zookeeper简介 - OPEN 开发经验库

一直对zookeeper的应用和原理比较迷糊，今天看一篇文章，讲得很通透，分享如下：

场景一

有这样一个场景：系统中有大约100w的用户，每个用户平 均有3个邮箱账号，每隔5分钟，每个邮箱账需要收取100封邮件，最多3亿份邮件需要下载到服务器中(不含附件和正文)。用20台机器划分计算的压力，从 多个不同的网路出口进行访问外网，计算的压力得到缓解，那么每台机器的计算压力也不会很大了。

通过我们的讨论和以往的经验判断在这场景中可以实现并行计算，但我们还期望能对并行计算的节点进行动态的添加/删除，做到在线更新并行计算的数目并且不会影响计算单元中的其他计算节点，但是有4个问题需要解决，否则会出现一些严重的问题：

1. 20台机器同时工作时，有一台机器down掉了，其他机器怎么进行接管计算任务，否则有些用户的业务不会被处理，造成用户服务终断。
2. 随着用户数量增加，添加机器是可以解决计算的瓶颈，但需要重启所有计算节点，如果需要，那么将会造成整个系统的不可用。
3. 用户数量增加或者减少，计算节点中的机器会出现有的机器资源使用率繁忙，有的却空闲，因为计算节点不知道彼此的运行负载状态。
4. 怎么去通知每个节点彼此的负载状态，怎么保证通知每个计算节点方式的可靠性和实时性。

先不说那么多专业名词，白话来说我们需要的是：1记录状态，2事件通知 ，3可靠稳定的中央调度器，4易上手、管理简单。

采用Zookeeper完全可以解决我们的问题，分布式计算中的协调员，观察者，分布式锁都可以作为zookeeper的关键词，在系统中利用Zookeeper来处理事件通知,队列,优先队列,锁,共享锁等功能，利用这些特色在分布式计算中发挥重要的作用。

场景二

假设我们我们有个20个搜索引擎的服务器(每个负责总索引中的一部分的搜索任务)和一个总服务器(负责向这20个搜索引擎的服务器发出搜索请求并合并 结果集),一个备用的总服务器(负责当总服务器宕机时替换总服务器),一个web的 cgi(向总服务器发出搜索请求).搜索引擎

的服务器中的15个服务器现在提供搜索服务,5个服务器正在生成索引.这20个搜索引擎的服务器经常要让正在 提供搜索服务的服务器停止提供服务开始生成索引,或生成索引的服务器已经把索引生成完成可以搜索提供服务了.使用Zookeeper可以保证总服务器自动 感知有多少提供搜索引擎的服务器并向这些服务器发出搜索请求,备用的总服务器宕机时自动启用备用的总服务器,web的cgi能够自动地获知总服务器的网络 地址变化.这些又如何做到呢?

1. 提供搜索引擎的服务器都在Zookeeper中创建znode,zk.create("/search/nodes/node1",

"hostname".getBytes(), Ids.OPEN_ACL_UNSAFE, CreateFlags.EPHEMERAL);

2.总服务器可以从Zookeeper中获取一个znode的子节点的列表,zk.getChildren("/search/nodes", true);

3.总服务器遍历这些子节点,并获取子节点的数据生成提供搜索引擎的服务器列表.

4.当总服务器接收到子节点改变的事件信息,重新返回第二步.

5.总服务器在Zookeeper中创建节点,zk.create("/search/master", "hostname".getBytes(), Ids.OPEN_ACL_UNSAFE, CreateFlags.EPHEMERAL);

6.备用的总服务器监控Zookeeper中的"/search/master"节点.当这个znode的节点数据改变时,把自己启动变成总服务器,并把自己的网络地址数据放进这个节点.

7.web的cgi从Zookeeper中"/search/master"节点获取总服务器的网络地址数据并向其发送搜索请求.

8.web的cgi监控Zookeeper中的"/search/master"节点,当这个znode的节点数据改变时,从这个节点获取总服务器的网络地址数据,并改变当前的总服务器的网络地址.

在我的测试中:一个Zookeeper的集群中,3个Zookeeper节点.一个leader,两个follower的情况下,停掉leader,然后两个follower选举出一个leader.获取的数据不变.我想Zookeeper能够帮助Hadoop做到:

Hadoop,使用Zookeeper的事件处理确保整个集群只有一个NameNode,存储配置信息等.

HBase,使用Zookeeper的事件处理确保整个集群只有一个HMaster,察觉HRegionServer联机和宕机,存储访问控制列表等.

zookeeper是什么

官方说辞：Zookeeper 分布式服务框架是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。

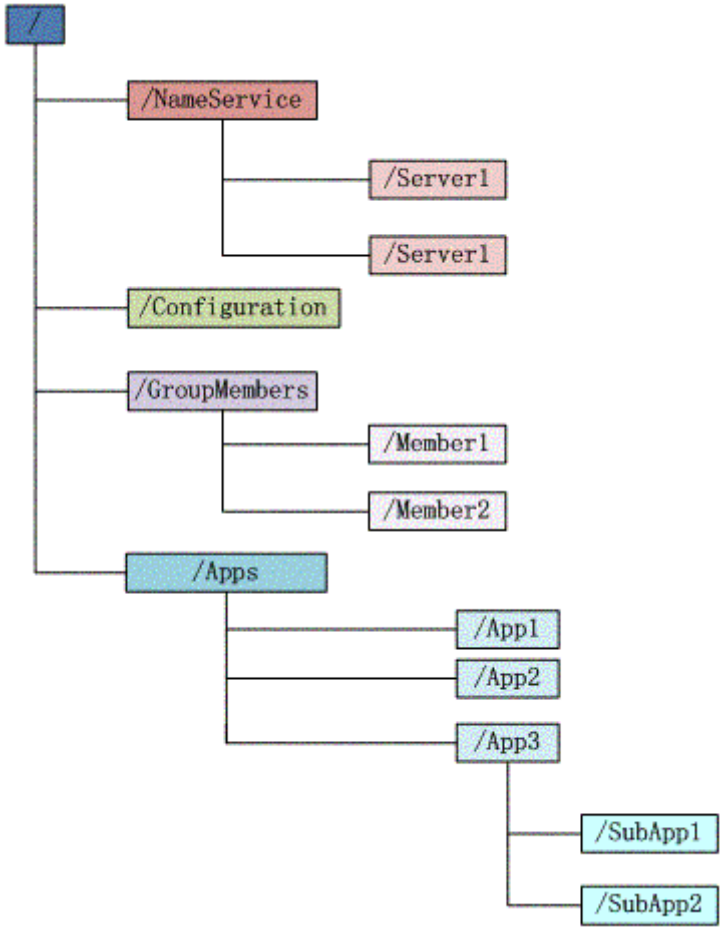
好抽象，我们改变一下方式，先看看它都提供了哪些功能，然后再看看使用它的这些功能能做点什么。

zookeeper提供了什么

简单的说，zookeeper=文件系统+通知机制。

1、 文件系统

Zookeeper维护一个类似文件系统的数据结构：



每个子目录项如 NameService 都被称作为 znode，和文件系统一样，我们能够自由的增加、删除znode，在一个znode下增加、删除子znode，唯一的不同在于znode是可以存储数据的。

有四种类型的znode：

1、PERSISTENT-持久化目录节点

客户端与zookeeper断开连接后，该节点依旧存在

2、 PERSISTENT_SEQUENTIAL-持久化顺序编号目录节点

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

3、EPHEMERAL-临时目录节点

客户端与zookeeper断开连接后，该节点被删除

4、EPHEMERAL_SEQUENTIAL-临时顺序编号目录节点

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

2、 通知机制

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

就这么简单，下面我们看看能做点什么呢？

我们能用zookeeper做什么

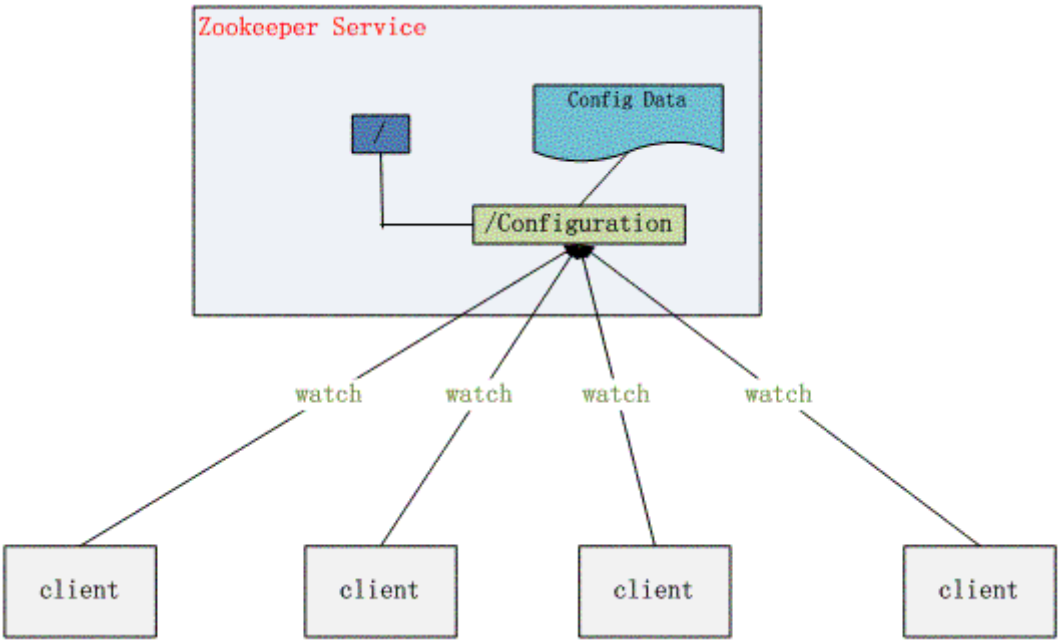
1、 命名服务

这个似乎最简单，在zookeeper的文件系统里创建一个目录，即有唯一的path。在我们使用tborg无法确定上游程序的部署机器时即可与下游程序约定好path，通过path即能互相探索发现，不见不散了。

2、 配置管理

程序总是需要配置的，如果程序分散部署在多台机器上，要逐个改变配置就变得困难。好吧，现在把这些配置全部放到zookeeper上去，保存在 Zookeeper 的某个目录节点中，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会

收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中就好。

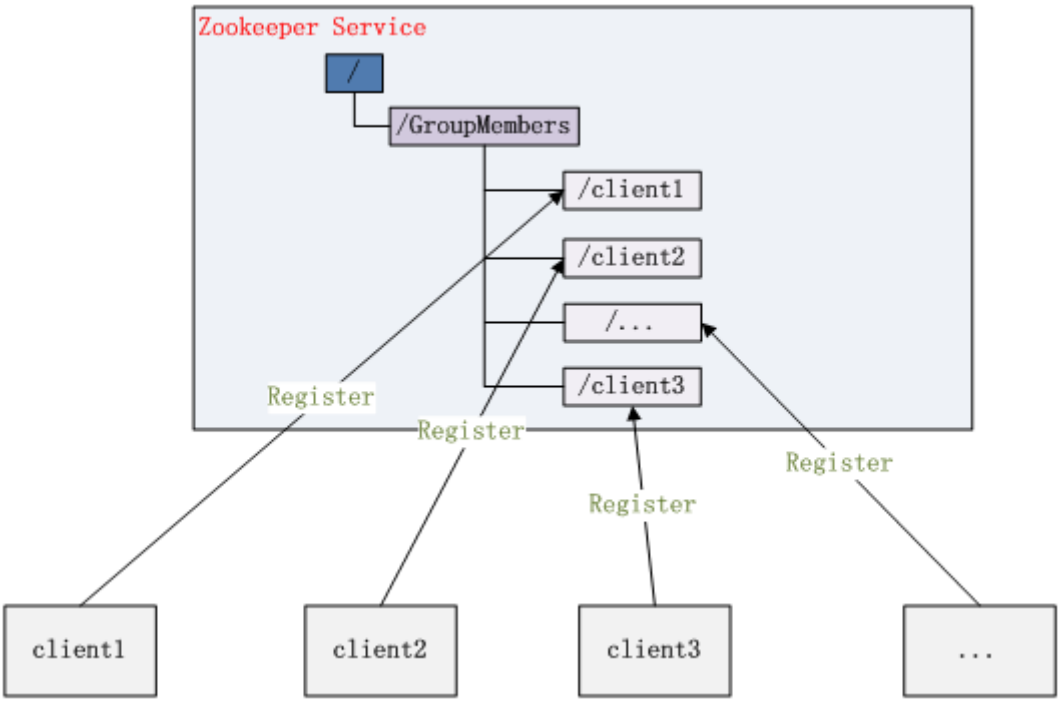


3、 集群管理

所谓集群管理无在乎两点：是否有机器退出和加入、选举master。

对于第一点，所有机器约定在父目录GroupMembers下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。新机器加入 也是类似，所有机器收到通知：新兄弟目录加入，highcount又有了。

对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为master就好。

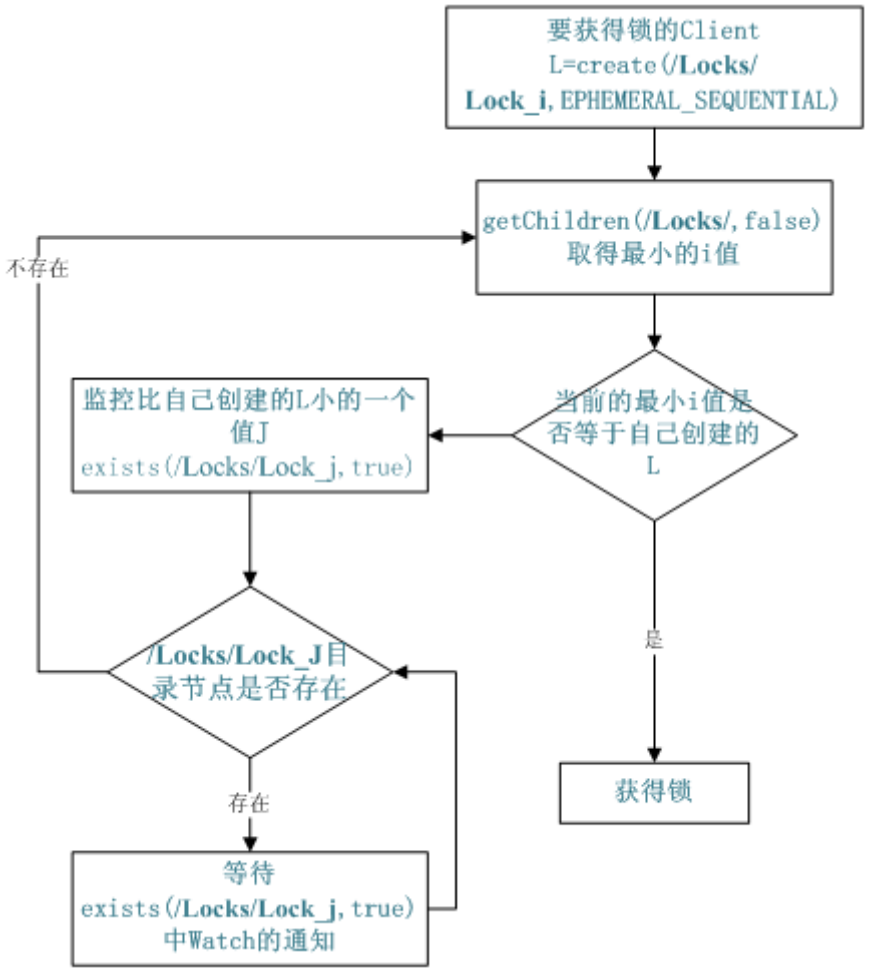


4、 分布式锁

有了zookeeper的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将zookeeper上的一个znode看作是一把锁，通过createznode的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。厕所有言：来也冲冲，去也冲冲，用完删除掉自己创建的distribute_lock 节点就释放出锁。

对于第二类， /distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选master一样，编号最小的获得锁，用完删除，依次方便。



5、队列管理

两种类型的队列：

1、 同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

2、队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。

终于了解完我们能用zookeeper做什么了，可是作为一个程序员，我们总是想狂热了解zookeeper是如何做到这一点的，单点维护一个文件系统没有什么难度，可是如果是一个集群维护一个文件系统保持数据的一致性就非常困难了。

分布式与数据复制

Zookeeper作为一个集群提供一致的数据服务，自然，它要在所有机器间做数据复制。数据复制的好处：

1、 容错
一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作；

2、提高系统的扩展能力
把负载分布到多个节点上，或者增加节点来提高系统的负载能力；

3、提高性能
让客户端本地访问就近的节点，提高用户访问速度。

从客户端读写访问的透明度来看，数据复制集群系统分下面两种：

1、写主(WriteMaster)
对数据的修改提交给指定的节点。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称读写分离；

2、写任意(Write Any)
对数据的修改可提交给任意的节点，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

对zookeeper来说，它采用的方式是写任意。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降（这 也是它建立observer的原因），而响应能力则取决于具体实现方式，是延迟复制保持最终一致性，还是立即复制快速响应。

我们关注的重点还是在如何保证数据在集群所有机器的一致性，这就涉及到paxos算法。

数据一致性与paxos算法

据说Paxos算法的难理解与算法的知名度一样令人敬仰，所以我们先看如何保持数据的一致性，这里有个原则就是：

在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。

Paxos算法解决的什么问题呢，解决的就是保证每个节点执行相同的操作序列。好吧，这还不简单，master维护一个全局写队列，所有写操作都必须 放入这个队列编号，那么无论我们写多少个节点，只要写操作是按编号来的，就能保证一致性。没错，就是这样，可是如果master挂了呢。

Paxos算法通过投票来对写操作进行全局编号，同一时刻，只有一个写操作被批准，同

时并发的写操作要去争取选票，只有获得过半数选票的写操作才会被 批准（所以永远只会有一个写操作得到批准），其他的写操作竞争失败只好再发起一轮投票，就这样，在日复一日年复一年的投票中，所有写操作都被严格编号排 序。编号严格递增，当一个节点接受了一个编号为100的写操作，之后又接受到编号为99的写操作（因为网络延迟等很多不可预见原因），它马上能意识到自己 数据不一致了，自动停止对外服务并重启同步过程。任何一个节点挂掉都不会影响整个集群的数据一致性（总2n+1台，除非挂掉大于n台）。

总结

Zookeeper 作为 Hadoop 项目中的一个子项目，是 Hadoop 集群管理的一个必不可少的模块，它主要用来控制集群中的数据，如它管理 Hadoop 集群中的 NameNode，还有 Hbase 中 Master Election、Server 之间状态同步等。

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，它包含一个简单的原语集，分布式应用程序可以基于它实现同步服务，配置维护和 命名服务等。Zookeeper是hadoop的一个子项目，其发展历程无需赘述。在分布式应用中，由于工程师不能很好地使用锁机制，以及基于消息的协调 机制不适合在某些应用中使用，因此需要有一种可靠的、可扩展的、分布式的、可配置的协调机制来统一系统的状态。Zookeeper的目的就在于此。本文简 单分析zookeeper的工作原理，对于如何使用zookeeper不是本文讨论的重点。

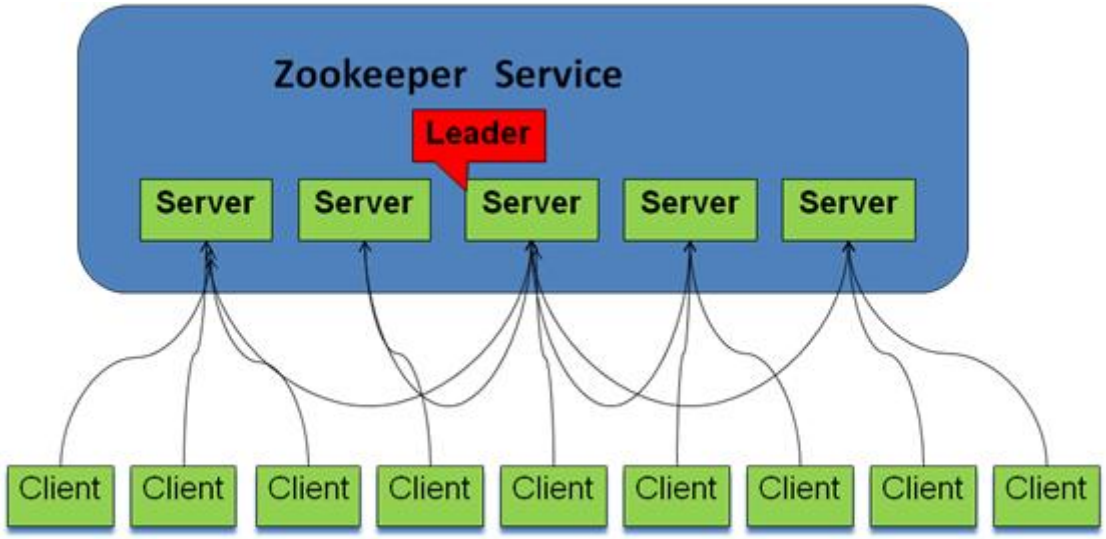
Zookeeper的基本概念

1.1 角色

Zookeeper中的角色主要有以下三类，如下表所示：

角色		描述
领导者（Leader）		领导者负责进行投票的发起和决议，更新系统状态
学习者（Learner）	跟随者（Follower）	Follower 用于接收客户请求并向客户端返回结果，在选主过程中参与投票
	观察者（ObServer）	ObServer 可以接收客户端连接，将写请求转发给 leader 节点。但 ObServer 不参加投票过程，只同步 leader 的状态。ObServer 的目的是为了扩展系统，提高读取速度
客户端（Client）		请求发起方

系统模型如图所示：



1.2 设计目的

1.最终一致性：client不论连接到哪个Server，展示给它都是同一个视图，这是zookeeper最重要的性能。

2 .可靠性：具有简单、健壮、良好的性能，如果消息m被到一台服务器接受，那么它将被所有的服务器接受。

3 .实时性：Zookeeper保证客户端将在一个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。但由于网络延时等原因，Zookeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用sync()接口。

4 .等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

5.原子性：更新只能成功或者失败，没有中间状态。

6 .顺序性：包括全局有序和偏序两种：全局有序是指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；偏序是指如果一个消息b在消息a后被同一个发送者发布，a必将排在b前面。

ZooKeeper的工作原理

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分 别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和 leader的状态同步以后，恢复模式就结束了。状态同步保证了leader

和Server具有相同的系统状态。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上 了zxid。实现中zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个 新的epoch，标识当前属于那个leader的统治时期。低32位用于递增计数。

每个Server在工作过程中有三种状态：

- LOOKING：当前Server不知道leader是谁，正在搜寻
- LEADING：当前Server即为选举出来的leader
- FOLLOWING：leader已经选举出来，当前Server与之同步

2.1 选主流程

当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的 Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。先介绍basic paxos流程：

1 .选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；

2 .选举线程首先向所有Server发起一次询问(包括自己)；

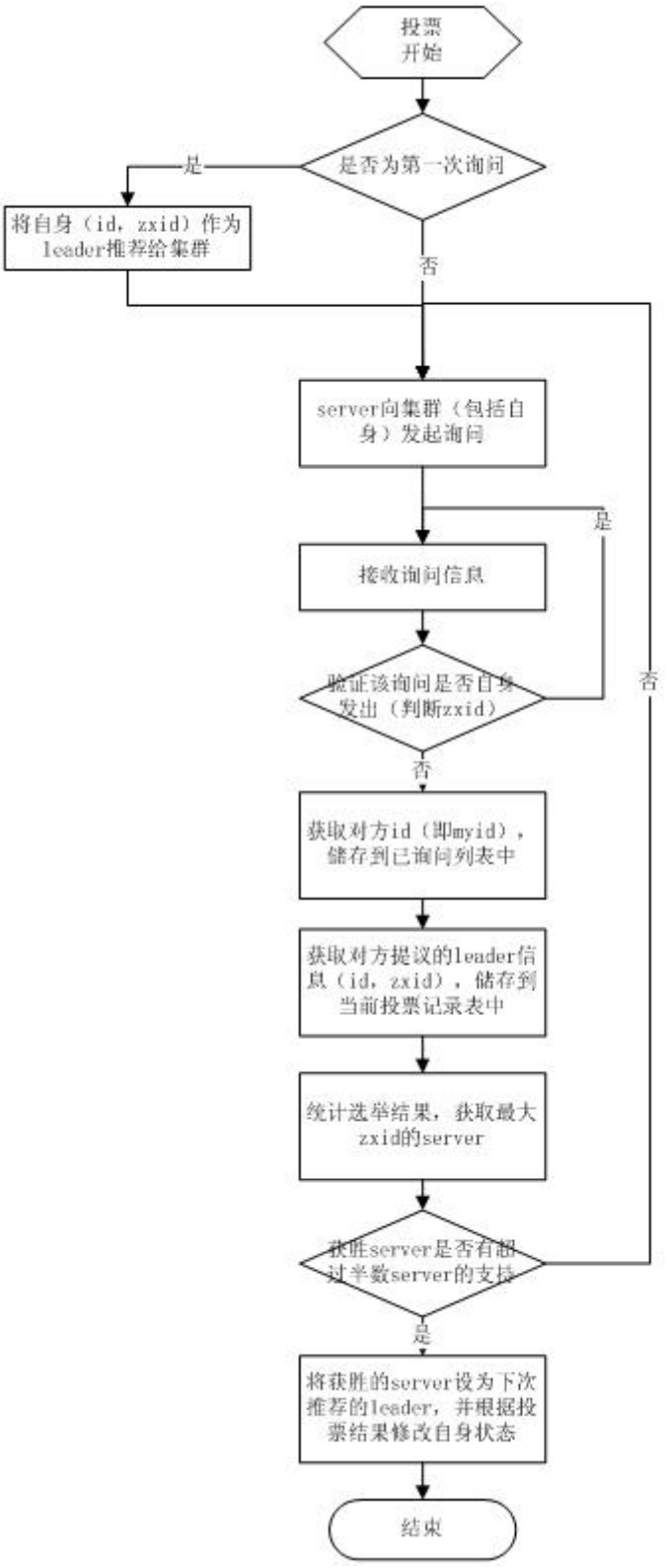
3 .选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；

4. 收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；

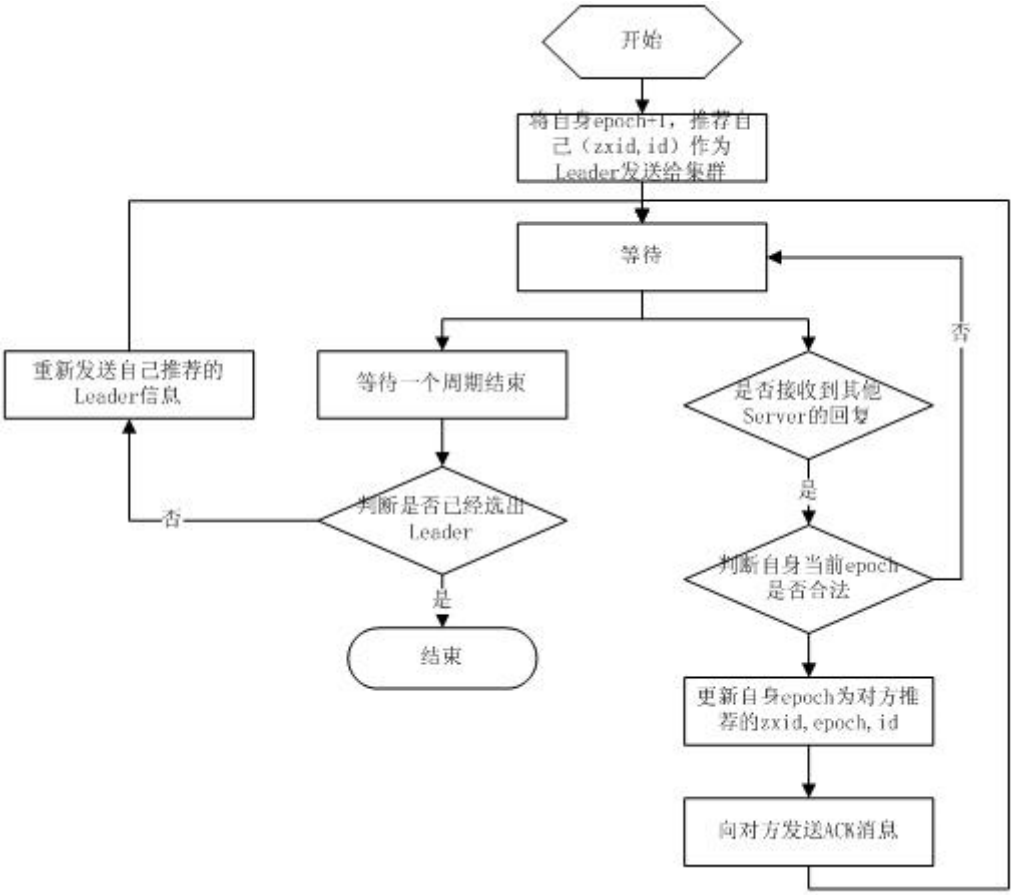
5. 线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数， 设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。

通过流程分析我们可以得出：要使Leader获得多数Server的支持，则Server总数必须是奇数2n+1，且存活的Server的数目不得少于n+1.

每个Server启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。选主的具体流程图如下所示：



fast paxos流程是在选举过程中，某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和 zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。其流程图如下所示：



2.2 同步流程

选完leader以后，zk就进入状态同步过程。

1. leader等待server连接；

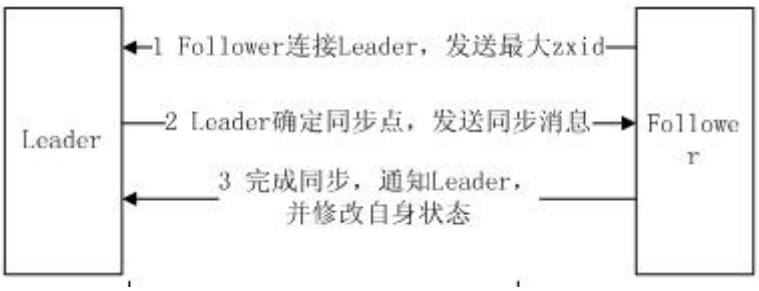
2 .Follower连接leader，将最大的zxid发送给leader；

3 .Leader根据follower的zxid确定同步点；

4 .完成同步后通知follower 已经成为uptodate状态；

5 .Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。

流程图如下所示：



2.3 工作流程

2.3.1 Leader工作流程

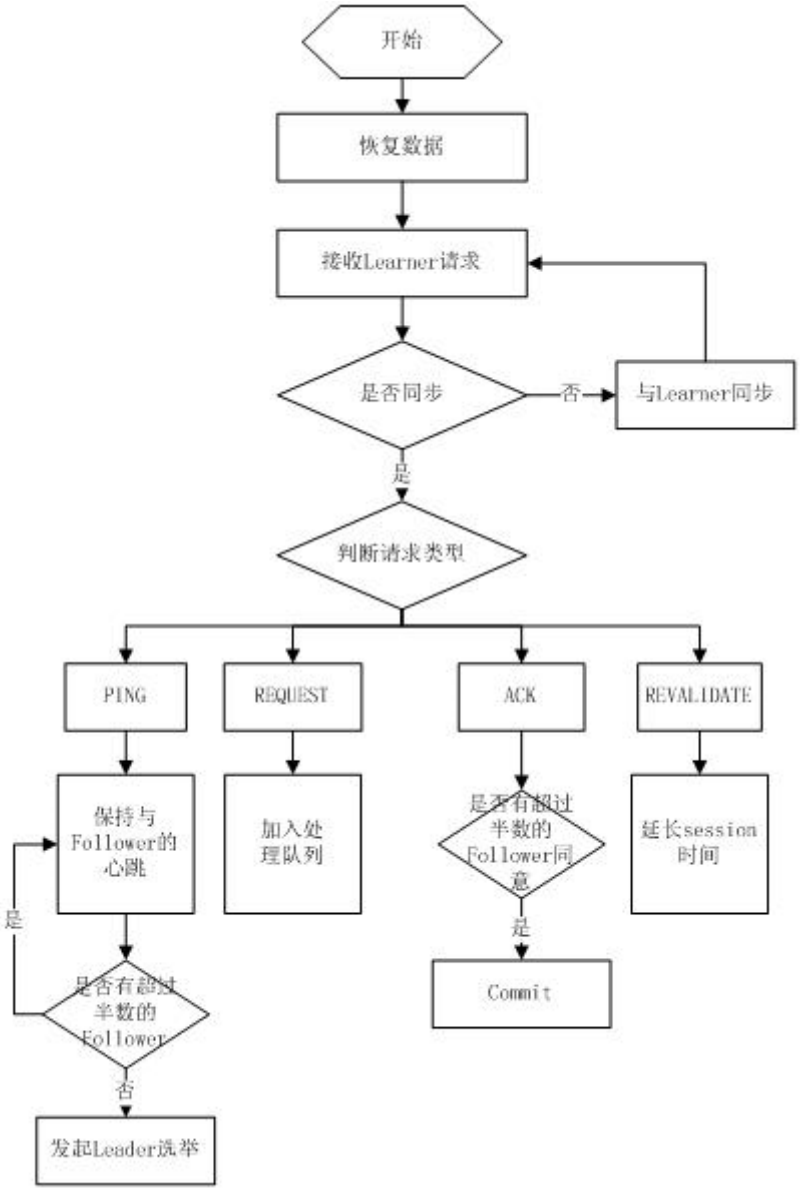
Leader主要有三个功能：

1 .恢复数据；

2 .维持与Learner的心跳，接收Learner请求并判断Learner的请求消息类型；

3 .Learner的消息类型主要有PING消息、REQUEST消息、ACK消息、REVALIDATE消息，根据不同的消息类型，进行不同的处理。

PING消息是指Learner的心跳信息；REQUEST消息是Follower发送的提议信息，包括写请求及同步请求；ACK消息是 Follower的对提议的回复，超过半数的Follower通过，则commit该提议；REVALIDATE消息是用来延长SESSION有效时间。
Leader的工作流程简图如下所示，在实际实现中，流程要比下图复杂得多，启动了三个线程来实现功能。



2.3.2 Follower工作流程

Follower主要有四个功能：

1. 向Leader发送请求（PING消息、REQUEST消息、ACK消息、REVALIDATE消息）；

2 .接收Leader消息并进行处理；

3 .接收Client的请求，如果为写请求，发送给Leader进行投票；

4 .返回Client结果。

Follower的消息循环处理如下几种来自Leader的消息：

1 .PING消息： 心跳消息；

2 .PROPOSAL消息：Leader发起的提案，要求Follower投票；

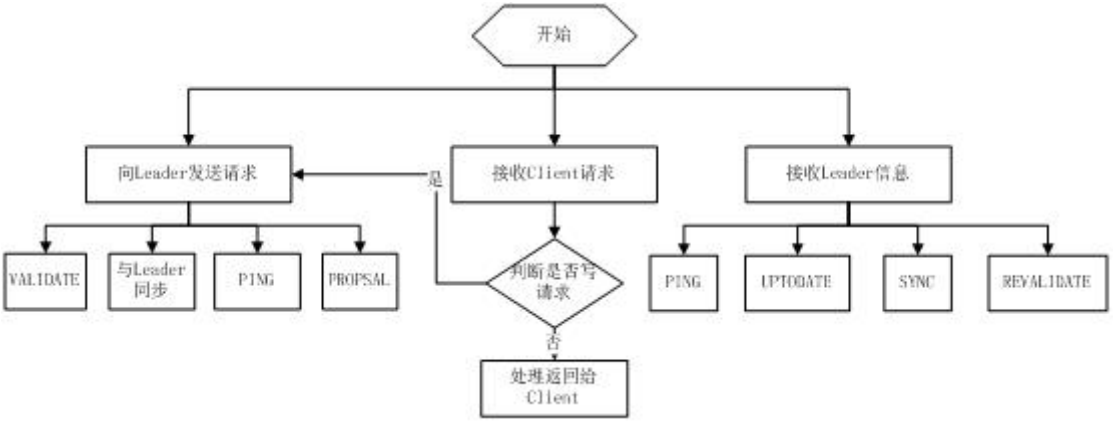
3 .COMMIT消息：服务器端最新一次提案的信息；

4 .UPTODATE消息：表明同步完成；

5 .REVALIDATE消息：根据Leader的REVALIDATE结果，关闭待revalidate的session还是允许其接受消息；

6 .SYNC消息：返回SYNC结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

Follower的工作流程简图如下所示，在实际实现中，Follower是通过5个线程来实现功能的。



对于observer的流程不再叙述，observer流程和Follower的唯一不同的地方就是observer不会参加leader发起的投票。

ZooKeeper典型使用场景一览

ZooKeeper是一个高可用的分布式数据管理与系统协调框架。基于对Paxos算法的实现，使该框架保证了分布式环境中数据的强一致性，也正是基 于这样的特性，使得zookeeper能够应用于很多场景。网上对zk的使用场景也有不少介绍，本文将结合作者身边的项目例子，系统的对zk的使用场景进 行归类介绍。 值得注意的是，zk并不是生来就为这些场景设计，都是后来众多开发者根据框架的特性，摸索出来的典型使用方法。因此，也非常欢迎你分享你在ZK使用上的奇 技淫巧。

场景类别	典型场景描述（ZK特性，使用方法）	应用中的具体使用	
		1. 索引信息和集群中机器节点状态存放在	

数据发布与订阅	发布与订阅即所谓的配置管理，顾名思义就是将数据发布到zk节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，地址列表等就非常适合使用。	zk的一些指定节点，供各个客户端订阅使用。2. 系统日志（经过处理后的）存储，这些日志通常2-3天后被清除。 3. 应用中用到的一些配置信息集中管理，在应用启动的时候主动来获取一次，并且在节点上注册一个Watcher，以后每次配置有更新，实时通知到应用，获取最新配置信息。 4. 业务逻辑中需要用到的一些全局变量，比如一些消息中间件的消息队列通常有个offset，这个offset存放在zk上，这样集群中每个发送者都能知道当前的发送进度。 5. 系统中有些信息需要动态获取，并且还会存在人工手动去修改这个信息。以前通常是暴露出接口，例如JMX接口，有了zk后，只要将这些信息存放到zk节点上即可。	

Name Service	这个主要是作为分布式命名服务，通过调用zk的create node api，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。	
分布通知/协调	ZooKeeper 中特有watcher注册与异步通知机制，能够很好的实现分布式环境下不同系统之间的通知与协调，实现对数据变更的实时处理。使用方法通常是不同系统都对 ZK上同一个znode进行注册，监听znode的变化（包括znode本身内容及子节点的），其中一个系统update了znode，那么另一个系统能 够收到通知，并作出相应处理。	1. 另一种心跳检测机制：检测系统和被检测系统之间并不直接关联起来，而是通过zk上某个节点关联，大大减少系统耦合。 2. 另一种系统调度模式：某系统有控制台和推送系统两部分组成，控制台的职责是控制推送系统进行相应的推送工作。管理人员在控制台作的一些操作，实际上是修改 了ZK上某些节点的状态，而zk就把这些变化通知给他们注册Watcher的客户端，即推送系统，于是，作出相应的推送任务。 3. 另一种工作汇报模式：一些类似于任务分发系统，子任务启动后，到zk来注册一个临时节点，并且定时将自己的进度进行汇报（将进度写回这

		个临时节点），这样任务管理者就能够实时知道任务进度。 总之，使用zookeeper来进行分布式通知和协调能够大大降低系统之间的耦合。
分布式锁	分布式锁，这个主要得益于ZooKeeper为我们保证了数据的强一致性，即用户只要完全相信每时每刻，zk集群中任意节点（一个zk server）上的相同znode的数据是一定是相同的。锁服务可以分为两类，一个是保持独占，另一个是控制时序。 所 谓保持独占，就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法是把zk上的一个znode看作是一把锁，通过create znode的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。 控 制时序，就是所有视图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局时序了。做法和上面基本类似，只是这里 /distribute_lock 已经预先存在，客户端在它下面创建临时有序节点（这个可以通过节点的属性控制：CreateMode.EPHEMERAL_SEQUENTIAL来指 定）。Zk的父节点（/distribute_lock）维持一份sequence,保证子节点创建的时序性，从而也形成了每个客户端的全局时序。	
	1. 集群机器监 控：这通常用于那种对集群中机器	

	状态，机器在线率有较高要求的场景，能够快速对集群中机器变化作出响应。这样的场景中，往往有一个监控系统，实时检测集群 机器是否存活。过去的做法通常是：监控系统通过某种手段（比如ping）定时检测每个机器，或者每个机器自己定时向监控系统汇报“我还活着”。 这种做法可行，但是存在两个比较明显的问题：1. 集群中机器有变动的时候，牵连修改的东西比较多。2. 有一定的延时。 利 用ZooKeeper有两个特性，就可以实时另一种集群机器存活性监控系统：a. 客户端在节点 x 上注册一个Watcher，那么如果 x 的子节点变化了，会通知该客户端。b. 创建EPHEMERAL类型的节点，一旦客户端和服务器的会话结束或过期，那么该节点就会消失。 例 如，监控系统在 /clusterServers 节点上注册一个Watcher，以后每动态加机器，那么就往 /clusterServers 下创建一个 EPHEMERAL类型的节点：/clusterServers/{hostname}. 这样，监控系统就能够实时知道机器的增减情况，至于后续处理就是监控系统的业务了。 2. Master选举则是zookeeper中最为经典的使用场景了。 在 分布式环境中，相同的业务应用分布在不同的机器上，有些业务逻辑（例如一些耗时的计算，网络I/O处理），往往只需要让整个集群中的某一台机器进行执行， 其余机器可以共享这个结果，这样可以大大减少重复劳动，提高性能，于是这个master选举便是这种场景下的碰到的主要问题。 利用ZooKeeper的强一致性，能够保证在分布式	1. 在搜索系统中，如果集群中每个机器都生成一份全量索引，不仅耗时，而且不能保证彼此之间索引数据一致。因此让集群中的Master来进行全量索引的生成， 然后同步到集群中其它机器。2. 另外，Master选举的容灾措施是，可以随时进行手动指定master，就是说应用在zk在无法获取master信息时，可以通过比如http方式，向 一个地方获取	
集群管理			

	高并发情况下节点创建的全局唯一性，即：同时有多个客户端请求创建 /currentMaster 节点，最终一定只有一个客户端请求能够创建成功。 利用这个特性，就能很轻易的在分布式环境中进行集群选取了。 另外，这种场景演化一下，就是动态Master选举。这就要用到 EPHEMERAL_SEQUENTIAL类型节点的特性了。 上 文中提到，所有客户端创建请求，最终只有一个能够创建成功。在这里稍微变化下，就是允许所有请求都能够创建成功，但是得有个创建顺序，于是所有的请求最终 在ZK上创建结果的一种可能情况是这样： /currentMaster/{sessionId}-1 , /currentMaster/{sessionId}-2 , /currentMaster/{sessionId}-3 ….. 每次选取序列号最小的那个机器作为Master，如果这个机器挂了，由于他创建的节点会马上小时，那么之后最小的那个机器就是Master了。	master。
分布式队列	队列方面，我目前感觉有两种，一种是常规的先进先出队列，另一种是要等到队列成员聚齐之后的才统一按序执行。对于第二种先进先出队列，和分布式锁服务中的控制时序场景基本原理一致，这里不再赘述。 第 二种队列其实是在FIFO队列的基础上作了一个增强。通常可以在 /queue 这个znode下预先建立一个/queue/num 节点，并且赋值为n（或者直接给/queue赋值n），表示队列大小，之后每次有队列成员加入后，就判断下是否已经到达队列大小，决定是否可以开始执行 了。这种用法的典型	

	场景是，分布式环境中，一个大任务Task A，需要在很多子任务完成（或条件就绪）情况下才能进行。这个时候，凡是其中一个子任务完成（就绪），那么就去 /taskList 下建立自己的临时时序节点 （CreateMode.EPHEMERAL_SEQUENTIAL），当 /taskList 发现自己下面的子节点满足指定个数，就可以进行下一步按序进行处理了。		

参考：