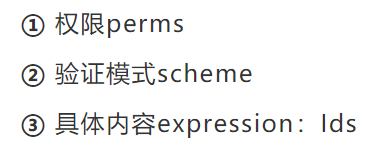
# [ZooKeeper机制架构](https://www.cnblogs.com/sunddenly/p/4133784.html)

## ZooKeeper权限管理机制

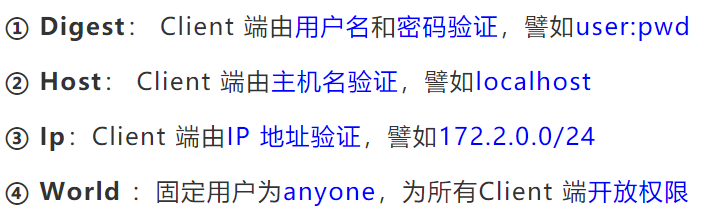
ZooKeeper 的权限管理亦即ACL 控制功能，使用ACL来对Znode进行访问控制。ACL的实现和Unix文件访问许可非常相似：它使用许可位来对一个节点的不同操作进行允许或禁止的权限控制。但是和标准的Unix许可不同的是，Zookeeper对于用户类别的区分，不止局限于所有者(owner)、组 (group)、所有人(world)三个级别。Zookeeper中，数据节点没有"所有者"的概念。访问者利用id标识自己的身份，并获得与之相应的不同的访问权限。ZooKeeper 的权限管理通过Server、Client 两端协调完成：

**(1) Server端**

一个ZooKeeper 的节点存储两部分内容：**数据**和**状态**，状态中包含ACL 信息。创建一个znode 会产生一个ACL 列表，列表中每个ACL 包括：



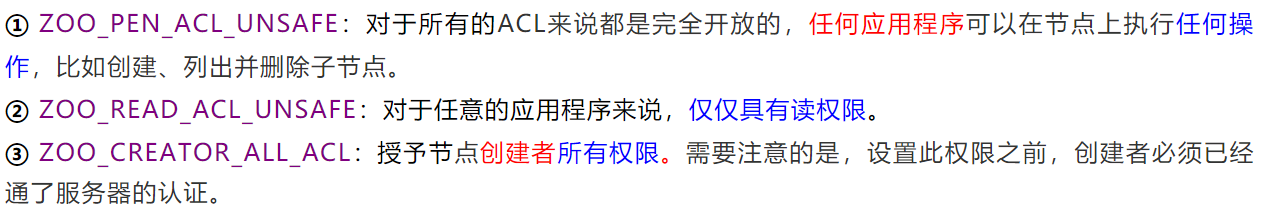
例如，当scheme="digest" 时， Ids 为用户名密码， 即"root ：J0sTy9BCUKubtK1y8pkbL7qoxSw"。ZooKeeper 提供了如下几种验证模式：



当会话建立的时候，客户端将会进行自我验证。权限许可集合如下：



另外，ZooKeeper Java API支持**三种标准的用户权限**，它们分别为：



Znode ACL 权限用一个int 型数字perms 表示，perms 的5 个二进制位分别表示set**a**cl、**d**elete、**c**reate、**w**rite、**r**ead。比如adcwr=0x1f，----r=0x1，a-c-r=0x15。

**(2) 客户端**

Client 通过调用addAuthInfo()函数设置当前会话的Author信息**（**针对Digest 验证模式**）**。Server 收到Client 发送的操作请求**（**除exists、getAcl 之外**）**，需要进行**ACL 验证**：对该请求携带的Author 明文信息加密，并与目标节点的ACL 信息进行比较，如果匹配则具有相应的权限，否则请求被Server 拒绝。

详情看代码：NewDigest

一次Client 对Znode 进行操作的验证ACL 的方式为：

**a) 遍历znode的所有ACL：**

**①** 对于每一个ACL，首先操作类型与权限（perms）匹配

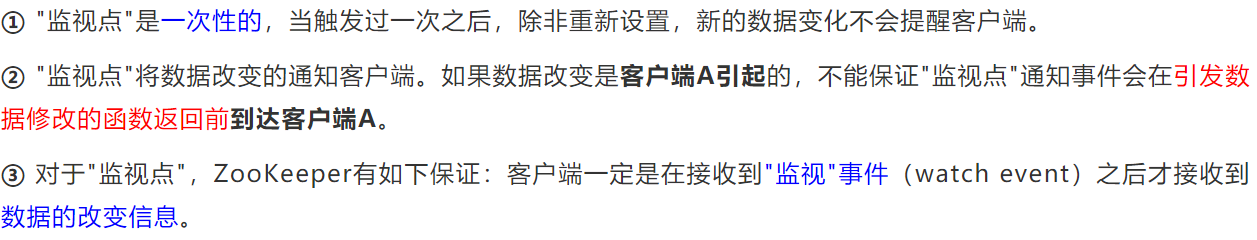
**②**只有匹配权限成功才进行session 的auth 信息与ACL 的用户名、密码匹配

**b)**如果两次匹配都成功，则允许操作；否则，返回权限不够error（rc=-102）

## Watch机制

Zookeeper客户端在数据节点上设置监视，则当数据节点发生变化时，客户端会收到提醒。ZooKeeper中的各种读请求，如getDate()，getChildren()，和exists()，都可以选择加"监视点"(watch)。"监视点"指的是一种**一次性的触发器**(trigger)，当受监视的数据发生变化时，该触发器会通知客户端。

1. **监视机制有三个关键点：**



**(2)** "**监视点**"保留在ZooKeeper服务器上，则当客户端连接到新的ZooKeeper服务器上时，所有需要被触发的相关"监视点"都会被触发。当客户端断线后重连，与它的相关的"监视点"都会自动重新注册，这对客户端来说是透明的。在以下情况，"监视点"会被错过：客户端B设置了关于节点A存在性的"监视点"，但B断线了，在B断线过程中节点A被创建又被删除。此时，B再连线后不知道A节点曾经被创建过。

**(3)** ZooKeeper的"监视"机制保证以下几点：

**①** "监视"事件的触发顺序和事件的分发顺序一致。

**②** 客户端将先接收到"监视"事件，然后才收到新的数据

**③** "监视"事件触发的顺序与ZooKeeper服务器上数据变化的顺序一致

**(4)**关于ZooKeeper"监视"机制的注意点：

**①** "监视点"是一次性的。

**②**由于"监视点"是一次性的，而且，从接收到"监视"事件到设置新"监视点"是有延时的，所以客户端可能监控不到数据的所有变化。

**③** 一个监控对象，只会被相关的通知触发一次。如果一个客户端设置了关于某个数据点exists和getData的监控，则当该数据被删除的时候，只会触发"文件被删除"的通知。

**④**当客户端断开与服务器的连接时，客户端不再能收到"监视"事件，直到重新获得连接。所以关于Session的信息将被发送给所有ZooKeeper服务器。由于当连接断开时收不到"监视"，所以在这种情况下，模块行为需要容错方面的设计。

## Session机制

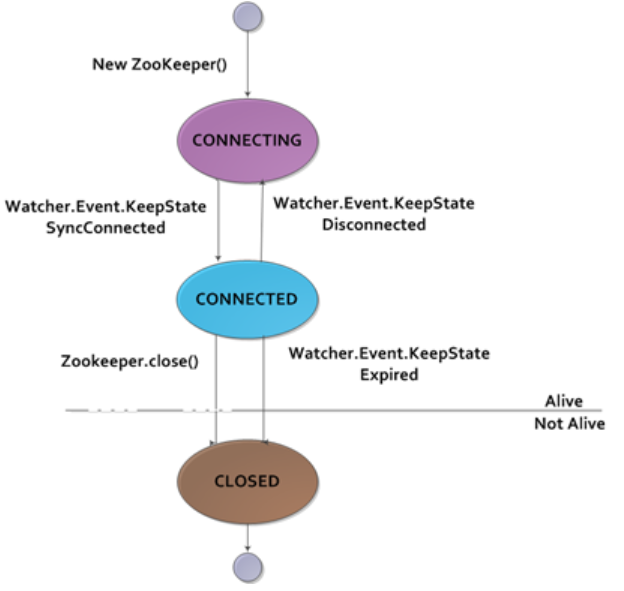
每个ZooKeeper客户端的配置中都包括集合体中服务器的列表。在启动时，客户端会尝试连接到列表中的一台服务器。如果连接失败，它会尝试连接另一台服务器，以此类推，直到成功与一台服务器建立连接或因为所有ZooKeeper服务器都不可用而失败。

一旦客户端与一台ZooKeeper服务器建立连接，这台服务器就会为该客户端创建一个新的会话。每个会话都会有一个超时的时间设置，这个设置由创建会话的应用来设定。如果服务器在超时时间段内没有收到任何请求，则相应的会话会过期。一旦一个会话已经过期，就无法重新打开，并且任何与该会话相关联的短暂znode都会丢失。会话通常长期存在，而且会话过期是一种比较罕见的事件，但对应用来说，如何处理会话过期仍是非常重要的。

只要一个会话空闲超过一定时间，都可以通过客户端发送ping请求（也称为心跳）保持会话不过期。ping请求由ZooKeeper的客户端库自动发送，因此在我们的代码中不需要考虑如何维护会话。这个时间长度的设置应当足够低，以便能档检测出服务器故障（由读超时体现），并且能够在会话超时的时间段内重新莲接到另外一台服务器。

ZooKeeper客户端可以自动地进行故障切换，切换至另一台ZooKeeper服务器。**并且关键的一点是**，在另一台服务器接替故障服务器之后，所有的会话和相关的短暂Znode仍然是有效的。**在故障切换过程中**，应用程序将收到断开连接和连接至服务的通知。当**客户端断开连接时**，观察通知将无法发送；但是当客户端成功恢复连接后，这些延迟的通知会被发送。当然，在客户端重新连接至另一台服务器的过程中，如果应用程序试图执行一个操作，这个操作将会失败。这充分体现了在真实的ZooKeeper应用中处理连接丢失异常的重要性。

## ZooKeeper实例状态



ZooKeeper对象在其生命周期中会经历几种不同的状态。你可以在任何时刻通过getState()方法来查询对象的状态：在试图与ZooKeeper服务**建立连接的过程中**，一个新建的ZooKeeper实例处于CONNECTING状态。一旦**建立连接**，它就会进入CONNECTED状态。

ZooKeeper实例可能失去或重新连接ZooKeeper服务，在CONNECTED和CONNECTING状态中切换。如果连接断开，watcher得到一个Disconnected事件。学要注意的是，这些状态的迁移是由ZooKeeper实例自己发起的，如果连接断开他将自动尝试自动连接。

如果任何一个close()方法被调用，或是会话由Expired类型的KeepState提示过期时，ZooKeeper可能会转变成第三种状态CLOSED。一旦处于CLOSED状态，ZooKeeper对象将不再是活动的了(可以使用states的isActive()方法进行测试)，而且不能被重用。客户端必须建立一个新的ZooKeeper实例才能重新连接到ZooKeeper服务。