[**ZooKeeper Programmer's Guide**](http://www.cnblogs.com/Xrinehart/p/3512509.html)

本文是为想要创建使用ZooKeeper协调服务优势的分布式应用的开发者准备的。本文包含理论信息和实践信息。

本指南的前四节对各种ZooKeeper概念进行较高层次的讨论。这些概念对于理解ZooKeeper是如何工作的，以及如何使用ZooKeeper来进行工作都是必要的。这几节没有代码，但却要求读者对分布式计算相关的问题较为熟悉。这四节是：

l         [ZooKeeper数据模型](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_2%20ZooKeeper%E6%95%B0%E6%8D%AE%E6%A8%A1%E5%9E%8B)

l         [ZooKeeper会话](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_3%20ZooKeeper%E4%BC%9A%E8%AF%9D)

l         [ZooKeeper观察](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_4%20ZooKeeper%E8%A7%82%E5%AF%9F)

l         [一致性保证](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_7%20%E4%B8%80%E8%87%B4%E6%80%A7%E4%BF%9D%E8%AF%81)

接下来的四节提供了实际编程的信息。这四节是：

l         [创建块：ZooKeeper操作指南](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_9%20%E5%88%9B%E5%BB%BA%E5%9D%97%EF%BC%9AZooKeeper%E6%93%8D%E4%BD%9C%E6%8C%87%E5%8D%97)

l         [绑定](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_8%20%E7%BB%91%E5%AE%9A)

l         [程序结构和简单示例](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_9.7%20%E7%A8%8B%E5%BA%8F%E7%BB%93%E6%9E%84%E5%92%8C%E7%AE%80%E5%8D%95%E7%A4%BA%E4%BE%8B)

l         [转向：常见问题和解决](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_10%20%E8%BD%AC%E5%90%91%EF%BC%9A%E5%B8%B8%E8%A7%81%E9%97%AE%E9%A2%98%E5%92%8C%E8%A7%A3%E5%86%B3)

本文最后的[附录](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#%E9%99%84%E5%BD%95)包含到其他有用的ZooKeeper相关信息的链接。

本文的大多数信息以可独立访问的参考材料的形式存在。但是，在编写第一个ZooKeeper应用程序之前，你应该至少读过[ZooKeeper数据模型](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_2%20ZooKeeper%E6%95%B0%E6%8D%AE%E6%A8%A1%E5%9E%8B)和[ZooKeeper基本操作](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_9%20%E5%88%9B%E5%BB%BA%E5%9D%97%EF%BC%9AZooKeeper%E6%93%8D%E4%BD%9C%E6%8C%87%E5%8D%97)。此外，[简单示例程序](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_9.7%20%E7%A8%8B%E5%BA%8F%E7%BB%93%E6%9E%84%E5%92%8C%E7%AE%80%E5%8D%95%E7%A4%BA%E4%BE%8B)也有助于理解ZooKeeper客户端应用程序的基本结构。

# 2 ZooKeeper 数据模型

ZooKeeper有一个分层的名字空间，跟分布式文件系统很相似。唯一的不同是，名字空间中的每个节点都可以有关联的数据和子节点。这就像一个允许文件也是目录的文件系统。节点路径总是表达为规则的、斜杠分隔的绝对路径，不存在相对路径。路径可以使用任何Unicode字符，但是需要遵循下列限制：

l         不能使用空字符(\?)。（这在C绑定中会导致问题）

l         因为不能正确显示，或者容易弄混淆，不能使用这些字符：\ - \和\ - \?。

l         不允许使用这些字符：\? - uF8FFF、\? - uFFFF、\uXFFFE - \uXFFFF(X是1到E之间的一个数字)、\? - \?。

l         可以使用小数点，但是不能单独使用.和..来指示路径中的节点，因为ZooKeeper不使用相对路径。/a/b/./c或者/a/b/../c是无效的。

l         记号zookeeper是保留的。

## 2.1 ZNode

ZooKeeper树中的节点称作**znode**。znode会维护一个包含数据修改和ACL修改版本号的Stat结构体，这个结构体还包含时间戳字段。版本号和时间戳让ZooKeeper可以校验缓存，协调更新。每次修改znode数据的时候，版本号会增加。客户端获取数据的同时，也会取得数据的版本号。执行更新或者删除操作时，客户端必须提供版本号。如果提供的版本号与数据的实际版本不匹配，则更新操作失败。（可以覆盖这个行为，更多信息请看……）

**注意：**

分布式应用工程中，node这个词可以指代主机、服务器、集群成员、客户端进程等等。ZooKeeper文档用znode指代数据节点；用server指代组成ZooKeeper服务的机器；用quorum peer指代组成集群的服务器；用client指代任何使用ZooKeeper服务的主机或者进程。

znode是程序员访问的主要实体，它有一些值得讨论的特征。

### 2.1.1  观察

客户端可以在znode上设置观察。对znode的修改将触发观察，然后移除观察。观察被触发时，ZooKeeper向客户端发送一个通知。关于观察的更多信息请看[ZooKeeper观察](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_4%20ZooKeeper%E8%A7%82%E5%AF%9F)。

### 2.1.2  数据存取

存储在名字空间中每个znode节点里的数据是原子地读取和写入的。读取操作获取节点的所有数据，写入操作替换所有数据。节点的访问控制列表(ACL)控制可以进行操作的用户。

ZooKeeper不是设计用来作为通用数据库或者大型对象存储的，而是用来存储协调数据的。协调数据的形式可能是配置、状态信息、聚合等等。各种形式的协调数据的一个共同特点是：它们通常比较小，以千字节来衡量。ZooKeeper客户端和服务器实现会进行检查，以保证znode数据小于1MB，但是平均的实际数据量应该远小于1MB。对较大数据的操作将导致某些操作比其他操作耗费更多时间，进而影响某些操作的延迟，因为需要额外的时间在网络和存储媒体间移动更多数据。如果需要大数据存储，通常方式是存储到块存储系统，如NFS或者HDFS中，然后在ZooKeeper中保存到存储位置的指针。

### 2.1.3  临时节点

ZooKeeper有临时节点的概念。临时节点在创建它的会话活动期间存在。会话终止的时候，临时节点被删除，所以临时节点不能有子节点。

### 2.1.4  顺序节点：唯一命名

创建znode时，可以要求ZooKeeper在路径名后增加一个单调增加的计数器部分。这个计数器相对于znode的父节点是唯一的。计数器的格式是0d，也就是带有0填充的10个数字（这种格式是为了方便排序），比如说，<path>0000000001。[队列接收](http://zookeeper.apache.org/doc/trunk/recipes.html#sc_recipes_Queues)节里有一个使用这种特征的例子。注意：用于存储下一个顺序号的计数器是一个由父节点维护的有符号整数（4字节），所以计数器将在超过2147483647的时候溢出（导致名字成为<path>-2147483647）。

## 2.2 ZooKeeper 中的时间

ZooKeeper以多种方式跟踪时间：

l         zxid

每次修改ZooKeeper状态都会收到一个zxid形式的时间戳，也就是ZooKeeper事务ID。事务ID是ZooKeeper中所有修改总的次序。每个修改都有唯一的zxid，如果zxid1小于zxid2，那么zxid1在zxid2之前发生。

对节点的每次修改将使得节点的版本号增加一。版本号有三种：version（znode数据修改的次数）、cversion（znode子节点修改的次数），以及aversion（znode的ACL修改次数）。

l         tick

多服务器ZooKeeper中，服务器使用tick来定义状态上传、会话超时、节点间连接超时等事件的时序。tick仅被最小会话超时（2倍的tick时间)间接使用：如果客户端要求小于最小会话超时的时间，服务器将告知客户端，实际使用的是最小会话超时。

l         真实时间

除了在创建和修改znode时将时间戳放入stat结构体中之外，ZooKeeper不使用真实时间，或者说时钟时间。

## 2.3 ZooKeeper的Stat 结构体

ZooKeeper中每个znode的Stat结构体由下述字段构成：

l         czxid：创建节点的事务的zxid

l         mzxid：对znode最近修改的zxid

l         ctime：以距离时间原点(epoch)的毫秒数表示的znode创建时间

l         mtime：以距离时间原点(epoch)的毫秒数表示的znode最近修改时间

l         version：znode数据的修改次数

l         cversion：znode子节点修改次数

l         aversion：znode的ACL修改次数

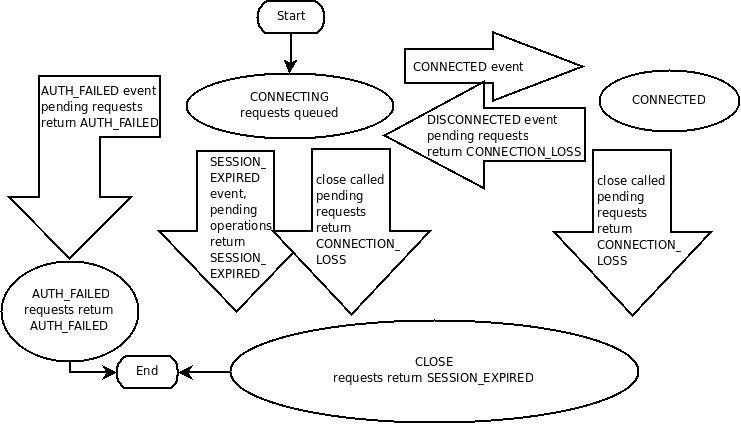
l         ephemeralOwner：如果znode是临时节点，则指示节点所有者的会话ID；如果不是临时节点，则为零。

l         dataLength：znode数据长度。

l         numChildren：znode子节点个数。

# 3 ZooKeeper 会话

客户端使用某种语言绑定创建一个到服务的句柄时，就建立了一个ZooKeeper会话。会话创建后，句柄处于CONNECTING状态，客户端库会试图连接到组成ZooKeeper服务的某个服务器；连接成功则进入到CONNECTED状态。通常操作中句柄将处于这两个状态之一。如果发生不可恢复的错误，如会话过期、身份鉴定失败，或者应用显式关闭，则句柄进入到CLOSED状态。下图显式了ZooKeeper客户端可能的状态转换：



要创建客户端会话，应用程序代码必须提供一个包含逗号分隔的列表的字符串，其中每个主机:端口对代表一个ZooKeeper服务器（例如，"127.0.0.1:4545"或者"127.0.0.1:3001,127.0.0.1:3002"）。ZooKeeper客户端库将试图连接到任意选择的一个服务器。如果连接失败，或者到服务器的连接断开，则客户端将自动尝试连接到列表中的下一个服务器，直到连接(重新)建立。

**3.2.0版新增加**：可以在连接字符串后增加可选的"chroot"后缀，这让客户端命令都是相对于指定的根的（类似于Unix的chroot命令)。例如，如果使用"127.0.0.1:4545/app/a"或者"127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002/app/a"，则客户端的根将是/app/a，所有路径将是相对于这个根的：获取/设置/foo/bar数据的操作将实际在/app/a/foo/bar上执行（从服务器来看）。这个特征在多用户环境中特别有用，某个特定ZooKeeper服务的每个用户可以使用不同的根。这让重用更加简单，用户应用在编码时以/为根，但实际的根位置（如/app/a）可以在部署时确定。

客户端取得ZooKeeper服务句柄时，ZooKeeper创建一个会话，由一个64位数标识，这个数将返回给客户端。如果连接到其他服务器，客户端将在连接握手时发送会话ID。出于安全考虑，服务器会为会话ID创建一个密码，ZooKeeper服务器可以校验这个密码。这个密码将在创建会话时与会话ID一同发送给客户端。与新的服务器重新建立会话的时候，客户端会和会话ID一同发送这个密码。

客户端库创建会话时需要的参数之一是毫秒表示的会话超时。客户端发送请求的超时值，服务器以可以分配给客户端的超时值回应。当前实现要求超时值最小是2倍的tickTime(在服务器配置文件中设置），最大是20倍的tickTime。客户端API可以获取商定的超时值。

从 服务集群分裂开来时，客户端（会话）将搜索会话创建时给出的服务器列表。最终，客户端至少和一个服务器重新建立连接，会话再次进入“已连接”状态（如果在 会话超时之前重新连接上），或者进入到“已过期”状态（如果在会话超时后才重新连接上）。不建议在断开连接时创建一个新的会话对象（即一个新的ZooKeeper.class对象，或者C绑定中的zookeeper句柄），因为客户端库会进行重新连接。特别是客户端库具有试探特征，可以处理“羊群效应”等问题。只需要在被通知会话已过期时创建新的会话（必须的）。

会话过期由ZooKeeper集群，而不是客户端来管理。客户端与集群建立会话时会提供上面讨论的超时值。集群使用这个值来确定客户端会话何时过期。集群在指定的超时时间内没有得到客户端的消息时发生会话过期。会话过期时集群将删除会话的所有临时节点，立即通知所有(观察节点的)客户端。此时已过期会话的客户端还是同集群断开连接的，不会被通知会话已经过期，直到(除非)客户端重新建立到集群的连接，这时候已过期会话的观察才会收到“会话已过期”通知。

已过期会话的观察看到的状态转换过程示例：

1．已连接：会话建立，客户端与集群通信中（客户端/服务器通信正常进行）

2．客户端从集群中分离

3．连接已断开：客户端失去同集群的连接

4．时间流逝，超时时间过后，集群让会话过期，客户端并不知道，因为它还是同集群断开连接的。

5．时间流逝，客户端与集群间的网络恢复正常。

6．已过期：最终客户端重新连接到集群，此时被通知会话已经过期。

建 立会话时的另一个参数是默认观察。客户端发生状态改变时观察会被通知。比如说，客户端将在失去同服务器的连接，或者会话过期时被通知。观察应该认为初始状 态是连接已经断开（在客户端库向观察发送任何状态改变事件之前）。新建连接时发送给观察的第一个事件通常是会话连接建立事件。

会话由客户端发送的请求保持为活动状态。如果要空闲一段将导致超时的时间，客户端将发送PING请求，保持会话是活动的。PING请求不仅让服务器知道客户端仍然是存活的，也让客户端可以确认，到服务器的连接依然是活动的。PING的时序足够保守，确保能够在合理的时间内检测到死掉的连接，重新连接到新的服务器。

一旦到服务器的连接成功建立，则进行同步或者异步操作时，通常有两种情况导致客户端库产生连接丢失事件（C绑定中的错误码，Java中的异常：关于绑定特定的细节，请看API文档）：

1．应用程序对已经不存活/有效的会话进行操作

2．在有到服务器的未决操作（例如，有一个进行中的异步调用）时，客户端断开同服务器的连接

**3.2.0版增加**：SessionMovedException。有一种称作SessionMovedException的 内部异常。通常客户端看不到这个异常。在某连接上收到一个会话请求，但是这个会话已经重建到另一个服务器上的时候会发生这种异常。导致这种错误的原因通常 是，客户端向服务器发送请求，但是数据分组被延迟，以致客户端超时并且连接到一个新的服务器。延迟的分组到达先前的服务器的时候，服务器检测到会话已经移 走，会关闭客户端连接。客户端通常看不到这个错误，因为客户端不会从较早的连接上读取数据（通常关闭了较早的连接）。两个客户端试图使用已保存的会话ID和密码重新建立相同的连接时会看到这种错误。其中一个客户端将重新建立连接，而另一个客户端会被断开连接（导致无限次地试图重新建立连接/会话）。

# 4 ZooKeeper 观察

ZooKeeper中的所有读操作：getData()、getChildren()和exists()，都有一个设置观察作为边效应的选项。ZooKeeper对观察的定义是：观察事件是在被观察数据发生变化时，发送给建立观察的客户端的一次性触发器。对于这个定义，有三点值得关注：

l         一次触发

观察事件将在数据修改后发送给客户端。比如说，如果客户端执行getData("/znode1",true)，然后/znode1的数据发生变化，或者被删除，则客户端将收到/znode1的观察事件。如果再次修改/znode1，则不会给客户端发送观察事件，除非客户端再执行一次读取操作，设置新的观察。

l         发送给客户端

这暗示着，在（导致观察事件被触发的）修改操作的成功返回码到达客户端之前，事件可能在去往客户端的路上，但是可能不会到达客户端。观察事件是异步地发送给观察者（客户端）的。ZooKeeper会保证次序：在收到观察事件之前，客户端不会看到已经为之设置观察的节点的改动。网络延迟或者其他因素可能会让不同的客户端在不同的时间收到观察事件和更新操作的返回码。这里的要点是：不同客户端看到的事情都有一致的次序。

l         为哪些数据设置观察

节点有不同的改动方式。可以认为ZooKeeper维护两个观察列表：数据观察和子节点观察。getData()和exists()设置数据观察。getChildren()设置子节点观察。此外，还可以认为不同的返回数据有不同的观察。getData()和exists()返回节点的数据，而getChildren()返回子节点列表。所以，setData()将为znode触发数据观察。成功的create()将为新创建的节点触发数据观察，为其父节点触发子节点观察。成功的delete()将会为被删除的节点触发数据观察以及子节点观察（因为节点不能再有子节点了），为其父节点触发子节点观察。

观察维护在客户端连接到的ZooKeeper服 务器中。这让观察的设置、维护和分发是轻量级的。客户端连接到新的服务器时，所有会话事件将被触发。同服务器断开连接期间不会收到观察。客户端重新连接 时，如果需要，先前已经注册的观察将被重新注册和触发。通常这都是透明的。有一种情况下观察事件将丢失：对还没有创建的节点设置存在观察，而在断开连接期 间创建节点，然后删除。

## 4.1 ZooKeeper 关于观察的保证

l         观察与其他事件、其他观察和异步回应是顺序的。ZooKeeper客户端库保证一切都是按顺序分发的。

l         客户端将在看到znode的新数据之前收到其观察事件。

l         观察事件的次序与ZooKeeper服务看到的更新次序一致。

## 4.2  关于观察需要记住的

l         观察是一次触发的：如果想在收到观察事件之后收到未来修改的通知，必须再次设置观察。

l         因为观察是一次触发的，而收到观察事件和发送新的请求、再次建立观察之间是有延迟的，所以不能可靠地观察到节点的所有修改。应该要准备处理在收到观察事件和再次设置观察之间，节点被多次修改的情况。（可以不处理，但至少要知道这种情况是可能的）

l         一个观察对象，或者函数/上下文对，只会因为某个通知而触发一次。比如说，对同一个文件使用exists和getData调用，设置相同的观察对象，然后文件被删除，则观察对象只会被调用一次，带有文件删除通知。

l         与服务器断开连接期间（比如说，服务器故障）不能收到任何观察事件，直到连接重新建立。因此，会话事件是发送给所有未决观察处理器的。可使用会话事件进入到安全模式：断开连接期间不会收到任何事件，进程应该谨慎操作。

# 5 使用ACL 的访问控制

ZooKeeper使用ACL控制对节点的访问。ACL的实现同Unix文件访问权限非常相似：采用权限位来定义允许/禁止的各种节点操作，以及位应用的范围。与标准Unix权限不同的是，ZooKeeper节点不由用户(文件所有者)、组和其他这三个标准范围来限制。ZooKeeper没有节点所有者的概念。取而代之的是，ACL指定一个ID集合，以及与这些ID相关联的权限。

还要注意的是，ACL仅仅用于某特定节点。特别是，ACL不会应用到子节点。比如说，/app只能被ip:172.16.16.1读取，/app/status可以被所有用户读取。ACL不是递归的。

ZooKeeper支持可插入式鉴权模式。使用scheme:id的形式指定ID，其中scheme是id对应的鉴权模式。比如说，ip:172.16.16.1是地址为172.16.16.1的主机的ID。

客户端连接到ZooKeeper，验证自身的时候，ZooKeeper将所有对应客户端的ID都关联到客户端连接上。客户端试图存取节点的时候，ZooKeeper会在节点的ACL中校验这些ID。ACL由(scheme:expression,perms)对组成。expression的格式是特定于scheme的。比如说，(ip:19.22.0.0/16,READ)给予任何IP地址以19.22开头的客户端以READ权限。

## 5.1 ACL 权限

ZooKeeper支持下述权限：

l         CREATE：可创建子节点

l         READ：可获取节点数据和子节点列表

l         WRITE：可设置节点数据

l         DELETE：可删除子节点

l         ADMIN：可设置节点权限

从WRITE权限中分离出CREATE和DELETE可以取得更好的访问控制。使用CREATE和DELETE的情况：

l         希望A可以设置节点数据，但是不能CREATE或者DELETE子节点。

l         没有DELETE的CREATE权限：客户端通过在某父目录中创建节点来创建请求。此时希望所有客户端可以添加节点，但是只有请求处理器可以删除节点。（这与文件的APPEND权限类似）

此外，ADMIN权限存在的原因是，ZooKeeper没有文件所有者的概念。某些情况下ADMIN权限可以指定实体的所有者。ZooKeeper不支持LOOKUP权限（目录上的、允许进行LOOKUP的执行权限位，即使不能列出目录内容）。每个用户都隐含地拥有LOOKUP权限。这仅仅让用户可以取得节点状态。（问题是，如果想对一个不存在的节点进行zoo\_exists()调用，没有权限可以检查）

### 5.1.1 内置的ACL 模式

ZooKeeper内置下述ACL模式：

l         **world**具有单独的ID，代表任何用户。

l         **auth**不使用任何ID，代表任何已确认用户。

l         **digest**使用username:password字符串来生成MD5散列值，用作ID。身份验证通过发送明文的username:password字符串来进行。用在ACL表达式中时将是username:base64编码的SHA1密码摘要。

l         **ip**使用客户端主机IP作为ID。ACL表达式的形式是addr/bits，表示addr的最高bits位将与客户端主机IP的最高bits位进行匹配。

### 5.1.2 ZooKeeper C客户端API

ZooKeeper C库提供下述常量：

l         const int ZOO\_PERM\_READ;//可读取节点的值，列出子节点

l         const int ZOO\_PERM\_WRITE;//可设置节点数据

l         const int ZOO\_PERM\_CREATE;//可创建子节点

l         const int ZOO\_PERM\_DELETE;//可删除子节点

l         const int ZOO\_PERM\_ADMIN;//可执行set\_acl()

l         const int ZOO\_PERM\_ALL;//OR连接的上述所有标志

下面是标准的ACL ID：

l         struct Id ZOO\_ANYONE\_ID\_UNSAFE;//('world','anyone')

l         struct Id ZOO\_AUTH\_IDS;//('auth','')

空的ZOO\_AUTH\_IDS标识字符串应该解释为“创建者的标识”。

ZooKeeper有三种标准ACL：

l         struct ACL\_vector ZOO\_OPEN\_ACL\_UNSAFE;//(ZOO\_PERM\_ALL,ZOO\_ANYONE\_ID\_UNSAFE)

l         struct ACL\_vector ZOO\_READ\_ACL\_UNSAFE;//(ZOO\_PERM\_READ,ZOO\_ANYONE\_ID\_UNSAFE)

l         struct ACL\_vector ZOO\_CREATOR\_ALL\_ACL;//(ZOO\_PERM\_ALL,ZOO\_AUTH\_IDS)

ZOO\_OPEN\_ACL\_UNSAFE是完全开放自由的ACL：任何应用程序可以对节点进行任何操作，以及创建、列出和删除子节点。ZOO\_READ\_ACL\_UNSAFE给予任何应用程序以只读访问权限。CREATE\_ALL\_ACL给予节点创建者所有权限。创建者在使用这种ACL创建节点之前，必须已经通过服务器的身份鉴别（比如说，使用"digest"模式）。

下述ZooKeeper操作用于处理ACL：

l         int zoo\_add\_auth(zhandle\_t\* zh,const char\* scheme,const char\* cert,int certLen,void\_completion\_t completion,const void\* data);

应用程序使用zoo\_add\_auth函数向服务器验证自身。如果想使用不同的模式和/或标识来进行身份验证，可以多次调用这个函数。

l         int zoo\_create(zhandle\_t\* zh,const char\* path,const char\* value,int valuelen,const struct ACL\_vector\* acl,int flags,char\* realpath,int max\_realpath\_len);

zoo\_create()创建新的节点。acl是与节点相关的ACL列表。父节点必须设置了CREATE权限位。

l         int zoo\_get\_acl(zhandle\_t\* zh,const char\* path,struct ACL\_vector\* acl,struct Stat\* stat);

这个函数返回节点的ACL信息。

l         int zoo\_set\_acl(zhandle\_t\* zh,const char\* path,int version,const struct ACL\_vector\* acl);

这个函数替换节点的ACL列表。节点必须设置了ADMIN权限。

下面是一段使用上述API来进行foo模式的身份验证，然后创建具有仅创建者可访问权限的临时节点/xyz的示例代码。

**注意**

这是一个展示如何与ZooKeeper ACL交互的非常简单的示例。更合适的C客户端实现示例请看../trunk/src/c/src/cli.c。

<……省略示例代码……>

# 6  插入式身份验证

ZooKeeper运行在各种使用不同身份验证模式的环境中，所以它有一个完全插入式的身份验证框架。内置的身份验证模式也是使用这个框架的。

要理解身份验证框架如何工作，首先必须理解两种主要的身份验证操作。框架首先要验证客户。这通常在客户端连接到服务器后立即进行，由验证客户端发送的信息，或者验证收集的关于客户端的信息，并且将其关联到连接两个步骤构成。框架进行的第二个操作是在ACL中找出客户端对应的实体。ACL实体就是<idspec,permissions>对。idspec可能是与连接关联的身份验证信息相匹配的简单字符串，或者是一个可以计算出身份验证信息的表达式。进行匹配是身份验证插件要实现的任务。下面是身份验证插件必须实现的接口：

第一个方法，getScheme返回标识插件的字符串。因为支持多种身份验证方法，所以每个身份验证凭证，或者说idspec总是带有scheme:前缀。ZooKeeper服务器使用身份验证插件返回的模式字符串来确定将模式应用到哪些id。

handleAuthentication在客户端发送与连接相关联的身份验证信息时被调用。客户端指定身份验证信息的模式。ZooKeeper服务器将信息传递给getScheme返回值与客户端传递的模式值相匹配的身份验证插件。handleAuthentication通常在确定身份验证信息不正确时返回错误，或者使用cnxn.getAuthInfo().add(new Id(getScheme(),data))将身份验证信息关联到连接。

身份验证插件与设置和使用ACL相关。为节点设置ACL时，ZooKeeper服务器会将条目的id部分传递给isValid(String id)方法。插件必须验证id具有正确的形式。比如说，ip:172.16.0.0/16是一个有效的id，但是ip:host.com则不是。

如果新的ACL含有auth条目，则isAuthenticated用于确定与连接相关联的身份验证信息是否要添加到ACL中。某些模式不应该包含在auth中。比如说，如果指定了auth，则客户端的IP地址不会被看作是id，不应该添加到ACL中。

检查ACL时，ZooKeeper调用matches(String id,String aclExpr)。函数需要将客户端的身份验证信息与相应的ACL条目进行匹配。为找出应用到客户端的条目，ZooKeeper服务器找出每个条目的模式，如果有客户端的、这个模式的身份验证信息，则matches(String id,String aclExpr)会被调用，id设置为先前通过handleAuthentication添加到连接的身份验证信息，aclExpr设置为ACL条目的id。身份验证插件使用其逻辑进行匹配，确定id是否包含在aclExpr中。

有两个内置的身份验证插件：id和digest。可通过系统属性添加额外的插件。ZooKeeper服务器启动时会查找以zookeeper.authProvider.开头的系统属性，将这些属性的值解释为身份验证插件的类名。可使用-Dzookeeper.authProvider.X=com.f.MyAuth来设置这些属性，或者在系统配置文件中添加类似于下面的条目：

应该注意，要确保后缀是唯一的。如果有重复的，如-Dzookeeper.authProvider.X=com.f.MyAuth和-Dzookeeper.authProvider.X=com.f.MyAuth2，只会使用一个。此外，所有服务器必须定义有同样的插件，否则客户端使用插件提供的身份验证模式连接到某些服务器时会有问题。

# 7  一致性保证

ZooKeeper是高性能、可伸缩的服务。读和写操作都设计为高速操作，虽然读比写更快。原因是在读操作中，ZooKeeper可返回较老的数据，这源自ZooKeeper的一致性保证：

l         顺序一致性：一个客户端的更新将以发送的次序被应用。

l         原子性：更新要么成功，要么失败，没有部分结果。

l         单一系统镜像：无论连接到哪个服务器，客户端将看到同样的视图。

l         可靠性：一旦应用了某更新，结果将是持久的，直到客户端覆盖了更新。这个保证有两个推论：

1．如果客户端得到成功的返回码，则更新已经被应用。某些失败情况下（通信错误、超时等），客户端不知道更新是否已经应用。我们采取措施保证最小化失败，但这个保证只对成功的返回码有效。（这称作是Paxos中的单一条件）

2．服务器从失败恢复时，客户端通过读请求或者成功更新看到的任何更新，都不会回滚。

l         及时性：保证客户端的系统视图在某个时间范围（大约为十几秒）内是最新的。在此范围内，客户端要么可看到系统的修改，要么检测到服务终止。

使用这些一致性保证，就可以很容易地单独在ZooKeeper客户端构建如领导者选举、护栏、队列以及可恢复的读写锁等高层功能。更多细节请看[Recipes and Solutions](http://zookeeper.apache.org/doc/trunk/recipes.html)。

注意：有时候开发者会错误地假定一个ZooKeeper实际上没有提供的保证：

l         跨客户端视图的并发一致性

ZooKeeper并不保证在某时刻，两个不同的客户端具有一致的数据视图。因为网络延迟的原因，一个客户端可能在另一个客户端得到修改通知之前进行更新。假定有两个客户端A和B。如果客户端A将一个节点/a的值从0修改为1，然后通知客户端B读取/a，客户端B读取到的值可能还是0，这取决于它连接到了哪个服务器。如果客户端A和B读取到相同的值很重要，那么客户端B应该在执行读取之前调用sync()方法。

所以，ZooKeeper本身不保证修改在多个服务器间同步地发生，但是可以使用ZooKeeper原语来构建高层功能，提供有用的客户端同步。（更多信息，请看[ZooKeeper Recipes](http://zookeeper.apache.org/doc/trunk/recipes.html)）

# 8  绑定

ZooKeeper客户端库以两种方式提供：Java和C。下面几节描述这两种绑定。

## 8.1 Java 绑定

ZooKeeper的Java绑定由两个包组成：org.apache.zookeeper和org.apache.zookeeper.data。组成ZooKeeper的其他包由内部使用或者是服务器实现的组成部分。org.apache.zookeeper.data由简单地用作容器的类构成。

ZooKeeper Java客户端使用的主要类是ZooKeeper类。这个类的两个构造函数的不同仅仅在于可选的会话ID和密码。ZooKeeper支持进程的不同实例间的会话恢复。Java程序可以将会话ID和密码保存到稳态存储中，然后重启、恢复程序先前实例使用的会话。

创建ZooKeeper对象的时候，会同时创建两个线程：一个IO线程和一个事件线程。所有IO在IO线程中发生（使用Java NIO）。所有事件回调则在事件线程中进行。重连到ZooKeeper服务器和维持心跳等会话维持活动在IO线程中进行。同步方法的回应也在IO线程中进行。所有异步方法的回应，以及观察事件则在事件线程中处理。对于这个设计，有一些事情需要注意：

l         所有同步调用和观察回调将按次序进行，一次一个。调用者可以进行任何想要的处理，但是在此期间不会处理其他回调。

l         回调不会阻塞IO线程或者同步调用的处理。

l         同步调用可能不会以正确的次序返回。比如说，假设客户端进行下述处理：提交一个watch设置为ture的、对节点/a的异步读取，然后在读取操作的完成回调中执行一个对/a的同步读取。（可能是不好的实现，但是是合法的，这只是一个简单的例子）

如果在异步读取和同步读取之间，对/a进行了修改，则客户端库将在同步读取返回之前接收到一个事件，表明/a已经被修改。但是因为完成回调阻塞了事件队列，同步读取将在观察事件被处理之前返回/a的新值。

最后，关于关闭的规则很直接：一旦被关闭或者接收到致命事件（SESSION\_EXPIRED和AUTH\_FAILED），ZooKeeper对象就变成无效的了。关闭后，两个线程被关闭，后续对zookeeper句柄的任何访问都将导致不确定的行为，应该避免。

## 8.2 C 绑定

C绑定有单线程和多线程库。多线程库易于使用，跟Java API非常相似。多线程库将创建用于处理连接维持和回调的IO线程与事件分发线程。通过暴露在多线程库中使用的事件循环，单线程库允许在事件驱动应用中使用ZooKeeper。

有两个共享库：zookeeper\_st和zookeeper\_mt。前者提供了异步API和回调，可集成到应用程序的事件循环中。这个库存在的目的仅仅是为了支持没有pthread可用，或者pthread不稳定的平台（如FreeBSD 4.x）。在其他场合，应用开发者应该链接zookeeper\_mt，它同时支持同步和异步API。

### 8.2.1  安装

如果从Apache代码仓库检出的代码创建客户端库，执行下面的步骤。如果从apache下载的工程源代码包开始创建，则跳到步骤3。

1．在ZooKeeper顶级目录（.../trunk）执行ant compile\_jute。这将在../trunk/src/c目录中创建"generated"目录。

2．修改当前目录为../trunk/src/c，执行autoreconf -if，以启动autoconf、automake和libtool。请确认安装了2.59或者更高版本的autoconf。跳到步骤4。

3．如果从工程源代码包开始创建，解压缩源代码包，cd到zookeeper-x.x.x/src/c目录。

4．执行./configure <your-options>以生成makefile。对于这一步，configure工具支持下述有用的选项：

l         --enable-debug 启用优化和调试信息。（默认是禁用的）

l         --without-syncapi 禁止同步API支持，不创建zookeeper\_mt库。（默认是启用的）

l         --disable-static 不创建静态库。（默认是启用的）

l         --disable-shared 不创建共享库。（默认是启用的）

l         注意：关于执行configure的一般信息，请看INSTALL文件。

5．执行make或者make install，创建并且安装库。

6．要生成ZooKeeper API的doxygen文档，可执行doxygen-doc。所有文档将放置到docs子目录中。默认情况下，这个命令只生成HTML。关于其他文档格式的信息，请执行./congiure --help。

### 8.2.2 使用C 客户端

要测试客户端，可运行ZooKeeper服务器（关于如何运行，请看工程wiki页面的指示），使用作为安装过程一部分创建的某个cli应用程序来连接到服务器。下面的例子显示了使用cli\_mt（多线程，与zookeeper\_mt库一同创建），但是也可以使用cli\_st（单线程，与zookeeper\_st库一同创建）：

这个客户端应用程序提供了一个执行简单ZooKeeper命令的Shell。成功启动并且连接到服务器之后，程序显示shell提示符。现在就可以输入ZooKeeper命令了。比如说，创建一个节点：

验证节点已经创建：

应该可以看到根节点的子节点列表。

在应用程序中使用ZooKeeper API时，应该记住：

1．包含ZooKeeper头文件：#include <zookeeper/zookeeper.h>

2．如果创建多线程客户端，请使用-DTHREADED编译器标志，以启用库的多线程版本，并且链接到zookeeper\_mt库。如果创建单线程客户端，不要使用-DTHREADED，并且链接到zookeeper\_st库。

关于Java和C的使用示例，请看[程序结构和简单示例](file:///F:\temp\Juziyue\HTMLEdit\CSAE\temp.html#_10%20%E7%A8%8B%E5%BA%8F%E7%BB%93%E6%9E%84%E5%92%8C%E7%AE%80%E5%8D%95%E7%A4%BA%E4%BE%8B)。

# 9 创建块：ZooKeeper 操作指南

本节描述开发者可对ZooKeeper服务器执行的所有操作。这些信息比本手册前面章节的内容要更底层，但是比ZooKeeper API参考的层次要高。

## 9.1  处理错误

Java和C绑定都可能报告错误。Java客户端绑定通过抛出KeeperException来报告错误，对异常对象调用code()可取得特定的错误码。C客户端绑定返回ZOO\_ERRORS枚举定义的错误码。在两个语言绑定中，API回调都指示结果值。关于所有可能的错误码及其含义的详细信息，请看API文档（Java绑定的javadoc，C绑定的doxygen）。

## 9.2 连接到ZooKeeper

## 9.3  读取操作

## 9.4  写入操作

## 9.5  处理观察

## 9.6 其他ZooKeeper 操作

# 10  程序结构和简单示例

# 11  转向：常见问题和解决

现在你了解ZooKeeper了，它高效、简单，你的程序可以工作，但是等等……，出了点问题了。下面是ZooKeeper用户遇到的一些陷阱：

1．使用观察的时候，必须处理已连接的观察事件。ZooKeeper客户端同服务器断开连接期间，不会收到修改通知，直到重新连接。如果观察一个节点的出现，则断开连接期间会错过节点的创建和删除事件。

2．必须测试ZooKeeper服务失败。一旦多数服务器不活动，ZooKeeper服务会失败。问题是：你的程序可以处理这种情况吗？在真实世界中，客户端到ZooKeeper的连接可能会断开（ZooKeeper服务器失败和网络分区是连接丢失的常见原因）。ZooKeeper客户端库会处理连接恢复，并且让你知道发生了什么，但是你必须保证可以正确恢复状态和任何已失败的未决请求。在实验室确认程序是正确的，而不是在产品中：用由多个服务器组成的ZooKeeper服务进行测试，并且进行一些重启。

3．客户端和服务器使用的服务器列表应该一致。如果客户端的列表只是真正的服务器列表的一部分，程序可以工作，虽然不是最优的；但是如果客户端包含不在集群中的服务器，则不能工作。

4．注意在哪里放置事务日志。事务日志是ZooKeeper中最关乎性能的部分。返回响应之前，ZooKeeper必须将事务同步到媒体中。专用事务日志设备是取得良好性能的关键。如果只有一个存储设备，把跟踪文件放到NFS中，并且增加snapshotCount；这不能解决问题，但是有一定的改善。

5．正确设置Java的最大堆大小。避免交换是非常重要的。大多数情况下，不必要地放入磁盘肯定会降低性能到不可接受的程度。记住，在ZooKeeper中，一切都是顺序的，如果一个请求触及磁盘，其他排队的请求也会触及磁盘。

为避免交换，试试将堆大小设置为拥有的物理内存大小减去操作系统和缓存需要的大小。确定最优堆大小的最好方法是执行负载测试。如果因为一些原因而不能进行测试，请采取保守估计，选择一个小于将导致交换的值。比如说，在4GB的机器上，3GB是一个保守的开始值。

除了正式的文档之外，开发者还有其他一些信息来源：

l         ZooKeeper白皮书

由Yahoo!研究院发布的关于ZooKeeper设计和性能的权威讨论。

l         API参考

关于ZooKeeper API的完整参考。

l         [Hadoup 2008峰会上的ZooKeeper演讲](http://us.dl1.yimg.com/download.yahoo.com/dl/ydn/zookeeper.m4v)

Yahoo!研究院的Benjamin Reed关于ZooKeeper的视频介绍。

l         [护栏和队列教程](http://wiki.apache.org/hadoop/ZooKeeper/Tutorial)

Flavio Junqueira编写的、关于使用ZooKeeper实现简单护栏和生产者-消费者队列的优秀教程。

l         [ZooKeeper：一个可靠的、可伸缩的分布式协调系统](http://wiki.apache.org/hadoop/ZooKeeper/ZooKeeperArticles)

Todd Hoff编写的一篇文章（07/15/2008）。

l         [ZooKeeper解决方案](http://zookeeper.apache.org/doc/trunk/recipes.html)

关于ZooKeeper各种同步解决方案实现的虚拟层面的讨论：事件处理、队列、锁和两阶段提交。