Zookeeper

2021年12月22日 9:38

etcd

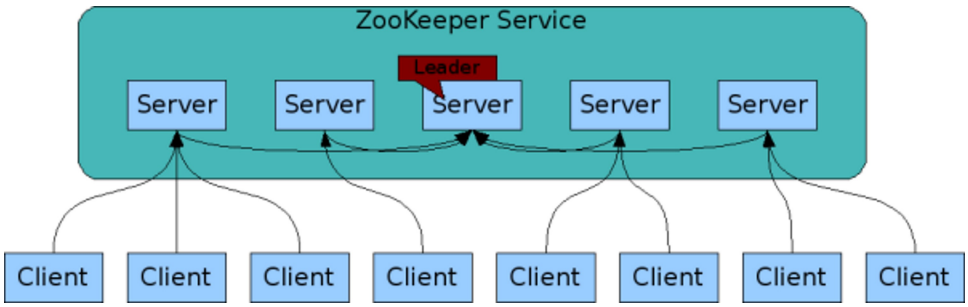
==========

zookeeper 的客户端调用过于复杂，Apache Curator 就是为了简化zookeeper客户端调用而生，利用它，可以更好的使用zookeeper

==========

分布式协调调度框架，主要用来解决 分布式集群中 应用系统的一致性问题。
本质上是 一个分布式的小文件存储系统。提供 基于类似于文件系统的目录树方式 的数据存储，并且可以对 树中的节点 进行有效管理。
客户端可以 监控存储在ZK内部的数据，从而可以达到 基于数据的 集群管理。比如：统一命名服务(dubbo)，分布式配置管理(solr的配置集中管理)，分布式消息队列(sub/pub)，分布式锁，分布式协调等功能

zk架构构成



Leader

zk集群工作的 核心角色
集群内部 各个服务器的 调度者
事务请求(写操作) 的唯一调度 和 处理者，保证集群事务处理的 顺序性；对于 create, setData, delete 等有写操作的请求，则需要 统一转发给 Leader 处理，leader 需要决定 编号，执行操作，这个过程 称为一个事务。

Follower

除非 客户端 非事务(读操作) 请求
转发 事务请求 给leader
参与 集群 Leader 选举投票

Observer

观察zk 集群的 最新状态 变化 并将 这些状态同步过来， 其对于 非事务请求 可以进行独立处理，对于 事务请求，则转发给 leader 服务器进行处理。
不参与任何 投票，只提供 非事务服务，通常用于 在不影响 集群事务处理能力的 前提下 提示 集群 非事务处理能力。 增加了集群的 并发的读请求。

zk特点

一个leader，多个follower 组成的集群
leader 负责进行投票的发起 和 决议，更新系统状态。
follower 用于 接收 客户请求 并向 客户端 返回结果，在 选举leader 过程中 参与投票。
集群中 只要有 半数以上 节点存活，zk集群就能正常服务
全局数据一致： 每个server 保存一份相同的数据副本，client 无论连接哪个 server，数据都是一致的。
更新请求 顺序进行
数据更新原子性。

zk搭建方式

3种，单机， 集群， 伪集群(一台服务器上运行多个zk实例)

znode类型

分为3大类

1. 持久性节点，创建后一直存在服务器，直到 删除操作
2. 临时性节点，声明周期 和 客户端绑定在一起， 客户端会话结束，节点会被删除掉。
   和持久性节点不同的是， 临时节点不能 创建 子节点。
3. 顺序性节点，在创建节点的时候，在节点名后面 加一个 数字后缀，来表示其顺序。

开发中，创建节点的时候， 可以通过组合 来生成 4种 节点类型： 持久节点，持久顺序节点，临时节点，临时顺序节点。

事务ID

zk中，事务是指 能改变 zk 服务器状态的 操作，包括 数据节点的 创建，删除，内容更新等操作。 对于每个事务请求，zk 会分配一个 全局唯一的 事务id，用 zxid 来表示，通常是一个 64位的数字，每个 zxid 对应一次 更新操作， 从这些 zxid中 可以 间接识别出 zk 处理这些 更新请求的 全局顺序。

znode的状态信息

```
[zk: localhost:2181(CONNECTED) 5] get -s /zk-persist
123
cZxid = 0x8
ctime = Mon Jan 03 17:09:50 CST 2022
mZxid = 0x8
mtime = Mon Jan 03 17:09:50 CST 2022
```

```
pZxid = 0x8
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 3
numChildren = 0
```

field	description
czxid	创建znode的zxid
mzxid	最近一次修改znode的zxid(创建、删除、set直系子节点、set自身节点都会计数)
pzxid	最近一次修改子节点的zxid(创建、删除直系子节点都会计数，set子节点不会计数)
ctime	创建znode的时间，单位毫秒
mtime	最近一次修改znode的时间，单位毫秒
version	修改znode的次数
cversion	修改子节点的次数(创建、删除直系子节点都会计数，set子节点不会计数)
aversion	该znode的ACL修改次数
ephemeralOwner	临时znode节点的session id，如果不是临时节点，值为0
dataLength	znode携带的数据长度，单位字节
numChildren	直系子节点的数量(不会递归计算孙节点)

watcher机制

zk使用 watcher 机制 实现分布式数据的 发布/订阅 功能。

zk允许客户端 想客户端注册一个 watcher 监听，当服务端的 一些指定事件触发了 这个 watcher，那么 zk 就会 向指定的 客户单发送一个 事件通知 来实现 分布式的 通知功能。

zk的watcher机制 主要包括： 客户端线程，客户端watcherManager,zookeeper服务器 3部分。

具体工作流程：

1. 客户端在向 zk 服务器 注册的同时，会将 watcher 对象存储在 客户端的 waterManager中。
2. 当zk服务器触发watcher事件后，会向客户端发送通知。
3. 客户端线程 从 watcherManager 中 取出 对应的 watcher 对象 来执行 回调逻辑。

```
create [-s] [-e] path data
    -s代表创建一个顺序节点；
    -e代表创建一个临时节点；
    若不指定，则创建一个持久节点。
```

ls命令：可以列出zk指定节点下的所有子节点，但只能查看指定节点下的第一级的所有子节点；

get命令：可以获取zk指定节点的数据内容和属性信息。

$ set [-s] [-v version] path data
其中data就是要更新的新内容，version表示数据版本。在zk中，节点的数据是有版本概念的，这个参数用于指定本次更新操作时基于ZNode的哪一个数据版本进行的。

delete [-v version] path
若删除节点存在子节点时，那么无法直接删除该节点，必须先删除子节点，再删除父节点。

==========

https://zookeeper.apache.org/doc/r3.8.0/zookeeperProgrammers.html

ZooKeeper Programmer's Guide
Developing Distributed Applications that use ZooKeeper

Introduction
本文档是为 希望常见 利用zk 协调服务的分布式应用程序的 开发人员提供 指南。

前4部分对各种 zk 概念进行 高层次的 讨论。这些对于 理解 zk 如何工作 及 如何使用它都是必要的。 不包含代码，但假定 你熟悉 与分布式计算相关的 问题，前4部分包括
The ZooKeeper Data Model
ZooKeeper Sessions
ZooKeeper Watches
Consistency Guarantees

接下来4个 提供了使用的 编程信息：
Building Blocks: A Guide to ZooKeeper Operations
Bindings
Gotchas: Common Problems and Troubleshooting
。。不要问为什么是3个， 就是3个。

最后有一个附录，包含了 指向其他有用的zk相关信息的link

The ZooKeeper Data Model

zk有一个 分层命名空间，很像分布式文件系统。唯一的区别是 命名空间中的 每个节点 都可以有 与其关联的 数据 及子节点。这就像 拥有了一个 允许文件也成为目录的 文件系统。节点的 路径 始终表示为 规范的，绝对的，斜杠分隔的路径。没有相对路径。能用在path中的unicode 字符有下面的 限制：

1. null字符(\u0000) 不能是path名 的一部分。(这会导致 c binding 出现问题)
2. 不能使用以下字符，因为它们显示不好，或呈现方式混乱，\u0001 - \u001F 和 \u007F
3. \u0009F
4. 下面的不允许：\ud800 – uF8FF, \uFFF0 – uFFFF.
5. .字符可以用作 另一个名称的一部分，但是 . 和 .. 不能单独用于 指示 路径上的 节点，因为 zk 不使用 相对路径。 下面是无效的： /a/b/./c /a/b/../c
6. token "zookeeper" 是保留的

ZNodes
zk tree 中 每个node 都被称为 znode。
znode 维护了一个 stat 结构，其中包括了 data change，acl change 的 version号。 stat结构也有时间戳。 版本号+时间戳 允许 zk 来 验证 cache，协调更新。
每次 znode 的 数据change， 版本号增加。
例如，每当客户端 检索data， 它会收到 数据的 版本。 当客户端执行update 或 delete，它必须 提供 它正在change的znode的数据的版本。 如果提供的 version 和 数据实际版本不匹配，则 update 会失败。 (这个行为可以被覆盖)

Note
在分布式应用工程中，node 可以指 主机，服务器，集群成员，客户端进程等。
在zk中，znode指 数据节点。 服务器是指 组成zk服务器的机器，quorum peers是指 组成一个整体的服务器，客户端是指 使用zk服务的 任何主机或进程。

znode是 程序员访问的 主要entity。它们有几个特点 值得一提：

1. watches
   客户端 可以在 znode 上设置 监视。 对该 znode 的 change 会触发 监视，然后 清除监视。当 watch 触发时，zk会向客户端发送通知。

2. Data Access
   存储在 命名空间中每个znode 的数据 是 原子 读和写的。
   读取，获取 与znode 关联的 所有数据字节
   写入 替换所有数据
   每个node 有 访问控制列表(ACL) 来限制 谁可以做什么
   zk不是为 通用数据库 或 大型对象存储 而设计的。 它管理协调数据(coordination data)。这些数据可以 以 配置，状态信息，集合点(rendezvous) 等形式出现。各种形式的协调数据的 一个共同特性是 它们相对较小：以 kb为单位。zk客户端和服务器 实现 sanity check，以确保 znode 的数据 小于 1m，但数据应该远小于这个值。
   在相对较大的数据上 进行操作 会导致 某些操作 比其他操作 花费更多时间，并且 会影响 某些操作的 延迟，因为通过 网络 将 大数据移动到 存储介质 上需要额外的时

间。
如果需要 大数据存储， 应该： 那数据放到 大容量存储系统上，如NFS，HDFS，并将指向 存储位置的 指针 存储在 zk 中。

3. Ephemeral Nodes 临时节点
zk也有 临时节点的概念。生命周期跟随 创建它的session。session结束，znode被删除。 因为这种行为，临时znode 不允许有 子节点。 可以使用 getEphemerals() API 来检索 session的 临时节点 列表。
getEphemerals() 检索 session 为 给定路径 创建的 临时节点列表。 如果路径为空，列出 session 的所有临时节点。
一个简单的用例是：如果 需要收集 session 的临时节点列表 以进行 重复数据输入检查， 并且 节点 是按顺序创建的，所以你不知道 重复检查的名字。这种情况下，使用这个api 来获取 session的 节点列表，这可能是 服务发现的 典型用例。。。。。。那就是 service instance 使用 session 连接zk， session 关闭/中断后， 临时节点就被删除，这个 service instance 就不可用。 。 不过 session的存活判断，还有 session 中断/关闭 后的 clean 。

4. Sequence Nodes -- Unique Naming
创建znode时，你还可以 请求 zk 在 路径末尾增加一个 递增的计数器。这个计数器 对于 父 znode 是唯一的。 格式是 %010d，即 10个数字(前缀0补足)。
这个功能的使用 参考 Queue Recipe.
Note:
保存下一个序列号的计数器 是 由 父znode 维护的 有符号int， 当超过 INT_MAX 时 ，变成 INT_MIN， 会 溢出。
。。。这个溢出 是可接受的，还是 error？ 最多 INT_MAX - INT_MIN 个 节点， 超过会怎么样？

5. Container Nodes (add in 3.6.0)
zk 有 container znode 的概念。 container znode 是 特殊用途的 znode， 可用于：leader，lock等。 当 container 的 最后一个 child 被删除时， 这个container 会在 未来某个时间点 被服务器删除。
由于这个属性，你应该 准备好 在 container znode 中 创建 child znode 时 收到 KeeperException.NoNodeException。 即，在 container znode 中 创建 child znode 时， 总是检查 KeeperException.NoNodeException，并且 在捕捉到这个 异常后， 重新创建 container znode。
。。。container znode 可能被服务器删除， 所以 需要 catch container znode 不存在的异常。

6. TTL Node (3.6.0)
在创建 PERSISTENT or PERSISTENT_SEQUENTIAL znode 时， 你可以 为 znode 设置一个 ttl (以 ms 为单位)。 如果 znode 在 ttl 内 没有被修改 且 没有 子节点， 它在 未来某个时间点 被服务器删除。。。。。。未来。。
Note:
ttl node 必须通过 系统属性 启用， 因为 默认下 是禁用的。 如果 你在没有 启用 的情况下，创建ttl node， 会抛出 KeeperException.UnimplementedException。

Time in ZK
zk以多种方式 跟踪时间

1. zxid，zk状态的 每次更改 都会收到 一个 zxid (zookeeper transaction id)。这 向 zk 公开了 所有change 的顺序。
2. Version numbers， 对node 的每个 change 会导致 node 的 某个version number 增加。有3个 version number：

version	znode 数据的change次数
cversion	znode子节点的chang次数
aversion	znode的ACL的change次数

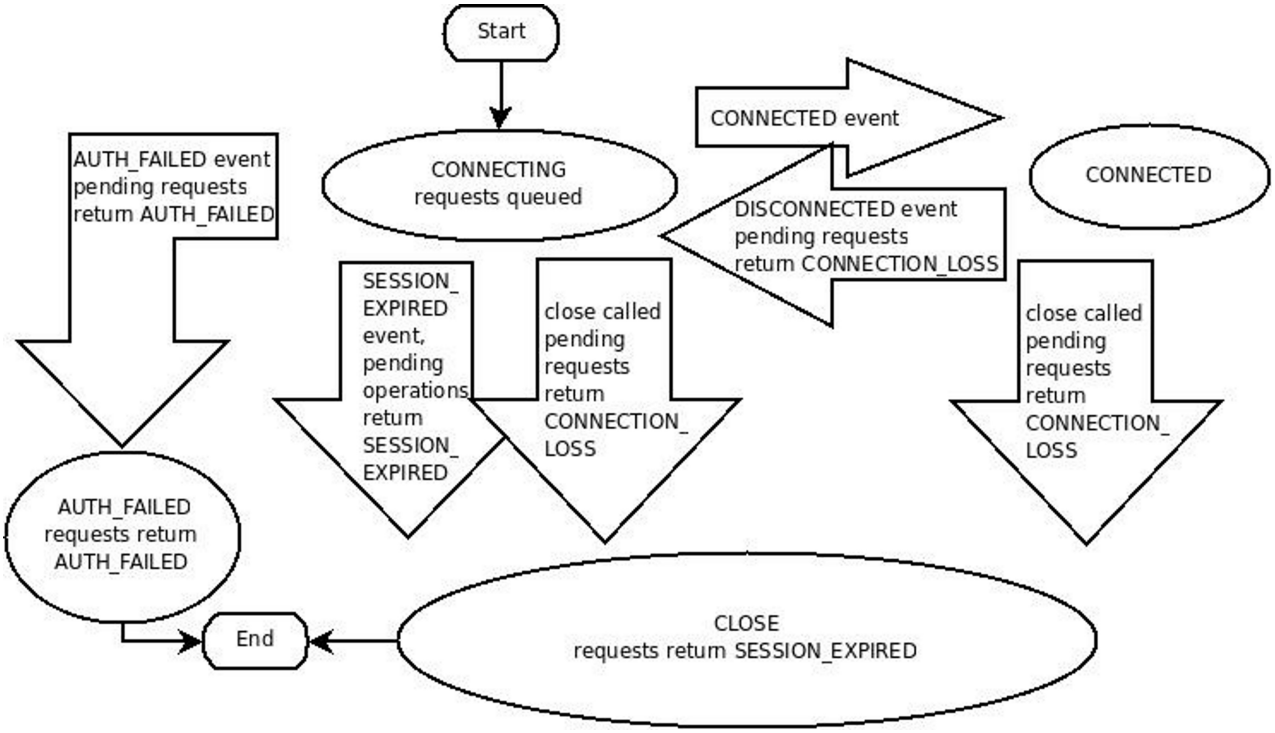
3. Ticks，当使用 multi-server zk时，服务器使用 ticks 来定义 event 的timing，例如 status upload,session timeout,peers 之间的 connection timeout 等 event。tick time 只是通过 最小 会话 超时 (2倍于 tick time) 来间接暴露； 如果 客户端请求的 session 最小超时 小于 最小会话超时，服务器会告诉 client ， session timeout 实际上是 最小会话超时。
4. read time，zk 不使用 real time，clock time， 除了 在 znode 创建 和 修改时 将 时间戳 放入 stat 结构中。

ZooKeeper Stat Structure

zk中 每个 znode 的 Stat 结构 由 下列字段组成：

czxid	The zxid of the change that caused this znode to be created.
mzxid	The zxid of the change that last modified this znode.
pzxid	The zxid of the change that last modified children of this znode.
ctime	The time in milliseconds from epoch when this znode was created.
mtime	The time in milliseconds from epoch when this znode was last modified.
version	The number of changes to the data of this znode.
cversion	The number of changes to the children of this znode.
aversion	The number of changes to the ACL of this znode.
ephemeralOwner	The session id of the owner of this znode if the znode is an ephemeral node. If it is not an ephemeral node, it will be zero.
dataLength	The length of the data field of this znode.
numChildren	The number of children of this znode.

ZooKeeper Sessions



zk客户端 使用 language binding 来创建 handle 指向 service， 来 建立 和 zk service 的 session。
一旦建立， handle 以 CONNECTING 状态开始，client 尝试连接到 组成zk 服务的 服务器之一，然后切换到 CONNECTTED 状态。
正常操作期间， 客户端 handle 将处于这2种状态 之一。
如果发生不可恢复的错误，如 session expiration(session过期)，身份认证失败，或 app 显式 关闭 handle，则 handle 变成 CLOSED 状态。

要创建 client session， app必须提供一个 连接字符串，是逗号分隔的 host:port 。每个对应于 zk 服务器 (如 "127.0.0.1:4545" or "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002")。 zk客户端库将选择 任意一个服务器并尝试连接。如果 此连接失败，或者客户端应该任何原因 和 服务器 断开连接，客户端将自动 尝试 列表的 下一个服务器，直到 (重新) 建立连接。

在3.2.0 中添加：可以在 connection string 后面加一个 chroot 后缀。 这会导致，运行的客户端命令 都是 相对于 这个 chroot 路径的 (类似于 unix 的 chroot 命令)。
这在 multi-tenant (多租户) 环境 中特别有用，在这种环境中， 特定 zk 服务的 每个 用户 都可能具有 不同的 root 权限。 这使得重用更加简单，因为每个用户 都可以 对他的 app 进行编码，就像它在 / 下一样， 而实际位置 可以在 部署时确定。
"127.0.0.1:4545/app/a" or "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002/app/a"，客户端会 在 /app/a 下， 所有的路径都是 相对于 这个路径的， 所以 "foo/bar" 最终是在 "/app/a/foo/bar" 下运行的。

当客户端获得 zk service 的 handle 时， zk 会创建一个 zk session，以 64bit 数字表示，并分配给 client。 如果 clien 连接到 不同的 zk 服务器，它将 发送 session id 作为 connection handshake 的一部分。作为一项安全措施， 服务器会为 session id 创建一个密码，所有的 zk server 都可以 validate。 密码 会和 session id 一起 发送给 客户端。 每次和 新server 重新建立 session时， client 都会发送 此 密码和session id。

创建zk session的 zk客户端库 的 调用 参数之一 是 session timeout (毫秒数)。 client 发送 它的超时设置 ， server 响应 它可以给予 client 的 超时设置。 目前的实现 要求 timeout 至少是 server配置的tickTime 的2倍 ，最多是 tickTIme 的20倍。 zk client api 允许 获得 协商后的 超时设置。

当客户端(session) 从 zk集群 中 变得分区时，它将开始搜索 session 创建期间指定的 服务器列表。 最终，当客户端 和 至少一个服务器之间的 连接重新建立时， session 会再次 转换到 CONNECTED 状态(如果在timeout时间内 重新连接)，或者 转换为 EXPIRED 状态(如果在timeout时间外 重新连接)。
不建议 创建新的session 对象(一个新 ZooKeeper.class 或 c binding中的 Zookeeper handle) 来 (处理) disconnection。 zk client 库 会为你 处理 重新连接。特别是 我们在客户端库 中内置了 启发式方法 来处理 诸如 羊群效应 等情况 。 只有在你 收到 session expiration (mandatory) 时 才创建一个 新会话。

session expiration 被 zk 集群本身 管理，而不是由 客户端。
当zk 客户端 和 集群建立session时，它会提供之前描述的 "超时"值。 集群使用 这个 值 来决定 客户端 session 是否 expire。当 集群在 指定的 timeout 周期内 没有 听到 心跳，就会发生 expire。在会话到期时，集群将 删除 该session 拥有的 所有 临时节点，并立即将 更改 通知到 所有连接的客户端。此时，过期的session 的客户端 和 集群是 disconnected的，所以它不会 收到 session过期的 通知 直到 它重新和 集群 建立 连接，它才会被告知 session expire。
客户端会一直处于 断开状态，直到 与集群重新建立 TCP 连接，此时 会话过期的 观察者 会收到 会话过期的 通知。
会话过期的 watcher(观察者) 看到的 会话过期的 状态转换 例子：

1. connected：session已经建立 并且 客户端 正在和 集群通信 (客户端，服务器 通信正常)
2. …客户端从集群中 分区
3. disconnected，客户端 失去了 和 集群的 连接
4. …一段时间后，在 timeout 时间后，集群 使得 session 过期，客户端没有看到 任何内容，因为它和 集群断开了 连接
5. …一段时间后， 客户端重新 和 集群 网络层连接。
6. expired，最终 客户端 重新连接到 集群，它被 通知 expiration

建立zk session方法 的 另一个参数是 默认观察者。 当 client 发生任何 状态 更改时，会通知 观察者。例如，如果 客户端 失去 和 服务器的 连接 或 client 的 session 过期，客户端将被通知。 watcher 应该认为 初始状态是 disconnected (比如在任何状态更改事件 被发送给watcher 之前)。在新connection的情况下， 发送给 观察者的 第一个事件 通常是 会话 连接事件。

会话通过 客户端发送的request 保持活动状态。如果 session idle 了一段 可能 会timeout 时间，client 发送 PING 请求 来让 session 处于alive。 PING请求不仅让 zk 服务器知道 client 仍然处于 活动状态，而且 还允许 client 验证 它和 zk服务器的 连接 是否处于 活动状态。 ping的 触发 足够保守，确保有 合理的时间 检测到 dead connection 并重写建立新 conn 到服务器。

一旦成功建立 到 服务器的连接， 基本上 有 2种情况 会使得 client lib 生成 connectionloss (这个是 c binding 的 result code， java 的 exception)， 当 执行 一个 同步 或异步 操作 且 满足 以下条件之一时：

1. 应用 在 非存活的 session 上调用操作

2. zk client 从 一个 有挂起操作的 server 上 断开连接。

在3.2.0 中添加 -- SessionMovedException 。 这是一个 内部异常，client 通常看不到。发生这个异常的原因是在 connection上 收到了 来自 与另一个server 重新建立的 session 的请求。 这个错误的正常原因是 client 向 server 发送请求，但 网络数据包 延迟，因此客户端 超时 并 连接到 新服务器。 当延迟的 数据包 到达 第一台服务器时，旧服务器检测到会话 已经移动，并关闭 客户端连接。 客户端通常不会看到这个错误，因为 它们不从 旧连接中读取。(旧连接通常被关闭)。可以看到这种情况的一种 情况是 但2个客户端 尝试使用保存的 session id 和 密码 重新建立 相同的 连接时。 其中一个客户端 将重新建立连接，第二个client 将断开连接 (导致 这2个客户端 无限期地 尝试 重建 conn/session)。

Updating the list of servers. 我们允许客户端 通过 提供一个 新的 逗号分隔的 host:port 列表 来更新 connection string，每个 host:port 对应一个 zk 服务器。 该函数 会 调用 概率load-balance算法，可能导致 客户端与 当前host 断开连接， 这个算法是为了 让 新list 中 每个server 的 连接数 统一。 如果客户端 连接的 当前host 不在 列表中，则 会 中断连接， 否则 将根据 server 的数量 是增加 还是 减少 以及 减少多少 来决定。

例如(第一个例子)，如果 之前的 connectString 包含 3台host， 现在的 list 包含 以前的3台 + 2台新的，连接到 3台主机的 40% 的client 会移动到 某台新host 来 load-balance. 这个算法会导致 client 以 0.4 的概率 断开 与当前host 的 连接，这种情况下， client 将连接到 随机选择的 2台新host 之一。

另一个例子，假设我们有5台主机，现在更新list 来 删除 2台主机，那些连接到 剩余3台主机的 客户端 依然保持连接， 连接到 被删除的2台的主机的 客户端 都需要 移动到 另外3台之一，随机选择。如果连接断开，客户端将进入特殊模式，在该模式下，他使用概率算法 选择新server 来连接，而不是 轮询。

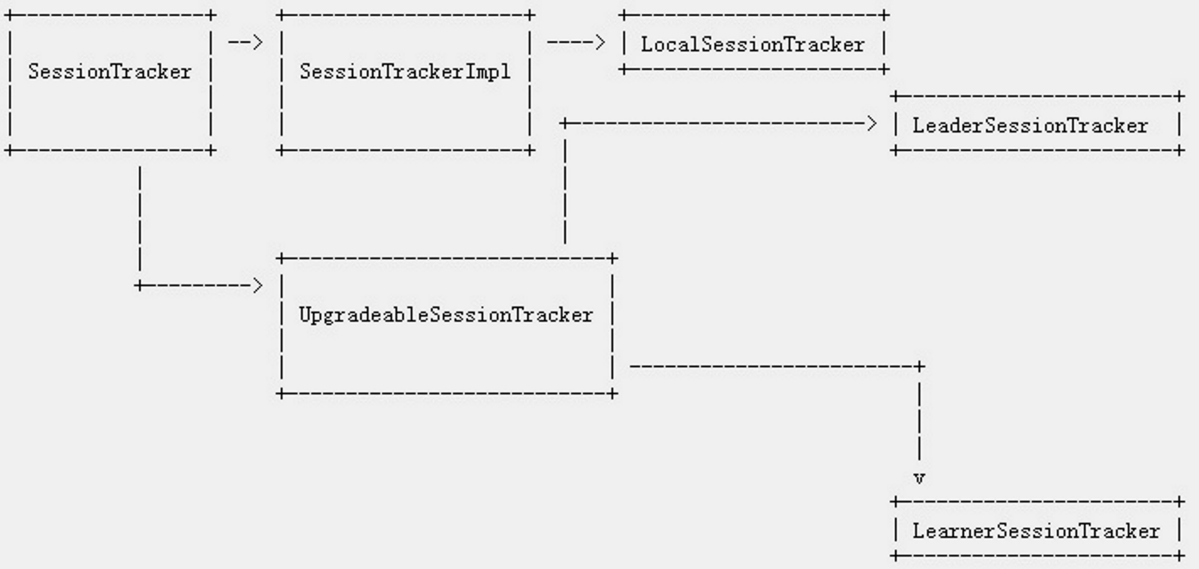
在第一个例子中，每个客户端 以0.4 的概率 断开连接， 但是一旦做出决定，它会尝试 连接到 随机的 新服务器，并且只有当它 无法连接到 任何新服务器时 才会 尝试连接到 那些旧服务器。在找到一台server， 或 尝试了 new list 中的所有server但都失败 后，client 返回到 正常操作模式，这种模式下，它会 从 connectString 中选择 任意服务器 并尝试 连接到它。 如果失败，继续 循环尝试 不同的 随机服务器。

Local session, 3.5.0增加， 主要由zookeeper-1147 实现。
背景：zk 中 session的 创建和 关闭 成本很高，因为它们需要 仲裁确认(quorum confirmation)， 当需要处理 数千个client的 connection 是，它们成为 zk 的瓶颈。所以在3.5.0后，增加了 新的 session 类型， local session， 它没有 普通(全局) session 的全部功能，这个功能 通过 打来 localSessionEnabled 来使用。

当 localSessionUpgradingEnabled 是disable
1. local session 不能创建 临时节点
2. 一旦 local session 丢失，用户 就无法 使用 session id/password 来 re-establish， session 和 它的 watch 将永远消失。
   Note: 丢失tcp连接 并不一定意味着 session 丢失。 如果可以在 会话 超时之前 与 同一个zk 重新建立连接，那么 client 可以继续 (它无法移动到 另一台服务器)
3. 当 local session 连接时，session 信息 仅在 client 连接的 zk server 上维护。leader 不知道 这个session的 创建，所以 没有 状态 被写入到 磁盘。
4. ping，expiration(过期)，其他session 状态由 当前 session 连接到的 server 处理。

当 localSessionUpgradingEnabled 启用

1. local session 可以 自动升级为 global session。
2. 创建 新session时，它会 保存在 本地 的 LocalSessionTracker 中。 随后 可以根据需要将其 升级为 全局session(如，创建临时节点)。如果 请求升级，则从 本地集合中删除 session，同时保持相同的 session id。
3. 目前，只有操作： 创建临时节点 需要 将 session 从local 升级为 global。 这是因为 临时节点的 创建 很大程度依赖于 global session。如果本地 session 可以在不升级到 global session 的情况下 创建 临时节点，则会导致 不同节点之间的 数据不一致。lead 还需要 知道 session 的 生命周期，以便在 关闭/到期 时 清除 临时节点。这需要一个 global session，因为 local session 绑定到 特定的服务器。
4. 升级过程中 一个session 即可以是 local ，也可以是 global， 但升级操作 不能被 2 个线程 同时调用。
5. ZooKeeperServer(standalone) 使用 SessionTrackerImpl； LeaderZookeeper使用 LeaderSessionTracker，它hold SessionTrackerImpl(global) 和 LocalSessionTracker(如果启用)； FollowerZooKeeperServer 和 ObserverZooKeeperServer 使用 LearnerSessionTracker,它 hold了 LocalSessionTracker。
   UML图：



Q&A

为什么需要选项 来禁用 local session upgrade?

在想要处理大量 客户端的 大型部署中，我们知道 客户端通过 观察者 进行 连接，而观察者 应该只是 local session。 所以 这更像是 防止 有人 意外创建 大量 临时节点 和 全局会话。

session在 几时 创建?

当前实现中，它会在 处理 ConnectRequest 和 createSession 请求到达 FinalRequestProcessor 时， 尝试 创建 local session

如果 session 的创建 被发送到 server A，然后客户端 断开， 然后连接到 其他Server B，往B 上再次发送它， 然后 断开连接， 然后 重新连回 A 。 会发生什么

What happens if the create for session is sent at server A and the client

disconnects to some other server B which ends up sending it again and then disconnects and connects back to server A?
。。？主要是 黄色的， 这是 从A断开 然后 到B 上去， 还是 原本就和B连着，然后断开？但是 客户端 能连多个 server ？ 不行吧。 但是 下面回答中 第一句是 重新连回B。。上面没有说啊。
。。还有，创建session 是从 A发出，还是 发到A？ 应该是 发到 A？ 应该是 本地创建了session，然后 告知 A ，但是下面是 A发出 createSession。。

> 当客户端 reconnect B时， sessionid 不存在于 B 的 local session tracker中。 所以 B 会 发送 验证包(validation packet)，如果A 发出的 CreateSession在 validate packet 之前到达 ，client 可以 连接。否则， client 将 收到 session 过期，因为仲裁 尚未知道 此会话。 如果 客户端 还尝试 再次 连回 A，则 会话已经从 local session tracker 中删除， 所以A需要 向 leader 发送一次 验证包。 结果应与 B 相同，具体取决于 请求的 时间。
>
> When a client reconnects to B, its sessionId won't exist in B's local session tracker. So B will send validation packet. If CreateSession issued by A is committed before validation packet arrive the client will be able to connect. Otherwise, the client will get session expired because the quorum hasn't know about this session yet. If the client also tries to connect back to A again, the session is already removed from local session tracker. So A will need to send a validation packet to the leader. The outcome should be the same as B depending on the timing of the request.

ZooKeeper Watches
zk中所有 read 操作， getData(),getChildren(),exists() 都可以选择 设置watch 作为 副作用。下面是 zk 对于 watch 的定义：watch事件 是一次性触发，发送到设置watch 的 client，当 watch的数据发生 变化时。3个关键点 在 watch 的定义中：

1. one-time trigger。 当数据发生变化时， 会向客户端发送一个 watch 事件。例如，如果客户端执行 getData("/znode1", true)之后 对 /znode1 的数据的更新或删除，客户端 会获得 /znode1 的watch event。 如果 znode1 再次改变，则不会 发送 任何 watch event，除非 client 进行了 另一次 read 以设置 新的watch
2. sent to the client。这意味着 event 在 去 发起watch的client 的路上，但可能 在 change操作的成功return的代码 到达 发起这个change操作的client 之前， 没有办法到达 发起watch的client。 watch event 异步发给 watcher。zk提供了 一个排序保证： client 会先收到 watch event 然后才会收到 change event。 网络延迟 或其他原因 可能导致 不同的client 在不同的 时间 看到 watch 和 更新的return code。 关键是 不同 client 看到的 所有 东西 都是 一个顺序的。
3. the data for which the watch was set。 指 节点 节点可以进行change 的 不同方式。 将zk 视为 维护2个 watch list 可能有所帮助：data watches 和 child watches。 getData() 和 exists() 设置 data watches。 getChildren() 设置 child watches。 或者，考虑根据 返回的 数据类型 设置 watch 可能有所帮助。getData(),exists() 返回 有关 节点数据的 信息，getChildren() 返回 子节点列表。 因此，setData() 将 触发 被set数据的 znode 的 watches (如果set数据成功的话)。成功的 create() 将触发正在 创建的 znode 的 data watch 和 parent znode 的 child watch。 成功的 delete() 将触发 正在删除的 znode 的 data watch 和 child watch (因为不能有children)，及 parent znode 的 child watch

watches 在 client 连接的 zk server 上 本地维护。 这允许 watch 在 set，maintain，dispatch 方面是 轻量级的。 当client 连接到 新server，任何session event 都会触发

watch。 和server 断开时，无法收到 watch。 client 重连时，任何以前注册的 watch 会被重新注册，并 在需要时 触发。一般来说，这一些都是透明地发生的。有一种情况 可能会丢失 watch： 如果在 断开连接时 创建和删除 znode，则会丢失 尚未 创建的znode的 watch。

New in 3.6.0: client 可以在 znode 上设置 永久的 递归watch，这些 watch 在被触发时不会 被删除， 会被 在watch 注册的znode 上 或 它的child (递归下去) 上的 change 触发。
New in 3.6.0: Clients can also set permanent, recursive watches on a znode that are not removed when triggered and that trigger for changes on the registered znode as well as any children znodes recursively.
。。。。

Semantics of Watches，语义
我们可以设置 watch 通过 3个 调用，这3个调用 读取了 zk的state ： exists，getData,getChildren。 下面展示了 watch 可以触发的 事件 和 启用watch的 call：

1. created event: 通过 exists 的call 启用
2. deleted event： exists,getData,getChildren
3. changed event: exists,getData
4. child event: getChildren

Persistent, Recursive Watches
3.6.0的新功能：上述 标准 watch的 一个变种， 你可以设置 一个 在触发时 不会被 移除的watch。 另外，这些watch 触发了 event type:
NodeCreated,NodeDeleted,NodeDataChanged。 并且，可选地，watch 可以 被 znode 和 它的 子孙znode 触发。 持久递归watch 不会触发 NodeChildrenChanged，因为这个是 多余的。

使用 addWatch() 来增加 永久watch。 触发语义 和 保证 和 标准watch 一样。唯一例外是递归持久watch 不会触发 child changed event，因为它们是多余的。 使用具有 watcher type WatcherType.Any 的 removeWatches() 来删除 持久watch。

Remove Watches
我们可以 通过调用 removeWatches 来 删除在znode 上注册的 watch。 此外，即使没有服务器连接，zk client 也可以通过 将local flag 设置为true 来 在本地删除 watch。
下面列表详细说明了 成功 移除 watch 后 将触发的 event：

1. Child Remove event: 通过 getChildren 添加的 watcher
2. Data Remove event: 通过 exists 或 getData 添加的 watcher
3. Persistent Remove event: 通过 添加 持久化watcher 添加的 watcher

What ZooKeeper Guarantees about Watches
对于watch，zk保证下面：

1. watches 相对于 其他event，其他watch，异步回复 是 进行排序的。 zk client 库 保证 按顺序分派所有内容。
2. client 会 先看到 它所watch的znode 的 watch event， 然后 看到 那个znode 上的 新数据。

3. zk的 watch event的 order 对应了 zk service 看到的 update 的 order。

Things to Remember about Watches

1. 标准watch是一次性触发器： 如果你收到 watch event 并希望 收到 未来的change 的通知，你 必须 设置另一个 watch
2. 由于标准watch是一次性的，并且 在 获取event 和 发送新请求以设置watch 之间 有延迟，所以 你可能无法 可靠地 看到 zk中 znode 发生的 每一次更高。 准备好处理 这种 case： 在收到watch event 和 设置新的watch 之间 数据被多次 修改 (你可能不在乎，但是至少要 意识到 它可能发生)
3. 对于一个给定的 notification， watch 或 function/context pair 只会被触发一次。例如，如果 为 一个文件 注册了 一个 会被 getData,exists 触发的 watch， 然后 这个文件 被删除了， 这个watch 只会被 调用一次，并且收到 文件的delete notification。
4. 当你 和 server 断开连接时 (如 server故障)，你不会 获得任何 watch 直到 连接被重建。出于这个原因， session events 被发送到 所有 未完成的 watch handlers。使用 session event 来进入 安全模式：在断开连接时 你不会收到 event ，因此 你的进程 在这个模式下 应该 谨慎行事。

ZooKeeper access control using ACLs

zk使用 acl 来控制 对它的znode 的 访问。 ACL 实现 和 unix file access 权限 类似，它使用 权限bit 来 允许/禁止 对于 znode 的 各种操作，以及这些bit 适用的范围。
和标准unix权限不同，zk znode 不受 user，group，word 这3个标准scope的限制。zk 没有 znode owner 的概念。
ACL 设置了 id 和 这些id具有的权限

另外注意，acl仅适用于特定的znode。 特别是 它不会递归应用到 children。例如，如果 /app 被限制为 只能 ip:172.16.16.1 读取，并且 /app/status 是 world readable， 那么任何人都可以读取 /app/status。
acl 不是递归的。

zk支持 可插拔的 auth 方案。Id 使用 格式： schema:expression ，schema 是 id对应的 auth schema。有效的expression 集合 由 schema 定义。 例如，ip:122.22.22.2 是使用 ip schema 的 地址为 122.22.22.2 的主机的 id， 而 digest:bob:password 是 使用 digest schema 的 名字是 bob的 用户的 id。

当客户端连接到 zk 并对其自身进行 身份验证时， zk 会将 客户端对应的 所有 id 与 client 相关联。 当client 尝试访问 节点时， 这些id 会根据 znode 的 acl 进行检查。
acl 由成对的 (scheme:expression,perms) 组成。 表达式的格式 特定于方案。例如，ip:111.11.0.0/16, READ 将read权限 授予 ip 地址 以 111.11 开头的 任何客户端。

ACL Permissions

支持下面权限：

CREATE: you can create a child node
READ: you can get data from a node and list its children.
WRITE: you can set data for a node
DELETE: you can delete a child node

ADMIN: you can set permissions

create 和 delete 从 write中分离出来，实现更精细的控制。
create 和delete 的 case 如下：
你希望A能够在 zk 节点上 进行设置， 但不能 创建或 删除 children。

只有create，没有delete： client 在 父目录中 创建 zk node。 你希望 所有client 都可以增加，但是 只有 request processor 可以删除。(类似 文件的 append 权限)

由于zk 没有 file owner 的概念，所以 存在 admin 权限。某种意义上，admin 权限 将 client 指定为 owner。
zk 不支持 lookup 权限 (可以查询目录，但是无法list 出目录)。每个人 都隐含地 拥有 lookup 权限。 这允许 你统计一个 节点，但仅此而已。 (问题是，如果你想在 一个 不存在的 节点上 调用 zoo_exists()， 没有 权限 来检查。)

admin 权限在 acl 方面也有特殊 作用： 为了检索 znode 用户的 acl，必须具有 read 或 admin 权限， 但如果没有 admin权限， digest hash value 将被屏蔽。

Builtin ACL Schemes
zk有以下 内置schema
- world
- auth
- digest
- ip
- x509

ZooKeeper C client API
用户建立conn时，用于设置权限的 c 方法 和代码。
跳

Pluggable ZooKeeper authentication

要理解 auth 框架的工作原理，首先要了解2个主要的 身份验证操作。
框架首先 必须对 client 进行身份验证。这通常在 client 连接到 server 后 立即完成，包括验证 从 client 发出 或收集的 关于client的 信息 并将其与 connection 相关联。
框架处理的 第二个操作是 在 acl 中查找与client 对应的 条目。 acl 条目是 <idspec, permission> pair。 idspec 可以是与 connection关联的 身份验证 的 简单string， 也可以是 针对该信息 进行eval 的expression，由 auth 插件的实现 来 进行匹配。
下面是 auth 插件 必须实现的 接口：

```
public interface AuthenticationProvider {
    String getScheme();
    KeeperException.Code handleAuthentication(ServerCnxn cnxn, byte authData[]);
    boolean isValid(String id);
    boolean matches(String id, String aclExpr);
    boolean isAuthenticated();
}
```

第一个方法 getScheme 返回标识 插件的 字符串。
。。。跳了。

3.6.0增加了 可插拔式auth的 抽象类

```
public abstract class ServerAuthenticationProvider implements
AuthenticationProvider {
    public abstract KeeperException.Code handleAuthentication(ServerObjs
serverObjs, byte authData[]);
    public abstract boolean matches(ServerObjs serverObjs, MatchValues
matchValues);
}
```

Consistency Guarantees 一致性保证
zk是一种高性能，可扩展的服务。 读取和写入都 为更快 而设计，尽管read 比 write 更快。 这样做的原因是，在读取的情况下，zk 可以服务较旧的数据，这是由于 zk 的一致性保证下面：

1. sequential consistency: 一个client 发出的update 会以它们 发送的顺序 被apply
2. Atomicity： update 要么成功，要么失败，没有中间态
3. single system image: client 会看到 service 的 相同的view， 不管 client 连接了哪个server， 即，cient 永远不会看到 system 旧的view，即使 client 的 同一个session 由于失败 而连接到 另一个server。
4. reliability：一旦update 被应用，它将 从那时起 永远存在，直到 client 覆盖掉 update，这个guarantee 有下面2个 推论：
    a. 如果 client 收到 成功的return code，那么 update 已经被 应用。 由于一些失败 (通信错误，超时，等) client 不知道 update 是否已经被应用。 我们采取一些步骤 来减少失败，但是 这个 guarantee 只有在 成功的return code 时 才 成立。
    b. client 通过 read请求 或 成功的update 看到的任何 update，不会被 rollback，当 服务从失败中 恢复过来时。
5. timeliness：client 看到的 system的view 被确保是 一定时间内(数十秒) 最新的。 要么在这个时间(几十秒)内，client 看到 system change， 要么 发现 服务挂了。

使用这些 一致性 保证，很容易 建立 高级功能，比如 leader election(选举)，barriers(屏障)，queues，read/write revocable(可撤回的，可废止的) lock。

Note
有时，开发者 错误地假定 一些 zk上实际不存在的 gurantee。比如
Simultaneously Consistent Cross-Client Views： zk 不保证 所有 instance 都是同步的，2个不同的client 将 具有 相同的 zk data 的view。 由于一些因素，如网络延迟， 一个客户端可能 执行了 一个 update， 但是另外一个客户端 还没有收到 change 的 通知。 考虑2个client，A和B。如果 A 设置了 znode /a 的值 从0 到1， 然后 告诉 B 去读取 /a， B可能读到 旧值 0，取决于 B 连到 哪台server。 如果 A 和B读取到 相同值 是非常重要的，B 应该调用 sync() 方法，在 执行read 前。 所以 zk 本身不保证 change 的occur 会被 synchronously 分发 到 所有 server， 但是 zk primitives 可以被用来 构建 高级 function，来提供 有用的 client synchronization。

Bindings
zk client libraries 有 2种语言： Java 和 C。 下面的章节描述了这些

Java Binding
2个package 构成了 zk java binding: org.apache.zookeeper 和
org.apache.zookeeper.data。
其他的 zk 的package 被 内部使用 或 是 server impl 的一部分
org.apache.zookeeper.data 包 由 generated class 组成，这些 仅用作 容器。

zk java client 用到的 主要的类是 ZooKeeper 类。它的2个 构造器 的差别仅仅是 可选的 session id 和 password。

==========

==========

https://zookeeper.apache.org/doc/r3.8.0/recipes.html

==========

==========