目　录

- 致谢
- Introduction
- 如果没有zookeeper世界会怎样
- Zookeeper的使用
- Part1:概念和基础
  - 简介
    - Zookeeper的使命
      - 没有Zookeeper世界会怎样
      - 哪些事情Zookeeper不做
      - Apache项目
      - 利用Zookeeper构建分布式系统
    - 示例：Master-Worker应用
      - Master失败
      - Worker失败
      - 通信失败
      - 任务总结
    - 为什么分布式协作如此困难
    - Zookeeper是成功的，但是有警告的
  - 初识Zookeeper
    - Zookeeper基础
      - API总览
      - Znode的不同模式
      - 监视器和通知
      - 版本
    - Zookeeper架构
      - Zookeeper仲裁
      - 会话
    - 开始使用Zookeeper
      - 第一个Zookeeper会话
      - 会话的生命周期和状态
      - 有仲裁者的Zookeeper
      - 实现一个原语：利用Zookeeper实现锁
    - 实现Master-Worker的例子

Master的角色

工人、任务和指派关系

Worker的角色

客户端的角色

小结

致谢

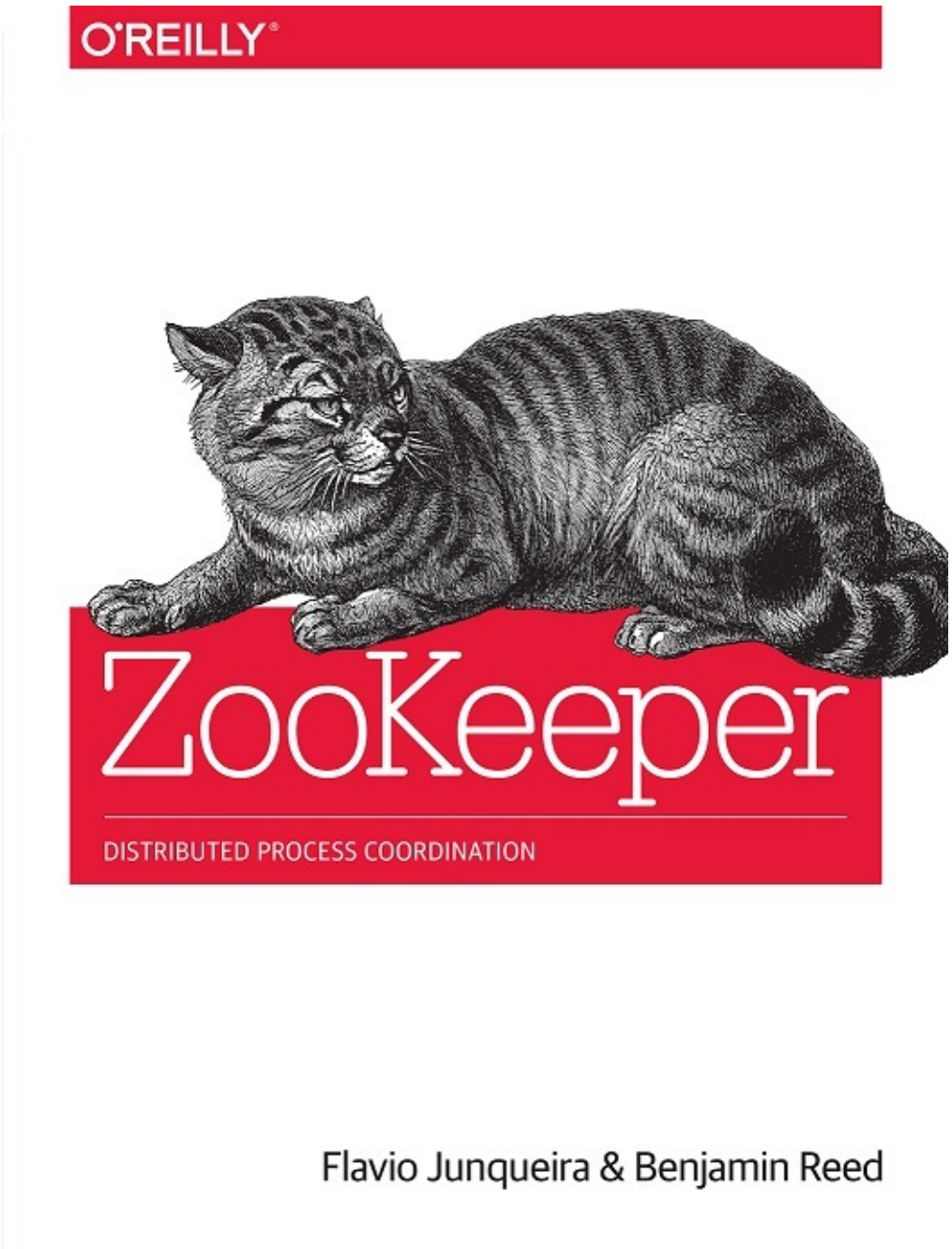
当前文档 《zookeeper:分布式进程协同》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-04-30。

Introduction

- zookeeper:分布式进程协同
  - Zookeeper:Distributed Process Coordination中文译本
  - GitBook阅读地址
  - GitHub阅读地址
- Progress
- MIT License
- 来源(书栈小编注)

zookeeper:分布式进程协同

Zookeeper:Distributed Process Coordination中文译本



GitBook阅读地址

GitHub阅读地址

Progress

很遗憾，因为授权的问题，不得不停止翻译的工作。本书已经有中文版的译本了，我后来才得知，所以我也不会取得中文版的翻译授权了。因为本人第一次翻译，事先没有搞清这些事情，才导致了现在

的情况。不得不说，十分遗憾，感谢关注本书翻译的伙伴。我相信已有的中文译本应该还不错，如果需要的伙伴可以去购买。So, that’s it, it’s over, thanks for your attention.

章节	完成情况
第一章	100%
第二章	100%
第三章	未开始
第四章	未开始
第五章	未开始
第六章	未开始
第七章	未开始
第八章	未开始
第九章	未开始
第十章	未开始

MIT License

来源(书栈小编注)

> *https://github.com/mymonkey110/zookeeper-book*

如果没有zookeeper世界会怎样

拥有Zookeeper使得一整个新类别的应用得以被开发？似乎并不是这样。Zookeeper简化了开发的进程，是的开发更加的敏捷，同时又能提供更加鲁棒性的实现。

以前系统也实现过像分布式锁管理者这样的组件或者使用分布式数据库用户协同服务。实际上，Zookeeper借用了很多来自以前系统的概念。然而，它不会暴露出一个锁服务的接口或者一个存储数据的通用接口。Zookeeper被设计成是专有的并且非常关注与协同任务。同时，它不会试着强加一组特定同步原语给开发者，这样使得能够实现的东西更加的灵活。

当然，没有Zookeeper也能构建分布式系统。然而，Zookeeper提供给开发者专注于应用逻辑而不是难懂的分布式系统概念的机会。没有Zookeeper进行分布式系统编程是可行的，但是会更加困难。

Zookeeper的使用

- Zookeeper的使用

Zookeeper的使用

Zookeeper对我们的意义就像解释螺丝刀对我们的意义一样。简而言之，螺丝刀允许我们拧螺丝，但是仅仅这样说不能表达出这个工具的能力。螺丝刀允许我们组装家具和电子设备，例如在墙上挂一副画。通过给出更多的这样的例子，我们对螺丝刀能做的事情有个大致的感觉，但是这样去描述也不够完整。

Zookeeper这样的系统能为我们做什么可以归结于这一点：它在分布式系统中能提供协同任务。协同任务一个有着多个进程参与的任务。比如，一个任务以协作或者协调为目的。协作意味着参与的进程需要一起做一些事情，一些进程需要执行某种动作让其他的进程继续运行。例如，在典型的master-worker架构中，当worker可用时它会通知master，Master随后会指派任务给worker。当遇到这种情况是竞争是不一样的，比如两个进程不能并发的运行，一个进程必须等待另外一个进程。还是那个master-worker的例子，我们只想要一个master，但是多个进程可能都会尝试变成master。多进程然后需要实现互斥操作。我们实际上希望获取领导权的任务像获取锁一样：那个获取了领导权锁的进程会变成master的角色。

如果你有过多线程编程的经验，你会意识到它们之间有很多的相似的问题。实际上，有着多个进程运行在同一台计算机上或者跨越多个计算机本质上没有什么区别。同步原语在多线程上线文中非常有用，同样在分布式环境中也一样有用。然而，一个重要的区别源于这样一个事实：在典型的非共享架构中，不同的计算器除了网络不会共享任何东西。虽然有很多的消息传递算法能实现同步原语，但是如果能依赖一个提供顺序排列属性的共享存储会容易实现很多，Zookeeper就提供这样的服务。

协同并不总是采用同步原语的形式，例如领导者选举或者锁服务。

Part1:概念和基础

- 概念和基础

概念和基础

任何对**Zookeeper**感兴趣的人都应该阅读本书。它描述了**Zookeeper**要解决的问题，以及它在设计过程中做出了哪些折中。

简介

过去，每个应用都是一个单独的程序，只运行在一台单CPU的计算机上。而今天，情况发生了变化。在这个大数据和云计算的时代，应用由许多独立的程序组成的，它们运行在各种各样的计算机上。

协调这些独立的程序远比写一个只运行在单个计算机上的程序困难很多。开发者很容易就陷入了协调的逻辑之中，而缺少时间去正确的编写应用的本身的逻辑。或者反过来说，开发者花了很短的时间来处理协同部分，简单的写了一个应急的主协调器，但是它是非常脆弱的，变成了一个不可靠的失败点。

Zookeeper被设计成鲁棒性的服务，允许应用开发者聚焦在他们的应用逻辑上而不是协同服务。受文件系统API的启发，Zookeeper提供了一个简单的API，允许开发者实现通用的协同任务，例如选举主服务器、管理组成员关系和管理元数据。Zookeeper提供应用层的库函数，有两个主要的实现-Java和C。同时，Zookeeper是一个Java实现的服务组件，运行在一个独立的服务集群上。由专门的服务集群来运行Zookeeper使得它有更强的容错性和扩展性。

设计一个有Zookeeper服务的应用程序时，理想的做法是分离应用数据和协同数据。比如，WEB邮件服务的用户对他们的邮件的内容更感兴趣，而不关心哪一台服务器在处理哪个特定邮箱的请求。邮件的内容是应用数据，然而哪个邮箱映射到哪个特定的邮件服务器是部分的协同数据（或者称为元数据）。Zookeeper集群是处理后者的。

Zookeeper的使命

- Apache HBase
- Apache Kafka
- Apache Solr
- Yahoo!Fetching Service
- Facebook Messages
- Zookeeper名字的由来

Zookeeper对我们的意义就像解释螺丝刀对我们的意义一样。简而言之，螺丝刀允许我们拧螺丝，但是仅仅这样说不能表达出这个工具的能力。螺丝刀允许我们组装家具和电子设备，例如在墙上挂一副画。通过给出更多的这样的例子，我们对螺丝刀能做的事情有个大致的感觉，但是这样去描述也不够完整。

Zookeeper这样的系统能为我们做什么可以归结于这一点：它在分布式系统中能提供协同任务。协同任务一个有着多个进程参与的任务。比如，一个任务以协作或者协调为目的。协作意味着参与的进程需要一起做一些事情，一些进程需要执行某种动作让其他的进程继续运行。例如，在典型的master-worker架构中，当worker可用时它会通知master，Master随后会指派任务给worker。当遇到这种情况是竞争是不一样的，比如两个进程不能并发的运行，一个进程必须等待另外一个进程。还是那个master-worker的例子，我们只想要一个master，但是多个进程可能都会尝试变成master。多进程然后需要实现互斥操作。我们实际上希望获取领导权的任务像获取锁一样：那个获取了领导权锁的进程会变成master的角色。

如果你有过多线程编程的经验，你会意识到它们之间有很多的相似的问题。实际上，有着多个进程运行在同一台计算机上或者跨越多个计算机本质上没有什么区别。同步原语在多线程上线文中非常有用，同样在分布式环境中也一样有用。然而，一个重要的区别源于这样一个事实：在典型的非共享架构中，不同的计算器除了网络不会共享任何东西。虽然有很多的消息传递算法能实现同步原语，但是如果能依赖一个提供顺序排列属性的共享存储会容易实现很多，Zookeeper就提供这样的服务。

协同并不总是采用同步原语的形式，例如领导者选举或者锁服务。一个进程告诉其他进程该做经常采用 配置元数据的方式进行。例如，在master-worker系统中，workers需要知道那些指派给他们的任务，而这些信息即使在master奔溃的情况下也要可用。

我们来看一些例子，来对Zookeeper的使用有一个更加具象的了解。

Apache HBase

HBase是一个经常和Hadoop一起使用的数据存储系统。在HBase中，Zookeeper被用作选举一个集群的领导者，追踪可用的服务器，维护集群的元数据。

Apache Kafka

Kafka是一个发布-订阅模式的消息系统。它用Zookeeper检测奔溃，实现主题发现，维护主题的生成者和消费者的状态。

Apache Solr

Solr是一个企业级的搜索平台。在分布式的形态中，它被成为SolrCloud，它用Zookeeper来存储关于集群的元数据，以及协调这些袁术的更新。

Yahoo!Fetching Service

Fetching Service实现了爬虫的部分工作，它能通过缓存内容数据有效的抓取WEB页面，同时又能确保WEB服务器的抓取策略能得到保留，比如那些robotx.txt文件。该服务使用Zookeeper来处理领导者选举、奔溃检查、元数据存储之类的任务。

Facebook Messages

这是一个Facebook应用，它整合了沟通的各个渠道：emails,SMS,Facebook Chat和现存的Facebook Inbox。它使用Zookeeper作为实现分片、故障转移和服务发现的控制器。

除了上面的例子，外面还有大量的实用案例。通过这个示例，让我们现在进行一些更加抽象的讨论。当使用Zookeeper进行编程时，开发者设计他们的应用程序作为连接到Zookeeper服务器的客户端，通过Zookeeper客户端的API进行某些操作。在Zookeeper的这些能力中，它主要提供一下几个能力：

- 强一致性、排序和持久化的保证
- 实现同步原语的能力
- 提供一种简单的方式来处理并发的各方面问题，这些问题在现实的分布式环境中经常导致不正确的行为

然而，Zookeeper并没有魔法，他不能开箱即用的解决所有的问题。所以理解Zookeeper提供的能力并了解它棘手的某些方面至关重要。本书的一个目标就是讨论处理这些问题的方式。我们涵盖了那些需要读者知道关于Zookeeper能为开发者做什么的基础知识。另外，我们还讨论了有Zookeeper的应用程序在实现过程中会出现的若干问题，帮助那些刚刚接触Zookeeper的开发者们。

Zookeeper名字的由来

Zookeeper室友雅虎研究院开发出来的。我们在Zookeeper上工作了一段时间，并把它推销给其他的团队，所以我们需要一个名字。那时候，团队和Hadoop的团队一起工作，并开发了各种各样的以动物名字为代号的项目，Apache Pig（猪）就是其中最有名的一个。当我们讨论不同动物的名字时，我们的经理想到好像听上去我们生活中一个动物园中。就是这样敲定的，分布式系统就是一个动物园。它们嘈杂并且难以管理，Zookeeper意味着让它们在管控之中。

本书采用猫来做封面也是合适的，因为早期的一片来自雅虎研究院的关于Zookeeper的文章把分布式进程管理描述的像管理一群猫。Zookeeper听上去要比一群猫要好点。

没有Zookeeper世界会怎样

拥有Zookeeper使得一整个新类别的应用得以被开发？似乎并不是这样。Zookeeper简化了开发的进程，是的开发更加的敏捷，同时又能提供更加鲁棒性的实现。

以前系统也实现过像分布式锁管理者这样的组件或者使用分布式数据库用户协同服务。实际上，Zookeeper借用了很多来自以前系统的概念。然而，它不会暴露出一个锁服务的接口或者一个存储数据的通用接口。Zookeeper被设计成是专有的并且非常关注与协同任务。同时，它不会试着强加一组特定同步原语给开发者，这样使得能够实现的东西更加的灵活。

当然，没有Zookeeper也能构建分布式系统。然而，Zookeeper提供给开发者专注于应用逻辑而不是难懂的分布式系统概念的机会。没有Zookeeper进行分布式系统编程是可行的，但是会更加困难。

哪些事情Zookeeper不做

Zookeeper服务器集群管理着与协同相关的至关重要的应用数据。Zookeeper不是用于大量数据存储的。对于存储大量的数据，有着非常多的选择，比如数据库和分布式文件系统。当设计一个有Zookeeper的应用程序时，理想的做法是从控制器中分离出应用数据或者协同数据。它们经常有着不同的需求，比如在一致性和持久性方面。

Zookeeper实现了一个核心的操作集合，使得对于许多分布式应用非常常见的任务得以实现。你知道多少个应用是需要一个master，或者需要追踪哪个进程是可相应的？然而，Zookeeper不会为你实现那些任务。它不会选举一个领导者，也不会追踪那些活着的进程。相反，它提供了许多工具来实现这些任务。开发者自己觉得他们要实现的是哪种协同任务。

Apache项目

Zookeeper是一个开源项目，托管在Apache软件基金会下面。它有一个项目管理委员会(PMC)来专门负责项目的管理和监督工作。只有Committer能检查patch，但是任务开发者都可以贡献一个patch。开发者只有在对项目作出贡献之后才能成为Committer。对项目作出贡献不局限于提交patch，它们可以有其他形式已经和社区的其他成员进行互动。我们在邮件列表上有着大量的讨论，关于新特性、来自用户的问题等等。我们十分鼓励开发者参与的社区中来，订阅邮件列表，参与讨论。如果你想通过某些项目和Zookeeper保持一个长期的关系，你会发现成为一个committer是非常值得的。

利用Zookeeper构建分布式系统

- 消息延迟
- 处理器速度
- 时钟漂移

对于分布式系统，现阶段有着许多的定义，对于本书而言，我们把它定义为有多个软件组件构成的系统，它们独立的并发的行的，并且跨越多个物理机。有很多理由以分布式的方式去设计一个系统。一个分布式系统有能力通过运行多个组件来发掘多个处理器的能力，也许是并发复制。因为某种策略的原因，一个系统可能是分布在不同的地方，比如在多个地方的服务器参与到单个应用中。

分离协同组件有个很多重要的优势。第一，它允许组件被独立的设计和实现。比如一个独立的组件能多个应用共享。第二，它使得一个系统架构师在协同部分的论证更加简单，当然这不是重点（至少在本书不是）。最后，它能使得一个系统能够将允许和管理协同组件分开。分开运行一个这样的组件能够简化产品中解决问题的任务。

软件组件运行在操作系统的进程中，大部分情况下是以多线程执行的。因此，Zookeeper服务器和客户端都是经常。通常，单个物理服务器（无论是单个机器还是在虚拟环境中的操作系统）运行着单个应用进程，虽然进程可能执行多线程来发掘现代处理器的多核能力。

分布式系统中进程有两大类通信方式：他们能直接通过网络交换消息，或者读写某个共享存储。Zookeeper使用共享存储的方式让应用实现协同和同步原语。但是共享存储自身需要进程间的网络通信和存储。强调网络通信是非常重要的，因为它是设计分布式系统诸多问题的其中一个源头。

在现实的系统中，需要十分注意一下几个问题：

消息延迟

消息随时都会发生延迟；比如，由于网络拥塞的原因。这样随时都会发生延迟的情况会导致不可预期的结果。比如，根据时钟来说，进程P可能在另外一进程Q发送消息请发送一条消息，但是进程Q的消息可能会先传递。

处理器速度

操作系统调度和超负荷运行可能导致消息处理随时会出现延迟。当一个进程发送一条消息给另外一个时，这条消息的总体延迟粗略的等于发送者处理时间加上传输的时间，再加上接受者处理的时间。如果发送和接受者需要被调度来进行处理，那么消息的延迟会更高。

时钟漂移

系统中使用时间的概念并不少见，比如决定什么时间在系统中发生什么事件。处理时时钟是不可靠的，和其他的处理器相比它们能随意的发生漂移。自然而然的，依赖于处理器时钟可能导致不正确的

决定。

这些问题中一个重要推论的就是非常难判断一个进程实际上是否已经奔溃还是其中任意一个因素引入了延迟。没有从一个进程中接受到信息可能因为这它已经奔溃了，可能是网络任意的延迟了最后一条消息，可能是其他的什么事情引起了延迟，或者处理时钟发什么漂移。这样无法区分原因的系统被称之为异步的。

数据中心通常由大批量的且大部分是统一的硬件组成。但是即使在数据中心，我们还是观察到这样问题对应用的影响，是因为在同一个应用中使用 多代硬件，还有即使在同一批硬件中也有轻微区别，它们对性能造成了很大影响。所有的这些事情使得分布式系统设计者的生活更加复杂了。

Zookeeper被精巧的设计成能更加简单的处理这些问题。Zookeeper不会使得问题消失或者把它们完全对应用屏蔽，但是它能使得问题变得更加的易于处理。Zookeeper实现了对分布式计算问题的解决方案，并把这些实现以一种更加直观的方式打包起来开放给开发者，至少，这是我们一直希望的。

示例：Master-Worker应用

- Master奔溃
- Worker奔溃
- Communication失败

我们已经抽象的讨论过了分布式系统，现在是时候给出让它更加具体了。让我们考虑一个在分布式系统中十分常用的架构：master-work架构。这种架构的一个重要例子就是HBase，一个Google的Bigtable的克隆版。从高层来看，master服务器(HMaster)负责追踪区域服务器(HRegionServer)的可用性并且指派区域给服务器。由于我们这里无法完整的介绍它，我们鼓励你查看HBase的文档以进一步阅读它是如何使用Zookeeper的细节的。我们的讨论只关注一个通用的master-worker架构。

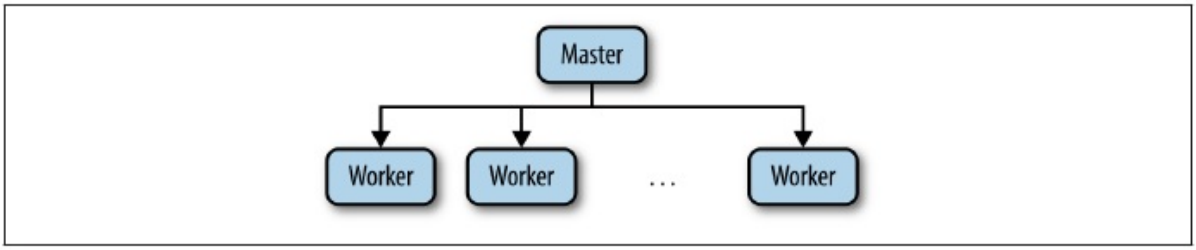


Figure 1-1. Master-worker example

通常，这种架构master进程负责追踪workers和任务的可用性，还要指派任务给workers。对Zookeeper来说，这种架构风格是很有代表性的，因为它阐述了很多常用的任务，比如选举master，追踪可用的worker，维护应用数据。

为了实现一个master-worker系统，我们必须解决下面三个关键的问题：

Master奔溃

如果Master发生错误变得不可用，系统不能分配新的任务或者重新分配那些来自worker已经失败的任务。

Worker奔溃

如果Worker奔溃，那些已经指定给它的任务就不会完成。

Communication失败

如果master和worker不能交换信息，那么worker可能就不能得到新的分配给它的任务。

为了处理这些问题，系统必须能可靠的选举出新的master如果前面的那个发生错误，决定哪一个

worker是可用的，以及决定何时一个worker相对去系统的其他部分变得陈旧。我们通过下面几段来看一下各个工作。

Master失败

为了应对master奔溃，我们需要有备用的master。当主master奔溃时，备用的master能接管主master的工作。然而，当涉及到master请求时， 故障转移就没有那么简单了。新的主master必须恢复当老的master奔溃时系统的状态。为了恢复master的状态，我们不能依赖拉取来自错误的master的状态，因为它已经奔溃了；我们必须从其他的地方来获取，这个地方就是Zookeeper。

恢复状态不是唯一重要的问题。假设这样一个情景：主master正常运行，但是备份master怀疑主master奔溃了。这种错误的怀疑是可能发生的，比如主master的负载非常高导致它的消息被延迟了。备份master会执行所有必要的步骤去接管主master的角色，最终可能会导致它开始执行主master的角色，变成了第二个主master。更糟糕的是，如果某些worker由于网络分区的原因无法和主master进行通信，它们最终可能会追随第二个主master。这种场景导致了一个被称之为“脑裂”(split-brain)的问题:系统内两个或者多个部分独立的发展，导致了行为的不一致性。作为接下来应对master失败的部分方案，避免“脑裂“场景的发生是至关重要的。

Worker失败

客户端提交任务给master，由master分配任务给可用的workers。这些worker接收被分配的任务，一旦这些任务被执行它们就会上报状态。master随后就会通知客户端执行的结果。

如果一个worker奔溃，所有分配给它的未完成的任务必须被重新分配。第一个要求就是给master能够检测worker奔溃的能力。master必须能够检测到何时一个worker奔溃了，同时也能确定哪些worker能够执行它的任务。当一个worker奔溃时，它可能最终部分的执行了任务，或者完全执行完了任务但是还未来得及上报结果。如果计算还有副作用的话，为了清理状态某些恢复步骤可能就很有必要了。

通信失败

- 仅有一次（Exactly-Once)和最多一次(At-Most-Once)的语义

如果一个worker和master失去连接，比如网络分区的原因，重新分配任务可能导致两个worker执行同一个任务。如果一个任务被执行多次是可以接受的，我们在重新分配任务时就可以不用校验第一个worker是否执行过任务。如果不能接受，那么应用必须能适应多个worker最终都可能执行该任务的可能。

> ### 仅有一次（*Exactly-Once)*和最多一次*(At-Most-Once)*的语义
>
> 对任务（比如领导者选举）使用锁并不能完全避免任务被执行多次，因为可能会存在以下几个连续的事件：
>
> 1. *Master M1*指定任务*T1*给*Worker W1*。
> 2. *W1*获取*T1*的锁，然后执行*T1*，接着释放锁。
> 3. *Master M1*怀疑*W1*奔溃了，然后重新分配任务*T1*给*Worker W2*。
> 4. *W2*获取了*T1*的锁，执行*T1*，接着释放锁。
>
> 这种情况下，*T1*的锁没有阻止任务被执行两次，因为两个*worker*之间在执行任务时没有重叠的步骤。为了处理那些需要仅有一次或者最多一次语义的场景，应用需要依赖特定的机制。比如说，应用数据包含时间戳，而一个任务需要修改这个应用数据，那么任务的成功执行取决于那个数据创建的时间戳的值。该应用在状态不是自动修改的情况下需要有能回滚部分变更的能力；否则，它可能最终导致一个不一致的状态。
>
> 我们通过这些讨论来阐述了为应用实现这些语义的难点。讨论这些语言的具体实现不在本书的范围内。

另外一个通信失败的重要问题是它们对同步原语（比如锁）产生的影响。因为节点可能奔溃，系统可能会有网络分区，锁会产生问题：如果一个节点奔溃或者被分区，锁可以阻止其他的节点继续运行。Zookeeper随后需要实现处理这种场景的机制。首先，它使得客户端指定在Zookeeper中的某些数据状态是临时的。其次，Zookeeper集群需要客户端周期性通知它们是存活的。如果一个客户端没有及时地通知集群，那么所有属于这个客户端的临时状态都会被删除。通过这两个机制，在奔溃和通信失败的情况下，我们能防止单独的客户端导致应用停止执行。

回想前面我们讨论过，在系统中我们无法控制消息的延迟，所以无法分辨一个客户端是否奔溃还是只是运行很慢。因此，当我们怀疑一个客户端已经奔溃时，我们需要假设它仅仅是运行地很慢，这样在未来它还有可能执行某些其他的动作。

任务总结

- 领导者选举
- 奔溃检测
- 组成员关系管理
- 元数据管理

在前面的描述中，我们可以对master-worker架构提取出下面几个要求：

领导者选举

有一个可用的master并且能分配任务给worker是至关重要的。

奔溃检测

master必须能够检查出worker奔溃或者失去联系。

组成员关系管理

master必须知道哪些worker是可用来执行任务的。

元数据管理

master和workers必须能够以一种可靠的方式来存储指派的任务信息和执行状态。

理想的情况下，每个暴露给应用的任务都必须采用原语的形式，完全对应用开发者屏蔽实现的细节。Zookeeper提供关键的机制去实现这种原语，使得开发者能够实现最满足它们需求的原语从而关注他们的应用逻辑。在本书中，我们经常提及像领导者选举或者奔溃检测这样任务的实现作为原语，是因为它们是分布式应用得以构建的基石。

为什么分布式协作如此困难

编写分布式应用会如此复杂，其中的一些原因是显而易见的。比如说，当我们的应用启动后，所有不同的进程需要找到应用程序的配置信息。随着时间的变化，配置信息也会发生变化。我们可以关闭所有的检测，重新分发配置文件并且重启，但是在配置过程可能会应用程序宕机的时间延长。

与配置相关的问题可以归结于组成员关系的问题。当负载发生变化时，我们希望能够添加或者移除新的机器和进程。

这个问题可以描述成功能性的问题，你可以为你的分布式应用设计解决方案；你也可以在不是前测试你的解决方案从而能够确定你正确的解决这个问题。当你在开发分布式应用时，你会遇到的真正的困难之处在于处理故障，特别是奔溃和通信故障。这些故障随时都可能发生，所以不太可能枚举所有需要处理的不同情况。

运行在单机上的应用和分布式应用的一个很大区别在于故障：分布式应用中，可能部分机器会发生故障。当一台机器奔溃时，运行在这台机器上的所有进程都会失败。

Zookeeper是成功的，但是有警告的

完美的解决方案是不可能的，我们再次重申Zookeeper不会解决分布式应用开发者将面临的所有的问题。然而，它确实为开发者提供了一个处理这些问题不错的框架。Zookeeper是建立在多年的分布式计算的研究工作之上的。Paxos和Virtual Synchrony对Zookeeper的设计有很大的影响的。当出现变化和状况时，它能无缝应对，同时提供给开发者一个框架用于应对那些无法被自动处理的状况。

Zookeeper最初在Yahoo被开发出来，被应用于大量的大型分布式应用中。我们注意到某些应用的分布式协同方面没有合适地处理，以至于系统部署后会单点失败或者十分的脆弱。另一方面，其他的开发者在分布式协同上花费了大量的时间以至于他们没有足够的资源来聚焦在应用的功能上。我们发现这些应用都有一些通用的基础的协同需求，所以我们打算设计一个通用的解决方案，它包含一些关键的要素使得我们可以只实现一次但是可以被很多不同的应用所使用。事实证明，Zookeeper比我们当初所想的还要更加通用和流行。

这些年来我们发现人们能简单地部署一个Zookeeper集群并使用它开发应用。但是真的如此简单吗？事实上，某些开发者并没有完全的理解在一些场景中需要开发者自己作出决定而不是Zookeeper。写作本书的其中一个目的就是让开发者理解他们需要做些什么来高效地使用Zookeeper以及为什么要这么做。

初识Zookeeper

上一章在比较高的层次上讨论了分布式应用的需求，它们在协同方面经常有着相同的需求。我们拿master-worker的例子来阐述了少数的我们描述过的通常使用的原语，这个例子是实际应用中比较有代表性的。我们现在正式介绍Zookeeper，它是一种实现协同原语的服务。

Zookeeper基础

几个用于协同的原语经常被跨多个应用所共享。因此，一种设计用于协同的服务就是提供一组原语，通过为每个原语创建实例来暴露调用接口，同时直接操作这些这些实例。比如，我们说分布式锁组成了一个重要的原语，并且提供了创建、获取和释放锁的调用。

然而，这样的设计有几个重要的缺陷。第一，我们要么在使用之前就列出一个原语的详细清单，要么继续扩展API来引入新的原语。第二，这种设计不能为那些使用这个服务的应用提供足够的灵活性以实现最适合它们的原语。

我们因此为Zookeeper选择了一条不一样的路。Zookeeper不直接提供原语。相反的，它提供一个像文件系统的API，由一个小规模的调用组成，这些调用使得应用能够实现他们自己的原语。我们通常使用recipes来表示这些原语的实现。Recipes包括Zookeeper的操作，这些操作能操控小的被称之为znode的数据节点。这些数据节点以树形的结构进行组织，像文件系统一样。Figure 2-1详细的描述了一个znode树。

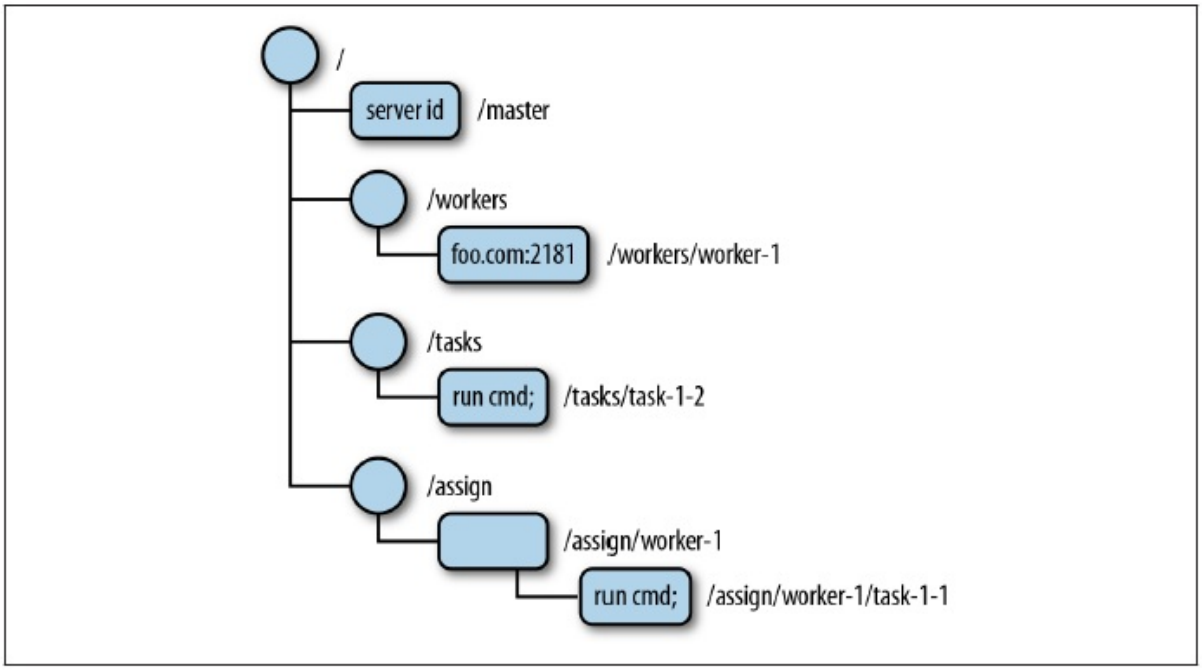


图2-1. Zookeeper数据树示例

数据的确实经常暗含了关于一个znode重要的信息。在master-worker的例子中，举个例子，缺少master节点（以下znode均译为节点）意味着当前没有master被选举出来。图2-1包含了少数几个其他的节点，它们在master-worker的配置中可能很有用。

- \/workers节点是系统中所有可用的worker的父节点。图2-1表明有一个worker(foo.com:2181)可用。如果worker变得不可用，那么它的节点应用从\/workers中删除。

- \/tasks节点是所有被创建和等待被worker执行的任务的父节点。master-worker应用的客户端在\/tasks下新增子节点用来代表新的任务，同时等待代表任务状态的节点。
- \/assigin节点是所有指派关系的父节点。当一个master指派一个任务给一个worker，它会在\/assign下面新增一个子节点。

API总览

Znode可能包含也可能不包含数据。如果一个znode包含数据，那么数据是以字节数组的形式来存储的。具体字节的组织形式因每个应用而异，并且Zookeeper不提高直接解析字节的方法。序列化的包可以用来处理存储在znode中的数据，例如Protocol Buffers, Thrift, Avro和MessagePack，但有时以UTF-8或者ASCII编码的字符串形式就足够了。

Zookeeper API暴露了下面几个操作：

- create \/path data 创建一个以\/path命名的znode节点，同时保护数据data
- delete \/path 删除路径为\/path的znode节点
- exists \/path 判断是否存储\/path路径的znode节点
- setData \/path data 设置路径为\/path的znode的节点数据为data
- getData \/path 获取路径为\/path的znode节点的数据
- getChildren \/path 获取路径为\/path的znode节点的孩子节点

非常重要的一点是Zookeeper不允许写或者读znode上数据。当设置znode上的数据或者读取时，数据内容是被整个替换或者读取的。

Zookeeper的客户端通过它们发出API调用来连接到Zookeeper服务并且建立一个会话。如果你着急的想使用Zookeeper，请直接跳到“会话”，那一节描述了如何通过一个命令行shell来运行zookeeper的命令。

Znode的不同模式

当创建一个新的znode时，你必须指定一个模式。不同的模式决定了znode不同的行为。

持久化的和临时的

一个znode节点要么是持久化的要么是临时的。一个持久化的znode节点只能通过调用*delete*方法来删除。相对的，一个临时的znode节点会在创建它的客户端奔溃或者断掉与zookeeper的连接时自动删除。

当应用需要在znode节点存储一些数据，同时这些数据在它们的创建者不再是系统的一部分时也需要得到保留时，持久化的znode节点就很有用了。比如，在master-worker的例子中，即使分配任务的master奔溃，我们也要维护分配给worker的任务指派关系。

临时zode节点的心智是只有当会话的创建者工作正常时，应用的某些部分必须存在。比如，master-worker例子中的master节点就是临时的。它的存在暗示着现在有一个master，并且它在运行中。如果master没有了而master节点还存在的话，那么系统就探测不到master奔溃了。这样会阻止系统继续运行，所以master节点必须和master一起消失。我们也可以用临时节点代表worker。如果一个worker不可用了，它的会话就会超时，同时它在\/workers下面的节点也会自动消失。

一个临时节点会在以下两种情况下被删除：

- 当客户端创建者的会话终结时，要么超时，要么被显示的关闭了。
- 当一个客户端删除它，不一样要是创建者。

因为临时节点在创建者会话超时时会删除，因此不允许临时节点有子节点。关于临时节点是否可以拥有临时子节点这个问题在社区里面有过很多讨论。这个特性在将来可能会成为现实，但是现在还不可用。

顺序化节点

一个znode节点可以被设置为顺序化的。一个顺序化节点可以被设置为唯一的、单调递增的int值。顺序号被追加在创建znode节点的路径后面。例如，如果一个客户端创建了一个以\/tasks\/task-开通的顺序节点，zookeeper会为它指派一个顺序号，比如说1，并把它追加到路径的后面。那么这个节点的路径就变为了\/tasks\/task-1.顺序节点提供了一种创建拥有独一无二名字节点的方式，同时，也很容易看得出来创建的顺序。

总之，一个znode节点有四种模式：持久化、临时的、持久顺序化和临时顺序化。

监视器和通知

因为zookeeper是以远程服务提供访问的，所以每次客户端想得到一个znode节点的内容时代价是很大的：这将带来很高的延迟同时相比于zookeeper的安装过程它需要更多的操作。考虑如图2-2所示的例子。第二个调用在\/tasks上调用getChildren将返回同样的值，一个空的集会，结果自然的然并卵的。

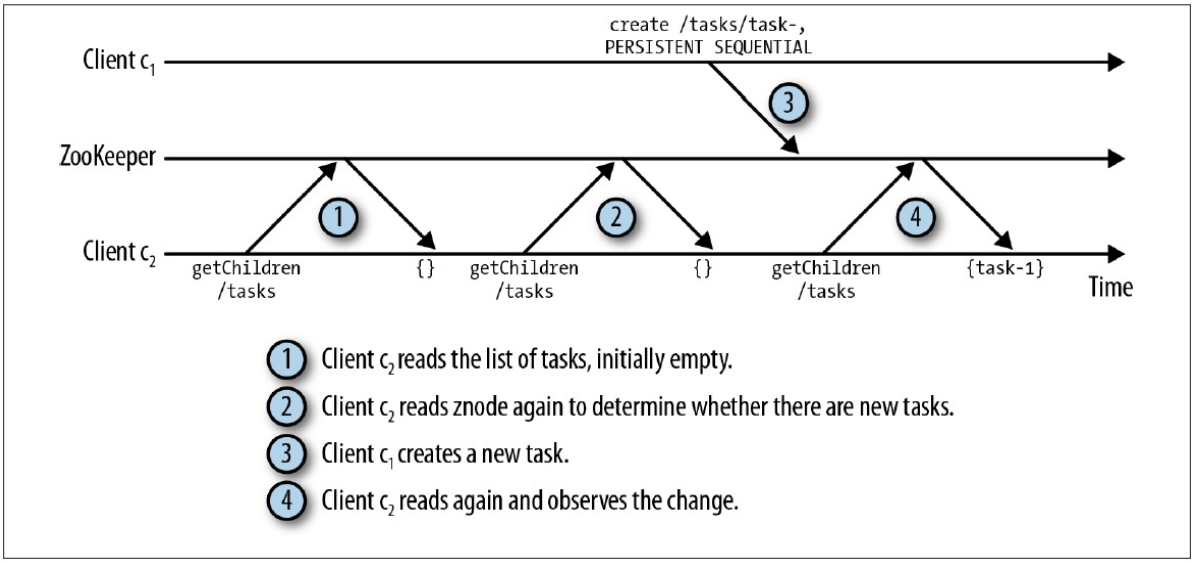


图 2-2 在同一个znode上执行多次读

这是轮询方式的一个常见问题。为了替代客户端轮询，我们采用了一种基于通知的机制：客户端向zookeeper注册它要接收关于znode节点的发生变化的通知。注册接收一个给定的znode上的通知叫做设置监视器。一个监视器是一次性的操作，意味着它只能触发一次通知。为了接收多个通知，客户端必须在接收通知时设置一个新的监视器。如图2-3所描述的那样，客户端只有在接收到表明\/tasks值发生变化的时候才会去读取一个新的值。

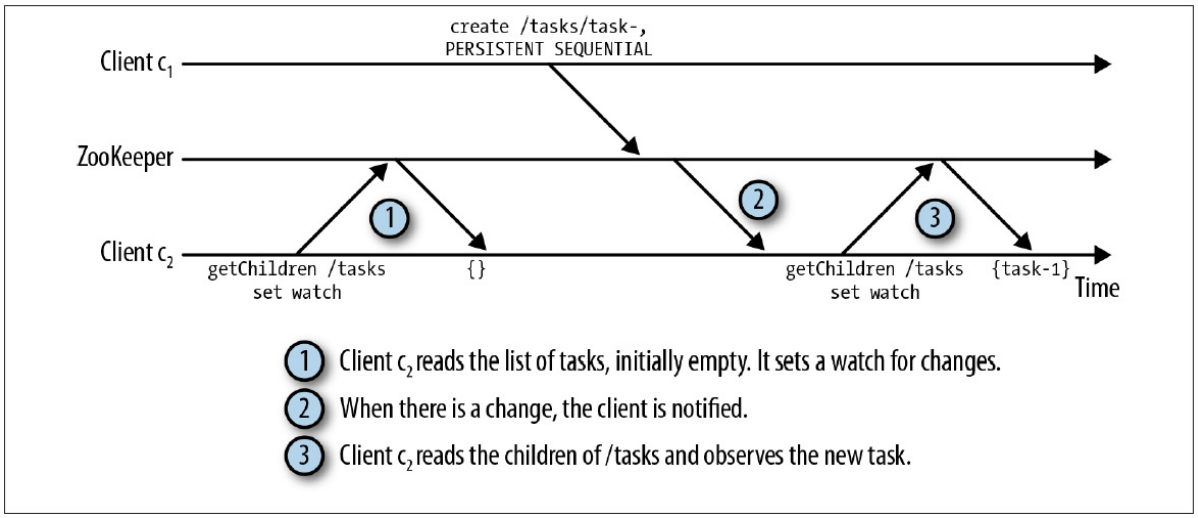


图2-3 利用通知来告知znode的变化

当使用通知时，有些事情需要注意。因为通知是一次性操作，有可能新的变化正好发生在接收通知和设置监视器之间（放心，你不会错过状态的变化）。让我们看一个例子来了解它是如果工作的。假设以下事件按顺序发生：

1. 客户端C1在\/tasks的节点上设置了数据的监视器。
2. 客户端C2在\/tasks下新增了一个任务。
3. 客户端C1接收到了通知。
4. 客户端C1设置一个新的监视器，但是在这之前，第三个客户端C3在\/tasks下新增了一个任务。

客户端C1z最终设置好了这个监视器，但是通知没有因为C3新增任务导致被触发。为了捕捉的这种变化，C1必须去读取\/tasks的状态。它确实确实这样做了，因为当设置监视器时，我们设置的是带有读取zookeeper状态的操作。自然，C1不会错过这种变化。

通知一个重要的保证是在任何发生在znode上的变化生效之前它会传递到客户端。如果一个客户端在一个znode上设置了监视器，同时有两个连续的在这个znode上的更新操作，那么客户端接收到通知发生在第一次更新之后和在它有机会通过读取znode来捕获第二次更新之前。关键点是通知保留了客户端观察到的发生顺序。虽然zookeeper状态变化传播到任意指定的客户端会变慢，但是我们保证客户端观察到zookeeper的状态变化按一个全局的顺序发生的。

Zookeeper提供不同类型的通知，这取决于监视器是如何回应设置的通知的。一个客户端可以设置一个znode数据变化的通知、子节点变化的通知和znode被创建和删除的通知。为了设置一个监视器，我们能使用API里面任意一个带读取zookeeper状态的调用。这些API调用给了一个可选参数*Watcher*对象，或者使用默认的监视器。在本章后面的*master-worker*的例子的讨论中和第四章，我们介绍了详细的使用该机制的细节。

版本

每个znode都有一个与它关联的版本号，每当它的数据发生变化时就增加。API中有一对操作可以被有条件的执行：setData和delete。两个调用都需要版本号作为一个入参，只有当客户端传递的版本号和服务器当前的版本号一致时才会执行成功。当有多个zookeeper的客户端在同一个znode上操作时版本号就很有用了。比如说，客户端C1向\/config节点写入一些配置信息。如果另外一个客户端C2并发的更新这个节点，那么C1的版本号就是明日黄花了，那么C1的setData操作b必须不会成功。使用版本号避免了这种情况。在这种情况下，当写回时，C1使用的版本号不匹配，那么操作就失败了。图2-4详细的描述了这一场景。

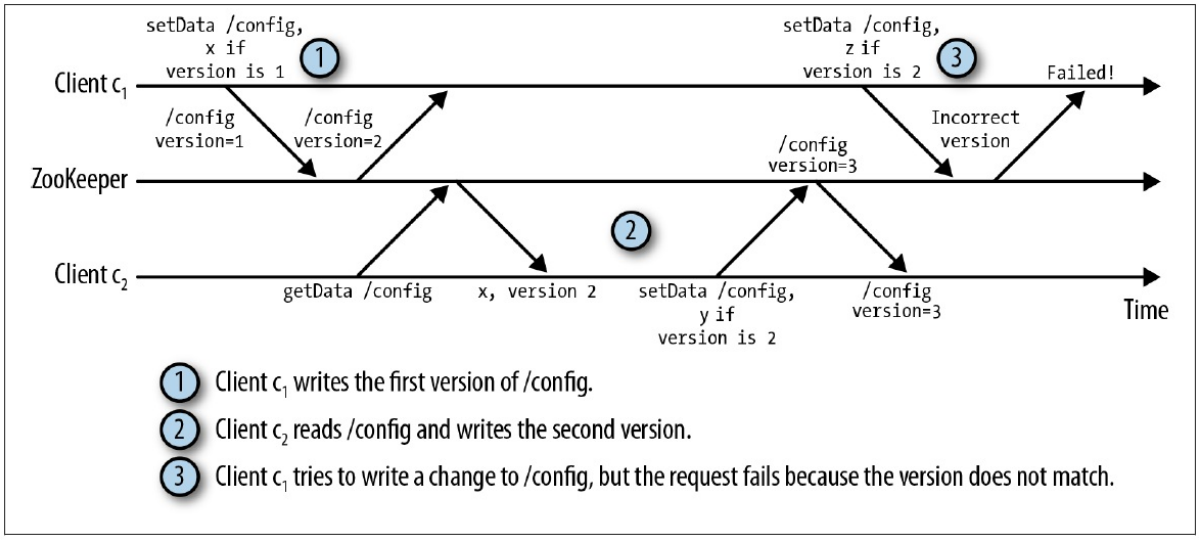


图2-4 使用版本号来防止并发更新引起的不一致性

Zookeeper架构

我们已经在高层次上讨论了zookeeper暴露给应用程序的操作，现在我们需要更多地理解它是如何工作的。应用程序通过客户端的库来调用zookeeper。客户端库复杂与zookeeper服务器之间的交互。

图2-5展示了客户端和服务器的关系。每个客户端都要导入客户端库，然后才能与zookeeper节点通信。

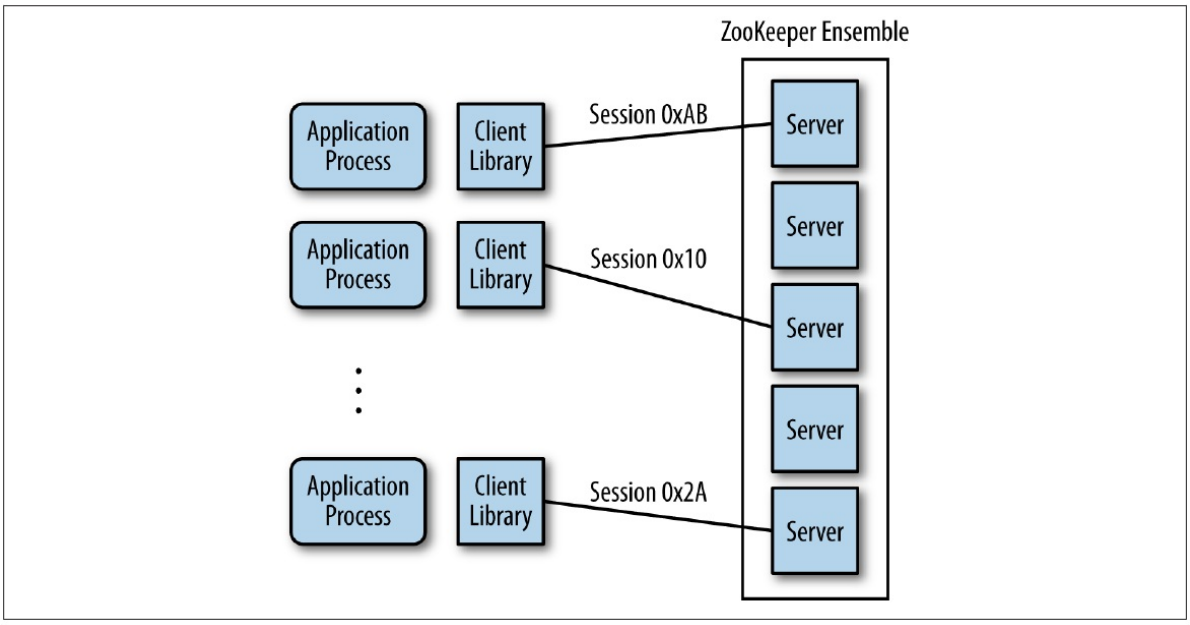


图2-5 Zookeeper架构总览

Zookeeper服务器运行在两种模式下面：单机模式和仲裁模式。单机模式显而易见只有一台服务器，并且zookeeper的状态不是复制的。仲裁模式中，一组zookeeper服务器，我们称之为Zookeeper集合，它们复制状态同时一起服务客户端的请求。从这个角度看，我们用”Zookeeper集合”来代表服务器的安装。这个安装可能包含单台服务器并已独立模式操作，也可能包含一组服务器并以集群模式操作。

Zookeeper仲裁

仲裁模式下，Zookeeper在集合中所有服务器上复制它的数据树。但是如果一个客户端不得不等每个服务器存储完数据才能继续运行的话，那延迟是不可接受的。在公共管理中，法定人数要求最少数量的立法者出席进行投票。而在zookeeper中，为了让其能工作，法定人数是需要最少数量正常运行的服务器。告诉客户端数据已经被安全存储之前，Zookeeper需要最少的服务器需存储客户端的数据。比如，我们一共有五台Zookeeper服务器，但是法定人数就是三台。只要任意三台服务器存储了数据，客户端就可以继续运行，另外两台服务器最终会追赶上来并存储数据。

为仲裁模式选定足够的服务器是很重要的。无论系统延迟和奔溃，仲裁者必须保证任何的更新请求Zookeeper都会积极的响应并保存，直到另外一个请求取代它。

为了理解其中的意义，让我们来通过一个例子来说明如果仲裁者过少会引起什么样的问题。比如说我们有五台服务器，仲裁者可以是其中任意的两台。现在服务器s1和s2说它们已经复制了创建\/z节点的请求。然后Zookeeper告诉客户端节点已经创建好了。现在假设这两台服务器在它们有机会复制新增的节点到其它服务器之前，因为网络分区的原因与其它服务器和客户端隔离任意长的时间。在这种状态下，Zookeeper服务能继续运行因为还有其它三台服务器可用，实际上根据我们的假设只需要两台服务器就可以了，但是这三台服务器都不知道创建了\/z的节点。自然，创建\/z节点的请求没有持久化。

这个例子就是在第一章提到过的脑裂场景之一。为了避免这个问题，这个例子中仲裁者的数量不得低于三个，这是五台服务器的大多数。为了继续运行，整个Zookeeper集合必须保证三台服务器可用。为了确认更新请求已经成功的完成了，Zookeeper集合需要至少三台服务器承认它们已经复制了这个状态。当然，如果Zookeeper集合想要继续运行，那么每个更新操作都要成功的完成，我们至少需要一台包含了更新操作副本的服务器可用（也就是说，仲裁者至少要有一台）。

采用这样一种架构，我们能容忍f台服务器的宕机，f小于总服务器数的一半。比如说，我们有5台服务器，那么最多能忍受2台服务器宕机。服务器的数量不能强制设为奇数的话，偶数会使得系统更加脆弱。举例来说我们有4台服务器，那么大多数就意味着3台服务器。然而，系统只能容忍1台服务器宕机，因为2台宕机使得系统就没有了大多数服务器。结果就是，4台服务器只允许一台服务器宕机，但是仲裁者相比之下就更多了，这意味着我们要得到更多确认来处理每个请求。底线是我们应该总数设置奇数台服务器。

我们可以设置仲裁者数量不同于大多数，当时这将会在更高级的章节进行讨论。我们会在第十章讨论。

会话

在Zookeeper集合处理任何请求之前，客户端必须先建立和服务器的连接。会话的概念相当重要，同时对于Zookeeper的操作也是至关重要的。客户端提交给Zookeeper的所有操作都与一个会话相关联。当一个会话因为任何原因结束时，那个会话期间建立的临时节点也会随之消失。

当一个客户端使用某种语言创建了Zookeeper的处理器时，它就建立和服务的会话。客户端起初只会连接集合中的一台服务器，也只会连接一台。它使用TCP连接与服务器通信，但是当它不再收到当前服务器的信息一段时间，它就会转移到其它服务器。转移会话连接到另外一台服务器是Zookeeper客户端库透明的处理的。

会话提高了顺序性的保障，这意味着在一个会话中请求的执行是按FIFO的顺序执行的。典型情况下，一个客户端只打开一个会话，那么所有的请求都是按FIFO的顺序执行的。如果客户端有多个并发会话，跨会话的FIFO顺序不一定会得到保障。来自同一个客户端连续的会话，即使它们时间上不重叠，也不一定会保障FIFO的顺序。在下面这个例子演示了它是如何发生的:

- 客户端建立一个会话，并发出两个连续的异步调用去创建\/tasks和\/workers
- 第一个会话过期
- 客户端建立另外一个会话，并发出一个异步调用去创建\/assigin

按这个顺序进行调用，有可能只会创建\/tasks和\/assigin，这样保证了第一个会话的FIFO顺序，但是跨越多个会话就得不到保证了。

开始使用Zookeeper

开始之前，你要先下载Zookeeper的发行版。Zookeeper是一个Apache项目，托管在http:\/\/zookeeper.apache.org上。如果你打开这个链接，你最终会得到一个文件名类似于zookeeper-3.4.5.tar.gz的压缩文件。在Linux、Mac OS X 或者其他类UNIX系统，你可以使用一下命令去解压这个发行版：

```
#tar -xvzf zookeeper-3.4.5.tar.gz
```

如果你使用Windows系统，你需要使用如WinZip这样的解压工具来解压。

你需要安装Java，运行Zookeeper需要Java6.

在发行版的目录的中你会找到一个*bin*目录，它包含了启动Zookeeper所需的脚本。这些脚本以.sh结尾，用以在UNIX平台(Linux,Mac OS X等等)上运行，还有以.cmd结尾的脚本是为Windows系统准备的。*conf*目录有配置文件。*lib*目录包含Java Jar包，它们是运行Zookeeper所需的第三方包。后面我们会涉及到你解压发行版的目录。我们把这个目录称之为{PATH_TO_ZK}。

第一个Zookeeper会话

让我们在本地以单机模式建立一个Zookeeper并创建一个会话。要达到此目的，我们使用伴随Zookeeper发行版在*bin*目录下的zkServer和zkCli工具。有经验的管理员会使用它们调试和管理，但现在它也非常适合初学者熟悉Zookeeper。

假设你下载并解压了一个Zookeeper的发行版，打开一个shell，切换目录(cd)到工程的根目录，重命名示例配置文件：

```
#mv conf/zoo_sample.cfg conf/zoo.cfg
```

虽然是可选的，但最好把data目录从\/tmp移出避免zookeeper填满你的根分区。你在zoo.cfg中可以改变它的位置：

```
dataDir=/users/me/zookeeper
```

最后启动服务器，执行以下命令：

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

```
JMX enabled by default
```

```
Using config: ../conf/zoo.cfg
```

```
Starting zookeeper ... STARTED
```

```
#
```

这条服务器命令使得Zookeeper服务器在后台运行起来。如果要在前台运行使得我们能看到服务器的输出，你可以运行下面的命令：

```
#bin/zkServer.sh start-foreground
```

这个选项让你有更加详细的输出，允许你看到服务器正在发生什么。

我们现在准备启动一个客户端。在工程的根目录下另起一个不同的shell，运行下面的命令：

```
#bin/zkCli.sh
```

```
.
```

```
.
```

```
.
```

```
<some omitted output>
```

```
.
```

```
.
```

```
.
2012-12-06 12:07:23,545 [myid:] - INFO [main:ZooKeeper@438] - (1)
Initiating client connection, connectString=localhost:2181
sessionTimeout=30000 watcher=org.apache.zookeeper.
ZooKeeperMain$MyWatcher@2c641e9a
Welcome to ZooKeeper!
2012-12-06 12:07:23,702 [myid:] - INFO [main-SendThread - (2)
(localhost:2181):ClientCnxn$SendThread@966] - Opening
socket connection to server localhost/127.0.0.1:2181.
Will not attempt to authenticate using SASL (Unable to
locate a login configuration)
JLine support is enabled
2012-12-06 12:07:23,717 [myid:] - INFO [main-SendThread - (3)
(localhost:2181):ClientCnxn$SendThread@849] - Socket
connection established to localhost/127.0.0.1:2181, initiating
session [zk: localhost:2181(CONNECTING) 0]
2012-12-06 12:07:23,987 [myid:] - INFO [main-SendThread - (4)
(localhost:2181):ClientCnxn$SendThread@1207] - Session
establishment complete on server localhost/127.0.0.1:2181,
sessionid = 0x13b6fe376cd0000, negotiated timeout = 30000
WATCHER::
WatchedEvent state:SyncConnected type:None path:null - (5)
```

1. 客户端启动建立会话的过程。
2. 客户端尝试连接localhost\/127.0.0.1:2181
3. 客户端连接成功，服务器开始初始化一个新的连接
4. 会话成功地初始化完成
5. 服务器发生SyncConnected事件给客户端

我们来看一下输出。有很多行告诉我们不同的环境变量是如何设置的以及客户端使用的什么Jar包。在这个例子中我们忽略他们，关注在会话的建立上，多花点时间分析你屏幕上的输出信息。

在输出的最后，我们看到了一些涉及到会话建立的日志。第一个是说“初始化连接”。这个消息表述了正在发生的事情，但是一个重要的附加信息是说它正在尝试连接到一台服务器，连接地址是客户端发

出的localhost\/127.0.0.1:2181。这个例子中，连接串只包含了localhost，所以这就是将要建立的一个连接。接下来我们看到一条关于SASL的消息，我们先忽略这条消息。紧接着是一条关于客户端与本地Zookeeper服务器建立TCP连接的消息。最后一条日志消息确认了连接已经建立，同时告诉我们连接的ID：0x13b6fe376cd0000。最后，客户端库以一个SyncConnected事件通知应用程序。应用应该事先Watcher对象来处理改事件。我们在下一节再讲更多的事件。

刚刚对Zookeeper有了更多的熟悉，让我们列出根路径下面的znode节点并创建一个znode。让我们首先确认数据树此时是空的，除了\/zookeeper节点，该节点标记了Zookeeper服务维护的元数据树：

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null

[zk: localhost:2181(CONNECTED) 0] ls /

[zookeeper]
```

这里发生了什么？我们执行了ls \/发现只有\/zookeeper。现在我们创建一个叫\/workers的节点并确认真的存在：

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null

[zk: localhost:2181(CONNECTED) 0]

[zk: localhost:2181(CONNECTED) 0] ls /

[zookeeper]

[zk: localhost:2181(CONNECTED) 1] create /workers ""

Created /workers

[zk: localhost:2181(CONNECTED) 2] ls /

[workers, zookeeper]

[zk: localhost:2181(CONNECTED) 3]
```

为了完成这个联系，我们删除znode并退出：

```
[zk: localhost:2181(CONNECTED) 3] delete /workers

[zk: localhost:2181(CONNECTED) 4] ls /

[zookeeper]

[zk: localhost:2181(CONNECTED) 5] quit

Quitting...

2012-12-06 12:28:18,200 [myid:] - INFO [main-EventThread:ClientCnxn$
```

```
EventThread@509] - EventThread shut down
```

```
2012-12-06 12:28:18,200 [myid:] - INFO [main:ZooKeeper@684] - Session:
```

```
0x13b6fe376cd0000 closed
```

观察到\/workers节点已经被删除了，同时会话已经被关闭了。为了清理干净，让我们关闭Zookeeper服务器：

```
# bin/zkServer.sh stop
```

```
JMX enabled by default
```

```
Using config: ../conf/zoo.cfg
```

```
Stopping zookeeper ... STOPPED
```

```
#
```

会话的生命周期和状态

一个会话的生命周期是说从它创建到结束，不论它是优雅的关闭还是因为超时。要知道一个会话内发生了什么，我们先要了解会话的可能状态已经它改变状态时可能有的事件。

大部分会话的状态字如其名：连接中（CONNECTING）、已连接（CONNECTED）、已关闭（CLOSED）和未连接（NOT_CONNECTED）。状态转化依赖于客户端和服务器间发生的不同事件。

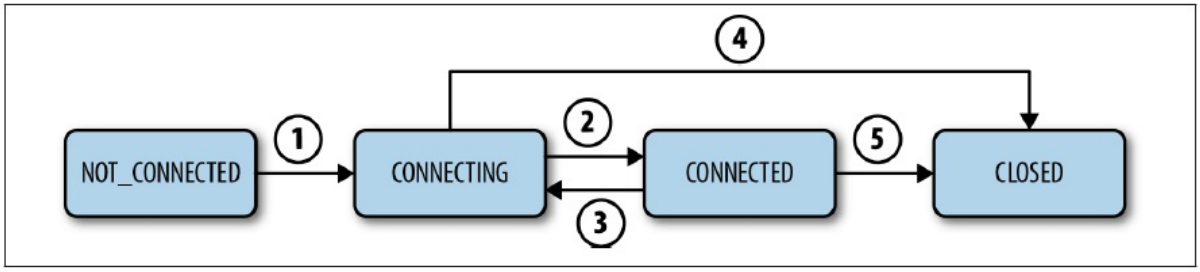


图2-6 会话状态和转化

一个会话始于未连接状态，然后转化到连接中（图2-6箭头1所示），同时初始化Zookeeper客户端。正常情况下，连接到一台Zookeeper服务器成功完成并且会话状态转化为已连接状态（箭头2）。当客户端丢失与Zookeeper服务器的连接或者不在接收到来自服务器的信息时，它的状态又转化回连接中（箭头3），然后尝试去找到另外一台Zookeeper服务器。如果它能找到另外一台服务器或者能重新与原先的服务器连接，一旦服务器确认了会话是有效的，它的状态又变为了已连接的状态。否则，会话就会过期，状态转化为已关闭（箭头4）的状态。应用能显示的关闭会话（箭头4和5）。

有一个重要的参数你要设置的是当创建会话时的超时时间，它是Zookeeper服务允许一个会话宣称它过期的时间。如果服务在时间t内没有收到指定会话相关的消息，它就会宣称会话过期了。客户端那边，如果它在时间1\/3 t内没有收到来自服务器的消息，它会发送一个心跳消息给服务器。在2\/3 t时，客户端开始寻找另外一台服务器，它有1\/3 t的时间来找到一台。

当尝试连接到一台不同的服务器时，非常重要的是该台服务器的Zookeeper状态至少要和客户端关注最后的Zookeeper状态一样新。如果一台服务器没有见过某个更新而客户端可能见过，那么客户端是不能连接到该台服务器的。Zookeeper决定在服务中按更新顺序来刷新。每个Zookeeper状态的变化对其他以执行的更新来说是完全顺序化的。 如果客户端在在位置i观察到了一个更新操作， 自然它 无法连接到一台服务器只见到了i’<i。在Zookeeper的实现中，有系统指派给每个更新操作的事务标识符确立了顺序。

图2-7描述了重连事务标识符(zxids)的使用。在客户端因为超时与s1断连之后，它尝试与s2连接，但是s2的状态是滞后的，并不能反映客户端已知的变化。然而，s3看到的变化和客户端一样，它可以安全的连接。

有仲裁者的Zookeeper

目前为止我们使用的配置文件是用于单机模式的。如果服务器起来了，那么服务就起来了，但是如果服务器宕机了，那么整个服务器也就随之挂了。这和当初提供一个可靠的协同服务的承诺有些不符。为了能真正得到可靠的服务，我们需要运行多台服务器。

幸运的是，即使我们只有一台机器也能运行多个服务器。我们只需要建立一个更高级的配置文件。

为了让服务器之间相互联系，它们需要一些联系人的信息。理论上服务器可以通过多播协议来发现对方，但是我们支持除了单一网络外运行Zookeeper集合分散在多个网络中以支持多个Zookeeper集合。

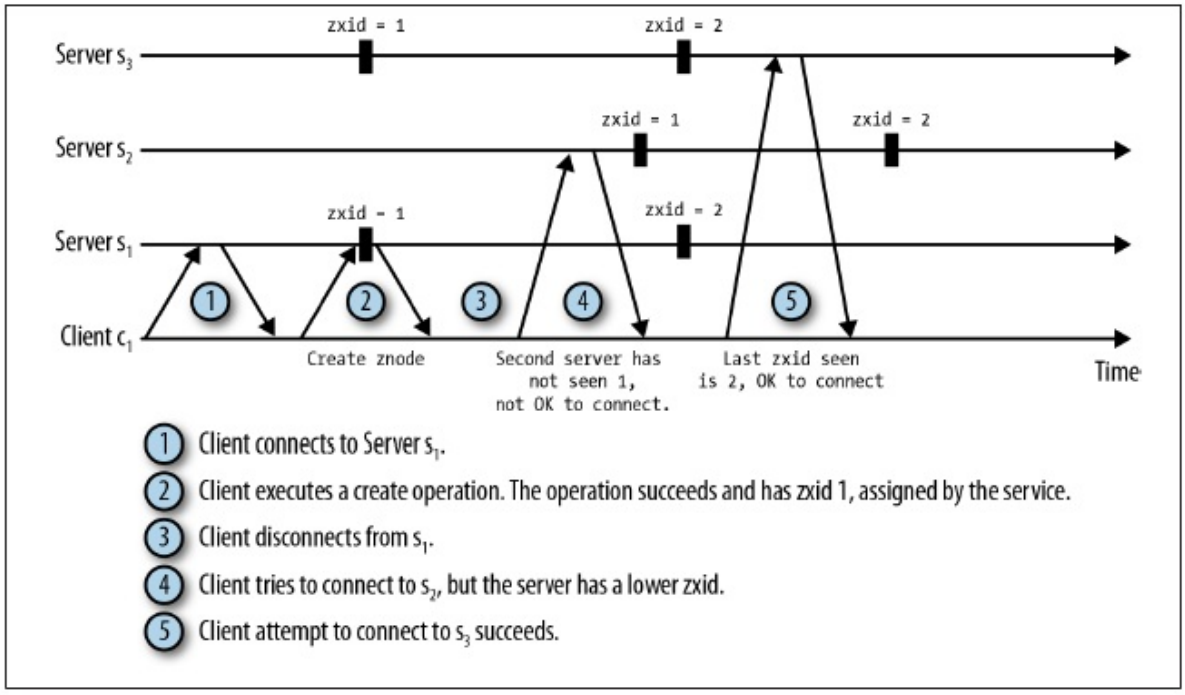


图2-7 客户端重连接例子

为了完成这个例子，我们使用如下配置文件：

```
tickTime=2000
```

```
initLimit=10
```

```
syncLimit=5
```

```
dataDir=./data
```

```
clientPort=2181
```

```
server.1=127.0.0.1    2223
```

```
server.2=127.0.0.1    3334
```

```
server.3=127.0.0.1    4445
```

我们关注最后三行，服务器.n的入口。剩下的普通的配置参数，会在第10章进行说明。

实现一个原语：利用Zookeeper实现锁

使用Zookeeper的一个简单例子就是通过锁来实现临界区。目前有多种锁（如读写锁，全局锁），使用Zookeeper来实现锁也有多种方式。这里我们讨论一个简单的秘方来说明应用如何使用Zookeeper，不考虑其它锁的变体。

假设我们现在有一个应用，其中有n个进程尝试获取一个锁。回想到Zookeeper是不直接暴露原语，所以我们需要使用Zookeeper的接口来操作一个znode以此来实现一个锁。为了得到锁，每个进程p都尝试创建一个znode节点，例如\/lock。如果p成功地创建了节点，那么它就获得了锁，也就能执行它的临界区的代码了。一个潜在的问题是进程p可能会奔溃，永远不会释放锁。这种情况下，没有其他的进程能够再次获取锁，系统可能会死锁。为了避免这种情况，当我们创建它的时候必须让\/lock节点是临时节点。

只要节点存在，其他想创建\/lock节点进程都会失败。所以，它们监视\/lock节点的变化，一旦它们检测到了\/lock节点被删除了就会再次尝试获取锁。当收到\/lock节点被删除的通知时，如果进程p′还对获取锁感兴趣的话，它会重复创建\/lock节点的步骤。如果另外一个进程已经创建了节点，那么继续监视它。

实现Master-Worker的例子

- Master
- Worker
- Client

在这一小节中，我们利用*zkCli*工具实现master-worker例子中某些功能。这个例子仅只有教学的目的，不推荐在构建系统中使用zkCli。使用zkCli的目的就是简化描述如何利用zookeeper实现协同的秘方，把大量的实现细节先放到一边。我们在下一章中再来看实现的细节。

master-worker模型有三个角色:

Master

master监视新的worker和任务，并分配任务给可用的worker。

Worker

worker向系统注册它们自己的信息，确保master能“看见”它们并能执行任务，同时监视新的任务。

Client

客户端创建新的任务并等待系统的响应。

让我们来过一遍不同的角色，了解每个角色需要执行的步骤。

Master的角色

因为只有一个进程能成为master，进程必须在zookeeper锁住领导权才能成为master。要达到此目的，进程需要创建一个名为\/master的临时节点：

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master1.example.com:2223" <1>
Created /master
[zk: localhost:2181(CONNECTED) 1] ls / <2>
[master, zookeeper]
[zk: localhost:2181(CONNECTED) 2] get /master <3>
"master1.example.com:2223"
cZxid = 0x67
ctime = Tue Dec 11 10:06:19 CET 2012
mZxid = 0x67
mtime = Tue Dec 11 10:06:19 CET 2012
pZxid = 0x67
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x13b891d4c9e0005
dataLength = 26
numChildren = 0
[zk: localhost:2181(CONNECTED) 3]
```

1. 创建一个master节点得到领导权。我们使用-e标志表明我们创建的临时节点。
2. 列出zookeeper树的根节点。
3. 得到\/master节点的元数据和内容数据。

刚刚发生了什么？我们先创建了一个临时节点\/master。以防其他人需要在Zookeeper外和它交互，我们在节点上加上了主机的信息。加上主机信息并不是必须的，这么做只是演示一下我们可以这么做。为了让节点成为临时节点，我们加上了-e标志。记住临时节点会在创建它的会话关闭或者过期时自动地删除。

假如现在我们有两个进程竞争领导master的角色，但是在任意时刻最多只能有一个激活的master。

其他的进程是备用的master。假设其他进程不知道master已经被选举出来了，它们也尝试创建\/master节点。我们看看发生了什么：

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master2.example.com:2223"
Node already exists: /master
[zk: localhost:2181(CONNECTED) 1]
```

Zookeeper告诉我们\/master节点已经存在了。以这种方式第二个进程就知道了已经有master了。然而，那个激活的master可能奔溃，后备的master需要接管激活master的角色。为了检测到这种情况，我们需要在\/master节点上按照下面的方式设置一个监视器：

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master2.example.com:2223"
Node already exists: /master
[zk: localhost:2181(CONNECTED) 1] stat /master true
cZxid = 0x67
ctime = Tue Dec 11 10:06:19 CET 2012
mZxid = 0x67
mtime = Tue Dec 11 10:06:19 CET 2012
pZxid = 0x67
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x13b891d4c9e0005
dataLength = 26
numChildren = 0
[zk: localhost:2181(CONNECTED) 2]
```

stat命令能够获得一个节点的属性，它允许我们在一个存在的节点上设置监视器。在路径后面设置参数为true来设置监视器。在本例中，当那个活跃的主master奔溃时，我们观察到了以下信息：

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master2.example.com:2223"
Node already exists: /master
[zk: localhost:2181(CONNECTED) 1] stat /master true
cZxid = 0x67
ctime = Tue Dec 11 10:06:19 CET 2012
```

```
mZxid = 0x67
mtime = Tue Dec 11 10:06:19 CET 2012
pZxid = 0x67
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x13b891d4c9e0005
dataLength = 26
numChildren = 0
[zk: localhost:2181(CONNECTED) 2]
WATCHER::
WatchedEvent state:SyncConnected type:NodeDeleted path:/master
[zk: localhost:2181(CONNECTED) 2] ls /
[zookeeper]
[zk: localhost:2181(CONNECTED) 3]
```

注意在最后输出的NodeDeleted事件。该事件表明那个主服务器的会话关闭了，要么就是过期了。注意到\/master节点不再存在了。备用的主服务器现在应该尝试通过创建\/master节点来变为主服务器。

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master2.example.com:2223"
Node already exists: /master
[zk: localhost:2181(CONNECTED) 1] stat /master true
cZxid = 0x67
ctime = Tue Dec 11 10:06:19 CET 2012
mZxid = 0x67
mtime = Tue Dec 11 10:06:19 CET 2012
pZxid = 0x67
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x13b891d4c9e0005
```

```
dataLength = 26
```

```
numChildren = 0
```

```
[zk: localhost:2181(CONNECTED) 2]
```

```
WATCHER::
```

```
WatchedEvent state:SyncConnected type:NodeDeleted path:/master
```

```
[zk: localhost:2181(CONNECTED) 2] ls /
```

```
[zookeeper]
```

```
[zk: localhost:2181(CONNECTED) 3] create -e /master "master2.example.com:2223"
```

```
Created /master
```

```
[zk: localhost:2181(CONNECTED) 4]
```

因为它成功地创建了\/master节点，现在客户端就变成了激活的master。

工人、任务和指派关系

在我们讨论worker和客户端的步骤之前，我们先来创建三个非常重要的父节点：\/workers，\/tasks和\/assign。

```
[zk: localhost:2181(CONNECTED) 0] create /workers ""
```

```
Created /workers
```

```
[zk: localhost:2181(CONNECTED) 1] create /tasks ""
```

```
Created /tasks
```

```
[zk: localhost:2181(CONNECTED) 2] create /assign ""
```

```
Created /assign
```

```
[zk: localhost:2181(CONNECTED) 3] ls /
```

```
[assign, tasks, workers, master, zookeeper]
```

```
[zk: localhost:2181(CONNECTED) 4]
```

这三个节点都是持久化的节点，并不包含数据。在本例中我们使用这些节点来告诉我们哪些worker是可用的，还有哪些任务等着被分配，同时分配给worker的任务。

在现实的应用中，这些节点要么是被主进程在分配任务之前创建的，要么是某些引导步骤创建的。不论它们是如何创建的，一旦它们被创建，主节点就需要监视孩子节点\/workers和\/tasks的变化：

```
[zk: localhost:2181(CONNECTED) 4] ls /workers true
```

```
[]
```

```
[zk: localhost:2181(CONNECTED) 5] ls /tasks true
```

```
[]
```

```
[zk: localhost:2181(CONNECTED) 6]
```

注意到我们在ls命令中使用了可选参数true，像我们之前在master上使用stat一样。true参数在本例中设置了相应节点的孩子节点变化的监视器。

Worker的角色

worker的第一步就是通知master它能执行任务。通过在\/workers下面创建临时节点来达到此目的。Worker使用它们的主机名来标示自己:

```
[zk: localhost:2181(CONNECTED) 0] create -e /workers/worker1.example.com
```

```
"worker1.example.com:2224"
```

```
Created /workers/worker1.example.com
```

```
[zk: localhost:2181(CONNECTED) 1]
```

从输出中可以确认节点已经被创建。回想到master已经在\/workers的孩子节点上设置了监视器。一旦worker在\/workers下面创建了节点，master节点就会观察到以下通知:

```
WATCHER::
```

```
WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/workers
```

接下来，worker需要创建一个父节点，\/assign\/worker1.example.com，为的是要接受任务分配。同时，通过执行带参数true的ls命令来监视新的任务:

```
[zk: localhost:2181(CONNECTED) 0] create -e /workers/worker1.example.com
```

```
"worker1.example.com:2224"
```

```
Created /workers/worker1.example.com
```

```
[zk: localhost:2181(CONNECTED) 1] create /assign/worker1.example.com ""
```

```
Created /assign/worker1.example.com
```

```
[zk: localhost:2181(CONNECTED) 2] ls /assign/worker1.example.com true
```

```
[]
```

```
[zk: localhost:2181(CONNECTED) 3]
```

worker现在准备好接受任务分配了。当我们讨论客户端的角色时顺带了解一下任务分配。

客户端的角色

客户端向系统中增加任务。在本例中，我们不关心任务的实际组成。现在我们假设客户端需要master-worker系统运行一个命令cmd。要在系统中增加一个任务，客户端执行下面的命令：

```
[zk: localhost:2181(CONNECTED) 0] create -s /tasks/task- "cmd"
```

```
Created /tasks/task-0000000000
```

为了让新增的任务有顺序，我们让任务节点是顺序性的，自然就形成了一个队列。客户端现在不得不等到任务被执行。一旦任务完成，那个执行任务的worker就会为task创建一个状态节点。当客户端看到任务的状态节点被创建时认为任务已经被执行了。自然，客户端需要监视状态节点的创建：

```
[zk: localhost:2181(CONNECTED) 1] ls /tasks/task-0000000000 true
```

```
[]
```

```
[zk: localhost:2181(CONNECTED) 2]
```

执行任务的worker创建了一个孩子节点\/tasks\/task-0000000000。这就是通过ls命令来监视\/tasks\/task-0000000000孩子节点的原因。

一旦任务节点被创建了，master就会观察到下面的事件：

```
[zk: localhost:2181(CONNECTED) 6]
```

```
WATCHER::
```

```
WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/tasks
```

master接下来检查新的任务，获取可用的worker列表，并把它指派给worker1.example.com：

```
[zk: 6] ls /tasks
```

```
[task-0000000000]
```

```
[zk: 7] ls /workers
```

```
[worker1.example.com]
```

```
[zk: 8] create /assign/worker1.example.com/task-0000000000 ""
```

```
Created /assign/worker1.example.com/task-0000000000
```

```
[zk: 9]
```

worker接着收到一个新任务被分配的通知：

```
[zk: localhost:2181(CONNECTED) 3]
```

```
WATCHER::
```

```
WatchedEvent state:SyncConnected type:NodeChildrenChanged
```

```
path:/assign/worker1.example.com
```

worker然后检查新的任务，看看任务是否被分配给它：

```
WATCHER::
```

```
WatchedEvent state:SyncConnected type:NodeChildrenChanged
```

```
path:/assign/worker1.example.com
```

```
[zk: localhost:2181(CONNECTED) 3] ls /assign/worker1.example.com
```

```
[task-0000000000]
```

```
[zk: localhost:2181(CONNECTED) 4]
```

一旦worker执行完了任务，它会在\/tasks下增加一个状态节点：

```
[zk: localhost:2181(CONNECTED) 4] create /tasks/task-0000000000/status "done"
```

```
Created /tasks/task-0000000000/status
```

```
[zk: localhost:2181(CONNECTED) 5]
```

然后客户端收到一个通知，检查结果：

```
WATCHER::
```

```
WatchedEvent state:SyncConnected type:NodeChildrenChanged
```

```
path:/tasks/task-0000000000
```

```
[zk: localhost:2181(CONNECTED) 2] get /tasks/task-0000000000
```

```
"cmd"
```

```
cZxid = 0x7c
```

```
ctime = Tue Dec 11 10:30:18 CET 2012
```

```
mZxid = 0x7c
```

```
mtime = Tue Dec 11 10:30:18 CET 2012
```

```
pZxid = 0x7e
```

```
cversion = 1
```

```
dataVersion = 0
```

```
aclVersion = 0
```

```
ephemeralOwner = 0x0
```

```
dataLength = 5
```

```
numChildren = 1
```

```
[zk: localhost:2181(CONNECTED) 3] get /tasks/task-0000000000/status
```

```
"done"
```

```
cZxid = 0x7e
```

```
ctime = Tue Dec 11 10:42:41 CET 2012
```

```
mZxid = 0x7e
```

```
mtime = Tue Dec 11 10:42:41 CET 2012
```

```
pZxid = 0x7e
```

```
cversion = 0
```

```
dataVersion = 0
```

```
aclVersion = 0
```

```
ephemeralOwner = 0x0
```

```
dataLength = 8
```

```
numChildren = 0
```

```
[zk: localhost:2181(CONNECTED) 4]
```

客户端检查状态节点的内容来判断任务发生了什么。本例中，任务被成功的执行了，结果是“done”。当然，任务可以更加复杂，甚至有另外的分布式系统参与。无论任务实际上是怎么样的，执行的机制和传递结果本质上都是一样的。

小结

本章中我们已经了解了大量的Zookeeper的基础知识。我们通过Zookeeper提供的API了解了它的基础功能，探寻了关于它的架构方面的某些重要概念，比如使用仲裁者用于复制。现阶段理解Zookeeper的复制协议并不重要，但是理解仲裁者的概念很重要因为当你部署Zookeeper时需要制定服务器的数量。另外一个重要的概念就是会话。会话语义对于Zookeeper的“承诺”至关重要因为大部分的“承诺”都涉及会话。

为了对如何使用Zookeeper工作有一个预先的理解，我们使用zkCli工具来访问Zookeeper服务器，并执行请求。我们使用这个工具来展示master-worker例子中大部分操作。当实现一个真实的Zookeeper应用时，你不应该使用这个工具。它大部分是用于调试和监控的目的。相反的，你应该使用Zookeeper提供的一种语言的绑定实现。在下一章中，我们使用Java来实现我们的例子。