### 简介

Apache Zookeeper 是由 Apache Hadoop 的 Zookeeper 子项目发展而来，现在已经成为了 Apache 的顶级项目。Zookeeper 为分布式系统提供了高效可靠且易于使用的协同服务，它可以为分布式应用提供相当多的服务，诸如统一命名服务，配置管理，状态同步和组服务等。 Zookeeper 接口简单，开发人员不必过多地纠结在分布式系统编程难于处理的同步和一致性问题上，你可以使用 Zookeeper 提供的现成(off-the-shelf)服务来实现分布式系统的配置管理，组管理，Leader 选举等功能。

Zookeeper 维护了大规模分布式系统中的常用对象，比如配置信息，层次化命名空间等，本文将从开发者的角度详细介绍 Zookeeper 的配置信息的意义以及 Zookeeper 的典型应用场景（配置文件的管理、集群管理、分布式队列、同步锁、Leader 选举、队列管理等）。

### Zookeeper 安装与配置

本文采用 Zookeeper-3.4.0 以基础介绍它的安装步骤以及配置信息，最新的代码可以到 [Zookeeper 的官网](http://zookeeper.apache.org/)：[http://zookeeper.apache.org/下](http://zookeeper.apache.org/%C3%A4%C2%B8%C2%8B)载。Zookeeper功能强大，但是安装却十分简单，下面重点以伪分布式模式来介绍 Zookeeper 的安装。

#### 伪分布式模式安装

Zookeeper 安装模式包括：单机模式，伪分布式模式和完全的集群模式。单机模式最简单，本文将跳过单机模式安装（单机模式安装步骤参见 [Zeekeeper 官方文档](http://zookeeper.apache.org/doc/current/zookeeperStarted.html)：<http://zookeeper.apache.org/doc/current/zookeeperStarted.html>），伪分布式模式与集群模式配置差别不大，由于手头机器有限，所以本文采用了在单台机器上伪分布式安装。

本文在Ubuntu 12.04上操作，Java 环境为 OpenJDK 1.7。安装 Zookeeper 前首先下载你需要的版本，暂时解压到指定目录（本文解压至~/zookeeper/目录下），并修改配置（可能需要多次修改配置文件），本次伪分布式模拟 5 个 Zookeeper 节点，事先在/tmpzookeeper目录下建立5个文件夹，分别命名 为：server001，server002，server003，server004，server005，然后在每个server00#文件夹下面新 建 data 和 logs 子文件夹。

Zookeeper 的配置文件主要在 conf 目录，包括zoo.cfg (zoo\_sample.cfg)和log4j.properties，修改 zoo\_sample.cfg，重命名为zoo.cgf，打开zoo.cfg，内容如下：

[复制代码](javascript:void(0);)

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can pass between

# sending a request and getting an acknowledgement

syncLimit=5

# the directory where the snapshot is stored.

# do not use /tmp for storage, /tmp here is just

# example sakes.

dataDir=/tmp/zookeeper

# the port at which the clients will connect

clientPort=2181

#

# Be sure to read the maintenance section of the

# administrator guide before turning on autopurge.

#

# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc\_maintenance

#

# The number of snapshots to retain in dataDir

#autopurge.snapRetainCount=3

# Purge task interval in hours

# Set to "0" to disable auto purge feature

#autopurge.purgeInterval=1

[复制代码](javascript:void(0);)

将内容修改为（server001节点的配置文件）：

[复制代码](javascript:void(0);)

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can pass between

# sending a request and getting an acknowledgement

syncLimit=5

# the directory where the snapshot is stored.

# do not use /tmp for storage, /tmp here is just

# example sakes.

dataDir=/tmp/zookeeper/server001/data

dataLogDir=/tmp/zookeeper/server001/logs

# the port at which the clients will connect

clientPort=2181

#

# Be sure to read the maintenance section of the

# administrator guide before turning on autopurge.

#

# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc\_maintenance

#

# The number of snapshots to retain in dataDir

#autopurge.snapRetainCount=3

# Purge task interval in hours

# Set to "0" to disable auto purge feature

#autopurge.purgeInterval=1

server.1=127.0.0.1:8881:7771

server.2=127.0.0.1:8882:7772

server.3=127.0.0.1:8883:7773

server.4=127.0.0.1:8884:7774

server.5=127.0.0.1:8885:7775

[复制代码](javascript:void(0);)

* tickTime：这个时间是作为 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。
* dataDir：顾名思义就是 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。
* clientPort：这个端口就是客户端连接 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。
* initLimit：这个配置项是用来配置 Zookeeper 接受客户端（这里所说的客户端不是用户连接 Zookeeper 服务器的客户端，而是 Zookeeper 服务器集群中连接到 Leader 的 Follower 服务器）初始化连接时最长能忍受多少个心跳时间间隔数。当已经超过 5个心跳的时间（也就是 tickTime）长度后 Zookeeper 服务器还没有收到客户端的返回信息，那么表明这个客户端连接失败。总的时间长度就是 5\*2000=10 秒
* syncLimit：这个配置项标识 Leader 与 Follower 之间发送消息，请求和应答时间长度，最长不能超过多少个 tickTime 的时间长度，总的时间长度就是 2\*2000=4 秒
* server.A=B：C：D：其中 A 是一个数字，表示这个是第几号服务器；B 是这个服务器的 ip 地址；C 表示的是这个服务器与集群中的 Leader 服务器交换信息的端口；D 表示的是万一集群中的 Leader 服务器挂了，需要一个端口来重新进行选举，选出一个新的 Leader，而这个端口就是用来执行选举时服务器相互通信的端口。如果是伪集群的配置方式，由于 B 都是一样，所以不同的 Zookeeper 实例通信端口号不能一样，所以要给它们分配不同的端口号。

然后将此 zookeeper 包拷贝至 /tmp/zookeeper/server001/ 目录下，并在 /tmp/zookeeper/server001/data/ 下建立一个myid文件，文件内容为 1，

echo "1" >> /tmp/zookeeper/server001/data/myid

继续修改~/zookeeper/目录中的zookeeper配置文件文件（server002的配置文件，注意 clientPort=2182，与 server001 中的 clientPort=2181 不同，后续修改配置均需设置不同的 clientPort），内容如下：

[复制代码](javascript:void(0);)

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can pass between

# sending a request and getting an acknowledgement

syncLimit=5

# the directory where the snapshot is stored.

# do not use /tmp for storage, /tmp here is just

# example sakes.

dataDir=/tmp/zookeeper/server002/data

dataLogDir=/tmp/zookeeper/server002/logs

# the port at which the clients will connect

clientPort=2182

#

# Be sure to read the maintenance section of the

# administrator guide before turning on autopurge.

#

# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc\_maintenance

#

# The number of snapshots to retain in dataDir

#autopurge.snapRetainCount=3

# Purge task interval in hours

# Set to "0" to disable auto purge feature

#autopurge.purgeInterval=1

server.1=127.0.0.1:8881:7771

server.2=127.0.0.1:8882:7772

server.3=127.0.0.1:8883:7773

server.4=127.0.0.1:8884:7774

server.5=127.0.0.1:8885:7775

[复制代码](javascript:void(0);)

然后将此 zookeeper 包拷贝至 /tmp/zookeeper/server002/ 目录下，并在 /tmp/zookeeper/server002/data/ 下建立一个myid文件，文件内容为 2，

echo "2" >> /tmp/zookeeper/server001/data/myid

依次修改配置文件，建立 server003，server004，server005节点文件夹，完成上述步骤后/tmp/zookeeper目录结构如下：

[复制代码](javascript:void(0);)

forhappy@forhappy-lenovo:/tmp/zookeeper$ tree -d -L 2

.

├── server001

│   ├── data

│   ├── logs

│   └── zookeeper-3.4.0

├── server002

│   ├── data

│   ├── logs

│   └── zookeeper-3.4.0

├── server003

│   ├── data

│   ├── logs

│   └── zookeeper-3.4.0

├── server004

│   ├── data

│   ├── logs

│   └── zookeeper-3.4.0

└── server005

├── data

├── logs

└── zookeeper-3.4.0

[复制代码](javascript:void(0);)

然后依次进入每个文件夹节点的zookeeper目录中，启动zookeeper服务，

$ bin/zkServer.sh start

如果一切顺利，Zookeeper 伪分布式模式安装成功，下面验证 Zookeeper 安装的正确性。

进入任意一个文件夹节点的zookeeper包所在的目录，执行一下命令：

$ bin/zkCli.sh -server 127.0.0.1:2181

执行成功后：

[复制代码](javascript:void(0);)

forhappy@forhappy-lenovo:/tmp/zookeeper/server001/zookeeper-3.4.0$ bin/zkCli.sh -server 127.0.0.1:2181

Connecting to 127.0.0.1:2181

Welcome to ZooKeeper!

WATCHER::

WatchedEvent state:SyncConnected type:None path:null

[zk: 127.0.0.1:2181(CONNECTED) 0]

[复制代码](javascript:void(0);)

help 帮助：

[复制代码](javascript:void(0);)

[zk: 127.0.0.1:2181(CONNECTED) 0] help

ZooKeeper -server host:port cmd args

connect host:port

get path [watch]

ls path [watch]

set path data [version]

rmr path

delquota [-n|-b] path

quit

printwatches on|off

create [-s] [-e] path data acl

stat path [watch]

close

ls2 path [watch]

history

listquota path

setAcl path acl

getAcl path

sync path

redo cmdno

addauth scheme auth

delete path [version]

setquota -n|-b val path

[复制代码](javascript:void(0);)

至此，Zookeeper 安装完成，下一篇博客将介绍 Zookeeper Java API，并给出 Zookeeper 典型的应用场景。

**简介**

Apache Zookeeper 是由 Apache Hadoop 的 Zookeeper 子项目发展而来，现在已经成为了 Apache 的顶级项目。Zookeeper 为分布式系统提供了高效可靠且易于使用的协同服务，它可以为分布式应用提供相当多的服务，诸如统一命名服务，配置管理，状态同步和组服务等。 Zookeeper 接口简单，开发人员不必过多地纠结在分布式系统编程难于处理的同步和一致性问题上，你可以使用 Zookeeper 提供的现成(off-the-shelf)服务来实现分布式系统的配置管理，组管理，Leader 选举等功能。

Zookeeper 维护了大规模分布式系统中的常用对象，比如配置信息，层次化命名空间等，本文将从开发者的角度详细介绍 Zookeeper 的配置信息的意义以及 Zookeeper 的典型应用场景（配置文件的管理、集群管理、分布式队列、同步锁、Leader 选举、队列管理等）。

[上一篇博客](http://www.cnblogs.com/haippy/archive/2012/07/19/2599989.html)主要讲了 Apache Zookeeper 的安装与配置，本文主要介绍 Zookeeper Java API。

Zookeeper 提供了原生的Java API，另外还有C调用接口，本文暂不介绍Zookeeper 的 C API。

**Zookeeper Java API**

Zookeeper 作为一个分布式服务框架，主要用来解决分布式集群中应用系统的一致性问题，它能提供基于类似于文件系统的目录节点树方式的数据存储，但是 Zookeeper 并不是用来专门存储数据的，它的作用主要是用来维护和监控你存储的数据的状态变化。通过监控这些数据状态的变化，从而可以达到基于数据的集群管理。

Zookeeper 客户端在连接 Zookeeper 服务器需要实例化一个**org.apache.zookeeper.ZooKeeper**对象，然后调用该类提供的接口与Zookeeper服务器进行交互。如果不指明，该类的所有方法均是线程安全的。

一旦 Zookeeper 客户端与服务器建立连接，客户端就会被分配一个会话ID(session ID)，客户端会定期向服务器端发送心跳以保持该会话有效。只要客户端会话有效，应用程序可以调用Zookeeper客户端的接口与服务器端进行交互。

下表主要介绍了**org.apache.zookeeper. ZooKeeper 方法(**[**该表摘自IBM Developerworks**](http://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/)**：**[**分布式服务框架 Zookeeper -- 管理分布式环境中的数据**](http://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/)**)**

|  |  |
| --- | --- |
| **方法名** | **方法功能描述** |
| [String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) create[(String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, byte[] data, [List](http://java.sun.com/javase/6/docs/api/java/util/List.html?is-external=true)[<](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true)ACL> acl, CreateMode createMode) | 创建一个给定的目录节点 path, 并给它设置数据，[CreateMode](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/CreateMode.html) 标 识有四种形式的目录节点，分别是 PERSISTENT：持久化目录节点，这个目录节点存储的数据不会丢失；PERSISTENT\_SEQUENTIAL：顺序自动编号的目录节点，这种目 录节点会根据当前已近存在的节点数自动加 1，然后返回给客户端已经成功创建的目录节点名；EPHEMERAL：临时目录节点，一旦创建这个节点的客户端与服务器端口也就是 session 超时，这种节点会被自动删除；EPHEMERAL\_SEQUENTIAL：临时自动编号节点。 |
| [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html) [exists](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#exists%28java.lang.String,%20boolean%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path,boolean watch) | 判断某个 path 是否存在，并设置是否监控这个目录节点，这里的 watcher 是在创建 ZooKeeper 实例时指定的 watcher，[exists](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#exists%28java.lang.String,%20boolean%29)方法还有一个重载方法，可以指定特定的watcher。 |
| [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html) [exists](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#exists%28java.lang.String,%20org.apache.zookeeper.Watcher%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, [Watcher](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/Watcher.html) watcher) | 重载方法，这里给 某个目录节点设置特定的 watcher，Watcher 在 ZooKeeper 是一个核心功能，Watcher 可以监控目录节点的数据变化以及子目录的变化，一旦这些状态发生变化，服务器就会通知所有设置在这个目录节点上的 Watcher，从而每个客户端都很快知道它所关注的目录节点的状态发生变化，而做出相应的反应。 |
| void [delete](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#delete%28java.lang.String,%20int%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, int version) | 删除 path 对应的目录节点，version 为 -1 可以匹配任何版本，也就删除了这个目录节点所有数据。 |
| [List](http://java.sun.com/javase/6/docs/api/java/util/List.html?is-external=true)<[String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true)> [getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch) | 获取指定 path 下的所有子目录节点，同样 [getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)方法也有一个重载方法可以设置特定的 watcher 监控子节点的状态。 |
| [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html) [setData](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#setData%28java.lang.String,%20byte[],%20int%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, byte[] data, int version) | 给 path 设置数据，可以指定这个数据的版本号，如果 version 为 -1 怎可以匹配任何版本。 |
| byte[] [getData](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getData%28java.lang.String,%20boolean,%20org.apache.zookeeper.data.Stat%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch, [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html) stat) | 获取这个 path 对应的目录节点存储的数据，数据的版本等信息可以通过 stat 来指定，同时还可以设置是否监控这个目录节点数据的状态。 |
| void [addAuthInfo](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#addAuthInfo%28java.lang.String,%20byte[]%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) scheme,byte[] auth) | 客户端将自己的授权信息提交给服务器，服务器将根据这个授权信息验证客户端的访问权限。 |
| [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html) [setACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#setACL%28java.lang.String,%20java.util.List,%20int%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, [List](http://java.sun.com/javase/6/docs/api/java/util/List.html?is-external=true)<[ACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/ACL.html)> acl, int version) | 给某个目录节点重新设置访问权限，需要注意的是 Zookeeper 中的目录节点权限不具有传递性，父目录节点的权限不能传递给子目录节点。目录节点 ACL 由两部分组成：perms 和 id。 Perms 有 ALL、READ、WRITE、CREATE、DELETE、ADMIN 几种  而 id 标识了访问目录节点的身份列表，默认情况下有以下两种： ANYONE\_ID\_UNSAFE = new Id("world", "anyone") 和 AUTH\_IDS = new Id("auth", "") 分别表示任何人都可以访问和创建者拥有访问权限。 |
| [List](http://java.sun.com/javase/6/docs/api/java/util/List.html?is-external=true)<[ACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/ACL.html)> [getACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getACL%28java.lang.String,%20org.apache.zookeeper.data.Stat%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html) stat) | 获取某个目录节点的访问权限列表 |

除了以上这些上表中列出的方法之外还有一些重载方法，如都提供了一个回调类的重载方法以及可以设置特定 Watcher 的重载方法，具体的方法可以参考 org.apache.zookeeper. ZooKeeper 类的 API 说明。

下面给出Java API 的基本操作：

[复制代码](javascript:void(0);)

// 创建一个与服务器的连接

ZooKeeper zk = new ZooKeeper("localhost:" + CLIENT\_PORT,

ClientBase.CONNECTION\_TIMEOUT, new Watcher() {

// 监控所有被触发的事件

public void process(WatchedEvent event) {

System.out.println("已经触发了" + event.getType() + "事件！");

}

});

// 创建一个目录节点

zk.create("/testRootPath", "testRootData".getBytes(), Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

// 创建一个子目录节点

zk.create("/testRootPath/testChildPathOne", "testChildDataOne".getBytes(),

Ids.OPEN\_ACL\_UNSAFE,CreateMode.PERSISTENT);

System.out.println(new String(zk.getData("/testRootPath",false,null)));

// 取出子目录节点列表

System.out.println(zk.getChildren("/testRootPath",true));

// 修改子目录节点数据

zk.setData("/testRootPath/testChildPathOne","modifyChildDataOne".getBytes(),-1);

System.out.println("目录节点状态：["+zk.exists("/testRootPath",true)+"]");

// 创建另外一个子目录节点

zk.create("/testRootPath/testChildPathTwo", "testChildDataTwo".getBytes(),

Ids.OPEN\_ACL\_UNSAFE,CreateMode.PERSISTENT);

System.out.println(new String(zk.getData("/testRootPath/testChildPathTwo",true,null)));

// 删除子目录节点

zk.delete("/testRootPath/testChildPathTwo",-1);

zk.delete("/testRootPath/testChildPathOne",-1);

// 删除父目录节点

zk.delete("/testRootPath",-1);

// 关闭连接

zk.close();

### 简介

Apache Zookeeper 是由 Apache Hadoop 的 Zookeeper 子项目发展而来，现在已经成为了 Apache 的顶级项目。Zookeeper 为分布式系统提供了高效可靠且易于使用的协同服务，它可以为分布式应用提供相当多的服务，诸如统一命名服务，配置管理，状态同步和组服务等。 Zookeeper 接口简单，开发人员不必过多地纠结在分布式系统编程难于处理的同步和一致性问题上，你可以使用 Zookeeper 提供的现成(off-the-shelf)服务来实现分布式系统的配置管理，组管理，Leader 选举等功能。

[英文原文地址](http://zookeeper.apache.org/doc/current/javaExample.html)：<http://zookeeper.apache.org/doc/current/javaExample.html>

### 一个简单的 Zookeeper Watch 客户端

为了介绍 Zookeeper Java API 的基本用法，本文将带你如何一步一步实现一个功能简单的  Zookeeper 客户端。该 Zookeeper 客户端会监视一个你指定 Zookeeper 节点 Znode， 当被监视的节点发生变化时，客户端会启动或者停止某一程序。

### 基本要求

该客户端具备四个基本要求：

* 客户端所带参数：
  + Zookeeper 服务地址。
  + 被监视的 Znode 节点名称。
  + 可执行程序及其所带的参数
* 客户端会获取被监视 Znode 节点的数据并启动你所指定的可执行程序。
* 如果被监视的 Znode 节点发生改变，客户端重新获取其内容并再次启动你所指定的可执行程序。
* 如果被监视的 Znode 节点消失，客户端会杀死可执行程序。

### 程序设计

一般而言，Zookeeper 应用程序分为两部分，其中一部分维护与服务器端的连接，另外一部分监视 Znode 节点的数据。在本程序中，**Executor** 类负责维护 Zookeeper 连接，**DataMonitor** 类监视 Zookeeper 目录树中的数据， 同时，Executor 包含了主线程和程序主要的执行逻辑，它负责少量的用户交互，以及与可执行程序的交互，该可执行程序接受你向它传入的参数，并且会根据被监视的 Znode 节点的状态变化停止或重启。

### Executor类

Executor 对象是本例程最基本的“容器”，它包括**Zookeeper** 对象和**DataMonitor**对象。

[复制代码](javascript:void(0);)

public static void main(String[] args) {

if (args.length < 4) {

System.err

.println("USAGE: Executor hostPort znode filename program [args ...]");

System.exit(2);

}

String hostPort = args[0];

String znode = args[1];

String filename = args[2];

String exec[] = new String[args.length - 3];

System.arraycopy(args, 3, exec, 0, exec.length);

try {

new Executor(hostPort, znode, filename, exec).run();

} catch (Exception e) {

e.printStackTrace();

}

}

public Executor(String hostPort, String znode, String filename,

String exec[]) throws KeeperException, IOException {

this.filename = filename;

this.exec = exec;

zk = new ZooKeeper(hostPort, 3000, this);

dm = new DataMonitor(zk, znode, null, this);

}

public void run() {

try {

synchronized (this) {

while (!dm.dead) {

wait();

}

}

} catch (InterruptedException e) {

}

}

[复制代码](javascript:void(0);)

回忆一下 Executor 的任务是根据 Zookeeper 中 Znode 节点状态改变所触发的事件来启动和停止你在命令行指定的可执行程序， 在上面的代码你可以看到，Executor 类在其构造函数中实例化 Zookeeper 对象时，将其自身的引用作为 Watch 参数传递给 Zookeeper 的构造函数，同时它也将其自身的引用作为 DataMonitorListener 参数传递给 DataMonitor 的构造函数。Executor 本身实现了以下接口：

public class Executor implements Watcher, Runnable, DataMonitor.DataMonitorListener {

...

**Watcher** 接口是在ZooKeeper Java API 中定义的。 ZooKeeper 用它来与“容器”（此处“容器”与上面的 Executor 类相似）进行通信，Watcher 只支持一个方法，即process(), ZooKeeper 用该函数来处理主线程可能感兴趣的事件，例如 Zookeeper 连接或会话的状态，本例中的“容器” Executor只是简单地把事件向下传递给 DataMonitor，具体如何处理事件是由 DataMonitor 决定的。本文只是简单地描述了如何使用 Watcher，通常情况下，Executor 或 与 Executor 类似的对象拥有 与Zookeeper 服务端的连接，但它可以将事件传递给其他对象，并有其它的对象处理该事件。

public void process(WatchedEvent event) {

dm.process(event);

}

**DataMonitorListener** 接口本身不是Zookeeper API 的一部分，它完全是一个自定义的接口，可以说是专门为本程序设计的。DataMonitor 对象使用该接口和“容器”（即 Executor 类）进行通信，DataMonitorListener 接口如下：

[复制代码](javascript:void(0);)

public interface DataMonitorListener {

/\*\*

\* The existence status of the node has changed.

\*/

void exists(byte data[]);

/\*\*

\* The ZooKeeper session is no longer valid.

\*

\* @param rc

\* the ZooKeeper reason code

\*/

void closing(int rc);

}

[复制代码](javascript:void(0);)

该接口在 DataMonitor 中定义，Executor 类实现该接口，当 Executor.exists() 被调用的时候，Executor 决定是否启动或停止事先指定的应用程序（回忆一下前文所说的，当 Znode 消失时 Zookeeper 客户端会杀死该可执行程序）。

当 Executor.closing() 被调用的时候，Executor 会根据 Zookeeper 连接永久性地消失来决定是否关闭自己。

你或许已经猜到，DataMonitor 对象根据 Zookeeper 状态变化来调用这些方法吧？

以下是 Executor 类中实现 DataMonitorListener.exists() 和 DataMonitorListener.closing()的代码：

[复制代码](javascript:void(0);)

public void exists( byte[] data ) {

if (data == null) {

if (child != null) {

System.out.println("Killing process");

child.destroy();

try {

child.waitFor();

} catch (InterruptedException e) {

}

}

child = null;

} else {

if (child != null) {

System.out.println("Stopping child");

child.destroy();

try {

child.waitFor();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

try {

FileOutputStream fos = new FileOutputStream(filename);

fos.write(data);

fos.close();

} catch (IOException e) {

e.printStackTrace();

}

try {

System.out.println("Starting child");

child = Runtime.getRuntime().exec(exec);

new StreamWriter(child.getInputStream(), System.out);

new StreamWriter(child.getErrorStream(), System.err);

} catch (IOException e) {

e.printStackTrace();

}

}

}

public void closing(int rc) {

synchronized (this) {

notifyAll();

}

}

[复制代码](javascript:void(0);)

### DataMonitor 类

DataMonitor 类是本程序 Zookeeper 逻辑的核心， 它差不多是异步的，并由事件驱动的。DataMonitor 构造函数如下：

[复制代码](javascript:void(0);)

public DataMonitor(ZooKeeper zk, String znode, Watcher chainedWatcher,

DataMonitorListener listener) {

this.zk = zk;

this.znode = znode;

this.chainedWatcher = chainedWatcher;

this.listener = listener;

// Get things started by checking if the node exists. We are going

// to be completely event driven

zk.exists(znode, true, this, null);

}

[复制代码](javascript:void(0);)

调用 ZooKeeper.exists() 检查指定的 Znode 是否存在，并设置监视，传递自身引用作为回调对象，在某种意义上，在 watch 触发时就会引起真实的处理流程。

当 ZooKeeper.exists() 操作在服务器端完成时，ZooKeeper API 会在客户端调用 completion callback：

[复制代码](javascript:void(0);)

public void processResult(int rc, String path, Object ctx, Stat stat) {

boolean exists;

switch (rc) {

case Code.Ok:

exists = true;

break;

case Code.NoNode:

exists = false;

break;

case Code.SessionExpired:

case Code.NoAuth:

dead = true;

listener.closing(rc);

return;

default:

// Retry errors

zk.exists(znode, true, this, null);

return;

}

byte b[] = null;

if (exists) {

try {

b = zk.getData(znode, false, null);

} catch (KeeperException e) {

// We don't need to worry about recovering now. The watch

// callbacks will kick off any exception handling

e.printStackTrace();

} catch (InterruptedException e) {

return;

}

}

if ((b == null && b != prevData)

|| (b != null && !Arrays.equals(prevData, b))) {

listener.exists(b);

prevData = b;

}

}

[复制代码](javascript:void(0);)

上述代码首先检查 Znode 是否存在，以及其他重大的不可恢复的错误。如果文件（或者Znode）存在，它将从 Znode 获取数据，如果状态发生变化再调用 Executor 的 exists() 回调函数。注意，getData 函数本省必须要做任何的异常处理，因为本身就有监视可以处理任何错误：如果节点在调用 ZooKeeper.getData() 之前被删除，ZooKeeper.exists() 就会触发回调函数，如果存在通信错误，在连接上的监视会在该连接重建之前触发相应的事件，同时引发相应的处理。

最后，DataMonitor 处理监视事件的代码如下：

[复制代码](javascript:void(0);)

public void process(WatchedEvent event) {

String path = event.getPath();

if (event.getType() == Event.EventType.None) {

// We are are being told that the state of the

// connection has changed

switch (event.getState()) {

case SyncConnected:

// In this particular example we don't need to do anything

// here - watches are automatically re-registered with

// server and any watches triggered while the client was

// disconnected will be delivered (in order of course)

break;

case Expired:

// It's all over

dead = true;

listener.closing(KeeperException.Code.SessionExpired);

break;

}

} else {

if (path != null && path.equals(znode)) {

// Something has changed on the node, let's find out

zk.exists(znode, true, this, null);

}

}

if (chainedWatcher != null) {

chainedWatcher.process(event);

}

}

[复制代码](javascript:void(0);)

如果客户端 Zookeeper 程序在会话失效时(Expired event)重新建立了通信信道(SyncConnected event) ，所有的会话监视会自动和服务器进行重连， (Zookeeper 3.0.0以上版本会重置之前设置的监视). 更多编程指南请参见 [ZooKeeper Watches](http://zookeeper.apache.org/doc/r3.4.3/zookeeperProgrammers.html#ch_zkWatches) 。 当 DataMonitor 获得了指定 Znode 的事件后，它将调用 ZooKeeper.exists() 来决定究竟发生了什么。

### 完整的程序：

Executor.java：

[复制代码](javascript:void(0);)

/\*\*

\* A simple example program to use DataMonitor to start and

\* stop executables based on a znode. The program watches the

\* specified znode and saves the data that corresponds to the

\* znode in the filesystem. It also starts the specified program

\* with the specified arguments when the znode exists and kills

\* the program if the znode goes away.

\*/

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import org.apache.zookeeper.KeeperException;

import org.apache.zookeeper.WatchedEvent;

import org.apache.zookeeper.Watcher;

import org.apache.zookeeper.ZooKeeper;

public class Executor

implements Watcher, Runnable, DataMonitor.DataMonitorListener

{

String znode;

DataMonitor dm;

ZooKeeper zk;

String filename;

String exec[];

Process child;

public Executor(String hostPort, String znode, String filename,

String exec[]) throws KeeperException, IOException {

this.filename = filename;

this.exec = exec;

zk = new ZooKeeper(hostPort, 3000, this);

dm = new DataMonitor(zk, znode, null, this);

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

if (args.length < 4) {

System.err

.println("USAGE: Executor hostPort znode filename program [args ...]");

System.exit(2);

}

String hostPort = args[0];

String znode = args[1];

String filename = args[2];

String exec[] = new String[args.length - 3];

System.arraycopy(args, 3, exec, 0, exec.length);

try {

new Executor(hostPort, znode, filename, exec).run();

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* We do process any events ourselves, we just need to forward them on.

\*

\* @see org.apache.zookeeper.Watcher#process(org.apache.zookeeper.proto.WatcherEvent)

\*/

public void process(WatchedEvent event) {

dm.process(event);

}

public void run() {

try {

synchronized (this) {

while (!dm.dead) {

wait();

}

}

} catch (InterruptedException e) {

}

}

public void closing(int rc) {

synchronized (this) {

notifyAll();

}

}

static class StreamWriter extends Thread {

OutputStream os;

InputStream is;

StreamWriter(InputStream is, OutputStream os) {

this.is = is;

this.os = os;

start();

}

public void run() {

byte b[] = new byte[80];

int rc;

try {

while ((rc = is.read(b)) > 0) {

os.write(b, 0, rc);

}

} catch (IOException e) {

}

}

}

public void exists(byte[] data) {

if (data == null) {

if (child != null) {

System.out.println("Killing process");

child.destroy();

try {

child.waitFor();

} catch (InterruptedException e) {

}

}

child = null;

} else {

if (child != null) {

System.out.println("Stopping child");

child.destroy();

try {

child.waitFor();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

try {

FileOutputStream fos = new FileOutputStream(filename);

fos.write(data);

fos.close();

} catch (IOException e) {

e.printStackTrace();

}

try {

System.out.println("Starting child");

child = Runtime.getRuntime().exec(exec);

new StreamWriter(child.getInputStream(), System.out);

new StreamWriter(child.getErrorStream(), System.err);

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

[复制代码](javascript:void(0);)

DataMonitor.java：

[复制代码](javascript:void(0);)

/\*\*

\* A simple class that monitors the data and existence of a ZooKeeper

\* node. It uses asynchronous ZooKeeper APIs.

\*/

import java.util.Arrays;

import org.apache.zookeeper.KeeperException;

import org.apache.zookeeper.WatchedEvent;

import org.apache.zookeeper.Watcher;

import org.apache.zookeeper.ZooKeeper;

import org.apache.zookeeper.AsyncCallback.StatCallback;

import org.apache.zookeeper.KeeperException.Code;

import org.apache.zookeeper.data.Stat;

public class DataMonitor implements Watcher, StatCallback {

ZooKeeper zk;

String znode;

Watcher chainedWatcher;

boolean dead;

DataMonitorListener listener;

byte prevData[];

public DataMonitor(ZooKeeper zk, String znode, Watcher chainedWatcher,

DataMonitorListener listener) {

this.zk = zk;

this.znode = znode;

this.chainedWatcher = chainedWatcher;

this.listener = listener;

// Get things started by checking if the node exists. We are going

// to be completely event driven

zk.exists(znode, true, this, null);

}

/\*\*

\* Other classes use the DataMonitor by implementing this method

\*/

public interface DataMonitorListener {

/\*\*

\* The existence status of the node has changed.

\*/

void exists(byte data[]);

/\*\*

\* The ZooKeeper session is no longer valid.

\*

\* @param rc

\* the ZooKeeper reason code

\*/

void closing(int rc);

}

public void process(WatchedEvent event) {

String path = event.getPath();

if (event.getType() == Event.EventType.None) {

// We are are being told that the state of the

// connection has changed

switch (event.getState()) {

case SyncConnected:

// In this particular example we don't need to do anything

// here - watches are automatically re-registered with

// server and any watches triggered while the client was

// disconnected will be delivered (in order of course)

break;

case Expired:

// It's all over

dead = true;

listener.closing(KeeperException.Code.SessionExpired);

break;

}

} else {

if (path != null && path.equals(znode)) {

// Something has changed on the node, let's find out

zk.exists(znode, true, this, null);

}

}

if (chainedWatcher != null) {

chainedWatcher.process(event);

}

}

public void processResult(int rc, String path, Object ctx, Stat stat) {

boolean exists;

switch (rc) {

case Code.Ok:

exists = true;

break;

case Code.NoNode:

exists = false;

break;

case Code.SessionExpired:

case Code.NoAuth:

dead = true;

listener.closing(rc);

return;

default:

// Retry errors

zk.exists(znode, true, this, null);

return;

}

byte b[] = null;

if (exists) {

try {

b = zk.getData(znode, false, null);

} catch (KeeperException e) {

// We don't need to worry about recovering now. The watch

// callbacks will kick off any exception handling

e.printStackTrace();

} catch (InterruptedException e) {

return;

}

}

if ((b == null && b != prevData)

|| (b != null && !Arrays.equals(prevData, b))) {

listener.exists(b);

prevData = b;

}

}

}

[复制代码](javascript:void(0);)

### ZooKeeper 典型的应用场景

Zookeeper 从设计模式角度来看，是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的注册，一旦这些数据的状态发生 变化，Zookeeper 就将负责通知已经在 Zookeeper 上注册的那些观察者做出相应的反应，从而实现集群中类似 Master/Slave 管理模式，关于 Zookeeper 的详细架构等内部细节可以阅读 Zookeeper 的源码

下面详细介绍这些典型的应用场景，也就是 Zookeeper 到底能帮我们解决那些问题？

### 统一命名服务（Name Service）

分布式应用中，通常需要有一套完整的命名规则，既能够产生唯一的名称又便于人识别和记住，通常情况下用树形的名称结构是一个理想的选择，树形的名称 结构是一个有层次的目录结构，既对人友好又不会重复。说到这里你可能想到了 JNDI，没错 Zookeeper 的 Name Service 与 JNDI 能够完成的功能是差不多的，它们都是将有层次的目录结构关联到一定资源上，但是 Zookeeper 的 Name Service 更加是广泛意义上的关联，也许你并不需要将名称关联到特定资源上，你可能只需要一个不会重复名称，就像数据库中产生一个唯一的数字主键一样。

Name Service 已经是 Zookeeper 内置的功能，你只要调用 Zookeeper 的 API 就能实现。如调用 create 接口就可以很容易创建一个目录节点。

### 配置管理（Configuration Management）

配置的管理在分布式应用环境中很常见，例如同一个应用系统需要多台 PC Server 运行，但是它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的配置项，那么就必须同时修改每台运行这个应用系统的 PC Server，这样非常麻烦而且容易出错。

像这样的配置信息完全可以交给 Zookeeper 来管理，将配置信息保存在 Zookeeper 的某个目录节点中，然后将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。

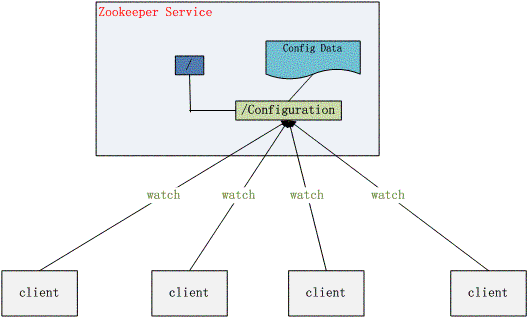


图 1. 配置管理结构图

### 集群管理（Group Membership）

Zookeeper 能够很容易的实现集群管理的功能，如有多台 Server 组成一个服务集群，那么必须要一个“总管”知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，集群中其它集群必须知道，从而做出调整重新分配服务策略。同样当增加集群的服务能力时，就会增加一台或多台 Server，同样也必须让“总管”知道。

Zookeeper 不仅能够帮你维护当前的集群中机器的服务状态，而且能够帮你选出一个“总管”，让这个总管来管理集群，这就是 Zookeeper 的另一个功能 Leader Election。

它们的实现方式都是在 Zookeeper 上创建一个 EPHEMERAL 类型的目录节点，然后每个 Server 在它们创建目录节点的父目录节点上调用 [getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch) 方法并设置 watch 为 true，由于是 EPHEMERAL 目录节点，当创建它的 Server 死去，这个目录节点也随之被删除，所以 Children 将会变化，这时 [getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)上的 Watch 将会被调用，所以其它 Server 就知道已经有某台 Server 死去了。新增 Server 也是同样的原理。

Zookeeper 如何实现 Leader Election，也就是选出一个 Master Server。和前面的一样每台 Server 创建一个 EPHEMERAL 目录节点，不同的是它还是一个 SEQUENTIAL 目录节点，所以它是个 EPHEMERAL\_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL\_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号，我们可以选择当前是最小编号的 Server 为 Master，假如这个最小编号的 Server 死去，由于是 EPHEMERAL 节点，死去的 Server 对应的节点也被删除，所以当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免了传统意义上单 Master 容易出现单点故障的问题。

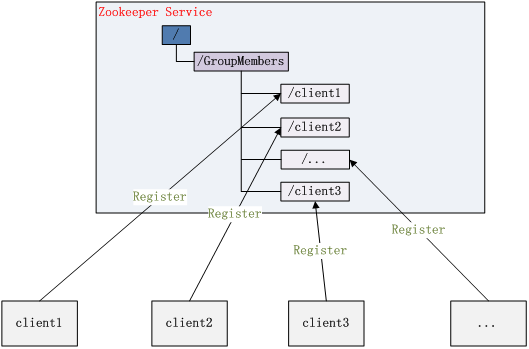


图 2. 集群管

这部分的示例代码如下，

[复制代码](javascript:void(0);)

void findLeader() throws InterruptedException {

byte[] leader = null;

try {

leader = zk.getData(root + "/leader", true, null);

} catch (Exception e) {

logger.error(e);

}

if (leader != null) {

following();

} else {

String newLeader = null;

try {

byte[] localhost = InetAddress.getLocalHost().getAddress();

newLeader = zk.create(root + "/leader", localhost,

ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

} catch (Exception e) {

logger.error(e);

}

if (newLeader != null) {

leading();

} else {

mutex.wait();

}

}

}

[复制代码](javascript:void(0);)

共享锁（Locks）

共享锁在同一个进程中很容易实现，但是在跨进程或者在不同 Server 之间就不好实现了。Zookeeper 却很容易实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL\_SEQUENTIAL 目录节点，然后调用 [getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)方法获取当前的目录节点列表中最小的目录节点是不是就是自己创建的目录节点，如果正是自己创建的，那么它就获得了这个锁，如果不是那么它就调用 [exists](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#exists%28java.lang.String,%20boolean%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch) 方法并监控 Zookeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，从而获得锁，释放锁很简单，只要删除前面它自己所创建的目录节点就行了。

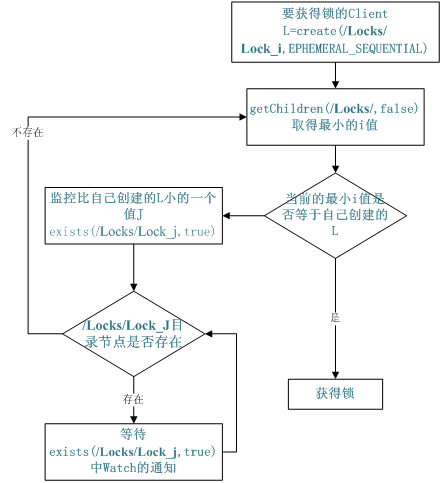


图 3. Zookeeper 实现 Locks 的流程图

同步锁的实现代码如下，同步锁的关键代码：

[复制代码](javascript:void(0);)

void getLock() throws KeeperException, InterruptedException{

List<String> list = zk.getChildren(root, false);

String[] nodes = list.toArray(new String[list.size()]);

Arrays.sort(nodes);

if(myZnode.equals(root+"/"+nodes[0])){

doAction();

}

else{

waitForLock(nodes[0]);

}

}

void waitForLock(String lower) throws InterruptedException, KeeperException {

Stat stat = zk.exists(root + "/" + lower,true);

if(stat != null){

mutex.wait();

}

else{

getLock();

}

}

[复制代码](javascript:void(0);)

### 队列管理

Zookeeper 可以处理两种类型的队列：

1. 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，这种是同步队列。
2. 队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

同步队列用 Zookeeper 实现的实现思路如下：

创建一个父目录 /synchronizing，每个成员都监控标志（Set Watch）位目录 /synchronizing/start 是否存在，然后每个成员都加入这个队列，加入队列的方式就是创建 /synchronizing/member\_i 的临时目录节点，然后每个成员获取 / synchronizing 目录的所有目录节点，也就是 member\_i。判断 i 的值是否已经是成员的个数，如果小于成员个数等待 /synchronizing/start 的出现，如果已经相等就创建 /synchronizing/start。

用下面的流程图更容易理解：

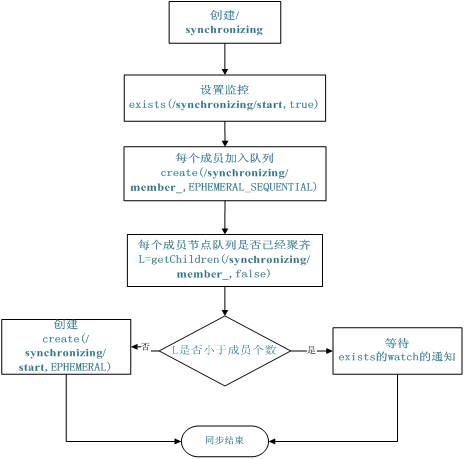


图 4. 同步队列流程图

同步队列的关键代码如下：

[复制代码](javascript:void(0);)

void addQueue() throws KeeperException, InterruptedException{

zk.exists(root + "/start",true);

zk.create(root + "/" + name, new byte[0], Ids.OPEN\_ACL\_UNSAFE,

CreateMode.EPHEMERAL\_SEQUENTIAL);

synchronized (mutex) {

List<String> list = zk.getChildren(root, false);

if (list.size() < size) {

mutex.wait();

} else {

zk.create(root + "/start", new byte[0], Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

}

}

[复制代码](javascript:void(0);)

当队列没满是进入 wait()，然后会一直等待 Watch 的通知，Watch 的代码如下：

[复制代码](javascript:void(0);)

public void process(WatchedEvent event) {

if(event.getPath().equals(root + "/start") &&

event.getType() == Event.EventType.NodeCreated){

System.out.println("得到通知");

super.process(event);

doAction();

}

}

[复制代码](javascript:void(0);)

FIFO 队列用 Zookeeper 实现思路如下：

实现的思路也非常简单，就是在特定的目录下创建 SEQUENTIAL 类型的子目录 /queue\_i，这样就能保证所有成员加入队列时都是有编号的，出队列时通过 getChildren( ) 方法可以返回当前所有的队列中的元素，然后消费其中最小的一个，这样就能保证 FIFO。

下面是生产者和消费者这种队列形式的示例代码，

生产者代码：

[复制代码](javascript:void(0);)

boolean produce(int i) throws KeeperException, InterruptedException{

ByteBuffer b = ByteBuffer.allocate(4);

byte[] value;

b.putInt(i);

value = b.array();

zk.create(root + "/element", value, ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT\_SEQUENTIAL);

return true;

}

[复制代码](javascript:void(0);)

消费者代码：

[复制代码](javascript:void(0);)

int consume() throws KeeperException, InterruptedException{

int retvalue = -1;

Stat stat = null;

while (true) {

synchronized (mutex) {

List<String> list = zk.getChildren(root, true);

if (list.size() == 0) {

mutex.wait();

} else {

Integer min = new Integer(list.get(0).substring(7));

for(String s : list){

Integer tempValue = new Integer(s.substring(7));

if(tempValue < min) min = tempValue;

}

byte[] b = zk.getData(root + "/element" + min,false, stat);

zk.delete(root + "/element" + min, 0);

ByteBuffer buffer = ByteBuffer.wrap(b);

retvalue = buffer.getInt();

return retvalue;

}

}

}

}

[复制代码](javascript:void(0);)

总结

Zookeeper 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。

本文介绍的 Zookeeper 的基本知识，以及介绍了几个典型的应用场景。这些都是 Zookeeper 的基本功能，最重要的是 Zoopkeeper 提供了一套很好的分布式集群管理的机制，就是它这种基于层次型的目录树的数据结构，并对树中的节点进行有效管理，从而可以设计出多种多样的分布式的数据管 理模型，而不仅仅局限于上面提到的几个常用应用场景。

**阅读指南——如何利用 Zookeeper 构建上层应用？**

本文将带你如何利用 Zookeeper 实现某些分布式应用所必需的高级功能。所有功能均可以在客户端按固定的模式实现，不需要 Zookeeper 的特殊支持，也希望 Zookeeper 社区能将这些具有固定实现模式的功能集成到 Zookeeper 客户端的程序库中，可以简化 Zookeeper 的使用并且还能使某些功能的实现标准化。

即便 Zookeeper 本身使用异步通知（*asynchronous* notifications），但却可以基于此构建同步的（*synchronous*）一致性原语，如队列和锁。你将看到 Zookeeper 实现这些功能是完全可能的，因为 Zookeeper 提供了强制的全序更新，并对外提供了保序接口。

注意下面的程序段试图采取最佳的编程实践，尤其是避免使用轮询（polling），定时器（timers）和其他任何可能造成“羊群效应（herd effect）”机制（“羊群效应”一般会带来网络流量的突增，限制系统的可扩展性）。

除了本文所列举的功能，我们还可以想象出其他很多实用的功能，比如可撤销的读写优先锁。本文提到的某些构建方式——比如锁，比较详细的阐述了使用 Zookepper 的关键点，其实你可以找到其他的例子，如事件处理和队列，但是，本节中的例子只是模拟相关的功能，在具体实践中需要考虑其他方面的因素。

**开箱即用的应用示例：命名服务，配置管理，组关系管理**

命名服务和配置管理是 Zookeeper 提供的最基本的应用，这两个功能可以直接用 Zookeeper 提供的 API 实现。

另外一个可以直接使用的功能是组关系管理，组在 Zookeeper 由一个 Znode 表示，组中的某个成员可以用组节点下的临时节点（Ephemeral Nodes）表示，当 Zookeeper 检测到节点故障时，节点成员中不正常的节点将会被自动地移除。

**屏障（Barriers）**

分布式系统使用屏障（*barriers*）来阻塞某一节点集的任务，直到满足特定的条件该节点集的所有节点才能正常向前推进，就像 屏障一样，在当条件不满足时，屏障阻塞了任务的执行，只有当条件满足后屏障才会被拆除，各节点才能推进自己正在执行的任务。Zookeeper 中实现屏障时指定一个屏障节点（barrier node），如果屏障节点存在，屏障就会生效，下面是伪代码：

1. 客户端在屏障节点上调用 ZooKeeper API  **exists()，***watch 设置为 true*.
2. 如果 **exists()** 返回 false，屏障消失，客户端可以推进的自己的工作。
3. 否则, **exists()** 返回 true，客户端等待屏障节点上监听事件的到来。
4. 如果监听事件被触发，客户端重新执行 **exists( )，** 再一次重复上述 1-3 步，直到屏障节点被移除。

**（双屏障）Double Barriers**

双屏障（Double barriers）使得所有客户端在进入和结束某一计算任务时都会得到同步。当足够的进程processes（注：此处指节点）加入到屏障时，才启动任 务，然后当任务完成时，离开屏障区，下面的代码段示意如何使用 Zookeeper 创建屏障节点。

伪代码中屏障节点用 *b* 表示，每个客户端进程（节点）  *p* 在进入屏障节点时注册事件，然后在离开时取消注册事件。进入屏障节点注册事件的代码如下表的 **Enter 程序段所示，** 在继续处理任务之前，它将等待客户端 *x* 进程的注册。（此处的 *x* 由你针对自己的系统决定）

|  |  |
| --- | --- |
| **Enter** | **Leave** |
| 1. 创建名称为 *b+“/”+p 的 Znode 节点* 2. 设置监视：**exists(*b* + ‘‘/ready’’, true)** 3. 创建子节点：**create( *n*, EPHEMERAL)** 4. **L = getChildren(b, false)** 5. 如果L的孩子数目小于 *x 的*, 则等待监视事件 6. 否则 **create(b + ‘‘/ready’’, REGULAR)** | 1. **L = getChildren(b, false)** 2. 如果没有任何子节点，则退出。 3. 如果 *p* 是 L 中唯一节点，则 delete(n) 并退出。 4. 如果 *p 是 L 中的序号最低的节点，则等待 P 中的序号最高节点。* 5. 否则 依然存在，则 **delete(*n*)** 并继续等待L中的最低节点。 6. 跳转 1 |

在进入屏障时，所有的进程（节点）监视一个准备好的节点（屏障节点），并创建一个临时节点作为屏障节点的孩子。除了最后进入屏障的节点外，每个进程 （节点）都等待屏障节点，直到第 5 行的条件出现。该进程（节点）创建第 x 个节点——即最后的进程（节点），它将会看到 x 个节点，并唤醒其他进程（节点），注意，所有的等待进程（节点）只是在退出的时候被唤醒，所以等待还是很高效的。

在退出屏障时，你不能设置 诸如 *ready* 的标志，因为你在等待进程节点退出，通过使用临时节点，进入屏障后失效的进程节点并不会阻止其他运行正确的节点完成任务。当进程节点准备推出屏障区时，它必须删除它的进程节点，并等待其他进程删除各自的进程节点。

当 b 没有的进程子节点时，进程（节点）就会退出屏障区。然而，为了效率起见，你可以使用序号最低的进程节点作为 ready 标志。所有其他准备退出屏障区的进程（节点）都监视序号最低的将要退出进程（节点）消失，序号最低的进程节点的拥有者则就等待其他任何一个节点的消失（选 择序号最高进程节点）。这意味着除了最后的一个进程节点外，其他的每个进程节点被删除时只要唤醒一个进程节点即可，当它被删除时就会唤醒其他的进程节点。

**队列（Queues）**

分布式队列是通用的数据结构，为了在 Zookeeper 中实现分布式队列，首先需要指定一个 Znode 节点作为队列节点（queue node）， 各个分布式客户端通过调用 create() 函数向队列中放入数据，调用create()时节点路径名带"queue-"结尾，并设置顺序和临时（*sequence* and *ephemeral*）节点标志。 由于设置了节点的顺序标志，新的路径名具有以下字符串模式："\_path-to-queue-node\_/queue-X"，X 是唯一自增号。需要从队列中移除数据的客户端首先调用 **getChildren( )** 函数，同时在队列节点（queue node）上将 *watch* 设置为 true，并处理最小序号的节点（即从序号最小的节点中取数据）。客户端不需要再一次调用 **getChildren( )，**队列中的数据获取完。如果队列节点中没有任何子节点，读取队列的客户端需要等待队列的监视事件通知。

**Priority Queues**

为了实现优先队列，你在普通队列上只需要简单的改变两处地方，首先，在某一元素被加入队列时，路径名以 "queue-YY" 结尾，YY 表示优先级，YY越小优先级越高，其次，从队列中移除一个元素时，客户端需要使用最新的孩子节点列表，这意味着如果队列节点上监视通知被触发，客户端需要 让先前获取的孩子节点列表无效。

**锁（Locks）**

完全分布式锁是全局同步的，这意味着在任何时刻没有两个客户端会同时认为它们都拥有相同的锁，使用 Zookeeper 可以实现分布式锁，和优先队列一样，我们需要首先定义一个锁节点（lock node）。

需要获得锁的客户端按照以下步骤来获取锁：

1. 调用 **create( )，**参数 pathname 为 "\_locknode\_/lock-"，并设置 *sequence* 和 *ephemeral* 标志。
2. 在所节点（lock node）上调用 **getChildren( )** ，不需要设置监视标志。 (为了避免“羊群效应”).
3. 如果在第 1 步中创建的节点的路径具有最小的序号后缀，那么该客户端就获得了锁。
4. 客户端调用 **exists( )** ，并在锁目录路径中下一个最小序号的节点上设置监视标志。
5. 如果 **exists( )** 返回 false，跳转至第 2 步，否则，在跳转至第 2 步之前等待前一部路径上节点的通知消息。

解锁协议非常简单：需要释放锁的客户端只需要删除在第 1 步中创建的节点即可。

注意事项：

* 一个节点的删除只会导致一个客户端被唤醒，因为每个节点只被一个客户端监视，这避免了羊群效应。
* 没有轮询和超时。
* 根据你实现锁的方式不同，不同的实现可能会带来大量的锁竞争，锁中断，调试锁等等。

**Shared Locks**

在基本的锁协议之上，你只需要做一些小的改变就可以实现共享锁（shared locks）：

|  |  |
| --- | --- |
| **获取读锁：** | **获取写锁：** |
| 1. Call **create( )** to create a node with pathname "\_locknode\_/read-". This is the lock node use later in the protocol. Make sure to set both the *sequence* and *ephemeral* flags. 2. Call **getChildren( )** on the lock node *without* setting the *watch* flag - this is important, as it avoids the herd effect. 3. If there are no children with a pathname starting with "write-" and having a lower sequence number than the node created in step **1**, the client has the lock and can exit the protocol. 4. Otherwise, call **exists( )**, with *watch* flag, set on the node in lock directory with pathname staring with "write-" having the next lowest sequence number. 5. If **exists( )** returns *false*, goto step **2**. 6. Otherwise, wait for a notification for the pathname from the previous step before going to step **2** | 1. Call **create( )** to create a node with pathname "\_locknode\_/write-". This is the lock node spoken of later in the protocol. Make sure to set both *sequence* and *ephemeral* flags. 2. Call **getChildren( )** on the lock node *without* setting the *watch* flag - this is important, as it avoids the herd effect. 3. If there are no children with a lower sequence number than the node created in step **1**, the client has the lock and the client exits the protocol. 4. Call **exists( ),** with *watch* flag set, on the node with the pathname that has the next lowest sequence number. 5. If **exists( )** returns *false*, goto step **2**. Otherwise, wait for a notification for the pathname from the previous step before going to step **2**. |

**Recoverable Shared Locks**

对共享锁做一些细小的改变，我们就可以使共享锁变成可撤销的共享锁：

在第 1 步，在获取读者和写者的锁协议中，在调用 **create( ) 后，**立即调用**getData( )**，并设置监视。如果客户端稍后收到了它在第一步创建节点的通知，它会再一次在该节点上调用 **getData( )**，并设置监视，查找 “unlock” 串。该信号会通知客户端必须释放锁。这是因为，依据共享锁协议，你可以通过在锁节点（lock node）上调用**setData()（将“unlock”写入该节点）**请求拥有该锁的客户端放弃该锁 。

注意该协议要求锁的拥有者也同意释放该锁，该协定非常重要，尤其是锁的拥有者需要在释放该锁前做一些处理。 当然，你也可以通过约定“撤销者可以在锁的拥有者一段时间没有删除该锁的情况下删除该锁节点”来实现可撤销的共享锁。

**两阶段提交（Two-phased Commit）**

两阶段提交协议可以让分布式系统的所有客户端决定究竟提交某一事务或还是终止该事务。

在 Zookeeper 中，你可以让协调者（coordinator）创建事务节点，比如，"/app/Tx"，从而实现一个两阶段提交协议。 当协调者（coordinator）创建了子节点时，子节点内容是未定义的，由于每个事务参与方都会从协调者接收事务，参与方读取每个子节点并设置监视。 然后每个参与方通过向与自身相关的 Znode 节点写入数据来投票“提交（commit）”或“中止（abort）”事务。一旦写入完成，其他的参与方会被通知到，当所有的参与方都投完票后，协调者就 可以决定究竟是“提交（commit）”或“中止（abort）”事务。注意，如果某些参与方投票“中止”，节点是可以决定提前“中止”事务的。

该实现方法有趣的地方在于协调者的唯一作用是决定参与方的组（the group of sites），创建 Zookeeper 节点， 将事务传播到相应的参与方，实际上，Zookeeper 可以通过将消息写入事务节点来传播事务。

上述讨论的方法存在两个明显的缺点，一是消息的复杂性，复杂度为 O(n²)，另外一个是仅通过临时节点不能判断某些参与方是否失效，为了利用临时节点检测参与方是否失效，必须参与方创建该节点。

为了解决第一个问题，你可以将系统设置成只有一个协调者可以收到事务节点状态的变化，一旦协调者达成意见后通知其他参与方， 该方法可扩展性较强，但是速度很慢，因为所有的通信都指向协调者。

为了解决第二个问题，你可以让参与方把事务传播到参与方，并让每个参与方创建自己的临时节点。

**Leader 选举（Leader Election）**

Zookeeper 实现 Leader 选举简单做法是在创建代表 “proposals” 客户端的 Znode 节点时设置 **SEQUENCE|EPHEMERAL** 标志。基本想法是创建一个节点，比如 "/election"，然后在创建子节点时"/election/n\_"设置标志 SEQUENCE|EPHEMERAL. 当设置顺序节点**SEQUENCE**标志时，Zookeeper 会在 "/election" 子节点的创建过程中自增子节点名称后缀的序号，最小后缀序号的 Znode 节点表示Leader。

然而，还没完，监视 Leader 失效也是非常重要的，当前的 Leader 失效后需要一个新的客户端起来接替旧的 Leader 的位置。一个简单的方式是让所有的应用进程监视当前序号最小的 Znode 节点， 并在当前 序号最小的 Znode 节点失效是检查他们是否为新的 Leader（注意当前序号最小的节点可能会随着 Leader 的消失而消失，他们可能是该Leader 节点的临时子节点）. 但是这会导致'羊群效应(herd effect)"：在当前 Leader 失效后，其他所有的进程（节点）将会收到通知，并在 "/election" 节点上执行 getChildren()来获取"/election"节点的子节点列表，如果客户端数目很大，它会使得Zookeeper服务器处理的操作次数急剧上 升。为了避免羊群效应，客户端只需要监视 Znode 节点中的下一个节点就足够。如果某个客户端收到了它正在监视的节点消失的通知，它将成为新的 Leader，因为此时没有其它的 Znode 节点的序号比它小。所以这就避免了羊群效应，并且客户端也没有必要监视同一个最小的 Znode 节点。

以下是伪代码：

假设 ELECTION 成为Leader 选举应用的路径，对于想要成为 Leader 的 Volunteer而言：

1. 创建 Znode 节点 z，路径名称为"ELECTION/n\_"并设置 SEQUENCE 和 EPHEMERAL 标志。
2. 假设 C 是"ELECTION"的子节点集合，  i 是 z 节点的序号。
3. 监视节点 "ELECTION/n\_j" 的改变，j 是满足 j < i 最小的序号，n\_j 是 C 节点集合中的某个节点。

当收到 Znode 节点删除的通知时：

1. 假设 C 是 “ELECTION” 新的子节点集合。
2. 如果 z 是 C 中的最小节点，则执行 Leader 选举流程。
3. 否则，监视节点 "ELECTION/n\_j" 的改变，j 是满足 j < i 最小的序号，n\_j 是 C 节点集合中的某个节点。

注意，在子节点列表中没有先遣节点的 Znode 并不意味着该节点的创建者知道它就是当前的Leader，应用程序可能需要考虑创建一个单独的 Znode 来确认该 Leader 已经执行了选举流程。