1. 上课约定须知

课程主题：**ZooKeeper-3.6.3 架构设计实现和源码深度剖析：ZooKeeper 核心功能深度剖析 和 企业级最佳应用实战**
上课时间：20:00 - 23:00
授课讲师：马中华
课间休息：21:30 - 22:00 之间择机休息 10 分钟，按照以往规律，大概在 21:40 左右，请各位做好完全准备
课前签到：如果进入直播间能清晰听见声音，清楚能看到画面，请在直播间扣 666 签到，非常感谢大家配合，如果有回声，或者直播不正常等，请及时反馈
听课建议：推荐使用 PC 网页观看直播，并且最好每隔 15 分钟左右刷新一下直播间以降低延迟！
课后答疑：23:00 下课之后，会有专门的集中答疑环节。

2. 本次课程大纲

今天这次课程的主题是：**ZooKeeper-3.6.3 架构设计实现和源码深度剖析：ZooKeeper 核心功能深度剖析 和 企业级最佳应用实战**

今天的主要内容有：

```
第一部分：ZooKeeper核心功能剖析
    01、ZooKeeper架构实现理解增强
    02、ZNode数据模型
    03、Watcher监听机制

第二部分：Zookeeper企业最佳实战
    04、应用场景分析
    05、实际企业案例的编码实现和运行演示
        案例1：分布式独占锁实现 Active Master 选举
        案例2：分布式时序锁实现分布式队列
        案例3：分布式集群配置管理
```

企业应用案例的目的：

```
1、要知道一些常用的流行的分布式技术底层使用 ZooKeeper 到底干了什么
2、我们可以用 ZooKeeper 解决企业生产的什么问题
```

HBase，Spark，Flink 的注册，选举，监听等就是 基于 ZooKeeper 实现的 。 咱们在看这些技术的源码的时候，都会看到涉及到基于 ZooKeeper 做 HA 的选举 或者 监听 或者注册等。

3. 课程详细内容：ZooKeeper 核心功能深度剖析 和 企业级最佳应用实战

3.1. ZooKeeper 核心功能和工作机制

Welcome to Apache ZooKeeper™

Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination.

What is ZooKeeper?

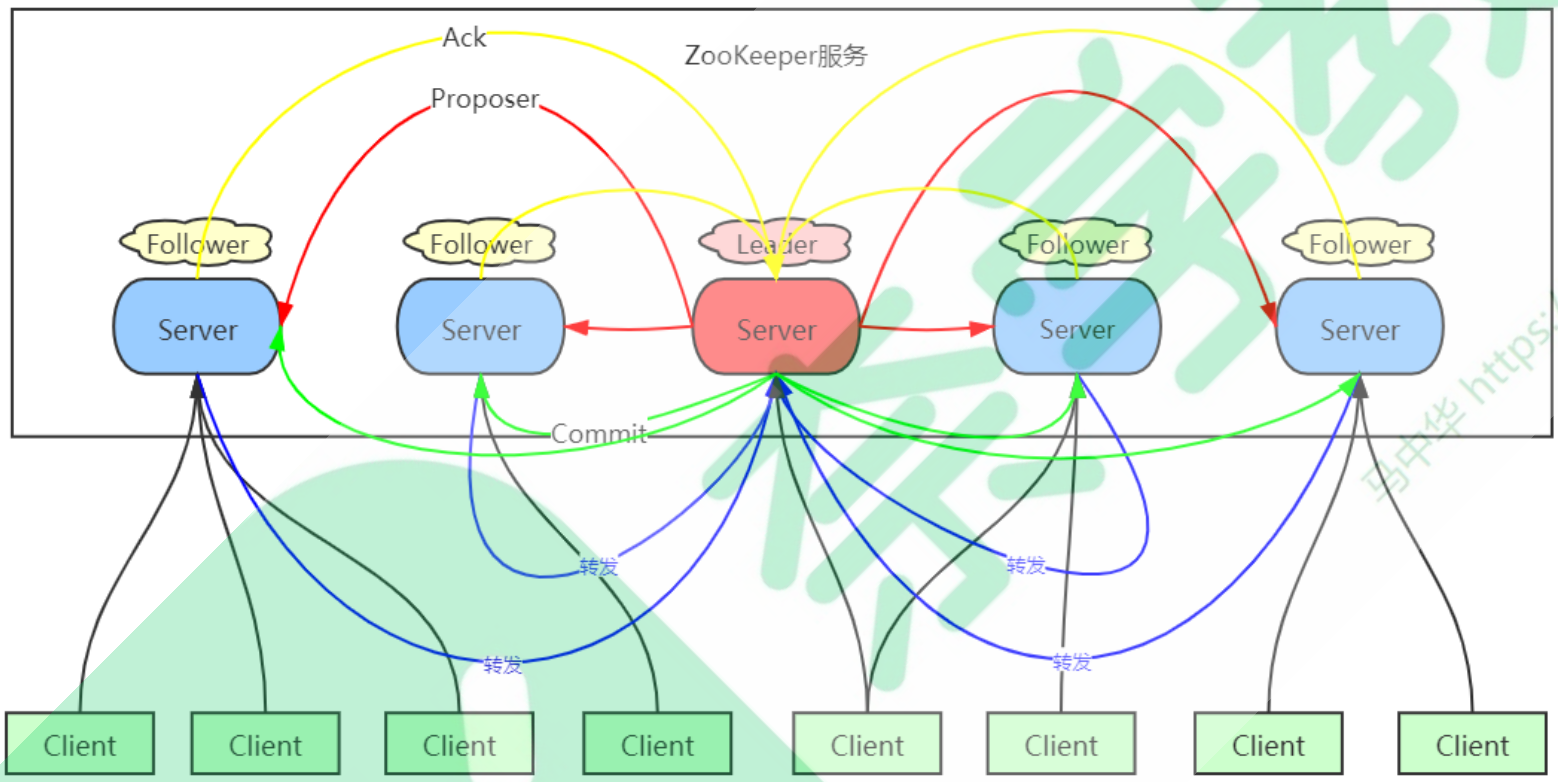
ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

3.1.1. ZooKeeper 架构理解增强

ZooKeeper 是一个分布式协调服务，劝架者，仲裁机构。 多个节点如果出现了意见的不一致，需要一个中间机构来调停！

ZooKeeper 就是一个小型的议会！当分布式系统中的多个节点，如果出现不一致，则把这个不一致的情况往 ZooKeeper 中写。ZooKeeper 会给你返回写成功的响应。但凡你接收到成功的响应，意味着 ZooKeeper 帮你达成了一致！

ZooKeeper 以一个集群的方式对外提供协调服务，**集群内部的所有节点都保存了一份完整的数据**。其中一个主节点用来做集群管理提供写数据服务，其他的从节点用来同步数据，提供读数据服务。这些从节点必须保持和主节点的数据状态一致。



黑色的线：客户端发送请求给服务端

蓝色的线：Follower转发客户端提交的写数据请求给 Leader

红色的线：Leader 广播 Proposal 给所有 Follower 服务器

黄色的线：所有 Follower 服务器返回 Ack 消息给 Leader

绿色的线：Server 发送提交事务的 Commit 命令给所有执行 Proposal 的 Follower 节点

1. ZooKeeper 是对等架构，工作的时候，会举行选举，变成 Leader + Follower 架构
2. 在 ZooKeeper 中，没有沿用 Master/Slave(主备)概念，而是引入了 Leader、Follower（正常合法公民）、Observer（剥夺政治权利） 三种角色概念。通过 Leader 选举算法来选定一台服务器充当 Leader 节点，Leader 机器为客户端提供读写服务，其他角色，也就是 Follower 提供读服务，在提供读服务的 Follower 和 Observer 中，唯一区别就是 Observer 不参与 Leader 选举过程，没有选举权和被选举权，因此 Observer 的作用就是可以在不影响写性能情况下提高集群读性能。
3. ZooKeeper 系统还有一种角色叫做 Observer，Observer 和 Follower 最大的区别就是 Observer 除了没有选举权 和 被选举权 以外，其他的和 Follower 完全一样
4. ZooKeeper 系统的 Leader 就相当于是一个全局唯一的分布式事务发起者，其他所有的 Follower 是事务参与者，拥有投票权
5. ZooKeeper 集群的最佳配置：比如 5,7,9,11,13 个这样的总结点数，Observer 若干，Observer 最好是在需要提升 ZooKeeper 集群服务能力的再进行扩展，而不是一开始就设置 Observer 角色！Follower 切记不宜太多！
6. Observer 的作用是 分担整个集群的读数据压力，同时又不增加分布式事务的执行压力，因为分布式事务的执行操作，只会在 Leader 和 Follower 中执行。Observer 只是保持跟 Leader 的同步，然后帮忙对外提供读数据服务，从而减轻 ZooKeeper 的数据读取服务压力。
7. ZooKeeper 中的所有数据，都在所有节点保存了一份完整的。所以只要所有节点保持状态一致的话，肯定是所有节点都可以单独对外提供读服务的。
8. ZooKeeper 集群中的所有节点的数据状态通过 ZAB 协议保持一致。ZAB 有两种工作模式：
    1. 崩溃恢复：集群没有 Leader 角色，内部在执行选举
    2. 原子广播：集群有 Leader 角色，Leader 主导分布式事务的执行，向所有的 Follower 节点，按照严格顺序广播事务
    3. 补充一点：实际上，ZAB 有四种工作模式，分别是：ELECTION，DISCOVERY，SYNCHRONIZATION，BROADCAST，源码剖析的时候详细讲解
9. ZooKeeper 系统中的 Leader 角色可以进行读，写操作，Follower 角色可以进行读操作执行，但是接收到写操作，会转发给 Leader 去执行。ZooKeeper 的所有事务操作在 Zookeeper 系统内部都是严格有序串行执行的。
10. ZooKeeper 系统虽然提供了存储系统（类文件系统：树形结构，每个节点不是文件也不是文件夹，是一个 znode ），但是这个存储，只是为自己实现某些功能做准备的，而不是提供出来，给用户存储大量数据的，所以，切忌往 ZooKeeper 中存储大量数据，甚至每个 Znode 节点的数据存储大小不能超过 1M
11. ZooKeeper 提供了 znode 节点的常规的增删改查操作，使用这些操作，可以模拟对应的业务操作，使用监听机制，可以让客户端立即感知这种变化。

12. ZooKeeper 集群和其他分布式集群最大的不同，在于 ZooKeeper 是不能进行线性扩展的。因为像 HDFS 的集群服务能力是和集群的节点个数成正比，但是 ZooKeeper 系统的节点个数到一定程度之后，节点数越多，反而性能越差。
13. ZooKeeper 实现了 A可用性、P分区容错性、C中的写入强一致性，丧失的是C中的读取一致性。ZooKeeper 并不保证读取的是最新数据。如果客户端刚好链接到一个刚好没执行事务成功的节点，也就是说没和 Leader 保持一致的 Follower 节点的话，是有可能读取到非最新数据的。如果要保证读取到最新数据，请使用 sync 回调处理。这个机制的原理：是先让 Follower 保持和 Leader 一致，然后再返回结果给客户端。
14. Leader 的内部，有三个服务端！
    1. 服务端1：为 Follower 和 Observer 提供同步服务
    2. 服务端2：为 Client 提供服务
    3. 服务端3：ZooKeeper 集群内部的选举机制

有一些知识点，涉及到非常多的细节，会在源码分析中讲到：

- 选举
- 状态同步
- 读写机制

YARN : ResourceManager 内部有很多服务端，但是有非常重要的四个服务端： NM，AM，JobClient, Admin

3.1.2. ZNode 数据模型

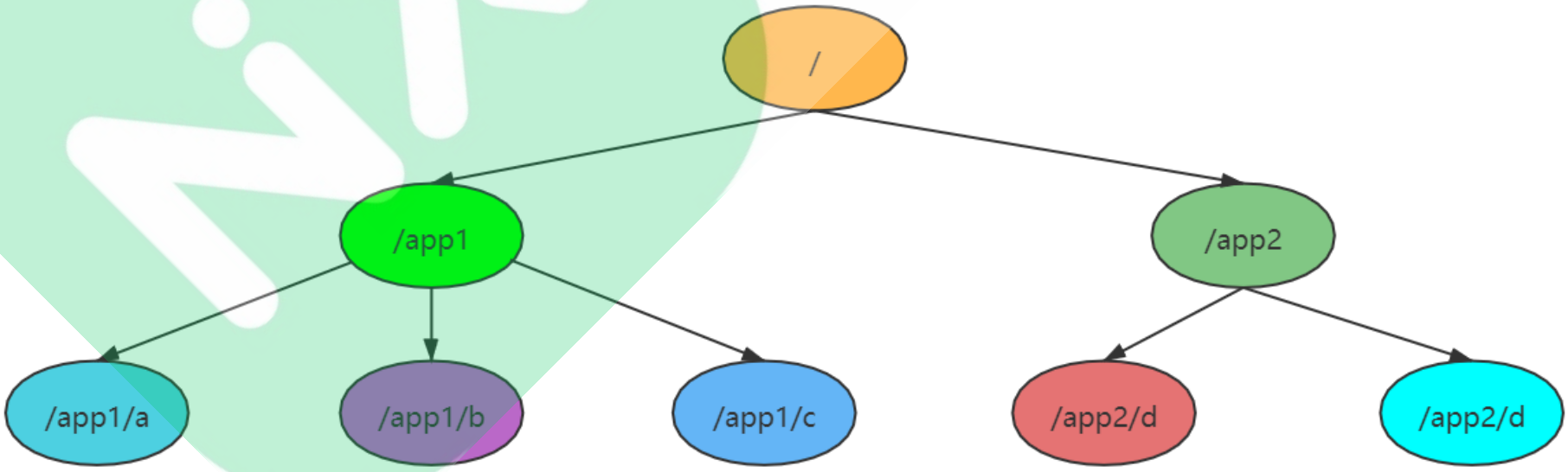
```
interface INode{}
class File implements INode{
    String data;
}
class Directory implements INode{
    List<INode> children
}

class Tree{
    Directory root;
}
```

```
class DataTree{
    DataNode root;
}
class DataNode{
    String data;
    DataNode parent;
    List<DataNode> children;
}
```

ZooKeeper 内部提供了一个数据存储系统，ZNode 数据模型是一个类 FileSystem 文件系统（为什么叫做类文件系统呢？因为正常文件系统中的每个节点要么是 directory 要么是 file，但是 znode 数据模型中的节点，没有这种区分，只有一个统一的概念，叫做 znode，但是既能存储数据，也能挂载子节点。所以既是文件夹，也是文件）

总结：ZooKeeper 的数据模型系统是一个类文件系统结构，每个节点称之为 ZNode，具体代码实现类是 DataNode，既不是文件夹，也不是文件，但是既具有文件夹的能力，也具有文件的能力。



如果自己要实现一棵树：

```
// DataNode 是这棵树上的一个节点的抽象，包含三个重要信息：可以存储数据，可以挂载一堆子节点，属于那个父 DataNode
class DataNode{
    // 存储节点的数据
    private Byte[] data;
    // 可以挂载一堆子节点
    private List<DataNode> children;
    // 当前节点有一个唯一的父节点，根节点除外
    private DataNode parent;
}
```

具体实现，可参考作业要求，也可以参考 ZooKeeper 源码实现！

关于 ZNode 的理解，我总结三个方面：

1、ZNode 的约束（ZNode 的节点存储的最大数据是 1M，最好不要超过 1kb）为什么？ZNode 的深度没有约束

```
每个节点都有相同的数据视图：每个节点都存储了这个 ZooKeeper 中的所有数据，每个节点的数据状态都和 leader 保持一致
1、同步的压力：写入过程至少要超过半数节点写成功才能认为该数据写成功，节点数越多，成功的难度越大
2、存储的压力：所有数据的规模超出单台服务器的存储能力
```

2、ZNode 的分类（按照生命周期分，或者按照是否带序列编号分）

```
1、按照生命周期：【临时节点 EPHEMERAL】 和 【永久节点 PERSISTENT】
    1、持久类型（显示创建，显示删除，只有当使用显示命令来删除节点，否则这个节点知道被创建成功，则会一直存在）
    2、临时类型/短暂类型（跟会话绑定，那个会话创建的这个节点，如果这个会话断开，则这个会话创建的所有临时节点被系统删除）。
        每个znode节点必然都是由某一个 session 创建的。如果当前这个 session 断开，那么该 znode 节点会被自动删除

    比如 HDFS ，Kafka，HBase, 等各种组件使用 ZooKeeper 的时候，都是先建立跟 ZooKeeper 的会话链接
    通过这个会话链接去创建一个 临时 znode 节点。如果这个链接断开，则 ZooKeeper 系统会自动删掉这个链接创建的所有临时节点
    MySQL： insert into table () values();
    client链接 mysql:  插入了一条临时数据，如果链接断开，这条数据就被自动删除了，如果链接一直存在则这条数据一直存在

2、按照是否带序列编号分，可以分为【带顺序编号】，和【不带顺序编号】两种类型的几点。每个节点都各自维护了一个序列编号，当前节点的序列编号是由它的父节点维护的，编号是自增序列编号，和 MySQL 的自增主键是一样的道理
    1、带      /a0000000001,/a0000000002
    2、不带    /a
    create /a "data"

3、总结一下，总共分成四种：
    1、CreateMode.PERSISTENT                    持久
    2、CreateMode.PERSISTENT_SEQUENTIAL         持久带序列编号
    3、CreateMode.EPHEMERAL                     临时
    4、CreateMode.EPHEMERAL_SEQUENTIAL          临时带序列编号
```

3、ZNode 的小知识补充

```
1、临时节点的下面不能挂载子节点，临时节点，只能作为叶子节点，临时节点的生命周期和会话绑定
2、其实 ZooKeeper-3.5 之后还提供了 Container 和 TTL 类型的节点，进一步的丰富了功能，在做源码分析的时候，具体讲解。
3、每个 ZNode 节点都有一个独一无二的绝对路径，没有相对路径的表示方式
4、如果创建的是带序列编号的节点，这个序列编号是由当前节点的父节点维护的
```

经典的用法：尽量少往 ZooKeeper 中写数据，写入的数据也不要特别大！ZooKeeper 只适合用来存储少量的关键数据！比如代表一个集群中到底谁是真正 active leader 的信息数据。为什么不适合写入大量数据的原因：

```
1、因为 ZooKeeper 系统内部每个节点都会做数据同步，在执行写请求的时候，事实上就是原子广播。待写入数据越大，原子广播的效率就越低，成功难度也越大。

2、所有的请求，都是严格的顺序串行执行，这个 ZooKeeper 集群在某一个时刻只能执行一个事务，如果上一个事务执行耗时，则会阻塞后面的请求的执行。

3、正因为每个节点都会存储一份完整的 ZooKeeper 系统数据，所以如果系统数据过大，甚至超过了单个 Follower 的存储能力了，系统服务大受影响甚至崩溃。

4、ZooKeeper 的设计初衷，就不是为了给用户提供一个大规模数据存储服务，而是提供了一个为了解决一些分布式问题而需要进行一些状态存储的数据模型系统。
```

3.1.3. Watcher 监听机制

ZooKeeper 提供了数据的发布订阅功能，多个订阅者（客户端）可同时监听某一特定主题对象（ZNode节点），当该主题对象的自身状态发生变化时（例如节点内容发生变化，节点下的子节点列表发生个数变化），系统会主动通知订阅者。

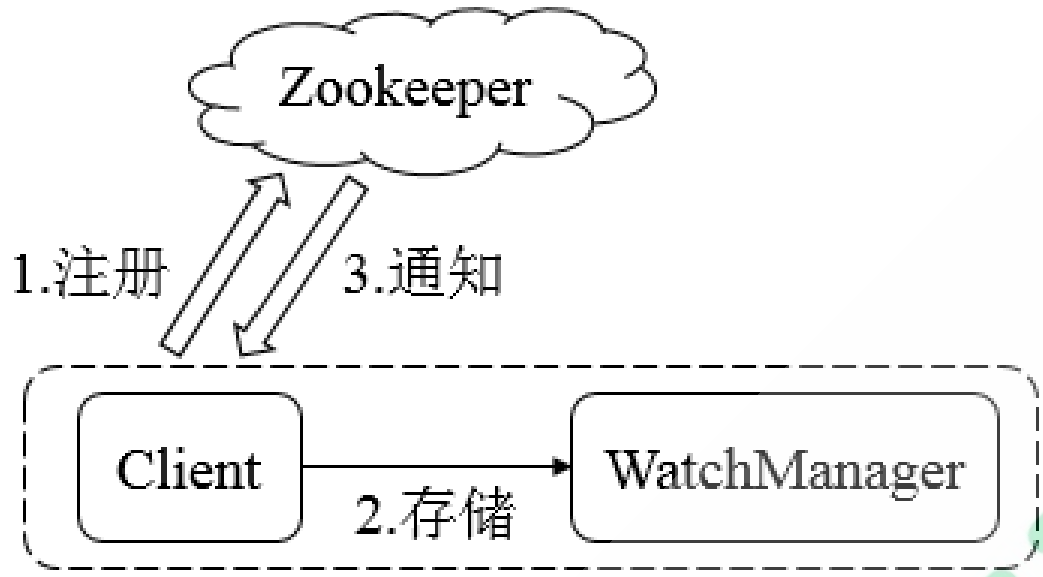
角色	客户端A	客户端B	系统/服务端
第一个例子	老板	访客	前台
第二个例子	学生	老师	公告栏
第三个例子	订阅者（普通用户）	发布者（小编）	系统（新闻App）

一句话总结：这种代码代码：观察者设计模式

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。Zookeeper 功能非常强大，可以实现诸如分布式应用配置管理、统一命名服务、状态同步服务、集群管理等

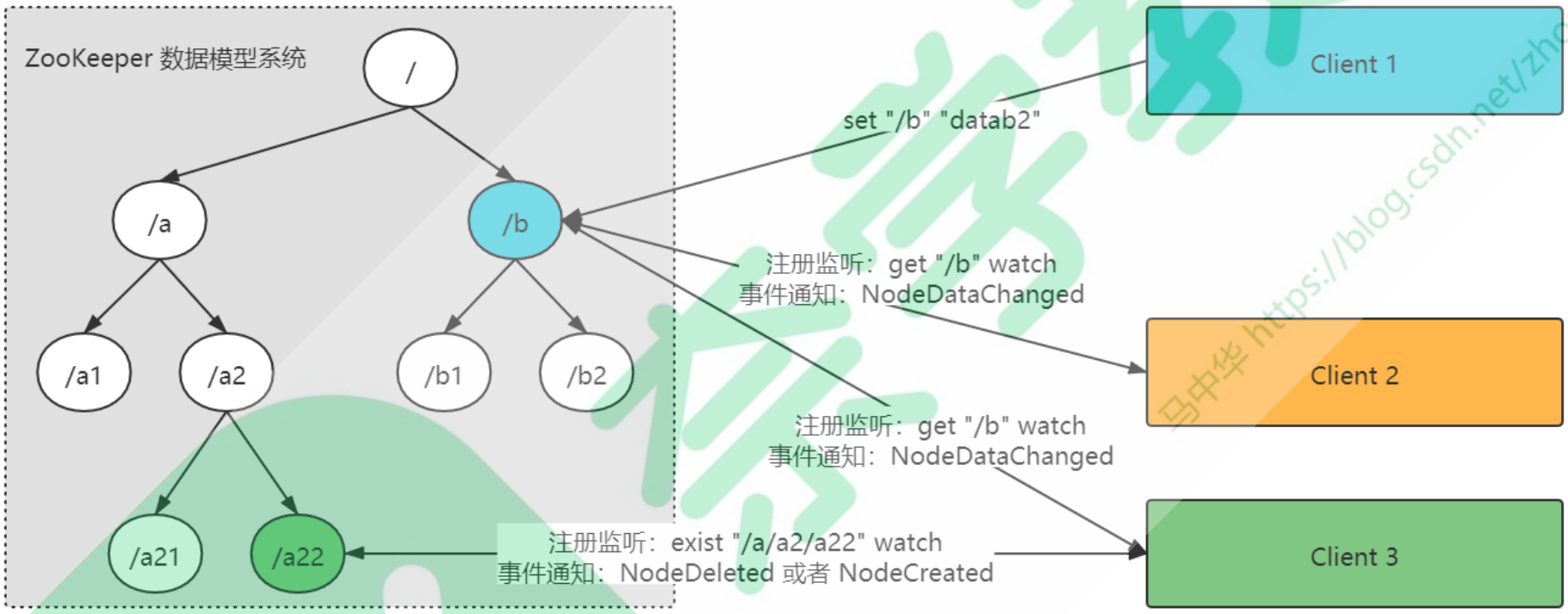
一个 Watch 事件是一个一次性的触发器，当被设置了 Watch 的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，以便通知它们。

ZooKeeper 的监听机制的工作原理：



客户端中，还存在一个 WatchManager 管理服务： 当前这个客户端到底注册了那些节点的那些监听都被分门别类的进行了管理。当这个客户端接收到 ZooKeeper 系统的事件通知：WatchedEvent， 那么 WatchManager 就会根据这个事件对象内部的 znodePath + type + state 来确定后续操作是什么。

我觉得官网的这个图太弱了，所以画了个新的：



1、Client2 关心 /b 节点的数据变化，所以通过 get /b watch 的方式，注册一个 Watcher，如果 Client 1 修改了 /b znode 节点的数据导致 /b 数据发生变化，则 ZK 系统会给注册了 /b 节点监听的 Client 2 一个事件通知，这样的话，Client2 是订阅，Client1 是发布

2、注意：注册监听的方式有多种，不同的方式关注不同的 ZK 系统变化，每一种变化都被抽象成一种具体的事件，所以事件也有多种

3、注册监听的方式：getChildren() + exists() + getData()

4、事件的类型有：NodeCreated + NodeDeleted + NodeDataChanged + NodeChildrenChanged

5、触发监听的方式：create() + delete() + setData()

ZooKeeper 采用 Watcher 机制实现了发布/订阅功能。该机制在被订阅者对象发生变化的时候会异步的通知客户端，因此客户端不必在 Watcher 注册后轮询阻塞，从而减轻客户端的压力。

客户端首先将 Watcher 注册到服务器上，同时将 Watcher 对象保存在客户端的 Watch 管理器中，当 Zookeeper 服务端监听的数据状态发生变化时，服务端会首先主动通知客户端，接着客户端的 Watch 管理器会触发相关 Watcher 来回调相应的处理逻辑，从而完成整体的数据发布/订阅流程。

监听器 Watcher 的定义：

```
// 监听器抽象
interface Watcher{
    // 该方法就是接收到服务端事件通知的监听回调方法。参数 event 有三个信息：
    // 1、KeeperState state      zk链接的状态
    // 2、String znodePath       发生事件的znode节点
    // 3、EventType type         事件的类型
    void process(WatchedEvent event) throw Execption;
}
```

客户端通过不同的方法注册 Watcher 的时候，会被抽象成不同的 Watcher 注册对象，然后放在 Request 中发送给 ZooKeeper 服务端：

序号	方法	意义	Watcher 注册对象
1	getData()	关注节点的数据变化	DataWatchRegistration
2	exists()	关注节点的存在与否的状态变化	ExistsWatchRegistration
3	getChildren()	关注节点的子节点个数变化	ChildWatchRegistration

不同的方法/动作，触发不同的监听响应，服务端给客户端返回的事件类型：

序号	方法	意义	事件类型	事件意义
1	setData()	更改节点数据事件，触发监听	EventType.NodeDataChanged	节点数据变化
2	create()	创建节点事件，触发监听	EventType.NodeCreated EventType.NodeChildrenChanged	节点创建 子节点个数变化
3	delete()	删除节点事件，触发监听	EventType.NodeDeleted EventType.NodeChildrenChanged	节点删除 子节点个数变化

这三个对象，是客户端根据用户调用不同的 方法注册的监听，他底层生成的不同的 Watch 注册对象，这个对象是需要发送给服务端的，只要发送给服务端，服务端就可以知道是哪个客户端对于哪个节点的什么变化感兴趣（注册监听），如果系统发生对应的变化，就可以通知客户端：返回 WatchedEvent

汇总一下，就是如下这张图：

创建Watch的API	触发事件				
	Create		Delete		setData
	znode	child	znode	child	znode
exists()	NodeCreated		NodeDeleted		NodeDataChanged
getData()			NodeDeleted		NodeDataChanged
getChildren()		NodeChildrenChanged	NodeDeleted	NodeChildrenChanged	NodeDataChanged

NodeCreated	创建节点时触发该事件
NodeChildrenChanged	当前节点下创建或者删除子节点触发该事件，修改子节点的数据不触发该事件
NodeDeleted	节点删除时触发该事件
NodeDataChanged	当前节点的数据被修改的时候触发该事件

```
注册监听：zk1.getChildren("/xxx/parent", watcher)   ==> 客户端 zk1 通过 getChildren 注册了一个监听： "/xxx/parent" 的子节点个数变化
触发监听：
    zk2.setData("/xxx/parent", "newdata")              ==> 触发客户端 zk1 就会收到 NodeDataChanged 事件
    zk2.createNode("/xxx/parent/son1")                 ==> 触发客户端 zk1 接收到 NodeChildrenChanged
    zk2.deleteNode("/xxx/parent/son2")                 ==> 触发客户端 zk1 接收到 NodeChildrenChanged
    zk2.setData("/xxx/parent/son2", "newData")         ==> 触发客户端 zk1 不会收到通知。通知监听了 /xxx/parent/son2 的客户端
```

见案例：

```java
package com.mazh.nx.zookeeper.core;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;

import java.io.IOException;

/**
 * Author：奈学教育
 * Description： 一个简单的 ZooKeeper 监听器应用程序编写规范！
 */
public class SimpleZKWatcherTest {

    static ZooKeeper zk = null;

    public static void main(String[] args) throws KeeperException, InterruptedException, IOException {
        // 请开始你的表演！

        // TODO_MA 注释： 第一步： 获取链接
```

```
        // 该抽象可以理解成：链接对象/会话对象/客户端
        // new ZooKeeper("bigdata02:2181,bigdata03:2181,bigdata04:2181", 5000, defaultWatcher)
        zk = new ZooKeeper("bigdata02:2181,bigdata03:2181,bigdata04:2181", 5000, new Watcher() {

            // TODO_MA 注释： 第三步： 回调方法
            // 如果说这个方法被调用了，就是意味着这个客户端收到了系统的一个响应事件
            // zk 系统发送事件通知到 客户端，是要走网络传输的。 但是网络是不可靠的（网络有延迟，或者数据丢回）
            @Override
            public void process(WatchedEvent event) {

                // 将接下来要执行的业务操作，疯转成一个任务，提交给线程池执行

                // 编写业务回调逻辑，这个地方为什么做 if else 判断，就是因为不同节点发生了不同的事件，必然会有不同的逻辑处理
                // 假设： 每隔 0.1 s 修改一次数据，但是在某一个监听响应中， process 执行了 1s 钟
                String path = event.getPath();
                Event.EventType type = event.getType();

                if(path.equals("/node_path") && type == Event.EventType.NodeDataChanged){
                    // 业务回调
                    // ...........

                    // 此处注册监听的目的，就是为了实现循环监听！
                    try {
                        byte[] data = zk.getData("/node_path", true, null);
                        System.out.println("回调中获取到的结果：" + new String(data));
                    } catch(KeeperException e) {
                        e.printStackTrace();
                    } catch(InterruptedException e) {
                        e.printStackTrace();
                    }
                }else if(path.equals("/node_path") && type == Event.EventType.NodeChildrenChanged){
                    // NodeChildrenChanged 只关心当前节点的子节点的个数是否发生了变化，不关心子节点的数据有没有发生变化
                }
            }
        });

        // TODO_MA 注释： 第二步： 注册监听，监听 /node_path 节点的数据是否改变
        // 两件事：获取节点的数据 + 给当前节点注册了一个监听
        // 通俗的话理解：我执行这句话代码，也就意味着，我对 zk 系统中的 /node_path 节点的数据的变化感兴趣，所以我注册了一个监听
        // 更通俗的：我告诉了 zk 系统，如果这个 znode 节点的数据发生了改变，我希望你能告诉我
        // 告诉我的方式，就是 zookeeper服务端会发送一个事件通知 WatchedEvent 过来,然后客户端去回调对应的监听方法 process()
        byte[] data1 = zk.getData("/node_path", true, null);
        byte[] data2 = zk.getData("/node_path", new Watcher(){
            @Override
            public void process(WatchedEvent event) {
                // 业务逻辑
            }
        }, null);
    }
}
```

最后一个小问题：请问，怎么实现循环监听/连续/反复监听？因为 ZooKeeper 的监听只会响应一次。

3.2. ZooKeeper 应用场景

ZooKeeper：分布式协调服务，劝架者，仲裁机构。基于它提供的两大核心功能：可以实现分布式场景中的各种疑难杂症！比如最经典的分布式锁的问题。

```
1、发布/订阅
2、命名服务
3、配置管理
4、集群管理
5、分布式锁
6、队列管理
7、负载均衡
```

3.2.1. 发布订阅

发布：触发事件发生的角色

订阅：接收事件结果的角色

应用服务器集群可能存在两个问题：

```
1、因为集群中有很多机器，当某个通用的配置发生变化后，怎么自动让所有服务器的配置同时生效？配置管理的问题
2、当集群中某个节点宕机，如何让集群中的其他节点感知？集群管理的问题
3、当集群中的 active Leader 宕机之后，怎么能让所有的 slave 快速感知新 active leader？
```

为了解决这两个问题，ZooKeeper 引入了 Watcher 机制来实现发布/订阅功能，能够让多个订阅者同时监听某一个主题对象，当这个主题对象自身状态发生变化时，会通知所有订阅者。

```
1、订阅者：在 zookeeper 这种架构实现中，其实就是 注册监听的客户端
2、发布者：在 zookepeer 这种架构实现中，其实就是 触发事件发生的客户端
```

数据发布/订阅即所谓的配置中心：发布者将数据发布到 ZooKeeper 的一个或一些列节点上，订阅者进行数据订阅，可以即时得到数据的变化通知。

消息/数据的发送有2种设计模式，推Push & 拉Pull。A 有一条消息，要让 B 知道：要么 A 主动告诉 B，要么 B 不断的来询问有没有新的消息，如果有，B 就能拿到。在推模中，服务端将所有数据更新发给订阅的客户端，而拉是由客户端主动发起请求获取最新数据。通常采用轮寻。比如 MapReduce 中的 MapTask 和 ReduceTask 之间的数据传输，采用的就是 拉取 的方式，大家可以思考一下，这是为什么?

ZooKeeper 采用推拉结合，客户端向服务端注册自己需要关注的节点的事件，一旦该节点数据发生该事件，服务器向客户端发送事件 Watcher 通知，客户端收到消息主动从服务端获取最新数据。**这种模式主要用于配置信息获取同步。**

3.2.2. 命名服务

通俗的理解，可以理解成是一个 唯一ID 生成服务

命名服务是分布式系统中较为常见的一类场景，分布式系统中，被命名的实体通常可以是集群中的机器、提供的服务地址或远程对象等，通过命名服务，客户端可以根据指定名字来获取资源的实体、服务地址和提供者的信息。ZooKeeper 也可帮助应用系统通过资源引用的方式来实现对资源的定位和使用，广义上的命名服务的资源定位都不是真正意义上的实体资源，在分布式环境中，上层应用仅仅需要一个全局唯一的名字。**ZooKeeper 可以实现一套分布式全局唯一 ID 的分配机制。**

Zookeeper 系统中的每个 znode 都有一个绝对唯一的路径！所以只要你创建成功了一个 znode 节点，也就意味着，你命名了一个全局唯一的名称！ 这是一种方式，另外，你也可以通过创建带顺序编号的节点来搞定：成功创建的带顺序编号的节点的名称，就是你要的命名。

```
create /order "aaa"
create -s /order/gid "bbbb"
```

由于 Zookeeper 可以创建顺序节点，保证了同一节点下子节点是唯一的，所以直接按照存放文件的方法，设置节点，比如一个路径下不可能存在两个相同的文件名，这种定义创建节点，就是全局唯一ID。切记：这种方式只是说能实现，不推荐在生产环境使用。这种方式的实现，就是利用到了 ZooKeeper 关于事务请求的严格顺序处理的机制，由于有多台机制执行分布式事务的存在，那还不如使用一台高性能的服务器做所有请求的顺序处理高效呢，而且还可自由定义命名规则。

请关注：专门用来做命名服务的算法：Twitter 的 SnowFlake 雪花算法，美团的 Leaf，滴滴的 TinyID 等，UUID

3.2.3. 集群管理

HDFS集群：

- 主节点：namenode： bigdata02, bigdata03
- 从节点： datanode： bigdata02, bigdata03, bigdata04, bigdata05

HBase 集群：

- 主节点：HMaster： bigdata02, bigdata05
- 从节点： HRegionServer： bigdata02, bigdata03, bigdata04, bigdata05

为什么 HBase 的从节点死掉了之后， HMaster 能立马知道，但是 HDFS 集群就不行？ Hbase 就是通过 ZK 来进行集群管理的。HDFS 没有通过 ZK 来进行集群管理的

ZooKeeper 集群管理可以理解成两点：**是否有机器退出和加入（从节点 管理）、选举 Master（主节点 管理）**。首先看问题：

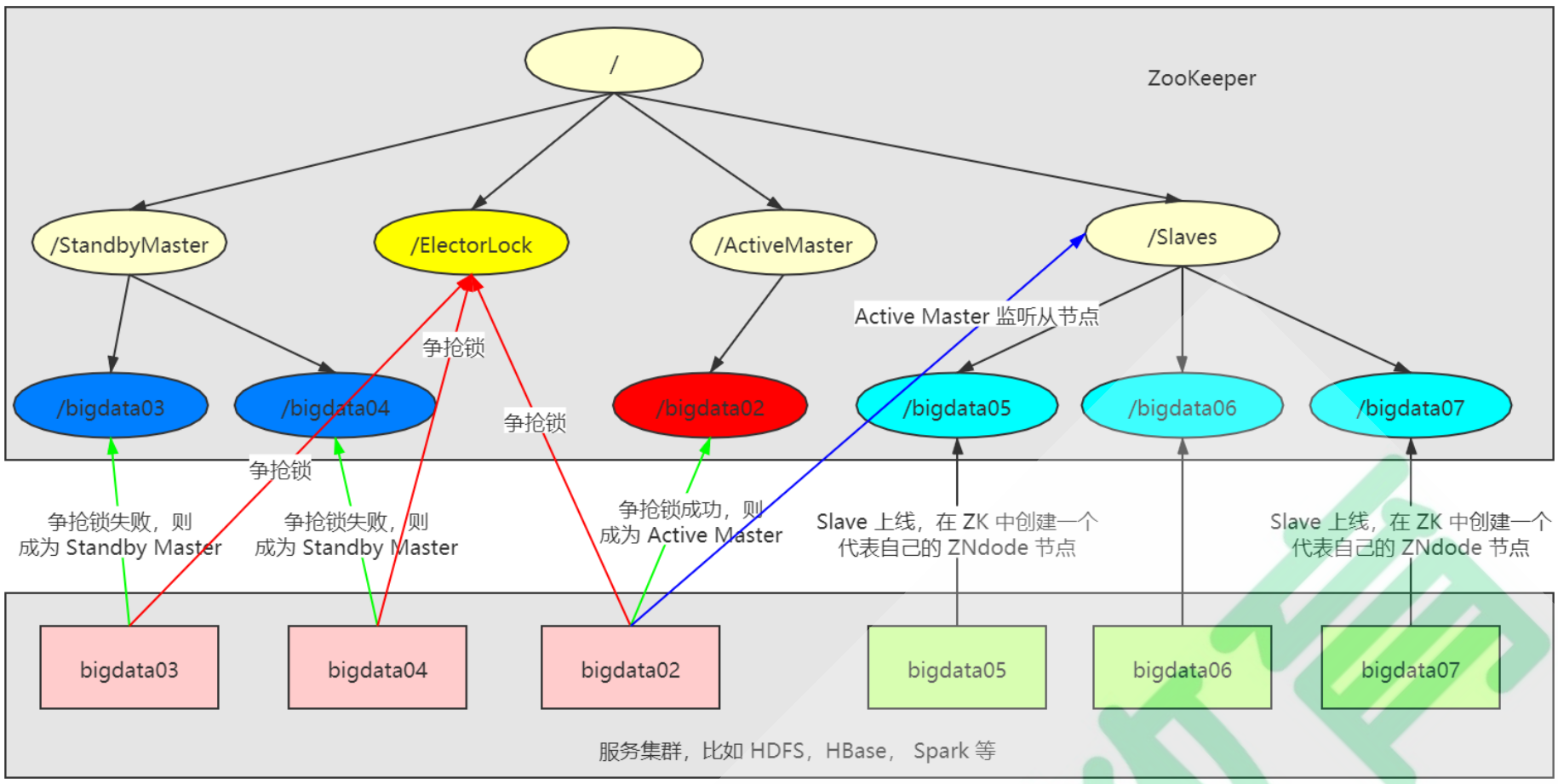
```
HDFS 中的 DataNode 如果死掉了，那么 NameNode 需要经过至少 630s 的默认时间，才会认为这个节点死掉！
HBase 中的 HRegionServer 如果死掉了，那么 HMaster 需要经过差不多 1s 的默认时间，才会认为这个节点死掉！
```

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，其所创建的代表该节点存活状态的临时目录节点被删除，所有其他机器都将收到通知：某个兄弟目录被删除，于是，所有人都知道：有兄弟节点挂掉了。新机器加入也是类似，所有机器收到通知：新兄弟目录加入，又多了个新兄弟节点。

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。当然，这只是其中的一种策略而已，选举策略完全可以由管理员自己制定。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。

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。（其实只要实现数据唯一性就可以做到选举，关系型数据库也可以，但是性能不好，设计也复杂）

利用 ZooKeeper 实现 集群管理：包括**集群从节点上下线即时感知管理**，和 **集群主节点选举管理**



注意：整个集群管理实现中，会有多种监听：
1、bigdata03,bigdata04 两个 master 由于是 Standby Master，所以需要监控 /ElectorLock 节点的删除事件，如果该节点被删除，意味着 Active Master 宕机
2、bigdata02 服务器需要监听 /Slaves 节点下的 子节点变化，增加了子节点，意味着新上线了 Slave，否则删除了 znode 节点的话，意味着宕机一台 Slave
3、bigdata05,bigdata06,bigdata07 需要监听 /ActiveMaster 节点下的唯一子节点的变化。如果 /ActiveMaster 下创建了一个子节点，意味着集群的 Active master 上线了，否则如果 /ActiveMaster 下的唯一 子 znode 被删除了，则意味着 active master 宕机了。
4、由于 active master 宕机，则 红色 /bigdata02 节点会被删除，同样，/ElectorLock 节点也会被删除。这样子，监听着 /ElectorLock 的 bigdata03 和 bigdata04 这两个 Standby Master 就会收到通知，然后参与选举，选举的具体实现是：多个 Standby 都尝试去创建 /ElectorLock 节点，谁创建成功了，就意味着，谁获取了成为 Active Master 的资格

3.2.4. 分布式锁

锁：并发编程中保证线程安全（一个 JVM 内部的多个线程的并发执行的安全）的一种机制：针对临界资源直接进行加锁的操作。 谁来操作，都需要先拿到钥匙！拿到操作许可！这个操作许可同时只能一个线程拿到。

分布式锁：分布式环境中，如果多个进程想要访问临界资源，则也需要进行加锁，但是比起单进程中的多线程加锁机制，分布式锁，还要考虑到网络通信的问题。为什么分布式环境中，各种问题会变得复杂，最大的原因，就是网络的不可靠：消息丢失 和 消息延迟

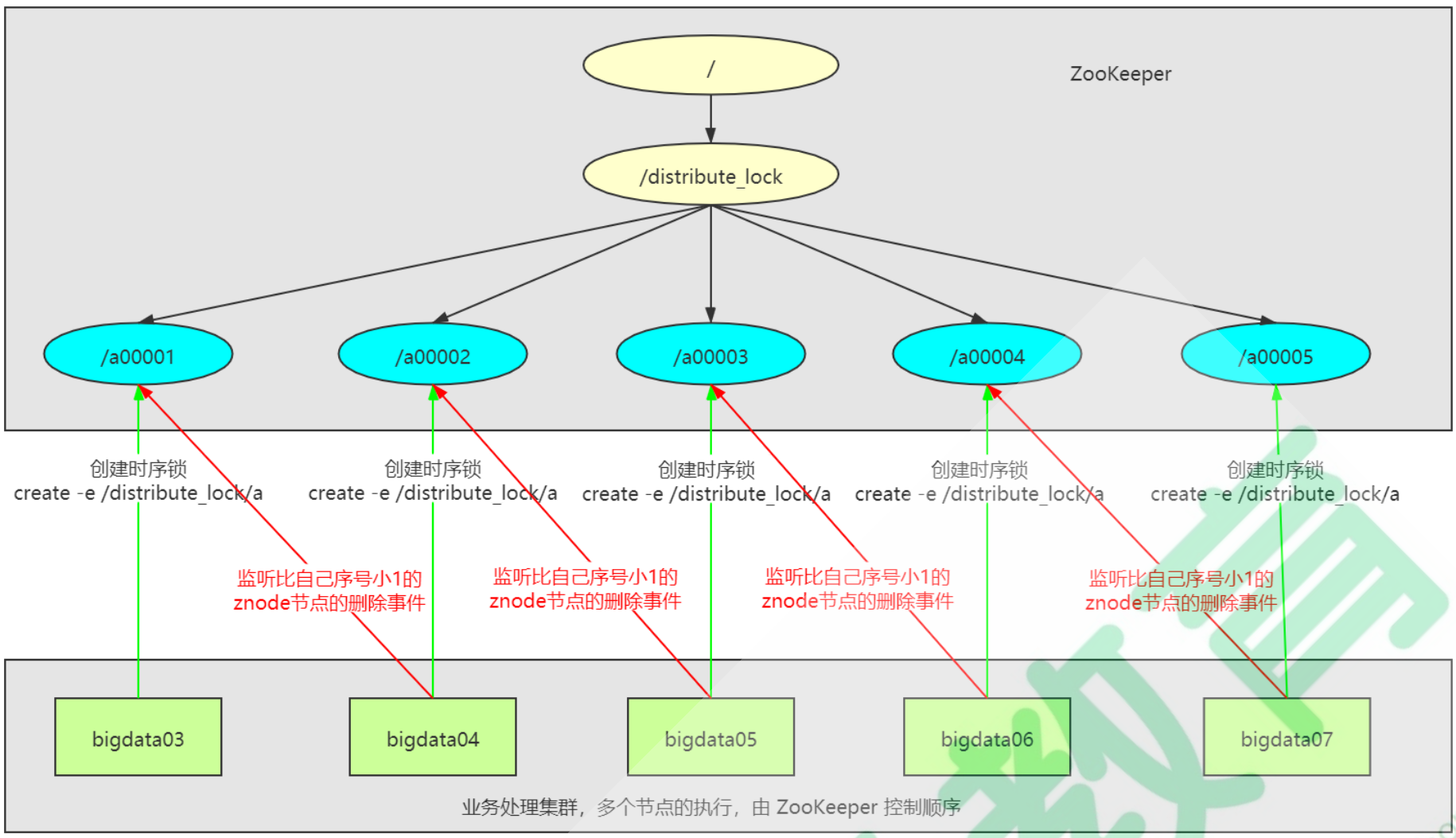
锁服务可以分为两三类： 独占锁，共享锁，时序锁

```
1、独占锁/写锁：对写加锁，保持独占，或者叫做排它锁，独占锁
2、共享锁/读锁：对读加锁，可共享访问，释放锁之后才可进行事务操作，也叫共享锁
3、时序锁：控制时序
```

对于第一类（独占锁），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代表拥有了这把锁。用完删除掉自己创建的 /distribute_lock 节点就释放出锁。

对于第二类（读写锁），我们在 ZooKeeper 上生成两个 znode，分别是：/lock_read 和 /lock_write，如果有一个客户端过来读取数据，则先判断 /lock_write 是否存在，如果不存在，则可以进行读取操作，同时创建一个 /lock_read 下的子节点代表读锁，读取完毕删除掉。如果有一个客户端过来写数据，则先判断 /lock_write 是否存在，再判断 /lock_read 下是否有读锁，如果都没有，则可以进行写操作。

对于第三类（时序锁），/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子节点，和选 Master 一样，编号最小的获得锁，用完删除，依次有序



叫号服务！bigdata03，bigdata04 他们就是一个工作线程！ 取得资格才能工作，他们采取叫号服务的方式来控制顺序

-e sequencetial 临时

-s persistent 带顺序编号

create -e /a "aa"

create -s /s "ss" ===> /s0000000001

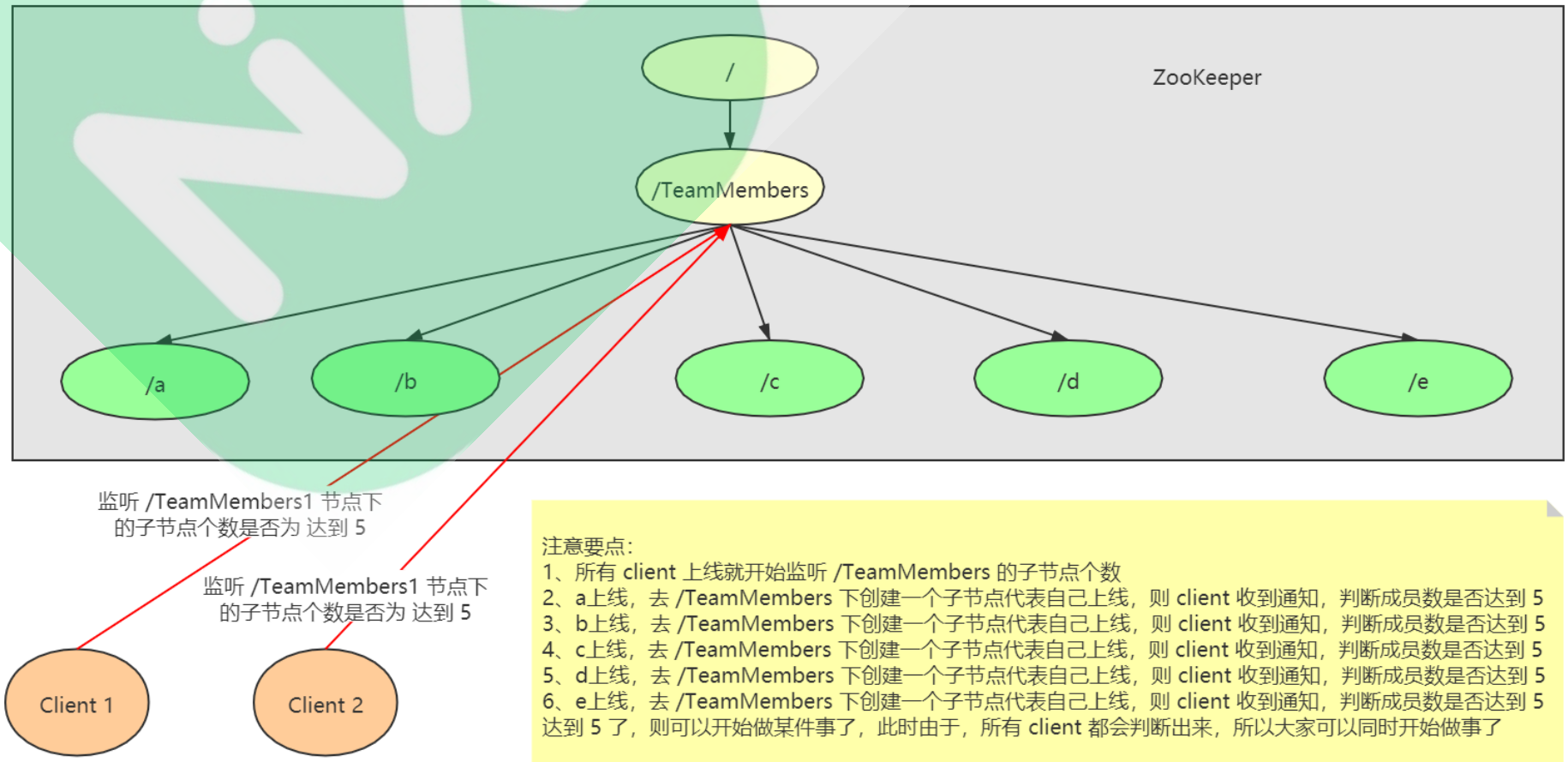
3.2.5. 分布式队列管理

两种类型的队列：

```
1、同步队列/分布式屏障/分布式栅栏：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、先进先出队列/顺序控制：队列按照 FIFO 方式进行入队和出队操作。和分布式时序锁一样。
```

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。



注意要点：
1、所有 client 上线就开始监听 /TeamMembers 的子节点个数
2、a上线，去 /TeamMembers 下创建一个子节点代表自己上线，则 client 收到通知，判断成员数是否达到 5
3、b上线，去 /TeamMembers 下创建一个子节点代表自己上线，则 client 收到通知，判断成员数是否达到 5
4、c上线，去 /TeamMembers 下创建一个子节点代表自己上线，则 client 收到通知，判断成员数是否达到 5
5、d上线，去 /TeamMembers 下创建一个子节点代表自己上线，则 client 收到通知，判断成员数是否达到 5
6、e上线，去 /TeamMembers 下创建一个子节点代表自己上线，则 client 收到通知，判断成员数是否达到 5
达到 5 了，则可以开始做某件事了，此时由于，所有 client 都会判断出来，所以大家可以同时开始做事了

3.2.6. 负载均衡

ZooKeeper 实现负载均衡本质上是利用 ZooKeeper 的配置管理功能，实现负载均衡的步骤：

```
1、服务提供者把自己的域名及 IP 端口映射注册到 zk 中。
2、服务消费者通过域名从 zk 中获取到对应的 IP 及端口，这里的 IP 及端口可能有多个，只是获取其中一个。
3、当服务提供者宕机时，对应的域名与 IP 的对应就会减少一个映射。
4、阿里的 dubbo 服务框架就是基于 zk 实现服务路由和负载均衡。
```

忠告：ZooKeeper 能实现负载均衡，但是跟 命名服务一样，不推荐使用！

3.2.7. 配置管理

我们都知道，HBase 和 Kafka 的架构设计实现中， ZooKeeper 的地位都非常的重要。他们都用 ZooKeeper 做了这么三件事：

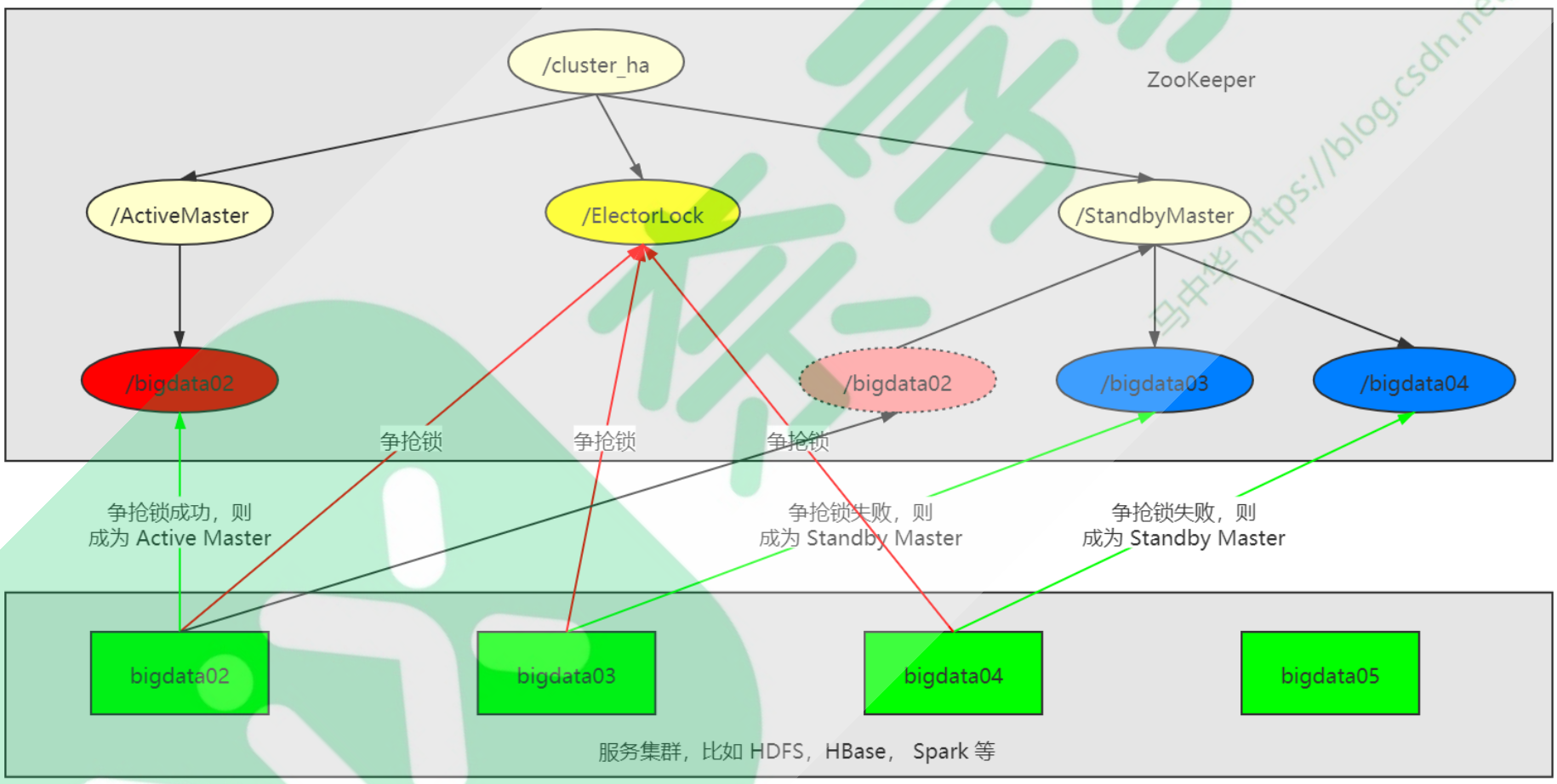
- 注册中心，利用 发布/订阅 功能实现
- 配置管理，集群中的相关重要元数据，就保存在 ZooKeeper 上
- HA 选举，HBase 的 HMaster，Kafka 中的 Controller 角色都是通过 ZooKeeper 来进行选举的

关于配置管理，后续有个代码演示可以让大家看看效果。

3.3. Zookeeper 最佳企业应用

3.3.1. 分布式独占锁实现 Active Master 选举

该分布式选举，所采用的是 分布式锁 的思路来实现的。整体架构思路：



整个选举机制，执行步骤解析：

初始状态：

```
/cluster_ha
    /ActiveMaster
    /StandbyMaster
```

bigdata02 上线，先自动成为 standby

```
/cluster_ha
    /ActiveMaster
    /StandbyMaster
        /bigdata02
```

然后去创建锁节点 ElectorLock

```
/cluster_ha
    /ActiveMaster
    /ElectorLock
    /StandbyMaster
        /bigdata02
```

如果 bigdata02 创建锁节点成功，则从 StandbyMaster 下删除自己，然后再 ActiveMaster 下创建自己

```
/cluster_ha
    /ActiveMaster
        /bigdata02
    /ElectorLock
    /StandbyMaster
```

到这个点为止，是 bigdata02 上线的状态：bigdata02 顺利成为 active

```
/cluster_ha
    /ActiveMaster
        /bigdata02
    /ElectorLock
    /StandbyMaster
```

如果 bigdata02 宕机，则对应 active znode 被删除

```
/cluster_ha
    /ActiveMaster
    /StandbyMaster
```

如果 bigdata02 为 active，然后 bigdata03 上线，则状态为：

```
/cluster_ha
    /ActiveMaster
        /bigdata02
    /ElectorLock
    /StandbyMaster
        /bigdata03
```

思路实现：

```
1、准备动作：Zookeeper 的状态：有 /ActiveMaster 节点和 /StandbyMaster 节点，但是下面都没有信息的，/ElectorLock 节点是不存在的

2、bigdata02 一上线，发现自动成为 standby 角色，所以把自己的信息注册到 /StandbyMaster 节点下

3、bigdata02 一上线，再去找 /ActiveMaster 节点下是否有子节点，如果有就证明有 active 的角色，如果没有，去争抢分布式锁，如果没有抢到，意味着，别人抢到了，别人成为 active，如果自己抢到了，则把自己的信息在 /StandbyMaster 里面删掉，再更新到 /ActiveMaster 节点下面

4、bigdata03 一上线，发现 /ActiveMaster 节点下面有子节点，有 active，就自动成为 standby 角色，会监听 /ActiveMaster znode 监听：NodeChildrenChagended。相当于告诉 ZooKeeper 系统，只要 /ActiveMaster 的子节点个数发生变化，系统就告诉我一声。 /ActiveMaster 下面只能有一个子节点。既然减少，那么 bigdata03 就会收到通知：原来的 active master 宕机了

5、bigdata04 上线，行为和 bigdata03 一致的

6、假设 bigdata02 宕机，bigdata04 跟 ZooKeeper 系统维持的会话就断开了，由于创建的锁和 /ActiveMaster 下面的子节点都是临时节点，当 bigdata02 一宕机，这两个节点就自动被 ZooKeeper 系统删除了， 由于 bigdata03 和 bigdata04 监听了这个事件，都会收到通知，则他们都知道了 active master 不在了，他们都去争抢成为 active 先去抢 /ElectorLock 锁。 最终创建这个 ElectorLock 锁成功的只会是一台服务器，所以谁创建成功，谁就成为 active ，没有创建锁节点 /ElectorLock 成功的就还是 standby master.
```

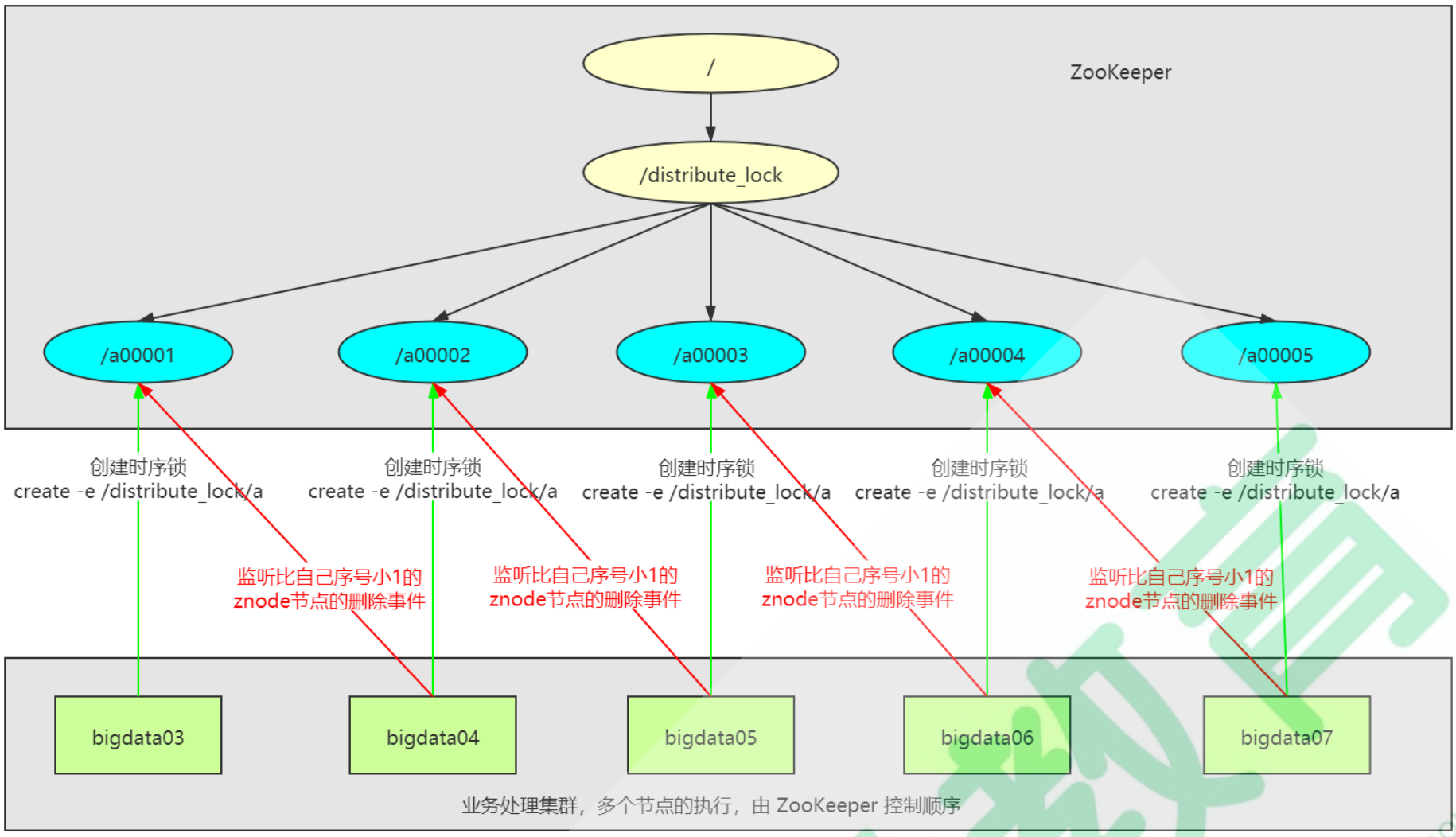
推荐：使用 Zookeeper的 API 框架： Curator 来实现 HA： LeaderLatch LeaderSelector , 减轻代码编写的复杂度。有兴趣的先研究，各大大数据技术组件，实现选举，都已经使用 Curator 来做封装实现了，比如 HBase，Flink 等。

3.3.2. 分布式时序锁实现分布式队列

其实解决方案有两种：但是这两种实现都讲的是独占锁的实现。独占锁的实现，在分布式选举的案例中，已经应用过了。

```
1、基于 Redis 的实现                    AP实现
2、基于 ZooKeeper的实现                 CP实现
```

所以在此，跟大家讲解一种分布式时序锁的实现：



3.3.3. 分布式集群配置管理

以 HDFS 的配置管理为例子，假设一个 HDFS 集群有 1000 个 datanode，现在管理员有一个操作：想把默认的 3 个副本这个参数，改成 4个副本，那么怎么高效实现呢?

```
现有简单方案：写一个脚本，修改配置文件，全局分发，然后滚动重启。
但是我最期望：如果我作为管理员，期望一种又简单又高效的做法：只要一更改某一个参数立即响应全局
```

结论：通过 ZooKeeper 的监听机制来实现，所有 datanode 都去监听 ZooKeeper 上的配置信息，如果任何一个客户端去更改了配置，则所有的 DataNode 都能立即收到通知，然后做相应回调处理即可。

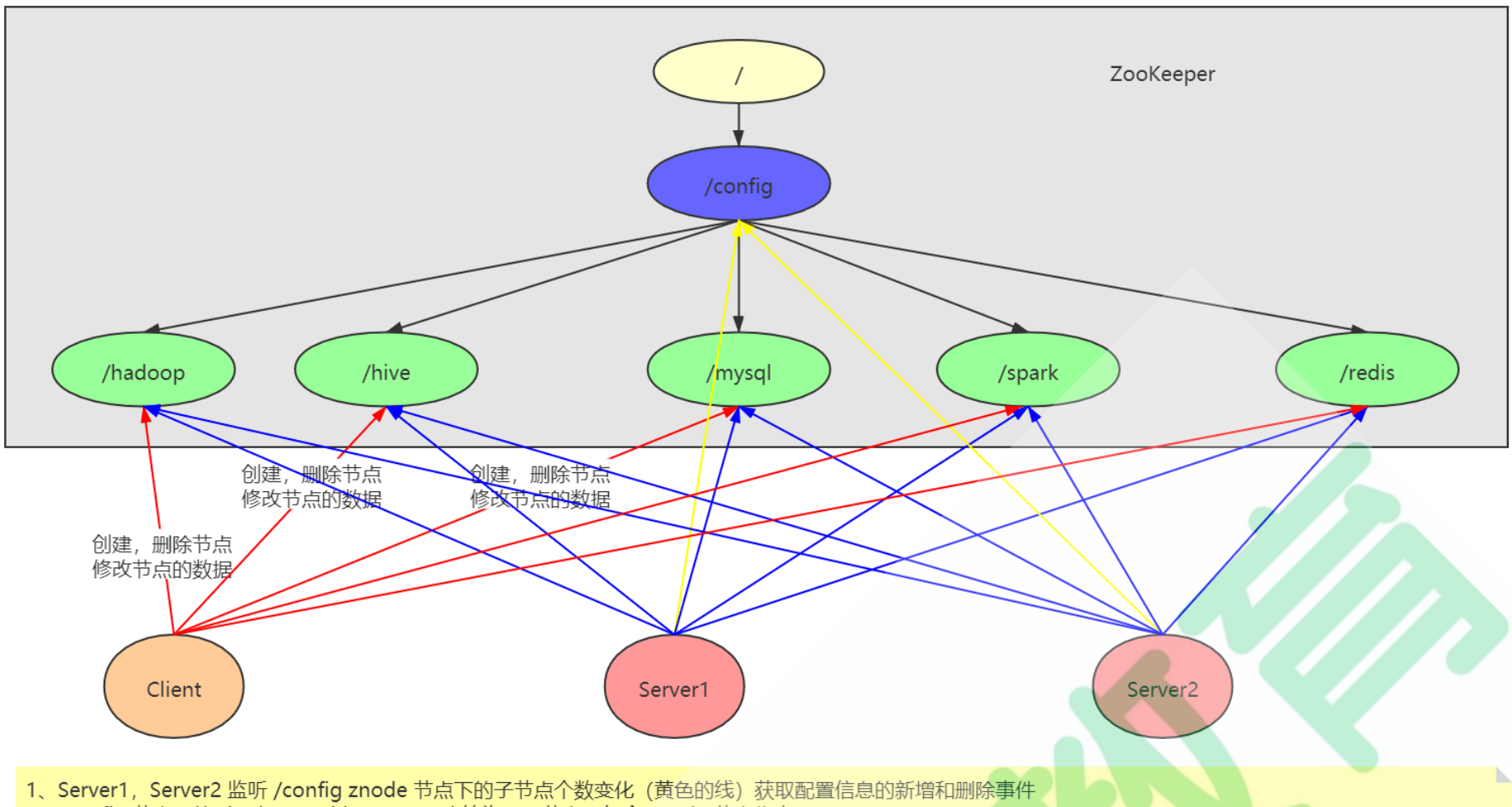
关于配置管理，一般涉及到三种常见动作：

```
1、新增配置
    两种方式：
    1、如果把所有的配置信息，都存储在一个节点，也就意味着，如果更改某一个配置，那么这个 znode 节点的值就发生了改变。 a:1-b:2 ==> a:1-b:2-c:3
        zk.getData()
    2、如果每一个配置，就通过一个 znode 节点来存储，znode 节点的名称是 key，znode 节点的值，就是 value，增加配置，其实就是多创建一个子 znode
        zk.getChildren()

2、删除配置
    两种方式：
    1、如果把所有的配置信息，都存储在一个节点，也就意味着，如果更改某一个配置，那么这个 znode 节点的值就发生了改变。 a:1-b:2-c:3 ==> a:1-b:2
        zk.getData()
    2、如果每一个配置，就通过一个 znode 节点来存储，znode 节点的名称是 key，znode 节点的值，就是 value，减少配置，其实就是删除一个子 znode
        zk.getChildren()

3、修改配置
    两种方式：
    1、如果把所有的配置信息，都存储在一个节点，也就意味着，如果更改某一个配置，那么这个 znode 节点的值就发生了改变。 a:1-b:2 ==> a:1-b:22
        zk.getData()
    2、如果每一个配置，就通过一个 znode 节点来存储，znode 节点的名称是 key，znode 节点的值，就是 value，修改配置，就是修改某个 znode 的数据
        zk.getData()
```

最终，我们采用了第二种方式，即使用一个 znode 来保存一项配置，最终的架构思路实现如下：



1、Server1，Server2 监听 /config znode 节点下的子节点个数变化（黄色的线）获取配置信息的新增和删除事件
2、/config 节点下的 /hadoop，/hive，/mysql 等为配置节点，每个 znode 节点代表了一项配置
3、Server1，Server2 监听 /hadoop，/hive，/mysql 等配置 znode 节点的数据变化事件（蓝色的线），感知配置信息的修改
4、Client 会发送请求到 ZooKeeper 集群中的 /config 节点下创建，或者删除，或者修改子节点的数据（红色的线）来进行配置的新增，删除，和修改动作

4. 本次课程总结

今天的主要内容有：

```
第一部分：ZooKeeper核心功能深度剖析
    01、ZNode数据模型
    02、Watcher监听机制

第二部分：Zookeeper企业级最佳应用实战
    03、应用场景分析
    04、实际企业案例的实现
```

企业应用案例的目的：

```
1、要知道一些常用的流行的分布式技术底层使用 ZooKeeper 到底干了什么以及我们能用 ZooKeeper 解决什么企业问题
2、自己知道基于ZooKeeper 实现这些企业需求的原理以及能用代码实现这些需求
3、为了将来研究其他技术组件的源码的执行细节打下扎实的基础
```

HBase，Spark，Flink 等实现 HA 就是 基于 ZooKeeper 。 到时候看源码的时候，都会涉及到基于 ZooKeeper 做 HA 的选举。

今天这节课的两个重点： 分布式独占锁 进行选举 + 集群管理 很多 分布式技术，都在用的一个方案

5. 本次课程作业

第一次课的作业：实现一个类，这个类中有四个抽象方法

```
package com.mazh.nx.zookeeper.exercise;

import org.apache.zookeeper.ZooKeeper;
import java.util.Map;

/**
 * 作者： 马中华：http://blog.csdn.net/zhongqi2513
 * 日期： 2021年04月09日 下午11:39:10
 *
 * 编程思维训练
 * 1、级联查看某节点下所有节点及节点值
 * 2、删除一个节点，不管有有没有任何子节点
 * 3、级联创建任意节点
 * 4、清空子节点
 *
 * 注意事项：
 * 1、关于方法名和参数列表，大家可以自行决定是否要修改。
```

```java
 * 2、请大家编写自己姓名的实现类，比如 ZKHomeWork_Zhangsan，实现 ZKHomeWork 接口，实现这四个方法。
 * 3、请大家写一个测试类，可以运行看效果的那种测试类
 */
public interface ZKHomeWork {

    /**
     * 级联查看某节点下所有节点及节点值
     */
    public Map<String, String> getChildNodeAndValue(String path, ZooKeeper zk) throws Exception;

    /**
     * 删除一个节点，不管有有没有任何子节点
     */
    public boolean rmr(String path, ZooKeeper zk) throws Exception;

    /**
     * 级联创建任意节点
     * /a/b/c/d/e
     */
    public boolean createZNode(String znodePath, String data, ZooKeeper zk) throws Exception;

    /**
     * 清空子节点
     */
    public boolean clearChildNode(String znodePath, ZooKeeper zk) throws Exception;
}
```