一致性，当然，并没有完全放弃一致性，而是最终的一致性(有延时)。如Dynamo、Cassandra、CouchDB等NoSQL数据库采用了此原则。
* 满足CP，不满足A。强调一致性性(C)和分区容错性(P)，放弃可用性性(A)。当出现网络分区时，受影响的服务需要等待数据一致，在等待期间无法对外提供服务。如Neo4J、HBase 、MongoDB、Redis等采用了此种设计原则。

1. 扩展 其他存储数据库

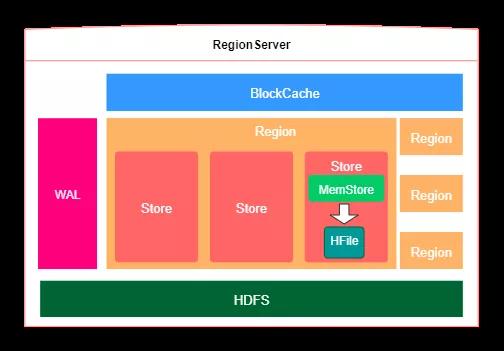
【HBase数据库】

[一文搞懂HBase的基本原理-hbase的工作原理 (51cto.com)](https://www.51cto.com/article/660015.html)

摘要：

### 微观架构

上一小节对HBase的整体架构进行了说明，接下来再看一下内部细节，如下图所示：展示了一台RegionServer的内部架构。

[](https://s2.51cto.com/oss/202104/27/6ab77eb3e6a20fbef80b3d15ecec1de7.jpg)

如上图所示：一个RegionServer可以存储多个region，Region相当于一个数据分片。每一个Region都有起 始rowkey和结束rowkey，代表了它所存储的row范围。在一个region内部，包括多个store，其中一个store对应一个列族，每个store的内部又包含一个MemStore，主要负责数据排序，等超过一定阈值之后将MemStore的数据刷到HFile文件，HFile文件时最终存储数据的地方。

**值得注意的是：**

一台RegionServer共用一个WAL(Write-Ahead Log)预写日志，如果开启了WAL，那么当写数据时会先写进WAL，可以起到容错作用。WAL是一个保险机制，数据在写到Memstore之前，先被写到WAL了。这样当故障恢复的时候可以从WAL中恢复数据。另外，每个Store都有一个MemStore，用于数据排序。一台RegionServer也只有一个BlockCache，用于读数据是进行缓存。

* WAL预写日志

\*\*Write Ahead Log (WAL)\*\*会记录HBase中的所有数据，WAL起到容错恢复的作用，并不是必须的选项。在HDFS上，WAL的默认路径是/hbase/WALs/,用户可以通过hbase.wal.dir进行配置。

WAL默认是开启的，如果关闭，可以使用下面的命令Mutation.setDurability(Durability.SKIP\_WAL)。WAL支持异步和同步的写入方式，异步方式通过调用下面的方法Mutation.setDurability(Durability.ASYNC\_WAL)。同步方式通过调用下面的方法：Mutation.setDurability(Durability.SYNC\_WAL)，其中同步方式是默认的方式。

关于异步WAL，当有Put、Delete、Append操作时，并不会立即触发同步数据。而是要等到一定的时间间隔，该时间间隔可以通过参数hbase.regionserver.optionallogflushinterval进行设定，默认是1000ms。

* MemStore

每个Store中有一个MemStore实例。数据写入WAL之后就会被放入MemStore。MemStore是内存的存储对象，只有当MemStore满了的时候才会将数据刷写(flush)到HFile中。

为了让数据顺序存储从而提高读取效率，HBase使用了LSM树结构来存储数据。数据会先在Memstore中 整理成LSM树，最后再刷写到HFile上。

关于MemStore，很容易让人混淆。数据在被刷到HFile之前，已经被存储到了HDFS的WAL上了，那么为什么还要在放入MemStore呢?其实很简单，我们都知道HDFS是不能修改的，而HBase的数据又是按照Row Key进行排序的，其实这个排序的过程就是在MemStore中进行的。值得注意的是：MemStore的作用不是为了加快写速度，而是为了对Row Key进行排序。

* HFile

HFile是数据存储的实际载体，我们创建的所有表、列等数据都存储在HFile里面。当Memstore达到一定阀值，或者达到了刷写时间间隔阀值的时候，HBaes会被这个Memstore的内容刷写到HDFS系统上，称为一个存储在硬盘上的HFile文件。至此，我们数据真正地被持久化到硬盘上。

[大数据入门：Hbase存储原理解析 (baidu.com)](https://baijiahao.baidu.com/s?id=1685049327472302700&wfr=spider&for=pc)

[HBase底层存储结构和原理 - jingyi\_up - 博客园 (cnblogs.com)](https://www.cnblogs.com/jing-yi/p/15534775.html)

1. 幂等性的实现

[玩转微服务接口幂等性与安全设计\_51CTO博客\_微服务 幂等](https://blog.51cto.com/u_8238263/6022647)

[(51条消息) 【微服务】如何保证接口的幂等性\_微服务接口幂等性方案\_金陵吴彦祖的博客-CSDN博客](https://blog.csdn.net/playadota/article/details/126162016)

[什么是幂等性？四种接口幂等性方案详解！ - mikechen的互联网架构 - 博客园 (cnblogs.com)](https://www.cnblogs.com/mikechenshare/p/16571394.html)【措辞有些问题，需要使用分布式锁都没有讲解清楚，容易误导人】

1. **微服务治理及分布式框架**
2. [微服务体系下的服务治理 - 简书 (jianshu.com)](https://www.jianshu.com/p/a6d5c7421076)
3. [服务治理的工作内容\_微服务\_阿泽\_InfoQ写作社区](https://xie.infoq.cn/article/97c6a47a2368e9a67b5bcc5df)
4. [服务治理治什么，10张图告诉你答案\_监控 (sohu.com)](https://www.sohu.com/a/461069579_115128)
5. [分布式应用框架Akka详解 - java的涟漪 - ITeye博客](https://www.iteye.com/blog/luo-yifan-2035833)

# 1.[Actor——Actor模型原理的通俗理解 (转) - 会飞的斧头 - 博客园 (cnblogs.com)](https://www.cnblogs.com/xingchong/p/16202290.html)

# 2.[Akka 架构简介](https://www.jianshu.com/p/dc9e39a90d5b)

3.[分布式应用框架Akka详解 - java的涟漪 - ITeye博客](https://www.iteye.com/blog/luo-yifan-2035833)

4. [深入解析actor 模型（一)： actor 介绍及在游戏行业应用 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/427806717)

1. JVM内存模型【todo 多线程相关总结.md文件整合进来：对象结构等系列连载文档】

[JVM内存模型和结构详解(五大模型图解) - mikechen的互联网架构 - 博客园 (cnblogs.com)](https://www.cnblogs.com/mikechenshare/p/16562589.html)

[JVM内存模型，你看这一篇就够了 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/411021919)【分布图没上一篇的好】

[jvm~xmx设置多少合适 - 张占岭 - 博客园 (cnblogs.com)](https://www.cnblogs.com/lori/p/13344220.html)

[JVM 参数配置及详解 -Xms -Xmx -Xmn -Xss 调优总结（点赞收藏） - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/490965101)

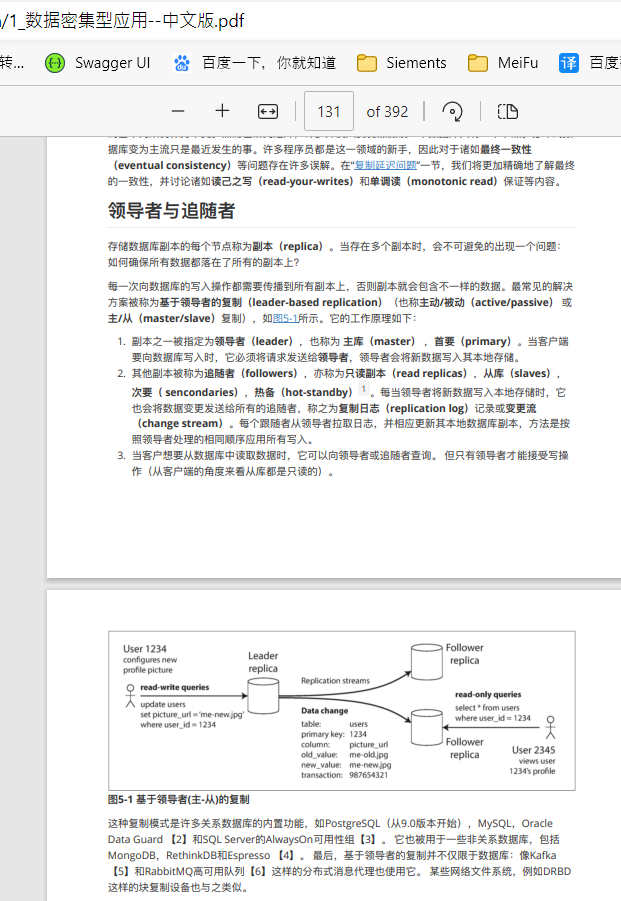
1. 分布式数据库：数据库高可用（水平/垂直扩展、复制/分区、集群）



**数据模型一文概要提炼：**

复制：

单变更 复制算法：领导者（single leader），多领导者（multi leader）和⽆领导者（leaderless）。⼏乎所有分布式数据库都使⽤这三种⽅法之⼀，



**主从结构（单主结构，选举算法）**

**同步复制与异步复制**

**半同步（semi-synchronous）：**保证你⾄少在两个节点上拥有最新的数据副 本：主库和同步从库。 这种配置有时也被称为

通常情况下，基于领导者的复制都配置为**完全异步**。 在这种情况下，如果主库失效且不可恢复，则任何尚未复制给从库的写⼊都会丢失。 这意味着即使已经向客户端确认成功，写⼊也不能保证持久

（Durable）。 然⽽，⼀个完全异步的配置也有优点：即使所有的从库都落后了，主库也可以继续处理写⼊。弱化的持久性可能听起来像是⼀个坏的折衷，⽆论如何，异步复制已经被⼴泛使⽤了，特别当有很多追随者，或追随者异地分布时。

**关于复制的研究**

对于异步复制系统⽽⾔，主库故障时有可能丢失数据。这可能是⼀个严重的问题，因此研究⼈员 仍在研究不丢数据但仍能提供良好性能和可⽤性的复制⽅法。 例如，链式复制【8,9】]是同步复 制的⼀种变体，已经在⼀些系统（如Microsoft Azure存储【10,11】）中成功实现。 复制的⼀致性与共识（consensus）（使⼏个节点就某个值达成⼀致）之间有着密切的联系

**拉取快照增量变更同步**

从库失效：追赶恢复

主库失效：故障切换



**复制日志的实现**

**基于语句的复制**

**传输预写式日志（WAL——Write Ahead Log）**

**逻辑⽇志复制（基于行）：**

复制和存储引擎使⽤不同的⽇志格式，这样可以使复制⽇志从存储引擎内部分离出来。

这种复制⽇志被称为逻辑⽇志，以将其与存储引擎的（物理）数据表示区分开来。

**基于触发器的复制**

**多主复制**

跨多个数据中⼼的多主复制

由于多主复制在许多数据库中都属于改装的功能，所以常常存在微妙的配置缺陷，且经常与其他数据库

功能之间出现意外的反应。例如⾃增主键、触发器、完整性约束等，都可能会有麻烦。因此，多主复制

往往被认为是危险的领域，应尽可能避免。

**无主复制**

单主复制、多主复制——都是这样的想法：客户端向⼀个

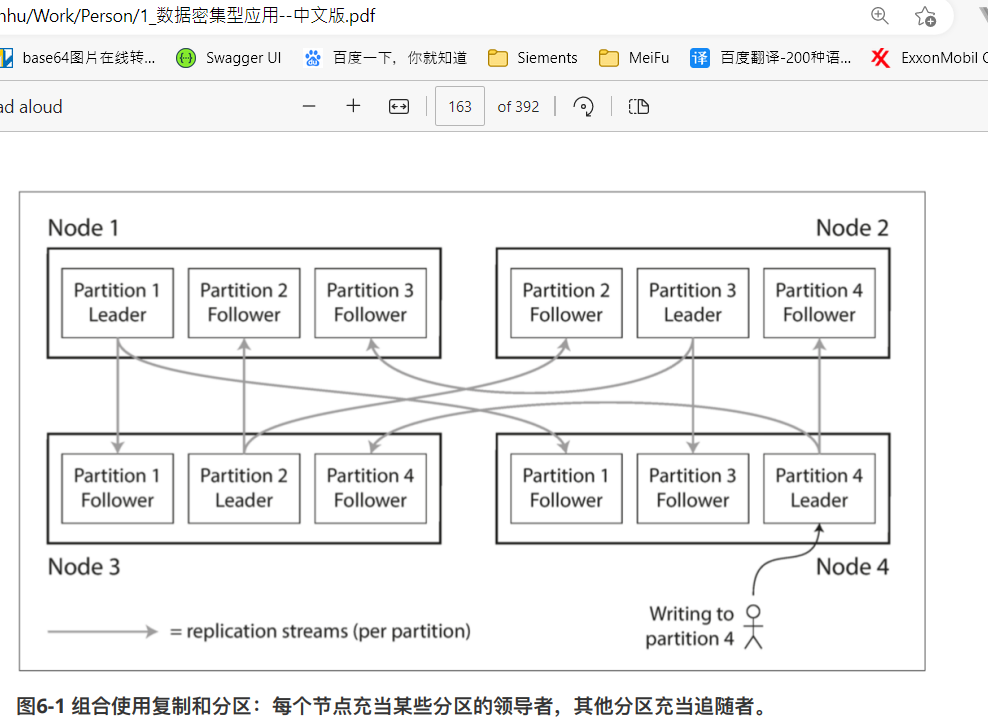
主库发送写请求，⽽数据库系统负责将写⼊复制到其他副本。主库决定写⼊的顺序，⽽从库按相同顺序

应⽤主库的写⼊。最早的⼀些的复制数据系统是⽆领导的（leaderless）【1,44】，但是在关系数据库主导的时代，这个想法几乎已被忘却。在亚马逊将其用于其内部的Dynamo系统 5 之后，它再一次成为数据库的一种时尚架构【37】。 Riak，Cassandra和Voldemort是由Dynamo启发的⽆领导复制模型的开源数据存储，所以这类数据库也被称为Dynamo风格。

1. **数据分区（partitions），也称为分片（sharding）**

分区主要是为了提升扩展性。分区数据库在20世纪80年代由Teradata和NonStop SQL【1】等产品率先推出，最近因为NoSQL数据库和基于Hadoop的数据仓库重新被关注。**有些系统是为事务性⼯作设计的，有些系统则⽤于分析（**参阅“[事务处理或分析]”）：这种差异会影响系统的运作⽅式，但是分区的基本原理均适⽤于这两种⼯作⽅式。

**分区与复制**



目标：平衡存储分布、加快检索速度

**键值数据的分区**

偏斜（skew）

热点（hot spot）

**根据键的范围分区**

分区边界可以由管理员手动选择，也可以由数据库自动选择（我们会在“重新平衡分区”中更详细地讨论分区边界的选择）。 Bigtable使用了这种分区策略，以及其开源等价物HBase 【2, 3】，RethinkDB和2.4版本之前的MongoDB 【4】。在每个分区中，我们可以按照一定的顺序保存键（参考“SSTables和LSM-树”）。好处是进行范围扫描非常简单，您可以将键作为联合索引来处理，以便在一次查询中获取多个相关记录（参阅“多列索引”）。

**根据键的散列分区**

这种技术擅⻓在分区之间分配键。分区边界可以是均匀间隔的，也可以是伪随机选择的（在这种情况下，该技术有时也被称为⼀致性哈希（consistent hashing）。

不幸的是，通过使⽤Key散列进⾏分区，我们失去了键范围分区的⼀个很好的属性：⾼效执⾏范围查询 的能⼒。曾经相邻的密钥现在分散在所有分区中，所以它们之间的顺序就丢失了。在MongoDB中，如 果您使⽤了基于散列的分区模式，则任何范围查询都必须发送到所有分区【4】。Riak 【9】， Couchbase 【10】或Voldemort不⽀持主键上的范围查询。 Cassandra采取了折衷的策略【11, 12, 13】。

**总结**

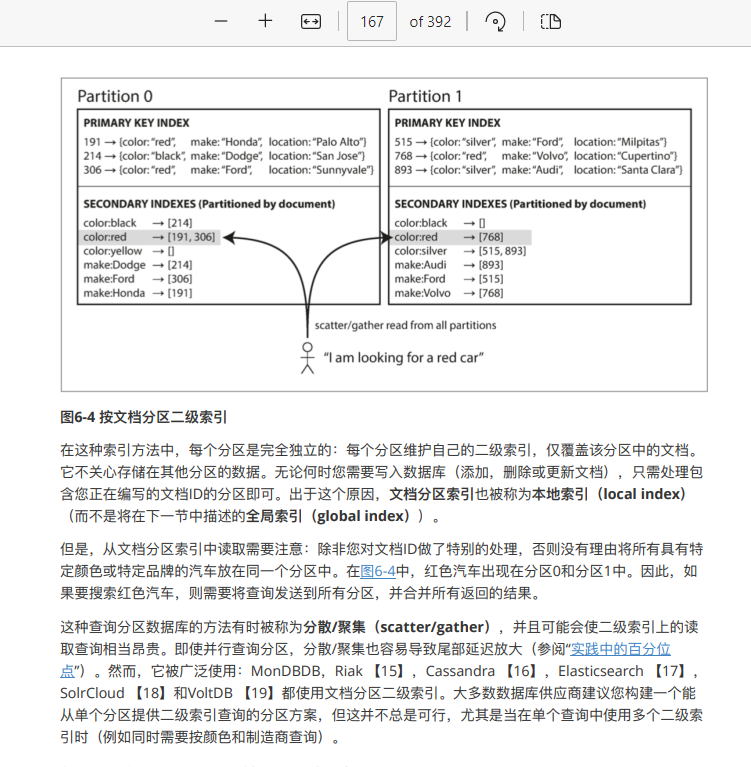
**负载倾斜与消除热点**

如前所述，哈希分区可以帮助减少热点。但是，它不能完全避免它们：在极端情况下，所有的读写操作都是针对同⼀个键的，所有的请求都会被路由到同⼀个分区。这种场景也许并不常⻅，但并⾮闻所未闻：例如，在社交媒体⽹站上，⼀个拥有数百万追随者的名⼈⽤户在做某事时可能会引发⼀场⻛暴【14】。这个事件可能导致⼤量写⼊同⼀个键（键可能是名⼈的⽤户ID，或者⼈们正在评论的动作的ID）。哈希策略不起作⽤，因为两个相同ID的哈希值仍然是相同的。

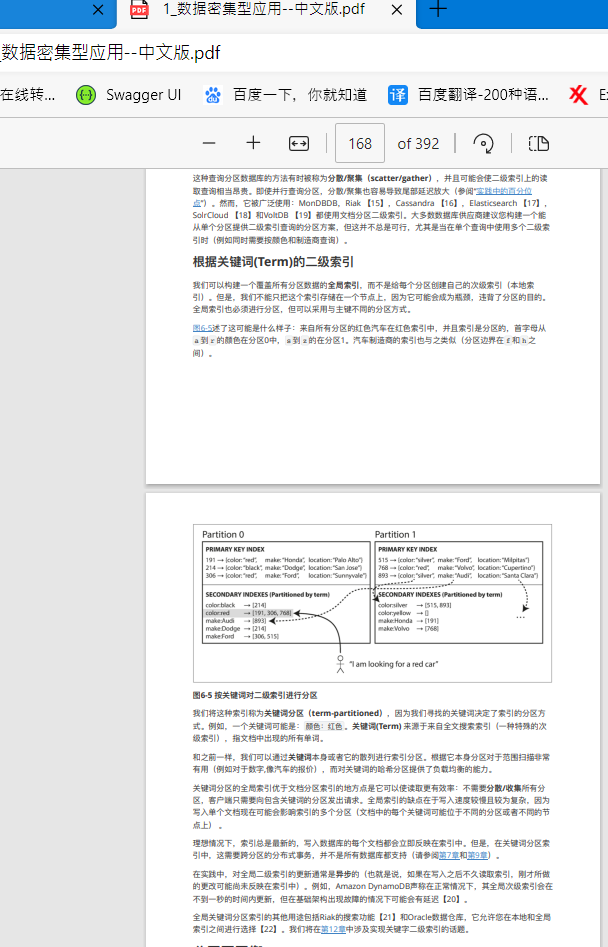
**分片与次级索引**

次级索引是关系型数据库的基础，并且在文档数据库中也很普遍。许多键值存储（如HBase和Voldemort）为了减少实现的复杂度而放弃了次级索引，但是一些（如Riak）已经开始添加它们，因为它们对于数据模型实在是太有用了。并且次级索引也是Solr和Elasticsearch等搜索服务器的基础。次级索引的问题是它们不能整齐地映射到分区。有两种用二级索引对数据库进⾏分区的方法：基于文档的分区（document-based）和基于关键词（term-based）的分区。

**按⽂档的二级索引**



**根据关键词(Term)的二级索引**



**分区再平衡**

**平衡策略**

**反面教材：hash mod N**

**固定数量的分区**

**动态分区**

**按节点比例分区**

**请求路由**

**本章小结**

**事务**

事务是应⽤程序将多个读写操作组 合成⼀个逻辑单元的⼀种⽅式。从概念上讲，事务中的所有读写操作被视作单个操作来执⾏：整个事务 要么成功（提交（commit））要么失败（中⽌（abort），回滚（rollback））。如果失败，应⽤程 序可以安全地重试。对于事务来说，应⽤程序的错误处理变得简单多了，因为它不⽤再担⼼部分失败的 情况了，即某些操作成功，某些失败（⽆论出于何种原因）。

ACID的含义：

**原子性（Atomicity）**：能够在错误时中止事务，丢弃该事务进行的所有写入变更的能力。 或许可中止性（abortability） 是更好的术语，但本书将继续使用原子性，因为这是惯用词。

**一致性（Consistency）**：对数据的一组特定陈述必须始终成立。

**隔离性（Isolation）：**同时执⾏的事务是相互隔离的。

**持久性（Durability）：**持久性是一个承诺，即一旦事务成功完成，即使发生硬件故障或数据库崩溃，写入的任何数据也不会丢失。完美的持久性是不存在的 ：如果所有硬盘和所有备份同时被销毁，那显然没有任何数据库能救得了你。

****

****

**弱隔离级别**

**读已提交**

**没有脏读**

**没有脏写**

**实现读已提交：**行锁（row-level lock） 来防止脏写：当事务想要修改特定对象（行或文档）时，它必须首先获得该对象的锁。然后必须持有该锁直到事务被提交或中止。一次只有一个事务可持有任何给定对象的锁；如果另一个事务要写入同一个对象，则必须等到第一个事务提交或中止后，才能获取该锁并继续。这种锁定是读已提交模式（或更强的隔离级别）的数据库自动完成的。

**快照隔离和可重复读**

**防止丢失更新**

**本章小结**

**分布式系统的麻烦**

**一致性与共识**

在“复制延迟问题”中，我们看到了数据库复制中发⽣的⼀些时序问题。如果你在同⼀时刻查看两个数据

库节点，则可能在两个节点上看到不同的数据，因为写请求在不同的时间到达不同的节点。⽆论数据库

使⽤何种复制⽅法（单主复制，多主复制或⽆主复制），都会出现这些不⼀致情况。

⼤多数复制的数据库⾄少提供了最终⼀致性，这意味着如果你停⽌向数据库写⼊数据并等待⼀段不确定

的时间，那么最终所有的读取请求都会返回相同的值【1】。换句话说，不⼀致性是暂时的，最终会⾃

⾏解决（假设⽹络中的任何故障最终都会被修复）。最终⼀致性的⼀个更好的名字可能是收敛

（convergence），因为我们预计所有的复本最终会收敛到相同的值【2】。

然⽽，这是⼀个⾮常弱的保证 —— 它并没有说什么什么时候副本会收敛。在收敛之前，读操作可能会

返回任何东⻄或什么都没有【1】。

**线性一致性**

**一致性 & 性能 & 可用性折衷**

**顺序保证**

**序列号顺序：**有一个更好的办法：我们可以使用序列号（sequence nunber）或时间戳（timestamp）来排序

事件。时间戳不一定来自时钟（或物理时钟，存在许多问题，如 “不可靠时钟” 中所述）。它可以来自一

个逻辑时钟（logical clock），这是一个用来生成标识操作的数字序列的算法，典型实现是使用一个每

次操作自增的计数器。

**兰伯特时间戳**

**全序广播：**单主复制通过选择一个节点作为主库来确定操作的全序，并在主库的单个CPU核上对所有操

作进行排序。接下来的挑战是，如果吞吐量超出单个主库的处理能力，这种情况下如何扩展系统；以及，如果主库失效（“处理节点宕机”），如何处理故障切换。在分布式系统文献中，这个问题被称为全序广播（total order broadcast）或原子广播（atomic broadcast）

**可靠交付（reliable delivery）：**没有消息丢失，如果消息被传递到⼀个节点，它将被传递到所有节点。

**全序交付（totally ordered delivery）：**消息以相同的顺序传递给每个节点。正确的全序⼴播算法必须始终保证可靠性和有序性，即使节点或⽹络出现故障。当然在⽹络中断的时候，消息是传不出去的，但是算法可以不断重试，以便在⽹络最终修复时，消息能及时通过并送达（当然它们必须仍然按照正确的顺序传递）。

**使用全序广播实现线性一致的存储**

**使用线性一致性存储实现全序广播**

**分布式事务与共识**

**领导选举**

**原⼦提交:** 我们将讨论两阶段提交（2PC, twophase commit）算法，这是解决原⼦提交问题最常⻅的办法，并在各种数据库、消息队列和应⽤服务 器中实现。事实证明2PC是⼀种共识算法，但不是⼀个⾮常好的算法【70,71】。 通过对2PC的学习。

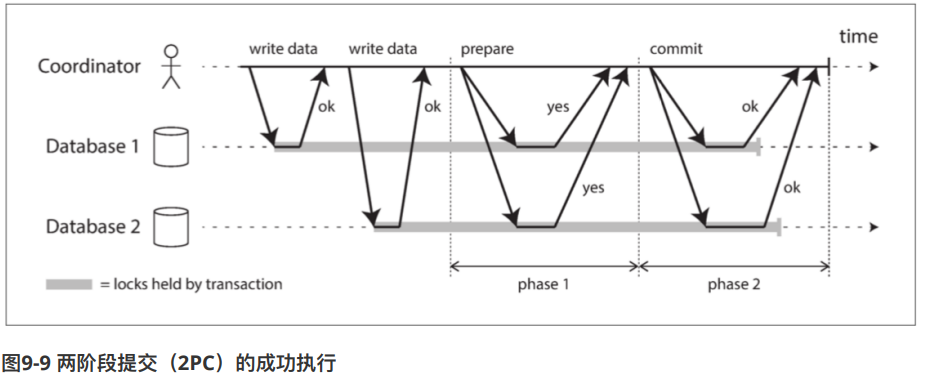
**两阶段提交简介** 两阶段提交（two-phase commit）是⼀种⽤于实现跨多个节点的原⼦事务提交的算法，即确保所有节 点提交或所有节点中⽌。 它是分布式数据库中的经典算法【13,35,75】。 2PC在某些数据库内部使⽤， 也以XA事务的形式对应⽤可⽤【76,77】（例如**Java Transaction API⽀持**）或以SOAP Web服务的 WSAtomicTransaction 形式提供给应⽤【78,79】。

图9-9说明了2PC的基本流程。2PC中的提交/中⽌过程分为两个阶段（因此⽽得名），⽽不是单节点事 务中的单个提交请求。 图9-9 两阶段提交（2PC）的成功执⾏。

**不要把2PC和2PL搞混了** 两阶段提交（2PC）和两阶段锁定（参阅“两阶段锁定（2PL）”）是两个完全不同的东⻄。 2PC在 分布式数据库中提供原⼦提交，⽽2PL提供可序列化的隔离等级。为了避免混淆，最好把它们看作 完全ᇿ⽴的概念，并忽略名称中不幸的相似性。

**2PC使⽤⼀个通常不会出现在单节点事务中的新组件：**协调者（coordinator）（也称为事务管理器 （transaction manager））。协调者通常在请求事务的相同应⽤进程中以库的形式实现（例如，嵌 ⼊在Java EE容器中），但也可以是单ᇿ的进程或服务。这种协调者的例⼦包括Narayana，JOTM，BTM 或MSDTC。

正常情况下，2PC事务以应⽤在多个数据库节点上读写数据开始。我们称这些数据库节点为参与者 （participants）。当应⽤准备提交时，协调者开始阶段 1 ：它发送⼀个准备（prepare）请求到每个 节点，询问它们是否能够提交。然后协调者会跟踪参与者的响应： 如果所有参与者都回答“是”，表示它们已经准备好提交，那么协调者在阶段 2 发出提交 （commit）请求，然后提交真正发⽣。 如果任意⼀个参与者回复了“否”，则协调者在阶段2 中向所有节点发送中⽌（abort）请求。 这个过程有点像⻄⽅传统婚姻仪式：司仪分别询问新娘和新郎是否要结婚，通常是从两⽅都收到“我愿 意”的答复。收到两者的回复后，司仪宣布这对情侣成为夫妻：事务就提交了，这⼀幸福事实会⼴播⾄所 有的参与者中。如果新娘与新郎之⼀没有回复”我愿意“，婚礼就会中⽌【73】

**三阶段提交** 两阶段提交被称为阻塞（blocking）原⼦提交协议，因为存在2PC可能卡住并等待协调者恢复的情况。 理论上，可以使⼀个原⼦提交协议变为⾮阻塞（nonblocking）的，以便在节点失败时不会卡住。但是 让这个协议能在实践中⼯作并没有那么简单。

**实践中的分布式事务**

**数据库内部的分布式事务** ⼀些分布式数据库（即在其标准配置中使⽤复制和分区的数据库）⽀持数据库节点之间的内部事务。例 如，VoltDB和MySQL Cluster的NDB存储引擎就有这样的内部事务⽀持。在这种情况下，所有参与事务 的节点都运⾏相同的数据库软件。

**异构分布式事务** 在异构（heterogeneous）事务中，参与者是两种或以上不同技术：例如来⾃不同供应商的两个数据 库，甚⾄是⾮数据库系统（如消息代理）。跨系统的分布式事务必须确保原⼦提交，尽管系统可能完全 不同。 数据库内部事务不必与任何其他系统兼容，因此它们可以使⽤任何协议，并能针对特定技术进⾏特定的 优化。因此数据库内部的分布式事务通常⼯作地很好。另⼀⽅⾯，跨异构技术的事务则更有挑战性。

**容错共识**

**成员与协调服务**

**本章⼩结**

**批处理**

**使用Unix⼯工具的批处理**

**MapReduce和分布式⽂文件系统**：MapReduce作业在分布式⽂文件系统上读写⽂文

件。在Hadoop的Map-Reduce实现中，该⽂文件系统被称为**HDFS**（**Hadoop**分布式⽂文件系统），⼀一个

Google⽂文件系统（GFS）的开源实现【19】。

除HDFS外，还有各种其他分布式⽂件系统，如GlusterFS和Quantcast File System（QFS）【20】。诸

如Amazon S3，Azure Blob存储和OpenStack Swift 【21】等对象存储服务在很多⽅方⾯面都是相似的5 。在本章中，我们将主要使⽤用HDFS作为示例例，但是这些原则适用于任何分布式文件系统。与⽹络连接存储（NAS）和存储区域⽹网络（SAN）架构的共享磁盘⽅方法相⽐比，HDFS基于⽆无共享原则（参⻅见第⼆二部分前⾔言）。共享磁盘存储由集中式存储设备实现，通常使⽤用定制硬件和专⽤用⽹网络基础设施（如光纤通道）。⽽而另⼀一⽅方⾯面，⽆无共享⽅方法不不需要特殊的硬件，只需要通过传统数据中心网络连接的计算机。HDFS包含在每台机器上运⾏行行的守护进程，对外暴暴露露⽹网络服务，允许其他节点访问存储在该机器上的⽂件（假设数据中心中的每台通⽤用计算机都挂载着⼀一些磁盘）。名为**NameNode**的中央服务器会跟踪哪个⽂文件块存储在哪台机器上（DataNode）。因此，HDFS在概念上创建了了⼀一个⼤大型⽂文件系统，可以使⽤用所有运⾏行行有守护进程的机器的磁盘。

**过程：**

**MapReduce作业执行、**

**Reduce端连接和分组、**

**GROUP BY、**

**Map端连接、**

**批处理工作流的输出:**

建立搜搜索引: Google最初使⽤用MapReduce是为其搜索引擎建⽴立索引，⽤用了了由5到10个MapReduce作业组成的⼯工作流

实现【1】。虽然Google后来也不不仅仅是为这个⽬目的⽽而使⽤用MapReduce 【43】，但如果从构建搜索索

引的⻆角度来看，更更能帮助理理解MapReduce。 （直⾄至今⽇日，Hadoop MapReduce仍然是为Lucene/Solr

构建索引的好⽅方法【44】）我们在“全⽂文搜索和模糊索引”中简要地了了解了了Lucene这样的全⽂文搜索索引是如何⼯工作的：它是⼀一个⽂文件（关键词字典），你可以在其中⾼高效地查找特定关键字，并找到包含该关键字的所有⽂文档ID列列表（⽂文章

列列表）。这是⼀一种⾮非常简化的看法 —— 实际上，搜索索引需要各种额外数据，以便便根据相关性对搜索

结果进⾏行行排名，纠正拼写错误，解析同义词等等 —— 但这个原则是成⽴立的。如果需要对⼀一组固定⽂文档执⾏行行全⽂文搜索，则批处理理是⼀一种构建索引的⾼高效⽅方法：Mapper根据需要对⽂文档集合进⾏行行分区，每个Reducer构建该分区的索引，并将索引⽂文件写⼊入分布式⽂文件系统。构建这样的⽂文档分区索引（参阅“分区和⼆二级索引”）并⾏行行处理理效果拔群。由于按关键字查询搜索索引是只读操作，因⽽而这些索引⽂文件⼀一旦创建就是不不可变的。如果索引的⽂文档集合发⽣生更更改，⼀一种选择是定期重跑整个索引⼯工作流，并在完成后⽤用新的索引⽂文件批量量替换以前的索引⽂文件。如果只有少量量的⽂文档发⽣生了了变化，这种⽅方法的计算成本可能会很⾼高。但它的优点

是索引过程很容易易理理解：⽂文档进，索引出。另⼀一个选择是，可以增量量建⽴立索引。如第3章中讨论的，如果要在索引中添加，删除或更更新⽂文档，Lucene会写新的段⽂文件，并在后台异步合并压缩段⽂文件。我们将在第11章中看到更更多这种增量量处理理。

**键值存储作为批处理输出:**

**批处理输出的哲学:**

**Hadoop与分布式数据库的对⽐:** 正如我们所看到的，Hadoop有点像Unix的分布式版本，其中HDFS是文件系统，而MapReduce是Unix

进程的怪异实现（总是在Map阶段和Reduce阶段运⾏行sort工具）。我们了解了如何在这些原语的基础上实现各种连接和分组操作。

当MapReduce论文发表时【1】，它从某种意义上来说 —— 并不新鲜。我们在前⼏节中讨论的所有处理和并⾏连接算法已经在十多年前所谓的大规模并行处理理（**MPP**， **massively parallel processing**）数据库中实现了【3,40】。比如Gamma database machine，Teradata和Tandem NonStop SQL就是这⽅面的先驱【52】。最大的区别是，MPP数据库专注于在一组机器上并⾏执⾏分析SQL查询，而MapReduce和分布式⽂件系统【19】的组合则更像是一个可以运⾏任意程序的通⽤操作系统。

**存储多样性:**数据库要求你根据特定的模型（例例如关系或⽂文档）来构造数据，⽽而分布式⽂文件系统中的⽂文件只是字节序列列，可以使⽤用任何数据模型和编码来编写。它们可能是数据库记录的集合，但同样可以是⽂文本，图像，视频，传感器器读数，稀疏矩阵，特征向量量，基因组序列列或任何其他类型的数据。

说白了，Hadoop开放了了将数据不不加区分地转储到HDFS的可能性，允许后续再研究如何进⼀一步处理理【53】。相⽐比之下，在将数据导入数据库专有存储格式之前，MPP数据库通常需要对数据和查询模式进行仔细的前期建模。

在纯粹主义者看来，这种仔细的建模和导⼊入似乎是可取的，因为这意味着数据库的⽤用户有更高质量的数

据来处理。然⽽实践经验表明，简单地使数据快速可⽤ —— 即使它很古怪，难以使⽤用，使用原始格式

—— 也通常要比事先决定理想数据模型要更有价值【54】。

这个想法与数据仓库类似（参阅“数据仓库”）：将大型组织的各个部分的数据集中在一起是很有价值

的，因为它可以跨越以前相分离的数据集进⾏连接。 MPP数据库所要求的谨慎模式设计拖慢了集中式数

据收集速度；以原始形式收集数据，稍后再操⼼模式的设计，能使数据收集速度加快（有时被称为“数据

湖（**data lake**）”或“企业数据中⼼心（**enterprise data hub**）”【55】）。不加区分的数据转储转移了解释数据的负担：数据集的⽣产者不再需要强制将其转化为标准格式，数据的解释成为消费者的问题（读时模式⽅方法【56】；参阅“⽂文档模型中的架构灵活性”）。如果⽣生产者和消费者是不不同优先级的不不同团队，这可能是⼀一种优势。甚⾄至可能不存在⼀一个理理想的数据模型，对于不同⽬目

的有不同的合适视⻆角。以原始形式简单地转储数据，可以允许多种这样的转换。这种⽅方法被称为寿司原

则（**sushi principle**）：“原始数据更更好”【57】。

因此，Hadoop经常被⽤于实现ETL过程（参阅“数据仓库”）：事务处理理系统中的数据以某种原始形式转

储到分布式文件系统中，然后编写MapReduce作业来清理理数据，将其转换为关系形式，并将其导⼊

MPP数据仓库以进⾏行行分析。数据建模仍然在进⾏行行，但它在⼀一个单独的步骤中进⾏，与数据收集相解耦。

这种解耦是可⾏的，因为分布式⽂件系统⽀持以任何格式编码的数据。

**处理模型多样性:** SQL查询语⾔允许以优雅的语法表达查询，⽽无需编写代码，使业务分析师用来做商业分析的可视化工具（例如Tableau）能够访问。另一⽅面，并⾮所有类型的处理都可以合理地表达为SQL查询。例如，如果要构建机器学习和推荐系统，或者使⽤关性排名模型的全文搜索引，或者执⾏图像分析，则很可能需要更一般的数据处理模型。这些类型的处理通常是特别针对特定应用的（例如机器学习的特征工程，机器翻译的⾃自然语⾔模型，欺诈预测的⻛险评估函数），因此它们不可避免地需要编写代码，而不仅仅是查询。

Hadoop⽣生态系统包括随机访问的OLTP数据库，如HBase（参阅“SSTables和LSM树”）和MPP⻛风格的分析型数据库，如Impala 【41】。 HBase与Impala都不不使⽤用MapReduce，但都使⽤用HDFS进⾏行行存储。它们是迥异的数据访问与处理理⽅方法，但是它们可以共存，并被集成到同⼀一个系统中。

**MapReduce之后**

**物化中间状态：**数据流引擎、容错、关于物化的讨论

**图与迭代处理：**

**Pregel处理理模型：**针对图批处理理的优化 —— 批量量同步并⾏行行（**BSP**）计算模型【70】已经开始流⾏行行起来。其中，Apache Giraph 【37】，Spark的GraphX API和Flink的Gelly API 【71】实现了了它。它也被称为**Pregel**模型，因为Google的Pregel论⽂文推⼴广了了这种处理理图的⽅方法【72】。回想⼀一下在MapReduce中，Mapper在概念上向Reducer的特定调⽤用“发送消息”，因为框架将所有具有相同键的Mapper输出集中在⼀一起。 Pregel背后有⼀一个类似的想法：⼀一个顶点可以向另⼀一个顶点“发送消息”，通常这些消息是沿着图的边发送的。在每次迭代中，为每个顶点调⽤用⼀一个函数，将所有发送给它的消息传递给它 —— 就像调⽤用Reducer⼀一样。与MapReduce的不不同之处在于，在Pregel模型中，顶点在⼀一次迭代到下⼀一次迭代的过程中会记住它的状态，所以这个函数只需要处理理新的传⼊入消息。如果图的某个部分没有被发送消息，那⾥就不需要做任何工作。这与Actor模型有些相似（参阅“分布式的Actor框架”——分布式事务的拆解方案：并行分布式计算框架，线性“事务”），除了顶点状态和顶点之间的消息具有容错性和耐久性，且通信以固定的⽅方式进⾏行行：在每次迭代中，框架递送上次迭代中发送的所有消息。Actor通常没有这样的时间保证。

**高级API和语⾔**

**向声明式查询语⾔言的转变**

**本章小结**

**流处理**

在第10章中，我们讨论了了批处理理技术，它读取⼀一组⽂文件作为输⼊入，并⽣生成⼀一组新的⽂文件作为输出。输出是衍⽣生数据（**derived data**）的⼀一种形式；也就是说，如果需要，可以通过再次运⾏行行批处理理过程来重新创建数据集。我们看到了了如何使⽤用这个简单⽽而强⼤大的想法来建⽴立搜索索引，推荐系统，做分析等等。然而，在第10章中仍然有⼀一个很⼤大的假设：即输⼊入是有界的，即已知和有限的⼤大⼩小，所以批处理知道它何时完成输入的读取。例例如，MapReduce核⼼心的排序操作必须读取其全部输⼊入，然后才能开始生成输出：可能发⽣生这种情况：最后⼀一条输⼊入记录具有最⼩小的键，因此需要第⼀一个被输出，所以提早开始输出是不可行的。实际上，很多数据是⽆无界限的，因为它随着时间的推移⽽而逐渐到达：你的⽤用户在昨天和今天产生了数据，明天他们将继续产⽣生更更多的数据。除⾮非你停业，否则这个过程永远都不不会结束，所以数据集从来就不会以任何有意义的⽅方式“完成”【1】。因此，批处理理程序必须将数据⼈人为地分成固定时间段的数据块，例例如，在每天结束时处理理⼀一天的数据，或者在每⼩小时结束时处理理⼀一⼩小时的数据。

日常批处理理中的问题是，输⼊入的变更更只会在⼀一天之后的输出中反映出来，这对于许多急躁的⽤用户来说太慢了了。为了了减少延迟，我们可以更更频繁地运⾏行行处理理 —— ⽐比如说，在每秒钟的末尾 —— 或者甚⾄至更更连续⼀一些，完全抛开固定的时间切⽚片，当事件发⽣生时就⽴立即进⾏行行处理理，这就是流处理理（**stream processing**）背后的想法。

**传递事件流**

**消息系统：**

**直接从⽣产者传递给消费者、**

**消息代理理、**

**消息代理理与数据库对⽐：**

**多个消费者**

**确认与重新交付**

**分区日志：**

**使用日志进行消息存储**

**日志与传统消息相比**

**消费者偏移量：**顺序消费⼀一个分区使得判断消息是否已经被处理理变得相当容易易：所有偏移量量⼩小于消费者的当前偏移量量的消息已经被处理理，⽽而具有更更⼤大偏移量量的消息还没有被看到。因此，代理理不不需要跟踪确认每条消息，只需要定期记录消费者的偏移即可。在这种⽅方法减少了了额外簿记开销，⽽而且在批处理理和流处理理中采⽤用这种⽅方法有助于提⾼高基于⽇日志的系统的吞吐量量。

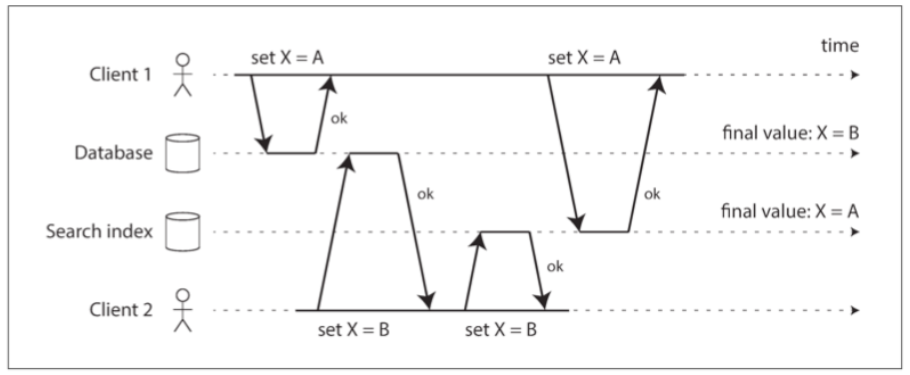
**磁盘空间使用：**在“消息传递系统”中，如果消费者⽆无法跟上⽣生产者发送信息的速度时，我们讨论了了三种选择：丢弃信息，进⾏行行缓冲或施加背压。在这种分类法⾥里里，基于⽇日志的⽅方法是缓冲的⼀一种形式，具有很⼤大，但⼤大⼩小固定的缓冲区（受可⽤用磁盘空间的限制）。

**当消费者跟不上生产者时**

**重播旧信息：**

**流与数据库**

**保持系统同步：**正如我们在本书中所看到的，没有一个系统能够满⾜所有的数据存储，查询和处理需求。在实践中，⼤多数重要应用都需要组合使⽤几种不同的技术来满⾜所有的需求：例如，使⽤OLTP数据库来为用户请求提供服务，使用缓存来加速常见请求，使⽤全文索引搜索处理搜索查询，使用数据仓库用于分析。每一个组件都有⾃己的数据副本，以⾃己的表示存储，并根据⾃己的目的进⾏优化。由于相同或相关的数据出现在了不同的地⽅，因此相互间需要保持同步：如果某个项⽬在数据库中被更更新，它也应当在缓存，搜索索引和数据仓库中被更新。对于数据仓库，这种同步通常由ETL进程执行（参见“数据仓库”），通常是先取得数据库的完整副本，然后执行转换，并批量量加载到数据仓库中 ——换句话说，批处理。我们在“批量工作流的输出”中同样看到了如何使用批处理创建搜索引，推荐系统和其他衍⽣数据系统。如果周期性的完整数据库转储过于缓慢，有时会使用的替代方法是双写（**dual write**），其中应用代码在数据变更时明确写⼊每个系统：



但是，双写有⼀些严重的问题，其中一个是竞争条件，如图11-4所示。在这个例子中，两个客户端同时想要更新⼀个项⽬目X：客户端1想要将值设置为A，客户端2想要将其设置为B。两个客户端⾸先将新值写入数据库，然后将其写入到搜索引。因为运⽓气不好，这些请求的时序是交错的：数据库⾸先看到来自客户端1的写入将值设置为A，然后来⾃自客户端2的写入将值设置为B，因此数据库中的最终值为B。搜索索引首先看到来⾃自客户端2的写⼊入，然后是客户端1的写⼊入，所以搜索索引中的最终值是A。即使没发⽣错误，这两个系统现在也永久地不⼀致了了。

图**11-4** 在数据库中**X**⾸首先被设置为**A**，然后被设置为**B**，⽽在搜索索引处，写⼊入以相反的顺序到达除⾮非有⼀一些额外的并发检测机制，例例如我们在“检测并发写⼊入”中讨论的版本向量量，否则你甚⾄至不不会意识到发⽣生了了并发写⼊一个值将简单地以⽆无提示⽅方式覆盖另一个值。双重写⼊入的另一个问题是，其中一个写⼊入可能会失败，⽽另⼀个成功。这是⼀个容错问题，⽽不是⼀个并发问题，但也会造成两个系统互相不一致的结果。确保它们要么都成功要么都失败，是原⼦提交问题的一个例子，解决这个问题的代价是昂贵的（参阅“原⼦子提交和两阶段提交（2PC）”）。如果你只有⼀个单领导者复制的数据库，那么这个领导者决定了了写⼊入顺序，⽽状态机复制⽅方法可以在数据库副本上⼯作。然⽽而，在图11-4中，没有单个主库：数据库可能有一个领导者，搜索索引也可能有⼀个领导者，但是两者都不不追随对⽅方，所以可能会发生冲突（参⻅见“多领导者复制“）。如果实际上只有⼀个领导者 —— 例例如，数据库 —— 而且我们能让搜索索引成为数据库的追随者，情况要好得多。但这在实践中可能吗？

**变更数据捕获：**变更更数据捕获的实现、初始快照、⽇日志压缩、变更更流的**API**⽀支持

**事件溯源：**我们在这里讨论的想法和事件溯源（ **Event Sourcing**）之间有⼀些相似之处，这是⼀个在领域驱动设计（**domain-driven design, DDD**）社区中折腾出来的技术。

**从事件⽇日志中派⽣生出当前状态**

**命令和事件**

**状态，流和不变性**

**不可变事件的优点**

**从同一时间日志中派生多个视图**

**并发控制：**

事件溯源和变更数据捕获的最大缺点是，事件日志的消费者通常是异步的，所以可能会出现这样的情况：⽤户会写⼊日志，然后从日志衍生视图中读取，结果发现他的写⼊还没有反映在读取视图中。我们之前在“读己之写”中讨论了这个问题以及可能的解决方案。

⼀种解决方案是将事件附加到日志时同步执行读取视图的更更新。⽽将这些写入操作合并为⼀个原⼦单元需要事务，所以要么将事件日志和读取视图保存在同一个存储系统中，要么就需要跨不同系统进行分布式事务。或者，你也可以使用在“使⽤全序⼴广播实现线性化存储”中讨论的方法。另⼀方面，从事件日志导出当前状态也简化了并发控制的某些部分。许多对于多对象事务的需求（参阅“单对象和多对象操作”）源于单个⽤户操作需要在多个不同的位置更改数据。通过事件溯源，你可以设计⼀个⾃包含的事件以表示一个⽤用户操作。然后⽤户操作就只需要在一个地方进行单次写⼊入操作 ——即将事件附加到⽇志中 —— 这个还是很容易使原⼦化的。如果事件日志与应⽤状态以相同的⽅、方式分区（例如，处理分区3中的客户事件只需要更新分区3中的应用状态），那么直接使用单线程日志消费者就不需要写入并发控制了。它从设计上一次只处理⼀个事件（参阅“真的串⾏执行”）。日志通过在分区中定义事件的序列列顺序，消除了并发性的不确定性【24】。如果⼀一个事件触及多个状态分区，那么需要做更多的工作，我们将在第12章讨论。

**不变的限制**

**流处理：**

到目前为⽌，本章中我们已经讨论了流的来源（⽤户活动事件，传感器器和写入数据库），我们讨论了流如何传输（直接通过消息传送，通过消息代理，通过事件日志）。剩下的就是讨论一下你可以⽤流做什么 —— 也就是说，你可以处理它。一般来说，有三种选项：

**1.** 你可以将事件中的数据写⼊数据库，缓存，搜索引或类似的存储系统，然后能被其他客户端查询。如图11-5所示，这是数据库与系统其他部分发⽣变更保持同步的好方法 —— 特别是当流消费者是写入数据库的唯一客户端时。如“批处理工作流的输出”中所讨论的，它是写⼊存储系统的流等价物。

**2.** 你能以某种方式将事件推送给⽤用户，例如发送报警邮件或推送通知，或将事件流式传输到可实时显示的仪表板上。在这种情况下，⼈是流的最终消费者。

**3.** 你可以处理一个或多个输入流，并产生⼀个或多个输出流。流可能会经过由几个这样的处理理阶段组成的流水线，最后再输出（选项1或2）。

在本章的剩余部分中，我们将讨论选项3：处理理流以产⽣其他衍生流。处理理这样的流的代码⽚段，被称为算⼦（**operator**）或作业（**job**）。它与我们在第10章中讨论过的Unix进程和MapReduce作业密切相关，数据流的模式是相似的：一个流处理理器以只读的⽅方式使⽤用输⼊流，并将其输出以仅追加的⽅式写⼊一个不同的位置。流处理中的分区和并行化模式也⾮常类似于第10章中介绍的MapReduce和数据流引擎，因此我们不再重复这些主题。基本的Map操作（如转换和过滤记录）也是一样的。与批量作业相⽐比的⼀个关键区别是，流不会结束。这种差异会带来很多隐含的结果。正如本章开始部分所讨论的，排序对无界数据集没有意义，因此无法使用排序合并联接（请参阅“Reduce端连接与分组”）。

**容错机制也必须改变：**对于已经运⾏了几分钟的批处理作业，可以简单地从头开始重启失败任务，但是对于已经运行数年的流作业，重启后从头开始跑可能并不是一个可行的选项。

**流处理的应⽤：**符合事件处理、流分析、维护**物化视图**、在流上搜索、消息传递和RPC

**时间推理：**事件时间与处理时间、知道什么时候准备好了、你用的是谁的时钟、窗口的类型

**流式连接：**流流连接（窗口连接）、流表连接（流扩展）、表表连接（维护物化视图）、连接的时间依赖性

**容错：**微批量与存档点、原子提交再现、幂等性、失败后重建状态

**本章小结**

**数据系统的未来**

**数据集成**

**分拆数据库**

**将事情做正确**

**为数据库使用端到端的参数：正好执行一次操作、抑制重复、操作标识符、在数据系统中应用端到端思考**

**强制约束：**

**唯一性约束需要达成共识**

**基于日志消息传递中的唯一性：**

日志确保所有消费者以相同的顺序看见消息——这种保证在形式上被称为**“全序广播”（total ordre boardcast）**并且等价于共识（参见“全序广播”）。在使用基于日志的消息传递的分拆数据库方法中，我们可以使用非常类似的方法来执行唯一约束。

流处理器在单线程上一次消费单个日志分区中的所有消息（参阅“与传统消息传递相比的日志”）。因此，如果日志是按有待确保唯一的值做的分区，则流处理器可以无歧义地，确定性地决定几个冲突操作中的哪一个先。例如，在多个用户尝试宣告相同用户名的情况下：

1. 每个对用户名的请求都被编码为一条消息，并追加到按用户名散列值确定的分区。
2. 流处理器依序读取日志中的请求，并使用本地数据库来追踪哪些用户名已经被占用了。对于所有申请可用用户名的请求，它都会记录该用户名，并向输出流发送一条成功消息。对于所有申请已占用用户名的请求，它都会向输出流发送一条拒绝消息。
3. 请求用户名的客户端监视输出流，等待与其请求相对应的成功或拒绝消息。

该算法基本上与“**使用全序广播实现先行一致性的存储**”中的算法相同。它可以简单地通过增加分区数扩展至较大的请求吞吐量，因为每个分区可以被独立处理。该方法不仅适用于唯一性约束，而且适用于许多其他类型的约束。其基本原理是，任何可能冲突的写入都会路由到相同分区并按顺序处理。正如“什么是冲突”与“写入偏差与幻读”中所述，冲突的定义可能取决于应用，但流处理器可以使用任意逻辑来验证请求。这个想法与Bayou在90年代开创的方法类似。

**多分区请求处理**

当涉及多个分区时，确保操作以原⼦⽅式执⾏且同时满⾜约束就变得很有趣了。在例12-2中，可能有三 个分区：⼀个包含请求ID，⼀个包含收款⼈账户，另⼀个包含付款⼈账户。没有理由把这三种东⻄放⼊ 同⼀个分区，因为它们都是相互独⽴的。 在数据库的传统⽅法中，执⾏此事务需要跨全部三个分区进⾏原⼦提交，这实质上是将该事务嵌⼊⼀个 全序，就这些分区上的所有其他事务⽽⾔。⽽这样就要求跨分区协调，不同的分区⽆法再独⽴地进⾏处 理，因此吞吐量可能会受到影响。 但事实证明，使⽤分区⽇志可以达到等价的正确性⽽⽆需原⼦提交： 1. 从账户A向账户B转账的请求由客户端提供⼀个唯⼀的请求ID，并按请求ID追加写⼊相应⽇志分 区。 2. 流处理器读取请求⽇志。对于每个请求消息，它向输出流发出两条消息：付款⼈账户A的借记指令 （按A分区），收款⼈B的贷记指令（按B分区）。被发出的消息中会带有原始的请求ID。 3. 后续处理器消费借记/贷记指令流，按照请求ID除重，并将变更应⽤⾄账户余额。 步骤1和步骤2是必要的，因为如果客户直接发送贷记与借记指令，则需要在这两个分区之间进⾏原⼦提 交，以确保两者要么都发⽣或都不发⽣。为了避免对分布式事务的需要，我们⾸先将请求持久化记录为 单条消息，然后从这第⼀条消息中衍⽣出贷记指令与借记指令。⼏乎在所有数据系统中，单对象写⼊都 是原⼦性的（参阅“单对象写⼊），因此请求要么出现在⽇志中，要么就不出现，⽆需多分区原⼦提交。 如果流处理器在步骤2中崩溃，则它会从上⼀个存档点恢复处理。这样做时，它不会跳过任何请求消 息，但可能会多次处理请求并产⽣重复的贷记与借记指令。但由于它是确定性的，因此它只是再次⽣成 相同的指令，⽽步骤3中的处理器可以使⽤端到端请求ID轻松地对其除重。 如果你想确保付款⼈的帐户不会因此次转账⽽透⽀，则可以使⽤⼀个额外的流处理器来维护账户余额并 校验事务（按付款⼈账户分区），只有有效的事务会被记录在步骤1中的请求⽇志中。 通过将多分区事务分解为两个不同分区⽅式的阶段，并使⽤端到端的请求ID，我们实现了同样的正确性 属性（每个请求对付款⼈与收款⼈都恰好⽣效⼀次），即使在出现故障，且没有使⽤原⼦提交协议的情 况下依然如此。使⽤多个不同分区⽅式的阶段与我们在“多分区数据处理”中讨论的想法类似（参阅“并发 控制”）

**及时性与完整性：**

数据流系统的正确性、

宽松地解释约束：如前所述，执⾏唯⼀性约束需要共识，通常通过在单个节点中汇集特定分区中的所有事件来实现。如果 我们想要传统的唯⼀性约束形式，这种限制是不可避免的，流处理也不例外。 然⽽另⼀个需要了解的事实是，许多真实世界的应⽤实际上可以摆脱这种形式，接受弱得多的唯⼀性： 如果两个⼈同时注册了相同的⽤户名或预订了相同的座位，你可以发送其中⼀个发消息道歉，并要 求他们选择⼀个不同的⽤户名。这种纠正错误的变化被称为**补偿性事务（compensating transaction）。**

无协调数据系统、

**信任但验证**

**维护完整性，尽管软件有Bug：**

**不要盲目信任承诺：**

**验证的文化：**

**为可审计性而设计：**

**端到端原则重现**

**用于可审计数据系统的工具：**

**做正确的事情**

**预测性分析：**偏见与分歧、责任与问责、反馈循环

**隐私和追踪：**监视、同意与选择的自由、隐私与数据使用、数据资产与权力、记者工业革命、立法与自律