Zookeeper 中文教程

作者 马军伟

目录

前言

本文档的电子书版本的来源是：转载自小马过河 - Zookeeper 文档目录的中文文档教程，原文链接是：
http://www.majunwei.com/category/201612011952003333/

关于作者

小马过河，致力于为广大 IT 技术爱好者提供官方、系统的中文学习资料；为了保证实现这一目标，我们整理的资料，全部来自于官网、图书、或有权威的技术专家；同时，我们非常欢迎有共同目标和热爱学习的朋友加入我们，共同学习和进步。

QQ 群：547104190

第 1 章 Zookeeper 简介

1.1 概述

Zookeeper 是一个分布式的、开源的分布式应用协调服务。它暴露了一组简单的基础原件，分布式应用可以在这些原件之上实现更高级别的服务，如同步、配置维护、群组、和命名。它被设计成容易编程实现的，并且使用一个常见的文件系统的树型结构的数据模型。它运行在 Java 中，并且绑定了 Java 和 C。

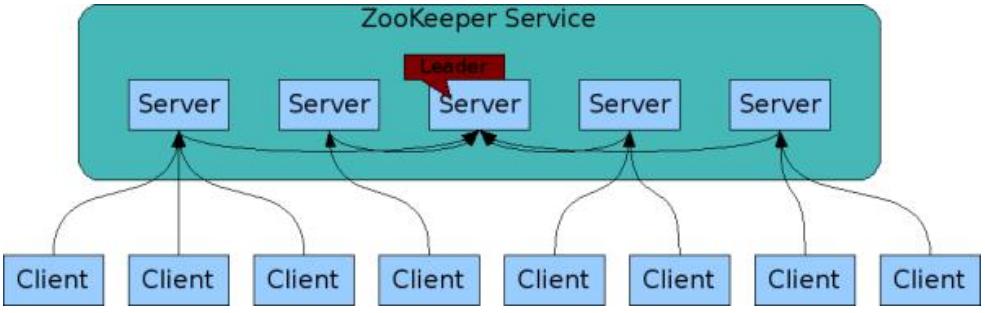
众所周知，协调服务很难做对。它们特别容易发生像文件竞争条件问题和死锁的错误。Zookeeper 的动机就是缓解分布式应用从头开始实现分布式协调服务的责任。

1.2 设计目标

简单。Zookeeper 允许程序通过一个共享的类似于标准文件系统的有组织的分层命名空间分布式处理协调。命名空间包括：数据寄存器 - 在 Zookeeper 中叫 znodes, 它和文件、目录一样。和一般文件系统的不同之处是，它的设计就是为了存储，Zookeeper 的数据保持在内存中，这就意味着它可以实现高吞吐量和低延迟的数据。

Zookeeper 的实现提供了一个优质的高性能、高可用，严格的访问顺序。Zookeeper 的性能方面意味着它可以用于大型的分布式系统。可靠性方面防止它成为一个单点故障。严格的顺序意味着可以在客户端实现复杂的同步原件。

复制。像分布式处理一样，Zookeeper 自己在处理协调的时候要复制多个主机。



Zookeeper 服务的组成部分必须彼此都知道彼此，它们维持了一个内存状态影像，连同事务日志和快照在一个持久化的存储中。只要大多数的服务器是可用的，Zookeeper 服务就是可用的。

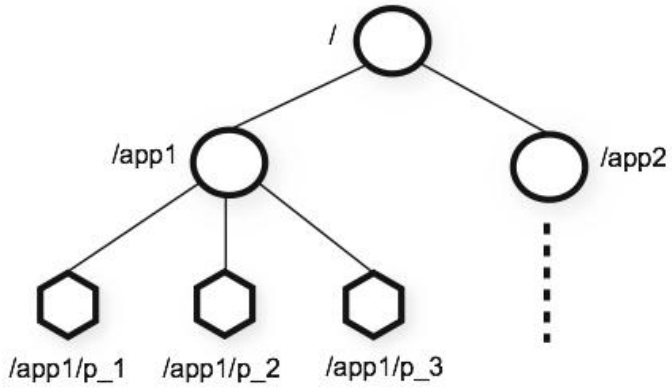
客户端连接到一个单独的服务。客户端保持了一个 TCP 连接，通过这个 TCP 连接发送请求、获取响应、获取 watch 事件、和发送心跳。如果这个连接断了，会自动连接到其他不同的服务器。

序列。Zookeeper 用数字标记每一个更新，用它来反射出所有的事务顺序。随后的操作可以使用这个顺序去实现更高级的抽象，例如同步原件。

快速。它在"Read-dominant"工作负载中特别快。Zookeeper 应用运行在数以千计的机器上，并且它通常在读比写多的时候运行的最好，读写大概比例在 10:1。

1.3 数据模型和分层的命名空间

Zookeeper 提供的命名空间非常像一个标准的文件系统。一个名字是一系列的以'/'隔开的一个路径元素。Zookeeper 命名空间中所有的节点都是通过路径识别。



1.4 节点和临时节点

不像标准的文件系统，Zookeeper 命名空间中的每个节点可以有数据也可以有子目录。它就像一个既可以是文件也可以是目录的文件系统。(Zookeeper 被设计成保存协调数据：状态信息，配置，位置信息，等等，所以每个节点存储的数据通常较小，通常在 1 个字节到数千字节的范围之内)我们使用术语 znode 来表明 Zookeeper 的数据节点。

znode 维持了一个 stat 结构，它包括数据变化的版本号、访问控制列表变化、和时间戳，允许缓存验证和协调更新。每当 znode 的数据有变化，版本号就会增加。例如，每当客户端检索数据时同时它也获取数据的版本信息。

命名空间中每个 znode 存储的数据自动的读取和写入的，读取时获得 znode 所有关联的数据字节，写入时替换所有的数据。每个节点都有一个访问控制列表来制约谁可以做什么。

Zookeeper 还有一个临时节点的概念。这些 znode 和 session 存活的一样长，session 创建时存活，当 session 结束，也跟着删除。

1.5 条件的更新和 watches

Zookeeper 支持 watches 的概念。客户端可以在 znode 上设置一个 watch。当 znode 发生变化时触发并移除 watch。当 watch 被触发时，客户端会接收到一个包说明 znode 有变化了。并且如果客户端和其中一台 server 中间的连接坏掉了，客户端就会收到一个本地通知。这些可以用来[tbd]。

1.6 保证

Zookeeper 是非常简单和高效的。因为它的目标就是，作为建设复杂服务的基础，比如同步。zookeeper 提供了一套保证，他们包括：

- 顺序一致性 - 来自客户端的更新会按顺序应用。
- 原子性 - 更新成功或者失败，没有局部的结果产生。
- 唯一系统映像 - 一个客户端不管连接到哪个服务端都会看到同样的视图。
- 可靠性- 一旦一个更新被应用，它将从更新的时间开始一直保持到一个客户端重写更新。
- 时效性 - 系统中的客户端视图在特定的时间点保证成为是最新的。

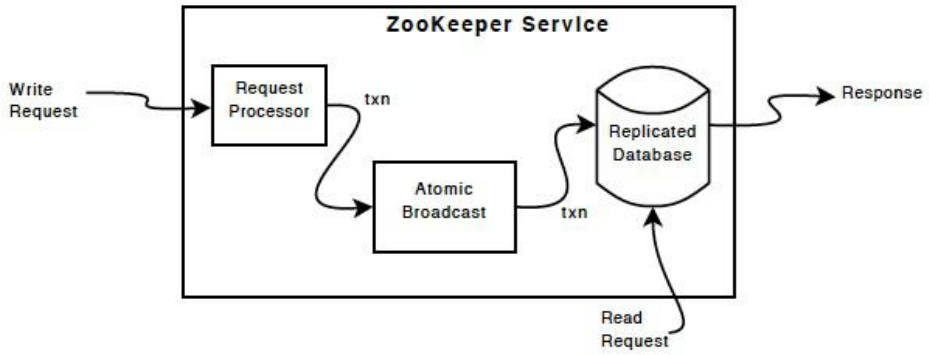
1.7 简单 API

Zookeeper 的设计目标的其中之一就是提供一个简单的程序接口。因此，它只支持这些操作：

- create - 在树形结构的位置中创建节点
- delete - 删除一个节点
- exists - 测试节点在指定位置上是否存在
- get data - 从节点上读取数据
- set data - 往节点写入输入
- get chilren - 检索一个节点的子节点列表
- sync - 等待传输数据

1.8 实现

Zookeeper Compnents 展示了 Zookeeper 服务的高级组件。除了请求处理器的异常之外，组成 Zookeeper 服务的服务器都会复制它们自己组件的副本。



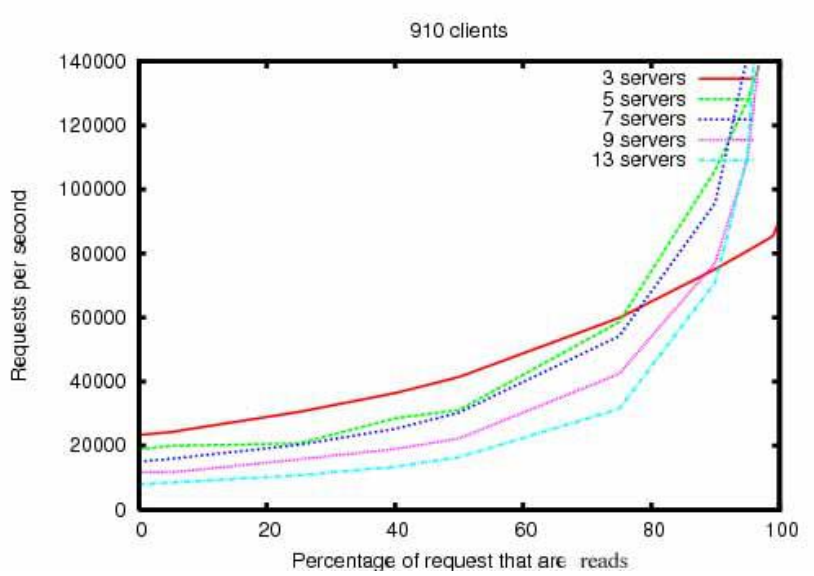
Replicated database 是一个内存数据库，它包含全部的数据树。为了可恢复性，更新记录保存到磁盘上，并且写入操作在应用到内存数据库之前被序列化到磁盘上。
每个 Zookeeper 服务端服务客户端。客户端正确的连接到一个服务器提交请求。每个服务端数据库的本地副本为读请求提供服务。服务的变化状态请求、写请求，被一个协议保护。
作为协议的一部分，所有的写操作从客户端转递到一台单独服务器，称为 leader。其他的 Zookeeper 服务器叫做 follows，它接收来自 leader 的消息建议并达成一致的消息建议。消息管理层负责在失败的时候更换 Leader 并同步 Follows。
Zookeeper 使用了一个自定义的原子消息协议。因为消息层是原子的，Zookeeper 可以保证本地副本从不出现偏差。当 leader 接受到一个写请求，它计算写操作被应用时系统的状态，并将捕获到的新状态转化进入事务。

1.9 使用

程序接口实现 Zookeeper 非常简单。然而，用它你可以实现更高级的操作，如同步基本数据，组成员，所有权，等等。一些分布式应用已经使用了它，[tbd,从白皮书和视频教程添加使用]，获取更多信息，查看[tbd]

1.10 性能

Zookeeper 被设计成高性能的。但真的是这样吗？Yahoo 的 Zookeeper 开发团队的结果表明了它就是这样的。它在应用中读比写多的时候有特别高的性能，因为在写的时候包含所有服务器状态同步。（读比写多是协调服务的典型案例）



上图是Zookeeper3.2版本在dual 2Ghz Xeon and two SATA 15K RPM 驱动配置的服务器上的吞吐量图像。一个驱动作为专门的Zookeeper日志装置。快照写进操作系统驱动。1k的写请求和1K的读取请求。"Servers" 表明了Zookeeper全体的大小，组成Zookeeper服务的服务器数量。接近于30台机器模仿客户端。Zookeeper全体被配置为leaders不允许客户端连接。

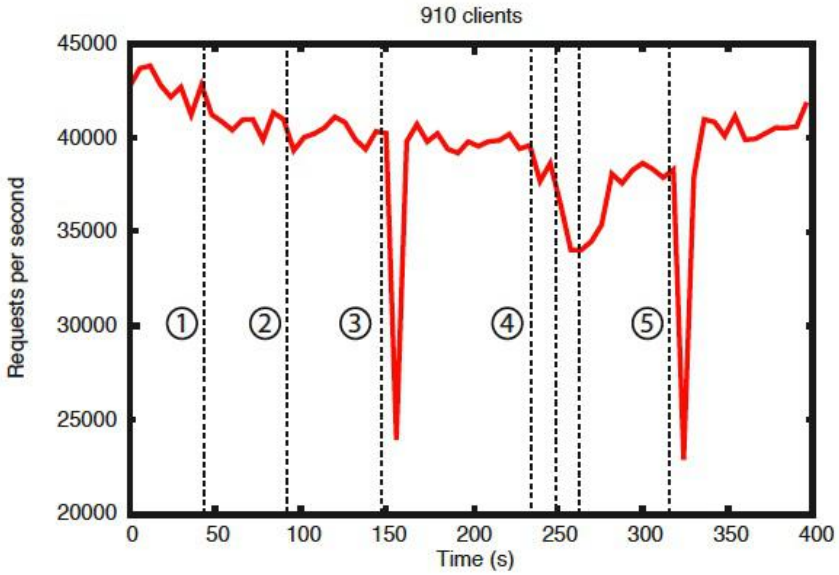
注意：相对于3.1的发布版本，在3.2的版本中读写性能都改善了。

测试基准也表明它是可靠的。Reliability in the Presence of Errors 展示了怎样部署应对各种失败。事件标记出以下几点：

- 一个follower的故障和恢复
- 不同的follower的故障和恢复
- leader的故障
- 两个followers的故障和恢复
- 另一个leader的故障

1.11 可靠性

展示我们启动了7台机器组成的Zookeeper服务注入超时故障的行为。我们之前运行了相同的饱和基准，但现在我们保持写的百分比在不变的30%，这是一个比较保守的工作负载比。



从这张图有几个重要的观察。第一，如果 follows 失败并快速的恢复，Zookeeper 能够维持一个高的吞吐量尽管有故障。但也许更重要的是，leader 选举算法允许系统快速恢复，足以预防吞吐量大幅下降。在我们的观测中，Zookeeper 用了不到 200ms 的时间就选出了一个新的 leader。第三，随着 follower 恢复，Zookeeper 一旦开始处理它们的请求，它能够再次提高吞吐量。

1.12 Zookeeper 项目

Zookeeper 在很多的工业应用中已经有了很多成功的使用。它在 Yahoo 消息代理中用它来做协调和故障恢复服务,它是一个高可扩展的 发布-订阅 模式的系统，它为复制和数据传递管理着数以千计的主题。它在 Yahoo 爬虫中的使用也非常迷人，它也管理着故障恢复。大量的 Yahoo 广告系统也使用 Zookeeper 实现它们可靠的服务。

鼓励所有的用户和开发者加入社区并贡献知识。查看 Zookeeper Project on Apache 获取更多信息。

第 2 章 Zookeeper 入门

2.1 简介

本章内容的主要的目的是让开发人员快速试用 Zookeeper，且包含了单个 Zookeeper 服务器的简单安装介绍，一些验证运行的命令，和一个简单的编程实例。最后，为了方便，还有一些复杂的安装部分，例如运行主从复制的部署，和优化事务日志。然而对于商业部署的完整介绍，请参考 ZooKeeper Administrator's Guide.

2.2 前提条件

查看管理员指南的 System Requirements

2.3 下载

获取 Zookeeper 的发布包，从 Apache Download Mirros 中下载一个最新的稳定版本。

2.4 单机操作

安装 Zookeeper 的单机模式非常简单。服务包含在一个单独的压缩文件中，所以安装只需要创建配置文件。

一旦你下载了一个 Zookeeper 的稳定的发布版本之后，解压并进入根目录。
启动 Zookeeper 之前需要一个配置文件。这是一个示例，创建一个文件在 conf/zoo.cfg:

```
1.      tickTime=2000
2.      dataDir=/var/lib/zookeeper
3.      clientPort=2181
```

这个文件本来可以叫任何名字，此时我们叫它 conf/zoo.cfg。修改 dataDir 的值为一个已经存在的(空的开始)文件夹。下面是每个字段的含义：

2.4.1 tickTime

Zookeeper 使用的基本时间，时间单位为毫秒。它用于心跳机制，并且设置最小的 session 超时时间为两倍心跳时间

2.4.2 dataDir

保存内存数据库快照信息的位置，如果没有其他说明，更新的事务日志也保存到数据库。

2.4.3 clientPort

监听客户端连接的端口。

现在你已经创建了配置文件，可以启动 Zookeeper：

```
1.      bin/zkServer.sh start
```

Zookeeper 的日志信息使用 log4j -- 更多详细信息可以在开发人员指南的 Logging 模块获取。你可以根据 log4j 的配置进入控制台查看日志信息(或日志文件)。这里列出了 Zookeeper 独立运行模式的步骤。没有主从复制，所以如果 Zookeeper 进程故障，服务就会停止。这对于大多数的开发情况是可以的，想要运行 Zookeeper 的主从复制模式，请参见 Running Replicated ZooKeeper.

2.5 管理 Zookeeper 存储

长时间运行的生产系统上的 Zookeeper 存储必须要在外部管理(datadir 和 logs)，查看 maintenance 部分获取更多信息。

2.6 连接 Zookeeper

如果 Zookeeper 在运行中，你可以有几个操作连接到它

JAVA： 使用

```
1.      bin/zkCli.sh -server 127.0.0.1:2181
```

这让你执行简单，类似文件的操作。

C：在 Zookeeper 源文件中的 src/c 子目录中通过运行 make cli_mt 或 make cli_st 编译 cli_mt(多线程的)或 cli_st(单线程的)。查看 src/c 中的 README 获取全部的详细信息。

你可以在 src/c 目录中运行程序：

```
LD_LIBRARY_PATH=. cli_mt 127.0.0.1:2181
```

或

```
LD_LIBRARY_PATH=. cli_st 127.0.0.1:2181
```

这将给你一个简单的 shell 脚本，在 Zookeeper 上执行文件系统的操作。

一旦连接上了，你可以看到一些信息：

```
Connecting to localhost:2181
log4j:WARN No appenders could be found for logger (org.apache.zookeeper.ZooKeeper).
log4j:WARN Please initialize the log4j system properly.
Welcome to ZooKeeper!
JLine support is enabled
[zkshell: 0]
```

从 shell 脚本，输入 help，可以获取一个从客户端可执行的命令列表：

```
[zkshell: 0] help
ZooKeeper host:port cmd args
        get path [watch]
        ls path [watch]
        set path data [version]
        delquota [-n|-b] path
        quit
        printwatches on|off
        createpath data acl
        stat path [watch]
        listquota path
        history
        setAcl path acl
        getAcl path
        sync path
        redo cmdno
        addauth scheme auth
        delete path [version]
        setquota -n|-b val path
```

从这里，你可以尝试一些简单的命令行接口找到一些感觉。第一，通过发行的列表命令开始，像 ls ：

```
[zkshell: 8] ls /
[zookeeper]
```

下一步，通过运行 create /zk_test my_data，创建一个新的 znode。这将创建一个新的 znode 节点和一个相关联的字符串"my_data"：

```
[zkshell: 9] create /zk_test my_data
Created /zk_test
```

使用 ls / 命令查看目录：

```
[zkshell: 11] ls /
[zookeeper, zk_test]
```

注意 zk_test 目录已经被创建了。

接下来，使用 get 命令验证数据是否与 znode 关联上了：

```
[zkshell: 12] get /zk_test
my_data
cZxid = 5
ctime = Fri Jun 05 13:57:06 PDT 2009
mZxid = 5
mtime = Fri Jun 05 13:57:06 PDT 2009
pZxid = 5
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0
dataLength = 7
numChildren = 0
```

我们可以是用 set 命令改变数据与 zk_test 的关联。像：

```
[zkshell: 14] set /zk_test junk
cZxid = 5
ctime = Fri Jun 05 13:57:06 PDT 2009
mZxid = 6
mtime = Fri Jun 05 14:01:52 PDT 2009
pZxid = 5
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0
dataLength = 4
numChildren = 0
```

```
[zkshell: 15] get /zk_test
junk
cZxid = 5
ctime = Fri Jun 05 13:57:06 PDT 2009
mZxid = 6
mtime = Fri Jun 05 14:01:52 PDT 2009
pZxid = 5
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0
dataLength = 4
numChildren = 0
```

(注意我们使用 get 在 setting data 之后，并且它确实改变了。)

最后，让我们 delete 节点：

```
[zkshell: 16] delete /zk_test
[zkshell: 17] ls /
[zookeeper]
[zkshell: 18]
```

先就这样，想探索更多，继续这个文档的其他部分和查看 Programmer's Guide.

2.7 Zookeeper 编程

Zookeeer 有一个 java 绑定和一个 C 绑定。它们在功能上是相同的。C 绑定有两种形式：单线程和多线程。它们的区别仅仅是怎么循环消息。获取更多消息，查看 Programming Examples in the ZooKeeper Programmer's Guide 使用不同 APIs 的实例代码。

2.8 运行主从复制的 Zookeeper

运行 Zookeeper 的独立模式方便评估、开发和测试。但是在生产中，你要运行 Zookeeper 的主从复制模式。相同应用中服务器主从复制组叫做 quorum，并且在主从复制模式中，在 quorum 中的所有服务器有相同的配置文件副本。配置文件和使用独立模式相似，但有一点点的区别，如：

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
```

```
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

新的条目，initLimit 是 Zookeeper 用它来限定 quorum 中的 Zookeeper 服务器连接到 Leader 的超时时间。syncLimit 限制了一个服务器从 Leader 多长时间超时。使用这两种超时，你指定的时间单位使用 tickTime.在这个例子中，initLimit 的超时时间是 5 个标记号，2000 毫秒一个标记，就是 10 秒。
条目 server.x 列出了构成 Zookeeper 服务的服务器。当服务启动时，它通过查找 data 目录中的 myid 文件知道是哪个服务。这个 myid 个文件包含了服务号，用 ASCII.
最后，注意每个服务器名称后面的两个端口号："2888"和"3888"。同事使用前面的端口连接到其他同事。这样的一个连接非常重要，以便于同事之间可以通讯，例如，对更新的顺序取得统一的意见。更具体的说，一个 Zookeeper 的服务器用这个端口连接 follower 到 leader。当一个新的 leader 产生时，follower 使用这个端口打开一个 TCP 连接，连接到 leader。因为默认的 leader 选举也使用 TCP。我们现在需要另一个端口用来 leader 选举。这是在服务器条目的第二个端口。

注意：如果你想在一台机器上测试多台服务器，在服务器配置文件为每个 server.x 指定 servername 为 localhost,和独有的 quorum & leader 选举端口（也就是 2888:3888, 2889:3889, 2890:3890 在上面的示例中）。当然分开 dataDir 和不同的 clientPort 也是非常重要的(在上面的主从复制示例中，在单个主机上运行，你仍然需要三个配置文件)。

2.9 其他优化

有一些其他的配置可以大大提高性能：

- 获取更新的低延迟，有一个专门的事务日志目录非常重要。默认情况下，事务日志作为一个数据快照和 myid 文件放入同一个目录。datalogDir 参数指示一个不同的目录用于事务日志。

第 3 章 深入 Zookeeper 客户端应用

3.1 概述

这篇文档是使用 Zookeeper 协调服务开发分布式应用的开发者编程指南。它包含概念性的和实用性信息。

这个指南开始的 4 个部分对 Zookeeper 的概念提出了高层次的论述。这对理解 Zookeeper 怎样工作和怎样使用它都非常重要。它不包含源代码，但是它呈现了分布式计算的相关问题。第一组的章节是：

- Zookeeper 数据模型
- Zookeeper 会话
- Zookeeper Watches
- 一致性保证

接下来的 4 个章节提供了实际的编程信息，它们是：

- Zookeeper 操作指南
- 绑定
- 程序结构，简单例子
- 常见问题和故障排除

文章的结尾有一个目录，它包含了一些其他有用的与 Zookeeper 有关的信息的链接。

这篇文档的大部分信息写成独立的容易理解的参考资料。然而，在开始你的第一个 Zookeeper 应用之前，你至少要读 Zookeeper 数据模型和 Zookeeper 基本操作上的部分。并且，简单的编程例子对理解 Zookeeper 客户端应用的基础结构非常有帮助。

3.2 Zookeeper 数据模型

Zookeeper 有一个分层的命名空间，非常像一个分布式的文件系统。不同的地方仅仅是命名空间里的每个节点可以有关联数据，也可以有子目录。它就像一个既可以是文件又可以是目录的文件系统。路径节点总是表示为标准的，绝对的，以

斜线为分隔符的路径；没有相对引用。任何 unicode 编码字符都可以用在路径上，但是有以下限制：

- null 字符(\u0000)不能作为路径名字的一部分(这将导致 C 绑定的问题)。
- 下面的字符不能用，因为它们显示不好或者显示混乱：\u0001 - \u0019 and \u007F - \u009F.
- 下面的字符不允许： \ud800 -uF8FFF, \uFFF0-uFFFF, \uXFFFE - \uXFFFF (其中 X 是一个数字：1-E), \uF0000 - \uFFFFF.
- "."字符可以作为名字的一部分，但是"."和".."不能单独使用。因为 Zokeeper 不使用相对路径。下面的路径将是无效的："/a/b/./c" or "/a/b/../c".
- "zookeeper"标记是预留的。

3.2.1 Znodes

Zookeeper 中的每个节点都被称为 znode。Znode 维护了一个 stat 结构，这个 stat 包含数据变化的版本号、访问控制列表变化。stat 结构还有时间戳。版本号和时间戳一起，可让 Zookeeper 验证缓存和协调更新。每次 znode 的数据发生了变化，版本号就增加。例如，无论何时客户端检索数据，它也一起检索数据的版本号。并且当客户端执行更新或删除时，客户端必须提供他正在改变的 znode 的版本号。如果它提供的版本号和真实的数据版本号不一致，更新将会失败。(这种行为可以被覆盖)

在分布式应用工程中，node 可以指的是一般的主机，一个服务器，全体成员的一员，一个客户端程序，等等。在 Zookeeper 的文档中，znode 指的是数据节点。Servers 指的是组成 Zookeeper 服务的机器；quorum peers 指的是组成全体的 servers；client 指的是任何使用 Zookeeper 服务的主机和程序。

Znode 是程序员使用的主要实体。这里需要提到几个有价值的特征：

Watches
客户端可以在 znode 上设置 watches。znode 的变化将会触发 watches 后清除 watches。触发 watches 时，Zookeeper 向客户端发送一个通知。更多关于 watches 的信息可以在 Zookeeper watches 章节查看。

数据访问
命名空间里的每个 znodes 上的数据存储都是原子性的读取和写入。读取时获取所有与 znode 有关的数据字节，写入时替换所有的数据字节。每个节点有一个访问控制列表用来限制谁可以做什么。

Zookeeper 不是设计成通用的数据库或者大数据对象存储。而是管理协调数据。这个数据的形式是配置表单，状态信息，集合点等等。各种形式的协调数据属性都非常小：经过测量在 KB 之内。Zookeeper 客户端和服务端实现检查确保 znode 有不到 1M 的数据，但是实际的数据要远小于平均值。在相对较大的数据上的

操作将会引起一些操作花费更多的时间并且影响一些操作的延迟，因为需要额外的时间在网络和存储设备之间移动数据。如果需要大数据存储，通常的做法是将数据存储进大存储器系统，如 NFS 和 HDFS，然后将存储指针和地址存储进 Zookeeper。

临时节点

Zookeeper 还有一个临时节点的概念。这些 znode 一旦 session 创建就存在，session 结束就被删除。因为这个特性，临时节点不允许有子节点。

序列节点 -- 唯一的命名

当创建 znode 的时候你还可以请求在路径的最后追加一个单调递增的计数器。这个计数器在父节点是唯一的。计数器有一个%010d --的格式，它是 10 位数用 0 做填充(计数器用这个方法格式化简化排序)，也就是：0000000001。查看 Queue Recipe 使用这个特性的例子。注释：计数器的序列号由父节点通过一个 int 类型维护，计数器当超过 2147483647 的时候将会溢出(-2147483647 将会导致)。

3.2.2 Zookeeper 里的计时

Zookeeper 通过多种方式追踪计时：

Zxid

每个 Zookeeper 状态的变化都以 zxid(事务 ID)的形式接收到标记。这个暴露了 Zookeeper 所有变化的总排序。每个变化都会有一个 zxid，并且如果 zxid1 早于 zxid2 则 zxid1 一定小于 zxid2。

版本号

节点的每个变化都会引起那个节点的版本号的其中之一增加。这三个版本号是 version(znode 的数据变化版本号),cversion(子目录的变化版本号)，和 aversion(访问控制列表的变化版本号)。

Ticks

当使用多服务器的 Zookeeper 时，服务器使用 ticks 定义事件的时间，如状态上传，会话超时，同事之间的连接超时等等。tick 次数只是通过最小的会话超时间接的暴露；如果一个客户端请求会话超时小于最小的会话超时，服务器就会告诉客户端会话超时实际上是最低会话超时时间。

Real time

Zookeeper 不使用实时或时钟时间，除了将时间戳加在 znode 创建和更新的 stat 结构上。

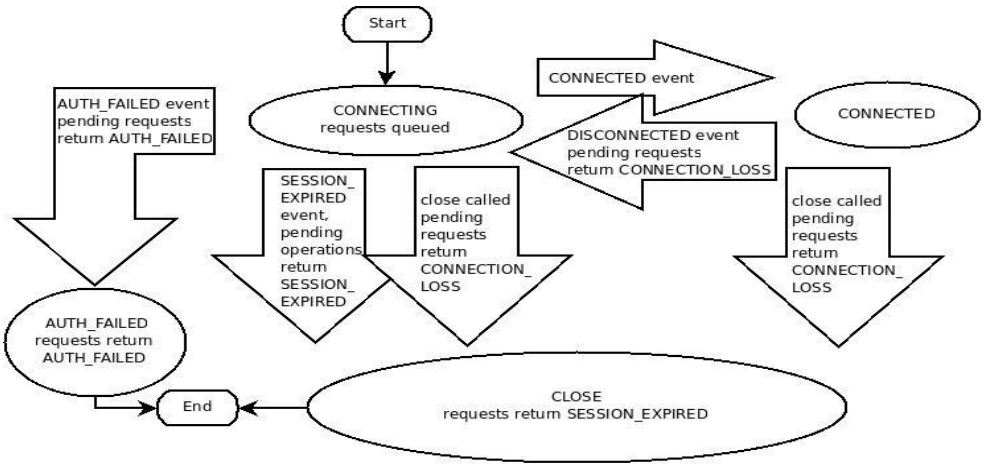
3.2.3 Zookeeper Stat 结构

Zookeeper 中的每个 znode 的 stat 机构都由下面的字段组成：

- czxid - 引起这个 znode 创建的 zxid
- mzxid - znode 最后更新的 zxid
- ctime - znode 被创建的毫秒数(从 1970 年开始)
- mtime - znode 最后修改的毫秒数(从 1970 年开始)
- version - znode 数据变化号
- cversion - znode 子节点变化号
- aversion - znode 访问控制列表的变化号
- ephemeralOwner - 如果是临时节点这个是 znode 拥有者的 session id。如果不是临时节点则是 0。
- dataLength - znode 的数据长度
- numChildren - znode 子节点数量

3.3 Zookeeper Sessions

Zookeeper 客户端通过使用语言绑定在服务上创建一个 handle 建立一个和 Zookeeper 服务的会话。一旦创建了，handle 从 CONNECTION 状态开始并且客户端库尝试连接到 Zookeeper 服务的服务器在这时候切换到 CONNECTED 状态。在正常运行期间会在这两个状态期间。如果发生不可恢复的错误，例如会话超时或授权失败，或应用明确的关闭处理器，handle 将会移动到 CLOSED 状态。下面的图像展示了 Zookeeper 客户端可能的状态变化流程：



创建客户端会话，应用代码必须提供一个以逗号分隔开的 host:port 的列表，每个对应一个 Zookeeper 服务(如:"127.0.0.1:4545"or"127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002")。Zookeeper 客户端会选择任意一个服务并尝试连接它。如果这

个连接失败，或者客户端变为 disconected，客户端会自动的尝试连接列表里的下一个服务器，直到建立连接。

3.2.0 添加：可以在连接字符串后面加入一个"chroot"可选项。这将相对于这个 root 解析执行客户端命令(类似于 unix 的 chroot 命令)。如果使用实例："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"客户端将根在"/app/a"并且所有的路径将相对于这个根 - 例如 getting/setting/等等。"/foo/bar"将会将操作变为"/app/a/foo/bar"。这个特性在每个用户的根目录都不同的多用户环境里特别有用。

客户端得到 Zookeeper 服务 handle 时，Zookeeper 创建一个 Zookeeper session，用 64 位的数字代表，分配到客户端。如果客户端连接到不同的 Zookeeper 服务器，他将发送 session id 作为连接握手的一部分。作为安全措施，服务器为 session id 创建一个任何 Zookeeper 服务器可以验证的密码。当建立会话是连同 session id 一起发送密码到客户端。客户端每当重建会话时都发送这个 session id 和密码。

Zookeeper 客户端调用创建 Zookeeper 会话的参数之一是以毫米为单位的会话超时时间。客户端发送请求超时，服务器响应一个可以给客户端的超时时间。当前的实现要求超时时间的最小值是 2 个 tickTime(服务端配置里设置)最大值是 20 个 tickTime。Zookeeper 客户端 API 允许访问协商的超时时间。

当客户端从 ZK 集群分隔它将开始搜索在 session 创建期间指定的服务器列表。最终，当客户端和服务器之间的连通性是重建时，session 将会转变为"connected"状态或者转为"expired"状态(冲过重建超时)。为断开创建一个 session 对象是不明智的。ZK 客户端类库会为你处理重连。我们启发式的构建客户端类库处理像“羊群效应”的事情等等。只有当通知 session 超时是建立新 session(强制的)。

session 逾期由 Zookeeper 集群自己管理，并不是客户端。当 ZK 客户端建立一个与集群的 session 时它提供一个上面描述的"timeout"值。这个值由集群确定什么时候客户端 session 逾期。当集群在指定的 session 超时周期内没有听到客户端(没有心跳)时发生。session 逾期，集群会删除全部 session 的临时节点并立即通知其他链接的客户端(watch 这些 znode 的客户端)。这时候逾期 session 的客户端仍然是 disconnected 的，它不会被通知 session 逾期知道他能够重新连接到集群。客户端将一直待在 disconnected 状态知道重新建立与集群的 TCP 连接，届时逾期 session 的 watcher 会收到"session expired"的通知。

逾期 session 的状态转化实例通过 watcher 查看：

- 'connected'：建立 session 且正在和集群通信
- ...客户端从集群分离
- 'disconnected'：客户端丢失集群的连接
- ...时间消失，'timeout'周期之后集群逾期 session，客户端什么都看不到
- ...时间消失，客户端收复集群的连通性

- 'expired'：最终客户端重连到集群，然后是逾期通知

建立 Zookeeper session 的另一个参数是默认的 watcher。任何发生在客户端的状态变化都会通知 Watchers。例如如果客户端丢失服务器的连通性客户端将会被通知，或者如果客户端 session 逾期，等等。这个 watcher 应该考虑初始化状态是 disconnected(也就是说要在任何状态事件变化之前发送到 watcher)。对于一个新连接，通常发送到 watcher 的第一个时间是 session 连接事件。

sesion 通过客户端发送请求保持存活。如果 session 空闲了一个超时时间，客户端会发送一个 PING 请求保持 session 存活。这个 PING 请求不仅可让 Zookeeer 服务器知道客户是活动的，而且可让客户端核查它的连接是活动的。PING 的时间是能够确保合理检测死链的时间并重连到新服务。

一旦成功的建立的到服务器的连接基本上有两种情况客户端生产丢失连接，当执行同步或异步操作时并且下面的其中之一：

- 应用在 session 上调用一个它不再存活的操作
- Zookeeper 在执行期间断开和服务器的连接

3.2.0 添加 -- SessionMovedException。有一个客户端通常不可见的 exception 叫做 SessionMovedException。这个异常发生在 session 重新创建在另一个服务器上时。一般引起这个错误的原因是客户端向服务端发送请求，但是网络延迟，以至于客户端超时并连接到新服务器。当延迟的包到达第一个服务器时，老服务器检查到 session 已经被移除，并关闭客户端连接。一般客户端看不到这个错误因为他们不从这些老连接读取数据。当两个客户端尝试使用保存的 sessionid 和 pass word 重新建立相同连接时可以看出来。其中一个客户端将会建立连接且第二个客户端会断开连接。

3. 4 ZooKeeper Watches

Zookeeper 里的所有读取操作 - getData(),getChildren()和 exists() - 都有设置 watch 的选项。这是 Zookeeper watch 的定义：watch 事件是 one-time 触发，向客户端发送设置 watch，当设置 watch 的数据变化时发生。在 watch 定义里有三个关键点：

- **一次触发** - 当数据有了变化时将向客户端发送一个 watch 事件。例如，如果一个客户端用 getData("/znode1",true)并且过一会之后/znode1 的数据改变或删除了，客户端将获得一个/znode1 的 watch 事件。如果/znode1 再次改变，将不会发送 watch 事件除非设置了新 watch。
- **发往客户端** - 这意味着事件发往客户端，但是可能在成功之前没到客户端。Watches 是异步发往 watchers。Zookeeper 提供一个顺序保证：在看到 watch 事件之前绝不会看到

变化。网络延迟或其他因素可能引起客户端看到 watches 并在不同时间返回 code。关键点是不同客户端看到的是一致性的顺序。

- **为数据设置 watch** - 一个节点可以有不同方式改变。它帮助 Zookeeper 维护两个 watches：data watches 和 child watches。getData()和 exists()设置 data watches。getChildren()设置 child watches。两者任选其一，它可以帮助 watches 根据类型返回。getData()和 exists()返回关于节点数据的信息，然而 getChildren()返回 children 列表。因此，setData()将会触发 znode 设置的 data watches。一个成功的 create()将会触发一个 datawatches 和一个父节点的 child watch。一个成功的 delete()将触发一个 data watch 和一个 child watch。

Watches 是在 client 连接到 Zookeeper 服务端的本地维护。这可让 watches 成为轻量的，可维护的和派发的。当一个 client 连接到新 server，watch 将会触发任何 session 事件。断开连接后不能接收到。当客户端重连，先前注册的 watches 将会被重新注册并触发如果需要。

3.4.1 Zookeeper 保证 watches 的什么

关于 watches，Zookeeper 维护这些保证：

- Watches 和其他事件、watches 和异步恢复都是有序的。Zookeeper 客户端保证每件事都是有序派发。
- 客户端在看到新数据之前先看到 watch 事件。
- 对应更新顺序的 Zookeeper watches 事件顺序由 Zookeeper 服务所见。

3.4.2 关于 Watches 要记住的事情

- Watches 是一次触发；如果你得到一个 watch 事件且想在将来的变化得到通知，必须设置另一个 watch。
- 因为 watches 是一次触发且在获得事件和发送请求得到 wathes 之间有延迟你不能可靠的看到发生在 Zookeeper 节点的每一个变化。准备好处理这个案例在获得事件和再次设置 watch 之间变化多次。(你可能不在意，但是至少认识到它可能发生)。
- 一个 watch 对象，或 function/context 对，对于指定的通知只能触发一次。例如，如果相同的文件通过 exists 和 getData 注册了相同的 watch 对象并且文件稍后删除了，watch 将只会触发文件的删除通知。
- 从服务端断开连接时(比如服务器故障)，将不会得到任何 watches 直到重新建立连接。因为这个原因 session 事件被发送到所有 watch 处理器。使用 session 事件进入安全模式：断开连接时不接收事件，所以在这个模式里你的程序应该采取保守。

3.5 Zookeeper 使用 ACLs 控制访问

Zookeeper 使用 ACLs 控制访问它的 znodes(Zookeeper 的数据节点)。ACL 实现非常类似于 UNIX 文件访问权限：它使用权限位允许/不允许一个简单和范围的各种操作。不像标准的 UNIX 权限，Zookeeper 节点不由三个标准范围(用户，组 和 world)限制。Zookeeper 没有 znode 所有者的概念。而是一个 ACLs 指定一组 ids 和与这些 ids 相关联的权限。

还要注意 ACL 只适用于特定的 znode。尤其不适用于 children。例如，如果/app 对 ip:192.168.1.56 是只读的并且/app/status 是全都可读的，任何人可以读取/app/status；ACLs 不是递归控制的。

Zookeeper 支持可插拔的权限认证方案。使用 scheme:id 的形式指定 Ids,scheme 是 id 对应的权限认证 scheme。例如，ip:172.16.16.1 是一个地址为 172.16.16.1 的 id.

客户端连接的 Zookeeper 并授权自己时，Zookeeper 联合所有对应客户端的 ids。这些 ids 在尝试访问节点时核查 znodes 的 ACLs。ACLs 由对组成(scheme:expression,permissions)。expression 的格式是针对 scheme。例如，(ip:19.22.0.0/16, READ)对开头 19.22 的 IP 地址的任意客户端提供读取权限。

3.5.1 ACL 权限

Zookeeper 支持下面的 permissions：

- CREATE：可以创建子节点
- READ：可以从节点获取数据并列出它的子节点
- WRITE：可以向节点设置数据
- DELETE：可以删除一个子节点
- ADMIN：可以设置权限
- CREATE 和 DELELTE 权限已经更细粒度的划分了 WRITE 权限

CREATE 和 DELETE 的案例是：
你想要 A 在 Zookeeper 节点上能够 set，但是不能 CREATE 或 DELETE 子节点。

CREATE 而没有 DELETE：客户端通过在父节点创建的 Zookeeper 节点创建请求。你想要所有客户端能 add，而只有 request processor 可以 delete。（这有点像文件的 APPEND 权限）

因为 Zookeeper 没有文件所有者的权限才有 ADMIN 权限。在某种意义上 ADMIN 权限指定实体作为拥有者。Zookeeper 不支持 LOOKUP 权限。每人都有 LOOKUP 权限。这可让你 stat 一个节点，但不能做其他的。

内嵌的 ACL schemes

Zookeeper 有下面的 schemes：

- world：有单独的 id，anyone,代表任何人。
- auth:不适用任何 id，代表任何授权的用户。
- digest：使用 username;password 字符串生成 MD5 哈希作为 ACL ID 身份。通过发送 username:password 明文授权。在 ACL 里使用时 expression 将会是 username:base64 编码的 SHA1 password 摘要。
- ip:使用客户端 IP 作为 ACL ID 身份。

3.6 可插拔的 Zookeeper 身份认证

Zookeeper 运行在各种不同身份认证 shcemes 多变的不同环境，所以它有一个完整的可插拔的身份认证框架。甚至内嵌的权限认证 schemes 使用可插拔的身份认证框架。

理解身份认证框架怎么工作，首先你必须理解两个主要的身份认证操作。框架首先必须授权客户端。通常客户端一连接到服务端就做了并且由验证信息组成。第二个操作通过框架处理找到对应客户端的 ACL 入口。ACL 入口是 对。idspec 可以是简单的字符串或者表达式。这是权限插件匹配的实现。这是一个授权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 返回标示插件的字符串。因为我们支持多个权限认证方法，一个身份认证证书或 idspec 一直有 scheme:前缀。Zookeeper 服务器使用认证插件返回的 scheme 确定 sheme 适用的 ids。

3.7 一致性保证

Zookeeper 是一个高性能可扩展的服务。读取和写入操作都非常快，但读取比写入块。这个读取案例的原因是 Zookeeper 可以提供老数据，Zookeeper 的一致性保证：

3.7.1 顺序一致性

客户端的更新将发送到序列

3.7.2 原子性

更新成功或失败 -- 没有局部结果

3.7.3 单一系统影像

客户端看到的服务端的视图都一样

3.7.4 可靠性

一旦更新应用了，它将会一直保持到下次更新覆盖。这个保证有两个推论：

- 如果客户端成功，更新就被应用。客户端看不到失败现象
- 客户端可以看到任何更新，通过读取请求或成功的更新

3.7.5 时效性

系统的客户端视图在特定的时间保证是最新的。

使用这些一致性保证很容易构建更高级的功能如领导选举，阻塞、队列和单独可撤销的锁定在客户端。

3.8 绑定

Zookeeper 客户端类库有两种语言：Java 和 C。下面的章节介绍他们。

3.8.1 Java 绑定

有两个组成 Zookeeper Java 绑定的包：org.apache.zookeeper 和 org.apache.zookeeper.data。其余组成 zookeeper 的包在内部使用或是服务实现的一部分。org.apache.zookeeper.data 包是使用简单容器生成的类。

Zookeeper java 客户端使用的主类是 zookeeper 类。它两个构造函数的不同仅仅是通过可选的 session id 和 password。Zookeeper 支持跨进程的 session 恢复。Ja

va 程序可以保持它的 session id 和密码到稳定的存储、重启和恢复更早程序实例使用的 session。

创建 Zookeeper 的时候，同时创建了两个线程：一个 IO 线程和事件线程。所有的 IO 都在在 IO 线程上(使用 Java NIO)。所有的事件回调都在事件线程上。Session 在 IO 线程上维护如重连 Zookeeper 服务和维护心跳。同步方法调用也在 IO 线程处理。所有异步调用和 watche 事件在事件线程上处理。这个设计有几个点需要注意：

- 所有异步调用的完成和 watcher 回调将被放置在序列中，一次一个。调用者可以做任何他们想做的处理，但是在处理期间不能做其他回调。
- 回调不锁定 IO 线程的处理或同步调用的处理。
- 同步调用可能是不正确的顺序。例如，假定一个客户端做以下的处理：对节点/a 设置 watch 为 true 发型一个异步读取，然后在读取的完成回调里执行一个 a 节点的同步读取。

最后，关闭规则非常简单：一旦一个 zookeeper 对象关闭或者接受一个致命的时间(SESSION_EXPIRED and AUTH_FAILED)，Zookeeper 对象变为无效的。在 close 上，两个线程停止并且 zookeeper handle 上的任何进一步的访问都是未定义的行为并拒绝。

3.9 常见问题和解决方案

现在你知道 Zookeeper 非常快,简单，等等。。。下面是 zookeeper 容易陷入的陷阱：

- 如果你正在使用 watches，你必须寻找连接的 watch 事件。Zookeeper 客户端从服务端断开连接时，直到重连后才能接收变化通知。如果你正在观察的 znode 存在，断开连接时如果 znode 创建或删除将丢失事件。
- 必须测试 Zookkeper 服务故障。Zookeeper 服务只要大多数的服务器正常就可以运行。问题是：你的应用可以处理它吗？现实中连接是可以断的。Zookeeper 客户端库负责恢复连接并让你知道发生了什么，但是你必须确定恢复你的状态和失败的未完成请求。在测试的时候找到他，而不是在生产力 - 测试 Zookeeper 服务器并重启他们。
- 客户端使用的 Zookeeper 服务列表必须匹配每个 Zookeeper 服务器的 Zookeeper 服务列表。
- 当心存储事务日志的地方。Zookeeper 的性能关键是事务日志。Zookeeper 在返回响应之前必须把事务同步到中介。专用的事务日志装置是保证良好性能的关键。将日志存储进繁忙的装置会影响性能。如果你只有一个存储装置，将追踪文件放置到 NFS 并增加快照数量；这不能消除问题但能缓解它。
- 正确的设置你的 Java 最多堆内存。这非常重要避免 swapping。不必要的进入磁盘肯定会降低性能。记住，在 Zookeeper，每件事都是有序的，所以如果一个请求点击磁盘，所有其他队列的请求也点击磁盘。避免 swapping，尝试设置堆大小为物理内存减去操作系统和缓存需要的内存。确定配置最佳堆内存的配置是运行负载测试。如果不能做，就估算一个值。例如，在 4G 的机器上，保守估计堆内存是 3G。

第 4 章 Zookeeper Java 实例

4.1 简单的 Watch 客户端

为了介绍 Zookeeper JAVA API，我们开发了一个简单的 watch 客户端。这个客户端监测通过启动和停止程序 Zookeeper 节点的变化。

4.1.1 要求

客户端有四个要求：

- Zookeeper 服务的地址
- 监控节点 znode 的名字
- 一个可执行文件的参数
- 获取与 znode 相关联的数据并启动可执行文件

znode 发生改变的话，会出现下面两种情况：

- 如果 znode 改变，客户端重新获取内容并重新启动可执行文件。
- 如果 znode 消失，客户端杀掉可执行文件。

4.1.2 程序设计

按照惯例，Zookeeper 应用分为两个单元，一个维持连接，另一个监控数据。在这个应用里，Executor 类维持 Zookeeper 连接，DataMonitor 类监控 Zookeeper 树形节点的数据。另外，Executor 包含主线程和执行逻辑。它负责什么用户交互存在，以及和你通过参数传入的可执行程序交互和哪个实例停止和重启，根据 znode 的状态。

4.2 Executor 类

Executor 对象是实例应用的主容器。它包含 Zookeeper 对象,DataMonitor，根据上面程序设计的描述。

```
1.          // from the Executor class...
2.
3.          public static void main(String[] args) {
4.              if (args.length < 4) {
```

```
5.                  System.err
6.                          .println("USAGE: Executor hostPort znode filename program
    [args ...]");
7.                  System.exit(2);
8.              }
9.              String hostPort = args[0];
10.             String znode = args[1];
11.             String filename = args[2];
12.             String exec[] = new String[args.length - 3];
13.             System.arraycopy(args, 3, exec, 0, exec.length);
14.             try {
15.                 new Executor(hostPort, znode, filename, exec).run();
16.             } catch (Exception e) {
17.                 e.printStackTrace();
18.             }
19.         }
20.
21.         public Executor(String hostPort, String znode, String filename,
22.                 String exec[]) throws KeeperException, IOException {
23.             this.filename = filename;
24.             this.exec = exec;
25.             zk = new ZooKeeper(hostPort, 3000, this);
26.             dm = new DataMonitor(zk, znode, null, this);
27.         }
28.
29.         public void run() {
30.             try {
31.                 synchronized (this) {
32.                     while (!dm.dead) {
33.                         wait();
34.                     }
35.                 }
36.             } catch (InterruptedException e) {
37.             }
38.         }
```

回想一下 Executor 的工作是启动和停止你通过命令行传入的可执行文件。它这样做通过 Zookeeper 对象应对激发的事件。就像上面你看到的代码，在 Zookeeper 构造里 Executor 将自身作为 Watcher 参数传入 Zookeeper 构造。他还将自身作为 DataMonitorListener 参数传入 DataMonitor 构造。每个 Executor 的定义，都实现下面的接口：

```
1.      public class Executor implements Watcher, Runnable, DataMonitor.DataMonitorLi
    stener { ...
```

Watcher 接口由 Zookeeper Java API 定义。Zookeeper 使用它交流回它的主容器。它只支持一个方法，process()，并且 Zookeeper 使用它交流主线程感兴趣的通用事件，比如 Zookeeper 连接的状态或者 Zookeeper session。这个例子里 Executor 简单的将事件指向到 DataMonitor 决定如何处理他们。简单的说明几点，按照惯例，Executor 或者一些类 Executor 对象“拥有”Zookeeper 连接，自由的委托事件到其他事件到其他对象。它还使用这个作为激发 watch 事件默认的管道（稍后详述）。

```
1.          public void process(WatchedEvent event) {
2.              dm.process(event);
3.          }
```

DataMonitorListener 接口，另一方面，不是 Zookeeper API 的一部分。它是这个实例应用的自定义接口。DataMonitor 对象使用它交流回它的容器，也是 Executor 对象。DataMonitorListener 接口看起来是这样的：

```
1.      public interface DataMonitorListener {
2.          /**
3.          * The existence status of the node has changed.
4.          */
5.          void exists(byte data[]);
6.
7.          /**
8.          * The ZooKeeper session is no longer valid.
9.          *
10.         * @param rc
11.         * the ZooKeeper reason code
12.         */
13.         void closing(int rc);
14.     }
```

这个接口在 DataMonitor 类里定义并在 Executor 类里实现。当 Executor.exists() 被调用时，Executor 决定是否启动或者关闭每个需求。回想要求说明当 znode 不再存在时杀掉可执行文件。

当 Executor.closing()被调用时，Executor 决定是否关闭它自己应对 Zookeeper 连接永久的消失。

你可能已经猜到，DataMonitor 是调用这些方法的对象,应对 Zookeeper 状态的变化。

这里是 DataMonitorListener.exists()和 DataMonitorListener.closing 的 Executor 的实现：

```
1.      public void exists( byte[] data ) {
2.          if (data == null) {
3.              if (child != null) {
4.                  System.out.println("Killing process");
5.                  child.destroy();
6.                  try {
7.                      child.waitFor();
8.                  } catch (InterruptedException e) {
9.                  }
10.             }
11.             child = null;
12.         } else {
13.             if (child != null) {
14.                 System.out.println("Stopping child");
15.                 child.destroy();
16.                 try {
17.                    child.waitFor();
18.                 } catch (InterruptedException e) {
19.                 e.printStackTrace();
20.                 }
21.             }
22.             try {
23.                 FileOutputStream fos = new FileOutputStream(filename);
24.                 fos.write(data);
25.                 fos.close();
26.             } catch (IOException e) {
27.                 e.printStackTrace();
28.             }
29.             try {
30.                 System.out.println("Starting child");
31.                 child = Runtime.getRuntime().exec(exec);
32.                 new StreamWriter(child.getInputStream(), System.out);
33.                 new StreamWriter(child.getErrorStream(), System.err);
34.             } catch (IOException e) {
35.                 e.printStackTrace();
36.             }
```

```
37.         }
38.     }
39.
40.     public void closing(int rc) {
41.         synchronized (this) {
42.             notifyAll();
43.         }
44.     }
```

4.3 DataMonitor 类

DataMonitor 类有 Zookeeper 逻辑的具体内容。它主要是异步和事件驱动。DataMonitor 在构造里就开始做事了：

```
1.      public DataMonitor(ZooKeeper zk, String znode, Watcher chainedWatcher,
2.              DataMonitorListener listener) {
3.          this.zk = zk;
4.          this.znode = znode;
5.          this.chainedWatcher = chainedWatcher;
6.          this.listener = listener;
7.
8.          // Get things started by checking if the node exists. We are going
9.          // to be completely event driven
10.         zk.exists(znode, true, this, null);
11.     }
```

调用 Zookeeper.exists 检查 znode 是否存在，设置一个 watch,并将自己的引用传入作为完成回调对象。从这个意义上，它开始做事了，因为当 watch 被触发时发生真正的处理。

注意：

不要迷惑 watch 回调的完成回调。Zookeeper.exists()完成回调，它恰好是 DataMonitor 对象里的 StatCallback.processResult()的实现，当在服务上异步设置 watch 操作完成时调用。

watch 的触发，换句话说，发送事件到 Executor 对象，因为 Executor 注册作为 Zookeeper 对象的 Watcher。

顺便插一句，你要注意 DataMonitor 还可以注册它自身作为这个特定 watch 事件的 Watcher。这是 Zookeeper3.0.3 的新特性。在这个例子里，而 DataMonitor 没有注册作为 Watcher。

Zookeeper.exists()操作在服务上完成时，Zookeeper API 在客户端调用这个完成回调：

```
1.      public void processResult(int rc, String path, Object ctx, Stat stat) {
2.          boolean exists;
3.          switch (rc) {
4.          case Code.Ok:
5.              exists = true;
6.              break;
7.          case Code.NoNode:
8.              exists = false;
9.              break;
10.         case Code.SessionExpired:
11.         case Code.NoAuth:
12.             dead = true;
13.             listener.closing(rc);
14.             return;
15.         default:
16.             // Retry errors
17.             zk.exists(znode, true, this, null);
18.             return;
19.         }
20.
21.         byte b[] = null;
22.         if (exists) {
23.             try {
24.                 b = zk.getData(znode, false, null);
25.             } catch (KeeperException e) {
26.                 // We don't need to worry about recovering now. The watch
27.                 // callbacks will kick off any exception handling
28.                 e.printStackTrace();
29.             } catch (InterruptedException e) {
30.                 return;
31.             }
32.         }
33.         if ((b == null && b != prevData)
34.                 || (b != null && !Arrays.equals(prevData, b))) {
35.             listener.exists(b);
36.             prevData = b;
```

```
37.         }
38.     }
```

代码首先检查 znode 存在性的错误代码,致命错误，和可恢复的错误。如果文件存在，它从 znode 获取数据，然后如果状态发生改变就调用 Executor 的 exists()回调。注意，没有必要做每个异常处理因为它已经检查等待任何引起错误的事情：如果在调用 Zookeeper.getData()之前删除了节点，通过 Zookeeper.exists()设置的 watch 事件触发一个回调；如果有通讯错误，当连接恢复的时候激发连接 watch 事件。

最后，注意 DataMonitor 怎么处理 watch 事件：

```
1.          public void process(WatchedEvent event) {
2.              String path = event.getPath();
3.              if (event.getType() == Event.EventType.None) {
4.                  // We are are being told that the state of the
5.                  // connection has changed
6.                  switch (event.getState()) {
7.                  case SyncConnected:
8.                      // In this particular example we don't need to do anything
9.                      // here - watches are automatically re-registered with
10.                     // server and any watches triggered while the client was
11.                     // disconnected will be delivered (in order of course)
12.                     break;
13.                 case Expired:
14.                     // It's all over
15.                     dead = true;
16.                     listener.closing(KeeperException.Code.SessionExpired);
17.                     break;
18.                 }
19.             } else {
20.                 if (path != null && path.equals(znode)) {
21.                     // Something has changed on the node, let's find out
22.                     zk.exists(znode, true, this, null);
23.                 }
24.             }
25.             if (chainedWatcher != null) {
26.                 chainedWatcher.process(event);
27.             }
28.         }
```

如果客户端 Zookeeper 类库可以在 session 超时之前重建通讯通道，所有的 session 的 watches 将自动的重连服务。在编程指南里查看 Zookeeper Watches 了解更多。在这个函数里有一点低，当 DataMonitor 获取 znode 的事件时，它调用 Zookeeper.exists()去找出发生了什么变化。

4.4 完整代码

4.4.1 Executor.java

```
1.      /**
2.       * A simple example program to use DataMonitor to start and
3.       * stop executables based on a znode. The program watches the
4.       * specified znode and saves the data that corresponds to the
5.       * znode in the filesystem. It also starts the specified program
6.       * with the specified arguments when the znode exists and kills
7.       * the program if the znode goes away.
8.       */
9.      import java.io.FileOutputStream;
10.     import java.io.IOException;
11.     import java.io.InputStream;
12.     import java.io.OutputStream;
13.
14.     import org.apache.zookeeper.KeeperException;
15.     import org.apache.zookeeper.WatchedEvent;
16.     import org.apache.zookeeper.Watcher;
17.     import org.apache.zookeeper.ZooKeeper;
18.
19.     public class Executor
20.         implements Watcher, Runnable, DataMonitor.DataMonitorListener
21.     {
22.         String znode;
23.
24.         DataMonitor dm;
25.
26.         ZooKeeper zk;
27.
28.         String filename;
29.
30.         String exec[];
```

```
31.
32.         Process child;
33.
34.         public Executor(String hostPort, String znode, String filename,
35.                 String exec[]) throws KeeperException, IOException {
36.             this.filename = filename;
37.             this.exec = exec;
38.             zk = new ZooKeeper(hostPort, 3000, this);
39.             dm = new DataMonitor(zk, znode, null, this);
40.         }
41.
42.         /**
43.          * @param args
44.          */
45.         public static void main(String[] args) {
46.             if (args.length < 4) {
47.                 System.err
48.                         .println("USAGE: Executor hostPort znode filename program
    [args ...]");
49.                 System.exit(2);
50.             }
51.             String hostPort = args[0];
52.             String znode = args[1];
53.             String filename = args[2];
54.             String exec[] = new String[args.length - 3];
55.             System.arraycopy(args, 3, exec, 0, exec.length);
56.             try {
57.                 new Executor(hostPort, znode, filename, exec).run();
58.             } catch (Exception e) {
59.                 e.printStackTrace();
60.             }
61.         }
62.
63.         /**********************************************************************
    ***
64.          * We do process any events ourselves, we just need to forward them on.
65.          *
66.          * @see org.apache.zookeeper.Watcher#process(org.apache.zookeeper.proto.W
    atcherEvent)
```

```
67.          */
68.         public void process(WatchedEvent event) {
69.             dm.process(event);
70.         }
71.
72.         public void run() {
73.             try {
74.                 synchronized (this) {
75.                     while (!dm.dead) {
76.                         wait();
77.                     }
78.                 }
79.             } catch (InterruptedException e) {
80.             }
81.         }
82.
83.         public void closing(int rc) {
84.             synchronized (this) {
85.                 notifyAll();
86.             }
87.         }
88.
89.         static class StreamWriter extends Thread {
90.             OutputStream os;
91.
92.             InputStream is;
93.
94.             StreamWriter(InputStream is, OutputStream os) {
95.                 this.is = is;
96.                 this.os = os;
97.                 start();
98.             }
99.
100.            public void run() {
101.                byte b[] = new byte[80];
102.                int rc;
103.                try {
104.                    while ((rc = is.read(b)) > 0) {
105.                        os.write(b, 0, rc);
```

```
106.                     }
107.                 } catch (IOException e) {
108.                 }
109.
110.             }
111.         }
112.
113.         public void exists(byte[] data) {
114.             if (data == null) {
115.                 if (child != null) {
116.                     System.out.println("Killing process");
117.                     child.destroy();
118.                     try {
119.                         child.waitFor();
120.                     } catch (InterruptedException e) {
121.                     }
122.                 }
123.                 child = null;
124.             } else {
125.                 if (child != null) {
126.                     System.out.println("Stopping child");
127.                     child.destroy();
128.                     try {
129.                         child.waitFor();
130.                     } catch (InterruptedException e) {
131.                         e.printStackTrace();
132.                     }
133.                 }
134.                 try {
135.                     FileOutputStream fos = new FileOutputStream(filename);
136.                     fos.write(data);
137.                     fos.close();
138.                 } catch (IOException e) {
139.                     e.printStackTrace();
140.                 }
141.                 try {
142.                     System.out.println("Starting child");
143.                     child = Runtime.getRuntime().exec(exec);
144.                     new StreamWriter(child.getInputStream(), System.out);
```

```
                    new StreamWriter(child.getErrorStream(), System.err);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

4.4.2 DataMonitor.java

```
/**
 * A simple class that monitors the data and existence of a ZooKeeper
 * node. It uses asynchronous ZooKeeper APIs.
 */
import java.util.Arrays;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.AsyncCallback.StatCallback;
import org.apache.zookeeper.KeeperException.Code;
import org.apache.zookeeper.data.Stat;

public class DataMonitor implements Watcher, StatCallback {

    ZooKeeper zk;

    String znode;

    Watcher chainedWatcher;

    boolean dead;

    DataMonitorListener listener;

    byte prevData[];

    public DataMonitor(ZooKeeper zk, String znode, Watcher chainedWatcher,
```

```
30.                 DataMonitorListener listener) {
31.             this.zk = zk;
32.             this.znode = znode;
33.             this.chainedWatcher = chainedWatcher;
34.             this.listener = listener;
35.             // Get things started by checking if the node exists. We are going
36.             // to be completely event driven
37.             zk.exists(znode, true, this, null);
38.         }
39.
40.         /**
41.          * Other classes use the DataMonitor by implementing this method
42.          */
43.         public interface DataMonitorListener {
44.             /**
45.              * The existence status of the node has changed.
46.              */
47.             void exists(byte data[]);
48.
49.             /**
50.              * The ZooKeeper session is no longer valid.
51.              *
52.              * @param rc
53.              *                the ZooKeeper reason code
54.              */
55.             void closing(int rc);
56.         }
57.
58.         public void process(WatchedEvent event) {
59.             String path = event.getPath();
60.             if (event.getType() == Event.EventType.None) {
61.                 // We are are being told that the state of the
62.                 // connection has changed
63.                 switch (event.getState()) {
64.                 case SyncConnected:
65.                     // In this particular example we don't need to do anything
66.                     // here - watches are automatically re-registered with
67.                     // server and any watches triggered while the client was
68.                     // disconnected will be delivered (in order of course)
```

```
69.                     break;
70.                 case Expired:
71.                     // It's all over
72.                     dead = true;
73.                     listener.closing(KeeperException.Code.SessionExpired);
74.                     break;
75.                 }
76.             } else {
77.                 if (path != null && path.equals(znode)) {
78.                     // Something has changed on the node, let's find out
79.                     zk.exists(znode, true, this, null);
80.                 }
81.             }
82.             if (chainedWatcher != null) {
83.                 chainedWatcher.process(event);
84.             }
85.         }
86.
87.         public void processResult(int rc, String path, Object ctx, Stat stat) {
88.             boolean exists;
89.             switch (rc) {
90.             case Code.Ok:
91.                 exists = true;
92.                 break;
93.             case Code.NoNode:
94.                 exists = false;
95.                 break;
96.             case Code.SessionExpired:
97.             case Code.NoAuth:
98.                 dead = true;
99.                 listener.closing(rc);
100.                return;
101.            default:
102.                // Retry errors
103.                zk.exists(znode, true, this, null);
104.                return;
105.            }
106.
107.            byte b[] = null;
```

```
108.            if (exists) {
109.                try {
110.                    b = zk.getData(znode, false, null);
111.                } catch (KeeperException e) {
112.                    // We don't need to worry about recovering now. The watch
113.                    // callbacks will kick off any exception handling
114.                    e.printStackTrace();
115.                } catch (InterruptedException e) {
116.                    return;
117.                }
118.            }
119.            if ((b == null && b != prevData)
120.                    || (b != null && !Arrays.equals(prevData, b))) {
121.                listener.exists(b);
122.                prevData = b;
123.            }
124.        }
125.    }
```

第 5 章 Zookeeper 阻塞和队列

5.1 简介

在这个教程里，我们展示了使用 Zookeeper 阻塞和生产者-消费者队列的简单实现。我们所说的类分别叫 Barrier 和 Queue。这些实例假定你至少有一台 Zookeeper 服务正在运行。

每个部分都使用下面的通用代码片段：

```
1.          static ZooKeeper zk = null;
2.          static Integer mutex;
3.
4.          String root;
5.
6.          SyncPrimitive(String address) {
7.              if(zk == null){
8.                  try {
9.                      System.out.println("Starting ZK:");
10.                     zk = new ZooKeeper(address, 3000, this);
11.                     mutex = new Integer(-1);
12.                     System.out.println("Finished starting ZK: " + zk);
13.                 } catch (IOException e) {
14.                     System.out.println(e.toString());
15.                     zk = null;
16.                 }
17.             }
18.         }
19.
20.         synchronized public void process(WatchedEvent event) {
21.             synchronized (mutex) {
22.                 mutex.notify();
23.             }
24.         }
```

所有类都继承 SyncPrimitive。通过这个方式，我们执行步骤都在 SyncPrimitive 的构造里。为了保持实例简单，我们创建一个 Zookeeper 对象第一次我们实例化一个 Barrier 对象或一个 Queue 对象，并且我们静态一个静态变量引用到这个对象。

阻塞和队列的序列实例检查 Zookeeper 对象是否存在。或者，我们可以用程序创建一个 Zookeeper 对象并传到 Barrier 和 Queue 的构造里。

我们使用 process()方法处理 watches 触发的通知。在下面的讨论中，我们展示设置 watches 的代码。watch 是内部结构使 Zookeeper 能够通知节点变化到客户端。例如，如果一个客户端正在等待其他客户端离开一个 barrier，然后它可以设置一个 watch 并等待修改特定的节点，它可以表明这是等待的结束。只要仔细看我们的例子这点就会很清晰。

5.2 阻塞

barrier 是一个元件，它能够启用同步计算开始和结束的一组流程。这个实现的一般方法是有一个 barrier 节点为成为各个处理节点的父节点的目的服务。假如我们叫 barrier 节点"/b1"。然后每个进程"p"创建一个节点"/b1/p"。一旦足够的进程创建了他们对应的节点，加入的进程就可以开始计算。

在这个例子里，每个进程实例化一个 Barrier 对象，并且它的构造带有几个参数：

- Zookeeper 服务的地址(如"zoo1.foo.com:2181")
- Zookeeper 上 barrier 节点的路径(如"/b1")
- 进程组的大小

Barrier 的构造将 Zookeeper 服务地址传入的父类的构造函数。如果 zk 实例不存在父类就创建一个 Zookeeper 实例。然后 Barrier 构造函数在 Zookeeper 上创建一个 barrier 节点,它是所有进程节点的父节点，我们把它叫做 root(注意这里不是 Zookeeper root"/")。

```
1.              /**
2.               * Barrier constructor
3.               *
4.               * @param address
5.               * @param root
6.               * @param size
7.               */
8.              Barrier(String address, String root, int size) {
9.                  super(address);
10.                 this.root = root;
11.                 this.size = size;
12.
13.                 // Create barrier node
```

```
14.                 if (zk != null) {
15.                     try {
16.                         Stat s = zk.exists(root, false);
17.                         if (s == null) {
18.                             zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
19.                                     CreateMode.PERSISTENT);
20.                         }
21.                     } catch (KeeperException e) {
22.                         System.out
23.                                 .println("Keeper exception when instantiating queue:
    "
24.                                         + e.toString());
25.                     } catch (InterruptedException e) {
26.                         System.out.println("Interrupted exception");
27.                     }
28.                 }
29.
30.                 // My node name
31.                 try {
32.                     name = new String(InetAddress.getLocalHost().getCanonicalHostN
   ame().toString());
33.                 } catch (UnknownHostException e) {
34.                     System.out.println(e.toString());
35.                 }
36.
37.             }
```

进入 barrier，一个进程调用 enter()。进程在 root 下创建一个表示它自己的节点，节点名字使用它的主机名的形式。然后它等待直到足够的进程进入到 barrier。进程通过使用"getChildren()"检查 root 节点的子节点数量，并且在这个例子里不到足够的数量时就一直等待通知。当 root 节点有变化是接收通知，进程必须设置一个 watch，并且通过调用"getChildren()"实现他。在这个代码里，我们的"getChildren()"有两个参数。第一个声明节点从哪里读，第二个是一个 boolean 值能够让进程设置一个 watch。这个代码里变量值是 true。

```
1.              /**
2.               * Join barrier
3.               *
4.               * @return
5.               * @throws KeeperException
6.               * @throws InterruptedException
```

```
7.              */
8.
9.             boolean enter() throws KeeperException, InterruptedException{
10.                zk.create(root + "/" + name, new byte[0], Ids.OPEN_ACL_UNSAFE,
11.                        CreateMode.EPHEMERAL_SEQUENTIAL);
12.                while (true) {
13.                    synchronized (mutex) {
14.                        List<String> list = zk.getChildren(root, true);
15.
16.                        if (list.size() < size) {
17.                            mutex.wait();
18.                        } else {
19.                            return true;
20.                        }
21.                    }
22.                }
23.            }
```

注意 enter()同时抛出了 KeeperException 和 InterruptedException，所以需要应用 catch 并处理这些异常。

一旦计算结束，进程调用 leave()离开 barrier。首先它删除它对应的节点，然后它获取 root 节点的子节点。如果至少有一个子节点，然后它等待通知(注意调用 getChidren()的第二个参数是 true，表示 Zookeeper 必须在 root 节点设置一个 watch)。接到通知时，它再次检查 root 节点是否有子节点。

```
1.             /**
2.              * Wait until all reach barrier
3.              *
4.              * @return
5.              * @throws KeeperException
6.              * @throws InterruptedException
7.              */
8.
9.             boolean leave() throws KeeperException, InterruptedException{
10.                zk.delete(root + "/" + name, 0);
11.                while (true) {
12.                    synchronized (mutex) {
13.                        List<String> list = zk.getChildren(root, true);
14.                            if (list.size() > 0) {
15.                                mutex.wait();
```

```
                        } else {
                            return true;
                        }
                    }
                }
            }
        }
```

5.3 生产者-消费者队列

生产者-消费者队列是一连串的生产和消费的分布式的数据结构重组，生产者创建新元素并添加到队列。消费者进程从集合中删除元素，并处理他们。在这个实现里，元素是简单的 integer。队列由一个根节点代表，并添加元素到队列，一个生产者进程创建一个新节点，根节点的子节点。

下面的代码片段是对象的构造。与 Barrier 对象一样，它首先调用父类的构造，SyncPrimitive，如果 Zookeeper 对象不存在就创建一个。然后它验证队列的根节点是否存在，并且如果不存在就创建。

```
1.              /**
2.               * Constructor of producer-consumer queue
3.               *
4.               * @param address
5.               * @param name
6.               */
7.              Queue(String address, String name) {
8.                  super(address);
9.                  this.root = name;
10.                 // Create ZK node name
11.                 if (zk != null) {
12.                     try {
13.                         Stat s = zk.exists(root, false);
14.                         if (s == null) {
15.                             zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
16.                                     CreateMode.PERSISTENT);
17.                         }
18.                     } catch (KeeperException e) {
19.                         System.out
20.                                 .println("Keeper exception when instantiating queue:
    "
```

```
                                    + e.toString());
                } catch (InterruptedException e) {
                    System.out.println("Interrupted exception");
                }
            }
        }
```

生产者进程调用"produce"添加一个元素到队列，并传入一个 integer 作为参数。添加元素到队列，使用"create()"方法创建一个新节点，并使用 SEQUENCE 标记指示 Zookeeper 追加根节点顺序计数器的值。用这种方法，我们利用队列的元素上的最终顺序，保证最老的元素是下一个消费的元素。

```
    /**
     * Add element to the queue.
     *
     * @param i
     * @return
     */

    boolean produce(int i) throws KeeperException, InterruptedException{
        ByteBuffer b = ByteBuffer.allocate(4);
        byte[] value;

        // Add child with value i
        b.putInt(i);
        value = b.array();
        zk.create(root + "/element", value, Ids.OPEN_ACL_UNSAFE,
                  CreateMode.PERSISTENT_SEQUENTIAL);

        return true;
    }
```

消费一个元素，消费者进程获得根节点的子节点，读取最小计数器值的节点，并返回元素。注意如果有冲突，然后两个竞争进程其中的一个将不能删除节点且删除操作会抛出一个异常。

调用 getChildren()方法返回词典式排序的子节点列表。因为词典式排序没有必要按照计数器的数字排序，我们需要确定哪个元素是最小的。为了确定哪个是最小的计数器值，我们便利集合，并删除每个节点的前缀"element"。

```
    /**
     * Remove first element from the queue.
     *
```

```
4.                 * @return
5.                 * @throws KeeperException
6.                 * @throws InterruptedException
7.                 */
8.                int consume() throws KeeperException, InterruptedException{
9.                    int retvalue = -1;
10.                   Stat stat = null;
11.
12.                   // Get the first element available
13.                   while (true) {
14.                       synchronized (mutex) {
15.                           List<String> list = zk.getChildren(root, true);
16.                           if (list.size() == 0) {
17.                               System.out.println("Going to wait");
18.                               mutex.wait();
19.                           } else {
20.                               Integer min = new Integer(list.get(0).substring(7));
21.                               for(String s : list){
22.                                   Integer tempValue = new Integer(s.substring(7));
23.                                   //System.out.println("Temporary value: " + tempValu
    e);
24.                                   if(tempValue < min) min = tempValue;
25.                               }
26.                               System.out.println("Temporary value: " + root + "/eleme
    nt" + min);
27.                               byte[] b = zk.getData(root + "/element" + min,
28.                                         false, stat);
29.                               zk.delete(root + "/element" + min, 0);
30.                               ByteBuffer buffer = ByteBuffer.wrap(b);
31.                               retvalue = buffer.getInt();
32.
33.                               return retvalue;
34.                           }
35.                       }
36.                   }
37.               }
38.           }
```

5.4 完整代码清单

5.4.1 SyncPrimitive.java

```
1.  import java.io.IOException;
2.  import java.net.InetAddress;
3.  import java.net.UnknownHostException;
4.  import java.nio.ByteBuffer;
5.  import java.util.List;
6.  import java.util.Random;
7.
8.  import org.apache.zookeeper.CreateMode;
9.  import org.apache.zookeeper.KeeperException;
10. import org.apache.zookeeper.WatchedEvent;
11. import org.apache.zookeeper.Watcher;
12. import org.apache.zookeeper.ZooKeeper;
13. import org.apache.zookeeper.ZooDefs.Ids;
14. import org.apache.zookeeper.data.Stat;
15.
16. public class SyncPrimitive implements Watcher {
17.
18.     static ZooKeeper zk = null;
19.     static Integer mutex;
20.
21.     String root;
22.
23.     SyncPrimitive(String address) {
24.         if(zk == null){
25.             try {
26.                 System.out.println("Starting ZK:");
27.                 zk = new ZooKeeper(address, 3000, this);
28.                 mutex = new Integer(-1);
29.                 System.out.println("Finished starting ZK: " + zk);
30.             } catch (IOException e) {
31.                 System.out.println(e.toString());
32.                 zk = null;
33.             }
34.         }
35.         //else mutex = new Integer(-1);
36.     }
37.
```

```
38.         synchronized public void process(WatchedEvent event) {
39.             synchronized (mutex) {
40.                 //System.out.println("Process: " + event.getType());
41.                 mutex.notify();
42.             }
43.         }
44.
45.         /**
46.          * Barrier
47.          */
48.         static public class Barrier extends SyncPrimitive {
49.             int size;
50.             String name;
51.
52.             /**
53.              * Barrier constructor
54.              *
55.              * @param address
56.              * @param root
57.              * @param size
58.              */
59.             Barrier(String address, String root, int size) {
60.                 super(address);
61.                 this.root = root;
62.                 this.size = size;
63.
64.                 // Create barrier node
65.                 if (zk != null) {
66.                     try {
67.                         Stat s = zk.exists(root, false);
68.                         if (s == null) {
69.                             zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
70.                                     CreateMode.PERSISTENT);
71.                         }
72.                     } catch (KeeperException e) {
73.                         System.out
74.                                 .println("Keeper exception when instantiating queue:
    "
75.                                         + e.toString());
```

```
76.                     } catch (InterruptedException e) {
77.                         System.out.println("Interrupted exception");
78.                     }
79.                 }
80.
81.                 // My node name
82.                 try {
83.                     name = new String(InetAddress.getLocalHost().getCanonicalHostN
    ame().toString());
84.                 } catch (UnknownHostException e) {
85.                     System.out.println(e.toString());
86.                 }
87.
88.             }
89.
90.             /**
91.              * Join barrier
92.              *
93.              * @return
94.              * @throws KeeperException
95.              * @throws InterruptedException
96.              */
97.
98.             boolean enter() throws KeeperException, InterruptedException{
99.                 zk.create(root + "/" + name, new byte[0], Ids.OPEN_ACL_UNSAFE,
100.                         CreateMode.EPHEMERAL_SEQUENTIAL);
101.                 while (true) {
102.                     synchronized (mutex) {
103.                         List<String> list = zk.getChildren(root, true);
104.
105.                         if (list.size() < size) {
106.                             mutex.wait();
107.                         } else {
108.                             return true;
109.                         }
110.                     }
111.                 }
112.             }
113.
```

```
114.            /**
115.             * Wait until all reach barrier
116.             *
117.             * @return
118.             * @throws KeeperException
119.             * @throws InterruptedException
120.             */
121.
122.            boolean leave() throws KeeperException, InterruptedException{
123.                zk.delete(root + "/" + name, 0);
124.                while (true) {
125.                    synchronized (mutex) {
126.                        List<String> list = zk.getChildren(root, true);
127.                            if (list.size() > 0) {
128.                                mutex.wait();
129.                            } else {
130.                                return true;
131.                            }
132.                        }
133.                    }
134.            }
135.        }
136.
137.        /**
138.         * Producer-Consumer queue
139.         */
140.        static public class Queue extends SyncPrimitive {
141.
142.            /**
143.             * Constructor of producer-consumer queue
144.             *
145.             * @param address
146.             * @param name
147.             */
148.            Queue(String address, String name) {
149.                super(address);
150.                this.root = name;
151.                // Create ZK node name
152.                if (zk != null) {
```

```
153.                 try {
154.                     Stat s = zk.exists(root, false);
155.                     if (s == null) {
156.                         zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
157.                                 CreateMode.PERSISTENT);
158.                     }
159.                 } catch (KeeperException e) {
160.                     System.out
161.                             .println("Keeper exception when instantiating queue:
    "
162.                                     + e.toString());
163.                 } catch (InterruptedException e) {
164.                     System.out.println("Interrupted exception");
165.                 }
166.             }
167.         }
168.
169.         /**
170.          * Add element to the queue.
171.          *
172.          * @param i
173.          * @return
174.          */
175.
176.         boolean produce(int i) throws KeeperException, InterruptedException{
177.             ByteBuffer b = ByteBuffer.allocate(4);
178.             byte[] value;
179.
180.             // Add child with value i
181.             b.putInt(i);
182.             value = b.array();
183.             zk.create(root + "/element", value, Ids.OPEN_ACL_UNSAFE,
184.                     CreateMode.PERSISTENT_SEQUENTIAL);
185.
186.             return true;
187.         }
188.
189.
190.         /**
```

```
191.             * Remove first element from the queue.
192.             *
193.             * @return
194.             * @throws KeeperException
195.             * @throws InterruptedException
196.             */
197.            int consume() throws KeeperException, InterruptedException{
198.                int retvalue = -1;
199.                Stat stat = null;
200.
201.                // Get the first element available
202.                while (true) {
203.                    synchronized (mutex) {
204.                        List<String> list = zk.getChildren(root, true);
205.                        if (list.size() == 0) {
206.                            System.out.println("Going to wait");
207.                            mutex.wait();
208.                        } else {
209.                            Integer min = new Integer(list.get(0).substring(7));
210.                            for(String s : list){
211.                                Integer tempValue = new Integer(s.substring(7));
212.                                //System.out.println("Temporary value: " + tempValu
    e);
213.                                if(tempValue < min) min = tempValue;
214.                            }
215.                            System.out.println("Temporary value: " + root + "/eleme
    nt" + min);
216.                            byte[] b = zk.getData(root + "/element" + min,
217.                                        false, stat);
218.                            zk.delete(root + "/element" + min, 0);
219.                            ByteBuffer buffer = ByteBuffer.wrap(b);
220.                            retvalue = buffer.getInt();
221.
222.                            return retvalue;
223.                        }
224.                    }
225.                }
226.            }
227.        }
```

```
228.
229.        public static void main(String args[]) {
230.            if (args[0].equals("qTest"))
231.                queueTest(args);
232.            else
233.                barrierTest(args);
234.
235.        }
236.
237.        public static void queueTest(String args[]) {
238.            Queue q = new Queue(args[1], "/app1");
239.
240.            System.out.println("Input: " + args[1]);
241.            int i;
242.            Integer max = new Integer(args[2]);
243.
244.            if (args[3].equals("p")) {
245.                System.out.println("Producer");
246.                for (i = 0; i < max; i++)
247.                    try{
248.                        q.produce(10 + i);
249.                    } catch (KeeperException e){
250.
251.                    } catch (InterruptedException e){
252.
253.                    }
254.            } else {
255.                System.out.println("Consumer");
256.
257.                for (i = 0; i < max; i++) {
258.                    try{
259.                        int r = q.consume();
260.                        System.out.println("Item: " + r);
261.                    } catch (KeeperException e){
262.                        i--;
263.                    } catch (InterruptedException e){
264.
265.                    }
266.                }
```

```
267.            }
268.        }
269.
270.        public static void barrierTest(String args[]) {
271.            Barrier b = new Barrier(args[1], "/b1", new Integer(args[2]));
272.            try{
273.                boolean flag = b.enter();
274.                System.out.println("Entered barrier: " + args[2]);
275.                if(!flag) System.out.println("Error when entering the barrier");
276.            } catch (KeeperException e){
277.
278.            } catch (InterruptedException e){
279.
280.            }
281.
282.            // Generate random integer
283.            Random rand = new Random();
284.            int r = rand.nextInt(100);
285.            // Loop for rand iterations
286.            for (int i = 0; i < r; i++) {
287.                try {
288.                    Thread.sleep(100);
289.                } catch (InterruptedException e) {
290.
291.                }
292.            }
293.            try{
294.                b.leave();
295.            } catch (KeeperException e){
296.
297.            } catch (InterruptedException e){
298.
299.            }
300.            System.out.println("Left barrier");
301.        }
302.    }
```

第 6 章 Zookeeper 常见问题和解决方案

6.1 Zookeeper 高级架构指南

在本文中，你会看到使用 Zookeeper 实现高级功能的指导方案。他们全部约定在客户端实现且不需要特殊的 Zookeeper 支持。希望社会各界在客户端类库里遵守这些约定提高易用性和鼓励标准化。

Zookeeper 最有趣的事情之一是虽然 Zookeeper 使用异步通知，你可以使用它构建同步一致性原件，如 queues 和 locks。如你将看到的那样，这是可行的因为 Zookeeper 在 updates 上强加了整体的顺序，并且有暴漏这个顺序的机制。

注意在下面的示例尝试采用最佳实践。特别的是，他们避免轮训，定时和任何导致"羊群效应"的东西，导致传输破裂和限制伸缩性。

有很多可以想象的有用的功能但不包括在这里 - 可撤销的读写优先级锁，只是一个例子。这里提到的一些结构 - locks，特别的是 - 说明某些问题，虽然你可以找到其他设计，如事件处理或 queues，执行相同方法更实际的方法。总之，这个章节的例子是为了激发想法。

6.2 开箱即用的应用程序：Name Service，Configuration，Group Membership

Name Service 和 configuration 是 Zookeeper 的两个主要应用。这两个功能由 Zookeeper API 直接提供。

另一个由 Zookeeper 直接提供的功能是 group membership。group 由一个节点代表。group 的成员在 group 节点下创建临时节点。Zookeeper 检测到故障的时候自动删除失败的成员节点。

6.3 阻塞

分布式系统使用 barriers 阻塞一组节点的处理直到遇见某个条件的时候才允许所有节点继续下去。Barriers 通过在 Zookeeper 指定一个 barrier 节点实现。如果 barriers 节点存在就在原位。这是假象的代码：

- 客户端在阻塞节点上调用 Zookeeper API 的 exists()方法，设置 watch 为 true。
- 如果 exists()返回 false，阻塞消失并且客户端继续。

- 另外，如果 exists()返回 true，客户端等待 Zookeeper 阻塞节点的 watch 事件。
- 当触发 watch 事件时，客户端重新调用 exists()，再次等待直到阻塞节点移除。

6.3.1 双重阻塞

双重阻塞可让客户端同步一个计算的开始和结束。当足够的进程加入了阻塞，进程开启他们的计算并一旦完成就离开阻塞。这个方法展示了怎么使用 Zookeeper 节点作为一个阻塞。

这个方法的代码的阻塞节点为 b。每个客户端进程 p 进入的时候注册到阻塞节点并在准备离开的时候注销。节点通过下面的 Enter 程序注册阻塞节点，它在开始运算之前等待直到 x 个客户端进程注册。(这里的 x 的值由你自己决定。)

Enter

1. 生成节点名字 n = b+“/”+p
2. 设置 watch: exists(b + "/ready", true)
3. 创建子节点: create(n, EPHEMERAL)
4. L = getChildren(b, false)
5. 如果 L 小于 x，等待 watch 事件
6. 否则 create(b + "/ready", REGULAR)

Leave

1. L = getChildren(b, false)
2. 如果没有子节点，退出
3. 如果 p 只是 L 的进程节点，删除并退出
4. 如果 p 是 L 的最小进程节点，等待 L 里最高的进程节点。
5. 否则删除如果仍然存在等待最小的进程节点
6. goto 1

进入的时候，所有的进程在 ready 节点上设置 watch 并创建一个临时节点作为阻塞节点的子节点。除了最后一个之外的每个进程进入阻塞并等待 ready 节点（在第 5 行）。进程创建第 x 个节点，最后的进程，将看到子节点列表里的 x 节点并创建 ready 节点，唤醒其他进程。注意等待的进程只在退出的时候唤醒，所以等待是高效的。

退出时，你不能使用 ready 这样的标记因为你正在监测进程节点离开。通过使用临时节点，阻塞之后的失败进程已经进入不阻止正确进程的完成。当进程准备好离开时，他们需要删除他们的进程节点并等待其他的进程做同样的事情。

当没有进程子节点存在时进程退出。然而，作为一个效率，你可以使用最低的进程节点作为 ready 标记。所有其他准备退出的进程监测最低的存在的进程节点离开，并且最低进程的拥有者监测任何其他进程节点(简单起见选择最高的)离开。

这意味着在每个节点删除只有单独的进程唤醒除了最后一个节点。当他删除时唤醒每一个。

6.4 队列

分布式队列是一个常见的数据结构。在 Zookeeper 里实现分布式队列，首先指定一个 znode 持有队列，也就是队列节点。分布式客户端通过调用 create()放进队列一些东西，包括路径名以"queue-"结尾、序列，并且在调用 create()的时候设置临时节点标记为 true。因为设置了序列标记，新路径的形式将是_path-to-queue-node_/queue-X，这里的 X 是单调递增的数字。一个客户端想要从队列删除调用 Zookeeper 的 getChildren()方法，在 queue 节点上设置 watch 为 true，并且使用最小的数值开始处理节点。客户端不需要发出另外的 getChildren()直到它耗尽从第一次调用 getChildren()获取的列表。如果在队列节点没有子节点，读取者等待 watch 通知再次检查队列。

注意

现在在 Zookeeper 秘诀目录里存在队列实现。它在发布包里--src/recipes/queue 目录

6.4.1 优先队列

实现一个优先队列，你只需要对一般的 queue recipe 做两个简单的修改。首先，添加进一个队列，路径结尾是"queue-YY"这里的 YY 是元素的优先级，数值越低，优先级越大。第二，当从队列移除时，如果触发了队列节点的 watch 通知,客户端使用最新的子列表，意味着客户端将原来获得的子列表置为无效。

6.5 锁

完整的分布式锁是全局同步的，意思是在任何时间点没有两个客户端持有相同的锁。他们可以使用 Zookeeper 实现。就像优先级队列，首先定义一个 lock 节点。

注意

现在在 Zookeeper 秘诀目录里有锁的实现。这是发布包里--src/recipes/lock

客户端想要获得一个锁要做以下事情：

- 调用 create()方法，参数是"_locknode_/lock_"的路径名、序列和临时节点标记
- 在 lock 节点上调用 getChildren()而不设置 watch 标记(这对避免羊群效应非常重要)
- 如果在第一步创建的路径名有最小序列值的后缀，客户端获得 lock 并退出协议

- 客户端在 lock 目录的下一个最小序列值的路径上使用 watch 标记调用 exists()
- 如果 exists()返回 false，进入第二步。否则，进入第二步之前等待上一步路径的通知。

解锁协议非常简单：客户端想要释放锁只需要简单的删除第一步创建的节点即可。

这里有一些要注意的事情：

- 一个节点的删除将会只引起一个客户端唤醒因为每个节点精确到一个客户端监测。用这种方法，你避免了羊群效应。
- 没有轮训和超时。
- 因为你实现锁的方式，非常容易看到竞争锁的成员、打破锁、调试锁问题等等。

6.5.1 共享锁

你在锁定协议上做一些改变就可以实现共享锁：

获得一个读取锁：

- 调用 create()方法创建一个"_locknode_/read-"的节点。这是稍后在协议里使用的 lock 节点。确保同时设置 sequence 和 ephemeral 标记。
- 在 lock 节点上调用 getChildren()方法而不设置 watch 标记 - 这非常重要，因为它避免羊群效应。
- 如果没有"write-"开头的子节点并且有一个比第一步创建的节点更小的序列号，客户端获得 lock 并可以离开协议。
- 否则，调用 exists()，使用 watch 标记，设置在 lock 目录的"write-"开头下一个更小序列号的子节点上。
- 如果 exists()返回 false，进入第二步。
- 否则，在进入第二步之前等待在上一步 pathname 的通知。

获得写入锁：

- 调用 create()创建一个"_locknode_/write-"的节点。这是协议里稍后说的 lock 节点。确保同时设置 sequence 和 ephemeral 标记。
- 不设置 watch 标记调用 getChildren()方法 - 这非常重要，因为它避免羊群效应。
- 如果没有序列号小于第一步设置节点序列号的子节点，客户端得到锁并退出协议。
- 调用 exists()，使用 watch 标记，设置在下一个更小序列号的节点上。
- 如果 exists()返回 false，进入第二步。否则，在进入第二步之前等待上一步 pathname 的通知。

注意

这个秘诀很可能创建一个羊群效应：当有一大批客户端等待一个读取锁，并且当最小序列号的"write-"节点被删除的同时或多或少的获得通知实际上。这是有效的行为：因为所有的等待的读者客户端应该释放因为他们有锁。羊群效应是指发布一个"herd"事实上只有一个或少量机器可以继续

6.5.2 可恢复的共享锁

稍微的修改一下共享锁协议，就可以实现可恢复的共享锁：

在 reader 和 wirter 锁协议的第一步，使用 watch 设置调用 getData，然后立刻调用 create()。如果客户端随后接收到在第一步创建的节点的通知，在那个节点上再次调用 getData()，设置 watch 并查找字符串"unlock"，它发送信号给客户端必须释放锁。这是因为，按照这个共享锁协议，你可以通过在 lock 节点调用 setData()要求客户端放弃锁，往那个节点写入"unlock"。

注意这个协议要求锁的持有者同意释放锁。这样的同意非常重要，特别是如果锁的持有者需要在释放锁之前做一些处理的时候。当然你可以一直实现可恢复的共享锁通过在协议里规定允许撤销者删除 lock 节点如果 lock 持有者在一定的时间之后还没有删除 lock。

6.6 两阶段提交

两阶段提交协议是一个让分布式系统里所有客户端同意提交或回滚事务的算法。

在 Zookeeper 里，你可以通过一个事务调节员节点实现两阶段提交，叫做"/app/Tx"，并且每个参与者叫做"/app/Tx/s_i"。协调者创建子节点时，它让内容定义。一旦事务里涉及到的每个参与者从协调者接收到事务，参与者读取每个子节点并设置 watch。然后每个参与者处理查询并通过写入他们各自节点投票"commit"或"abort"。一旦写入完成，就通知其他参与者，并且如果所有参与者投完票，他们可以决定"abort"或"commit"。注意一个节点提早可以决定"abort"。

这个实现有趣的一面是协调者的唯一角色是决定参与者，创建 Zookeeper 节点，并传到事务到相应的参与者。事实上，甚至传播事务可以通过 Zookeeper 写入事务节点实现。

在上面的描述中有两个重要的缺陷。一个是消息复杂性，它是 $O(n^2)$。第二个是通过临时节点不能检测失败。使用临时节点发现参与者的失败，参与者创建节点是非常必要的。

解决第一个问题，你可以只得到事务节点变化的通知，然后通知参与者一旦协调者达到一个决定。注意这个方法是可以扩展的，但他是很慢的，因为它要求所有通讯经过协调者。

解决第二个问题，你可以让协调者传播事务到参与者，并让每个参与者创建他们自己的临时节点。

6.7 领导者选举

使用 Zookeeper 做领导人选举的简单方法是创建代表客户端的"proposals"的节点的时候使用 SEQUENCE|EPHEMERAL 标记。方法是有一个节点，叫做"/election"，这样每个节点创建一个子节点"/election/n_"同时加上 SEQUENCE|EPHEMERAL 标记。sequence 标记，Zookeeper 自动的追加比之前子节点更高的序列号。最小序列号的是领导者。

这还不是全部。重要的是检测领导者的故障，以便于在当前领导者故障的时候选举新领导者。一般的解决方案是在所有的应用进程检测当前最小的节点，并当最小节点走开时检测它是不是新领导者(注意因为节点是临时的，如果领导者故障最小的节点会消失)。但是这会引起一个羊群效应：当前领导者故障后，其他所有进程接收到一个通知，并在"/election"上执行 getChildren()获得当前"/election"的子节点列表。如果客户端的数量非常大，它会引起 Zookeeper 服务处理操作的高峰。避免羊群效应，检测节点序列的下一个节点就够了。如果客户端接收一个节点通知，然后在这个没有更小节点的案例中，它变成新的领导者。注意这通过所有客户端检测相同节点避免羊群效应。

这是假想的代码：

让 ELECTION 成为应用选择的路径。自愿者成为领导者。

- 创建节点 z，路径是"ELECTION/n_"同时指定 SEQUENCE 和 EPHEMERAL 标记
- 让 C 成为"ELECTION"的子节点，并且 i 是 z 节点的序列号
- 检测"ELECTION/n_j"的变化，这里的 j 是最大序列号,这里 j 小于 i 并且 n_j 是 C 里的节点

接收节点删除的通知：

- 让 C 成为 ELECTION 的新子节点
- 如果 z 是 C 里的最小节点，然后执行领导者过程
- 否则，检测"ELECTION/n_j"的变化，这里的 j 是最大序列号，这里 j 小于 i 并且 n_j 是 C 里的节点

注意子节点列表里没有之前的节点并不意味着节点的创建者知道他是当前的领导者。应用可能考虑创建一个单独的节点去通知领导者已经执行了领导者选举过程。

第 7 章 Zookeeper 管理与部署

7.1 部署

本章节包含了 Zookeeper 部署信息以及以下主题：

- 系统要求
- 集群设置
- 单机服务和开发

前两个部分假定你在生产环境如数据中心安装 Zookeeper。最后的部分包含你在有限的基础上安装 Zookeeper - 评估、测试或者开发 - 而不是在生产

7.1.1 系统要求

支持的平台

- GNU/LINUX 支持服务端和客户端的开发和生产。
- Sun Solaris 支持服务端和客户端的开发和生产。
- FreeBSD 只支持客户端的开发和生产，Java NIO 选择器在 FreeBSD JVM 已经不支持了。
- Win32 只支持服务端和客户端的开发。
- MacOSX 只支持服务端和客户端的开发。

必要的软件

Zookeeper 运行在 JDK1.6 或更高版本以上。它作为 Zookeeper 服务的整体运行。对于整体，三个 Zookeeper 服务器是最小的推荐数量，并且我们建议它们运行在不同的机器上。在 Yahoo,Zookeeper 通常部署到专用的 RHEL 容器，双核处理器，2GB 内存，80GB 的硬盘驱动。

7.1.2 集群设置

对于可靠的 Zookeeper 服务，你应该部署 Zookeeper 在一个集群环境里。只要集群的多数服务可用，集群服务就是可用的。因为 Zookeeper 要求一个大多数，进群最好使用奇数个机器。例如，4 台主机的 Zookeeper 只可以处理单机的故障；如果两个主机故障，剩下的两个机器不能成为大多数。然而，5 台主机的 Zookeeper 可以处理两个机器的故障。

下面是设置成为集群服务的步骤。这些步骤应该在集群里的每台主机上执行：

1. 安装 Java JDK。你可以使用系统的原生包装系统，或者从 JDK 下载：http://java.sun.com/javase/downloads/index.jsp

2. 设置 Java 堆大小。这对于避免 swapping 非常重要，它将严重降低 Zookeeper 的性能。确定正确值，使用负载测试，并确定低于引起 swap 的的限制。保守的 - 对于 4GB 的机器使用最大堆大小 3GB。

3. 安装 Zookeeper 服务包。可以从以下地址下载：http://zookeeper.apache.org/releases.html

首先，新建一个配置文件。这个文件可以叫任何名字。使用下面的设置作为出发点：

```
tickTime=2000

dataDir=/var/lib/zookeeper/

clientPort=2181

initLimit=5

syncLimit=2

server.1=zoo1:2888:3888

server.2=zoo2:2888:3888

server.3=zoo3:2888:3888
```

你可以在配置参数部分找到这些配置的含义和其他的配置项。这里有以下几点要解释：Zookeeper 集群的每个主机应该知道集群里的每个主机。通过 server.id=host:port:port 的形式来完成。参数 host 和 port 非常明确。通过为每个主机创建一个 myid 文件把 server id 归于每个主机，这个文件放在配置文件参数 dataDir 指定的 data 目录里。

- myid 文件只包含主机 id 的文字。所以服务器 1 的 myid 只包含文字"1"，没有其他任何东西。id 值在集群中必须是唯一的并且应该在 1 到 255 之间。
- 如果你设置了配置文件，你可以启动 Zookeeper 服务：

```
$ java -cp zookeeper.jar:lib/slf4j-api-1.6.1.jar:lib/slf4j-log4j12-1.6.1.jar:lib/log4j-1.2.15.jar:conf \ org.apache.zookeeper.server.quorum.QuorumPeerMain zoo.cfg
```

QuorumPeerMain 启动 Zookeeper 服务，同事还注册 JMX 管理 beans 他允许通过 JMX 管理控制台管理。Zookeeper JMX document 包含 JMX 管理 Zookeeper 的详细说明。

通过连接到主机测试你的部署：

在 Java，你可以运行下面的命令执行简单的操作

```
$ java -cp zookeeper.jar:lib/slf4j-api-1.6.1.jar:lib/slf4j-log4j12-1.6.1.jar:lib/log4j-1.2.15.jar:conf:src/java/lib/jline-0.9.94.jar \ org.apache.zookeeper.ZooKeeperMain -server 127.0.0.1:2181
```

在 C，你可以编译单线程客户端或者多线程客户端：Zookeeper 源码里的 c 子目录。这个编译单线程客户端：

```
$ make cli_st
```

这个编译多线程客户端：

```
$ make cli_mt
```

运行程序给你一个运行简单类似文件系统操作的 shell。使用多线程客户端连接到 Zookeeper，例如，你运行：

```
$ cli_mt 127.0.0.1:2181
```

7.1.3 单机服务器和开发人员设置

如果你以开发的目的安装 Zookeeper，你可以向安装 Zookeeper 的单机服务器实例，然后安装 Java 或 C 客户端类库绑定到你的开发机。

安装单服务器实例的步骤可上面的类似，除了配置文件更简单了。你可以在 Zookeeper 入门指南的安装和运行 Zookeeper 单机模式的章节里找到完整说明。

安装客户端类库的资料，参考 Zookeeper 开发人员指南的绑定章节。

7.2 管理

本章节包含关于运行和维护 Zookeeper 的资料，并包含了这些主题：

设计 Zookeeper 部署；配备。

Zookeeper 的优势和局限

- 管理
- 维护
- 监管
- 监控
- 记录
- 故障排除
- 配置参数

- Zookeeper 命令：四字母单词
- 数据文件管理
- 避免事情
- 最佳实践

7.2.1 设计 Zookeeper 部署

Zookeeper 的可靠性取决于两个基本设想。

- 只有少数服务会失败。这里的 Failure 意思是主机损坏，或者在网络里发生一些错误使服务从多数里分开。
- 部署机器操作正确。操作正确意思是正确的执行代码，有正常工作的时钟，且有一致性执行的网络组件存储。

下面的章节包含 Zookeeper 管理员最大化这些设想成为事实的概率的注意事项。其中的一些是跨机的注意事项，以及在部署里每个机器的其他一些应该考虑的事情。

7.2.2 跨机器要求

为了 Zookeeper 服务是可用的，可以互相通信的机器必须是多数可用的。创建一个可以容错 F 个机器的部署，你应该部署 2*F+1 个机器。因此，一个由三台机器组成的部署可以处理一个故障，一个五台机器组成的部署可以处理两个故障。注意一个由 6 台机器组成的部署只可以处理两个故障因为 3 台机器不是多数。由于这个原因，Zookeeper 部署通常由奇数个机器组成。

为了实现容错的最大概率你应该尝试使机器故障独立。例如，如果多数的机器共享相同的网关，网关故障可能引起一个关联故障并降低服务。共享电路、冷却系统也是如此，等等。

7.2.3 单机要求

如果 Zookeeper 必须和其他应用竞争访问资源如存储介质、CPU、网络或内存，它的性能将会受到显著影响。Zookeeper 有强大的耐用性保证，意味着他使用存储介质在操作允许完成之前记录变化。你应该意识到这个依赖性，并且用心确保 Zookeeper 操作没有通过介质绑定。给你一些降低这些的建议：

- Zookeeper 的事务日志必须在专用设备上。(一个专用分区不够。)Zookeeper 按顺序写日志，和其他应用分享你的日志设备可能引起寻找和争夺，这反过来又可能导致多次延迟。
- 不要将 Zookeeper 放置在可能引起 swap 的环境。为了 Zookeeper 的及时性功能，它简直不能允许 swap。因此，确保最大的堆大小不是把大于真实可用内存给 Zookeeper。关于这个更多资料，查看下面的 Things to Avoid。

7.2.4 配备

7.2.5 Zookeeper 的优势和局限：管理与维护

Zookeeper 集群的中长期维护你必须意识到：

持续的数据目录清理
Zookeeper 数据目录包含由服务存储的 znode 持久化拷贝文件。有快照和事务日志文件。znodes 的变化追加到事务日志，偶尔当日志变大的时候，所有 znode 当前状态的快照写入到文件系统。这个快照取代所有的以往日志。

使用默认配置的时候，Zookeeper 服务器不会删除老版本快照和日志文件(查看下面的 autopurge)，这是操作员的责任。每个服务环境都是不同的因此管理这些文件的要求可能是不同的。

PurgeTxnLog 功能实现了一个管理员可以使用的简单保留策略。API docs 包含调用规则的详细资料。

在下面的例子中最近一次的快照统计和他们相应的日志被保留并删除其他的。典型的值应该大于 3。这可以在 Zookeeper 服务器上运行一个计划任务清理每天的日志。

```
java -cp zookeeper.jar:lib/slf4j-api-1.6.1.jar:lib/slf4j-log4j12-1.6.1.jar:lib/log4
j-1.2.15.jar:conf org.apache.zookeeper.server.PurgeTxnLog-n
```

自动的清除快照和对应事务日志在 3.4.0 版本引入并且可以通过下面的配置参数启用：autopurge.snapRetainCount 和 autopurge.purgeInterval。关于这个的更多信息，查看下面的高级配置。

调试日志清理(log4j)
查看本文档的 logging 章节。它希望你使用内置的 log4j 功能设置滚动文件追加。发布文件 tar 包里的 conf/log4j.properties 提供了一个这样的实例。

7.2.6 监理

你可能想有一个管理进程管理每个 Zookeeper 服务进程(JVM)。Zookeeper 服务器的"fail fast"设计意味着如果发生错误他就停止(进程退出)。Zookeeper 服务集群是高度可靠的，这意味着当服务器从集群离开他始终是可用的和服务请求。此外，因为集群是"自我修复的"，故障的服务器一旦重启，就会自动的重新加入集群而不需要人工干预。

监管进程像 daemontoos 或 SMF(监管进程可以有其他的选择，可以根据个人喜好使用，这里只是两个例子)管理你的 Zookeeper 服务去吧如果进程非正常的离开他讲自动的重启并快速重新加入集群。

7.2.7 监控

Zookeeper 服务可以通过两个途径监控：

- 通过使用 4 个字母单词命令
- JMX。查看你环境/要求适合的章节

7.2.8 记录

Zookeeper 使用 log4j1.2 作为它的日志基础设施。Zookeeper 默认的 log4j.properties 文件在 conf 目录里。Log4j 要求 log4j.properties 在工作目录里(Zookeeper 运行目录)或可以从 classpath 可访问的路径里。

获取更多资料，查看 log4j 手册的 Log4j Default Initialization Procedure

7.2.9 故障排除

因为文件损坏服务器不来；

因为 Zookeeper 服务器里的事务日志的一些文件损坏，服务器可能不能读取他的数据库且不能出现。在加载 Zookeeper 数据库时你会看到一些 IOException。在这种情况下，确保集群中其他服务器是正常工作的。在命令端口使用"stat"命令查看他们是否是正常的。检查集群中其他所有服务是正常的之后，你可以继续清理损坏服务器的数据库。删除 datadir/version-2 和 datalogdir/version-2 的所有文件。重启服务器。

7.2.10 配置参数

Zookeeper 的行为由 Zookeeper 配置文件控制。这个文件是这样设计的精确相同的文件可以由组成 Zookeeper 服务的所有服务器使用(假定磁盘格局都是一样的)。如果服务器使用不同的配置文件，必须确保所有不同配置文件的服务器列表。

最小配置
以下是必须定义在配置文件的最小配置关键词：

clientPort

监听客户端连接的端口；这是客户端尝试连接的端口。

dataDir

Zookeeper 存储内存数据库快照的位置，除非有特殊说明，更新的事务日志也存储进数据库。

注意

注意你存放事务日志的地方。一个专用的事务日志装在是保持良好性能的关键。存放日志到忙碌的装置会严重影响性能。

tikeTime

单个 tick 的长度，它是 Zookeeper 使用的基本时间单位，以毫秒为计量单位。它用于控制心跳和超时。例如，最小的 session 超时是两倍 ticks。

高级配置

本部分的配置是可选的。你可以使用它们进一步优化 Zookeeper 服务的运行状况。有一些可以通过 Java 系统属性设置，一般是 zookeeper.keyword 的形式。精确的可用属性，在下面提到。

dataLogDir

(没有 Java 系统属性)

这个选项将决定机器将事务日志写到 dataLogDir 而不是 dataDir。这允许使用专用的日志设备，帮助你避免日志和快照之间的竞争。

注意

专用的日志设备对吞吐量和稳定的延迟有很大的影响。强烈建议你使用专用的日志设备并设置 dataLogDir 指向那个设备的目录，然后确保指定 dataDir 不在那个设备上。

globalOutstandingLimit

(Java 系统属性：zookeeper.globalOutstandingLimit)

客户端提交请求的速度快于 Zookeeper 处理他们的速度，特别是如果有很多客户端。为了防止 Zookeeper 由于排队请求而耗尽内存，Zookeeper 将限制客户端在系统里最多只有不超过 globalOutstandingLimit 个未完成的请求。默认限制是 1000。

preAllocSize

(Java 系统属性：zookeeper.preAllocSize)

为了避免在 preAllocSize kb 的块的事务日志里寻找分配的空间。默认的块大小是 64M。改变块大小的原因之一是如果快照更多就减少块大小(查看 snapCount)。

snapCount

(Java 系统属性：zookeeper.snapCount)

Zookeeper 记录事务到一个事务日志。snapCount 个事务写入日志文件之后启动一个快照并创建一个新的事务日志文件。默认的 snapCount 是 100000。

traceFile

(Java 系统属性：requestTraceFile)

如果定义了这个选项，请求将会被记录的名为 traceFile.year.month.day 的跟踪文件。使用这个选项提供有用的调试信息，但是会影响性能。(注意：系统属性没有 zookeeper 前缀，并且配置变量名和系统属性不一样。是的 - 是不一致的，让人讨厌)

maxClientCnxns

(没有 Java 系统属性)

限制单个客户端的并发连接数(在 socket 级别)，通过 IP 地址识别，Zookeeper 集群的单个成员。这用于预防某些类的 Dos 攻击，包括文件描述符耗尽。默认是 60.设置为 0 是完全的删除并发连接的限制。

clientPortAddress

3.3.0 新属性：监听客户端连接的地址(ipv4,ipv6 或 hostname)；换言之，客户端尝试连接的地址。这是可选的，默认的我们以绑定的方式在服务器上任何连接到 clientPort 的 address/interface/nic 都将被接受。

minSessionTimeout

(没有 Java 系统属性)

3.3.0 新属性：服务器允许客户端交涉的最小 session 超时时间。默认是 2 倍的 tickTime。

maxSessionTimeout

(没有 Java 系统属性)

3.3.0 新属性：服务器允许客户端交涉的最大 session 超时时间。默认是 20 倍的 tickTime。

fsync.warningthresholdms

(Java 系统属性：fync.warningthresholdms)

3.3.4 新加入：事务日志所用时间大于这个值的时候输出警告消息。值的单位是毫秒默认值是 1000.这个值只可以作为系统属性设置。

autopurge.snapRetainCount

(没有 Java 系统属性)

3.4.0 新加入：启用时，Zookeeper 自动清洗功能保留 autopurge.snapRetainCount 个最近的快照和各自 dataDir/dataLogDir 对应的事务日志并删除先前的。默认值是 3.最小值是 3.

autopurge.purgeInterval

(没有 Java 系统属性)

3.4.0 新加入：触发清洗任务的时间间隔，以小时为单位。设置正整数(1 或更高)启用自动清洗。默认是 0。

syncEnabled

(Java 系统属性：zookeeper.observer.syncEnabled)

3.4.6,3.5.0 更新：观察者记录事务和快照到硬盘。这降低观察者重启的恢复时间。设置"false"禁用这个功能。默认是"true"

集群选项

本章节的选项设计用于服务器的全员 -- 换句话说，部署集群时。

electionAlg

(没有 Java 系统属性)

用于选举实现。"0"代表原始的 UDP 版本，"1"代表快速领导者选举的不授权的 UDP 版本，"2"代表授权的快速领导者选举的 UDP 版本，"3"代表快速领导者选举的 TCP 版本。当前，默认值是 3.

注意

领导者选举 0,1,2 的实现现在是弃用的。我们计划在下个发布版本里移除他们，到那时只有 FastLeaderElection 可用。

initLimit

(没有 Java 系统属性)

时间数量，单位是 ticks，允许追随者连接和同步 leader。根据需要增加这个值，如果 Zookeeper 管理的数据量非常大。

leaderServers

(Java 系统属性：zookeeper.leaderServers)

Leader 接受客户端连接。默认值是"yes"。leader 机器协调更新。默认是 yes，它意味着 leader 接受客户端连接。

注意

Zookeeper 集群大于 3 台机器时强烈建议开启领导者选举。

server.x=[hostname]:nnnnn[:nnnnn]，等等

(没有 Java 系统属性)

组成 Zookeeper 机器的服务器。当服务启动时，通过查找数据目录里的 myid 文件确定它是哪个服务器。那个文件包含服务器号，ASCII 的形式，并且应该匹配左边 server.x 里的 x。

使用 Zookeeper 服务的服务器的客户端必须匹配每个 Zookeeper 服务。

有两个端口号 nnnnn。第一个用于连接领导者，第二个用于领导者选举。领导者选举端口只在 electionAlg 是 1、2、3 时必要。如果 electionAlg 是 0，第二个端口就没必要。如果你想在单机上测试多服务，每个服务要使用不同的端口。

syncLimit

(没有 Java 系统属性)

时间总数，以 ticks 为单位，运行追随者同步 Zookeeper。如果追随者失败太久，就掉线。

group.x=nnnnn[:nnnnn]

(没有 Java 系统属性)

启用分层的法定人数构造。"x"是 group 标识符,"="后面的数字对应服务标识符。左边的任务是冒号分隔的服务标识符。注意 groups 必须是不相交的并且所有 groups 联盟必须是 Zookeeper 全体。

weight.x=nnnnn

(没有 Java 系统属性)

兼用"group"，当形成法定人数时它给服务分配一个权重。投票时这个值对应服务的权重。Zookeeper 的一些部分要求投票如领导者选举和自动广播协议。默认服务器的权重是 1.如果配置定义了 groups，但不是权重，将会给所有的服务器权重设置为 1.

cnxTimeout

(Java 系统属性：zookeeper.cnxTimeout)

设置领导者选举通知打开连接的超时值。只适用于 electionAlg3。

注意

默认值是 5 秒

身份认证 & 授权选项

本章节的选项允许通过服务执行控制身份认证/授权。

zookeeper.DigestAuthenticationProvider.superDigest

(Java 系统属性：zookeeper.DigestAuthenticationProvider.superDigest)

默认这个功能是禁用的

3.2 新加入：使用 zookeeper 全体管理员能够以超级用户的身份访问 znode。特别的是对于授权超级用户的用户没有 ACL 检查

org.apache.zookeeper.server.auth.DigestAuthenticationProvider 可以用于生产 superDigest，用"super:"的参数调用它。规定生成的"super:"作为系统属性值当启动每个服务时。

当授权 Zookeeper 服务通过"digest"的计划和"super:"的 authdata 时。注意 digest 认证 通过服务，它是谨慎的做法在本地使用这个授权方法或在安全协议之上。

实验性的选项/功能

目前处于试验阶段的新特性

只读模式的服务

(Java 系统属性：readonlymode.enabled)

3.4.0 加入：设置为 ture 启用只读模式支持(默认不启用)。在这个模式里 ROM 客户端可以从

ZK 服务里读取数据，但是不能写入数据和从其他客户端看到变化。查看 ZOOKEEPER-784 查看更多。

不安全选项

下面的选项可以用，但用的时候要小心。每个的风险说明和变量一起。

forceSync

(Java 系统属性:zookeeper.forceSync)

在完成处理更新之前要求通过更新事务日志的媒介。如果这个选项设置为 no，Zookeeper 将不要求同步更新媒介。

jute.maxbuffer:

(Java 系统属性：jute.maxbuffer)

此选择系可以通过 Java 系统属性设置。没有 zookeeper 前缀。它指定 znode 中可以存储的数据的最大值。默认是 0xffffff,或低于 1M。如果这个选项变了，系统属性必须在所有服务器和客户端上设置，否则会出现问题。这确实是一个合理性检查。Zookeeper 用来存储数据大约在 KB 大小。

skipACL

(Java 系统属性：zookeeper.skipACL)

跳过 ACL 检查。这会增加吞吐量，但是会像所有人打开完整的访问权限

quorumListenOnAllIPs

设置为 true 时，Zookeeper 服务器将监听所有同行可用 IP 地址的连接，并且不只是服务器里配置的地址。它影响连接处理 ZAB 协议和快速领导者选举协议。默认是 false。

使用 Netty 框架通信

3.4 新加入：Netty 是一个基于 NIO 的客户端/服务端通信框架，它简化了 Java 应用很多复杂的网络层通信。此外 Netty 框架支持数据加密(SSL)和身份验证(证书)。还有可选择的功能可以独立的开启或关闭。

在版本 3.4 之前 Zookeeper 一直直接使用 NIO，然而在版本 3.4 和以后的版本支持 NIO 选择 Netty。NIO 仍然是默认值，然而基于 Netty 的通信可以通过设置环境变量"zookeeper.serverCnxnFactory"为"org.apache.zookeeper.server.NettyServerCnxnFactory"使用。可以在客户端或服务端设置这个选项，一般两边都设置，这个由你决定。

TBD - netty 的调优选项

TBD - 怎么管理数据加密

TBD - 怎么管理证书

7.2.11 Zookeeper 命令：四个字母的单词

Zookeeper 支持一小组命令。每个命令由四个字母组成。你通过 telnet 或 nc 在客户端端口发行命令到 Zookeeper。

三个比较有趣的命令："stat"给出了服务器和连接的客户端的一般信息，"srvr"和"cons"分别提供服务器上和连接的扩展细节。

conf

3.3.0 加入：打印服务配置详情

cons

3.3.0 加入：列出所有连接到这个服务器的客户端完整的 connection/session 详情。包括接收/发送数据包的数量，session id，操作延迟，最后执行的操作，等等。

crst

3.3.9 加入：重置所有连接的统计信息。

dump

列出未交付的 session 和临时节点。这只适用于领导者。

envi

打印服务环境详情。

ruok

测试服务器是否正常运行在一个无错误状态。如果正在运行服务器回复 imok。否则不响应。"imok"的回复并不表明服务器已经加入 quorum，至少说服务进程是活动的并绑定到指定的客户端端口。使用"stat"获取详细信息。

srst

重置服务器统计

srvr

3.3.0 加入：列出服务器的完整详情。

stat

列出服务器和连接的客户端摘要信息。

wchs

3.3.0 加入：列出服务器 watch 的摘要信息。

wchc

3.3.0 加入：列出服务器 watch 的详细信息，通过 session。这个输出相关 watch 的 session 清单。注意，这个选项取决于 watch 的数量可能开销比较大，使用时要小心。

wchp

3.3.0 加入：列出服务器 watch 的详细信息，通过路径。这个输出相关 session 的路径清单。注意，这个选项取决于 watch 的数量可能开销比较大，使用时要小心。

mntr

3.4.0 加入：输出可以监控集群状态的变量清单。

```
1.      $ echo mntr | nc localhost 2185
2.      zk_version  3.4.0
3.      zk_avg_latency  0
4.      zk_max_latency  0
```

```
zk_min_latency  0
zk_packets_received 70
zk_packets_sent 69
zk_outstanding_requests 0
zk_server_state leader
zk_znode_count   4
zk_watch_count  0
zk_ephemerals_count 0
zk_approximate_data_size    27
zk_followers    4                   - only exposed by the Leader
zk_synced_followers 4               - only exposed by the Leader
zk_pending_syncs    0               - only exposed by the Leader
zk_open_file_descriptor_count 23    - only available on Unix platforms
zk_max_file_descriptor_count 1024   - only available on Unix platforms
```

输出和 java 属性格式兼容并且内容可能随着时间变动。你的脚本应该改变。

这里有个 ruok 命令的实例

```
$ echo ruok | nc 127.0.0.1 5111
imok
```

7.2.12 数据文件管理

Zookeeper 存储它的数据到一个数据目录和它的事务日志到一个事务日志目录。默认这两个目录是一样的。服务器可以配置存储事务日志文件到一个单独的目录里。使用专用的设备存储事务日志可以增加吞吐量和减缓延迟。

数据目录
数据目录里有两个文件：

myid - 包含一个单独的可读的整型数字 ASCII 文本，代表 server id。

snatshop.- 保留数据树的模糊快照。

每个 Zookeeper 服务器有一个唯一 id。这个 id 用于两个地方：myid 文件和配置文件。myid 文件标识服务器。配置文件通过它的 server id 暴露每个服务器的联系方式。当一个 Zookeeper 服务实例启动，它从 myid 文件读取它的 id，然后使用这个 id，从配置文件读取它应该监听的端口。

数据目录的 snapshot 文件是 Zookeeper 服务采取快照、更新的模糊快照。snapshot 文件名字的后缀是 zxid，在开启快照时最后提交的事务 id。因此，快照包括在快照过程中数据树变化的子集。然后快照可能不对应任何数据树，因为这个

原因我们叫它为模糊的快照。Zookeeper 仍然可以使用这个快照因为它运用了更新的等幂性质。使用模糊的快照回复事务日志 Zookeeper 得到日志结尾的系统状态。

日志目录

日志目录包含 Zookeeper 事务日志。在任何更新发生之前，Zookeeper 确保代表更新的事务写入不易丢失的存储。每次启用新的日志文件快照就开始。日志文件的后缀是写到日志的第一个 zxid。

文件管理

快照和日志文件的格式在单个 Zookeeper 服务器和复制的 Zookeeper 服务器的不同配置之间没有变化。因此，你可以从运行的 Zookeeper 服务器上拉取这些文件到部署的机器。

使用老的日志和快照文件，你可以考虑 Zookeeper 服务的上一个状态和恢复状态。LogFormatter 类可让管理员查看日志里的事务。

Zookeeper 服务创建快照和日志文件，当从不删除它们。数据和日志文件的保留策略由 Zookeeper 服务以外的实现。服务本身只需要最后完整的模糊快照和日志文件。查看 maintennance 章节可以了解 Zookeeper 存储维护的保留策略设置。

7.2.13 要避免的事情

下面的问题可以通过 Zookeeper 正确的配置避免：

- 不一致的服务器清单

客户端使用的 Zookeeper 服务器清单必须和每个 Zookeeper 服务的一致。每个 Zookeeper 服务配置文件的服务器清单应该和其他的一致。

- 不正确的事务日志放置

Zookeeper 性能的关键部分是事务日志。Zookeeper 在响应之前同步事务到媒介。一个专用的事务日志装置是保持良好性能的关键。将日志放置在繁忙的装置上会严重影响性能。如果你只有一个存储设备，将跟踪文件放置在 NFS 上并增加 snapshotCount；这不能解决问题，但能改善。

- 不正确的 Java 堆大小

你应该特别注意正确的设置 Java 最大堆大小。特别的是，你不应该营造 Zookeeper 交换磁盘的情况。磁盘可让 Zookeeper 死亡。每个事情都是有序的，所以如果处理一个请求交换磁盘，所有其他队列里的请求很可能会做同样的事情。磁盘，不要 SWAP。

7.2.14 最佳实践

为了获得最佳效果，注意下面的优秀 Zookeeper 实践清单：

对于 multi-tennant 安装查看 Zookeeper"chroot"支持的详细章节，这对部署多个应用/服务连接到单个 Zookeeper 集群非常有用。

7.3 配额

Zookeeper 包含 namespace 和 bytes quotas。你可以使用 ZookeeperMain 类设置 quotas。Zookeeper 打印 WARN 消息如果用户超过分配给他们的 quota。消息打印在 Zookeeper 的 log 里。

```
$java -cp zookeeper.jar:src/java/lib/log4j-1.2.15.jar/conf:src/java/lib/jline-0.9.9
4.jar \ org.apache.zookeeper.ZooKeeperMain -server host:port
```

上面的命令提供给你了一个使用 quotas 的命令行选项。

7.4 设置配额

你可以使用 setquota 设置 Zookeeper 节点上的 quota。它有一个设置配额的选项-n(namespace)和-b(bytes)。

Zookeeper quota 存储在 Zookeeper 自身的/zookeeper/quota。只有管理员才可以设置更改/zookeeper/quota，禁用其他用户。

7.5 配额清单

你可以使用 listquota 列出 Zookeeper 节点上的 quota。

7.6 删除配额

你可以使用 delquota 删除 Zookeeper 节点上的 quota。

7.7 JMX

Apache Zookeeper 已经扩展了对 JMX 的支持，允许你展示和管理 Zookeeper 整体服务。

本篇文章假定你有 JMX 的基础知识。查看 Sun JMX Technology 页面快速学习 JMX。

查看 JMX 管理指南纤细了解建立 VM 实例的本地和远程管理。默认包含的 zkServer.sh 只支持本地管理 - 查看链接文档启用支持远程管理(超过了本文档的范围)。

7.7.1 启动启用 JMX 的 Zookeeper

org.apache.zookeeper.server.quorum.QuorumPeerMain 类将启动了一个 JMX 可管理的 Zookeeper 服务。这个类在初始化期间注册适当的 MBeans 支持实例的 JMX 监控和管理。查看 bin/zkServer.sh 使用 QuorumPeerMain 启动 Zookeeper 的实例。

7.7.2 运行 JMX 控制台

有很多可用的 JMX 控制台可以连接到运行的服务。Java JDK 有一个简单的 JMX 控制台叫做 jconsole，它可以用于连接 Zookeeper 并检查运行的服务。一旦你使用 QuorumPeerMain 启动 jconsole 启动了 Zookeeper，它通常在 JDK_HOME/bin/jconsole。当"新连接"窗口展示连接到本地进程或使用远程连接。默认，VM 的"overview"标签被展示。选择"MBeans"标签。

你现在应该在左边看 org.apache.ZookeeperService。扩展这一项取决于你怎么启动服务，你将能够监控和管理相关的各种服务功能。

还要注意 Zookeeper 还注册 log4j Mbeans。在左手边相同的部分你会看到"log4j"。展开它通过 JMX 管理 log4j。特别感兴趣的是能够动态改变日志级别通过自定义 appender 和 root 阀值。禁用 Log4j MBean 注册可以在启动 Zookeeper 时传送 -Dzookeeper.jmx.log4j.disable=true 到 JVM。

7.7.3 Zookeeper MBean 参考

这个表格详细列出了复制的 Zookeeper 整体里的服务器的 JMX。这是生产环境典型案例。

MBean	MBean 对象名字	描述

Quorum	ReplicateServer_id<#>	代表 Quorum 或全体 - 所有集群成员的父级。注意对象名字包含 JMX 连接的服务的"myid"
LocalPeer|RemotePeer	replica.<#>	代表一个本地或远程对的。注意对象名字包含服务的"myid"
Leader	Leader	表明父级复制品是领导者并且提供那个服务的属性/操作。注意 Leader 是 ZookeeperServer 的子类，所以它提供与 ZooeeperServer 节点相关的所有信息
Follower	Follower	表明父级复制品是一个跟随者并且提供那个服务的属性/操作。注意 Follower 是 ZookeeperServer 的子类，所以它提供与 ZookeeperServer 节点相关的所有信息。
DataTree	InMemoryDataTree	内存中 znode 数据库的统计信息，还有访问数据的统计信息的操作。InMemoryDataTrees 是 ZookeeperServer 节点的子类
ServerCnxn	<session_id>	每个客户端连接的统计信息，和那些连接的操作。注意对象名字是连接的十六进制的 session id。

这个表格详细列举了单台服务的 JMX。通常单台只用于开发环境。

MBean	MBean 对象名	描述
ZookeeperServer	StandaloneServer_port<#>	运行的服务的统计信息，还有重置这些属性的操作。注意对象名包含服务的客户端端口。
DataTree	InMemoryDataTree	znode 内存数据库的统计信息，还有访问数据统计信息的操作。
ServerCnxn	<session_id>	每个客户端连接的统计信息，还有那些连接的操作。注意对象名字是连接的十六进制形式的 session id

第 8 章 Zookeeper 内部构件

8.1 简介

本文档包含 Zookeeper 内部运作的资料。到目前为止，它讨论这些主题：

- 原子广播
- 日志

8.2 原子广播

Zookeeper 的核心是一个保持所有服务同步的原子消息系统。

8.2.1 保证，属性和定义

Zookeeper 通过下面的内容使用消息系统提高特殊保证：

可靠的交付

如果消息 m 由一个服务器投递，它最终将由所有服务投递。

总序

如果 a 消息在 b 消息之前由一个服务器投递，所有服务器对 a 消息的投递都在 b 之前。如果 a 和 b 是传递消息，a 将在 b 之前投递或者 b 在 a 之前投递。

因果顺序

如果消息 b 在消息 a 之后发送，a 由 b 的发送者投递,a 的顺序一定在 b 之前。如果发送者在发送 b 之后发送 c，c 的顺序一定在 b 之后。

Zookeeper 消息系统还需要是高效、可靠和易于实现和维护的。我们大量使用消息，所以我们需要系统能够处理每秒数以千计的请求。虽然我们可能需要至少 K +1 个正确的服务发送新消息，我们必须能够从相关故障恢复，如断电。当我们实现系统时我们只有很少的时间和稀缺的工程资源，所以我们需要一个易接近的工程和易于实现的协议。我们发现我们的协议满足所有的这些目标。

我们的协议假定我们可以在服务器间构建点对点 FIFO(先入先出)通道。而类似的服务通常假定消息投递可以丢失和重排序消息。我们 FIFO 通道的设想是对使用 TCP 通讯非常实用的。特别的是我们依靠下面的 TCP 性质：

有序传递

数据在相同顺序里投递并发送，消息 m 在所有在 m 之前投递的消息发送之后投递。(这个推论是如果消息 m 丢失，m 消息以后的所有消息都会丢失)

关闭之后无消息

一旦 FIFO 通道关闭，将不会再从它接收消息。

FLP 证明在异步分布式系统里如果故障不能取得共识。为了确保在面对故障时取得共识我们使用超时时间。然而，我们依靠时间判断活性而不是正确性。所以，如果超时时间停止消息系统可能会挂，但是它不会违反它的保证。

描述 Zookeeper 消息协议时我们会谈论数据包、建议和信息：

数据包

通过 FIFO 通道发送的字节序列

建议

一个单位的协议。建议通过交换数据包获得一致。多数的建议包含信息，然而 NEW_LEADER 建议是不对应消息的建议实例。

信息

序列字节是原子广播到所有服务。信息放入一个提议并在投递之前同意。

如上所述，Zookeeper 保证消息的总序，并且他还保证提议的总序。Zookeeper 通过 zxid 暴露总序。所有的提议将会有一个 zxid 标记当它被提议并准确的反应总排序。提议被发送到所有 Zookeeper 服务并提交当法定人数承认提议的时候。如果提议包含一个信息，当提交提议时信息将被投递。确认的意思是服务器记录了这个提议道持久化存储。我们的法定人数有要求，任何一对法定人数必须只是有一个服务。我们通过要求所有法定人数的大小(n/2+1)确保，这里的 n 是组成 Zookeeper 服务的服务器数量。

zxid 有两个部分：时代和一个计数器。在我们的实现中 zxid 是一个 64 位的数字。我们使用高位 32 为记录时代，低位 32 位记录计数器。因为它有两个部分代表 zxid，一个数字和一堆证书，(epoch，count)。epoch 数字代表领导阶层的变化。每当一个新领导产生，它会获得自己的 epoch 数字。我们有一个简单的算法分配

唯一的 zxid：领导者简单的增加 zxid 去为每个提议获得唯一的 zxid。领导活动将保证只有一个领导使用指定的 epoch，所有我们简单的算法保证每个提议将由一个唯一 id。

Zookeeper 消息由两个阶段组成：

领袖激活

在这个阶段一个领袖建立系统的正确状态并准备开始提出建议。

活动消息

在这个阶段一个领袖接收信息到建议并协调消息投递。

Zookeeper 是一个全盘协议。我们不关注个人建议，而把协议流看作一个整体。我们严格的顺序可让我们有效的并大大简化我们的协议。领袖激活体现这个整体概念。一个领袖变为活动的只在法定追随者已经同步了领导者，他们有相同的状态。这个状态有所有的提议构成，领袖认为已经提交了并且提议追溯领袖,NEW_LEADER 提议。

8.2.2 领袖激活

领袖激活包括领导者选举。当前在 Zookeeper 我们有两个领导者选举算法：LeaderElection 和 FastLeaderElection(AuthFastLeaderElection 是 FastLeaderElection 的一个变种，它用 UDP 可让服务器之心给一个简单的授权形式避免 IP 欺骗)。Zookeeper 消息传递不关心选举的精确算法，只有以下的持有：

领导者见过所有追随着的最高 zxid。

- 服务器的法定人数已经提交到后续的领导者。
- 只有这个两个要求，第一，追随者中最高的 zxid 需要坚持正确的操作。第二个要求，追随者的法定人数只需要保持高概率。我们将重检查第二个要求，所以如果在领导者选举或之后发生故障且法定人数丢失，我们将通过放弃领袖激活恢复并运行另一个选举。

领导者选举之后将指定一个单独的服务作为领导者并开始等待追随者连接它。其余的服务将尝试连接到领导者。领导者将通过发送他们丢失的提议同步追随者，或者如果追随者丢失太多提议，它将发送一个完整的状态快照到追随者。

有一个极端状况，一个有提议的追随者，U，领导者没有看到。提议在顺序中看到，所有 U 的提议将由一个比领导者更大的 zkid。因为提交的提议一定要让服务器的法定人数看到，并且服务器法定人数选举领导者看不到 U，你的提议还没提交，所有他们可以被丢弃。当追随者连接的领导者，领导者会告诉追随者丢弃 U。

新的领导者建立一个 zxid 开始使用新的提议，e，最大的 zxid 看到并设置下一个 zxid 使用(e+1,0)，领导者同步追随者之后，它提出 NEW_LEADER 提议。一旦 NEW_LEADER 提议提交，领导者将激活并开始接受和发行提议。

这一切听起来复杂，但这是领导者激活过程中的基本规则：

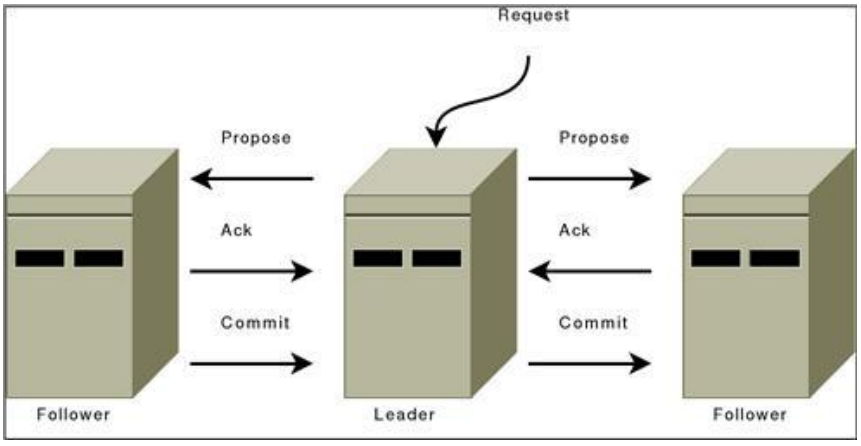
- 追随者确认 NEW_LEADER 提议在它同步 leader 之后。
- 追随者只从单个服务确认给定的 zxid 的 NEW_LEADER 提议。
- 新领导者将提交 NEW_LEADER 提议，当法定人数的追随者确认时。
- 追随者将提交从领导者接收到的任何状态，当 NEW_LEADER 提议提交时。
- 新领导者将不接收新提议，直到 NEW_LEADER 提议完成提交。

如果领导者选举异常终止，我们没有问题因为 NEW_LEADER 提议不会提交，因为领导者没有法定人数。当这个发生时，领导者和剩下的追随者会超时并回到领导者选举。

8.2.3 活动通知

领导者激活做了所有的重担。一旦领导者加冕它可以启动爆破提议。只要他保持领导者没有其他领导者可以出现，因为没有其他领导者能够获得追随者法定人数。如果新领导者出现，意味着领导者丢失了法定人数，并且新的领导者将清理任何混乱的东西。

Zookeeper 消息传递操作类似于一个经典的两阶段提交。



所有的通讯通道都是先进先出的，所有每件事都是有序的。特别的是下面的操作约束：

- 领导者使用相同的顺序发送提议所有的追随者。此外，这个顺序遵循请求接收的顺序。因为我们使用 FIFO 通道，意味着追随者也在顺序中接收提议。

- 追随者有序的处理他们接收到的消息。这意味着消息将在顺序中确认并且领导者将从追随者有序的接收确认，由于 FIFO 通道。它还意味着如果消息m已经写入到非易失的存储，所有在m消息之前消息也已经写入到非易失性的存储。
- 领导者将发行一个 COMMIT 到所有的追随者，一旦法定的追随者确认消息。因为消息在顺序中确认，COMMIT 将被领导者接收的顺序发送。
- COMMITs 被有序的处理。追随者投递一个提议消息当提议被提交时

8.2.4 总结

它为什么工作？特别的是，为什么提议被新领导者赞成总数包含任何已经提交的提议？首先，所有提议有一个唯一 zxid，所有不像其他的协议，我们从没必要担心出现两个相同的 zxid；追随者有序的看到并记录提议；提议有序的提交；在一个时间点只有一个活动的领导者因为在特定时间追随者只追随单独的领导者；新领导者从之前的 epoch 看到所有提交的提议因为它从法定数量的服务已经看到最高的 zxid；上一个时代任何没有提交的提议将由新领导者提交在它活动之前。

8.2.5 比较

这不仅仅是 Multi-Paxos 吗？不是，Multi-Paxos 需要某种程度上的保证，只有一个单独的协调者。我们不指望这样的保证。反而我们使用领导者激活去恢复。

这不只是 Paxos 吗？你的活动通知阶段看上去仅仅是 Paxos 的第二阶段?实际上，到我们的活动消息看起来像两阶段提交而不需要处理终止。活动通知从某种意义上看是不同的。如果我们不维持严格的数据包 FIFO 顺序，这一切面临崩溃。另外，我们的领导者激活阶段从这方面看也是不同的。特别的，我们使用的时期可让我们跳过没提交的提议块并不关心给定 zxid 的重复提议。

8.3 法定人数

原子性广播和领导者选举使用法定人数的概念保证系统的一致性视图。默认的，Zookeeper 使用大多数的法定人数，这意味着每个协议投票的发生要求多数人投票。一个实例是确认领导者提议：领导者值可以提交一旦接收到法定人数的服务确认。

8.4 日志

Zookeeper 使用 slf4j 作为日志抽象层。版本 1.2 开始选择 log4j 作为最终的日志实现。为了更好的嵌入支持，计划在将来放弃决定选择，最终的日志实现交给用户。因此，在代码里总是使用 slf4j api 写日志生命，而不是在运行是配置 log4j。注意 slf4j 没有 FATAL 级别，前面的消息在 FATAL 级别已经移动到 ERROR 级别。Zookeeper 的 log4j 配置信息，查看 Zookeeper 管理员指南的日志章节。

8.4.1 开发者指引

请跟随 slf4j 手册当在代码中建立日志声明时。还要阅读 FAQ on perfomance，创建日志声明时。补丁审核员将寻找下面的：

正确级别的日志

slf4j 日志有若干个级别。正确的选择非常重要。为了更好的降低严重程度：

- ERROR 级别指定可让程序仍然继续运行的错误事件
- WARN 级别表示潜在的有害情况
- INFO 级别表示报告的消息，强调在粗粒度级别应用程序的进展。
- DEBUG 级别表示细粒度的报告事件，有利于调试应用。
- TRACE 级别表示比 DEBUG 更细粒度的报告事件。
- Zookeeper 运行在生产环境通常使用 INFO 级别或更高级别的日志消息输出到日志。

使用标准的 slf4j 风格

静态消息日志

```
LOG.debug("process completed successfully!");
```

需要创建参数化的消息时，使用格式化锚。

```
LOG.debug("got {} messages in {} minutes",new Object[]{count,time});
```

命名

记录器应该在使用它们的类后命名。

```
public class Foo {

   private static final Logger LOG = LoggerFactory.getLogger(Foo.class);

   ....

   public Foo() {

      LOG.info("constructing Foo");
```

异常处理

```
try {
```

```
 // code
} catch (XYZException e) {
 // do this
 LOG.error("Something bad happened", e);
 // don't do this (generally)
 // LOG.error(e);
 // why? because "don't do" case hides the stack trace
 // continue process here as you need... recover or (re)throw
}
```