知识回顾

2018年8月23日 9:06

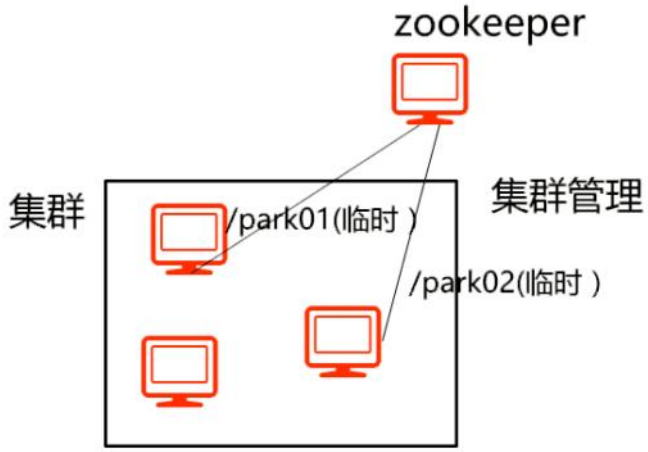
知识点

1.Zookeeper是一个分布式（集群环境）的协调服务框架

2.Zookeeper集群管理，实现思路：

每台客户端服务器在启动时连接zk服务器，并注册自己的临时节点，然后通过监听机制，监听节点被删除的事件。

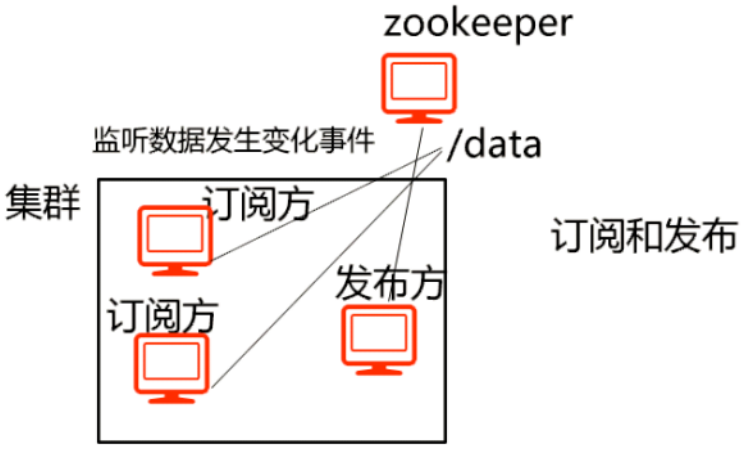
所以zookeeper的应用范围很广泛，只要是集群，都可以使用zookeeper。



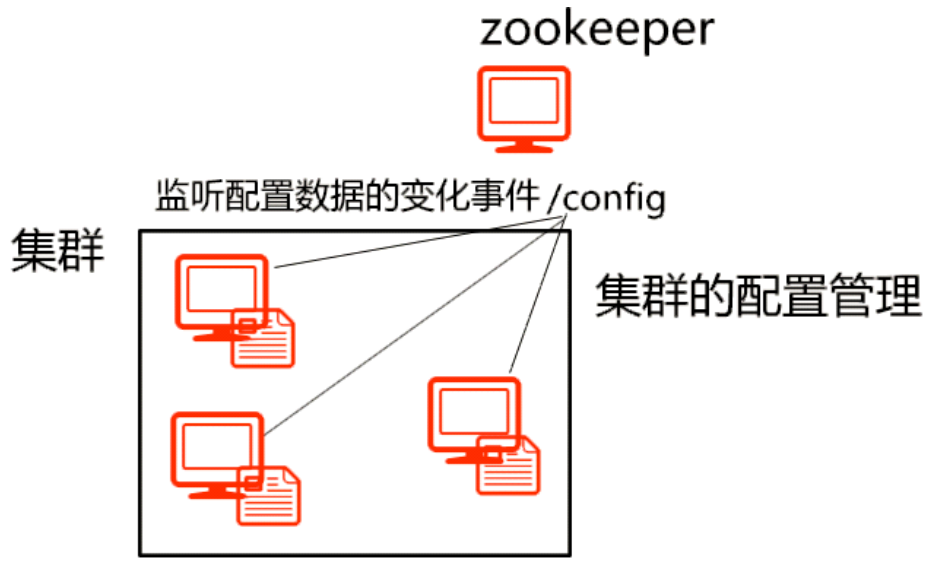
3.zookeeper可以做统一的命名服务。可以利用zk路径唯一性的来实现。

4.zookeeper实现数据的订阅和发布。

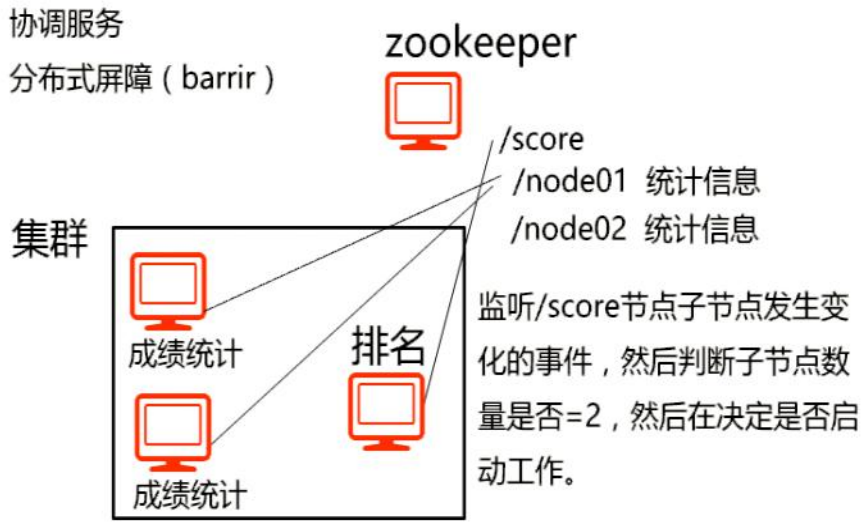
订阅方监听数据发布方节点数据产生变化事件。



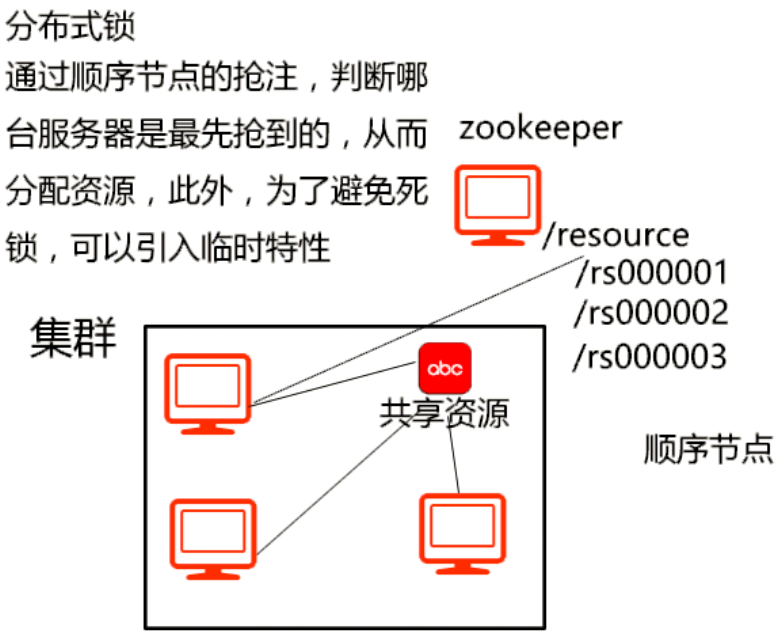
5.zookeeper实现集群统一的配置信息管理



6.zookeeper实现协调服务通知



7.zookeeper实现分布式锁



8.总结：zookeeper实现各种分布式服务的手段：各类型节点+监听机制

9.zookeeper单机模式安装，解压完之后，进入zk安装目录的conf目录，把zoo_sample.conf 重命名为 zoo.cfg，下一步，进入bin 目录，启动zk服务

10.zk单机模式用于练习或测试环境，不能用于生产环境，因为单点故障问题。

11.zk集群的搭建，需要配置zoo.cfg，两个必配：

①data.dir zk存储数据的本地目录

②

server.1=ip:2888:3888

server.2=ip:2888:3888

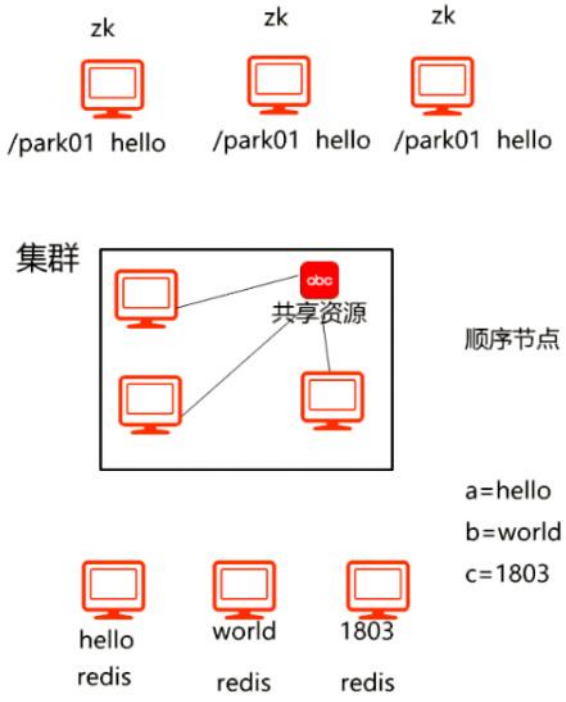
server.3=ip:2888:3888

数字是选举id，要求必须能够比较出大小。2888原子广播端口，3888选举端口。

配置完之后，在本地目录，创建myid文件，然后写入当前服务器的id号。

最后就是启动zk服务集群。

12.操作zk时，都是以znode路径来操作的，每个znode都可以存储数据，数据会在zk服务器的内存里存一份，供用户快速访问，基于这个特点，不能用zk存储海量数据，一是zk的应用就是存储海量数据的（是做协调服务的），二是基于内存，不能存储海量数据。所以，zk存储的数据是元数据信息，配置信息，服务器的节点数据。



13.选举机制。目的是通过一定的规则（算法）选出zk集群的leader。

pk原则：

①先比较每台zk服务的事务id，谁大谁当Leader

②如果事务id比较不出来，就比较选举id，谁大谁当Leader

③必须要满足过半性

14.zk中过半性的体现：

①选举过程的过半同意

②zk服务集群要保证正常工作，必须要满足过半性，称为过半存活。

所以，一般在工作中，zk集群数量一般都是奇数个

拓展：2PC算法

分布式下的数据一致性问题

对于一个将数据副本分布在不同分布式节点上的系统来说，如果对第一个节点的数据进行了更新操作并且更新成功后，却没有使得第二个节点上的数据得到相应的更新，于是在对第二个节点的数据进行读取操作时，获取的依然是老数据（或称为脏数据），这就是典型的分布式数据不一致情况。

为了解决分布式数据一致性问题，在长期的探索研究过程中，涌现出了一大批经典的一致性协议和算法，其中最著名的就是二阶段提交协议、三阶段提交协议和Paxos算法。

2PC

2PC，是Two-Phase Commit的缩写，即二阶段提交，是计算机网络尤其是在数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务处理过程中能够保持原子性和一致性而设计的一种算法。通常，二阶段提交协议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，**绝大部分的关系型数据库都是采用二阶段提交协议来完成分布式事务处理的**，利用该协议能够非常方便地完成所有分布式事务参与者的协调，统一决定事务的提交或回滚，从而能够有效地保证分布式数据一致性，因此二阶段提交协议被广泛地应用在许多分布式系统中。

提交过程

二阶段提交协议是将事务的提交过程分成了两个阶段来进行处理，其执行流程如下。

阶段一：提交事务请求+执行事务

1. 事务询问。

协调者向所有的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。

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。

由于上面讲述的内容在形式上近似是协调者组织各参与者对一次事务操作的投票表态过程，因此二阶段提交协议的阶段一也被称为“投票阶段”，即各参与者投票表明是否要继续执行接下去的事务提交操作。

阶段二：事务提交

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，包含以下两种可能。

执行事务提交

假如协调者从所有的参与者获得的反馈都是Yes响应，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消息后，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。

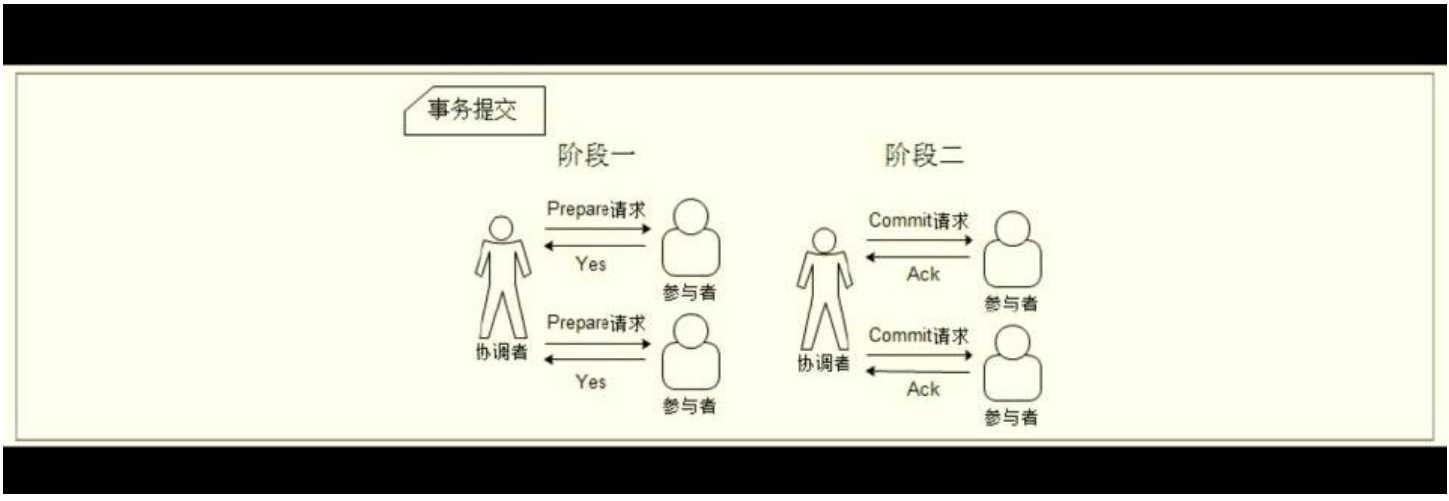
3. 反馈事务回滚结果。

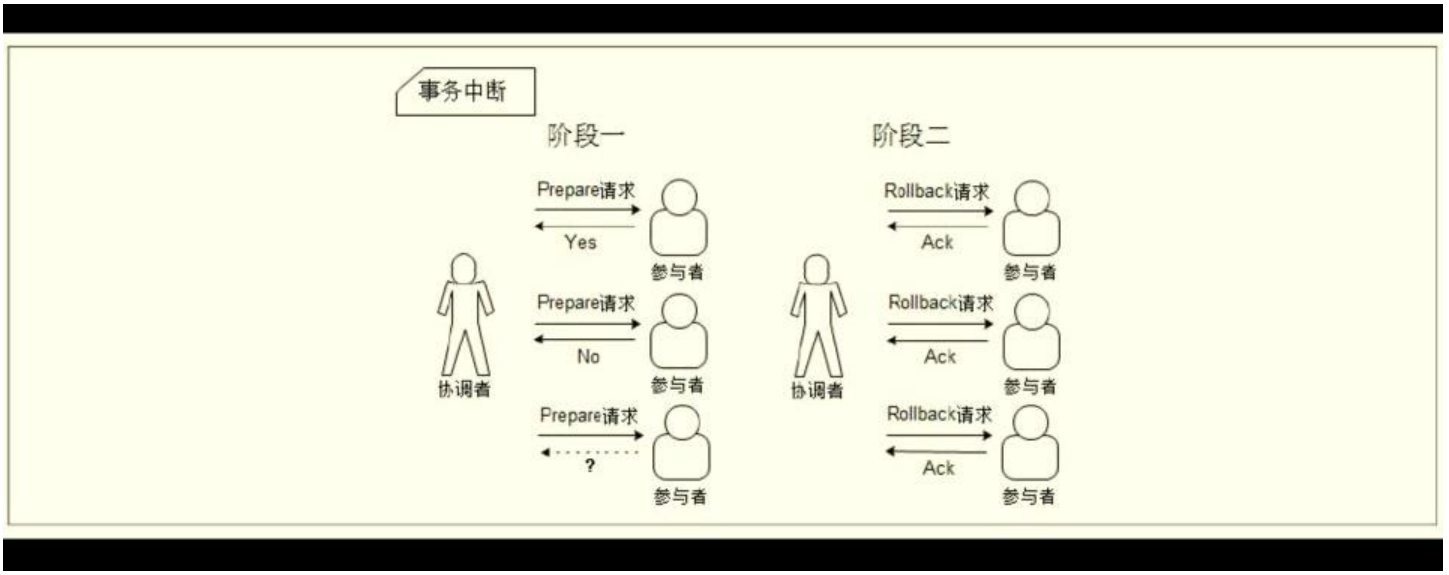
参与者在完成事务回滚之后，向协调者发送Ack消息。

4. 中断事务。

协调者接收到所有参与者反馈的Ack消息后，完成事务中断。

以上就是二阶段提交过程中，前后两个阶段分别进行的处理逻辑。简单地讲，二阶段提交将一个事务的处理过程分为了投票和执行两个阶段，其核心是对每个事务都采用先尝试后提交的处理方式，因此也可以将二阶段提交看作一个强一致性的算法，下图分别展示了二阶段提交过程中“事务提交”和“事务中断”两种场景下的交互流程。





优缺点

二阶段提交协议的优点：原理简单，实现方便。

二阶段提交协议的缺点：同步阻塞、单点问题、太过保守（只要有一个失败，则整个事务失败）。

同步阻塞

二阶段提交协议存在的最明显也是最大的一个问题就是同步阻塞，这会极大地限制分布式系统的性能。在二阶段提交的执行过程中，所有参与该事务操作的逻辑都处于阻塞状态，也就是说，**各个参与者在等待其他参与者响应的过程中，将无法进行其他任何操作**。所以性能较低

单点问题

在上面的讲解过程中，相信读者可以看出，**协调者**的角色在整个二阶段提交协议中起到了非常重要的作用。**一旦协调者出现问题**，那么整个二阶段提交流程将无法运转

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。换句话说，二阶段提交协议没有设计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。**

ZAB协议

2018年8月22日 15:21

概述

ZAB（Zookeeper Atomic Broadcast）协议是为分布式协调服务ZooKeeper专门设计的一种**支持崩溃恢复的原子广播协议**,它是一种特别为ZooKeeper设计的崩溃可恢复的原子消息广播算法。这个算法是一种类2PC算法，在2PC基础上做的改进

协议介绍

ZAB协议包括两种基本的模式，分别是：

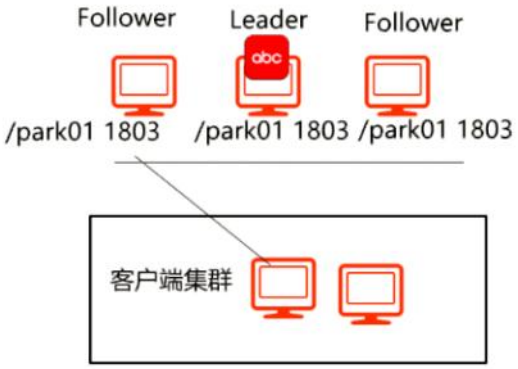
1）消息原子广播（保证数据一致性）

2）崩溃恢复（解决2pc算法的单点问题）

消息原子广播

在ZooKeeper中，**主要依赖ZAB协议来实现分布式数据一致性**，基于该协议，ZooKeeper实现了一种主备模式的系统架构来保持集群中各副本之间数据的一致性，实现分布式数据一致性的这一过程称为消息广播（原子广播）。

具体的，**ZooKeeper使用一个单一的主进程（Leader服务器）来接收并处理客户端的所有事务请求**，并采用ZAB的原子广播协议，将服务器数据的状态变更以事务Proposal的形式广播到所有的副本进程（Follower或Observer)上去。即：所有事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器被称为Leader服务器，而余下的其他服务器则成为Follower服务器或observer。



1.当客户端发现读请求是，任何一台都可以提供读服务，
2.当客户端发现写请求（事务），如果Followr收到事务请求，会将事务转交给Leader来处理

Leader服务器负责将一个客户端事务请求转换成一个事务Proposal（提议），并将该Proposal分发给集群中所有的Follower服务器。之后Leader服务器需要等待所有Follower服务器的反馈，一旦超过半数的服务器（Follower+Leader服务器）本身进行了正确的反馈后，那么Leader就会再次向所有的Follower服务器分发Commit消息，要求其进行提交。较比与2PC算法，ZAB引入了过半性思想。

ZAB协议的消息广播过程使用的是一个原子广播协议，类似于一个二阶段提交过程。针对客户端的事务请求，Leader服务器会为其生成对应的事务Proposal，并将其发送给集群中其余所有的机器，然后再分别收集各自的选票，最后进行事务提交。

需要明确的是，ZAB协议中涉及的二阶段提交过程则与2PC不同。在ZAB协议的二阶段提交过程中，移除了中断逻辑，中断逻辑移除意味着我们可以在过半的Follower服务器已经反馈Ack之后就开始提交事务Proposal了，而不需要等待集群中所有的Follower服务器都反馈响应。

当然，在这种简化了的二阶段提交模型下，是无法处理Leader服务器崩溃退出而带来的数据不一致问题的，因此在ZAB协议中添加了另一个模式，即采用崩溃恢复模式来解决这个问题。另外，整个消息广播协议是基于具有FIFO特性的TCP协议来进行网络通信的，因此能够很容易地保证消息广播过程中消息接收与发送的顺序性。

在整个消息广播过程中，Leader服务器会为每个事务请求生成对应的Proposal来进行广播，并且在广播事务Proposal之前，Leader服务器会首先为这个事务Proposal分配一个全局单调递增的唯一ID，我们称之为事务ID（即ZXID）。由于ZAB协议需要保证每一个消息严格的因果关系，因此必须将每一个事务Proposal按照其ZXID的先后顺序来进行排序与处理。

具体的，在消息广播过程中，Leader服务器会为每一个Follower服务器都各自分配一个单独的队列，然后将需要广播的事务Proposal依次放入这些队列中去，并且根据FIFO策略进行消息发送。每一个Follower服务器在接收到这个事务Proposal之后，都会首先将其以事务日志的形式写入到本地磁盘中去，并且在成功写入后反馈给Leader服务器一个Ack响应。当Leader服务器接收到超过半数Follower的Ack响应后，就会广播一个Commit消息给所有的Follower服务器以通知其进行事务提交，同时Leader自身也会完成对事务的提交，而每一个Follower服务器在接收到Commit消息后，也会完成对事务的提交。

崩溃恢复

当整个服务框架在启动过程中，或是当Leader服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB协议就会进入恢复模式并选举产生新的Leader服务器。

当选举产生了新的Leader服务器，同时集群中已经有过半的机器与该Leader服务器完成了状态同步之后，ZAB协议就会退出恢复模式。其中，所谓的状态同步是指数据同步，用来保证集群中存在过半的机器能够和Leader服务器的数据状态保持一致。
当集群中已经有过半的Follower服务器完成了和Leader服务器的状态同步，那么整个服务框架就可以进入消息广播模式了。

当一台同样遵守ZAB协议的服务器启动后加入到集群中时，如果此时集群中已经存在一个Leader服务器在负责进行消息广播，那么新加入的服务器就会自觉地进入数据恢复模式：找到Leader所在的服务器，

并与其进行数据同步，然后一起参与到消息广播流程中去。

ZooKeeper设计成只允许唯一的一个Leader服务器来进行事务请求的处理。Leader服务器在接收到客户端的事务请求后，会生成对应的事务提案并发起一轮广播协议；而如果集群中的其他机器接收到客户端的事务请求，那么这些非Leader服务器会首先将这个事务请求转发给Leader服务器。

当Leader服务器出现崩溃退出或机器重启，亦或是集群中已经不存在过半的服务器与该Leader服务器保持正常通信时，那么在重新开始新一轮的原子广播事务操作之前，所有进程首先会使用崩溃恢复协议来使彼此达到一个一致的状态，于是整个ZAB流程就会从消息广播模式进入到崩溃恢复模式。

一个机器要成为新的Leader，必须获得过半进程的支持，同时由于每个进程都有可能会崩溃，因此，在ZAB协议运行过程中，前后会出现多个Leader，并且每个进程也有可能会多次成为Leader。进入崩溃恢复模式后，只要集群中存在过半的服务器能够彼此进行正常通信，那么就可以产生一个新的Leader并再次进入消息广播模式。举个例子来说，一个由3台机器组成的ZAB服务，通常由1个Leader、2个Follower服务器组成。某一个时刻，假如其中一个Follower服务器挂了，整个ZAB集群是不会中断服务的，这是因为Leader服务器依然能够获得过半机器（包括Leader自己）的支持。

Zookeeper观察者

观察者

虽然客户端直接连接到投票选举的Zookeeper成员执行良好，但这个架构很难扩展到大量的客户端。问题就是因为我么添加了更多的投票成员，**写入性能下降**。这是由于这样的事实：一个写入操作要求共识协议**至少是整体的一半**，因此投票的成本随着投票者越多会显著增加。

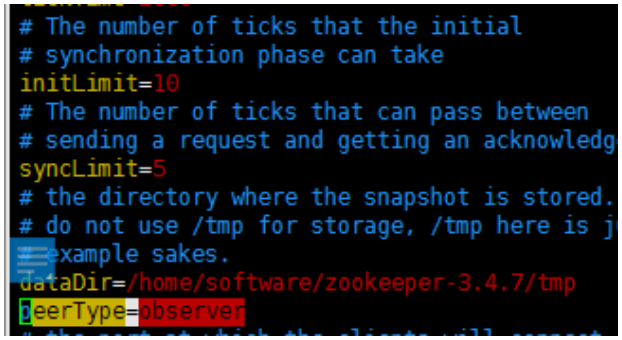
我们引入了一个新的Zookeeper节点类型叫做Observer（观察者），它帮助处理这个问题并进一步完善了Zookeeper的可扩展性。**观察者不参与投票，它只监听投票的结果**，不是导致了他们的共识协议。除了这个简单的区别，观察者精确的和追随者一样运行 - 客户端可能链接他们并发送读取和写入请求。**观察者像追随者一样转发这些请求到领导者，而他们只是简单的等待监听投票的结果。**正因为如此，我们可以尽可能多的增加观察者的数量，而不影响投票的性能。

观察者还有其他优势。因为他们不投票，他们不是Zookeeper整体的主要组件。**因此他们可以故障，或者从集群断开连接，而不影响Zookeeper服务的可用性。**对用户的好处是观察者可以连接到比追随者更不可靠的网络。事实上，观察者可以用于从其他数据中心和Zookeeper服务通信。观察者的客户端会看到快速的读取，因为所有的读取都在本地，并且写入导致最小的网络开销，因为投票协议所需的消息数量更小。

怎么使用观察者

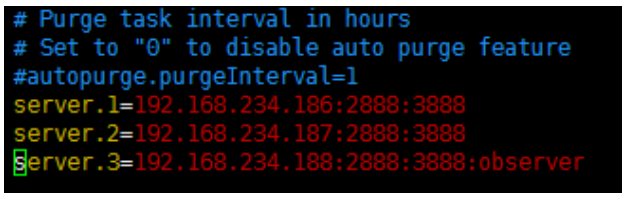
使用观察者设置Zookeeper全员非常简单，只需要在原来的配置文件上改两个地方。第一，在要设置的那个节点的配置文件设置为观察者，必须放置这一行：

```
peerType=observer
```



这一行告诉Zookeeper的服务是一个观察者。第二，在每个服务配置文件里，必须在观察者定义行添加:observer。例如：

server.1:localhost:2181:3181:observer



这个告诉其他服务server.1是一个观察者，并且他们不需要期望他选举。这是你在Zookeeper集群中添加观察者需要的所有配置。现在你可以连接到它好像是一个普通的追随者。尝试一下，通过运行：

bin/zkCli.sh -server localhost:2181

这里的localhost:2181是每个配置文件里指定的观察者的hostname和端口号。你应该查看一个命令行提示，通过类似于ls的操作查询Zookeeper服务。

Zookeeper配置详解

参数名	说明
clientPort	客户端连接server的端口，即对外服务端口，一般设置为2181吧。
dataDir	存储快照文件snapshot的目录。默认情况下，事务日志也会存储在这里。 follower和Leader服务节点都会有自己的事务日志。 ZK会在特定条件下会触发一次快照（snapshot），将当前服务节点的状态以快照文件的形式dump到磁盘上去，即snapshot文件。此外，每生成一次快照文件，就会生成一个对应的事务日志文件 快照数据文件名为：snapshot.x，而事务日志文件对应为：log.x。
dataLogDir	事务日志输出目录。 正常运行过程中，针对所有事务操作，在返回客户端“事务成功”的响应前，ZK会确保已经将本次事务操作的事务日志写到磁盘上，只有这样，事务才会生效。
tickTime	ZK中的一个时间单元。ZK中所有时间都是以这个时间单元为基础，进行整数倍配置的。 比如客户端的超时连接，是有一个上下限。下限：2*ticktime=4000 上限：20*ticktime=40000 4000~40000,如果API指定的时间不在此范围内，则用对应的下限或上限时间。
initLimit	Follower在启动过程中，会从Leader同步所有最新数据，然后确定自己能够对外服务的起始状态。Leader允许F在 **initLimit** 时间内完成这个工作。通常情况下，我们不用太在意这个参数的设置。如果ZK集群的数据量确实很大了，F在启动的时候，从Leader上同步数据的时间也会相应变长，因此在这种情况下，有必要适当调大这个参数了。 **默认是：10*ticktime**
syncLimit	在运行过程中，Leader负责与ZK集群中所有机器进行通信，例如通过一些心跳检测机制，来检测机器的存活状态。如果L发出心跳包在syncLimit之后，还没有从F那里收到响应，那么就认为这个F已经不在线了。 **默认是：5*ticktime**
minSessionTimeout maxSessionTimeout	Session超时时间限制，如果客户端设置的超时时间不在这个范围，那么会被强制设置为最大或最小时间。默认的Session超时时间是在2 * **tickTime ~ 20 * tickTime 这个范围**
snapCount	每进行snapCount次事务日志输出后，触发一次快照(snapshot), 此时，ZK会生成一个snapshot.*文件，同时创建一个新的事务日志文件log.*。默认是100000。这是一种情况 此外，在产生新Leader时，也会生成新的快照文件，（同时会生成对应的事务文件）

	此参数，在实际工作可以适当减少，比如5000次或1万次。
autopurge.purgeInterval	3.4.0及之后版本，ZK提供了自动清理事务日志和快照文件的功能，这个参数指定了清理频率，单位是小时，需要配置一个1或更大的整数，默认是0，表示不开启自动清理功能。 此参数，如果要配置，自动清理周期一般是一周或一个月
server.x=[hostname]:nnnnn[:nnnnn]	这里的x是一个数字，与myid文件中的id是一致的。右边可以配置两个端口，第一个端口用于F和L之间的数据同步和其它通信，第二个端口用于Leader选举过程中投票通信。
jute.maxbuffer	控制每个znode节点都可以存储数据的大小。每个节点最大数据量，默认是1M。
globalOutstandingLimit	最大请求堆积数。默认是1000。ZK运行的时候， 尽管server已经没有空闲来处理更多的客户端请求了，但是还是允许客户端将请求提交到服务器上来，以提高吞吐性能。当然，为了防止Server内存溢出，这个请求堆积数还是需要限制下的。 在工作中，如果想提高zk的并发容忍能力，可以将此参数调大。具体的调节 要取决于服务器的内存。配成20000~50000
preAllocSize	预先开辟磁盘空间，用于后续写入事务日志。默认是64M，每个事务日志大小就是64M。
electionAlg	默认为3，即基于TCP fast paxos election 选举算法。zookeeper的ZAB协议是类2PC算法，ZAB算法改进的原型算法是Paxos算法。 在3.4版本后，1 2对应的选举算法都是UDP，已弃用，所以此项配置不要更改。
leaderServers	默认情况下，Leader是会接受客户端连接，并提供正常的读写服务。但是，如果你想让Leader专注于集群中机器的协调，那么可以将这个参数设置为no，这样一来，**会提高整个zk集群性能。**
maxClientCnxns=60	控制的每一台zk服务器能处理的客户端并发请求数。 实际工作中，都会将此参数调大。200~350

Zookeeper集群指令

2015年12月1日 18:19

nc安装使用

1.上传nc安装包

2.执行：rpm -ivh nc-1.84-24.el6.x86_64.rpm

nc ：netcat 是一个小型的网络通信工具，可以发起TCP请求

Zookeeper集群命令

可以通过Linux nc 工具来查看Zookeeper集群服务状态（掌握3个即可）

netcat（网络工具）

1. 执行：echo stat|nc 127.0.0.1 2181

查看哪个节点（想看哪个节点，就写那个节点的ip即可）被选择作为follower或者leader

Clients:

/127.0.0.1:40743[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0

Received: 2

Sent: 1

Connections: 1

Outstanding: 0

Zxid: 0x500000002

Mode: follower

Node count: 8

2. 执行：echo ruok|nc 127.0.0.1 2181

测试是否启动了该Server，若回复imok表示已经启动。

3. 执行：echo conf | nc 127.0.0.1 2181

输出相关服务配置的详细信息。

clientPort=2181

dataDir=/home/software/zookeeper-3.4.7/tmp/version-2

dataLogDir=/home/software/zookeeper-3.4.7/tmp/version-2

tickTime=2000

maxClientCnxns=60

minSessionTimeout=4000

maxSessionTimeout=40000

serverId=1
initLimit=10
syncLimit=5
electionAlg=3
electionPort=3888
quorumPort=2888
peerType=0

4. echo kill | nc 127.0.0.1 2181 ,关掉server

Zookeeper特性总结

数据一致性

client不论连接到哪个Zookeeper，展示给它都是同一个视图，即查询的数据都是一样的。这是zookeeper最重要的性能。

原子性

对于事务决议的更新，只能是成功或者失败两种可能，没有中间状态。**要么都更新成功，要么都不更新。**即，要么整个集群中所有机器都成功应用了某一事务，要么都没有应用，一定不会出现集群中部分机器应用了改事务，另外一部分没有应用的情况。

可靠性

一旦zk服务端成功的应用了一个事务，并完成对客户端的响应，那么该事务所引起的服务端状态变更将会一直保留下来，除非有另一个事务又对其进行了改变。

实时性

Zookeeper保证客户端将在非常短的时间间隔范围内获得服务器的更新信息，或者服务器失效的信息，或者指定监听事件的变化信息。（前提条件是：网络状况良好）

顺序性

如果在一台服务器上消息a在消息b前发布，则在所有Server上消息a都将在消息b前被发布。底层是通过递增的事务id（zxid）来实现的。

过半性

zookeeper集群必须有半数以上的机器存活才能正常工作。因为只有满足过半数，才能满足选举机制选出Leader。因为只有过半，在做事务决议时，事务才能更新。

所以一般来说，zookeeper集群的数量最好是奇数个。

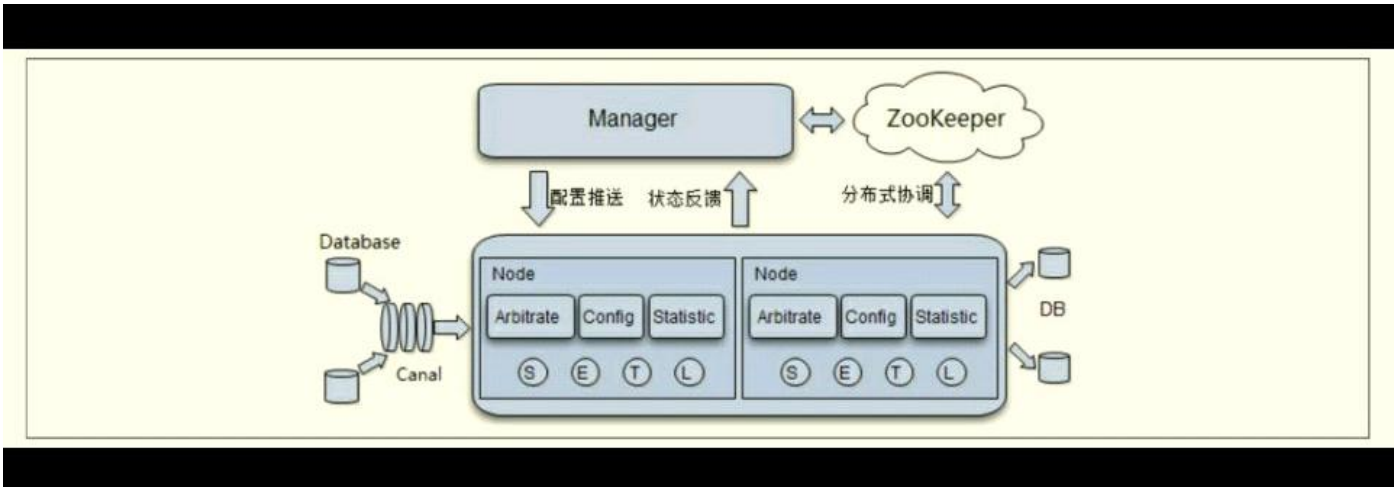
课外阅读：阿里—otter

Otter是阿里巴巴于2013年8月正式开源的一个由纯Java语言编写的分布式数据库同步系统，主要用于异地双A机房的数据库数据同步，致力于解决长距离机房的数据同步及双A机房架构下的数据一致性问题。目前项目主页地址为

https://github.com/alibaba/otter

由项目主要负责人，同时也是资深的开源爱好者agapple持续维护。

项目名Otter取自“水獭”的英文单词，寓意数据搬运工，是一个定位为基于数据库增量日志解析，在本机房或异地机房的MySQL/Oracle数据库之间进行准实时同步的分布式数据库同步系统。Otter的第一个版本可以追溯到2004年，初衷是为了解决阿里巴巴中美机房之间的数据同步问题，从4.0版本开始开源，并逐渐演变成一个通用的分布式数据库同步系统。其基本架构如下图所示。



Otter基本架构示意图

从上图中，我们可以看出，在Otter中也是使用ZooKeeper来实现一些与分布式协调相关的功能，下面我们将从Otter的分布式SEDA模型调度和面向全球机房服务的ZooKeeper集群搭建两方面来讲解Otter中的ZooKeeper使用。

分布式SEDA模型调度

为了更好地提高整个系统的扩展性和灵活性，在Otter中将整个数据同步流程抽象为类似于ETL的处理模型，具体分为四个阶段（Stage）。

Select：数据接入。

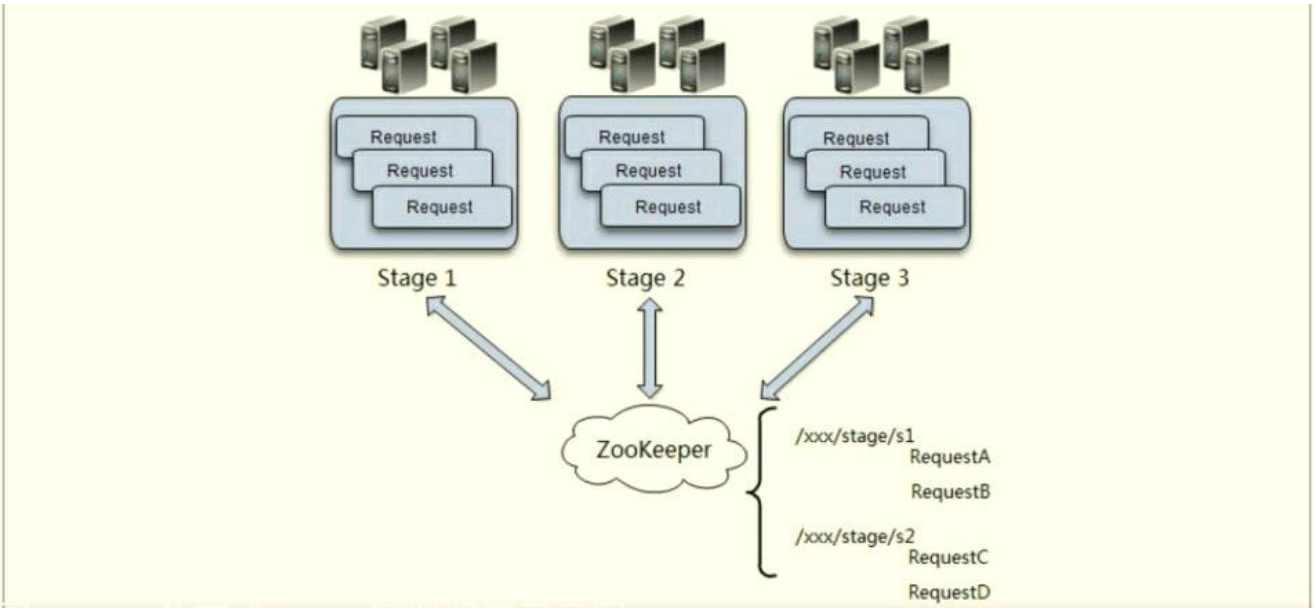
Extract：数据提取。

Transform：数据转换。

Load：数据载入。

其中Select阶段是为了解决数据来源的差异性，比如可以接入来自Canal的增量数据，也可以接入其他系统的数据源。Extract/Transform/Load阶段则类似于数据仓库的ETL模型，具体可分为数据Join、数据转化和数据Load等过程

同时，为了保证系统的高可用性，SEDA的每个阶段都会有多个节点进行协同处理。如下图所示是该SEDA模型的示意图。



分布式SEDA模型调度示意图

Stage管理

Stage管理主要就是维护一组工作线程，在接收到Schedule的Event任务信号后，分配一个工作线程来进行任务处理，并在任务处理完成后，反馈信息到Schedule。

Schedule调度

Schedule调度主要是指基于ZooKeeper来管理Stage之间的任务消息传递，其具体实现逻辑如下。

1. 创建节点。

Otter首先会为每个Stage在ZooKeeper上创建一个节点，例如/seda/stage/s1，其中s1即为该Stage的名称，每个任务事件都会对应于该节点下的一个子节点，例如/seda/stage/s1/RequestA。

2. 任务分配。

当s1的上一级Stage完成RequestA任务后，就会通知“Schedule调度器”其已完成了该请求。根据预先定义的Stage流程，Schedule调度器便会在Stage s1的目录下创建一个RequestA的子节点，告知s1有一个新的请求需要其处理——以此完成一次任务的分配。

3. 任务通知。

每个Stage都会有一个Schedule监听线程，利用ZooKeeper的Watcher机制来关注ZooKeeper中对应Stage节点的子节点变化，比如关注s1就是关注/seda/stage/s1的子节点的变化情况。此时，如果步骤2中调度器在s1的节点下创建了一个RequestA，那么ZooKeeper就会通过Watcher机制通知到该Schedule线程，然后Schedule就会通知Stage进行任务处理——以此完成一次任务的通知。

4. 任务完成。

当s1完成了RequestA任务后，会删除s1目录下的RequestA任务，代表处理完成，然后继续步骤2，分配下一个Stage的任务。

在上面的步骤3中，还有一个需要注意的细节是，在真正的生产环境部署中，往往都会由多台机器共同组成一个Stage来处理Request，因此就涉及多个机器节点之间的分布式协调。

如果s1有多个节点协同处理，每个节点都会有该Stage的一个Shedule线程，其在s1目录变化时都会收到通知。在这种情况下，往往可以采取抢占式的模式，尝试在RequestA目录下创建一个lock节点，谁创建成功就可以代表当前谁抢到了任务，而没抢到该任务的节点，便会关注该lock节点的变化（因为一旦该lock节点消失，那么代表当前抢到任务的节点可能出现了异常退

出，没有完成任务），然后继续抢占模型。

中美跨机房ZooKeeper集群的部署

由于Otter主要用于异地双A机房的数据库同步，致力于解决长距离机房的数据同步及双A机房架构下的数据一致性问题，因此其本身就有面向中美机房服务的需求，也就会有每个机房都要对ZooKeeper进行读写操作的需求。于是，希望可以部署一个面向全球机房服务的ZooKeeper集群，保证读写数据一致性。

这里就需要使用ZooKeeper的Observer功能了。从3.3.0版本开始，ZooKeeper新增了Observer模式，该角色提供只读服务，且不参与事务请求的投票，主要用来提升整个ZooKeeper集群对非事务请求的处理能力。

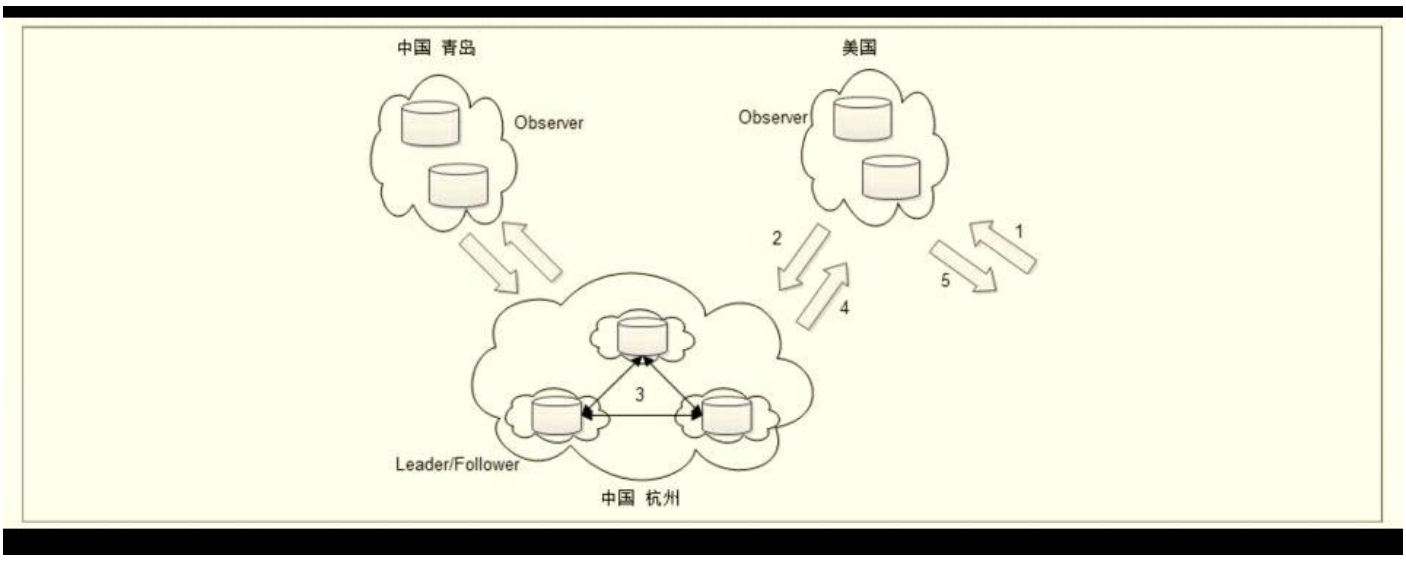
因此，借助ZooKeeper的Observer特性，Otter将ZooKeeper集群进行了三地部署。

杭州机房部署Leader/Follower集群，为了保障系统高可用，可以部署3个机房。每个机房的部署实例可为1/1/1或者3/2/2的模式。

美国机房部署Observer集群，为了保证系统高可用，可以部署2个机房，每个机房的部署实例可以为1/1。

青岛机房部署Observer集群。

下图所示是ZooKeeper集群三地部署示意图。



当美国机房的客户端发起一个非事务请求时，就直接从部署在美国机房的Observer ZooKeeper读取数据即可，这将大大减少中美机房之间网络延迟对ZooKeeper操作的影响。而如果是事务请求，那么美国机房的Observer就会将该事务请求转发到杭州机房的Leader/Follower集群上进行投票处理，然后再通知美国机房的Observer，最后再由美国机房的Observer负责响应客户端。

上面这个部署结构，不仅大大提升了ZooKeeper集群对美国机房客户端的非事务请求处理能力，同时，由于对事务请求的投票处理都是在杭州机房内部完成，因此也大大提升了集群对事务请求的处理能力。

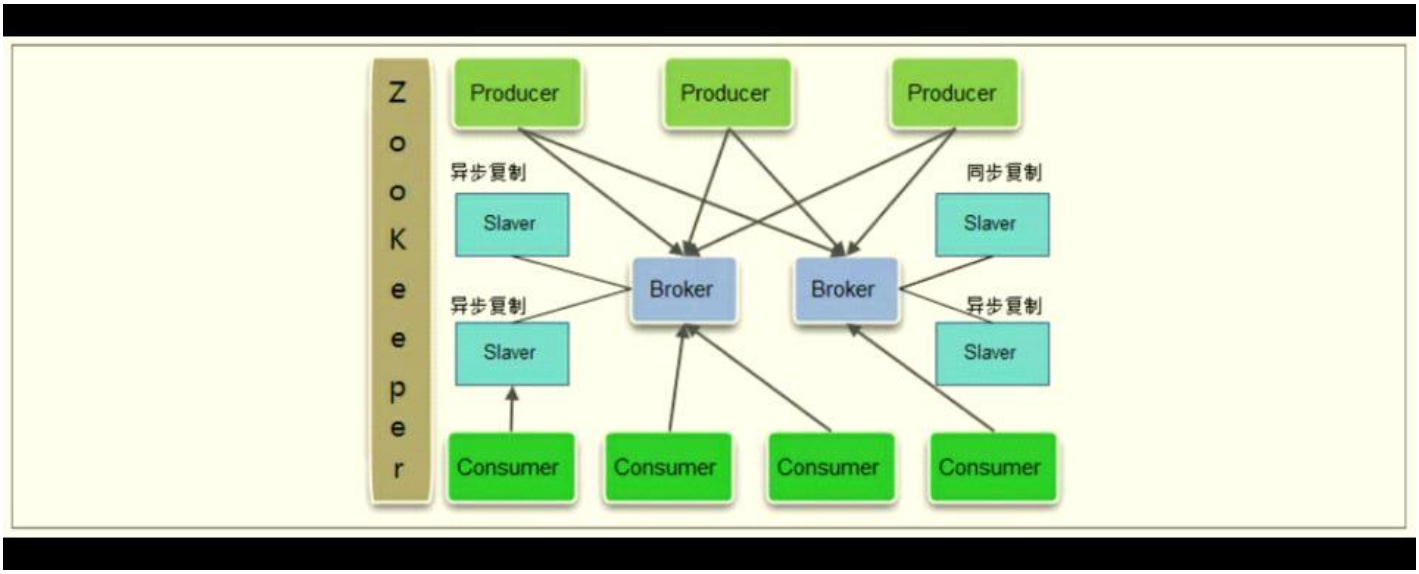
课外阅读：阿里—Metamorphosis

Metamorphosis是阿里巴巴中间件团队的killme2008和wq163于2012年3月开源的一个Java消息中间件，目前项目主页地址为：

https://github.com/killme2008/Metamorphosis

由开源爱好者及项目的创始人killme2008和wq163持续维护。关于消息中间件，相信读者应该都听说过JMS规范，以及一些典型的开源实现，如ActiveMQ和HornetQ等，Metamorphosis也是其中之一。

Metamorphosis是一个高性能、高可用、可扩展的分布式消息中间件，其思路起源于LinkedIn的Kafka，但并不是Kafka的一个简单复制。Metamorphosis具有消息存储顺序写、吞吐量大和支持本地XA事务等特性，适用于大吞吐量、顺序消息、消息广播和日志数据传输等分布式应用场景，目前在淘宝和支付宝都有着广泛的应用，其系统整体部署结构如下图所示。



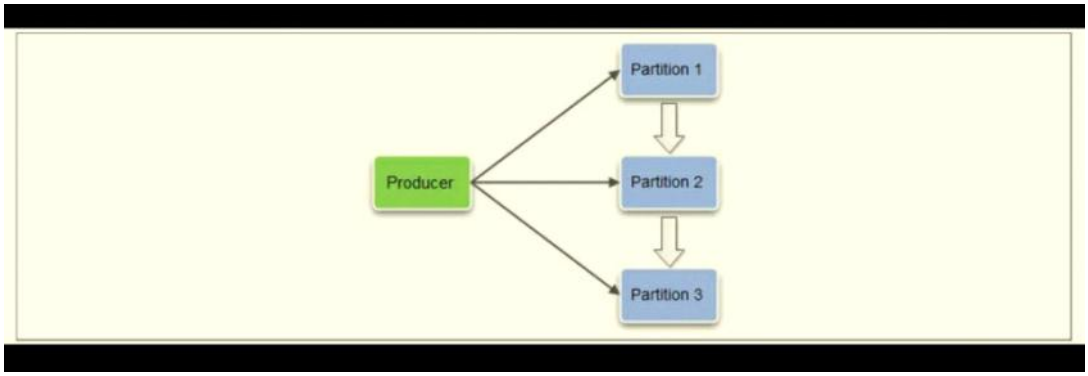
Metamorphosis整体部署结构

和传统的消息中间件采用推（Push）模型所不同的是，Metamorphosis是基于拉（Pull）模型构建的，由消费者主动从Metamorphosis服务器拉取数据并解析成消息来进行消费，同时大量依赖ZooKeeper来实现负载均衡和Offset的存储。

生产者的负载均衡

和Kafka系统一样，Metamorphosis假定生产者、Broker和消费者都是分布式的集群系统。生产者可以是一个集群，多台机器上的生产者可以向相同的Topic发送消息。而服务器Broker通常也是一个集群，多台Broker组成一个集群对外提供一系列的Topic消息服务，生产者按照一定的路由规则向集群里某台Broker发送消息，消费者按照一定的路由规则拉取某台Broker上的消息。每个Broker都可以配置一个Topic的多个分区，但是在生产者看来，会将一个Topic在所有Broker上的所有分区组成一个完整的分区列表来使用。

在创建生产者的时候，客户端会从ZooKeeper上获取已经配置好的Topic对应的Broker和分区列表，生产者在发送消息的时候必须选择一台Broker上的一个分区来发送消息，默认的策略是一个轮询的路由规则，如下图所示。



Metamorphosis生产者消息分区发送示意图

生产者在通过ZooKeeper获取分区列表之后，会按照Broker Id和Partition的顺序排列组织成一个有序的分区列表，发送的时候按照从头到尾循环往复的方式选择一个分区来发送消息。考虑到我们的Broker服务器软硬件配置基本一致，因此默认的轮询策略已然足够。

在Broker因为重启或者故障等因素无法提供服务时，Producer能够通过ZooKeeper感知到这个变化，同时将失效的分区从列表中移除，从而做到Fail Over。需要注意的是，因为从故障到生产者感知到这个变化有一定的延迟，因此可能在那一瞬间会有部分的消息发送失败。

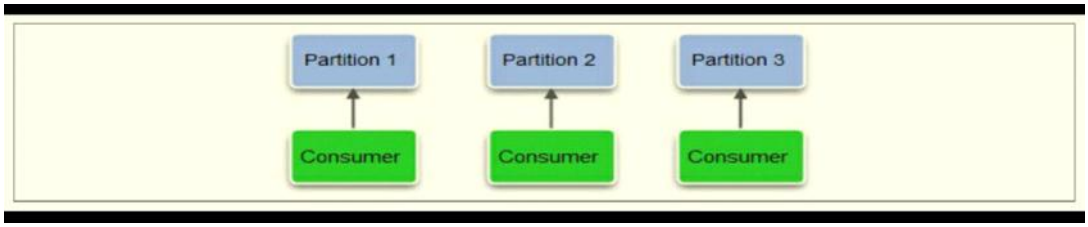
消费者的负载均衡

消费者的负载均衡则会相对复杂一些，我们这里讨论的是单个分组内的消费者集群的负载均衡，不同分组的负载均衡互不干扰。消费者的负载均衡跟Topic的分区数目和消费者的个数紧

密相关，我们分几个场景来讨论。

消费者数和Topic分区数一致

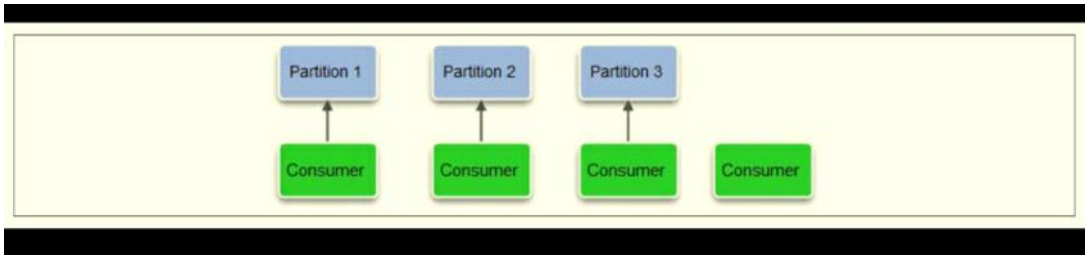
如果单个分组内的消费者数目和Topic总的分区数目相同，那么每个消费者负责消费一个分区中的消息，一一对应，如下图所示。



消费者数和Topic分区数一致情况下的分区消息消费示意图

消费者数大于Topic分区数

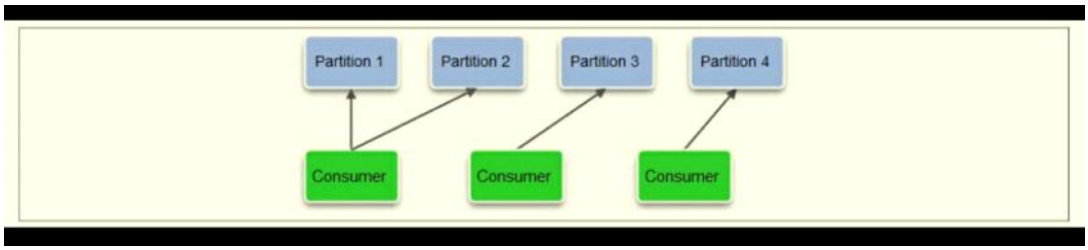
如果单个分组内的消费者数目比Topic总的分区数目多，则多出来的消费者不参与消费，如下图所示。



消费者数大于Topic分区数情况下的分区消息消费示意图

消费者数小于Topic分区数

如果分组内的消费者数目比Topic总的分区数目小，则有部分消费者需要额外承担消息的消费任务，具体如图6-34所示。



消费者数小于Topic分区数情况下的分区消息消费示意图

当分区数目（n）大于单个Group的消费者数目（m）的时候，则有n%m个

消费者需要额外承担1/n的消费任务，我们假设n无限大，那么这种策略还是能够达到负载均衡的目的的。

综上所述，单个分组内的消费者集群的负载均衡策略如下。

每个分区针对同一个Group只能挂载一个消费者，即每个分区至多同时允许被一个消费者进行消费。

如果同一个Group的消费者数目大于分区数目，则多出来的消费者将不参与消费。

如果同一个Group的消费者数目小于分区数目，则有部分消费者需要额外承担消费任务。

Metamorphosis的客户端会自动处理消费者的负载均衡，将消费者列表和分区列表分别排序，然后按照上述规则做合理的挂载。

从上述内容来看，合理地设置分区数目至关重要。如果分区数目太小，则有部分消费者可能闲置；如果分区数目太大，则对服务器的性能有影响。

在某个消费者发生故障或者发生重启等情况时，其他消费者会感知到这一变化（通过ZooKeeper的“节点变化”通知），然后重新进行负载均衡，以保证所有的分区都有消费者进行消费。

消息消费位点Offset存储

为了保证生产者和消费者在进行消息发送与接收过程中的可靠性和顺序性，同时也是为了尽可能地保证避免出现消息的重复发送和接收，Metamorphosis会将消息的消费记录Offset记录到

ZooKeeper上去，以尽可能地确保在消费者进行负载均衡的时候，能够正确地识别出指定分区的消息进度。

课外阅读：阿里—Dubbo

Dubbo是阿里巴巴于2011年10月正式开源的一个由Java语言编写的分布式服务框架，致力于提供高性能和透明化的远程服务调用方案和基于服务框架展开的完整SOA服务治理方案。目前项目主页地址为:

https://github.com/alibaba/dubbo。

Dubbo的核心部分包含以下三块。

远程通信：提供对多种基于长连接的NIO框架抽象封装，包括多种线程模型、序列化，以及“请求-响应”模式的信息交换方式。

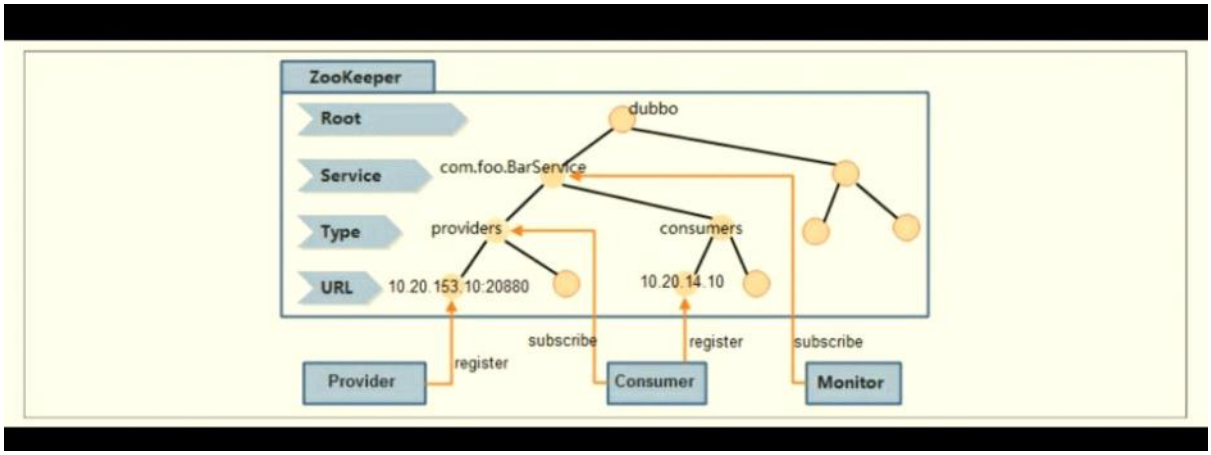
集群容错：提供基于接口方法的远

程过程透明调用，包括对多协议的支持，以及对软负载均衡、失败容错、地址路由和动态配置等集群特性的支持。

自动发现：提供基于注册中心的目录服务，使服务消费方能动态地查找服务提供方，使地址透明，使服务提供方可以平滑地增加或减少机器。

注册中心是RPC框架最核心的模块之一，用于服务的注册和订阅。在Dubbo的实现中，对注册中心模块进行了抽象封装，因此可以基于其提供的外部接口来实现各种不同类型的注册中心，例如数据库、ZooKeeper和Redis等。在本书前面部分我们已经多次提到，ZooKeeper是一个树形结构的目录服务，支持变更推送，因此非常适合作为Dubbo服务的注册中心，下面我们着重来看基于ZooKeeper实现的Dubbo注册中心。

在Dubbo注册中心的整体架构设计中，ZooKeeper上服务的节点设计如下图所示。



基于ZooKeeper实现的注册中心节点结构示意图

/dubbo：这是Dubbo在ZooKeeper上创建的根节点。

/dubbo/com.foo.BarService：这是服务节点，代表了Dubbo的一个服务。

/dubbo/com.foo.BarService/providers：这是服务提供者的根节点，其子节点代表了每一个服务的真正提供者。

/dubbo/com.foo.BarService/cons

sumers：这是服务消费者的根节点，其子节点代表了每一个服务的真正消费者。

结合图6-36，我们以“com.foo.BarService”这个服务为例，来说明Dubbo基于ZooKeeper实现的注册中心的工作流程。

服务提供者

服务提供者在初始化启动的时候，会首先在ZooKeeper的/dubbo/com.foo.BarService/providers节点下创建一个子节点，并写入自己的URL地址，这就代表了“com.foo.BarService”这个服务的一个提供者。

服务消费者

服务消费者会在启动的时候，读取并订阅ZooKeeper上/dubbo/com.foo.BarService/providers节点下的所有子节点，并解析出所有提供者的URL

地址来作为该服务地址列表，然后开始发起正常调用。

同时，服务消费者还会在ZooKeeper的/dubbo/com.foo.BarService/consumers节点下创建一个临时节点，并写入自己的URL地址，这就代表了“com.foo.BarService”这个服务的一个消费者。

监控中心

监控中心是Dubbo中服务治理体系的重要一部分，其需要知道一个服务的所有提供者和订阅者，及其变化情况。因此，监控中心在启动的时候，会通过ZooKeeper的/dubbo/com.foo.BarService节点来获取所有提供者和消费者的URL地址，并注册Watcher来监听其子节点变化。

另外需要注意的是，所有提供者在ZooKeeper上创建的节点都是临时节点，利用的是临时节点的生命周期和客户端会话相关的特性，因此一旦提供者所在的机器出现故障导致该提供者无法对外提供服务时，该临时节点就会自动从ZooKeeper上删除，这样服务的消费者和监控中心都能感知到服务提供者的变化。

在ZooKeeper节点结构设计上，以服务名和类型作为节点路径，符合Dubbo订阅和通知的需求，这样保证了以服务为粒度的变更通知，通知范围易于控制，即使在服务的提供者和消费者变更频繁的情况下，也不会对ZooKeeper造成太大的性能影响。

课外阅读：阿里—Canal

Canal是阿里巴巴于2013年1月正式开源的一个由纯Java语言编写的基于MySQL数据库Binlog实现的增量订阅和消费组件。目前项目主页地址为:

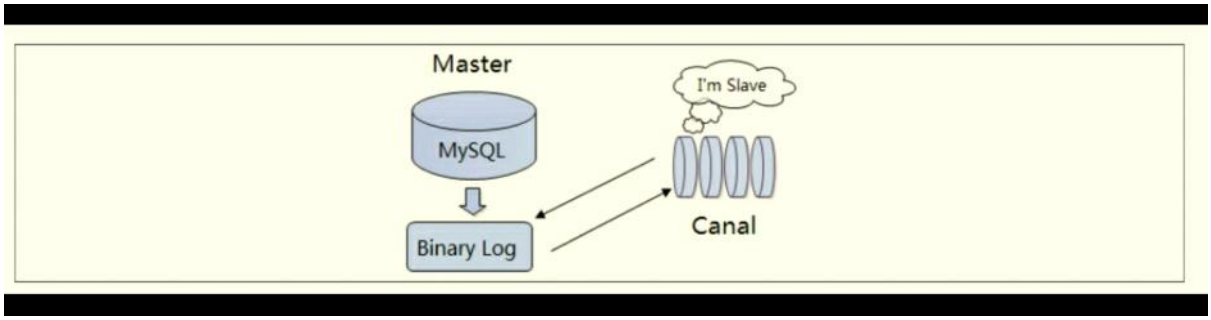
https://github.Com/alibaba/canal

由项目主要负责人，同时也是资深的开源爱好者agapple持续维护。

项目名Canal取自“管道”的英文单词，寓意数据的流转，是一个定位为基于MySQL数据库的Binlog增量日志来实现数据库镜像、实时备份和增量数据消费的通用组件。

早期的数据库同步业务，大多都是使用MySQL数据库的触发器机制（即Trigger）来获取数据库的增量变更。不过从2010年开始，阿里系下属各公司开始逐步尝试基于数据库的日志解析来获取增量变更，并在此基础上实现数据的同步，由此衍生出了数据库的增量订阅和消费业务——Canal项目也由此诞生了。

Canal的工作原理相对比较简单，其核心思想就是模拟MySQL Slave的交互协议，将自己伪装成一个MySQL的Slave机器，然后不断地向Master服务器发送Dump请求。Master收到Dump请求后，就会开始推送相应的Binary Log给该Slave（也就是Canal）。Canal收到Binary Log，解析出相应的Binary Log对象后就可以进行二次消费了，其基本工作原理如下图所示。

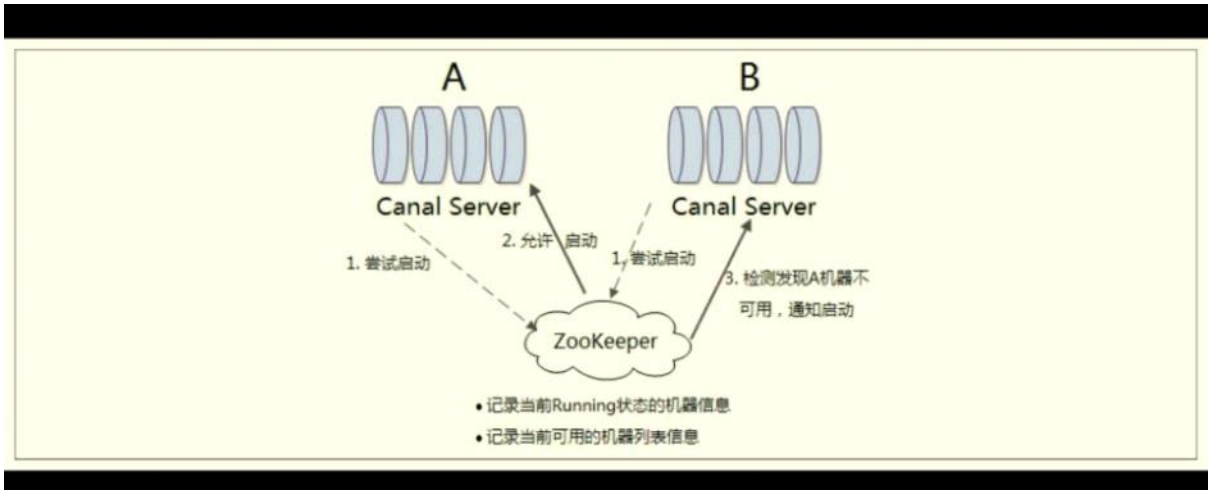


Canal基本工作原理示意图

Canal Server主备切换设计

在Canal的设计中，基于对容灾的考虑，往往会配置两个或更多个Canal Server来负责一个MySQL数据库实例的数据增量复制。另一方面，为了减少Canal Server的Dump请求对MySQL Master所带来的性能影响，就要求不同的Canal Server上的instance在同一时刻只能有一个处于Running状态，其他的instance都处于Standby状态，这就使

得Canal必须具备主备自动切换的能力。在Canal中，整个主备切换过程控制主要是依赖于ZooKeeper来完成的，如下图所示。



Canal Server主备切换机制

1. 尝试启动。

每个Canal Server在启动某个Canal instance的时候都会首先向ZooKeeper进行一次尝试启动判断。具体的做法是向ZooKeeper创建一个相同的临时节点，哪个Canal Server创建成功了，那么就让哪个Server启动。

以“example”这个instance为例来说明，所有的Canal Server在启动的时候，都会去创建/otter/canal/destinations/example/running节点，并且无论有多少个Canal Server同时并发启动，ZooKeeper都会保证最终只有一个Canal Server能够成功创建该节点。

2. 启动instance。

假设最终IP地址为

10.20.144.51的Canal Server成功创建了该节点，那么它就会将自己的机器信息写入到该节点中去：

{"active":true,"address":"

10.20.144.51:11111","cid":1}

并同时启动instance。而其他Canal Server由于没有成功创建节点，于是就会将自己的状态置为Standby，同时对/otter/canal/destinations/example/running节点注册Watcher监听，以监听该节点的变化情况。

3. 主备切换。

Canal Server在运行过程中，难免会发生一些异常情况导致其无法正常工作，这个时候就需要进行主备切换了。基于ZooKeeper临时节点的特性，当原本处于Running状态的Canal Server因为挂掉或网络等原因断开了与ZooKeeper的连接，那么/otter/canal/destinations/example/running节点就会在一段时间后消失。

由于之前处于Standby状态的所有Canal Server已经对该节点进行了监听，因此它们在接收到ZooKeeper发送过来的节点消失通知后，会重复进行步骤1——以此实现主备切换。

下面我们再来看看在主备切换设计过程中最容易碰到的一个问题，就是“假死”。所谓假死状态是指，Canal Server所在服务器的网络出现闪断，导致ZooKeeper认为其会话失效，从而释放了Running节点——但此时Canal Server对应的JVM并未退出，其工作状态是正常的。

在Canal的设计中，为了保护假死状态的Canal Server，避免因瞬间Running节点失效导致instance重新分布带来的资源消耗，所以设计了一个策略：

状态为Standby的Canal Server在收到Running节点释放的通知后，会延迟一段时间抢占Running节点，而原本处于Running状态的instance，即Running节点的拥有者可以不需要等待延迟，直接取得Running节点。

这样就可以尽可能地保证假死状态下一些无谓的资源释放和重新分配了。目前延迟时间的默认值为5秒，即Running节点针对假死状态的保护期为5秒。

Canal Client的HA设计

Canal Client在进行数据消费前，首先当然需要找到当前正在提供服务的Canal Server，即Master。在上面“主备切换”部分中我们已经讲到，针对每一个数据复制实例，例如example，都会在/otter/canal/destinations/example/running节点中记录下当前正在运行的Canal Server。因此，Canal Client只需要连接ZooKeeper，并从对应的节点上读取Canal Server信息即可。

1. 从ZooKeeper中读取出当前处于Running状态的Server。

Canal Client在启动的时候，会首先从/otter/canal/destinations/example/running节点上读取出当前处于Running状态的Server。同时，客户端也会将自己的信息注册到ZooKeeper的/otter/canal/destinations/example/1001/running节点上，其中“1001”代表了该客户端的唯一标识，其节点内容如下：

```
{"active":true,"address":"
10.12.48.171:50544","clientId":1001}
```

2. 注册Running节点数据变化的监听。

由于Canal Server存在挂掉的风险，因此Canal Client还会对/otter/canal/destinations/example/running节点注册一个节点变化的监听，这样一旦发生Server的主备切换，Client就可以随时感知到。

3. 连接对应的Running Server进行数据消费。

数据消费位点记录

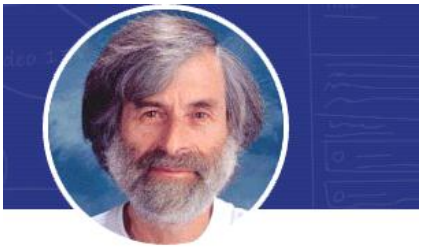
由于存在Canal Client的重启或其他变化，为了避免数据消费的重复性和顺序错乱，Canal必须对数据消费的位点进行实时记录。数据消费成功后，Canal Server会在ZooKeeper中记录

下当前最后一次消费成功的Binary Log位点，一旦发生Client重启，只需要从这最后一个位点继续进行消费即可。具体的做法是在ZooKeeper的/otter/canal/destinations/example/1001/cursor节点中记录下客户端消费的详细位点信息：

{"@type":"com.alibaba.otter.canal.protocol.position.LogPosition","identity":{"slaveId":-1,"sourceAddress":{"address":"

10.20.144.15","port":3306}},"postion":{"included":false,"journalName":"mysql-bin.002253","position":2574756,"timestamp":1363688722000}}

课外阅读：Paxos算法



Leslie Lamport

Leslie B. Lamport is an American computer scientist. Lamport is best known for his seminal work in distributed systems and as the initial developer of the document preparation system LaTeX. Leslie Lamport was the winner of the 2013 Turing Award for imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems, in which several autonomous computers communicate with each other by passing messages. He devised important algorithms and developed formal modeling and verification protocols that improve the quality of real distributed systems. These contributions have resulted in improved correctness, performance, and reliability of computer systems.

Read more about Leslie Lamport's extensive work and publication archive at: www.lamport.org.

Leslie Lamport（莱斯利·兰伯特）

Paxos算法的作者Leslie Lamport（莱斯利·兰伯特）及其对计算机科学尤其是分布式计算领域的杰出贡献。作为2013年的新科图灵奖得主，现年73岁的Lamport是计算机科学领域一位拥有杰出成就的传奇人物，其先后多次荣获ACM和IEEE以及其他各类计算机重大奖项。Lamport对时钟同步算法、面包店算法、拜占庭将军问题以及Paxos算法的创造性研究，极大地推动了计算机科学尤其是分布式计算的发展，全世界无数工程师得益于他的理论，其中Paxos算法的提出，正是Lamport多年的研究成果。

说起Paxos理论的发表，还有一段非常有趣的历史故事。Lamport早在1990年就已经将其对Paxos算法的研究论文The Part-Time Parliament提交给ACM TOCS Jnl.的评审委员会了，但是由于Lamport“创造性”地使用了故事的方式来

进行算法的描述，导致当时委员会的工作人员没有一个能够正确地理解他对算法的描述，时任主编要求Lamport使用严谨的数据证明方式来描述该算法，否则他们将不考虑接受这篇论文。遗憾的是，Lamport并没有接收他们的建议，当然也就拒绝了对论文的修改，并撤销了对这篇论文的提交。在后来的一个会议上，Lamport还对此事耿耿于怀：“为什么这些搞理论的人一点幽默感也没有呢？”

幸运的是，还是有人能够理解Lamport那公认的令人晦涩的算法描述的。1996年，来自微软的Butler Lampson在WDAG96上提出了重新审视这篇分布式论文的建议，在次年的WDAG97上，麻省理工学院的Nancy Lynch也公布了其根据Lamport的原文重新修改后的Revisiting the Paxos Algorithm，“帮助”Lamport用数学的形式化术语定义并证明了Paxos算法。于是在1998年的ACM TOCS上，这篇延迟了9年的论文终于被接受了，也标志着Paxos算法正式被计算机科学接受并开始影响更多的工程师解决分布式一致性问题。

后来在2001年，Lamport本人也做出了让步，这次他放弃了故事的描述方式，而是使用了通俗易懂的语言重新讲述了原文，并发表了Paxos Made Simple——当然，Lamport甚为固执地认为他自己的表述语言没有歧义，并且也足够让人明白Paxos算法，因此不需要数学来协助描述，于是整篇文章还是没有任何数学符号。好在这篇文章已经能够被更多的人理解，相信绝大多数的Paxos爱好者也都是从这篇文章开始慢慢进入了Paxos的神秘世界。
由于Lamport个人自负固执的性格，使得Paxos理论的诞生可谓一波三折。关于Paxos理论的诞生过程，后来也成为了计算机科学领域被广泛流传的学术趣事。

拜占庭将军问题的提出

1982年，Lamport与另两人共同发表了论文The Byzantine Generals Problem，提出了一种计算机容错理论。在理论描述过程中，为了将所要描述的问题形象的表达出来，Lamport设想出了下面这样一个场景：

拜占庭帝国有许多支军队，不同军队的将军之间必须制订一个统一的行动计划，从而做出进攻或者撤退的决定，同时，各个将军在地理上都是被分隔开来的，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，这些叛徒可以任意篡改消息，从而达到欺骗将军的目的。

这就是著名的“拜占庭将军问题”。实际上拜占庭将军问题是一个分布式环境下的协议问题，拜占庭帝国军队的将军

们必须全体一致的决定是否攻击某一支敌军。

Paxos算法的诞生

Lamport在1990年提出了一个理论上的一致性解决方案，同时给出了严格的数学证明。鉴于之前采用故事类比的方式成功的阐述了“拜占廷将军问题”，因此这次Lamport同样用心良苦地设想出了一个场景来描述这种一致性算法需要解决的问题，及其具体的解决过程：

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。值得注意的是，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的传递消息，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。

这就是论文The Part-Time Parliament中提到的兼职议会，而Paxos算法名称的由来也是取自论文中提到的Paxos小岛。在这个论文中，Lamport压根没有说Paxos小岛是虚构出来的，而是煞有介事的说是考古工作者发现了Paxos议会事务的手稿，从这些手稿猜测Paxos人开展议会的方法。因此，在这个论文中，Lamport从问题的提出到算法的推演论证，通篇贯穿了对Paxos议会历史的描述。

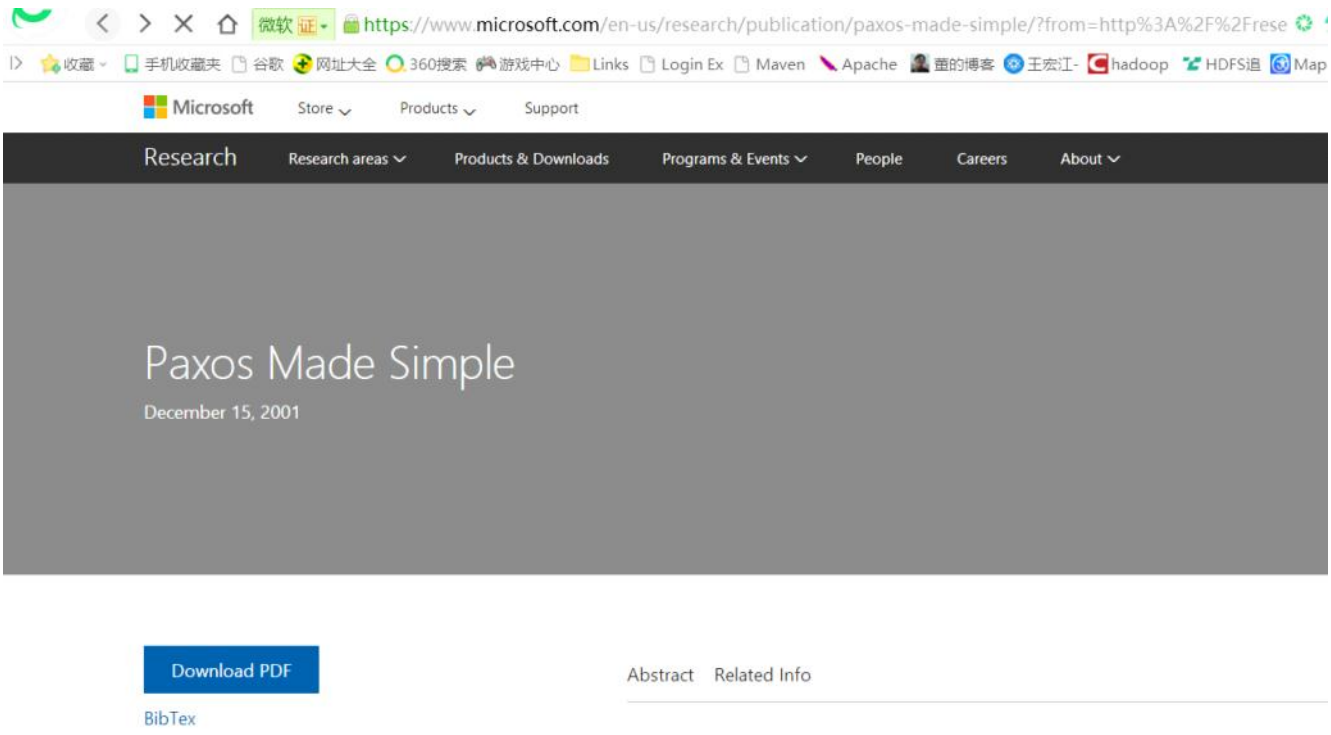
算法陈述

Paxos算法实际上也是一个类2pc算法，而重点是引入了“过半性”的投票理念，通俗地讲就是少数服从多数的原则。此外，Paxos算法支持分布式节点角色之间的轮换，即当协调者出现问题后，参与者可以变成协调者工作。这极大地避免了分布式单点的出现，因此Paxos算法既解决了无限期等待问题，也提高了性能，是目前来说最优秀的分布式一致性协议之一。

课外阅读

论文下载地址：

https://www.microsoft.com/en-us/research/publication/paxos-made-simple/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Fpaxos-simple.pdf


https://www.microsoft.com/en-us/research/publication/paxos-made-simple/?from=http%3A%2F%2Frese
Microsoft
Store
Products
Support
Research
Research areas
Products & Downloads
Programs & Events
People
Careers
About
Paxos Made Simple
December 15, 2001
Download PDF
BibTex
Abstract
Related Info