**Zookeeper的核心概念：**

**ZNode**

Znode就是核心结构，Zookeeper服务中是由大量的Znode构成。Znode一般是由客户端建立和修改，作为信息或标志的载体，甚至本身就是标志。

Znode可以设置为持久（PERSISTENT）或临时（EPHEMERAL），区别在于临时的节点若断开连接后就自动删除。建立节点时可选择是否使用序列号命名（SEQUENTIAL），若启用则会自动在节点名后加入唯一序列编号。

**Session**

作为客户端和Zookeeper服务之间交互的凭证。

**Watch**

当客户端对节点信息进行查询操作之后，可以选择是否设置一个Watch。其作用就是当本次查询的数据在服务器端发生变化之后，会对设置Watch的客户端发送通知。一次发送之后，就将删除该Watch，以后的变更或不再设置Watch则不会通知。

**ACLs**

节点的权限限制使用ACL，如增删改查操作。

**Zookeeper的服务器安装：**

1、下载对应版本号的tar.gz文件

2、使用 tar xzvf zookeeper-3.4.2.tar.gz -C ./ 解压

3、设置，将conf/zoo.example.cfg复制到conf/zoo.cfg或者手动建立一个新的。

4、启动Zookeeper服务：bin/zkServer.sh start

5、启动客户端连接：bin/zkCli.sh -server 127.0.0.1:2181（此处在本机，且使用了默认端口，且在Java环境中）

6、使用命令：ls、get、set等。

7、关闭Zookeeper服务：bin/zkServer.sh stop

**Zookeeper代码编写：**

代码编写部分比较简单，因为暴露的接口很少，主要复杂在于项目如何使用节点以及节点信息。

启动Zookeeper服务之后，客户端代码进行节点的增删，Watch的设置，内容的改查等。

此处建议查看官方的《Programming with ZooKeeper - A basic tutorial》部分，当中举了两个例子来模拟分布式系统的应用。

代码基本没有问题，唯一需要注意的就是：若之间按照原版进行调试时，有可能在调用

 Stat s = zk.exists(root, false);

这句代码时会出现一个异常，当中包括“KeeperErrorCode = ConnectionLoss for”。

这个问题引起的原因可以看一下代码

                System.out.println("Starting ZK:");  
                zk = new ZooKeeper(address, 3000, this);  
                mutex = new Integer(-1);  
                System.out.println("Finished starting ZK: " + zk);

最后一行有打印出Zookeeper目前的信息，若未修改的原代码，此处的State应当是CONECTING。连接中的时候去验证是否存在节点会报错。解决的方法也很简单，就是等到Zookeeper客户端以及完全连接上服务器，State为CONECTED之后再进行其他操作。给出代码示例：

// 使用了倒数计数，只需要计数一次  
private CountDownLatch connectedSignal = new CountDownLatch(1);   
SyncPrimitive(String address) {  
    if(zk == null){  
        try {  
            System.out.println("Starting ZK:");  
            zk = new ZooKeeper(address, 3000, this);  
            mutex = new Integer(-1);  
            connectedSignal.await(); // 等待连接完成  
            System.out.println("Finished starting ZK: " + zk);  
        } catch (IOException e) {  
            System.out.println(e.toString());  
            zk = null;  
        } catch (InterruptedException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
    //else mutex = new Integer(-1);  
}  
synchronized public void process(WatchedEvent event) {  
    // 此处设立在Watch中会在状态变化后触发事件  
    if (event.getState() == KeeperState.SyncConnected) {  
        connectedSignal.countDown();// 倒数-1  
    }  
      
        synchronized (mutex) {  
            //System.out.println("Process: " + event.getType());  
            mutex.notify();  
        }  
}

这样就可以正确运行代码了。

**Zookeeper的应用场景及方式：**

此处是为引用，原地址为（<http://rdc.taobao.com/team/jm/archives/1232> ）

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** |
| **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。 | 1. 索引信息和集群中机器节点状态存放在zk的一些指定节点，供各个客户端订阅使用。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。 |  |
| **集群管理** | 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。 | 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。 |  |

最近使用Zookeeper作为配置管理服务，因为配置数据有很高的安全要求，需要有权限控制，也就是需要登录才能看到Zookeeper上面的数据。   
  
Zookeeper对权限的控制是节点级别的，而且不继承，即对父节点设置权限，其子节点不继承父节点的权限。   
  
Zookeeper提供了几种认证方式   
\* world：有个单一的ID，anyone，表示任何人。   
\* auth：不使用任何ID，表示任何通过验证的用户（是通过ZK验证的用户？连接到此ZK服务器的用户？）。   
\* digest：使用 用户名：密码 字符串生成MD5哈希值作为ACL标识符ID。权限的验证通过直接发送用户名密码字符串的方式完成，   
\* ip：使用客户端主机ip地址作为一个ACL标识符，ACL表达式是以 addr/bits 这种格式表示的。ZK服务器会将addr的前bits位与客户端地址的前bits位来进行匹配验证权限。   
  
digest方式比较适合我们的业务，因此采用此种方式对Zookeeper进行权限控制。   
  
创建节点数据时：

**Java代码  [收藏代码](javascript:void())**

1. List<ACL> acls = **new** ArrayList<ACL>(2);
3. Id id1 = **new** Id("digest", DigestAuthenticationProvider.generateDigest("admin:admin123"));
4. ACL acl1 = **new** ACL(ZooDefs.Perms.ALL, id1);
6. Id id2 = **new** Id("digest", DigestAuthenticationProvider.generateDigest("guest:guest123"));
7. ACL acl2 = **new** ACL(ZooDefs.Perms.READ, id2);
9. acls.add(acl1);
10. acls.add(acl2);
12. ZooKeeper zk = **new** ZooKeeper("127.0.0.1:2181", 10000, **new** DefaultWatcher());
13. zk.create("/test", **new** **byte**[0], acls, CreateMode.PERSISTENT);

登录Zookeeper读取节点数据时：

**Java代码  [收藏代码](javascript:void())**

1. ZooKeeper zk = **new** ZooKeeper("127.0.0.1:2181", 10000, **new** DefaultWatcher());
2. zk.addAuthInfo("digest", "guest:guest123".getBytes());
3. **byte**[] value = zk.getData("/test", **null**, **new** Stat());

这样对Zookeeper上的节点数据设置多个用户，用于不同的权限操作。

老样子，先上连接，第一个是淘宝综合业务平台的（前几个月还叫通用产品团队呢我记得）

http://rdc.taobao.com/team/jm/archives/947，其实基本上需要的他这里都有了，只是……太专业了，什么都说了唯独没说要怎么用。另一个是http://ec2-23-21-252-232.compute-

1.amazonaws.com/wordpress/?p=14，写的基本上也到位了，就是格式太差，看了半天看不到重点，所以今天再写一篇不完全自主知识产权的文章，稍微总结一下zookeeper中的权限管理。

其实综合业务（每次都想打“通用产品”）的文章写的已经很具体了不过可以稍微总结一下：

先来看下ACL的数据结构：  
每一个znode节点上都可以设置一个访问控制列表，数据结构为List  
  
ACL  
+--perms int （allow What）  
+--id Id  
    +--scheme String （Who）  
    +--id String      （How）  
一个ACL对象就是一个Id和permission对，用来表示哪个/哪些范围的Id（Who）在通过了怎样的鉴权（How）  
  
之后，就允许进行那些操作（What）：Who How What；permission（What）就是一个int表示的位码，每一  
  
位代表一个对应操作的允许状态。类似unix的文件权限，不同的是共有5种操作：CREATE、READ、WRITE、  
  
DELETE、ADMIN(对应更改ACL的权限)；Id由scheme（Who）和一个具体的字符串鉴权表达式id(How)构成，用  
  
来描述哪个/哪些范围的Id应该怎样被鉴权。Scheme事实上是所使用的鉴权插件的标识。id的具体格式和语  
  
义由scheme对应的鉴权实现决定。

1、在zookeeperC的API里面，设置ACL的时候需要传入一个类型为structACL的结构指针，structACL结构类

型的定义如下：

structACL\_vector {

   int32\_t count;

   struct ACL \*data;

};

可以看出，该结构包含两个字段：一个用于记录结构中ACL结构个数的，另一个应该是一个指向ACL结构数组

的指针（数组长度就由count决定了），ACL结构的定义如下：

struct ACL {

   int32\_tperms;

   struct Idid;

};

这个结构也是两个字段：perms的意思就是permission，记录着权限，权限值从0到31说白了就是五个bit位

分别记录五种权限，从高位到低位分别是admin、delete、create、write、read，简称adcwr，权限的具体

含义不解释。另一个字段又是一个结构Id，（要处理这么样一个结构果断提防内存泄露）该结构的定义如下

：

struct Id {

   char \*scheme;

   char \* id;

};

到这里算是清楚了，里面两个字段：scheme标志着权限管理策略，目前有四个：world、auth、digest和ip

，具体不解释。Id字段是真正用来做权限匹配的，匹配成功就算你拥有权限。

整个结构综合业务那边解释的很清楚了，就不在说了，只是把具体定义给大家看看，说白了，如果要设置权

限，就要弄这么样一个结构出来，把需要的字段都填好，还要填对（比如如果scheme是“digest”的话，id

必须具有“user：password”的格式）。

2、可以设置节点权限的API主要有两个，一个是create，一个是setACL，前者是在创建的时候就设置了，后

者是另行设置，直观上来讲，后者设置能不能成功那都是不一定的，因为起码你要有a权限的。其实，前者

也不好说了，经典反例请看综合业务博客，不解释。

3、每个节点在创建的时候肯定要设置一个权限的，最经常设置的就是ZOO\_OPEN\_ACL\_UNSAFE，也就是说

adcwr= 31、scheme = “world”、 id = “anyone”。简单地说：随便哪个家伙，想干什么都行。根据综

合业务博客，在一个zookeeper session创建以后会为该session创建默认的Id，其中scheme= “ip”、 id

就是客户端的IP地址，在访问节点的时候，zookeeper内部会将这个权限信息附加在消息中发送，消息在处

理的时候会将消息中的Id信息与节点的ACL信息进行匹配，对于ZOO\_OPEN\_ACL\_UNSAFE权限的节点，任何匹配

都会是成功的。

4、除了在session创建时默认构造的Id，客户还可以指定自己的权限，这需要用到zoo\_add\_auth函数，该函

数就是用来重新声明自己的Id。比如，如果客户端要访问一个节点时发现权限不够，如果它知道节点的权限

设置，它就可以重新设置自己的Id，并达到访问节点的目的。

5、当scheme为“digest”的时候，id字段必须具有“user：password”的格式，当我们调用zoo\_add\_auth

函数时，password代表的字符串会经过SHA1加密。而调用getACL函数时我们看到的password字段代表的字符

串已经是加密后的字符串了。因此“digest”策略下，在调用setACL时，password部分的字符串必须是密文

，而调用zoo\_add\_auth函数password部分的字符串必须是明文。

用最简单的话总结，创建节点的时候会分配权限和验证Id，用户可以通过zoo\_add\_auth函数设置自己的验证

Id，同时用户可以通过setACL重设权限和验证Id，如果验证成功用户将拥有该节点赋予该验证Id的权限。