



Apache Zookeeper 3.4 中文教程

Collect Author: boonya Date:2017-04-10

<http://blog.csdn.net/boonya>

Apache Zookeeper: <http://zookeeper.apache.org/doc/trunk/index.html>

目录

1	概述.....	4
1.1	欢迎: 协调分布式系统 ZooKeeper	5
1.1.1	ZooKeeper 概述.....	5
1.1.2	开发商.....	5
1.1.3	管理员和运营商.....	5
1.1.4	贡献者.....	5
1.1.5	其他 ZooKeeper 文档.....	6
1.1.6	BookKeeper 文档.....	6

1.2	概述: ZooKeeper 分布式应用程序的分布式协调服务	6
1.3	入门: 使用 ZooKeeper 协调分布式应用程序	13
1.3.1	先决条件	13
1.3.2	下载	13
1.3.3	独立操作	13
1.3.4	管理 ZooKeeper 存储	14
1.3.5	连接到 ZooKeeper	14
1.3.6	编程到 ZooKeeper	17
1.3.7	运行复制 ZooKeeper	17
1.3.8	其他优化	18
2	开发者	18
2.1	API 文档	18
2.2	程序员开发使用 ZooKeeper 的分布式应用程序	18
2.2.1	介绍	19
2.2.2	ZooKeeper 数据模型	20
2.2.3	ZooKeeper 会话	22
2.2.4	ZooKeeper 观察者 (Watcher)	25
2.2.5	使用 ACL 的 ZooKeeper 访问控制	26
2.2.6	可插拔 ZooKeeper 认证	30
2.2.7	一致性保证	32
2.2.8	绑定	33
2.2.9	构建块: ZooKeeper 操作指南	35
2.2.10	程序结构, 简单示例	36
2.2.11	Gotchas: 常见问题和故障排除	36
2.3	ZooKeeper Java 示例	37
2.3.1	一个简单的 Watcher 客户端	38
2.3.2	执行者类	38
2.3.3	DataMonitor 类	41
2.3.4	完整的来源列表	44
2.4	障碍和队列: 使用 ZooKeeper 编程 - 一个基础教程	50
2.4.1	介绍	51
2.4.2	障碍	52
2.4.3	生产者 - 消费者队列	54
2.4.4	完整的源代码清单	57
2.5	ZooKeeper 方法和解决方案	64
2.5.1	使用 ZooKeeper 创建高级构造的指南	64
3	BookKeeper 入门指南	69
3.1	入门: 设置 BookKeeper 来编写日志。	69
3.1.1	先决条件	69
3.1.2	下载	69
3.1.3	LocalBookKeeper	70
3.1.4	建立书店	70
3.1.5	设置 ZooKeeper	70
3.1.6	例	70

3.2	BookKeeper 概述.....	71
3.2.1	BookKeeper 介绍.....	71
3.2.2	稍微详细一点.....	72
3.2.3	簿记员元素和概念.....	72
3.2.4	簿记员初步设计.....	73
3.2.5	簿记员元数据管理.....	74
3.2.6	结算分类帐.....	75
3.3	BookKeeper 管理员指南.....	75
3.3.1	部署.....	75
3.3.2	系统要求.....	76
3.3.3	运行的书.....	76
3.3.4	ZooKeeper 元数据.....	76
3.4	BookKeeper 程序编程入门.....	76
3.4.1	实例化 BookKeeper。.....	77
3.4.2	创建分类帐.....	77
3.4.3	将条目添加到分类帐。.....	78
3.4.4	关闭分类帐.....	79
3.4.5	打开分类帐.....	79
3.4.6	从分类账中读.....	80
3.4.7	删除分类帐.....	81
4	管理和部署.....	81
4.1	ZooKeeper 管理员指南：部署和管理指南.....	81
4.1.1	部署.....	82
4.1.2	管理.....	85
4.2	ZooKeeper 配额指南：部署和管理指南.....	100
4.2.1	配额.....	100
4.3	ZooKeeper JMX.....	101
4.3.1	JMX.....	101
4.3.2	启动使用 JMX 的 ZooKeeper.....	101
4.3.3	运行 JMX 控制台.....	101
4.3.4	ZooKeeper MBean 参考.....	102
4.4	Zookeeper 观察员.....	102
4.4.1	观察者：缩放 ZooKeeper 而不伤害写入性能.....	103
4.4.2	如何使用观察者.....	103
4.4.3	示例用例.....	104
5	贡献者：Zookeeper 内部.....	104
5.1	介绍.....	104
5.2	原子广播.....	104
5.2.1	担保，财产和定义.....	104
5.2.2	领导激活.....	106
5.2.3	主动消息.....	107
5.2.4	概要.....	108
5.2.5	比较.....	108
5.3	法定人数.....	108

5.4	记录.....	109
5.4.1	开发者指南.....	109
6	常见问题: FAQ.....	110
6.1	ZooKeeper 的状态转换是什么?	111
6.2	如何处理 CONNECTION_LOSS 错误?	111
6.3	我应该如何处理 SESSION_EXPIRED?	111
6.4	是否有一个简单的方法来过期测试?	112
6.5	为什么 NodeChildrenChanged 和 NodeDataChanged 观察事件不会返回有关更改的更多信息?	112
6.6	有什么选择 - 升级 ZooKeeper 的过程?	113
6.7	如何调整 ZooKeeper 系列(群集)?	113
6.8	我可以在负载均衡器后面运行一个集群集群吗?	114
6.9	群集关闭时, ZK 会话会发生什么?	115
7	补充内容.....	115
7.1	基于 Zookeeper 的服务注册与发现架构.....	115
7.1.1	服务提供者.....	115
7.1.2	服务注册中心.....	116
7.1.3	服务消费者.....	116
7.1.4	服务注册发现方案.....	116
7.2	DUBBO 分布式服务架构与 Zookeeper 实现服务提供和消费	118
7.2.1	背景.....	118
7.2.2	需求.....	119
7.2.3	架构.....	120
7.2.4	资源.....	122
7.2.5	文档.....	122
7.2.6	在 SpringMVC 框架中实现服务发布和消费	123
7.3	使用 RMI + ZooKeeper 实现远程调用框架.....	130
7.3.1	发布 RMI 服务.....	130
7.3.2	调用 RMI 服务.....	133
7.3.3	RMI 服务的局限性.....	133
7.3.4	使用 ZooKeeper 提供高可用的 RMI 服务	134
7.4	Zookeeper 编程客户端.....	143

1 概述

1.1 欢迎：协调分布式系统 ZooKeeper

ZooKeeper 是分布式应用程序的高性能协调服务。 它在一个简单的界面中公开了诸如[命名](#)，[配置管理](#)，[同步和组服务等常用服务](#)，因此您不必从头开始编写它们。 您可以使用现成的方式来实现共识，团体管理，领导选举和存在协议。 您可以为自己的具体需求搭建它。

以下文档介绍了使用 ZooKeeper 开始使用的概念和步骤。 如果您有更多的问题，请询问[邮件列表](#)或浏览档案。

1.1.1 ZooKeeper 概述

客户开发人员，管理员和贡献者的技术概述文件

- [概述](#) - ZooKeeper 的鸟瞰图，包括设计理念和架构
- [入门](#) - 开发人员为 ZooKeeper 安装，运行和编程的教程式指南
- [发行说明](#) - 新的开发人员和面向用户的功能，改进和不兼容性

1.1.2 开发商

使用 ZooKeeper Client API 的开发人员文档

- [API 文档](#) - 对 ZooKeeper Client API 的技术参考
- [程序员指南](#) - [ZooKeeper](#) 的客户端应用程序开发人员指南
- [ZooKeeper Java 示例](#) - 一个简单的 Zookeeper 客户端应用程序，用 Java 编写
- [障碍和队列教程](#) - [屏障和队列](#)的示例实现
- [ZooKeeper 食谱](#) - 分布式应用程序中常见问题的更高级解决方案

1.1.3 管理员和运营商

ZooKeeper 部署的管理员和操作工程师的文档

- [管理员指南](#) - 系统管理员和任何可能部署 ZooKeeper 的[人的指南](#)
- [配额指南](#) - ZooKeeper 配额系统管理员指南。
- [JMX](#) - 如何在 ZooKeeper 中启用 JMX
- [分层法定人数](#)
- [观察员](#) - 无投票的集体成员，轻松提高 ZooKeeper 的可扩展性

1.1.4 贡献者

为 ZooKeeper 开源项目贡献的开发人员文档

- [ZooKeeper 内部](#) - 关于 ZooKeeper 内部工作的各种主题

1.1.5 其他 ZooKeeper 文档

- [维基](#)
- [常问问题](#)

1.1.6 BookKeeper 文档

BookKeeper 是一个高可用系统，可实现高性能预写日志记录。它使用 ZooKeeper 作为元数据，这是 ZooKeeper contrib 的主要原因。

- [henn，又是什么](#)
- [好的，现在我该怎么做呢](#)
- [真棒，但如何将其与我的应用程序集成？](#)
- [我可以流字节而不是条目吗？](#)

1.2 概述：ZooKeeper 分布式应用程序的分布式协调服务

ZooKeeper

- [ZooKeeper：分布式应用程序的分布式协调服务](#)
 - [设计目标](#)
 - [数据模型和层次命名空间](#)
 - [节点和短暂节点](#)
 - [有条件的更新和观察者\(Watcher\)](#)
 - [保证](#)
 - [简单的 API](#)
 - [实施](#)
 - [用途](#)
 - [性能](#)
 - [可靠性](#)
 - [ZooKeeper 项目](#)

ZooKeeper 是一种用于分布式应用程序的分布式，开源协调服务。它暴露了一组简单的原语，分布式应用程序可以基于实现更高级别的服务进行同步，配置维护以及组和命名。它被设计为易于编程，并使用在文件系统熟悉的目录树结构之后设计的数据模型。它运行在 Java 中，并且具有 Java 和 C 的绑定。

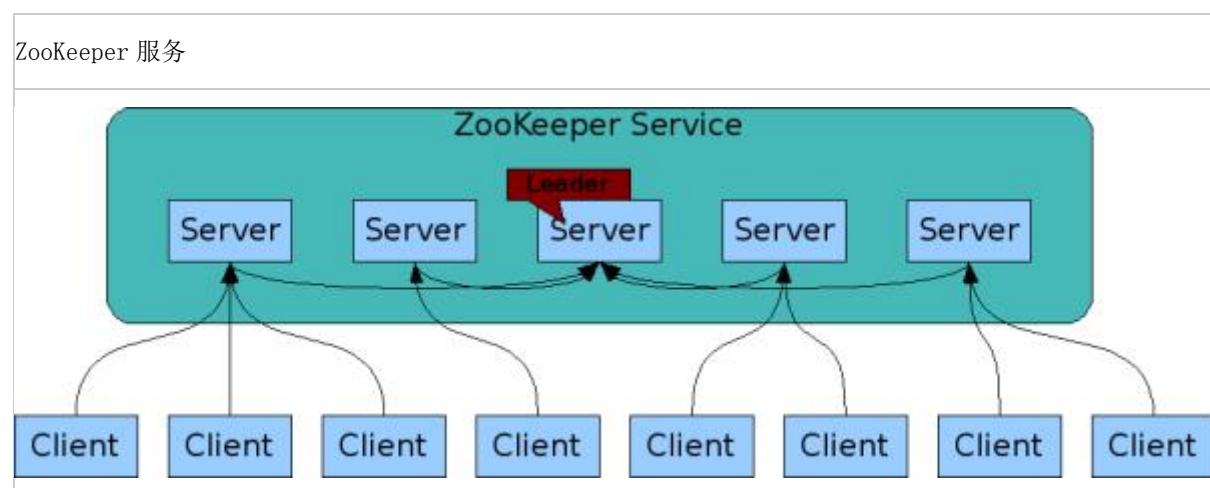
协调服务是非常难以正确的。他们特别容易出现种族条件和僵局等错误。ZooKeeper 背后的动机是缓解分布式应用程序从头开始执行协调服务的责任。

1.2.1.1 设计目标

ZooKeeper 很简单 ZooKeeper 允许分布式进程通过与标准文件系统类似的共享分层命名空间相互协调。名称空间由 ZooKeeper 语法中的数据寄存器（称为 znodes）组成，这些类似于文件和目录。与专为存储设计的典型文件系统不同，ZooKeeper 数据保存在内存中，这意味着 ZooKeeper 可以实现高吞吐量和低延迟数。

ZooKeeper 实现高性能，高可用性，严格有序的访问非常重要。ZooKeeper 的性能方面意味着它可以在大型分布式系统中使用。可靠性方面使其不会成为单点故障。严格的排序意味着复杂的同步原语可以在客户端实现。

ZooKeeper 被复制。 像它所协调的分布式进程一样，ZooKeeper 本身也是被复制到一组称为合奏的主机上。



构成 ZooKeeper 服务的服务器必须彼此了解。他们维护状态的内存映像以及持久存储中的事务日志和快照。只要大多数服务器可用，ZooKeeper 服务将可用。

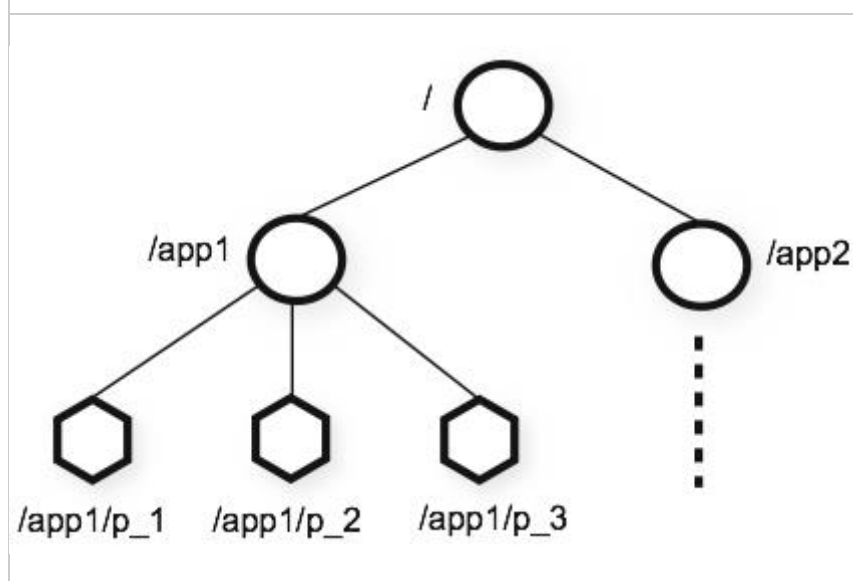
客户端连接到一个 ZooKeeper 服务器。客户端维护 TCP 连接，通过它发送请求，获取响应，获取 Watcher 事件并发送心跳。如果到服务器的 TCP 连接中断，客户端将连接到不同的服务器。

ZooKeeper 订购。 ZooKeeper 将每个更新标记为反映所有 ZooKeeper 事务顺序的数字。后续操作可以使用该命令来实现更高级的抽象，例如同步原语。

ZooKeeper 很快。 它在“读主导”工作负载中特别快。ZooKeeper 应用程序在数千台机器上运行，并且在写入次数比写入次数多 10 到 1 的情况下，性能最好。

1.2.1.2 数据模型和层次命名空间

ZooKeeper 提供的名称空间与标准文件系统类似。名称是以斜杠（/）分隔的路径元素序列。ZooKeeper 的名称空间中的每个节点都由路径标识。



1.2.1.3 节点和短暂节点

与标准文件系统不同，ZooKeeper 命名空间中的每个节点都可以与其相关联的数据以及子节点。就像有一个允许文件也是目录的文件系统。（ZooKeeper 设计用于存储协调数据：状态信息，配置，位置信息等，因此存储在每个节点上的数据通常很小，字节到千字节范围）。我们使用术语 *znode* 来表明我们正在谈论 ZooKeeper 数据节点。

Znodes 维护统计结构，其中包括数据更改，ACL 更改和时间戳的版本号，以允许缓存验证和协调更新。每次 znode 的数据发生变化时，版本号都会增加。例如，每当客户端检索数据时，它也会收到数据的版本。

存储在命名空间中的每个 znode 处的数据以原子方式读取和写入。读取获取与 znode 相关联的所有数据字节，写入替换所有数据。每个节点都有一个访问控制列表（ACL），它限制谁能做什么。

ZooKeeper 还具有短暂节点的概念。只要创建 znode 的会话处于活动状态，就会存在这些 znodes。当会话结束时，znode 被删除。当您想实现 [TBD] 时，临时节点很有用。

1.2.1.4 有条件的更新和观察者 (Watcher)

ZooKeeper 支持 *观察者 (Watcher)* 的概念。客户端可以在 znode 上设置观察者 (Watcher)。当 znode 更改时，观察者 (Watcher) 将被触发并移除。当观察者 (Watcher) 触发时，客户端接收到一个数据包，说明 znode 已经改变了。并且如果客户端与 Zoo Keeper 服务器之一的连接断开，客户端将收到本地通知。这些可以用于 [TBD]。

1.2.1.5 保证

ZooKeeper 非常快，非常简单。 由于其目标是建设更为复杂的服务，如同步化的基础，它提供了一套保证。 这些是：

- 顺序一致性 - 客户端的更新将按照发送的顺序进行应用。
- 原子性 - 更新成功或失败。 没有部分结果。
- 单一系统映像 - 客户端将看到与服务器连接的服务器相同的服务视图。
- 可靠性 - 一旦应用了更新，它将从当前持续到客户端覆盖更新。
- 及时性 - 系统的客户端视图在一定时间内保证是最新的。

有关这些的更多信息以及如何使用它们，请参阅 *[tbd]*

1.2.1.6 简单的 API

ZooKeeper 的设计目标之一是提供一个非常简单的编程接口。 因此，它仅支持以下操作：

创建

在树中的某个位置创建一个节点

删除

删除一个节点

存在

测试节点是否存在于某个位置

获取数据

从节点读取数据

设置数据

将数据写入节点

得到孩子

检索节点的子节点列表

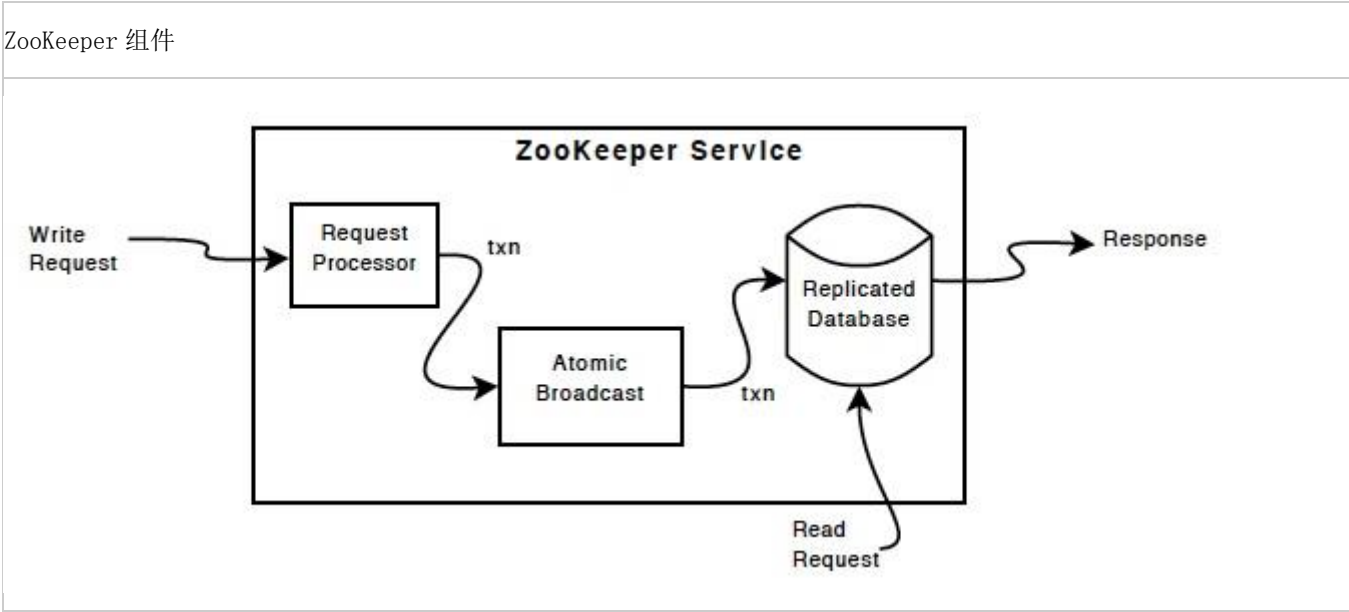
同步

等待数据传播

有关这些更深入的讨论，以及如何使用它们来实现更高层次的操作，请参考 *[tbd]*

1.2.1.7 实施

ZooKeeper 组件显示 ZooKeeper 服务的高级组件。 除了请求处理器之外，组成 ZooKeeper 服务的每个服务器都会复制其每个组件的自己的副本。



复制数据库是包含整个数据树的内存数据库。 将更新记录到磁盘以获取可恢复性，并将写入序列化到磁盘，然后才能应用到内存数据库。

每个 ZooKeeper 服务器都为客户端服务。 客户端连接到一个服务器以提交 irequests。 从每个服务器数据库的本地副本服务器读取请求。 更改服务状态，写入请求的请求由协议协议进行处理。

作为协议协议的一部分，客户端的所有写入请求都将转发到单个服务器，称为**主管**。 称为**关注者**的其他 ZooKeeper 服务器从领导者接收消息提议，并同意消息传递。 消息传递层负责替代领导者的失败，并与领导者同步追随者。

ZooKeeper 使用自定义的原子消息协议。 由于消息层是原子的，所以 ZooKeeper 可以保证本地副本不会发散。 当领导者收到写请求时，它会计算要应用写入时系统的状态，并将其转换为捕获此新状态的事务。

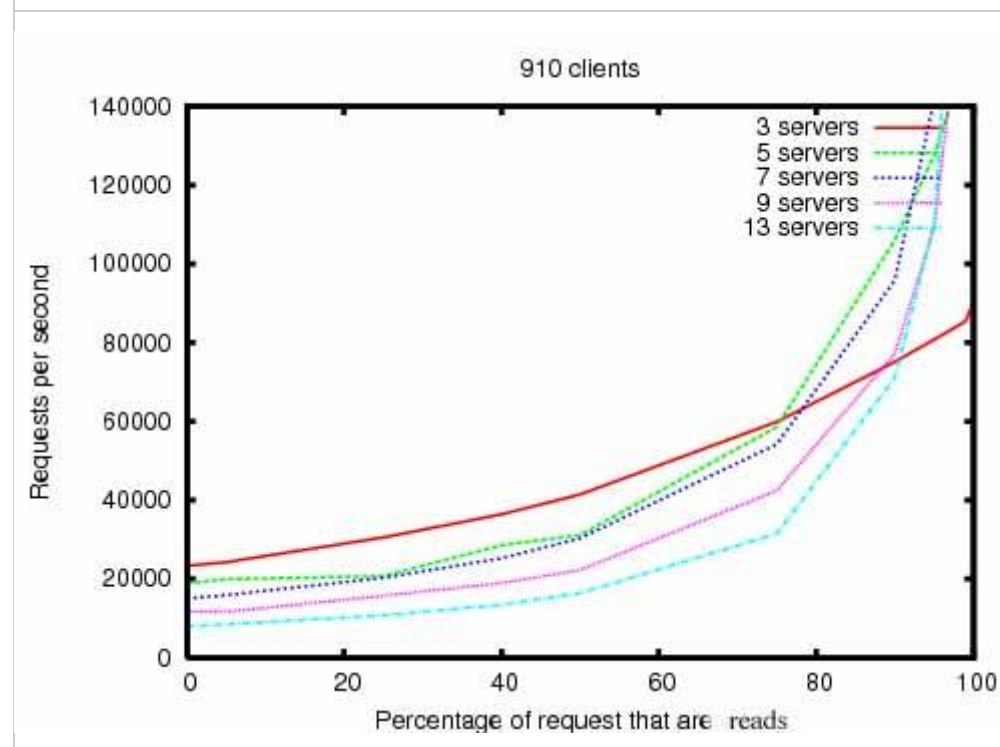
1.2.1.8 用途

ZooKeeper 的编程界面故意简单。 然而，您可以实现更高阶的操作，例如同步原语，组成员资格，所有权等。一些分布式应用程序已使用它： [tbd : 从白皮书和视频演示中添加用途。) 有关详细信息，请参阅 [tbd]

1.2.1.9 性能

ZooKeeper 的设计是高性能的。 但是呢 Yahoo! Research 的 ZooKeeper 开发团队的结果表明它是。 （请参见 [ZooKeeper 吞吐量作为读写比率的变化](#) 。）读取超出写入的应用程序中性能特别高，因为写入涉及同步所有服务器的状态。 （协调服务通常情况下读取数量超过写入次数。）

ZooKeeper 吞吐量随读写速率的变化而变化



图形 [ZooKeeper 吞吐量作为读写比率变化](#) 是 ZooKeeper 3.2 版本在具有双 2Ghz 至强和两个 SATA 15K RPM 驱动器的服务器上运行的吞吐量图。 一个驱动器用作专用的 ZooKeeper 日志设备。 快照已写入 OS 驱动器。 写入请求是 1K 写入，读取是 1K 读取。 “服务器”表示 ZooKeeper 系统的大小，构成服务的服务器数量。 大约使用其他 30 台服务器来模拟客户端。 ZooKeeper 系列被配置为使领导者不允许与客户端的连接。

注意

在 3.2 版本中，r / w 性能比[以前的 3.1 版本](#)提高了约 2 倍。

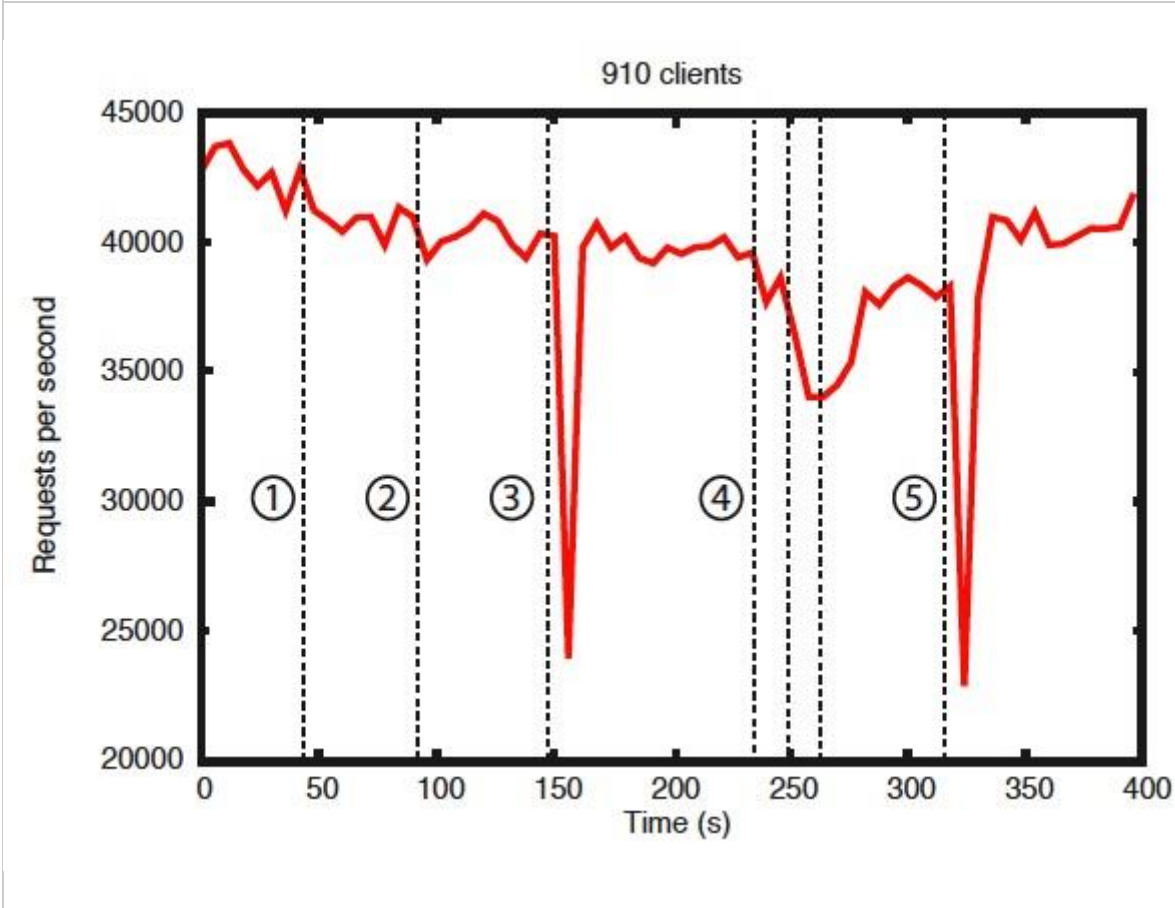
基准也表明它也是可靠的。 [出现错误的可靠性](#)显示部署如何响应各种故障。 图中标有的事件如下：

1. 追随者的失败和恢复
2. 失败和恢复不同的追随者
3. 领导失败
4. 失败和恢复的两个追随者
5. 另一位领导失败

1.2.1.10 可靠性

为了显示系统随着时间的推移，注意到我们运行了一个由 7 台机器组成的 ZooKeeper 服务。 我们执行了与以前相同的饱和基准，但这次我们将写入百分比保持在 30%，这是我们预期工作负荷的保守比例。

出现错误的可靠性



这是这个图表的一些重要观察。 首先，如果追随者失败并迅速恢复，那么 ZooKeeper 尽管发生故障仍然能够维持高吞吐量。 但也许更重要的是，领导选举算法可以使系统恢复得足够快，以防止吞吐量大幅下降。 在我们的观察中，ZooKeeper 需要不到 200ms 才能选出新的领导者。 第三，随着追随者的恢复，ZooKeeper 能够在开始处理请求后再次提高吞吐量。

1.2.1.11 ZooKeeper 项目

ZooKeeper 已经成功应用于许多工业应用。 它在 Yahoo! 中用作 Yahoo! Message Broker 的协调和故障恢复服务，Yahoo! Message Broker 是一个高度可扩展的发布订阅系统，用于管理数千个用于复制和数据传送的主题。 它由用于 Yahoo! crawler 的获取服务使用，它还管理故障恢复。 一些 Yahoo! 广告系统也使用 ZooKeeper 来实现可靠的服务。

鼓励所有用户和开发人员加入社区并贡献他们的专长。 有关更多信息，请参阅 [Apache 上的 Zookeeper 项目](#) 。

1.3 入门：使用 ZooKeeper 协调分布式应用程序

ZooKeeper 入门指南

- [入门：使用 ZooKeeper 协调分布式应用程序](#)
 - [先决条件](#)
 - [下载](#)
 - [独立操作](#)
 - [管理 ZooKeeper 存储](#)
 - [连接到 ZooKeeper](#)
 - [编程到 ZooKeeper](#)
 - [运行复制 ZooKeeper](#)
 - [其他优化](#)

本文档包含有关使用 ZooKeeper 快速入门的信息。 它主要针对开发人员希望尝试，并且包含单个 ZooKeeper 服务器的简单安装说明，一些验证它正在运行的命令以及一个简单的编程示例。 最后，为了方便起见，有一些关于更复杂的安装的部分，例如运行复制的部署以及优化事务日志。 但是，有关商业部署的完整说明，请参阅 “ [ZooKeeper 管理员指南](#) ” 。

1.3.1 先决条件

请参阅“管理指南”中的 “ [系统要求](#) ”。

1.3.2 下载

要获得 ZooKeeper 发行版，请从其中一个 Apache 下载镜像下载最新的[稳定版本](#)。

1.3.3 独立操作

以独立模式设置 ZooKeeper 服务器是直接的。 服务器包含在单个 JAR 文件中，因此安装包括创建配置。

一旦你下载了一个稳定的 ZooKeeper release 解压缩它和 cd 到根

要启动 ZooKeeper，您需要一个配置文件。 这里是一个示例，在 `conf / zoo.cfg` 中创建它：

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

这个文件可以被称为任何东西，但是为了讨论这个问题，它调用它 `conf / zoo.cfg`。 更改 `dataDir` 的值以指定一个现有的（空的开始）目录。 以下是每个字段的含义：

`tickTime`

ZooKeeper 使用的基本时间单位（毫秒）。 它用于做心跳，最小会话超时将是 `tickTime` 的两倍。

`dataDir`

存储内存数据库快照的位置，除非另有规定，更新数据库的事务日志。

`clientPort`

端口监听客户端连接

现在您创建了配置文件，您可以启动 ZooKeeper：

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

ZooKeeper 使用 log4j 记录消息 - “程序员指南”的“[记录](#)”部分提供了更多详细信息。 根据 log4j 配置，您将看到进入控制台的日志消息（默认）和/或日志文件。

这里概述的步骤运行在独立模式下的 ZooKeeper。 没有复制，所以如果 ZooKeeper 进程失败，服务将会关闭。 这对于大多数开发环境都很好，但是要在复制模式下[运行 ZooKeeper](#)，请参阅[运行复制的 ZooKeeper](#)。

1.3.4 管理 ZooKeeper 存储

对于长时间运行的生产系统，ZooKeeper 存储必须从外部进行管理（`dataDir` 和日志）。 有关详细信息，请参阅[维护](#)部分。

1.3.5 连接到 ZooKeeper

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

这使您可以执行简单的类文件操作。

连接完毕后，您应该会看到：

```
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 中，键入 `help` 以获取可以从客户端执行的命令列表，如下所示：

```
[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
    create path data acl
    stat path [watch]
    listquota path
    history
    setAcl path acl
    getAcl path
    sync path
    redo cmdno
    addauth scheme auth
    delete path [version]
    deleteall path
    setquota -n|-b val path
```

从这里，您可以尝试一些简单的命令来了解这个简单的命令行界面。 首先，通过发出 `list` 命令，如在 `ls` 中，产生：

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

接下来，通过运行 `create / zk_test my_data` 来创建一个新的 `znode`。 这将创建一个新的 `znode` 并将字符串“`my_data`”与节点相关联。 你应该看到：

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

发出另一个 `ls /` 命令来查看目录的样子：

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

请注意，`zk_test` 目录已经创建。

接下来，通过运行 `get` 命令验证数据是否与 `znode` 相关联，如：

```
[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
```

我们可以通过发出 set 命令来更改与 zk_test 关联的数据，如：

```
[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，我们在设置数据后做了一个确定，确实发生了变化。

最后，我们通过下列方式删除节点：

```
[zkshell: 16] delete /zk_test
```



```
[zkshell: 17] ls /  
[zookeeper]  
[zkshell: 18]
```

就是这样。 要了解更多信息，请继续阅读本文档的其余部分，并参阅“[程序员指南](#)”。

1.3.6 编程到 ZooKeeper

ZooKeeper 具有 Java 绑定和 C 绑定。 它们在功能上相当。 C 绑定存在两种变体：单线程和多线程。 这些不同之处仅在于消息循环如何完成。 有关更多信息，请参阅“[ZooKeeper 程序员指南](#)”中的[编程示例](#)，了解使用不同 API 的示例代码。

1.3.7 运行复制 ZooKeeper

以独立模式运行 ZooKeeper，便于评估，开发和测试。 但在生产中，您应该以复制模式运行 ZooKeeper。 同一应用程序中的复制服务器组称为[仲裁](#)，而在复制模式下，仲裁中的所有服务器都具有相同配置文件的副本。

注意

对于复制模式，至少需要三台服务器，强烈建议您拥有奇数个服务器。 如果您只有两台服务器，那么您处于这样的情况：如果其中一台服务器出现故障，则没有足够的机器来形成大多数法定人数。 两台服务器本身的稳定性比单个服务器稳定，因为有两个单点故障。

复制模式所需的 `conf / zoo.cfg` 文件类似于独立模式下使用的文件，但有一些区别。 这是一个例子：

```
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
```

新条目 `initLimit` 是超时 ZooKeeper 用于限制 ZooKeeper 服务器在法定人群中连接到领导者的时间长度。 条目 `syncLimit` 限制服务器可以从领导者多远的距离。

使用这两个超时时间，您可以使用 `tickTime` 指定时间单位。 在这个例子中，`initLimit` 的超时时间为 2000 刻度，或 10 秒钟。

表单 `server.X` 的条目列出构成 ZooKeeper 服务的服务器。 当服务器启动时，它通过查找数据目录中的文件 `myid` 来知道它是哪个服务器。 该文件包含服务器编号，以 ASCII 格式显示。

最后，请注意每个服务器名称后面的两个端口号：“2888”和“3888”。对等体使用前端口连接到其他对等体。这样的连接是必要的，使得对等体可以进行通信，例如，以商定更新的顺序。更具体地说，一个 ZooKeeper 服务器使用这个端口来连接追随者到领导者。当新的领导者出现时，追随者使用此端口打开与领导者的 TCP 连接。因为默认领导选举也使用 TCP，所以我们目前需要另外一个端口进行领导选举。这是服务器条目中的第二个端口。

注意

如果要在单个计算机上测试多个服务器，请将服务器名称指定为 localhost，在服务器的每个服务器上指定唯一的仲裁和前导选举端口（例如上述示例中的 2888: 3888, 2889: 3889, 2890 : 3890）配置文件。当然，单独的 *dataDir* 和不同的 *clientPort* 也是必需的（在上面的复制示例中，在单个本地主机上运行，您仍然有三个配置文件）。

请注意，在单个机器上设置多个服务器将不会产生任何冗余。如果发生导致机器死机的事情，所有的 zookeeper 服务器都将处于脱机状态。完全冗余要求每个服务器都有自己的机器。它必须是完全独立的物理服务器。同一物理主机上的多个虚拟机仍然容易受到该主机完全失败的影响。

1.3.8 其他优化

有几个其他配置参数可以大大提高性能：

- 要获得更新的低延迟，重要的是拥有一个专用的事务日志目录。默认情况下，事务日志将放在与数据快照和 *myid* 文件相同的目录中。 *dataLogDir* 参数指示用于事务日志的不同目录。
- *[tbd: 其他配置参数是什么?]*

2 开发者

2.1 API 文档

<http://zookeeper.apache.org/doc/current/api/index.html>

2.2 程序员开发使用 ZooKeeper 的分布式应用程序

- [介绍](#)
- [ZooKeeper 数据模型](#)
 - [ZNodes](#)
 - [观察者\(Watcher\)](#)
 - [数据访问](#)
 - [短暂节点](#)
 - [序列节点 - 唯一命名](#)
 - [时间在 ZooKeeper](#)

- [ZooKeeper 统计结构](#)
- [ZooKeeper 会话](#)
- [ZooKeeper 观察者\(Watcher\)](#)
- [观察者\(Watcher\) 语义](#)
- [什么 ZooKeeper 保证关于观察者\(Watcher\)](#)
- [关于观察者\(Watcher\) 的事情](#)
- [使用 ACL 的 ZooKeeper 访问控制](#)
- [ACL 权限](#)
 - [内置 ACL 方案](#)
 - [ZooKeeper C 客户端 API](#)
- [可插拔 ZooKeeper 认证](#)
- [一致性保证](#)
- [绑定](#)
- [Java 绑定](#)
- [C 绑定](#)
 - [安装](#)
 - [建立您自己的 C 客户端](#)
- [构建块：ZooKeeper 操作指南](#)
- [处理错误](#)
- [连接到 ZooKeeper](#)
- [阅读操作](#)
- [写作操作](#)
- [处理观察者\(Watcher\)](#)
- [小型 Zookeeper 行动](#)
- [程序结构，简单示例](#)
- [Gotchas：常见问题和故障排除](#)

2.2.1 介绍

本文档是开发者希望创建利用 ZooKeeper 协调服务的分布式应用程序的指南。 它包含概念和实用信息。

本指南的前四部分介绍了各种 ZooKeeper 概念的更高层次的讨论。 为了了解 ZooKeeper 如何工作以及如何使用它们，这些都是必需的。 它不包含源代码，但它确实会熟悉与分布式计算相关的问题。 第一组中的部分是：

- [ZooKeeper 数据模型](#)
- [ZooKeeper 会话](#)
- [ZooKeeper 观察者\(Watcher\)](#)
- [一致性保证](#)

接下来的四节提供实用的编程信息。 这些是：

- [构建块：ZooKeeper 操作指南](#)
- [绑定](#)

- [程序结构，简单示例](#) [tbd]
- [Gotchas: 常见问题和故障排除](#)

本书总结了[附录](#)，其中包含与其他有用的 ZooKeeper 相关信息的链接。

本文档中的大部分信息都是作为独立参考资料编写的。但是，在开始第一个 ZooKeeper 应用程序之前，您应该至少阅读 [ZooKeeper 数据模型](#)和 [ZooKeeper 基本操作上的章节](#)。此外，[简单编程示例](#) [tbd]有助于了解 ZooKeeper 客户端应用程序的基本结构。

2.2.2 ZooKeeper 数据模型

ZooKeeper 具有层次结构的名称空间，非常类似于分布式文件系统。唯一的区别是命名空间中的每个节点都可以拥有与其相关联的数据以及子节点。它就像有一个允许文件也是一个目录的文件系统。节点的路径总是表示为规范的，绝对的，斜线分隔的路径；没有相对参考。任何 unicode 字符都可以在路径中使用，具有以下约束：

- 空字符（\ u0000）不能是路径名的一部分。（这会导致 C 绑定的问题）
- 以下字符无法使用，因为它们显示不正确，或以令人困惑的方式呈现：\ u0001 - \ u0019 和 \ u007F - \ u009F。
- 不允许使用以下字符：\ ud800 -uF8FFF, \ uFFF0 - uFFFF。
- “。” 字符可以用作另一个名称的一部分，但是“。” 和“..” 不能单独用于指示沿路径的节点，因为 ZooKeeper 不使用相对路径。以下内容将无效：“/a/b/. /c” 或“/a/b/.. /c”。
- 令牌“zookeeper”被保留。

2.2.2.1 ZNodes

ZooKeeper 树中的每个节点都称为 *znode*。Znodes 维护一个统计结构，包括数据更改版本号，acl 更改。统计结构也有时间戳。版本号与时间戳一起允许 ZooKeeper 验证缓存并协调更新。每次 znode 的数据发生变化时，版本号都会增加。例如，每当客户端检索数据时，它也会收到数据的版本。当客户端执行更新或删除时，它必须提供正在更改的 znode 的数据版本。如果它提供的版本与数据的实际版本不匹配，则更新将失败。（可以覆盖此行为。有关详细信息，请参阅...） [tbd ...]

注意

在分布式应用程序工程中，单词 *节点* 可以指通用主机，服务器，组合成员，客户端进程等。在 ZooKeeper 文档中，*znodes* 是指数据节点。*服务器* 是指构成 ZooKeeper 服务的机器；*法定对等体* 是指构成集合的服务器；*客户端* 是指使用 ZooKeeper 服务的任何主机或进程。

znode 是程序员需要注意的主要抽象。Znode 有几个特点，值得一提。

2.2.2.1.1 观察者(Watcher)

客户端可以在 znode 上设置观察者(Watcher)。对 znode 的更改会触发观察者(Watcher)，然后清除观察者(Watcher)。当观察者(Watcher)触发时，ZooKeeper 会向客户端发送通知。有关观察者(Watcher)的更多信息，请参见“[ZooKeeper 观察者\(Watcher\)](#)”一节。

2.2.2.1.2 数据访问

存储在命名空间中的每个 znode 处的数据以原子方式读取和写入。读取获取与 znode 相关联的所有数据字节，写入替换所有数据。每个节点都有一个访问控制列表 (ACL)，它限制谁能做什么。

ZooKeeper 不是设计为一般数据库或大型对象存储。而是管理协调数据。该数据可以以配置，状态信息，会合等的形式出现。各种形式的协调数据的共同属性是它们相对较小：以千字节为单位。ZooKeeper 客户端和服务端实现具有完整性检查，以确保 znode 具有少于 1M 的数据，但数据应远远小于平均值。在相对较大的数据大小上运行将导致一些操作比其他操作花费更多的时间，并且会影响某些操作的延迟，因为在网络上移动更多数据到存储介质所需的额外时间。如果需要大量数据存储，处理此类数据的通常模式是将其存储在大容量存储系统（如 NFS 或 HDFS）上，并将指针存储到 ZooKeeper 中的存储位置。

2.2.2.1.3 短暂节点

ZooKeeper 还具有短暂节点的概念。只要创建 znode 的会话处于活动状态，就会存在这些 znodes。当会话结束时，znode 被删除。由于这种行为，短暂的 znodes 不允许有孩子。

2.2.2.1.4 序列节点 - 唯一命名

创建 znode 时，您也可以请求 ZooKeeper 在路径的末端附加单调增加的计数器。这个计数器是父 znode 唯一的。计数器的格式为 %010d - 这是 10 位，0（零）填充（计数器以这种方式进行格式化以简化排序），即“<path> 0000000001”。有关使用此功能的示例，请参阅[队列配方](#)。注意：用于存储下一个序列号的计数器是由父节点维护的有符号 int（4 字节），当递增超过 2147483647（导致名称“<path> -2147483647”）时，计数器将溢出。

2.2.2.2 时间在 ZooKeeper

ZooKeeper 追踪时间多种方式：

- **Zxid**
对 ZooKeeper 状态的每次更改都会以 *zxid*（ZooKeeper Transaction Id）的形式接收邮票。这显示了对 ZooKeeper 的所有更改的总排序。每个更改都将有一个唯一的 *zxid*，如果 *zxid1* 小于 *zxid2*，则 *zxid1* 发生在 *zxid2* 之前。
- **版本号**

对节点的每次更改将导致该节点的其中一个版本号的增加。 三个版本号是版本（znode 的数据更改数），cversion（znode 的子项更改次数）和 aversion（znode 的 ACL 更改次数）。

- **踢**

当使用多服务器 ZooKeeper 时，服务器使用 ticks 来定义事件的时间，例如状态上传，会话超时，对等体之间的连接超时等。勾选时间仅通过最小会话超时（2 次刻度时间）间接暴露； 如果客户端请求超过最小会话超时的会话超时，服务器将告诉客户端会话超时实际上是最小会话超时。

- **即时的**

除了在 znode 创建和 znode 修改之后将时间戳放入统计结构中，ZooKeeper 不使用实时或时钟时间。

2.2.2.3 ZooKeeper 统计结构

ZooKeeper 中每个 znode 的 Stat 结构由以下字段组成：

- **czxid**

导致此 znode 创建的更改的 zxid。

- **mzxid**

最后修改此 znode 的更改的 zxid。

- **pzxid**

最后修改 znode 子项的 zxid。

- **时间**

创建这个 znode 的时间（以毫秒为单位）。

- **mtime**

这个 znode 最后一次被修改的时间（以毫秒为单位）。

- **版**

这个 znode 数据的更改次数。

- **转换**

这个 znode 的孩子的更改次数。

- **厌恶**

该 znode 的 ACL 的更改次数。

- **短暂的老板**

如果 znode 是短暂节点，则该 znode 的所有者的会话 ID。 如果它不是短暂节点，它将为零。

- **dataLength**

该 znode 的数据字段的长度。

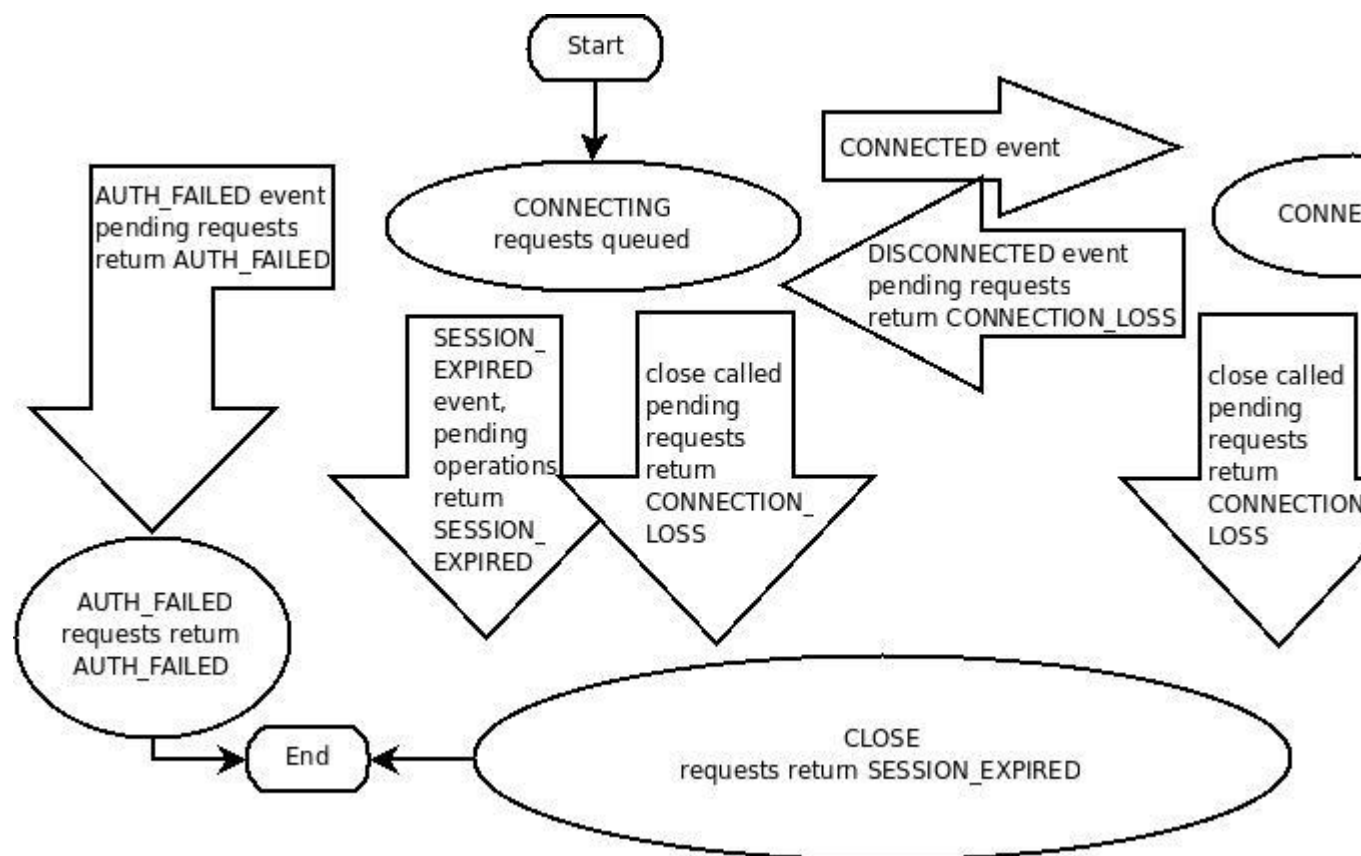
- **numChildren**

这个 znode 的孩子的数量。

2.2.3 ZooKeeper 会话

ZooKeeper 客户端通过使用语言绑定创建对服务的句柄来与 ZooKeeper 服务建立会话。 一旦创建，手柄开始处于 CONNECTING 状态，客户端库尝试连接到构成 ZooKeeper 服务的其中一个服务器，此时它切换到 CONNECTED 状态。 在正常运行期间将处于这两种状态之一。 如果出现不可恢复的

错误，例如会话过期或身份验证失败，或者如果应用程序显式关闭句柄，则句柄将移至 CLOSED 状态。 下图显示了 ZooKeeper 客户端可能的状态转换：



要创建客户端会话，应用程序代码必须提供连接字符串，其中包含逗号分隔的主机：端口对列表，每个对应于一个 ZooKeeper 服务器（例如“127.0.0.1:4545”或“127.0.0.1:3000, 127.0.0.1:3001, 127.0.0.1:3002”）。 ZooKeeper 客户端库将选择一个任意的服务器并尝试连接到它。 如果此连接失败，或者由于任何原因客户端与服务器断开连接，客户端将自动尝试列表中的下一个服务器，直到（重新）建立连接。

在 3.2.0 中添加：可选的“chroot”后缀也可以附加到连接字符串。 这将运行客户端命令，同时解释相对于此根的所有路径（类似于 unix chroot 命令）。 如果使用的话，这个例子看起来像：“127.0.0.1:4545/app/a”或“127.0.0.1:3000, 127.0.0.1:3001, 127.0.0.1:3002/app/a”，客户端将被植根于“/ app / a”，所有的路径都是相对于这个根目录，即获取/设置/ etc ... “/ foo / bar”将导致运行在“/ app / a / foo / bar”服务器的角度）。 此特性在多租户环境中特别有用，其中特定 ZooKeeper 服务的每个用户可以根植不同。 这使得重新使用更简单，因为每个用户可以将他/她的应用程序编码为“/”，而实际位置（说/ app / a）可以在部署时确定。

当客户端获取 ZooKeeper 服务的句柄时，ZooKeeper 将创建一个 ZooKeeper 会话，以 64 位数表示，它分配给客户端。 如果客户端连接到不同的 ZooKeeper 服务器，它将发送会话 ID 作为连接握手的一部分。 作为安全措施，服务器为任何 ZooKeeper 服务器可以验证的会话 ID 创建一个密码。当客户端建立会话时，将密码发送给具有会话 ID 的客户端。 当客户端重新建立与新服务器的会话时，客户端将使用会话 ID 发送此密码。

ZooKeeper 客户端库调用创建 ZooKeeper 会话的参数之一是会话超时（以毫秒为单位）。客户端发送请求的超时，服务器响应可以给客户端的超时。当前的实现需要超时时间为 tickTime（在服务器配置中设置）的最小值的 2 倍，最多为 tickTime 的 20 倍。ZooKeeper 客户端 API 允许访问协商超时。

当客户端（会话）从 ZK 服务集群分区时，它将开始搜索在会话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，会话将再次转换到“已连接”状态（如果在会话超时值内重新连接），或者它将转换到“过期”状态（如果会话超时后重新连接）。不建议在断开连接时创建一个新的会话对象（c 绑定中的一个新的 ZooKeeper.class 或 zookeeper 句柄）。ZK 客户端库将为您处理重新连接。特别是，我们将客户端库中的启发式内置到处理诸如“群体效应”之类的东西... 只有当通知会话到期（强制性）时才创建一个新的会话。

会话到期由 ZooKeeper 集群本身管理，而不是由客户端管理。当 ZK 客户端与群集建立会话时，它提供了上面详述的“超时”值。群集使用此值来确定客户端的会话何时过期。当集群在指定的会话超时时间内没有听到客户端（即没有心跳）时，会发生到期。在会话到期时，集群将删除该会话拥有的任何/所有短暂节点，并立即通知任何/所有连接的客户端的更改（任何人 Watcher 这些 znodes）。此时，过期会话的客户端仍然与集群断开连接，除非可以重新建立与集群的连接，否则将不会通知会话到期。客户端将保持断开状态，直到与集群重新建立 TCP 连接，届时，过期会话的观察者将收到“会话到期”通知。

过期会话的观察者所看到的过期会话的示例状态转换：

1. 'connected': 会话建立，客户端与群集通信（客户端/服务器通信正常运行）
2. 客户端从集群分区
3. 'disconnect': 客户端已经失去与集群的连接
4. 时间过去，在超时时间之后，集群会过期，客户端看不到与集群断开连接
5. 时间流逝，客户端恢复与集群的网络级连接
6. 'expired': 最终客户端重新连接到集群，然后通知到期

ZooKeeper 会话建立呼叫的另一个参数是默认观察者。在客户端发生任何状态更改时通知观察者。例如，如果客户端失去与服务器的连接，客户端将被通知，或者客户端的会话过期等等。这个观察者应该考虑初始状态被断开（即在任何状态改变事件被发送到观察者之前客户端 lib）。在新连接的情况下，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请求保持活动。如果会话空闲一段时间，会超时会话，客户端将发送一个 PING 请求以保持会话活动。此 PING 请求不仅允许 ZooKeeper 服务器知道客户端仍然处于活动状态，而且还允许客户端验证其与 ZooKeeper 服务器的连接是否仍然有效。PING 的时间足够保守，以确保合理的时间来检测死连接并重新连接到新的服务器。

一旦与服务器的连接成功建立（连接），基本上有两种情况，即客户端 lib 生成连接失败（结果代码在 c 绑定，Java 中的异常 - 请参阅 API 文档中绑定特定的详细信息）当同步或执行异步操作，并且其中一个成立：

1. 应用程序调用不再有效/有效的会话的操作
2. 当对该服务器有待处理的操作时，ZooKeeper 客户端与服务器断开连接，即有一个挂起的异步调用。

在 3.2.0 中添加 - `SessionMovedException` 。 有一个内部异常，通常被称为 `SessionMovedException` 的客户端看不到。 发生此异常是因为在已在不同服务器上重新建立的会话的连接上接收到请求。 这个错误的正常原因是向服务器发送请求的客户端，但是数据包被延迟，因此客户端超时并连接到新的服务器。 当延迟的数据包到达第一个服务器时，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此错误，因为它们不会从这些旧连接中读取。（旧连接通常是关闭的。）可以看到这种情况的一种情况是当两个客户端尝试使用保存的会话 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，第二个客户端将断开连接（导致该对尝试无限期地重新建立连接/会话）。

2.2.4 ZooKeeper 观察者 (Watcher)

ZooKeeper 中的所有读取操作 - `getData()` , `getChildren()` 和 `exists()` 可以将观察者 (Watcher) 设置为副作用。 这是 ZooKeeper 对观察者 (Watcher) 的定义: 观察者 (Watcher) 事件是一次性触发，发送到设置观察者 (Watcher) 的客户端，当观察者 (Watcher) 设置的数据更改时发生。 观察者 (Watcher) 的定义有三个要点要考虑:

- **一次触发**

当数据发生变化时，一个观察事件将发送给客户端。 例如，如果客户端执行一个 `getData("/znode1", true)`，然后更改 `/znode1` 的数据被更改或删除，客户端将获取 `/znode1` 的监视事件。 如果 `/znode1` 再次更改，则不会发送 Watcher 事件，除非客户端已经完成另一个设置新观察者 (Watcher) 的读取。

- **发送给客户端**

这意味着事件正在到客户端的路上，但是在成功返回代码到更改操作到达发起更改的客户端之前可能无法到达客户端。 观察者 (Watcher) 异步发送给观察者。 ZooKeeper 提供了一个订购保证: 一个客户端将永远不会看到一个变化，它已经设置了一个观察者 (Watcher)，直到它第一次看到观察者 (Watcher) 事件。 网络延迟或其他因素可能导致不同的客户端在不同时间看到更新的观察者 (Watcher) 和返回代码。 关键是，不同客户端所看到的一切都将具有一致的顺序。

- **观察者 (Watcher) 设置的数据**

这是指节点可以改变的不同方式。 它有助于将 ZooKeeper 视为维护两个观察者 (Watcher) 列表: 数据观察者 (Watcher) 和儿童观察者 (Watcher)。 `getData()` 和 `exists()` 设置数据观察者 (Watcher)。 `getChildren()` 设置子观察者 (Watcher)。 或者，它可能有助于考虑根据返回的数据类型设置观察者 (Watcher)。 `getData()` 和 `exists()` 返回有关节点数据的信息，而 `getChildren()` 返回一个子列表。 因此，`setData()` 将触发正在设置的 `znode` 的数据观察者 (Watcher) (假设该组成功)。 一个成功的 `create()` 将触发正在创建的 `znode` 的数据监视和一个子监视器的父 `znode`。 一个成功的 `delete()` 将触发一个数据观察者 (Watcher) 和一个子观察者 (Watcher) (因为不再有孩子)，因为 `znode` 被删除以及一个子监视器的父 `znode`。

观察者 (Watcher) 在本地维护客户端连接的 ZooKeeper 服务器。 这允许观察者 (Watcher) 设置，维护和发送轻量级。 当客户端连接到新的服务器时，观察者 (Watcher) 将被触发任何会话事件。 手机与服务器断开连接时将不会收到观察者 (Watcher)。 当客户端重新连接时，如果需要，任何先前注册的观察者 (Watcher) 将被重新注册并触发。 一般来说，这一切都是透明的。 有一种可能会丢失观察者 (Watcher) 的情况: 如果在断开连接时创建和删除了 `znode`，则会丢失尚未创建的 `znode` 的观察者 (Watcher)。

2.2.4.1 观察者(Watcher)语义

我们可以通过读取 ZooKeeper 状态的三个调用来设置观察者(Watcher): exists, getData 和 getChildren。以下列表详细介绍了观察者(Watcher)可以触发的事件以及启用它们的呼叫:

- **创建事件:**
启用呼叫存在。
- **已删除事件:**
启用调用存在, getData 和 getChildren。
- **更改事件:**
启用调用存在和 getData。
- **儿童活动:**
通过调用 getChildren 启用。

2.2.4.2 什么 ZooKeeper 保证关于观察者(Watcher)

关于观察者(Watcher), ZooKeeper 保留这些保证:

- 观察者(Watcher)是针对其他事件, 其他观察者(Watcher)和异步回复进行排序的。ZooKeeper 客户端库确保一切都按顺序发送。
- 在看到与该 znode 对应的新数据之前, 客户端将看到它正在 Watcher 的 znode 的 watch 事件。
- 来自 ZooKeeper 的 Watcher 事件的顺序对应于 ZooKeeper 服务所看到的更新顺序。

2.2.4.3 关于观察者(Watcher)的事情

- 观察者(Watcher)是一次触发; 如果您收到一个 Watcher 活动, 并且想要收到未来变化的通知, 则必须设置另一个观察者(Watcher)。
- 因为观察者(Watcher)是一次触发器, 并且在获取事件和发送新的请求以获取 Watcher 之间存在延迟, 因此您无法可靠地看到 ZooKeeper 中的节点发生的每个更改。准备好处理 znode 在获取事件和再次设置观察者(Watcher)之间多次更改的情况。(您可能不在乎, 但至少意识到可能会发生。)
- 观察者(Watcher)对象或功能/上下文只对给定的通知将被触发一次。例如, 如果同一个 Watch 对象被注册为一个存在, 并且对同一个文件进行了一个 getData 调用, 然后该文件被删除, 那么只能通过该文件的删除通知来调用该监视对象一次。
- 当断开与服务器的连接时(例如, 当服务器发生故障)时, 在连接重新建立之前, 您将不会得到任何观察者(Watcher)。因此, 会话事件将发送到所有未完成的处理程序。使用会话事件进入安全模式: 您不会在断开连接时收到事件, 因此您的进程应该以该模式保守运行。

2.2.5 使用 ACL 的 ZooKeeper 访问控制

ZooKeeper 使用 ACL 来控制对其 znode (ZooKeeper 数据树的数据节点) 的访问。ACL 实现与 UNIX 文件访问权限非常相似: 它使用权限位来允许/不允许针对节点的各种操作以及应用该位的范

围。与标准 UNIX 权限不同，ZooKeeper 节点不受用户（文件所有者），组和世界（其他）的三个标准作用域的限制。ZooKeeper 没有 znode 所有者的概念。相反，ACL 会指定与这些 ids 相关联的 ids 和权限集。

还要注意，ACL 只适用于特定的 znode。特别是它不适用于儿童。例如，如果 `/app` 只能由 `ip: 172.16.16.1` 读取，并且 `/app / status` 是可读的，任何人都可以读取 `/app / status`；ACL 不递归。

ZooKeeper 支持可插拔认证方案。Id 使用形式 `scheme id` 指定，其中 `scheme` 是 id 对应的认证方案。例如，`ip: 172.16.16.1` 是一个地址为 `172.16.16.1` 的主机的 ID。

当客户端连接到 ZooKeeper 并对其进行身份验证时，ZooKeeper 会将与客户端对应的所有 ids 与客户端连接相关联。当客户端尝试访问节点时，会针对 znode 的 ACL 检查这些 ID。ACL 由 `(scheme: expression, perms)` 对组成。该表达式的格式特定于该方案。例如，对 `(ip: 19.22.0.0/16, READ)` 给予具有以 19.22 开头的 IP 地址的任何客户端的 `READ` 权限。

2.2.5.1 ACL 权限

ZooKeeper 支持以下权限：

- **CREATE**：您可以创建一个子节点
- **READ**：您可以从节点获取数据并列出其子项。
- **WRITE**：您可以设置节点的数据
- **删除**：您可以删除子节点
- **ADMIN**：你可以设置权限

对于更精细的访问控制，`CREATE` 和 `DELETE` 权限已被打破了 `WRITE` 权限。`CREATE` 和 `DELETE` 的情况如下：

您希望 A 能够在 ZooKeeper 节点上进行设置，但无法 `CREATE` 或 `DELETE` children。

`CREATE` without `DELETE`：客户端通过在父目录中创建 ZooKeeper 节点来创建请求。您希望所有客户端能够添加，但只有请求处理器才能删除。（这类似于文件的 `APPEND` 权限。）

此外，`ADMIN` 权限在那里，因为 ZooKeeper 没有文件所有者的概念。在某种意义上，`ADMIN` 权限将实体指定为所有者。ZooKeeper 不支持 `LOOKUP` 权限（即使您无法列出目录，也可以在目录上执行权限位以允许您使用 `LOOKUP`）。每个人都隐含有 `LOOKUP` 权限。这可以让你统计一个节点，但是没有其他的。（问题是，如果要在不存在的节点上调用 `zoo_exists()`，则无权检查。）

2.2.5.1.1 内置 ACL 方案

ZooKeeper 具有以下内置方案：

- **世界**有一个 id，`任何人`，代表任何人。

- **auth** 不使用任何 id，表示任何经过身份验证的用户。
- **digest** 使用用户名: password 字符串生成 MD5 哈希值，然后将其用作 ACL ID 身份。 通过以明文形式发送用户名: password 进行认证。 在 ACL 中使用时，表达式将是用户名: base64 编码的 SHA1 密码摘要。
- **ip** 使用客户端主机 IP 作为 ACL ID 身份。 ACL 表达式的格式为 *addr / bits*，其中 *addr* 的最高有效位与客户端主机 IP 的最高有效位匹配。

2.2.5.1.2 ZooKeeper C 客户端 API

ZooKeeper C 库提供以下常量：

- `const int ZOO_PERM_READ;` //可以读取节点的值并列出其子节点
- `const int ZOO_PERM_WRITE;` //可以设置节点的值
- `const int ZOO_PERM_CREATE;` //可以创建孩子
- `const int ZOO_PERM_DELETE;` //可以删除孩子
- `const int ZOO_PERM_ADMIN;` //可以执行 `set_acl()`
- `const int ZOO_PERM_ALL;` //所有上述标志 OR'd 在一起

以下是标准 ACL ID：

- `struct Id ZOO_ANYONE_ID_UNSAFE;` // ('world', 'anyone')
- `struct Id ZOO_AUTH_IDS;` // ('auth', '')

ZOO_AUTH_IDS 空的身份字符串应该被解释为“创建者的身份”。

ZooKeeper 客户端提供三种标准 ACL：

- `struct ACL_vector ZOO_OPEN_ACL_UNSAFE;` // (ZOO_PERM_ALL, ZOO_ANYONE_ID_UNSAFE)
- `struct ACL_vector ZOO_READ_ACL_UNSAFE;` // (ZOO_PERM_READ, ZOO_ANYONE_ID_UNSAFE)
- `struct ACL_vector ZOO_CREATOR_ALL_ACL;` // (ZOO_PERM_ALL, ZOO_AUTH_IDS)

所有 ACL 都可以免费使用 ZOO_OPEN_ACL_UNSAFE：任何应用程序都可以在节点上执行任何操作，并可以创建，列出和删除其子项。 ZOO_READ_ACL_UNSAFE 是任何应用程序的只读访问。 CREATE_ALL_ACL 向节点的创建者授予所有权限。 创建者必须已经由服务器进行身份验证（例如，使用“摘要”方案）才能创建具有此 ACL 的节点。

以下 ZooKeeper 操作涉及 ACL：

- `int zoo_add_auth (zhandle_t * zh, const char * scheme, const char * cert, int certLen, void_completion_t completion, const void * data);`

该应用程序使用 `zoo_add_auth` 函数来对服务器进行身份验证。 如果应用程序想要使用不同的方案和/或身份进行身份验证，则可以多次调用该函数。

- `int zoo_create (zhandle_t * zh, const char * path, const char * value, int valuelen, const struct ACL_vector * acl, int flags, char * realpath, int max_realpath_len);`

zoo_create (...) 操作创建一个新节点。 acl 参数是与节点相关联的 ACL 列表。 父节点必须设置 CREATE 权限位。

- `int zoo_get_acl (zhandle_t * zh, const char * path, struct ACL_vector * acl, struct Stat * stat);`

此操作返回节点的 ACL 信息。

- `int zoo_set_acl (zhandle_t * zh, const char * path, int version, const struct ACL_vector * acl);`

此功能用新的代替节点的 ACL 列表。 该节点必须具有 ADMIN 权限集。

以下是使用上述 API 的示例代码，使用 “ foo ” 方案对其进行身份验证，并创建具有创建权限的短暂节点 “ / xyz ”。

注意

这是一个非常简单的例子，旨在显示如何与 ZooKeeper ACL 进行交互。 有关 C 客户端实现的示例，请参阅... / trunk / src / c / src / cli.c

```
#include <string.h>
#include <errno.h>

#include "zookeeper.h"

static zhandle_t * zh;

/ **
 *在这个例子中，这个方法可以获得你的证书
 *环境 - 你必须提供
 * /
char * foo_get_cert_once (char * id) {return 0; }

/ **看守功能 - 这个例子是空的，不是你该的
 *以实际代码* /
void watcher (zhandle_t * zzh, int type, int state, const char *
path,
              void * watcherCtx) {}

int main (int argc, char argv) {
    char 缓冲区[512];
    char p [2048];
    char * cert = 0;
    char appId [64];

    strcpy (appId, "example.foo_test" );
```

```

cert = foo_get_cert_once (appId) ;
if (cert != 0) {
    fprintf (stderr,
        "appid [%s]的证书是[%s] \n", appId, cert) ;
    strncpy (p, cert, sizeof (p) -1) ;
    免费 (证书) ;
} else {
    fprintf (stderr, "appid [%s] not found \n" 的证书, appId) ;
    strcpy (p, "dummy" ) ;
}

zoo_set_debug_level (ZOO_LOG_LEVEL_DEBUG) ;

zh = zookeeper_init ( "localhost: 3181" , watcher, 10000, 0, 0,
0) ;
if (! zh) {
    返回 errno;
}
if (zoo_add_auth (zh, "foo", p, strlen (p) , 0,0) != ZOK)
    返回 2;

struct ACL_CREATE_ONLY_ACL [] = {{ZOO_PERM_CREATE, ZOO_AUTH_IDS}};
struct ACL_vector CREATE_ONLY = {1, CREATE_ONLY_ACL};
int rc = zoo_create (zh, "/ xyz", "value", 5, &CREATE_ONLY,
ZOO_EPHEMERAL,
                    buffer, sizeof (buffer) -1) ;

/ **此操作将失败, 并出现 ZNOAUTH 错误* /
int buflen = sizeof (buffer) ;
结构统计
rc = zoo_get (zh, "/ xyz", 0, buffer, &buflen, &stat) ;
if (rc) {
    fprintf (stderr, "%s \n 的错误%d", rc, __LINE__ ) ;
}

zookeeper_close (zh) ;
返回 0;
}

```

2.2.6 可插拔 ZooKeeper 认证

ZooKeeper 在各种不同的环境中运行各种不同的认证方案, 因此它具有完全可插拔的认证框架。 即使内置认证方案也使用可插拔认证框架。

要了解认证框架的工作原理，首先必须了解两种主要的认证操作。 框架首先必须验证客户端。 这通常在客户端连接到服务器时完成，并且包括验证从客户端发送或收集的信息，并将其与连接相关联。 框架处理的第二个操作是查找与客户端对应的 ACL 中的条目。 ACL 条目为 `< idspec, permissions >` 对。 `idspec` 可以是与连接相关联的认证信息的简单字符串匹配，也可以是针对该信息进行评估的表达式。 这是由认证插件来实现的。 以下是认证插件必须实现的界面：

```
公共接口 AuthenticationProvider {
    String getScheme () ;
    KeeperException.Code handleAuthentication (ServerCnxn cnxn, byte
authData []) ;
    boolean isValid (String id) ;
    布尔匹配 (String id, String aclExpr) ;
    boolean isAuthenticated () ;
}
```

第一种方法 `getScheme` 返回标识插件的字符串。 因为我们支持多种身份验证方法，所以身份验证凭证或 `idspec` 将始终以 `scheme` 为前缀。 ZooKeeper 服务器使用验证插件返回的方案来确定该方案适用于哪些 ID。

当客户端发送与连接关联的身份验证信息时，将调用 `handleAuthentication`。 客户端指定信息对应的方案。 ZooKeeper 服务器将信息传递给验证插件，该验证插件的 `getScheme` 与客户端传递的方案相匹配。 `handleAuthentication` 的实现者通常会在确定信息不正确时返回错误，或者将使用 `cnxn.getAuthInfo ()` 将信息与连接相关联。 `add (new Id (getScheme (), data))` 。

验证插件涉及设置和使用 ACL。 当为 `znode` 设置 ACL 时，ZooKeeper 服务器会将该条目的 `id` 部分传递给 `isValid (String id)` 方法。 验证该标识符是否具有正确的形式。 例如， `ip: 172.16.0.0/16` 是一个有效的 `id`，但 `ip: host.com` 不是。 如果新 ACL 包含“`auth`”条目，则使用 `isAuthenticated` 来查看与该连接相关的该方案的认证信息是否应添加到 ACL 中。 一些方案不应包括在验证中。 例如，如果指定了 `auth`，客户端的 IP 地址不被视为应该添加到 ACL 的 ID。

在检查 ACL 时， ZooKeeper 会调用 `match (String id, String aclExpr)` 。 需要将客户端的认证信息与相关 ACL 条目进行匹配。 要找到适用于客户端的条目，ZooKeeper 服务器将找到每个条目的方案，如果该方案的客户端存在验证信息，则 `匹配 (String id, String aclExpr)` 将被调用， `ID` 设置为验证以前通过 `handleAuthentication` 添加到连接的信息， `aclExpr` 设置为 ACL 条目的标识。 认证插件使用自己的逻辑和匹配方案来确定 `id` 是否包含在 `aclExpr` 中 。

有两个内置的验证插件： `ip` 和 `digest` 。 额外的插件可以使用系统属性进行添加。 在启动时，ZooKeeper 服务器将查找以“`zookeeper.authProvider`”开头的系统属性。 并将这些属性的值解释为验证插件的类名。 可以使用 `-Dzookeeper.authProvider.X = com.f.MyAuth` 设置这些属性，或者在服务器配置文件中添加如下所示的条目：

```
authProvider.1 = com.f.MyAuth
authProvider.2 = com.f.MyAuth2
```

应注意确保财产的后缀是唯一的。如果有重复的东西，如 `-Dzookeeper.authProvider.X = com.f.MyAuth` `-Dzookeeper.authProvider.X = com.f.MyAuth2`，只有一个将被使用。所有服务器也必须具有相同的插件定义，否则，使用插件提供的认证方案的客户端将连接到某些服务器时出现问题。

2.2.7 一致性保证

ZooKeeper 是一款高性能，可扩展的服务。读取和写入操作都被设计为快速，但读取速度比写入速度更快。原因是在阅读的情况下，ZooKeeper 可以提供较旧的数据，这反过来是由于 ZooKeeper 的一致性保证：

顺序一致性

客户端的更新将按照发送的顺序进行应用。

原子性

更新成功或失败 – 没有部分结果。

单系统图像

无论连接到哪个服务器，客户端都将看到服务的相同视图。

可靠性

一旦应用更新，它将从该时间开始持续到客户端覆盖更新。这种担保有两个推论：

1. 如果客户端获得成功的返回代码，则更新将被应用。在某些故障（通信错误，超时等）中，客户端将不知道更新是否已应用。我们采取措施尽量减少故障，但保证只有成功的退货代码。（这被称为 Paxos 的 *单调性条件*。）
2. 通过读取请求或成功更新，客户端看到的任何更新在从服务器故障恢复时都不会回滚。

及时性

系统的客户端视图保证是在一定时间内（数十秒左右）更新的。系统更改将在此绑定内的客户端看到，或者客户端将检测到服务中断。

使用这些一致性保证，只需在 ZooKeeper 客户端（不需要 ZooKeeper 所需的添加）即可轻松构建更高级别的功能，例如领导选举，障碍，队列和读/写可撤销锁。有关详细信息，请参阅[配方和解决方案](#)。

注意

有时候，开发人员错误地假定 ZooKeeper 并不实际做出另一个保证。这是：

同时一致的跨客户端视图

ZooKeeper 并不保证在每个实例的时候，两个不同的客户端将具有相同的 ZooKeeper 数据视图。由于网络延迟等因素，一个客户端可能会在另一个客户端收到更改通知之前执行更新。考虑两个客户端 A 和 B 的场景。如果客户端 A 将 `znode / a` 的值从 0 设置为 1，则告知客

客户端 B 读取 / a，客户端 B 可能会读取旧值 0，具体取决于哪个服务器它连接到。如果客户端 A 和客户端 B 读取相同的值很重要，客户端 B 应该在执行其读取之前从 ZooKeeper API 方法调用 `sync()` 方法。

因此，ZooKeeper 本身并不能保证在所有服务器之间同步进行更改，但是 ZooKeeper 原语可用于构建提供有用的客户端同步的更高级别的功能。（有关更多信息，请参阅 [ZooKeeper 食谱](#)。 [tbd : ...] ）。

2.2.8 绑定

ZooKeeper 客户端库有两种语言：Java 和 C。以下几节介绍这些。

2.2.8.1 Java 绑定

有两个组成 ZooKeeper Java 绑定的包：`org.apache.zookeeper` 和 `org.apache.zookeeper.data`。构成 ZooKeeper 的其他软件包在内部使用或是服务器实现的一部分。`org.apache.zookeeper.data` 包由生成的类组成，它们简单地用作容器。

ZooKeeper Java 客户端使用的主要类是 `ZooKeeper` 类。它的两个构造函数只有可选的会话 ID 和密码不同。ZooKeeper 支持进程的实例的会话恢复。Java 程序可以将其会话 ID 和密码保存到稳定的存储，重新启动和恢复程序早期实例使用的会话。

当创建 ZooKeeper 对象时，也会创建两个线程：IO 线程和事件线程。所有 IO 都发生在 IO 线程上（使用 Java NIO）。所有事件回调都发生在事件线程上。会话维护例如重新连接到 ZooKeeper 服务器并维护心跳在 IO 线程上完成。同步方法的响应也在 IO 线程中处理。对事件线程处理对异步方法和监视事件的所有响应。有一些事情要注意，从这个设计的结果：

- 异步调用和观察器回调的所有完成将按顺序进行，一次一个。呼叫者可以进行任何处理，但在此期间不会处理其他回调。
- 回调不阻止 IO 线程的处理或同步调用的处理。
- 同步呼叫可能不会以正确的顺序返回。例如，假设客户端执行以下处理：发出 `node / a` 的异步读取，将 `watch` 设置为 `true`，然后在读取的完成回调中执行 `/ a` 的同步读取。（也许不是很好的做法，但也不是非法的，这只是一个简单的例子。）

请注意，如果异步读取和同步读取之间的 `/ a` 有变化，客户端库将在同步读取的响应之前接收到表示 `/ a` 更改的监视事件，但由于完成回调阻塞事件队列，在处理 `watch` 事件之前，同步读取将返回新值 `/ a`。

最后，与关机关联的规则很简单：一旦 ZooKeeper 对象关闭或接收到一个致命事件（`SESSION_EXPIRED` 和 `AUTH_FAILED`），ZooKeeper 对象将变为无效。关闭，两个线程关闭，并且 `zookeeper` 手柄上的任何进一步访问都是未定义的行为，应该避免。

2.2.8.2 C 绑定

C 绑定具有单线程和多线程库。多线程库最容易使用，与 Java API 最相似。该库将创建一个 IO 线程和一个用于处理连接维护和回调的事件调度线程。单线程库允许通过暴露多线程库中使用的事件循环来将 ZooKeeper 用于事件驱动的应用程序。

该软件包包含两个共享库：zookeeper_st 和 zookeeper_mt。前者仅提供用于集成到应用程序事件循环中的异步 API 和回调。这个库存在的唯一原因是支持平台是一个 *pthread* 库不可用或不稳定（即 FreeBSD 4.x）。在所有其他情况下，应用程序开发人员应与 zookeeper_mt 进行链接，因为它包括对 Sync 和 Async API 的支持。

2.2.8.2.1 安装

如果您从 Apache 存储库中的退房构建客户端，请按照以下步骤进行操作。如果您从从 apache 下载的项目源代码包构建，请跳到步骤 3。

1. 从 ZooKeeper 顶级目录（... / trunk）运行 `ant compile_jute`。这将在... / trunk / src / c 下创建一个名为“generated”的目录。
2. 将目录更改为... / trunk / src / c 并运行 `autoreconf -if` 以引导 **autoconf**，**automake** 和 **libtool**。确保安装了 **autoconf 版本 2.59** 或更高版本。跳到步骤 4。
3. 如果您正在从项目源包构建，请将源 tarball 和 cd 解压缩到 `zookeeper-xxx / src / c` 目录。
4. 运行 `./configure <your-options>` 来生成 makefile。以下是配置实用程序支持的一些选项，可用于此步骤：
 - `--enable-debug`
启用优化并启用调试信息编译器选项。（默认情况下禁用）
 - `--without-syncapi`
禁用 Sync API 支持；zookeeper_mt 库将不会被构建。（默认情况下启用）
 - `- 静态静态`
不要建立静态库。（默认情况下启用）
 - `- 不可共享`
不要建立共享库。（默认情况下启用）

5. 注意

6. 有关运行 **configure** 的一般信息，请参阅 INSTALL。
7. 运行 `make` 或 `make install` 构建库并进行安装。
8. 要为 ZooKeeper API 生成 doxygen 文档，请运行 `make doxygen-doc`。所有文档将被放置在名为 docs 的新子文件夹中。默认情况下，此命令仅生成 HTML。有关其他文档格式的信息，请运行 `./configure --help`

2.2.8.2.2 建立您自己的 C 客户端

为了能够在你的应用程序中使用 ZooKeeper API，你必须记住

1. 包含 ZooKeeper 标题：`#include <zookeeper / zookeeper.h>`
2. 如果您正在构建多线程客户机，请使用 `-DTHREADED` 编译器标志编译以启用库的多线程版本，然后与 `zookeeper_mt` 库进行链接。如果您正在构建单线程客户端，请勿使用 `-DTHREADED` 进行编译，并确保与 `zookeeper_st` 库进行链接。

注意

有关 C 客户端实现的示例，请参阅... / trunk / src / c / src / cli.c

2.2.9 构建块：ZooKeeper 操作指南

本节将调查开发人员可以针对 ZooKeeper 服务器执行的所有操作。它比本手册中较早的概念章节低级别，但比 ZooKeeper API Reference 更高级别。它涵盖这些主题：

- [连接到 ZooKeeper](#)

2.2.9.1 处理错误

Java 和 C 客户端绑定都可能会报错。Java 客户端绑定通过抛出 `KeeperException` 这样做，对异常调用代码（）将返回特定的错误代码。C 客户端绑定返回在枚举 `ZOO_ERRORS` 中定义的错误代码。API 回调指示两种语言绑定的结果代码。有关可能的错误及其含义的完整详细信息，请参阅 API 文档（Java 的 `javadoc` for Java，`doxygen` for C）。

2.2.9.2 连接到 ZooKeeper

2.2.9.3 阅读操作

2.2.9.4 写作操作

2.2.9.5 处理观察者 (Watcher)

2.2.9.6 小型 Zookeeper 行动

2.2.10 程序结构，简单示例

[tbd]

2.2.11 Gotchas: 常见问题和故障排除

所以现在你知道 ZooKeeper。它的快速，简单，您的应用程序工作，但等待... 有什么问题。以下是 ZooKeeper 用户陷入的一些陷阱：

1. 如果您使用观察者 (Watcher)，则必须查找连接的观察者 (Watcher) 事件。当 ZooKeeper 客户端与服务器断开连接时，在重新连接之前，您将不会收到更改通知。如果您正在 Watcherznode 来存在，则如果在断开连接时创建并删除了 znode，那么您将会错过该事件。
2. 您必须测试 ZooKeeper 服务器故障。只要大多数服务器都是活动的，ZooKeeper 服务就能够幸免于难。要问的问题是：你的应用程序可以处理吗？在现实世界中，客户与 ZooKeeper 的连接可能会破裂。（ZooKeeper 服务器故障和网络分区是连接丢失的常见原因。）ZooKeeper 客户端库负责恢复您的连接，并告诉您发生了什么，但是您必须确保恢复您的状态和任何未完成的请求。了解您是否在测试实验室中正确使用，而不是在生产中使用由几台服务器组成的 ZooKeeper 服务进行测试，并对其进行重新启动。
3. 客户端使用的 ZooKeeper 服务器列表必须与每个 ZooKeeper 服务器具有的 ZooKeeper 服务器列表相匹配。如果客户端列表是 ZooKeeper 服务器的真实列表的一部分，但是如果客户端列出 ZooKeeper 服务器不在 ZooKeeper 群集中，那么事情可以工作，但不是最佳。
4. 小心你把那个事务日志放在哪里。ZooKeeper 最具性能关键的部分是事务日志。在返回响应之前，ZooKeeper 必须将事务同步到媒体。专用的事务日志设备是一致的良好性能的关键。将日志放在繁忙的设备上会不利地影响性能。如果您只有一个存储设备，请将跟踪文件放在 NFS 上并增加 snapshotCount；它不能消除这个问题，但可以减轻这个问题。

5. 正确设置您的 Java 最大堆大小。 避免交换是非常重要的。 不必要地进入磁盘将几乎肯定会降低您的性能不可接受。 请记住，在 ZooKeeper 中，一切都是有序的，所以如果一个请求命中磁盘，所有其他排队的请求都会到达磁盘。

为避免交换，尝试将堆叠的物理内存数量减少到减少操作系统和缓存所需的数量。 为配置确定最佳堆大小的最佳方法是运行负载测试。 如果由于某种原因你不能在你的估计中保守，并选择一个远低于限制的数字，这将导致你的机器交换。 例如，在 4G 机器上，3G 堆是一个保守的估计。

在正式文档之外，ZooKeeper 开发人员还有其他几个信息来源。

ZooKeeper 白皮书[tbd : find url]

由 Yahoo! Research 对 ZooKeeper 的设计和性能进行定论

API 参考[tbd : find url]

完整引用 ZooKeeper API

2008 年 Hadou 峰会上的 ZooKeeper Talk

由 Yahoo! Research 的 Benjamin Reed 发行的 ZooKeeper 视频介绍

屏障和队列教程

Flavio Junqueira 的优秀 Java 教程，使用 ZooKeeper 实现简单的屏障和生产者 - 消费者队列。

ZooKeeper - 可靠，可扩展的分布式协调系统

Todd Hoff 的一篇文章（07/15/2008）

ZooKeeper 食谱

使用 ZooKeeper 实现各种同步解决方案的虚拟讨论：事件句柄，队列，锁和两阶段提交。

[tbd]

任何人都可以想到的任何其他好的来源...

2.3 ZooKeeper Java 示例

- 一个简单的观察客户端
 - 要求
 - 程序设计
- 执行者类
- DataMonitor 类
- 完整的来源列表

2.3.1 一个简单的 Watcher 客户端

要介绍 ZooKeeper Java API，我们在这里开发一个非常简单的 Watcher 客户端。该 ZooKeeper 客户端通过启动或停止程序来监视 ZooKeeper 节点的更改并进行响应。

2.3.1.1 要求

客户有四个要求：

- 它需要参数：
 - ZooKeeper 服务的地址
 - 那么 znode 的名字就是被监视的
 - 具有参数的可执行文件。
- 它获取与 znode 相关联的数据，并启动可执行文件。
- 如果 znode 更改，客户端将重新启动内容并重新启动可执行文件。
- 如果 znode 消失，客户端将杀死可执行文件。

2.3.1.2 程序设计

通常，ZooKeeper 应用程序分为两个单元，一个维护连接，另一个用于监视数据。在此应用程序中，名为 **Executor** 的类维护 ZooKeeper 连接，并且名为 **DataMonitor** 的类监视 ZooKeeper 树中的数据。此外，Executor 包含主线程并包含执行逻辑。它负责什么样的用户交互，以及与您作为参数传递的可执行程序的交互以及根据 znode 的状态关闭和重新启动示例（根据要求）。

2.3.2 执行者类

Executor 对象是示例应用程序的主容器。它包含 **ZooKeeper** 对象，**DataMonitor**，如上面在 程序设计中所述。

```
// from the Executor class...

public static void main(String[] args) {
    if (args.length < 4) {
        System.err
            .println("USAGE: Executor hostPort znode filename program [args ...]");
        System.exit(2);
    }
    String hostPort = args[0];
    String znode = args[1];
    String filename = args[2];
    String exec[] = new String[args.length - 3];
    System.arraycopy(args, 3, exec, 0, exec.length);
}
```

```

        try {
            new Executor(hostPort, znode, filename, exec).run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public Executor(String hostPort, String znode, String filename,
        String exec[]) throws KeeperException, IOException {
        this.filename = filename;
        this.exec = exec;
        zk = new ZooKeeper(hostPort, 3000, this);
        dm = new DataMonitor(zk, znode, null, this);
    }

    public void run() {
        try {
            synchronized (this) {
                while (!dm.dead) {
                    wait();
                }
            }
        } catch (InterruptedException e) {
        }
    }
}

```

回想一下，执行者的工作是启动和停止在命令行上传入的可执行文件。 它是为了响应 ZooKeeper 对象触发的事件。 正如你可以在上面的代码中看到的，Executor 将一个引用作为 ZooKeeper 构造函数中的 Watcher 参数传递给自己。 它还将对 DataMonitorListener 参数的引用传递给 DataMonitor 构造函数。 根据 Executor 的定义，它实现了这两个接口：

```

public class Executor implements Watcher, Runnable, DataMonitor.DataMonitorListener {
    ...
}

```

Watcher 界面由 ZooKeeper Java API 定义。 ZooKeeper 使用它来回传到其容器。 它只支持一种方法 process()，而 ZooKeeper 使用它来通信主线程将被插入的通用事件，例如 ZooKeeper 连接或 ZooKeeper 会话的状态。本示例中的执行器简单地将这些事件转发到 DataMonitor 来决定如何处理它们。 它只是为了说明一点，按照惯例，Executor 或一些类似 Executor 的对象“拥有” ZooKeeper 连接，但可以将事件委托给其他事件到其他对象。 它也使用它作为防火观察事件的默认通道。（稍后再来）

```

public void process(WatchedEvent event) {
    dm.process(event);
}

```

另一方面，DataMonitorListener 接口不属于 ZooKeeper API。它是一个完全定制的界面，专为此示例应用程序而设计。DataMonitor 对象使用它来回传给它的容器，它也是 Executor 对象。

DataMonitorListener 接口如下所示：

```
public interface DataMonitorListener {  
    /**  
     * The existence status of the node has changed.  
     */  
    void exists(byte data[]);  
  
    /**  
     * The ZooKeeper session is no longer valid.  
     *  
     * @param rc  
     * the ZooKeeper reason code  
     */  
    void closing(int rc);  
}
```

该接口在 DataMonitor 类中定义，并在 Executor 类中实现。当执行 Executor.exists（）时，执行器根据要求决定是启动还是关闭。回想一下，当 znode 不再存在时，需要说的是杀死可执行文件。

当调用 Executor.closing（）时，Executor 决定是否关闭自己，以响应 ZooKeeper 连接永久消失。

您可能已经猜到，DataMonitor 是调用这些方法的对象，以响应 ZooKeeper 状态的更改。

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

```
public void exists (byte [] data) {  
    if (data == null) {  
        if (child != null) {  
            System.out.println ( "Killing process" );  
            child.destroy ();  
            尝试{  
                child.waitFor ();  
            } catch (InterruptedException e) {  
            }  
        }  
        child = null;  
    } else {  
        if (child != null) {
```



```

        System.out.println ( “Stopping child” ) ;
        child.destroy () ;
        尝试{
            child.waitFor () ;
        } catch (InterruptedException e) {
            e.printStackTrace () ;
        }
    }
    尝试{
        FileOutputStream fos = new FileOutputStream (filename) ;
        fos.write (data) ;
        fos.close () ;
    } catch (IOException e) {
        e.printStackTrace () ;
    }
    尝试{
        System.out.println ( “Starting child” ) ;
        child = Runtime.getRuntime () 。exec (exec) ;
        新的 StreamWriter (child.getInputStream () ,
System.out) ;
        新 StreamWriter (child.getErrorStream () , System.err) ;
    } catch (IOException e) {
        e.printStackTrace () ;
    }
}

public void closing (int rc) {
    synchronized (this) {
        notifyAll () ;
    }
}
}

```

2.3.3 DataMonitor 类

DataMonitor 类具有 ZooKeeper 逻辑的肉体。 它主要是异步和事件驱动。 DataMonitor 在构造函数中使用：

```

public DataMonitor(ZooKeeper zk, String znode, Watcher
chainedWatcher,
    DataMonitorListener listener) {
    this.zk = zk;
    this.znode = znode;
    this.chainedWatcher = chainedWatcher;
}

```

```

        this.listener = listener;

        // Get things started by checking if the node exists. We are
going
        // to be completely event driven
        zk.exists(znode, true, this, null);
    }

```

对 ZooKeeper.exists () 的调用检查是否存在 znode，设置一个观察者 (Watcher)，并将一个引用自身 (this) 作为完成回调对象传递给它。 在这个意义上，它会踢东西，因为当观察者 (Watcher) 被触发时，真正的处理就会发生。

注意

不要将完成回调与观察者 (Watcher) 回调混淆。 当 *监视* 操作的异步 *设置* (由 ZooKeeper.exists ()) 在服务器上完成时， ZooKeeper.exists () 完成回调恰好是在 DataMonitor 对象中实现的 StatCallback.processResult () 方法 。

另一方面，触发观察者 (Watcher) 将向 *Executor* 对象发送一个事件，因为 Executor 注册为 ZooKeeper 对象的 Watcher。

除此之外，您可能会注意到，DataMonitor 也可以将自己注册为此特定 Watcher 事件的观察者。 这是 ZooKeeper 3.0.0 (支持多个观察者) 的新功能。 但是在这个例子中，DataMonitor 不会注册为 Watcher。

当 ZooKeeper.exists () 操作在服务器上完成时，ZooKeeper API 会在客户端上调用此完成回调：

```

public void processResult(int rc, String path, Object ctx, Stat stat) {
    boolean exists;
    switch (rc) {
    case Code.Ok:
        exists = true;
        break;
    case Code.NoNode:
        exists = false;
        break;
    case Code.SessionExpired:
    case Code.NoAuth:
        dead = true;
        listener.closing(rc);
        return;
    default:
        // Retry errors
        zk.exists(znode, true, this, null);
        return;
    }
}

```

```

byte b[] = null;
if (exists) {
    try {
        b = zk.getData(znode, false, null);
    } catch (KeeperException e) {
        // We don't need to worry about recovering now. The watch
        // callbacks will kick off any exception handling
        e.printStackTrace();
    } catch (InterruptedException e) {
        return;
    }
}
if ((b == null && b != prevData)
    || (b != null && !Arrays.equals(prevData, b))) {
    listener.exists(b);
    prevData = b;
}
}

```

代码首先检查 znode 存在，致命错误和可恢复错误的错误代码。如果文件（或 znode）存在，它将从 znode 获取数据，然后调用 Executor 的 exists（）回调，如果状态已更改。注意，它不需要为 getData 调用执行异常处理，因为它具有挂起的任何可能导致错误的监视器：如果节点在调用 ZooKeeper.getData（）之前被删除，则由 ZooKeeper 设置的监视事件.exists（）触发回调；如果发生通信错误，则在连接重新启动时触发连接观察事件。

最后，请注意 DataMonitor 如何处理监视事件：

```

public void process(WatchedEvent event) {
    String path = event.getPath();
    if (event.getType() == Event.EventType.None) {
        // We are being told that the state of the
        // connection has changed
        switch (event.getState()) {
            case SyncConnected:
                // In this particular example we don't need to do anything
                // here - watches are automatically re-registered with
                // server and any watches triggered while the client was
                // disconnected will be delivered (in order of course)
                break;
            case Expired:
                // It's all over
                dead = true;
                listener.closing(KeeperException.Code.SessionExpired);

```

```

        break;
    }
} else {
    if (path != null && path.equals(znode)) {
        // Something has changed on the node, let's find out
        zk.exists(znode, true, this, null);
    }
}
}
if (chainedWatcher != null) {
    chainedWatcher.process(event);
}
}
}

```

如果客户端 ZooKeeper 库可以在会话到期之前（过期事件）将 ZooKeeper 重新建立通信通道（SyncConnected 事件），所有会话的观察者（Watcher）将自动与服务器重新建立（观察者（Watcher）自动重置 ZooKeeper 3.0.0）。有关更多信息，请参阅[程序员指南](#)中的 [ZooKeeper 观察者（Watcher）](#)。在这个功能中，当 DataMonitor 获取一个 znode 的一个事件时，这个函数有点低一点，它调用了 ZooKeeper.exists（）来找出改变的内容。

2.3.4完整的来源列表

Executor.java

```

/**
 * A simple example program to use DataMonitor to start and
 * stop executables based on a znode. The program watches the
 * specified znode and saves the data that corresponds to the
 * znode in the filesystem. It also starts the specified program
 * with the specified arguments when the znode exists and kills
 * the program if the znode goes away.
 */
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;

public class Executor
    implements Watcher, Runnable, DataMonitor.DataMonitorListener
{
    String znode;

```

```

DataMonitor dm;

ZooKeeper zk;

String filename;

String exec[];

Process child;

public Executor(String hostPort, String znode, String filename,
    String exec[]) throws KeeperException, IOException {
    this.filename = filename;
    this.exec = exec;
    zk = new ZooKeeper(hostPort, 3000, this);
    dm = new DataMonitor(zk, znode, null, this);
}

/**
 * @param args
 */
public static void main(String[] args) {
    if (args.length < 4) {
        System.err
            .println("USAGE: Executor hostPort znode filename program [args ...]");
        System.exit(2);
    }
    String hostPort = args[0];
    String znode = args[1];
    String filename = args[2];
    String exec[] = new String[args.length - 3];
    System.arraycopy(args, 3, exec, 0, exec.length);
    try {
        new Executor(hostPort, znode, filename, exec).run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/*****
 * We do process any events ourselves, we just need to forward them on.
 *
 * @see org.apache.zookeeper.Watcher#process(org.apache.zookeeper.proto.WatcherEvent)

```

```

    */
    public void process(WatchedEvent event) {
        dm.process(event);
    }

    public void run() {
        try {
            synchronized (this) {
                while (!dm.dead) {
                    wait();
                }
            }
        } catch (InterruptedException e) {
        }
    }

    public void closing(int rc) {
        synchronized (this) {
            notifyAll();
        }
    }

    static class StreamWriter extends Thread {
        OutputStream os;

        InputStream is;

        StreamWriter(InputStream is, OutputStream os) {
            this.is = is;
            this.os = os;
            start();
        }

        public void run() {
            byte b[] = new byte[80];
            int rc;
            try {
                while ((rc = is.read(b)) > 0) {
                    os.write(b, 0, rc);
                }
            } catch (IOException e) {
            }
        }
    }

```

```

    }

    public void exists(byte[] data) {
        if (data == null) {
            if (child != null) {
                System.out.println("Killing process");
                child.destroy();
                try {
                    child.waitFor();
                } catch (InterruptedException e) {
                }
            }
            child = null;
        } else {
            if (child != null) {
                System.out.println("Stopping child");
                child.destroy();
                try {
                    child.waitFor();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            try {
                FileOutputStream fos = new FileOutputStream(filename);
                fos.write(data);
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                System.out.println("Starting child");
                child = Runtime.getRuntime().exec(exec);
                new StreamWriter(child.getInputStream(), System.out);
                new StreamWriter(child.getErrorStream(), System.err);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

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,
        DataMonitorListener listener) {
        this.zk = zk;
        this.znode = znode;
        this.chainedWatcher = chainedWatcher;
        this.listener = listener;
        // Get things started by checking if the node exists. We are going
        // to be completely event driven
        zk.exists(znode, true, this, null);
    }

    /**
     * Other classes use the DataMonitor by implementing this method
     */
    public interface DataMonitorListener {
        /**
         * The existence status of the node has changed.

```



```

    */
void exists(byte data[]);

/**
 * The ZooKeeper session is no longer valid.
 *
 * @param rc
 *         the ZooKeeper reason code
 */
void closing(int rc);
}

public void process(WatchedEvent event) {
    String path = event.getPath();
    if (event.getType() == Event.EventType.None) {
        // We are being told that the state of the
        // connection has changed
        switch (event.getState()) {
            case SyncConnected:
                // In this particular example we don't need to do anything
                // here - watches are automatically re-registered with
                // server and any watches triggered while the client was
                // disconnected will be delivered (in order of course)
                break;
            case Expired:
                // It's all over
                dead = true;
                listener.closing(KeeperException.Code.SessionExpired);
                break;
        }
    } else {
        if (path != null && path.equals(znode)) {
            // Something has changed on the node, let's find out
            zk.exists(znode, true, this, null);
        }
    }
    if (chainedWatcher != null) {
        chainedWatcher.process(event);
    }
}

public void processResult(int rc, String path, Object ctx, Stat stat) {
    boolean exists;
    switch (rc) {

```

```

    case Code.Ok:
        exists = true;
        break;
    case Code.NoNode:
        exists = false;
        break;
    case Code.SessionExpired:
    case Code.NoAuth:
        dead = true;
        listener.closing(rc);
        return;
    default:
        // Retry errors
        zk.exists(znode, true, this, null);
        return;
}

byte b[] = null;
if (exists) {
    try {
        b = zk.getData(znode, false, null);
    } catch (KeeperException e) {
        // We don't need to worry about recovering now. The watch
        // callbacks will kick off any exception handling
        e.printStackTrace();
    } catch (InterruptedException e) {
        return;
    }
}

if ((b == null && b != prevData)
    || (b != null && !Arrays.equals(prevData, b))) {
    listener.exists(b);
    prevData = b;
}
}
}

```

2.4 障碍和队列：使用 ZooKeeper 编程 – 一个基础教程

- [介绍](#)
- [障碍](#)
- [生产者 – 消费者队列](#)
- [完整的源代码清单](#)

2.4.1 介绍

在本教程中，我们将使用 ZooKeeper 显示屏障和生产者 - 消费者队列的简单实现。我们称各自的类别为障碍和队列。这些示例假定您至少运行了一个 ZooKeeper 服务器。

两个原语都使用以下代码摘要：

```
static ZooKeeper zk = null;
static Integer mutex;

String root;

SyncPrimitive(String address) {
    if(zk == null) {
        try {
            System.out.println("Starting ZK:");
            zk = new ZooKeeper(address, 3000, this);
            mutex = new Integer(-1);
            System.out.println("Finished starting ZK: " + zk);
        } catch (IOException e) {
            System.out.println(e.toString());
            zk = null;
        }
    }
}

synchronized public void process(WatchedEvent event) {
    synchronized (mutex) {
        mutex.notify();
    }
}
```

两个类都扩展了 SyncPrimitive。以这种方式，我们执行 SyncPrimitive 的构造函数中所有原语通用的步骤。为了简化示例，我们首次实例化了一个屏障对象或一个队列对象，我们创建了一个 ZooKeeper 对象，我们声明一个静态变量，它是对该对象的引用。Barrier 和 Queue 的后续实例检查 ZooKeeper 对象是否存在。或者，我们可以让应用程序创建一个 ZooKeeper 对象并将其传递给 Barrier 和 Queue 的构造函数。

我们使用 process() 方法处理由于观察者(Watcher)而触发的通知。在下面的讨论中，我们提供了设置观察者(Watcher)的代码。观察者(Watcher)是内部结构，使 ZooKeeper 能够向客户端通知节点的更改。例如，如果客户端正在等待其他客户端离开屏障，那么它可以设置一个监视并等待对特定节点的修改，这可以指示它是等待的结束。一旦我们了解这些例子，这一点就会变得清楚。

2.4.2 障碍

屏障是使一组进程能够同步计算的开始和结束的原语。该实现的一般思想是具有用于作为单个进程节点的父节点的障碍节点。假设我们称为屏障节点“/ b1”。每个进程“p”然后创建一个节点“/ b1 / p”。一旦足够的进程创建了其对应的节点，连接的进程就可以开始计算。

在这个例子中，每个进程实例化一个 Barrier 对象，它的构造函数作为参数：

- ZooKeeper 服务器的地址（例如，“zoo1.foo.com:2181”）
- ZooKeeper 上的屏障节点的路径（例如，“/ b1”）
- 进程组的大小

Barrier 的构造函数将 Zookeeper 服务器的地址传递给父类的构造函数。父类创建一个 ZooKeeper 实例（如果不存在）。Barrier 的构造函数然后在 ZooKeeper 上创建一个屏障节点，它是所有进程节点的父节点，我们称之为 root（**注意：**这不是 ZooKeeper 根“/”）。

```
/**
 * Barrier constructor
 *
 * @param address
 * @param root
 * @param size
 */
Barrier(String address, String root, int size) {
    super(address);
    this.root = root;
    this.size = size;

    // Create barrier node
    if (zk != null) {
        try {
            Stat s = zk.exists(root, false);
            if (s == null) {
                zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
                    CreateMode.PERSISTENT);
            }
        } catch (KeeperException e) {
            System.out
                .println("Keeper exception when instantiating queue: "
                    + e.toString());
        } catch (InterruptedException e) {
            System.out.println("Interrupted exception");
        }
    }
}
```

```

        // My node name
        try {
            name = new
String(InetAddress.getLocalHost().getCanonicalHostName().toString());
        } catch (UnknownHostException e) {
            System.out.println(e.toString());
        }
    }
}

```

要进入障碍，进程调用 `enter()`。该进程在根下创建一个节点来表示它，使用其主机名来形成节点名称。然后等待足够的进程进入障碍。一个进程通过使用“`getChildren()`”检查根节点具有的子节点数，并且在没有足够的情况下等待通知。要在根节点发生更改时收到通知，进程必须设置一个观察者(Watcher)，并通过调用“`getChildren()`”来执行。在代码中，我们有“`getChildren()`”有两个参数。第一个声明节点要读取，第二个是一个布尔标志，使进程能够设置一个观察者(Watcher)。在代码中，标志是真的。

```

/**
 * Join barrier
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */

boolean enter() throws KeeperException, InterruptedException{
    zk.create(root + "/" + name, new byte[0], Ids.OPEN_ACL_UNSAFE,
        CreateMode.EPHEMERAL_SEQUENTIAL);
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);

            if (list.size() < size) {
                mutex.wait();
            } else {
                return true;
            }
        }
    }
}
}

```

请注意，enter（）抛出 KeeperException 和 InterruptedException，因此应用程序可以捕获并处理这些异常。

一旦计算完成，一个进程调用 leave（）离开障碍。 首先删除其对应的节点，然后获取根节点的子节点。 如果至少有一个小孩，那么它会等待一个通知（obs：请注意，调用 getChildren（）的第二个参数为 true，这意味着 ZooKeeper 必须在根节点上设置一个 watch）。 在接收到通知后，它再次检查根节点是否有任何子节点。

```
/**
 * Wait until all reach barrier
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */

boolean leave() throws KeeperException, InterruptedException{
    zk.delete(root + "/" + name, 0);
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);
            if (list.size() > 0) {
                mutex.wait();
            } else {
                return true;
            }
        }
    }
}
```

2.4.3 生产者 - 消费者队列

生产者 - 消费者队列是分组数据结构，一组进程用于生成和使用项目。 生产者流程创建新元素并将其添加到队列中。 消费者进程从列表中删除元素，并处理它们。 在这个实现中，元素是简单的整数。 队列由根节点表示，并且要向队列添加元素，生产者进程将创建一个新节点，即根节点的子节点。

代码的以下摘录对应于对象的构造函数。 与 Barrier 对象一样，它首先调用父类 SyncPrimitive 的构造函数，如果不存在，则会创建一个 ZooKeeper 对象。 然后它验证队列的根节点是否存在，如果不存在则创建。

```
/**
 * Constructor of producer-consumer queue
 *
```

```

    * @param address
    * @param name
    */
    Queue(String address, String name) {
        super(address);
        this.root = name;
        // Create ZK node name
        if (zk != null) {
            try {
                Stat s = zk.exists(root, false);
                if (s == null) {
                    zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
                        CreateMode.PERSISTENT);
                }
            } catch (KeeperException e) {
                System.out
                    .println("Keeper exception when instantiating queue: "
                        + e.toString());
            } catch (InterruptedException e) {
                System.out.println("Interrupted exception");
            }
        }
    }
}

```

生产者进程调用“produce ()”将一个元素添加到队列中，并将整数作为参数传递。要向队列中添加元素，该方法使用“create ()”创建一个新节点，并使用 SEQUENCE 标志来指示 ZooKeeper 附加与根节点关联的序列器计数器的值。以这种方式，我们对队列的元素施加了一个总的顺序，从而保证队列中最旧的元素是下一个元素。

```

/**
    * 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;
    }
}
```

要消耗元素，消费者进程获取根节点的子节点，读取具有最小计数器值的节点，并返回该元素。 请注意，如果存在冲突，则两个竞争进程之一将无法删除该节点，并且删除操作将抛出异常。

调用 `getChildren()` 以字典顺序返回子项列表。 由于字典顺序不必遵循计数器值的数字顺序，因此我们需要确定哪个元素是最小的。 要确定哪个计数器值最小，我们遍历列表，并从每个元素中删除前缀“元素”。

```
/**
 * Remove first element from the queue.
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */
int consume() throws KeeperException, InterruptedException{
    int retvalue = -1;
    Stat stat = null;

    // Get the first element available
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);
            if (list.size() == 0) {
                System.out.println("Going to wait");
                mutex.wait();
            } else {
                Integer min = new Integer(list.get(0).substring(7));
                for(String s : list){
                    Integer tempValue = new Integer(s.substring(7));
                    //System.out.println("Temporary value: " + tempValue);
                    if(tempValue < min) min = tempValue;
                }
                System.out.println("Temporary value: " + root + "/element" + min);
                byte[] b = zk.getData(root + "/element" + min,
                    false, stat);
                zk.delete(root + "/element" + min, 0);
                ByteBuffer buffer = ByteBuffer.wrap(b);
                retvalue = buffer.getInt();
            }
        }
    }
}
```



```

    }
    //else mutex = new Integer(-1);
}

synchronized public void process(WatchedEvent event) {
    synchronized (mutex) {
        //System.out.println("Process: " + event.getType());
        mutex.notify();
    }
}

/**
 * Barrier
 */
static public class Barrier extends SyncPrimitive {
    int size;
    String name;

    /**
     * Barrier constructor
     *
     * @param address
     * @param root
     * @param size
     */
    Barrier(String address, String root, int size) {
        super(address);
        this.root = root;
        this.size = size;

        // Create barrier node
        if (zk != null) {
            try {
                Stat s = zk.exists(root, false);
                if (s == null) {
                    zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
                        CreateMode.PERSISTENT);
                }
            } catch (KeeperException e) {
                System.out
                    .println("Keeper exception when instantiating queue: "
                        + e.toString());
            } catch (InterruptedException e) {
                System.out.println("Interrupted exception");
            }
        }
    }
}

```

```

        }
    }

    // My node name
    try {
        name = new
String(InetAddress.getLocalHost().getCanonicalHostName().toString());
    } catch (UnknownHostException e) {
        System.out.println(e.toString());
    }

}

/**
 * Join barrier
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */

boolean enter() throws KeeperException, InterruptedException{
    zk.create(root + "/" + name, new byte[0], Ids.OPEN_ACL_UNSAFE,
        CreateMode.EPHEMERAL_SEQUENTIAL);
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);

            if (list.size() < size) {
                mutex.wait();
            } else {
                return true;
            }
        }
    }
}

/**
 * Wait until all reach barrier
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */

```

```

boolean leave() throws KeeperException, InterruptedException{
    zk.delete(root + "/" + name, 0);
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);
            if (list.size() > 0) {
                mutex.wait();
            } else {
                return true;
            }
        }
    }
}

/**
 * Producer-Consumer queue
 */
static public class Queue extends SyncPrimitive {

    /**
     * Constructor of producer-consumer queue
     *
     * @param address
     * @param name
     */
    Queue(String address, String name) {
        super(address);
        this.root = name;
        // Create ZK node name
        if (zk != null) {
            try {
                Stat s = zk.exists(root, false);
                if (s == null) {
                    zk.create(root, new byte[0], Ids.OPEN_ACL_UNSAFE,
                        CreateMode.PERSISTENT);
                }
            } catch (KeeperException e) {
                System.out
                    .println("Keeper exception when instantiating queue: "
                        + e.toString());
            } catch (InterruptedException e) {
                System.out.println("Interrupted exception");
            }
        }
    }
}

```

```

        }
    }
}

/**
 * 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;
}

/**
 * Remove first element from the queue.
 *
 * @return
 * @throws KeeperException
 * @throws InterruptedException
 */

int consume() throws KeeperException, InterruptedException{
    int retvalue = -1;
    Stat stat = null;

    // Get the first element available
    while (true) {
        synchronized (mutex) {
            List<String> list = zk.getChildren(root, true);
            if (list.size() == 0) {
                System.out.println("Going to wait");
                mutex.wait();
            } else {

```



```

        }
    } else {
        System.out.println("Consumer");

        for (i = 0; i < max; i++) {
            try{
                int r = q.consume();
                System.out.println("Item: " + r);
            } catch (KeeperException e) {
                i--;
            } catch (InterruptedException e) {

            }
        }
    }
}

public static void barrierTest(String args[]) {
    Barrier b = new Barrier(args[1], "/b1", new Integer(args[2]));
    try{
        boolean flag = b.enter();
        System.out.println("Entered barrier: " + args[2]);
        if(!flag) System.out.println("Error when entering the barrier");
    } catch (KeeperException e) {

    }

    } catch (InterruptedException e) {

    }

    // Generate random integer
    Random rand = new Random();
    int r = rand.nextInt(100);
    // Loop for rand iterations
    for (int i = 0; i < r; i++) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {

        }
    }
    try{
        b.leave();
    } catch (KeeperException e) {

```

```
        } catch (InterruptedException e) {  
  
        }  
        System.out.println("Left barrier");  
    }  
}
```

2.5 ZooKeeper 方法和解决方案

- [使用 ZooKeeper 创建高级构造的指南](#)
 - [开箱即用应用程序：名称服务，配置，组成员资格](#)
 - [障碍](#)
 - [双重障碍](#)
 - [队列](#)
 - [优先级队列](#)
 - [锁定](#)
 - [共享锁](#)
 - [可恢复共享锁](#)
 - [两阶段提交](#)
 - [领导选举](#)

2.5.1 使用 ZooKeeper 创建高级构造的指南

在本文中，您将找到使用 ZooKeeper 实现更高阶功能的准则。所有这些都是在客户端实施的约定，不需要 ZooKeeper 的特别支持。希望社区能够在客户端库中捕获这些约定，以减轻其使用和鼓励标准化。

ZooKeeper 最有趣的一件事是即使 ZooKeeper 使用异步通知，也可以使用它来构建同步的一致性原语，比如队列和锁。如你所见，这是可能的，因为 ZooKeeper 对更新施加了一个整体的顺序，并且具有暴露这种排序的机制。

请注意，下面的配方尝试采用最佳做法。特别是，它们避免了轮询，计时器或其他任何会导致“群体效应”的事件，导致流量突发和限制可扩展性。

有许多有用的功能可以被想象为不包括在这里 – 可撤销的读写优先级锁，仅作为一个例子。而且这里提到的一些结构 – 特别是锁定 – 说明某些点，即使您可能会发现其他构造，例如事件句柄或队列，更实际的执行相同功能的方法。一般来说，本节中的例子旨在激发思想。

2.5.1.1 开箱即用应用程序：名称服务，配置，组成员资格

名称服务和配置是 ZooKeeper 的两个主要应用程序。这两个功能由 ZooKeeper API 直接提供。

ZooKeeper 直接提供的另一个功能是*组成员资格* 。 该组由一个节点表示。 组成员在组节点下创建临时节点。 当 ZooKeeper 检测到故障时，异常异常的成员节点将被自动删除。

2.5.1.2 障碍

分布式系统使用*障碍*来阻止对一组节点的处理，直到满足所有节点才允许进行的条件。 障碍通过指定一个障碍节点在 ZooKeeper 中实现。 如果屏障节点存在，屏障就位。 这是伪代码：

1. 客户端在屏障节点上调用 ZooKeeper API 的 **exists()** 函数，将 *watch* 设置为 true。
2. 如果 **exists()** 返回 false，则屏障消失，客户端继续进行
3. 否则，如果 **exists()** 返回 true，则客户端等待来自 ZooKeeper 的观察事件作为屏障节点。
4. 当监视事件被触发时，客户机重新发出 **exists()** 调用，再次等到屏障节点被移除。

2.5.1.2.1 双重障碍

双重障碍使客户端能够同步计算的开始和结束。 当足够的流程加入障碍物时，流程开始计算，一旦完成就离开障碍物。 该配方显示如何使用 ZooKeeper 节点作为屏障。

该配方中的伪代码表示屏障节点为 *b* 。 每个客户端进程 *p* 在进入时注册屏障节点，并在准备离开时注销。 节点通过下面的*输入*过程向屏障节点注册，它在进行计算之前等待直到 *x* 客户端进程寄存器。（这里的 *x* 取决于您为系统确定。）

输入	离开
<div>1. 创建一个名字 $n = b + "/" + p$</div> <div>2. 设置观察者(Watcher): exists (<i>b</i> +''/ ready'', true)</div> <div>3. 创建 child: create (<i>n</i> , EPHEMERAL)</div> <div>4. L = getChildren (b, false)</div> <div>5. 如果 L 中的孩子少于 <i>x</i> , 请等待观看事件</div> <div>6. else create (b +''/ ready'', REGULAR)</div>	<div>1. L = getChildren (b, false)</div> <div>2. 如果没有孩子, 退出</div> <div>3. 如果 <i>p</i> 只是 L 中的进程节点, 则删除 (n) 并退出</div> <div>4. 如果 <i>p</i> 是 L 中最低的进程节点, 则等待 L 中的最高进程节点</div> <div>5. else delete (<i>n</i>) 如果仍然存在并等待 L 中的最低进程节点</div> <div>6. goto 1</div>

在进入时，所有进程在可用节点上观察，并创建一个短暂节点作为障碍节点的子节点。 每个进程，但最后一个进入障碍，并等待就绪节点出现在第 5 行。创建第 *x* 个节点（最后一个进程）的进程将在子列表中看到 *x* 个节点，并创建就绪节点，唤醒其他流程。 请注意，等待进程只有在退出时才醒来，所以等待是有效的。

在退出时，您不能使用诸如*准备好的*标志，因为您正在观看进程节点的消失。 通过使用短暂节点，输入障碍物后失败的过程不会阻止正确的过程完成。 当进程准备离开时，他们需要删除其进程节点并等待所有其他进程执行相同操作。

当没有作为 *b* 的孩子的进程节点没有进程时退出。 但是，作为效率，您可以使用最低的进程节点作为就绪标志。 准备退出的所有其他进程观察最低的现有进程节点离开，并且最低进程的所有者

监视任何其他进程节点（为了简单起见，最高）为止消失。这意味着除了最后一个节点之外，除了最后一个节点之外，每个节点删除只有一个进程将被唤醒，这些节点在删除时唤醒每个节点。

2.5.1.3 队列

分布式队列是一种常见的数据结构。要在 ZooKeeper 中实现分布式队列，首先指定一个 znode 来保存队列节点。分布式客户端通过调用具有以“queue-”结尾的路径名称的 `create()`，将 `create()` 调用中的 *序列*和*临时*标志设置为 `true`，从而将其放入队列中。因为*序列*标志被设置，新的路径名将具有 `_path-to-queue-node_ / queue-X` 的形式，其中 X 是单调递增的数字。要从队列中删除的客户端会调用 ZooKeeper 的 `getChildren()` 函数，在队列节点上将 *watch* 设置为 `true`，并开始处理数量最少的节点。客户端不需要发出另一个 `getChildren()`，直到它耗尽从第一个 `getChildren()` 调用获取的列表。如果队列中没有子节点，则读者等待观察通知再次检查队列。

注意

在 ZooKeeper `recipes` 目录中现在存在一个 Queue 实现。这是与 `release-src / recipes / queue` 目录一起分发的。

2.5.1.3.1 优先级队列

要实现优先级队列，只需对通用队列配方进行两次简单更改即可。首先，要添加到队列中，路径名以“queue-YY”结尾，其中 YY 是具有较低数字的元素的优先级，表示较高优先级（就像 UNIX）。第二，当从队列中删除时，客户端使用最新的子列表，这意味着如果监视通知触发队列节点，则客户端将使先前获得的子列表无效。

2.5.1.4 锁定

完全分布的全局同步锁，意味着任何时间的快照，没有两个客户端认为它们持有相同的锁。这些可以使用 ZooKeeper 来实现。与优先级队列一样，首先定义一个锁定节点。

注意

现在在 ZooKeeper 食谱目录中存在一个 Lock 实现。这是与 `release-src / recipes / lock` 目录一起发布的。

希望获得锁的客户请执行以下操作：

1. 调用具有路径名为“`_locknode_ / lock-`”的 `create()`，并设置*序列*和*临时*标志。
2. 在锁节点上调用 `getChildren()` 而不设置观察者(Watcher)标志（这对于避免这种效应很重要）。
3. 如果在步骤 1 中创建的路径名具有最低的序列号后缀，则客户端具有锁定并且客户端退出协议。

4. 客户端调用 `exists()` ，其中 `watch` 标记在锁定目录中的路径上设置，具有下一个最低序列号。
5. 如果 `exists()` 返回 `false`，请转到步骤 2 。 否则，请等待上一步骤中的路径名通知，然后再转到步骤 2 。

解锁协议非常简单：希望释放锁的客户端只需删除在步骤 1 中创建的节点。

以下是一些注意事项：

- 删除节点只会导致一个客户端唤醒，因为每个节点只有一个客户端被监视。 这样你就可以避免群体效应。
- 没有投票或超时。
- 由于实现锁定的方式，很容易看到锁争用的数量，中断锁，调试锁定问题等。

2.5.1.4.1 共享锁

您可以通过对锁协议进行一些更改来实现共享锁：

获取读锁：

1. 调用 `create()` 创建一个路径名为 “ `_locknode_ / read-` ” 的节点。 这是协议中稍后使用的锁节点。 设置 *序列* 和 *短暂* 标志。
2. 在锁节点上调用 `getChildren()` ， 而不设置 *观察者(Watcher)* 标志 - 这很重要，因为它可以避免群体效应。
3. 如果没有一个路径名以 “ `write-` ” 开头并且序列号低于步骤 1 中创建的节点的子节点，客户端就有锁定并且协议。
4. 否则，使用 `watch` 标志调用 `exists()` ，在 `lock` 目录中的节点上设置路径名，并使用具有下一个最低序列号的 “`write-`” 。
5. 如果 `exists()` 返回 `false` ，则转到步骤 2 。
6. 否则，请等待上一步骤中的路径名通知，然后再转到步骤 2

注意

似乎这个配方可能会产生一个群体效应：当有一大群客户端等待读取锁定时，并且当删除具有最低序列号的 “ `写入` ” 节点时，所有这些都会同时得到或多或少的通知。 事实上。 这是有效的行为：因为所有等待读者客户端应该被释放，因为他们有锁。 畜群效应是指实际上只能使用一台或几台机器才能释放 “群” 。

2.5.1.4.2 可恢复共享锁

通过对 Shared Lock 协议进行微小修改，可以通过修改共享锁协议来使共享锁可撤销：

在步骤 1 中，既获得读写器锁定协议，也可以在调用 `create()` 之后立即调用 `getData()` 。 如果客户端随后在步骤 1 中为其创建的节点接收到通知，则在该节点上执行另一个 `getData()` ，并设置 `watch` 并查找字符串 “`unlock`”，该消息向客户端发出必须释放锁定的字符串。 这是因为根据这个共享锁协议，您可以通过在锁节点上调用 `setData()` 来向客户端请求锁放置锁定，将 “`unlock`” 写入该节点。

请注意，此协议要求锁持有人同意释放锁。 这种同意很重要，特别是如果锁定架需要在释放锁之前进行一些处理。 当然，您可以随时通过 *Freaking Laser Beams* 实施可撤销共享锁，通过在协议中规定允许撤销者删除锁定节点，如果在一段时间内锁定不被锁定夹删除。

2.5.1.5 两阶段提交

两阶段提交协议是一种算法，它允许分布式系统中的所有客户端同意提交事务或中止。

在 ZooKeeper 中，您可以通过协调器创建一个交易节点来实现两阶段提交，例如 `/ app / Tx` 和每个参与站点的一个子节点，例如 `/ app / Tx / s_i`。 协调器创建子节点时，将保留内容未定义。 一旦参与交易的每个站点从协调器接收到该事务，该站点将读取每个子节点并设置一个观察者 (Watcher)。 然后每个站点处理查询并通过写入其相应的节点投票“提交”或“中止”。 一旦完成写作，其他站点就会被通知，一旦所有站点都进行了所有的投票，他们就可以决定“中止”或“提交”。 请注意，如果某个站点投票为“中止”，节点可以提前决定“中止”。

该实现的一个有趣的方面是，协调器的唯一作用是决定站点组，创建 ZooKeeper 节点，并将事务传播到相应的站点。 事实上，即使传播事务也可以通过 ZooKeeper 将其写入事务节点来完成。

上述方法有两个重要的缺点。 一个是消息复杂度，这是 $O(n^2)$ 。 第二个是不可能通过短暂的节点检测站点的故障。 要使用临时节点检测站点的故障，站点创建节点是必要的。

为了解决第一个问题，您只能向协调器通知事务节点的更改，然后在协调器达成决定后通知站点。 请注意，这种方法是可扩展的，但它也较慢，因为它需要通过协调器进行所有通信。

为了解决第二个问题，您可以让协调者将事务传播到站点，并让每个站点创建自己的短暂节点。

2.5.1.6 领导选举

使用 ZooKeeper 进行领导选举的一种简单方法是在创建代表客户端的“提案”的 znodes 时使用 **SEQUENCE | EPHEMERAL** 标志。 这个想法是有一个 znode，说 `/选举`，这样每个 znode 都使用两个标志 **SEQUENCE | EPHEMERAL** 创建一个子节点 `/ election / n_`。 使用序列标记，ZooKeeper 会自动附加一个序列号，该序列号先前附加到 `/ election` 的子节点。 创建具有最小附加序列号的 znode 的过程是前导。

那不是全部的。 重要的是要注意领导的失败，这样一个新的客户就会成为当前领导者失败的新领导者。 一个微不足道的解决方案是让所有应用程序进程监视当前最小的 znode，并在最小的 znode 消失时检查它们是否是新的领导者（请注意，如果引导失败，则最小的 znode 会消失，因为节点是短暂的）。 但是这会产生一个群体效应：由于当前领导者的失败，所有其他进程都会收到通知，并执行 `getChildren / election` 以获取当前的 `/ election` 子项列表。 如果客户端数量庞大，则会导致 ZooKeeper 服务器必须处理的操作数量上升。 为了避免群体效应，在 znodes 序列上观察下一个 znode 就足够了。 如果一个客户端收到一个通知，表示它正在观看的 znode 已经消失，那么在没有较小的 znode 的情况下，它将成为新的领导者。 请注意，这不会让所有客户端都看到相同的 znode，从而避免了群体效应。

这是伪代码：

让选择成为应用程序的选择之路。 自愿成为领导者：

1. 使用序列和 EPHEMERAL 标志创建具有路径 “ELECTION / n_” 的 znode z；
2. 让 C 成为 “选择” 的孩子，我是 z 的序列号；
3. 观察 “ELECTION / n_j” 的变化，其中 j 是最大的序列号，使得 $j < i$ 和 n_j 是 C 中的 znode；

收到 znode 删除通知后：

1. 让 C 成为 ELECTION 的新一批子女；
2. 如果 z 是 C 中的最小节点，则执行 leader 过程；
3. 否则，请注意 “ELECTION / n_j” 上的更改，其中 j 是最大序列号，使得 $j < i$ 和 n_j 是 C 中的 znode；

请注意，在子列表中没有之前的 znode 的 znode 并不意味着该 znode 的创建者知道它是当前的领导者。 应用程序可能会考虑创建一个单独的 znode 来确认领导执行领导程序。

3 BookKeeper 入门指南

- 入门：设置 BookKeeper 来编写日志。
 - 先决条件
 - 下载
 - LocalBookKeeper
 - 建立书店
 - 设置 ZooKeeper
 - 例

3.1 入门：设置 BookKeeper 来编写日志。

本文档包含有助于您快速启动 BookKeeper 的信息。 它的目标主要是开发人员愿意尝试，并为简单的 BookKeeper 安装和简单的编程示例提供简单的安装说明。 有关进一步的编程细节，请参阅 BookKeeper 程序员指南 。

3.1.1 先决条件

请参阅“管理指南”中的“ 系统要求 ”。

3.1.2 下载

BookKeeper 与 ZooKeeper 一起分发。 要获得 ZooKeeper 发行版，请从其中一个 Apache 下载镜像下载最新的稳定版本。

3.1.3 LocalBookKeeper

在 org.apache.bookkeeper.util 下，您会发现一个名为 LocalBookKeeper.java 的 java 程序，可以让您在单个机器上运行 BookKeeper。从性能角度看，这远不是理想的，但该程序对于测试和教育目的都是有用的。

3.1.4 建立书店

如果你是大胆的，而不仅仅是在本地运行的东西，那么你需要在不同的服务器上运行预订。您需要至少三个书籍才能开始。

对于每个 bookie，我们需要执行如下命令：

```
java -cp。： ./ zookeeper- <version> -bookkeeper.jar: ./ zookeeper-  
<version> .jar \: lib / slf4j-api-1.6.1.jar: lib / slf4j-log4j12-1.6.1。 jar:  
lib / log4j-1.2.15.jar -Dlog4j.configuration = log4j.properties \  
org.apache.bookkeeper.proto.BookieServer 3181 127.0.0.1:2181 /  
path_to_log_device / \ / path_to_ledger_device /
```

“/ path_to_log_device /” 和 “/ path_to_ledger_device /” 是不同的路径。此外，3181 端口是一个 bookie 监听来自客户端的连接请求的端口。127.0.0.1:2181 是 ZooKeeper 服务器的主机名: 端口。在此示例中，独立的 ZooKeeper 服务器在端口 2181 上本地运行。如果我们有多个 ZooKeeper 服务器，则该参数将是与其对应的所有主机名: 端口值的逗号分隔列表。

3.1.5 设置 ZooKeeper

ZooKeeper 代表 BookKeeper 客户端和订阅者存储元数据。为了获得最小的 ZooKeeper 安装与 BookKeeper 配合使用，我们可以设置一个以独立模式运行的服务器。一旦我们使服务器运行，我们需要创建一些 znodes：

1. /分类帐
2. /分类帐/可用
3. 对于每个 bookie，我们添加一个 znode，使得 znode 的名称是机器名称和该书正在侦听的端口号的连接。例如，如果 bookie.foo.com 上运行的 bookie 正在侦听 3181 端口，那么我们添加一个 znode /ledgers/available/bookie.foo.com:3181 。

3.1.6 例

在下面的代码摘录中，我们：

1. 创建分类账
2. 写入分类帐；
3. 关闭分类帐；
4. 打开同一分类帐阅读；

```

        5. 从分类账中读取;
        6. 再次关闭分类帐;

        LedgerHandle lh = bkc.createLedger(ledgerPassword);
        ledgerId = lh.getId();
        ByteBuffer entry = ByteBuffer.allocate(4);

        for(int i = 0; i < 10; i++){
            entry.putInt(i);
            entry.position(0);
            entries.add(entry.array());
            lh.addEntry(entry.array());
        }

        lh.close();
        lh = bkc.openLedger(ledgerId, ledgerPassword);

        Enumeration<LedgerEntry> ls = lh.readEntries(0, 9);
        int i = 0;
        while(ls.hasMoreElements()){
            ByteBuffer origbb = ByteBuffer.wrap(
                entries.get(i++));

            Integer origEntry = origbb.getInt();
            ByteBuffer result = ByteBuffer.wrap(
                ls.nextElement().getEntry());

            Integer retrEntry = result.getInt();
        }
        lh.close();

```

3.2 BookKeeper 概述

- [BookKeeper 概述](#)
- [BookKeeper 介绍](#)
- [稍微详细一点...](#)
- [簿记员元素和概念](#)
- [簿记员初步设计](#)
- [簿记员元数据管理](#)
- [结算分类帐](#)

3.2.1 BookKeeper 介绍

BookKeeper 是可靠地记录记录流的复制服务。在 BookKeeper 中，服务器是“bookies”，日志流是“分类帐”，每个单元的日志（又称记录）都是“分类帐条目”。BookKeeper 设计可靠；存

储分类帐的服务器可能会崩溃，损坏数据，丢弃数据，但只要有足够的身份正常运行，整个服务的行为正常。

BookKeeper 的初始动机来自于 HDFS 的主题。 Namenodes 必须以可靠的方式记录操作，以便在崩溃的情况下进行恢复。 我们发现 BookKeeper 的应用程序远远超出了 HDFS。 基本上，任何需要附加存储的应用程序都可以用 BookKeeper 替换它们的实现。 BookKeeper 具有缩放服务器数量的优势。

在较高层次上，簿记客户端从客户端应用程序接收条目并将其存储到一组书籍中，并且拥有这样的服务有一些优点：

- 我们可以使用针对这样的服务进行了优化的硬件。 我们目前认为，这样一个系统只能用于磁盘 I / O;
- 我们可以有一个服务器池实现这样一个日志系统，并在多个服务器之间共享;
- 我们可以使用这样的池具有更高的复制制度，如果与其应用使用的硬件相比，必要的硬件更便宜，这是有道理的。

3.2.2 稍微详细一点...

BookKeeper 实现了高可用性的日志，并且它被设计为提前写入记录。 除了由于服务的复制性质而引起的高可用性外，由于条带化，它提供了高吞吐量。 当我们在集合的一个子集中写入条目并在可用的仲裁中旋转写入时，我们可以通过读取和写入的服务器数量增加吞吐量。 可扩展性是在这种情况下由于使用仲裁可能实现的属性。 其他复制技术，如状态机复制，不启用这样的属性。

应用程序首先在通过本地 BookKeeper 客户端实例写入预订之前创建分类帐。 创建分类帐后，BookKeeper 客户端将分类帐的元数据写入 ZooKeeper。 每个分类帐目前都有一个作家。 该作者必须执行一个关闭的分类帐操作，然后任何其他客户端才能读取它。 如果分类帐的作者没有正确关闭分类帐，因为例如，在有机会关闭分类帐之前它已经崩溃了，那么试图打开分类帐的下一个客户端会执行一个程序来恢复。 关闭一个分类帐基本上由写入到分类帐到 ZooKeeper 的最后一个条目组成，恢复过程只需找到正确写入的最后一个条目，并将其写入 ZooKeeper。

请注意，当尝试打开分类帐时，此恢复过程将自动执行，并且不需要明确的操作。 虽然两个客户端可能会尝试同时恢复分类帐，但只有一个将成功，第一个能够为分类帐创建关闭 znode。

3.2.3 簿记员元素和概念

BookKeeper 使用四个基本元素：

- **分类帐** : 分类帐是一系列条目，每个条目都是一个字节序列。 条目按顺序写入分类帐，最多一次。 因此，分类帐具有仅附加语义;
- **BookKeeper 客户端** : 客户端与 BookKeeper 应用程序一起运行，它使应用程序能够对分类帐执行操作，如创建分类帐和写入;
- **Bookie** : **Bookie** 是一个 BookKeeper 存储服务器。 Bookies 存储分类帐的内容。 对于任何给定的分类帐 L，我们称之为集合存储 L 的内容的组合。为了表现，我们在集合的每个订阅者上存储

只有分类帐的片段。 也就是说，当将条目写入分类帐时，我们会分条，以便将每个条目写入组合的小组。

- **元数据存储服务**：BookKeeper 需要元数据存储服务来存储与分类帐和可用的预订相关的信息。 我们目前使用 ZooKeeper 进行这样的任务。

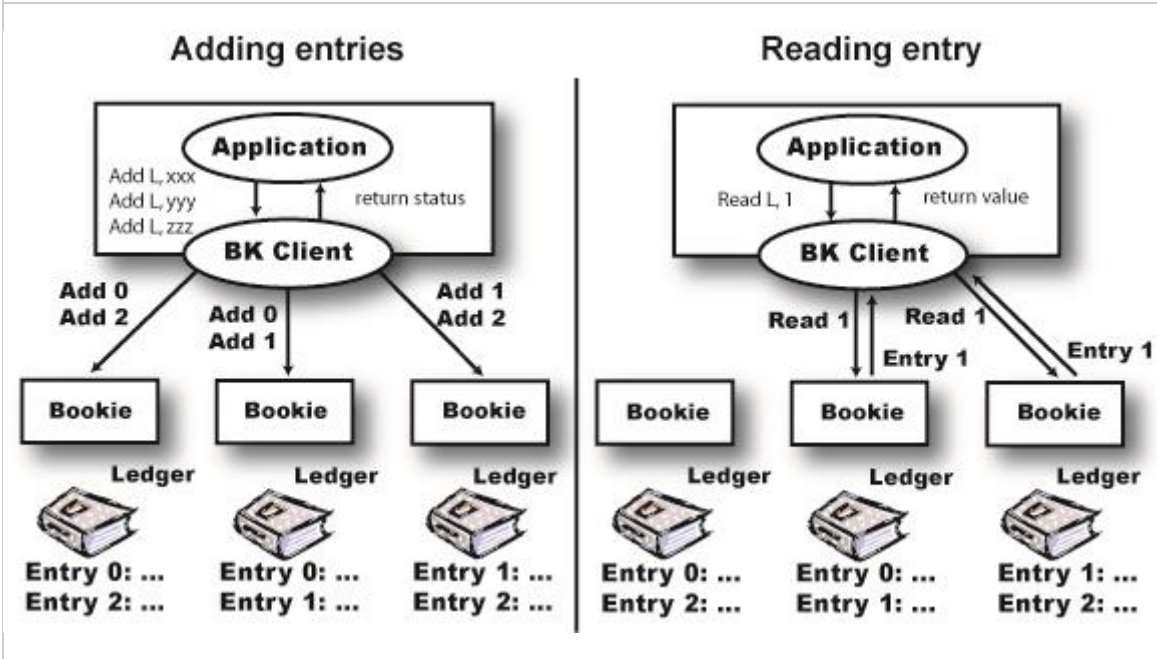
3.2.4 簿记员初步设计

一套 bookies 实现 BookKeeper，我们使用基于 Quorum 的协议来复制数据。 现有分类帐基本上有两个操作：读取和追加。 以下是完整的 API 列表（[模式详细信息](#)）：

- 创建分类帐：创建一个新的空白分类帐；
- 开立分类帐：打开现有的分类帐阅读；
- 添加条目：将记录同步或异步添加到分类帐；
- 阅读条目：从分类帐同步或异步读取一系列条目

只有一个客户端可以写入分类帐。 一旦该分类帐关闭或客户端出现故障，则不能添加更多条目。（我们利用这种行为提供强有力的保证。）分类帐中不会有差距。 手指破碎，书籍被操纵时，人们会变得粗暴或者被关在监狱，所以没有删除或改变条目。

BookKeeper 概述



BookKeeper 的简单使用是实现一个预先提前的事务日志。 服务器维护内存中数据结构（例如使用周期性快照），并在应用更改之前记录该结构的更改。 应用程序服务器在启动时创建分类帐，并将分类帐 ID 和密码存储在众所周知的地方（可能是 ZooKeeper）。 当需要进行更改时，服务器将更改信息的条目添加到分类帐，并在 BookKeeper 成功添加条目时应用更改。 服务器甚至可以使用 `asyncAddEntry` 排列许多更改以实现高更改吞吐量。 BookKeeper 仔细记录更改顺序，并按顺序调用完成功能。

当应用程序服务器死机时，备份服务器将上线，获取最后一个快照，然后它将打开旧服务器的分类帐，并从拍摄时间读取所有条目。（由于它不知道最后一个条目号，它将使用 MAX_INTEGER）。一旦所有的条目都被处理完成，它将关闭分类帐并开始一个新的使用。

一个客户端图书馆负责与书籍的沟通和管理条目号码。 条目具有以下字段：

输入字段

领域	类型	
帐号	长	此条目的分类帐的编号
条目编号	长	此条目的 ID
最后确认（ LC ）	长	最后记录的条目的 ID
数据	字节[]	条目数据（由应用程序提供）
验证码	字节[]	消息验证码，包括条目的所有其他字段

客户端库生成分类帐条目。 没有一个字段被书籍修改，只有前三个字段被书籍解释。

要添加到分类帐，客户端使用分类帐号生成上述条目。 条目号将比生成的最后一个条目多一个。 LC 字段包含 BookKeeper 成功记录的最后一个条目。 如果客户端一次写入一个条目，则 LC 是最后一个条目 ID。 但是，如果客户端正在使用 asyncAddEntry，则飞行中可能会有许多条目。 当满足以下两个条件时，考虑记录条目：

- 该条目已被法定人数接受
- 具有较低条目 ID 的所有条目已被法定人数接受

LC 现在似乎是神秘的，但现在解释我们如何使用它还时尚早； 只是微笑，继续前进。

一旦所有其他字段都已输入，客户端将生成一个包含所有以前字段的认证码。 然后将该条目发送到要记录的订单的法定人数。 任何失败都将导致入场券被发送到新的法定人数。

要阅读，客户端库最初联系一个 bookie 并开始请求条目。 如果条目丢失或无效（例如 MAC 错误），客户端将向另一个预订者发出请求。 通过使用仲裁写作，只要有足够的书签，我们将保证最终能够读取一个条目。

3.2.5 簿记员元数据管理

有一些元数据需要提供给 BookKeeper 客户端：

- 可用的书籍；
- 分类帐列表；
- 用于给定分类账的预订名单；
- 分类帐的最后一个条目；

我们在 ZooKeeper 中保留这些信息。Bookies 使用短暂节点来指示它们的可用性。客户端使用 znodes 来跟踪分类帐创建和删除，并且还知道分类帐的结尾和用于存储分类帐的预订。Bookies 还会看清分类帐列表，以便他们可以清理被删除的分类帐。

3.2.6 结算分类帐

由于 BookKeeper 的持久性保证，关闭分类帐和找到最后一个分类帐的过程很困难：

- 如果一个条目已被成功记录，它必须是可读的。
- 如果一个条目被读取一次，它必须始终可以被读取。

如果分类帐优雅地关闭，ZooKeeper 将会有最后一个条目，一切都会顺利进行。但是，如果正在编写分类帐的 BookKeeper 客户端死机，则需要进行一些恢复。

有问题的条目是分类帐末尾的条目。当 BookKeeper 客户端死机时，可能会有航班中的条目。如果条目只有一个 bookie，该条目不应该是可读的，因为条目将消失，如果该预订失败。如果该条目仅在一个簿册上，那并不意味着条目没有成功记录；记录条目的其他书签可能会失败。

使一切工作的诀窍是对最后一个条目有一个正确的想法。我们大致分三步做：

1. 查找具有最高记录条目 *LC* 的条目；
2. 找到连续记录的最高记录，*LR*；
3. 确保 *LC* 和 *LR* 之间的所有条目都在一个议价的数量上；

3.3 BookKeeper 管理员指南

- [部署](#)
 - [系统要求](#)
 - [运行的书](#)
 - [ZooKeeper 元数据](#)

3.3.1 部署

本节包含有关部署 BookKeeper 的信息，并涵盖以下主题：

- [系统要求](#)
- [运行的书](#)
- [ZooKeeper 元数据](#)

第一部分告诉你需要多少机器。第二个解释如何引导 bookies（BookKeeper 存储服务器）。第三部分解释了我们如何使用 ZooKeeper 和我们对 ZooKeeper 的要求。

3.3.2 系统要求

一个典型的 BookKeeper 安装包括一套书籍和一套 ZooKeeper 副本。 预订的确切数量取决于法定人数模式，所需吞吐量和同时使用此安装的客户端数量。 最小数量为 3，用于自我验证（存储消息认证码和每个条目），4 个用于通用（不存储每个条目的消息认证码字），并且没有上限的预订。 事实上，增加订单数量可以提高吞吐量。

对于性能，我们要求每个服务器至少有两个磁盘。 在这种情况下，可以运行带有单个磁盘的预订，但性能会显着降低。 当然，它适用于一个磁盘，但性能明显较低。

对于 ZooKeeper，对于副本的数量没有约束。 在独立模式下运行 ZooKeeper 的单台机器对于 BookKeeper 来说已经足够了。 出于弹性目的，运行 ZooKeeper 可能会在多个服务器的法定模式下运行。 有关如何使用多个副本配置 ZooKeeper 的详细信息，请参阅 ZooKeeper 文档

3.3.3 运行的书

要运行一个 bookie，我们执行以下命令：

```
java -cp。： ./ zookeeper- <version> -bookkeeper.jar: ./ zookeeper-  
<version> .jar \: ../ log4j / apache-log4j-1.2.15 / log4j-1.2.15.jar -  
Dlog4j.configuration = log4j.properties \  
org.apache.bookkeeper.proto.BookieServer 3181 127.0.0.1:2181 /  
path_to_log_device / \ / path_to_ledger_device /
```

参数为：

- 书店听的港口号码；
- 具有主机名：port 格式的 ZooKeeper 服务器的逗号分隔列表；
- 日志设备的路径（存储预订日志）；
- 分类帐设备路径（存储分类帐条目）；

理想情况下， / path_to_log_device / 和 / path_to_ledger_device / 都位于不同的设备中。

3.3.4 ZooKeeper 元数据

对于 BookKeeper，我们需要一个 ZooKeeper 安装来存储元数据，并将 ZooKeeper 服务器列表作为参数传递给 BookKeeper 类（ org.apache.bookkeeper.client, BookKeeper ）的构造函数。 要安装 ZooKeeper，请检查 [ZooKeeper 文档](#) 。

3.4 BookKeeper 程序编程入门

- [实例化 BookKeeper。](#)
- [创建分类帐](#)
- [将条目添加到分类帐。](#)

- [关闭分类帐](#)
- [打开分类帐](#)
- [从分类账中读](#)
- [删除分类帐](#)

3.4.1 实例化 BookKeeper。

使用 BookKeeper 的第一步是实例化一个 BookKeeper 对象：

```
org.apache.bookkeeper.BookKeeper
```

有三本 BookKeeper 构造函数：

```
public BookKeeper (String servers) 抛出 KeeperException, IOException
```

哪里：

- 服务器是逗号分隔的 ZooKeeper 服务器列表。

```
public BookKeeper (ZooKeeper zk) 抛出 InterruptedException, KeeperException
```

哪里：

- zk 是一个 ZooKeeper 对象。 当应用程序也使用 ZooKeeper 并且想要有一个 ZooKeeper 的实例时，这个构造函数很有用。

```
public BookKeeper (ZooKeeper zk, ClientSocketChannelFactory channelFactory) 抛出 InterruptedException, KeeperException
```

哪里：

- zk 是一个 ZooKeeper 对象。 当应用程序也使用 ZooKeeper 并且想要有一个 ZooKeeper 的实例时，这个构造函数很有用。
- channelFactory 是一个 netty 通道对象（ org.jboss.netty.channel.socket ）。

3.4.2 创建分类帐

在向 BookKeeper 写入条目之前，有必要创建一个分类帐。 使用当前的 BookKeeper API，可以同时或异步地创建分类帐。 以下方法属于 org.apache.bookkeeper.client.BookKeeper 。

同步通话

```
public LedgerHandle createLedger (int ensSize, int qSize, DigestType type, byte passwd []) throws KeeperException, InterruptedException, IOException, BKException
```

哪里：

- ensSize 是书籍的数量（合奏大小）；
- qSize 是写入仲裁大小；
- type 是与条目一起使用的摘要类型：MAC 或 CRC32。
- passwd 是授权客户端写入正在创建的分类帐的密码。

分类帐上的所有其他操作都将通过返回的 LedgerHandle 对象进行调用。

为了方便起见，我们提供一个默认参数（3, 2, VERIFIABLE）的 createLedger，它需要的只有两个输入参数是摘要类型和密码。

异步调用：

```
public void asyncCreateLedger (int ensSize, int qSize, DigestType type, byte
passwd [], CreateCallback cb, Object ctx)
```

参数与同步版本相同，但 cb 和 ctx 除外。 CreateCallback 是 org.apache.bookkeeper.client.AsyncCallback 中的接口，实现它的类必须实现一个名为 createComplete 的方法，该方法具有以下签名：

```
void createComplete (int rc, LedgerHandle lh, Object ctx) ;
```

哪里：

- rc 是一个返回代码（请参考一个列表的 org.apache.bookkeeper.client.BKException）；
- lh 是操纵分类帐的 LedgerHandle 对象；
- ctx 是用于问责的控制对象。它基本上可以是应用程序所满足的任何对象。

作为参数传递的 ctx 对象用于创建分类帐的调用是在回调中返回的一个。

3.4.3 将条目添加到分类帐。

一旦我们有一个通过调用创建分类帐获得的分类帐处理，我们可以开始写入条目。与创建分类帐一样，我们可以同步和异步地写入。以下方法属于 org.apache.bookkeeper.client.LedgerHandle。

同步通话

```
public long addEntry (byte [] data) throws InterruptedException
```

哪里：

- 数据是一个字节数组；

对 addEntry 的调用返回操作的状态（有关列表，请参阅 org.apache.bookkeeper.client.BKDefs）；

异步调用：

```
public void asyncAddEntry (byte [] data, AddCallback cb, Object ctx)
```

它还需要一个字节数组作为要作为条目存储的字节序列。 另外，它需要一个回调对象 `cb` 和一个控制对象 `ctx` 。 回调对象必须在 `org.apache.bookkeeper.client.AsyncCallback` 中实现 `AddCallback` 接口，实现它的类必须实现一个名为 `addComplete` 的方法，该方法具有以下签名：

```
void addComplete (int rc, LedgerHandle lh, long entryId, Object ctx) ;
```

哪里：

- `rc` 是一个返回码（请参考 `org.apache.bookkeeper.client.BKDefs` 列表）；
- `lh` 是操纵分类帐的 `LedgerHandle` 对象；
- `entryId` 是与此请求相关联的条目的标识符；
- `ctx` 是用于问责的控制对象。 它可以是应用程序所满足的任何对象。

3.4.4 关闭分类帐

一旦客户完成写作，它将关闭分类帐。 以下方法属于 `org.apache.bookkeeper.client.LedgerHandle` 。

同步关闭：

```
public void close () throws InterruptedException
```

它不需要输入参数。

异步关闭：

```
public void asyncClose (CloseCallback cb, Object ctx) throws  
InterruptedException
```

它需要一个回调对象 `cb` 和一个控制对象 `ctx` 。 回调对象必须在 `org.apache.bookkeeper.client.AsyncCallback` 中实现 `CloseCallback` 接口，实现它的类必须实现一个名为 `closeComplete` 的方法，该方法具有以下签名：

```
void closeComplete (int rc, LedgerHandle lh, Object ctx)
```

哪里：

- `rc` 是一个返回码（请参考 `org.apache.bookkeeper.client.BKDefs` 列表）；
- `lh` 是操纵分类帐的 `LedgerHandle` 对象；
- `ctx` 是用于问责的控制对象。

3.4.5 打开分类帐

要从分类账中读取，客户必须先打开它。 以下方法属于 `org.apache.bookkeeper.client.BookKeeper` 。

同步打开:

```
public LedgerHandle openLedger (long lId, DigestType type, byte passwd [])  
throws InterruptedException, BKException
```

- 分类帐 ID 是分类帐标识符;
- type 是与条目一起使用的摘要类型: MAC 或 CRC32。
- passwd 是访问分类帐的密码 (仅在 VERIFIABLE 分类帐的情况下使用);

异步打开:

```
public void asyncOpenLedger (long lId, DigestType type, byte passwd [],  
OpenCallback cb, Object ctx)
```

它还需要一个分类帐标识符和一个密码。 另外, 它需要一个回调对象 cb 和一个控制对象 ctx 。 回调对象必须在 org.apache.bookkeeper.client.AsyncCallback 中实现 OpenCallback 接口 , 实现它的类必须实现一个名为 openComplete 的方法, 该方法具有以下签名:

```
public void openComplete (int rc, LedgerHandle lh, Object ctx)
```

哪里:

- rc 是一个返回码 (请参考 org.apache.bookkeeper.client.BKDefs 列表);
- lh 是操纵分类帐的 LedgerHandle 对象;
- ctx 是用于问责的控制对象。

3.4.6 从分类账中读

读取呼叫可以请求一个或多个连续条目。 以下方法属于 org.apache.bookkeeper.client.LedgerHandle 。

同步阅读:

```
public Enumeration <LedgerEntry> readEntries (long firstEntry, long lastEntry)  
throws InterruptedException, BKException
```

- firstEntry 是要读取的条目序列中的第一个条目的标识符;
- lastEntry 是要读取的条目序列中最后一个条目的标识符。

异步读取:

```
public void asyncReadEntries (long firstEntry, long lastEntry, ReadCallback  
cb, Object ctx) throws BKException, InterruptedException
```

它还需要第一个和最后一个条目标识符。 另外, 它需要一个回调对象 cb 和一个控制对象 ctx 。 回调对象必须在 org.apache.bookkeeper.client.AsyncCallback 中实现

ReadCallback 接口，实现它的类必须实现一个名为 readComplete 的方法，该方法具有以下签名：

```
void readComplete (int rc, LedgerHandle lh, Enumeration <LedgerEntry> seq,
Object ctx)
```

哪里：

- rc 是一个返回码（请参考 org.apache.bookkeeper.client.BKDefs 列表）；
- lh 是操纵分类帐的 LedgerHandle 对象；
- seq 是枚举<LedgerEntry>对象，用于包含请求的条目列表；
- ctx 是用于问责的控制对象。

3.4.7 删除分类帐

一旦客户完成了分类帐，并且确定没有人会再次需要读取，他们可以删除分类帐。以下方法属于 org.apache.bookkeeper.client.BookKeeper。

同步删除：

```
public void deleteLedger (long lId) throws InterruptedException, BKException
```

- lId 是分类帐标识符；

异步删除：

```
public void asyncDeleteLedger (long lId, DeleteCallback cb, Object ctx)
```

它需要一个分类帐标识符。另外，它需要一个回调对象 cb 和一个控制对象 ctx。回调对象必须在 org.apache.bookkeeper.client.AsyncCallback 中实现 DeleteCallback 接口，实现它的类必须实现一个名为 deleteComplete 的方法，该方法具有以下签名：

```
void deleteComplete (int rc, Object ctx)
```

哪里：

- rc 是一个返回码（请参考 org.apache.bookkeeper.client.BKDefs 列表）；
- ctx 是用于问责的控制对象。

4 管理和部署

4.1 ZooKeeper 管理员指南：部署和管理指南

- 部署
 - 系统要求
 - 支持的平台
 - 所需软件

- 群集（多服务器）设置
- 单服务器和开发人员设置
- 行政
- 设计 ZooKeeper 部署
- 跨机器要求
- 单机要求
- 配置
- 需要考虑的事项：ZooKeeper 的优点和局限性
- 管理
- 保养
- 正在进行的数据目录清理
- 调试日志清理（log4j）
- 监督
- 监控
- 记录
- 故障排除
- 配置参数
- 最低配置
- 高级配置
- 群集选项
- 认证和授权选项
- 实验选项/功能
- 不安全的选项
- 使用 Netty 框架进行沟通
- ZooKeeper 命令：四个字母的单词
- 数据文件管理
- 数据目录
- 日志目录
- 文件管理
- 要避免的事情
- 最佳做法

4. 1. 1 部署

本节包含有关部署 Zookeeper 的信息，并涵盖以下主题：

- 系统要求
- 群集（多服务器）设置
- 单服务器和开发人员设置

前两节假设您有兴趣在生产环境（如数据中心）中安装 ZooKeeper。最后一节将介绍如何在有限的基础上设置 ZooKeeper - 用于评估，测试或开发 - 但不在生产环境中。

4.1.1.1 系统要求

4.1.1.1.1 支持的平台

ZooKeeper 由多个组件组成。一些组件被广泛支持，其他组件仅在较小的一组平台上受支持。

- **客户端**是 Java 客户端库，由应用程序用于连接到 ZooKeeper 集合。
- **服务器**是在 ZooKeeper 集合节点上运行的 Java 服务器。
- **Native Client** 是 C 中实现的客户端，类似于 Java 客户端，由应用程序用于连接到 ZooKeeper 系统。
- **Contrib** 是指多个可选的附加组件。

以下矩阵描述了在不同操作系统平台上运行每个组件的承诺级别。

支持矩阵		
操作系统	客户	
GNU / Linux	发展与生产	发展与生产
Solaris	发展与生产	发展与生产
FreeBSD	发展与生产	发展与生产
视窗	发展与生产	发展与生产
Mac OS X	只有发展	只有发展

对于没有明确提到的矩阵中支持的操作系统，组件可能会或可能不起作用。ZooKeeper 社区将修复为其他平台报告的明显错误，但没有完全支持。

4.1.1.1.2 所需软件

ZooKeeper 在 Java 中运行，1.6 或更高版本（JDK 6 或更高版本）。它作为 ZooKeeper 服务器的集合运行。三个 ZooKeeper 服务器是组合的最小推荐尺寸，我们还建议它们在不同的机器上运行。在 Yahoo !，ZooKeeper 通常部署在专用的 RHEL 盒子上，配备双核处理器，2GB 内存和 80GB IDE 硬盘。

4.1.1.2 群集（多服务器）设置

为了可靠的 ZooKeeper 服务，您应该将 ZooKeeper 部署在称为合奏的群集中。只要大部分合奏成功，该服务将可用。因为 Zookeeper 需要多数，最好使用奇数机器。例如，使用四台机器，ZooKeeper 只能处理单台机器的故障；如果两台机器发生故障，剩下的两台机器不构成多数。但是，有五台机器，ZooKeeper 可以处理两台机器的故障。

注意

如“[ZooKeeper 入门指南](#)”所述，容错群集设置至少需要三台服务器，强烈建议您使用奇数个服务器。

通常，三台服务器足够用于生产安装，但为了在维护期间获得最大可靠性，您可能希望安装五台服务器。使用三台服务器，如果在其中一台服务器上进行维护，那么在维护期间，其他两台服务器之间的故障将很容易出现。如果你有五个运行，你可以采取一个维护，并且知道如果其他四个突然失败，你仍然可以。

您的冗余考虑应包括您的环境的所有方面。如果您有三个 ZooKeeper 服务器，但是它们的网络电缆都插入同一个网络交换机，则该交换机的故障将占用整个系统。

以下是设置将成为集合的一部分的服务器的步骤。这些步骤应该在集合中的每个主机上执行：

1. 安装 Java JDK。 您可以使用系统的本地打包系统，或者从以下位置下载 JDK：
<http://java.sun.com/javase/downloads/index.jsp>
2. 设置 Java 堆大小。 这对避免交换非常重要，这将严重降低 ZooKeeper 的性能。 要确定正确的值，请使用负载测试，并确保您远远低于将导致交换的使用限制。 保守 - 对于 4GB 机器，最大堆大小为 3GB。
3. 安装 ZooKeeper 服务器包。 它可以从下载：
<http://zookeeper.apache.org/releases.html>
4. 创建一个配置文件。 这个文件可以调用任何东西。 使用以下设置作为起点：
5. `tickTime = 2000`
6. `dataDir = / var / lib / zookeeper /`
7. `clientPort = 2181`
8. `initLimit = 5`
9. `syncLimit = 2`
10. `server.1 = zoo1: 2888: 3888`
11. `server.2 = zoo2: 2888: 3888`
`server.3 = zoo3: 2888: 3888`

您可以在配置参数一节中找到这些和其他配置设置的含义。 一个字，虽然在这里几个：作为 ZooKeeper 系列的一部分的每台机器都应该了解该系列中的其他所有机器。 您可以使用表单 **server.id = host: port: port** 的一系列行完成此操作。 **主机和端口的参数** 很简单。 您可以通过为配置文件参数 **dataDir** 指定的名为 myid 的文件来创建一个名为 myid 的文件，每个服务器位于该服务器的数据目录中。

12. myid 文件由仅包含该机器 ID 的文本的一行组成。 所以服务器 1 的 myid 将包含文本“1”，没有别的。 该集合中的 id 必须是唯一的，并且应该具有 1 到 255 之间的值。
13. 如果您的配置文件已设置，您可以启动 ZooKeeper 服务器：

```
$ 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 服务器， [JMX](#) 管理 bean 也被注册，允许通过 JMX 管理控制台进行管理。 [ZooKeeper JMX 文档](#) 包含有关使用 JMX 管理 ZooKeeper 的详细信息。 有关启动服务器实例的示例，请参阅发行版中包含的脚本 `bin / zkServer.sh`。
14. 通过连接到主机来测试您的部署：
在 Java 中，您可以运行以下命令来执行简单的操作：

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

4.1.1.3 单服务器和开发人员设置

如果要将 ZooKeeper 设置为开发目的，您可能需要设置一个单一的 ZooKeeper 服务器实例，然后在开发机器上安装 Java 或 C 客户端库和绑定。

设置单个服务器实例的步骤与上述类似，但配置文件更简单。您可以在“[ZooKeeper 入门指南](#)”的“[单服务器模式下安装和运行 ZooKeeper](#)”一节中找到完整的说明。

有关安装客户端库的信息，请参阅“[ZooKeeper 程序员指南](#)”中的“[绑定](#)”部分。

4.1.2 管理

本节包含有关运行和维护 ZooKeeper 的信息，并涵盖以下主题：

- [设计 ZooKeeper 部署](#)
- [配置](#)
- [需要考虑的事项：ZooKeeper 的优点和局限性](#)
- [管理](#)
- [保养](#)
- [监督](#)
- [监控](#)
- [记录](#)
- [故障排除](#)
- [配置参数](#)
- [ZooKeeper 命令：四个字母的单词](#)
- [数据文件管理](#)
- [要避免的事情](#)
- [最佳做法](#)

4.1.2.1 设计 ZooKeeper 部署

ZooKeeper 的可靠性取决于两个基本假设。

1. 部署中只有少数的服务器将失败。在这种情况下失败意味着机器崩溃，或网络中的一些错误，从而将服务器与多数服务器分开。
2. 部署的机器正常运行。正确操作意味着正确执行代码，使时钟能够正常工作，并使存储和网络组件始终如一地执行。

以下部分包含 ZooKeeper 管理员的考虑因素，以最大限度地提高这些假设的概率。其中一些是跨机器注意事项，其他一些是您应该为部署中的每台机器考虑的事情。

4.1.2.2 跨机器要求

要使 ZooKeeper 服务处于活动状态，必须存在大多数可以相互通信的非故障机器。要创建可以容忍 F 机故障的部署，您应该依靠部署 $2xF + 1$ 机器。因此，由三台机器组成的部署可以处理一个故障，并且部署五台机器可以处理两个故障。请注意，六台机器的部署只能处理两个故障，因为三台机器不是多数机器。因此，ZooKeeper 的部署通常由奇数机器组成。

为了达到容忍故障的最高概率，您应该尽量使机器故障独立。例如，如果大多数机器共享相同的交换机，则该交换机的故障可能导致相关故障并降低服务。共享电源电路，冷却系统等也是如此。

4.1.2.3 单机要求

如果 ZooKeeper 必须与其他应用程序争取访问存储媒体，CPU，网络或内存等资源，其性能将会显著下降。ZooKeeper 具有强大的耐用性保证，这意味着它将使用存储介质来记录更改，然后才允许负责更改的操作完成。您应该知道这种依赖性，如果您想要确保 ZooKeeper 操作不被您的媒体所阻止，请非常小心。以下是您可以采取的一些措施，尽量减少这种退化：

- ZooKeeper 的事务日志必须在专用设备上。（专用分区不够。）ZooKeeper 顺序地写入日志，而不用寻求与其他进程共享您的日志设备可能导致搜索和争用，从而导致多秒延迟。
- 不要将 ZooKeeper 放在可能导致交换的情况下。为了使 ZooKeeper 能够以任何时间的方式运行，根本无法进行交换。因此，确保给 ZooKeeper 的最大堆大小不大于 ZooKeeper 可用的实际内存量。有关更多信息，请参阅下面的事项。

4.1.2.4 配置

4.1.2.5 需要考虑的事项：ZooKeeper 的优点和局限性

4.1.2.6 管理

4.1.2.7 保养

一个 ZooKeeper 集群需要很长时间的维护，但是您必须注意以下事项：

4.1.2.7.1 正在进行的数据目录清理

ZooKeeper 数据目录包含由特定服务集合存储的 znodes 的持久副本的文件。这些是快照和事务日志文件。随着对 znode 的更改，这些更改将附加到事务日志中，有时，当日志增长时，所有 znode 的当前状态的快照将被写入文件系统。此快照取代所有以前的日志。

使用默认配置时，ZooKeeper 服务器不会删除旧的快照和日志文件（请参见下面的 autopurge），这是操作员的责任。每个服务环境都不同，因此管理这些文件的要求可能因安装（例如备份）而异。

PurgeTxnLog 实用程序实现了管理员可以使用的简单的保留策略。[API 文档](#)包含有关调用约定（参数等）的详细信息。

在以下示例中，最后计数快照及其对应的日志将被保留，其他日志将被删除。通常，<count>的值应大于 3（尽管不是必需的，但是在不太可能的情况下，最近的日志已经损坏，因此提供了 3 个备份）。这可以作为 ZooKeeper 服务器计算机上的 cron 作业运行，以便每天清理日志。

```
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. PurgeTxnLog <dataDir> <snapDir> -n
<count>
```

在版本 3.4.0 中引入了自动清除快照和相应的事务日志，可以通过以下配置参数 **autopurge.snapRetainCount** 和 **autopurge.purgeInterval** 启用。有关更多信息，请参阅下面的[高级配置](#)。

4.1.2.7.2 调试日志清理（log4j）

请参阅登录本文档的部分。预计您将使用内置的 log4j 功能来设置滚动文件追加程序。release tar 的 conf / log4j.properties 中的示例配置文件提供了一个示例。

4.1.2.8 监督

您将需要一个管理每个 ZooKeeper 服务器进程（JVM）的监控进程。ZK 服务器设计为“快速失败”，意味着如果出现无法恢复的错误，它将关闭（进程退出）。作为一个 ZooKeeper 服务集群是高可靠性的，这意味着服务器可能会下降，整个群集仍然是活动的并且提供服务请求。此外，由于集群是“自我修复”，故障服务器一旦重新启动，将自动重新加入该组合以进行任何手动交互。

管理进程如 [daemontools](#) 或 [SMF](#)（其他监控过程的选项也可以使用，由您自己决定使用哪一个，这只是两个例子），管理您的 ZooKeeper 服务器可确保如果进程异常退出，将自动重新启动，并将快速重新加入群集。

4.1.2.9 监控

ZooKeeper 服务可以通过两种主要方式进行监控：1) 命令端口通过使用 [4 个字母的单词](#) 和 2) [JMX](#)。请参阅适用于您的环境/要求的部分。

4.1.2.10 记录

ZooKeeper 使用 **log4j** 版本 1.2 作为其日志记录基础设施。 ZooKeeper 默认 `log4j.properties` 文件驻留在 `conf` 目录中。 Log4j 要求 `log4j.properties` 位于工作目录（运行 ZooKeeper 的目录）或可从类路径访问。

有关详细信息，请参阅 log4j 手册的 [Log4j 默认初始化过程](#)。

4.1.2.11 故障排除

由于文件损坏，服务器不会出现

由于 ZooKeeper 服务器的事务日志中的某些文件损坏，服务器可能无法读取其数据库并无法启动。 您将在加载 ZooKeeper 数据库时看到一些 `IOException`。 在这种情况下，请确保您的系统中的所有其他服务器正常工作。 在命令端口使用“`stat`”命令，看看它们是否健康。 验证所有其他集成服务器已启动后，您可以继续清理损坏的服务器数据库。 删除 `datadir / version-2` 和 `datalogdir / version-2` 中的所有文件。 重新启动服务器。

4.1.2.12 配置参数

ZooKeeper 的行为由 ZooKeeper 配置文件管理。 该文件的设计使得假设磁盘布局相同，构成 ZooKeeper 服务器的所有服务器都可以使用完全相同的文件。 如果服务器使用不同的配置文件，则必须注意确保所有不同配置文件中的服务器列表相匹配。

4.1.2.12.1 最低配置

以下是配置文件中必须定义的最低配置关键字：

`clientPort`

端口监听客户端连接； 即客户端尝试连接的端口。

`dataDir`

ZooKeeper 将存储内存数据库快照的位置，除非另有说明，更新数据库的事务日志。

注意

把事务日志放在哪里。 专用的事务日志设备是一致的良好性能的关键。 将日志放在繁忙的设备上会不利地影响性能。

`tickTime`

单个刻度的长度，它是 ZooKeeper 使用的基本时间单位，以毫秒为单位。它用于调节心跳和超时。例如，最小会话超时将是两个 ticks。

4.1.2.12.2 高级配置

该部分中的配置设置是可选的。您可以使用它们来进一步微调 ZooKeeper 服务器的行为。一些也可以使用 Java 系统属性设置，一般是 `zookeeper.keyword` 的形式。确切的系统属性（如果可用）如下所示。

`dataLogDir`

（无 Java 系统属性）

该选项将指示机器将事务日志写入 `dataLogDir` 而不是 `dataDir`。这允许使用专用日志设备，并且有助于避免日志记录和快照之间的竞争。

注意

拥有专用的日志设备对吞吐量和稳定延迟有很大的影响。强烈推荐使用日志设备，并将 `dataLogDir` 设置为指向该设备上的目录，然后确保将 `dataDir` 指向不在该设备上的目录。

`globalOutstandingLimit`

（Java 系统属性：`zookeeper.globalOutstandingLimit`。）

客户端可以比 ZooKeeper 更快地提交请求，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 将会限制客户端，使系统中没有超过 `globalOutstandingLimit` 未完成的请求。默认值为 1,000。

`preAllocSize`

（Java 系统属性：`zookeeper.preAllocSize`。）

为了避免查找，ZooKeeper 将以 `preAllocSize` 千字节的块分配事务日志文件中的空间。默认块大小为 64M。更改块大小的一个原因是如果更快地拍摄快照，则减小块大小。（另请参见 `snapCount`）。

`snapCount`

（Java 系统属性：`zookeeper.snapCount`。）

ZooKeeper 将事务记录到事务日志。将 `snapCount` 事务写入日志文件后，将启动快照并创建新的事务日志文件。默认 `snapCount` 为 100,000。

`maxClientCnxns`

（无 Java 系统属性）

限制由 IP 地址标识的单个客户端可能对 ZooKeeper 系列的单个成员进行的并发连接数（在套接字级别）。这用于防止某些类别的 DoS 攻击，包括文件描述符耗尽。默认值为 60。将此设置为 0 将完全删除并发连接的限制。

clientPortAddress

3.3.0 中的新功能：用于监听客户端连接的地址（ipv4, ipv6 或 hostname）；也就是客户端尝试连接的地址。这是可选的，默认情况下，我们绑定的方式是任何连接到 **clientPort** 的任何地址/接口/ nic 在服务器将被接受。

minSessionTimeout

（无 Java 系统属性）

3.3.0 中的新功能：服务器允许客户端协商的最小会话超时（以毫秒为单位）。默认为 **tickTime** 的 2 倍。

maxSessionTimeout

（无 Java 系统属性）

3.3.0 中的新功能：服务器将允许客户端协商的最大会话超时（以毫秒为单位）。默认为 **tickTime** 的 20 倍。

fsync.warningthresholdms

（Java 系统属性：**zookeeper.fsync.warningthresholdms**）

3.3.4 中的新功能：当事务日志（WAL）中的 fsync 需要比此值更长时，将向日志输出警告消息。这些值以毫秒为单位指定，默认值为 1000。该值只能设置为系统属性。

autopurge.snapRetainCount

（无 Java 系统属性）

3.4.0 中的新功能：启用后，ZooKeeper 自动清除功能将分别保留在 dataDir 和 dataLogDir 中的 autopurge.snapRetainCount 最新快照和相应的事务日志，并删除其余的。默认为 3。最小值为 3。

autopurge.purgeInterval

（无 Java 系统属性）

3.4.0 中的新增：必须触发清除任务的时间间隔（以小时为单位）。设置为正整数（1 及以上）以启用自动清除。默认为 0。

syncEnabled

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

3.4.6, 3.5.0 中的**新功能**: 观察者现在记录事务, 默认情况下将写入快照写入磁盘, 与参与者一样。 这样可以减少重启时观察者的恢复时间。 设置为“false”以禁用此功能。 默认为“true”

4.1.2.12.3 群集选项

本节中的选项设计用于与服务器集成 - 即部署服务器群集时使用。

选举

(无 Java 系统属性)

选举执行使用。 值为“0”对应于原始的基于 UDP 的版本, “1”对应于非认证的基于 UDP 的快速前导选举版本, “2”对应于基于认证的快速领导选举的基于 UDP 的版本, “3”对应于基于 TCP 的快速领导选举版本。 目前, 算法 3 是默认值

注意

领导选举 0, 1 和 2 的执行现在已被**弃用**。 我们有意在下一个版本中删除它们, 此时只有 FastLeaderElection 可用。

initLimit

(无 Java 系统属性)

时间量 (以 tick 为单位) (请参阅 [tickTime](#)), 以允许关注者连接并同步到领导者。 如果 ZooKeeper 管理的数据量较大, 则根据需要增加此值。

leaderServes

(Java 系统属性: `zookeeper.leaderServes`)

领导接受客户端连接。 默认值为“yes”。 领导机器协调更新。 为了更高的更新吞吐量, 在读取吞吐量方面, 领导者可以配置为不接受客户端并专注于协调。 该选项的默认值为 yes, 这意味着领导者将接受客户端连接。

注意

强烈建议您在集合中拥有三个以上的 ZooKeeper 服务器时, 请务必重新启动首选项。

`server.x = [hostname]:nnnnn [:nnnnn]`等

(无 Java 系统属性)

服务器组成 ZooKeeper 合奏。 当服务器启动时, 它通过在数据目录中查找文件 `myid` 来确定哪个服务器。 该文件包含服务器编号, 以 ASCII 格式, 并且应该在此设置左侧的 **server.x** 中匹配 **x** 。

组成 ZooKeeper 服务器的客户端使用的服务器列表必须与每个 ZooKeeper 服务器的 ZooKeeper 服务器列表相匹配。

有两个端口号 `nnnnn` 。 第一个追随者用来连接领导者，第二个是领导人的选举。 领导选举端口只有在选举 A1 是 1, 2 或 3（默认）时才需要。 如果选择 A1 为 0，则不需要第二个端口。 如果要在单台机器上测试多台服务器，则可以为每台服务器使用不同的端口。

`syncLimit`

（无 Java 系统属性）

时间量（以 tick 为单位）（见 `tickTime` ），允许关注者与 ZooKeeper 同步。 如果追随者落后于领导者，他们将被丢弃。

`group.x = nnnnn [: nnnnn]`

（无 Java 系统属性）

启用分层仲裁结构。“x”是组标识符，“=”号后面的数字对应于服务器标识符。 作业的左侧是一个冒号分隔的服务器标识符列表。 请注意，组必须是不相交的，并且所有组的联合必须是 ZooKeeper 集合。

你会在[这里](#)找到一个例子

`weight.x = nnnnn`

（无 Java 系统属性）

与“组”一起使用时，会在形成仲裁时为服务器分配权重。 这样的值对应于投票时服务器的权重。 ZooKeeper 有几个部分需要投票，如领导选举和原子广播协议。 默认情况下，服务器的权重为 1. 如果配置定义了组而不是权重，那么将为所有服务器分配值 1。

你会在[这里](#)找到一个例子

`cnxTimeout`

（Java 系统属性：`zookeeper.cnxTimeout` ）

设置用于打开连接的超时值，用于领导选举通知。 只适用于您使用 `electAlg 3`。

注意

默认值为 5 秒。

`4lw.commands.whitelist`

（Java 系统属性：`zookeeper.4lw.commands.whitelist` ）

3.4.10 中的新功能：此属性包含逗号分隔的四字母命令的列表。 引入了对 ZooKeeper 可以执行的命令集进行细粒度控制，所以用户可以根据需要关闭某些命令。 默认情况下，如果没有指定属性，则它包含除“wchp”和“wche”之外的所有支持的四个字母的单词命令。 如果指定了该属性，那么仅启用白名单中列出的命令。

以下是禁用四字母命令的其余部分的 stat, ruok, conf 和 isro 命令的配置示例:

```
4lw.commands.whitelist = stat, ruok, conf,  
isro
```

用户也可以使用星号选项, 因此不必在列表中逐个包含每个命令。 例如, 这将使所有四个字母的单词命令:

```
4lw.commands.whitelist = *
```

4.1.2.12.4 认证和授权选项

本节中的选项允许控制由服务执行的身份验证/授权。

zookeeper.DigestAuthenticationProvider.superDigest

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

默认情况下, 此功能被**禁用**

3.2 中的新功能: 使 ZooKeeper 系统管理员能够以“超级”用户身份访问 znode 层次结构。 特别地, 对于被认证为超级用户的用户, 不发生 ACL 检查。

org.apache.zookeeper.server.auth.DigestAuthenticationProvider 可用于生成 superDigest, 并使用“super: <password>”的一个参数进行调用。 在启动集合的每个服务器时, 提供生成的“super: <data>”作为系统属性值。

当向 ZooKeeper 服务器 (从 ZooKeeper 客户端) 进行身份验证时, 会传递“digest”和 authdata “super: <password>”的方案。 请注意, digest auth 将 authdata 以明文形式传递到服务器, 只有在本地主机 (而不是通过网络) 或通过加密连接使用此验证方法将是谨慎的。

isro

3.4.0 中的新功能: 测试服务器是否以只读模式运行。 如果处于只读模式, 则服务器将以“ro”响应, 如果不是只读模式, 则将“rw”响应。

gtmk

获取当前跟踪掩码为十进制格式的 64 位带符号长整型值。 请参阅 stmk 了解可能的值。

stmk

设置当前跟踪掩码。 跟踪掩码为 64 位, 其中每个位启用或禁用服务器上的特定类别的跟踪记录。 必须将 Log4J 配置为首先启用 TRACE 级别才能查看跟踪记录消息。 跟踪掩码的位对应于以下跟踪日志记录类别。

跟踪掩码位值	
0b0000000000	未使用，保留供将来使用。
0b0000000010	记录客户端请求，不包括 ping 请求。
0b0000000100	未使用，保留供将来使用。
0b0000001000	记录客户端 ping 请求。
0b0000010000	记录从当前领导者的仲裁对等体接收的数据包，不包括 ping 请求。
0b0000100000	记录客户端会话的添加，删除和验证。
0b0001000000	记录观看事件的交付到客户端会话。
0b0010000000	记录从当前领导者的仲裁对等体接收的数据包。
0b0100000000	未使用，保留供将来使用。
0b1000000000	未使用，保留供将来使用。

64 位值中的所有剩余位都未使用，并保留供将来使用。 通过计算记录值的按位 OR 来指定多个跟踪记录类别。 默认跟踪掩码为 0b0100110010。 因此，默认情况下，跟踪日志记录包括客户端请求，从领导和会话接收的数据包。

要设置不同的跟踪掩码，请发送一个包含 stmk 四字母字的请求，后跟跟踪掩码表示为 64 位带符号的长整型值。 此示例使用 Perl 包函数构建一个跟踪掩码，该跟踪掩码启用上述所有跟踪日志记录类别，并将其转换为具有大字节顺序的 64 位带符号长整型值。 结果附加到 stmk 并使用 netcat 发送到服务器。 服务器以十进制格式响应新的跟踪掩码。

```
$ perl -e "print 'stmk', pack('q>', 0b0011111010)" | nc localhost 2181
250
```

4.1.2.12.5 实验选项/功能

目前被认为是实验性的新功能。

只读模式服务器

(Java 系统属性: `readonlymode.enabled`)

3.4.0 中的新功能: 将此值设置为 true 将启用只读模式服务器支持（默认情况下禁用）。 ROM 允许客户端请求 ROM 支持的会话连接到服务器，即使服务器可能从法定人数分

隔。在这种模式下，ROM 客户端仍然可以从 ZK 服务读取值，但是无法写入值并查看其他客户端的更改。有关详细信息，请参阅 ZOOKEEPER-784。

4.1.2.12.6 不安全的选项

以下选项可能很有用，但使用它们时要小心。每个人的风险以及该变量所做的解释一并解释。

`forceSync`

(Java 系统属性: `zookeeper.forceSync`)

在完成处理更新之前，需要将更新同步到事务日志的介质。如果此选项设置为否，ZooKeeper 将不需要将更新同步到媒体。

`jute.maxbuffer:`

(Java 系统属性: `jute.maxbuffer`)

此选项只能设置为 Java 系统属性。没有 `zookeeper` 前缀。它指定可以存储在 `znode` 中的数据的大小。默认值为 `0xfffff`，或低于 1M。如果更改此选项，则必须在所有服务器和客户端上设置系统属性，否则将出现问题。这真的是一个健康检查。ZooKeeper 旨在存储大小为千字节数量的数据。

`skipACL`

(Java 系统属性: `zookeeper.skipACL`)

跳过 ACL 检查。这会导致吞吐量的提高，但可以向所有人开放数据树的完全访问。

`quorumListenOnAllIPs`

当设置为 `true` 时，ZooKeeper 服务器将监听来自其所有可用 IP 地址的对等体的连接，而不仅仅是在配置文件的服务器列表中配置的地址。它影响处理 ZAB 协议和快速领导选举协议的连接。默认值为 `false`。

4.1.2.12.7 使用 Netty 框架进行沟通

3.4 中的新功能 `Netty` 是一种基于 NIO 的客户端/服务器通信框架，它简化了（通过 NIO 直接使用）Java 应用程序的网络级通信的许多复杂性。另外 `Netty` 框架内置了对加密（SSL）和认证（证书）的支持。这些是可选功能，可单独打开或关闭。

在 3.4 之前，ZooKeeper 一直使用 NIO，但是在版本 3.4 和更高版本中，`Netty` 作为 NIO（替代）的选项被支持。NIO 继续是默认的，但是通过将环境变量 `“zookeeper.serverCnxnFactory”` 设置为 `“org.apache.zookeeper.server.NettyServerCnxnFactory”`，可以使用基于 `Netty` 的通信来代替 NIO。您可以选择在客户端或服务上进行设置，通常您需要将其设置在两者上，但这是您自行决定的。

TBD – netty 的调整选项 – 目前没有一个是 netty 具体的，但我们应该添加一些。 Esp 最大限制在 netty 创建的读者工作线程数。

TBD – 如何管理加密

TBD – 如何管理证书

4. 1. 2. 13 ZooKeeper 命令：四个字母的单词

ZooKeeper 响应一小组命令。 每个命令由四个字母组成。 您可以通过 telnet 或 nc 在客户端向 ZooKeeper 发出命令。

三个更有趣的命令：“stat”提供了有关服务器和连接的客户端的一些一般信息，而“srvr”和“cons”分别提供了服务器和连接的扩展详细信息。

conf

3. 3. 0 中的新功能：打印有关投放配置的详细信息。

缺点

3. 3. 0 中的新功能：列出连接到此服务器的所有客户端的完整连接/会话详细信息。 包括接收/发送的数据包数，会话 ID，操作延迟，执行的最后操作等的信息...

crst

3. 3. 0 中的新功能：重置所有连接的连接/会话统计信息。

倾倒

列出了未完成的会话和短暂节点。 这只适用于领导。

环境

打印有关服务环境的详细信息

你还行吗

测试服务器是否处于非错误状态。 如果服务器正在运行，将使用 imok 响应。 否则它根本不会做出回应。

“imok”的响应不一定表示服务器已加入仲裁，只是服务器进程是活动的并且绑定到指定的客户端。 使用“stat”来了解状态 wrtum 和客户端连接信息。

srst

重置服务器统计

srvr

3. 3. 0 中的新功能：列出服务器的完整详细信息。

统计

列出服务器和连接的客户端的简要信息。

wchs

3.3.0 中的新功能：列出有关服务器的观察者(Watcher)的简要信息。

wchc

3.3.0 中的新功能：按会话列出服务器观察者(Watcher)的详细信息。 这将输出与相关监视（路径）的会话（连接）列表。 请注意，根据观察者(Watcher)的数量，此操作可能很昂贵（即影响服务器性能），请仔细使用。

wchp

3.3.0 中的新功能：根据路径列出服务器观察者(Watcher)的详细信息。 这将输出具有关联会话的路径列表（znodes）。 请注意，根据观察者(Watcher)的数量，此操作可能很昂贵（即影响服务器性能），请仔细使用。

mntr

3.4.0 中的新增功能：输出可用于监视群集运行状况的变量列表。

```
$ echo mntr | nc localhost 2185
```

```
zk_version 3.4.0
zk_avg_latency 0
zk_max_latency 0
zk_min_latency 0
zk_packets_received 70
zk_packets_sent 69
zk_outstanding_requests 0
zk_server_state 领导
zk_znode_count 4
zk_watch_count 0
zk_ephemerals_count 0
zk_approximate_data_size 27
zk_followers 4 - 仅由领导者公开
zk_synced_followers 4 - 仅由领导者公开
zk_pending_syncs 0 - 仅由领导者公开
zk_open_file_descriptor_count 23 - 仅在 Unix 平台上可用
zk_max_file_descriptor_count 1024 - 仅在 Unix 平台上可用
```

输出与 java 属性格式兼容，内容可能随时间而变化（添加新密钥）。 您的脚本应该会改变。

注意：某些键是平台特定的，一些键仅由领导者导出。

输出包含多行，格式如下：

键\t值

这是一个 **ruok** 命令的例子：

```
$ echo ruok | nc 127.0.0.1 5111  
我可以
```

4.1.2.14 数据文件管理

ZooKeeper 将其数据存储在数据目录中，其事务日志存储在事务日志目录中。默认情况下，这两个目录是一样的。服务器可以（并且应该）被配置为将事务日志文件存储在与数据文件不同的目录中。当事务日志驻留在专用日志设备上时，吞吐量增加和延迟减少。

4.1.2.14.1 数据目录

该目录中有两个文件：

- **myid** - 包含代表服务器 ID 的可读取 ASCII 文本中的单个整数。
- **快照**。<zxid> - 保存数据树的模糊快照。

每个 ZooKeeper 服务器都有唯一的 ID。这个 id 在两个地方使用：**myid** 文件和配置文件。**myid** 文件标识与给定数据目录对应的服务器。配置文件列出了其服务器标识的每个服务器的联系人信息。当 ZooKeeper 服务器实例启动时，它从 **myid** 文件中读取其 ID，然后使用该 id 从配置文件中读取它应该侦听的端口。

存储在数据目录中的快照文件是模糊快照，这意味着在 ZooKeeper 服务器拍摄快照的时候，更新正在数据树中发生。快照文件名称的后缀是快照开始时上次提交的事务的 **zxid**（ZooKeeper 事务标识）。因此，快照包括在快照正在进行时发生的数据树更新的子集。然后，快照可能不对应于任何实际存在的数据树，因此我们将其称为模糊快照。尽管如此，ZooKeeper 可以使用此快照进行恢复，因为它可以利用其更新的等级特性。通过对模糊快照重播事务日志，ZooKeeper 将在日志结束时获取系统的状态。

4.1.2.14.2 日志目录

日志目录包含 ZooKeeper 事务日志。在进行任何更新之前，ZooKeeper 确保将表示更新的事务写入非易失性存储。每次启动快照时都会启动一个新的日志文件。日志文件的后缀是写入该日志的第一个 **zxid**。

4.1.2.14.3 文件管理

快照和日志文件的格式在独立的 ZooKeeper 服务器和复制的 ZooKeeper 服务器的不同配置之间不会改变。因此，您可以将这些文件从运行的复制的 ZooKeeper 服务器拉回到具有独立的 ZooKeeper 服务器的开发机器进行故障排除。

使用较旧的日志和快照文件，您可以查看之前状态的 ZooKeeper 服务器，甚至恢复该状态。 LogFormatter 类允许管理员查看日志中的事务。

ZooKeeper 服务器创建快照和日志文件，但不会删除它们。 数据和日志文件的保留策略在 ZooKeeper 服务器之外实现。 服务器本身只需要最新的完整模糊快照和从该快照开始的日志文件。 有关设置保留策略和维护 ZooKeeper 存储的详细信息，请参阅本文档中的[维护](#)部分。

注意

存储在这些文件中的数据未被加密。 在 ZooKeeper 中存储敏感数据的情况下，需要采取必要措施以防止未经授权的访问。 此类措施在 ZooKeeper 外部（例如，控制对文件的访问），并取决于其部署的各个设置。

4.1.2.15 要避免的事情

以下是通过正确配置 ZooKeeper 可以避免的一些常见问题：

不一致的服务器列表

客户端使用的 ZooKeeper 服务器列表必须与每个 ZooKeeper 服务器具有的 ZooKeeper 服务器列表相匹配。 如果客户端列表是真实列表的一部分，事情可以正常工作，但如果客户端列出了不同 ZooKeeper 集群中的 ZooKeeper 服务器，事情将会变得奇怪。 另外，每个 Zookeeper 服务器配置文件中的服务器列表应该彼此一致。

传递日志不正确的位置

ZooKeeper 最具性能的关键部分是事务日志。 在返回响应之前，ZooKeeper 会将事务同步到媒体。 专用的事务日志设备是一致的良好性能的关键。 将日志放在繁忙的设备上会不利地影响性能。 如果您只有一个存储设备，请将跟踪文件放在 NFS 上并增加 snapshotCount； 它不能消除这个问题，但它应该减轻这个问题。

不正确的 Java 堆大小

您应该特别注意正确设置 Java 最大堆大小。 特别是，您不应该创建 ZooKeeper 切换到磁盘的情况。 磁盘死亡到 ZooKeeper。 一切都是有序的，所以如果处理一个请求交换磁盘，所有其他排队的请求可能会相同。 磁盘。 不要切换

在您的估计中保守：如果您有 4G 的 RAM，则不要将 Java 最大堆大小设置为 6G 甚至 4G。 例如，由于操作系统和缓存也需要内存，所以更有可能为 4G 机器使用 3G 堆。 估计系统需要的堆大小的最佳和唯一推荐做法是运行负载测试，然后确保您远远低于将导致系统交换的使用限制。

公开访问的部署

ZooKeeper 系列预计将在可信赖的计算环境中运行。 因此建议在防火墙后部署 ZooKeeper。

4.1.2.16 最佳做法

为了获得最佳效果，请注意以下 Zookeeper 的良好做法：

对于多功能安装，请参阅 ZooKeeper “chroot” 支持部分，当将许多应用程序/服务部署到单个 ZooKeeper 群集时，这非常有用。

4.2 ZooKeeper 配额指南：部署和管理指南

- [配额](#)
 - [配置配额](#)
 - [上市配额](#)
 - [删除配额](#)

4.2.1 配额

ZooKeeper 具有命名空间和字节配额。 您可以使用 ZooKeeperMain 类来设置配额。 如果用户超过分配给它们的配额，ZooKeeper 将打印 *WARN* 消息。 消息打印在 ZooKeeper 的日志中。

```
$ bin / zkCli.sh -server host: port
```

上面的命令给你一个使用配额的命令行选项。

4.2.1.1 配置配额

您可以使用 *setquota* 在 ZooKeeper 节点上设置配额。 它可以选择使用 *-n*（用于命名空间）和 *-b*（对于字节）设置配额。

ZooKeeper 配额存储在 ZooKeeper 本身/ *zookeeper* /配额中。 要禁止其他人更改配额，请为/ *zookeeper* /配额设置 ACL，以便只有管理员才能读取和写入。

4.2.1.2 上市配额

您可以使用 *listquota* 在 ZooKeeper 节点上列出配额。

4.2.1.3 删除配额

您可以使用 *delquota* 删除 ZooKeeper 节点上的配额。

4.3 ZooKeeper JMX

- [JMX](#)
- [启动使用 JMX 的 ZooKeeper](#)
- [运行 JMX 控制台](#)
- [ZooKeeper MBean 参考](#)

4.3.1 JMX

Apache ZooKeeper 对 JMX 有广泛的支持，允许您查看和管理 ZooKeeper 服务器。

本文假设您具有 JMX 的基础知识。 请参阅 [Sun JMX 技术页面](#) 以开始使用 JMX。

有关设置 VM 实例的本地和远程管理的详细信息，请参阅 “[JMX 管理指南](#)”。 默认情况下，`zkServer.sh` 仅支持本地管理 - 查看链接文档以支持远程管理（超出本文档范围）。

4.3.2 启动使用 JMX 的 ZooKeeper

类 `org.apache.zookeeper.server.quorum.QuorumPeerMain` 将启动一个 JMX 可管理的 ZooKeeper 服务器。 该类在启动期间注册正确的 MBean，以支持实例的 JMX 监视和管理。 有关使用 `QuorumPeerMain` 启动 ZooKeeper 的一个示例，请参阅 `bin / zkServer.sh`。

4.3.3 运行 JMX 控制台

有许多 JMX 控制台可以连接到正在运行的服务器。 对于这个例子，我们将使用 Sun 的 `jconsole`。

Java JDK 附带一个名为 `jconsole` 的简单 JMX 控制台，可用于连接 ZooKeeper 并检查正在运行的服务器。 一旦您使用 `QuorumPeerMain` 启动 ZooKeeper 启动 `jconsole`，通常位于 `JDK_HOME / bin / jconsole`

当显示“新连接”窗口时，可以连接到本地进程（如果 `jconsole` 在与 Server 相同的主机上启动）或使用远程进程连接。

默认情况下，显示虚拟机的“概述”选项卡（这是了解虚拟机 btw 的好方法）。 选择“MBeans”选项卡。

您现在应该在左侧看到 `org.apache.ZooKeeperService`。 展开此项目，并根据您如何启动服务器，您将能够监视和管理各种与服务相关的功能。

还要注意，ZooKeeper 也会注册 `log4j` MBean。 在左侧的同一部分，您将看到“log4j”。 展开以通过 JMX 管理 `log4j`。 特别感兴趣的是通过编辑 `appender` 和根阈值动态地更改记录级别的能

力。 Log4j 可以通过在启动 ZooKeeper 时将 `-Dzookeeper.jmx.log4j.disable = true` 传递给 JVM 来禁用 MBean 注册。

4.3.4 ZooKeeper MBean 参考

此表详细说明了参与复制的 ZooKeeper 系统（即不是独立）的服务器的 JMX。 这是生产环境的典型案例。

MBeans, 他们的名字和描述		
MBean	MBean 对象名称	
Quorum	ReplicatedServer_id <#>	<代表所有集群成员的 Quorum 或 Ensemble - parent。 请注意，对象名称包括您的 JMX 代理连接到的服务器（名称）>
LocalPeer RemotePeer	副本	表示本地或远程对等体（即参与集合的服务器）。 请注意，对象名称包括服务器的“myid”（名称后缀）。
LeaderElection	领导选择	代表一个正在进行的 ZooKeeper 集群领导选举。 提供关于选举的信息，例如何时开始。
Leader	领导	表示父副本是领导者，并为该服务器提供属性/操作。 请注意，Leader 是 ZooKeeperServer 的子类，因此它提供
Follower	追随者	表示父副本是跟随者，并为该服务器提供属性/操作。 请注意，Follower 是 ZooKeeperServer 的子类，因此它提
DataTree	InMemoryDataTree	在内存中统计 znode 数据库，还可以对数据进行更精细（更加计算密集）统计的操作（如临时计数）。 InMemory
ServerCnxn	<session_id>	统计每个客户端连接，也对这些连接进行操作（如终止）。 注意对象名称是十六进制形式的连接的会话 ID。

此表详细说明了独立服务器的 JMX。 通常，独立的仅用于开发环境。

MBeans, 他们的名字和描述		
MBean	MBean 对象名称	
ZooKeeperServer	StandaloneServer_port <#>	运行服务器上的统计信息，还有操作来重置这些属性。 请注意，对象名称包括服务器的客户端端口（名称后缀）。
DataTree	InMemoryDataTree	在内存中统计 znode 数据库，还可以对数据进行更精细（更加计算密集）统计的操作（如临时计数）。
ServerCnxn	<session_id>	统计每个客户端连接，也对这些连接进行操作（如终止）。 注意对象名称是十六进制形式的连接的会话 ID。

4.4 Zookeeper 观察员

- [观察者：缩放 ZooKeeper 而不伤害写入性能](#)
- [如何使用观察者](#)
- [示例用例](#)

4. 4. 1 观察者：缩放 ZooKeeper 而不伤害写入性能

虽然 ZooKeeper 通过让客户直接连接到合奏的投票成员来表现得非常好，但是这种架构使得很难扩展到大量的客户端。 问题在于，当我们添加更多的投票成员时，写入性能下降。 这是因为一个写作操作需要（一般地）一个合奏中的至少一半的节点的协议，因此，在添加更多的选民的情况下，投票的成本可以显著增加。

我们推出了一种称为 *Observer* 的新型 ZooKeeper 节点，它可以帮助解决这个问题，进一步提高 ZooKeeper 的可伸缩性。 观察员是一个没有投票权的成员，只能听取投票结果，而不是达成协议的协议。 除了这个简单的区别，观察者的功能与关注者完全相同 - 客户端可能会连接到它们并向其发送读取和写入请求。 观察员将这些请求转发给领导者，就像“追随者”一样，但他们只是等待听取投票结果。 因此，我们可以尽可能多地增加观察员的数量，而不会损害观察员的表现。

观察员有其他优势。 因为他们不投票，它们不是 ZooKeeper 合奏的关键部分。 因此，它们可能会失败或与群集断开连接，而不会损害 ZooKeeper 服务的可用性。 用户的好处是观察者可能通过比追随者更不可靠的网络链接进行连接。 事实上，观察者可能会用来与另一个数据中心的 ZooKeeper 服务器通话。 观察员的客户将看到快速读取，因为所有的读取都是在本地提供的，并且写入会导致最小的网络流量，因为没有投票协议所需的消息数量较少。

4. 4. 2 如何使用观察者

设置使用观察者的 ZooKeeper 系统非常简单，只需要对配置文件进行两次更改。 首先，在要作为 Observer 的每个节点的配置文件中，您必须放置此行：

```
peerType = observer
```

该行告诉 ZooKeeper 服务器是一个观察者。 其次，在每个服务器配置文件中，您必须添加：observer 到每个 Observer 的服务器定义行。 例如：

```
server.1: localhost: 2181: 3181: observer
```

这告诉每个其他服务器 server.1 是一个观察者，他们不应该期望它投票。 这是您将 Zoo 观察器添加到 ZooKeeper 群集中所需要做的所有配置。 现在你可以连接到它，就像它是一个普通的追随者。 尝试一下，运行：

```
$ bin / zkCli.sh -server localhost: 2181
```

其中 localhost: 2181 是每个配置文件中指定的 Observer 的主机名和端口号。 您应该看到一个命令行提示符，通过它可以发出像 *ls* 这样的命令来查询 ZooKeeper 服务。

4. 4. 3 示例用例

观察员的两个示例用例如下所示。事实上，无论您希望如何扩展 ZooKeeper 系列的客户数量，或者希望将整体关键部分与客户端请求的负担隔离开来，观察者是一个很好的架构选择。

- 作为数据中心桥：在两个数据中心之间形成 ZK 集合是一个有问题的尝试，因为数据中心之间的延迟的高差异可能导致错误的正故障检测和分区。但是，如果集合完全在一个数据中心中运行，而第二个数据中心仅运行“观察者”，则分区不会因为集合保持连接而有问题。观察员的客户仍然可以看到并发出建议。
- 作为消息总线的链接：有些公司表示有兴趣将 ZK 用作持久可靠的消息总线的组件。观察者将为此工作提供一个自然的融合点：可以使用插件机制将观察者看到的提议流附加到发布订阅系统，而不再加载核心合奏。

5 贡献者：Zookeeper 内部

- [介绍](#)
- [原子广播](#)
- [担保，财产和定义](#)
- [领导激活](#)
- [主动消息](#)
- [概要](#)
- [比较](#)
- [法定人数](#)
- [记录](#)
- [开发者指南](#)
- [登录在正确的水平](#)
- [使用标准 slf4j 成语](#)

5.1 介绍

本文档包含有关 ZooKeeper 内部工作的信息。到目前为止，它讨论了这些主题：

- [原子广播](#)
- [记录](#)

5.2 原子广播

ZooKeeper 的核心是将所有服务器保持同步的原子信息系统。

5.2.1 担保，财产和定义

ZooKeeper 使用的消息系统提供的具体保证如下：

可靠的交货

如果一个消息 m 由一个服务器传递，则最终将由所有服务器传递。

总订单

如果一个服务器在消息 b 之前传送了一条消息，则所有服务器将在 b 之前传送一个消息。如果 a 和 b 被传递消息，则在 b 或 b 将在 a 之前传送 a 将被递送。

原因订单

如果在 b 的发送者发送消息 a 之后发送消息 b ，则必须在 b 之前订购 a 。发送方发送 c 后， c 必须在 b 之后进行订购。

ZooKeeper 消息系统还需要高效，可靠，易于实施和维护。我们大量使用消息传递，因此我们需要系统能够每秒处理数千个请求。尽管我们至少需要 $k + 1$ 个正确的服务器来发送新的消息，但是我们必须能够从相关的故障（如断电）中恢复。当我们实施系统时，我们几乎没有时间和少量的工程资源，所以我们需要一个工程师可以访问的协议，并且易于实现。我们发现我们的协议满足了所有这些目标。

我们的协议假设我们可以在服务器之间构建点对点 FIFO 通道。虽然类似的服务通常假设可能丢失或重新排序消息的消息传递，但是由于我们使用 TCP 进行通信，所以我们对 FIFO 通道的假设非常实用。具体来说，我们依靠 TCP 的以下属性：

有序交货

数据按照发送的相同顺序进行传送，并且仅在传送 m 之前发送的所有消息之后才传送消息 m 。（这样做的结论是，如果消息 m 丢失， m 后的所有消息将丢失。）

关闭后没有留言

一旦 FIFO 通道关闭，就不会收到任何消息。

FLP 证明，如果可能出现故障，则在异步分布式系统中无法实现共识。为了确保我们在失败的情况下达成共识，我们使用超时。但是，我们依靠时代的活泼而不是正确的。因此，如果超时停止工作（例如，时钟故障），消息系统可能挂起，但不会违反其保证。

在描述 ZooKeeper 消息协议时，我们将讨论数据包，提议和消息：

包

通过 FIFO 通道发送的字节序列

提案

一个协议单位。通过与 ZooKeeper 服务器的数量交换数据包来商定提案。大多数提案都包含邮件，但是 NEW_LEADER 提案是与邮件不对应的提案的示例。

信息

要将所有 ZooKeeper 服务器原子地广播的字节序列。在提交之前提交的信息，并在交付之前达成一致。

如上所述，ZooKeeper 保证了邮件的总顺序，并且还保证了提案的总顺序。 ZooKeeper 使用 ZooKeeper 事务 ID (*zxid*) 公开总排序。 提出建议时，所有提案都将加上 *zxid*，并且完全反映总排序。 提案将发送到所有 ZooKeeper 服务器，并在其中的法定人数承认提案时提交。 如果提案包含消息，则在提交提案时将传递消息。 确认意味着服务器已将建议记录到永久存储。 我们的法定人数要求任何一对法定人数必须至少有一个共同的服务器。 我们通过要求所有仲裁都具有大小 ($n / 2 + 1$) 来确保这一点，其中 *n* 是构成 ZooKeeper 服务的服务器数量。

zxid 有两部分：时代和计数器。 在我们的实现中，*zxid* 是一个 64 位的数字。 我们使用高阶 32 位的时期和低位 32 位的计数器。 因为它有两部分代表 *zxid* 作为一个数字和一个整数 (*epoch*, *count*)。 时代代表领导层的变化。 每当新领导人上台时，它将拥有自己的时代。 我们有一个简单的算法来为一个提案分配一个唯一的 *zxid*：领导者只需增加 *zxid*，为每个提案获得唯一的 *zxid*。 领导激活将确保只有一个领导者使用给定的时期，所以我们的简单算法保证每个提案都有唯一的 ID。

ZooKeeper 消息传递包括两个阶段：

领导激活

在这个阶段，领导者建立了系统的正确状态，并准备开始提出建议。

活动消息

在这个阶段，领导者接受消息来提出和协调消息传递。

ZooKeeper 是一个整体协议。 我们不把重点放在个别的建议上，而是把整个建议的流程看成一个。 我们严格的订购使我们能够有效地做到这一点，大大简化了我们的协议。 领导激励体现了这个整体概念。 一个领导者只有当一个法定的追随者（领导者也被视为追随者，你总是投票给自己）与领导者同步，他们有相同的状态。 这个国家包括领导人认为已经提出的所有提案，以及跟随领导者的建议 NEW_LEADER 提案。 （希望你在想自己， 领导人认为的一套提案是否包括了所有真正提出的建议 ， 答案是肯定的 ， 下面我们明白为什么）

5.2.2 领导激活

领导激活包括领导选举。 我们目前在 ZooKeeper 中有两个领先的选举算法：LeaderElection 和 FastLeaderElection (AuthFastLeaderElection 是 FastLeaderElection 的一个变体，它使用 UDP，并允许服务器执行简单的身份验证形式以避免 IP 欺骗)。 ZooKeeper 消息传递不在乎选择领导者的确切方法长久以来如下：

- 领导人已经看到了所有追随者中最高的 *zxid*。
- 服务器的法定人数已经承诺跟随领导者。

在这两个要求中，只有第一个，追随者需要保持正确操作的最高 *zxid*。 第二个要求是法定的追随者，只需要高概率地保持。 我们要重新检查第二个要求，所以如果在领导人选举和法定人数丧失之前或之后发生失败，我们将通过放弃领导激活和进行另一次选举来恢复。

领导选举之后，单个服务器将被指定为领导者，并开始等待追随者连接。 其余的服务器将尝试连接到领导。 领导者将通过发送任何缺失的提案与追随者进行同步，或者如果追随者缺少太多提案，它将向跟随者发送状态的完整快照。

有一个角落的情况下，有一个提案的追随者 U，没有被领导者看到。 提出的建议是按顺序进行的，所以 U 的提案的 zxids 将高于领导者看到的 zxid。 追随者必须在领导人当选之后到达，否则，由于已经看到更高的 zxid，追随者将被选为领导者。 由于所提交的提案必须被服务器的数量所限定，而选举领导人的服务器数量并未看到 U，所以您的提案尚未提交，因此可以将其丢弃。 当追随者连接领导者时，领导将告诉跟随者舍弃 U。

一个新的领导者建立一个 zxid 来开始使用新的提案，通过获得它所看到的最高 zxid 的时代 e，并将下一个 zxid 设置为 $(e + 1, 0)$ ，使领导者与一个跟随者同步将提出一个 NEW_LEADER 提案。 一旦提交了 NEW_LEADER 提案，领导者将激活并开始接收和发布提案。

这一切听起来很复杂，但这里是领导激活期间的基本操作规则：

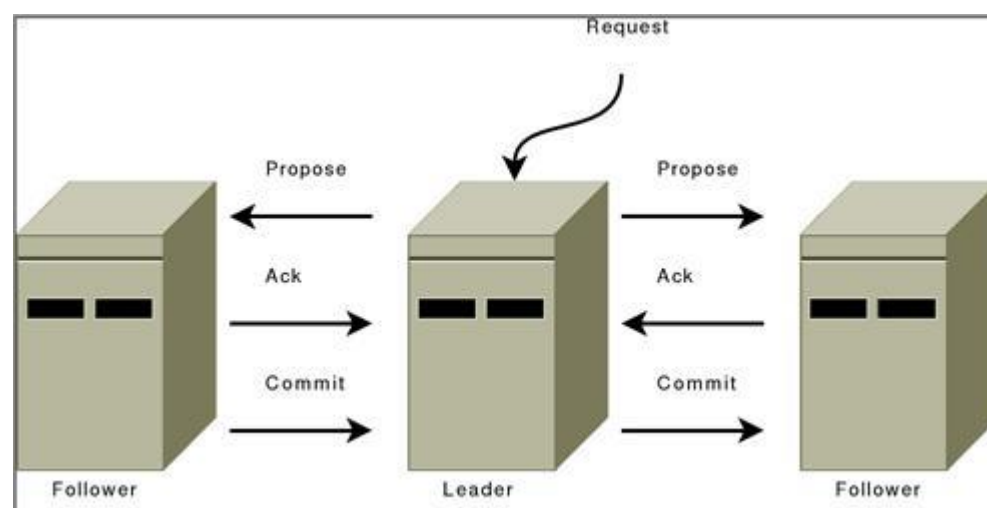
- 跟随者在与领导者同步之后，将确认 NEW_LEADER 提案。
- 追随者只能使用来自单个服务器的给定 zxid 来确认 NEW_LEADER 提案。
- 一位新的领导人将在法定人数确认后提交 NEW_LEADER 提案。
- 当 NEW_LEADER 提案为 COMMIT 时，追随者将承担从领导者收到的任何状态。
- 在 NEW_LEADER 提案已被 COMMITED 之前，新的领导者将不会接受新的提案。

如果领导人选举错误地终止，我们没有问题，因为 NEW_LEADER 的提案不会被提交，因为领导人不会有法定人数。 当这种情况发生时，领导人和任何剩余的追随者将暂停，并返回领导人的选举。

5.2.3 主动消息

领导激活做的都很重。 一旦领导人加冕，他就可以开始爆破提案。 只要他仍然是领导人，没有其他领导人可以出现，因为没有其他领导人能够获得法定的追随者。 如果一个新的领导人出现，这意味着领导人已经失去了法定人数，新领导人将在领导激活期间清理遗留下来的任何混乱。

ZooKeeper 消息传递类似于经典的两阶段提交。



所有通信通道都是 FIFO，所以一切都是按顺序完成的。 具体来说，遵守以下操作限制：

- 领导者使用相同的顺序向所有追随者发送建议。 此外，此顺序遵循接收请求的顺序。 因为我们使用 FIFO 通道，这意味着追随者也按顺序接收提案。

- 关注者按照收到的顺序处理消息。 这意味着消息将按顺序被确认，由于 FIFO 通道，引导者将按顺序从接收者接收 ACK。 这也意味着如果消息\$ m \$已写入非易失性存储，则在\$ m \$之前提出的所有消息都已写入非易失性存储。
- 只要法定人数确认了一个信息，领导者将向所有追随者发出 COMMIT。 由于消息是按顺序确认的，所以由发送者按照接收者的顺序发送 COMMIT。
- COMMITs 按顺序处理。 当提案提交时，关注者提供建议消息。

5.2.4 概要

所以你去。 为什么它工作？ 具体来说，为什么一个新领导层认为的一套提案总是包含实际上提出的任何提案？ 首先，所有提案都有一个独特的 zxid，与其他协议不同，我们从不担心为同一个 zxid 提出两个不同的值； 追随者（领导也是追随者）按顺序查看和记录提案； 提案是按顺序进行的； 一次只有一个领导者，因为追随者一次只跟随一个领导人， 一个新的领导者已经看到了从前一个时代的所有提出的建议，因为它看到了从法定服务器的最高 zxid； 新领导人以前的历史纪录提出的任何未经授权的提案将由该领导人在其变得活跃之前承诺。

5.2.5 比较

这不是只是多 Paxos 吗？ 不，Multi-Paxos 需要一些确保只有一个协调者的方法。 我们不指望这样的保证。 相反，我们使用领导激活来恢复领导层的变革或老领导者相信他们仍然活跃。

这不是 Paxos 吗？ 您的活跃消息阶段看起来就像 Paxos 的第二阶段？ 实际上，对我们来说，活动消息看起来就像 2 阶段提交，而不需要处理中止。 在这种情况下，活动消息传递与交叉提议排序要求不同。 如果我们不对所有数据包进行严格的 FIFO 排序，那么它们都会分开。 此外，我们的领导激活阶段与两者不同。 特别是，我们使用时代可以让我们跳过未提交的提案，不用担心给定的 zxid 的重复提议。

5.3 法定人数

原子广播和领导人选举使用法定人数的概念来保证系统的一致观点。 默认情况下，ZooKeeper 使用多数法定人数，这意味着在其中一个协议中发生的每次投票都需要多数人投票。 一个例子是承认一个领导者提案：领导者只有在收到来自法定服务器的确认后才能提交。

如果我们从我们使用多数提取真正需要的属性，那么我们只需要保证通过投票来验证一个操作的进程组（例如，承认领导者提议）在至少一个服务器中成对相交。 使用多数保证这样的财产。 然而，还有其他方法来构建不同于多数的法定人数。 例如，我们可以为服务器的投票分配权重，并且说一些服务器的投票更重要。 要获得法定人数，我们得到足够的票数，所有票数的总和大于所有权重总和的一半。

使用权重的不同结构在广域部署（共同位置）中是有用的，是一种层次化的结构。 通过这种结构，我们将服务器分为不相交的组，并为进程分配权重。 为了形成法定人数，我们必须从大多数 G 组中获得足够的服务器，这样对于 g 中的每个组，g 中的总和大于 g 中的权重总和的一半。 有趣的是，这种结构使得法定人数更小。 例如，如果有 9 个服务器，我们将它们分成 3 组，并为每个服务器分配一个权重，然后我们可以形成大小为 4 的仲裁。注意，两个进程组成的子集大多数

大多数组中的每个服务器必须具有非空交集。可以合理地预期，大多数协同位置将以大概率使大多数服务器可用。

使用 ZooKeeper，我们为用户提供了配置服务器以使用群体的多数仲裁，权重或层次结构的能力。

5.4 记录

Zookeeper 使用 `slf4j` 作为记录层的抽象层。版本 1.2 中的 `log4j` 被选为现在的最终日志记录实现。为了更好的嵌入支持，计划在未来决定向最终用户选择最终的日志记录实现。因此，始终使用 `slf4j` api 在代码中写入日志语句，但是配置 `log4j` 以了解如何在运行时登录。请注意，`slf4j` 没有 FATAL 级别，FATAL 级别的以前的消息已被移动到 ERROR 级别。有关为 ZooKeeper 配置 `log4j` 的信息，请参阅“[ZooKeeper 管理员指南](#)”的“[日志记录](#)”部分。

5.4.1 开发者指南

在代码中创建日志语句时，请遵循 [slf4j 手册](#)。在创建日志语句时，请阅读有关性能的 [FAQ](#)。补丁审核人员将寻找以下内容：

5.4.1.1 登录在正确的水平

在 `slf4j` 中有几个记录级别。选择一个是很重要的。按照更高到更低的严重程度：

1. 错误级别指定可能仍允许应用程序继续运行的错误事件。
2. WARN 级别指定潜在的有害情况。
3. INFO 级别指定在粗粒度级别突出显示应用程序进度的信息性消息。
4. DEBUG 级别指定调试应用程序最有用的细粒度信息事件。
5. TRACE 级别指定比 DEBUG 更精细的信息事件。

ZooKeeper 通常以生产方式运行，使得 INFO 级别严重性和更高（更严重）的日志消息输出到日志。

5.4.1.2 使用标准 `slf4j` 成语

静态消息记录

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

但是，当需要创建参数化消息时，请使用格式化锚点。

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

命名

记录器应该以使用它们的类命名。

```
public class Foo {  
    private static final Logger LOG = LoggerFactory.getLogger(Foo.class);  
    ....  
}
```

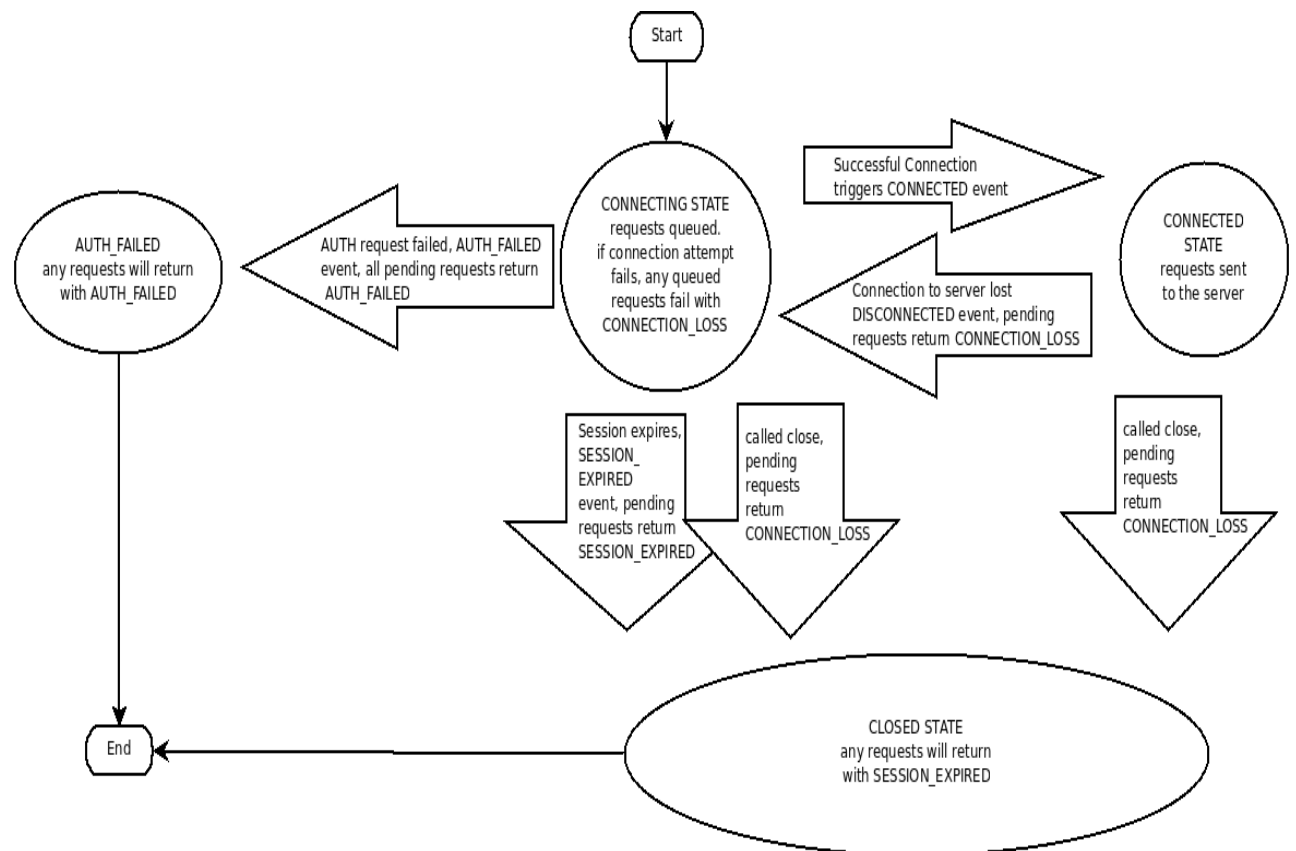
```
public Foo() {  
    LOG.info("constructing Foo");  
}
```

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  
}
```

6 常见问题：FAQ

6.1 ZooKeeper 的状态转换是什么？



6.2 如何处理 CONNECTION_LOSS 错误？

CONNECTION_LOSS 意味着客户端和服务端之间的链接断开。这并不一定意味着请求失败。如果您正在执行创建请求，并且在请求到达服务器之后并且在返回响应之前链接已中断，则创建请求将成功。如果在数据包进入电线之前链接断开，则创建请求失败。不幸的是，客户端库没有办法知道，所以它返回 CONNECTION_LOSS。程序员必须弄清楚请求是成功还是需要重试。通常这是以应用程序的具体方式完成的。成功检测的示例包括检查要创建的文件的存在或检查要修改的 znode 的值。

当客户端（会话）从 ZK 服务集群分区时，它将开始搜索在会话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，会话将再次转换到“已连接”状态（如果在会话超时值内重新连接），或者它将转换到“过期”状态（如果会话超时后重新连接）。ZK 客户端库将自动处理重新连接。特别是，我们将客户端库中的启发式内置到处理诸如“群体效应”之类的东西... 只有当通知会话到期（强制性）时才创建一个新的会话。

6.3 我应该如何处理 SESSION_EXPIRED？

SESSION_EXPIRED 自动关闭 ZooKeeper 句柄。在正确运行的群集中，您不应该看到 SESSION_EXPIRED。这意味着客户端与 ZooKeeper 服务分开了更多的会话超时，ZooKeeper 决定客

户端死机。由于 ZooKeeper 服务是实地的，所以客户应该考虑自己死亡，并恢复。如果客户端只是从 ZooKeeper 读取状态，恢复就意味着重新连接。在更复杂的应用程序中，恢复意味着重建临时节点，争取领导角色，并重建已发布的状态。

图书馆作家应该意识到过期状态的严重性，而不是尝试从中恢复。相反，库应该返回致命错误。即使图书馆只是从 ZooKeeper 阅读，图书馆的用户也可能正在使用 ZooKeeper 进行其他操作，这需要更复杂的恢复。

会话到期由 ZooKeeper 集群本身管理，而不是由客户端管理。当 ZK 客户端与集群建立会话时，它提供一个“超时”值。群集使用此值来确定客户端的会话何时过期。当集群在指定的会话超时时间内没有听到客户端（即没有心跳）时，会发生到期。在会话到期时，集群将删除该会话拥有的任何/所有短暂节点，并立即通知任何/所有连接的客户端的更改（任何人观看这些 znodes）。此时，过期会话的客户端仍然与集群断开连接，除非可以重新建立与集群的连接，否则将不会通知会话到期。客户端将保持断开状态，直到与集群重新建立 TCP 连接，届时，过期会话的观察者将收到“会话到期”通知。

过期会话的观察者所看到的过期会话的示例状态转换：

1. 'connected'：会话建立，客户端与群集通信（客户端/服务器通信正常运行）
2. 客户端从集群分区
3. 'disconnect'：客户端已经失去与集群的连接
4. 时间过去，在超时时间之后，集群会过期，客户端看不到与集群断开连接
5. 时间流逝，客户端恢复与集群的网络级连接
6. 'expired'：最终客户端重新连接到集群，然后通知到期

6.4 是否有一个简单的方法来过期测试？

是的，一个 ZooKeeper 句柄可以占用会话 ID 和密码。此构造函数用于在应用程序失败后恢复会话。例如，应用程序可以连接到 ZooKeeper，将会话 ID 和密码保存到文件，终止，重新启动，读取会话 ID 和密码，并重新连接到 ZooKeeper，而不会丢失会话和相应的临时节点。由程序员确定会话 ID 和密码不会传递到应用程序的多个实例，否则可能会导致问题。

在测试的情况下，我们想要导致一个问题，所以要显式地过期应用程序连接到 ZooKeeper 的会话，保存会话 ID 和密码，使用该 ID 和密码创建另一个 ZooKeeper 句柄，然后关闭新的句柄。由于两个句柄引用相同的会话，所以第二个句柄的关闭将会使会话失效，从而导致第一个句柄上的 SESSION_EXPIRED。

6.5 为什么 NodeChildrenChanged 和 NodeDataChanged 观察事件不会返回有关更改的更多信息？

当 ZooKeeper 服务器生成更改事件时，它会准确了解更改的内容。在我们最初的 ZooKeeper 实现中，我们使用 change 事件返回了这些信息，但是事实证明这是不可能正确使用的。可能有正确的使用方法，但我们从未见过正确使用的情況。问题是观察者(Watcher)用于了解最新的变化。（否则，你只需要定期获取。）大多数程序员似乎都想念的东西，当他们要求这个功能时，

观察者(Watcher)是一次触发。 观察以下数据更改的情况：一个进程在“/ a”上执行一个 getData，其中 watch 设置为 true 并获取“v1”，另一个进程将“/ a”更改为“v2”，稍后将“/ a”更改为“v3”。 第一个进程将会看到“/ a”被更改为“v2”，但不知道“/ a”现在是“/ v3”。

6.6 有什么选择 – 升级 ZooKeeper 的过程？

有两种主要方法： 1) 完全重新启动或 2) 滚动重启。

在完全重新启动的情况下，您可以执行更新的代码/配置/ etc ...，停止系统中的所有服务器，切换代码/配置，并重新启动 ZooKeeper 系列。 如果您以编程方式（脚本通常，即不是手动），重新启动可以以秒为单位完成。 因此，在此期间，客户端将失去与 ZooKeeper 集群的连接，但它会像客户端一样像网络分区。 一旦 ZooKeeper 系统重新启动，所有现有的客户端会话将被维护并重新建立。 显然，这种方法的一个缺点是，如果您遇到任何问题（在测试工具上测试/分级这些更改总是一个好主意），集群可能会比预期的要长。

第二个选项，对于许多用户来说，最好是进行“滚动重启”。 在这种情况下，您可以一次升级 ZooKeeper 系统中的一台服务器； 关闭服务器，升级代码/配置等等，然后重新启动服务器。 服务器将自动重新加入法定人数，使用当前的 ZK 领导更新其内部状态，并开始提供客户端会话。 作为滚动重新启动的结果，而不是完全重新启动，管理员可以在升级过程中监视集合，如果遇到任何问题，可能会回滚。

6.7 如何调整 ZooKeeper 系列（群集）？

一般来说，当确定 ZooKeeper 服务节点的数量（集合的大小）时，您需要考虑可靠性而不是性能。

可靠性：

单个 ZooKeeper 服务器（独立）本质上是一个不可靠性的协调器（单个服务节点故障导致 ZK 服务）。

3 服务器集合（您需要跳转到 3 而不是 2，因为 ZK 基于简单的多数投票工作）允许单个服务器失败，服务仍然可用。

所以如果你希望可靠性至少要 3。我们通常建议在“在线”生产服务环境中有 5 台服务器。 这允许您将服务器服务器（即计划维护）取消，并且仍然能够在服务中断时能够维持其中一个服务器的意外中断。

性能：

添加 ZK 服务器时，写入性能实际上会降低，而读取性能稍微增

加：[http : //zookeeper.apache.org/doc/current/zookeeper0ver.html#Performanc](http://zookeeper.apache.org/doc/current/zookeeper0ver.html#Performanc)
[e](http://zookeeper.apache.org/doc/current/zookeeper0ver.html#Performance)

查看[此页面](http://twitter.com/phunt)进行调查 Patrick Hunt (<http://twitter.com/phunt>) 在独立服务器和大小为 3 的集合中查看了运营延迟。您会注意到，运行独立 ZK 系列的单核心机器（1 服务

器) 仍然能够处理每秒 15k 的请求。 这比大多数应用程序需要的数量级大 (如果他们正确使用 ZooKeeper - 即作为协调服务, 而不是替换数据库, 文件存储, 缓存等)

6.8 我可以在负载均衡器后面运行一个集群集群吗?

在分布式系统中, 从套接字 I / O 角度看, 有两种类型的服务器故障。

1. 服务器由于硬件故障和操作系统崩溃/挂起, Zookeeper 守护程序挂起, 临时/永久网络中断, 网络交换机异常等: 客户端无法立即查明故障, 因为没有响应实体。 因此, zookeeper 客户端必须依靠超时来识别故障。
2. Dead zookeeper 进程 (守护进程): 由于操作系统将响应关闭的 TCP 端口, 客户端将在套接字连接或套接字 I / O 上的“对等重置”时获得“连接被拒绝”。 客户立即注意到另一端失败。

以下是 ZK 客户端在每种情况下对服务器的响应方式。

1. 在这种情况下 (前者), ZK 客户端依靠心跳算法。 ZK 客户端以 2/3 的 recv 超时 (Zookeeper_init) 检测服务器故障, 然后如果只给出一个合成器, 则在每个 recv 超时时间内重试相同的 IP。 如果提供两个以上的集体 IP, ZK 客户端将立即尝试下一个 IP。
2. 在这种情况下, ZK 客户端将立即检测到故障, 并且假设仅给出一个集合 IP, 每秒重试连接。 如果给出多个合奏 IP (大多数安装属于此类别), 则 ZK 客户端将立即重试下一个 IP。

请注意, 在两种情况下, 当指定多个集成 IP 时, ZK 客户端立即重试下一个 IP, 而不会延迟。

在一些安装中, 最好在负载均衡器 (如硬件 L4 交换机, TCP 反向代理或 DNS 循环) 后面运行集群, 因为这样的设置允许用户简单地使用一个主机名或 IP (或 VIP) 进行集群集群, 还有一些检测服务器故障。

但是, 这些负载均衡器对服务器故障的反应有微妙的差异。

- 硬件 L4 负载均衡器: 此设置涉及一个 IP 和主机名。 L4 交换机通常自行进行心跳, 从而从其 IP 列表中删除不响应的主机。 但是这也取决于故障检测的相同超时方案。 L4 可能会将您重定向到无响应的服务器。 如果硬件 LB 检测到服务器故障足够快, 此设置将始终将您重定向到实时集成服务器。
- DNS 轮询: 此设置涉及一个主机名和一个 IP 列表。 ZK 客户端正确使用 DNS 查询返回的 IP 列表。 因此, 此设置的工作方式与 zookeeper_init 的多个主机名 (IP) 参数相同。 缺点是当集群配置更改为服务器添加/删除时, 可能需要一段时间来传播所有 DNS 服务器和 DNS 客户端缓存 (例如 nsd) TTL 问题中的 DNS 条目更改。

总而言之, 除了集群重新配置的情况下, DNS RR 与集合 IP 参数的列表一样好。

事实证明, DNS RR 存在一个小问题。 如果您正在使用诸如 zktop.py 之类的工具, 则不会处理由 DNS 服务器返回的主机 IP 列表。

6.9 群集关闭时，ZK 会话会发生什么？

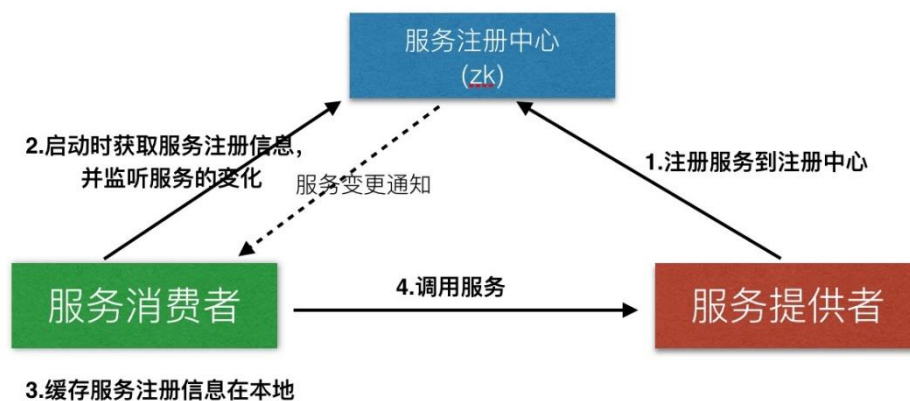
想象一下，客户端连接到 ZK，会话超时时间为 5 秒，管理员将整个 ZK 群集关闭以进行升级。群集关闭几分钟，然后重新启动。

在这种情况下，客户端能够重新连接和刷新其会话。由于会话超时由领导跟踪，所以当集群重新启动时，会话将再次以新鲜超时重新计时。因此，只要客户端在领导者当选后的前 5 秒内连接，它将重新连接而不会过期，并且维护在停机之前的任何短暂节点。

当领导人崩溃并选出新的领导人时，也会表现出相同的行为。在极限情况下，如果领导者快速翻转，会话永远不会过期，因为他们的计时器不断重置。

7 补充内容

7.1 基于 Zookeeper 的服务注册与发现架构



在此架构中有三类角色：服务提供者，服务注册中心，服务消费者。

7.1.1 服务提供者

服务提供者作为服务的提供方将自身的服务信息注册到服务注册中心中。服务信息包含：

- 隶属于哪个系统
- 服务的 IP，端口
- 服务的请求 URL
- 服务的权重等等

7.1.2 服务注册中心

服务注册中心主要提供所有服务注册信息的中心存储，同时负责将服务注册信息的更新通知实时的 Push 给服务消费者（主要是通过 Zookeeper 的 Watcher 机制来实现的）。

7.1.3 服务消费者

服务消费者主要职责如下：

- 服务消费者在启动时从服务注册中心获取需要的服务注册信息
- 将服务注册信息缓存在本地
- 监听服务注册信息的变更，如接收到服务注册中心的服务变更通知，则在本地缓存中更新服务的注册信息
- 根据本地缓存中的服务注册信息构建服务调用请求，并根据负载均衡策略（随机负载均衡，Round-Robin 负载均衡等）来转发请求
- 对服务提供方的存活进行检测，如果出现服务不可用的服务提供方，将从本地缓存中剔除

服务消费者只在自己初始化以及服务变更时会依赖服务注册中心，在此阶段的单点故障通过 Zookeeper 集群来进行保障。在整个服务调用过程中，服务消费者不依赖于任何第三方服务。

7.1.4 服务注册发现方案

7.1.4.1 方案一：DNS

DNS 作为服务注册发现的一种方案，它比较简单。只要在 DNS 服务上，配置一个 DNS 名称与 IP 对应关系即可。定位一个服务只需要连接到 DNS 服务器上，随机返回一个 IP 地址即可。由于存在 DNS 缓存，所以 DNS 服务器本身不会成为一个瓶颈。

这种基于 Pull 的方式不能及时获取服务的状态的更新（例如：服务的 IP 更新等）。如果服务的提供者出现故障，由于 DNS 缓存的存在，服务的调用方会仍然将请求转发给出现故障的服务提供方；反之亦然。

7.1.4.2 方案二：Dubbo

Dubbo 是阿里巴巴推出的分布式服务框架，致力于解决服务的注册与发现，编排，治理。它的优点如下：

- 功能全面，易于扩展
- 支持各种序列化协议（JSON，Hession，java 序列化等）

- 支持各种 RPC 协议（HTTP，Java RMI，Dubbo 自身的 RPC 协议等）
- 支持多种负载均衡算法
- 其他高级特性：服务编排，服务治理，服务监控等

缺点如下：

- 只支持 Java，对于 Python 没有相应的支持
- 虽然已经开源，但是没有成熟的社区来运营和维护，未来升级可能是个麻烦
- 重量级的解决方案带来新的复杂性

7.1.4.3 方案三：Zookeeper

Zookeeper 是什么？按照 Apache 官网的描述是：

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

参照官网的定义，它能够做：

1. 作为配置信息的存储的中心服务器
2. 命名服务
3. 分布式的协调
4. Master 选举等

在定义中特别提到了命名服务。在调研之后，Zookeeper 作为服务注册与发现的解决方案，它有如下优点：

1. 它提供的简单 API
2. 已有互联网公司(例如：Pinterest，Airbnb)使用它来进行服务注册与发现
3. 支持多语言的客户端
4. 通过 Watcher 机制实现 Push 模型，服务注册信息的变更能够及时通知服务消费方

缺点是：

1. 引入新的 Zookeeper 组件，带来新的复杂性和运维问题
2. 需自己通过它提供的 API 来实现服务注册与发现逻辑（包含 Python 与 Java 版本）

我们对上述几个方案的优缺点权衡之后，决定采用了基于 Zookeeper 实现自己的服务注册与发现。

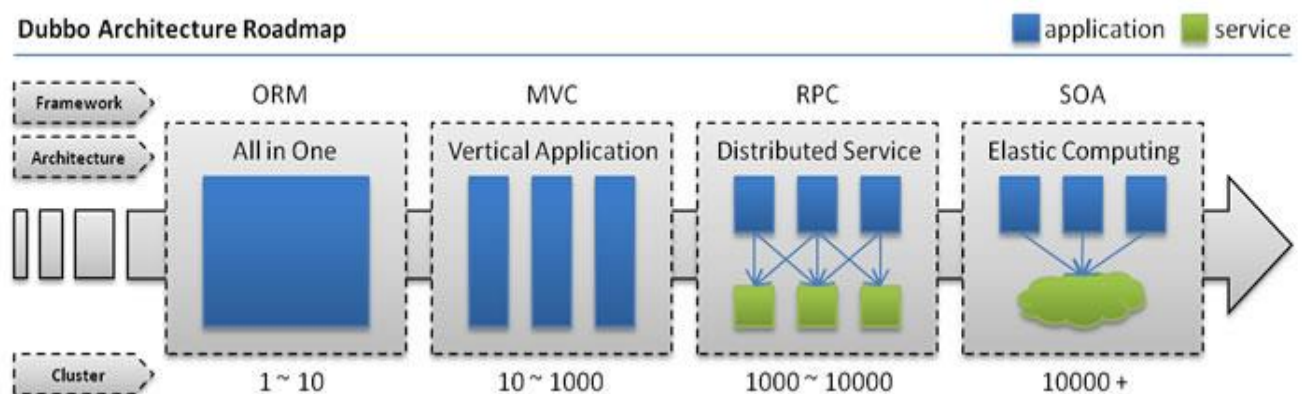
7.2 DUBBO 分布式服务架构与 Zookeeper 实现服务提供和消费

文章地址: <http://blog.csdn.net/boonya/article/details/69397962>

DUBBO 是一个分布式服务框架, 致力于提供高性能和透明化的 RPC 远程服务调用方案, 是阿里巴巴 SOA 服务化治理方案的核心框架, 每天为 2,000+ 个服务提供 3,000,000,000+ 次访问量支持, 并被广泛应用于阿里巴巴集团的各成员站点。

7.2.1 背景

随着互联网的发展, 网站应用的规模不断扩大, 常规的垂直应用**架构**已无法应对, 分布式服务架构以及流动计算架构势在必行, 亟需一个治理系统确保架构有条不紊的演进。



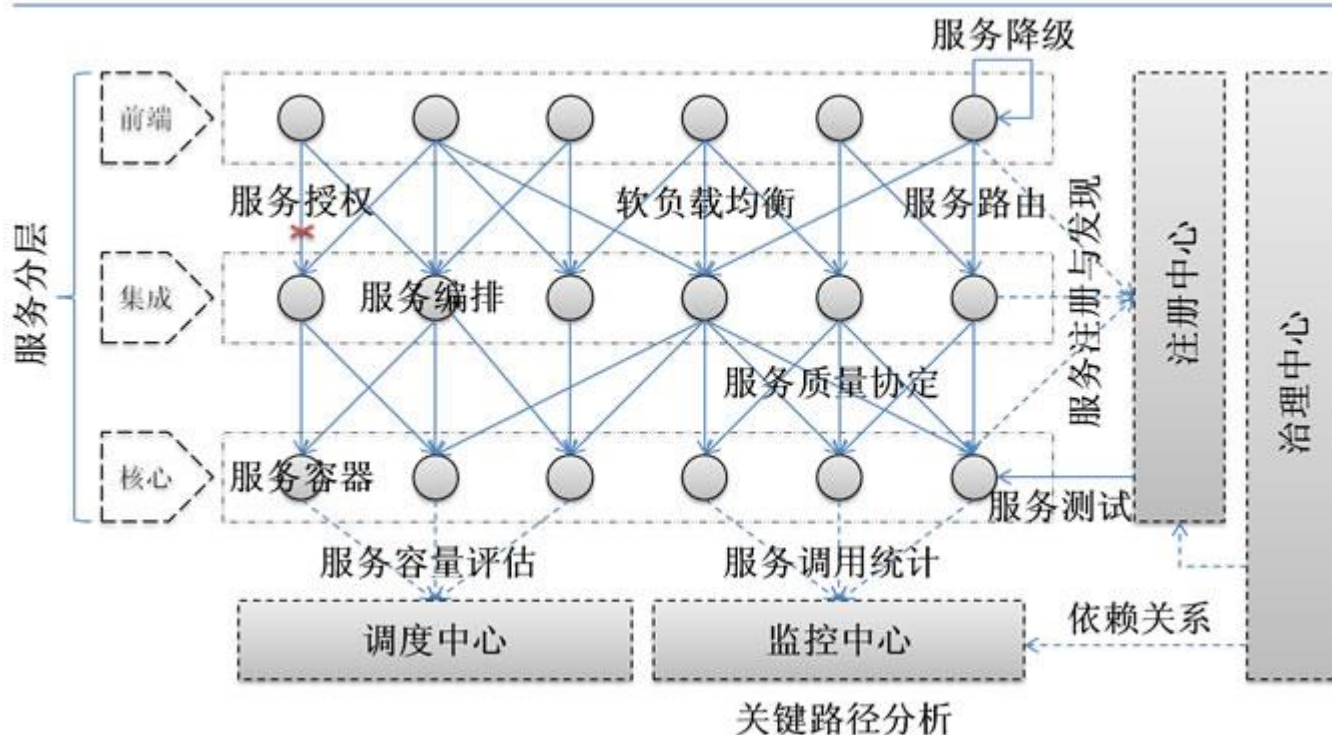
- **单一应用架构**
 - 当网站流量很小时, 只需一个应用, 将所有功能都部署在一起, 以减少部署节点和成本。
 - 此时, 用于简化增删改查工作量的 **数据访问框架(ORM)** 是关键。
- **垂直应用架构**
 - 当访问量逐渐增大, 单一应用增加机器带来的加速度越来越小, 将应用拆成互不相干的几个应用, 以提升效率。
 - 此时, 用于加速前端页面开发的 **Web 框架(MVC)** 是关键。
- **分布式服务架构**
 - 当垂直应用越来越多, 应用之间交互不可避免, 将核心业务抽取出来, 作为独立的服务, 逐渐形成稳定的服务中心, 使前端应用能更快速的响应多变的市场需求。
 - 此时, 用于提高业务复用及整合的 **分布式服务框架(RPC)** 是关键。

- 流动计算架构

- 当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。
- 此时，用于提高机器利用率的 **资源调度和治理中心(SOA)** 是关键。

7.2.2 需求

Dubbo服务治理



在大规模服务化之前，应用可能只是通过 RMI 或 Hessian 等工具，简单的暴露和引用远程服务，通过配置服务的 URL 地址进行调用，通过 F5 等硬件进行负载均衡。

(1) 当服务越来越多时，服务 URL 配置管理变得非常困难，F5 硬件负载均衡器的单点压力也越来越大。

此时需要一个服务注册中心，动态的注册和发现服务，使服务的位置透明。

并通过在消费方获取服务提供方地址列表，实现软负载均衡和 Failover，降低对 F5 硬件负载均衡器的依赖，也能减少部分成本。

(2) 当进一步发展，服务间依赖关系变得错综复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。

这时，需要自动画出应用间的依赖关系图，以帮助架构师理清关系。

(3) 接着，服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？

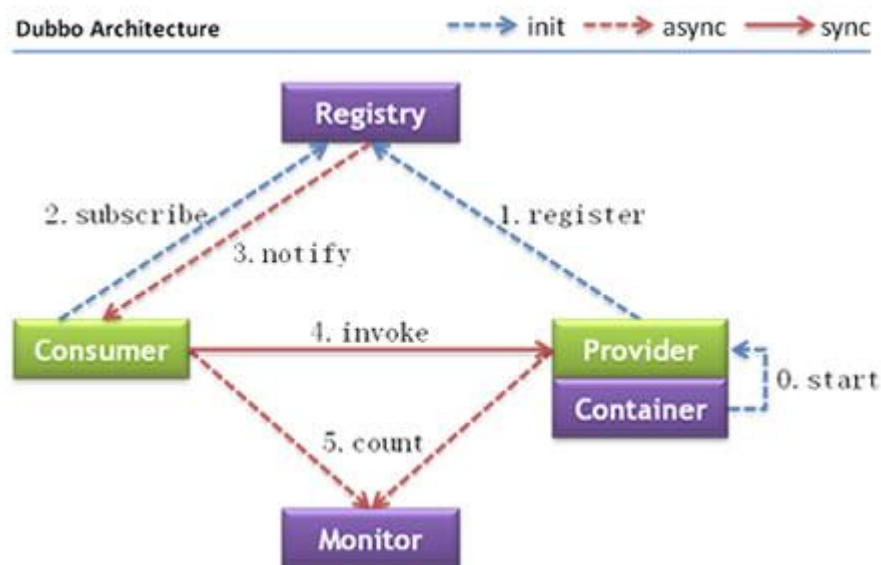
为了解决这些问题，第一步，要将服务现在每天的调用量，响应时间，都统计出来，作为容量规划的参考指标。

其次，要可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阈值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

以上是 Dubbo 最基本的几个需求，更多服务治理问题参见：

http://code.alibabatech.com/blog/experience_1402/service-governance-process.html

7.2.3 架构



节点角色说明：

- **Provider**：暴露服务的服务提供方。
- **Consumer**：调用远程服务的服务消费方。
- **Registry**：服务注册与发现的注册中心。
- **Monitor**：统计服务的调用次数和调用时间的监控中心。
- **Container**：服务运行容器。

调用关系说明：

- 0. 服务容器负责启动，加载，运行服务提供者。
- 1. 服务提供者在启动时，向注册中心注册自己提供的服务。
- 2. 服务消费者在启动时，向注册中心订阅自己所需的服务。
- 3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
- 4. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
- 5. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

(1) 连通性：

- 注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小
- 监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示
- 服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销
- 服务消费者向注册中心获取服务提供者地址列表，并根据负载均衡算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销
- 注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外
- 注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者
- 注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表
- 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者

(2) 健壮性：

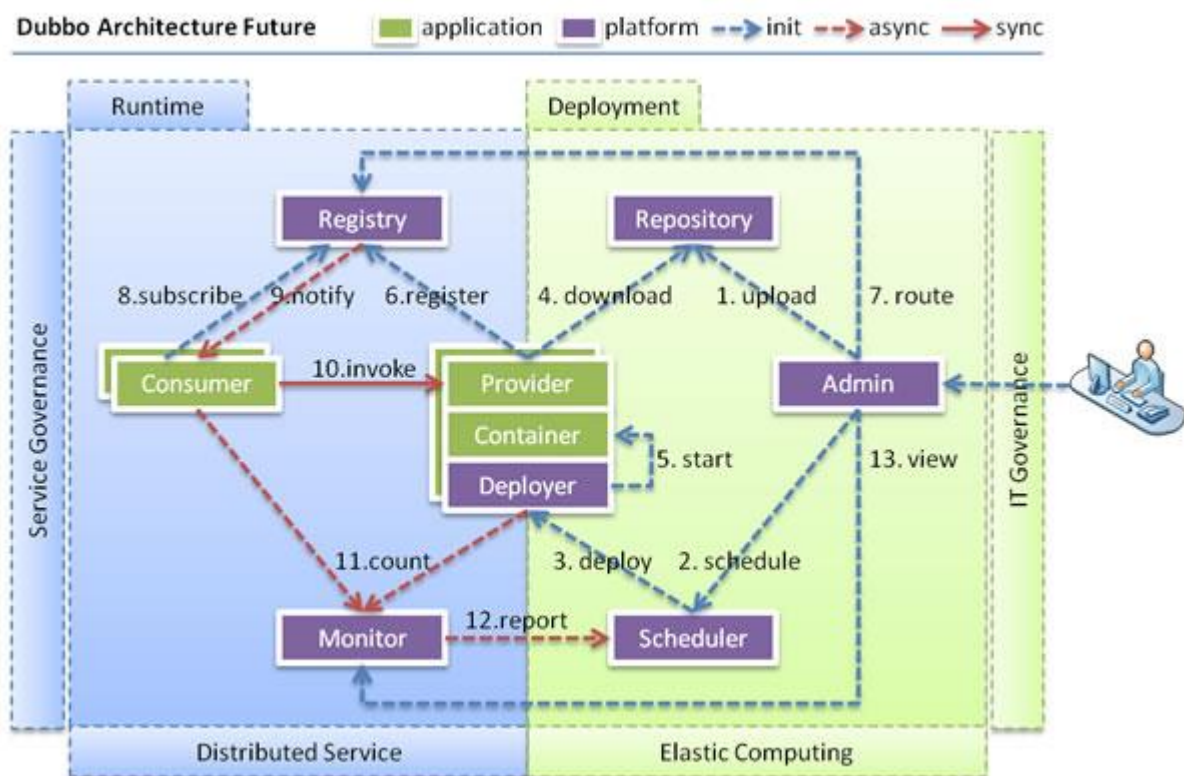
- 监控中心宕掉不影响使用，只是丢失部分采样数据
- 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
- 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
- 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
- 服务提供者无状态，任意一台宕掉后，不影响使用
- 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

(3) 伸缩性：

- 注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心
- 服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者

(4) 升级性：

- 当服务集群规模进一步扩大，带动 IT 治理结构进一步升级，需要实现动态部署，进行流动计算，现有分布式服务架构不会带来阻力：



- **Deployer:** 自动部署服务的本地代理。
- **Repository:** 仓库用于存储服务应用发布包。
- **Scheduler:** 调度中心基于访问压力自动增减服务提供者。
- **Admin:** 统一管理控制台。

7.2.4 资源

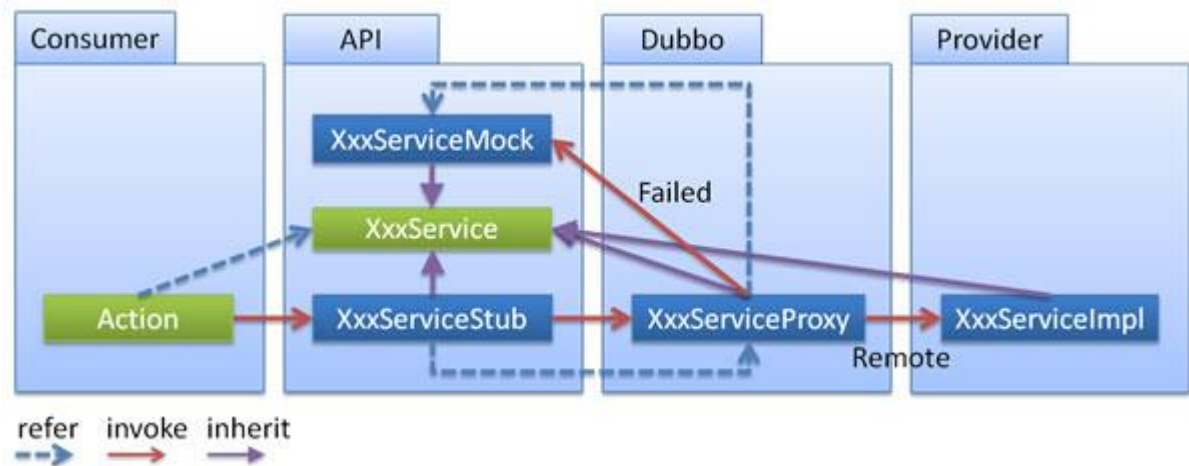
- 源码: <https://github.com/alibaba/dubbo>
- 下载: <http://repo1.maven.org/maven2/com/alibaba/dubbo>
- 微博: <http://weibo.com/dubbo>

7.2.5 文档

- [用户指南](#)
- [开发指南](#)
- [管理指南](#)
- [常见问题](#)

7.2.6 在 SpringMVC 框架中实现服务发布和消费

我们主要基于应用层面通过 zookeeper 注册接口服务。



7.2.6.1 安装 Zookeeper

请参考: [Apache Zookeeper 单个节点测试环境与集群环境设置](#)

7.2.6.2 在 Tomcat 部署 dubbo 管理界面

1. 下载 dubbo 管理界面

<http://download.csdn.NET/detail/u013286716/7041185>

2. 解压对应的 war 包，这里的 war 的版本是 2.5.3

3. 将解压后的文件全部覆盖 ROOT 下的文件；

4. 修改 \$WEB-INF\dubbo.properties 文件：

C

```
dubbo.registry.address=zookeeper://192.168.234.128:2181
```

```
dubbo.admin.root.password=root
```

```
dubbo.admin.guest.password=guest
```

注：192.168.234.128 是 Zookeeper 安装的服务器 IP 地址，对外服务端口是 2181.

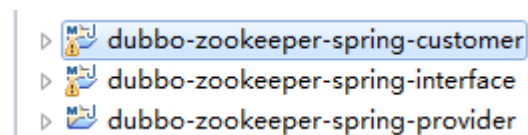
5. 到 Tomcat/bin 下启动 startup.bat/startup.sh;
6. 访问 Tomcat dubbo 管理界面, 用户名和密码都是: root;



注:访问地址自然是启动的 Tomcat 的 IP 和端口。

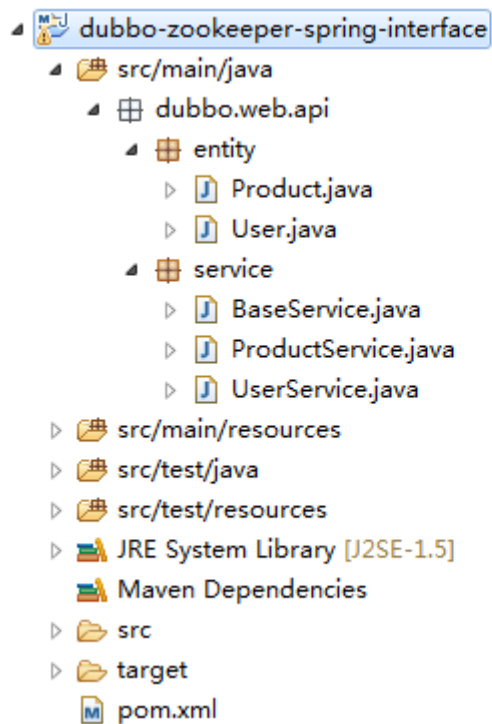
7.2.6.3 通过代码实现 Provider、API 和 Customer

首先创建三个项目分别为用户提供接口服务、定义接口和接口消费。



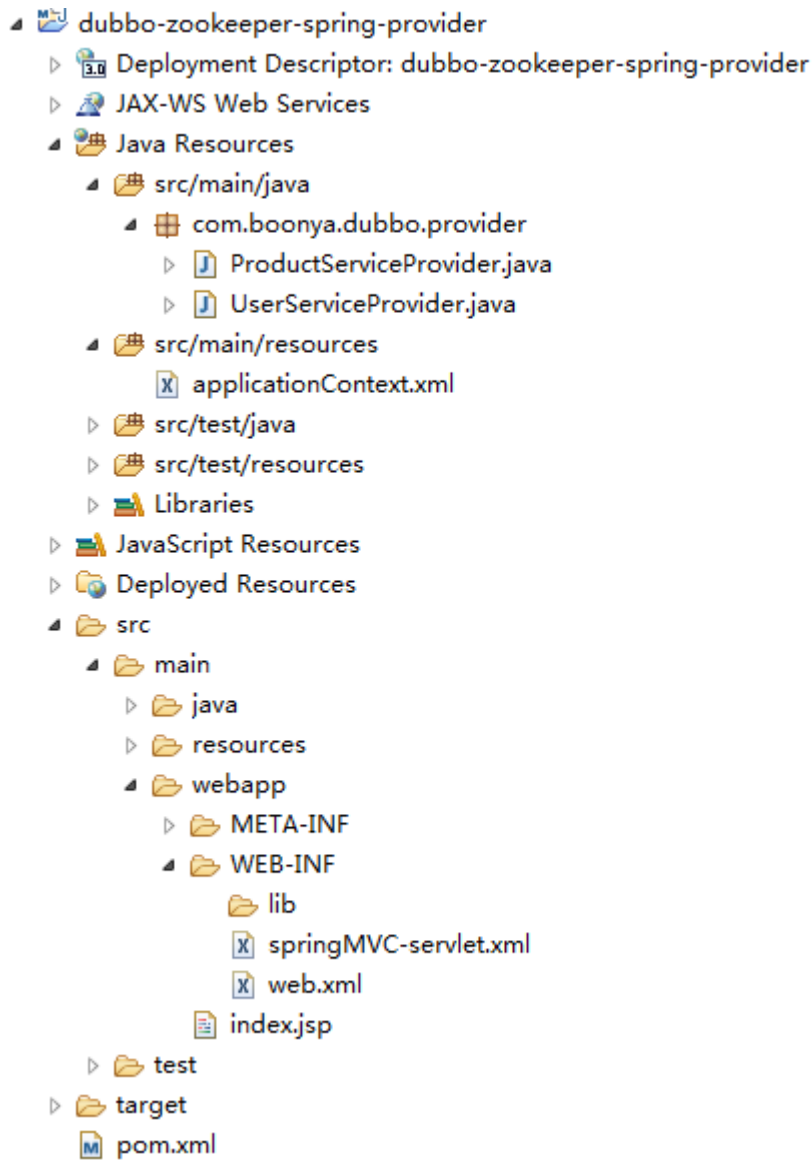
7.2.6.4 定义服务接口-Interface

通过定义普通的接口服务来实现接口的定义, 很普通的接口, 此处不再贴出代码。使用 Maven 进行依赖配置, 项目打包类型设置为 jar。



7.2.6.5 定义服务提供者-Provider

服务提供者用于实现前面定义的接口，下面的 `UserServiceProvider` 和 `ProductServiceProvider` 分别实现了前面定义的 `dubbo.web.api.service.UserService` 和 `dubbo.web.api.service.ProductService` 接口。



需要在 applicationContext.xml 文件配置的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://code.alibabatech.com/schema/dubbo
                           http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

    <!-- 提供方应用信息，用于计算依赖关系 -->
    <dubbo:application name="webapp-api-provider" />

    <!-- 使用 zookeeper 注册中心暴露服务地址 -->
    <dubbo:registry address="zookeeper://192.168.234.128:2181" />
```

```

    <!-- 用 dubbo 协议在 20881 端口暴露服务（如果有多个提供者，则端口号不能重复） -->
    <dubbo:protocol name="dubbo" port="20881" />

    <!-- 声明需要暴露的服务接口（interface 是全路径名，不能自定义） -->
    <dubbo:service interface="dubbo.web.api.service.UserService" ref="userService" />
    <!-- 和本地 bean 一样实现服务 -->
    <bean id="userService" class="com.boonya.dubbo.provider.UserServiceProvider"/>

    <!-- 声明需要暴露的服务接口（interface 是全路径名，不能自定义） -->
    <dubbo:service interface="dubbo.web.api.service.ProductService"
ref="productService" />
    <!-- 和本地 bean 一样实现服务 -->
    <bean id="productService"
class="com.boonya.dubbo.provider.ProductServiceProvider"/>

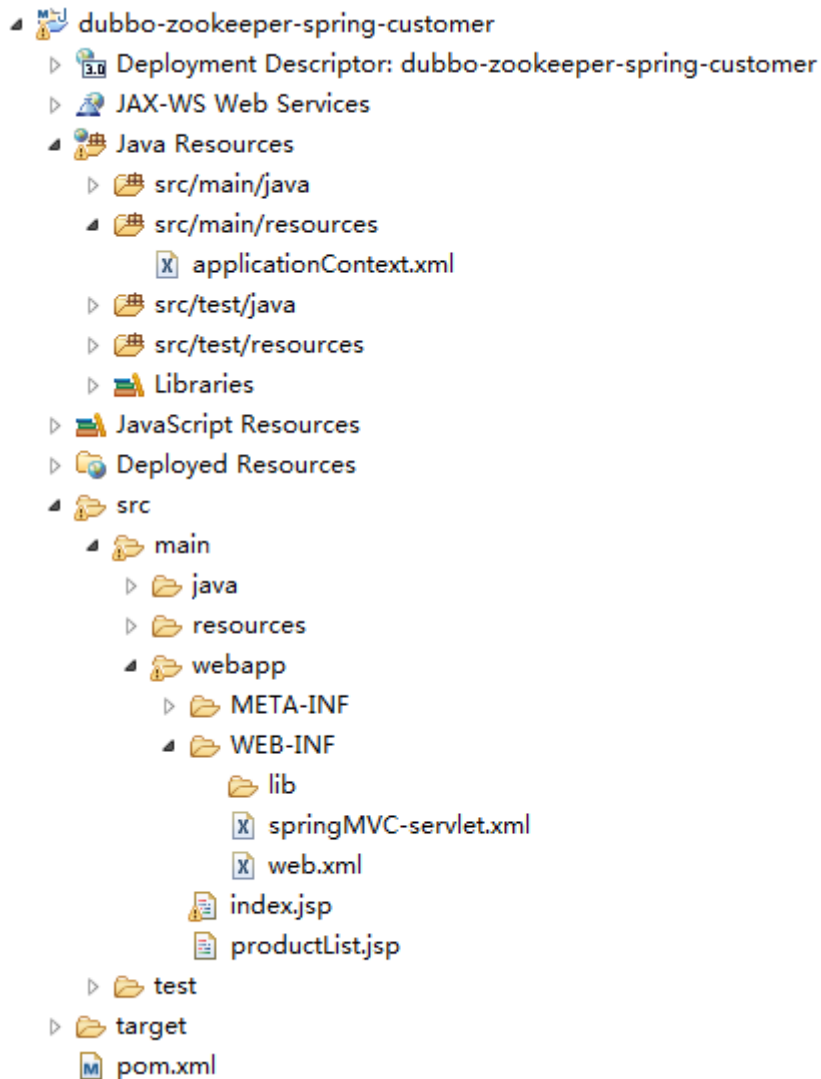
</beans>

```

该类的配置主要用于注册提供者接口服务。

7.2.6.6 定义服务消费者-Customer

服务消费者用于实现前面定义的接口，通过 Zookeeper 发现服务。我们可以通过前面定义的接口，一旦有提供者发起了相关的接口操作服务消费者即可收到信息。



applicationContext.xml 文件的配置:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://code.alibabatech.com/schema/dubbo
                           http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

    <!-- 消费方应用信息，用于计算依赖关系 -->
    <dubbo:application name="webapp-api-customer" />

    <!-- 使用 zookeeper 注册中心暴露服务地址 -->
    <dubbo:registry address="zookeeper://192.168.234.128:2181" />

    <!-- 生成远程服务代理，可以和本地 bean 一样使用 userService -->
```



```

        <dubbo:reference id="userService" interface="dubbo.web.api.service.UserService" />

        <dubbo:reference id="productService"
interface="dubbo.web.api.service.ProductService" />

</beans>

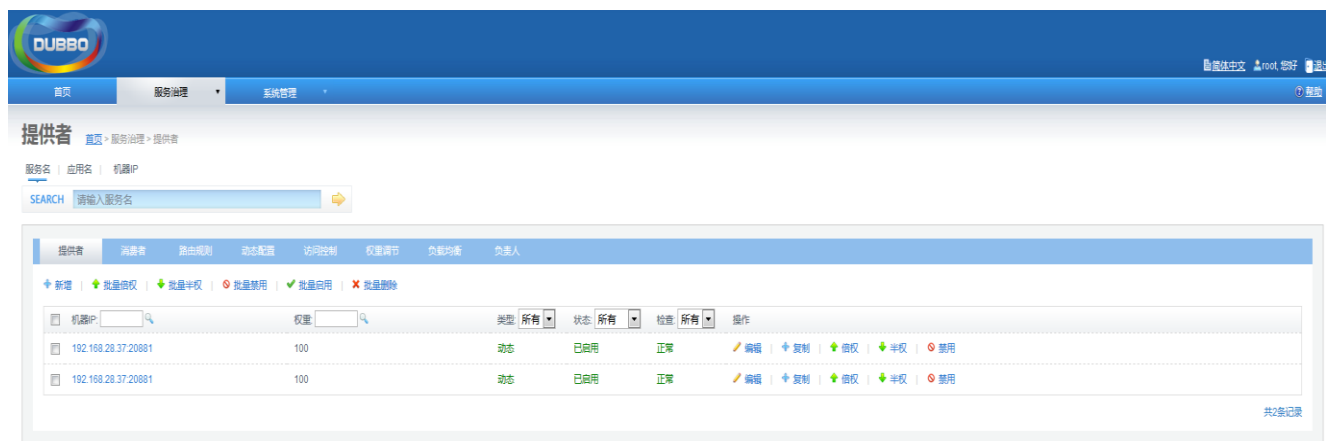
```

通过以上注册后，消费者可以对 dubbo.web.api.service.UserService 和 dubbo.web.api.service.ProductService 接口进行直接调用。

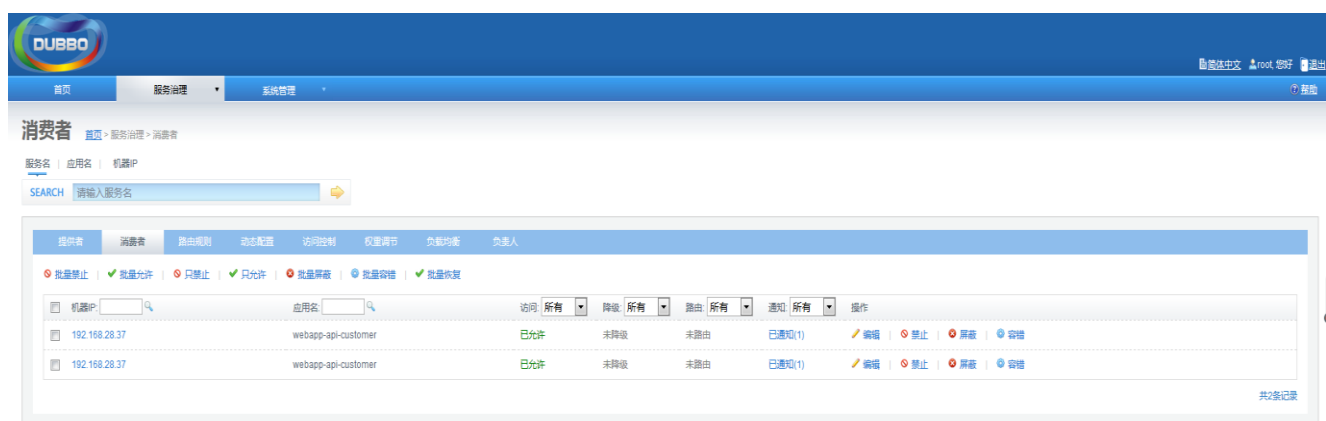
7.2.6.7 启动测试

首先启动 dubbo-zookeeper-spring-provider 然后再启动 dubbo-zookeeper-spring-customer 项目，前提是 Zookeeper 已启动。

提供者：



消费者：



7.2.6.8 示例下载

代码已上传 Github: <https://github.com/SunflowersOfJava/dubbo-zookeeper-spring.git>

7.3 使用 RMI + ZooKeeper 实现远程调用框架

原文地址: <http://www.tianshouzhi.com/api/tutorials/zookeeper/219>

在 Java 世界里,有一种技术可以实现“跨虚拟机”的调用,它就是 RMI (Remote Method Invocation, 远程方法调用)。例如,服务 A 在 JVM1 中运行,服务 B 在 JVM2 中运行,服务 A 与 服务 B 可相互进行远程调用,就像调用本地方法一样,这就是 RMI。在分布式系统中,我们使用 RMI 技术可轻松将 服务提供者 (Service Provider) 与 服务消费者 (Service Consumer) 进行分离,充分体现组件之间的弱耦合,系统架构更易于扩展。

本文先从通过一个最简单的 RMI 服务与调用示例,让读者快速掌握 RMI 的使用方法,然后指出 RMI 的局限性,最后笔者对此问题提供了一种简单的解决方案,即使用 ZooKeeper 轻松解决 RMI 调用过程中所涉及的问题。

下面我们就从一个最简单的 RMI 示例开始吧!

7.3.1 发布 RMI 服务

发布一个 RMI 服务,我们只需做三件事情:

1. 定义一个 RMI 接口
2. 编写 RMI 接口的实现类
3. 通过 JNDI 发布 RMI 服务

7.3.1.1 定义一个 RMI 接口

RMI 接口实际上还是一个普通的 Java 接口,只是 RMI 接口必须继承 `java.rmi.Remote`,此外,每个 RMI 接口的方法必须声明抛出一个 `java.rmi.RemoteException` 异常,就像下面这样:

```
1. package zookeeper_rmi;  
2.  
3. import java.rmi.Remote;  
4. import java.rmi.RemoteException;  
5.
```

```
6. public interface HelloService extends Remote {
7.
8.     String sayHello(String name) throws RemoteException;
9. }
```

继承了 Remote 接口，实际上是让 JVM 得知该接口是需要用于远程调用的，抛出了 RemoteException 是为了让调用 RMI 服务的程序捕获这个异常。毕竟远程调用过程中，什么奇怪的事情都会发生（比如：断网）。需要说明的是，RemoteException 是一个“受检异常”，在调用的时候必须使用 try...catch... 自行处理。

7.3.1.2 编写 RMI 接口的实现类

实现以上的 HelloService 是一件非常简单的事情，但需要注意的是，我们必须让实现类继承 java.rmi.server.UnicastRemoteObject 类，此外，必须提供一个构造器，并且构造器必须抛出 java.rmi.RemoteException 异常。我们既然使用 JVM 提供的这套 RMI 框架，那么就必须按照这个要求来实现，否则是无法成功发布 RMI 服务的，一句话：我们得按规矩出牌！

```
1. package zookeeper_rmi;
2.
3. import java.rmi.RemoteException;
4. import java.rmi.server.UnicastRemoteObject;
5.
6. public class HelloServiceImpl extends UnicastRemoteObject implements
   s HelloService {
7.
8.     private static final long serialVersionUID = 1L;
9.
10.    protected HelloServiceImpl() throws RemoteException {
11.    }
12.
13.    @Override
14.    public String sayHello(String name) throws RemoteException {
15.        return String.format("Hello %s", name);
16.    }
17. }
```

为了满足 RMI 框架的要求，我们确实做了很多额外的工作（继承了 UnicastRemoteObject 类，抛出了 RemoteException 异常），但这些工作阻止不了我们发布 RMI 服务的决心！我们可以通过 JVM 提供的 JNDI（Java Naming and Directory Interface，Java 命名与目录接口）这个 API 轻松发布 RMI 服务。

7.3.1.3 通过 JNDI 发布 RMI 服务

发布 RMI 服务，我们需要告诉 JNDI 三个基本信息：1. 域名或 IP 地址（host）、2. 端口号（port）、3. 服务名（service），它们构成了 RMI 协议的 URL（或称为“RMI 地址”）：

```
1. rmi://<host>:<port>/<service>
```

如果我们是本地发布 RMI 服务，那么 host 就是“localhost”。此外，RMI 默认的 port 是“1099”，我们也可以自行设置 port 的值（只要不与其它端口冲突即可）。service 实际上是一个基于同一 host 与 port 下唯一的服务名，我们不妨使用 Java 完全类名来表示吧，这样也更容易保证 RMI 地址的唯一性。

对于我们的示例而言，RMI 地址为：

```
1. rmi://localhost:1099/demo.zookeeper.remoting.server.HelloServiceImpl
```

我们只需简单提供一个 main() 方法就能发布 RMI 服务，就像下面这样：

```
1. package zookeeper_rmi;
2.
3. import java.rmi.Naming;
4. import java.rmi.registry.LocateRegistry;
5.
6. public class RmiServer {
7.
8.     public static void main(String[] args) throws Exception {
9.         int port = 1099;
10.        String url = "rmi://localhost:1099/demo.zookeeper.remoting.server.HelloServiceImpl" ;
11.        LocateRegistry.createRegistry(port);
12.        Naming.rebind(url, new HelloServiceImpl());
13.    }
14. }
```

需要注意的是，我们通过 `LocateRegistry.createRegistry()` 方法在 JNDI 中创建一个注册表，只需提供一个 RMI 端口号即可。此外，通过 `Naming.rebind()` 方法绑定 RMI 地址与 RMI 服务实现类，这里使用了 `rebind()` 方法，它相当于先后调用 `Naming` 的 `unbind()` 与 `bind()` 方法，只是使用 `rebind()` 方法来得更加痛快而已，所以我们选择了它。

运行这个 `main()` 方法，RMI 服务就会自动发布，剩下要做的就是写一个 RMI 客户端来调用已发布的 RMI 服务。

7.3.2 调用 RMI 服务

同样我们也使用一个 `main()` 方法来调用 RMI 服务，相比发布而言，调用会更加简单，我们只需要知道两个东西：1. RMI 请求路径、2. RMI 接口（一定不需要 RMI 实现类，否则就是本地调用了）。数行代码就能调用刚才发布的 RMI 服务，就像下面这样：

```
1. package zookeeper_rmi;
2.
3. import java.rmi.Naming;
4.
5. public class RmiClient {
6.
7.     public static void main(String[] args) throws Exception {
8.         String url = "rmi://localhost:1099/demo.zookeeper.remoti
ng.server.HelloServiceImpl" ;
9.         HelloService helloService = (HelloService) Naming.looku
p( url);
10.        String result = helloService .sayHello("Jack");
11.        System. out.println(result );
12.    }
13. }
```

当我们运行以上 `main()` 方法，在控制台中看到“Hello Jack”输出，就表明 RMI 调用成功。

7.3.3 RMI 服务的局限性

可见，借助 JNDI 这个所谓的命名与目录服务，我们成功地发布并调用了 RMI 服务。实际上，JNDI 就是一个注册表，服务端将服务对象放入到注册表中，客户端从注册表中获取服务对象。在服务端我们发布了 RMI 服务，并在 JNDI 中进行了注册，此时就在服务端创建了一个 Skeleton（骨架），当客户端第一次成功连接 JNDI 并获取远程服务对象后，立马就在本地创建了一个 Stub（存根），远程通信实际上是通过 Skeleton 与 Stub 来完成的，数据是基于 TCP/IP 协议，在“传输层”上发送的。毋庸置疑，理论上 RMI 一定比 WebService 要快，毕竟 WebService 是基于 HTTP 的，而 HTTP 所携带的数据是通过“应用层”来传输的，传输层较应用层更为底层，越底层越快。

既然 RMI 比 WebService 快，使用起来也方便，那么为什么我们有时候还要用 WebService 呢？

其实原因很简单，WebService 可以实现跨语言系统之间的调用，而 RMI 只能实现 Java 系统之间的调用。也就是说，RMI 的跨平台性不如 WebService 好，假如我们的系统都是用 Java 开发的，那么当然首选就是 RMI 服务了。

貌似 RMI 确实挺优秀的，除了不能跨平台以外，还有那些问题呢？

笔者认为有两点局限性：

1. RMI 使用了 Java 默认的序列化方式，对于性能要求比较高的系统，可能需要使用其它序列化方案来解决（例如：Proto Buffer）。

2. RMI 服务在运行时难免会存在出故障，例如，如果 RMI 服务无法连接了，就会导致客户端无法响应的现象。

在一般的情况下，Java 默认的序列化方式确实已经足以满足我们的要求了，如果性能方面如果不是问题的话，我们需要解决的实际上是第二点，也就是说，让使系统具备 HA（High Availability，高可用性）。

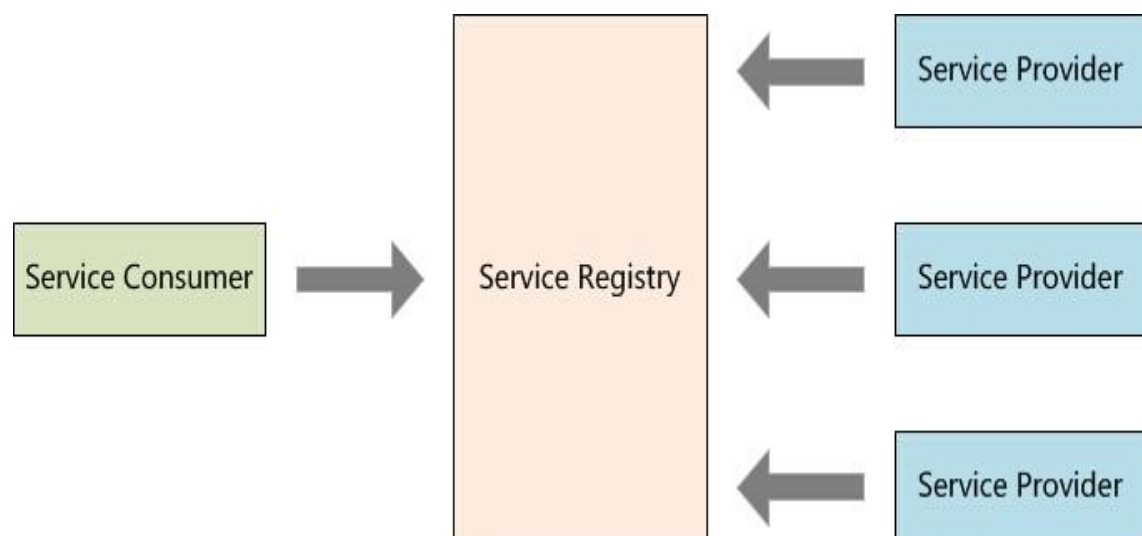
7.3.4 使用 ZooKeeper 提供高可用的 RMI 服务

ZooKeeper 是 Hadoop 的一个子项目，用于解决分布式系统之间的数据一致性问题。如果读者尚不了解 ZooKeeper 的工作原理与使用方法，可以通过以下链接来了解：

* ZooKeeper 官网

* 分布式服务框架 ZooKeeper - 管理分布式环境中的数据

要想解决 RMI 服务的高可用性问题，我们需要利用 ZooKeeper 充当一个 服务注册表（Service Registry），让多个 服务提供者（Service Provider）形成一个集群，让 服务消费者（Service Consumer）通过服务注册表获取具体的服务访问地址（也就是 RMI 服务地址）去访问具体的服务提供者。如下图所示：



需要注意的是，服务注册表并不是 Load Balancer（负载均衡器），提供的不是“反向代理”服务，而是“服务注册”与“心跳检测”功能。

利用服务注册表来注册 RMI 地址，这个很好理解，那么“心跳检测”又如何理解呢？说白了就是通过服务中心定时向各个服务提供者发送一个请求（实际上建立的是一个 Socket 长连接），如果长期没有响应，服务中心就认为该服务提供者已经“挂了”，只会从还“活着”的服务提供者中选出一个做为当前的服务提供者。

也许读者会考虑到，服务中心可能会出现单点故障，如果服务注册表都坏掉了，整个系统也就瘫痪了。看来要想实现这个架构，必须保证服务中心也具备高可用性。

ZooKeeper 正好能够满足我们上面提到的所有需求。

1. 使用 ZooKeeper 的临时性 ZNode 来存放服务提供者的 RMI 地址，一旦与服务提供者的 Session 中断，会自动清除相应的 ZNode。
2. 让服务消费者去监听这些 ZNode，一旦发现 ZNode 的数据（RMI 地址）有变化，就会重新获取一份有效数据的拷贝。
3. ZooKeeper 与生俱来的集群能力（例如：数据同步与领导选举特性），可以确保服务注册表的高可用性。

7.3.4.1 服务提供者

需要编写一个 ServiceProvider 类，来发布 RMI 服务，并将 RMI 地址注册到 ZooKeeper 中（实际存放在 ZNode 上）。

```
1. package zookeeper_rmi;
2.
3. import java.io.IOException;
4. import java.net.MalformedURLException;
5. import java.rmi.Naming;
6. import java.rmi.Remote;
7. import java.rmi.RemoteException;
8. import java.rmi.registry.LocateRegistry;
9. import java.util.concurrent.CountDownLatch;
10.
11. import org.apache.zookeeper.CreateMode;
12. import org.apache.zookeeper.KeeperException;
13. import org.apache.zookeeper.WatchedEvent;
14. import org.apache.zookeeper.Watcher;
15. import org.apache.zookeeper.ZooDefs;
```

```
16. import org.apache.zookeeper.ZooKeeper;
17. import org.slf4j.Logger;
18. import org.slf4j.LoggerFactory;
19.
20. public class ServiceProvider {
21.
22.     private static final Logger LOGGER = LoggerFactory.getLogger
        (ServiceProvider.class);
23.
24.     // 用于等待 SyncConnected 事件触发后继续执行当前线程
25.     private CountDownLatch latch = new CountDownLatch(1);
26.
27.     // 发布 RMI 服务并注册 RMI 地址到 ZooKeeper 中
28.     public void publish(Remote remote , String host, int port) {
29.         String url = publishService(remote , host , port
        ); // 发布 RMI 服务并返回 RMI 地址
30.         if (url != null) {
31.             ZooKeeper zk = connectServer(); // 连接 ZooKeeper 服务器并获取 ZooKeeper 对象
32.             if (zk != null) {
33.                 createNode( zk, url); // 创建 ZNode 并将 RMI 地址放入 ZNode 上
34.             }
35.         }
36.     }
37.
38.     // 发布 RMI 服务
39.     private String publishService(Remote remote, String host , int port) {
40.         String url = null ;
41.         try {
42.             url = String.format( "rmi://%s:%d/%s", host , port , remote.getClass().getName());
43.             LocateRegistry.createRegistry(port);
44.             Naming.rebind(url, remote);
45.             LOGGER.debug("publish rmi service (url: {})
        " , url );
46.         } catch (RemoteException | MalformedURLException e) {
47.             LOGGER.error(" " , e );
48.         }
49.         return url ;
50.     }
51. }
```



```

52.      // 连接 ZooKeeper 服务器
53.      private ZooKeeper connectServer() {
54.          ZooKeeper zk = null;
55.          try {
56.              zk = new ZooKeeper(Constant.ZK_CONNECTION_STRING,
57.                  Constant.ZK_SESSION_TIMEOUT, new Watcher() {
58.                      @Override
59.                      public void process(WatchedEvent event) {
60.                          if (event.getState() == Event.KeeperState.SyncConnected) {
61.                              latch.countDown(); // 唤醒当前正在执行的线程
62.                          }
63.                      }
64.                  });
65.              latch.await(); // 使当前线程处于等待状态
66.          } catch (IOException | InterruptedException e) {
67.              LOGGER.error("", e);
68.          }
69.          return zk;
70.      }
71.      // 创建 ZNode
72.      private void createNode(ZooKeeper zk, String url) {
73.          try {
74.              byte[] data = url.getBytes();
75.              String path = zk.create(Constant.ZK_PROVIDER_PATH, data, ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL); // 创建一个临时性且有序的 ZNode
76.              LOGGER.debug("create zookeeper node ({} => {})", path, url);
77.          } catch (KeeperException | InterruptedException e) {
78.              LOGGER.error("", e);
79.          }
80.      }
81.  }

```

涉及到的 Constant 常量，见如下代码：

```

1. package zookeeper_rmi;
2.
3. public interface Constant {
4.     String ZK_CONNECTION_STRING = "localhost:2181";

```

```
5.         int ZK_SESSION_TIMEOUT = 5000;
6.         String ZK_REGISTRY_PATH = "/registry";
7.         String ZK_PROVIDER_PATH = ZK_REGISTRY_PATH + "/provider" ;
8.     }
```

注意：我们首先需要使用 ZooKeeper 的客户端工具创建一个持久性 ZNode，名为“/registry”，该节点是不存放任何数据的，可使用如下命令：

```
1. create /registry null
```

7.3.4.2 服务消费者

服务消费者需要在创建的时候连接 ZooKeeper，同时监听 /registry 节点的 NodeChildrenChanged 事件，也就是说，一旦该节点的子节点有变化，就需要重新获取最新的子节点。这里提到的子节点，就是存放服务提供者发布的 RMI 地址。需要强调的是，这些子节点都是临时性的，当服务提供者与 ZooKeeper 服务注册表的 Session 中断后，该临时性节点会被自动删除。

```
1. package master_worker_example;
2.
3. import java.io.IOException;
4. import java.net.MalformedURLException;
5. import java.rmi.ConnectException;
6. import java.rmi.Naming;
7. import java.rmi.NotBoundException;
8. import java.rmi.Remote;
9. import java.rmi.RemoteException;
10. import java.util.ArrayList;
11. import java.util.List;
12. import java.util.concurrent.CountDownLatch;
13. import java.util.concurrent.ThreadLocalRandom;
14.
15. import org.apache.zookeeper.KeeperException;
16. import org.apache.zookeeper.WatchedEvent;
17. import org.apache.zookeeper.Watcher;
18. import org.apache.zookeeper.ZooKeeper;
19. import org.slf4j.Logger;
20. import org.slf4j.LoggerFactory;
21.
22. import zookeeper_rmi.Constant;
23.
24. public class ServiceConsumer {
25.
```

```

26.     private static final Logger LOGGER = LoggerFactory.getLogger
        (ServiceConsumer.class);
27.
28.     // 用于等待 SyncConnected 事件触发后继续执行当前线程
29.     private CountDownLatch latch = new CountDownLatch(1);
30.
31.     // 定义一个 volatile 成员变量，用于保存最新的 RMI 地址（考虑到
        该变量或许会被其它线程所修改，一旦修改后，该变量的值会影响到所有线程）
32.     private volatile List<String> urlList = new ArrayList<>();
33.
34.     // 构造器
35.     public ServiceConsumer() {
36.         ZooKeeper zk = connectServer(); // 连接 ZooKeeper 服
        务器并获取 ZooKeeper 对象
37.         if (zk != null) {
38.             watchNode( zk); // 观察 /registry 节点的所有子
        节点并更新 urlList 成员变量
39.         }
40.     }
41.
42.     // 查找 RMI 服务
43.     public <T extends Remote> T lookup() {
44.         T service = null ;
45.         int size = urlList.size();
46.         if (size > 0) {
47.             String url;
48.             if (size == 1) {
49.                 url = urlList .get(0); // 若 urlList
        中只有一个元素，则直接获取该元素
50.                 LOGGER.debug("using only url: {}" , url
        1 );
51.             } else {
52.                 url = urlList.get(ThreadLocalRandom.curre
        nt().nextInt( size)); // 若 urlList 中存在多个元素，则随机获取一个元素
53.                 LOGGER.debug("using random url: {}
        " , url );
54.             }
55.             service = lookupService( url); // 从 JNDI 中
        查找 RMI 服务
56.         }
57.         return service ;
58.     }
59.
60.     // 连接 ZooKeeper 服务器

```

```

61.         private ZooKeeper connectServer() {
62.             ZooKeeper zk = null;
63.             try {
64.                 zk = new ZooKeeper(Constant.ZK_CONNECTION_STRIN
G, Constant.ZK_SESSION_TIMEOUT , new Watcher() {
65.                     @Override
66.                     public void process(WatchedEvent even
t) {
67.                         if (event .getState() == Event.
KeeperState.SyncConnected ) {
68.                             latch.countDown(); // 唤
醒当前正在执行的线程
69.                         }
70.                     }
71.                 });
72.                 latch.await(); // 使当前线程处于等待状态
73.             } catch (IOException | InterruptedException e) {
74.                 LOGGER.error(" ", e );
75.             }
76.             return zk ;
77.         }
78.
79.         // 观察 /registry 节点下所有子节点是否有变化
80.         private void watchNode(final ZooKeeper zk) {
81.             try {
82.                 List<String> nodeList = zk.getChildren(Constant.
ZK_REGISTRY_PATH, new Watcher() {
83.                     @Override
84.                     public void process(WatchedEvent even
t) {
85.                         if (event .getType() == Event.E
ventType.NodeChildrenChanged ) {
86.                             watchNode( zk); // 若子
节点有变化，则重新调用该方法（为了获取最新子节点中的数据）
87.                         }
88.                     }
89.                 });
90.                 List<String> dataList = new ArrayList<>(); /
/ 用于存放 /registry 所有子节点中的数据
91.                 for (String node : nodeList) {
92.                     byte[] data = zk.getData(Constant. ZK_R
EGISTRY_PATH + "/" + node, false , null); // 获取 /registry 的子
节点中的数据
93.                     dataList.add(new String(data));

```

```

94.         }
95.         LOGGER.debug("node data: {}" , dataList );
96.         urlList = dataList ; // 更新最新的 RMI 地址
97.     } catch (KeeperException | InterruptedException e) {
98.         LOGGER.error("", e );
99.     }
100. }
101.
102. // 在 JNDI 中查找 RMI 远程服务对象
103. @SuppressWarnings( "unchecked")
104. private <T> T lookupService(String url) {
105.     T remote = null ;
106.     try {
107.         remote = (T) Naming.lookup( url);
108.     } catch (NotBoundException | MalformedURLException |
109.             RemoteException e) {
110.         if (e instanceof ConnectException) {
111.             // 若连接中断，则使用 urlList 中第一个
112.             // RMI 地址来查找（这是一种简单的重试方式，确保不会抛出异常）
113.             LOGGER.error("ConnectException -> url: {}" , url );
114.             if (urlList .size() != 0) {
115.                 url = urlList .get(0);
116.                 return lookupService( url);
117.             }
118.             LOGGER.error("", e );
119.         }
120.         return remote ;
121.     }
122. }

```

7.3.4.3 发布服务

我们需要调用 ServiceProvider 的 publish() 方法来发布 RMI 服务，发布成功后也会自动在 ZooKeeper 中注册 RMI 地址。

```

1. package zookeeper_rmi;
2.
3. public class Server {
4.
5.     public static void main(String[] args) throws Exception {
6.         if (args .length != 2) {

```

```

7.          System.err.println("please using command: java
a Server <rmi_host> <rmi_port>");
8.          System.exit(-1);
9.      }
10.
11.      String host = args[0];
12.      int port = Integer.parseInt(args[1]);
13.
14.      ServiceProvider provider = new ServiceProvider();
15.
16.      HelloService helloService = new HelloServiceImpl();
17.      provider.publish(helloService, host, port);
18.
19.      Thread.sleep(Long.MAX_VALUE);
20.  }
21. }

```

注意：在运行 Server 类的 main() 方法时，一定要使用命令行参数来指定 host 与 port，例如：

```

1. java Server localhost 1099
2. java Server localhost 2099

```

以上两条 Java 命令可在本地运行两个 Server 程序，当然也可以同时运行更多的 Server 程序，只要 port 不同就行。

7.3.4.4 调用服务

通过调用 ServiceConsumer 的 lookup() 方法来查找 RMI 远程服务对象。我们使用一个“死循环”来模拟每隔 3 秒钟调用一次远程方法。

```

1. package zookeeper_rmi;
2.
3. import master_worker_example.ServiceConsumer;
4.
5. public class Client {
6.
7.     public static void main(String[] args) throws Exception {
8.         ServiceConsumer consumer = new ServiceConsumer();
9.
10.        while (true) {
11.            HelloService helloService = consumer.lookup();
12.            String result = helloService.sayHello("Jack");

```

```
13.                System.out.println(result );
14.                Thread.sleep(3000);
15.            }
16.        }
17.    }
```

4.5 使用方法根据以下步骤验证 RMI 服务的高可用性：

1. 运行两个 Server 程序，一定要确保 port 是不同的。
2. 运行一个 Client 程序。
3. 停止其中一个 Server 程序，并观察 Client 控制台的变化（停止一个 Server 不会导致 Client 端调用失败）。
4. 重新启动刚才关闭的 Server 程序，继续观察 Client 控制台变化（新启动的 Server 会加入候选）。
5. 先后停止所有的 Server 程序，还是观察 Client 控制台变化（Client 会重试连接，多次连接失败后，自动关闭）。


5 总结通过本文，我们尝试使用 ZooKeeper 实现了一个简单的 RMI 服务高可用性解决方案，通过 ZooKeeper 注册所有服务提供者发布的 RMI 服务，让服务消费者监听 ZooKeeper 的 Znode，从而获取当前可用的 RMI 服务。此方案局限于 RMI 服务，对于任何形式的服务（比如：WebService），也提供了一定参考。




如果再配合 ZooKeeper 自身的集群，那才是一个相对完美的解决方案，对于 ZooKeeper 的集群，请读者自行实践。

由于笔者水平有限，对于描述有误之处，还请各位读者提出建议，并期待更加优秀的解决方案。

7.4 Zookeeper 编程客户端






<https://github.com/apache/curator>


apache / curator
 mirrored from [git://git.apache.org/curator.git](https://git.apache.org/curator.git)

 Watch 91
  Star 573
  Fork 354


[Code](#)
[Pull requests 23](#)
[Projects 0](#)
[Pulse](#)
[Graphs](#)

















Mirror of Apache Curator

 1,884 commits
  208 branches
  102 releases
  58 contributors
  Apache-2.0

Branch: master
 [New pull request](#)

[Create new file](#)
[Upload files](#)
[Find file](#)
[Clone or download](#)

 randgalt Closes #212 Latest commit 9Feb31 7 hours ago

 curator-client	check empty connection string in FixedEnsembleProvider	a month ago
 curator-examples	[maven-release-plugin] prepare for next development iteration	a month ago
 curator-framework	[maven-release-plugin] prepare for next development iteration	a month ago
 curator-recipes	Merge branch 'CURATOR-391'	22 days ago
 curator-test	[maven-release-plugin] prepare for next development iteration	a month ago
 curator-x-discovery-server	[maven-release-plugin] prepare for next development iteration	a month ago
 curator-x-discovery	[maven-release-plugin] prepare for next development iteration	a month ago
 curator-x-rpc	[maven-release-plugin] prepare for next development iteration	a month ago
 src	minor reformat	24 days ago
 .gitignore	don't ignore patch files	4 years ago
 DEPENDENCIES	Defeat Maven's auto generated DEPENDENCIES file	4 years ago
 LICENSE	CURATOR-119 Match LICENSE file to ASF 'Gold Copy'	3 years ago
 NOTICE	Updated copyright	3 years ago
 README.md	[CURATOR-390] Create better summary of what Apache Curator is/does	27 days ago
 doap.rdf	updated the DOAP file	7 days ago
 pom.xml	add fangmin to the developers section	10 days ago

What's is Apache Curator?

Apache Curator is a Java/JVM client library for Apache ZooKeeper[1], a distributed coordination service.

Apache Curator includes a high-level API framework and utilities to make using Apache ZooKeeper much easier and more reliable. It also includes recipes for common use cases and extensions such as service discovery and a Java 8 asynchronous DSL. For more details, please visit the project website: <http://curator.apache.org/>

[1] Apache ZooKeeper <https://zookeeper.apache.org/>