软件架构与中间件

涂志莹 tzy_hit@hit.edu.cn

苏统华 thsu@hit.edu.cn

哈尔滨工业大学

软件架构与中间件

Software Architecture and Middleware

第4章

数据层的软件架构技术

第4章 数据层的软件架构技术

4.1 数据驱动的软件架构演化

4.2 数据读写与主从分离

4.3 数据分库分表

4.4 数据缓存

4.5 非关系型数据库

4.6 数据层架构案例

4.4
数据缓存

1、数据缓存的基本概念

2、本地缓存

3、分布式缓存

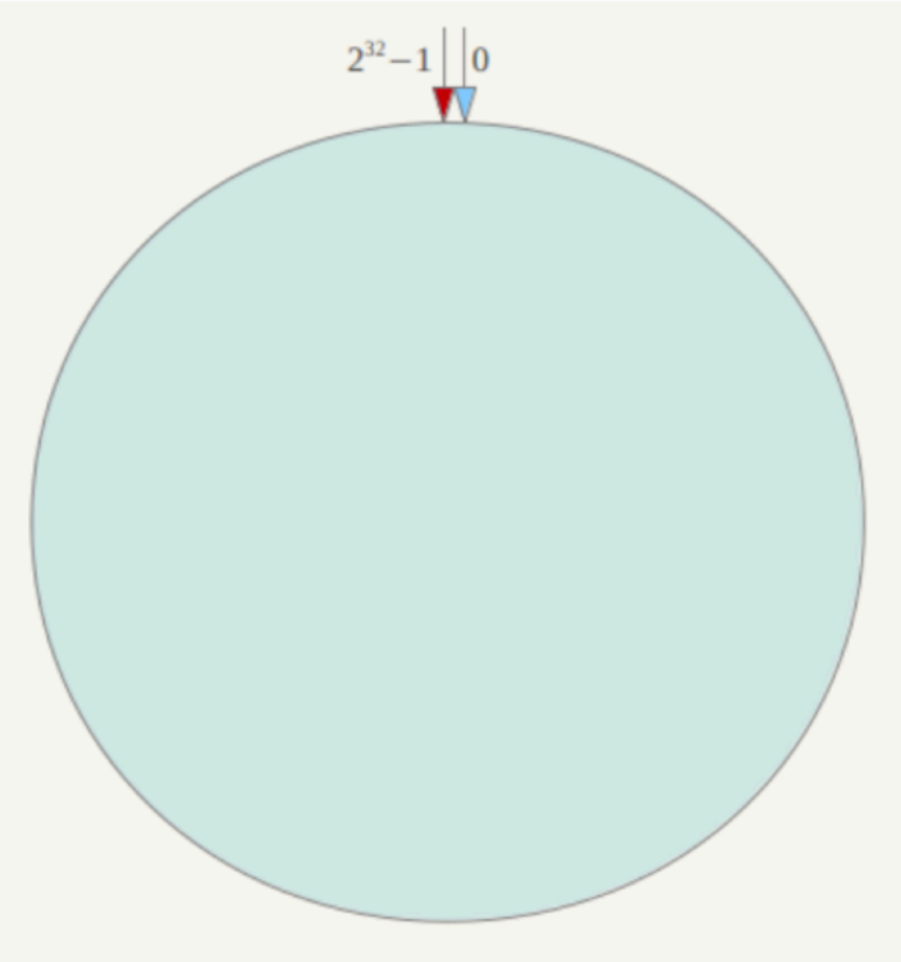
4、缓存问题讨论

4.4.3 分布式缓存

- 分布式缓存的关注点
- Redis

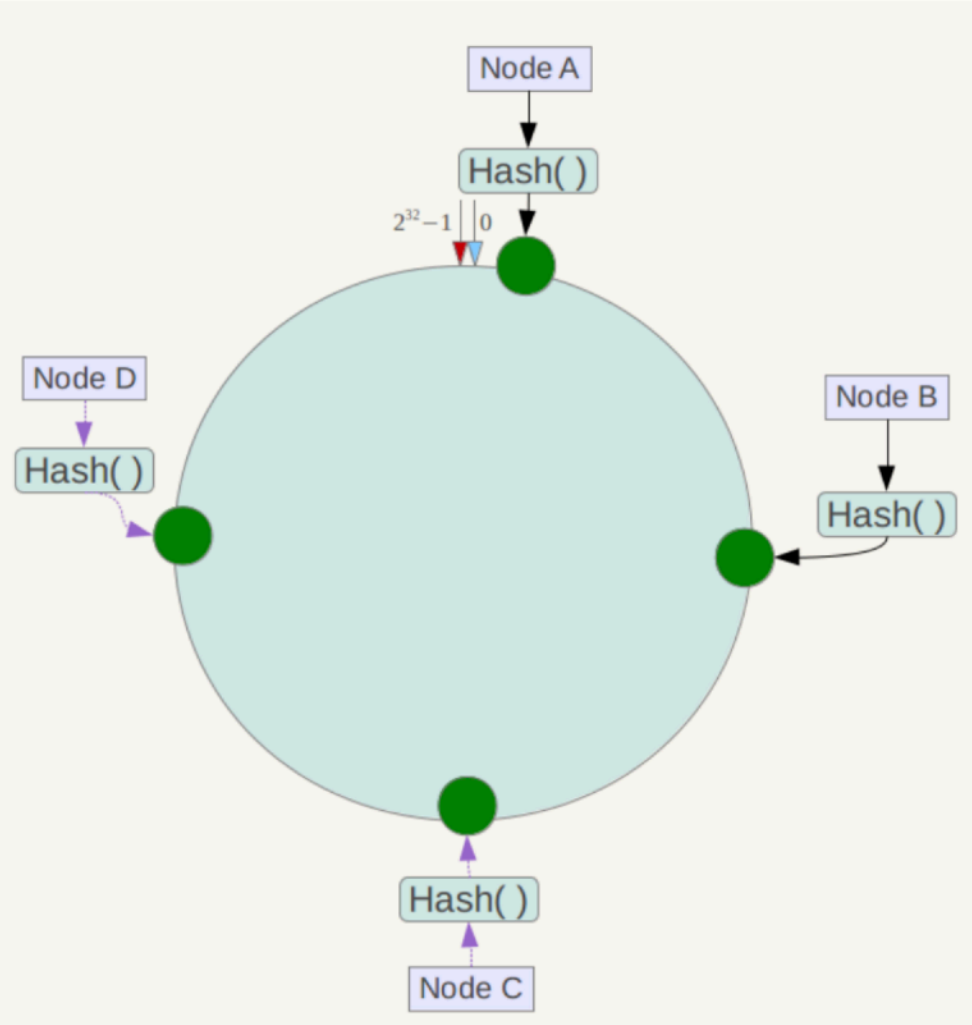
分布式缓存的关注点

分布式缓存的迁移-一致性哈希



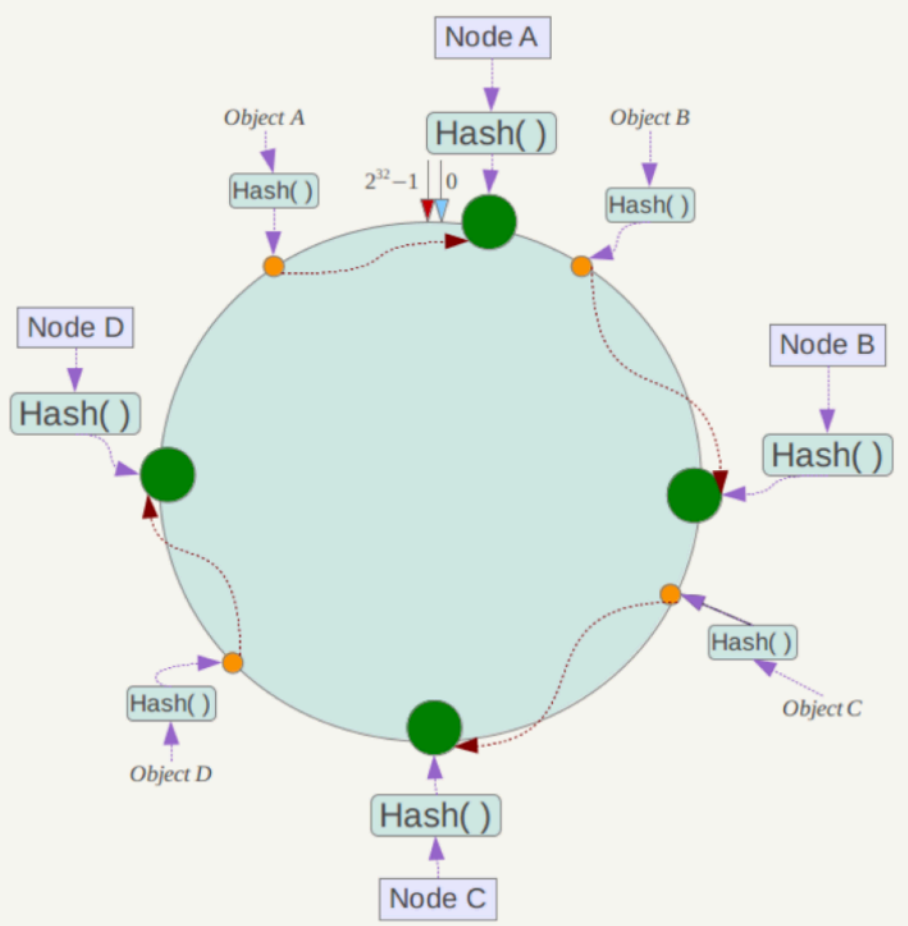
一致性哈希将整个哈希值空间组织成一个虚拟的圆环（哈希环），如假设某哈希函数H的值空间为0-2^{32}-1（即哈希值是一个32位无符号整形），整个空间按顺时针方向组织。0和2^{32}-1在零点钟方向重合

分布式缓存的迁移-一致性哈希



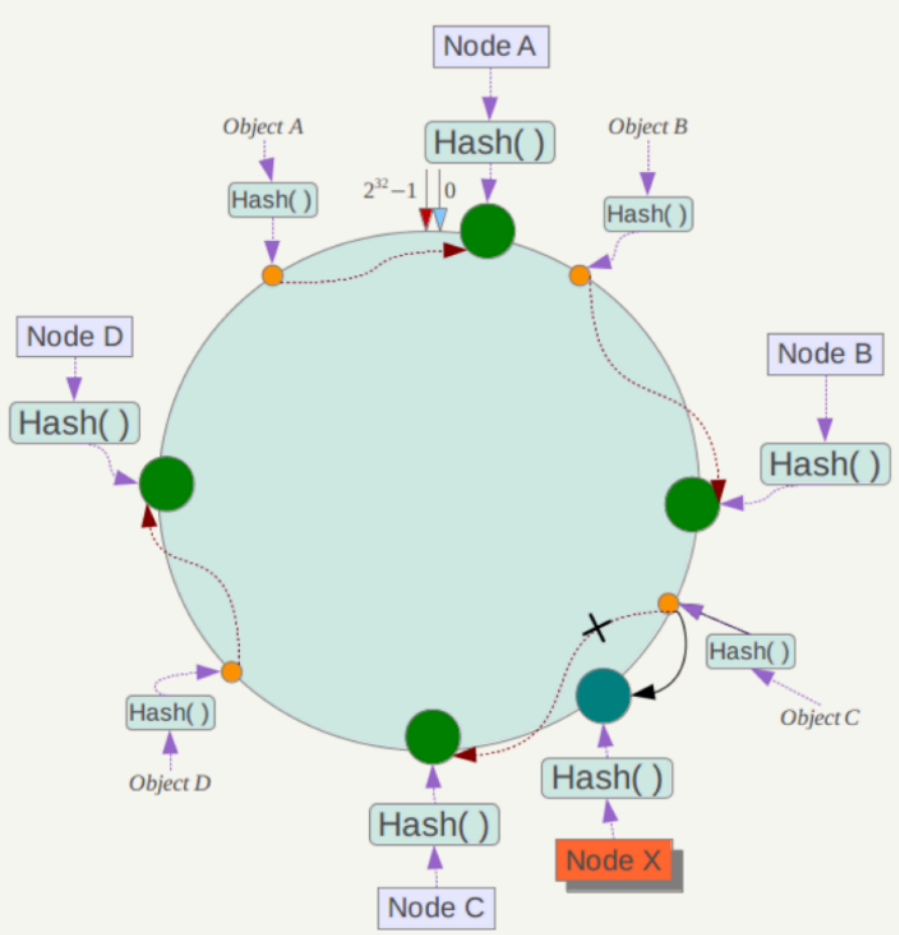
将各个服务器使用Hash进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置

分布式缓存的迁移-一致性哈希



将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器

分布式缓存的迁移-一致性哈希

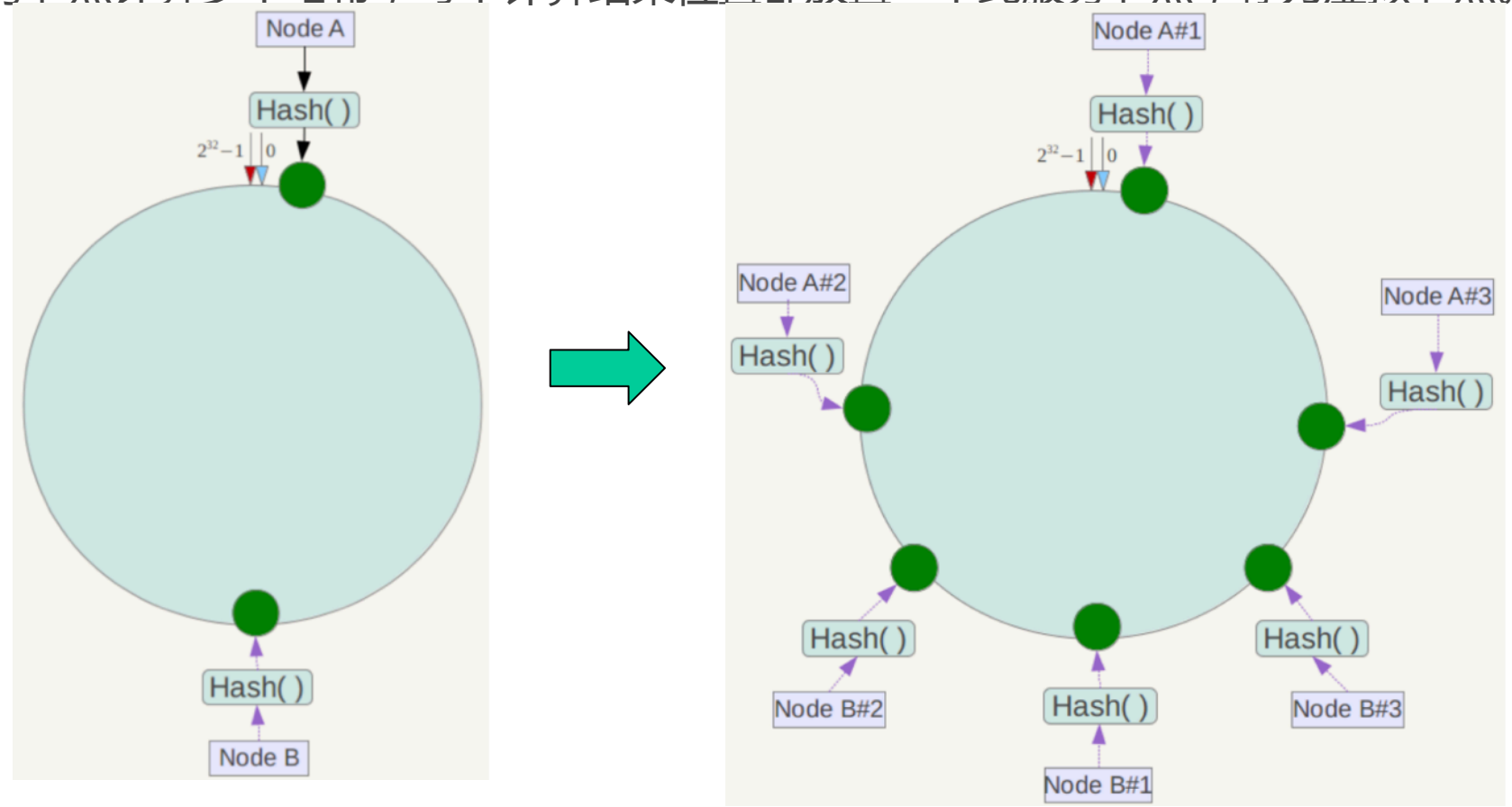


分析一致性哈希算法的容错性和可扩展性：

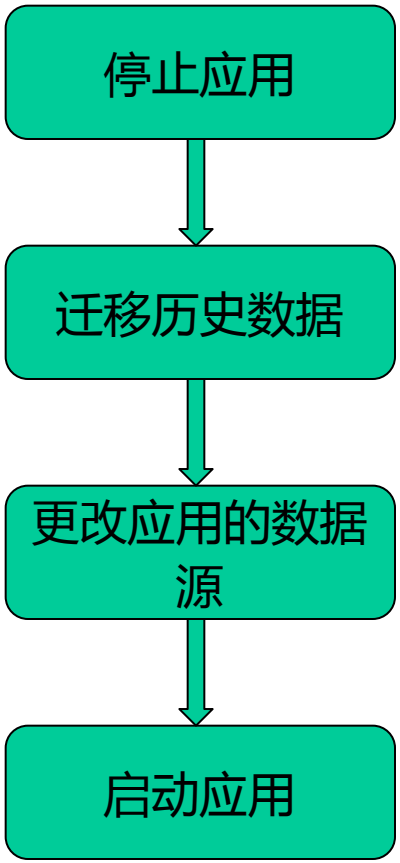
- 现假设Node C不幸宕机，可以看到此时对象A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。
- 如果在系统中增加一台服务器Node X，此时对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X 。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

分布式缓存的迁移-一致性哈希

- 一致性哈希算法在服务节点太少时，容易因为节点分布不均匀而造成数据倾斜问题。此时必然造成大量数据集中到一个节点上。
- 为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。



分布式缓存的迁移-停机迁移



- 1、停机应用，先将应用停止服务
- 2、迁移历史数据，按照新的规则把历史数据迁移到新的缓存数据集群中
- 3、更改应用的数据源配置，指向新的缓存集群
- 4、重新启动应用

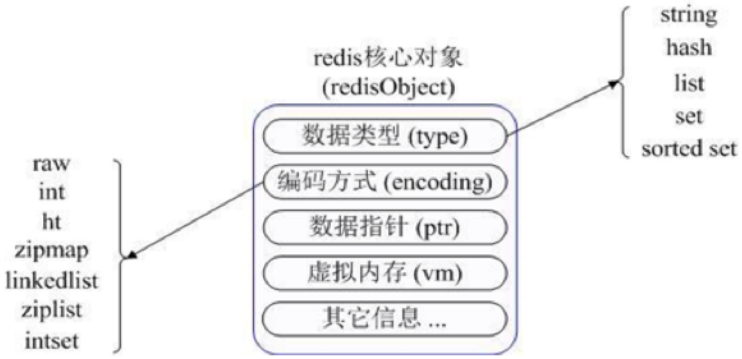
- 该方式简单，高效，能够有效避免数据的不一致，但需要由业务方评估影响，一般在晚上访问量较小，或者非核心服务的场景下比较适用。

Redis

基本介绍

- Redis是一个远程内存数据库（非关系型数据库），性能强劲，具有复制特性以及解决问题而生的独一无二的数据模型。它可以存储键值对与5种不同类型的值之间的映射，可以将存储在内存的键值对数据持久化到硬盘，可以使用复制特性来扩展读性能
- Redis还可以使用客户端分片来扩展写性能。内置了 复制（replication），LUA脚本（Lua scripting），LRU驱逐事件（LRU eviction），事务（transactions）和不同级别的 磁盘持久化（persistence），并通过 Redis哨兵（Sentinel）和自动分区（Cluster）提供高可用性（high availability）

数据模型



- Redis内部使用一个redisObject对象来标识所有的key和value数据，其中包括：
  - type代表一个value对象具体是何种数据类型
  - encoding是不同数据类型在Redis内部的存储方式，比如——type=string代表value存储的是一个普通字符串，那么对应的encoding可以是raw或是int，如果是int则代表Redis内部是按数值类型存储和表示这个字符串。

- 对象的编码方式：
  - 字符串可以被编码为raw（一般字符串）或Rint（为了节约内存，Redis会将字符串表示的64位有符号整数编码为整数来进行储存）；
  - 列表可以被编码为ziplist或linkedlist，ziplist是为节约大小较小的列表空间而作的特殊表示；
  - 集合可以被编码为intset或者hashtable，intset是只储存数字的小集合的特殊表示；
  - hash表可以编码为zipmap或者hashtable，zipmap是小hash表的特殊表示；
  - 有序集合可以被编码为ziplist或者skiplist格式，ziplist用于表示小的有序集合，而skiplist则用于表示任何大小的有序集合。

数据淘汰策略

- Volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
- Volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
- Volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
- Allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
- Allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
- No-enviction（驱逐）：禁止驱逐数据

数据淘汰内部实现

- 消极方法：
  - 在主键被访问时如果发现它已经失效，那么就删除它
  - 触发时机：在实现GET，MGET，HGET，LRANGE等所有涉及到读取数据的命令时都会调用
- 积极方法：
  - 周期性地从设置了失效时间的主键中选择一部分失效的主键删除
  - 触发时机：Redis的时间事件，即每隔一段时间就中断一下完成一些指定操作

持久化方法

- RDB（Redis DataBase）
  - 默认的持久化方案，数据库的快照（snapshot）以二进制的方式定时保存到磁盘中
- AOF（Append Only File）
  - 以协议文本的方式，将所有对数据库进行过写入的命令（及其参数）记录到AOF文件，以此达到记录数据库状态的目的

4.4
数据缓存

1、数据缓存的基本概念

2、本地缓存

3、分布式缓存

4、缓存问题讨论

4.4.4 缓存的问题

- 数据一致性
- 缓存穿透
- 缓存雪崩
- 缓存高可用
- 缓存热点

数据一致性

- 因为缓存属于持久化数据的一个副本，所以不可避免的会出现数据不一致问题，如脏读或读不到数据的情况。
- 数据不一致，一般是因为网络不稳定或节点故障导致问题出现的常见3个场景以及解决方案：
  - 先写缓存，再写数据库：【描述】缓存写成功，但写数据库失败或响应延迟，则下次读取（并发读）缓存时，就出现脏读；【解决】这个写缓存的方式，本身就是错误的，需要改为先写持久化介质，再写缓存的方式
  - 先写数据库，再写缓存：【描述】写数据库成功，但写缓存失败，则下次读取（并发读）缓存时，则读不到数据；【解决1】根据写入缓存的响应来进行判断，如果缓存写入失败，则回滚数据库操作。该方法增加了程序的复杂度；【解决2】缓存使用时，假如读缓存失败，先读数据库，再回写缓存
  - 缓存异步刷新：【描述】指数据库操作和写缓存不在一个操作步骤中，比如在分布式场景下，无法做到同时写缓存或需要异步刷新；【解决】根据日志中用户刷新数据的时间间隔，以及针对数据可能产生不一致的时间，进行同步操作

缓存穿透

- 缓存穿透指的是使用不存在的key进行大量的高并发查询，这导致缓存无法命中，每次请求都要穿透到后端数据库系统进行查询，使得数据库压力过大，甚至导致数据库服务崩溃。
- 解决方案：
  - 通常将空值缓存起来，再次接收到同样的查询请求时，若命中缓存并值为空，就会直接返回，不会透传到数据库，避免缓存穿透
  - 对恶意的查询攻击，可以对查询条件设置规则，不符合条件产生规则的直接拒绝

缓存并发

- 缓存并发的问题通常发生在高并发的场景下，当一个缓存key过期时，因为访问这个缓存key的请求量较大，多个请求同时发现缓存过期，因此多个请求会同时访问数据库来查询最新数据，并且回写缓存，这样会造成应用和数据库的负载增加，性能降低，由于并发较高，甚至会导致数据库崩溃。

缓存并发-解决方案

- 分布式锁：
  - 使用分布式锁，保证对于每个key同时只有一个线程去查询后端服务，其他线程没有获得分布式锁的权限，因此只需要等待即可。该方式将高并发的压力转移到了分布式锁，因此对分布式锁的考验很大。
- 本地锁：
  - 与分布式锁类似，通过本地锁的方式来限制只有一个线程去数据库中查询数据，而其他线程只需等待，等前面的线程查询到数据后再访问缓存。但是，这种方法只能限制一个服务节点只有一个线程取数据库中查询，如果一个服务有多个节点，则会有多个数据库查询操作，也就是说在节点数量较多的情况下并没有完全解决缓存并发的问题。

缓存并发-解决方案（续）

- 软过期：
  - 软过期指对缓存中的数据设置失效时间，就是不使用缓存服务提供的过期时间，而是业务层在数据中存储过期时间信息，由业务程序判断是否过期并更新，在发现了数据即将过期时，将缓存的时效延长，程序可以派遣一个线程去数据库中获取最新的数据，其他线程会先继续使用旧数据并等待，直至派遣线程获取最新数据后再更新缓存。
  - 也可以通过异步更新服务来更新设置软过期的缓存，这样应用层就不用关心缓存并发的问题。

缓存雪崩

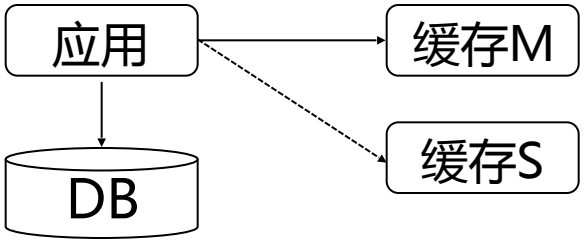
- 缓存雪崩指缓存服务器重启或者大量缓存集中在某一个时间段内失效，业务系统需要重新生成缓存，给后端数据库造成瞬时的负载升高的压力，甚至导致数据库崩溃。

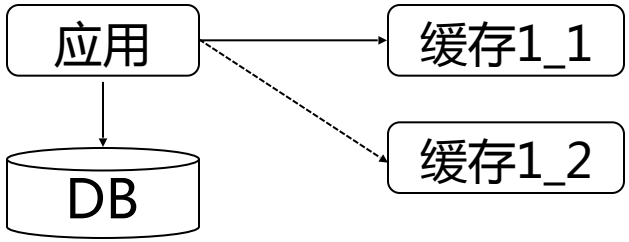
缓存雪崩-解决方案

- 更新锁机制：对缓存更新操作进行加锁保护，保证只有一个线程能进行缓存更新
- 失效时间分片机制：对不同的数据使用不同的失效时间，甚至对相同的数据、不同的请求使用不同的失效时间
  - 例如，当对缓存的user数据中的每个用户的数据设置不同的缓存过期时间，可以定义一个基础时间，如10秒，然后加上一个两秒以内的随机数，则过期时间为10～12秒，这样就可以避免雪崩。
- 后台更新机制：由后台线程来更新缓存，并不是业务线程来更新缓存
  - 后台线程除了定时更新缓存，还要频繁去读取缓存
  - 业务线程发现缓存失效后，通过消息队列发送一条消息通知后台线程更新缓存（适合业务刚上线时缓存预热）

缓存雪崩-解决方案

- 缓存集群：可以做缓存的主从与缓存水平分片





缓存高可用

- 缓存是否高可用，需要根据实际的场景而定，并不是所有业务都要求缓存高可用，需要结合具体业务，具体情况进行方案设计，例如临界点是否对后端的数据库造成影响。
- 主要解决方案：
  - 分布式：实现数据的海量缓存
  - 复制：实现缓存数据节点的高可用

缓存热点

- 一些特别热点的数据，高并发访问同一份缓存数据，导致缓存服务器压力过大。
- 解决：
  - 复制多份缓存副本，把请求分散到多个缓存服务器上，减轻缓存热点导致的单台缓存服务器压力

第4章 数据层的软件架构技术

4.1 数据驱动的软件架构演化

4.2 数据读写与主从分离

4.3 数据分库分表

4.4 数据缓存

4.5 非关系型数据库

4.6 数据层架构案例

4.5
非关系型数据库

1、NoSQL概述

2、NoSQL在系统架构中的应用

3、常用的NoSQL数据库

4.5.1 NoSQL概述

- NoSQL的基本概念
- NoSQL数据库分类

NoSQL的基本概念

NoSQL是什么

- NoSQL概念在2009年被提了出来
  - NoSQL最常见的解释是“non-relational”，“Not Only SQL”
- NoSQL中使用最多的是Key-value存储型，其他的还包含文档型、列存储型、图型数据库、XML数据库等
- 在NoSQL概念提出之前，已有类似的数据库应用，但较少用于Web互联网应用，如cdb、qdbm、bdb等数据库

传统关系数据库的瓶颈

- 传统的关系数据库具有不错的性能，高稳定性，使用简单，功能强大，被业界广泛认可。甚至，其中的MySQL为互联网的发展作出了卓越的贡献。
- 关系型数据面临的问题：
  - 扩展困难：由于存在类似Join这样多表查询机制，使得数据库在扩展方面很艰难；
  - 读写慢：这种情况主要发生在数据量达到一定规模时由于关系型数据库的系统逻辑非常复杂，使得其非常容易发生死锁等的并发问题，所以导致其读写速度下滑严重；
  - 成本高：企业级数据库的License价格很惊人，并且随着系统的规模变大，成本将不断上升；
  - 有限的支撑容量：现有关系型解决方案还无法支撑大型互联网应用级的海量的数据存储，如Google等。

NoSQL诞生的原因

- 数据库访问的新需求
  - 低延迟的读写速度：应用快速地反应能极大地提升用户的满意度
  - 支撑海量的数据和流量：对于搜索这样大型应用而言，需要利用PB级别的数据和能应对百万级的流量；
  - 大规模集群的管理：系统管理员希望分布式应用能更简单的部署和管理；
  - 庞大运营成本的考量：IT管理者们希望在硬件成本、软件成本和人力成本能够有大幅度地降低；

NoSQL数据库共有原则

- 假设失效是必然发生的
  - NoSQL实现都建立在硬盘、机器和网络都会失效这些假设之上
  - 我们不能彻底阻止这些失效，因此需要让系统能够在即使非常极端的条件下也能应付这些失效
- 对数据进行分区
  - 最小化失效带来的影响，也将读写操作的负载分布到了不同的机器上
- 保存同一数据的多个副本
  - 大部分NoSQL实现都基于数据副本的热备份来保证连续的高可用性
  - 一些实现提供了API，可以控制副本的复制，也就是说，当存储一个对象时，可以在对象级指定希望保存的副本数
- 查询支持
  - 在这个方面，不同的实现有本质的区别。不同的实现的一个共性在于哈希表中的Key-value匹配

关系型数据库 VS NoSQL

关系型数据库	NoSQL
• 表都是存储一些格式化的数据结构	• 以键值对存储，它的结构不固定
• 每个元组字段的组成都一样	• 每一个元组可以有不一样的字段
• 即使不是每个元组都需要所有字段，但数据库会为每一个元组分配所有的字段	• 每个元组可以根据需要增加一些自己的键值对
• 这样的结构可以便于表与表之间进行连接等操作	• 不会局限于固定的结构，可以减少一些时间和空间的开销
• 分布式关系型数据库中强调的ACID	• 对于许多互联网应用来说，对于一致性要求可以降低，而可用性（Availability）的要求则更为明显，从而产生了弱一致性的理论BASE
• ACID的目的就是通过事务支持，保证数据的完整性和正确性	• BASE这个模型是反ACID模型

关系型数据库 VS NoSQL

- NoSQL数据库仅仅是关系数据库在某些方面（性能、扩展）的一个弥补
  - 单从功能上讲，NoSQL的几乎所有功能，在关系数据库上都能够满足
  - 一般会把NoSQL和关系数据库进行结合使用，各取所长，各得其所
- 在某些应用场合，例如一些配置的关系键值映射存储，用户名和密码的存储，Session会话存储等
  - 用NoSQL完全可以替代关系型数据库存储，不但具有更高的性能，而且开发也更便捷

NoSQL的优缺点

优点	缺点
• 简单的拓展：轻松添加新的节点来扩展集群	• 不提供对SQL的支持：如果不支持SQL这样的工业标准，将会对用户产生一定的学习和应用迁移成本，尽管现在有第三方工具或中间件缓解此弊端，但仍为非标准化支持
• 快速的读写：由于逻辑简单，且部分数据库可进行纯内存操作（如Redis），这使得其性能非常出色，单节点每秒可以处理超过10万次读写操作	• 支持的特性不够丰富：现有产品所提供的功能都比较有限，大多数NoSQL数据库都不支持事务，也不像MS SQL Server和Oracle那样能提供各种附加功能，比如BI和报表等
• 低廉的成本：大多数NoSQL的共有特点，就是开源，没有昂贵的License成本	• 现有产品在逐步成熟中：大多数产品都是开源产品的新贵，有待继续完善，包括与之相关的生态构建。

NoSQL的优缺点

- NoSQL数据库的出现，弥补了关系数据（比如MySQL）在某些方面的不足，能极大的节省开发成本和维护成本。
- MySQL和NoSQL都有各自的特点和使用的应用场景，两者的紧密结合将会丰富系统在数据架构层面上的可能性，为不同的工程问题提供不同的解决方案。
- 关系数据库关注在关系上，NoSQL关注在存储上。

NoSQL数据库分类

按数据模型分类-三大主流

- Column-oriented（列式）
  - 主要围绕着“列（Column）”，而非“行（Row）”进行数据存储
  - 属于同一列的数据会尽可能地存储在硬盘同一页（Page）中
  - 大多数列式数据库都支持Column Family这个特性
  - 每次查询都会处理很多数据，但涉及的列并不多
  - 特点：比较适合汇总（Aggregation）和数据仓库类应用
- Key-value
  - 类似常见的HashTable，一个key对应一个value，但是其能提供非常快的查询速度、大的数据存放量和高并发操作
  - 非常适合通过主键对数据进行查询和修改等操作，虽然不支持复杂的操作，但可通过上层的开发来弥补这个缺陷
- Document（文档）
  - 类似key-value，但提供嵌入式文档的存储方式
  - 经常查询的数据存储在同一个文档中，而不是存储在表中，如果一个应用程序需要存储不同的属性以及大量的数据，那么该类数据库将会是一个好选择

按数据模型分类-全分类

类型	部分代表	特点
列存储	Hbase Cassandra Hypertable	顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩对针对某一列或者某几列的查询有非常大的IO优势。
文档存储	MongoDB CouchDB	文档存储一般用类似json的格式存储，存储的内容是文档型的。这样也就有有机会对某些字段建立索引，实现关系数据库的某些功能。
key-value存储	Tokyo Cabinet / Tyrant Berkeley DB MemcacheDB Redis	可以通过key快速查询到其value。一般来说，存储不管value的格式，照单全收。（Redis包含了其他功能）
图存储	Neo4J FlockDB	图形关系的最佳存储。使用传统关系数据库来解决的话性能低下，而且设计使用不方便。
对象存储	db4o Versant	通过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	Berkeley DB XML BaseX	高效的存储XML数据，并支持XML的内部查询语法，比如XQuery,Xpath。

按CAP分类-CP

记在A4上

- 关注一致性和分区容忍性
- 这类数据库可支持在无法确保数据一致性的情况下拒绝提供服务，损失可用性，主要的CP类数据库有：
  - BigTable（Column-oriented）
  - Hypertable（Column-oriented）
  - HBase（Column-oriented）
  - MongoDB（Document）
  - Terrastore（Document）
  - Redis（Key-value）
  - Scalaris（Key-value）
  - MemcacheDB（Key-value）
  - Berkeley DB（Key-value）

按CAP分类-AP

记在A4上

- 关注可用性和分区容忍性
- 这类数据库主要以实现“最终一致性（Eventual Consistency）”来确保可用性和分区容忍性，主要的AP类数据库有：
  - Cassandra（Column-oriented）
  - CouchDB（ Document ）
  - SimpleDB（ Document ）
  - Riak（Document）
  - Dynamo（Key-value）
  - Voldemort（Key-value）
  - Tokyo Cabinet（Key-value）
  - KAI（Key-value）

4.5
非关系型数据库

1、NoSQL概述

2、NoSQL在系统架构中的应用

3、常用的NoSQL数据库

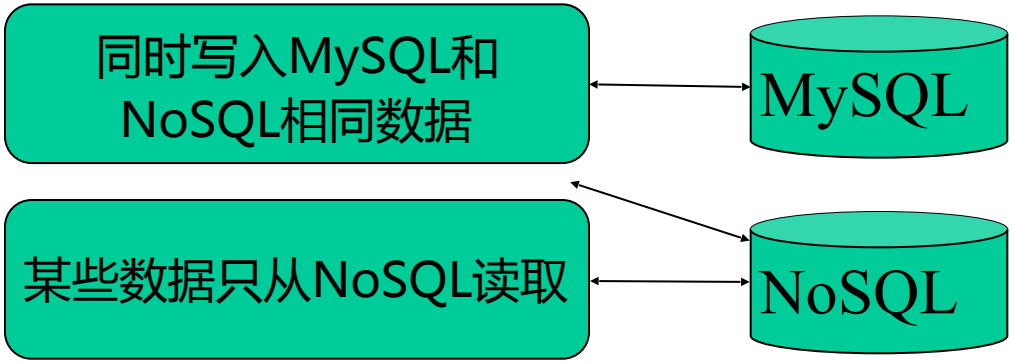
4.5.2 NoSQL在系统架构中的应用

- 以NoSQL为辅
- 以NoSQL为主
- 以NoSQL为缓存

以NoSQL为辅

NoSQL作为镜像

- 不改变原有的以MySQL作为存储的架构，使用NoSQL作为辅助镜像存储，用NoSQL的优势辅助提升性能。
- 在原有基于MySQL数据库的架构上增加了一层辅助的NoSQL存储
- 在写入MySQL数据库后，同时写入到NoSQL数据库，让MySQL和NoSQL拥有相同的镜像数据
- 在某些可以根据主键查询的地方，使用高效的NoSQL数据库查询



NoSQL作为镜像

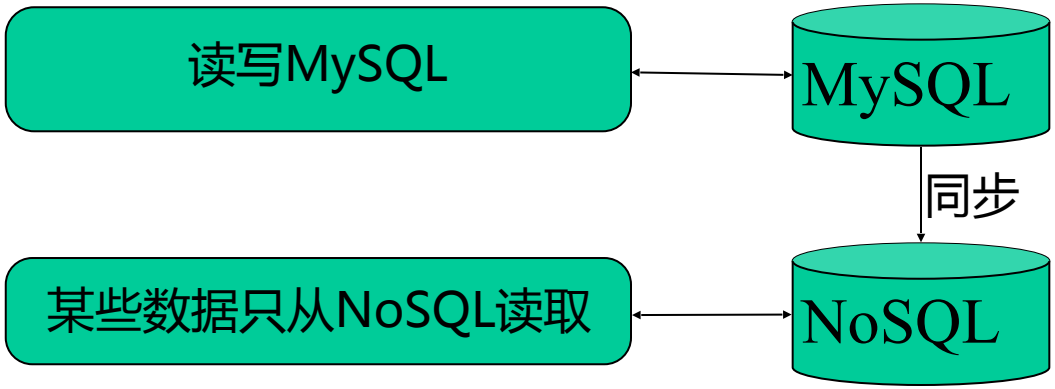
```
1       //data为我们要存储的数据对象
2       data.title="title";
3       data.name="name";
4       data.time="2009-12-01 10:10:01";
5       data.from="1";
6       id=DB.Insert(data);//写入MySQL数据库
7       NoSQL.Add(id,data);//以写入MySQL产生的自增id为主键写入NoSQL数据库
```

一致性要求

```
1   //写入数据的示例伪代码
2       //data为我们要存储的数据对象
3       bool status=false;
4       DB.startTransaction();//开始事务
5       id=DB.Insert(data);//写入MySQL数据库
6       if(id>0){
7           status=NoSQL.Add(id,data);//以写入MySQL产生的自增id为主键写入NoSQL数据库
8       }
9       if(id>0 && status==true){
10          DB.commit();//提交事务
11      }else{
12          DB.rollback();//不成功，进行回滚
13      }
```

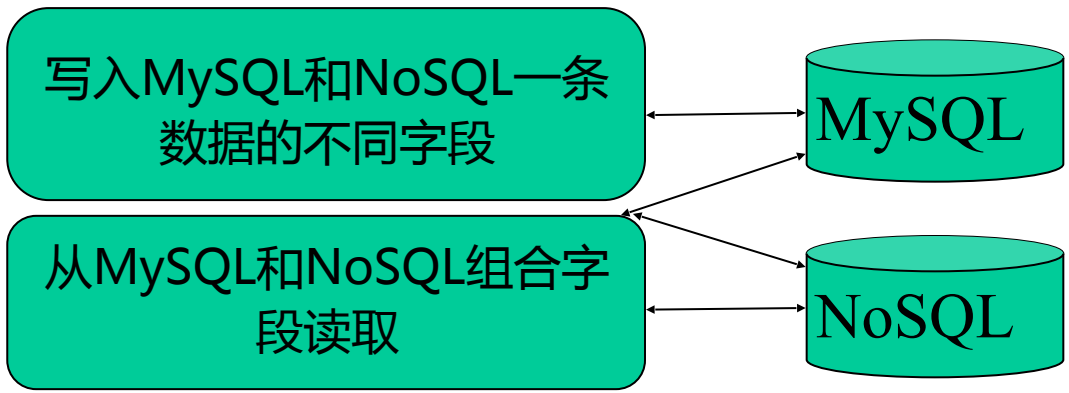
NoSQL作为镜像-同步模式

- 通过MySQL把数据同步到NoSQL中，是一种对写入透明，但是具有更高技术难度的一种模式
- 适用于现有的比较复杂的老系统，通过修改代码不易实现，可能引起新的问题，同时也适用于需要把数据同步到多种类型的存储中
- MySQL到NoSQL同步的实现可以使用MySQL UDF函数，MySQL binlog的解析来实现。可以利用现有的开源项目来实现，比如：MySQL memcached UDFs：从通过UDF操作Memcached协议



MySQL和NoSQL组合

- MySQL中只存储需要查询的小字段，NoSQL存储所有数据
- 把需要查询的字段，一般都是数字，时间等类型的小字段存储于MySQL中，根据查询建立相应的索引
- 其他不需要的字段，包括大文本字段都存储在NoSQL中
- 在查询的时候，先从MySQL中查询出数据的主键，然后从NoSQL中直接取出对应的数据即可



NoSQL架构应用 → 为辅

MySQL和NoSQL组合

```
    //写入数据的示例伪代码

    //data为我们要存储的数据对象
    data.title="title";
    data.name="name";
    data.time="2009-12-01 10:10:01";
    data.from="1";
    bool status=false;
    DB.startTransaction();//开始事务
    id=DB.Insert("INSERT INTO table (from) VALUES(data.from)");//写入MySQL数据库,只写from需要where查询的字段
    if(id>0){
        status=NoSQL.Add(id,data);//以写入MySQL产生的自增id为主键写入NoSQL数据库
    }
    if(id>0 && status==true){
        DB.commit();//提交事务
    }else{
        DB.rollback();//不成功，进行回滚
    }
```

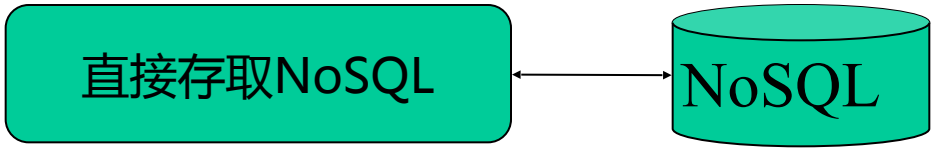
优点：

* 节省MySQL的IO开销。由于MySQL只存储需要查询的小字段，不再负责存储大文本字段，这样就可以节省MySQL存储的空间开销，从而节省MySQL的磁盘IO。
* 提高MySQL Query Cache缓存命中率。我们知道query cache缓存失效是表级的，在MySQL表一旦被更新就会失效，经过这种字段的分离，更新的字段如果不是存储在MySQL中，那么对query cache就没有任何影响。而NoSQL的Cache往往都是行级别的，只对更新的记录的缓存失效。
* 提升MySQL主从同步效率。由于MySQL存储空间的减小，同步的数据记录也减小了，而部分数据的更新落在NoSQL而不是MySQL，这样也减少了MySQL数据需要同步的次数。
* 提高MySQL数据备份和恢复的速度。由于MySQL数据库存储的数据的减小，很容易看到数据备份和恢复的速度也将极大的提高。 比以前更容易扩展。NoSQL天生容易扩展。经过这种优化，MySQL性能也得到提高。

以NoSQL为主

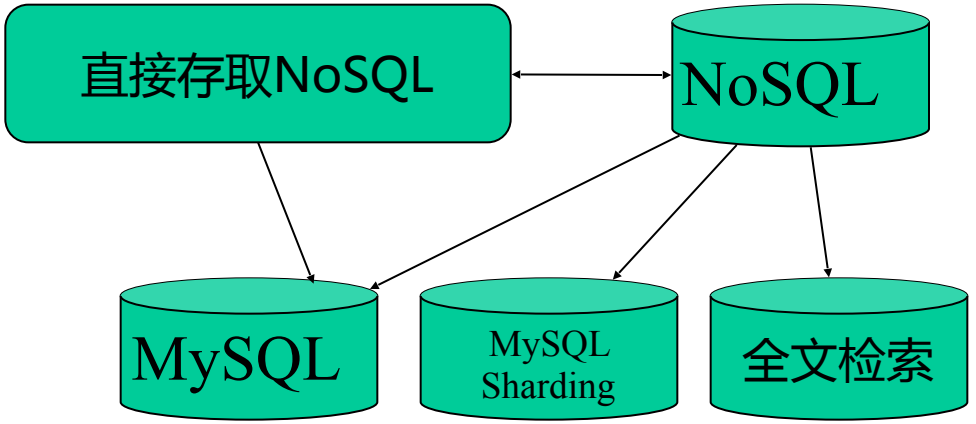
纯NoSQL架构

- 在一些数据结构、查询关系非常简单的系统中，可以只使用NoSQL即可以解决存储问题。
- 在一些数据库结构经常变化，数据结构不定的系统中，就非常适合使用NoSQL来存储。
  - 例如监控系统中的监控信息的存储，可能每种类型的监控信息都不太一样。
- 有些NoSQL数据库已经具备部分关系数据库的关系查询特性，他们的功能介于key-value和关系数据库之间，却具有key-value数据库的性能。基本满足绝大多数互联网应用的查询需求。



以NoSQL为数据源

- 数据直接写入NoSQL，再通过NoSQL同步协议复制到其他存储
- 根据应用的逻辑来决定去相应的存储获取数据
- 应用程序只负责把数据直接写入到NoSQL数据库，然后通过NoSQL的复制协议，把NoSQL的每次写入，更新，删除操作都复制到MySQL数据库中
- 同时，也可以通过复制协议把数据同步复制到全文检索实现强大的检索功能
- 这种架构需要考虑数据复制的延迟问题，与主从中的复制延迟问题一样。



以NoSQL为缓存

以NoSQL为缓存

- 由于NoSQL数据库天生具有高性能、易扩展的特点，所以常常结合关系数据库，存储一些高性能的、海量的数据。
- 从另外一个角度看，根据NoSQL的高性能特点，它同样适合用于缓存数据。
- 用NoSQL缓存数据可以分为内存模式和磁盘持久化模式。

内存模式

- Memcached提供了相当高的读写性能，在互联网发展过程中，一直是缓存服务器的首选。
- NoSQL数据库Redis又为我们提供了功能更加强大的内存存储功能。跟Memcached比，Redis的一个key可以存储多种数据结构Strings、Hashes、Lists、Sets、Sorted sets。
- Redis不但功能强大，而且它的性能完全超越了Memcached
  - 除了支持多种数据结构之外，Redis还提供了更加易于使用的api和操作性能，比如对缓存的list数据的修改。

磁盘持久化模式

- 虽然基于内存的缓存服务器具有高性能，低延迟的特点，但是内存成本高，内存数据易失却不容忽视
- 大部分互联网应用的特点都是数据访问有热点，也就是说，只有一部分数据是被频繁使用的。
- 其实NoSQL数据库内部也是通过内存缓存来提高性能的，通过一些比较好的算法
  - 把热点数据进行内存缓存
  - 非热点数据存储到磁盘
  - 以节省内存占用
- 使用NoSQL来做缓存，由于其不受内存大小的限制，可以把一些不常访问、不怎么更新的数据也缓存起来。

4.5
非关系型数据库

1、NoSQL概述

2、NoSQL在系统架构中的应用

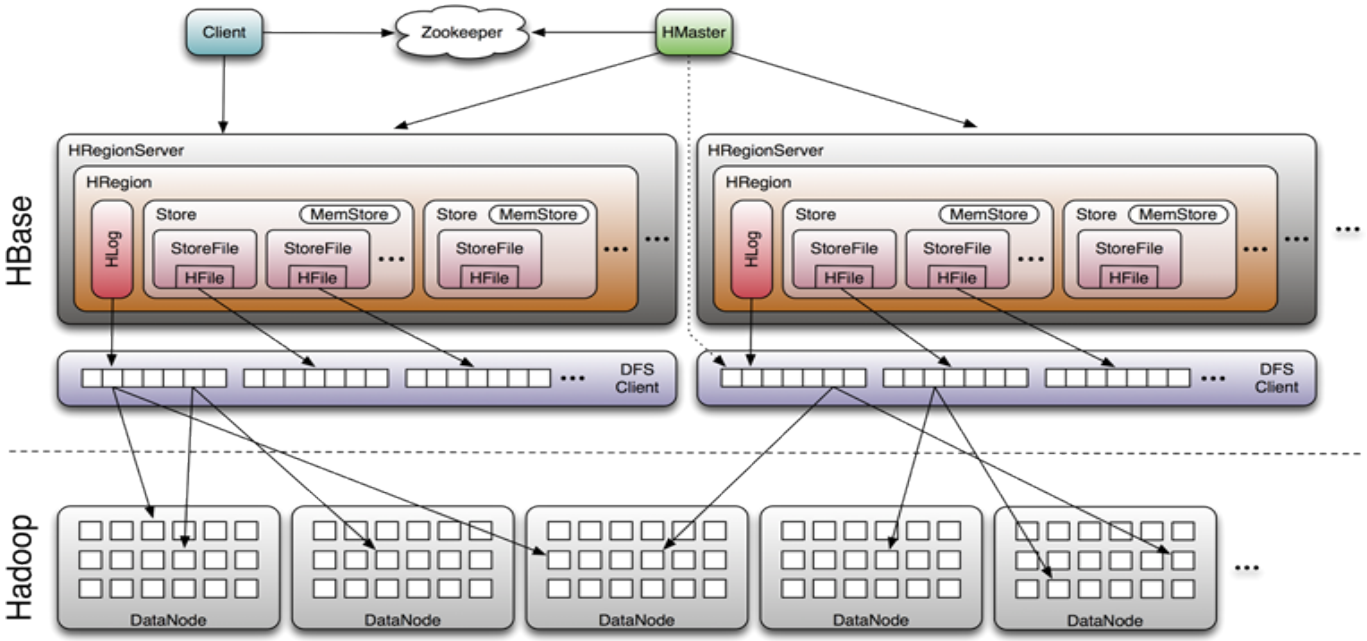
3、常用的NoSQL数据库

4.5.2 常用的NoSQL数据库

记下几种类型

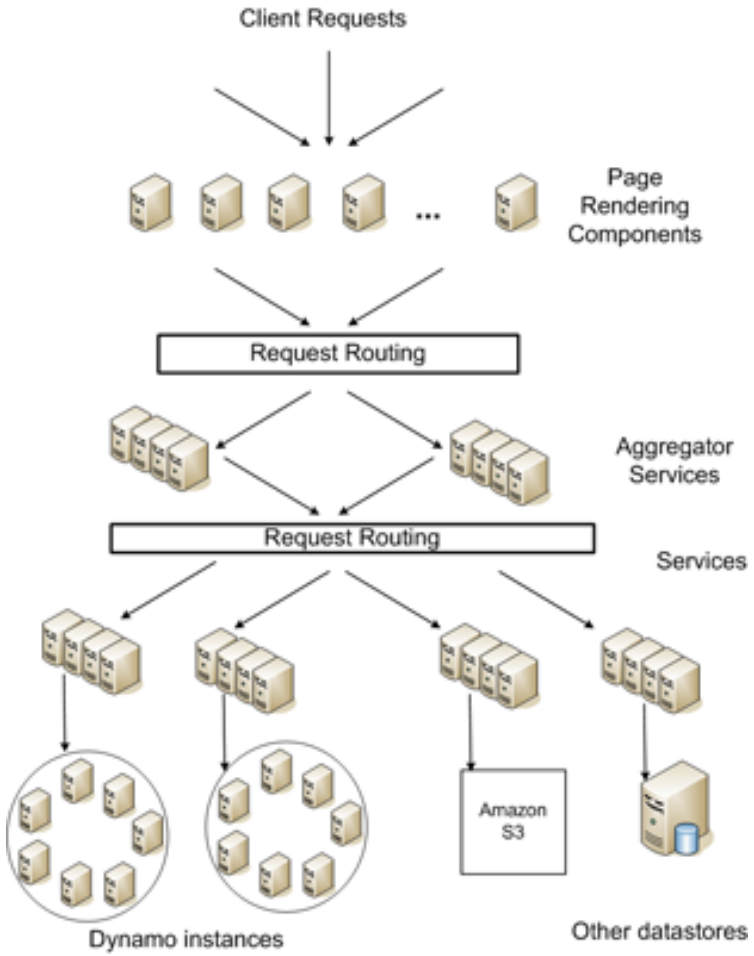
Hadoop： Hbase

- HBase: Handoop Databse，基于Key-Value和列族数据库的NoSQL
  - 利用Hadoop HDFS作为其文件存储系统：高可靠性的底层存储
  - 利用Hadoop MapReduce来处理HBase中的海量数据：高性能的计算能力
  - 利用Zookeeper作为协同服务：稳定服务和Failover机制。



Amazon：Dynamo

- Dynamo：具有高可用性和高扩展性的分布式数据存储系统。
  - 去中心化的分布式系统，由多个物理异构的机器组成；每台机器存放一部分数据；数据的备份同步完全由系统自己完成；
  - 每台机器可随意添加或去除；单台机器故障不会影响系统对外的可用性。
- 数据存储格式：键值（Key-Value）
- 数据分区：Consistent Hashing
- 数据复制：在Consistent Hashing Ring上面，用后续结点为前面的结点做备份。



Facebook：Cassandra

- Apache Cassandra：Facebook的开源分布式Key-Value存储系统
  - 由一堆数据库节点共同构成一个分布式网络服务；
  - 对Cassandra 的写操作，会被复制到其他节点上；
  - 对Cassandra的读操作，会被路由到某个节点上去读取；
  - 易于扩展。
- Cassandra以Amazon Dynamo为基础，结合了Google BigTable基于列族(Column Family)的数据模型。

Redis

- Redis(Remote Dictionary Server): 一种Key-Value存储系统。
- Redis在保持键值数据库简单快捷特点的同时，又吸收了关系数据库的优点，使它处于关系数据库和键值数据库之间。
  - Key-Value不仅能保存Strings类型的数据，还能保存Lists类型和Sets类型的数据。
- 将数据存储于内存中，或被配置为虚拟内存。
- 采用两种方式以实现数据持久化：
  - 使用截图方式，将内存中的数据不断写入磁盘；
  - 使用类似MySQL的日志方式，记录每次更新的日志。
  - 前者性能较高，但可能会引起一定程度的数据丢失；后者相反。
- 支持主从副本。

MongoDB

- MongoDB：分布式文档存储数据库
  - 介于关系数据库和非关系数据库之间，数据模型不局限于“关系”或“Key-Value”；
  - 面向集合存储，易存储对象类型的数据：数据被分组存储在数据集中，被称为一个集合(Collection)；每个集合在数据库中都有一个唯一标识名，并可包含无限数目的文档，类似于RDMBS的Table，但无需定义Schema；
  - 存储在集合中的文档，被存储为Key-Value的形式；键用于唯一标识一个文档(为字符串类型)，而值则可以是各种复杂的文件类型；
  - 高性能、易部署、易使用，存储数据非常方便。

作业4

- 针对KWIC问题，我们要提供一个全球10亿用户可用的Web应用，请给出该系统的分布式架构设计方案
  - 用户可以从多种终端访问应用，e.g. 手机端、PC端
  - 可以通过网络上传待处理文件，文件可能超过10000行，历史数据需要存储并提供检索功能
  - 设计方案应考虑数据层的典型架构技术，实现存储的高可用和高性能
  - 应能够与作业3的计算层面架构形成有机整体
  - 截止日期：+2周

第4章

数据层的软件架构技术

Thanks for listening

涂志莹、苏统华
哈尔滨工业大学计算机学院
企业与服务计算研究中心