1.上传zk安装包

2.解压

3.配置（先在一台节点上配置）

3.1添加一个zoo.cfg配置文件

$ZOOKEEPER/conf

mv zoo\_sample.cfg zoo.cfg

3.2修改配置文件（zoo.cfg）

dataDir=/itcast/zookeeper-3.4.5/data

server.1=itcast05:2888:3888

server.2=itcast06:2888:3888

server.3=itcast07:2888:3888

3.3在（dataDir=/itcast/zookeeper-3.4.5/data）创建一个myid文件，里面内容是server.N中的N（server.2里面内容为2）

echo "1" > myid

3.4将配置好的zk拷贝到其他节点

scp -r /itcast/zookeeper-3.4.5/ itcast06:/itcast/

scp -r /itcast/zookeeper-3.4.5/ itcast07:/itcast/

3.5注意：在其他节点上一定要修改myid的内容

在itcast06应该讲myid的内容改为2 （echo "6" > myid）

在itcast07应该讲myid的内容改为3 （echo "7" > myid）

4.启动集群

分别启动zk

./zkServer.sh start

Zookeeper 配置说明

zookeeper的默认配置文件为zookeeper/conf/zoo\_sample.cfg，需要将其修改为zoo.cfg。其中各配置项的含义，解释如下：

1.tickTime：CS通信心跳时间

Zookeeper 服务器之间或客户端与服务器之间维持心跳的时间间隔，也就是每个 tickTime 时间就会发送一个心跳。tickTime以毫秒为单位。

tickTime=2000

2.initLimit：LF初始通信时限

集群中的follower服务器(F)与leader服务器(L)之间初始连接时能容忍的最多心跳数（tickTime的数量）。

initLimit=5

3.syncLimit：LF同步通信时限

集群中的follower服务器与leader服务器之间请求和应答之间能容忍的最多心跳数（tickTime的数量）。

syncLimit=2

4.dataDir：数据文件目录

Zookeeper保存数据的目录，默认情况下，Zookeeper将写数据的日志文件也保存在这个目录里。

dataDir=/home/michael/opt/zookeeper/data

5.clientPort：客户端连接端口

客户端连接 Zookeeper 服务器的端口，Zookeeper 会监听这个端口，接受客户端的访问请求。

clientPort=2181

6.服务器名称与地址：集群信息（服务器编号，服务器地址，LF通信端口，选举端口）

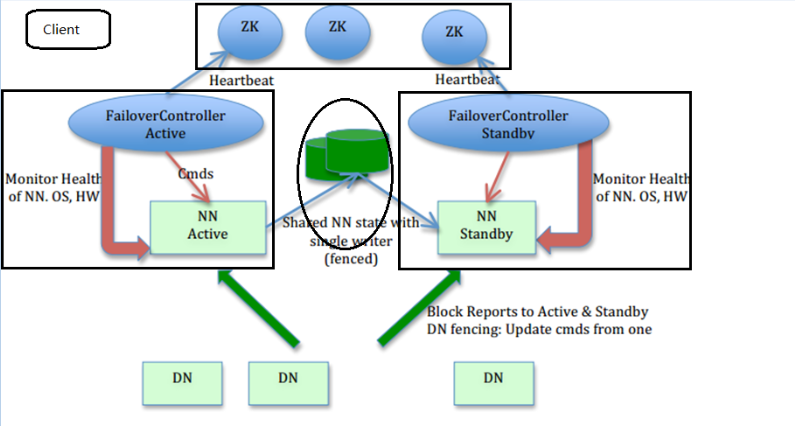
这个配置项的书写格式比较特殊，规则如下：

server.N=YYY:A:B

server.1=itcast05:2888:3888

server.2=itcast06:2888:3888

server.3=itcast07:2888:3888



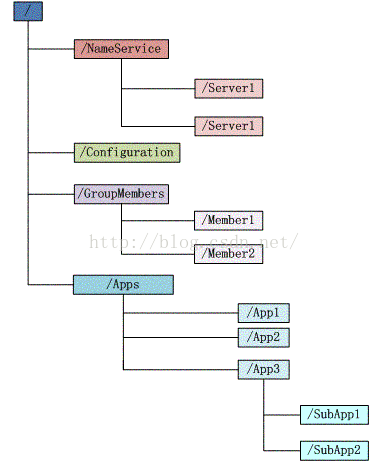
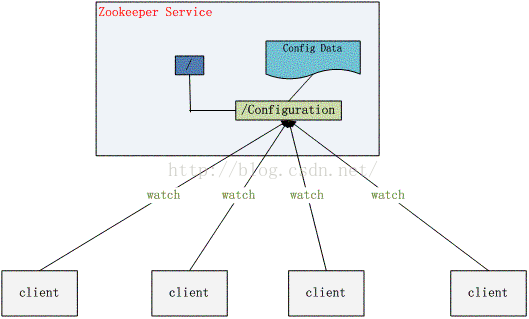
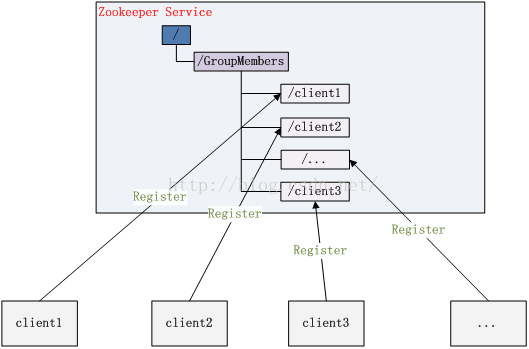
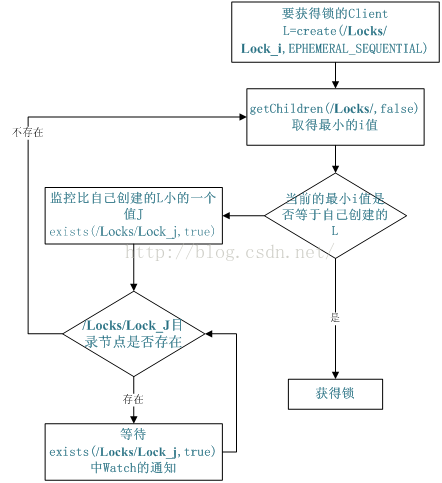
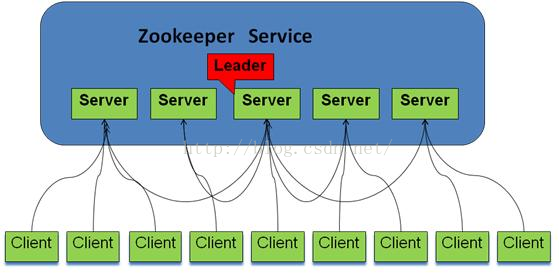
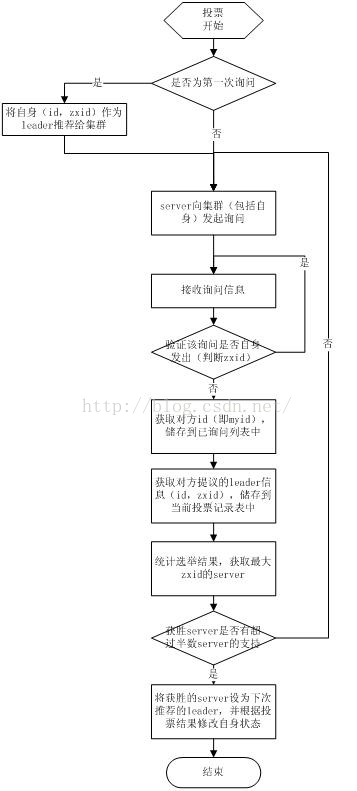
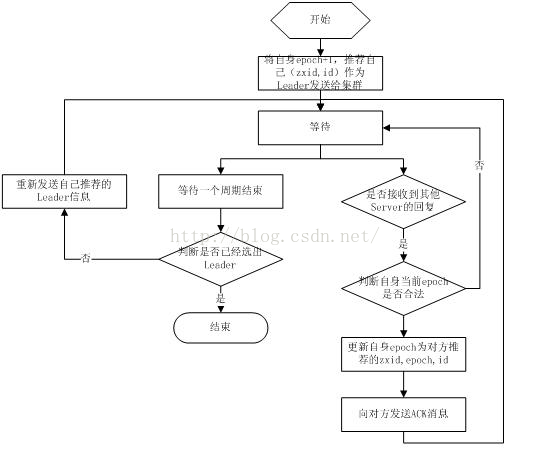
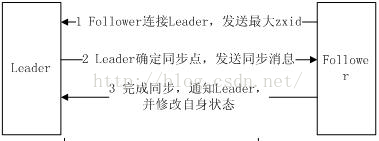
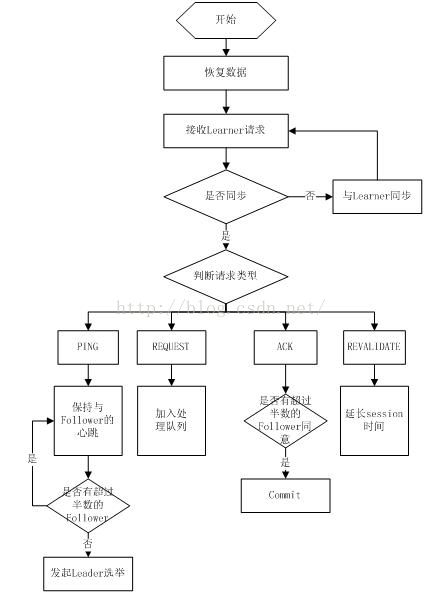
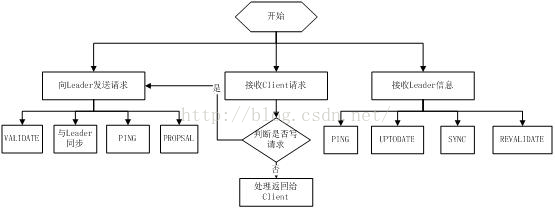
Zookeeper提供了

1)文件系统

2)通知机制

3.Zookeeper文件系统

1.ZooKeeper是什么？  
ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，是Google的Chubby一个开源的实现，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户  
  
2.ZooKeeper提供了什么？  
  
1)文件系统  
  
2)通知机制  
  
3.Zookeeper文件系统

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   
  
<ignore\_js\_op>   
  
4.Zookeeper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。  
  
5.Zookeeper做了什么？  
  
1.命名服务   2.配置管理   3.集群管理   4.分布式锁  5.队列管理  
  
6.Zookeeper命名服务  
  
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。  
  
7.Zookeeper的配置管理  
  
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。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  
  
<ignore\_js\_op>   
  
8.Zookeeper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。  
  
<ignore\_js\_op>   
  
9.Zookeeper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。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。  
  
<ignore\_js\_op>   
  
10.Zookeeper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。  
  
11.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。  
  
12.Zookeeper角色描述  
  
<ignore\_js\_op>   
  
13.Zookeeper与客户端  
  
<ignore\_js\_op>   
  
14.Zookeeper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。   
  
15.Zookeeper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。  
  
16.Zookeeper 下 Server工作状态  
  
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  
  
17.Zookeeper选主流程(basic paxos)  
  
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。  
  
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。选主的具体流程图所示：   
  
<ignore\_js\_op>   
  
18.Zookeeper选主流程（fast paxos）  
  
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。  
  
<ignore\_js\_op>   
  
19.Zookeeper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。  
  
<ignore\_js\_op>   
  
20.Zookeeper工作流程-Leader  
  
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。  
  
<ignore\_js\_op>   
  
21.Zookeeper工作流程-Follower  
  
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。  
  
<ignore\_js\_op>

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解决很多分布式问题。网上对ZK的应用场景也有不少介绍，本文将结合作者身边的项目例子，系统地对ZK的应用场景进行一个分门归类的介绍。

值得注意的是，ZK并非天生就是为这些应用场景设计的，都是后来众多开发者根据其框架的特性，利用其提供的一系列API接口（或者称为原语集），摸索出来的典型使用方法。因此，也非常欢迎读者分享你在ZK使用上的奇技淫巧。

|  |
| --- |
| **ZooKeeper典型应用场景一览** |
| **数据发布与订阅（配置中心）** |
| 发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到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，服务式服务框架的服务地址列表等就非常适合使用。 |
| * 应用中用到的一些配置信息放到ZK上进行集中管理。这类场景通常是这样：应用在启动的时候会主动来获取一次配置，同时，在节点上注册一个Watcher，这样一来，以后每次配置有更新的时候，都会实时通知到订阅的客户端，从来达到获取最新配置信息的目的。 * 分布式搜索服务中，索引的元信息和服务器集群机器的节点状态存放在ZK的一些指定节点，供各个客户端订阅使用。 * 分布式日志收集系统。这个系统的核心工作是收集分布在不同机器的日志。收集器通常是按照应用来分配收集任务单元，因此需要在ZK上创建一个以应用名作为path的节点P，并将这个应用的所有机器ip，以子节点的形式注册到节点P上，这样一来就能够实现机器变动的时候，能够实时通知到收集器调整任务分配。 * 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的发问。通常是暴露出接口，例如JMX接口，来获取一些运行时的信息。引入ZK之后，就不用自己实现一套方案了，只要将这些信息存放到指定的ZK节点上即可。   **注意**：在上面提到的应用场景中，有个默认前提是：数据量很小，但是数据更新可能会比较快的场景。 |
| **负载均衡** |
| 这里说的负载均衡是指软负载均衡。在分布式环境中，为了保证高可用性，通常同一个应用或同一个服务的提供方都会部署多份，达到对等服务。而消费者就须要在这些对等的服务器中选择一个来执行相关的业务逻辑，其中比较典型的是消息中间件中的生产者，消费者负载均衡。 |
| 消息中间件中发布者和订阅者的负载均衡，linkedin开源的KafkaMQ和阿里开源的[metaq](http://metaq.taobao.org/)都是通过zookeeper来做到生产者、消费者的负载均衡。这里以metaq为例如讲下： **生产者负载均衡**：metaq发送消息的时候，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，因此metaq在运行过程中，会把所有broker和对应的分区信息全部注册到ZK指定节点上，默认的策略是一个依次轮询的过程，生产者在通过ZK获取分区列表之后，会按照brokerId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。  **消费负载均衡：**  在消费过程中，一个消费者会消费一个或多个分区中的消息，但是一个分区只会由一个消费者来消费。MetaQ的消费策略是：   * 每个分区针对同一个group只挂载一个消费者。 * 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。   在某个消费者故障或者重启等情况下，其他消费者会感知到这一变化（通过 zookeeper watch消费者列表），然后重新进行负载均衡，保证所有的分区都有消费者进行消费。 |
| **命名服务(Naming Service)** |
| 命名服务也是分布式系统中比较常见的一类场景。在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。被命名的实体通常可以是集群中的机器，提供的服务地址，远程对象等等——这些我们都可以统称他们为名字（Name）。其中较为常见的就是一些分布式服务框架中的服务地址列表。通过调用ZK提供的创建节点的 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。 |
| 阿里巴巴集团开源的分布式服务框架Dubbo中使用ZooKeeper来作为其命名服务，维护全局的服务地址列表，[点击这里](http://code.alibabatech.com/wiki/display/dubbo/Home)查看Dubbo开源项目。在Dubbo实现中：  **服务提供者**在启动的时候，向ZK上的指定节点/dubbo/${serviceName}/providers目录下写入自己的URL地址，这个操作就完成了服务的发布。  **服务消费者**启动的时候，订阅/dubbo/${serviceName}/providers目录下的提供者URL地址， 并向/dubbo/${serviceName} /consumers目录下写入自己的URL地址。  **注意**，所有向ZK上注册的地址都是临时节点，这样就能够保证服务提供者和消费者能够自动感应资源的变化。  另外，Dubbo还有针对服务粒度的监控，方法是订阅/dubbo/${serviceName}目录下所有提供者和消费者的信息。 |
| **分布式通知/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 |
| * 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。 * 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。 * 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 |
| **集群管理与Master选举** |
| * 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：   1. 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。 2. 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。   * Master选举则是zookeeper中最为经典的应用场景了。   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。 |
| * 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。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master。 * 在Hbase中，也是使用ZooKeeper来实现动态HMaster的选举。在Hbase实现中，会在ZK上存储一些ROOT表的地址和 HMaster的地址，HRegionServer也会把自己以临时节点（Ephemeral）的方式注册到Zookeeper中，使得HMaster可以随时感知到各个HRegionServer的存活状态，同时，一旦HMaster出现问题，会重新选举出一个HMaster来运行，从而避免了 HMaster的单点问题 |
| **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。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。 |
| **分布式队列** |
| 队列方面，简单地讲有两种，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。对于第一种先进先出队列，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。 |