Zookeeper Watcher

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和协调，实现对数据变更的实时处理，使用方法是不同系统都对ZK上同一个znode进行注册，监听znode的变化（包括znode本身内容和子节点），其中一个系统更新znode，那么另外一个系统能够收到通知，并作出相应处理。

1. Watcher接口类

在Zookeeper中，接口类Watcher定义了事件通知相关的逻辑，包含了KeeperState和EventType两个枚举类，分别代表通知状态和事件类型，接口中比较重要的方法：

abstract public void process(WatchedEvent event)

这个方法，用于处理事件通知，每个实现类都应该实现合适的处理逻辑。参数WatchedEvent封装了KeeperState及EventType两个枚举类，以及触发事件对应的ZK节点的path。

1. Watcher的注册

Watcher接口提供了基本的回调方法用于处理来自服务器的通知，只要实现这个接口，并传给服务器即可，Zookeeper的构造方法：

ZooKeeper(String connectString, int sessionTimeout, Watcher watcher)

第三个参数是Watcher，构造中将Watcher注册到Zookeeper中，该Watcher会成为当前ZK session的默认Watcher实现。在构造方法中：

watchManager.defaultWatcher = watcher;

WatchManager管理及处理ClientCnxn object的事件。

另外一个注册的地方是接口方法中：

public Stat exists(String path, boolean watch)

public List<String> getChildren(String path, boolean watch)

public byte[] getData(String path,boolean watch,Stat stat)

public void register(Watcher watcher)

以getChildren为例，源码如下：

public List<String> getChildren(final String path, Watcher watcher){

.................................

if (watcher != null) {

wcb = new ChildWatchRegistration(watcher, clientPath);

}.......................

ReplyHeader r = cnxn.submitRequest(h, request, response, wcb);

....................................

}

当getchildren调用成功，Watcher会在访问的路径上设置该Watch，如果对该path进行设置data或者删除data那么，会触发该Watcher操作。ClentCnxn.submitRequest提交请求后，将Request对象加入到toProcess队列中，在接下来的循环中会调用FinalRequestProcessor#pro

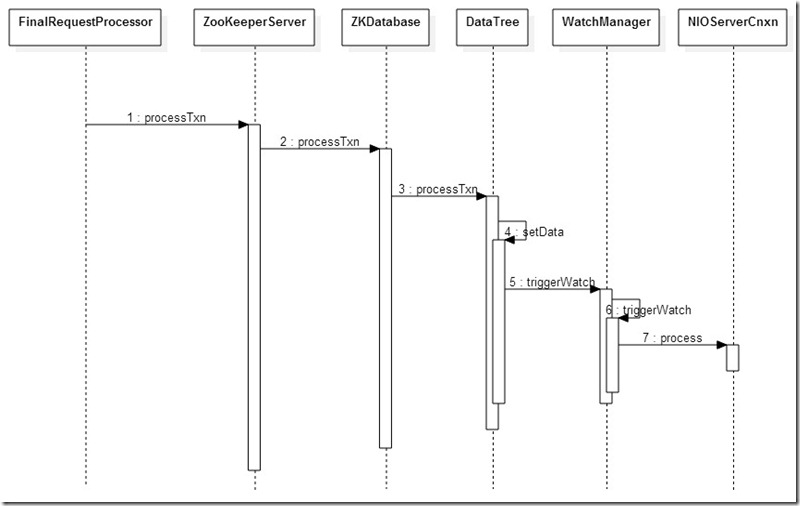
cessRequest方法进行处理。在FinalRequestProcessor中，最终会调用ZKDatabase的方法获取znode信息（DataTree.getData），同时会设置watcher。

1. Event的通知

在2.2.2中，如果是setData修改数据请求，DataTree.setData方法将txn应用到znode上，同时触发Wathcher，并发送notification给Client端。DataTree.setData中源码如下所示：

dataWatches.triggerWatch(path, EventType.NodeDataChanged);

其setData请求的时序图如下所示：



注：http://www.linuxidc.com/Linux/2015-02/113899.htm

1. Watcher Example

下面是官网给出的一个Watcher使用案例：

http://zookeeper.apache.org/doc/trunk/javaExample.html

Zookeeper应用程序分成两部分，Executor类维护Zookeeper连接，其包含了主线程和程序的主要执行逻辑，负责用户以及可执行程序的交互，可执行程序根据被监视的Znode节点的状态变化停止或者重启。DataMonitor监视Zookeeper目录树中的数据，根据ZNode节点的变化进行相应的操作。之间的关系如下图所示：



Executor的任务时根据Zookeeper中Znode节点状态改变所触发的事件来启动和停止在命令行中指定的可执行程序。Executor在构造中，实例化Zookeeper对象，并将自身引用作为Watch参数传递给Zookeeper的构造函数，同时将自身引用作为DataMonitorListener参数传递给DataMonitor的构造函数。

Executor继承Watcher接口，Zookeeper用其与Exectutor通信，并通过process方法完成事件的处理。这个例子中，Executor只是简单的把事件向下传给DataMonitor，具体的处理由DataMonitor决定。

DataMonitorListener用于DataMonitor与Executor进行通信，Executor中实现DataMonitor

-Listener的exists及closing的代码。

DataMonitor是本例子中Zookeeper逻辑的核心，异步的由事件驱动。构造方法中，调用zookeeper.exists方法，检查指定的znode是否存在，并设置监视，传递自身作为回调对象。Zookeeper.exist方法是在服务端完成的，Zookeeper API会在客户端调用completion callback（DataMonitor#processResult）。源码如下：

public void processResult(rcmpath,ctc,stat){

......

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

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

listener.exists(b);

prevData = b;

}

......

}

如果状态变化(b != prevData)，如果状态变化调用Executor的exists回调变化，将数据写入文件，并执行参数中的命令。

当监控的Znode变化时，出触发Executor的process方法，其最终调用DataMonitor

#process，源码如下所示：

public void process(WatchedEvent event) {

String path = event.getPath();

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

switch(event.getState()){

case SyncConnected:

break;

case Expired:

dead = true;

listener.closing(Code.SessionExpired);

break;

}

}else {

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

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

}

}

if (chainerWatcher != null){

chainerWatcher.process(event);

}

}

根据EventType的不同类型，做不同的处理。