非关系型数据库

于钧一　乔艺萌
杨浩哲　姜美锐

排版：谢颖
版本号：V1.0
修订时间：2018.12

山软智库-让知识回归平凡

目录

第一章　NoSQL 简介

作者：杨浩哲

1.1　实际用例

- **新浪微博**：Redis+MySQL，使用 200 多台物理机
- **淘宝数据平台**：Oceanbase，使用的 Tair 在 2010 年 6 月 30 日对外开源
- **视觉中国**：MongoDB
- **优酷**：在线评论使用 MongoDB；运营数据分析使用 Hadoop/Hbase
- **飞信空间**：使用 HandlerSocket
- **豆瓣社区**：使用 BeansDB

1.2 NoSQL 的产生背景

2011 年，中国互联网行业持有数据总量达到 1.9EB（1EB 字节相当于 10 亿 GB）2011 年，全球被创建和复制的数据总量为 1.8ZB（1.8 万亿 GB） 2013 年，生成这样规模的信息量只需 10 分钟 2015 年，全球被创建和复制的数据总量将增长到 8.2EB 以上。

2020 年，全球电子设备存储的数据将暴增达到 40ZB ... 暴增的海量数据标识着我们进入了**大数据时代**。

1.3 大数据

1.3.1 大数据的四个特征

- **大量化**：数据一直都在以每年 50%的速度增长，也就是说每两年就增长一倍（大数据摩尔定律）
- **快速化**：从数据的生成到消耗，时间窗口非常小，可用于生成决策的时间非常少
- **多样化**：大数据是由结构化和非结构化数据组成的

非结构化数据类型多样：邮件、视频、微博、位置信息、链接信息、手机呼叫、网页点击、长微博...

- **价值化**：价值密度低，商业价值高

以视频为例，连续不间断监控过程中，可能有用的数据仅仅有一两秒，但是具有很高的商业价值

1.4 普通数据和大数据的对比

	普通数据	大数据
数据规模（基本单位）	MB	GB 以上

	普通数据	大数据
数据类型	种类单一，且以结构化数据为主	种类繁多，且存在结构化、半结构化、非结构化的数据
模式（Schema）和数据的关系	先有模式后有数据	难以确定模式，多为先有数据后有模式
处理对象	数据仅为处理对象	以数据为辅助解决其他问题的工具
处理工具	一般只需要一种方式（One Size Fits All）	需要多种方式（No SIze Fits All）

1.5 关系数据库和 NoSQL 的对比（重点）

1.5.1 关系数据库的优缺点

优点：

- 通用性和高性能
- 能保持数据的一致性（事务处理）
- 能够保证最小冗余
- 能实现复杂查询如 JOIN
- 拥有成熟的技术

缺点：

- 不适合在分布式环境中的向外扩展

解释：1、随着用户数的不断增加，中心的数据库服务器的压力会越来越大；如果只为主服务器增加从服务器，那么从服务器只能响应读操作（适用于读操作大于写操作的情形）；如果增加主服务器，能够减少负载，但是更新处理的冲突可能会造成数据的不一致，这样对于读写请求在应用服务器中需要进行判断分配给合适的主数据库。 2、如果增加 cache（如 Memcached），就需要考虑 cache 和数据库数据的一致性，保证数据库的更新能在很短的时间内反映到 cache 中。 3、写操作主要集中在主服务器上，主服务器无法承担负载压力后只能采用向上扩展方式，开销很大。 4、还可以将数据进行水平或垂直切分分布到多个节点，但代价同样很高，同时会引入分布式事务以及跨节点的 join、排序操作（跨节点的 join 请求会耗费大量的网络资源，如果在应用层进行整合，则会加重应用层的负担）

- 难以支持高并发读写

为保障可用性，数据通常保存副本。 关系数据库本质上支持事务，那么就要保证 ACID 特性中的原子性和一致性，因此数据的更新必须在所有副本间同步，会带来一定的延迟。只有极个别云中的关系数据库也支持读副本(Amazon RDS MySQL)，允许副本间数据不一致，但这种副本无法在主副本故障时，替代主副本，仍然需要存在同步备份的副本。

缺点（整合 2）：

- 不适合在分布式环境中的向外扩展
- 所有的应用服务器都共享一个中心的数据库服务器
- 难以支持高并发读写
- 不擅长进行**大量数据的写入处理**、
- 对于**字段不固定的应用**无法进行**表结构的变更及建立索引**
- **对简单查询需要快速返回结果的处理**效率较低

总结： 关系型数据库在**大量数据的写入处理**、**表结构变更及建立索引**、**字段不固定的应用**、**对简单查询需要快速返回结果的处理**方面存在不足

1.5.2 NoSQL 的优点

- 易于数据的分散
- 提升性能和增大规模
- 模式自由
- 扩展性好

1.6 典型的 NoSQL 类型

存储类型	代表解决方案	特点
列存储	HBase	按列存储，适用于数据压缩，对一个或几个字段进行查询的效率很高
文档存储	MongoDB	保证海量数据存储的同时，具有良好的查询性能，用类 Json 格式进行存储
Key-Value 存储	Redis（永久性和临时性兼具）、MemcacheDB（临时性键值存储）	具有极高的并发读写性能。通过 key 迅速查找到 value，但只能通过 key 查询
图数据库	Neo4J	图形关系的最佳存储模式
对象数据库	db4o，Versant	类似面向对象语言的语法操作数据库，通过对象的方式存取数据
xml 数据库	BaseX、Berkerey DB XML	高效存储 XML 数据，并支持 XML 的内部查询语法

1.7 NoSQL 类型解释

1.7.1 键值存储

即抛弃了表的概念，所有的数据都以键值对的形式存储（这样的话，不支持模糊查询），具体的存储方法，分为**临时性**，**永久性**和**两者兼具**三种。
键值存储支持键上自有的隐形索引，但键值存储一般不支持事务处理机制。

很关键的一点是：键值存储引擎并不在意"值"的内部结构，一个值上可以存储任意信息（可以是一个 xml 文档，一个 json 对象，或者其他任意序列化形式），它**依赖客户端对"值"进行解释和管理**

临时性键值存储：即所有数据保存在内存中，服务退出时数据丢失
优点是存取速度非常快，理论上最快

永久性键值存储： 即将数据存储在磁盘上，（学过操作系统的应该知道，这是相对永久，仍然需要各种容灾处理）
而 Redis 则是两者兼具，其既在内存中存储，又定时根据数据的变动速度，动态的将数据存储到磁盘中。

1.7.2 基于文档的数据库

如 MongoDB，这种存储没有强制的架构，同时支持**嵌套类型**，如**楼中楼的多层回复**等，存储方式一般是基于 json 字符串、xml 文档或是某些类似的能够嵌套的文档结构。

和键值存储一样，该种存储方式不需要定义表结构，但是可以像关系数据库一样使用复杂的查询来查找内容，这是键值存储不具备的。
和键值存储不同的是，文档存储关心文档的内部结构，因此文档存储支持二级索引，从而可以对任意字段进行高效查询。

为了便于**管理数据**，应用对要检索的数据采取一些约定，或者利用存储引擎的能力将不同的文档划分成不同的集合。

1.7.3 面向列的数据库

列式存储以流的方式在列中存储所有的数据。对于任何记录，索引都可以快速地获取列上的数据。

如 HBASE，这种数据库比较适合处理海量高维的（稀疏）数据，复杂的数据更适合存储在列式数据库中。列式数据库也支持行检索，但也是通过从每个列获取匹配的列值重新组成列的方式进行的。

课堂上的一个例子是，为了做人物画像，需要一个人在医院的所有购药的药物种类的记录，一个人只会有几种或者十几种的购药记录，但药物种类有几十万种，如果以一

个人为一行数据的话，使用关系型数据库会有很大的开销，这样就需要面向列的数据库来应对。

第二章　数据一致性理论

作者：杨浩哲

2.1 数据一致性

2.1.1 强一致性

解释：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
注意：单副本数据**容易保证强一致性**；多副本数据需要使用**分布式事务协议**
也叫**即时一致性**

2.1.2 弱一致性

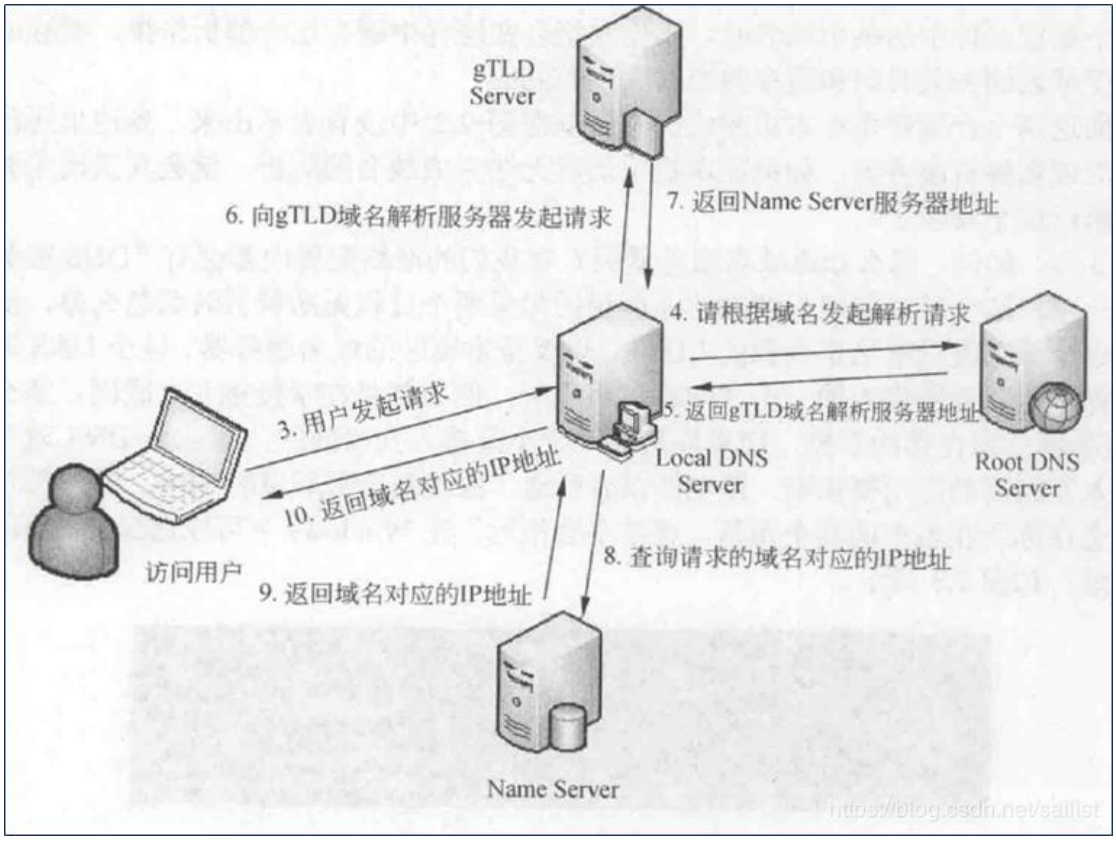
假如 A 先写入了一个值到存储系统，存储系统不能保证后续 A，B，C 的读取操作能读取到最新值
不一致性窗口（区间）：从 A 写入到后续操作 A，B，C 读取到最新值这一段时间
注意：弱一致性不保证可以读到最新的数据值，**即使过了不一致性窗口也可能读不到**

2.1.3 最终一致性

最终一致性是弱一致性的一种特例
假如 A 先写了一个值到存储系统，存储系统保证如果在 A，B，C 后续读取之前没有其它写操作更新同样的值的话，最终所有的读取操作都会读取到 A 写入的最新值
不一致性窗口"的区间和弱一致性相同，且其大小依赖于以下的几个因素（如果没有失败产生）：

- 交互延迟
- 系统的负载
- 以及复制技术中 replica 的个数（这个可以理解为 master/slave 模式中，slave 的个数）

最终一致性方面最出名的系统可以说是 DNS 系统，当更新一个域名的 IP 以后，根据配置策略以及缓存控制策略的不同，最终所有的客户都会看到最新的值

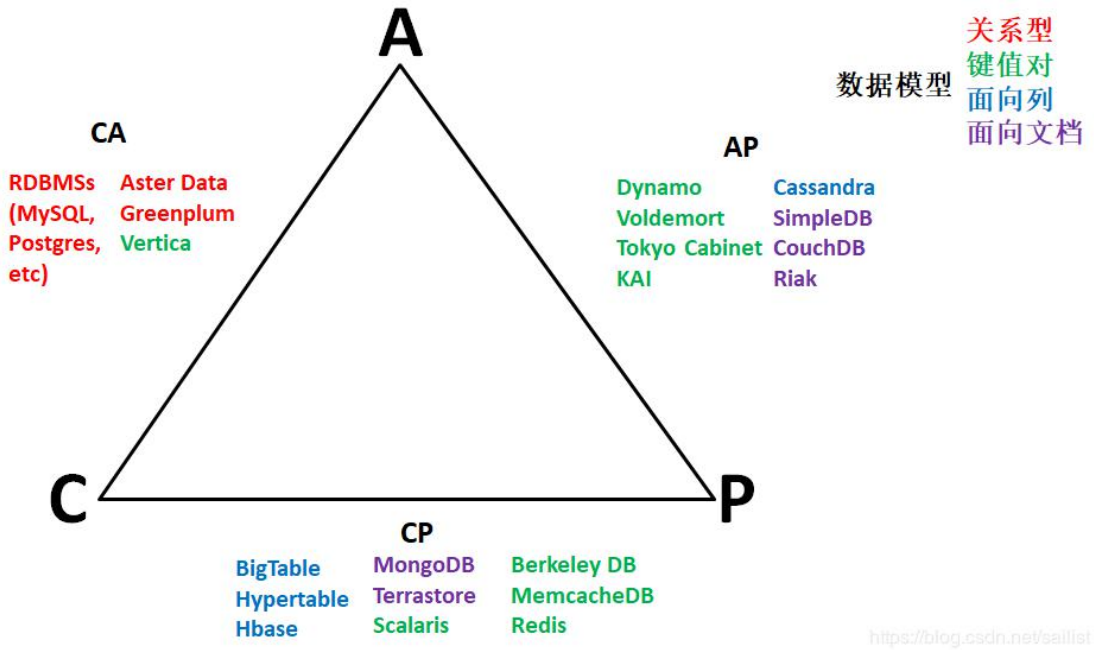


2.2 XAP理论

一个分布式系统不可能满足一致性，可用性和分区容错性这三个需求，最多只能**同时满足两个**

- **C**：Consistency 一致性：数据一致更新，所有数据变动都是同步的（所有节点访问同一份最新的数据副本）
- **A**：Availability 可用性：某个节点的宕机不会影响其他节点继续完成操作
- **P**：Tolerance of network Parition 分区容忍性：能容忍网络分区（如果集群中的机器被分成了两部分，这两部分不能互相通信，系统能够继续正常工作）

2.2.1 CAP 理论的应用实例



2.3 BASE 理论

核心思想是：即使无法做到**强一致性**，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到**最终一致性**。BASE 模型完全不同于 ACID 的强一致性模型，主要用于 NoSQL，牺牲高一致性，获得可用性或可靠性

- **Basically Availble 基本可用**：支持分区失败(e.g. sharding 碎片划分数据库)
- **Soft-state 软状态/柔性事务**：状态可以有一段时间不同步，异步。
- **Eventural Consistency 最终一致性**：最终数据是一致的就可以了，而不是时时高一致。

如刷新视频，会出现新的视频，但追加的数据不同（最终会相同--最终一致性）

2.3.1 关系数据库的 ACID 理论（回顾对比用）

高一致性 + 可用性 很难进行分区

- **Atomicity 原子性**：一个事务中所有操作都必须全部完成，要么全部不完成。
- **Consistency 一致性**： 在事务开始或结束时，数据库应该在一致状态。
- **Isolation 隔离性**：事务将假定只有它自己在操作数据库，彼此不知晓。
- **Durability 持久性**：一旦事务完成，就不能返回。

2.4 数据一致性实现技术

2.4.1 NWR 模型

根据 CAP 理论，一致性，可用性和分区容错性最多只能满足两个，因此需要在一致性和可用性之间做一平衡 NWR 模型是 Amazon 分布式存储引擎 Dynamo 中使用的技术，将 CAP 的选择权交给了用户，让用户自己选择 CAP 中的哪两个。

NWR 模型中：（下文中备份也可以理解为节点）

- N = N 个备份，即复制的节点数量
- R = 至少读取 R 个备份（成功读操作的最小节点数）
- W = 要写入至少 W 份才认为成功（成功写操作的最小节点数）
- 只需要 W+R > N，就可以保证强一致性

在分布式系统中，一般都要有容错性，所以一般 N>3

W+R > N， 所以 R > N-W 所以读取的份数一定要比总备份数减去确保写成功的节点的差值要大。根据抽屉原理（是吧？），这个时候的读操作，肯定会读到一个最新的写操作。

优化写性能(AP)（写多读少） 配置 W=1，N=R 这样，写完一个副本就成功，其他的副本异步复制即可；而 R=N，那么读取数据的时候需要读全部节点的数据，（这个时候可能会出现数据冲突，需要用户自行判断冲突数据）。

优化读性能(CP)（读多写少） 配置 W=N，R=1 这时，写完所有的副本才算成功，而读的时候只需读一个副本即可（*只能保证读任意节点都一致，不能保证是最新，此时可能还没有写成功?*）。

应用：如使用 NoSQL 做 cache 等业务的时候（只要写好了就可以读）？

平衡读写性能(AC) 当数据不多，单台能搞定，且不需要容错和扩展性的时候，可以配置 N=1（只有一份数据），根据公式 W+R>N，则 W=1，R=1。这种情况就简化为单机问题了。

2.4.2 NWR 补充内容（用于加深理解）

NWR 模型的一些设置会造成脏数据的问题，可能每次的读写操作都不在同一个结点上，于是会出现一些结点上的数据并不是最新版本，但却进行了最新的操作。

所以，Amazon Dynamo 引了数据版本的设计。也就是说，如果你读出来数据的版本是 v1，当你计算完成后要回填数据后，却发现数据的版本号已经被人更新成了 v2，那么服务器就会拒绝你。版本这个事就像“乐观锁”一样。

但是，对于分布式和 NWR 模型来说，版本也会有恶梦的时候——就是版本冲的问题，比如：我们设置了 N=3 W=1，如果 A 结点上接受了一个值，版本由 v1 -> v2，但还没有来得及同步到结点 B 上（异步的，应该 W=1，写一份就算成功），B 结点上还是 v1 版本，此时，B 结点接到写请求，按道理来说，他需要拒绝掉，但是他一方面并不知道

别的结点已经被更新到 v2，另一方面他也无法拒绝，因为 W=1，所以写一分就成功了。于是，出现了严重的版本冲突。

Amazon 的 Dynamo 把版本冲突这个问题巧妙地回避掉了——版本冲这个事交给用户自己来处理。

于是，Dynamo 引入了 Vector Clock（矢量钟？！）这个设计。这个设计让每个结点各自记录自己的版本信息，也就是说，对于同一个数据，需要记录两个事：1）谁更新的我，2）我的版本号是什么。

下面，我们来看一个操作序列：

1）一个写请求，第一次被节点 A 处理了。节点 A 会增加一个版本信息(A，1)。我们把这个时候的数据记做 D1(A，1)。 然后另外一个对同样 key 的请求还是被 A 处理了于是有 D2(A，2)。这个时候，D2 是可以覆盖 D1 的，不会有冲突产生。

2）现在我们假设 D2 传播到了所有节点(B 和 C)，B 和 C 收到的数据不是从客户产生的，而是别人复制给他们的，所以他们不产生新的版本信息，所以现在 B 和 C 所持有的数据还是 D2(A，2)。于是 A，B，C 上的数据及其版本号都是一样的。

3)如果我们有一个新的写请求到了 B 结点上,于是 B 结点生成数据 D3(A,2; B,1)，意思是：数据 D 全局版本号为 3，A 升了两新，B 升了一次。这不就是所谓的代码版本的 log 么？

4）如果 D3 没有传播到 C 的时候又一个请求被 C 处理了，于是，以 C 结点上的数据是 D4(A,2; C,1)。

5）好，最精彩的事情来了：如果这个时候来了一个读请求，我们要记得，我们的 W=1 那么 R=N=3，所以 R 会从所有三个节点上读，此时，他会读到三个版本：
A 结点：D2(A,2) B 结点：D3(A,2; ?B,1); C 结点：D4(A,2; ?C,1) 6）这个时候可以判断出，D2 已经是旧版本（已经包含在 D3/D4 中），可以舍弃。

6）但是 D3 和 D4 是明显的版本冲突。于是，交给调用方自己去做版本冲突处理。就像源代码版本管理一样。

很明显，上述的 Dynamo 的配置用的是 CAP 里的 A 和 P。

2.5 两阶段提交协议

在分布式系统中，每个节点只能知道自己的操作是否成功，不能知道其他节点操作是否成功。因此，当一个事务跨越多个节点时（多个节点都有操作），为了**保持事务的 ACID 特性**，需要引入一个**协调者**来统一掌控所有节点(参与者)的操作结果并最终反馈给这些节点是否进行提交。

二阶段提交(Two-phaseCommit) 是指，在计算机网络以及数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法(Algorithm)。

2.5.1 算法描述

参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。

2.5.2 两个阶段

- **第一阶段：** 准备阶段（投票阶段）
- **第二阶段：** 提交阶段（执行阶段）。

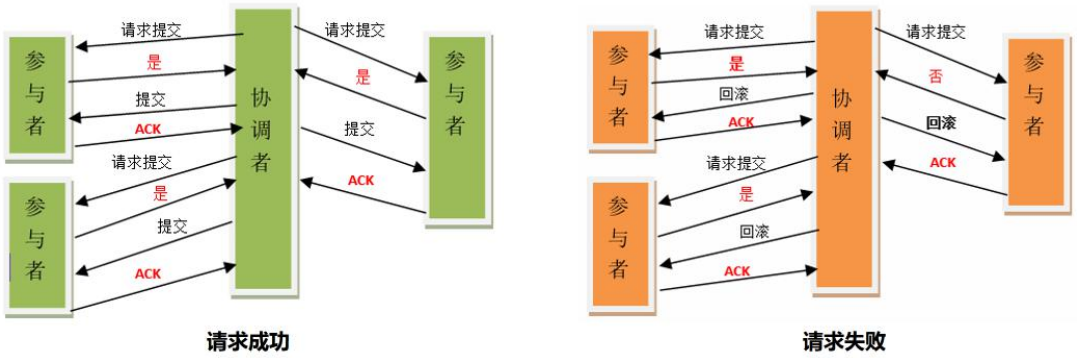
2.5.3 准备阶段

以一个协调者三个参与者为例

- 协调者向三个参与者分别发送请求提交请求，等待参与者应答是/否
- 请求提交请求中包含了要执行的操作，**参与者**如果可以执行，就执行协调者发起的事务操作，写日志，并返回是，如果因为某些原因不能执行或者执行失败，就返回否

2.5.4 提交阶段

- 协调者接收参与者返回的消息
- 如果均为是，协调者向三个参与者发送提交请求，等待所有参与者完成提交、释放所有事务处理过程中使用的锁资源、返回完成消息后，完成事务
- 如果有一个以上为否，则协调者向三个参与者发送回滚请求，等待所有参与者完成回滚应答 ACK 即回滚完成消息后，取消事务



PS：不管最后结果如何，第二阶段都会结束当前事务。

2.5.5 优缺点

优点：实现简单

缺点：

- **同步阻塞**：执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。
- **单点故障**：由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）
- **数据不一致**：在二阶段提交的阶段二中，当协调者向参与者发送 commit 请求之后，发生了局部网络异常或者在发送 commit 请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了 commit 请求。而在这部分参与者接到 commit 请求之后就会执行 commit 操作。但是其他部分未接到 commit 请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象

- **二阶段无法解决的问题**：协调者再发出 commit 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交

第三章　HBase

作者：姜美锐

引言：HBase 是列存储类型的非关系数据库的代表解决方案。特点是按列存储，适用于数据压缩，对一个或几个字段进行查询的效率很高。

3.1 列存储数据库

3.1.1 列式存储对比关系型数据库

关系型数据库

Name	Age	Gender	Birthday
a	20	M	1990-10-1
b	40	F	1970-8-24
c	30	M	1980-1-18
…	…	…	…

列存储NoSQL

Name : a	Age : 20	Gender : M	Birthday : 1990-10-1	Hobby : travel
Name : b	Age : 40	Birthday : 1970-8-24	Tel : 12345678	
Name : c	Age : 30	Gender : M		
…	…	…	…	…

目的：　列式存储主要解决关系型数据库中的稀疏表问题。
存储区别：　与关系模型存储记录不同，列式存储以流的方式在列中存储所有的数据。对于任何记录，索引都可以快速地获取列上的数据。
检索：　列式存储支持行检索，但这需要从每个列获取匹配的列值，并重新组成行

3.2 HBase 简介

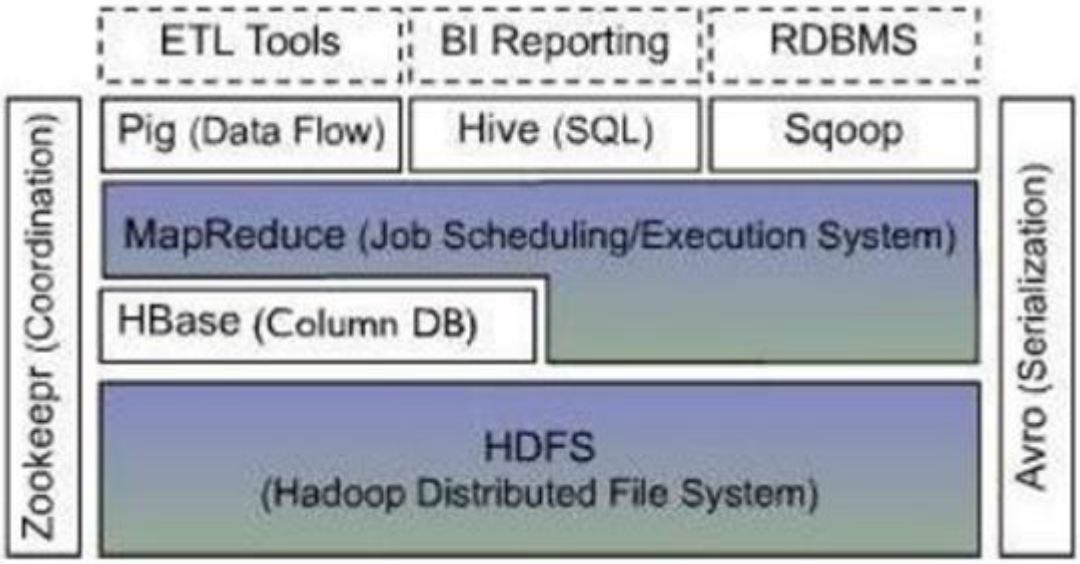
3.2.1 HBase

起源：　HBase 是 Google BigTable 的开源实现，模仿并提供了基于 Google 文件系统的 BigTable 数据库的所有功能
定义：　HBase（Hadoop Database）是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用 HBase 技术可在廉价 PC Server 上搭建起大规模结构化存储集群。
目标：　处理非常庞大的表（超过 10 亿行数据，并且有数百万列元素组成的数据表）
扩展方式：　HBase 主要依靠**横向**扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力
检索方式：　HBase 仅能通过**主键**(row key)和主键的 **range** 来检索数据，仅支持单行事务(可通过 Hive 支持来实现多表连接等复杂操作)

存储数据形式： **非结构化**和**半结构化**的松散数据
HBase 中表的特点：

- 大：上亿行、上百万列
- 面向列 ： 面向列(族)的存储和权限控制，列(族)独立检索
- 稀疏 ： 为空(null)的列不占用存储空间，表设计的非常稀疏

3.2.2 Hadoops 生态系统



- HBase 位于结构化存储层
- HDFS 为 HBase 提供了高可靠性的底层存储支持
- MapReduce 为 HBase 提供了高性能的计算能力
- Zookeeper 为 HBase 提供了稳定服务和失败恢复机制
- Pig 和 Hive 还为 HBase 提供了高层语言支持，使得在 HBase 上进行数据统计 处理变的非常简单
- Sqoop 则为 HBase 提供了方便的 RDBMS 数据导入功能，方便数据迁移

3.2.3 HBase VS 传统数据库

HBase 特点

- HBase 是按照 BigTable 开发的，是一个稀疏的、分布的、持续多维度的排序映射数组
- HBase 基于列模式的映射数据库，只能表示很简单的键-数据的映射关系，大大简化了传统的关系数据库

关系数据库特点

- 面向磁盘存储和索引结构
- 多线程访问
- 基于锁的同步访问机制
- 基于 log 记录的恢复机制

两者区别

1. **数据类型**

- HBase 只有简单的字符串类型，所有类型都由用户自己处理，它只保存字符串
- 关系数据库有丰富的类型选择和存储方式

2．**数据操作**
- HBase 操作只有很简单的插入、查询、删除、清空等，表和表之间是分离的，没有复杂的表和表之间的关系，所以不能(也没必要)实现表和表之间的关联等操作
- 传统的关系数据通常有各种各样的函数、连接操作

3．**存储模式**
- HBase 是基于列存储的，每个列族都有若干个文件保存，不同列族的文件是分离的
- 传统的关系数据库是基于表格结构和行模式保存的

4．**数据维护**
- HBase 不是真正的更新，实际上是插入了新的数据。旧有的版本仍然会保留。
- 传统关系数据库里面是替换修改

5．**可伸缩性**
- 能够轻易的增加或者减少（在硬件出错时）硬件数量，而且对错误的兼容性比较高
- 传统的关系数据库通常需要增加中间层才能实现类似的功能

3.3 HBase 数据模型

3.3.1 HBase 结构

Row Key	Timestamp	Column Family	
		URI	Parser
r1	t3	url=http://www.taobao.com	title=天天特价
	t2	host=taobao.com	
	t1		
r2	t5	url=http://www.alibaba.com	content=每天…
	t4	host=alibaba.com	

- HBase 的索引是行关键字、列关键字和时间戳
- 数据都是字符串，没有类型
- 每一行都有一个可排序的主键和任意多的列
- 同一张表里面的每一行数据都可以有截然不同的列

3.3.2 HBase 的行和列

- **列**：`<family>:<label>`
  由字符串组成，每一张表有一个 family 集合，这个集合是固定不变的，相当于表的结构。
  只能通过改变表结构来改变，但是 label 值相对于每一行来说都是可以改变的。
  同一个 family 里面的数据存储在同一个目录底下。
- **行**：
  HBase 的写操作是锁行的，每一行都是一个原子元素，都可以加锁

3.3.3 HBase 模型中的三个重要概念

- **行键（Row Key）**
  HBase 表的主键，表中的记录按照行键排序。行键用来检索记录的主键
- **时间戳（Timestamp）**
  每次数据操作对应的时间戳，可看作是数据的版本号，不同版本通过时间戳来进行索引。每个更新都是一个新的版本，而 HBase 会保留一定数量的版本，这个值是可以设定的。
  客户端可以选择获取距离某个时间最近的版本，或者一次获取所有版本
- **列族（Column Family）**
  表在水平方向有一个或者多个列族组成，一个列族中可以由任意多个列组成。
  列族支持动态扩展，所有列均以二进制格式存储，用户需要自行进行类型转换

3.3.4 HBase 的概念视图和物理视图

概念视图

一个表可以想象成一个大的映射关系，通过主键，或者主键+时间戳，可以定位一行数据（由于是稀疏数据，所以某些列可以是空白）

Row Key	Time Stamp	Column "contents:"	Column "anchor:"		Column "mime:"
"com.cnn.www"	t9		"anchor:cnnsi.com"	"CNN"	
	t8		"anchor:my.look.ca"	"CNN.com"	
	t6	"<html>..."			"text/html"
	t5	"<html>..."			
	t5	"<html>..."			

物理视图

在物理存储上面，表实际是按照列来保存的。概念视图中空白的列实际不会被存储，当请求这些空白的单元格的时候，会返回 null 值

Row Key	Time Stamp	Column "contents:"
"com.cnn.www"	t6	"<html>..."
	t5	"<html>..."
	t3	"<html>..."

Row Key	Time Stamp	Column "anchor:"	
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"

3.3.5 HBase 的增删改查规则

创建表：创建表时，要指定表中有哪些列族(模式)
更改列族：增加或删除列族，需要修改表的模式。
修改表模式需要先将表设置为不可用(disable)，模式修改完成再启用表(enable)
查询：只能通过 row key 访问

1. 通过单个 row key 访问
2. 通过 row key 的范围访问
3. 全表扫描

3.4 HBase 访问

类型	特点	场合
Native Java API	最常规和高效的访问方式	适合Hadoop MapReduce作业并行批处理HBase表数据
HBase Shell	HBase的命令行工具，最简单的接口	适合HBase管理使用
Thrift Gateway	利用Thrift序列化技术，支持C++、PHP、Python等多种语言	适合其他异构系统在线访问HBase表数据
REST Gateway	解除了语言限制	支持REST风格的Http API访问HBase
Pig	使用Pig Latin流式编程语言来处理HBase中的数据	适合做数据统计
Hive	简单	当需要以类似SQL语言方式来访问HBase的时候

3.4.1 命令行访问

操作	命令
读	get
写	put
扫描	scan
删除	delete

详细命令请对照闫中敏老师的 ppt

3.4.2 Java API

请参照闫中敏老师的 ppt

3.5 HBase 的实现

3.5.1 HBase 的主要功能组件

- **库函数**：链接到每个客户端
- 一个 **HMaster 主服务器**
- 许多个 **HRegion 服务器**

一个 HBase 中存储了许多表。每个表都是一个 HRegion 集合，每个 HRegion 包含了位于某个域区间内的所有数据。
具体存储形式请参考闫中敏老师的 ppt

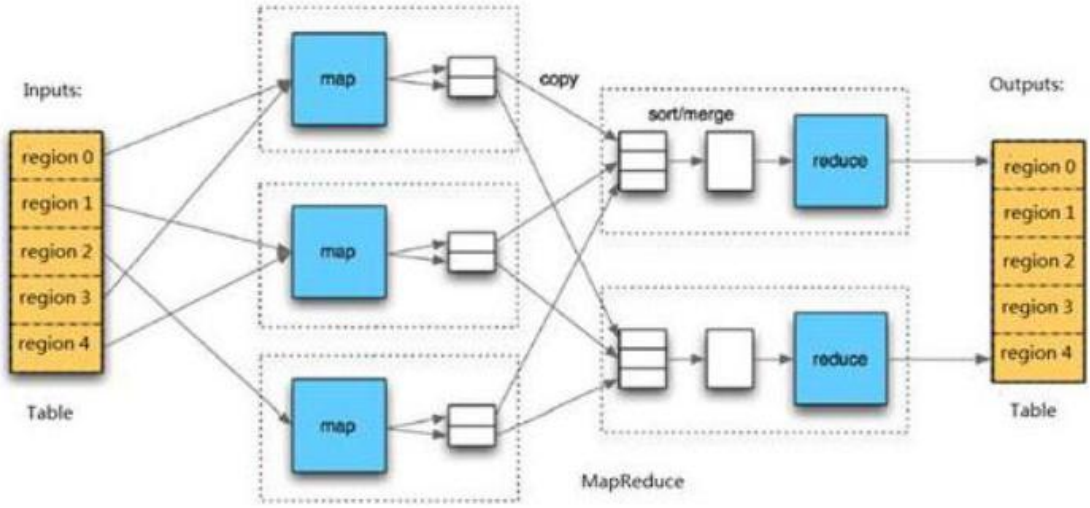
3.5.2 HDFS

组成：一个 HDFS 集群是由一个 NameNode 和若干个 DataNode 组成

- NameNode 作为主服务器，管理文件系统的命名空间和客户端对文件的访问操作
- DataNode 管理存储的数据

3.6 MapReduce

MapReduce 流程：



函数	输入	输出	说明
Map	$<k_1,v_1>$	List($<k_2,v_2>$)	1.将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理 2.每一个输入的$<k_1,v_1>$会输出一批$<k_2,v_2>$。$<k_2,v_2>$是计算的中间结果
Reduce	$<k_2,\text{List}(v_2)>$	$<k_3,v_3>$	输入的中间结果$<k_2,\text{List}(v_2)>$中的$\text{List}(v_2)$表示是一批属于同一个k_2的value

Map 与 Reduce 之间还存在一个 Shuffle 过程，可以结合数据科学导论的 MapReduce 相关课件学习详细过程。

第四章　Redis 数据库

作者：于钧一

以学生表，课程表，选课表的数据引入：　关系数据库这样存储

学生表 (Student)

Sno	Sname	Sex	Age
201201	Edward	M	32
201202	Catty	F	23

课程表 (Course)

Cno	Cname	Credit
1001	DataBase	3
1002	DataStructure	2

选课表(SC)

Sno	Cno	Score
201201	1001	98
201201	1002	95
201202	1001	95
201202	1002	86

Redis 会这样存储，可以等到本章看完再来看这里

key	value
Sno: 201201	{Sname: 'Edward', Sex: 'M', Age: 32}
Sno: 201202	{Sname: 'Catty', Sex: 'F, Age: 23}
Cno:1001	{ 'DataBase', '3'}
Cno:1002	{ 'DataStructure', '2'}
201201_1001	'98'
201201_1002	'95'
201202_1001	'95'
201202_1002	'86'

Hash类型

List类型

String类型

4.1 Redis 简介

1. Redis 是一个 **key-value** 数据库，　具有极高的并发读写性能，通过 key 迅速查找到 value，但只能通过 key 查询。key-value 很像是我们 java 中学过的 map 结构，以"键值对"存储数据，每一个键（key）都会对应一个唯一的值（value），类似哈希表，可以通过 key 来添加、查询、删除，这样的数据结构性能极高，扩展性亦良好。
2. 支持的数据类型包括 **string、list、set、zset(有序集合)和 hash**
3. 支持 push/pop、add/remove、集合并交差等丰富的操作，而且操作都是**原子性**的，支持各种不同方式的排序

4. 为了获得优异的性能，Redis 采用了内存中(in-memory)数据集(dataset)的方式。同时，Redis 支持数据的持久化，每隔一段时间将数据集转存到磁盘上(snapshot)，或者在日志尾部追加每一条操作命令(append only file,aof)。
5. 提供各种语言支持，java，python，cpp.....

4.2 Redis 数据模型

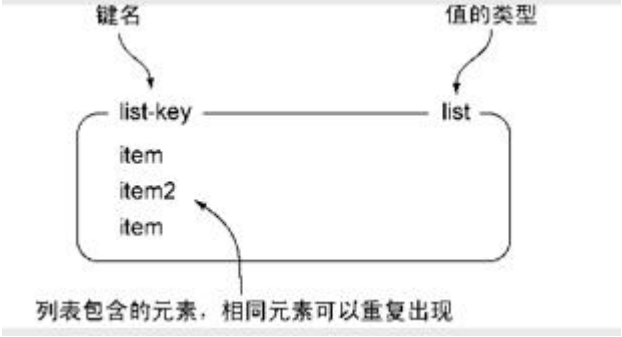
4.2.1 Key 相关概念

1. 二进制安全概念 在 C 语言中，以'/0'来表示一个字符串的结尾，这种根据某种特殊的标志来解析的字符串是二进制不安全的。Redis 的 key 是非二进制安全的字符串类型，values 都是二进制安全的
   str = "1234\0123"
   c 语言： strlen(str)=4
   二进制安全语言： strlen(str)=7
2. Redis 本质上一个 key-value 数据库，所以我们首先关注 key。首先 key 也是字符串类型，由于 key 不是 binary safe 的字符串，所以像“my key”和“mykey\n”这样包含空格和换行的 key 是不允许的。
3. 我们在使用的时候可以自己定义一个 Key 的格式，例如
   $object-type:id:field$ Key 不要太长。占内存，查询慢 Key 不要太短。
   u:1000:pwd 不如 user:1000:password 可读性好
4. Key 相关的一些命令 见闫中敏老师的 ppt，不在此罗列。唯一提醒的是凡是涉及到删除的命令，慎用。

4.2.2 value 的五种数据类型

1. String 可以保存二进制字节的序列，包括字符串、整数或者浮点数。相关命令在闫中敏老师的 ppt 罗列。 ① String 作为整数或者浮点数时，可以递增或者递减任意数值。整数的范围和运行平台的长整数的范围相同(32 位带符号整数或者 64 位带符号整数)，浮点数的范围与 IEEE754 标准的双精度浮点数相同 ② Redis 这种将 String 作为整数或者浮点数的能力，增加了灵活性
2. List 一个链表，链表上的每个节点都包含了一个字符串。从链表的两端推入或者弹出元素；根据偏移量对链表进行修剪(trim)；读取单个或者多个元素；根据值查找或者移除元素。相关命令在闫中敏老师的 ppt 罗列。

① List 是简单地字符串列表，通过插入顺序排序，一个列表结构可以有序地存储多个字符串，和表示字符串时使用的方法一样，可以添加一个元素到 Redis 列表的头部或尾部，列表的最大长度为 $2^{32}-1$ 个元素，列表中可以出现相同元素

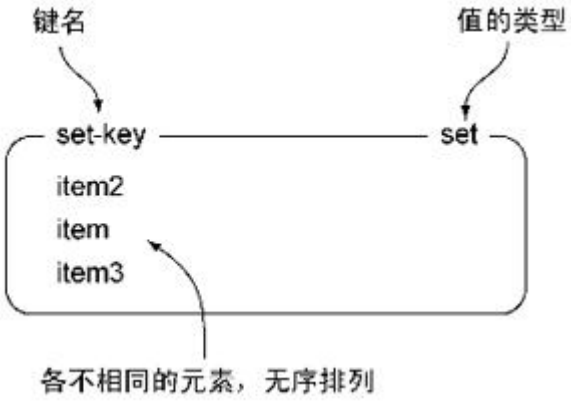


② Redis 的 list 类型其实就是一个每个子元素都是 string 类型的双向链表。我们可以通过 push，pop 操作从链表的头部或者尾部添加删除元素。这使得 list 既可以用作栈，也可以用作队列。

③ list 的 pop 操作还有阻塞版本的。当我们 pop 一个 list 对象时，如果 list 是空，或者不存在，会立即返回 nil。但是阻塞版本的 pop 直到等待超时或发现可弹出元素为止。当然可以加超时时间，超时后也会返回 nil。为什么要阻塞版本的 pop 呢，主要是为了避免轮询。

3. Set 包含字符串的无序收集器(unordered collection)，并且被包含的**每个字符串都是独一无二、各不相同的**。添加、获取、移除单个元素；检查一个元素是否存在于集合中；计算交集、并集、差集；从集合里面随机获取元素。相关命令在闫中敏老师的 ppt 罗列。

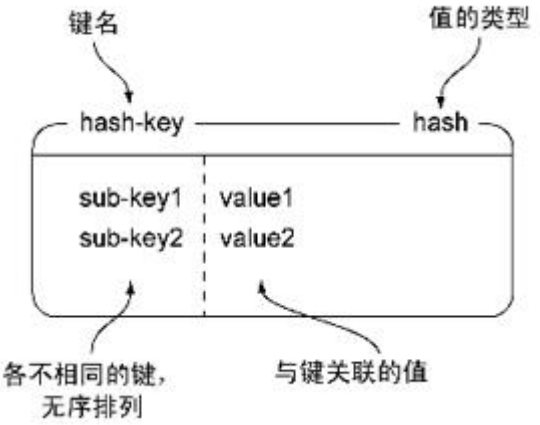
① set 元素最大可以包含 $2^{32}-1$ 个元素。



② set 的是通过 hash table 实现的，hash table 会随着添加或者删除自动的调整大小。通过使用散列表来保证自己存储的每个字符串都是各不相同的

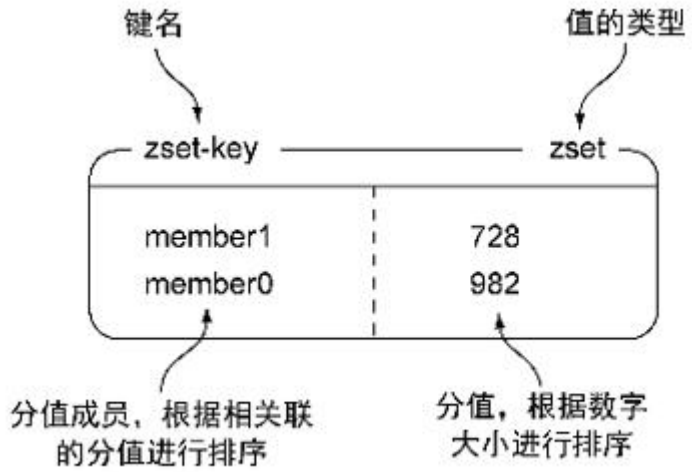
③ 关于 set 集合类型除了基本的添加删除操作，其他有用的操作还包含集合的取并集(union)，交集(intersection)，差集(difference)

4. Hash 包含键值对的无序散列表。添加、获取、移除单个键值对；获取所有键值对。相关命令在闫中敏老师的 ppt 中罗列。 ① 哈希可以存储多个键值对之间的映射。哈希存储的值既可以是字符串又可以是数字值，并且用户同样可以对散列存储的数字值执行自增操作或者自减操作。



5. ZSet 字符串成员(member)与浮点数分值(score)之间的有序映射，元素的排列顺序由分值的大小决定。添加、获取、删除单个元素；根据分值范围(range)或者成员来获取元素。相关命令在闫中敏老师的 ppt 罗列。

① 有序集合和散列一样，都用于存储键值对（注意这里的键值对仅仅是为了排序而成的键值对，更贴切的理解还是有顺序的 set，key 存储的是真正的内容，value 值可以只是帮助给 key 值排序的，也可以是具有真实意义的值）。有序集合的键被称为成员(member)，每个成员都是独一无二的。有序集合的值则被称为分值(score)，分值必须为浮点数。有序集合是 Redis 里面唯一一个既可以根据成员访问元素(这一点和散列一样)，又可以根据分值以及分值的排列顺序来访问元素的结构



② 集合排序 一个有序结合的每个成员都可以排序，就是为了按照有序集合排序获取它们，按分值从最小到最大排序。虽然成员都是独一无二的，但是分数值可以重复

4.3 Redis 其他管理命令

4.3.1 持久化

1. Redis 为了内部数据的安全考虑，会把本身的数据以文件形式保存到硬盘中一份，在服务器重启之后会自动把硬盘的数据恢复到内存(redis)的里边，数据保存到硬盘的过程就称为“持久化”效果。
2. 持久化方式一：Snapshotting(快照) 该持久化默认开启，一次性把 redis 中全部的数据保存一份存储在硬盘中。可以配置自动做快照持久化的方式。我们可以配置 Redis 在 n 秒内如果超过 m 个 key 被修改就自动做快照。 ① 如果数据量特别大，就不合适频繁进行快照持久化 ② 快照优势在于快照的文件适合备份，劣势在于可能会丢失数据（意外宕机后只能恢复到最后一次快照）
3. 持久化方式二： Append-only file(aof) aof 比快照方式有更好的持久化性，是由于在使用 aof 持久化方式时，Redis 会将每一个收到的写命令都通过 write 函数追加到文件中(默认是 appendonly.aof)。当 Redis 重启时会通过重新执

行文件中保存的写命令在内存中重建整个数据库的内容。 ① 由于 OS 会在内核中缓存 write 做的修改，所以可能不是立即写到磁盘上。这样 aof 方式的持久化也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉 Redis 我们想要通过 fsync 函数强制 os 写入到磁盘的时机。推荐的做法是每秒钟强制写入磁盘一次, 在性能和持久化方面做了很好的折中。② 优势在于不容易丢失数据，劣势在于 aof 的体积会很大，系统重启重新构建 redis 内存会花很长时间

4.3.2 主从复制

1. 当数据量变得庞大的时候，读写分离可以帮助系统优化性能 Redis 采用了 Master/Slave(主从机制)进行数据复制。
2. 主从复制：主节点负责写数据，从节点负责读数据，主节点定期把数据同步到从节点保证数据的一致性。主从复制允许多个 slave serve 拥有和 master server 相同的数据库副本。下面是 **Redis 主从复制**的一些特点 ① master 可以有多个 slave ② 除了多个 slave 连到相同的 master 外, slave 也可以连接其他 slave 形成图结构 ③ 主从复制不会阻塞 master。也就是说当一个或多个 slave 与 master 进行初次同步数据时，master 可以继续处理 client 发来的请求。相反 slave 在初次同步数据时则会阻塞（阻塞的意思是直到同步完成不处理其他业务），不能处理 client 的请求。 ④ 主从复制可以用来提高系统的可伸缩性(我们可以用多个 slave 专门用于 client 的读请求，比如 sort 操作可以使用 slave 来处理)，也可以用来做简单的数据冗余。 ⑤ 可以在 master 禁用数据持久化，只在 slave 上配置数据持久化
3. 主从同步 数据可已从主服务器向任意数量的从服务器上同步，包括一主多从和级联结构。 ① 全量同步：全量同步是一次性同步全部数据 ② 增量同步：只同步两个数据库不同的部分 ③ Redis 主从同步的策略：在 slave 初次连接上 master 时，首先采取全量同步，之后一直才用增量同步，只有增量同步出了问题或是有其他需求时，会采取全量同步。

4.3.3 事务

1. Redis 事务允许一组命令在单一步骤中执行
2. 事务有两个属性 ① 在一个事务中的所有命令作为单个独立的操作顺序执行 ② Redis 事务是原子的，原子意味着要么所有的命令都执行，要么都不执行
3. 在 Redis 中，事务的所有命令都将会被串行化的顺序执行，一个事务执行期 Redis 不会再为其他客户端的请求提供任何服务
4. Redis 的事务无法回滚
5. Redis 对事务的支持目前还比较简单。Redis 只能保证一个 client 发起的事务中的命令可以连续的执行，而中间不会插入其他 client 的命令。 Multi 事务开始 Exec 执行事务 Discard 放弃事务 放弃执行事务内的所有命令 Watch 监听 key Unwatch 放弃所有 key 的监听
6. watch 命令会监视给定的 key，当 exec 时如果监视的 key 从调用 watch 后发生过变化，则整个事务会失败。注意 watch 的 key 是对整个连接有效的，和事务一样，如果连接断开，监视和事务都会被自动清除。

7. 事务中出现语法错误，只要有一个命令有语法错误，执行 EXEC 命令后 Redis 就会直接返回错误，连语法正确的命令也不会执行(Redis 2.6.5 之前的版本会忽略有语法错误的命令，然后执行事务中其他语法正确的命令)
8. 事务中出现运行错误：运行错误指在命令执行时出现的错误，比如使用散列类型的命令操作集合类型的键，这种错误在实际执行之前 Redis 是无法发现的，所以在事务里这样的命令是会被 Redis 接受并执行的。如果事务里的 一条命令出现了运行错误，事务里其他的命令依然会继续执行(包括出错命令之后的命令)。

4.3.4 乐观锁和悲观锁

1. 锁的存在都是为了更好的解决并发访问造成的数据不一致性的的问题，乐观锁可以认为是一种在最后提交的时候**检测冲突**的手段，而悲观锁则是一种**避免冲突**的手段
2. 乐观锁：应用在系统层面和数据的业务逻辑层次上的锁（实质上并没有加锁），只是利用程序处理并发，假定当某个用户读取某数据时没有其他用户来访问修改这个数据，在最后事务提交时刻检查这个数据的版本以确定没有被其他用户修改，开销比较小。乐观锁大部分是基于版本控制实现。 ① 优：避免悲观锁涉及到的不断加锁解锁而导致的大开销，降低用户等待时间，提升大并发下的系统性能表现，乐观锁更适合读比较多的场景 ② 劣：如果系统冲突较多，每次都只能在提交数据时发现业务事务将要失败，意味着本次白做了而且还要重新进行计算，造成相当大的代价（重做的代价，发现失败太迟的代价）。而且乐观锁无法解决读脏数据的问题。
3. 悲观锁：完全依赖于数据库锁的机制实现的，在数据库中可以使用 Repeatable Read 的隔离级别(可重复读)来实现悲观锁，它完全满足悲观锁的要求(加锁)。在读取的时候就对数据进行加锁， 在该用户读取数据的期间，其他任何用户都不能来修改该数据，但是其他用户是可以读取该数据的，只有当自己读取完毕才释放锁。（可同时读，一读谁也不能写） ① 优：能避免冲突的发生 ② 劣：开销大，加锁时间长，对于并发的访问性支持不好
4. watch 命令和乐观锁 在 Redis 的事务中，WATCH 命令可用于提供 CAS (check-and-set)功能，我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在 WATCH 之后有任何 Key 的值发生了变化，EXEC 命令执行的事务都将被放弃，同时返回 Null multi-bulk 应答以通知调用者事务执行失败。注意以上的说明仅仅针对单个连接，如果有多个客户端在同时执行这段 watch 的代码，就会出现 race condition（和操作系统里提到的概念一样）代码的写法见闫中敏老师的 ppt

4.3.5 发布订阅

发布订阅(pub/sub)是一种消息通信模式。订阅者可以通过 subscribe 和 pubscribe 命令向 Redis server 订阅自己感兴趣的消息类型，Redis 将信息类型成为通道。当发布者通过 publish 命令向 Redis server 发送特定类型的信息时，订阅该信息类型的全部 client 都会收到此消息。

4.3.6 管道

1. Redis 是一个 cs 模式的 tcp server，使用和 http 类似的请求响应协议。一个 client 可以通过一个 socket 连接发起多个请求命令。每个请求命令发出后 client 通常会阻塞并等待 Redis 服务处理，Redis 处理完后请求命令后会将结果通过响应报文返回给 client。如果一条命令 server 回应一次传输过程花费的时间较长，所以用 pipeline 的方式从 client 打包多条命令一起发出，不需要等待单条命令的响应返回，而 Redis 服务端会处理完多条命令后会将多条命令的处理结果打包到一起返回给客户端。
2. 管道(pipeline)可以一次性发送多条命令并在执行完后一次性将结果返回，pipeline 通过减少客户端与 redis 的通信次数来实现降低往返延时时间，而且 Pipeline 实现的原理是队列，而队列的原理是时先进先出，这样就保证数据的顺序性。
3. 适应 pipeline 的场景：批量的将数据写入 redis，允许一定比例的写入失败 比如 10000 条一下进入 redis，可能失败了 2 条无所谓，后期有补偿机制就行了，比如短信群发这种场景
4. 不适应 pipeline 的场景： 对可靠性要求很高，每次操作都需要立即知道操作是否成功，是否数据已经写进 redis 了

4.3.7 Redis 分区

1. 分区是将数据分割成多个 Redis 实例，每个实例将只包含键子集
2. 分区的好处 ① 它允许更大的数据库，使用多台计算机的内存总和；如果不分区，只是一台计算机有限的内存可以支持的数据存储 ② 它允许按比例在多内核和多个计算机计算，以及网络带宽向多台计算机和网络适配器
3. 分区的劣势 ① 涉及多个键的操作通常不支持。例如，如果它们被存储在被映射到不同的 Redis 实例键，则不能在两个集合之间执行交集 ② 涉及多个键时，Redis 事务无法使用 ③ 分区粒度是一个键，所以它不可能使用一个键和一个非常大的有序集合分享一个数据集 ④ 当使用分区，数据处理比较复杂，比如要处理多个 RDB/AOF 文件，使数据备份需要从多个实例和主机聚集持久性文件
4. 分区类型。假设我们有四个 Redis 实例：R0，R1，R2，R3，分别表示用户如：user:1，user:2，...，，对既定的 key 有多种不同方式来选择这个 key 存放在哪个实例中。 ① 范围分区：指定 key 在某一个范围。如：用户从 ID 为 0 至 10000 将进入实例 R0，而用户 10001 到 20000 将进入实例 R1 等等 ② 哈希分区：用一个哈希函数(例如，模数函数)将键转换为数字数据，然后存储在不同的 Redis 实例

4.3.8 Redis 备份

1. 注意区分 SAVE 命令和 BGSAVE 命令的区别：一个同步，一个异步 SAVE 直接调用 rdbSave ，阻塞 Redis 主进程，直到保存完成为止。在主进程阻塞期间，服务器不能处理客户端的任何请求。 BGSAVE 则 fork 出一个子进程，子进程负责调用 rdbSave ，并在保存完成之后向主进程发送信号，通知保存已完成。 Redis 服务器在 BGSAVE 执行期间仍然可以继续处理客户端的请求。

4.3.9 恢复 Redis 数据

4.3.10 Redis 安全

4.3.9 和 4.3.10 知道有这么回事就好，老师讲的都是直接的操作方式了，详细了解见闫中敏老师的 ppt

4.4 Redis 使用实例

1. 适用场景 数据高并发的读写 海量数据的读写 对扩展性要求高的数据
2. 不适用场景 需要事务支持(关系型数据库) 基于 sql 结构化查询储存，关系复杂
3. 仔细读懂简单的文章投票网站的后端这个例子，注意其中为了实现功能恰好使用的数据结构。加深理解。为什么使用 set，为什么使用 zset？

第五章 MongoDB

作者：杨浩哲

5.1 简介

MongoDB 是一种非关系型的数据库，其存储类型是文档存储

5.1.1 文档存储的特点

保证海量数据存储的同时，具有良好的查询性能。用类 Json 格式进行存储

5.1.2 MongoDB 的简单介绍

- C++语言编写
- 基于分布式文件存储的开源数据库系统
- 在高负载的情况下，添加更多的节点，可以保证服务器性能
- 旨在为 WEB 应用提供可扩展的高性能数据存储解决方案

5.1.3 MongoDB 的存储方式（文档存储）

- 将数据存储为一个文档，数据结构由**键值(key value)对**组成
- **字段值**可以包含**其他文档**，**数组**及**文档数组**

文档类似于 JSON 对象

5.1.4 MongoDB 的特点

- 面向集合存储，易存储对象类型的数据

- 模式自由
- 支持动态查询
- 支持完全索引，包含内部对象
- 支持查询
- 支持复制和故障恢复
- 使用高效的二进制数据存储，包括大型对象（如视频）
- 自动处理碎片，以支持云计算层次的扩展性
- 支持 RUBY，PYTHON，JAVA，C++，PHP，C#等多种语言
- 文件存储格式为 BSON（一种 JSON 的扩展）
- 可通过网络访问

5.1.5 MongoDB 的适用场景

- **网站数据**
  - 非常适合实时的插入，更新与查询
  - 具备网站**实时数据存储**所需的**复制**及**高度伸缩性**
- **缓存**
  - 适合作为信息基础设施的**缓存层**
  - **避免过载**：在系统重启之后，由 Mongo 搭建的**持久化缓存层**可以避免下层的数据源过载
- **大尺寸、低价值的数据**
  - 传统的关系型数据库存储某些数据代价可能会比较昂贵（在此之前，很多时候程序员往往会选择传统的文件进行存储）
- **高伸缩性的场景**
  - 非常适合由数十或数百台服务器组成的数据库
  - Mongo 的路线图中已经包含对 MapReduce 引擎的内置支持（什么是路线图？）
  - 内置对 MapReduce 引擎的支持
- **用于对象及 JSON 数据的存储**
  - Mongo 的 **BSON 数据格式**非常适合**文档化格式的存储及查询**

5.1.6 MongoDB 不适用的场景

- 高度事务性的系统

如银行或会计系统。传统的关系型数据库目前还是更适用于需要大量原子性复杂事务的应用程序

- 传统的商业智能应用
  - 针对特定问题的 BI 数据库会产生高度优化的查询方式
  - 数据仓库可能是更合适的选择
- 需要 SQL 的问题

5.2 数据模型

SQL 术语/概念	MongoDB 术语/概念	解释/说明
database	database	数据库
table	collection	数据库表/集合
row	document	数据记录行/文档
column	field	数据字段/域
table join	无	表连接（MongoDB 不支持）
primary key	primary key	主键，MongoDB 自动将_id 字段设置为主键

Mongodb 中基本的概念是文档、集合、数据库

5.2.1 SQL->MongoDB

id	user_name	email	age	city
1	mark hanks	mark@abc.com	25	Los Angeles
2	Richard Peter	richard@abc.com	31	Dallas

=>

```
{
  "_id":ObjectId("146bb52dE524270060001f"),
  "age":25,
  "city":"Los Angeles",
  "email":"mark@abc.com",
  "user_name":"mark hanks"
}
{
  "_id":ObjectId("bb52dE5242700E0001"),
  "age":31,
  "city":"Dallas",
  "email":"richard@abc.com",
  "user_name":"Richard Peter"
}
```

注意："_id"在集合内唯一标识一行

5.2.2 MongoDB 中的数据类型（了解）

除了普通的 String，Integer，Boolean，Double，Arrays（各种类型的数组），TimeStamp（时间戳），Date 外，还提供了一些特殊的格式：

数据类型	描述
Min/Max keys	将一个值与 BSON（二进制的 JSON）元素的最低值和最高值相对比
Object	用于内嵌文档
Null	用于创建空值
Symbol	符号，基本等同于字符串类型，但一般采用特殊符号类型的语言
Object ID	对象 ID，用于创建文档的 ID
Binary Data	二进制数据

数据类型	描述
Code	用于存储 js 代码
Regular expression	用于存储正则表达式

5.3 MongoDB 操作（了解）

5.3.1 连接

- shell 连接

连接本地数据库服务器，端口是默认的。

```
mongodb://localhost
```

使用用户名fred，密码foobar登录localhost的admin数据库。

```
mongodb://fred:foobar@localhost
```

使用用户名fred，密码foobar登录localhost的baz数据库。

```
mongodb://fred:foobar@localhost/baz
```

连接 replica pair, 服务器1为example1.com服务器2为example2。

```
mongodb://example1.com:27017,example2.com:27017
```

连接 replica set 三台服务器 (端口 27017, 27018, 和27019):

```
mongodb://localhost,localhost:27018,localhost:27019
```

连接 replica set 三台服务器, 写入操作应用在主服务器 并且分布查询到从服务器。

```
mongodb://host1,host2,host3/?slaveOk=true
```

直接连接第一个服务器，无论是replica set一部分或者主服务器或者从服务器。

```
mongodb://host1,host2,host3/?connect=direct;slaveOk=true
```

当你的连接服务器有优先级，还需要列出所有服务器，你可以使用上述连接方式。

安全模式连接到localhost:

```
mongodb://localhost/?safe=true
```

以安全模式连接到replica set，并且等待至少两个复制服务器成功写入，超时时间设置为2秒。

```
mongodb://host1,host2,host3/?safe=true;w=2;wtimeoutMS=2000
```


5.3.2 增删改查

- 查看所有数据库：`show dbs`
- 创建：`use [database_name]`

如果没有创建会自动创建，但这个时候使用 `show dbs` 是看不到刚创建的数据库的，必须真正的插入了数据后该数据库才会被创建

- 删除：`db.dropDatabase()`删除当前的数据库
- 集合（表）操作：创建集合，获取某集合，查看所有集合，删除集合（这里集合基本等同于表）
- 插入：`db.[collection_name].insert(...something...)`
- 查看：`db.[collection_name].find()`
- 条件操作符：`$type`
- 还有其他的赋值，更新，删除，查询等操作..

- 聚合操作

具体可以查看菜鸟教程
（http://www.runoob.com/mongodb/mongodb-tutorial.html）

5.3.3 子文档的存储方法

有嵌入式文档和引用式文档两种类型

嵌入式文档	引用式文档

"_id": ObjectId("52ttc33cd85242t43E000001"),
"contact":"987654321",
"dob":3
"address":[
{
"building":"22A,Indiana apt",
"pincode":123456
},
{
"building":"341A,Acropolis Apt",
"pincode":786543
}
]
	"_id": ObjectId("52ttc33cd85242t43E000001"),
	"contact":"987654321",
	"dob":3
	"address":[
	ObjectId("52ttc33cd85242t43E000041"),
	ObjectId("52ttc33cd85242t43E000021"),
	
	]

可以直接查询到详细信息（但是有可能会存在冗余信息）	如果查询，需要两次查询，第一次查询ObjectId，第二次查询ID来获取详细信息

5.3.4 索引

MongDB 支持按某个字段，包括内嵌字段进行排序。

- 如果没有索引，MongoDB 在读取数据时必须扫描集合中的每个文件并选取那些符合查询条件的记录
- 而扫描全集合的查询效率是非常低的，特别在处理大量的数据时，查询可以要花费几十秒甚至几分钟，这对网站的性能是非常致命的

具体操作（索引的删除和建立方式）参考菜鸟教程-MongoDB 索引

索引的查询限制:

- 索引不能被包含正则表达式，非操作符，算术运算符，$where 子句的查询使用
- 一个集合中索引不能超过 64 个
- 索引名长度不超过 125 个字符
- 一个复合索引最多有 31 个字段

通过 explain 命令可以查看语句是否使用索引

5.3.5 JAVA 连接

略

5.4 数据复制

复制是将数据同步在多个服务器的过程

5.4.1 复制的作用

- 保障数据的安全性
- 数据高可用性（24*7）
- 灾难恢复
- 无需停机维护（如备份，重建索引，压缩）
- 分布式读取数据

5.4.2 复制需求

- 复制**最少**需要两个节点：一个是主节点，负责处理客户端请求，其余均为从节点，负责复制主节点上的数据。
- MongoDB 搭配中一主一从和一主多从均可

5.4.3 数据复制原理

主节点记录在其上的所有操作生成**操作日志（oplog）**，从节点**定期轮询**主节点获取这些操作，然后对自己的数据副本执行这些操作，从而保证从节点的数据与主节点一致

5.4.4 MongoDB 的副本集特性

副本集在主机宕机后，副本会接管主节点成为主节点而不会宕机（常见的主从模式在主机宕机后所有服务将停止）

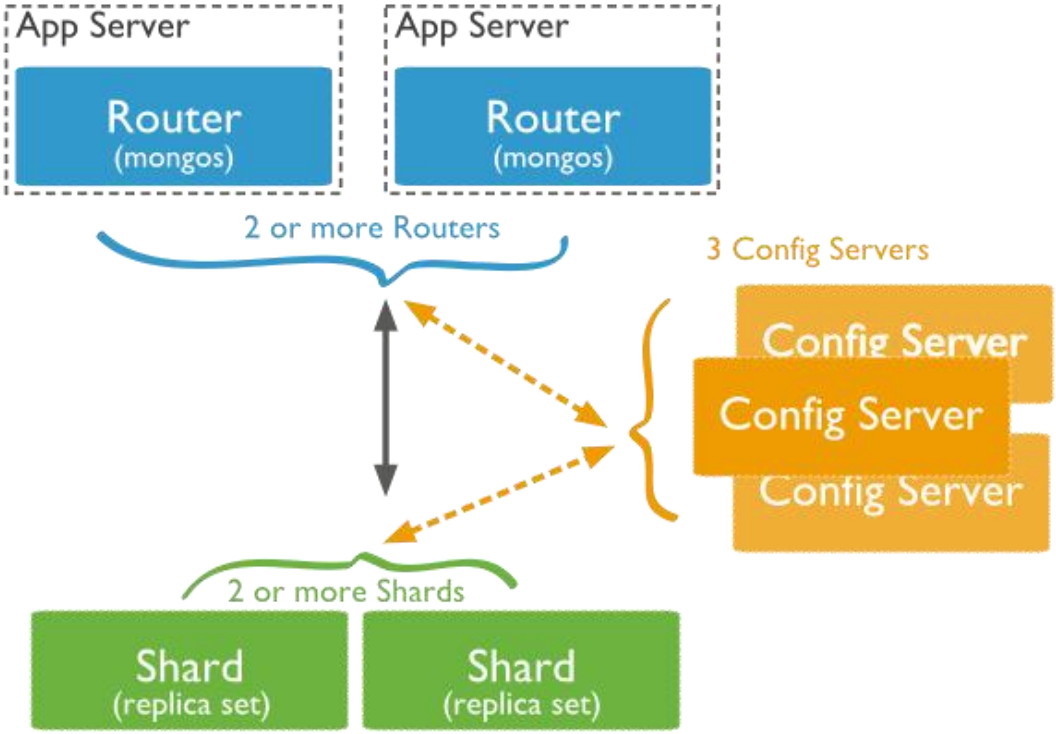
5.5 分片

当 MongoDB 存储海量的数据时，一台机器可能不足以存储数据，也可能不足以提供可接受的读写吞吐量

5.5.1 使用分片的原因

- 复制所有的写入操作到主节点
- 延迟的敏感数据会在主节点查询
- 单个副本集限制在 12 个节点
- 请求量巨大时会出现内存不足
- 本地磁盘不足而垂直扩展价格昂贵

5.5.2 MongoDB 的分片集群结构



Shard：用于存储实际的数据块，实际生产环境中一个 shard server 角色可由几台机器组各一个 replica set 承担，防止主机单点故障 **Config Server**：mongod 实例，存储了整个 ClusterMetadata，其中包括 chunk 信息。**Query Routers**：前端路由，客户端由此接入，且让整个集群看上去像单一数据库，前端应用可以透明使用。

5.6 MongoDB 的其他特性（了解）

5.6.1 数据备份

mongodump 命令来备份 MongoDB 数据，导出所有数据到指定目录中

```
mongodump -h dbhost -d dbname -o dbdirectory
```

5.6.2 数据恢复

mongorestore 命令来恢复备份的数据

```
mongorestore -h dbhost -d dbname --directoryperdb dbdirectory
```

5.6.3 运行情况监控

mongostat 命令

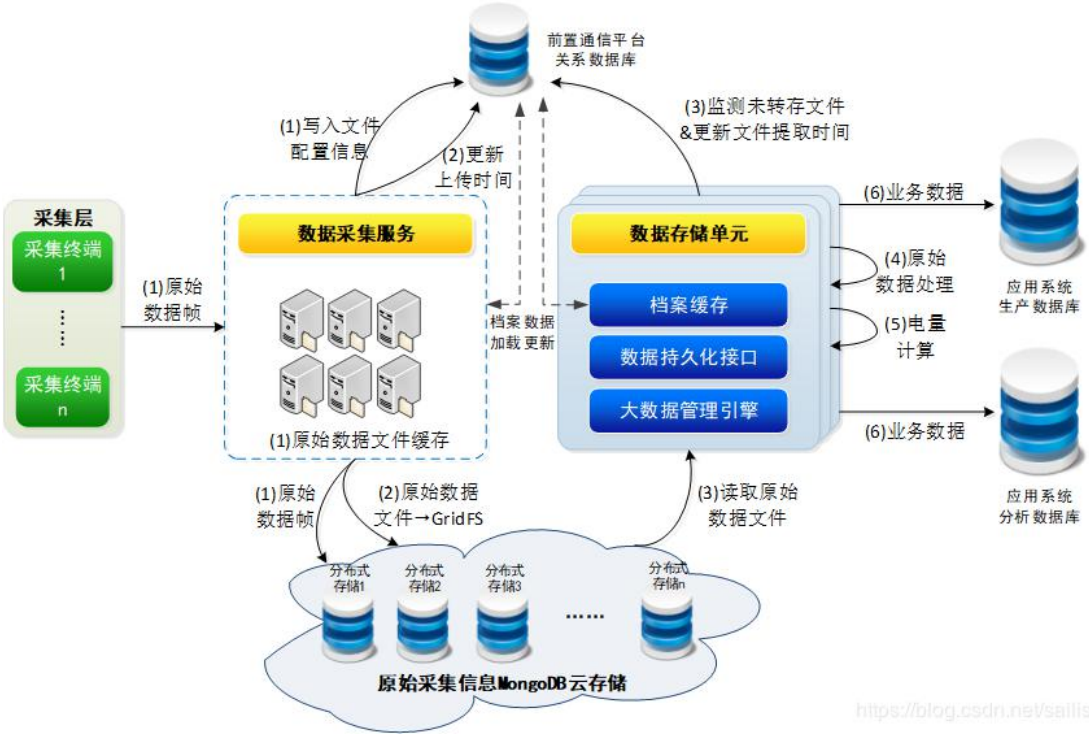
- mongostat 是 Mongodb 自带的状态检测工具，在命令行下使用
- 它会间隔固定时间获取 mongodb 的当前运行状态，并输出
- 如果发现数据库突然变慢或者有其他问题的话，第一手的操作就考虑采用 mongostat 来查看 mongo 的状态

mongotop 命令

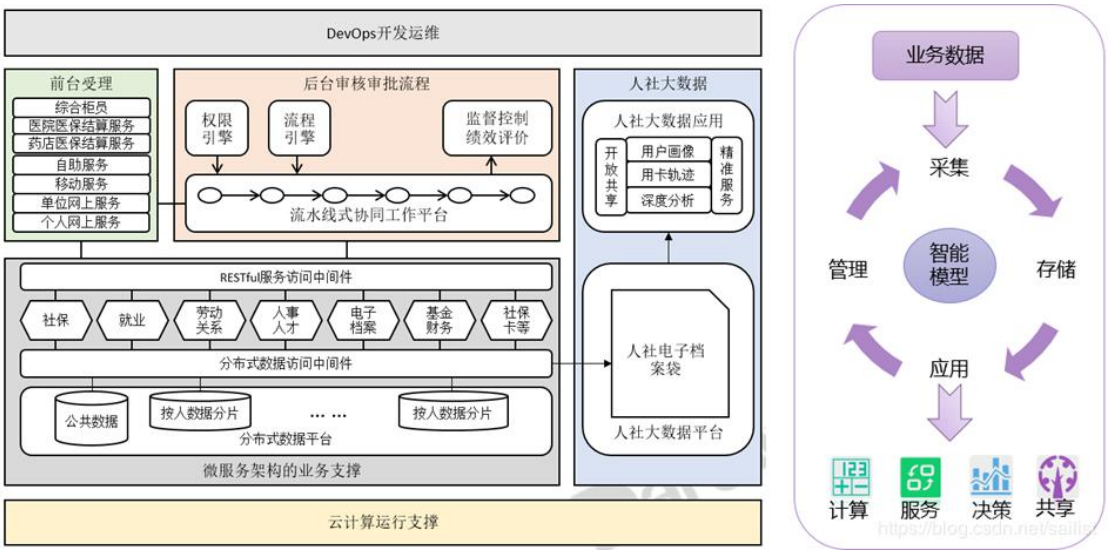
- mongotop 提供了一个方法，用来跟踪一个 MongoDB 的实例，查看哪些大量的时间花费在读取和写入数据
- mongotop 提供每个集合的水平的统计数据
- 默认情况下，mongotop 返回值的每一秒

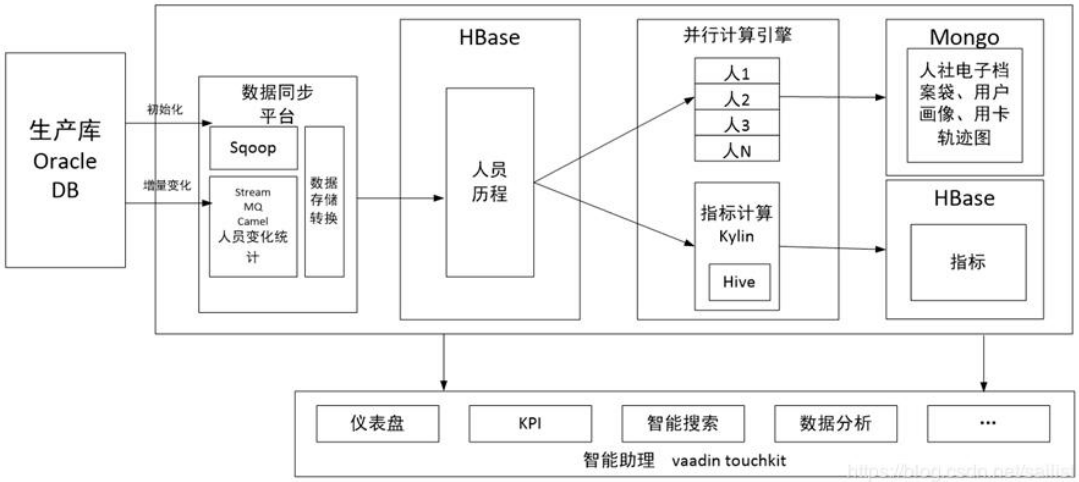
5.7 MongoDB 应用案例

5.7.1 用电信息采集



5.7.2 人社大数据平台





第六章 Neo4j

作者:乔艺萌

6.1 Neo4j 简介

6.1.1 产生原因

关系数据库和其他 NoSQL 数据库缺乏联系的表达，图数据库拥抱联系

6.1.2 简介

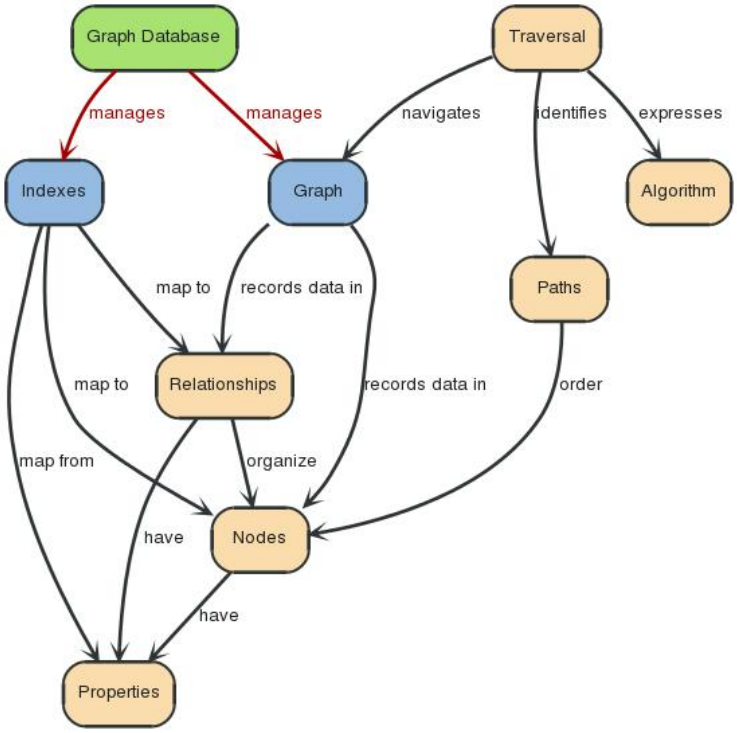
1. 类型：NoSQL 图形数据库
2. 定义：Neo4j 使用图相关的概念来描述数据模型，把数据保存为图中的节点以及节点之间的关系
3. 两个基本概念——节点和边：节点表示实体，边表示实体之间的关系；节点和边都可以有自己的属性；不同实体通过各种不同的关系关联起来，形成复杂的对象图
4. 查找和遍历主要使用深度优先搜索和广度优先搜索两种方式

6.1.3 特点

1. 完整的 ACID 支持
2. 高可用性
3. 轻易扩展到上亿级别的节点和关系
4. 通过遍历工具高速检索数据

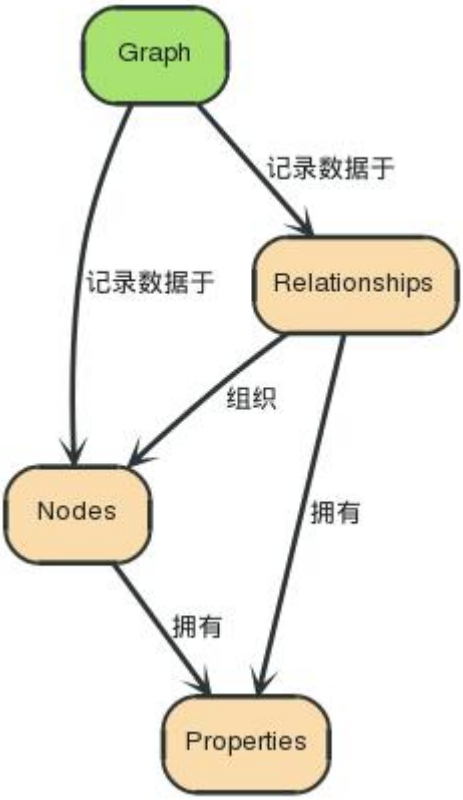
6.2 Neo4j 数据模型

6.2.1 概览

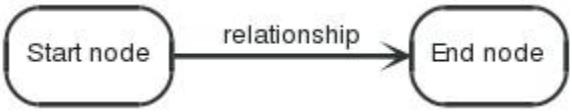


6.2.2 基本概念

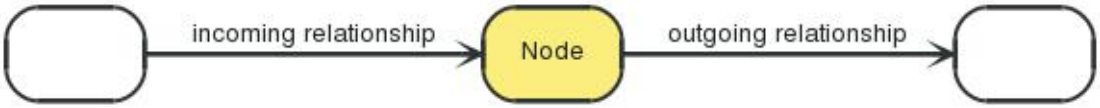
1. 图



2. 节点
   - 节点经常被用于表示实体，可以有属性
   - 最简单的节点只有一个属性，如 `name: Marko`，只有名字一个属性
3. 关系
   - 一个关系连接两个节点，必须有一个开始节点和结束节点
   - 关系也可以有属性

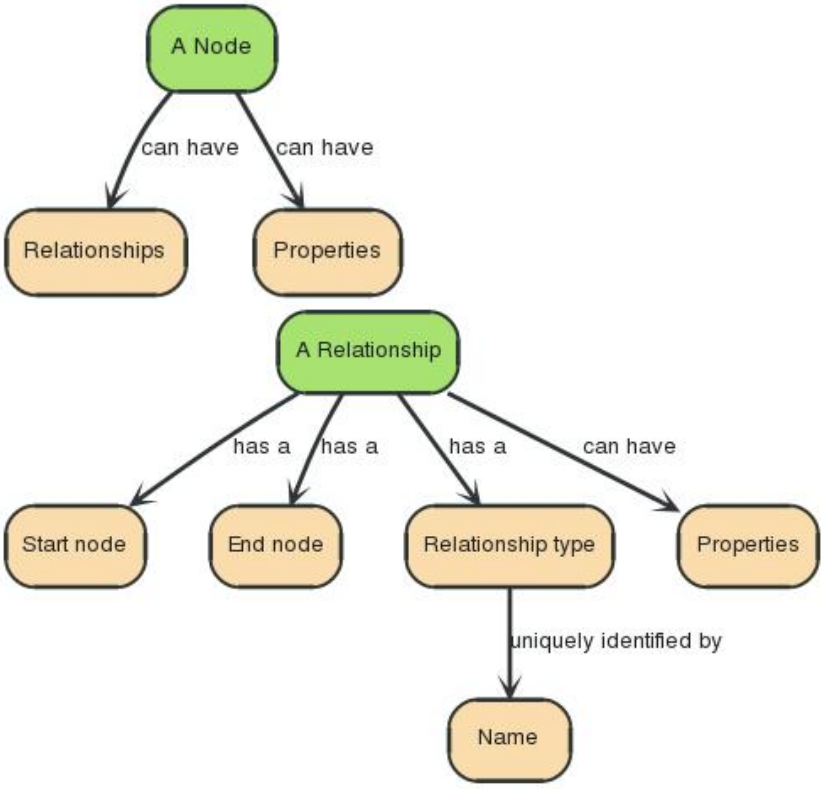


   - 关系总是直接相连的，所以对于一个节点来说，与他关联的关系看起来有输入/输出两个方向

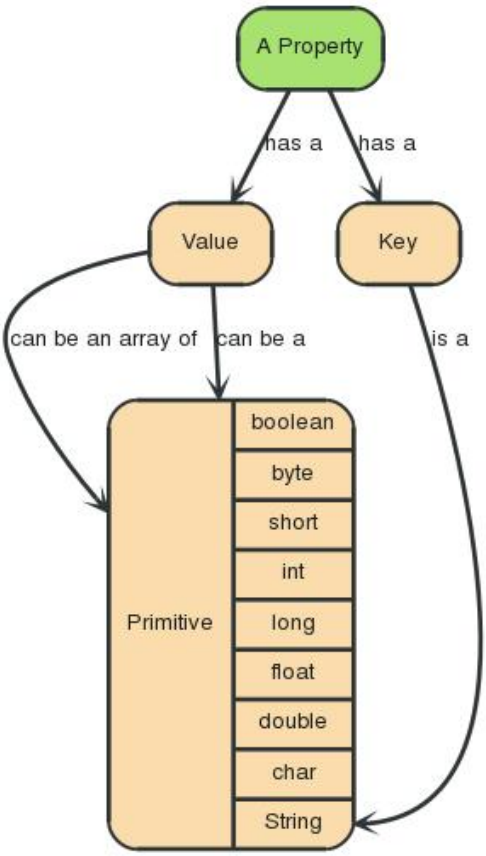


   - 通过 `outgoing` 关系进行正向搜索，通过 `incoming` 关系进行逆向搜索，可以找到很多关联数据

4. 节点、关系和属性

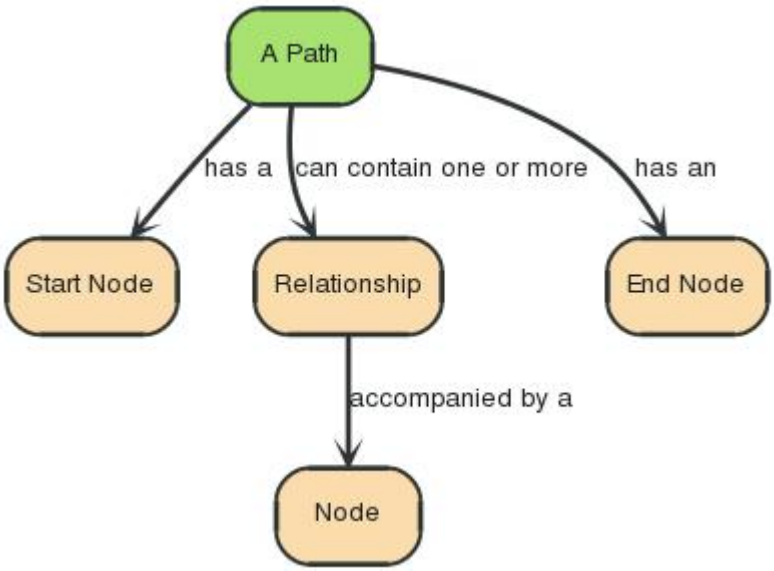


5. 属性
   由键值对组成：键名是字符串，属性值要么是原始值，要么是原始值类型的一个数组



6．路径

路径由至少一个节点，通过各种关系连接组成，经常是作为一个查询或者遍历的结果



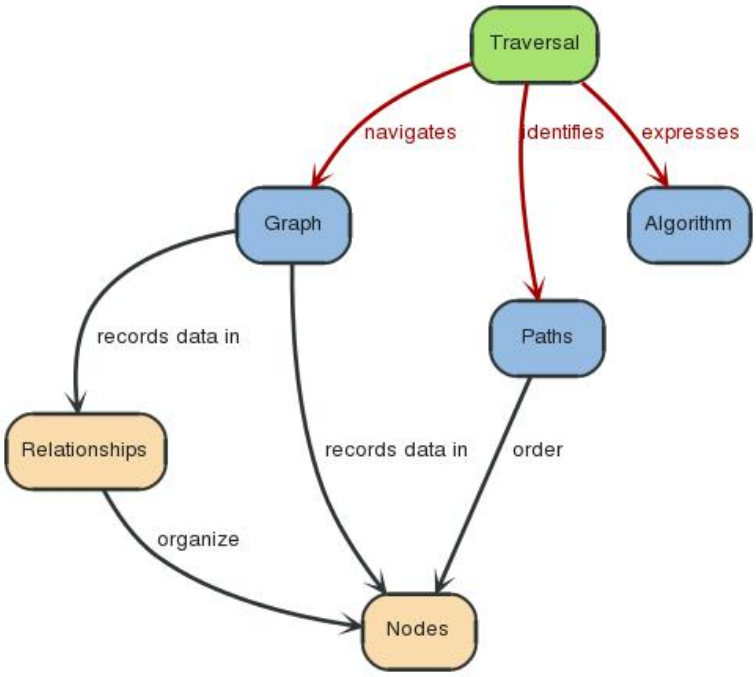
7．遍历

定义：遍历一张图就是按照一定的规则，跟随他们的关系，访问关联的节点集合

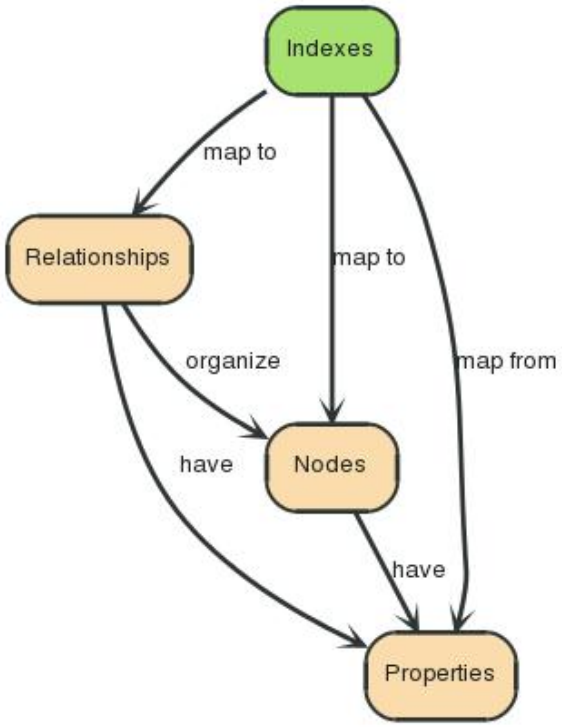
要素：

- 遍历的路径：通常用关系的类型和方向来表示

- 遍历的顺序：常见的遍历顺序有深度优先和广度优先两种
- 遍历的唯一性：可以指定在整个遍历中是否允许经过重复的节点、关系或路径
- 遍历过程的决策器：用来在遍历过程中判断是否继续进行遍历，以及选择遍历过程的返回结果
- 起始节点：遍历过程的起点



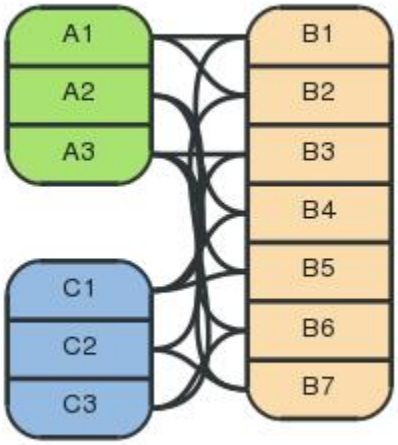
8．索引



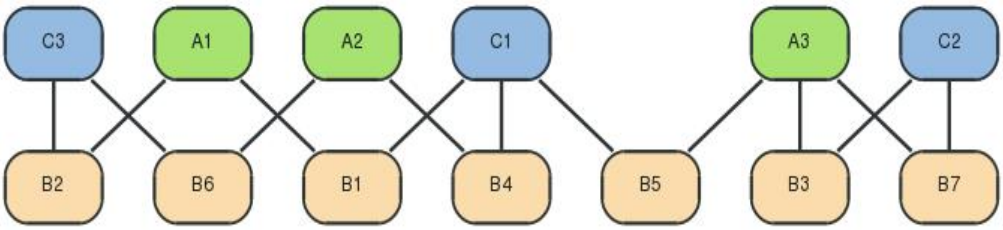
6.2.3 其他数据库与 Neo4j 的转换

1. 关系数据库->Neo4j

关系数据库

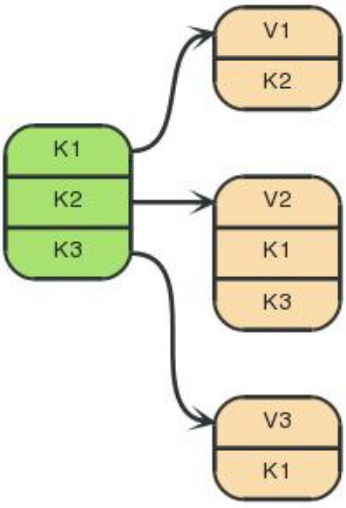


Neo4j

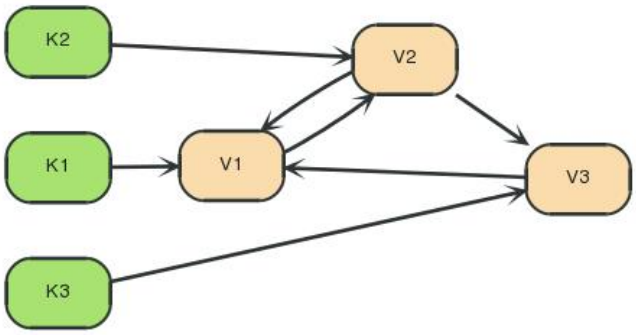


2. 键值对型数据库->Neo4j

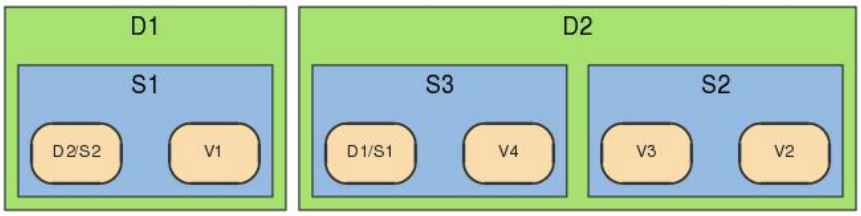
关系数据库



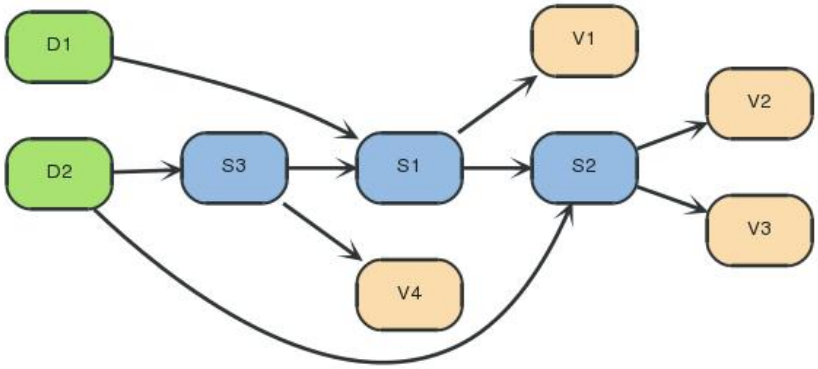
Neo4j



3. 文档型数据库->Neo4j
关系数据库



Neo4j



6.3 Neo4j 应用

/6.3.1 注意区分模型（schema）和实例（instance）（重点）

以老师上课的例子为例

/

1. 要求描述数据库结构/schema 时，描述节点和边的构成：
   节点：A、B、C
   边：A->B、A->C、C->B
2. 要求描述数据库实例时（a1、b1、c1 分别是 A、B、C 的某个实例值）：

/

6.3.2 schema 描述方式举例——社交网络

/
Person
friend: 连接两个不同 Person 实例的关系（不能连接自己）
status: 连接到最近的 StatusUpdate
statusUpdate
next: 指向在主线上的下一个 StatusUpdate，是在当前这个状态更新之前发生的

第七章 其他数据库

作者：乔艺萌

本章了解即可，没有需要掌握的内容
数据库分类
/

7.1 其他 NoSQL 数据库

NoSQL 数据库比较/
几种非关系型数据库

- DynamoDB （键值对）
- Cassandra （混合型）
- CouchDB （文档型）
- HyperGraphDB （图）

7.2 NewSQL 数据库

7.2.1 含义

NewSQL 是对各种新的可扩展/高性能数据库的简称

- 具有 NoSQL 对海量数据的存储管理能力
- 保持了传统数据库支持 ACID 和 SQL 等特性

7.2.2 特点

- 支持关系数据模型
- 使用 SQL 作为其主要接口

7.2.3 举例——VoltDB

7.3 云数据库

定义：云数据库是指被优化或部署到一个虚拟计算环境中的数据库

7.4 各种数据库的选择

/Q：使用 NoSQL 还是关系数据库？ A：没有绝对的回答，需要根据应用场景来决定到底用什么 建议：

- 如果关系数据库在你的应用场景中，完全能够很好的工作，而你又是非常善于使用和维护关系数据库的，那么完全没有必要迁移到 NoSQL 上。如果是在金融，电信等以数据为王的关键领域，目前使用的是Oracle数据库来提供高可靠性的，除非遇到特别大的瓶颈，不然也别贸然尝试 NoSQL。
- 如果在你的应用场景中有以下情况，可以尝试使用 NoSQL 数据库：数据库表模式经常变化；对于数据一致性要求很低，要求高并发读写；业务模型不是非常复杂，事务少，大表关联少；数据量大，而服务器等硬件资源有限，也无法承受 Oracle 等解决方案的高成本。
- NoSQL 和关系数据库结合使用，各取所长，需要使用关系特性的时候使用关系数据库，需要使用 NoSQL 特性的时候使用 NoSQL 数据库，各得其所。
- NoSQL 和关系数据库结合使用；NoSQL 作为缓存服务器；NoSQL 代替关系数据库。
- 规避 NoSQL 数据库风险。虽然 NoSQL 数据库能带来很多便利，但是在应用的时候也要考虑风险。NoSQL 数据库本身也需要进行备份（冷备和热备）。或者可以考虑使用两种 NoSQL 数据库，出现问题后可以进行切换。

==========

感谢对智库的支持，如发现本知识见解有错误或对结构内容有你的观点，欢迎扫码给我们留言。

扫码关注公众号
获取最新版本和更多科目知识见解