# 分布式存储系统

* 分布式存储系统包括分布式文件系统、NoSQL和分布式数据库。

# 分布式文件系统

* 对一些图片、大文本的存储，使用数据库就不合适了。可以考虑的一个方案是NAS网络存储设备。它是一种专用数据存储服务器。它以数据为中心，将存储设备与服务器彻底分离，集中管理数据，从而释放带宽、提高性能、降低总拥有成本、保护投资。其成本远远低于使用服务器存储，而效率却远远高于后者。
* 不过NAS本身的IO吞吐性能及扩展性在大型网站中会表现出明显的不足，另外的一个方案是分布式文件系统。
* 比如GFS，HDFS，FastDFS。

# 块存储&文件存储&对象存储

# 块存储

* 块级是指以扇区为基础，一个或连续的扇区组成一个块，也叫物理块。它是在文件系统与块设备（例如：磁盘驱动器)之间。
* 典型设备：磁盘阵列，硬盘，虚拟硬盘
* 存储方案：
  + 1) DAS（Direct Attach STorage)：是直接连接于主机服务器的一种储存方式，每一台主机服务器有独立的储存设备，每台主机服务器的储存设备无法互通，需要跨主机存取资料时，必须经过相对复杂的设定，若主机服务器分属不同的操作系统，要存取彼此的资料，更是复杂，有些系统甚至不能存取。通常用在单一网络环境下且数据交换量不大，性能要求不高的环境下，可以说是一种应用较为早的技术实现。
  + 2)SAN（Storage Area Network)：是一种用高速（光纤)网络联接专业主机服务器的一种储存方式，此系统会位于主机群的后端，它使用高速I/O 联结方式， 如 SCSI, ESCON 及 Fibre- Channels。一般而言，SAN应用在对网络速度要求高、对数据的可靠性和安全性要求高、对数据共享的性能要求高的应用环境中，特点是代价高，性能好。例如电信、银行的大数据量关键应用。它采用SCSI 块I/O的命令集，通过在磁盘或FC（Fiber Channel)级的数据访问提供高性能的随机I/O和数据吞吐率，它具有高带宽、低延迟的优势，在高性能计算中占有一席之地，但是由于SAN系统的价格较高，且可扩展性较差，已不能满足成千上万个CPU规模的系统。

# 文件存储

* 文件级是指文件系统，单个文件可能由于一个或多个逻辑块组成，且逻辑块之间是不连续分布。逻辑块大于或等于物理块整数倍。扇区→物理块→逻辑块→文件系统
* 通常，NAS产品都是文件级存储。

# 对象存储

* 对象存储同兼具SAN高速直接访问磁盘特点及NAS的分布式共享特点。
* 核心是将数据通路（数据读或写)和控制通路（元数据)分离，并且基于对象存储设备（Object-based Storage Device，OSD)构建存储系统，每个对象存储设备具有一定的智能，能够自动管理其上的数据分布。
* 对象存储结构组成部分（对象、对象存储设备、元数据服务器、对象存储系统的客户端)：

## 对象

* 对象是系统中数据存储的基本单位，一个对象实际上就是文件的数据和一组属性信息（Meta Data)的组合，这些属性信息可以定义基于文件的RAID参数、数据分布和服务质量等，而传统的存储系统中用文件或块作为基本的存储单位，在块存储系统中还需要始终追踪系统中每个块的属性，对象通过与存储系统通信维护自己的属性。在存储设备中，所有对象都有一个对象标识，通过对象标识OSD命令访问该对象。通常有多种类型的对象，存储设备上的根对象标识存储设备和该设备的各种属性，组对象是存储设备上共享资源管理策略的对象集合等。

## 对象存储设备

* 对象存储设备具有一定的智能，它有自己的CPU、内存、网络和磁盘系统，OSD同块设备的不同不在于存储介质，而在于两者提供的访问接口。OSD的主要功能包括数据存储和安全访问。目前国际上通常采用刀片式结构实现对象存储设备。OSD提供三个主要功能：
  + （1) 数据存储。OSD管理对象数据，并将它们放置在标准的磁盘系统上，OSD不提供块接口访问方式，Client请求数据时用对象ID、偏移进行数据读写。
  + （2) 智能分布。OSD用其自身的CPU和内存优化数据分布，并支持数据的预取。由于OSD可以智能地支持对象的预取，从而可以优化磁盘的性能。
  + （3) 每个对象元数据的管理。OSD管理存储在其上对象的元数据，该元数据与传统的inode元数据相似，通常包括对象的数据块和对象的长度。而在传统的NAS系统中，这些元数据是由文件服务器维护的，对象存储架构将系统中主要的元数据管理工作由OSD来完成，降低了Client的开销。

## 元数据服务器（Metadata Server，MDS)

* MDS控制Client与OSD对象的交互，主要提供以下几个功能：
  + （1) 对象存储访问。
* MDS构造、管理描述每个文件分布的视图，允许Client直接访问对象。MDS为Client提供访问该文件所含对象的能力，OSD在接收到每个请求时将先验证该能力，然后才可以访问。
  + （2) 文件和目录访问管理。
* MDS在存储系统上构建一个文件结构，包括限额控制、目录和文件的创建和删除、访问控制等。
  + （3) Client Cache一致性。
* 为了提高Client性能，在对象存储系统设计时通常支持Client方的Cache。由于引入Client方的Cache，带来了Cache一致性问题，MDS支持基于Client的文件Cache，当Cache的文件发生改变时，将通知Client刷新Cache，从而防止Cache不一致引发的问题。

## 对象存储系统的客户端Client

* 为了有效支持Client支持访问OSD上的对象，需要在计算节点实现对象存储系统的Client，通常提供POSIX文件系统接口，允许应用程序像执行标准的文件系统操作一样。

# GFS

* HDFS就是基于Java的类GFS的实现。

# NoSQL

* NoSQL涵盖的范围很广，基本上处于分布式文件系统和关系型数据库之间的系统都被归为NoSQL的范畴。

# NoSQL数据模型

# Key-Value

* 键值对，没有办法进行高效的范围查询

# Ordered Key-Value

* Key是有序的，可以解决基于Key的范围查询的效率问题。不过在这个模型中，Value本身的内容和结构是由应用来负责解析和存储的，如果在多个应用中去使用的话，这种方式并不直观也不方便。

# BigTable

* BigTable对Value进行了Schema的支持，Value是由多个Column Family组成，Column Family内部是Column，Column Family不能动态扩展，而Column Family内部的Column是可以动态扩展的。

# Document

* Document数据库有两个非常大的进步，一个是可以在Value中自定义复杂的Schema，而不再仅仅是Map的嵌套；另一个是对索引的支持

# Graph

* 图数据库可以看作是从Ordered Key-Value数据库发展而来的一个分支。主要是支持图结构的数据模型。

# 分布式数据库

# 分布式数据库解决方案

* RDBMS -> NoSQL -> NewSQL
* 在RDBMS基础上有一些分布式数据库中间件，比如MyCat；另一种则是新型数据库，称为NewSQL，一般是兼容某一种RDBMS以保证用户使用无障碍（比如兼容MySQL)，比如TiDB。
* TiDB
* TiDB 是新一代开源分布式 NewSQL 数据库，模型受 Google Spanner / F1 论文的启发, 实现了自动的水平伸缩，强一致性的分布式事务，基于 Raft 算法的多副本复制等重要 NewSQL 特性。 TiDB 结合了 RDBMS 和 NoSQL 的优点，部署简单，在线弹性扩容和异步表结构变更不影响业务， 真正的异地多活及自动故障恢复保障数据安全，同时兼容 MySQL 协议，使迁移使用成本降到极低。

# 中间件的局限性

## 性能

* 基于 MySQL 的方案它的天花板在哪里，它的天花板特别明显。有一个思路是能不能通过 MySQL 的 server 把 InnoDB 变成一个分布式数据库，听起来这个方案很完美，但是很快就会遇到天花板。因为 MySQL 生成的执行计划是个单机的，它认为整个计划的 cost 也是单机的，我读取一行和读取下一行之间的开销是很小的，比如迭代 next row 可以立刻拿到下一行。实际上在一个分布式系统里面，这是不一定的。
* 另外，你把数据都拿回来计算这个太慢了，很多时候我们需要把我们的 expression 或者计算过程等等运算推下去，向上返回一个最终的计算结果，这个一定要用分布式的 plan，前面控制执行计划的节点，它必须要理解下面是分布式的东西，才能生成最好的 plan，这样才能实现最高的执行效率。
* 比如说你做一个 sum，你是一条条拿回来加，还是让一堆机器一起算，最后给我一个结果。 例如我有 100 亿条数据分布在 10 台机器上，并行在这 10台机器我可能只拿到 10 个结果，如果把所有的数据每一条都拿回来，这就太慢了，完全丧失了分布式的价值。聊到 MySQL 想实现分布式，另外一个实现分布式的方案就是 Proxy。但是 Proxy 本身的天花板在那里，就是它不支持分布式的 transaction，它不支持跨节点的 join，它无法理解复杂的 plan，一个复杂的 plan 打到 Proxy 上面，Proxy 就傻了，我到底应该往哪一个节点上转发呢，如果我涉及到 subquery sql 怎么办？所以这个天花板是瞬间会到，在传统模型下面的修改，很快会达不到我们的要求。

## 高可用（运维)

* 另外一个很重要的是，MySQL 支持的复制方式是半同步或者是异步，但是半同步可以降级成异步，也就是说任何时候数据出了问题你不敢切换，因为有可能是异步复制，有一部分数据还没有同步过来，这时候切换数据就不一致了。前一阵子出现过某公司突然不能支付了这种事件，今年有很多这种类似的 case，所以微博上大家都在说“说好的异地多活呢？”……
* 为什么传统的方案在这上面解决起来特别的困难，天花板马上到了，基本上不可能解决这个问题。另外是多数据中心的复制和数据中心的容灾，MySQL 在这上面是做不好的。

## SQL支持

* 基于中间件来进行分库， 确实对 SQL 有阉割的情况，并不是所有sql都能够支持。主要原因是数据被拆分了。而数据一旦被拆分到多个节点，则： 1.复杂的join查询2. 同时更新多个数据库节点的sql语句这两类SQL的支持难度，就比较高。这也是目前市面上所有中间件都无法满足的两点。复杂的join查询之所以难以支持，是因为要跨节点join；同时更新多个节点的sql难以支持，是因为很难解决多个节点的并发一致性问题。但是除了这两点之外，其他的sql类型，一款中间件是能够努力做到的。

# NoSQL的局限性

* 无法支持ACID，强一致性事务
* 不支持SQL，需要手工设计数据分布与查询

# 数据库从单机到分布式的挑战和应对

# 垂直拆分

* 将同一个库中的表分到不同的库中
* 垂直拆分的影响：
  + 1)单机ACID被打破，要么放弃原来的单机事务，修改实现，要么引入分布式事务
  + 2)一些JOIN操作会变得比较困难，需要应用或者其他方式来解决
  + 3)靠外键去进行约束的场景会受影响

# 水平拆分

* 水平拆分会带来如下影响：
  + 1)ACID被打破
  + 2)JOIN操作被影响
  + 3)靠外键去进行约束的场景会有影响
  + 4)依赖单库的自增序列生成唯一ID会受影响
  + 5)针对单个逻辑意义上的表的查询要跨库了

# 分布式事务

## 2PC（2-Phase-Commit)

### 介绍

* 是在分布式环境下保证事务原子性和一致性设计的算法，也被认为是一种一致性协议，用来保证分布式系统数据的一致性。目前，绝大多数数据库都是采用2PC协议来完成分布式事务处理的。
* 它分成两个阶段，先由一方进行提议(propose)并收集其他节点的反馈(vote)，再根据反馈决定提交(commit)或中止(abort)事务。我们将提议的节点称为协调者(协调者)，其他参与决议节点称为参与者(参与者s, 或cohorts)。
* 另一种情况：
* 回滚所有资源。

#### 阶段一：提交事务请求 投票阶段

* 1、事务询问
* 协调者向所有参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待各参与者的响应。
* 2、执行事务
* 各参与者节点执行事务操作，并将Undo和Redo信息记入事务日志中。
* 3、各参与者向协调者反馈事务询问的响应
* 如果参与者成功执行了事务操作，那么就反馈给协调者yes响应，表示事务可以执行；如果参与者没有成功执行事务，那么就反馈给协调者no响应，表示事务不可以执行。

#### 阶段二：执行事务提交 执行阶段

* 在阶段二中，协调者会根据各参与者的反馈情况来决定最终是否可以进行事务提交操作，正常情况下，包括以下两种可能：

##### 执行事务提交

* 假如协调者从所有的参与者获得反馈都是yes，那么就会执行事务提交。
* 1、发送提交请求
* 协调者向所有参与者节点发出commit请求
* 2、事务提交
* 参与者接收到commit请求后，会正式执行事务提交操作，并在完成提交之后释放在整个事务执行期间占用的事务资源。
* 3、反馈事务提交结果
* 参与者在完成事务提交之后，向协调者发送ACK消息。
* 4、完成事务
* 协调者接收到所有参与者反馈的ACK消息，完成事务

##### 中断事务

* 假如任何一个参与者向协调者反馈了no响应，或者在等待超时之后，协调者无法接收到所有参与者的反馈响应，那么就会中断事务。
* 1、发送回滚请求
* 协调者向所有参与者节点发出rollback请求
* 2、事务回滚
* 参与者接收到rollback请求后，会利用其在阶段一中记录的Undo信息来中事务回滚操作，并在完成回滚之后释放在整个事务执行期间占用的资源。
* 3、反馈事务回滚结果
* 参与者在完成事务回滚之后，向协调者发送ACK消息
* 4、中断事务
* 协调者接收到所有参与者反馈的ACK消息，完成事务中断。

### 优缺点

* 优点是原理简单，实现方便
* 缺点：同步阻塞、单点问题、数据不一致、太过保守

#### 同步阻塞

* 两阶段提交中的第二阶段, 协调者需要等待所有参与者发出yes请求, 或者一个参与者发出no请求后, 才能执行提交或者中断操作. 这会造成长时间同时锁住多个资源, 造成性能瓶颈, 如果参与者有一个耗时长的操作, 性能损耗会更明显.

#### 单点问题

* 协调者的角色在整个2PC中起到了非常重要的作用。一旦协调者出现问题，那么整个2PC流程将无法运转。如果协调者是在阶段二出现问题的话，那么其他参与者将会一直处于锁定事务资源的状态中，而无法继续完成事务操作。

#### 数据不一致/脑裂

* 在2PC的阶段二，即在执行事务提交的时候，当协调者向所有的参与者发送commit请求之后，发生了局部网络异常或者是协调者在尚未完全发送完commit请求之前自身发生了崩溃，导致最终只有部分参与者收到了commit请求。于是，这部分收到了commit请求的参与者就会进行事务的提交，而其他没有收到commit请求的参与者则无法进行事务提交，于是出现数据不一致的情况。

#### 太过保守

* 如果在协调者指示参与者进行事务提交询问的过程中，参与者出现故障而导致协调者始终无法获取到所有参与者的响应信息的话，这时协调者只能依靠其自身的超时机制来判断是否需要中断事务，这样的策略显得比较保守。2PC没有涉及较为完善的容错机制，任意一个节点的失败都会导致整个事务的失败。

### XA 基于2PC的分布式事务规范

* X/Open组织提出了一个分布式事务的规范——XA。了解XA先要了解X/Open组织定义的分布式事务处理模型——X/Open DTP（X/Open Distributed Transaction Processing Reference Model)模型。在该模型中定义了三个组件，即Application Progream,Resource Manager和Transaction Manager。
* AP：应用程序，可以理解为使用DTP模型的程序，它定义了事务边界，并定义了构成该事务的应用程序的特点操作
* RM：资源管理器，可以理解为一个DBMS系统。应用程序通过资源管理器对资源进行控制，资源必须实现XA定义的接口，资源管理器提供了存储共享资源的支持。
* TM：事务管理器，负责协调和管理事务，提供给AP应用程序编程接口并管理资源管理器。事务管理器向事务指定标识，监视它们的进程，并负责处理事务的完成和失败。事务分支标识（XID)由TM指定，以标识一个RM内的全局事务和指定分支。它是TM中日志与RM中日志之间的相关标记，2PC提交或回滚时需要XID，以便在系统启动时执行再同步操作，或在需要时允许管理员执行试探操作。
* 在这三个组件中，AP可以与TM、RM通信，TM与RM之间可以互相通信。DTP模型中定义了XA接口，TM和RM通过XA接口进行双向的通信。
* AP和RM是一定需要的，而事务管理器TM是我们额外引入的。之所以要引入TM，是因为在分布式系统中，两台机器理论上无法达到一致的状态，需要引入一个单点进行协调。RM控制着全局事务，管理事务的生命周期，并协调资源。
* DTP中还定义了几个概念：
  + 1)事务：一个事务是一个完整的工作单位，由多个独立的计算任务组成，这多个任务在逻辑上是原子的
  + 2)全局事务：一次性操作多个RM的事务就是全局事务
  + 3)分支事务：在全局事务中，每一个RM有自己独立的任务，这些任务的集合就是这个资源管理器的分支任务
  + 4)控制线程：用来表示一个工作线程，主要是关联AP、TM和RM三者的线程，也就是事务上下文环境。
  + 1)AP和RM之间，可以使用RM自身提供的native API进行交互，这种方式就是使用RM的传统方式，并且这个交互不在TM的管理范围内。另外，当AP和RM之间需要进行分布式事务时，AP需要得到对RM的连接（由TM管理)，然后使用XA的native API来进行交互。
  + 2)AP和TM之间，使用的是TX接口，它用于对事务进行控制，包括启动事务、提交事务和回滚事务。
  + 3)TM和RM之间是通过XA接口进行交互的。TM管理到了RM的连接，并实现了2PC

### 实现

* Atomikos
* 在JavaEE平台下，WebLogic、Webshare等主流商用的应用服务器提供了JTA的实现和支持。而在Tomcat下是没有实现的，这就需要借助第三方的框架Jotm、Automikos等来实现，两者均支持spring事务整合。

## TCC（Try-Confirm-Commit)

### 介绍

* TCC, 是基于补偿型事务的AP系统的一种实现, 具有最终一致性.  
  下面以客户购买商品时的付款操作为例进行讲解:
* Try:
* 完成所有的业务检查(一致性),预留必需业务资源(准隔离性);
* 体现在本例中, 就是确认客户账户余额足够支付(一致性), 锁住客户账户, 商户账户(准隔离性).
* Confirm:
* 使用Try阶段预留的业务资源执行业务(业务操作必须是幂等的), 如果执行出现异常, 要进行重试.
* 在这里就是执行客户账户扣款, 商户账户入账操作.
* Cancle:
* 释放Try阶段预留的业务资源, 在这里就是释放客户账户和商户账户的锁;
* 如果任一子业务在Confirm阶段有操作无法执行成功, 会造成对业务活动管理器的响应超时, 此时要对其他业务执行补偿性事务. 如果补偿操作执行也出现异常, 必须进行重试, 若实在无法执行成功, 则事务管理器必须能够感知到失败的操作, 进行log(用于事后人工进行补偿性事务操作或者交由中间件接管在之后进行补偿性事务操作).

### 优点

* 对比与前面提到的两阶段提交法, 有两大优势:
* TCC能够对分布式事务中的各个资源进行分别锁定, 分别提交与释放, 例如, 假设有AB两个操作, 假设A操作耗时短, 那么A就能较快的完成自身的try-confirm-cancel流程, 释放资源. 无需等待B操作. 如果事后出现问题, 追加执行补偿性事务即可.
* TCC是绑定在各个子业务上的(除了cancel中的全局回滚操作), 也就是各服务之间可以在一定程度上”异步并行”执行。

### 适用场景

* • 严格一致性
* • 执行时间短
* • 实时性要求高
* 举例: 红包, 收付款业务.

### 实现

* https://github.com/liuyangming/ByteTCC

## 异步确保型/可靠消息最终一致（基于消息中间件,要求MQ支持事务消息->目前阿里闭源版MQ支持)

### 介绍

* 通过将一系列同步的事务操作变为基于消息执行的异步操作, 避免了分布式事务中的同步阻塞操作的影响.
* 这个方案真正实现了两个服务的解耦, 解耦的关键就是异步消息和补偿性事务。
* 执行步骤如下:
* MQ发送方发送远程事务消息到MQ Server;
* MQ Server给予响应, 表明事务消息已成功到达MQ Server.
* MQ发送方Commit本地事务.
* 若本地事务Commit成功, 则通知MQ Server允许对应事务消息被消费; 若本地事务失败, 则通知MQ Server对应事务消息应被丢弃.
* 若MQ发送方超时未对MQ Server作出本地事务执行状态的反馈, 那么需要MQ Server向MQ发送方主动回查事务状态, 以决定事务消息是否能被消费.
* 当得知本地事务执行成功时, MQ Server允许MQ订阅方消费本条事务消息.
* 需要额外说明的一点, 就是事务消息投递到MQ订阅方后, 并不一定能够成功执行. 需要MQ订阅方主动给予消费反馈(ack)
* 如果MQ订阅方执行远程事务成功, 则给予消费成功的ack, 那么MQ Server可以安全将事务消息移除;
* 如果执行失败, MQ Server需要对消息重新投递, 直至消费成功.

### 适用场景

* • 执行周期较长
* • 实时性要求不高

### 实现

* RocketMQ

## 最大努力通知型（基于消息中间件，定期校对)

### 介绍

* 这是分布式事务中要求最低的一种, 也可以通过消息中间件实现, 与前面异步确保型操作不同的一点是, 在消息由MQ Server投递到消费者之后, 允许在达到最大重试次数之后正常结束事务.
* 1.业务活动的主动方，在完成业务处理之后，向业务活动的被动方发送消息，允许消息丢失。
* 2.主动方可以设置时间阶梯型通知规则，在通知失败后按规则重复通知，直到通知N次后不再通知。
* 3.主动方提供校对查询接口给被动方按需校对查询，用于恢复丢失的业务消息。
* 4.业务活动的被动方如果正常接收了数据，就正常返回响应，并结束事务。
* 5.如果被动方没有正常接收，根据定时策略，向业务活动主动方查询，恢复丢失的业务消息。

### 适用场景

* 交易结果消息的通知等.

### 实现

* 相比于可靠消息最终一致方案，最大努力通知方案设计上比较简单，主要是由两部分构成。
* 1.实时消息服务（MQ)：接收主动方发送的MQ消息。
* 2.通知服务子系统：监听MQ消息，当收到消息后，向被动方发送通知（一般是URL方式)，同时生成通知记录。如果没有接收到被动方的返回消息，就根据通知记录进行重复通知。
* 最大努力通知方案实现方式比较简单，本质上就是通过定期校对，适用于数据一致性时间要求不太高的场合，其实不把它看作是分布式事务方案，只认为是一种跨平台的数据处理方案也是可以的。

# 多机Sequence/分布式全局唯一序列号

- 1)唯一性  
- 2)连续性

* 如果仅考虑ID的唯一性，那么可以使用UUID，但是连续性不好
* 如果要追求连续性，可以考虑：把所有ID集中放在一个地方进行管理，对每个ID序列独立管理，每台机器上使用ID时就从这个ID生成器中取。这里有如下几个问题需要解决：
  + 1)性能问题：每次都远程取ID会有资源损耗。一种改进方案是一次取一段ID，然后缓存在本地，这样就不需要每次都去远程的生成器上取ID了。但是也会带来问题，如果应用取了一段ID，正在用时完全宕机了，那么一些ID号就浪费不可用了。
  + 2)生成器的稳定问题：ID生成器作为一个无状态的集群存在，其可用性要靠整个集群来保证。
  + 3)存储的问题。底层存储的选择空间较大，需要根据不同类型进行对应的容灾方案。下面介绍两种方式：
* 独立ID生成器方案：在底层使用一个独立的存储来记录每个ID序列当前的最大值，并控制并发更新。
* 生成器嵌入到应用：在每个应用上完成生成器要做的工作，即读取可用的ID或ID段，然后给应用的请求使用。
* 因为这种方式没有中心的控制节点，并且不希望生成器之间还有通信，因此数据的ID并不是严格按照进入数据库的顺序而增大的。

## Snowflake

### 介绍

- 41-bit 的时间可以表示（1L<<41)/(1000L\*3600\*24\*365)=69 年的时间，

* 10-bit 机器可以分别表示1024 台机器。如果我们对IDC 划分有需求，还可以将
* 10-bit 分5-bit 给IDC，分5-bit 给工作机器。这样就可以表示32 个IDC，每个
* IDC 下可以有32 台机器，可以根据自身需求定义。12 个自增序列号可以表示2^12
* 个ID，理论上snowflake 方案的QPS 约为409.6w/s，这种分配方式可以保证在任何一个IDC 的任何一台机器在任意毫秒内生成的ID 都是不同的。

### 源码

/\*\*  
 \* SnowFlake的结构如下(每部分用-分开):<br>  
 \* 0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 - 000000000000 <br>  
 \* 1位标识，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负数是1，所以id一般是正数，最高位是0<br>  
 \* <p>  
 \* 41位时间截(毫秒级)，注意，41位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截 - 开始时间截)  
 \* 得到的值)，这里的的开始时间截，一般是我们的id生成器开始使用的时间，由我们程序来指定的（如下下面程序IdWorker类的startTime属性)。  
 \* 41位的时间截，可以使用69年，年T = (1L << 41) / (1000L \* 60 \* 60 \* 24 \* 365) = 69<br>  
 \* <p>  
 \* 10位的数据机器位，可以部署在1024个节点，包括5位datacenterId和5位workerId<br>  
 \* <p>  
 \* 12位序列，毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生4096个ID序号<br>  
 \* 加起来刚好64位，为一个Long型。<br>  
 \* SnowFlake的优点是，整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞(由数据中心ID和机器ID作区分)，并且效率较高，经测试，SnowFlake每秒能够产生26万ID左右。  
 \*/  
public class SnowflakeIdWorker {  
 /\*\*  
 \* 开始时间截 (2015-01-01)  
 \*/  
 private final long twepoch = 1420041600000L;  
  
 /\*\*  
 \* 机器id所占的位数  
 \*/  
 private final long workerIdBits = 5L;  
  
 /\*\*  
 \* 数据标识id所占的位数  
 \*/  
 private final long dataCenterIdBits = 5L;  
  
 /\*\*  
 \* 支持的最大机器id，结果是31 (这个移位算法可以很快的计算出几位二进制数所能表示的最大十进制数)  
 \*/  
 private final long maxWorkerId = -1L ^ (-1L << workerIdBits);  
  
 /\*\*  
 \* 支持的最大数据标识id，结果是31  
 \*/  
 private final long maxDataCenterId = -1L ^ (-1L << dataCenterIdBits);  
  
 /\*\*  
 \* 序列在id中占的位数  
 \*/  
 private final long sequenceBits = 12L;  
  
 /\*\*  
 \* 机器ID向左移12位  
 \*/  
 private final long workerIdShift = sequenceBits;  
  
 /\*\*  
 \* 数据标识id向左移17位(12+5)  
 \*/  
 private final long dataCenterIdShift = sequenceBits + workerIdBits;  
  
 /\*\*  
 \* 时间截向左移22位(5+5+12)  
 \*/  
 private final long timestampLeftShift = sequenceBits + workerIdBits + dataCenterIdBits;  
  
 /\*\*  
 \* 生成序列的掩码，这里为4095 (0b111111111111=0xfff=4095)  
 \*/  
 private final long sequenceMask = -1L ^ (-1L << sequenceBits);  
  
 /\*\*  
 \* 工作机器ID(0~31)  
 \*/  
 private long workerId;  
  
 /\*\*  
 \* 数据中心ID(0~31)  
 \*/  
 private long dataCenterId;  
  
 /\*\*  
 \* 毫秒内序列(0~4095)  
 \*/  
 private long sequence = 0L;  
  
 /\*\*  
 \* 上次生成ID的时间截  
 \*/  
 private long lastTimestamp = -1L;  
  
 //==============================Constructors=====================================  
  
 /\*\*  
 \* 构造函数  
 \*  
 \* @param workerId 工作ID (0~31)  
 \* @param dataCenterId 数据中心ID (0~31)  
 \*/  
 public SnowflakeIdWorker(long workerId, long dataCenterId) {  
 if (workerId > maxWorkerId || workerId < 0) {  
 throw new IllegalArgumentException(String.format("worker Id can't be greater than %d or less than 0", maxWorkerId));  
 }  
 if (dataCenterId > maxDataCenterId || dataCenterId < 0) {  
 throw new IllegalArgumentException(String.format("datacenter Id can't be greater than %d or less than 0", maxDataCenterId));  
 }  
 this.workerId = workerId;  
 this.dataCenterId = dataCenterId;  
 }  
  
 // ==============================Methods==========================================  
  
 /\*\*  
 \* 获得下一个ID (该方法是线程安全的)  
 \*  
 \* @return SnowflakeId  
 \*/  
 public synchronized long nextId() {  
 long timestamp = timeGen();  
  
 //如果当前时间小于上一次ID生成的时间戳，说明系统时钟回退过这个时候应当抛出异常  
 if (timestamp < lastTimestamp) {  
 throw new RuntimeException(  
 String.format("Clock moved backwards. Refusing to generate id for %d milliseconds", lastTimestamp - timestamp));  
 }  
  
 //如果是同一时间生成的，则进行毫秒内序列  
 if (lastTimestamp == timestamp) {  
 sequence = (sequence + 1) & sequenceMask;  
 //毫秒内序列溢出  
 if (sequence == 0) {  
 //阻塞到下一个毫秒,获得新的时间戳  
 timestamp = tilNextMillis(lastTimestamp);  
 }  
 }  
 //时间戳改变，毫秒内序列重置  
 else {  
 sequence = 0L;  
 }  
  
 //上次生成ID的时间截  
 lastTimestamp = timestamp;  
  
 //移位并通过或运算拼到一起组成64位的ID  
 return ((timestamp - twepoch) << timestampLeftShift) //  
 | (dataCenterId << dataCenterIdShift) //  
 | (workerId << workerIdShift) //  
 | sequence;  
 }  
  
 /\*\*  
 \* 阻塞到下一个毫秒，直到获得新的时间戳  
 \*  
 \* @param lastTimestamp 上次生成ID的时间截  
 \* @return 当前时间戳  
 \*/  
 protected long tilNextMillis(long lastTimestamp) {  
 long timestamp = timeGen();  
 while (timestamp <= lastTimestamp) {  
 timestamp = timeGen();  
 }  
 return timestamp;  
 }  
  
 /\*\*  
 \* 返回以毫秒为单位的当前时间  
 \*  
 \* @return 当前时间(毫秒)  
 \*/  
 protected long timeGen() {  
 return System.currentTimeMillis();  
 }  
  
  
 /\*\*  
 \* 测试  
 \*/  
 public static void main(String[] args) {  
 SnowflakeIdWorker idWorker = new SnowflakeIdWorker(0, 0);  
 for (int i = 0; i < 1000; i++) {  
 long id = idWorker.nextId();  
 System.out.println(Long.toBinaryString(id));  
 System.out.println(id);  
 }  
 }  
}

### 优点

* ● 毫秒数在高位，自增序列在低位，整个ID 都是趋势递增的。
* ● 不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成 ID 的性能也是非常高的。
* ● 可以根据自身业务特性分配 bit位，非常灵活。

### 缺点

* ● 强依赖机器时钟，如果机器上时钟回拨，会导致发号重复或者服务会处于不可用状态。
* 为什会时间回拨？
* 第一：人为操作，在真实环境一般不会有那个傻逼干这种事情，所以基本可以排除。
* 第二： 由于有些业务等需要，机器需要同步时间服务器（在这个过程中可能会存在时间回拨，查了下我们服务器一般在10ms以内（2小时同步一次))。

## Flicker

* 基于MySQL自增ID的机制。
* 创建一张表
* create table tickets (
* id bigint primary key auto\_increment,
  + stub char(1) not null default ‘’ unique
* );
* 在一个事务中
* replace into tickets(stub) values(‘a’); 不存在（主键索引或唯一索引)则插入
* select last*insert*id();
* 这样我们就能拿到不断增长且不重复的id了。如果要解决单点问题，可以启动多台数据库生成id，通过区分auto\_increment的起始值和步长来生成奇偶数的ID。

# 应对多机的数据查询

## 跨库JOIN

* 解决方案：
  + 1)在应用层把原来数据库的JOIN操作分成多次的数据库操作
  + 2)数据冗余
  + 3)借助外部系统（例如搜索引擎)来解决一些跨库的问题

## 外键约束

* 如果要对分库后的单库做外键约束，就要求分库后每个单库的数据是内聚的，否则就只能靠应用层的判断、容错等方式了。

## 跨库查询

* 分库分表后要对查询结果在应用上合并：
  + 1)排序，即多个来源的数据查询出来后，在应用层进行排序的工作。如果从数据库中查询出的数据是已经排好序的，那么在应用层要进行的就是多路归并排序；如果查询的数据未排序，就要进行一个全排序。
  + 2)函数处理，即使用max、min、sum、count等函数对多个数据来源的值进行相应的函数处理
  + 3)求平均值，从多个数据来源进行查询时，需要把SQL改为查询sum和count，然后对多个数据来源的sum求和、count求和后，计算平均值
  + 4)非排序分页：是同等步长地在多个数据源上分页处理，还是同等比例地分页处理。同等步长是指分页的每页中，来自不同数据源的记录数是一样的；同等比例的意思是，分页的每页中，来自不同数据源的数据占这个数据源符合条件的数据总数的比例是一样的。
  + 5)排序后分页：取第一页时，应该考虑的最极端情况是最终合并后的结果可能都来自一个数据源，所以我们需要从每个数据源取足一页的数据。对于第二页，需要把是每个数据源的前两页都取回来进行归并排序。越往后，承受的负担越重。

# 分布式数据库中间件的设计与实现

# 对外提供数据访问层功能的方式

* 第一种是为用户提供专有API，它的通用性很差。
* 第二种是通用的方式，数据层自身可以作为一个JDBC的实现，暴露出JDBC的接口给应用，此时应用的成本就很低了，和使用远程数据库的JDBC驱动的方式是一样的，迁移成本也很低。
* 第三种是基于ORM或类ORM接口的方式，比如MyBatis。

# 按照数据层流程的顺序看数据层设计

## SQL解析

* 有两个问题，一个是对SQL支持的程度，是否需要支持所有的SQL；
* 二是支持多少SQL的方言。
* 具体解析可以用antle、javacc。
* 通过SQL解析或者提示方式得到了相关信息后，下一步就是进行规则处理，从而确定要执行这个SQL的目标库。

## 规则处理

* 如何将数据分散到不同的数据库和表中?

### 固定哈希算法

* 固定哈希的方式为，根据某个字段比如id取模，然后将数据分散到不同的数据库和表中。除了根据id取模，还经常会根据时间维度来存储数据，这一般用于数据产生后相关日期不进行修改的情况。根据时间取模多用在日志类或者其他与时间维度密切相关的场景。通常将周期性的数据放在一起，这样进行数据备份、迁移或者现有数据的清空都会很方便。

### 一致性哈希

* 见《面试---7. 分布式理论---问题》

## SQL改写

* 对于应用给数据层执行的SQL，除了根据规则确定数据源外，我们可能还需要修改SQL。
* 我们的数据表从原来的单库单表变为了多库多表，这些分布在不同数据库中的表的结构一样，但是表名未必一样。如果把原来的表分布在多库并且每个库只有一个表的话，那么这些表是可以同名的，但是如果单库中不止一个表，那就不能用同样的名字，一般是在逻辑表名后加后缀。
* 在命名表时有一个需要作出选择之处，就是不同库中的表名是否要一样？如果每个表的名字都是唯一的，看起来似乎不太优雅，但是可以避免很多误操作。另外表名唯一在进行路由和数据迁移时比较便利。
* 除了修改表名，SQL中一些提示中用到的索引名，在分库分表时也需要进行相应的修改，需要从逻辑上的名字变为对应数据库中物理的名字。
* 另外，还有一个需要修改SQL的地方，就是在进行跨库计算平均值的时候，需要分别在各个库求sum和count，然后合并求平均。

## 选择数据源

* 规则部分可以确定一组数据源，而在这里需要确定是具体的某个数据源。一张表经历了分库分表后，我们会给分库后的库都提供备库。分库是把数据分到了不同的数据分组中。我们决定了数据分组后，还需要决定访问分组中的哪个库。这些库一般是一写多读的，根据当前要执行的SQL特点（读，写)、是否在事务中以及各个库的权重规则，计算得到这次SQL请求要访问的数据库。

### 完整数据源

* 单个DataSource下面对应了多个数据库，以及一组规则。

### 分组数据源

* groupDataSource，分组的DataSource，用于管理整个业务数据库集群中的一组数据库。groupDataSource相对于完整的DataSource来说，可以不管理具体的规则，也可以不进行SQL的解析。它是作为一个相对基础的数据源提供给业务的，那么groupDataSource重点解决的问题是，要在访问这个分组中的数据库时，解决访问具体数据库的选择问题，具体ide选择策略是groupDataSource要完成的重点工作，包括根据事务、读/写等特性选择主备、以及根据权重的不同在库间进行选择。

### 对比

* 如果采用完整的DataSource，对于应用来说只会看到一个DataSource，可以少关心很多事情，不过可能会收到DataSource本身的限制；如果采用groupDataSource会有更大的自主权。
* 如果采用完整的DataSource，对于后端业务的数据库集群的管理会更方便，例如我们可以进行一些扩容、缩容的工作而不需要应用太多的感知；而使用groupDataSource则意味着绑定了分组数量，这样要进行扩容、缩容时是需要应用进行较多配合的。虽然使用groupDataSource不能进行整体的扩容、缩容，但是可以进行组内的扩容、缩容、主备切换等工作，这也是groupDataSource最大的价值。在一些活动或者可预期的访问高峰前，可以给每个分组挂载上备库，通过配置管理中心的配置更新下线数据库，以及进行主备库的切换。

### 数据源封装

* 对于某个具体数据库，可以使用第三方数据源的组件配置。这种方式的最大缺点是不够动态，并且对于进行SQL执行的降级隔离等业务稳定性方面没有很多支持。如果我们通过AtomDataSource把单个数据库的数据源的配置集中存储，那么在定期更换密码、进行机房迁移等需要更改IP地址或改变端口时就会非常方便。另外，通过AtomDataSource也可以帮助我们完成在单表上的SQL连接隔离，以及禁止某些SQL的执行等和稳定性相关的工作。
* 把整体DataSource分层后为应用提供的三层数据源实现：

## 执行SQL和结果处理

* 在SQL执行的部分，比较重要的是对异常的处理和哦按段，需要能够从异常中明确判断出数据库不可用的情况，而关于执行结果的处理，在之前一些特殊情况中都已经提及，这里不再重复了。

# 独立部署的数据访问层实现方式

* 首先从数据层的物理部署来说可以分为jar包方式和Proxy的方式。
* 如果采用Proxy方式的话，客户端与Proxy之间的协议有两种选择：数据库协议和私有协议。
  + 1)采用数据库协议时，应用就会把Proxy看做一个数据库，然后使用数据库本身提供的JDBC的实现就可以连接Proxy。因为应用到Proxy、Proxy到DB采用的都是数据库协议，所以，如果使用的是同样的协议，例如都是MySQL协议，那么在一些场景下就可以减少一次MySQL协议到对象再到MySQL协议的转换。不过采用这种方式时Proxy要完全实现一套相关数据库的协议，这个成本是比较高的，此外，应用到Proxy之间也没有办法做到连接复用。
  + 2)采用私有协议时，Proxy对外的通信协议是我们自己设计的，并且需要一个独立的数据层客户端，这个协议的好处是，Proxy的实现会相对简单一些，并且应用到Proxy之前的连接是可以复用的。

# 读写分离的挑战和应对

* 读写分离的场景下，需要解决把主库的数据复制到备库中去。
* 如果是多从库对应一主库的情况下，并且数据结构相同，可以使用MySQL的Replication来解决这个问题。
* 假如主从库数据结构不同，那么需要将主库上的写操作都存放至数据同步服务器，然后由数据同步服务器将数据同步到从库。
* Extractor负责把数据源变更的信息加入到数据分发平台，Applier的作用是把这些变更应用到相应的目标上。

# 平滑迁移

* 迁移过程中不停机，保证系统持续服务。
  + 1)双写
  + 2)全量+增量：先复制全量数据，再复制增量数据
* 在开始进行数据迁移时，记录增量的日志，在迁移结束后，再对增量的变化进行处理。在最后，可以把要被迁移的数据的写暂停，保证增量日志都处理完毕后，再切换规则，放开所有的写，完成迁移工作。
  + 1)开始迁移，开始记录数据库的数据变更的增量日志
  + 2)数据开始复制到新库，并且也有更新进来
  + 3)当全量迁移结束后，把增量日志中的数据也进行迁移
  + 4)进行数据比对，记录源库和新库的数据不同
  + 5)停止对源库中的写操作，进行增量日志的处理
  + 6)更新路由规则，所有新数据的读写就到了新库

# 分布式缓存

* 缓存穿透
* 分布式文件系统 分布式存储系统（GFS、HDFS、fastDFS)、存储模型（skipList、LSM等)
* 自己实现一个缓存 LinkedHashMap？按访问顺序排序

# 缓存分类

* 1、按宿主层次划分
  + 1)本地缓存/进程间缓存：可以分为堆内缓存，堆外缓存。堆内缓存对GC影响较大，堆外缓存又会增加额外的序列化和反序列化开销
  + 2)进程间缓存：可以在本机单独启动一个进程专门放缓存，通过Domain Socket通信
  + 3)远程缓存：跨服务器访问的缓存，如Redis
  + 4)二级缓存：本地缓存与远程缓存的结合，对于不易改变但访问量巨大的数据，可以放到本地缓存中
* 2、按存储介质划分
  + 1)内存缓存
  + 2)持久化缓存
* 3、按架构层次划分
* 页面缓存、浏览器缓存、Web服务器缓存、反向代理缓存、应用级缓存

# 缓存使用场景

# 用缓存来管理存储

* 这种方式中，应用是不直接操作存储的，存储由缓存来控制。对于应用的逻辑来说这很简单，但是对于缓存来说，因为需要保证数据写入缓存后能够存入存储中，所以缓存本身的逻辑会复杂写，需要有很多操作日志及故障恢复。

# 应用直接管理缓存和存储

* 在这种方式中，应用直接与缓存和存储进行交互，一般的做法是应用在写数据时更新存储，然后失效缓存数据，而在读数据时首先读缓存，如果缓存中没有数据，那么再去读存储，并且把数据写入缓存。

# 合理使用缓存

# 频繁修改的数据

* 如果缓存中保存的是频繁修改的数据，就会出现数据写入缓存后，应用还来不及读取缓存，数据就已经失效的情形，徒增系统负担。一般来说，数据的读写比在2:1以上，即写入一次缓存，在数据更新前至少读取两次，缓存才有意义。

# 没有热点的访问

* 不可能将所有数据都缓存起来，只能将最新访问的数据缓存起来，而将历史数据清理出缓存。如果应用系统访问数据没有热点，不遵循二八定律，那么缓存没有意义。

# 数据不一致与脏读

* 一般会对缓存设置失效时间，一旦超过失效时间，就要从数据库中重新加载，因此应用要容忍一定时间的数据不一致。如卖家已经编辑了商品属性，但是需要过一段时间才能被买家看到。在互联网应用中，这种延迟通常是可以接受的。还有一种策略是数据更新时立即更新缓存，不过这也会带来更多系统开销和事务一致性的问题。

# 缓存可用性/缓存雪崩

* 当缓存服务崩溃时，数据库会因为完全不能承受如此大的压力而宕机，进而导致整个网站不可用。这种情况被称为缓存雪崩。
* 实践中，有的网站通过缓存热备等手段提高缓存可用性。
* 由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。
* 解决：将缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

# 缓存预热

* 缓存中存放过的是热点数据，热点数据又是缓存系统利用LRU对不断访问的数据筛选淘汰出来的。这个过程需要花费较长的时间。新启动的缓存系统如果没有任何数据，在重建缓存数据的过程中，系统的性能和数据库负载都不太好，那么最好在缓存系统启动时就把热点数据加载好，这个缓存预加载手段叫做缓存预热（warm up)。

# 缓存穿透

* 如果因为不恰当的业务，或者恶意攻击持久高并发地请求某个不存在的数据（或者缓存失效)，由于缓存没有保存该数据，所有的请求都会落到数据库上，会对数据库造成很大压力，甚至崩溃。一个简单的对策是将不存在的数据也缓存起来（value置为null)。
* 最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

# 分布式消息队列

* 作用：
  + 1)解耦：A系统完成后，需要通知B系统，此时可以用RPC的方式来解决。
* AService#doSomething(){
* BService.doSomething();
* }
* 但是如果修改需求，还要通知C、D系统，怎么实现？
* 还是用RPC的话，修改AService#doSomething()
* AService#doSomething() {
* BService.doSomething();
* CService.doSomething();
* DService.doSomething();
* }
* 如果使用消息队列，那么可以这样做：
* AService#doSomething() {
* mq.send(topic: “doSomething” ,data: data);
* }
* B、C、D系统均订阅该topic
* mq.subscibe(topic:”doSomething”,callback);
* 需要增加其他系统时，只需要在该系统中增加订阅该topic的操作即可，不需要修改既往代码。
  + 2)异步：削峰
* 短时间大量的任务可以先放入队列中慢慢处理，避免引起阻塞。

# 消息发送一致性/事务消息

* 消息发送一致性是指产生消息的业务动作与消息发送的一致。如果业务操作成功了，那么由这个操作产生的消息一定要发送出去，否则就丢失消息了。如果这个行为没有发生或者失败，那么就不应该把消息发送出去。
* 一种实现是JMS中的XA支持，引入了分布式事务。
* 事务消息也可以解决这个问题。
* 一个分布式事务被拆为一个本地事务和一个消息发送。
* 而消息发送的前提是本地事务执行成功，本地事务提交后，消息才会发送出去；否则会取消该消息的发送。
* 流程如下：
  1. Producer向Broker发送Prepared消息，可能会发送失败
  2. 执行本地事务
  3. 如果本地事务执行成功，则发送Confirm消息；如果失败，那么回滚本地事务，取消发送Confirm消息，Broker会删除Prepared消息。
  4. Producer发送Confirm消息时可能会发送失败，此时消息的状态仍为Prepared。
  5. Broker接收到Confirm消息时，会将该消息推送给Consumer。
  6. 设置Scheduler去向Producer轮询Prepared消息的当前状态，称为消息回查。消息回查主要目的是检测Confirm消息发送失败的情况。
  7. Consumer接收到消息，执行本地事务。
  8. 本地事务执行成功时，会返回给Broker一个ACK，执行失败时，Broker会定期重新发送给Consumer该消息，超过重试次数时可以选择不再重试。
* RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改消息的状态。
* 为解决确认消息发送失败的问题（消息回查)，RocketMQ会定期扫描消息集群中的事务消息，如果发现了Prepared消息，它会向消息发送端(生产者)确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

# 解决消息中间件与使用者的强依赖问题

* 解决消息发送一致性时，会造成消息中间件变成了业务应用的必要依赖。如果消息中间件系统出现问题，就会导致业务操作无法继续进行，即便当时业务应用和业务操作的资源都是可用的。
* 解决这个问题有三种思路：
  + 1)提供消息中间件系统的可靠性，但无法实现完全可靠
  + 2)对于消息中间件系统中影响业务操作进行的部分，使其可靠性与应用自身的可靠性相同
  + 3)可以提供弱依赖的支持，能够较好地保证一致性
* 第二种方案就是要保证业务能操作成功，就需要消息能够入库成功。
* 该方案有几种设计思路：
  + 1)应用和消息中间件一起操作消息表结构
  + 2)消息中间件不直接操作消息表结构
  + 3)应用本地记录消息结构

# 应用和消息中间件一起操作消息表结构

* 我们把消息中间件所需要的消息表与业务数据表放到同一个业务数据库中，这样，业务应用就可以把业务操作和写入消息作为一个本地事务来完成，然后再通知消息中间件有消息可以发送，这样就解决了一致性的问题。消息中间件会定时去轮询业务数据库，找到需要放松的消息，取出内容后进行发送。这个方案对业务系统有如下三个影响：
  + 1)需要用业务自己的数据库承载消息数据
  + 2)需要让消息中间件去访问业务数据库
  + 3)需要业务操作的对象是一个数据库

# 消息中间件不直接操作消息表结构

* 消息中间件不再直接与业务数据库打交道。消息表还是放在业务数据库中，完全由业务数据库来控制消息的生成、获取、发送及重试的策略。比较多的逻辑从消息中间件的服务端移动到消息中间件的客户端，并且在业务应用上执行。消息中间件更多的是管理接收消息的应用，并且当有消息从业务应用发过来后就只管理投递，把原来的调度、重头、投递等逻辑分到了客户端和服务端两边。
* 这两种方式已经解决了大部分的问题，但是它们都要求业务操作是支持事务的数据库操作。

# 应用本地记录消息结构

* 可以考虑把本地磁盘作为一个消息存储，如果消息中间件不可用，又不愿或不能侵入业务自己的数据库时，可以把本地磁盘作为存储消息的地方，等待消息中间件回复后，再把消息送到消息中间件中。所有的投递、重试等管理，仍然是在消息中间件中进行，而本地磁盘的定位只是对业务应用上发送消息一定成功的一个保证。
* 这种方式的风险是，如果消息中间件不可用，而且写入本地磁盘的数据也坏了的话，那么消息就丢失了。这确实是个问题，所以，从业务数据上进行消息补发才是最彻底的容灾的手段，因为只有这样才能保证只要业务数据在，就一定可以有办法恢复消息。
* 将本地磁盘作为消息存储的格式有两种用法，一是作为一致性发送消息的解决方案的容灾手段，该方式平时不工作，出现问题时才切换到该方式上；二是直接使用该方式来工作，这样可以控制业务操作本身调用发送消息的接口的处理时间（异步发送？)，此外也有机会在业务应用和消息中间件间做一些批处理的工作。

# 消息模型

# JMS Queue模型 / P2P

* 应用1和2发送消息到JMS服务器，这些消息根据到达的顺序形成一个队列，应用3和4进行消息的消费。注意，应用3和4收到的消息是不同的，在JMS Queue方式下，如果Queue里面的消息被一个应用消费了，那么连接到JMS Queue上的另一个应用是收不到这个消息的，也就是说所有连接到这个JMS Queue上的应用共同消费了所有的消息。消息从发送端发送出来时不能确定最终会被哪个应用消费，但是可以明确的是只有一个应用回去消费这条消息。也被称为Peer to Peer模型。

# JMS Topic模型

* 从发送消息的部分和JMS Topic内部的逻辑来看，JMS Topic和JMS Queue是一样的，二者最大的差别在于消息接收的部分，在Topic模型中，接收消息的应用3和4是可以独立收到所有到达Topic的消息的。也被称为Pub/Sub模型。

# 我们需要什么样的消息模型

- 1)消息发送方和接收方都是集群  
- 2)同一个消息的接收方可能有多个集群进行消息的处理  
- 3)不同集群对于同一条消息的处理不能相互干扰

* Queue和Topic都无法满足我们的需求，前者不支持独立消费，后者同一个集群内部各个机器会收到重复的消息。
* 我们可以把集群和集群之间对消息的消费当做Topic模型来处理，而集群内部的各个具体应用处理对消息的消费当做Queue模型来处理。可以引入ClusterID，用它来标识不同的集群，集群内部的各个应用实例的连接使用同样的ClusterID。
* 如果一定要使用JMS的话，有一个变通的做法。
* 不过这种级联方式相对比较繁重，是多个独立的JMS服务器之间的连接，这比在消息中间件服务器内部进行处理要复杂的多。好处是基本可以直接使用JMS的实现，这里需要注意的是从Topic中发消息分派到不同的Queue中时，需要由独立的中转的消息订阅者来完成，并且对同一个Queue的中转只能由一个连接完成。

# 消息订阅

# 非持久订阅

* 非持久订阅是消息接收者和消息中间件之间的消息订阅的关系的存续，与消息接收者自身是否处于运行状态有直接关系。也就是说，当消息接收者应用启动时，就建立了订阅关系，这时可以收到消息，而如果消息接收者应用结束了，那么消息订阅关系也就不存在了，这时的消息是不会为消息接收者保留的。

# 持久订阅

* 持久订阅的含义是，消息订阅关系一旦建立，除非应用显式地取消订阅关系，否则这个订阅关系将一直存在；而订阅关系建立后，消息接收者会接收到所有消息，如果消息接收者应用停止，那么这个消息也会保留，等待下次应用启动后再投递给消息接收者。

# 消息可靠性

* 在持久订阅前提下，整个消息系统是如何保证消息可靠的呢？
* 消息从发送端应用到接收端应用，中间有三个阶段需要保证可靠：
  + 1)消息发送者将消息发送到消息中间件
  + 2)消息中间件将消息存入消息存储
  + 3)消息中间件把消息投递给消息接收者

# 消息发送端的可靠性保证

* 消息从发送者发送到消息中间件，只有当消息中间件及时、明确地返回成功，才能确定消息可靠到达消息中间件了；返回错误、出现异常、超时等情况，都表示消息发送到消息中间件这个动作失败。这里需要注意的是对异常的处理，可能出现的问题是在不注意的情况下吃掉了异常，从而导致错误的判断结果。

# 消息存储的可靠性保证

* 要么是完全自主实现持久存储部分的代码，要么利用现有的存储系统实现。
* 现有的存储系统有比较多的选择，有关系型数据库、分布式文件系统和NoSQL。

## 基于文件的消息存储

## 基于数据库的消息存储

* 就消息中间件而言，我们希望尽量避免获取数据时的表关联查询，所以希望一个消息只用一个单行的数据来解决。
* 如果单机出现硬件故障，需要考虑数据的容灾方案。
  + 1)单机Raid
  + 2)多机的数据同步，要求不能有延迟
  + 3)应用双写

## 基于双机内存的消息存储

* 使用文件系统或数据库来进行消息存储时，因为磁盘IO的原因，系统性能都会受到限制。一个改进方案是用混合方式进行存储的管理。可以采用的一个方法是用双机的内存来保证数据的可靠性。正常情况下，消息持久存储是不工作的，而基于内存来存储消息则能够提供很高的吞吐量。一旦一个机器出现故障，则停止另一台机器的数据写操作，并把当前数据落盘。
* 只要不遇到两台基于内存的消息中间件机器同时出故障的情况，并且当一台出问题时，另一台将当前内存的消息写入持久存储的过程中不出问题的话，消息是很安全的。这种方式适用于消息到了消息中间件后大部分消息能够及时被消费掉的情况，它可以很好地提高性能。

# 消息投递的可靠性保证

* 消息中间件需要显式地接收到接收者确认消息处理完毕的信号（ACK)才能删除消息。
* 投递处理的第一个可优化之处是，在进行投递时一定要采用多线程的方式处理。
* 一种方式是每个线程处理一条消息，并且等待处理结束后再进行下一条消息的处理。这种方式在正常情况下没有问题，而遇到异常情况时，例如订阅者集群有一个很慢的订阅者，复杂投递的所有线程会慢慢地被堵死，因此都需要等待这个慢的订阅者返回。
* 另一种方式是把处理消息结果返回的处理工作放到另外的线程池中完成，也就是投递线程完成消息到网络的投递后就可以接着处理下一个消息，保证投递的环节不会被堵死。而等待返回结果的消息可以先放到内存中，不占用线程资源，等有了最后的结果时，再放入另外的线程池中处理。这种方式把占用线程池的等待方式变为了靠网络收到消息处理结果后的主动响应方式。
* 更新数据库也可以采用batch的方式。
* 第二个可优化之处是一个应用上有多个订阅者订阅同样的信息，如果不加以优化，我们会向这个机器发送多次同样的信息。
  + 1)单机多订阅者共享连接
  + 2)消息只发送一次，然后传到单机的多订阅者生成多个实例处理。

# 消息系统扩容

# 消息中间件扩容

* 消息中间件本身没有持久状态，扩容比较容易。主要是让消息的发送者和订阅者能够感知到有新的消息中间件机器加入到了集群，这是通过注册查找中心实现的。
* 在同一个存储中如何区分存储的消息是来自于哪个消息中间件应用的？可以给每条消息加一个server字段，当有新加入的消息中间件时，会使用新的server值。
* 如果有消息中间件应用长期不可用的话，我们就需要加入一个和它具有相同server标识的机器来代替它，或者通过这个消息中间件进入到消息系统中但还没有完成投递的消息分给其他机器处理，也就是让另一台机器承担剩余消息的投递工作。

# 消息存储扩容

* 假设：不用保证消息顺序；不提供Pull方式获取消息
* 为了在分库分表情况下主动根据某些条件进行数据查询，就必须确切知道数据存储在哪个数据库的哪张表，对这种情况的扩容就比较复杂。但在假设前提下，我们不需要支持外部主动根据条件查询消息的，因为：
  + 1)消息发送到消息中间件时，消息中间件将消息入库，这时消息中间件是明确知道消息存储在哪里的，并会进行消息的投递调度，所以一定能找到消息。
  + 2)由于在内存中进行调度的消息数量有限，因此我们会调度存储在数据库中的消息。而在调度时，我们更关心的是那些符合发送条件的消息，所以这个调度必然是需要跨库跨表的。在这个过程中，需要投递的消息会把相关索引信息加载到内存，在这个过程后，内存中的调度信息就自然有了存储节点信息。

# 消息重复

* 消息重复的产生原因：
  + 1)消息发送端应用的消息重复发送：
* 消息发送端发送消息给消息中间件，消息中间件收到消息并成功存储，而此时消息中间件出了问题，导致应用没有收到消息发送成功的返回，因而进行重试
* 消息中间件因为负载高，响应变慢，成功把消息存储到消息存储中后，返回“成功”这个结果时超时
* 消息中间件将消息成功写入消息存储，在返回结果时网络出现问题，导致应用发送端重试，而重试时网络恢复，由此导致重复
* 一个解决办法是，重试发送消息时使用同样的消息id，而不要在消息中间件产生消息id。
  + 2)消息到了消息存储，由消息中间件向外投递时产生重复：
* 消息被投递到消息接收者进行处理，处理完毕后应用出问题了，消息中间件不知道消息处理结果，会进行重试
* 消息被投递到消息接收者进行处理，处理完毕后网络出问题了，消息中间件没有收到消息处理结果，会进行重试
* 消息被投递到消息接收者进行处理，处理时间比较长，消息中间件因为消息超时会重试
* 消息被投递到消息接收者进行处理，处理完毕后消息中间件收到结果，但是遇到消息存储故障，没能更新投递状态，会进行重试
* 一种解决办法是引入分布式事务，还有一种是保证消息接收者的消息处理是幂等的。
* 在JMS中，消息接收者对收到的消息进行确认，有以下几种选择：
  + 1)AUTO\_ACKNOWLEDGE：自动确认，当JMS的消息接收者收到消息后，JMS的客户端会自动进行确认。但是消息确认时可能消息还没来得及处理或者尚未处理完成，所以这种确认方式对于消息投递处理来说是不可靠的。
  + 2)CLIENT\_ACKONWLEDGE：客户端确认，客户端如果要确认消息处理成功，告诉服务器确认消息时，需要主动调用Message#acknowledge方法以确认。
  + 3)DUPS*OK*ACKNOWLEDGE：在消息接收方的消息处理函数执行结束后进行确认，一方面保证了消息处理结束后才进行确认，另一方面也不需要客户端主动调用acknowledge方法了。

# 消息投递的其他属性支持

# 消息优先级

* 一般情况下消息是先到先投递，消息优先级的属性可以支持根据优先级来确定投递顺序。

# 订阅者消息处理顺序和分级订阅

* 对于同样的消息，可能会希望有些订阅者处理结束后再让其他订阅者进行处理，一种方案是可以设定优先处理的订阅集群，也就是我们这里的订阅者消息处理顺序的属性，可以在这个字段上设置有些处理的集群ID，另一种方案是分级订阅。
* 我们把优先接收者和一般接收者的接收分开，优先接收者处理成功后主动把消息投递到另外的消息中间件，然后一般接收者接收新产生的消息。

# 自定义属性

* 消息自身的创建时间、类型、投递次数等熟悉属于消息的基础属性，在消息体外，支持自动以的属性会很方便，比如消息过滤，以及接收端对消息的处理。

# 局部顺序

* 局部顺序是指在众多的消息中，和某件事情相关的多条消息之间有顺序，而多件事情之间的消息则没有顺序。
* 在消息中间件内部，有非常多的逻辑上独立的队列。支持局部有序需要消息上有一个属性，即区分某个消息应该与哪些消息一起排队的属性字段。

# 顺序消息

* 在顺序消息的场景下，对于消息接收端的设计从原来的Push变为了Pull，这是为了让消息接收者更好地控制消息的接收和处理，而消息中间件自身的逻辑也进行了简化。
* 在消息中间件内部，有多个物理上的队列，进入到每个队列的消息则是严格按照顺序被接收和消费的，而消息中间件单机内部的队列之间是互不影响的。
* 具体实现中，消息的存储就写到本地文件中，采用的是顺序写入的方式，其基本思路和前面基于文件的存储比较类似。二者的差别是，这个场景不存在文件的空洞，因为消息必须按照顺序去消息，所以，一个消息接收者在每一个它所接收的消息队列上有一个当前消费消息的位置，对于这个接收者来说，这个位置之前的消息就已经完成消费了。在同一个队列中，不同的消费者分别维护自己的指针，并且通过指针的回溯，可以把消息的消费恢复到之前的某个位置继续处理。
* 如果需要对消息进行补发，那么移动接收端的消费消息的指针就可以完成了。

# 单机多队列的问题与优化

* 如果单机的队列数量特别多，性能就会明显地下降，原因是队列数量过多时，消息接入就接近于随机写了。一个改进措施是把发送到这台机器的消息数据进行顺序写入，然后再根据队列做一个索引，每个队列的索引是独立的，其中保存的只是相对于存储数据的物理队列的索引位置。
* 这样改进的好处是：
  + 1)队列轻量化，单个队列数据量非常小
  + 2)对磁盘的访问串行化，避免磁盘竞争，不会因为队列增加导致IOWAIT增高
* 缺点：
  + 1)写虽然是顺序写，但读却变成了随机读
  + 2)读一条消息时，会先读逻辑队列，再读物理队列，增加了开销
  + 3)需要保证物理队列和逻辑队列完全一致性，增加了编程复杂度。

# 解决本地消息存储的可靠性

* 复制
  + 1)把单个的消息中间件机器变为主备两个节点，slave节点订阅master节点上的所有消息。这是一个异步的操作，存在着丢失消息的可能，比较类似于MySQL的Replication。（异步复制？)
  + 2)同步复制而非订阅，master收到消息后会主动写往slave，并且收到slave的响应后才向消息发送者返回“成功”。（同步双写？)
* 第二种方式更加安全和保险。

# 队列扩容

* 基本的策略是让向一个队列写入数据的消息发送者能够知道应该把消息写入迁移到新的队列中，并且也需要让消息订阅者知道，当前的队列消费完数据后需要迁移到新队列去消息消息。
* 关键点：
  + 1)原队列在开始扩容后需要有一个标志，即便有新消息过来，也不再接收
  + 2)通知消息发送端新的队列的位置
  + 3)对于消息接收端，对原来队列的定位会收到新旧两个位置，当旧队列的数据接收完毕后，则会只关心新队列的位置，完成切换。

# Pull/Pull

# 高并发下的幂等策略分析

# 为什么需要幂等

* 业务开发中，经常会遇到重复提交的情况，无论是由于网络问题无法收到请求结果而重新发起请求，或是前端的操作抖动而造成重复提交情况。
* 在交易系统，支付系统这种重复提交造成的问题有尤其明显，比如：
* 用户在APP上连续点击了多次提交订单，后台应该只产生一个订单；
* 向支付宝发起支付请求，由于网络问题或系统BUG重发，支付宝应该只扣一次钱。
* 很显然，幂等接口认为，外部调用者会存在多次调用的场景，为了防止重试对数据状态的改变，需要将接口的设计为幂等的。

# 什么情况下需要保证幂等性

* 以SQL为例，有下面三种场景，只有第三种场景需要开发人员使用其他策略保证幂等性：
* SELECT col1 FROM tab1 WHER col2=2，无论执行多少次都不会改变状态，是天然的幂等。
* UPDATE tab1 SET col1=1 WHERE col2=2，无论执行成功多少次状态都是一致的，因此也是幂等操作。
* UPDATE tab1 SET col1=col1+1 WHERE col2=2，每次执行的结果都会发生变化，这种不是幂等的。

# 保证幂等策略

* 幂等需要通过唯一的业务单号来保证。也就是说相同的业务单号，认为是同一笔业务。使用这个唯一的业务单号来确保，后面多次的相同的业务单号的处理逻辑和执行效果是一致的。
* 下面以支付为例，在不考虑并发的情况下，实现幂等很简单：①先查询一下订单是否已经支付过，②如果已经支付过，则返回支付成功；如果没有支付，进行支付流程，修改订单状态为‘已支付’。

# 防重复提交策略

* 上述的保证幂等方案是分成两步的，第②步依赖第①步的查询结果，无法保证原子性的。在高并发下就会出现下面的情况：第二次请求在第一次请求第②步订单状态还没有修改为‘已支付状态’的情况下到来。既然得出了这个结论，余下的问题也就变得简单：把查询和变更状态操作加锁，将并行操作改为串行操作。

## 乐观锁

* 如果只是更新已有的数据，没有必要对业务进行加锁，设计表结构时使用乐观锁，一般通过version来做乐观锁，这样既能保证执行效率，又能保证幂等。例如：
* UPDATE tab1 SET col1=1,version=version+1 WHERE version=#version#
* 不过，乐观锁存在失效的情况，就是常说的ABA问题，不过如果version版本一直是自增的就不会出现ABA的情况。（从网上找了一张图片很能说明乐观锁，引用过来，出自Mybatis对乐观锁的支持)

## 防重表

* 使用订单号orderNo做为去重表的唯一索引，每次请求都根据订单号向去重表中插入一条数据。第一次请求查询订单支付状态，当然订单没有支付，进行支付操作，无论成功与否，执行完后更新订单状态为成功或失败，删除去重表中的数据。后续的订单因为表中唯一索引而插入失败，则返回操作失败，直到第一次的请求完成（成功或失败)。可以看出防重表作用是加锁的功能。

## 分布式锁

* 这里使用的防重表可以使用分布式锁代替，比如Redis。订单发起支付请求，支付系统会去Redis缓存中查询是否存在该订单号的Key，如果不存在，则向Redis增加Key为订单号。查询订单支付已经支付，如果没有则进行支付，支付完成后删除该订单号的Key。通过Redis做到了分布式锁，只有这次订单订单支付请求完成，下次请求才能进来。相比去重表，将放并发做到了缓存中，较为高效。思路相同，同一时间只能完成一次支付请求。

## token令牌

* 这种方式分成两个阶段：申请token阶段和支付阶段。
* 第一阶段，在进入到提交订单页面之前，需要订单系统根据用户信息向支付系统发起一次申请token的请求，支付系统将token保存到Redis缓存中，为第二阶段支付使用。
* 第二阶段，订单系统拿着申请到的token发起支付请求，支付系统会检查Redis中是否存在该token，如果存在，表示第一次发起支付请求，删除缓存中token后开始支付逻辑处理；如果缓存中不存在，表示非法请求。

# 分布式RPC/服务框架

* RPC框架（包括整体的一些框架理论，通信的netty，序列化协议thrift，protobuff等)
* 服务化的好处：
  + 1)模块化拆分，独立维护，减少重复代码，提高代码质量
  + 2)核心相对稳定，修改和发布次数减少
  + 3)底层资源由服务层管理，结构更加清晰，利于提高效率

# 服务调用端的设计与实现

# 服务框架的使用方式

* 使用Spring进行引入。
* 因为Java有动态代理的支持，所以在完成远程调用的时候，使用一个通用的对象就可以解决问题了，而不需要像很多语言一样需要通过类似IDL（Interface Description Language，接口描述语言)的方式定义，然后生成代理存根代码，再分别与调用端和被调用端一起编译。
* 需要配置三个基础的属性
* interfaceName
* version
* group

## 配置

### interfaceName

* 接口名称。在进行远程通信时CustomerBean必须知道被调用的接口是哪一个，才能生成对这个接口的代理，以供本地调用，所以这是一个必备的属性。

### version

* 版本号。在实际的场景中，接口是存在变化的可能性的，有的是因为实现代码本身重构的原因，也有的是因为业务的发展变化需要修改接口中已有方法的参数或者返回值，以满足新的需求。如果直接这样变化，那就要求所有使用的地方一起修改，一起升级，这在一个大型的分布式系统中代价是非常高的。解决这个问题的方法有两种，一是如果需要修改方法的参数或返回值，就新增一个方法，始终保持已有方法不变，不过这样对导致在过渡期间代码相对臃肿；另一种方案就是通过版本号进行区分隔离。

### group

* 分组。分组的好处 是如果对同一个接口的远程服务有很多机器，我们可以把这些远程服务的机器归组，然后调用者可以选择不同的分组来调用，这样就可以将不同调用者对于同一服务的调用进行隔离了。

## 容器

* 还有两个问题需要解决，一是服务框架自身的部署方式问题，二是实现自己的服务框架所依赖的一些外部jar包与应用自身依赖jar包的冲突问题。
* 第一个问题：服务框架自身的部署方式问题。一种方案是把服务框架作为应用的一个依赖包并与应用一起打包。通过这种方式，服务框架就变为了应用的一个库，并随应用启动。存在的问题是，如果要升级服务框架，就需要更新应用本身，因为服务框架是与应用打包放在一起的，并且服务框架没有办法接管classloader，也就不能做一些隔离以及包的实现替换工作。
* 另外一种方案是把服务框架作为容器一部分，这里是针对Web应用来说的，而Web应用一般用JBoss、Tomcat等作为容易，我们就要遵循不同容器所支持的方法，把服务框架作为容器的一部分。
* Jar包冲突问题：将服务框架自身用的类与应用用到的类控制在User-Defined Class Loader，这样就实现了相互间的隔离。Web容器对于多个Web应用的处理，以及OSGi对于不同Bundle的处理都采用了类似的方法。此外，我们在实际中还会遇到需要在运行时统一版本的情况，那就需要服务框架比应用优先启动，并且把一些需要统一的jar包放到User-Defined Class Loader锁公用的祖先“ClassLoader”中。

# 通信方式

* 动态代理对象被调用后会进行如同服务请求方的处理，要完成寻址等工作。
* 有两种方式进行远程服务调用，第一种是通过中间的代理来解决。
* 我们这里的服务框架的设计采用的是另一种控制方案：调用者和提供者直接建立连接的方式，并且引入了一个服务注册查找中心的服务。
* 服务注册查找中心并不处在调用者和服务提供者之间，服务注册查找中心对于调用者来说，只是提供可用的服务提供者的列表。处于效率的考虑，并不是在每次调用远程服务前都通过这个服务注册查找中心来查找可用地址，而是把地址缓存在调用者本地，当有变化时主动从服务注册查找中心发起通知，告诉调用者可用的服务提供者列表的变化。
* 当客户端拿到可用的服务提供者的地址列表后，如何为当次的调用进行选择就是路由要解决的问题了。首先要考虑的就是集群的负载均衡。具体到负载均衡的实现，随机、轮询、权重是比较常见的实现方式，其中权重方式一般是指动态权重的方式，可以根据响应时间等参数来进行计算。在服务提供者的机器能力对等的情况下，采用随机和轮询这两种方式比较容易实现；在被调用的服务集群的机器能力不对等的情况下，使用权重计算的方式来进行路由比较合适。具体策略可以参考硬件负载均衡设备以及LVS、HAProxy等替代硬件负载均衡设备的系统所支持的策略。

# 基于接口、方法、参数的路由

* 从系统可用性和经济性的角度考虑，控制执行速度较慢的方法对正常情况的营销是比较合理的思路，第一种思路是增加资源保证系统的能力是超出需要的，第二种思路是隔离这些资源，从而使得快慢不同、重要级别不同的方法之间互不影响。
* 从客户端的角度来说，控制同一个集群中不同服务的路由并进行请求的隔离是一种可行方案。虽然集群中每台机器部署的代码是一样的，提供的服务也是一样的，但是通过路由的策略，我们让对于某些服务的请求到一部分机器，让另一些服务的请求到另一部分机器。
* 客户端的路由导致了请求的分流。在具体实现上，我们一般采用的方式把路由规则进行集中管理，在具体调用端的服务框架上获取规则后进行路由的处理，具体来说，是根据服务定位提供服务的那个集群的地址，然后与接口路由规则中的地址一起取交集，得到的地址列表再进行接下来的负载均衡算法，最后得到一个可用的地址去调用。
* 以上是基于接口的路由。可以基于接口的具体方法来进行路由。该方式只是在通过接口定位到服务地址列表后，根据接口加方法名从规则中得到一个服务地址列表，再和刚才的地址列表取交集。
* 一般到基于方法的路由就够用了，需要对一些特定参数进行特殊处理的情况才会使用基于参数的路由。

# 多机房场景

* 每个机房都有自己的容量上线，如果网站的规模非常大，那就需要多个机房了。机房之间的距离和分工决定了我们应该采用什么样的架构和策略，这里进讨论距离比较近的同城机房的情况。
* 如果不做任何处理，服务注册查找中心会把服务提供者1的所有机器看做一个集群，尽管它们分布在两个机房。这样，分布在两个机房的调用者1就会对等地看待分布在不同机房的服务提供者1的机器。如果能避免跨机房调用，就能提升系统稳定性。
* 有两种方案可以实现这个想法：一是在服务注册查找中心做一些工作，通过它来甄别不同机房的调用者集群，给它们不同服务提供者的地址。另一种方式是通过路由来完成。大概思路是，服务注册查找中心给不同机房的调用者相同的服务提供者列表，我们在服务框架内部进行地址过滤，过滤的原则一般是基于接口等路由规则进行集中管理配置。在具体实践中，一方面需要考虑两个甚至多个机房的部署能力是否对等，也就是说通过路由使服务都走本地的话，负载是否均衡。此外还有一个异常的情况需要考虑，即如果某个机房的服务提供者大面积不可用，而另外机房的服务提供者是正常运行并且有余量提供服务，那么如何让服务提供者大面积不可用的机房的调用者调用远程的服务呢，这个是需要解决的问题。

# 服务调用者的流控处理

* 流量控制一般来说是有两种，一种是0-1开关，也就是说完全打开不进行流控；另一种是设定一个固定的值，表示每秒可以进行的请求次数，超过这个请求次数的话就拒绝对远程的请求。那些被流量控制拒绝的请求，可以直接返回给调用者，也可以进行排队。
* 基于下面两个维度进行控制：
  + 1)根据服务端自身的接口、方法做控制，针对不同的接口、方法设置不同的阈值，这是为了使服务端的不同接口、方法之间的负载不相互影响。
  + 2)根据来源做控制，也就是对于同样的接口、方法，根据不同来源设置不同的限制。这一般用在比较基础的服务上，也就是在多个集群使用同样的服务时，根据请求来源的不同等级等进行不同的流控处理。

# 序列化

* 如果在整个分布式系统中的调用者或者服务听着要使用Java以外的语言来实现，那么序列化和反序列化的方式就要支持跨语言。第二，性能开销也是需要注意的点。第三，还需要足以序列化后的长度。
* 我们从两个方面来看协议的部分，一个是用于通信的数据报文的自定义协议，另一个是远程过程调用本身的协议。
* 前者比如XML或者JSON，后者比如HTTP或TCP。

# 网络通信实现

* BIO、NIO、AIO。

# 异步服务调用

- 1)Oneway：只管发送请求而不关心结果的方式。

* 可以看到Oneway方式非常简单，只需要把要发送的数据放入数据队列，然后就可以继续处理后续的任务了；而IO线程也只需要从数据队列中得到数据，然后通过Socket连接送出去就好了。Oneway方式不关心对方是否收到了数据，也不关心对方收到数据后做什么或有什么返回，这就基本等价于一个不保证可靠送达的通知。
  + 2)Callback：这种方式下请求方发送请求后会继续执行自己的操作，等对方有响应时给一个回调。
  + 3)Future
  + 4)可靠异步：消息中间件

# 服务提供端的设计与实现

# 暴露远程服务

* 服务需要注册到服务注册查找中心后才能被服务调用者发现。所以ProviderBean需要将自己所代表的服务注册到服务注册查找中心。另外，当请求调用端定位到提供服务的机器并且请求被送达到提供服务的机器上后，在本机也需要有一个服务与具体对象的对应关系。ProviderBean也需要在本地注册服务和对应服务实例的关系。

# 请求处理

* 接收到请求后，通过协议解析及反序列化，可以得到请求发送端调用服务方法的具体信息，再根据其中的服务名称、版本号找到本地提供服务的具体对象，然后再用传过来的参数调用相关对象的方法就可以了。

# 线程池隔离

* 服务提供端的工作线程是一个线程池，路由到本地的服务请求会被放入这个线程池执行，如果客户端没有提供接口或方法进行路由，我们就可以在服务提供端进行控制，也就是进行服务端线程池隔离。具体的做法其实十分类似于请求调用方根据接口、方法、参数进行的路由。在服务提供端，工作线程池不止一个，而是多个，当定位到服务后，我们根据服务名称、方法、参数来确定具体执行服务调用的线程池是哪个。这样不同的线程池就是隔离的，不会出现争抢线程资源的情况。

# 服务提供端的流控处理

* 在服务提供者看来，不同来源的服务调用者、0-1的开关以及限制具体数值的QPS的方式都需要实现。并且在服务提供者这里，某个服务或者方法可以对不同服务调用者进行不同对待。这样的做法就是对不同的服务调用者进行分级，确保优先级高的服务调用者而被优先提供服务。

# 服务升级

* 对于服务升级，会遇到两种情况。第一种情况是接口不变，只是代码本身进行完善。这样的情况处理起来比较简单，因为提供给使用者的接口、方法都没有变，只是内部的服务实现有变化。这种情况下，采用灰度发布的方式验证然后全部发布就可以了。第二种情况是需要修改原有的接口，这又分成以下两种情况：
* 一是在接口中添加方法，这个情况比较简单，直接增加方法就行了。并且需要使用新方法的调用者就使用新方法，原来的调用者继续使用原来的方法即可。
* 二是要对接口的某些方法修改调用的参数列表，有几种方式来应对：
  + 1)对使用原来方法的代码都进行修改，，然后和服务端一起发布。这种方法不太可行，因为这要求我们同时发布多个系统，而且一些系统可能并不会从调整参数后的方法那里受益。
  + 2)通过版本号来解决。使用老方法的系统继续调用原来版本的服务，而需要使用新方法的系统则使用新版本的服务。
  + 3)从设计方法上考虑参数的扩展性，比如参数为Map<String,Object>。这个方法不太好，不直观，而且对参数的校验会比较复杂。

# 服务治理

* 服务治理可以分为管理服务和查看服务。

# 服务查看

# 服务管理

# 服务框架与ESB的对比

* ESB的概念是从SOA发展过来的，它是对多样系统中的服务调用者和服务提供者的解耦。ESB本身也可以解决服务化的问题，它提供了服务暴露、接入、协议转换、数据格式转换、路由等方面的支持。ESB和服务框架主要有两个差异。第一，服务框架是一个点对点的模型，而ESB是一个总线式的模型；第二，服务框架基本上是面向同构的系统，不会重点考虑整合的需求，而ESB会更多地考虑不同厂商所提供服务的整合。

# 分布式一致性

# CAP+BASE

# CAP

* CAP定理是2000年，由 Eric Brewer 提出来的。Brewer认为在分布式的环境下设计和部署系统时，有3个核心的需求，以一种特殊的关系存在。这里的分布式系统说的是在物理上分布的系统。
* CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。  
  - Consistency（一致性)：所有的节点在同一时间读到相同的数据。这就是数据上的一致性，当数据写入成功后，所有的节点会同时看到这个新的数据  
  - Availability（可用性)：保证无论是成功还是失败，每个请求都能够收到一个反馈。这就是数据的可用性。重点是系统一定要有响应。  
  - Partition-Tolerance（分区容错性)：即使系统中有部分问题或者有消息的丢失，系统仍能够继续运行，这被称为分区容错性，也就是在系统的一部分出问题时，系统仍能继续工作。
* 因此，根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则、满足 CP 原则和满足 AP 原则三大类：
* CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。
* 一种较为简单的做法是将所有的数据（或者是仅仅与事务相关的数据)都放在一个分布式节点上。这样的做法虽然无法100%地保证系统不会出错，但至少不会碰到由于网络分区带来的负面影响，放弃P的同时也意味着放弃了系统的可扩展性。
* CP - 满足一致性，分区容错性的系统。
* 一旦系统遇到网络分区或者其他故障时，那么受到影响的服务需要等待一定的时间，因此在等待期间系统无法对外提供正常服务，即不可用。
* AP - 满足可用性，分区容错性的系统，通常可能对一致性要求低一些。  
  放弃一致性指的是放弃强一致性，而保留数据的最终一致性，这样的系统无法保证数据保持实时的一致性，但是能够承诺的是，数据最终会达到一个一致的状态。这就引入了一个时间窗口的概念，具体多久能够达到数据一致性取决于系统的设计，主要包括数据副本在不同节点之间的复制时间长短。
* CA 传统单机数据库
* AP 很多NoSQL
* CP 网络的问题可能会让整个系统不可用

# BASE

* BASE就是为了解决关系数据库强一致性引起的问题而造成可用性降低而提出的解决方案。
* BASE其实是下面三个术语的缩写：
* 基本可用（Basically Available)
* 软状态（Soft state)
* 最终一致（Eventually consistent)
* 它的思想是通过让系统放松对某一时刻数据一致性的要求来换取系统整体伸缩性和性能上改观。为什么这么说呢，缘由就在于大型系统往往由于地域分布和极高性能的要求，不可能采用分布式事务来完成这些指标，要想获得这些指标，我们必须采用另外一种方式来完成，这里BASE就是解决这个问题的办法。

## 基本可用

* 分布式系统在出现不可预知故障的时候，允许损失部分可用性，比如
  + 1)响应时间上的损失：出现故障时响应稍慢
  + 2)功能上的损失：部分用户可能会被引导到一个降级页面

## 软状态

* 允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

## 最终一致性

* 系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。
* 在实际工程实践中，最终一致性存在以下五类主要变种：

# 数据一致性模型

* 一些分布式系统通过复制数据来提高系统的可靠性和容错性，并且将数据的不同的副本存放在不同的机器，由于维护数据副本的一致性代价高，因此许多系统采用弱一致性来提高性能，一些不同的一致性模型也相继被提出。

- 强一致性： 要求无论更新操作实在哪一个副本执行，之后所有的读操作都要能获得最新的数据。  
- 弱一致性：用户读到某一操作对系统特定数据的更新需要一段时间，我们称这段时间为“不一致性窗口”。  
- 最终一致性：是弱一致性的一种特例，保证用户最终能够读取到某操作对系统特定数据的更新。

* 常用的锁实现算法有Lamport bakery algorithm （俗称面包店算法)， 还有Paxos算法以及乐观锁。
* Paxos算法是2PC的升级版，比2PC更加轻量级的保证一致性的协议。

# 一致性协议

# 2PC

* 见上面的2PC

# 3PC（3 Phase Coimmit)

* 是2PC的改进版，其将2PC的提交事务请求过程一分为二，形成了由CanCommit、PreCommit和doCommit三个阶段组成的事务处理协议。
* 在2PC中一个参与者的状态只有它自己和协调者知晓，假如协调者提议后自身宕机，一个参与者又宕机，其他参与者就会进入既不能回滚、又不能强制commit的阻塞状态，直到参与者宕机恢复。这引出两个疑问：
* 能不能去掉阻塞，使系统可以在commit/abort前回滚(rollback)到决议发起前的初始状态？
* 当次决议中，参与者间能不能相互知道对方的状态，又或者参与者间根本不依赖对方的状态？

## 阶段一：CanCommit

* 1、事务询问
* 协调者向所有的参与者发送一个包含事务内容的canCommit请求，询问是否可以执行事务提交操作，并开始等待各参与者的响应
* 2、各参与者向协调者反馈事务询问的响应
* 参与者接收到来自协调者的canCommit请求后，正常情况下，如果其自身认为可以顺利执行事务，那么会反馈yes响应，并进入预备状态，否则反馈no响应。

## 阶段二：PreCommit

* 在阶段二中，协调者会根据各参与者的反馈情况来决定是否可以进行事务的PreCommit操作，正确情况下，包含两种可能。

### 执行事务预提交

* 假如协调者从所有的参与者获得的反馈都是yes，那么就会执行事务预提交。
* 1、发送预提交请求
* 协调者向所有参与者节点发出preCommit请求，并进入Prepared阶段
* 2、事务预提交
* 参与者接收到preCommit请求后，会执行事务操作，并将Undo和Redo信息记录到事务日志中
* 3、各参与者向协调者反馈事务执行的响应
* 如果参与者成功执行了事务操作，那么就会反馈给协调者ACK响应，同时等待最终的指令：提交（commit)或中止（abort)。如果失败，那么会反馈给协调者no响应。

### 中断事务

* 假如任何一个参与者向协调者反馈了no响应，或者在等待超时之后，协调者尚无法接收到所有参与者的反馈响应，那么就会中断事务。
* 1、发送中断请求
* 协调者向所有参与者节点发出abort请求
* 2、中断事务
* 无论是收到来自协调者的abort请求，或者是在等待协调者请求过程中出现超时，参与者都会中断事务

## 阶段三：doCommit

* 该阶段将进行真正的事务提交，会存在以下两种可能的情况

### 执行提交

* 1、发送提交请求
* 假设协调者处于正常工作状态，并且它接收到了来自所有参与者的ACK响应，那么它将从“预提交”状态转换到“提交”状态，并向所有的参与者发送doCommit请求。
* 2、事务提交
* 参与者接收到doCommit请求后，会正式执行事务提交操作，并在完成提交之后释放在整个事务执行阶段占用的事务资源
* 3、反馈事务提交结果
* 参与者在完成事务提交之后，向协调者发送ACK消息
* 4、完成事务
* 协调者接收到所有参与者反馈的ACK消息后，完成事务。

### 中断事务

* 进入这一阶段，假设协调者处于正常工作状态，并且有任意一个参与者向协调者反馈no响应，或者在等待超时之后，协调者尚无法接收到所有参与者的反馈响应，那么就会中断事务。
* 1、发送中断请求
* 协调者向所有的参与者发送abort请求
* 2、事务回滚
* 参与者接收到abort请求后，会利用其在阶段二记录的Undo信息来执行事务回滚操作，并在完成回滚之后释放在整个事务执行期间占用的资源。
* 3、反馈事务回滚结果
* 参与者在完成事务回滚之后，向协调者发送ACK消息
* 4、中断事务
* 协调者接收到所有参与者反馈的ACK消息后，中断事务。
* 一旦进入阶段三，可能会存在以下两种故障：
  + 1)协调者出现问题
  + 2)协调者和参与者之间的网络出现故障
* 无论出现哪种情况，都会导致参与者无法及时接收到来自协调者的doCommit或abort请求，针对这种异常情况，参与者会在等待超时之后，继续进行事务提交。

## 优缺点

* 优点：相较于2PC，最大的优点是降低了参与者的阻塞范围，并且能够在出现单点故障后继续达成一致
* 缺点：在参与者接收到preCommit消息后，如果网络出现分区，此时协调者所在的节点和参与者无法进行正常的网络通信，此时该参与者依然会进行事务的提交，这必然会出现数据的不一致性。

# Paxos

* 如何在一个可能发生机器宕机或者网络异常的分布式系统中，快速且正确地在集群内部对某个数据的值达成一致，不会破坏数据一致性。

## 拜占庭将军问题

* 拜占庭帝国有许多支军队，不同军队的将军之间必须制定一个统一的行动计划，从而做出仅供或者撤退的决定，同时，各个将军在地理上都是被分隔开的，只能依靠军队的通讯员来进行通讯。然而，在所有的通讯员中可能会存在叛徒，这些叛徒可以篡改信息，从而达到欺骗将军的目的。
* 事实上，大多数系统都是部署在同一个局域网中的，因此消息被篡改的情况非常罕见。另一方面，由于硬件和网络原因造成消息不完整的问题，只需一套简单的校验算法既可避免。因此，在实际工程时间中，可以假设不存在拜占庭将军问题，也就是假设所有消息都是完整的，没有被篡改的，那么，在这种情况下，需要什么样的算法来保证一致性呢？

## Paxos场景

* 在古希腊有一个叫做Paxos的小岛，岛上采用议会的形式来通过法令，议会中的议员通过信使进行消息的传递。议员和信使都是兼职的，他们随时有可能会离开议会厅，并且信使可能会重复地传递消息，也可能一去不复返。因此，议会协议要保证在这种情况下法令仍然能够正确的产生，并且不会出现冲突。

## 算法

* 将所有节点都写入同一个值，且被写入后不再更改。
* proposer将发起提案（value)给所有accpetor，超过半数accpetor获得批准后，proposer将提案写入accpetor内，最终所有accpetor获得一致性的确定性取值，且后续不允许再修改。
* 一、两个操作：
* Proposal Value：提议的值；
* Proposal Number：提议编号，可理解为提议版本号，要求不能冲突；
* 二、三个角色：
* Proposer：提议发起者。Proposer 可以有多个，Proposer 提出议案（value)。所谓 value，可以是任何操作，比如“设置某个变量的值为value”。不同的 Proposer 可以提出不同的 value，例如某个Proposer 提议“将变量 X 设置为 1”，另一个 Proposer 提议“将变量 X 设置为 2”，但对同一轮 Paxos过程，最多只有一个 value 被批准。
  + Acceptor：提议接受者；Acceptor 有 N 个，Proposer 提出的 value 必须获得超过半数(N/2+1)的 Acceptor批准后才能通过。Acceptor 之间完全对等独立。
* Learner：提议学习者。上面提到只要超过半数accpetor通过即可获得通过，那么learner角色的目的就是把通过的确定性取值同步给其他未确定的Acceptor。
* 三、协议过程

### 准备阶段

#### 第一阶段A：Proposer发送Prepare

* Proposer生成全局唯一且递增的提案ID（比如时间戳+IP+序列号)，向Acceptor集群发送请求，无需携带提案内容，只携带提案ID即可（提案ID称为Pn)。

#### 第一阶段B：Acceptor应答Proposer

* Acceptor收到提案请求Pn过后，做出以下约定：
  + 1)不再应答<= Pn的请求
  + 2)对于Accept请求也不做处理
* Acceptor会做的处理：
  + 1)应答前要保存/持久化当前提案ID proposalID
  + 2)如果当前请求的提案ID（Pn)大于此前存放的proposalID，则将proposalID更新为当前ID（应答)。

### 接受阶段

#### 第二阶段A：Proposer发送Accept

* Proposer收集到多数派（过半)应答Prepare阶段的返回值后，从中选择proposalID最大的提案内容，作为要发起Accept的提案，如果这个提案内容为空值，则可以随意决定提案内容。然后携带上当前proposalID，向Acceptor集群发送Accept请求。

#### 第二阶段B：Acceptor应答Accept

* Acceptor接收到Accept请求后，检查不违背自己之前作出约定的情况下，保存/持久化当前proposalID和提案内容。最后Proposer收集到多数派应答的Accept回复后，形成决议。

# Quorum/NWR

* 借鉴了Paxos的思想，实现上更加简洁，同样解决了在多个节点并发写入时的数据一致性问题。
* 这个协议有三个关键字N、R、W。
* • N代表数据所具有的副本数。
* • R表示读取一个数据需要读取的拷贝的份数（值越大读性能越差)
* • W表示更新一个数据对象时需要确保成功的份数（值越大写性能越差)
* 该策略中，只需要保证R+W>N，就可以保证强一致性。
* 例如：N=3，W=2，R=2，那么表示系统中数据有3个不同的副本，当进行写操作时，需要等待至少有2个副本完成了该写操作系统才会返回执行成功的状态，对于读操作，系统有同样的特性。由于R + W > N，因此该系统是可以保证强一致性的。
* R + W> N会产生类似Quorum的效果。该模型中的读（写)延迟由最慢的R(W)副本决定，有时为了获得较高的性能和较小的延迟，R和W的和可能小于N，这时系统不能保证读操作能获取最新的数据。
* 如果R + W > N，那么分布式系统就会提供强一致性的保证，因为读取数据的节点和被同步写入的节点是有重叠的。在关系型数据管理系统中，如果N=2，可以设置为W=2，R=1，这是比较强的一致性约束，写操作的性能比较低，因为系统需要2个节点上的数据都完成更新后才将确认结果返回给用户。
* 如果R + W ≤ N，这时读取和写入操作是不重叠的，系统只能保证最终一致性，而副本达到一致的时间则依赖于系统异步更新的实现方式，不一致性的时间段也就等于从更新开始到所有的节点都异步完成更新之间的时间。
* R和W的设置直接影响系统的性能、扩展性与一致性。如果W设置为1，则一个副本完成更改就可以返回给用户，然后通过异步的机制更新剩余的N-W的副本；如果R设置为1，只要有一个副本被读取就可以完成读操作，R和W的值如较小会影响一致性，较大则会影响性能，因此对这两个值的设置需要权衡。
* 下面为不同设置的几种特殊情况:
  1. 当W=1,R=N时，系统对写操作有较高的要求，但读操作会比较慢，若N个节点中有节点发生故障，那么读操作将不能完成。
  2. 当R=1,W=N时，系统对读操作有较高性能、高可用，但写操作性能较低，用于需要大量读操作的系统，若N个节点中有节点发生故障，那么些操作将不能完成。
     1. 当R=Q,W=Q(Q=N/2+1)时，系统在读写性能之间取得平衡，兼顾了性能和可用性。

# Raft

* Raft把一致性问题分解成为三个小问题:
* leader election 选举
* log replication 日志复制
* safety 安全性

## 基本概念

* 每个Server有三个状态: leader, follower, candidate
* follower: 不发request而只会回复leader和candidate的request.
* leader: 处理client发过来的请求
* candidate: leader的候选人
* Raft把时间分为terms. 每一个term开始时都进行一次选举. 每一个term里最多有一个leader, 或者没有leader.

## RPC实现

* 算法需要两种RPC

### RequestVote RPC

* 由candidates在选举过程中发起,当另外一个server收到这个RPC之后, 只有当对方term和log都至少和自己的一样新的时候才会投赞成票,收到多数赞成票的candidate会当选leader.

### AppendEntries RPC

* 由leader发起用来分发日志, 强迫follwer的log和自己一致.

## Leader election

* 如果一个follower在election timeout的时间里没有收到leader的信息,就进入新的term,转成candidate,给自己投票,发起选举 RequestVote RPC. 这个状态持续到发生下面三个中的任意事件：
  + 1)它赢得选举
  + 2)另外有Server赢得选举
  + 3)1个term过去了,还是没有选举结果（Split Votes)
  + 为什么会有3)这个情况呢,就是当如果大家同时发起选举,都投给自己,那就没有Server能够得到多数选票了,这个时候就要进入下一个term,再选一次. 为了避免这个情况持续发生,每个Server的election timeout被随机的设成不同的值,所以先timeout的就可以先发起下一次选举.
* 初始状态 ABC 都是 Follower，然后发起选举这时有三种可能情形发生。下图中前二种都能选出 Leader，第三种则表明本轮投票无效（Split Votes)，每方都投给了自己，结果没有任何一方获得多数票。之后每个参与方随机休息一阵（Election Timeout)重新发起投票直到一方获得多数票。这里的关键就是随机 timeout，最先从 timeout 中恢复发起投票的一方向还在 timeout 中的另外两方请求投票，这时它们就只能投给对方了，很快达成一致。

## Log replication

* 选好leader之后就可以分发log啦.
* 每一个log都有一个log index 和 term number. 当大多数的follower都复制好这个log时,就说这个log是committed,可以执行了. Leader 记住已经commit的最大log index, 用它来分发下一个 AppendEntries RPC. 这个和TCP里段的编号的作用是一样的.
* 当一个leader重新选出来时,它的log和follower的log可能不一致,那么它会强制所有的follower都和自己的log一致.首先leader要找到和follower之间的最大的编号一致的log,然后覆盖掉那之后的log.
* 数据的流向只能从 Leader 节点向 Follower 节点转移。当 Client 向集群 Leader 节点提交数据后，Leader 节点接收到的数据处于未提交状态（Uncommitted)，接着 Leader 节点会并发向所有 Follower 节点复制数据并等待接收响应，确保至少集群中超过半数节点已接收到数据后再向 Client 确认数据已接收。一旦向 Client 发出数据接收 Ack 响应后，表明此时数据状态进入已提交（Committed)，Leader 节点再向 Follower 节点发通知告知该数据状态已提交。

## Safety

### Follower宕机

* 但是到目前为止仍然不能保证安全性.比如说, 当leader在commit log时, 某follower宕机,然后这个follower后来被选为leader,它会覆盖掉现在follwer那些已经committed log, 由于这些log是已经执行过的,所以结果不同的机器就执行不同的指令. 在选举过程中,再加多一个限制就可以防止这种情况发生, 即:
* Leader completeness property:
* 对于任意一个term, leader都要包含所有在之前term里committed的log。

### Leader宕机

#### 数据到达 Leader 节点前

* 这个阶段 Leader 挂掉不影响一致性

#### 数据到达 Leader 节点，但未复制到 Follower 节点

* 这个阶段 Leader 挂掉，数据属于未提交状态，Client 不会收到 Ack 会认为超时失败可安全发起重试。Follower 节点上没有该数据，重新选主后 Client 重试重新提交可成功。原来的 Leader 节点恢复后作为 Follower 加入集群重新从当前任期的新 Leader 处同步数据，强制保持和 Leader 数据一致。

#### 数据到达 Leader 节点，成功复制到 Follower 所有节点，但还未向 Leader 响应接收

* 这个阶段 Leader 挂掉，虽然数据在 Follower 节点处于未提交状态（Uncommitted)但保持一致，重新选出 Leader 后可完成数据提交，此时 Client 由于不知到底提交成功没有，可重试提交。针对这种情况 Raft 要求 RPC 请求实现幂等性，也就是要实现内部去重机制。

#### 数据到达 Leader 节点，成功复制到 Follower 部分节点，但还未向 Leader 响应接收

* 这个阶段 Leader 挂掉，数据在 Follower 节点处于未提交状态（Uncommitted)且不一致，Raft 协议要求投票只能投给拥有最新数据的节点。所以拥有最新数据的节点会被选为 Leader 再强制同步数据到 Follower，数据不会丢失并最终一致。

#### 数据到达 Leader 节点，成功复制到 Follower 所有或多数节点，数据在 Leader 处于已提交状态，但在 Follower 处于未提交状态

* 这个阶段 Leader 挂掉，重新选出新 Leader 后的处理流程和阶段 3 一样。

#### 数据到达 Leader 节点，成功复制到 Follower 所有或多数节点，数据在所有节点都处于已提交状态，但还未响应 Client

* 这个阶段 Leader 挂掉，Cluster 内部数据其实已经是一致的，Client 重复重试基于幂等策略对一致性无影响。

#### 网络分区导致的脑裂情况，出现双 Leader

* 网络分区将原先的 Leader 节点和 Follower 节点分隔开，Follower 收不到 Leader 的心跳将发起选举产生新的 Leader。这时就产生了双 Leader，原先的 Leader 独自在一个区，向它提交数据不可能复制到多数节点所以永远提交不成功。向新的 Leader 提交数据可以提交成功，网络恢复后旧的 Leader 发现集群中有更新任期（Term)的新 Leader 则自动降级为 Follower 并从新 Leader 处同步数据达成集群数据一致。

# MVCC

* 多版本并发控制（Multiversion concurrency control)
* 是一种宽松的设计，读写相互不阻塞。
* 与MVCC相对的，是基于锁的并发控制，Lock-Based Concurrency Control。
* MVCC协议中，每个用户在连接数据库时看到的是一个具有一致性状态的镜像，每个事务在提交到数据库之前对其他用户均是不可见的。当事务需要更新数据时，不会直接覆盖以前的数据，而是生成一个新的版本的数据，因此一条数据会有多个版本存储，但是同一时刻只有最新的版本号是有效的。因此，读的时候就可以保证总是以当前时刻的版本的数据可以被读到，不论这条数据后来是否被修改或删除。

# Gossip

* 在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，最终所有节点的状态都会达成一致。每个节点可能知道所有其他节点，也可能仅知道几个邻居节点，只要这些节可以通过网络连通，最终他们的状态都是一致的，当然这也是疫情传播的特点。
* 要注意到的一点是，即使有的节点因宕机而重启，有新节点加入，但经过一段时间后，这些节点的状态也会与其他节点达成一致，也就是说，Gossip天然具有分布式容错的优点。
* Gossip是一个带冗余的容错算法，更进一步，Gossip是一个最终一致性算法。虽然无法保证在某个时刻所有节点状态一致，但可以保证在”最终“所有节点一致，”最终“是一个现实中存在，但理论上无法证明的时间点。
* 因为Gossip不要求节点知道所有其他节点，因此又具有去中心化的特点，节点之间完全对等，不需要任何的中心节点。实际上Gossip可以用于众多能接受“最终一致性”的领域：失败检测、路由同步、Pub/Sub、动态负载均衡。
* 但Gossip的缺点也很明显，冗余通信会对网路带宽、CUP资源造成很大的负载，而这些负载又受限于通信频率，该频率又影响着算法收敛的速度，后面我们会讲在各种场合下的优化方法。

## 状态的传播

* A节点首先知道了msg，它首先将此msg传播给集群中的部分节点（比如相邻节点)，B和C。后者再将其传递到它们所选择的部分节点。以此类推，最终来自于A的msg在数轮交互后被传播到了集群中的所有节点。
* 在分布式系统中，msg可能是某个节点所感知的关于其他节点是否宕机的认识；也可能是数据水平拆分的缓存集群中，关于哪些hash桶分布在哪些节点上的信息。每个结点起初只掌握部分状态信息，不断从其他节点收到gossip msg，每个节点逐渐地掌握到了整个集群的状态信息，因此解决了状态同步的第一个问题：全局状态的获取。
* 对于集群中出现的部分网络分隔，因为信息也能通过别的路径传播到整个集群，所以可以解决。

## 状态的一致

* 状态同步的第二个问题：对于同一条状态信息，不同的节点可能掌握的值不同，也能通过基于gossip通信思路构建的协议包版本得到解决。
* 以Reids水平拆分的集群为例：
* 此时各个节点预先通过某种协议（比如gossip)得知了集群的状态全集，此时新加入了节点D。
  + D分担了C的的8号哈希桶，此时C/D和集群中的其他节点就C所拥有哪些hash桶这件事产生了产生了分歧：A、B认为C目前有6、7、8号哈希桶。此时通过gossip消息体引入版本号，使得关于C的最新状态信息（只有6、7)在全集群达到一致。
  + 例如B收到来自A和C的gossip消息时会将版本号最新的消息（来自C的v2)更新到自己的本地副本中。
* 各个节点的本地副本保存的集群全量状态也可以用来表示各个节点的存活状态。
* 例如A和C的网络断开，但A和C本身都正常运行，此时A和C互相无法通信，C会将A标记为dead、对于中心化思路的协议，如果C恰好是中心节点，那么A不可用的信息将会同步到集群的所有节点上，使得这些节点将其实可用的A也标记为宕机。而基于gossip这类去中心化的协议进行接收到消息后实现逻辑扩展（例如只有当接收到大多数的节点关于A已经宕机的消息时，才更新A的状态)，最终保证A不被误判为宕机

## 特性总结

* gossip的核心是在去中心化结构下，通过信息的部分传递，达到全集群的状态信息传播，传播的时间收敛在O(logn)以内，N是结点数量。同时基于gossip协议，可以构建出状态一致的各种方案。

# 分布式锁

* 分布式锁 zk实现分布式锁的方式；redis实现；数据库乐观锁
* 从实现的复杂性角度（从低到高): Zookeeper >= 缓存 > 数据库
* 从性能角度（从高到低): 缓存 > Zookeeper >= 数据库
* 从可靠性角度（从高到低): Zookeeper > 缓存 > 数据库

# 数据库悲观锁

* 直接建一张表，里面记录锁定的方法名 时间 即可。
* CREATE TABLE methodLock (
  + id int(11) NOT NULL AUTO\_INCREMENT COMMENT '主键',
  + method\_name varchar(64) NOT NULL DEFAULT '' COMMENT '锁定的方法名',
  + desc varchar(1024) NOT NULL DEFAULT '备注信息',
* update\_time timestamp NOT NULL DEFAULT CURRENT*TIMESTAMP ON UPDATE CURRENT*TIMESTAMP COMMENT '保存数据时间，自动生成',
* PRIMARY KEY (id),
* UNIQUE KEY uidx\_method\_name (method\_name) USING BTREE
* ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';
* 获得锁：
* select \* from methodLock where method\_name = #{currentMethod} for update;
* 释放锁：
* commit

## 缺点

- 1)这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。  
- 2)这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

* 这里还可能存在另外一个问题，虽然我们对method\_name 使用了唯一索引，并且显示使用for update来使用行级锁。但是，MySql会对查询进行优化，即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。

# 数据库乐观锁

* select state,version from methodLock where method\_name = #{currentMethod};
* update methodLock set state = ‘locked’,version = version + 1,update*time = now() where method*name = #{currentMethod} and state = ‘unlock’ and version = #{version}
* 乐观锁只能解决持久化是DB数据的一次更新问题。假如你的数据不是在DB，或者一个过程有三个数据的更新操作，线程A更新了数据1和数据2，线程B更新了数据3，乐观锁就不能起作用。

# 缓存

* 相比于用数据库来实现分布式锁，基于缓存实现的分布式锁的性能会更好一些。
* 目前有很多成熟的分布式产品，包括Redis、memcache、Tair等。
* 获取锁的使用，使用setnx加锁，将值设为当前的时间戳，再使用expire设置一个过期值。
* 获取到锁则执行同步代码块，没获取则根据业务场景可以选择自旋、休眠、或做一个等待队列等拥有锁进程来唤醒（类似Synchronize的同步队列),在等待时使用ttl去检查是否有过期值，如果没有则使用expire设置一个。
* 执行完毕后，先根据value的值来判断是不是自己的锁，如果是的话则删除，不是则表明自己的锁已经过期，不需要删除。（此时出现由于过期而导致的多进程同时拥有锁的问题)

## 优点

* 性能好，实现起来较为方便。

## 缺点

* 通过超时时间来控制锁的失效时间并不是十分的靠谱。
* 非可重入

# Zookeeper

* 基于zookeeper临时有序节点可以实现的分布式锁。
* 大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

## 优点

- 1)无单点问题。ZK是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。  
  
- 2)持有锁任意长的时间，可自动释放锁。使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在ZK中创建一个临时节点，一旦客户端获取到锁之后突然挂掉（Session连接断开)，那么这个临时节点就会自动删除掉。其他客户端就可以再次获得锁。这避免了基于Redis的锁对于有效时间(lock validity time)到底设置多长的两难问题。实际上，基于ZooKeeper的锁是依靠Session（心跳)来维持锁的持有状态的，而Redis不支持Sesion。  
  
- 3)可阻塞。使用Zookeeper可以实现阻塞的锁，客户端可以通过在ZK中创建顺序节点，并且在节点上绑定监听器，一旦节点有变化，Zookeeper会通知客户端，客户端可以检查自己创建的节点是不是当前所有节点中序号最小的，如果是，那么自己就获取到锁，便可以执行业务逻辑了。  
  
- 4)可重入。客户端在创建节点的时候，把当前客户端的主机信息和线程信息直接写入到节点中，下次想要获取锁的时候和当前最小的节点中的数据比对一下就可以了。如果和自己的信息一样，那么自己直接获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。

* zookeeper第三方库Curator客户端，这个客户端中封装了一个可重入的锁服务。

## 缺点

* zookeeper实现的分布式锁其实存在一个缺点，那就是性能上可能并没有缓存服务那么高。因为每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。ZK中创建和删除节点只能通过Leader服务器来执行，然后将数据同不到所有的Follower机器上。

# 注册查找中心/软负载中心

# 职责

* 软负载中心有两个最基础的职责：
  + 1)聚合地址信息
  + 2)生命周期感知
* 注册查找中心需要对服务的上下线自动感知，并且根据这个变化去更新服务地址数据，形成新的地址列表后，把数据传给需要数据的调用者或者消息的发送者和接收者。

# 结构

* 包括客户端和服务端。服务端主要负责感知提供服务的机器是否在线，聚合提供者的机器信息，并且负责把数据传给使用数据的应用。客户端承载了两个角色，作为服务提供者，客户端主要是把服务提供者挺服务的具体信息主动传给服务端，并且随着提供服务的变化去更新数据；而作为服务使用者，客户端主要是向服务端告知自己所需要的数据并负责去更新数据，还要进行本地的缓存。
* 注册查找中心中内部有三部分重要的数据：
  + 1)聚合数据：聚合后的地址信息列表通过dataId和group就可以定位到一个唯一的键值对
  + 2)订阅关系：服务使用者/数据订阅者把自己所需要的数据信息告诉注册查找中心，这就是一个订阅关系。订阅的粒度和聚合树的粒度是一致的，就是通过dataId和group来确定数据，那么会有dataId、group到数据订阅者的分组Id（consumerGroupId)的一个映射关系。当聚合的数据有变化时，也是通过订阅关系的数据找到需要通知的数据订阅者，然后去进行数据更新通知。
  + 3)连接数据：连接到注册查找中心的节点和注册查找中心已经建立的连接管理。连接数据以groupId作为key，然后对应管理这个物理连接的，基于长连接。当订阅的数据发生变化时，通过订阅关系找到需要通知的groupId，然后进行数据发送，完成对应用的数据更新。

# 内容聚合

* 内容聚合需要完成的工作主要有是两个：
  + 1)保证数据正确性：并发场景下的数据聚合的正确性；另外需要考虑的是发布数据的机器短时间上下线的问题。
  + 2)高效地聚合数据
* 可以用一个Map来存储数据，用dataId和groupId作为key，对应的value就是聚合后的数据。
* Map<dataId,Map<groupId,Listaddress>>
* 有几个关键点要注意

## 并发下数据正确性的保证

* 加锁或并发容器

## 数据更新、删除的顺序保证

* NIO+Selector时更新、新增数据和连接断开要去删除数据就可能在两个线程中处理。而如果是发布数据后很快断开，那么保证在内部按照顺序来处理就很关键。因为如果顺序不保证，我们就可能先处理了删除数据，然后再处理新增，这样数据就不对了。一个解决的办法是在插入数据时判断当前产生数据的发布方的连接是否存在。

## 大量数据同时插入、更新时的性能保证

* 并发容器选用ConcurrentHashMap。
* 对于同样的dataId，group对应的数据保存，采用LinkedList需要加锁。根据dataId、group进行分线程的处理，我们可以保证同一个dataId，group的数据是在同一个线程中处理，这样可以把整个数据结构变成一个不需要锁的数据结构。
* 对于同样dataId、group的增改删是可以分线程处理的，读取自然也可以分线程。分线程的话可以实现任务队列来实现。

# 服务上下线的感知

* 当服务可用时，需要自动把服务加到地址列表中，而服务不可用时，需要自动从列表中删除。
* 主要有两种实现方式：

## 通过客户端和服务器的连接感知

* 无论是消息发布者还是接收者都与注册查找中心的服务器维持一个长连接，可以通过长连接上的心跳来判断服务发布者是否还在线。
* 可能存在的问题：
  + 1)注册查找中心自身的负载很高时，可能会来不及处理心跳树，会以为心跳超时而判定服务不在线。
  + 2)如果服务发布者到注册查找中心的网络有问题，而服务发布者到服务使用者的网络没有问题，也会造成感知的问题。
* 解决办法是在注册查找中心的客户端上增加逻辑，当收到注册查找中心通知应用下线数据时，需要服务调用者进行验证才能接收这个通知。但是这个方法带来的是对每个服务提供者的一次额外验证。

## 通过对于发布数据中提供的地址接口进行连接的检查

* 通过外部的一个主动检查的方式去进行判断是一个补偿的方式，也就是当长连接的相关感知判断服务应用已经下线时，不直接认定这个服务已经下线，而是交给另一个独立的监控应用去验证这个服务是否已经不在了，方法一般是通过之前发布的地址、端口进行一下连接的验证，如果不能连接，则确认机器下线了。
* 不过这种方式同样存在一个问题，即进行检查确认的这个系统也可能与服务提供者之间存在问题，同样需要服务调用者进行最终确认才能解决。

# 数据分发

## 数据分发与消息订阅的区别

* 消息中间件的订阅、消息的接收和注册查找中心的订阅、消息的接收有什么区别？
  + 1)消息中间件需要保证数据不丢失，每条消息都应该送到相关的订阅者；注册查找中心只需要保证最新数据送到相关的订阅者，不需要保证每次的数据变化都能让最终订阅者感知。
  + 2)订阅者分组。在消息中间件中，同一个集群中的不同机器是分享所有消息的（Queue模型)，因为这个消息只要同一集群中的一台机器去处理就行了。而在注册查找中心中，需要把这个数据分发给所有的机器（Topic)。

## 提升数据分发性能

- 1)数据压缩  
- 2)全量和增量的选择

* 建议刚开始的实现中采用简单的方式，也就是传送全量数据，当全量的数据很大时，就需要考虑采用增量传送的方式来实现了。

# 针对服务化的特性支持

## 数据分组

* 分组group主要是为了隔离。分组本身就是一个命名空间，用来把相同dataId的内容分开。分组主要用在下面两种场景：
  + 1)根据环境进行区分：适用于线下的环境，我们在线下开发、测试的环境中，需要对不同的环境、项目进行隔离和区分，而分组就可以很好地支持这一功能。
  + 2)分优先级的隔离：适用于线上运行系统的隔离。可以把提供相同服务的提供者用组的概念分开，重要的服务使用者会有专门的组来提供服务，而其他的服务使用者可能会共用一个默认的组。
* 关于分组的方式，需要支持指定分组的API设置方式，以及根据IP地址自动归组的方式，根据IP地址自动进行归组可以带来更大的灵活性和运维的便利性。

## 提供自动感知以外的上下线开关

* 机器的上下线还需要通过指令而非机器状态来控制，通过指令直接从注册查找中心使机器下线。
* 之所在机器的状态外进行控制，主要有以下两点考虑：
  + 1)优雅地停止应用：应该先从服务列表中去掉这个机器，等待当时正在执行的服务结束，然后再停止应用。
  + 2)保持应用场景，用于排错：遇到服务的问题时，可以把出问题的服务留下一台进行故障定位和场景分析。这时需要把这台机器从服务列表中拿下来，以免有新的请求进来造成服务的失败。

## 维护管理路由规则

* 注册查找中心可以维护路由规则，不过这些数据与服务地址列表的特性不同。前者是持久化的，后者是非持久化的。

# 集群

* 集群会带来的问题：
  + 1)数据管理问题：集群下数据应该怎么维护保存
  + 2)连接管理问题：数据发布者与数据订阅者的连接应该怎么管理

## 数据统一管理方案

* 把数据聚合放在一个地方，这样负责管理连接的机器就是无状态的了。
* 整个结构分为三层，聚合数据这一层就是在管理数据；注册查找中心的机器是无状态的；对于数据发布者和订阅者来说，选择注册查找中心集群中的任何一台机器连接皆可。
* 对这个方案可以做一个修改，即把注册查找中心集群中的机器职责分开，就是把聚合数据的任务和推送数据的任务分到专门的机器上处理。
* 数据发布者和订阅者的连接是分开管理的，而集群中的应用分工更加明确。为了提升性能，在注册查找中心负责数据推送的机器上是可以对聚合数据做缓存的。

## 数据对等管理方案

* 将数据分散到注册查找中心的节点上，并且把自己节点管理的数据分发到其他节点上，从而保证每个结点都有整个集群的全部数据，并且这些节点的角色是对等的。
* 同样的，使用注册查找中心的数据发布者和订阅者只需要去连接注册查找中心中的任何一台机器就可以了，数据发布者只需要把数据发布给这一台机器，而数据订阅者只需要从这一台机器上进行订阅。
* 在注册查找中心内部，各个节点之间会进行数据的同步。如果注册查找中心A需要把数据同步给注册查找中心B，那么A就作为一个数据发布者把数据发布给B就可以了。B基本可以按照一个普通的数据发布者来处理A，差别是当B需要把自己的数据发布给其他节点时，从A收到的数据是不需要发布的，因为A自己会去发布。
* 这个方式可以复用现有的注册查找中心的客户端，不过也带来了同步效率的问题，可以通过批处理来提高效率。
* 同样可以把集群内部的节点进行职责划分，一种是进行数据分发的节点，另一种是进行数据聚合的节点。负责数据聚合的节点之间是没有连接的，负责数据分发的节点之间也是没有连接的，负责数据聚合和负责数据分发的节点是有连接的。

# Zookeeper ectd Consul

* Zookeeper（ZAB，Paxos)
* Zookeeper是这种类型的项目中历史最悠久的之一，它起源于Hadoop，帮助在Hadoop集群中维护各种组件。它非常成熟、可靠，被许多大公司（YouTube、eBay、雅虎等)使用。其数据存储的格式类似于文件系统，如果运行在一个服务器集群中，Zookeper将跨所有节点共享配置状态，每个集群选举一个领袖，客户端可以连接到任何一台服务器获取数据。
* Zookeeper的主要优势是其成熟、健壮以及丰富的特性，然而，它也有自己的缺点，其中采用Java开发以及复杂性是罪魁祸首。尽管Java在许多方面非常伟大，然后对于这种类型的工作还是太沉重了，Zookeeper使用Java以及相当数量的依赖使其对于资源竞争非常饥渴。因为上述的这些问题，Zookeeper变得非常复杂，维护它需要比我们期望从这种类型的应用程序中获得的收益更多的知识。这部分地是由于丰富的特性反而将其从优势转变为累赘。应用程序的特性功能越多，就会有越大的可能性不需要这些特性，因此，我们最终将会为这些不需要的特性付出复杂度方面的代价。
* etcd（Raft)
* etcd是一个采用HTTP协议的健/值对存储系统，它是一个分布式和功能层次配置系统，可用于构建服务发现系统。其很容易部署、安装和使用，提供了可靠的数据持久化特性。它是安全的并且文档也十分齐全。
* etcd比Zookeeper是比更好的选择，因为它很简单，然而，它需要搭配一些第三方工具才可以提供服务发现功能。
* Consul（Gossip)
* Consul是强一致性的数据存储，使用gossip形成动态集群。它提供分级键/值存储方式，不仅可以存储数据，而且可以用于注册器件事各种任务，从发送数据改变通知到运行健康检查和自定义命令，具体如何取决于它们的输出。
* 与Zookeeper和etcd不一样，Consul内嵌实现了服务发现系统，所以这样就不需要构建自己的系统或使用第三方系统。这一发现系统除了上述提到的特性之外，还包括节点健康检查和运行在其上的服务。
* Zookeeper和etcd只提供原始的键/值队存储，要求应用程序开发人员构建他们自己的系统提供服务发现功能。而Consul提供了一个内置的服务发现的框架。客户只需要注册服务并通过DNS或HTTP接口执行服务发现。其他两个工具需要一个亲手制作的解决方案或借助于第三方工具。
* Consul为多种数据中心提供了开箱即用的原生支持，其中的gossip系统不仅可以工作在同一集群内部的各个节点，而且还可以跨数据中心工作。

# 配置管理中心

* 开源的配置管理中心比较流行是的disconf。
* 在最初的时候，只有注册查找中心，除了管理服务地址列表外，路由规则、消息的订阅关系等也都在注册查找中心保存。但这些数据的特性并不相同，可以从数据是否需要持久以及数据是否需要聚合两个维度对数据进行分类。
* 持久指的是数据本身与发布者的生命周期无关的，典型的是持久订阅关系、路由规则、数据访问层的分库分表规则和数据库配置等；非持久指的是和发布者生命周期有关的，比如服务地址列表。此外，服务地址列表、订阅关系等数据是需要聚合的，而路由规则及一些设置项的内容则不需要聚合。
* 注册查找中心管理的是非持久的数据，配置管理中心管理的是持久数据，二者都可以支持聚合的数据。
* 对于配置管理中心来说，最为关心是稳定性和各种异常情况下的容灾策略，其次是性能和数据分发的延迟。配置管理中心存储的基本都是各个应用集群、中间件产品的关键管理配置信息，以及一些配置开关。
* 我们通过主备的持久存储来保存持久数据，一般采用关系型数据库。
* 配置管理中心集群是由多个节点组成，这些节点是对等地，都可以提供数据给应用端，也都可以接收数据的更新请求并更改数据库，这些节点之间互不依赖。
* 在配置管理中心的单个节点中，我们部署了Nginx和一个Web应用，Web应用主要负责完成相关的程序逻辑（数据库操作)以及根据IP等的分组操作。单机的本地文件则是为了容灾和提升性能，客户端进行数据获取时，最后都是从Nginx直接获取本地文件并把数据返回给请求段。
* 对于配置管理中心的使用分为两部分：
  + 1)提供给应用使用的客户端。主要是业务应用通过客户端去获取配置信息和数据，用于数据的读取。应用本身不去修改配置数据，而是根据配置来决定和更改自身应用的行为。
  + 2)为控制台或者控制脚本提供管理SDK
* 这个SDK包括了对数据的读写，提供管理SDK可以进行配置数据的更改。

# 客户端实现和容灾策略

* 客户端通过HTTP协议与配置管理中心进行通信。采用HTTP协议而不是私有协议可以更方便地支持多种语言的客户端，而且可以方便地进行测试和问题定位。可以采用HTTP长轮询的方式，数据分发的实时性比短轮询要好很多，和Socket长连接方式大体相同。HTTP长轮询是HTTP短轮询和Socket长连接的折中。
* 容灾：
  + 1)数据缓存：每次收到服务端更新后对数据的缓存。当服务端因忙而不能及时响应数据获取请求时，为应用提供一个可选的获取数据的方案。使用本地的缓存不能保证获取最新的数据，但是能保证获得比较新的数据。在一些场景下，应用需要的是获得相应的数据然后继续业务逻辑，是否是当下最细的数据可能不那么关键。
  + 2)数据快照：数据快照保存的是最近几次更新数据，数据是比缓存的数据旧一些，但是会保持最近的多个版本。数据快照用于服务端出现问题并且由于各种原因不能使用数据缓存时，例如缓存的最新的数据配置是一个有问题的配置，如果这是服务端不正常的话，就可以从更早几个版本的数据快照中进行恢复。

# 服务端实现和容灾策略

* 相比通过Web应用从数据库中获取数据，然后再把数据传给Nginx，通过Nginx返回本地文件的数据要快很多，能够很好地提高系统的吞吐量。除了作为静态化进行加速意外，本地文件处理还有一个重要的职责是进行数据库的容灾。有了本地文件，数据的读取就不再走数据库了，读取配置数据不需要数据库的参数。
* 数据库是共享的，文件是本地私有的。后者是缓存。
* 在服务端需要做的另一件事情是和数据库的数据同步：
  + 1)通过当前服务端更新数据库。由管理SDK的请求送到当前的服务端，服务端需要去更新数据库的数据，同时服务端也更新自身的本地文件，还可以通知其他机器去更新数据，不过只是传送一个更新数据的通知，而不是传送所有数据，并且这个通知也不是更新其他服务端数据的唯一方式。
  + 2)定时检查服务端的数据与数据库中数据的一致性。这是为了确保服务端本地文件和数据库内容的一致性，前面提到的如果数据更新通知不能送达其他服务端，那么其他服务端就要靠定时地检查来保证与数据库中数据的一致性。
* 此外，根据IP地址的分组处理也是服务端的Web应用需要处理好的逻辑。

# 数据库策略

* 数据库在设计时需要支持配置的版本管理，也就是随着配置内容的更改，老的版本是需要保留，这主要是为了方便进行配置变更的比对和回滚。而数据库本身需要主备进行数据的容灾考虑。

# Disconf

* http://disconf.readthedocs.io/zh\_CN/latest/
* Distributed Configuration Management Platform(分布式配置管理平台)
* 专注于各种「分布式系统配置管理」的「通用组件」和「通用平台」, 提供统一的「配置管理服务」。
* Disconf是百度开源出来的一款基于Zookeeper的分布式配置管理软件。目前很多公司都在使用，包括滴滴、百度、网易、顺丰等公司。通过简单的界面操作就可以动态修改配置属性，还是很方便的。使用Disconf后发现的一大好处是省却应用很多配置，而且配置可以自动load，实时生效。
* 通过简单的注解类方式 托管配置。托管后，本地不需要此配置文件，统一从配置中心服务获取。
* 当配置被更新后，注解类的数据自动同步。
* 基于Nginx、Tomcat、Zookeeper、Redis和MySQL。

# 分布式搜索引擎

# 爬虫问题

* 对于全网搜索来说，需要通过爬虫去获取被检索的网站的网页信息。在站内搜索中，我们同样需要可以发现、获取要被搜索的内容的系统。对于内部搜索来说，进入搜索系统中的数据的来源、格式及要求更新的频率都是已知的，这位我们根据数据变化来更新索引带来了很大的便利。
* 更新索引的方式一般有如下两种：
  + 1)定时从数据库拉取，称为增量dump。这要求数据库记录中有一个记录变更时间的字段，而这个字段需要有索引。增量dump开始前，需要进行全量dump构造初始化数据。增量的时间间隔一般会在分钟级，这会引起明显延时。
  + 2)通过数据变更的通知，及时通知搜索引擎构建索引，即时性会很好，不过带来的系统压力也比较大，适用于对实时性要求很高的场景。

# 倒排索引

* 正排索引：
* 可以通过文章找到这篇文章中的关键词，但是如果给定关键词，要找到该词出现在哪些文章中？
* 相对于正排索引，倒排索引是把原来作为值的内容拆分为索引的key，而原来用作索引的key则变成了value。搜索引擎比数据库的like查询更高效的原因也在于倒排索引。
* 如何确定建立倒排索引的关键字呢？取决于如何对要索引的内容进行分词。

# 查询预处理

* 查询预处理主要负责对用户输入的搜索内容进行分词及分词后的分析，包括一些同义词的替换及纠错等，这一部分是在使用搜索引擎前对于要搜索内容的梳理环节，而这部分的工作也会影响到最后搜索结果的质量。

# 相关度计算

* 当经过了查询分析器的处理后，查询会在搜索引擎上被执行，对于返回的结果，我们需要计算和搜索内容的相关度后展示给用户。相关度计算是在不指定按照某个字段排序的基础上对搜索结果的排序，排序的规则就是被搜索到的内容与要搜索的内容之间的相关度。
* 相关度的计算方式很多，例如有向量空间模型、概率模型等方法。

# 分布式数据计算

* 从实时性角度来讲，我们可以把计算分为实时计算和离线计算。

# 离线计算

* 离线计算是业务产生的数据离开生产环境后进行的计算，就是把业务数据从在线存储中移动到离线存储中，然后进行数据处理的过程。MapReduce模型是非常著名和常用的。

# MapReduce

* 在Map阶段，我们根据设定的规则把整体数据集映射给不同的Worker来处理，并且声称各自的处理结果。而在Reduce阶段，是对前面处理过的数据进行聚合，形成最后的结果。
* MapReduce模型让我们能够使用统一的模型和方式来使用集群中多机，降低了使用成本。
* Hadoop是MapReduce的一个来源实现，Hadoop使用HDFS进行数据存储， 而Spark提供了基于内存的集群计算的支持。Spark本身是为集群计算中特定类型的工作而设计，例如进行机器学习的算法训练，而基于内存的方式使得Spark的速度非常快。

# 在线计算

* 比较常见的方式是流式计算，Storm是使用的比较广泛的一个框架。

# 发布系统

* 发布系统应该完成的任务

# 分发应用

* 我们需要提供自动高效并且容易操作的机制来把经过测试的程序包分发到线上的应用中，这里我们一般会采用Web的操作方式，通过专用通道把应用程序包从线下环境传送到线上的发布服务器。

# 启动校验

* 当我们完成应用程序包的分发工作后，需要去停止当前应用上的程序，并完成新应用的启动。应用重新启动后，我们需要进行校验从而完成这台应用服务器上的应用发布。对应用的校验一般是由应用自身提供一个检测脚本或者页面，发布系统执行这个脚本或者访问页面后来判断返回的结果。
* 在停止应用时，如果采用暴力方式，就会影响当时正在执行的请求，所以需要优雅地关闭。但是如果持续有新请求进入的话，是很难优雅关闭应用的，所以需要控制不能有新请求进入。这就需要在负载均衡或者注册查找中心上将当前应用移除，之后再关闭应用（结束所有请求后关闭)，然后进行新应用的启动及检查，检查通过后，再把这个应用加入到负载均衡或者注册查找中心上，并对外提供应用。

# 灰度发布

* 会对新应用进行分批发布，逐步扩大新应用在整个集群中的比例直至最后全部完成。

# 产品改版Beta

* 面向最终用户的应用产品的改版会改变用户的习惯，对于这样的改变我们不会一刀切地直接推行，而会提供新旧应用的共存。应用本身会根据策略引流用户，对于发布系统来说，把新旧两个应用作为两个应用集群处理就行了。

# 应用监控系统

* 监控系统主要分为监视和控制两部分

# 监视

# 数据监视维度

* 监视的数据主要包括系统数据和应用自身的数据。系统数据比如CPU使用率，内存使用情况，交换分区使用情况，当前系统负载，IO情况等；应用自身的数据比如调用次数、成功率、响应时间、异常数量等维度的数据。

# 数据记录方式

* 系统自身的数据已经被记录到了本地磁盘上，应用的数据一般也是存放在应用自身的目录中。对于应用数据的记录，我们首先会用定时统计的方式记录一些量很大的信息。对于一个提供服务的应用，我们一般并不直接记录每次调用的信息，而是记录一段时间内的总调用次数、总响应时间这样的信息，而对于异常等信息，则每条都会予以记录。

# 数据采集方式

* 采集方式有应用服务器主动推送给监控中心以及等待监控中心拉取两种方式。

# 展现与告警

* 监控中心采集服务器收集的数据会集中存储，采用图表的方式可以提供Web页面的展示，并且根据设置的告警条件和接收人进行告警。

# 控制

* 控制是应用启动后在运行期对应用的行为改变。对于应用的运维，最低的要求是出现问题时 可以通过重启应用解决，但是我们还是需要更加精细化地控制应用，其实比较多的控制是进行降级和一些切换。
* 降级是我们遇到大量请求且不能扩容的情况时所进行的功能限制的行为，可能针对某个功能的所有使用者进行限制，也可能是根据不同使用者来进行限制。
* 切换更多的是当依赖的下层系统出现故障并且需要手工进行切换时的一个管理。这些控制一般是通过开关、参数设置来完成。

# 接口网关

# 定义

* 接口网关，顾名思义，是企业 IT 在系统边界上提供给外部访问内部接口服务的统一入口。这里的外部可以指客户端、浏览器或者第三方应用等，在这种情况下，接口网关可以有多种定位：
* 提供后端服务面向 Web App 或者 Mobile App 的 APIGateway
* 作为开放平台面向 Partner 的 OpenAPI
* 主要功能：请求路由，安全认证

# 请求路由

* 企业提供内外两网，在没有接口网关时，提供外部服务的应用需要部署在外网。随着服务的增多，部署在外网的应用越来越多，在服务的安全压力与维护成本增大的情况下，需要一个统一的接口网关“隔离”内外服务。企业提供的服务（无论内部服务还是外部服务)均部署在内网，而由部署在外网的网关接受请求，并路由到内网服务。在这种情况下，既有利于对外屏蔽企业内部服务部署细节，提供统一的服务访问地址，又便于管理与维护内外部服务接口，便于演进与重构服务。这是接口网关提供请求路由的作用。

# 安全认证

* 在没有接口网关时，企业对外服务直接由外部访问，身份验证与数据加解密等工作都需要每一个对外服务本身去处理，增加了服务本不该有的职责，并且增加了服务开发的难度与工作量。实际在大多数情况下，可以将身份验证与数据加解密等安全工作可以从服务抽离，统一由接口网关负责处理。接口网关作为入口，对外验证调用方的 IP，身份以及接口访问权限等，并且可以解密数据后再将请求路由到服务。这是接口网关提供安全认证的功能。

# Nginx+OpenResty

* OpenResty 是一个通过 Lua 扩展 Nginx 实现的可伸缩的 Web 平台。其核心是基于 Nginx 一个 C 模块将 Lua 语言嵌入到 Nginx 服务器中，对外提供一套完整的 Lua Web 的 API，并透明支持非阻塞 I/O，提供协程 —— “轻量级线程”、定时器等，从而极大地降低了高性能服务端的开发难度和开发周期。
* OpenResty 将两个极为优秀的组件 Nginx 与 Lua 进行糅合，一方面保留了 Nginx 高性能 web 服务特征，另一方面有提供 Lua 特性在极少损失性能情况下便于业务功能的开发。根据官网介绍，OpenResty 非常便于用来搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。

# 限流

* 在实际的系统架构中，用户请求可能会经过多级才会到达应用节点，比如：nginx-->gateway-->应用。如果条件允许，可以在尽量靠前的位置做限流设置，这样可以尽早的给用户反馈，也可以减少后续层级的资源浪费。不过毕竟在应用内增加限流配置的开发成本相对来说较低，并且可能会更灵活，所以需要根据团队实际情况而定了。nginx做限流设置可以使用Lua+Redis配合来实现；应用内限流可以使用RateLimiter来做。当然都可以通过封装来实现动态配置限流的功能，比如【ratelimiter-spring-boot-starter】

# 基于方法调用的限流

* 限流是对系统的出入流量进行控制，防止大流量出入，导致资源不足，系统不稳定。

## 限制瞬时并发数

- AtomicInteger atomic = new AtomicInteger(1)

* function(){
* try {
* if(atomic.incrementAndGet() > 限流数) {
* //熔断逻辑
* } else {
* //处理逻辑
* }
* } finally {
* atomic.decrementAndGet();
* }
* }

## 限制时间窗最大请求数

* 即一个时间窗口内的请求数，如想限制某个接口/服务每秒/每分钟/每天的请求数/调用量。如一些基础服务会被很多其他系统调用，比如商品详情页服务会调用基础商品服务调用，但是怕因为更新量比较大将基础服务打挂，这时我们要对每秒/每分钟的调用量进行限速；

LoadingCache<Long, AtomicLong> counter =  
 CacheBuilder.newBuilder()  
 .expireAfterWrite(2, TimeUnit.SECONDS)  
 .build(new CacheLoader<Long, AtomicLong>() {  
 @Override  
 public AtomicLong load(Long seconds) throws Exception {  
 return new AtomicLong(0);  
 }  
 });  
long limit = 1000;  
while(true) {  
 //得到当前秒  
 long currentSeconds = System.currentTimeMillis() / 1000;  
 if(counter.get(currentSeconds).incrementAndGet() > limit) {  
 System.out.println("限流了:" + currentSeconds);  
 continue;  
 }  
 //业务处理  
}

* 使用Guava的Cache来存储计数器，过期时间设置为2秒（保证1秒内的计数器是有的)，然后我们获取当前时间戳然后取秒数来作为KEY进行计数统计和限流，这种方式也是简单粗暴，刚才说的场景够用了。

## 令牌桶

* 假如用户配置的平均发送速率为r，则每隔1/r秒一个令牌被加入到桶中
* 假设桶中最多可以存放b个令牌。如果令牌到达时令牌桶已经满了，那么这个令牌会被丢弃
* 当流量以速率v进入，从桶中以速率v取令牌，拿到令牌的流量通过，拿不到令牌流量不通过，执行熔断逻辑
* 参考令牌桶的算法描述，一般思路是在RateLimiter-client放一个重复执行的线程，线程根据配置往令牌桶里添加令牌，这样的实现由如下缺点：
* 需要为每个令牌桶配置添加一个重复执行的线程
* 重复的间隔精度不够精确：线程需要每1/r秒向桶里添加一个令牌，当r >1000 时间线程执行的时间间隔根本没办法设置（从后面性能测试的变现来看RateLimiter-client 是可以承担 QPS > 5000 的请求速率)
* 基于上述的令牌桶算法
* 将添加令牌改成触发式的方式，取令牌时做添加令牌的动作
* 在取令牌的时候，通过计算上一次添加令牌和当前的时间差，计算出这段时间应该添加的令牌数，然后往桶里添加
* curr*mill*second = 当前毫秒数
* last*mill*second = 上一次添加令牌的毫秒数
* r = 添加令牌的速率
* reserve*permits = (curr*mill*second-last*mill\_second)/1000 \* r
* 添加完令牌之后再执行取令牌逻辑
* 单节点下GOOGLE GUAVA 提供的工具库中 RATELIMITER 类是非常好用的，底层也是基于令牌桶的思想。

# 基于IP的限流

* 使用Lua+Redis来限制IP的访问频率
* --查询redis中保存的ip的计数器
* ip*count, err = conn:get(BUSINESS.."-COUNT-"..ngx.var.remote*addr)
* if ip*count == ngx.null then --如果不存在，则将该IP存入redis，并将计数器设置为1、该KEY的超时时间为ip*time\_out
  + res, err = conn:set(BUSINESS.."-COUNT-"..ngx.var.remote\_addr, 1)
* res, err = conn:expire(BUSINESS.."-COUNT-"..ngx.var.remote*addr, ip*time\_out)
* else
* ip*count = ip*count + 1 --存在则将单位时间内的访问次数加1
* if ip*count >= ip*max*count then --如果超过单位时间限制的访问次数，则添加限制访问标识，限制时间为ip*block\_time
  + res, err = conn:set(BUSINESS.."-BLOCK-"..ngx.var.remote\_addr, 1)
* res, err = conn:expire(BUSINESS.."-BLOCK-"..ngx.var.remote*addr, ip*block\_time)
* else
* res, err = conn:set(BUSINESS.."-COUNT-"..ngx.var.remote*addr,ip*count)
* res, err = conn:expire(BUSINESS.."-COUNT-"..ngx.var.remote*addr, ip*time\_out)
* end
* end

# Kong

* Kong是一款基于Nginx\_Lua模块写的高可用，易扩展由Mashape公司开源的API Gateway项目。由于Kong是基于Nginx的，所以可以水平扩展多个Kong服务器，通过前置的负载均衡配置把请求均匀地分发到各个Server，来应对大批量的网络请求。
* Kong主要有三个组件：
* Kong Server ：基于nginx的服务器，用来接收API请求。
* Apache Cassandra/PostgreSQL ：用来存储操作数据。
* Kong dashboard：官方推荐UI管理工具，当然，也可以使用 restfull 方式 管理admin api。