Zookeeper

——Learning Report

ACADEMIC YEAR: 2019-2020

SEMESTER 1

张磊

UCAS 中国科学院大学

(zhanglei171@mails.ucas.ac.cn)

目录

0. 声明

大家好，我于 2019 年秋季参加中国科学院大学“面向的对象程序设计”课程，并选择 ZooKeeper 作为源码学习环节的目标项目，本篇文章主要分为三个部分：功能分析与建模、核心流程设计分析、高级设计意图分析。

本次源码分析使用的版本号为 ZooKeeper 3.4.13，主要关注第二部分的 Leader 选举的核心流程分析，可能会弱化第一部分功能分析和第三部分高级设计意图分析的部分内容，文中摘录了许多我认为总结的比较好的对 ZooKeeper 中重要概念的介绍，感兴趣的读者可以自行查阅，参考文献会在每一部分最后的小结部分中给出，如果是网上的文章会给出相应的网址，项目源码地址：

https://github.com/apache/zookeeper

由于本人初学 java，对 Zookeeper 难免有理解错误的地方，欢迎大家批评指正，共同进步。

张磊 2019 年 11 月 4 日 星期一

www.zhanglei171@mails.ucas.ac.cn

第一部分

重要功能和概念简介

一. ZooKeeper 的诞生

从计算机诞生之后的几十年里，大多数我们平时使用的应用服务都是在一个单独的单处理器计算机上运行的程序。但是今天，应用服务已经发生了很大的变化。今天的应用服务都是由很多个独立的程序组成，并且这些独立的程序分别运行在不同的计算机上。

开发一个程序时，要想协同一个应用中的多个程序工作是一件非常棘手的事。开发这样的应用，很容易让很多开发人员陷入如何使多个程序协同工作的逻辑中，最后导致没有时间更好地思考和实现他们自己的应用程序逻辑；又或者开发人员对协同逻辑关注不够，只是用很少的时间开发了一个简单脆弱的主协调器，导致不可靠的单一失效点。[1]

所以，ZooKeeper 诞生了。

ZooKeeper 的设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用。这样便使得应用开发人员可以更多关注应用本身的逻辑，而不是协同工作。

简单地说，ZooKeeper 的功能就是：**在分布式系统中协作多个任务。**

[1] ZooKeeper 分布式过程协同技术讲解 机械工业出版社

二. ZooKeeper 的特点

ZooKeeper 主要有以下 4 个特点：

- 顺序一致性：从同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKeeper 中去。
- 原子性：所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。
- 单一系统映像 ：无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。
- 可靠性：一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖。

三. ZooKeeper 的重要概念

3.1 会话（session）

Session 指的是 ZooKeeper 服务器与客户端会话。在 ZooKeeper 中，一个客户端连接是指客户端和服务器之间的一个 TCP 长连接。客户端启动的时候，首先会与服务器建立一个 TCP 连接，从第一次连接建立开始，客户端会话的生命周期也开始了。通过这个连接,客户端能够通过心跳检测与服务器保持有效的会话,也能够向 Zookeeper 服务器发送请求并接受响应，同时还能够通过该连接接收来自服务器的 Watch 事件通知。Session 的 sessionTimeout 值用来设置一个客户端会话的超时时间。当由于服务器压力太大、网络故障或是客户端主动断开连接等各种原因导致客户端连接断开时，只要在 sessionTimeout 规定的时间内能够重新连接上集群中任意一台服务器，那么之前创建的会话仍然有效。

在为客户端创建会话之前，服务端首先会为每个客户端都分配一个 sessionID。由于 sessionID 是 Zookeeper 会话的一个重要标识,许多与会话相关的运行机制都是基于这个 sessionID 的,因此,无论是哪台服务器为客户端分配的 sessionID，都务必保证全局唯一。

3.2 Znode

在谈到分布式的时候，我们通常说的“节点"是指组成集群的每

一台机器。然而，在 Zookeeper 中，“节点"分为两类，第一类同样是指构成集群的机器，我们称之为机器节点；第二类则是指数据模型中的数据单元，我们称之为数据节点——Znode。

Zookeeper 将所有数据存储在内存中，数据模型是一棵树（Znode Tree)，其中由斜杠（/）进行分割的路径，就是一个 Znode，例如/foo/path1。每个上都会保存自己的数据内容，同时还会保存一系列属性信息。

在 Zookeeper 中，node 可以分为持久节点和临时节点两类。所谓持久节点是指一旦这个 Znode 被创建了,除非主动进行 Znode 的移除操作，否则这个 Znode 将一直保存在 Zookeeper 上。而临时节点就不一样了，它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。另外，ZooKeeper 还允许用户为每个节点添加一个特殊的属性：SEQUENTIAL.一旦节点被标记上这个属性，那么在这个节点被创建的时候,Zookeeper 会自动在其节点名后面追加上一个整型数字，这个整型数字是一个由父节点维护的自增数字。

3.3 版本

在前面我们已经提到，ZooKeeper 的每个 Znode 上都会存储数据，对应于每个 Znode，ZooKeeper 都会为其维护一个叫做 Stat 的数据结构，Stat 中记录了这个 Znode 的三个数据版本，分别是 version（当前 Znode 的版本）、cversion（当前 Znode 子节点的版

本）和 cversion（当前 Znode 的 ACL 版本）。

3.4 Watcher

Watcher（事件监听器），是 ZooKeeper 中的一个很重要的特性。ZooKeeper 允许用户在指定节点上注册一些 Watcher，并且在一些特定事件触发的时候，ZooKeeper 服务端会将事件通知到感兴趣的客户端上去，该机制是 ZooKeeper 实现分布式协调服务的重要特性。

3.5 ACL

ZooKeeper 采用 ACL（AccessControlLists）策略来进行权限控制，类似于 UNIX 文件系统的权限控制。ZooKeeper 定义了如下 5 种权限。

- **CREATE** ： 创建子节点的权限；
- **READ** ： 获取节点数据和子节点列表的权限；
- **WRITE** ： 更新节点数据的权限；
- **DELETE** ： 删除子节点的权限；
- **ADMIN** ： 设置节点 ACL 的权限；

其中需要特别注意的是，CREATE 和 DELETE 这两种权限都是针对子节点的权限控制。

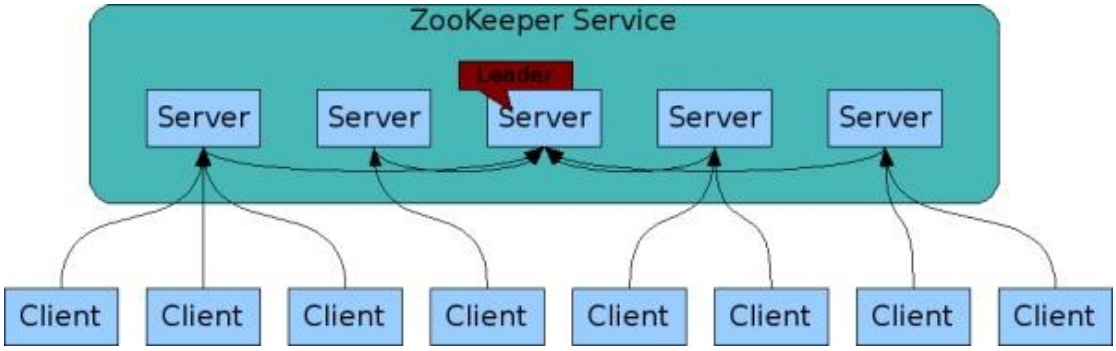
四. ZooKeeper 的设计目标

4.1 简单的数据模型

ZooKeeper 允许分布式进程通过共享的层次结构命名空间进行相互协调，这与标准文件系统类似。名称空间由 ZooKeeper 中的数据寄存器组成——称为 znode，这些类似于文件和目录。与为存储设计的典型文件系统不同，ZooKeeper 数据保存在内存中，这意味着 ZooKeeper 可以实现高吞吐量和低延迟。

4.2 可构建集群

为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么 ZooKeeper 本身仍然是可用的。客户端在使用 ZooKeeper 时，需要知道集群机器列表，通过与集群中的某一台机器建立 TCP 连接来使用服务，客户端使用这个 TCP 链接来发送请求、获取结果、获取监听事件以及发送心跳包。如果这个连接异常断开了，客户端可以连接到另外的机器上。



上图中每一个 Server 代表一个安装 Zookeeper 服务的服务器。

组成 ZooKeeper 服务的服务器都会在内存中维护当前的服务器状态，并且每台服务器之间都互相保持着通信。集群间通过 Zab 协议（Zookeeper Atomic Broadcast）来保持数据的一致性。

4.3 顺序访问

对于来自客户端的每个更新请求，ZooKeeper 都会分配一个全局唯一的递增编号，这个编号反应了所有事务操作的先后顺序，应用程序可以使用 ZooKeeper 这个特性来实现更高层次的同步原语。这个编号也叫做时间戳——zxid（Zookeeper Transaction Id）

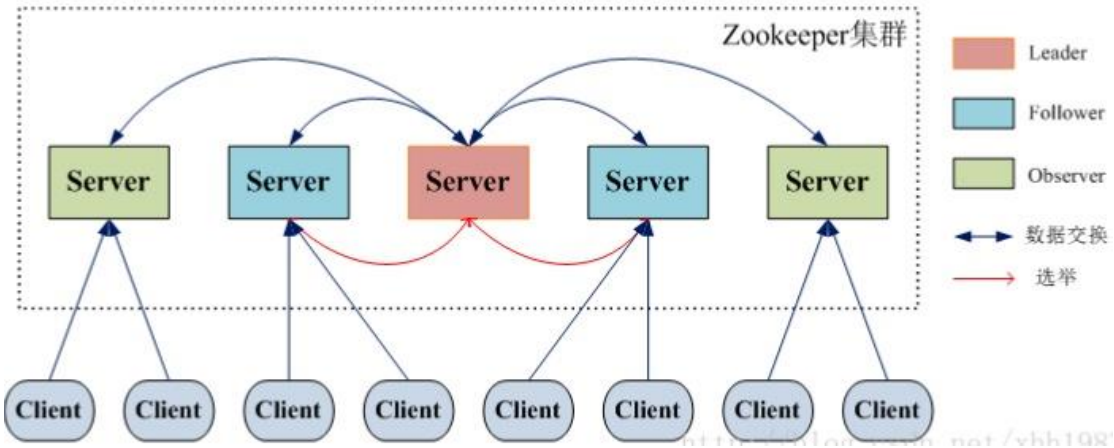
4.4 高性能

ZooKeeper 是高性能的。在“读”多于“写”的应用程序中尤其地高性能，因为“写”会导致所有的服务器间同步状态。(“读”多于“写”是协调服务的典型场景。)

五. ZooKeeper 的集群角色

最典型集群模式： Master/Slave 模式（主备模式）。在这种模式中，通常 Master 服务器作为主服务器提供写服务，其他的 Slave 服务器从服务器通过异步复制的方式获取 Master 服务器最新的数据提供读服务。

但是，在 ZooKeeper 中没有选择传统的 Master/Slave 概念，而是引入了 Leader、Follower 和 Observer 三种角色。



ZooKeeper 集群中的所有机器通过一个 Leader 选举过程来选定一台称为 “Leader” 的机器，Leader 既可以为客户端提供写服务又能提供读服务。除了 Leader 外，Follower 和 Observer 都只能提供读服务。Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。

角色		主要工作描述
领导者		1. 事务请求的唯一调度和处理者，保证集群事务处理的顺序性； 2. 集群内部各服务器的调度者
学习者（Learner）	跟随者（Follower）	1. 处理客户端非事务请求，转发事务请求给Leader服务器 2. 参与事务请求Proposal的投票 3. 参与Leader选举的投票
	观察者（Observer）	Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的"过半写成功"策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。
客户端（Client）		请求发起方

当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进人恢复模式并选举产生新的 Leader 服务器。这个过程大致是这样的：

1. **Leader election（选举阶段）**：节点在一开始都处于选举阶段，只要有一个节点得到超半数节点的票数，它就可以当选准 leader。
2. **Discovery（发现阶段）**：在这个阶段，followers 跟准 leader 进行通信，同步 followers 最近接收的事务提议。
3. **Synchronization（同步阶段）**:同步阶段主要是利用 leader 前一阶段获得的最新提议历史，同步集群中所有的副本。同步完成之后 准 leader 才会成为真正的 leader。
4. **Broadcast（广播阶段）**：到了这个阶段，Zookeeper 集群才能正式对外提供事务服务，并且 leader 可以进行消息广播。同时如果有新的节点加入，还需要对新节点进行同步。

六. ZooKeeper & ZAB 协议 & Paxos 算法

6.1 ZAB 协议 & Paxos 算法

ZooKeeper 并没有完全采用 Paxos 算法 ，而是使用 ZAB 协议作为其保证数据一致性的核心算法。另外，在 ZooKeeper 的官方文档中也指出，ZAB 协议并不像 Paxos 算法那样，是一种通用的分布式一致性算法，它是一种特别为 Zookeeper 设计的崩溃可恢复的原子消息广播算法。

6.2 ZAB 协议

ZAB（ZooKeeper Atomic Broadcast 原子广播） 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。 在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

6.3 崩溃恢复和消息广播

ZAB 协议包括两种基本的模式，分别是 **崩溃恢复和消息广播**。当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进入恢复模式并选举产生新的 Leader 服务器。

当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该 Leader 服务器完成了状态同步之后，ZAB 协议就会退出

恢复模式。其中，**所谓的状态同步是指数据同步，用来保证集群中存在过半的机器能够和 Leader 服务器的数据状态保持一致。**

当集群中已经有过半的 Follower 服务器完成了和 Leader 服务器的状态同步，那么整个服务框架就可以进人消息广播模式了。当一台同样遵守 ZAB 协议的服务器启动后加入到集群中时，如果此时集群中已经存在一个 Leader 服务器在负责进行消息广播，那么新加入的服务器就会自觉地进人数据恢复模式：找到 Leader 所在的服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。

如上所述，ZooKeeper 设计成只允许唯一的一个 Leader 服务器来进行事务请求的处理。Leader 服务器在接收到客户端的事务请求后，会生成对应的事务提案并发起一轮广播协议；而如果集群中的其他机器接收到客户端的事务请求，那么这些非 Leader 服务器会首先将这个事务请求转发给 Leader 服务器。

七. 小结

通过这段时间的阅读了解，我基本了解了 ZooKeeper 的由来、功能作用、特点、重要的概念、设计目标、集群角色以及 ZAB 协议和 Paxos 算法总算有了初步的了解。

但是由于没有亲自使用过 ZooKeeper，所以我对 ZooKeeper 的具体功能还是处于一种懵懂的状态，而且本部分摘取了很多前人的文章，自己的东西非常少，在之后的学习过程中我会尝试亲自使用 ZooKeeper，然后在使用中总结出自己的东西。

参考文献：

1. ZooKeeper：分布式过程协同技术详解 ZooKeeper：Distributed Process Coordination [美] 荣凯拉（Junqueira, F.）[美] 里德（Reed， B.）著
2. https://github.com/Snailclimb/JavaGuide/blob/master/docs/system-design/framework/ZooKeeper.md

第二部分

功能分析与需求建模

(Leader Election)

一. ZooKeeper 简单示例

ZooKeeper 最基本的一个功能就是数据同步。当 ZooKeeper 处于集群模式的时候（也就是有多个服务器的时候），此时假设我们有两个服务器 A 和 B，有两个客户端 C1 和 C2，C1 与 A 连接，C2 与 B 连接。

此时，客户端 C1 向服务器 A 发送数据，比如要创建一个/test 结点，并写入数据 666，写入成功后，客户端 C2 访问/test 结点的数据，那么此时，服务器 B 应该返回客户端 C1 写入的数据 666。

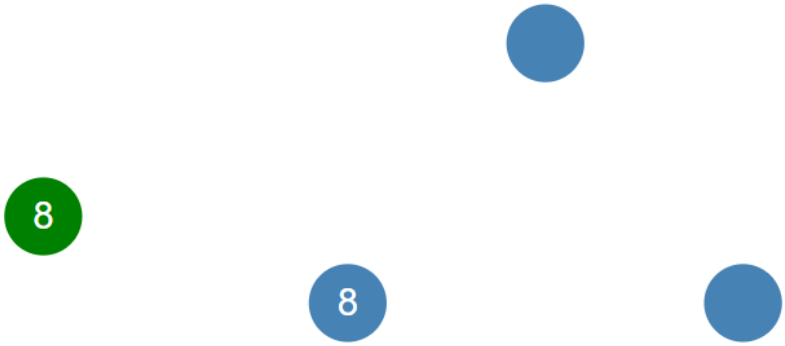
基于 watcher 的功能，ZooKeeper 还可以实现发布和订阅功能。

二. 数据同步

数据的同步在只有一个服务器的时候是非常简单地：

绿色客户端发送信息数据给服务器，不需要同步：

但是在集群模式下，数据同步就变得不那么容易了：



那么，如何同步多个服务器之间的数据呢？这就是我们这一部分需要讨论的重点了。

三. 人类的选举

在讨论这个问题之前我们需要先来研究一下选举，在后面我们会看到服务器的选举其实和人类的选举是十分类似的。

1. 人类选举的基本原理

小到选班长、组长，大到选地方领导，我们应该都经历过投票，如果你是一个非常具有正义感的人,那么你在投票时一定会把票投给自己认为能力比较强的人。如果你已经选好了一个人，但是又突然出现了能力更强的人，那么你可能会改选这个更强的人。选票将会放在投票中,最后从投票箱中进行统计,获得票数最多的人当选。

在这样一个选举过程中我们提炼出四个基本概念：

1. **个人能力：**投我认为能力最强的人，这是投票的基本规则；
2. **改票：**能力最强的人是逐渐和其他人沟通之后的结果，类似改票，先投给 A，但是后来发现 B 更厉害，则改为投 B；
3. **投票箱：**所有人公用一个投票箱；
4. **领导者：**获得投票数最多的人为领导者；

有了这个概念，我们就可以来看看 ZooKeeper 的选举了。

四. ZooKeeper 的选举

首先 ZooKeeper 只有在集群模式下才需要进行选举。

在选举中 Zookeeper 需要实现上面人类选举的四个基本概念：

1. **个人能力：**我们说 Zookeeper 可以看做是一个数据库，那么集群中节点的数据越新就代表此节点能力越强，而在 Zookeeper 中可以通过事务 id(zxid)来表示数据的新旧，一个节点的 zxid 越大则该节点的数据越新。所以 Zookeeper 选举时会根据 zxid 的大小来作为投票的基本规则。

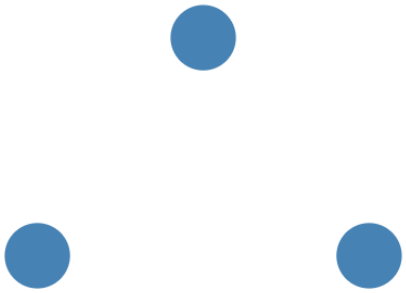
2. **改票：**Zookeeper 集群中的某一个节点在开始进行选举时，首先认为自己的数据是最新的，会先投自己一票，并且把这张选票发送给其他服务器，这张选票里包含了两个重要信息：zxid 和 sid，sid 表示这张选票投的服务器 id，zxid 表示这张选票投的服务器上最大的事务 id，同时也会接收到其他服务器的选票。
   接收到其他服务器的选票后，可以根据选票信息中的 zxid 来与自己当前所投的服务器上的最大 zxid 来进行比较，如果其他服务器的选票中的 zxid 较大，则表示自己当前所投的机器数据没有接收到的选票所投的服务器上的数据新，所以本节点需要改票，改成投给和刚刚接收到的选票一样。

3. **投票箱：**Zookeeper 集群中会有很多节点，和人类选举不一样，Zookeeper 集群并不会单独去维护一个投票箱应用，而是在每个节点内存里利用一个数组来作为投票箱。每个节点里都有一个投票箱，节点会将自己的选票以及从其他服务器接收到的选票放在这个投票箱中。因为集群节点是相互交互的，并且选票的 PK 规则是一致的，所以每个节点里的这个投票箱所存储的选票都会是一样的，这样也可以达到公用一个投票箱的目的。

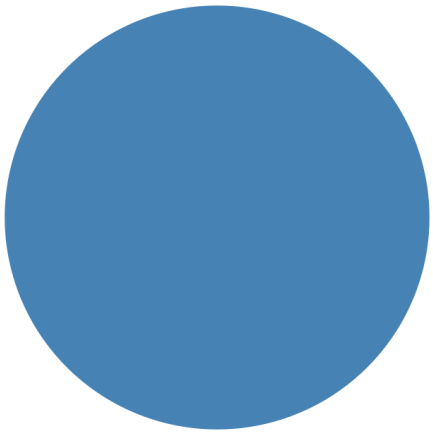
4. **领导者：**Zookeeper 集群中的每个节点，开始进行领导选举后，会不断的接收其他节点的选票，然后进行选票 PK，将自己的选票修改为投给数据最新的节点，这样就保证了，每个节点自己的选票代表的都是自己暂时所认为的数据最新的节点，再因为其他服务器的选票都会存储在投票箱内，所以可以根据投票箱里去统计是否有超过一半的选票和自己选择的是同一个节点，都认为这个节点的数据最新，一旦整个集群里超过一半的节点都认为某一个节点上的数据最新，则该节点就是领导者。

五. 具体选举过程说明

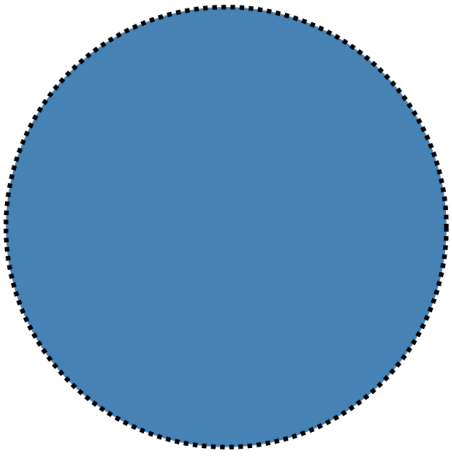
我们来看 3 台服务器的情况：



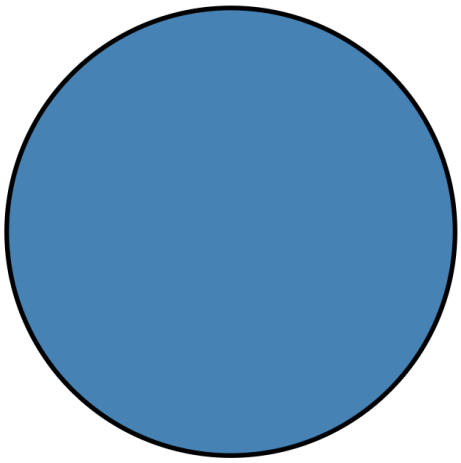
在 raft 中每台服务器存在三种可能的状态：



The *Follower* state,

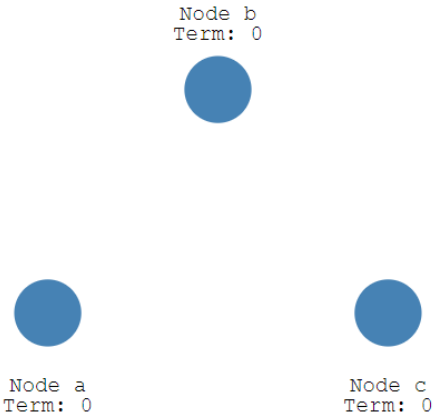


the *Candidate* state,

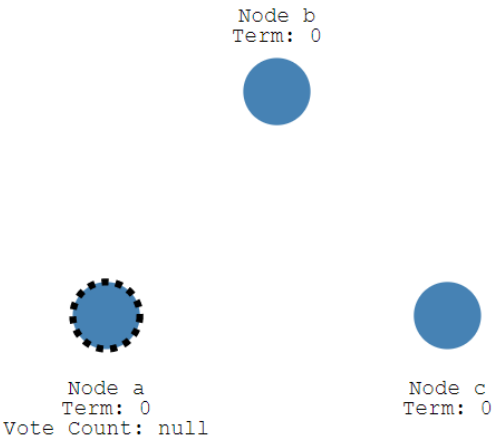


or the *Leader* state.

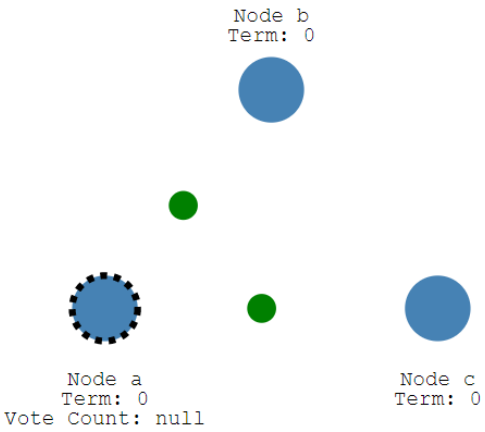
下面我们把刚才 3 个结点服务器命名如下：



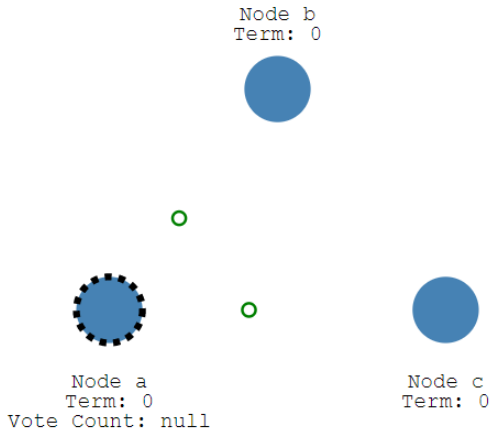
如果集群中没有 leader，那么 follower 就会变成 candidate 状态，



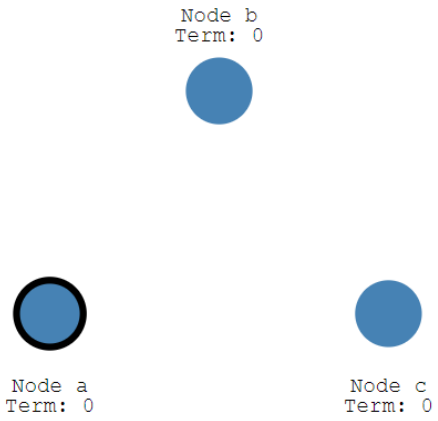
Candidate 会发送一个请求询问其他成员的投票：



其他成员也会回复 Candidate 的询问，



然后按照少数服从多数的原则，选出 Leader，



这个过程就是我们要讨论的重点：Leader Election。

六. 小结

在这一部分中，我们通过 ZooKeeper 的一个最基本的数据同步功能引出了 Leader 选举，然后通过与人类社会的选举类比，将 ZooKeeper 的 Leader 选举给抽象了出来，大致理解了整个 Leader 选举的过程。

但是在服务器运行的过程当中可能会出现一些其他的情况，导致 Leader 的变更，此时又要重新进行选举，但是大致的流程都是一样的，我就不再细讲了。有需要的同学可以通过下方的参考文献中的链接去看下一下选举的过程，以及出现问题了应该怎么办。

下一部分，我们将会深入到函数和类当中，看一下，ZooKeeper 是如何运用面向对象的方法，完成 Leader 选举的。

参考文献

1. http://thesecretlivesofdata.com/raft/

第三部分

核心流程设计分析

(Leader Election)

一. Leader 选举入口

ZooKeeper 有两种模式，一种单机模式，一种集群模式。其中：

1. **ZooKeeperServer 表示单机模式中的一个 zkServer;**
2. **QuorumPeer 表示集群模式中的一个 zkServer;**

QuorumPeer 类的定义如下：

```
public class QuorumPeer extends ZooKeeperThread implements QuorumStats.Provider
```

定义表明 QuorumPeer 是一个 ZooKeeperThread，即 QuorumPeer 是一个线程。

当集群中的某一个台 zkServer 启动时 QuorumPeer 类的 start 方法将被调用。

QuorumPeer 的 start 方法：

```
public synchronized void start() {
    // 加载数据库
    loadDataBase();
    // 开启读取线程
    // 还没有完成领导选举，ZooKeeper还不能提供服务
    // 所以此时客户端即使连接上了服务器也会被拒绝
    cnxnFactory.start();//server端负责与client的连接
    // 进行领导者选举，确定服务器的角色
    // 再对各个服务器角色进行对应的初始化
    startLeaderElection();
    // 到这里完成了Leader选举的应用层和传输层的初始化
    // 本类的run方法
    super.start();// 由于它本身就是个线程，所以调用start方法就是直接run
}
```

1. **loadDataBase()：**

   zkServer 中有一个内存数据库对象 ZKDatabase，zkServer 在启动时需要将已被持久化的数据加载到内存中，也就是加载至 ZKDatabase;

2. **cnxnFactory.start()：**

   这一步会开启一个线程来接收客户端请求，但是需要注意，这一步执行完后虽然成功开启了一个线程，并且也可以接收客户端线程，但是因为现在 zkServer 还没有经过初始化，实际上会把请求拒绝掉，直到 zkServer 初始化完成才能正常的接收请求;

3. **startLeaderElection()：**

   这个名字非常有误导性，这里其实并没有开始进行 Leader 的选举，只是做了一些初始化;

4. **super.start()：**

   继续启动，包括进行领导者选举、zkServer 初始化;

二. Leader 的选举策略

QuorumPeer 类的 startLeaderElection 会进行领导者选举初始化。领导者选举在 Zookeeper 中有 3 种实现：

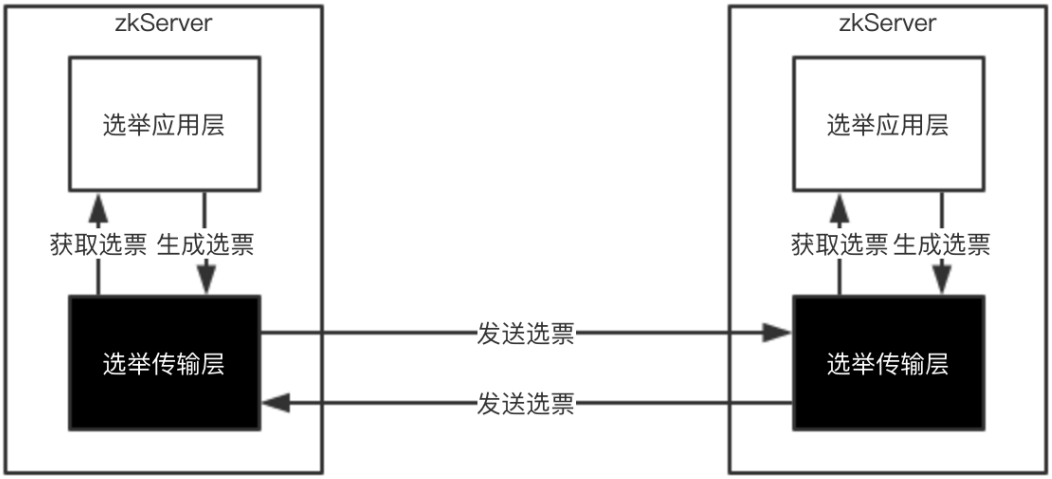
```
protected Election createElectionAlgorithm(int electionAlgorithm){
    Election le=null;

    //TODO: use a factory rather than a switch
    switch (electionAlgorithm) {
    case 0:
        le = new LeaderElection(this);
        break;
    case 1:
        le = new AuthFastLeaderElection(this);
        break;
    case 2:
        le = new AuthFastLeaderElection(this, true);
        break;
    case 3:
        // 初始化负责各台服务器之间底层Leader选举过程的网络通信
        qcm = createCnxnManager();
        QuorumCnxManager.Listener listener = qcm.listener;
        if(listener != null){
            // 启动之前初始化的Listener
            listener.start();
            // 默认采用FastLeaderElection
            le = new FastLeaderElection(this, qcm);
        } else {
            LOG.error("Null listener when initializing cnx manager");
        }
        break;
    default:
        assert false;
```

其中 LeaderElection、AuthFastLeaderElection 已经被标为过期，不建议使用，所以现在用的都是快速领导者选举 FastLeaderElection，这里也只介绍 FastLeaderElection。

三. FastLeaderElection 策略

FastLeaderElection 实现架构图：



四. 传输层初始化

从架构图我们可以发现，快速领导者选举实现架构分为两层：应用层和传输层。所以初始化核心就是初始化传输层。

初始化步骤：

1. **初始化 QuorumCnxManager;**
2. **初始化 QuorumCnxManager.Listener;**
3. **运行 QuorumCnxManager.Listener;**
4. **运行 QuorumCnxManager;**
5. **返回 FastLeaderElection 对象;**

QuorumCnxManager 介绍：

QuorumCnxManager 就是传输层实现，QuorumCnxManager 中几个重要的属性：

- `final ConcurrentHashMap<Long, ArrayBlockingQueue<ByteBuffer>> queueSendMap;`
- `final ConcurrentHashMap<Long, SendWorker> senderWorkerMap;`
- `public final ArrayBlockingQueue<Message> recvQueue;`
- `public final Listener listener;`

传输层的每个 zkServer 需要发送选票信息给其他服务器，这些选票信息来至应用层，在传输层中将会按服务器 id 分组保存在 queueSendMap 中。

传输层的每个 zkServer 需要发送选票信息给其他服务器，SendWorker 就是封装了 Socket 的发送器，而 senderWorkerMap 就是用来记录其他服务器 id 以及对应的 SendWorker 的。

传输层的每个 zkServer 将接收其他服务器发送的选票信息，这些选票会保存在 recvQueue 中，以提供给应用层使用。

Listener 负责开启 socket 监听。

五. 应用层初始化

服务器在进行领导者选举时，在发送选票时也会同时接受其他服务器的选票，FastLeaderElection 类也提供了和传输层类似的实现，将待发送的选票放在 sendqueue 中，由 Messenger.WorkerSender 发送到传输层 queueSendMap 中。

同样，由 Messenger.WorkerReceiver 负责从传输层获取数据并放入 recvqueue 中。

这样在应用层了，只需要将待发送的选票信息添加到 sendqueue 中即可完成选票信息发送，或者从 recvqueue 中获取元素即可得到选票信息。

在构造 FastLeaderElection 对象时，会对 sendqueue、recvqueue 队列进行初始化，并且运行 Messenger.WorkerSender 与 Messenger.WorkerReceiver 线程。

六. 快速领导者选举实现

快速领导者选举的最终实现其实就在 super.start 方法中：

QuorumPeer 类是一个 ZooKeeperThread 线程，super.start()实际就是运行一个线程，相当于运行 QuorumPeer 类中的 run 方法，这个方法也是集群模式下 Zkserver 启动最核心的方法。

总结一下 QuorumPeer 类的 start 方法：

1. **加载持久化数据到内存；**
2. **初始化领导者选举策略；**
3. **初始化快速领导者选举传输层；**
4. **初始化快速领导者选举应用层；**
5. **开启主线程；**

接下来我们着重来分析一下主线程内的逻辑。

七. 快速领导者选举主线程

主线程伪代码如下图：

```
while (服务是否正在运行) {
    switch (当前服务器状态) {
        case LOOKING:
            // 领导者选举
            setCurrentVote(makeLEStrategy().lookForLeader());
            break;
        case OBSERVING:
            try {
                // 初始化为观察者
            } catch (Exception e) {
                LOG.warn("Unexpected exception",e );
            } finally {
                observer.shutdown();
                setPeerState(ServerState.LOOKING);
            }
            break;
```

```
        case FOLLOWING:
            try {
                // 初始化为跟随者
            } catch (Exception e) {
                LOG.warn("Unexpected exception",e);
            } finally {
                follower.shutdown();
                setPeerState(ServerState.LOOKING);
            }
            break;
        case LEADING:
            try {
                // 初始化为领导者
            } catch (Exception e) {
                LOG.warn("Unexpected exception",e);
            } finally {
                leader.shutdown("Forcing shutdown");
                setPeerState(ServerState.LOOKING);
            }
        break;
    }
}
```

根据伪代码可以看到，当服务器状态为 LOOKING 时会进行领导者选举，所以我们着重来看领导者选举。

lookForLeader：

当服务器状态为 LOOKING 时会调用 FastLeaderElection 类的 lookForLeader 方法，这就是领导者选举的应用层。

该过程可分为 9 个步骤：

1. 初始化一个投票箱：

```
HashMap<Long, Vote> recvset = new HashMap<Long, Vote>();
```

2. 更新选票，将票投给自己：

```
updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());
```

3. 发送选票：

```
sendNotifications();
```

4. 不断获取其他服务器的投票信息，直到选出 Leader：

```
while ((self.getPeerState() == ServerState.LOOKING) && (!stop)){
    // 从recvqueue中获取接收到的投票信息
    Notification n = recvqueue.poll(notTimeout, TimeUnit.MILLISECONDS);

    if (获得的投票为空) {
        // 连接其他服务器
    } else {
        // 处理投票
    }
}
```

5. 连接其他服务器：

因为在这一步之前，都只进行了服务器的初始化，并没有真正的去与其他服务器建立连接，所以在这里建立连接。

6. 处理投票：

判断接收到的投票所对应的服务器的状态，也就是投此票的服务器的状态：

```
switch (n.state) {
    case LOOKING:
        // PK选票、过半机制验证等
        break;
    case OBSERVING:
        // 观察者节点不应该发起投票，直接忽略
        break;
    case FOLLOWING:
    case LEADING:
        // 如果接收到跟随者或领导者节点的选票，
        // 则可以认为当前集群已经存在Leader了，
        // 直接return，退出lookForLeader方法。
}
```

7. PK 选票：

```
if (接收到的投票的选举周期 > 本服务器当前的选举周期) {
    // 修改本服务器的选举周期为接收到的投票的选举周期
    // 清空本服务器的投票箱（表示选举周期落后，重新开始投票）
    // 比较接收到的选票所选择的服务器与本服务器的数据谁更新，
    // 本服务器将选票投给数据较新者发送选票
} else if(接收到的投票的选举周期 < 本服务器当前的选举周期){
    // 接收到的投票的选举周期落后了，本服务器直接忽略此投票
} else if(选举周期一致) {
    // 比较接收到的选票所选择的服务器与本服务器当前所选择的服务器的数据谁更新，
    // 本服务器将选票投给数据较新者发送选票
}
```

8. 过半机制验证：

本服务器的选票经过不停的 PK 会将票投给数据更新的服务器，PK 完后，将接收到的选票以及本服务器自己所投的选票放入投票箱中，然后从投票箱中统计出与本服务器当前所投服务器一致的选票数量，判断该选票数量是否超过集群中所有跟随者的一半（选票数量 > 跟随者数量/2），如果满足这个过半机制就选出了一个准 Leader。

9. 最终确认：

选出准 Leader 之后，再去获取其他服务器的选票，如果获取到的选票所代表的服务器的数据比准 Leader 更新，则准 Leader 卸职，继续选举。如果没有准 Leader 更新，则继续获取投票，直到没有获取到选票，则选出了最终的 Leader。

Leader 确定后，其他服务器的角色也确定好了。

八. 快速领导者选举主线程

根据伪代码我们可以发现，只有当集群中服务器的角色确定了之后，while 才会进行下一次循环，当进入下一次循环后，就会根据服务器的角色进入到对应的初始化逻辑,初始化完成之后才能对外提供服务。

因此，选举代码实现了一个非常重要的功能：

ZooKeeper 集群在进行领导者选举的过程中不能对外提供服务。

九. 小结

在这一部分中，我们深入代码，根据代码的执行流程，分析了ZooKeeper 集群的快速领导选举的过程，对整个选举过程的层次有了更加清晰地认识，对选举的过程也理解的更加深刻了。

由于这部分内容非常晦涩难懂，查询网上的资料也没有讲得很详细的，所以我听了几节网课，这部分的很多内容也是我从网课中学到的，我也看了许多讲课老师写的资料，如果大家想要再了解一点其他的关于 ZooKeeper 的知识，比如脑裂和二次提交，也可以去看一下，地址和资料见下文参考文献。

参考文献

1. https://www.yuque.com/renyong-jmovm/kb/fukd3b

2. 网课：鲁班学院 周瑜老师 ZooKeeper 小课

第四部分

高级设计意图分析

一. 本部分简介

在完成了功能分析与需求建模，以及 Leader Election 的核心流程设计分析后，我们来探讨一下 ZooKeeper 中使用到的设计模式，和高级设计意图。

由于时间的关系，我只选取了 ZooKeeper 中最基本的一个设计模式进行分析——观察者模式。

二. 观察者模式

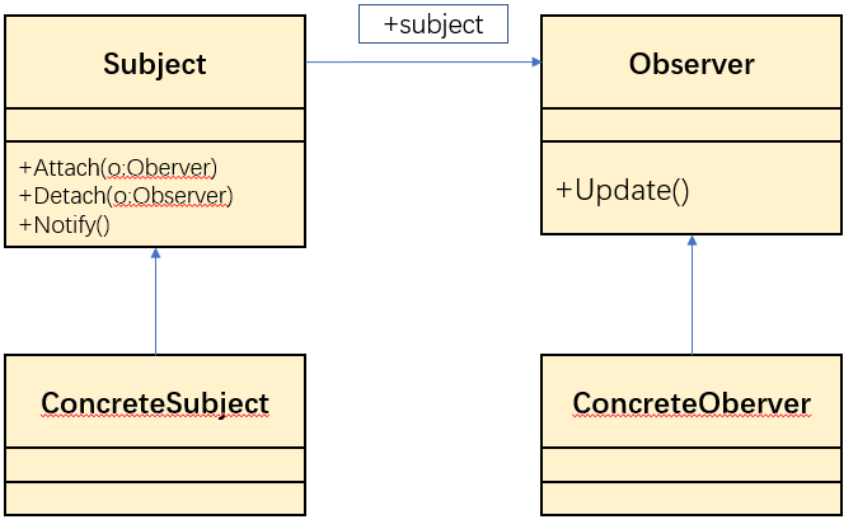
1. 观察者模式与 ZooKeeper：

观察者模式属于行为型模式，又叫发布订阅模式。ZooKeeper 中的发布订阅功能的核心——Watcher 机制就是分布式系统中的观察者模式的实例；

2. 定义：

观察模式定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新；

3. 观察者模式类图：



4. 角色：

1) 被观察者：Subject 定义一个被观察者必须实现职责，包括动态增加，删除，通知观察者；
2) 观察者：Observer 接受到观察者修改消息，执行自身逻辑；
3) 具体观察者：ConctreteSubject 继承 Subject，拥有自己的业务逻辑，具有被观察者基本功能，对某些事件进行通知；
4) 具体被观察者：ConcreteObserver 具体观察者，在接受到被观察者变更消息后，进行各自业务处理；

5. 观察者模式的优点：

1) 观察者和被观察者之间是抽象耦合，容易拓展；
2) 通过触发机制可以创建成一种链式触发机制,形成多级触发；

6. 观察者模式的缺点：

1) 执行效率：

当通知观察者是顺序执行时，需要考虑整个观察者列表的数量，对整个通知事件执行效率的影响，可以考虑使用异步通知，同时尽量避免多级触发事件；

2) 循环依赖：

当观察者和被观察之间形成循环依赖，会导致循环调用，比如 A 改变通知 B，B 改变通知 C，C 改变通知 A，注意避免循环依赖的发生；

三. ZooKeeper 的发布与订阅功能

发布与订阅是 ZooKeeper 提供的最基本的功能之一，而 ZooKeeper 的发布订阅功能又是基于 Watcher 机制实现的。所以，下面我们来看看 ZooKeeper 是怎么使用观察者模式实现 Watcher 机制的。

1. 发布与订阅的基本概念：

1) 发布订阅模式可以看成一对多的关系：多个订阅者对象同时监听一个主题对象，这个主题对象在自身状态发生变化时，会通知所有的订阅者对象，使他们能够自动的更新自己的状态。
2) 发布订阅模式，可以让发布方和订阅方，独立封装，独立改变，当一个对象的改变，需要同时改变其他的对象，而且它不知道有多少个对象需要改变时，可以使用发布订阅模式
3) 发布订阅模式在分布式系统的典型应用有， 配置管理和服务发现。
    i. 配置管理:是指如果集群中机器拥有某些相同的配置，并且这些配置信息需要动态的改变,我们可以使用发布订阅模式，对配置文件做统一的管理，让这些机器各自订阅配置文件的改变,当配置文件发生改变的时候这些机器就会得到通知,把自己的配置文件更新为最新的配置

ii. 服务发现:是指对集群中的服务上下线做统一的管理,每个工作服务器都可以作为数据的发布方,向集群注册自己的基本信息,而让模型机器作为订阅方,订阅工作服务器的基本信息,当工作服务器的基本信息发生改变时如上下线,服务器的角色和服务范围变更,监控服务器就会得到通知,并响应这些变化。

四. Watcher 机制

1. 客户端注册监听:

每个客户端创建 Zookeeper 成功连接之后都会开启两个后台线程,SendThread 和 EventThread。

SendThread 负责处理该客户端和 zk 服务器的所有通讯。

EventThread 主要负责处理服务端的通知推送进行的回调。

首先将Watcher和需要监视的路径包装到DataWatchRegistration。然后调用 submitRequest 封装请求发送给 Zookeeper 服务器。

```
public byte[] getData(final String path, Watcher watcher, Stat stat)
    throws KeeperException, InterruptedException
{
    final String clientPath = path;
    PathUtils.validatePath(clientPath);

    // the watch contains the un-chroot path
    WatchRegistration wcb = null;
    if (watcher != null) {
        wcb = new DataWatchRegistration(watcher, clientPath);
    }
```

```
final String serverPath = prependChroot(clientPath);

RequestHeader h = new RequestHeader();
h.setType(ZooDefs.OpCode.getData);
GetDataRequest request = new GetDataRequest();
request.setPath(serverPath);
request.setWatch(watcher != null);
GetDataResponse response = new GetDataResponse();
ReplyHeader r = cnxn.submitRequest(h, request, response, wcb);
if (r.getErr() != 0) {
    throw KeeperException.create(KeeperException.Code.get(r.getErr()),
            clientPath);
}
if (stat != null) {
    DataTree.copyStat(response.getStat(), stat);
}
return response.getData();
```

在 Zookeeper 中，Packet 是最小的通信单元。但是实际上 DataWatchRegistration 并不会被序列化传输给服务器端，只是 request.setWatch 设置了标志。

在发送了 getData 请求之后，如果成功收到了响应，SendThread 线程会处理这个响应，并且在最后 finally 块中调用 finishPacket 方法进行响应之后的操作，包括本地客户端需要注册 Watcher。

```
void readResponse(ByteBuffer incomingBuffer) throws IOException {
    ByteBufferInputStream bbis = new ByteBufferInputStream(…
    BinaryInputArchive bbia = BinaryInputArchive.getArchive(bbis);
    ReplyHeader replyHdr = new ReplyHeader();

    replyHdr.deserialize(bbia, "header");
    if (replyHdr.getXid() == -2) {…
    }
    if (replyHdr.getXid() == -4) {…
    }
    if (replyHdr.getXid() == -1) {…
    }

    // If SASL authentication is currently in progress, construct and
    // send a response packet immediately, rather than queuing a
    // response as with other packets.
    if (clientTunneledAuthenticationInProgress()) {…
    }

    Packet packet;
    synchronized (pendingQueue) {…
    }
    /*…
    try {…
    } finally {
        finishPacket(packet);
    }
}
```

finishPacket 方法包括判断请求是否包含 Watcher 的监听。

```
private void finishPacket(Packet p) {
    if (p.watchRegistration != null) {
        p.watchRegistration.register(p.replyHeader.getErr());
    }

    if (p.cb == null) {
        synchronized (p) {
            p.finished = true;
            p.notifyAll();
        }
    } else {
        p.finished = true;
        eventThread.queuePacket(p);
    }
}
```

如果包含了 Watcher，则调用 register 方法进行注册到对应的(path->Set)映射表中，客户端对 Watcher 的管理主要在 ZKWatchManager 这个类中。

```
private static class ZKWatchManager implements ClientWatchManager {
    //getData注册的Watcher
    private final Map<String, Set<Watcher>> dataWatches =
        new HashMap<String, Set<Watcher>>();
    //exist操作注册的Watcher
    private final Map<String, Set<Watcher>> existWatches =
        new HashMap<String, Set<Watcher>>();
    //getChildren注册的监听
    private final Map<String, Set<Watcher>> childWatches =
        new HashMap<String, Set<Watcher>>();
```

2. 服务器端处理监听请求：

服务端接受到客户端的请求之后，FinalRequestProcessor，processRequest 方法判断该请求是否要注册 Watcher。如果需要注册 Watcher，则在服务器端的 list 中添加 watcher。

```
public synchronized void addWatch(String path, Watcher watcher) {
    HashSet<Watcher> list = watchTable.get(path);
    if (list == null) {
        // don't waste memory if there are few watches on a node
        // rehash when the 4th entry is added, doubling size thereafte
        // seems like a good compromise
        list = new HashSet<Watcher>(4);
        watchTable.put(path, list);
    }
    list.add(watcher);
```

3. 服务器端触发监听：

服务器端会从之前管理的 Watcher 映射中得到该路径上的 Watcher 集合，并调用 process 方法，其实就是发送通知。

```
@Override
synchronized public void process(WatchedEvent event) {
    ReplyHeader h = new ReplyHeader(-1, -1L, 0);
    if (LOG.isTraceEnabled()) {
        ZooTrace.logTraceMessage(LOG, ZooTrace.EVENT_DELIVERY_TRACE_MASK,
                                 "Deliver event " + event + " to 0x"
                                 + Long.toHexString(this.sessionId)
                                 + " through " + this);
    }

    // Convert WatchedEvent to a type that can be sent over the wire
    WatcherEvent e = event.getWrapper();

    sendResponse(h, e, "notification");
}
```

4. 客户端回调监听

客户端接受到响应之后，会判断响应头标识中的 XID 是否为-1，如果是-1，则标识这是一个通知类型的响应。

客户端会不断从 waitingEvents 中取通知事件，然后调用 processEvent 方法处理该事件。

```
private void processEvent(Object event) {
    try {
        if (event instanceof WatcherSetEventPair) {
            // each watcher will process the event
            WatcherSetEventPair pair = (WatcherSetEventPair) event;
            for (Watcher watcher : pair.watchers) {
                try {
                    watcher.process(pair.event);
                } catch (Throwable t) {
                    LOG.error("Error while calling watcher ", t);
                }
            }
        } else {
```

processEvent 方法取出本次事件需要通知的所有集合，并进行循环调用每个 Watcher 的 process 方法。

五. 小结

在这一部分中，我们分析了观察者模式在 ZooKeeper 中的应用，也就是 Watcher 机制的实现，这部分如果只是看流程图的话还是挺简单的，但是具体到代码实现的时候，看着还是挺头疼的。另外也由于时间比较紧迫，我没来得及每行代码都仔细的去调试，去看，所以我参考了官方文档和一些博主们的博文，看了他们对 ZooKeeper 中的 Watcher 机制的分析，下方是本部分的参考文献。

参考文献

1. http://www.zzcblogs.top/2018/11/06/zookeeper-watcher%E6%9C%BA%E5%88%B6/
2. https://draveness.me/zookeeper-chubby
3. https://chenmingyu.top/design/
4. 官方手册

第五部分

结语

结语

最最艰苦的一个学期终于要结束了。虽然其他课程压力比较大，每个周都被实验课压得喘不过气，但是好在每周五还能在课上听老师讲讲“单口相声”，放松一下心情。

虽然我之前没有接触过 JAVA，但是在这个学期阅读源码的过程中，好像已经慢慢学到了许多 JAVA 编程的语法，另外就是我也是第一次接触到 ZooKeeper 这样的顶级开源项目，我第一次把源码下载下来的时候，面对里边众多的文件，都不知道该从何看起。

后来，我咨询了窦文生老师，部署安装了环境，亲自在虚拟机里跑起了 ZooKeeper 服务器，试了下 ZooKeeper 的一些创建节点，设置节点值，读取节点值的操作后，开始对 ZooKeeper 有了感觉，原来分布式就是这么个意思。

再后来，我又陆续的搜寻资料，看了几节专门讲 ZooKeeper 的网课，并阅读了一些博主的优质博文，深入了解了 ZooKeeper 从启动到完成 Leader 选举，对外提供服务，再到后来的 Watcher 机制的实现的过程。但是，对于 Watcher 机制的实现我还是有许多疑问，不像 Leader 选举那样清楚每一步是怎么实现的，等考完试假期间我再自己跑一遍，看一下它到底是怎么实现的，总的来说阅读 ZooKeeper 源码的收获还是蛮大的。

最后，感谢王伟老师和唐震助教。王伟老师不仅讲课有趣，还很关心我们的学业压力，哈哈。选则项目的时候我犹豫不决，助教耐心的鼓励我不要担心，去尝试 A 类项目，就感觉助教人超级好，

这学期问了助教很多问题，每次问问题助教都很快就回复我了，真的超级感动。