Zookeeper 面试题

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的 Zookeeper 面试题的大保健。

而题目的难度，尽量按照从容易到困难的顺序，逐步下去。

Zookeeper 是什么?

ZooKeeper 是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 具有如下特性：

- 顺序一致性(有序性)

  > 从同一个客户端发起的事务请求，最终将会严格地按照其发起顺序被应用到 Zookeeper 中去。
  >
  > 有序性是 Zookeeper 中非常重要的一个特性。
  >
  > - 所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid(Zookeeper Transaction Id)。
  > - 而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个 Zookeeper 最新的 zxid 。

- 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，即整个集群要么都成功应用了某个事务，要么都没有应用。

- 单一视图

  > 无论客户端连接的是哪个 Zookeeper 服务器，其看到的服务端数据模型都是一致的。

- 可靠性

  > 一旦服务端成功地应用了一个事务，并完成对客户端的响应，那么该事务所引起的服务端状态变更将会一直被保留，除非有另一个事务对其进行了变更。

- 实时性

  > Zookeeper 保证在一定的时间段内，客户端最终一定能够从服务端上读取到最新的数据状态。

Zookeeper 对于读写请求有所不同：

- 客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 Zookeeper 机器来处理。
- 对于写请求，这些请求会同时发给其他 Zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 Zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

Chubby 是什么？和 Zookeeper 对比你怎么看?

- Chubby 是 Google 的，完全实现 Paxos 算法，不开源。
- Zookeeper 是 Chubby 的开源实现，使用 ZAB 协议(Paxos 算法的变种)。

Zookeeper 的 Java 客户端都有哪些?

- Zookeeper 自带的 zkclient
- Apache 开源的 Curator

> 实际项目中，采用 Curator 居多。因为，功能更加强大。

具体的使用，可以看看 《ZK 客户端操作》 文章。

另外，Zookeeper 没有特别好用的 GUI 工具，有需要的胖友，可以看看 ZooInspector ，凑活能用。

Zookeeper 的设计目标?

- 1、简单的数据结构，Zookeeper 使得分布式程序能够通过一个共享的树形结构的名字空间来进行相互协调，即 Zookeeper 服务器内存中的数据模型由一系列被称为 ZNode 的数据节点组成，Zookeeper 将全量的数据存储在内存中，以此来提高服务器吞吐、减少延迟的目的。
- 2、可以构建集群 Zookeeper 集群通常由一组机器构成，组成 Zookeeper 集群的而每台机器都会在内存中维护当前服务器状态，并且每台机器之间都相互通信。
- 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，这个编号反映了所有事务操作的先后顺序。
- 4、高性能，Zookeeper 和 Redis 一样全量数据存储在内存中，100%读请求压测 QPS 12-13W 。

> 没具体测试过，比想象中的高。感兴趣的胖友，可以看看 《ZooKeeper 的一个性能测试》 。

Zookeeper 有哪些应用场景?

Zookeeper 的功能很强大，应用场景很多，结合我们实际工作中使用 Dubbo 框架的情况，Zookeeper 主要是做注册中心用。

- 基于 Dubbo 框架开发的提供者、消费者都向 Zookeeper 注册自己的 URL ，消费者还能拿到并订阅提供者的注册 URL ，以便在后续程序的执行中去调用提供者。
- 而提供者发生了变动，也会通过 Zookeeper 向订阅的消费者发送通知。

当然，Zookeeper 能提供的不仅仅如此，再例如：

- 统一命名服务。

> 命名服务是指通过指定的名字来获取资源或服务的地址，利用zk创建一个全局的路径，即时唯一的路径，这个路径就可以作为一个名字，指向集群中机器或者提供服务的地址，又或者一个远程的对象等。

- 分布式锁服务。

> 这个比较好理解，Zookeeper 实现的分布式锁的可靠性会比 Redis 实现的分布式锁高，当然相对来说，性能会低。

- 配置管理。

> 例如说，Spring Cloud Config Zookeeper ，就实现了基于 Zookeeper 的 Spring Cloud Config 的实现，提供配置中心的服务。

- 注册与发现。

  > 是否有机器加入或退出
  >
  > 所有机器约定在父目录下创建临时目录节点，然后监听父目录节点下的子节点变化。一旦有机器挂掉，该机器与 ZooKeeper 的连接断开，其所创建的临时目录节点也被删除，所有其他机器都收到通知：某个节点被删除了。

- Master 选举。

  > 基于 Zookeeper 实现分布式协调，从而实现主从的选举。这个在 Kafka、Elastic-Job 等等中间件，都有所使用到。

- 分布式锁。

  > 有了 ZooKeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分成两类，一个是保持独占，另一个是控制时序。
  >
  > - 1、保持独占，我们把 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也即拥有了这把锁。用完删除掉自己创建的 `/distribute_lock` 节点就释放出锁。
  > - 2、控制时序，`/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节点，和 Master 一样，编号最小的获得锁，用完删除，依次方便。

- 队列管理

  > 两种类型的队列。
  >
  > - 1、同步队列，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。在约定的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。
  > - 2、队列按照 FIFO 方式进行入队和出队操作。和分布式锁服务中的控制时序的场景基本原理一致，入列有编号，出列按编号。创建 PERSISTENT_SEQUENTIAL 节点，创建成功时 Watcher 通知等待的队列，队列删除序列号最小的节点以消费。此场景下，znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息丢失的问题。

当然，详细的可以看看 《Zookeeper 技术浅析》 文章。另外，该问对 Zookeeper 的“特点”介绍，也要重点看看。

上述的很多功能，在 Apache Curator 已经默认提供实现了，直接调用 API 即可使用。

作为服务注册中心，Eureka 比 Zookeeper 好在哪里?

参见 《作为服务注册中心，Eureka 比 Zookeeper 好在哪里》 文章。

比较重要的原因是，注册中心对可用性比一致性有更高的要求，也就是说，能够容忍在异常情况下，读取到几分钟前的数据。

Zookeeper 提供了什么?

- 1、文件系统。
- 2、通知机制。

Zookeeper 的文件系统是什么?

Zookeeper 提供一个多层级的节点命名空间(节点称为 znode)。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。

Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M 。

📣 Zookeeper 有哪几种节点类型?

- PERSISTENT 持久节点

  > 创建之后一直存在，除非有删除操作，创建节点的客户端会话失效也不影响此节点。

- PERSISTENT_SEQUENTIAL 持久顺序节点

  > 跟持久一样，就是父节点在创建下一级子节点的时候，记录每个子节点创建的先后顺序，会给每个子节点名加上一个数字后缀。

- EPHEMERAL 临时节点

  > 创建客户端会话失效（注意是会话失效，不是连接断了），节点也就没了。不能建子节点。

- EPHEMERAL_SEQUENTIAL 临时顺序节点

  > 基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

如下是整理的 Elastic-Job-Lite 使用 Zookeeper 作为存储的明细：

${namespace}		路径				用途	值	节点类型
	${JOB_NAME}	leader	election	instance		主节点信息	【空串】	临时节点
				latch		主节点选举锁		永久节点（分布式锁）
			sharding	necessary		重新分配作业分片标记	【空串】	持久节点
				processing		正在分配作业分片标识	【空串】	临时节点
			failover	items	${ITEM_INDEX}	作业分片失效转移标记	【空串】	永久节点
				latch		作业失效转移锁		永久节点（分布式锁）
		guarantee	started	${ITEM_INDEX}		开始执行作业分片	【空串】	永久节点
			completed	${ITEM_INDEX}		结束执行作业分片	【空串】	永久节点
		servers	${IP}			服务器信息	【空串】	永久节点
		config				Lite作业配置	LiteJobConfiguration JSON字符串	永久节点
		instances	${JOB_INSTANCE_ID			作业节点信息	【空串】	临时节点
		sharding	${ITEM_INDEX}	instance		分配的作业节点	${JOB_INSTANCE_ID}	
				running		作业分片运行中标记	【空串】	临时节点
				failover		作业分片失效转移标记	【空串】	临时节点
				misfired		作业分片被错过执行标记	【空串】	永久节点
				diabled		作业分片被禁用标记	【空串】	永久节点

Zookeeper 的通知机制是什么?

Zookeeper 允许客户端向服务端的某个 znode 注册一个 Watcher 监听，当服务端的一些指定事件触发了这个 Watcher ，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类型做出业务上的改变。

整个流程如下：

> 具体的过程，下面每个小问题，进行说明。

- 第一步，客户端注册 Watcher 。
- 第二步，服务端处理 Watcher 。
- 第三步，客户端回调 Watcher 。

Watcher 的特性总结：

- 1、一次性。

> 无论是服务端还是客户端，一旦一个 Watcher 被触发， Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。
>
> 😈 注意哟，这个特性可以变成一个面试题「Zookeeper 对节点的 watch 监听通知是永久的吗？」。
>
> 如果我们使用 Apache Curator 作为操作 Zookeeper 的客户端，它可以帮我们自动透明的实现持续的 watch 操作，非常方便。

- 2、客户端串行执行。

> 客户端 Watcher 回调的过程是一个串行同步的过程。

- 3、轻量级 Watch 机制。

> - Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。
> - 客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用`boolean`类型属性进行了标记。

- 4、Watcher event 异步发送 Watcher 的通知事件从 Server 发送到Client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过Socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee ，即客户端监听事件后，才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。**Zookeeper 只能保证最终的一致性，而无法保证强一致性。**
- 5、可以注册 Watcher 的操作：getData、exists、getChildren 。
- 6、可以触发 Watcher 的操作：create、delete、setData 。
- 7、当一个 Client 连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 Client 重新连接时，如果需要的话，所有先前注册过的watch ，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode 的 exists watch ，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

😈 看了这么多特性总结，最最最重要的是**【一次性】**。

> ：下面三个步骤，选择性了解即可。面试如果问到，就当倒霉。

📣 **第一步，客户端注册 Watcher 实现?**

- 1、调用 getData、getChildren、exist 三个 API ，传入Watcher 对象。
- 2、标记请求 request ，封装 Watcher 到 WatchRegistration 。
- 3、封装成 Packe t对象，发服务端发送 request 。
- 4、收到服务端响应后，将 Watcher 注册到 ZKWatcherManager 中进行管理。
- 5、请求返回，完成注册。

📣 **第二步，服务端处理 Watcher 实现?**

- 1、服务端接收 Watcher 并存储。

> 接收到客户端请求，处理请求判断是否需要注册 Watcher ，需要的话将数据节点的节点路径和 ServerCnxn(ServerCnxn 代表一个客户端和服务端的连接，实现了 Watcher 的 process 接口，此时可以看成一个 Watcher 对象)存储在 WatcherManager 的 WatchTable 和 Watch2Paths 中去。

- 2、Watcher 触发。

> 以服务端接收到 setData 事务请求触发 NodeDataChanged 事件为例：

> - 封装 WatchedEvent：
>
>> 将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点路径封装成一个 WatchedEvent对象
>
> - 查询 Watcher：
>
>> 从 WatchTable 中根据节点路径查找 Watcher 。
>>
>> - 没找到： 说明没有客户端在该数据节点上注册过 Watcher 。
>> - 找到： 提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher (**从这里可以看出 Watcher 在服务端是一次性的，触发一次就失效了**)。

- 3、调用 process 方法来触发 Watcher 。

> 这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件通知。

📣 第三步，客户端回调 Watcher 实现?

客户端 SendThread 线程接收事件通知，交由 EventThread 线程回调Watcher 。

客户端的 Watcher 机制同样是一次性的，一旦被触发后，该 Watcher 就失效了。

Zookeeper 采用什么权限控制机制?

> 在网上看到一个「你们的 Zookeeper 的节点加密是用的什么方式？」问题，应该也是问这个。

目前，在 Linux/Unix 文件系统中，使用 UGO(User/Group/Others) 权限模型，也是使用最广泛的权限控制方式。是一种粗粒度的文件系统权限控制模式。

> 一般我们管理后台，采用的 RBAC 居多，和 UGO 比较类似，差别在于一般将权限分配给 Role ，而不是直接给 User 。

对于 Zookeeper ，它采用 ACL（Access Control List）访问控制列表。包括三个方面：

- 权限模式（Scheme）

> - IP： 从 IP 地址粒度进行权限控制
> - 【常用】Digest： 最常用，用类似于 `username:password` 的权限标识来进行权限配置，便于区分不同应用来进行权限控制。
> - World： 最开放的权限控制方式，是一种特殊的 digest 模式，只有一个权限标识 `“world:anyone”` 。
> - Super： 超级用户。

- 授权对象

> 授权对象指的是权限赋予的用户或一个指定实体，例如 IP 地址或是机器等。

- 权限 Permission

> - CREATE： 数据节点创建权限，允许授权对象在该 znode 下创建子节点。
> - DELETE： 子节点删除权限，允许授权对象删除该数据节点的子节点。
> - READ： 数据节点的读取权限，允许授权对象访问该数据节点并读取其数据内容或子节点列表等。
> - WRITE： 数据节点更新权限，允许授权对象对该数据节点进行更新操作。
> - ADMIN： 数据节点管理权限，允许授权对象对该数据节点进行 ACL 相关设置操作。

📣 Chroot 特性是什么?

Zookeeper 3.2.0 版本后，添加了 Chroot 特性。该特性允许每个客户端为自己设置一个命名空间。如果一个客户端设置了 Chroot ，那么该客户端对服务器的任何操作，都将会被限制在其自己的命名空间下。

通过设置 Chroot ，能够将一个客户端应用于 Zookeeper 服务端的一颗子树相对应，在那些多个应用公用一个 Zookeeper 进群的场景下，对实现不同应用间的相互隔离非常有帮助。

> ：貌似实际还用的比较少。

Zookeeper 的会话管理是怎么样的?

ZooKeeper 的每个客户端都维护一组服务端信息，在创建连接时由应用指定，客户端随机选择一个服务端进行连接，连接成功后，服务端为每个连接分配一个唯一标识。

- 客户端在创建连接时可以指定溢出时间，客户端会周期性的向服务端发送 PING 请求来保持连接。

  > 如果客户端异常下线，或者网络问题，导致一段时间没心跳给 Zookeeper 服务端，则会被 Zookeeper 标记为下线。

- 当客户端检测到与服务端断开连接后，客户端将自动选择服务端列表中的另一个服务端进行重连。客户端允许应用修改服务端列表，但修改可能导致客户端与服务端的重连。

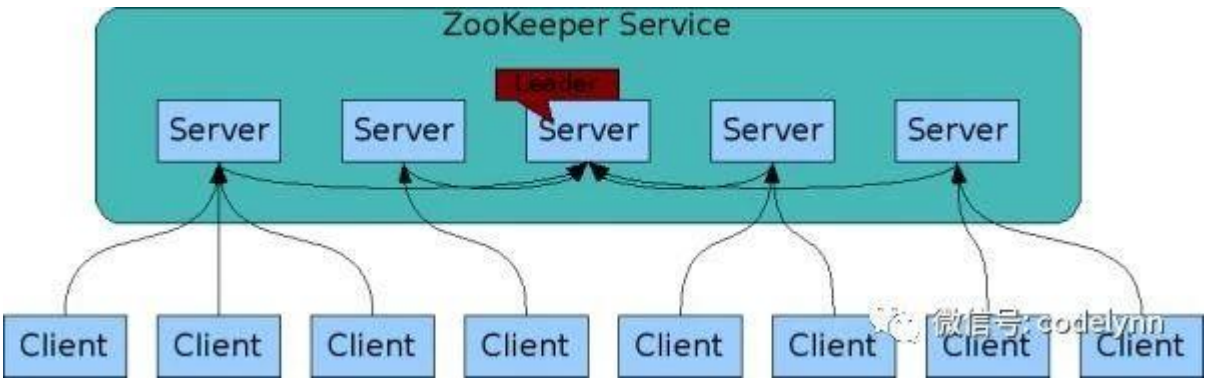
详细的，推荐阅读如下两篇文章：

- 《ZooKeeper session 管理》
- 《ZooKeeper 技术内幕：会话》更原理层面。

Zookeeper 的部署方式?

Zookeeper 有两种部署方式：

- 1、单机
- 2、集群

> Zookeeper 集群，是一个由多个 Server 组成，一个 Leader，多个 Follower。（这个不同于我们常见的 Master/Slave 模式）Leader 为客户端服务器提供读写服务，除了 Leader 外其他的机器只能提供读服务。
>
> 每个 Server 保存一份数据副本全数据一致，分布式读 Follower ，写由 Leader 实施更新请求转发，由 Leader 实施更新请求顺序进行，来自同一个 Client 的更新请求按其发送顺序依次执行数据更新原子性，一次数据更新要么成功，要么失败。
>
> 全局唯一数据视图，Client 无论连接到哪个 Server，数据视图都是一致的实时性，在一定事件范围内，Client 能读到最新数据。
>
> 
>

一般来说，测试环境部署单机，而生产环境必须必须必须部署集群。

📣 集群中的机器角色有哪些?

集群中一共有三种角色:

- 1、Leader
  - 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。
  - 集群内部各服务的调度者。
- 2、Follower
  - 处理客户端的非事务请求，转发事务请求给 Leader 服务器。
  - 参与事务请求 Proposal 的投票。
  - 参与 Leader 选举投票。
- 3、Observer

  3.3.0 版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础上提升集群的非事务处理能力。
  - 处理客户端的非事务请求，转发事务请求给 Leader 服务器
  - 不参与任何形式的投票。

  如果 ZooKeeper 集群的读取负载很高，或者客户端多到跨机房，可以设置一些 Observer 服务器，以提高读取的吞吐量。Observer 和 Follower 比较相似，只有一些小区别:
  - 首先 Observer 不属于法定人数，即不参加选举也不响应提议，也不参与写操作的“过半写成功”策略;
  - 其次是 Observer 不需要将事务持久化到磁盘，一旦 Observer 被重启，需要从 Leader 重新同步整个名字空间。

在一个集群中，最少需要 3 台。或者保证 2N + 1 台，即奇数。为什么保证奇数？主要是为了选举算法。

📣 集群如果有 3 台机器，挂掉 1 台集群还能工作吗？挂掉 2 台呢?

记住一个原则：过半存活即可用。所以挂掉 1 台可以继续工作，挂掉 2 台不可以工作。

📣 集群支持动态添加机器吗?

在 3.5 版本开始，支持动态扩容。

而在 3.5 版本之前，Zookeeper 在这方面不太好。所以需要如下两种方式:

- 全部重启：关闭所有 Zookeeper 服务，修改配置之后启动。不影响之前客户端的会话。
- 逐个重启：顾名思义。这是比较常用的方式。

📣 Zookeeper 下 Server 工作状态?

服务器具有四种状态，分别是:

- LOOKING 寻找 Leader 状态

  当服务器处于该状态时，它会认为当前集群中没有 Leader ，因此需要进入 Leader 选举状态。
- FOLLOWING 跟随者状态

  表明当前服务器角色是 Follower 。
- LEADING 领导者状态

> 表明当前服务器角色是 Leader 。

- OBSERVING 观察者状态

> 表明当前服务器角色是 Observer 。

ZooKeeper 的工作原理?

ZooKeeper 的核心是原子广播，这个机制保证了各个 Server 之间的同步。实现这个机制的协议叫做 **Zab** 协议。Zab 协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）：

- 选主：当服务启动或者 Leader 崩溃后，Zab 就进入了恢复模式，当新的 Leader 被选举出来，且大多数 Server 完成了和 Leader 的状态同步以后，恢复模式就结束了。

> 更加详细的描述。
>
> 当整个 Zookeeper 集群刚刚启动，或者 Leader 服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式。
>
> - 首先，选举产生新的Leader服务器。
> - 然后，集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步。
> - 当集群中超过半数机器与该Leader服务器完成数据同步之后，退出恢复模式进入消息广播模式，

- 同步：状态同步保证了 Leader 和 Server 具有相同的系统状态。

> 更加详细的描述。
>
> Leader 服务器开始接收客户端的事务请求，生成事务提案来进行事务请求处理。

📣 ZooKeeper 是如何保证事务的顺序一致性的?

ZooKeeper 采用了递增的事务 id 来识别，所有的 proposal（提议）都在被提出的时候加上了 zxid 。zxid 实际上是一个 64 位数字。

- 高 32 位是 epoch 用来标识 Leader 是否发生了改变，如果有新的 Leader 产生出来，epoch会自增。
- 低 32 位用来递增计数。

当新产生的 peoposal 的时候，会依据数据库的两阶段过程，首先会向其他的 Server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

📣 ZooKeeper 集群中个服务器之间是怎样通信的?

Leader 服务器会和每一个 Follower/Observer 服务器都建立 TCP 连接，同时为每个 Follower/Observer 都创建一个叫做 LearnerHandler 的实体。

- LearnerHandler 主要负责 Leader 和 Follower/Observer 之间的网络通讯，包括数据同步，请求转发和 Proposal 提议的投票等。
- Leader 服务器保存了所有 Follower/Observer 的 LearnerHandler 。

📣 ZAB 和 Paxos 算法的联系与区别?

Paxos 算法是分布式选举算法，Zookeeper 使用的 ZAB 协议（Zookeeper 原子广播）。

二者有相同的地方：

- 都有一个 Leader，用来协调 N 个 Follower 的运行
- Leader 要等待超半数的 Follower做 出正确反馈之后才进行提案。

- 二者都有一个值来代表 Leader 的周期。ZAB 协议中，每个 Proposal 中都包含一个 epoch 值来代表当前的 Leader周期，Paxos中名字为 Ballot 。

不同的地方在于：

- ZAB 用来构建高可用的分布式数据主备系统（Zookeeper），Paxos 是用来构建分布式一致性状态机系统。

Paxos 算法、ZAB 协议要想讲清楚可不是一时半会的事儿，自 1990 年莱斯利·兰伯特提出 Paxos 算法以来，因为晦涩难懂并没有受到重视。后续几年，兰伯特通过好几篇论文对其进行更进一步地解释，也直到 06 年谷歌发表了三篇论文，选择 Paxos 作为 Chubby cell 的一致性算法，Paxo s才真正流行起来。

对于普通开发者来说，尤其是学习使用 Zookeeper 的开发者明确一点就好：分布式 Zookeeper 选举 Leader 服务器的算法，与 Paxos 有很深的关系。

Zookeeper 的选举过程?

当 Leader 崩溃，或者 Leader 失去大多数的 Follower，这时 Zookeeper 进入恢复模式，恢复模式需要重新选举出一个新的 Leader，让所有的 Server 都恢复到一个正确的状态。

Zookeeper 的选举算法有两种：一种是基于 basic paxos 实现的，另外一种是基于 fast paxos 算法实现的。系统默认的选举算法为 fast paxos 。

> 相对详细的，胖友可以看看 《【分布式】Zookeeper的Leader选举》 和 《Zookeeper 源码分析 —— Zookeeper Leader 选举算法》 。
>
> - 不同阶段的选举流程
>   - 服务器启动时期的 Leader 选举。
>   - 服务器运行时期的 Leader 选举。
> - 三种选举算法
>   - LeaderElection： 使用 basic paxos 算法。
>   - FastLeaderElection： 使用 fast paxos 算法。
>   - AuthFastLeaderElection： 在 FastLeaderElection 的基础上，增加认证。
>   - 最终在 Zookeeper 3.4.0 版本之后，只保留 FastLeaderElection 版本。

😈 看下面的原理描述，还是有点懵逼。等后面自己去撸下源码，可能会清晰一些。

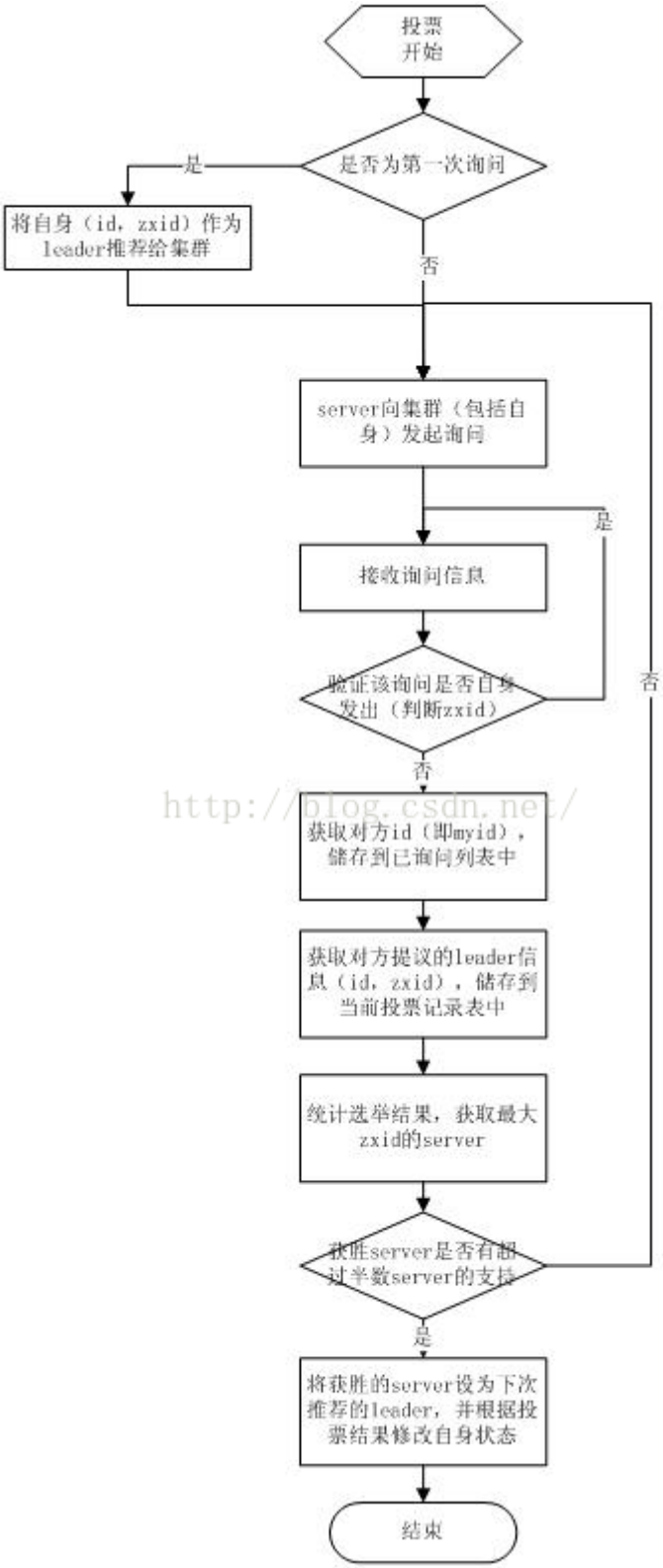
📣 Zookeeper 选主流程(basic paxos)?

> 选择性了解。

- 1、选举线程由当前 Server 发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的 Server 。
- 2、选举线程首先向所有 Server 发起一次询问(包括自己)。
- 3、选举线程收到回复后，验证是否是自己发起的询问(验证 zxid 是否一致)，然后获取对方的 id(myid)，并存储到当前询问对象列表中，最后获取对方提议的 Leader相关信息(id，zxid)，并将这些信息存储到当次选举的投票记录表中。
- 4、收到所有 Server 回复以后，就计算出 zxid 最大的那个 Server ，并将这个 Server 相关信息设置成下一次要投票的 Server 。
- 5、线程将当前 zxid 最大的 Server 设置为当前 Server 要推荐的 Leader ，如果此时获胜的 Server 获得 `n/2+1` 的 Server 票数，设置当前推荐的 Leader 为获胜的 Server ，将根据获胜的 Server 相关信息设置自己的状态，否则，继续这个过程，直到 Leader 被选举出来。

通过流程分析我们可以得出：要使 Leader 获得多数 Server 的支持，则 Server 总数必须是奇数 `2n+1` ，且存活的 Server 的数目不得少于 `n+1` 。每个 Server 启动后都会重复以上流程。

在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的 Server 还会从磁盘快照中恢复数据和会话信息，Zookeeper 会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。



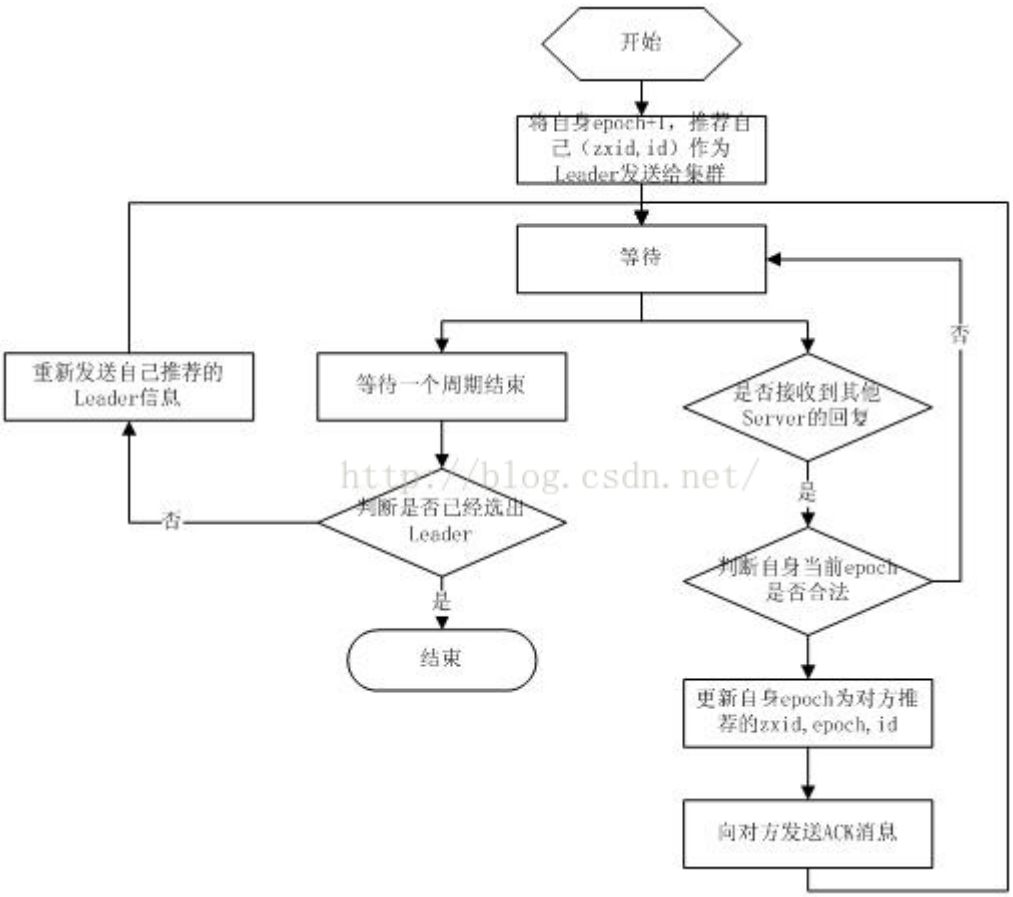
📣 Zookeeper 选主流程(fast paxos)?

> 重点了解。这块在 《Zookeeper 源码分析 —— Zookeeper Leader 选举算法》 写的比较详细。

由于 LeaderElection 收敛速度较慢，所以 Zookeeper 引入了 FastLeaderElection 选举算法，FastLeaderElection 也成了Zookeeper默认的Leader选举算法。

FastLeaderElection 是标准的 Fast Paxos 的实现。它首先向所有 Server 提议自己要成为 Leader ，当其它 Server 收到提议以后，解决 epoch 和 zxid 的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息。重复这个流程，最后一定能选举出Leader。

FastLeaderElection 算法通过异步的通信方式来收集其它节点的选票，同时在分析选票时又根据投票者的当前状态来作不同的处理，以加快 Leader 的选举进程。



为什么 Zookeeper 集群推荐节点数是单数?

在统计投票时，有个过半的概念，大于集群机器数量的一半，即大于或等于(`n/2+1`)。那么我们来看看如下的统计：

集群数量	至少正常运行数量	允许挂掉的数量
2	2 的半数为 1，半数以上最少为 2	0
3	3 的半数为 1.5，半数以上最少为 2	1
4	4 的半数为 2，半数以上最少为 3	1
5	5 的半数为 2.5，半数以上最少为 3	2
6	6 的半数为 3，半数以上最少为 4	2

通过以上可以发现：

- 3 台服务器和 4 台服务器都最多允许 1 台服务器挂掉，5 台服务器和 6 台服务器都最多允许 2 台服务器挂掉，明显 4 台服务器成本高于 3 台服务器成本，6 台服务器成本高于 5 服务器成本。
- 这是由于半数以上投票通过决定的。所以，Zookeeper 集群推荐节点数是单数。

简单的来说，**节省资源**！

另外，因为 Zookeeper 使用一致性协议，过多的节点，反倒会降低性能。😈

📣 Zookeeper 是否需存在脑裂?

按道理说，Zookeeper 选举不会存在脑裂问题，因为需要 `n / 2 + 1` 投票通过，才能执行对应的写操作。但是听朋友说，实际场景下，貌似发生过脑裂问题。关于这块，心里也不太有底，欢迎在星球一起讨论。

- 《Zookeeper 已经分布式环境中的假死脑裂》

  > 认为存在脑裂问题，以及提供怎么解决。

- 《zookeeper（二）常见问题汇总》

  > 认为不会存在脑裂问题。

📣 机器中为什么会有 Leader?

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 Leader 选举。

Zookeeper 的同步流程?

选完 Leader 以后，Zookeeper 就进入状态同步过程。

- 1、Leader 等待 Server 连接。
- 2、Follower 连接 Leader ，将最大的 zxid 发送给 Leader 。
- 3、Leader 根据 Follower 的 zxid 确定同步点。
- 4、完成同步后通知 Follower 已经成为 update 状态。
- 5、Follower 收到 update 消息后，又可以重新接受 Client 的请求进行服务了。

当然，同步流程并不是像上述描述的这么简单，具体的，还是得看看 《Zookeeper Leader 和 Learner 的数据同步》 。

666. 彩蛋

估计大多数胖友，学习 Zookeeper 的过程，是因为使用 Dubbo 时，需要使用到 Zookeeper 作为注册中心，然后快速搭建了下。然后，断断续续看了下 Zookeeper 的文章。☺ 就当是复习。面对的时候，比较重点的几个问题是：

- Zookeeper 的选举过程?
- Zookeeper 如何提供分布式锁?
- Zookeeper 的一些应用场景?

参考与推荐如下文章：

- 《ZooKeeper 学习总结（3）—— ZooKeeper 常见面试题》
- 《Zookeeper 常见面试题》
- 《Zookeeper 面试题》
- 《如果有人问你 ZooKeeper 是什么，就把这篇文章发给他》