

提供各种IT类书籍pdf下载，如有需要，请QQ:2404062482

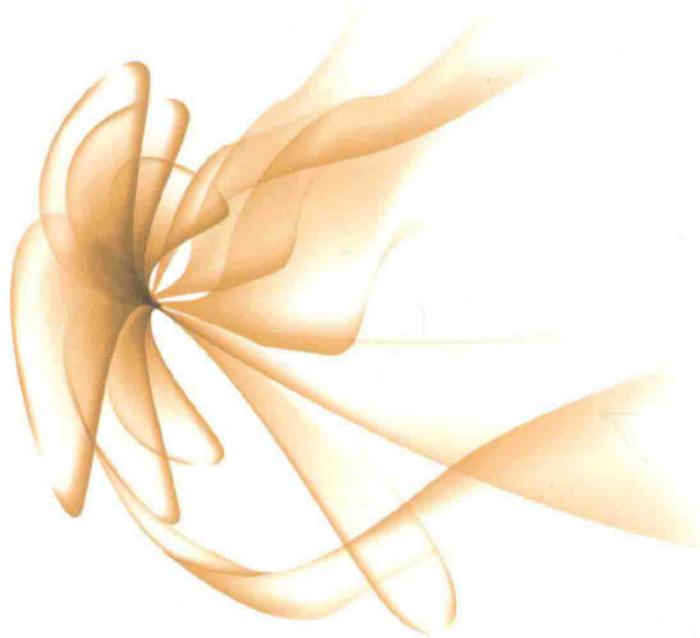
注：链接至淘宝，不喜者勿入！整理那么多资料也不容易，请多多见谅！非诚勿扰！

[更多此类书籍](#)



绝技源于江湖、将军发于卒伍，本书包含作者从程序员到首席架构师十多年职业生涯所积累的实战经验。

这不是一本讲怎么使用Hadoop的书，而是一本讲实现Hadoop功能的书，本书系统讲解构建大规模分布式系统的核心技术和实现方法，包含开源的代码，手把手教你掌握分布式技术。



Architecture and Design of Large Scale
Distributed System

大规模分布式系统 架构与设计实战

彭渊◎著



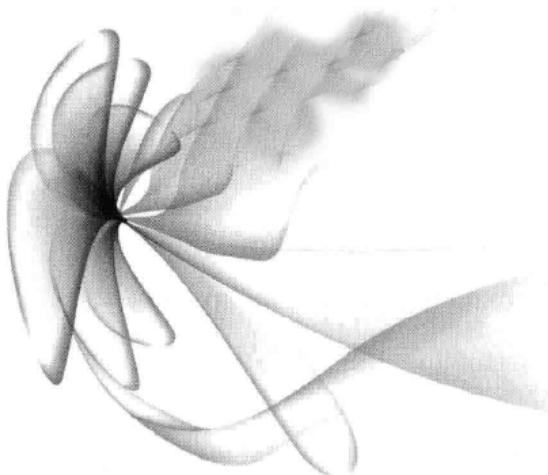
CD-ROM

技术丛书

Architecture and Design of Large Scale
Distributed System

大规模分布式系统 架构与设计实战

彭 淵◎著



机械工业出版社

图书在版编目 (CIP) 数据

大规模分布式系统架构与设计实战 / 彭渊著. —北京：机械工业出版社，2014.1
(大数据技术丛书)

ISBN 978-7-111-45503-5

I. 大… II. 彭… III. 分布式计算机系统—系统设计 IV. TP338.8

中国版本图书馆CIP 数据核字 (2014) 第013468号

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书是作者从程序员到首席架构师十多年职业生涯的实战经验总结，系统讲解构建大规模分布式系统的核心技术与实现方法，包含作者开源的Fourinone系统的设计与实现过程，手把手教你掌握分布式技术。通过学习这个系统的实现方法与相关的理论，读者可快速掌握分布式系统的理论并设计自己的分布式系统。

本书从分布式计算的基本概念开始，解剖了众多流行概念的本质，深入讲解分布式系统的基本原理与实现方式，包括 master-slave 结构、消息中枢模式、网状直接交互模式、并行结合串行模式等，以及 Fourinone 系统的架构、实现分布式功能的示例。接下来详细介绍分布式协调、分布式缓存、消息队列、分布式文件系统、分布式作业调度平台的设计与实现方法，不仅包括详细的架构原理、算法，还给出了实现步骤、核心 API、实现代码。随书附带的光盘包括书中示例代码以及 Fourinone 系统源代码。

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑：吴 怡

北京市荣盛彩色印刷有限公司印刷

2014 年 2 月第 1 版第 1 次印刷

186mm × 240mm • 15 印张

标准书号：ISBN 978-7-111-45503-5

ISBN 978-7-89405-278-0 (光盘)

定 价：59.00 元 (附光盘)

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

前　　言

在大数据、云计算如火如荼的今天，各类技术产品顺应潮流层出不穷。大家是不是有这种感觉：**Hadoop** 还没学完，**Storm** 就来了；**Storm** 刚学会安装配置，**Spark**、**Hama**、**Yarn** 等又一起出现了；同时国内外各大云平台厂商，如 **Google**、亚马逊、阿里云等，还在推各自应用开发平台……要学习的东西太多了，就是这样疲于奔命地学，刚学会了某个产品的安装配置与开发步骤，没多久它又过时了。

这么多千姿百态的分布式技术和产品背后有没有某些共性的东西呢？能让我们换了马甲还能认出它，让我们超越学习每个产品的“安装配置开发”而掌握背后的精髓呢？有没有可能学一反三，学一招应万招，牢牢掌握好技术的船舵，穿越一次次颠覆性的技术浪潮？本书的目的就是为你揭示分布式技术的核心内幕，透彻理解其精髓，站在浪潮之巅。

因此，这不是一本讲如何使用 **Hadoop** 的书，而是一本讲实现 **Hadoop** 功能的书，是一本讲如何简化实现分布式技术核心功能的书。这不是一本空谈概念、四处摘抄的书，而是来源于作者十多年来在私企、港企、外包、创业、淘宝、华为等企业打拼，从底层程序员一路走到首席架构师的实战经验总结。绝技源于江湖，将军发于卒伍，这本书讲的是你在课本上学不到核心技术，无论你是在中国什么样的 IT 企业做什么样的分布式应用，这本书对你都具备参考性。

本书面向千千万万战斗在一线攻城拔寨的程序员、工程师们，你可以有很多基础，也可以从头开始，本书尽量做到深入浅出和通俗易懂，希望帮助你降低分布式技术的学习成本，帮助你更容易完成工作任务，更轻松地挣钱。

本书根据分布式技术的主要应用，分别介绍分布式并行计算的基本概念、分布式协调、分布式缓存、消息队列、分布式文件系统、分布式作业调度平台等，详细阐述分布式各技术的架构原理和实现方式，并附带大量示例，便于读者实际操作运行。基于本书原理，作者用 **Java** 实现并开源了 **Fourinone** 框架，这是一个高效的分布式系统，归纳在 150KB 源码里，代码不到 1 万行，让你能够轻松掌握。学习开发核心技术的诀窍是多动手，建议读者运行本书附带的大量 **DEMO**，在运行后细细体会分布式的理论，进行反思和总结。本书归纳的设计思想和算法不局限于某个框架，读者领会

后可以用任何语言来实现自己的分布式系统。

本书各章有一定的独立性，阅读本书的方式比较自由，可以从头开始，也可以随性翻阅。从第2章开始，每章都有理论部分与示例，读者可以先运行 DEMO，不清楚的地方再回看原理；也可以先看原理，再运行 DEMO 加深理解。由于时间的限制，且本书写作的时期是在作者最为忙碌和事业的转折时期，匆忙中，难免出错，请朋友们海涵，并提出意见以便于今后纠正。最后感谢机械出版社华章公司所有幕后编辑的大量工作，感谢所给予我帮助与支持的领导和朋友。

本书所有源码附带在光盘里。你也可以登录开源地址下载，开源地址：<http://code.google.com/p/fourinone>

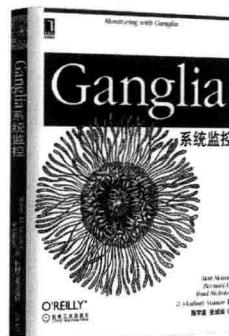
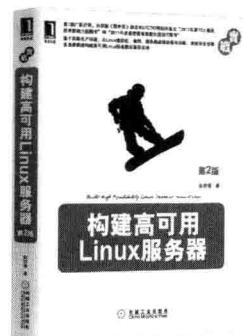
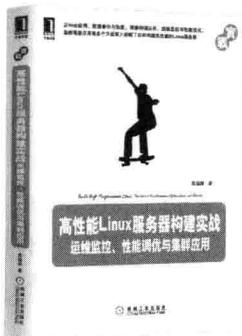
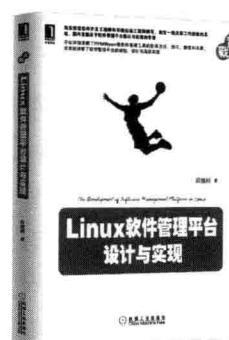
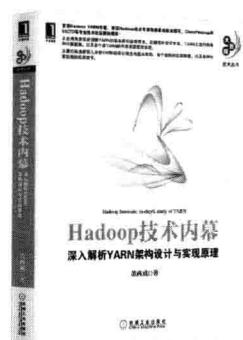
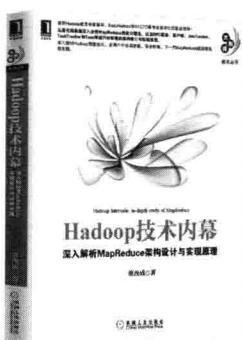
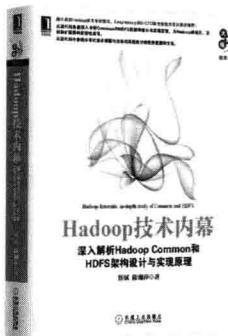
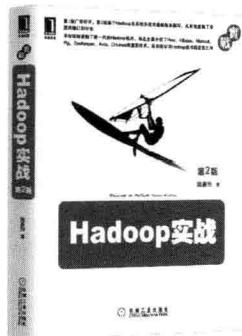
作者联系方式：邮箱：Fourinone@yeah.net

QQ 群 1：1313859

QQ 群 2：241116021

QQ 群 3：23321760

推荐阅读



目 录

前 言

第1章 概述	1
1.1 分布式计算、并行计算、云计算概述	1
1.2 分布式产品Hadoop、ZooKeeper、HBase概述	6
1.3 Fourinone的产生背景	12
第2章 分布式并行计算的原理与实践	14
2.1 分布式并行计算模式	14
2.1.1 最初想到的master-slave结构	14
2.1.2 “包工头-职介所-手工仓库-工人”模式	15
2.1.3 基于消息中枢的计算模式	17
2.1.4 基于网状直接交互的计算模式	18
2.1.5 并行结合串行模式	22
2.1.6 包工头内部批量多阶段处理模式	23
2.1.7 计算集群模式和兼容遗留计算系统	24
2.1.8 工人计算的服务化模式	26
2.2 跟Hadoop的区别	28
2.3 关于分布式的一些概念与产品	30
2.4 配置文件和核心API介绍	35
2.5 实践与应用	36
2.5.1 一个简单的示例	36
2.5.2 工头工人计算模式更完整的示例	39
2.5.3 工人合并互相say hello的示例	44

2.5.4 实现Hadoop经典实例Word Count	48
2.5.5 分布式多机部署的示例	52
2.5.6 分布式计算自动部署的示例	53
2.5.7 计算过程中的故障和容灾处理	57
2.5.8 计算过程中的相关时间属性设置	60
2.5.9 如何在一台计算机上一次性启动多个进程	63
2.5.10 如何调用C/C++程序实现	68
2.5.11 如何中止工人计算和超时中止	68
2.5.12 使用并行计算大幅提升递归算法效率	73
2.5.13 使用并行计算求圆周率 π	81
2.5.14 从赌钱游戏看PageRank算法	86
2.5.15 使用并行计算实现上亿排序	96
2.5.16 工人服务化模式应用示例	104
2.6 实时流计算	107
第3章 分布式协调的实现	111
3.1 协调架构原理简介	111
3.2 核心API	113
3.3 权限机制	115
3.4 相对于ZooKeeper的区别	116
3.5 与Paxos算法的区别	117
3.6 实践与应用	119
3.6.1 如何实现公共配置管理	119
3.6.2 如何实现分布式锁	126
3.6.3 如何实现集群管理	129
3.6.4 多节点权限操作示例	134
3.6.5 领导者选举相关属性设置	137
第4章 分布式缓存的实现	139
4.1 小型网站或企业应用的缓存实现架构	139
4.2 大型分布式缓存系统实现过程	140
4.3 一致性哈希算法的原理、改进和实现	147

4.4	解决任意扩容的问题	152
4.5	解决扩容后数据均匀的问题	153
4.6	分布式Session的架构设计和实现	154
4.7	缓存容量的相关属性设置	156
4.8	缓存清空的相关属性设置	158
第5章	消息队列的实现	162
5.1	闲话中间件与MQ	162
5.2	JMS的两种经典模式	163
5.3	如何实现发送接收的队列模式	164
5.4	如何实现主题订阅模式	168
第6章	分布式文件系统的实现	173
6.1	FTTP架构原理解析	174
6.2	搭建配置FtpAdapter环境	177
6.3	访问集群文件根目录	179
6.4	访问和操作远程文件	181
6.5	集群内文件复制和并行复制	184
6.6	读写远程文件	187
6.7	解析远程文件	189
6.8	并行读写远程文件	191
6.9	批量并行读写远程文件和事务补偿处理	194
6.10	如何进行整型读写	198
6.11	基于整型读写的上亿排序	205
第7章	分布式作业调度平台的实现	219
7.1	调度平台的设计与实现	219
7.2	资源隔离的实现	224
7.3	资源调度算法	226
7.4	其他作业调度平台简介	227
7.4.1	其他MPI作业资源调度技术	227
7.4.2	Mesos和Yarn简介	229

第1章

概 述

在概述分布式核心技术之前，我们有必要先概括阐述一下分布式计算、并行计算、云计算等相关概念，以及市场上流行的相关技术产品，如 Hadoop 生态体系，然后再结合背景引出我们为什么要归纳出一个轻量级的分布式框架。本章为后续章节的背景。本章意在使读者对分布式技术话题的前因后果先有所了解。

由于只是概述，我们对涉及的分布式计算概念和 Hadoop 生态体系只是蜻蜓点水地带过，目的仅是让读者了解到这些内容大致是什么，详细的使用方法和开发方法可以参考其他书籍。

1.1 分布式计算、并行计算、云计算概述

1. 什么是分布式计算

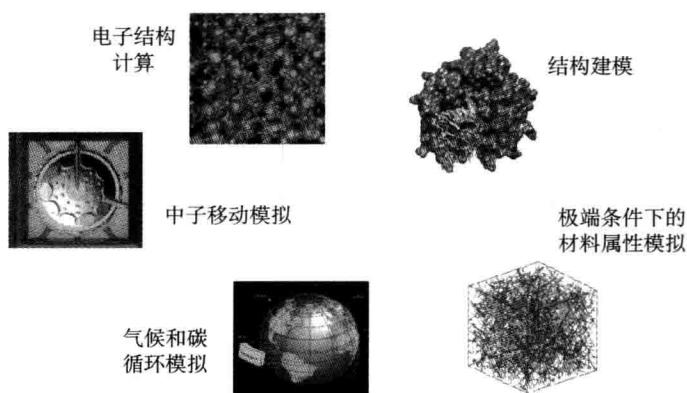


图 1-1 科学中的分布式计算

经科学研究发现：目前存在很多万亿次计算实例，其中涉及的问题都需要非常巨大的计算能力才能解决。这类问题很多还是跨学科的、极富挑战性的、人类亟待解决的科研课题，如图 1-1 所示。

除此之外还有很多研究项目需要巨大的计算能力，例如：

1) 解决较为复杂的数学问题，例如：GIMPS（寻找最大的梅森素数）。

梅森素数 (Mersenne Prime) 是指形如 $2^p - 1$ 的正整数，其中指数 p 是素数，常记为 M_p 。若 M_p 是素数，则称为梅森素数。 $p=2, 3, 5, 7$ 时， M_p 都是素数，但 $M_{11}=2047=23 \times 89$ 不是素数，是否有无穷多个梅森素数是数论中未解决的难题之一。截至 2012 年 7 月已累计发现 47 个梅森素数，最大的是 $p=43,112,609$ ，此时 M_p 是一个 12,978,189 位数。

值得一提的是，中国的一位数学家算出了梅森素数的分布规律图，并用简练的数学公式描述了出来。如果借助计算机的并行计算，也许会对寻找该数字分布规律有所帮助。

2) 研究寻找最为安全的密码系统，例如：RC-72（密码破解）。

3) 生物病理研究，例如：Folding@home（研究蛋白质折叠、误解、聚合，以及由此引起的相关疾病）。

4) 各种各样疾病的药物研究，例如：United Devices（对抗癌症的有效药物）。

5) 信号处理，例如：SETI@Home（寻找地球外文明）。

由上不难看出，这些项目都很庞大，都需要惊人的计算量，仅由单个电脑或个人在一个能让人接受的时间内计算完成是决不可能的。在以前，这些问题都应该由超级计算机来解决。但是，超级计算机的造价和维护非常昂贵，这不是一个普通的科研组织能承受的。随着科学的发展，一种廉价的、高效的、维护方便的计算方法应运而生——分布式计算！

所谓**分布式计算**其实是一门计算机科学，它研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。

最近的一个分布式计算项目已经使用世界各地成千上万位志愿者的计算机来进行操作，利用这些闲置计算能力，通过因特网，你可以分析来自外太空的电信号，并探索可能存在的外星智慧生命；你可以寻找超过 1000 万位数字的梅森素数；你也可以寻找并发现对抗艾滋病病毒的更为有效的药物。

讨论

我们能否利用访问淘宝网的几千万个用户的电脑做一次分布式计算？

这里仅仅点拨一下，回答该问题其实涉及侵犯用户隐私。用户用电脑上网，安装了很多客户端软件，有的软件是拥有本地所有权限的，它们是否在偷偷用用户的电脑干其他私活，用户也不知道。这曾经引发过国内两大客户端巨头的官司。但是由此可以看出，千千万万用户的电脑是可

以利用起来做计算的，当然计算必须围绕一个消息中枢模式进行，因为用户电脑间的网络结构千差万别，无法直接连接。

2. 什么是并行计算

并行计算其实早就有了，所有大型编程语言都支持多线程，多线程就是一种简单的并行计算方式，多个程序线程并行地争抢 CPU 时间。

并行计算（Parallel Computing）是指同时使用多种计算资源解决计算问题的过程。并行计算的主要目的是快速解决大型且复杂的计算问题。此外还包括：利用非本地资源节约成本，即使用多个“廉价”计算资源取代大型计算机，同时克服单个计算机上存在的存储器限制问题。

传统上，串行计算是指在单个计算机（具有单个中央处理单元）上执行软件写操作。CPU 逐个使用一系列指令解决问题，但在每一个时刻只能执行一种指令。并行计算是在串行计算的基础上演变而来的，它努力仿真自然世界中的事务状态：一个序列中众多同时发生的、复杂且相关的事件。

为利用并行计算，通常计算问题表现为以下特征：

- 将工作分解成离散部分，有助于同时解决；
- 随时并及时地执行多个程序指令；
- 多计算资源下解决问题的耗时要少于单个计算资源下的耗时。

并行计算是相对于串行计算来说的，所谓并行计算分为时间上的并行和空间上的并行。时间上的并行就是指流水线技术，而空间上的并行则是指用多个处理器并发地执行计算。

3. 并行计算与串行计算的关系

并行计算与串行计算的关系如图 1-2 所示。

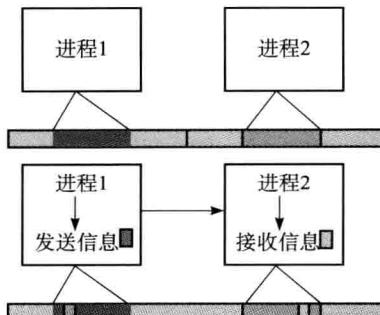


图 1-2 串行（上图），并行（下图）

结合图 1-2，对串行计算和并行计算分析如下：

- 传统的串行计算，分为“指令”和“数据”两个部分，并在程序执行时“独立地申请和占有”内存空间，且所有计算均局限于该内存空间。
- 并行计算将进程相对独立的分配于不同的节点上，由各自独立的操作系统调度，享有独立的CPU和内存资源（内存可以共享）；进程间相互信息交换是通过消息传递进行的。

4. 什么是云计算

云计算是一种理念，是旧瓶子装新酒，它实际上是分布式技术+服务化技术+资源隔离和管理技术（虚拟化），如图1-3所示。商业公司对云计算都有自己的定义，例如：

- 一种计算模式：把IT资源、数据、应用作为服务通过网络提供给用户（如IBM公司）。
- 一种基础架构管理方法论：把大量的高度虚拟化的资源管理起来，组成一个大的资源池，用来统一提供服务（如IBM公司）。
- 以公开的标准和服务为基础，以互联网为中心，提供安全、快速、便捷的数据存储和网络计算服务（如Google公司）。

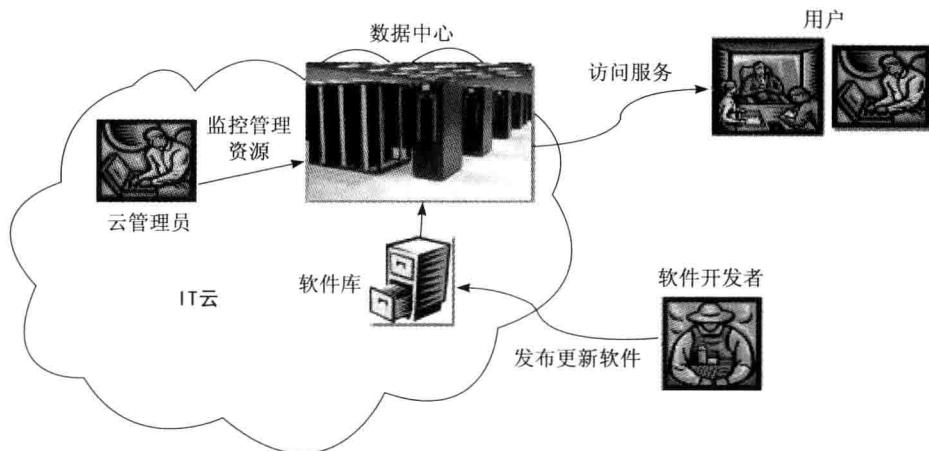
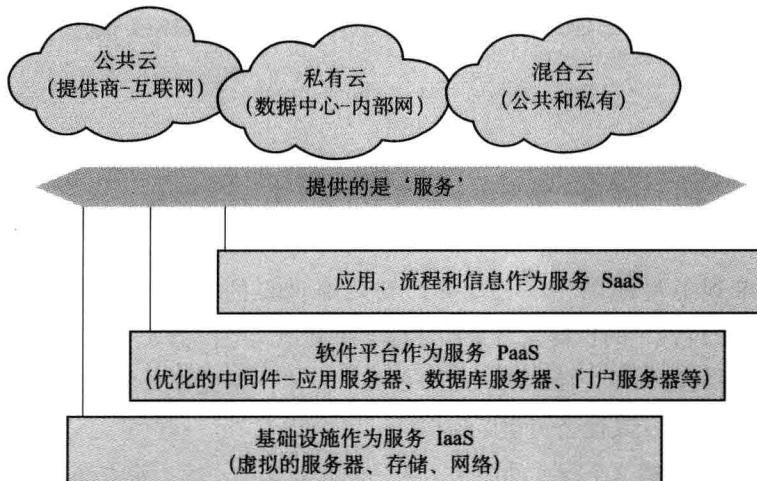


图1-3 云计算示意图

通俗意义上的云计算往往是上面这个架构图包含的内容，开发者利用云API开发应用，然后上传到云上托管，并提供给用户使用，而不关心云背后的运维和管理，以及机器资源分配等问题。

虚拟化和服务化是云计算的表现形式（参见图1-4）：



- 虚拟化技术包括：资源虚拟化、统一分配监测资源、向资源池中添加资源。虚拟化的技术非常多，有的是完全模拟硬件的方式去运行整个操作系统，比如我们熟悉的 VMWare，可以看做重量级虚拟化产品。也有通过软件实现的，共享一个操作系统的轻量级虚拟化，比如 Solaris 的 Container、Linux 的 lxc(关于 cgroup 的方式我们在 7.2 节也会谈到)。虚拟化的管理、运维多数是通过工具完成的，比如 Linux 的 VirtManager、VMWare 的 vSphere、VMWare 的 vCloud 等等。
- 服务思想包括：
 - 软件即服务 (Software-as-a-Service, SAAS)。是目前最为成熟的云计算服务模式。在这种模式下，应用软件安装在厂商或者服务供应商那里，用户可以通过某个网络来使用这些软件。这种模式具有高度的灵活性、可靠性和可扩展性，因此能够降低客户的维护成本和投入，而且运营成本也得以降低。最著名的例子就是 Salesforce.com。
 - 平台即服务 (Platform-as-a-Service, PAAS)。提供了开发平台和相关组件，软件开发者可以在这个开发平台之上开发新的应用，或者使用已有的各种组件，因此可以不必购买开发、质量控制或生产服务器。Salesforce.com 的 Force.com、Google 的 App Engine 和微软的 Azure(微软云计算平台) 都采用了 PAAS 的模式。
 - 基础设施作为服务 (Infrastructure as a Service, IAAS)。通过互联网提供了数据中心、硬件和软件基础设施资源。IAAS 可以提供服务器、操作系统、磁盘存储、数据库和 / 或信息资源。用户可以像购买水电煤气一样购买这些基础设施资源使用。

1.2 分布式产品 Hadoop、ZooKeeper、HBase 概述

1. Hadoop

说到云计算技术和产品，不能不提到 Google 这家企业。曾经，微软是 IT 行业的象征，号称只招最聪明的人。十年后，微软逐渐疲软了下来，Google 这家企业取而代之号称只招最聪明的人。

从搜索引擎到卫星地图，到云计算，到风靡世界的 Android，到现在的无人汽车、Google 眼镜，以及传说中的机器人之父和在他家里满地爬的机器人……这个行业总有这么一家企业成为最高科技的领袖，控制着大部分的核心技术，聚拢着大部分的一流人才，垄断着最大份额的市场，几乎让其他公司望而却步。

因此，我们不难理解这个行业的结果，第一的企业吃肉，第二喝点汤，其他都亏损。

也许在学校读书时，考试成绩前十名都是好学生，但是 IT 行业的社会竞争，必须要做到第一，成为所在领域的老大才有利可图。

再简单回顾一下云计算相关技术产品的发展史：

- 2002 ~ 2004：Apache Nutch。
- 2004 ~ 2006：Google 发表 GFS 和 MapReduce 相关论文。Apache 在 Nutch 中实现 HDFS 和 MapReduce。
- 2006 ~ 2008：Hadoop 项目从 Nutch 中分离。
- 2008 年 7 月，Hadoop 赢得 Terabyte Sort Benchmark。

从上面我们可以看到早在 2004 ~ 2006 年，Google 就发表了两篇与 GFS 和 MapReduce 相关的论文，分别是分布式文件系统和基于它的 Map/Reduce 并行计算。Apache 的开源项目 Hadoop 便是根据这两篇论文的思想实现的 Java 版本，Hadoop 引起关注是在它赢得了一次 TB 排序比赛：Hadoop Wins Terabyte Sort Benchmark: One of Yahoo's Hadoop clusters sorted 1 terabyte of data in 209 seconds, which beat the previous record of 297 seconds in the annual general purpose (Daytona) terabyte sort benchmark. This is the first time that either a Java or an open source program has won.

用了大量的机器在 209 秒完成 1TB 排序，提升了之前 297 秒的记录。

值得一提的是，Hadoop 的作者 Doug Cutting 同时也是 Lucene 和 Nutch 的作者，早年供职 Yahoo，后来担任 Apache 软件基金会主席。

Hadoop 的名字来源于 Doug Cutting 儿子的一个宠物名。Hadoop 从最初的 HDFS 分布式文件系统发展到后来的 Hadoop+ZooKeeper+Hive+Pig+HBase 生态体系。

HDFS 提供了一个可扩缩的、容错的、可以在廉价机器上运行的分布式文件系统，按行

进行存储，按 64MB 块进行文件拆分。

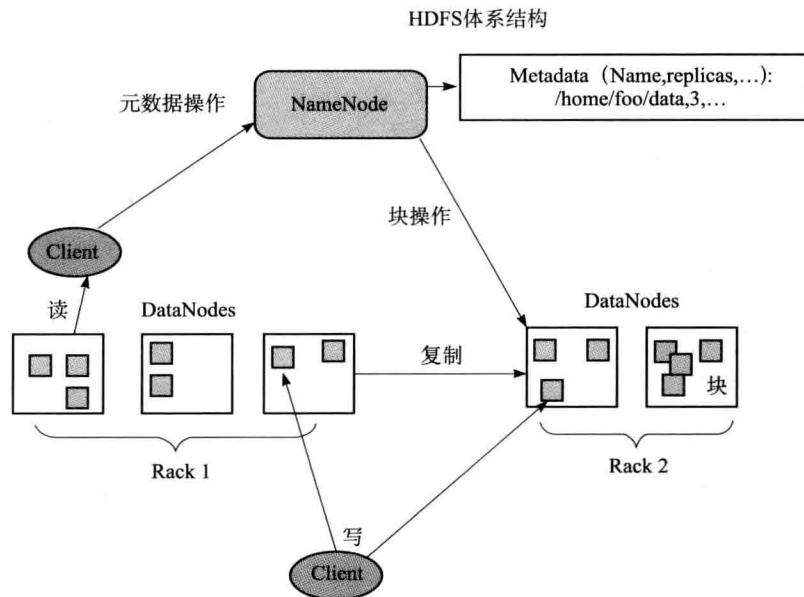


图 1-5 HDFS 体系结构

我们可以看到，HDFS 的架构是一个 NameNode 和多个 DataNode 的结构（参见图 1-5）：

□ NameNode

存储 HDFS 的元数据 (metadata)。

管理文件系统的命名空间 (namespace)。

创建、删除、移动、重命名文件和文件夹。

接收从 DataNode 来的 Heartbeat 和 Blockreport。

□ DataNode

存储数据块。

执行从 NameNode 来的文件操作命令。

定时向 NameNode 发送 Heartbeat 和 Blockreport。

除了提供分布式文件存储外，Hadoop 还提供基于 Map/Reduce 的框架，进行按行的并行分析，可以用来查询和计算。

图 1-6 是以 Word Count 为例子演示 Map/Reduce 机制的图，学习过 Hadoop 的人一般都看到过，有趣的是，我们在 2.1.4 节也会基于 Fourinone 的方式实现一个 Word Count。

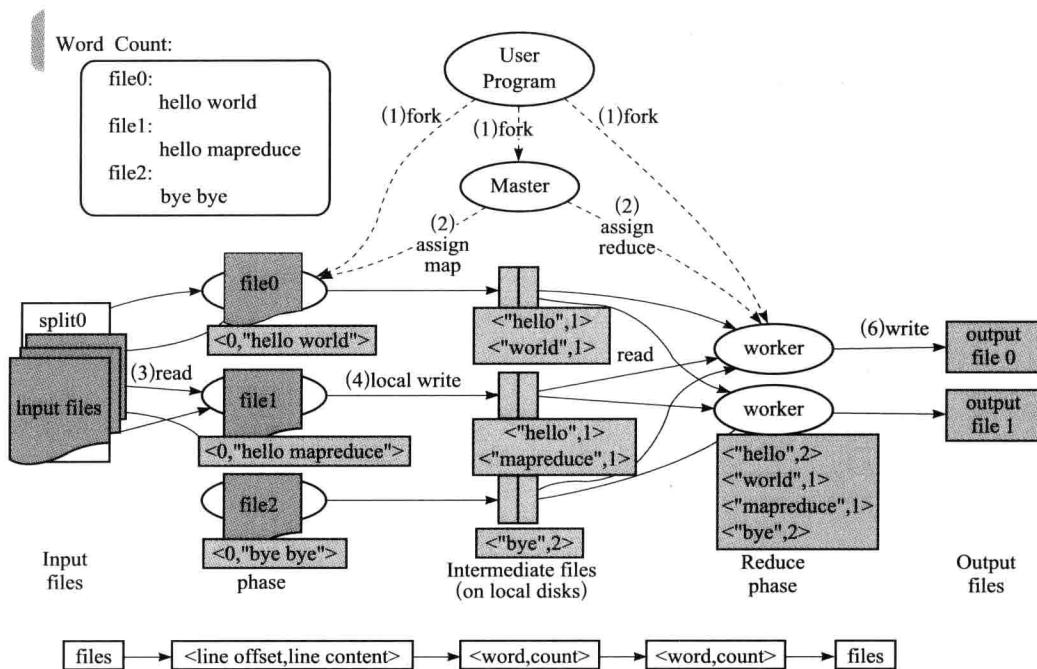


图 1-6 Word Count

图 1-7 中 Hadoop 的 Map/Reduce 实际上是 1.0 版，也许 Hadoop 项目组也意识到该框架的局限性，在目前 Map/Reduce2.0（Yarn）版中实际上已经完全进行了重构，整个设计思想是做成一个资源和任务的调度框架，再也不是 Google 的 Map/Reduce 相关论文阐述的内容了。关于 Yarn 我们在 7.4.2 节也会谈到。

关于 Hadoop 的详细资料可以参考以下信息：

http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html

<http://labs.google.com/papers/gfs.html>

2. ZooKeeper

ZooKeeper 在 Hadoop 生态体系中是作为协同系统出现的，为什么会独立出一个协同系统呢？我们看看跟分布式协同相关的一些重要概念。

- **分布式协同系统：**大型分布式应用通常需要调度器、控制器、协同器等管理任务进程的资源分配和任务调度，为避免大多数应用将协同器嵌入调度控制等实现中，造成系统扩充困难、开发维护成本高，通常将协同器独立出来设计成为通用、可伸缩的协同系统。

- **ZooKeeper：**Hadoop 生态系统的协同实现，提供协调服务，包括分布式锁、统一命

名等。

- Chubby：Google 分布式系统中的协同实现和 ZooKeeper 类似。
- Paxos 算法：1989 年由莱斯利·兰伯特提出，此算法被认为是处理分布式系统消息传递一致性的最好算法。
- 领导者选举：计算机集群中通常需要维持一个领导者的服务器，它负责进行集群管理和调度等，因此集群需要在启动和运行等各个阶段保证有一个领导者提供服务，并且在发生故障和故障恢复后能重新选择领导者。

当前业界分布式协同系统的主要实现有 ZooKeeper 和 Chubby，ZooKeeper 实际上是 Google 的 Chubby 的一个开源实现。ZooKeeper 的配置中心实现更像一个文件系统，文件系统中的所有文件形成一个树形结构，ZooKeeper 维护着这样的树形层次结构，树中的结点称为 znode，每个 znode 存储的数据有小于 1MB 的大小限制。ZooKeeper 提供了几种 znode 类型：临时 znode、持久 znode、顺序 znode 等，用于不同的一致性需求。在 znode 发生变化时，通过“观察”（watch）机制可以让客户端得到通知。可以针对 ZooKeeper 服务的“操作”来设置观察，该服务的其他操作可以触发观察。ZooKeeper 服务的“操作”包括一些对 znode 添加修改获取操作。ZooKeeper 采用一种类似 Paxos 的算法实现领导者选举，以解决集群宕机的一致性和协同保障。总体上说，ZooKeeper 提供了一个分布式协同系统，包括配置维护、名字服务、分布式同步、组服务等功能，并将相关操作接口提供给用户。

ZooKeeper 的结构如图 1-7 所示。

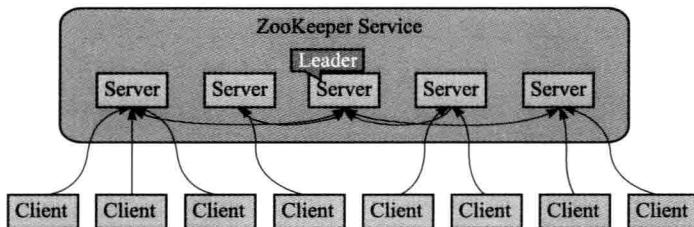


图 1-7 ZooKeeper 的结构示意

ZooKeeper 大致工作过程如下：

1) 启动 ZooKeeper 服务器集群环境后，多个 ZooKeeper 服务器在工作前会选举出一个 Leader，若在接下来的工作中这个被选举出来的 Leader 死了，而剩下的 ZooKeeper 服务器会知道这个 Leader 死掉了，系统会在活着的 ZooKeeper 集群中会继续选出一个 Leader，选举出 Leader 的目的是在分布式的环境中保证数据的一致性。

2) 另外，ZooKeeper 支持 watch(观察) 的概念。客户端可以在每个 znode 结点上设置一个 watch。如果被观察服务端的 znode 结点有变更，那么 watch 就会被触发，这个 watch 所属的客户端将接收到一个通知包被告知结点已经发生变化。若客户端和所连接的 ZooKeeper 服

务器断开连接时，其他客户端也会收到一个通知，也就是说一个 ZooKeeper 服务器端可以服务于多个客户端，当然也可以是多个 ZooKeeper 服务器端服务于多个客户端。

我们这里只是讲述了 ZooKeeper 协同的基本概念，第 3 章我们还会详细讲如何实现这样的协同系统，并与 ZooKeeper 进行一些对比。

3. HBase

HBase 是 NoSQL 技术的产物，NoSQL 风行后，很多互联网应用需要一个面向键 / 值的列存储数据库，并可以支持水平扩充，当然 Google 在这个领域又走在了前面。

HBase 是 Google Bigtable 的开源实现。Google Bigtable 利用 GFS 作为其文件存储系统，HBase 利用 Hadoop HDFS 作为其文件存储系统；Google 运行 MapReduce 来处理 Bigtable 中的海量数据，HBase 同样利用 Hadoop MapReduce 来处理海量数据；Google Bigtable 利用 Chubby 作为协同服务，HBase 利用 ZooKeeper 作为对应的功能。

图 1-8 是 HBase 的架构，里面的 HDFS 也就是 Hadoop 中的分布式文件系统。对里面主要的核心组件简单介绍如下：

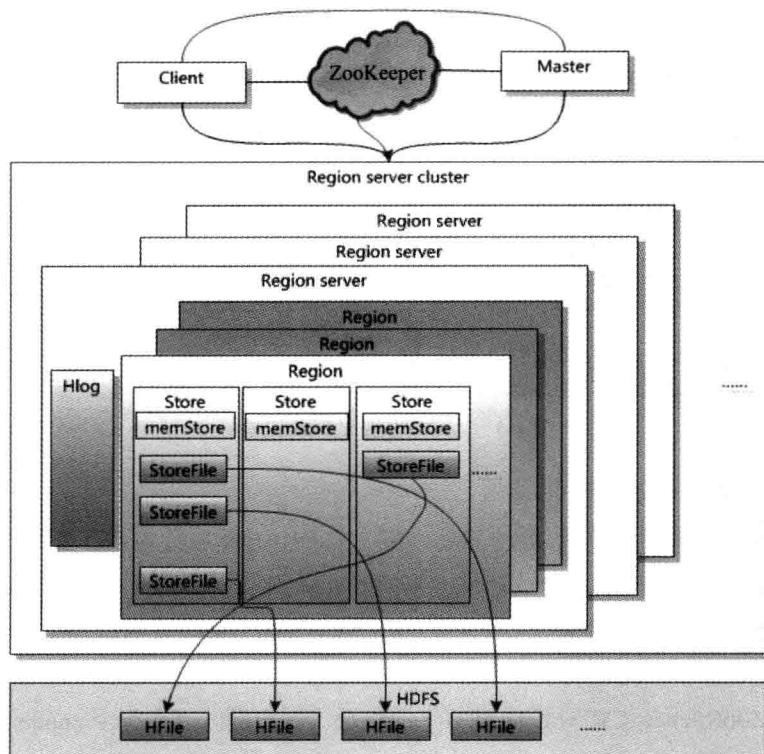


图 1-8 HBase

(1) Client

Client 包含访问 HBase 的接口，维护着一些 cache 来加快对 HBase 的访问，比如 Region 的位置信息。

(2) ZooKeeper

保证任何时候，集群中都只有一个 Master。存储所有 Region 的寻址入口。实时监控 Region server 的状态，将 Region server 的上线和下线信息实时通知给 Master。存储 HBase 的 schema 包括有哪些表，每个表有哪些列。

(3) Master

为 Region server 分配 Region。负责 Region server 的负载均衡。发现失效的 Region server 并重新分配其上的 Region。GFS 上的垃圾文件回收。处理 schema 更新请求。

(4) Region server

Region server 维护 Master 分配给它的 Region，处理对这些 Region 的 IO 请求。Region server 负责切分在运行过程中变得过大的 Region。

我们走马观花般地过了一遍 Hadoop 主要的技术和产品，当然 Hadoop 体系包括的还有很多，我们只是描述了它最主要的部分，图 1-9 是 Hadoop 产品的生态图，有十多个，估计需要工程师们挑灯苦学了。

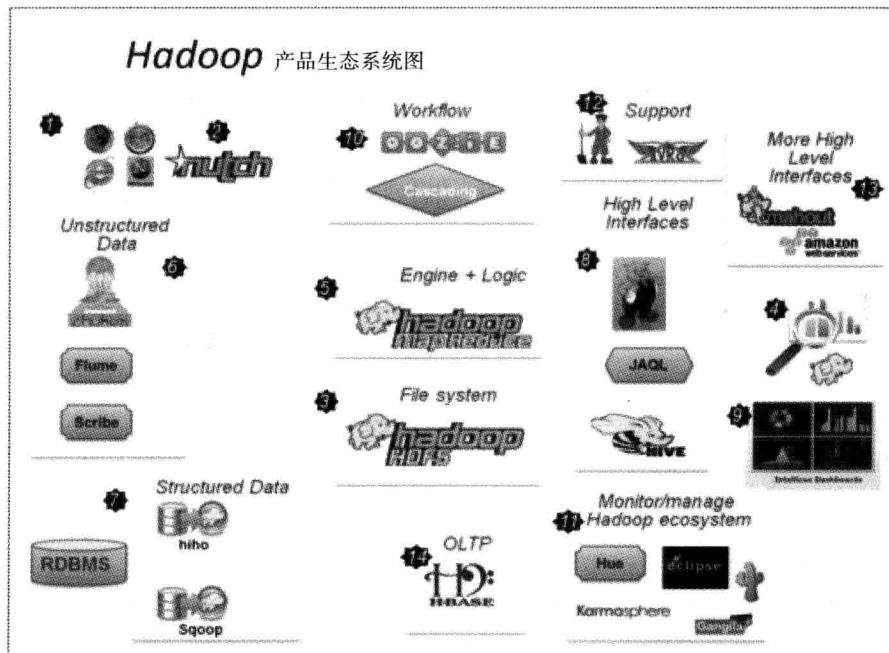


图 1-9 Hadoop 产品生态环境

1.3 Fourinone 的产生背景

1. 使用 Hadoop 时碰到的问题

笔者最开始尝试大数据并行计算分析是为了解决淘宝网的秒杀作弊问题。秒杀曾经是最成功的电商促销活动，往往在短时间内造成平时很多倍的销量，当然这也给系统造成很大的压力，但是这些都不是问题，关键是出现了像蟑螂一样的东西，怎么都灭不掉，那就是秒杀器。

当市场上不断出现秒杀器后，想用手工秒杀商品越来越难了，秒杀活动几乎全部被秒杀器软件垄断。于是买家开始大量抱怨不想参加秒杀了，卖家也不想发货给使用秒杀器作弊的人，甚至社会上流言淘宝网搞虚假秒杀，严重扰乱了市场秩序。

虽然技术人员也尝试屏蔽秒杀器作弊，但是一直收效甚微，比如尝试把验证码设计得很复杂，比如多加一些验证参数。写过秒杀器软件的人都明白，这样的防范把自己折磨得很辛苦，但是对秒杀器是不起作用的，因为秒杀器实际上是一个复杂的模拟 HTTP 交互的网络通信软件，它反复跟网站服务器交互 HTTP 报文，如果发现报文参数不够，会不断变更报文直到通过，因此只要秒杀器写得足够好，理论上就无法屏蔽它。

因此，唯一的办法是要深入分析 HTTP 报文本身的差异性。秒杀器和浏览器制造的报文是有区别的，这个区别或者大或者小，都是可以通过技术手段区分出来的，这是一个识别模型。由于每日大型、小型秒杀的订单数量太大，为了在短时间内识别出每个订单报文是否作弊，我们考虑用并行计算多机完成，为了完成这个技术任务，我们首先想到了用 Hadoop。

按照 Hadoop Map/Reduce 框架的开发步骤，我们要实现 Map 和 Reduce 接口，取出每行，执行一段逻辑，打包好，启动 jobtracker……我们学会了按照这个步骤开发后，尝试套用 Map/Reduce 框架会发现一些的问题，比如：

- 1) HTTP 协议报文一个请求占多行，各行之前有一定逻辑关系，不能简单以行拆分和合并。
- 2) 复杂的中间过程的计算套用 Map/Reduce 不容易构思，如大数据的组合或者迭代，多机计算并不只有拆分合并的需求（请参考 2.1.11 节）。
- 3) Hadoop 实现得太复杂，API 枯燥难懂，不利于程序员迅速上手并驾驭。
- 4) Map/Reduce 容易将逻辑思维框住，业务逻辑不连贯，容易让程序员在使用过程中总是花大量时间去搞懂框架本身的实现。
- 5) 在一台机器上未能很直接看出并行计算优势。
- 6) 比较难直观设计每台计算机干哪些事情，每台计算机部署哪些程序，只能按照开发步骤学习，按照提供方式运行。
- 7) 没有一个简单易用的 Windows 版，需要模仿 Linux 环境，安装配置复杂。

2. 抽取一个简化的并行计算框架

我们从秒杀作弊分析延伸出了想法，提取和归纳了分布式核心技术，建立了一个简化的并行计算框架用于业务场景需要。我们的思路有以下几点：

- 1) 对并行计算 Map/Reduce 和 forkjoin 等思想进行对比和研究。
- 2) 侧重于小型灵活的、对业务逻辑进行并行编程的框架，基于该框架可以简单迅速建立并行的处理方式，较大程度提高计算效率。可以在本地使用，也可以在多机使用，使用公共内存和脚本语言配置，能嵌入式使用。
- 3) 可以并行高效进行大数据和逻辑复杂的运算，并对外提供调用服务，所有的业务逻辑由业务开发方提供，未来并行计算平台上可以高效支持秒杀器作弊分析、炒信软件分析、评价数据分析、账务结算系统等业务逻辑运算。

4) 可以试图做得更通用化，满足大部分并行计算需求。

下一章我们顺着这个思路深入解析并行计算模式的设计过程。

第 2 章

分布式并行计算的原理与实践

本章首先讲述分布式并行计算的各种设计模式和原理机制，并进一步说明 Fourinone 与市场上其他并行计算产品和技术的区别，最后手把手演示大量并行计算案例。

本章讲述的分布式并行计算思想、模式、技巧、实现，与编程语言无关，读者可以用多台计算机来尝试编程。

2.1 分布式并行计算模式

2.1.1 最初想到的 master-slave 结构

当我们最初构思使用几台计算机去设计一个分布式并行计算系统时，很自然就会想到 master-slave（m-s）的结构，由一台计算机作为主调度者，然后几台计算机根据调度完成任务，如图 2-1 所示。

实际上，这个最简单的并行计算结构目前仍然存在于很多分布式系统中，master 通常实现为一个 SocketServer，slave 与之保持心跳，并且互相通信传送任务，最后由 master 汇集结果。这个结构一目了然，容易被 C/C++ 或者 Java 实现。或者另外一种变体就是在一台计算机上的多线程实现，由线程共享变量代替通信，多线程实现的 master-slave 结构也广泛存在于各种应用中，对于经验并不多的工程师，也能自己写一个多线程实现的 master-slave 一展身手。

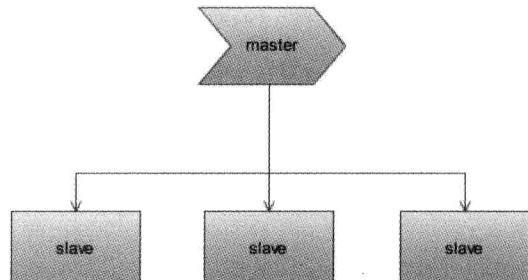


图 2-1 master-slave 结构

但是实践的时间长点，我们会发现这个最简单的并行计算结构存在一些缺陷。首先，各个 slave 需要获取 master 的存在并连接它，master 必须做为一个服务程序存在，它跟 slave 之间是一种紧密耦合的连接状态，master 必须一直存在于集群中，它虽然孤立，但是它是中心领导，它太重要了，因此不能有任何问题。

另外，我们观察 master 的职责，它除了分配任务给 slave 执行外，还承担着负责协同一致性等角色，比如它要接受 slave 的注册，如果 slave 死掉，需要感知等等，承担的责任太多了。

因此 master 最好能将协同部分的职责分离出来，它只负责任务调度部分，为了减少故障，它最好不要做为服务程序一直存在。

2.1.2 “包工头 – 职介所 – 手工仓库 – 工人” 模式

30 年的改革开放，中国经济总量已经排到了世界第二的位置上，有部分原因是农民工在加工生产行业几十年辛勤的劳动和努力，是他们使中国经济获得了今天的地位。

为了表示对中国农民工的敬意，Fourinone 提出一种简化的分布式并行计算的设计模型，模仿现实中生产加工链式加并行处理的“包工头 / 农民工 / 手工仓库 / 职介所”方式设计分布式计算，如图 2-2 所示。

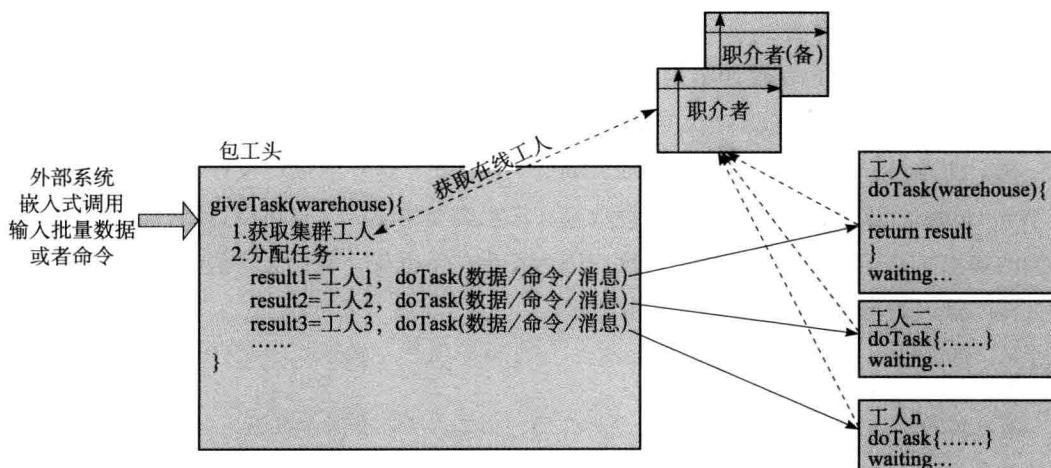


图 2-2 “包工头 – 职介所 – 手工仓库 – 工人” 模式

这个图能帮助认识整个计算结构，并产生一个总体印象：工头的 `giveTask` 方法可以由开发者实现，工头先获取线上工人数量，然后调用各个工人的 `doTask` 方法，让工人们并行完成任务，工人的 `doTask` 方法也可以由开发者实现……

就是这么一个简化的框架思想，初步理解了它，我们再进一步看看各个角色的详细含义：

“职介所”(ParkServer)可以部署在一台独立计算机，它在每次分布式计算时给“包工头”介绍“工人”，当“工人”加入集群时首先在“职介所”进行登记，然后“包工头”去“职介所”获取可用于计算的“工人”，然后“职介所”会继续和“工人”保持松散的联系，以便在有新的工程时继续介绍该“工人”。

“工人”(MigrantWorker)为一个计算节点，可以部署在多个机器，它的业务逻辑由开发者自由实现，计算时，“工人”到“输入仓库”获取“包工头”分配的输入资源，再将计算结果放回“输出仓库”返回给“包工头”。

“包工头”(Contractor)负责承包一个复杂项目的一部分，可以理解为一个分配任务和调度程序，它的业务逻辑由开发者自己实现，开发者可以自由控制调度过程，比如按照“工人”的数量将源数据切分成多少份，然后远程分配给“工人”节点进行计算处理，它处理完的中间结果数据不限制保存在HDFS里，而可以自由控制保存在分布式缓存、数据库、分布式文件里。但是通常情况下，输入数据多半分布在“工人”机器上，这样有利于“工人”本地进行计算处理和生成结果，避免通过“包工头”进行网络传送耗用。“包工头”实际上是一个任务的调度者，它通过getWaitingWorkers方法获取线上工人并行调度任务执行，“包工头”是一个并行计算应用的程序入口，它不是一个服务程序，运行完成就退出。

“手工仓库”(WareHouse)为输入输出设计(参见后面的图2-4)，让计算的资源独立于计算的角色(包工头，工人)，包工头和工人的数据交互都是通过手工仓库，它可以当做远程请求的参数，同时也是返回结果的对象。“手工仓库”可以存放任意类型的对象，它本身就是一个map的实现。

我们注意WareHouse doTask(WareHouse inhouse)这样的接口设计，它的输入输出都是WareHouse，意味着它可以接受任意数量、任意类型的输入，也可以返回任意数量、任意类型的输出，这样能做到最大灵活程度，因为作为一个框架，我们无法假定开发用户设计出什么样的输入输出参数，以及什么样的类型。

注意

有人容易认为，包工头是将数据发到职介所，由工人去职介所拿数据(当然这种基于消息中枢的计算方式也是可以支持，下个章节会介绍)，但是图2-2展示的默认最常用的计算模型，是包工头直接跟工人交互的。也就是职介所只起介绍作用，介绍给工头后就不干涉他们之间分配任务干活了(因此计算过程中，职介所挂掉也不会影响工头/工人完成计算，这对计算结构的健壮性来说很重要)，但是介绍后职介所还会跟工人保持一种松散的联系，询问工人工作的怎么样，联系方式是否变了，等等。联系是松散的意味着不影响工人工作，可以半个月1个月跟职介所联系一次，时间可以自由设置，但是不能中断联系。保持松散联系的目的是为了在下一次有工头来

职介所找工人时，是否继续介绍该工人，如果该工人死机，跟职介所已经失去了联系，就不能再介绍给工头。“包工头 - 职介所 - 工人”整个作业过程跟我们现实社会中的场景完全吻合，这样设计有利于更直观形象地理解这背后的并行计算思想。

最后我们再归纳一下各个角色职责：

- 包工头负责分配任务，开发者实现分配任务接口。
- 农民工负责执行任务，开发者实现任务执行接口。
- 职介所负责协同一致性等处理（登记，介绍，保持联系）。
- 手工仓库负责输入输出数据交换。

2.1.3 基于消息中枢的计算模式

由于 Fourinone 提供简单 MQ 的支持，因此包工头和工人之间的交互也可以以消息中枢的模式进行，包工头将需要工人执行的任务 / 命令 / 消息发送到消息中枢，工人监听消息中枢上各自的消息队列，一旦自己的队列有新内容，便接收执行任务，各个工人以并行的方式进行，如图 2-3 所示。

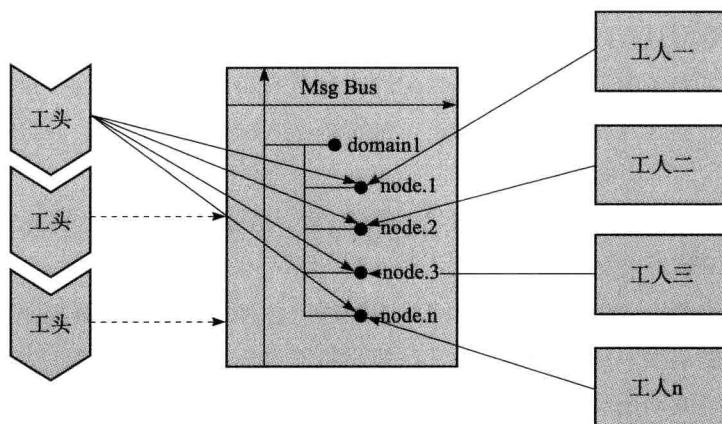


图 2-3 消息中枢模式

如果要支持消息中枢模式，在 Fourinone 里只需要修改一个配置即可，在 config.xml 配置文件里：

```
<PROPSROW DESC="COMPUTEMODE">
<MODE DESC="DEFAULT">0</MODE>
<MODE>1</MODE>
</PROPSROW>
```

可以看到默认的计算模式选项为 0，代表工头工人直接交互模式（下章会介绍），如果将

默认改为 1，便是消息中枢模式。



提示

对于开发者来说，实现程序上一行代码都不用改动，`doTask` 和 `giveTask` 仍然按照实现的内容执行，只不过它们背后的交互机制完全不同，模式 0 是直接交互，模式 1 是把任务发到消息中枢上交互。

消息中枢模式的优点：最直接的优点就是包工头机器和工人机器可以互相不可见，可以互相网络不通，只要它们都跟消息中枢机器能连接即可。比如我们一个网吧或者一个公司的机器，可以访问外面的互联网服务器，但是互联网服务器却不能直接连接到局域网网关内的每台机器，那么使用模式 0 的方式肯定连接不了，使用模式 1 消息中枢是最好的方式，因为如果包工头在另外一个局域网网络里，那么公共的互联网服务器是大家都可以连接的，但是包工头和工人却无法直接连接。比如每日访问淘宝网站的有几千万用户 PC 机，他们可以直接连接淘宝服务器，但是反过来淘宝的工头程序无法连接到每台用户 PC，如果能使用消息中枢模式利用用户这几千台 PC 做一次并行计算，将产生一个巨大的计算能力。

消息中枢模式的缺点：所有的任务发送到消息中枢来实现异步传递，消息中枢本身很容易形成瓶颈，因为它只有一台机器，容量有限，也容易形成故障单点，很明显，太大数据的任务或者太频繁、太多的消息请求都对它的性能有严重影响，它只适合数量不多连续性质的增量消息，但是不适合各种并行计算场景，特别是输入输出数据庞大，并且耗费网络带宽很大的计算，比如我们下面的 2.3 节还会谈及 MPI 方式的并行计算。

所有的任务通过消息中枢中转而不是直接交互，数据多了一个中转环节，经过我们测试，显然速度上不如直接交互的计算快。

消息中枢模式意味着包工头无法直接对工人进行各种复杂点的计算控制，因为这两者隔离开来了，所以框架很多高级功能支持（比如避免工人任务重复调用、工人任务中止或者超时中止、工人网络波动抢救期设置等等）在消息中枢模式下使用不了，以及我们下一节会谈到工人和工人之间传递数据也使用不了。消息中枢模式只能完成最简单基本的计算。

因此，框架默认和推荐的方式是直接交互模式，该模式几乎能完成所有的分布式并行计算需求了，只有工头 / 工人无法直接连接的网络环境才考虑使用消息中枢模式。

2.1.4 基于网状直接交互的计算模式

“包工头 - 职介所 - 手工仓库 - 工人”的基本结构和角色已经在前面介绍过了，这里重点介绍计算过程中的工头和工人的交互机制。先看看框架默认和推荐的网状直接交互模式。

前面我们基本了解到工头通过在 `giveTask` 实现中，调用工人的 `doTask` 来分配任务，并

轮询检查 doTask 结果是否完成，如果选择网状直接交互模式，也就是默认的设置：

```
<MODE DESC="DEFAULT">0</MODE>
```

那么这个调用过程是直接调用，而不会通过消息中枢中转，也就是如果计算过程中，把 parkserver 关掉，对计算也没有影响，反之则不行。

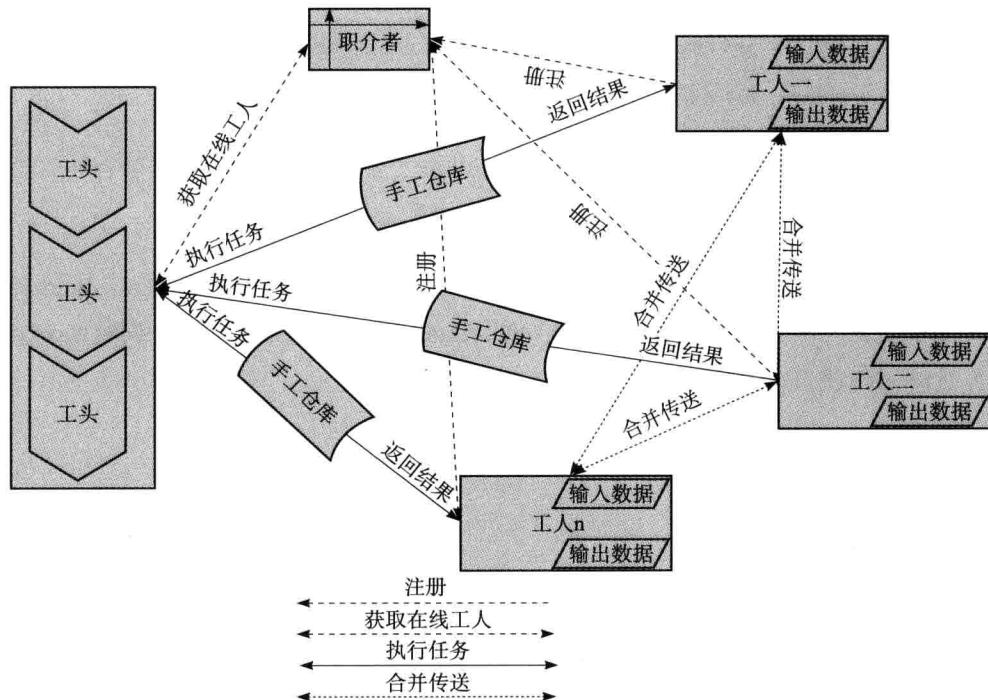


图 2-4 网状直接交互模式

通过图 2-4 我们观察到一个重要的特性，就是工人之间也是可以直接交互的。这个机制主要使用在并行计算过程中的合并，各个工人在计算过程中互相进行数据合并。在每个工人的 doTask 实现里，框架提供了一系列 API 帮助获取到集群中其他工人集合，如下所示：

```
// 获取除该工人外其他所有相同类型的工人
protected abstract Workman[] getWorkerElse();

// 获取其他某个相同类型的工人, index 为在集群中序号
protected abstract Workman getWorkerIndex(int index);

// 获取除该工人外其他所有工人, workerType 为工人类型
protected abstract Workman[] getWorkerElse(String workerType);

// 获取其他某个工人, workerType 为工人类型, index 为在集群中序号
protected abstract Workman getWorkerIndex(String workerType, int index);
```

```

// 获取某台机器上的工人, workerType 为工人类型, host 为 ip, port 为端口
protected abstract Workman getWorkerElse(String workerType, String host, int
port);

// 获取集群中所有工人, 包括该工人自己
protected abstract Workman[] getWorkerAll();

// 获取集群中所有类型为 workerType 的工人
protected abstract Workman[] getWorkerAll(String workerType);

// 获取自己在集群中的位置序号
protected abstract int getSelfIndex();

// 接收来自其他工人的发送内容
protected abstract boolean receive(WareHouse inhouse);

```

上面 API 中获取集群其他工人返回结果是 Workman，代表该名其他工人，Workman 有一个 receive 方法可供调用，通常调用该工人的 receive 方法向它发送数据，同时每个工人都需要实现 receive 接口，工人之间的整个交互过程如图 2-5 所示。

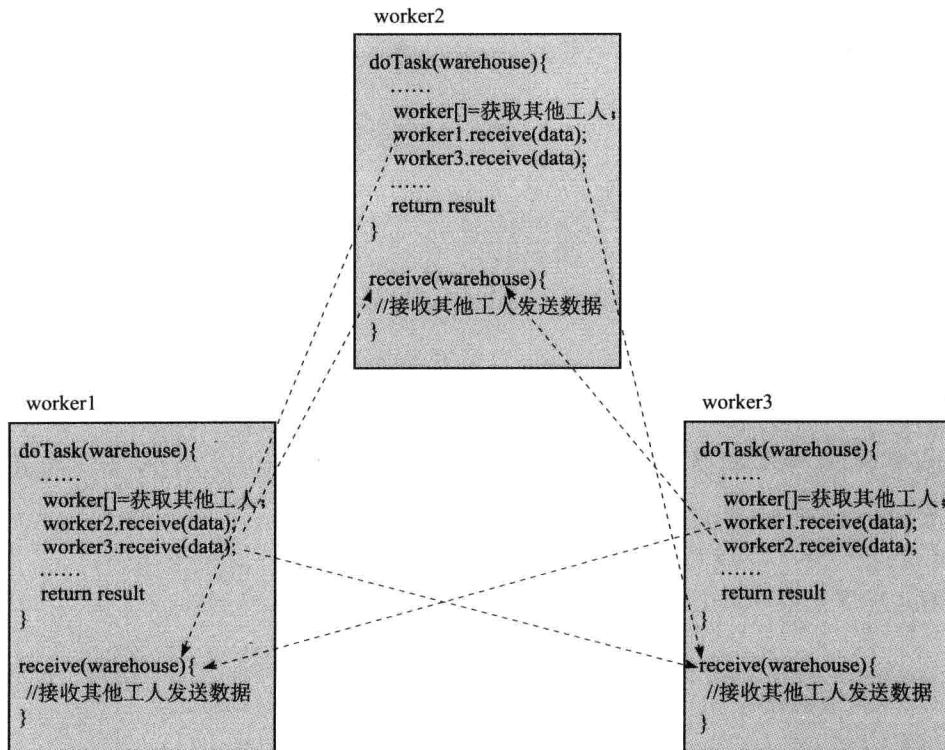


图 2-5 工人间互相交互

图中示例了 3 个工人间的交互过程：

- 1) 包工头会并发地调用 3 个工人的 doTask，让他们完成任务；
- 2) 每个工人在计算过程中，如果需要跟其他工人交互，应先获取其他工人集合；
- 3) 依次调用其他工人的 receive 方法，将数据发送给该工人；
- 4) 在各自的 receive 方法实现中，接收其他工人发送过来的数据；
- 5) 如果所有工人的 doTask 调用完成了，那么所有工人的 receive 接收也完成了。



注意

对比前面的介绍，我们发现这里工人获取其他工人并且交互都是独立一套 API，为什么不直接使用包工头的 getWaitingWorkers 获取工人集合，再调用每个工人的 doTask 传递数据呢？如果工头和工人共用一套交互机制，很容易出现工头工人都在等待 doTask 执行完成，会产生死锁问题（可以参考银行家算法），下面谈及跟 MPI 的区别时也会提到，因此框架从设计上就避免了开发者使用产生死锁问题，分别用独立的 API 将“工头 - 工人”、“工人 - 工人”的交互隔离开来。这样让开发者可以轻松设计并行计算而完全不用考虑死锁等复杂问题，这些让框架去考虑，开发者专注在实现计算逻辑本身上即可。

我们总结一下并行计算过程中数据和计算的关系：

比如 Hadoop 的方式，是将计算 jar 包发到数据节点上执行，计算向数据移动；

还有的实时计算 Storm，是将数据通过消息发到节点上计算，数据向计算移动。

一般这类计算平台软件会将数据和计算的关系固化下来，开发者无法自己决定，必须按照计算平台规范开发，上传 job 进行执行。

但是 Fourinone 的数据和计算的关系相当灵活，可以根据需求自由设计，几乎不受限制，如下所示：

- 如果数据小，可以由工头分配任务时放在 Warehouse 里直接传给工人。
- 如果数据大，可以直接保存在工人机器上，Warehouse 里只放工头的命令，工人收到命令后直接在本地读取数据计算；或者工头把数据地址、数据库表连接信息等发给工人用于计算，但是不传数据。
- 如果计算结果小，各工人可以直接返回结果给工头汇总。
- 如果计算结果大，各工人直接存放本地，返回完成状态告诉工头。
- 如果中间结果小，可以返回工头合并再做为条件重新安排任务。
- 如果中间结果大，可以多次通过 receive 方式跟其他工人合并。
- 怎么做完全取决于开发者的需求和设计。

2.1.5 并行结合串行模式

我们现实中很多需求不仅需要并行方式加快执行，很多时候还必须遵循串行的先后次序执行。

比如生产 PC 机，必须要把主板、硬盘、CPU、内存等配件生产齐全后，才能进行组装，如果其中一个配件未生产齐全，无法完成后续组装。但是生产主板、硬盘、CPU、内存每一个配件的过程又都是并行的，现实生活中电子工厂生产线上大量的工人并行作业，同时生产各种配件。

因此，我们需要框架支持“并行 + 串行”的模式才能满足更多的需求，如图 2-6 所示。

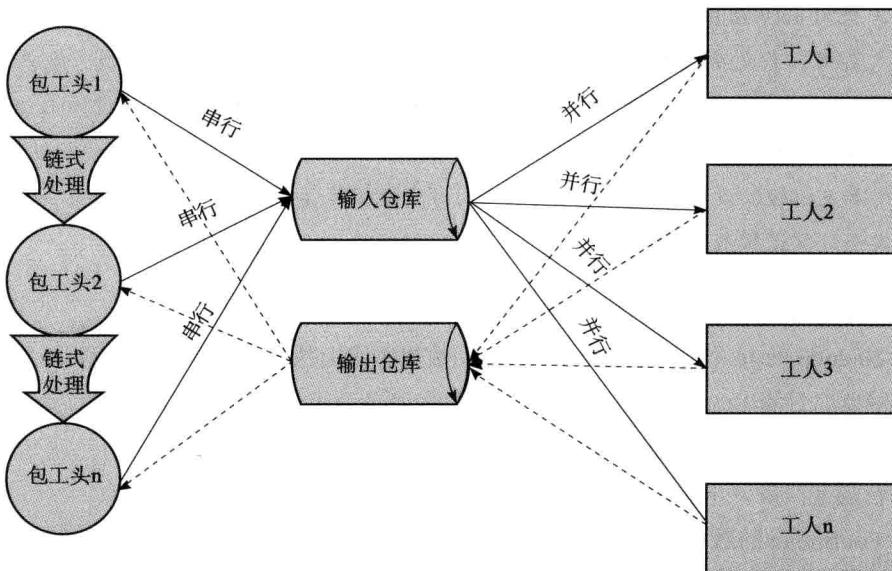


图 2-6 并行结合串行模式

我们可以看到，多个包工头可以衔接在一起，以加工链式的方式向下执行，同时每一个链环节又可以让多个工人并行完成。这样做好处有很多：

- 我们知道分布式并行计算多用于解决数据庞大计算复杂的应用，那么这个计算过程最好能够拆分开来，不光是将计算数据拆分到不同计算机，计算过程也能拆分为多个环节去做，工业界叫做“工艺流程”，如果将一个复杂漫长的计算过程混在一起并不是一个好的做法。
- 计算过程混在一起没有环节，首先比较难监控和处理异常，如果把包工头划分为多个环节很容易对计算过程进行监控，出现异常容易锁定环节，我们可以知道一个时间长的计算任务目前运行到什么环节了，问题出在哪个环节，每个环节完成还可以发送消

息事件进行相关协作。

- 对照环节的切分思想，我们再看看如何满足 Map/Reduce 计算，那么实际上 Map 是一个环节，Reduce 是第二个环节，如果是更灵活的需求，比如 Map/Reduce/Map/Reduce…，或者 Map/Reduce/Reduce…。如果我们按照 Map/Reduce 的方式套就很难设计了，但是转换为一到多个链式环节去处理就很容易。

因此，包工头实现链式的多环节处理，有利于我们将复杂计算进行拆分，并且可以深入控制计算过程，通过能构思整个计算“工艺流程”的设计，能保证业务逻辑的完整性，而不至于将业务分散到框架的各个 Map 或者 Reduce 接口中，拆分得支离破碎，难以理解整个计算过程设计。

2.1.6 包工头内部批量多阶段处理模式

通过上一节，我们了解到包工头之间的链式多环节处理，但是由于多个包工头通常是由多个类角色实现的，代表不同的调度角色和任务，那么这是一种粗粒度的环节划分，是否在一个包工头里面也是可以进行细粒度的划分呢？

Fourinone 框架提供了包工头内部的批量处理，批量处理就是由多个工人完成各自的任务，必须等到最慢的一个工人做完，才统一返回结果。我们可看到批量处理 doTaskBatch 方法的定义（详见框架源码）：

```
WareHouse[] doTaskBatch(WorkerLocal[] wks, WareHouse wh)
```

输入参数：WorkerLocal[] 是集群工人集合，WareHouse 是任务。

输出参数：WareHouse[] 是工人计算结果集合。

doTaskBatch 方法的含义也就是让 WorkerLocal[] 中每个工人执行 WareHouse 任务，然后返回一个结果集 WareHouse[]，其序号对应 WorkerLocal[]，代表每个工人的完成结果。

doTaskBatch 是一个堵塞方法，直到 WorkerLocal[] 中所有工人计算完成，才返回最后的结果集。

因此，在包工头内部的逻辑设计中，我们可以通过多个 doTaskBatch 批量处理，将 giveTask 划分为多个细粒度的阶段处理，比如多个步骤。如图 2-7 所示。

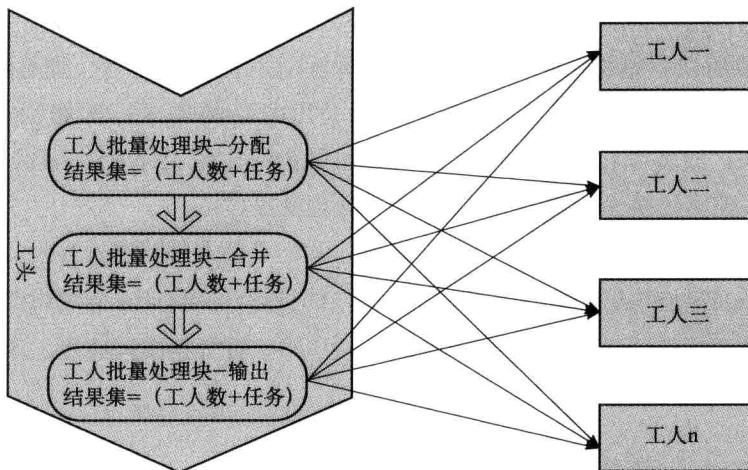


图 2-7 内部批量多阶段处理模式

所以，现在我们通过包工头的链式处理和内部的批量处理，可以粗粒度或细粒度地设计并行计算过程中的环节和阶段划分，这样能更灵活地满足不同层面的计算过程划分需求。在下面的章节中我们还会谈到业界的 BSP 等并行计算方式，其核心思想也就是对计算过程环节划分的功能支持，但是没有粗细粒度的考虑。

2.1.7 计算集群模式和兼容遗留计算系统

从上面我们知道多个包工头外部可以分环节处理，虽然每个包工头可以调用多个工人并行处理，但是多个包工头之间是一种串式处理，我们能否让多个包工头之间也并行处理呢？比如一个包工头带领几个工人做饭，另外一个包工头带领几个工人炒菜，做饭和炒菜不需要先后顺序，可以并行进行。

曾经收到一些开发者的反馈，说框架应该增加一个总工头角色，总工头可以给多个工头分配任务，再由每个工头给多个工人分配任务，最后所有工人和工头都是并行作业。

我们顺着这个思路完善，如果总工头分配了任务，各个工头完成后，需要任务合并怎么办，都汇集到总工头这个角色来合并吗，这样对于数据大的结果并不妥，各工头之间能否合并呢，如果要支持，那么是否跟工人之间的合并机制类似了呢，能否避免重复设计？

另外，随着计算集群的扩充，总工头负责的计算任务的范围也需要扩大，能否不改变设计便集成到另外一个包括它的计算集群中去，那么是否还需要总 - 总工头，总工头上面又总工头，总工头和总工头之间是否也要考虑增加合并机制呢？

这样设计会越来越复杂和臃肿。

实际上灵活运用工头工人组合机制就可以解决这个问题，组成计算集群模式不需要额外

设计其他角色。如图 2-8 所示。

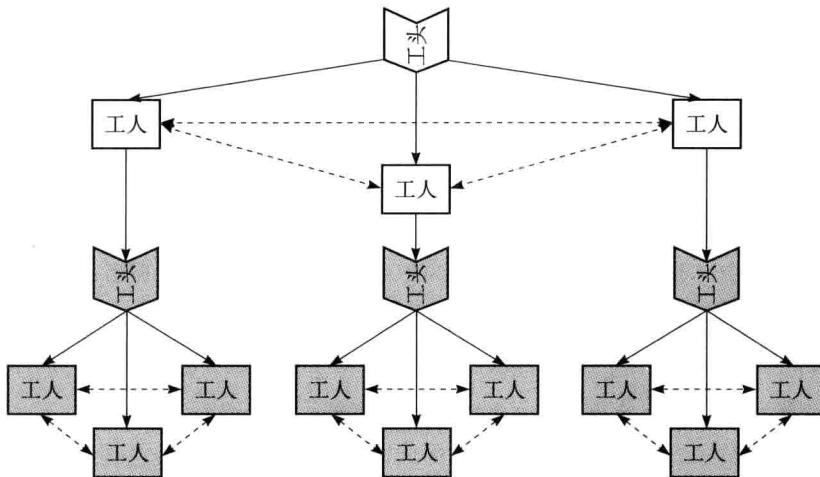


图 2-8 计算集群模式

我们通过图 2-8 可以看到：这里总共示范了 4 组“工头 - 工人”计算单元，它们之间是有层级的，下面阴影部分的 3 组计算单元是任务执行者，上面白色部分相当于一个“总工头”角色，我们发现它是通过上层工人调用下层工头方式集成其他并行计算单元的，在工人的 doTask 实现内部，新增一个其他计算单元的工头类，管理该工头的生命周期，并且等待该工头完成并行计算任务后，获取结果，这个中间结果可以返回给最上层的工头汇总，也可以上层工人之间再进行合并。

通过这样的计算单元组件衔接的方式，我们可以横向扩充更多的“工头 - 工人”计算单元进来，也可以纵向延升到更高级的工头，工头上面集成更多的工头，组成一个大的计算集群。

同时，每一个并行计算单元并不一定都是同一时期开发的，也不一定是同一套 Java 技术或者基于 Fourinone 计算框架开发的，它可以是 MPI 的，也可以是其他一个多台计算机的计算集群，我们可以在工人实现里面通过脚本调用方式启动它（详见下面章节 2.5.9 节和 2.5.10 节），这样就能实现新旧系统、将不同技术平台的历史遗留系统统一集成。

为了深刻理解该并行计算集群结构，我们联想一下现实加工生产行业的类似情形：

假设我们是苹果公司，我们将手机生产组装的任务外包给中国的企业做，假设富士坑组装手机壳、比阿迪生产摄像头、还有一些小公司组装其他配件，苹果公司内部跟这些外包公司对接的都是各个部门，这些部门的负责人就好比是上层的工人，他们直接指挥下层公司的老板——计算单元的包工头，每个生产外包公司的老板分配任务让各自工厂成千上万的工人生产。每个生产外包公司都在并行生产各种配件，每个公司的工人都在并行生产某种配件，

整体就是一个大的并行计算集群，最后汇总到苹果公司形成最终的手机产品。

几乎完全吻合我们现实社会中的生产加工原材料半成品承包模式。

实际上在上层工人调用下层工头时，可以根据一些条件流向的，比如满足 a 条件，分配给 A 工头，满足 b 条件，分配给 B 工头，最后形成一个作业流，我们在后面 2.3 节讲 DAG 时，也会谈到这个问题。

2.1.8 工人计算的服务化模式

在分布式应用中，非常基础也非常广泛的一项应用就是服务化，服务是 SOA 架构中最核心的单元，可以说所有的 SOA 都是服务的组成和上层运用。

什么是服务，大家了解到 WebService 是服务，MQ 也可以看做一个服务，Http 可以看做服务，Socket 服务端也可以看做服务，Ftp 也可以做服务，等等。

我们听说过这么多的技术被叫做服务，那么如何来归纳服务的概念呢？

最简陋的理解：服务是一个监听程序，它说白了就是提供服务器在某个端口监听，并提供根据业务定义的输入输出数据格式交换接口。

因此，我们写一个 Socket 服务程序，它一直监听等待，它只接受某种格式的消息头，并返回某种格式的数据，那么这就是一个原始简陋的服务。

当然有更标准的服务 WebService，它提供一个基于 XML 的 WSDL 服务定义，无论什么样的程序语言平台，我们都遵循这个 XML 输入输出标准，如何查看这个服务标准呢，访问它的 WSDL 文件，然后各自的程序语言平台去实现各自语言数据格式到 XML 数据格式的转换，分布式通信交互都是通过 XML 的消息方式进行，WSDL 严格定义了服务的内容和数据格式，而且形成标准。

不过，我发现很多公司并不喜欢 WebService 的标准服务（我怀疑可能开发有点繁琐，因此工程师难上手而回避它，好像更喜欢 restful 方式的简单服务），而更多是自己实现一个类似上面说的 Socket 原始简陋的服务，当然他们的实现技术千差万别，一般不会基于底层写 TCP 通信，而会借助 MINA、NETTY 这样的通信组件解决通信交互问题，然后再自己做点序列化的工作，再集成到 Spring 等 Bean 管理容器中。最后完成的这个服务产品肯定没有 WSDL 的标准定义，在其之上也比较难做 ESB 等标准服务编排等 SOA 架构，它更多的是用在公司内部的服务化需求中，而很少用在公司对外的服务集成中。

实际上在金融行业，银行保险公司对外提供的服务交换接口中，银监会有着严格的服务标准定义，比如银保通定义了银行和保险公司数据交换的详细格式和内容，体现在他们系统对外的服务接口上，必须严格遵循该项标准。否则金融行业对外的系统交互需求特别多，如果各个公司都遵循自己的服务标准，几乎建立不了公共标准的集成应用。

我们理解了一些服务的概念，再回头来看看 Fourinone 做为一个分布式框架，对服务性质的功能的支持，那么可以肯定的是提供不了 WebService 的标准跨平台服务，因为框架的侧重点不是在服务化的深入和标准上，但是由于分布式技术与服务的相似性，可以实现上面理解的最原始简陋的服务。



提示

实际上工人都是一个个的服务，doTask 是一个通用的服务接口。只不过通常在并行计算中工人只对一个工头提供服务，默认不允许多个工头对它调用，它们默认并不是一个广泛意义的服务，但是我们可以通过修改配置文件将工人改成一个广泛的服务。

在 config.xml 配置文件的工人模块：

```
<PROPSROW DESC="WORKER">
    <SERVICE>false</SERVICE>
</PROPSROW>
```

我们可以看到 SERVICE 项的默认配置是 FALSE，将它改为 TRUE，那么工人变成一个服务概念，由于服务需要支持多个客户端同时访问，每个工人服务可以接受多个工头调用，工头在这里变成一个服务客户端概念，WareHouse 在这里变成了一个输入输出的格式定义，因为 WareHouse 是一个通用的 MAP，它什么都可以装，也什么都可以返回。最后我们的通用服务接口看上去像这样：

```
Map outputdata doService(Map inputdata)
```

只不过这里名字不叫 doService，而叫 doTask。

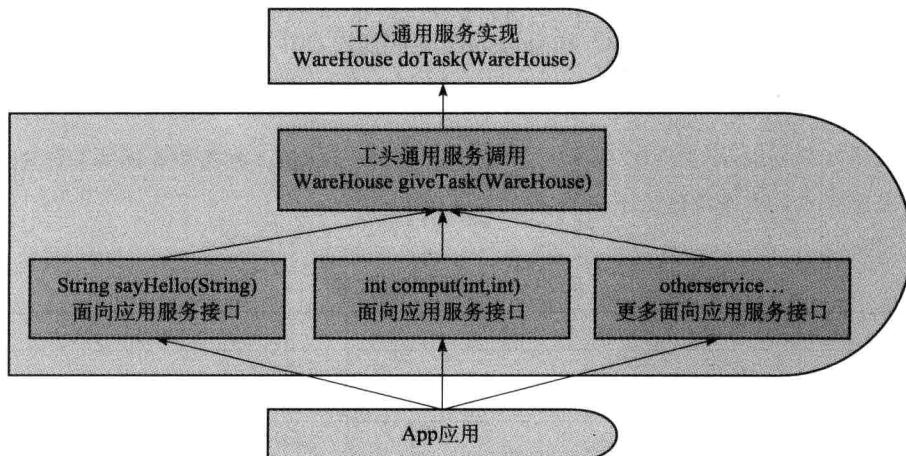


图 2-9 工人计算服务化模式

但是我们面向用户的时候多半不是上面通用服务接口的样子，而是有业务含义的特定服务接口，比如像下面这个样子：

```
String sayHello(String name)
```

这个特定服务输入一个字符串参数 `name`，根据逻辑返回另外一个字符串结果。按照我们设计软件的通常做法，一般会将背后原始基础的通用服务隐藏起来，把面向用户的接口暴露给它们，而在用户接口的实现里面调用原始的通用基础服务 `doTask`。在下面 2.5.1 节的例子里我们就可以看到。

另外还有个地方要注意，使用工头获取线上工人服务需要输入一个类型然后得到一个集合，这实际上意味着我们可以让工人指定提供某种类型的服务，并且可以有多个工人提供这种类型的服务，换句话说，提供某种服务的地址可以有多个，比如 3 个服务器都可以提供 `hello` 服务，我们可以负载均衡的访问他们其中一个，这恰恰是大型分布式服务系统所需要的。

根据之前我们了解的 `doTask` 的调用，在并行计算环境下需要轮询结果，如果是服务化运用，包工头实际上是在提供客户端的功能，`doTask` 实际上是服务调用，那么我们可以异步方式调用，轮询结果；也可以同步调用，使用前面讲的 `doTaskBatch`，等待结果完成后再返回。

工人服务化详细实践和示例请参见 2.5.16 节。

2.2 跟 Hadoop 的区别

对于初学者，特别是学习过一些 Hadoop 知识的读者，可能阅读到这里会产生疑惑，本书描述的分布式核心技术跟 Hadoop 之间的到底有哪些区别。

在表 2-1 中，我们全面列举两者之间的区别，但是仅供参考，不是对 Hadoop 产品持否认态度，我们一贯尊重 Hadoop 作者和 Hadoop 的使用者。

注意

该对比是在 Fourinone-1.11.09 版本，在 2.0 版以后的 Fourinone 打包部署提供了自动部署，并且增加了完整分布式文件操作功能。

表 2-1 两个系统的比较

	Fourinone-1.11.09	Hadoop-0.21.0
体积	82K	71M
依赖关系	就一个 jar，没有依赖	约 12 项 jar 包依赖
配置	就一个配置文件	较多配置文件和复杂属性
集群搭建	简单，每台机器放一个 jar 和配置文件	复杂，需要 Linux 操作基础和 ssh 等复杂配置，还需要较多配置文件配置

(续)

	Fourinone-1.11.09	Hadoop-0.21.0
计算模式	提供两种计算模式：包工头和工人直接交互方式，包工头和工人通过消息中枢方式交互，后者不需要工人节点可直接访问	计算更多倾向于文件数据的并行读取，而非计算过程的设计。JobTracke 跟 TaskTracker 直接交互，查询 NameNode 后，TaskTracker 直接从 DataNode 获取数据
并行模式	$N \times N$ ，支持单机并行，也支持多机并行，多机多实例并行	$1 \times N$ ，不支持单机并行，只支持多机单实例并行
内存方式	支持内存方式设计和开发应用，并内置完整的分布式缓存功能	以 HDFS 文件方式进行数据处理，内存方式计算支持很弱
文件方式	自带文件适配器处理 IO	HDFS 处理文件 IO
计算数据要求	任意数据格式和任意数据来源，包括来自数据库，分布式文件，分布式缓存等	HDFS 内的文件数据，多倾向于带换行符的数据
调度角色	包工头，可以有多个，支持链式处理，也支持大包工头对小包工头的调度	JobTracke，通常与 NameNode 一起
任务执行角色	农民工，框架支持设计多种类型的工人用于拆分或者合并任务	TaskTracker，通常与 DataNode 一起
中间结果数据保存	手工仓库，或者其他任意数据库存储设备	HDFS 中间结果文件
拆分策略	自由设计，框架提供链式处理对于大的业务场景进行环节拆分数据的存储和计算拆分根据业务场景自定义	以 64M 为拆分进行存储，以行为拆分进行计算，实现 map 接口，按行处理数据进行计算
合并策略	自由设计，框架提供农民工节点之间的合并接口，可以互相交互设计合并策略，也可以通过包工头进行合并	TaskTracker 不透明，较少提供程序控制，合并策略设计复杂，实现 Reduce 接口进行中间数据合并逻辑实现
内存耗用	无需要制定 JVM 内存，按默认即可，根据计算要求考虑是否增加 JVM 内存	需要制定 JVM 内存，每个进程默认 1G，常常 NameNode, jobtracker 等启动 3 个进程，耗用 3G 内存
监控	框架提供多环节链式处理设计支持监控过程，通过可编程的监控方式，给予业务开发方最大灵活的监控需求实现，为追求高性能不输出大量系统监控 Log	输出较多的系统监控 Log，如 Map 和 Reduce 百分比等，但是会牺牲性能，业务监控需要自己实现
打包部署	脚本工具	上传 jar 包到 jobtracker 机器
平台支撑	支持跨平台，Windows 支持良好	多倾向于支持 Linux，Windows 支持不佳，需要模拟 Linux 环境，并且建议只用于开发学习
其他	协同一致性、分布式缓存、通信队列等跟分布式计算关系密切的功能支持	不支持

总结：Hadoop 并不是为了追求一个并行计算的框架而设计，提供快捷和灵活的计算方式去服务各种计算场景，它更多的是一个分布式文件系统，提供文件数据的存储和查询，它的 Map/Reduce 更倾向于提供并行计算方式进行文件数据查询。

Fourinone 和 Hadoop 运行 Word Count 的对比测试（平均 4 核 4G 配置，输入数据为文件）如表 2-2 所示。

表 2-2 运行对比

	Fourinone- 1.11.09(n × 4)	Fourinone- 1.11.09(n × 1)	Hadoop- 0.21.0(n × 1)
3 台机器 × 256M	4s	12s	72s
3 台机器 × 512M	7s	30s	140s
3 台机器 × 1G	14s	50s	279s
19 台机器 × 1G	21s	60s	289s
10 台机器 × 2G	29s		
5 台机器 × 4G	60s		

N×4: Fourinone 可以充分利用单机并行能力, 4 核计算机可以 4 个并行实例计算, Hadoop 目前只能 N×1; 如果要完成 20G 的数据, 实际上 Fourinone 只需要使用 5 台机器用 60 秒完成, 比使用 19 台机器完成 19G 的 Hadoop 节省了 14 台机器, 并提前了 200 多秒。

2.3 关于分布式的一些概念与产品

从业 IT 行业就要做好心理准备, 这个行业的技术日新月异, 产品层出不穷, Hadoop 还没学完, Spark 就来了, Spark 刚学会安装, Storm 又来了……一个产品刚学会了安装配置开发步骤, 没多久它又过时了, 很多工程师就这样最终被拖累了, 拖疲了, 拖的放弃了技术路线, 转向管理岗位。

这么多千姿百态的技术和产品背后有没有某些共性的东西呢? 能让我们换了马甲还能认出它, 能让我们超越学习每个产品的“安装配置开发”而掌握背后的精髓呢, 这样学一反三, 学一招应万招, 能够牢牢掌握好技术的船舵, 穿越一次次颠覆性的技术浪潮。

我们前面章节学习了 Fourinone 的很多分布式技术思想, 那么当我们再次碰到市场上其他千姿百态的技术时, 能否具备了一些识别能力, 具备了一些悟性, 帮助我们认清楚这些技术的本质和用途, 避免重复性学习呢。

我们发现, 市场上和分布式技术一起高频率出现还有这些技术名词: 离线计算、实时计算、迭代计算、Mpi、Spark、Storm、BSP、DAG……我们接下来争取用简练的语言指出这些技术和产品的本质特征。

离线计算: 它不是技术, 也不是产品, 而是一种分布式计算应用方式。Hadoop 应用大部分都是离线计算, 也就是数据通常保存在分布式文件系统中, 在离线状态下去分析这些数据, 类似离线状态的数据分析应用有很多, 日志分析也属于一种离线计算, 很多数据挖掘算法也是基于离线计算的。通常离线计算要分析的数据量很大, 一般用并行计算方式提高效率。

实时计算: 其实多数是指一种增量计算, 尽量让计算过程在短时间完成, 而不是先导入多少数据到一个存储位置, 然后再花费数个小时去分析这些数据, 这是跟离线计算的本质区别, 实时计算的数据是增量性的, 少量的, 变化的, 来了就能算的。实时计算也是一种分布

式计算应用方式。

迭代计算：这种计算方式使用 Hadoop 的 Map/Reduce 并不好弄，它有个最大的区别，就是需要考虑阶段，上一阶段的结果是下一阶段的条件，而且为了高效，最好中间结果保存内存，不要频繁读写文件，迭代计算在数据挖掘聚类算法、机器学习算法等等有很多应用，主要因为这些算法要大量训练，逐渐靠近精确结果，所以中间结果会做为下一轮条件。在 2.1.11 节我们讲述了一个迭代计算的简单例子。

这三种计算应用方式实际上是目前互联网行业运用最多的。下面介绍一些分布计算的产品。

1. Storm

国外 Twitter 公司的一款用于实时分析的软件，后来将其开源，Twitter 这家企业技术实力并不强，规模也不能算大，国内新浪微博跟它类似，但是市场规模要更大。Storm 是一个集成性质的软件产品，核心的消息通信和分布式协调都是使用其他开源软件，要运行 Storm，需要依赖 Apache ZooKeeper、JZMQ(ZeroMQ)，ZooKeeper 用于管理集群中的不同组件的协同，JZMQ(ZeroMQ) 是其核心的内部消息系统。抛开 Storm 本身的安装配置开发步骤，我们认识到其流式计算方式，主要是指增量数据通过消息分发，多个消息接收者并行接收处理这样的一个机制，我们再回头看 Fourinone 能否也实现这样的机制。



提示

跟 Storm 集成 ZooKeeper 和 JZMQ(ZeroMQ) 去实现其核心功能不同，Fourinone 的分布式协调和消息队列功能都是框架自带的功能，拥有自己的实现，底层技术上完全不依赖第三方厂商。

我们回头看 2.1.2 节“包工头 - 职介所 - 手工仓库 - 工人模式”的架构图会发现可以完全实现上面 Storm 的计算机制，通过工人分发任务调用，可以将数据以消息的形式封装到 warehouse 里，调用每个工人并行完成对数据消息的处理，而且根据前面章节的详细阐述，可以支持消息中枢的方式（跟 Storm 一样），也可以支持直接工头工人直接调用方式（这种方式更高效，避免消息中枢瓶颈），两种方式只需要配置修改一下，实现程序无须变动。

由于一直以来国内的工程师都是以学习使用国外开源软件作为生存技能，所以作为国外软件的 Storm，在国内仍然拥有一些追随者，当然这也离不开 Twitter 公司的应用场景。但是我们看到 Fourinone 对分布式技术设计思想的归纳是覆盖这个应用范围的，在软件分布式技术这个层面，我们已经拥有相关的核心技术，并能做得更超越。Fourinone 努力将分布式实现技术做到傻瓜化和普及化，让工程师能更快更轻松的掌握。

2. Spark

这实际上是加州伯克利大学学生的作品，年少轻狂、意气风发的学生们开发了一款基于

内存的用于迭代计算的框架，立刻便风靡世界。有时真佩服这种国外学校的市场影响力，不用做什么推广就产生市场效应，特别是 Spark 还公开表示自己并不成熟稳定就能吸引大量追随者。Spark 建立在一个叫做 mesos 的资源调度框架上（我们会在第 7 章介绍到 mesos），mesos 也是伯克利大学学生们的产物，后来开源后成为 Apache 项目。在 mesos 的 Github 官方 Wiki 上用黑色醒目字体写了这么句话：Please note that Mesos is still in beta. Though the current version is in use in production at Twitter, it may have some stability issues in certain environments. (<https://github.com/mesos/mesos/wiki>)，公开表示还不稳定。

Spark 用比较少的 Scala 代码实现，跟 Hadoop 基于分布式文件 IO 操作方式不同，Spark 尽可能利用内存去做迭代计算，并使用 mesos 管理机器资源分配。

Fourinone 则是通过多个包工头多环节链式处理和包工头内部多阶段处理的粗细粒度方式支持迭代类型计算，对于内存的使用提供完整的单机小型缓存和多机分布式缓存功能（详见第 4 章）。因此，通过提供多环节计算支持和分布式缓存功能，也能实现 Spark 基于内存完成迭代计算的机制。

在互联网公司内部，Spark 相对于 Storm 来说，使用的场景要少很多，这个原因作者也观察思考过，虽然加州伯克利大学耀眼的光环仍然吸引着大量研究者，但是有一点估计是哑巴吃黄连，就是用起来比较难，学习成本大，出了问题难以处理。因此导致工程师真正使用 Spark 建立应用的场景很少。就好比西藏虽然很美很理想，但是氧气稀薄，紫外线太厉害，对皮肤的杀伤力太大，让人很难拥有。还有一点永远不会变，那就是工程师和研究人员不同，研究人员不能研究太通俗易懂、简单直观的东西，但是工程师最终只会支持自己能学会的、容易学会的、容易谋生的工具。EJB 被 Hibernate 打败就是一个活生生例子，曾经的 EJB 是分布式技术领域的重要代表，为什么 EJB2.0 几乎到死亡的边缘，原因可能很多，但有一点是肯定的，因为它开发配置复杂难用，而 Hibernate 更轻量级和简单易用，其实 Hibernate 只能做 O/R 映射，根本解决不了分布式框架的问题，但是这不影响它取代 EJB，因为大部分开发人员对 EJB 的理解就是当做 O/R 映射来用。

3. MPI

MPI (Message Passing Interface) 是消息传递并行程序设计的标准之一，它是一个规范或者是一个库，但是不包括实现，目前最新版本为 MPI-2 (1997 年发布)，MPI 能完成并行机的进程间通信，当前的实现版本有 MPICH2 和 OPENMPI，目前广泛用于互联网企业的广告算法和迭代算法，阿里和国内其他大型互联网公司都有成百上千的 MPI 计算集群。

MPI 定义了进程间的通信接口，比如 `mpi_send` 和 `mpi_recv` 等两个进程发送接收最常用接口，由此可见，基于 MPI 可以利用多个进程并行作业和交互，相对于上面介绍的 Hadoop、Storm 等计算平台，MPI 能实现更灵活的并行计算方式，MPI 更接近开发工具包和开发规范，

在这点上跟 Fourinone 有很大的相似性，比如 giveTask 和 doTask 跟 mpi_send 和 mpi_recv 在并行计算上有相似点。

但是 MPI 仅仅定义出进程间的通信接口和交互方式，它缺乏对并行计算模式的设计归纳和角色抽象，比如没有任务调度角色包工头，没有多环节处理模式设计，没有独立的协调者角色，等等。因此基于 MPI 开发并行计算的难度要更大，编程要基于进程通信接口，缺乏一个完善而灵活的框架去简化复杂度和屏蔽编程错误，比如多进程合并的死锁问题，MPI 开发者需要自己在开发中考虑。另外 MPI 的调度是一个突出问题，由于缺乏专门的任务调度角色，MPI 的启动借助于操作系统进程管理器，把各个 MPI 进程当作操作系统进程孤立地启动和管理，因此 MPI 的调度方案一直是阿里、百度等公司进行研究的项目（我们在 7.4.1 “其他 MPI 作业资源调度技术” 中会详细谈到）。

MPI 在中国著名高校里作为研究生课程，有比较深的普及度，但是在业界开源社区并不火热，多数应用于 Linux 下的 C 环境开发，Java 开源社区涉及的很少，几乎没有太知名的 Java 版 MPI 实现。

MPI 虽然原始，开发运行较繁琐，但仍不失为一种灵活的并行计算方式。比如前面我们举的并行计算递归，或者并行计算求圆周率这些灵活计算的例子，使用 Hadoop/STORM/Spark 几乎做不了，使用 MPI 可以做出，但是实现肯定比 Fourinone 要麻烦。

4. BSP

Apache 发布了一款 Hama 的并行计算软件牵扯出了 BSP (Bulk Synchronous Parallel) 思想，工程师们感慨万千，老外又出新产品和技术了，赶紧学啊，指南在哪里，helloworld 例子在哪里，安装部署文档在哪里，又要开始学习新技术的苦逼日子了……其实我们不妨先从原理上分析一下该技术的来龙去脉和核心思想，再看看我们掌握的已有技术是否已经覆盖了。

Hama 基于 BSP 实现了 Google 的 Pregel 的思想，提供一个相对于 Map/Reduce 和 MPI 更灵活的计算模式，大致由并行进程、消息通信层、栅栏同步层组成，并划分成一系列的 superstep。官方资料强调 BSP 相对于 MPI 更侧重解决通信密集型计算，并对计算过程用一些参数指标，比如：进程数、进出消息数、消息大小等等，试图预估出整个计算需要耗费的时间。BSP 相对于 MPI 的优化主要还是设计模型上，通信接口等优化不了什么。提出一个栅栏的设计，其实就是划分阶段和环节，用于迭代类型。还有就是基于消息的进程间通信交互，用发布接收消息模式代替点对点通信。

Google 相关论文是：Pregel: A System for Large-Scale Graph Processing。

我们看到，BSP 思想的核心其实就是将复杂计算过程划分多个阶段，每阶段等待最慢的进程计算完才能进行下一个阶段，并且试图通过一些参数指标估算出并行计算时间，但是这个时间较难估计，跟用户业务实现复杂度相关。而 Fourinone 对阶段的粗细粒度设计，BSP

并没有这样的考虑。

5. DAG

我们对任务调度 DAG（有向无环图）的认识更多是从微软的分布式并行计算平台 Dryad 来的，Dryad 系统的总体构建用来支持有向无环图（Directed Acycline Graph, DAG）类型数据流的并行程序。Dryad 的整体框架根据程序的要求完成调度工作，自动完成任务在各个节点上的运行。在 Dryad 平台上，每个 Dryad 工作或并行计算过程被表示为一个有向无环图，如图 2-10 所示。

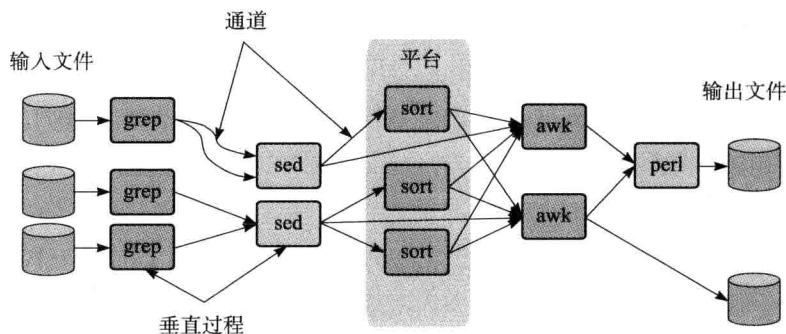


图 2-10 DAG (有向无环图)

上面的描述很抽象，但简单来说，DAG 最有价值的地方是构成了一个任务并行流的概念，比如我们的任务共有几个子任务，第一个子任务需要 20 台计算机并行计算完成，第二子任务需要 40 台计算机并行计算完成，第三个子任务需要 30 台机器完成，现在我们来设计任务调度，由于业务需求，任务一 + 条件一 + 任务三是我们需要的，另外一个需求任务一 + 条件二 + 任务二是我们需要，每个子任务都是多个计算机并行计算，并且子任务之间有先后顺序，同时需要匹配一定条件，反过来观察我们的计算流程中的节点，刚好构成了一个有向无环图的树结构。

那么 Fourinone 如何实现这样的任务流结构，多个任务之间有先后顺序，而且要匹配流转条件，同时每个任务又是并行处理。我们回头看看 2.1.7 节“计算集群模式和兼容遗留计算系统”的工头工人计算集群模式，每一个工头计算单元就是子任务，它同时由多个工人并行处理，在计算集群模式的架构图里我们看不到先后顺序，各工头计算单元之间是平行的，但是我们根据计算集群的扩充性很容易就能延伸出，工头计算单元可以前后衔接，衔接方式是通过在工人实现里面调用，由于工人实现是灵活的，所以也可以在衔接时增加条件判断，根据需要选择下一个子任务交给哪个工头单元完成。因此，通过计算集群的衔接方式，我们能很容易实现出一个 DAG 的结构出来。

还有更多的技术和产品我们就不一一分析了。

通过对以上对市场上高频率出现的分布式技术和产品分析，我们发现一个道理，其实分布式技术都是那套原理，分布式技术的产品只不过往不同方面的侧重有所不同，然后就包装出一个新的概念，比如支持多阶段包装出 BSP，支持消息分流包装出 Storm，支持计算任务流包装出 DAG……其实我们用一个灵活的框架就可以概括这些不同侧重面的应用了，这样节省了我们大量的学习成本，并且开阔了眼界。碰到国外新技术和新产品，我们应该学会思考一下看看它是穿了什么马甲，而不要急着进行“安装配置开发步骤”这样无休止地盲目跟学。

2.4 配置文件和核心 API 介绍

Fourinone 开发包自带了一个 config.xml 的配置文件，运行时必须要有。为了降低门槛，更容易上手，配置文件和核心 API 都尽量简化，XML 配置里一般都不需要改，需要修改的每个例子里都有涉及。同样一般常用的 API 例子都有说明，没说明的一般不用，API 多了不容易上手，例子已经基本覆盖所有 API。下面把使用最多的核心 API 说明一下（ParkServer 的 API 详见第 3 章），参见表 2-3。

表 2-3 核心 API 说明

分类	名称	说明
工头	WareHouse giveTask(WareHouse inhouse)	实现分配工人要做的任务
	WorkerLocal[] getWaitingWorkers(String workerType)	获取集群中等待的工人
	WareHouse[] doTaskBatch(WorkerLocal[] wks, WareHouse wh)	所有工人批量完成给定任务处理
	doProject(WareHouse inhouse)	工头开始项目启动
	toNext	多个包工头链式处理
工人	WareHouse doTask(WareHouse inhouse);	实现工头分配的任务
	waitWorking(String workerType)	等待工作状态，指定工人类型
	Workman[] getWorkerAll();	获取所有的工人
	Workman[] getWorkerElse();	获取除自己外的其他工人
	Workman[] getWorkerElse(String workerType)	获取其他某种类型的工人
	Workman getWorkerIndex(int index)	获取第 index 位工人
	Workman getWorkerIndex(String workerType, int index)	获取某种类型的第 index 位工人
	int getSelfIndex();	获取自己在工作中的位置
	boolean receive(WareHouse inhouse)	接收来自其他工人的传递

2.5 实践与应用

2.5.1 一个简单的示例

本节用最简洁的代码示范 Fourinone 如何进行分布式计算，如上面章节所述，Fourinone 采用一种工头链式结合工人并行的计算结构简化分布式计算，能够通俗易懂，并能深入控制整个计算过程。

完成一个并行计算需要工头（SimpleCtor）、工人（SimpleWorker）、职介所（ParkServerDemo）3 个角色，如下所示：

- **SimpleCtor**: 是一个工头实现，它实现 giveTask 接口，并通过 getWaitingWorkers 获取线上工人节点（工人节点为一个独立进程，它可以独立部署一台机器也可以一台机器部署多个），并调用该工人的 doTask 方法完成任务，传入的任务是一句“hello”的话。注意工人的 doTask 方法是一个异步调用，它会马上返回一个 result，但是没有值，需要轮循 result 是否有值为止，有值就代表工人已经处理完该任务了。这样做是因为当多个任务分配给多个工人完成时，它们之间是并行的，不会等待前个工人完成再去分配下一个工人任务。
- **SimpleWorker**: 是一个工人实现，它实现 doTask 接口，从 WareHouse 获取到工头的传入参数 word，并回应“hello word”，它的输入输出类型都是 WareHouse，WareHouse 是一个 map 结构，可以放置任何类型的对象。SimpleWorker 通过 waitWorking 开始等待任务，waitWorking 需要输入一个参数，给该工人指定一个类型描述，在更复杂的应用中，可以设计多种类型的工人，比如有的做任务处理，有的做任务结果合并。
- **ParkServerDemo**: 是负责分布式计算过程的协同服务，它必须启动才能完成分布式计算。

运行步骤：

1) 启动 ParkServerDemo (它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定)，结果如图 2-11 所示：

```
Java -classpath fourinone.jar; ParkServerDemo
```

```
D:\demo\comutil\test>java -cp fourinone.jar; ParkServerDemo
三月 20, 2013 3:22:32 下午
INFO: wantBeMaster.....
三月 20, 2013 3:22:33 下午
INFO: get one of other parks for init parkInfo.....
三月 20, 2013 3:22:34 下午
INFO: setMaster(localhost:1888):true
三月 20, 2013 3:23:24 下午 [Park] [askLeader]
INFO: receive askLeader.....
三月 20, 2013 3:23:24 下午
INFO: setMaster(localhost:1888):true
三月 20, 2013 3:23:57 下午 [Park] [askLeader]
INFO: receive askLeader.....
三月 20, 2013 3:23:57 下午
INFO: setMaster(localhost:1888):true
```

图 2-11 ParkServerDemo

2) 运行 SimpleWorker (它的 IP 端口已经在配置文件的 WORKER 部分的 SERVERS 指定), 如果如图 2-12 所示:

```
java -cp fourinone.jar; SimpleWorker
```

```
D:\demo\comutil\test>java -cp fourinone.jar; SimpleWorker
三月 21, 2013 2:48:16 下午
INFO: getLeaderPark.....
三月 21, 2013 2:48:16 下午
INFO: leader server is(localhost:1888)
Hello from Contractor.
终止批处理操作吗(Y/N)?
```

```
D:\demo\comutil\test>
D:\demo\comutil\test>
```

```
D:\demo\comutil\test>java -cp fourinone.jar; SimpleWorker
三月 21, 2013 2:52:53 下午
INFO: getLeaderPark.....
三月 21, 2013 2:52:53 下午
INFO: leader server is(localhost:1888)
Hello from Contractor.
```

图 2-12 SimpleWorker

运行 SimpleCtor, 结果如图 2-13 所示:

```
java -cp fourinone.jar; SimpleCtor
```

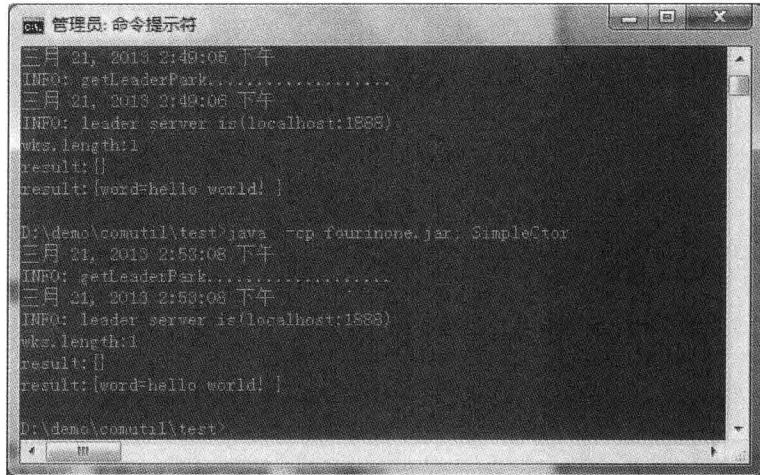


图 2-13 SimpleCtor



以上程序启动时都需要配置文件 config.xml, 可以将配置文件、程序 class 文件、fourinone.jar 三者放到相同目录中, 如果 class 文件有包名, 需要放在包根目录处, 这样能默认找到。也可以通过 BeanContext.setConfigFile 指定其他目录位置, 特别是使用 Eclipse 会自动生成 class 目录和运行目录, 导致具体路径不清晰, 可以尝试指定绝对路径, 或者慢慢调试改成相对路径。

掌握 Fourinone 最基本的工头工人分布式计算方式后, 可以进一步学习另一个完整的 demo, 会示范多个任务多个工人的分配和结果轮循以及多个工头的链式处理方式。

Demo 完整源码如下:

```

// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
    public static void main(String[] args){
        BeanContext.startPark();
    }
}

// SimpleWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;

public class SimpleWorker extends MigrantWorker
{
    public WareHouse doTask(WareHouse inhouse)
    {
        String word = inhouse.getString("word");
        System.out.println(word+" from Contractor.");
    }
}

```

```

        return new WareHouse("word", word+" world!");
    }

    public static void main(String[] args)
    {
        SimpleWorker mw = new SimpleWorker();
        mw.waitWorking("simpleworker");
    }
}

// SimpleCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.ArrayList;

public class SimpleCtor extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("simpleworker");
        System.out.println("wks.length:"+wks.length);

        WareHouse wh = new WareHouse("word", "hello");
        WareHouse result = wks[0].doTask(wh);

        while(true){
            if(result.getStatus() == WareHouse.READY)
            {
                System.out.println("result:"+result);
                break;
            }
        }

        return null;
    }

    public static void main(String[] args)
    {
        SimpleCtor a = new SimpleCtor();
        a.giveTask(null);
    }
}

```

2.5.2 工头工人计算模式更完整的示例

从前面章节的原理介绍里知道，我们现实中的分布式计算存在多个环节，比如有的任务拆分，有的计算结果合并，或者多个拆分和合并，它们之间是串行关系，也就是合并必须等待拆分和计算完成才能进行，同时每个拆分或者合并的任务又都是并行的过程。

CtorDemo：是包含了3个工头实例，对应3个环节，链式处理，实现过程获取到线上工

人节点，进行调用，所有的分配任务和中间结果存储都由自己实现处理。

这里简单地将 20 条数据分配给多个工人处理。数据用 `data` 变量表示，`j` 用来记录计算结果，如果 `j==20`，标志结束。任务初始为一个 `id` 的字符传给工头实例 1，工头加上自己名称的描述和数据 `data` 后传给工人处理，工人再加上自己的名称和处理信息返回给工头，工头实例 1 处理完再传给工头实例 2，直到 3 个工头都链式处理完，这里将上一个工头的处理结果又当做下一个工头的输入。

注意

工头和工人之间是异步调用，会马上返回，需要检查结果是否完成。

```
WareHouse[] hmarr = new WareHouse[wks.length];
```

这里通过 `hmarr` 数组来记录每次每个工人任务分配的结果，需要轮循 `hmarr` 的每个结果是否已经计算完成，如果计算完成就设置为 `null`，进行新的任务安排。

- `WorkerDemo`: 是一个工人实现，工人可以指定某种类型，比如有的工人用于计算，有的用于合并，也都是自己实现。这里只是简单地在工头传入的 `id` 后加上自己的名称信息代表处理。

```
waitWorking("localhost", Integer.parseInt(args[1]), "workdemo");
```

该方法进行任务等待，其中 3 个参数分别指定工人监听 ip、工人监听端口、工人类型。

- `ParkServerDemo`: 分布式计算过程的协同服务 park。

另外，工头和工人之间的计算交互有两种模式，一种是工头直接调用工人，一种是通过 park 消息中枢调用工人，可以在配置文件里配置 COMPUTEMODE 的默认值进行指定，默认是直接调用方式。

部署：将 `CtorDemo`、`WorkerDemo`、`ParkServerDemo` 分别部署在不同机器或者同台机器不同进程，`Worker` 可以有多个。

运行步骤：

- 1) 启动 `ParkServerDemo`（它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定）：

```
Java -classpath fourinone.jar; ParkServerDemo
```

- 2) 运行 `WorkerDemo`，通过传入不同的端口和名称参数指定多个 `Worker`，这里假设在同一机演示，ip 设置为 `localhost`，如果如图 2-14 所示：

```
java -cp fourinone.jar; WorkerDemo aaa 2008
java -cp fourinone.jar; WorkerDemo bbb 2009
java -cp fourinone.jar; WorkerDemo ccc 2010
```

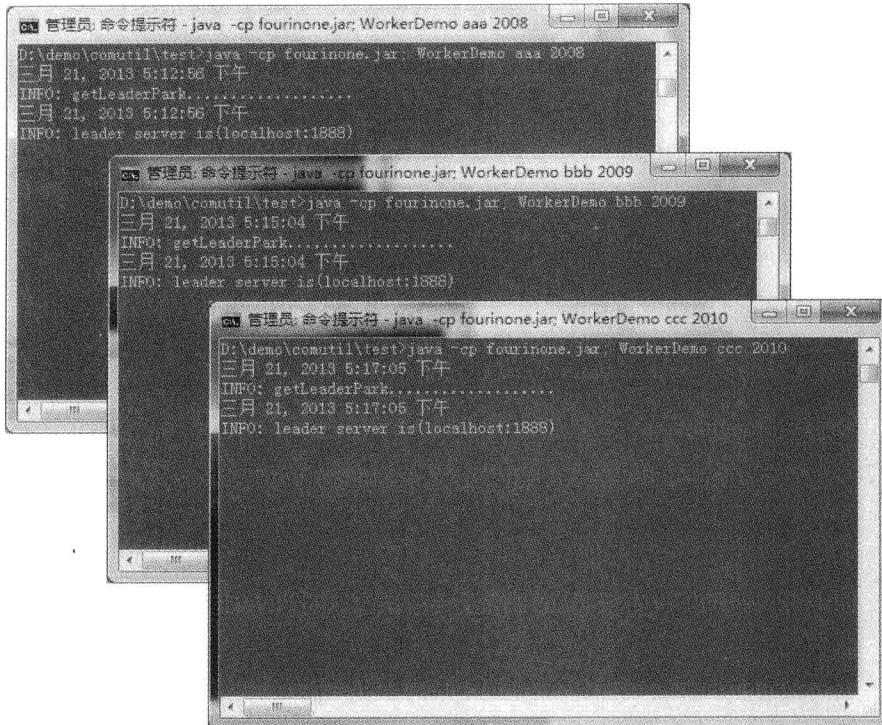


图 2-14 WorkerDemo

3) 运行 CtorDemo：

```
java -cp fourinone.jar; CtorDemo
```

可以看到工头窗口的输出如图 2-15 所示。



图 2-15 CtorDemo

总共三个包工头链式处理，先后将任务分配给 3 个工人并行执行，窗口内的信息“ThreeCtor16-ccc”代表“第三个工头的第 16 条任务分配给 ccc 工人执行”。最后将所有包工头的执行结果汇总输出。

三个工人窗口的信息输出如下：

```

    [aaa] 管理员: 命令提示符 - java -cp fourinone.jar;WorkerDemo aaa 2008
    aaa inhouse:OneCtor19
    aaa inhouse:TwoCtor0
    aaa inhouse:TwoCtor3
    aaa inhouse:TwoCtor6
    aaa inhouse:OneCtor15
    aaa inhouse:OneCtor18
    aaa inhouse:OneCtor21
    aaa inhouse:TwoCtor1
    aaa inhouse:TwoCtor7
    aaa inhouse:TwoCtor10
    aaa inhouse:OneCtor14
    aaa inhouse:OneCtor17
    aaa inhouse:OneCtor20
    aaa inhouse:TwoCtor2
    aaa inhouse:TwoCtor5
    aaa inhouse:TwoCtor8
    aaa inhouse:TwoCtor11
    aaa inhouse:TwoCtor14
    aaa inhouse:TwoCtor17
    aaa inhouse:TwoCtor20
    aaa inhouse:ThreeCtor2
    aaa inhouse:ThreeCtor5
    aaa inhouse:ThreeCtor9
    aaa inhouse:ThreeCtor12
    aaa inhouse:ThreeCtor16
    aaa inhouse:ThreeCtor18
    aaa inhouse:ThreeCtor21

    [bbb] 管理员: 命令提示符 - java -cp fourinone.jar;WorkerDemo bbb 2009
    bbb inhouse:OneCtor15
    bbb inhouse:OneCtor18
    bbb inhouse:OneCtor21
    bbb inhouse:TwoCtor1
    bbb inhouse:TwoCtor7
    bbb inhouse:TwoCtor10
    bbb inhouse:OneCtor14
    bbb inhouse:OneCtor17
    bbb inhouse:OneCtor20
    bbb inhouse:TwoCtor2
    bbb inhouse:TwoCtor5
    bbb inhouse:TwoCtor8
    bbb inhouse:TwoCtor11
    bbb inhouse:TwoCtor14
    bbb inhouse:TwoCtor17
    bbb inhouse:TwoCtor20
    bbb inhouse:ThreeCtor2
    bbb inhouse:ThreeCtor5
    bbb inhouse:ThreeCtor9
    bbb inhouse:ThreeCtor12
    bbb inhouse:ThreeCtor16
    bbb inhouse:ThreeCtor18
    bbb inhouse:ThreeCtor21

    [ccc] 管理员: 命令提示符 - java -cp fourinone.jar;WorkerDemo ccc 2010
    ccc inhouse:OneCtor14
    ccc inhouse:OneCtor17
    ccc inhouse:OneCtor20
    ccc inhouse:TwoCtor2
    ccc inhouse:TwoCtor5
    ccc inhouse:TwoCtor8
    ccc inhouse:TwoCtor11
    ccc inhouse:TwoCtor14
    ccc inhouse:TwoCtor17
    ccc inhouse:TwoCtor20
    ccc inhouse:ThreeCtor2
    ccc inhouse:ThreeCtor5
    ccc inhouse:ThreeCtor9
    ccc inhouse:ThreeCtor12
    ccc inhouse:ThreeCtor16
    ccc inhouse:ThreeCtor18
    ccc inhouse:ThreeCtor21
  
```

图 2-16 工人窗口结果

每个工人窗口输出了执行每个工头的任务，“bbb inhouse:ThreeCtor17”表示 bbb 工人执行第 3 个工头的第 17 条任务。可以看到每个工人都是以并行争抢方式去执行包工头的任务。

Demo 完整源码如下：

```

// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
    public static void main(String[] args){
        BeanContext.startPark();
    }
}

// WorkerDemo
  
```

```
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;

public class WorkerDemo extends MigrantWorker
{
    private String workname;
    public WorkerDemo(String workname)
    {
        this.workname = workname;
    }

    public WareHouse doTask(WareHouse inhouse)
    {
        String v = inhouse.getString("id");
        System.out.println(workname+" inhouse:"+v);
        return new WareHouse("id",v+"-"+workname+"-");
    }

    public static void main(String[] args)
    {
        WorkerDemo wd = new WorkerDemo(args[0]);
        wd.waitWorking("localhost",Integer.parseInt(args[1]),"workdemo");
    }
}

// CtorDemo
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.ArrayList;

public class CtorDemo extends Contractor
{
    private String ctorname;

    CtorDemo(String ctorname)
    {
        this.ctorname = ctorname;
    }

    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("workdemo");
        System.out.println("wks.length:"+wks.length);

        String outStr = inhouse.getString("id");
        WareHouse[] hmarr = new WareHouse[wks.length];

        int data=0;
        for(int j=0;j<20;)
        {
            for(int i=0;i<wks.length;i++)
            {
                if(hmarr[i]==null){
                    WareHouse wh = new WareHouse();
                    wh.put("id",ctorname+(data++));
                    hmarr[i] = wh;
                }
            }
        }
    }
}
```

```
        hmarr[i] = wks[i].doTask(wh);
    }
    else if(hmarr[i].getStatus()!=WareHouse.NOTREADY)
    {
        System.out.println(hmarr[i]);
        outStr+=hmarr[i];
        hmarr[i]=null;
        j++;
    }
}
}

inhouse.setString("id", outStr);
return inhous;
}

public static void main(String[] args)
{
    Contractor a = new CtorDemo("OneCtor");
    a.toNext(new CtorDemo("TwoCtor")).toNext(new CtorDemo ("ThreeCtor"));
    WareHouse house = new WareHouse("id","begin ");
    System.out.println(a.giveTask(house,true));
}
```

2.5.3 工人合并互相 say hello 的示例

假设你已经看过前面的分布式计算上手 demo 指南，对 Fourinone 基本的分布式并行计算方式有了初步了解。

本 demo 演示了工头和几个工人之间互相 sayhello 的简单例子，从而了解到集群计算节点之间互相交互，以及工头批量处理和工人互相传递数据（多用于合并）的功能。

- **HelloCtor:** 是一个工头实现，它实现 giveTask 接口，它首先通过 getWaitingWorkers 获取到一个线上工人的集合，然后通过 doTaskBatch 进行批量任务处理，这里工头向每个工人说句“hello”打招呼。doTaskBatch 有两个参数，分别是工人集合和任务，该方法会等到每个工人都执行完该任务才返回，因此使用 doTaskBatch 不需要轮循检查每一个调用结果，它是一个批量处理。为了节省资源利用，工头运行结束后不会退出 jvm，可以使用 exit 方法强行退出。
- **HelloWorker:** 是一个工人实现，这里它实现了 doTask 和 receive 接口，分别用于被工头和其他工人调用。doTask 实现了被工头调用执行任务的内容，这里该工人向工头和其他工人“say hello”招呼，并告诉自己的名字。它通过 getWorkerElse 获取到集群中除自己以外的其他工人，getWorkerElse 可以传入一个参数指定工人类型，然后依次调用其他工人的 receive 方法传递信息。receive 实现了该工人被其他工人调用的处理内

容，参数 WareHouse 由其他工人传入，它返回一个 boolean 值，可以代表接收和处理是否成功。这里简单的将其他工人的问候输出。

运行步骤：

- 1) 启动 ParkServerDemo (它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定)：

```
Java -classpath fourinone.jar; ParkServerDemo
```

- 2) 运行一到多个 HelloWorker (传入 3 个参数，依次是该工人的名字、IP 或者域名、端口)：

```
java -cp fourinone.jar; HelloWorker aaa localhost 2008
java -cp fourinone.jar; HelloWorker bbb localhost 2009
```

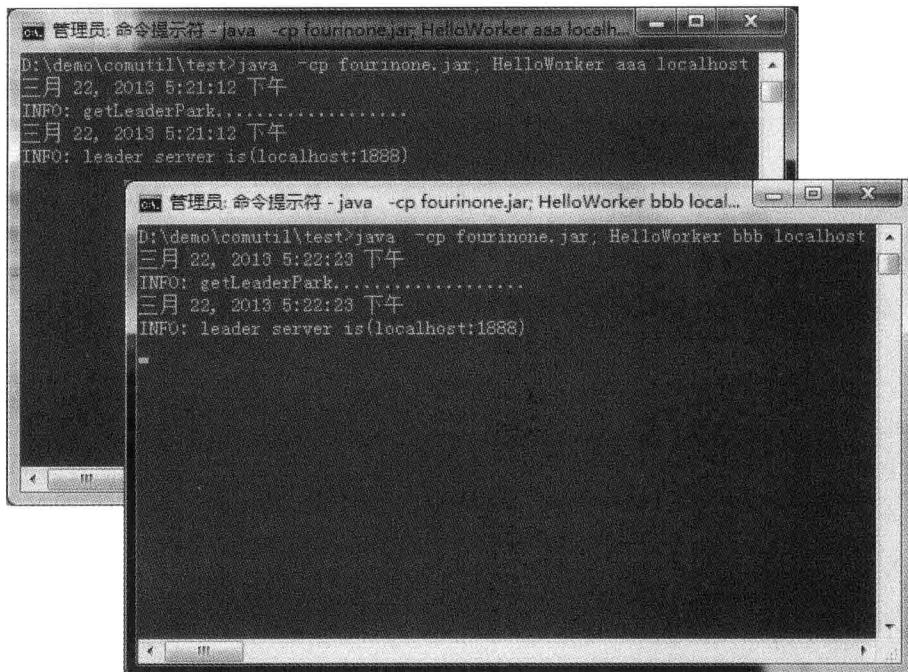


图 2-17 HelloWorker

- 3) 运行 HelloCtor：

```
java -cp fourinone.jar; HelloCtor
```

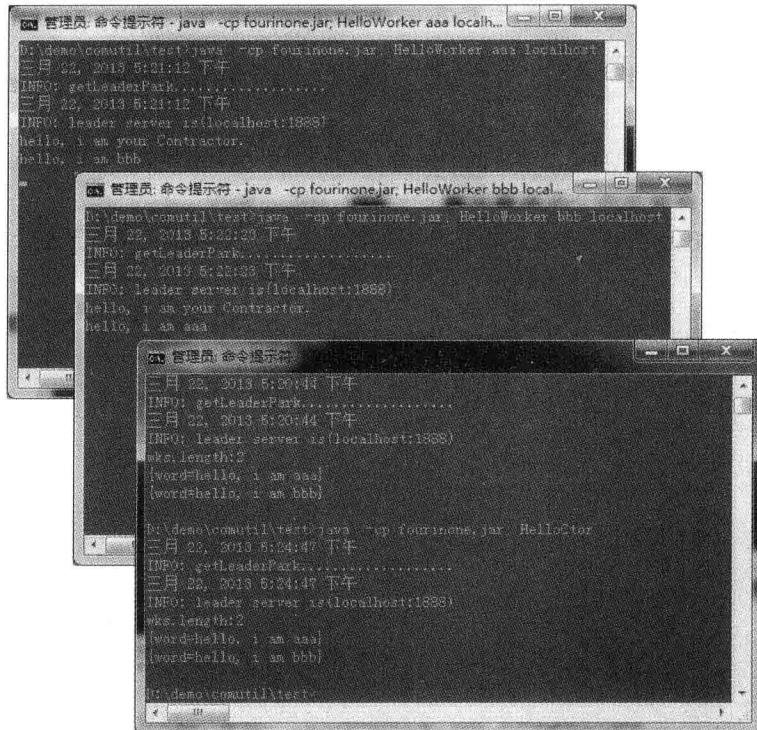


图 2-18 HelloCtor

从上面可以看到，工头和每个工人都收到来自对方的 hello 招呼。



注意

`doTaskBatch` 会等集群中最慢的一个工人完成任务才统一返回，如果希望能让机器运行快的机器在完成后能马上又分配新的任务，而不用等待，实现能者多劳，可以不使用 `doTaskBatch`，而采用逐个调用每个工人的 `doTask` 并轮循结果状态的方式实现，具体请参考前面的分布式计算完整 demo。

实际上，工头对工人的调用是通过 `doTask`，工人对工人的调用是通过 `receive`。`doTask` 用于工头分配任务，`receive` 多用于工人之间合并传递数据，每个工人都可以同时向其他工人传递数据，并接收来自其他工人的数据。集群中每个工人向其他工人传递数据都完成了，也就意味着每个工人都接收完成了（详细“工人 - 工人”交互原理可参见 2.1 节）。

Demo 完整源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
```

```
public static void main(String[] args){
    BeanContext.startPark();
}
}

// HelloWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
import com.fourinone.Workman;

public class HelloWorker extends MigrantWorker
{
    private String name;
    public HelloWorker(String name){
        this.name = name;
    }

    public WareHouse doTask(WareHouse inhouse)
    {
        System.out.println(inhouse.getString("word"));
        WareHouse wh = new WareHouse("word", "hello, i am "+name);
        Workman[] wms = getWorkerElse("helloworlder");
        for(Workman wm:wms)
            wm.receive(wh);
        return wh;
    }

    public boolean receive(WareHouse inhouse)
    {
        System.out.println(inhouse.getString("word"));
        return true;
    }

    public static void main(String[] args)
    {
        HelloWorker mw = new HelloWorker(args[0]);
        mw.waitWorking(args[1],Integer.parseInt(args[2]),"helloworlder");
    }
}

// HelloCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;

public class HelloCtor extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("helloworlder");
        System.out.println("wks.length:"+wks.length);
        WareHouse wh = new WareHouse("word", "hello, i am your Contractor.");
        WareHouse[] hmarr = doTaskBatch(wks, wh);

        for(WareHouse result:hmarr)
```

```

        System.out.println(result);

    return null;
}

public static void main(String[] args)
{
    HelloCtor a = new HelloCtor();
    a.giveTask(null);
    a.exit();
}
}

```

2.5.4 实现 Hadoop 经典实例 Word Count

很多人是通过 Word Count 入门分布式并行计算，该 demo 演示了 Hadoop 的经典实例 Word Count 的实现。

- 输入数据： n 个数据文件，每个 1G 大小，为了方便统计，每个文件的数据由“aaa bbb ... ccc”（由空格分割的 1k 单词组）不断复制组成。
- 输出数据：输出这 $n \times 1G$ 个数据文件中的每个单词总数。

Fourinone 简单实现思路：假设有 n 台计算机，将这 n 个 1G 数据文件放置在每台计算机上，每台计算机各自统计 1G 数据，然后合并得到结果。

- WordcountCT：为一个工头实现，它把需要处理的数据文件名称发给各个工人，然后用一个 `HashMap<String, Integer>` wordcount 的 map 用来装结果。
- WordcountWK：为一个工人实现，它按照每次读取 8M 的方式处理文件数据，将文件大小除以 8M 得到总次数，每次处理过程将字符串进行空格拆分，然后放入本地一个 MAP 里，完成后将此 MAP 发给工头。
- ParkServerDemo：分布式计算过程的协同服务 park。

运行步骤：

1) 启动 ParkServerDemo（它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定）：

```
Java -classpath fourinone.jar; ParkServerDemo
```

2) 运行一到多个 WordcountWK，通过传入不同的端口指定多个 Worker，这里假设在同一机演示，IP 设置为 localhost：

```
java -cp fourinone.jar; WordcountWK 2008
java -cp fourinone.jar; WordcountWK 2009
java -cp fourinone.jar; WordcountWK 2010
```



图 2-19 WordcountWK

3) 运行 WordcountCT，传入文件路径（假设多个工人处理相同数据文件）：

```
java -cp fourinone.jar; WordcountCT D:\demo\comutil\test\data\inputdata.txt
```

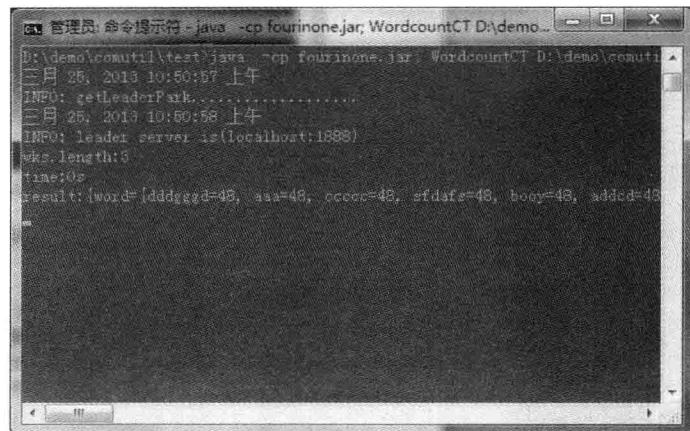


图 2-20 WordcountCT

我们可以看到，3个工人每个计算相同的数据文件，总共的单词总数如上所示。现实场景中工人应该分布在不同计算机，每台计算机上的数据文件不相同（如果数据文件不同，每次按照64K获取处理截断单词的情况，请参考6.7节的文件解析处理），这里演示清楚了并行计算原理和过程，可以根据需求去灵活设计。

如果将以上实现部署到分布式环境里，它是 $1 \times n$ 的并行计算模式，也就是每台机器一个计算实例，Fourinone可以支持充分利用一台机器的并行计算能力，可以进行 $n \times n$ 的并行计算模式，比如，每台机器4个实例，每个只需要计算256M，总共1G，这样整体的速度会大幅上升。请参见表2-2。

Demo完整源码如下：

```
// inputdata.txt
cccc world good dddfg word googl booy aaa dddgggd sfdafs addcd cccc world
good dddfg word googl booy aaa dddgggd sfdafs addcd cccc world good dddfg word
googl booy aaa dddgggd sfdafs addcd cccc world good dddfg word googl booy aaa
dddgggd sfdafs addcd cccc world good dddfg word googl booy aaa dddgggd sfdafs addcd
cccc world good dddfg word googl booy aaa dddgggd sfdafs addcd cccc world good
dddfg word googl booy aaa dddgggd sfdafs addcd cccc world good dddfg word googl
booy aaa dddgggd sfdafs addcd cccc world good dddfg word googl booy aaa dddgggd
sfafs addcd cccc world good dddfg word googl booy aaa dddgggd sfdafs addcd cccc
world good dddfg word googl booy aaa dddgggd sfdafs addcd cccc world good dddfg
word googl booy aaa dddgggd sfdafs addcd cccc world good dddfg word googl booy
aaa dddgggd sfdafs addcd cccc world good dddfg word googl booy aaa dddgggd sfdafs
addcd cccc world good dddfg word googl booy aaa dddgggd sfdafs addcd cccc world
good dddfg word googl booy aaa dddgggd sfdafs addcd
```

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
    public static void main(String[] args){
        BeanContext.startPark();
    }
}

// WordcountWK
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
import com.fourinone.FileAdapter;
import com.fourinone.FileAdapter.ReadAdapter;
import java.util.StringTokenizer;
import java.util.HashMap;
import java.util.ArrayList;
import java.io.File;

public class WordcountWK extends MigrantWorker
```

```

{
    public WareHouse doTask(WareHouse inhouse)
    {
        String filepath = inhouse.getString("filepath");
        long n=64;//FileAdapter.m(8)
        long num = (new File(filepath)).length()/n;
        FileAdapter fa = null;
        ReadAdapter ra = null;
        byte[] bts = null;
        HashMap<String, Integer> wordcount = new HashMap<String, Integer>();
        fa = new FileAdapter(filepath);
        for(long i=0;i<num;i++){
            ra = fa.getReader(i*n, n);
            bts = ra.readAll();
            StringTokenizer tokenizer = new StringTokenizer(new String(bts));
            while(tokenizer.hasMoreTokens()){
                String curword = tokenizer.nextToken();
                if(wordcount.containsKey(curword))
                    wordcount.put(curword, wordcount.get(curword)+1 );
                else
                    wordcount.put(curword, 1 );
            }
        }
        fa.close();
        return new WareHouse("word", wordcount);
    }

    public static void main(String[] args)
    {
        WordcountWK mw = new WordcountWK();
        mw.waitWorking("localhost",Integer.parseInt(args[0]),"wordcount");
    }
}

// WordcountCT
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Date;

public class WordcountCT extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("wordcount");
        System.out.println("wks.length:"+wks.length);
    }
}

```

```

WareHouse[] hmarr = doTaskBatch(wks, inhouse);

HashMap<String, Integer> wordcount = new HashMap<String, Integer>();
for(WareHouse hm:hmarr)
{
    HashMap<String, Integer> wordhm = (HashMap<String, Integer>)hm.get ("word");
    for(Iterator<String> iter=wordhm.keySet().iterator();iter.hasNext();){
        String curword = iter.next();
        if(wordcount.containsKey(curword))
            wordcount.put(curword, wordcount.get(curword)+ wordhm.
                get(curword));
        else
            wordcount.put(curword, wordhm.get(curword));
    }
}

return new WareHouse("word", wordcount);
}

public static void main(String[] args)
{
    Contractor a = new WordcountCT();
    long begin = (new Date()).getTime();
    WareHouse result = a.giveTask(new WareHouse("filepath", args[0])); //eg:
        "D:\\demo\\parallel\\a\\three.txt"
    long end = (new Date()).getTime();
    System.out.println("time:"+ (end-begin)/1000+"s");
    System.out.println("result:"+result);
}
}
}

```

2.5.5 分布式多机部署的示例

在前面的分布式计算上手 demo，分布式计算 sayhello，分布式计算完整 demo 中，对于工人（worker）、工头（ctor），parkserver 在多台计算机上的部署和配置：

每台计算机放置实现类：fourinone.jar、config.xml 文件即可，parkserver 的 IP 配置要保持一致（也可以用域名）：

假设有 192.0.0.1，192.0.0.2，192.0.0.3 三台计算机，应如下部署：

- 192.0.0.1 上部署：parkserver 类字节码文件，fourinone.jar，config.xml。其中 config 的 PARK 部分的 SERVERS 配置为：

```
<SERVERS>192.0.0.1:1888,localhost:1889</SERVERS>
```

第二个是备份 parkserver，没有可以不设置。

- 192.0.0.2 上部署：worker 实现类字节码文件，fourinone.jar，config.xml。其中 config 的 PARK 部分的 SERVERS 配置指定上面 parkserver 的 ip 端口（192.0.0.1:1888）。可以在 WORKER 部分的 SERVERS 指定该工人机器 ip192.0.0.2 端口。也可以在 waitWorking 方法程序指定工人机器 ip 端口。
- 192.0.0.3 上部署：ctor 实现类字节码文件、fourinone.jar，config.xml。其中 config 的 PARK 部分的 SERVERS 配置指定上面 parkserver 的 ip 端口（192.0.0.1:1888）。ctor 是嵌入式的类，它不需要指定自己的 ip 端口。

启动顺序：

- 1) parkserver
- 2) 多个 worker
- 3) ctor



注意

由于 demo 为了方便本地演示，默认配置都是 localhost，多机环境请改为实际 ip，避免出现连接问题。

配置原理说明如下：由于工人需要向职介者注册（parkserver），因此需要配置指定 ip 端口告诉 parkserver；由于工头和工人都需要知道 parkserver 地址，所以 parkserver 的 ip 端口需要配置为一致；由于工头通过 park 获取到工人 ip，然后直接跟工人交互，因此工人不需要知道工头 ip，工头本身也不是服务程序，所以工头启动不需要配置自己 ip 端口（参见前面的图 2-2）。

2.5.6 分布式计算自动部署的示例

对于一个分布式计算的应用，如果不需要自动部署，将工头工人程序文件分别部署到相应机器运行即可，不需要过多配置。

Fourinone 可以支持自动化 class 和 jar 包部署，class 和 jar 包只需放在工头机器上，各工人机器会自动获取并执行，兼容操作系统，不需要进行安全密钥复杂配置。

假设有一个分布式计算 job，包括三个程序文件：

- JobCtor：包工头实现
- JobWorker：农民工实现
- JobHelp：工人帮助类

如果需要自动部署，可以将上面三个 class 文件都放置在工头机器上，并在工头实现里指

定工人实现类。在 JobCtor 里通过下面代码设置：

```
wks[0].setWorker(new JobWorker());
```

运行步骤如下：

- 1) 启动 ParkServerDemo (它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定)

```
Java -classpath fourinone.jar; ParkServerDemo
```

- 2) 由于工人的实现类不放置在工人机器上，那么请在每台工人节点机器上启动 MigrantWorker 类，结果如图 2-21 所示：

```
java -cp fourinone.jar; com.fourinone.MigrantWorker localhost 2008 simpleworker
```

这里启动 MigrantWorker 指定的几个参数依次是“工人节点 IP 信息”、“端口信息”、“工人类型”。

如果是在本地模拟，可以在两个不同的目录进行，一个目录启动 MigrantWorker，另外一个目录包含有 JobCtor、JobWorker 和 JobHelp 字节码文件。

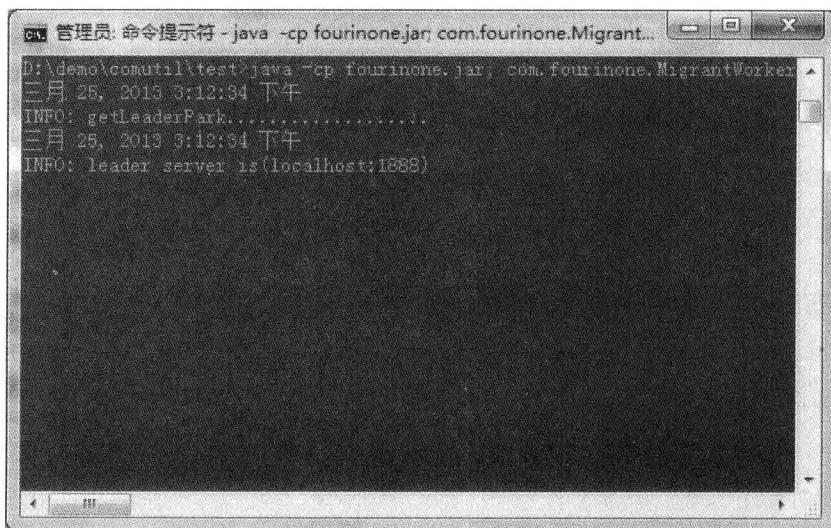


图 2-21 MigrantWorker

- 3) 运行 JobCtor，结果如图 2-22 所示：

```
java -cp fourinone.jar; JobCtor
```

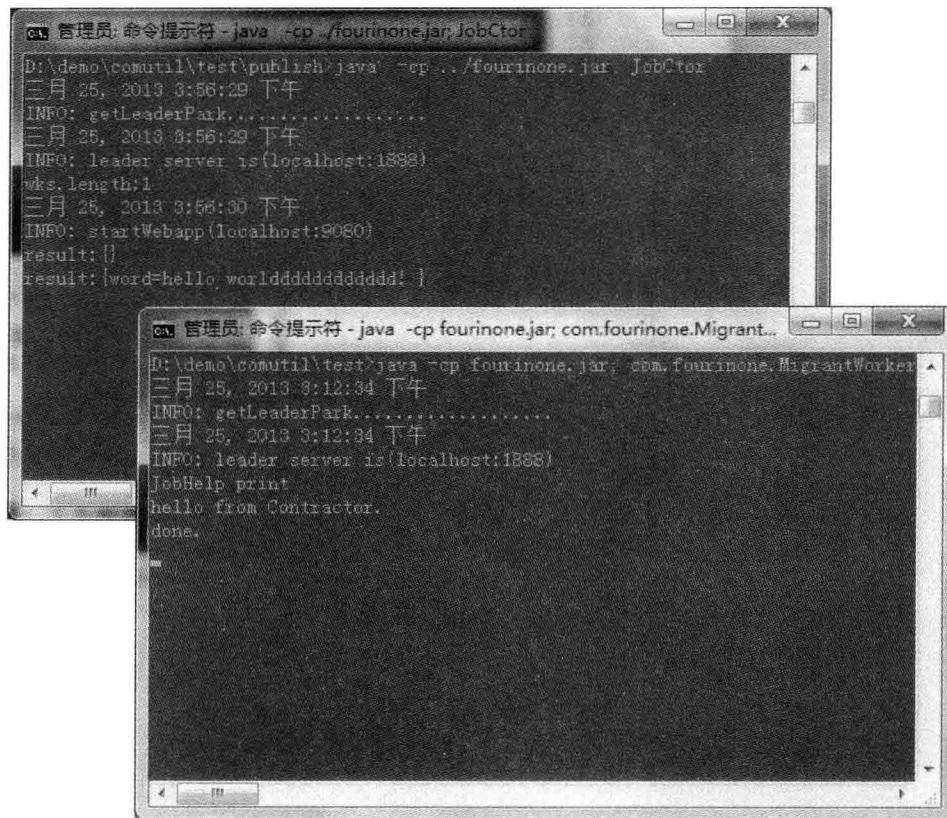


图 2-22 JobCtor

可以看到工头动态分发 JobWorker 和 JobHelp 到工人节点机器上执行，每台工人机器节点只需要有 fourinone.jar 和 config.xml 即可。

如果将上面三个 class 文件打包成 jar 包，需要在上面启动 MigrantWorker 时增加一个 jar 名称参数，表示会自动请求执行该 jar 包。



注意

如果使用自动部署，工头机器会自动启动 HTTP 服务，HTTP 服务 IP 和端口在 WEBAPP 部分的 SERVERS 配置，需要设置为工头机器 ip，并保持工头和工人机器的 SERVERS 配置一致。如果在同台机器上模拟自动部署，需要注意 parkServer 和工头不重复启动 http 服务（可以将 park 部分配置的 <STARTWEBAPP>false</STARTWEBAPP> 设置为关闭）。

有包名时注意：

- 1) 工头、工人在同一个包下比如 com.job，需要在包根目录启动工头：

```
java -cp fourinone.jar; com/job/JobCtor
```

2) 工头、工人不在同一个包下, 如:

```
x/job/JobCtor.class
x/com/job/JobWorker.class
```

x 为根目录, 在 x 目录下运行:

```
java -cp fourinone.jar; job/JobCtor
```

Demo 完整源码如下:

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
    public static void main(String[] args){
        BeanContext.startPark();
    }
}

// JobWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
import com.fourinone.FileAdapter;
public class JobWorker extends MigrantWorker
{
    public WareHouse doTask(WareHouse inhouse)
    {
        JobHelp.print();
        String word = inhouse.getString("word");
        System.out.println(word+" from Contractor.");
        // 停止 2 秒模拟执行任务
        try{Thread.sleep(2000L);}catch(Exception ex){}
        System.out.println("done.");
        return new WareHouse("word", word+" worldoooooooooooo! ");
    }
}

// JobCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;

public class JobCtor extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("simpleworker");
        System.out.println("wks.length:"+wks.length);
        wks[0].setWorker(new JobWorker());
    }
}
```

```

WareHouse wh = new WareHouse("word", "hello");
WareHouse result = wks[0].doTask(wh);
System.out.println("result:"+result);

while(true){
    if(result.isReady()){
        System.out.println("result:"+result);
        break;
    }
}
return null;
}

public static void main(String[] args)
{
    JobCtor a = new JobCtor();
    a.giveTask(null);
    a.exit();
}
}
}

```

2.5.7 计算过程中的故障和容灾处理

使用 Fourinone 可以完成大部分分布式并行计算需求，但是计算过程中的故障和容灾处理是怎么进行的呢，这里详细分析一下。

总的来说，Fourinone 框架不会在设计中抛弃错误不处理或者容忍错误导致框架崩溃，框架通常会捕获所有的错误反馈给开发者去处理，但是框架本身不自作主张，替开发者考虑处理方案，只有这样框架才能从特定场景中抽象出来，给开发者更灵活的发挥和去满足各种更复杂业务容错情况。

那么框架究竟关注和不关注哪个层面的故障呢？

并行计算过程中，通常有两种类型的故障：一种是系统故障引起的计算中断（宕机和网络故障），一种是业务逻辑意义上的错误数据。前者是框架关注的，后者是业务逻辑开发者关注的。

系统故障导致网络断掉或者宕机，框架会捕获故障信息并通告，工头在检验工人执行状态时会获知，并进行相应的业务上的故障处理，比如重发或者单独记录日志。业务逻辑意义上的错误数据，通常在工人的业务实现逻辑里去判断，比如计算结果的金额为负数是一个不符合业务要求的错误数据，这个是由开发者去控制，框架不做业务逻辑上的错误处理。

针对故障，框架又是怎样容灾的呢？

通常一个典型的分布式计算结构，由工头、工人、中介所组成，我们详细分析一下这几个角色在故障时各自如何容灾：

工头是嵌入式的，他不是一个服务程序，由嵌入他的系统 new 工头类并管理他的生命周期

期，工头不存在恢复或者容灾的概念，就好比我们写一个 Helloworld 的 main 函数，很少考虑程序运行到 hello, world 没有输出时就宕机了。但是如果嵌入工头的系统是一个定时执行的计算任务时，也许要考虑容灾，因为涉及单点问题，可以让两个工头竞争一个分布式锁实现（详细参考 3.6.2 分布式锁 demo）。

工人和职介所是服务程序，如果工人节点故障，职介所会实时感知，工头分配计算时会获取到最新活跃工人数量，如果是职介所节点故障，Fourinone 实现了领导者选举机制，会实时切换到备份职介所上（详细参考 3.6.1 统一配置管理主备领导者切换）。

换句话说，如果一个工人节点在计算开始前发生故障不可用，工头通过 getWaiting Workers 获取可用工人时不会包括该工人节点，因为职介所会感知每个工人的可用状态。

如果工人在计算过程中发生故障，框架会进行截获，然后提前返回计算结果，并设置结果的状态为异常。

也就是正常完成计算时：result.getStatus()==WareHouse.READY

计算过程发生故障中断时：result.getStatus()==WareHouse.EXCEPTION

这样工头就可以根据检查结果的状态，来做故障时的容灾处理。

实际上也可以在工人的 doTask 实现方法内部捕捉业务异常，由开发者根据程序实现自由决定。

以下 demo 演示了 Fourinone 计算过程中的故障容灾处理：

- **FaultCtor**: 是一个工头实现，它调用集群中一个工人 doTask 执行任务，然后轮询该结果，判断结果是否完成或者是否异常，如果结果状态为异常，则打印消息。实际上这里只是简单演示机制，现实场景中，可以将任务先记录，工人执行成功后再删除并跳转下一个任务，如果异常则继续重发其他工人执行该任务，或者采用其他故障策略，统一记录到错误日志，在其他时间再另行排查处理。
- **FaultWorker**: 是一个工人实现，它模拟了一个任务执行，睡眠了 8 秒钟，然后再制造一个空指针异常。该工人模拟了两种系统异常状况，计算过程中可以关闭它，或者等待它运行到空指针异常查看效果，注意这里 doTask 本身是不抛出和捕捉异常的，由框架去处理。

运行步骤：

1) 编译 demo 的 java 类：

```
Java -classpath fourinone.jar; *.java
```

2) 启动 ParkServerDemo (它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定)

```
Java -classpath fourinone.jar; ParkServerDemo
```

3) 运行 FaultWorker (传入端口号参数)

```
Java -classpath fourinone.jar; FaultWorker 2008
```

4) 运行 FaultCtor

```
Java -classpath fourinone.jar; FaultCtor
```

运行后工人进入 8 秒中“任务执行”，这时可以将该工人进程关闭，然后会查看到工头界面输出 something wrong about wks[0] result，说明框架已经屏蔽系统故障并反馈到任务结果的异常状态中，如果 8 秒中内不关闭，会引发另外一个空指针异常，产生同样的异常状态。

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// FaultWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
import com.fourinone.Workman;

public class FaultWorker extends MigrantWorker
{
    public WareHouse doTask(WareHouse inhouse)
    {
        System.out.println(inhouse.getString("word"));
        try{Thread.sleep(8000L); }catch(Exception ex){}
        String[] strs = null;
        System.out.println(strs.length);
        WareHouse wh = new WareHouse("word", "hello ");
        return wh;
    }

    public static void main(String[] args)
    {
        FaultWorker mw = new FaultWorker();
        mw.waitWorking("localhost",Integer.parseInt(args[0]),"faultworker");
    }
}
```

```

// FaultCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.ArrayList;

public class FaultCtor extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("faultworker");
        System.out.println("wks.length:"+wks.length);

        WareHouse wh = new WareHouse("word", "hello");
        WareHouse result = wks[0].doTask(wh);
        System.out.println("result:"+result);

        while(true){
            if(result.getStatus() == WareHouse.READY){
                System.out.println("result:"+result);
                break;
            }
            else if(result.getStatus() == WareHouse.EXCEPTION){
                System.out.println("something wrong about wks[0] result");
                //doTask(wh) again or put wh into log
                break;
            }
        }

        return null;
    }

    public static void main(String[] args)
    {
        FaultCtor a = new FaultCtor();
        a.giveTask(null);
        a.exit();
    }
}

```

2.5.8 计算过程中的相关时间属性设置

我们有时在计算过程中，发现 ParkServer 窗口输出的信息如图 2-23 所示。



图 2-23 工人忙碌提示

如果 ParkServer 窗口出现“INFO: _worker_faultworker.13D7C5BDC9A-11FF07AC4566 cant be deleted or not exist!”的信息，并不是框架出故障了，而是表明：您的工人当前太繁忙了，无法按照跟职介所约定的心跳时间保持联系，职介所长时间没有收到该名工人的联系，认为他已经不活跃了，将其从活跃工人列表中删除，于是出现了上面的提示信息。不过职介所将工人从活跃列表删除，并不影响本次计算，因为职介所将工人介绍给包工头后，计算过程中是包工头和工人直接交互，不再依赖职介所。受影响的是下次计算时，包工头再次到职介所获取工人时，职介所不会再推荐该名工人了。

在 config.xml 的 Park 部分配置里，有下面两个时间配置项：

```
<HEARTBEAT>3000</HEARTBEAT>
<MAXDELAY>0</MAXDELAY>
```

HEARTBEAT 是心跳时间，默认配置是 3000 毫秒，也就是 3 秒钟工人和职介所保持一次联系，可以根据计算需要调整这个心跳时间。

注意

在同台机器，工人、职介所、包工头共用一个配置文件时，修改 HEARTBEAT 即可。如果工人、职介所、职介所在不同机器，拥有各自的配置文件，需要将各自文件的 HEARTBEAT 配置为一致。

我们从上面注意到，工人和职介所失去联系，可能仅仅是因为计算太忙碌，而不是因为工人机器故障宕机，同样的情况还有网络波动等，因此，为了让职介所不轻易删除掉工人节

点，可以设置一个抢救期，只要是在抢救期内能恢复联系，那么不认为该工人节点死亡，过了抢救期，才判定死亡。

MAXDELAY 是抢救期时间，默认配置是 0 毫秒，也就是关闭状态，默认不进行抢救期判断，如果需要，修改 MAXDELAY 为一个大于 0 的值。

为了演示抢救期的效果，我们还是使用前面 FaultWorker 的例子，只不过是把配置项改为：

```
<HEARTBEAT>6</HEARTBEAT>
<MAXDELAY>8</MAXDELAY>
```

我们把心跳时间和抢救时间改的非常小，这个时间具体多小可以根据当前机器调整，心跳时间很小时，工人来不及和职介所联系，于是会达到我们演示效果的目的。

启动 ParkServerDemo 以及 FaultWorker，之后出现图 2-24 所示的结果。



图 2-24 心跳抢救时间设置

我们从上面的信息可以看到

```
WARNING: _worker_faultworker, 13D80B483C2-697540AA5E1 slow and weak heartbeat!
```

表明心跳联系缓慢，已经处在抢救期内，如果超过了抢救期，那么最后会判定节点死亡并从职介所删除。

在计算很繁忙的时候，为了不影响工人和职介所保持联系，可以将 HEARTBEAT 时间设置的比较长一些，比如 30 秒或者以上，但是也意味着集群节点真的故障宕机，需要 30 秒延时才能感知，因此，可以通过进一步配置 MAXDELAY 抢救时间，既避免工人计算繁忙时跟职介所失去联系，也可以避免集群状态感知的太长时间延迟。

2.5.9 如何在一台计算机上一次性启动多个进程

我们之前看到一个完整的并行计算应用，需要启动很多个实例，parkserver、多个工人、工头，很多时候需要在一台计算机上启动多个工人，一般都是手工启动，特别是使用 Linux 终端时，一个窗口一个窗口登陆上去启动很麻烦，当然也可以写一个批处理脚本一次性启动多个工人，这里我们介绍通过使用 Fourinone 自带的程序 API 来启动多个进程。

1. 如何串行或并行的启动进程

BeanContext 提供了 start 和 tryStart 两种方式启动进程，start 进程如下：

```
public static int start(String... params)
public static int start(Map env, FileAdapter fa, String... params)
```

start 为串行方式，启动多个需要先后依次完成，params 为命令参数，可以将一行脚本命令按照空格分成多个参数输入，比如：javac *.java，写为：

```
start( "javac" , " *.java" );
```

也可以通过 Map env 指定命令运行环境变量和通过 FileAdapter fa 指定程序运行目录：

```
public static StartResult<Integer> tryStart(String... params)
public static StartResult<Integer> tryStart(Map env, FileAdapter fa, String... params)
```

tryStart 的使用跟 start 类似，只不过它是并行方式，启动多个无须先后等待，可以同时执行，它返回一个 StartResult 对象，可以通过轮询 StartResult 的状态查看进程是否成功完成结束：

- StartResult.getStatus() == StartResult.NOTREADY，代表进程还在运行中，未完成或结束
- StartResult.getStatus() == StartResult.READY，代表进程已经完成结束
- StartResult.getStatus() == StartResult.EXCEPTION，代表进程运行出了错误



注意

如果一个进程是服务监听的，那么它启动后会一直堵塞，因此通过 start 方式启动不会返回，如果通过 tryStart 方式启动可以立即返回，但是 StartResult 的状态一直为 NOTREADY（进程运行中），所以一个服务化进程启动后不能通过 StartResult 的状态为 READY 来判断它是否启动完成，可以根据进程启动耗时设置一个等待时间等待它启动就绪，详见下面 demo。

2. 如何杀死进程和超时杀死进程

StartResult 提供了 kill 方法杀死它所属的进程：public void kill()

也可以在获取状态时，传入一个超时时间去比对判断是否超时，如果超时那么会自动杀死该进程，并返回异常状态（但是 kill 返回是完成状态）：public int getStatus(long timeout)，这里 timeout 为毫秒数。

比如在轮询获取状态时每次输入一个 30 秒时间，getStatus(30000)，那么进程运行超过了 30 秒，会被强行中止并返回。

这个方法与后面的 2.5.11 节谈到的中止工人计算方式不同，这里是真正的杀死进程方式，而不是请求通知方式。

注意

start 和 tryStart 使用非常灵活，可以启动任何进程脚本命令或者批处理命令，但是需要注意父子进程的关系，这里只能杀死父进程，但是杀不了由父进程启动的子进程，比如启动了一个 cmd.bat 的批处理命令，如果使用 kill 中止，只能杀死 cmd.bat 这个进程本身，但是无法杀死在这个批处理中启动的其他进程。因此，如果需要使用 kill，最好不要有进程嵌套启动，请转换为一个个的单独进程启动。

3. 如何输出进程运行的日志信息

如果采用了 start 方式启动进程，那么默认会在系统窗口输出日志，因为多个 start 会串行先后输出日志，不存在冲突。但是如果采用 tryStart 方式启动进程，由于是并行方式同时执行，如果都在系统窗口输出日志存在冲突，导致日志信息错乱，因此默认不输出。

但是 StartResult 提供了 print 方法可以单独异步方式输出到一个文件中：

```
public void print(String logpath)
```

logpath 为文件路径，可以是绝对路径，也可以是相对路径，比如：print("log/worker.log");

注意

实际上，对于批处理命令，也可以在启动时通过加入 “>>” 参数方式输出日志，比如：tryStart(“build.bat”，“>>log/park.log”，“2>&l”），但是这种方式不是所有情况都适合，如果是启动 java 或者 javac 这样的命令使用 “>>”，会被当成一个普通的 java 输入参数，而不会被 shell 识别为重定向输出，因为 java 不是 .bat 批处理命令，无法启动 shell 环境。

下面这个 demo 演示了在一个 HelloMain 程序里面运行前面的“互相 say hello 例子”，所有的进程都在在一个 main 函数里启动、运行和中止退出。

运行步骤：输入如下命令，结果如图 2-25 所示：

```
java -cp fourinone.jar; HelloMain
```

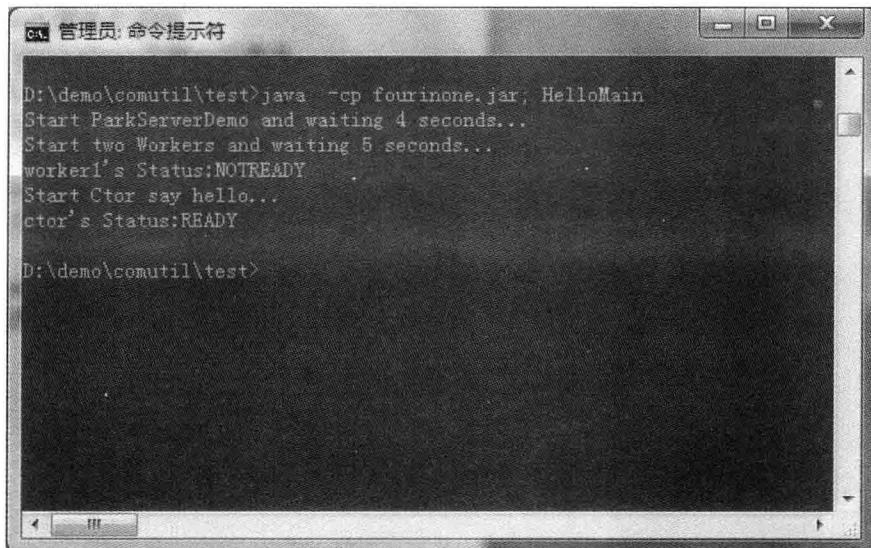


图 2-25 HelloMain

HelloMain 首先使用 java 命令启动 ParkServerDemo，并且输出日志到 log/park.log，并等待 4 秒钟保证 ParkServer 启动完成。

然后启动两个 HelloWorker，分别输出日志到 log/worker1.log、log/ worker2.log，并等待 5 秒钟保证两个 HelloWorker 启动完成，注意观察 worker1 启动完成后打印它的状态仍然为 NOTREADY，因为它是一个服务化进程。

最后启动 HelloCtor，输出日志到 log/ctor.log，由于 HelloCtor 不是一个服务化进程，因此可以通过轮询判断它的结果是否为 READY，如果已完成，那么 kill 掉以上 Park Server 和 2 个 HelloWorker 进程，HelloCtor 本身不需要 kill，因为非服务化进程运行结束会退出。

如果打开进程管理器观察，会发现运行时一共有 5 个 java 进程启动（一个 main 函数本身的主进程，1 个 parkserver、2 个 HelloWorker 和 1 个 HelloCtor 共 4 个子进程），运行结束，这 5 个 java 进程全部消失。如图 2-26 所示。

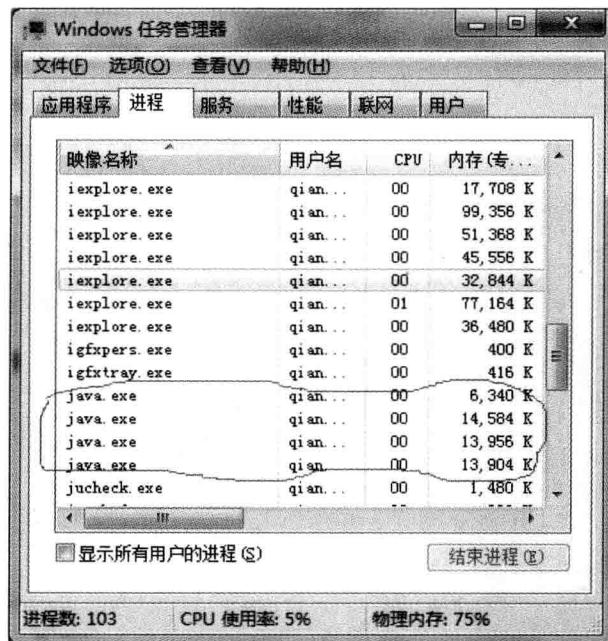


图 2-26 进程查看器

打开 log 目录下，可以看到生成了 park.log、worker1.log、worker2.log、ctor.log4 个日志文件，里面记录了进程运行过程的信息，如图 2-27 所示。



图 2-27 日志文件

完整 demo 源码如下：

```
//HelloMain
import com.fourinone.StartResult;
```

```
import com.fourinone.BeanContext;

public class HelloMain
{
    public static void main(String[] args)
    {
        //five process:a main process and four child process
        System.out.println("Start ParkServerDemo and waiting 4 seconds...");
        StartResult<Integer> parkserver =
            BeanContext.tryStart("java","-cp","fourinone.jar;","ParkServerDemo");
        parkserver.print("log/park.log");
        try
        {
            Thread.sleep(4000);
        }
        catch(Exception ex)
        {}

        System.out.println("Start two Workers and waiting 5 seconds...");
        StartResult<Integer> worker1 =
            BeanContext.tryStart("java","-cp","fourinone.jar;","HelloWorker","work
            er1","localhost","2008");
        worker1.print("log/worker1.log");
        StartResult<Integer> worker2 =
            BeanContext.tryStart("java","-cp","fourinone.jar;","HelloWorker","work
            er2","localhost","2009");
        worker2.print("log/worker2.log");
        try
        {
            Thread.sleep(5000);
        }
        catch(Exception ex)
        {}
        System.out.println("worker1's Status:"+worker1.getStatusName());

        System.out.println("Start Ctor say hello...");
        StartResult<Integer> ctor =
            BeanContext.tryStart("java","-cp","fourinone.jar;","HelloCtor");
        ctor.print("log/ctor.log");
        while(true)
        {
            if(ctor.getStatus()!=StartResult.NOTREADY)
            {
                System.out.println("ctor's Status:"+ctor.getStatusName());
                parkserver.kill();
                worker1.kill();
                worker2.kill();
                break;
            }
        }
    }
}
```

2.5.10 如何调用 C/C++ 程序实现

我们知道，工头的 giveTask 和工人的 doTask 都是由开发者自己实现，如果采用 C/C++ 实现的逻辑如何跟基于 Java 的框架交互呢？

方式一、采用脚本调用输出日志文件方式。

最直接的方式采用上一节讲述的进程调用方式，在工人实现里使用 start 或者 tryStart 调用 C/C++ 的运行脚本，脚本运行结果输出到日志文件里，然后从工人通过获取日志文件的内容并返回给工头。

方式二、jni 方式实现 C/C++ 和 Java 交互。

过程如下：

- 1) 编写带有 native 声明的方法的 Java 类。
- 2) 使用 javac 命令编译所编写的 Java 类。
- 3) 使用 javah Java 类名生成扩展名为 h 的头文件。
- 4) 使用 C/C++ 实现本地方法。
- 5) 将 C/C++ 编写的文件生成动态连接库。

在第 4 步中，C/C++ 实现可以通过 JNIEnv * env 去操作 Java 类和方法，JNIEnv 可以当做一个面向 C/C++ 的 jni 库。

方式三、socket 的 TCP/IP 通信方式。

将 C 实现的算法逻辑包装成 Socket Server，通过在工人的 doTask 里发送 TCP 通信包到 C 服务端，通过网络通信的方式实现交互。

综合建议：第一种方式最简单直接，对开发者没有太多要求，特别是 C/C++ 实现的算法本身的输出结果就是文件的情况，采用第一种方式最合适。只有输入输出程序对象希望实现数据转换时才考虑第二、第三种方法。

2.5.11 如何中止工人计算和超时中止

在并行计算的典型应用中，通常会让多计算机各自寻解或者并行搜索，当其中一台计算机找到解后，应该中止其他计算机继续计算；或者就是在某一个时间范围内寻找，超时便要全部中止。所以计算过程中对工人计算的中止功能是非常有用的（见图 2-28）。具体实施如下：

- 工人的中止功能是通过提供 interrupt 和 isInterrupted 方法实现的。
- 包工头调用 interrupt 方法中止工人计算。
- 工人检查 isInterrupted 方法的 boolean 返回值响应中止请求并返回结果。

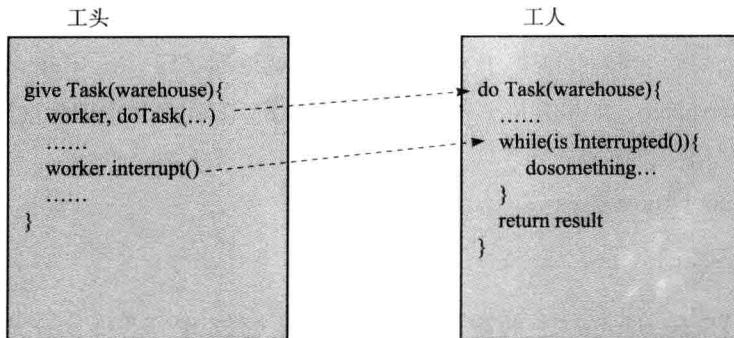


图 2-28 工人计算中止

详细过程描述：包工头在调用工人的 `doTask` 后，轮询结果是否完成，如果其中一个完成，可以通过调用其他工人的 `interrupt` 方法进行中止任务，在工人的 `doTask` 实现里面，需要通过 `isInterrupted` 判断是否被中止，如果是，停止计算并返回。

要注意以下问题：

1) 跟强行杀死进程或者线程的实现方式不同（请区别“一次性启动多个工人”里谈到的杀死进程方式），`interrupt` 在这里实际上是一种请求通知机制，由工头根据计算过程的进展发起中止请求，由工人在 `doTask` 实现逻辑中检查 `isInterrupted` 响应该请求（如果工人不听指挥拒绝执行请求，那包工头也没办法）。

2) 为什么要这样设计而不让框架直接强行杀死进程呢，这是为了让开发者能自由控制中止的策略和可以在中止时做一些保存/备份/日志等善后工作，并可以决定正常运行完成和中止时分别返回什么结果给工头，从而有更大处理的灵活度。

如果工人是公共服务状态（配置文件 `worker` 部分的 `<SERVICE>true</SERVICE>`，默认为 `false`），调用 `interrupt` 无效，并会产生 `InterruptedException`，因为公共服务状态下的工人不能被中止，它需要一直做为服务程序存在。

另外，工头对 `interrupt` 的调用需要在 `doTask` 的执行过程中，否则没有效果，因为 `interrupt` 调用后会检查当前是否有任务在执行，如果尚未发生任务调用或者已经执行完成，便认为无效。

这个 demo 演示了一个查找随机数的例子，由几台机器同时各自获取 10 万以内的随机数，看谁最先获取到 888 这个数字，一旦某台机器获取到，中止其他机器的寻找，整体完成计算。

`CancelCtor`：是一个工头实现，它的 `giveTask` 实现中，首先获取集群工人数量，然后调用各工人的 `doTask` 开始计算，并将各自结果保存起来轮询检查，它的程序结构使用了 3 个 `for` 循环完成检查和中止，第一个 `for` 用于记录完成的结果数，第二个 `for` 轮询各结果是否完成，当找到 888 的结果后，用第三个 `for` 中止掉其他工人计算。

CancelWorker: 是一个工人实现，它的 doTask 实现中通过一个 while 循环不断的生产 10 万以内的随机数，然后判断是否等于 888，如果找到就返回结果。在 while 循环同时，它还会不断检查 isInterrupted，如果其他工人已经找到该数字，它便会马上中止计算并返回。

运行步骤（在本机模拟）：

1) 编译 demo 的 Java 类：

```
javac -classpath fourinone.jar; *.java
```

2) 启动 ParkServerDemo (它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定)：

```
java -classpath fourinone.jar; ParkServerDemo
```

3) 运行一到多个 CancelWorker (传入一个端口号参数区分不同工人，结果如图 2-29 所示)：

```
java -classpath fourinone.jar; CancelWorker 2008
java -classpath fourinone.jar; CancelWorker 2009
java -classpath fourinone.jar; CancelWorker 2010
```



图 2-29 CancelWorker

4) 运行 CancelCtor, 结果如图 2-30 所示:

```
Java -classpath fourinone.jar; CancelCtor
```



图 2-30 CancelCtor

可以看到，在第 1 个工人找到 888 后，整体结束计算，工头将结果输出。

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// CancelWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
import com.fourinone.FileAdapter;
import java.util.List;
import java.util.Collections;
import java.util.Random;

public class CancelWorker extends MigrantWorker
```

```

{
    public WareHouse doTask(WareHouse inhouse)
    {
        int n = 0;
        Random rd = new Random();
        while(!isInterrupted()){
            n=rd.nextInt(100000);
            System.out.println(n);
            if(n==888)
                break;
        }
        return new WareHouse("result", n);
    }

    public static void main(String[] args)
    {
        CancelWorker mw = new CancelWorker();
        mw.waitWorking("localhost",Integer.parseInt(args[0]),"cancelworker");
    }
}

// CancelCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.ArrayList;

public class CancelCtor extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("cancelworker");
        System.out.println("wks.length:"+wks.length);

        WareHouse[] hmarr = new WareHouse[wks.length];
        for(int i=0;i<wks.length;i++){
            hmarr[i] = wks[i].doTask(new WareHouse());//3
        }

        for(int j=0;j<hmarr.length;){//记录完成的结果数
            for(int i=0;i<wks.length;i++){//检查结果是否完成
                if(hmarr[i]!=null&&hmarr[i].getStatus()==WareHouse.READY){
                    System.out.println(i+"："+hmarr[i]);
                    //找到 888 后，停止其他工人计算
                    if((Integer)hmarr[i].getObj("result")==888){
                        for(int k=0;k<wks.length;k++){
                            if(k!=i)
                                wks[k].interrupt();
                        }
                    }
                    hmarr[i]=null;
                    j++;
                }
            }
        }
    }
}

```

```

    return null;
}

public static void main(String[] args)
{
    CancelCtor a = new CancelCtor();
    a.giveTask(null);
    a.exit();
}
}
}

```

如果我们要定义一个计算时间，超时便中止计算，可以通过四种方法完成：

方法一：工头分配完任务后开始看时间，超时便指挥各工人停止。

工头自行检查超时，工头在调用 doTask 时开始计时，每次轮询结果时检查是否超时，超时便调用 interrupt 通知工人进行中止。

方法二：工头要求工人自觉，工人自己看时间，超时自觉停止。

工人自行检查超时，在工人的 doTask 实现逻辑里加入计时检查，如果超时便退出返回结果（注意和方法三的区别）。

方法三：框架调用超时抛异常方式。

工人不自行检查超时，框架检查到工人 doTask 计算超时抛出系统异常，中断任务调用。如果要使用该方式，请将配置文件 config.xml 中：

```

<PROPSROW DESC="WORKER">
<TIMEOUT DESC="FALSE">2</TIMEOUT>
</PROPSROW>

```

TIMEOUT DESC 设置为 TRUE，2 表示超时时间，小时为单位，这里默认是 2 小时，也就是如果工人执行 doTask 超过 2 小时仍未完成，框架放弃调用，抛出系统异常。

方法四：doTask 的 interrupt 方式。

为了方便超时中止，框架也提供了一个便利方法：

```
public WareHouse doTask(WareHouse inhouse, long timeoutseconds);
```

也就是在调用 doTask 时，可以传入一个超时时间参数（秒为单位），它的实际效果就相当于 doTask+interrupt，超时自动调用 interrupt 请求工人中止。

2.5.12 使用并行计算大幅提升递归算法效率

无论什么样的并行计算方式，其终极目的都是为了有效利用多机多核的计算能力，并能灵活满足各种需求。相对于传统基于单机编写的运行程序，如果使用该方式改写为多机并行

程序，能够充分利用多机多核 CPU 的资源，使得运行效率得到大幅度提升，那么这是一个好的靠谱的并行计算方式，反之，又难使用又难直接看出并行计算优势，还要耗费大量学习成本，那就不是一个好的方式。

由于并行计算在互联网应用的业务场景都比较复杂，如海量数据商品搜索、广告点击算法、用户行为挖掘，关联推荐模型等等，如果以真实场景举例，初学者很容易被业务本身的复杂度绕晕了头。因此，我们需要一个通俗易懂的例子来直接看到并行计算的优势。

数字排列组合是个经典的算法问题，它很通俗易懂，适合不懂业务的人学习，我们通过它来发现和运用并行计算的优势，可以得到一个很直观的体会，并留下深刻的印象。问题如下：

请写一个程序，输入 M，然后打印出 M 个数字的所有排列组合（每个数字为 1, 2, 3, 4 中的一个）。比如：M=3，输出：

1, 1, 1

1, 1, 2

.....

4, 4, 4

共 64 个



注意

这里是使用计算机遍历出所有排列组合，而不是求总数，如果只求总数，可以直接利用数学公式进行计算了。

1. 单机解决方案

通常，我们在一台电脑上写这样的排列组合算法，一般用递归或者迭代来做，我们先分别看看这两种方案。

(1) 单机递归

可以将 n ($1 \leq n \leq 4$) 看做深度，输入的 m 看做广度，得到以下递归函数（完整代码见下面源码 CombTest.java）

```
public void comb(String str) {
    for(int i=1;i<n+1;i++){
        if(str.length()==m-1) {
            System.out.println(str+i);
            total++;
        }else
            comb(str+i);
    }
}
```

但是当 m 数字很大时，会超出单台机器的计算局限导致缓慢，太大数字的排列组合在一台计算机上几乎很难运行出，不光是排列组合问题，其他类似遍历求解的递归或回溯等算法也都存在这个问题，如何突破单机计算性能的问题一直困扰着我们。

(2) 单机迭代

我们观察到，求的 m 个数字的排列组合，实际上都可以在 $m-1$ 的结果基础上得到。比如：

$m=1$ ，得到排列为 1,2,3,4，记录该结果为 $r(1)$

$m=2$ ，可以由 $(1,2,3,4) * r(1) = 11,12,13,14,21,22, \dots, 43,44$ 得到，记录该结果为 $r(2)$

由此， $r(m) = (1,2,3,4) * r(m-1)$

如果我们从 1 开始计算，每轮结果保存到一个中间变量中，反复迭代这个中间变量，直到算出 m 的结果为止，这样看上去也可行，仿佛还更简单。

但是如果我们估计一下这个中间变量的大小，估计会吓一跳，因为当 $m=14$ 的时候，结果已经上亿了，一亿个数字，每个数字有 14 位长，并且为了得到 $m=15$ 的结果，我们需要将 $m=14$ 的结果存储在内存变量中用于迭代计算，无论以什么格式存，几乎都会遭遇到单台机器的内存局限，如果排列组合数字继续增大下去，结果便会内存溢出了。

2. 分布式并行计算解决方案

我们看看如何利用多台计算机来解决该问题，同样以递归和迭代的方式进行分析。

(1) 多机递归

做分布式并行计算的核心是需要改变传统的编程设计观念，将算法重新设计按多机进行拆分和合并，有效利用多机并行计算优势去完成结果。

我们观察到，将一个 n 深度 m 广度的递归结果记录为 $r(n,m)$ ，那么它可以由 $(1,2,\dots,n) * r(n,m-1)$ 得到：

$$r(n,m) = 1 * r(n,m-1) + 2 * r(n,m-1) + \dots + n * r(n,m-1)$$

假设我们有 n 台计算机，每台计算机的编号依次为 1 到 n ，那么每台计算机实际上只要计算 $r(n,m-1)$ 的结果就够了，这里实际上将递归降了一级，并且让多机并行计算。

如果我们有更多的计算机，假设有 $n*n$ 台计算机，那么：

$$r(n,m) = 11 * r(n,m-2) + 12 * r(n,m-2) + \dots + nn * r(n,m-2)$$

拆分到 $n*n$ 台计算机上就将递归降了两级了。

可以推断，只要我们的机器足够多，能够线性扩充下去，我们的递归复杂度会逐渐降低，并且并行计算的能力会逐渐增强。如图 2-31 所示。

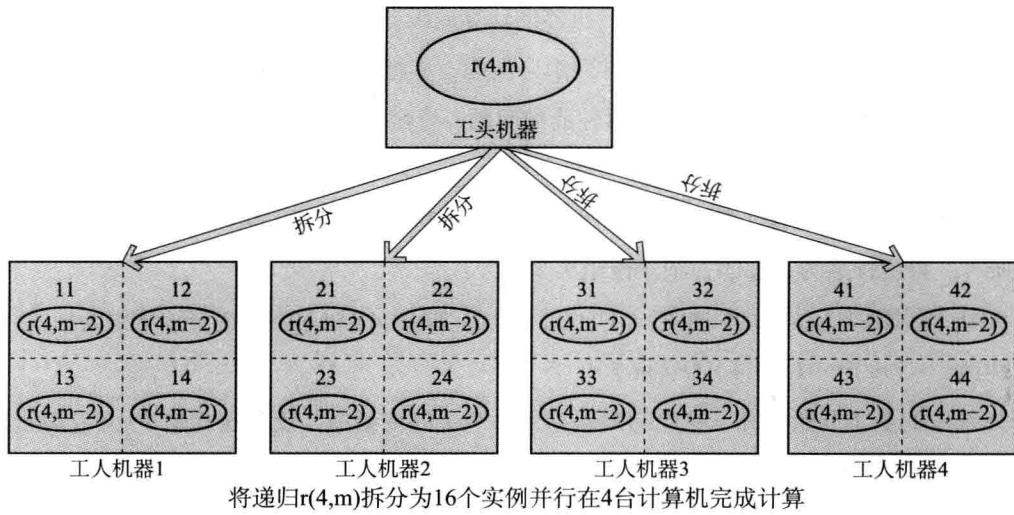


图 2-31 递归多机多实例并行计算

这里是进行拆分设计的分析，假设每台计算机只跑 1 个实例，实际上每台计算机可以跑多个实例（如图 2-3 所示），我们从下面的例子可以看到，这种并行计算的方式相对传统单机递归有大幅度的效率提升。

- ParkServerDemo：负责工人注册和分布式协调。
- CombCtor：是一个包工头实现，它负责接收用户输入的 m，并将 m 保存到变量 comb，和线上工人总数 wknum 一起传给各个工人，下达计算命令，并在计算完成后累加每个工人的结果数量，得到一个结果总数。
- CombWorker：是一个工人实现，它接收到工头发的 comb 和 wknum 参数用于递归条件，并且通过获取自己在集群的位置 index，作为递归初始条件用于降级，它找到一个排列组合会直接在本机输出，但是计数保存到 total，然后将本机的 total 发给包工头统计总体数量。

运行步骤：

为了方便演示，我们在一台计算机上运行：

1) 启动 ParkServerDemo：它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定，代码如下所示：

```
java -cp fourinone.jar; ParkServerDemo
```

2) 启动 4 个 CombWorker 实例：传入 2 个参数，依次是 ip 或者域名、端口（如果在同一台机器可以 ip 相同，但是端口不同），这里启动 4 个工人是由于 $1 \leq n \leq 4$ ，每个工人实例

刚好可以通过集群位置 index 进行任务拆分，代码如下：

```
java -cp fourinone.jar; CombWorker localhost 2008
java -cp fourinone.jar; CombWorker localhost 2009
java -cp fourinone.jar; CombWorker localhost 2010
java -cp fourinone.jar; CombWorker localhost 2011
```

3) 运行 CombCtor 查看计算时间和结果，如下所示：

```
java -cp fourinone.jar; CombCtor 14
```

表 2-4 是在一台普通 4CPU 双核 2.4GHz 内存 4G 开发机上和单机递归 CombTest 的测试对比。

表 2-4 测试对比

	M=14	M=15	M=16	CPU 利用率
单机递归计算	10 秒	41 秒	169 秒	29%
单机并行计算	6 秒	26 秒	112 秒	99%
多机并行计算	按机器数量成倍提升效率			99%

通过测试结果我们可以看到：

- 1) 可以推断，由于单机的性能限制，无法完成 m 值很大的计算。
- 2) 同是单机环境下，并行计算相对于传统递归提升了将近 1.6 倍的效率，随着 m 的值越大，节省的时间越多。
- 3) 单机递归的 CPU 利用率不高，平均 20% ~ 30%，在多核时代没有充分利用机器资源，造成 CPU 闲置浪费，而并行计算则能打满 CPU，充分利用机器资源。
- 4) 如果是多机分布式并行计算，在 4 台机器上，采用 4×4 的 16 个实例完成计算，效率还会成倍提升，而且机器数量越多，计算越快。
- 5) 单机递归实现和运行简单，使用 C 或者 Java 写个 main 函数完成即可，而分布式并行程序，则需要利用并行框架，以包工头 + 多个工人的全新并行计算思想去完成。

(2) 多机迭代

我们最后看看如何构思多机分布式迭代方式实现。

思路一：根据单机迭代的特点，我们可以将 n 台计算机编号为 1 到 n

第一轮统计各工人发送编号给工头，工头合并得到第一轮结果 $\{1, 2, 3, \dots, n\}$

第二轮，工头将第一轮结果发给各工人做为计算输入条件，各工人根据自己编号累加，返回结果给工头合并，得到第二轮结果： $\{11, 12, 13, 1n, \dots, n1, n2, n3, nn\}$

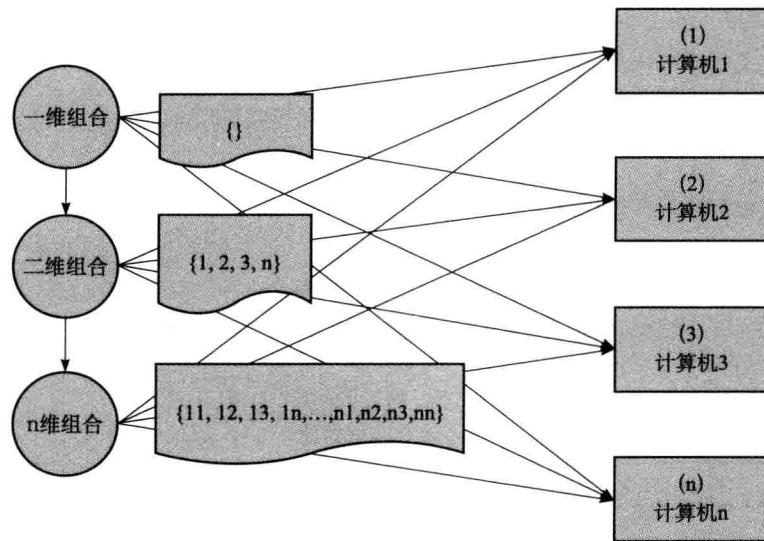


图 2-32 多机迭代

这样迭代下去，直到 m 轮结束，如图 2-32 所示。

但很快就会发现，工头合并每轮结果是个很大的瓶颈，很容易内存不够导致计算崩溃。

思路二：如果对思路一改进，各工人不发中间结果给工头合并，而采取工人之间互相合并方式，将中间结果按编号分类，通过 `receive` 方式（工人互相合并及 `receive` 使用可参见前面 `sayhello demo`），将属于其他工人编号的数据发给对方。这样一定程度避免了工头成为瓶颈，但是经过实践发现，随着迭代变大，中间结果数据越来越大，工人合并耗用网络也越来越大，如果中间结果保存在各工人内存中，随着 m 变的更大，仍然存在内存溢出危险。

思路三：继续改进思路二，将中间结果变量不保存内存中，而每次写入文件（详见第六章对分布式文件的简化操作），这样能避免内存问题，但是增加了大量的文件 IO 消耗。虽然能运行出结果，但是并不高效。

总结：或许分布式迭代在这里并不是最好的做法，上面的多机递归更合适。由于迭代计算的特点，需要将中间结果进行保存，作为下一轮计算的条件，如果为了利用多机并行计算优势，又需要反复合并产生中间结果，所以导致对内存、带宽、文件 IO 的耗用很大，处理不当容易造成性能低下。

我们早已经进入多 CPU 多核时代，但是我们的传统程序设计和算法还停留在过去单机应用，因此合理利用并行计算的优势来改进传统软件设计思想，能为我们带来更大效率的提升。

以下是分布式并行递归的 demo 源码：

```
// CombTest
```

```

import java.util.Date;
public class CombTest
{
    int m=0,n=0,total=0;
    CombTest(int n, int m){
        this.m=m;
        this.n=n;
    }
    public void comb(String str)
    {
        for(int i=1;i<n+1;i++){
            if(str.length()==m-1) {
                //System.out.println(str+i);// 打印出组合序列
                total++;
            }
            else
                comb(str+i);
        }
    }

    public static void main(String[] args)
    {
        CombTest ct = new CombTest(Integer.parseInt(args[0]),
                                   Integer.parseInt(args[1]));
        long begin = (new Date()).getTime();
        ct.comb("");
        System.out.println("total:"+ct.total);
        long end = (new Date()).getTime();
        System.out.println("time:"+((end-begin)/1000)+"s");
    }
}

// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
    public static void main(String[] args){
        BeanContext.startPark();
    }
}

// CombCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.Date;
public class CombCtor extends Contractor
{
    public WareHouse giveTask(WareHouse wh)
    {
        WorkerLocal[] wks = getWaitingWorkers("CombWorker");
        System.out.println("wks.length:"+wks.length+";"+wh);
        wh.setObj("wknum",wks.length);
        WareHouse[] hmarr = doTaskBatch(wks, wh);// 批量执行任务，所有工人完成才返回
        int total=0;
        for(WareHouse hm:hmarr)

```

```

        total+=(Integer)hm.getObj("total");
        System.out.println("total:"+total);
        return wh;
    }

    public static void main(String[] args)
    {
        CombCtor a = new CombCtor();
        WareHouse wh = new WareHouse("comb", Integer.parseInt(args[0]));
        long begin = (new Date()).getTime();
        a.doProject(wh);
        long end = (new Date()).getTime();
        System.out.println("time:"+ (end-begin)/1000+"s");
        a.exit();
    }
}

//CombWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
public class CombWorker extends MigrantWorker
{
    private int m=0,n=0,total=0,index=-1;

    public WareHouse doTask(WareHouse wh)
    {
        total=0;
        n = (Integer)wh.getObj("wknum");
        m = (Integer)wh.getObj("comb");
        index = getSelfIndex()+1;
        System.out.println("index:"+index);
        comb(index+"");
        System.out.println("total:"+total);
        return new WareHouse("total",total);
    }

    public void comb(String str)
    {
        for(int i=1;i<n+1;i++){
            if(str.length()==m-1) {
                //System.out.println(str+i);// 打印出组合序列
                total++;
            }
            else
                comb(str+i);
        }
    }

    public static void main(String[] args)
    {
        CombWorker mw = new CombWorker();
        mw.waitWorking(args[0],Integer.parseInt(args[1]),"CombWorker");
    }
}

```

2.5.13 使用并行计算求圆周率 π

关于圆周率大家再熟悉不过了，我们从课本上学习到早在一千多年前，祖冲之将圆周率计算到 3.1415926 到 3.1415927 之间……计算机诞生后，计算圆周率被用来检测计算机的硬件性能，昼夜燃烧 CPU 看会不会出问题……另外一些人也想看看这个无限延伸的神秘数字背后是否有规律，是否能发现一些宇宙的秘密……

提起圆周率，不能不提及 Fabrice Bellard，他被认为是一位计算机天才，在业界有着重要的影响。1996 年他编写了一个简洁但是完整的 C 编译器和一个 Java 虚拟机 Harissa。Fabrice Bellard 发明的 TinyCC 是 GNU/Linux 环境下最小的 ANSI C 语言编译器，是目前号称编译速度最快的 C 编译器。Fabrice Bellard 杰作众多且涉及广泛，1998 年编写了一个简洁的 OpenGL 实现 TinyGL，2003 年开发了 Emacs 克隆 QEmacs，2005 年还设计了一个廉价的数字电视系统。如图 2-33 所示。

Fabrice Bellard 使用一台普通的台式电脑，完成了冲击由超级计算机保持的圆周率运算记录的壮举，他使用台式机将圆周率计算到了小数点后 2.7 万亿位，超过了由目前排名世界第 47 位的 T2K Open 超级计算机于去年 8 月份创造的小数点后 2.5 万亿位的记录。

Bellard 使用的电脑是一台基于 2.93GHz Core i7 处理器的电脑，这部电脑的内存容量是 6GB，硬盘则使用的是五块 RAID-0 配置的 1.5TB 容量的希捷 7200.11，系统运行 64 位 Red Hat Fedora 10 操作系统，文件系统则使用 Linux 的 ext4。

这次计算出来的圆周率数据占去了 1137GB 的硬盘容量，Bellard 花了 103 天的时间计算出了这样的结果。

计算圆周率的方法有很多种，下面介绍几种。

(1) 微积分割圆法

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x) \Big|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4}$$

令函数 $f(x) = 4/(1+x^2)$

则有 $\int_0^1 f(x)dx = \pi$

$$\begin{aligned}\pi &\approx \sum_{i=1}^n / \left(\frac{2 \times i - 1}{2 \times N} \right) \times \frac{1}{N} \\ &= \frac{1}{N} \times \sum_{i=1}^n f\left(\frac{i-0.5}{N}\right)\end{aligned}$$

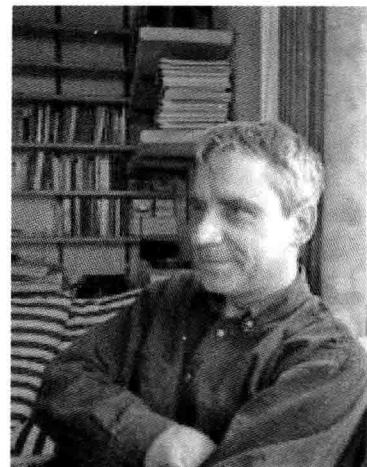
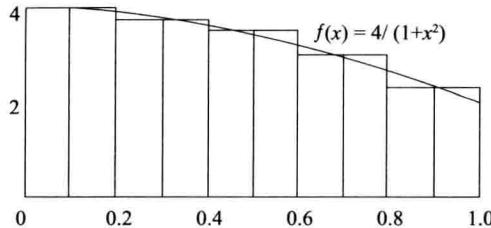


图 2-33 Fabrice Bellard



(2) 利用便于计算机计算的丘德诺夫斯基公式法

$$\pi = \frac{426880\sqrt{10005}}{\sum_{n=0}^{\infty} \frac{(6n)!(13591409+545140134n)}{(3n)!(n!)^3(-640320)^{3n}}}$$

不过这些计算方法都比较复杂，难以让读者理解和使用并行计算来求。所幸数学上的泰勒级数是个好东西，它将微积分的东西改成用无限级数来表示，这样很容易进行并行计算分解：

$$\pi = 4 * \sum (-1)^n + 1 / (2n - 1)$$

或者写为：

$$\pi = 4 * (1 - 1/3 + 1/5 - 1/7 + \dots)$$

也可以得到：

$$\pi n = \pi n - 1 + (-1)^n + 1 / (2n - 1), \text{ 也就是可以通过迭代前面的 } \pi \text{ 值去求当前 } \pi \text{ 值。}$$

我们根据上面公式先写个单机程序，如下所示：

```
public class PiTest
{
    public static void main(String[] args)
    {
        double pi=0.0;
        for(double i=1.0;i<1000000001d;i++){
            pi += Math.pow(-1,i+1) / (2*i-1);
        }
        System.out.println(4*pi);
    }
}
```

运行以上程序，并对照 π 的标准值：3.141592653589793238462643383279…

- 如果 $i < 10000$ ，得到 $\pi = 3.1416926635905345$ （从阴影部分以后不精确了）
- 如果 $i < 1000000$ ，得到 $\pi = 3.1415936535907742$ （同上）
- 如果 $i < 1000000000$ ，得到 $\pi = 3.1415926525880504$ （同上）

可以看到，当迭代的轮数越大，求出的 π 值越精确。

由于是无限累加，我们可以很容易改成并行程序求解，比如 $i=4n$ ，可以分成 4 段并行求

解，再将 4 部分和合并起来得到最终 π 值。假设我们有 4 台计算机，并行计算设计如图 2-34 所示。

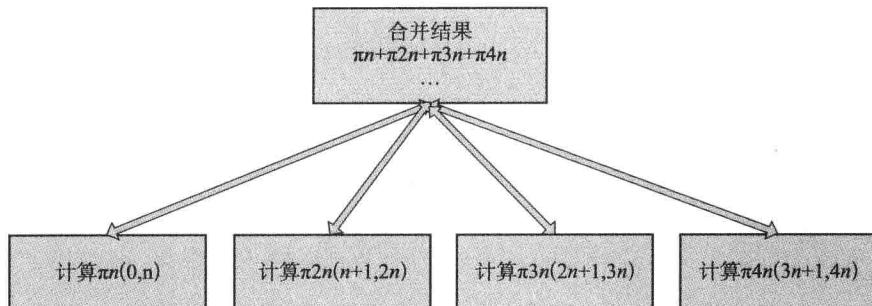


图 2-34 并行计算方式求圆周率

程序实现的方法如下：

- **PiWorker**：是一个 π 计算工人实现，我们可以看到它通过命令行输入一个计算 π 值的起始值和结束值，我们同时启动 4 个 PiWorker 实例，启动时指定不同的起始结束参数。
- **PiCtor**：是一个 π 计算包工头实现，它的实现很简单，获取到线上工人后，通过 doTaskBatch 进行阶段计算，等待每个工人计算完成后，将各工人返回的 π 计算结果合并累加。

运行步骤：

- 1) 启动 ParkServerDemo，它的 IP 端口已经在配置文件指定，如图 2-35 所示。

```
java -cp fourinone.jar; ParkServerDemo
```

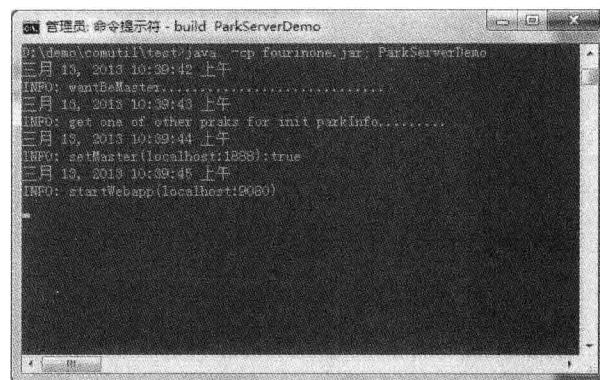


图 2-35 ParkServerDemo

- 2) 运行 4 个 PiWorker，将迭代 100 000 000 轮的计算拆分到 4 个工人并行完成，这里方

便演示是在同一台机器上，现实应用中可以在多台计算机上完成，代码如下，如图 2-36 所示。

```
java -cp fourinone.jar; PiWorker localhost 2008 1 250000000
java -cp fourinone.jar; PiWorker localhost 2009 2500000000 500000000
java -cp fourinone.jar; PiWorker localhost 2010 500000000 750000000
java -cp fourinone.jar; PiWorker localhost 2011 750000000 100000000
```

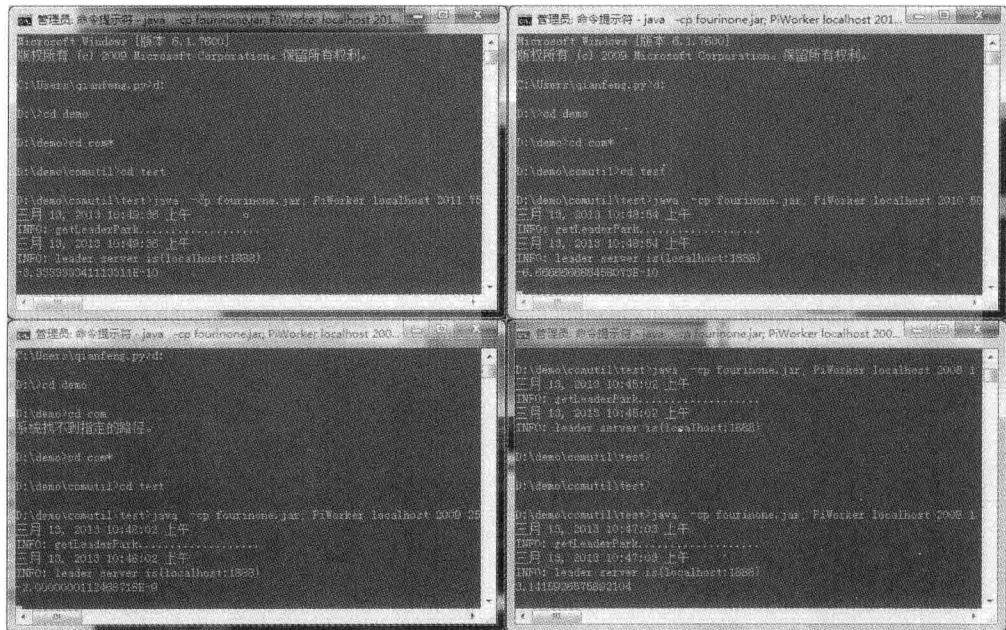


图 2-36 PiWorker

3) 运行 PiCtor，代码如下，如图 2-37 所示。

```
java -cp fourinone.jar; PiCtor
```

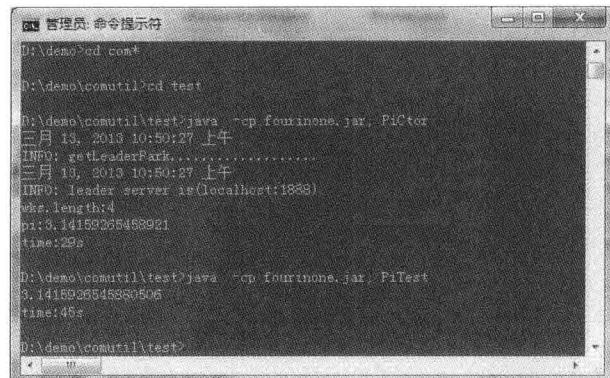


图 2-37 PiCtor



提示

4个工人实例在同台机器并行完成计算 π 值的时间为 29 秒，如果是运行单机程序 PiTest 完成的时间在 45 秒，精准度都是到小数点后 8 位“3.14159265”，但是耗时上有明显差距，如果多机多实例，效率还会进一步提升，并行计算性能提升分析可以参考 2.5.12 节“使用并行计算大幅提升递归算法效率”。

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// PiWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;

public class PiWorker extends MigrantWorker
{
    public double m=0.0,n=0.0;

    public PiWorker(double m, double n){
        this.m = m;
        this.n = n;
    }

    public WareHouse doTask(WareHouse inhouse)
    {
        double pi=0.0;
        for(double i=m;i<n;i++){
            pi += Math.pow(-1,i+1) / (2*i-1) ;
        }

        System.out.println(4*pi);
        inhouse.setObj("pi",4*pi);

        return inhouse;
    }

    public static void main(String[] args)
    {
        PiWorker mw = new PiWorker(Double.parseDouble(args[2]),Double.
            parseDouble(args[3]));
        mw.waitWorking(args[0],Integer.parseInt(args[1]),"PiWorker");
    }
}
```

```

// PiCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.Date;

public class PiCtor extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("PiWorker");
        System.out.println("wks.length:"+wks.length);

        WareHouse[] hmarr = doTaskBatch(wks, inhouse);

        double pi=0.0;
        for(WareHouse result:hmarr){
            pi = pi + (Double)result.getObj("pi");
        }

        System.out.println("pi:"+pi);
        return inhouse;
    }

    public static void main(String[] args)
    {
        PiCtor a = new PiCtor();
        long begin = (new Date()).getTime();
        a.giveTask(new WareHouse());
        long end = (new Date()).getTime();
        System.out.println("time:"+ (end-begin)/1000+"s");
        a.exit();
    }
}

```

2.5.14 从赌钱游戏看 PageRank 算法

谈到并行计算应用，会有人想到 PageRank 算法，我们有成千上万的网页分析链接关系确定排名先后，借助并行计算完成是一个很好的场景。长期以来，Google 的发明 PageRank 算法吸引了很多人学习研究，据说当年 Google 创始者兴奋地找到 Yahoo 公司，说他们找到一种更好的搜索引擎算法，但是被 Yahoo 公司技术人员泼了冷水，说他们关心的不是更好的技术，而是搜索的盈利。后来 Google 包装成了“更先进技术的新一代搜索引擎”的身份，逐渐取代了市场，并实现了盈利。

由于 PageRank 算法有非常高的知名度和普及度，我们接下来以 PageRank 算法为例讲述“并行计算 + 数据算法”的经典搭配，并且这种“海量数据并行处理、迭代多轮后收敛”的分析过程也跟其他的数据挖掘或者机器学习算法应用类似，能起到很好的参考作用。

下面是 PageRank 算法的公式：

$$PR(A) = \left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \dots \right) q + 1 - q$$

我们其实可以直接阐述该公式本身，并介绍如何使用并行计算套用上面公式得到各网页的 PageRank 值，这样虽然通过并行计算方式完成了 PageRank 计算，但是大家仍然不明白上面的 PageRank 公式是怎么来的。

我们把这个 PageRank 算法公式先放在一边，看看一个赌钱的游戏。有甲、乙、丙三个人赌钱，他们的输赢关系如下：

- 甲的钱输给乙和丙。
- 乙的钱输给丙。
- 丙的钱输给甲。

例如，甲、乙、丙各有本钱 100 元，按照以上输赢关系，玩一把下来：

- 甲输给乙 50 元、输给丙 50 元。
- 乙输给丙 100 元。
- 丙输给甲 100 元。

如果仅是玩一把的话很容易算出谁输谁赢。但如果他们几个人维持这样的输赢关系，赢的钱又投进去继续赌，这样一轮一轮赌下去的话，最后会是什么样子呢？

我们可以写个单机程序看看，为了方便计算，初始本钱都设为 1 块钱，用 $x1$ 、 $x2$ 、 $x3$ 表示甲、乙、丙：

```
double x1=1.0, x2=1.0, x3=1.0;
```

用 $x1_income$ 、 $x2_income$ 、 $x3_income$ 代表每赌一把后各人赢的钱，根据输赢关系列出如下表达式：

```
double x2_income = x1/2.0;
double x3_income = x1/2.0+x2;
double x1_income = x3;
```

最后再把各人赢的钱覆盖掉本钱，继续往下计算。

完整程序如下：

```
// Gamble 单机程序
public class Gamble {
    public static double x1=1.0, x2=1.0, x3=1.0;

    public static void playgame() {
```

```

        double x2_income=x1/2.0;
        double x3_income=x1/2.0+x2;
        double x1_income=x3;
        x1=x1_income;
        x2=x2_income;
        x3=x3_income;
        System.out.println("x1:"+x1+", x2:"+x2+", x3:"+x3 );
    }

    public static void main(String[] args){
        for(int i=0;i<500;i++){
            System.out.print(" 第 "+i+" 轮 ");
            playgame();
        }
    }
}

```

我们运行 500 轮后，看到结果如图 2-38 所示。

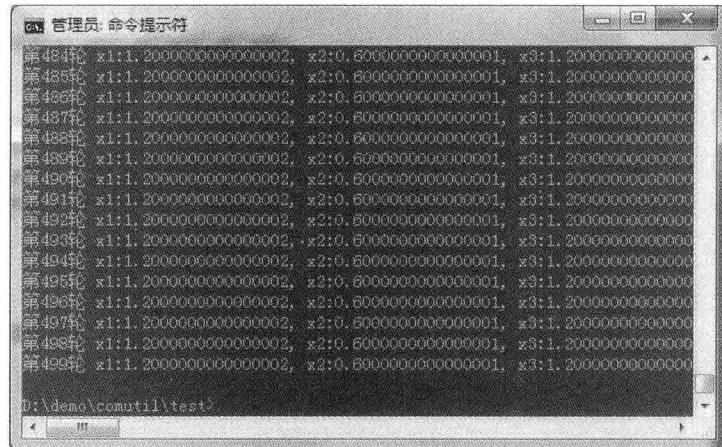


图 2-38 Gamble

我们发现，从 107 轮后，各人的输赢结果就一直是
 $x1:1.2000000000000002, x2:0.6000000000000001, x3:1.2000000000000002$
.....

可能你都没想到会有这么个规律，这样一直赌下去，虽然各人每轮有输有赢，但是多轮后的输赢结果居然保持平衡，维持不变了。用技术术语来说就是多轮迭代后产生了收敛，用俗话来讲，就是玩下去甲和丙是不亏的，乙不服输再继续赌下去，也不会有扳本的机会的。

我们再把输赢关系稍微改一下，丙的钱输给甲和乙：

```

double x2_income=x1/2.0+x3/2.0;
double x3_income=x1/2.0+x2;
double x1_income=x3/2.0;

```

运行 10000 轮后，发现又收敛了：

```
x1:0.6666666666666667, x2:1.0, x3:1.3333333333333333
```

不过这次就变成了“甲是输的，乙保本，丙是赢的”，我们发现收敛的结果可用于排名，如果给他们做一个赌王排名的话，很显然：“丙排第一，乙第二，甲第三”。

那么这样的收敛是在所有情况下都会发生吗，什么情况不会收敛呢？

我们回过头观察上面的输赢关系，甲、乙、丙三人互相各有输赢，导致钱没有流走，所以他们三人才一直可以赌下去，如果把输赢关系改一下，让甲只输钱，不赢钱，关系如下：

```
double x2_income=x1/2.0+x3/2.0;
double x3_income=x1/2.0+x2;
double x1_income=0;
```

那么运行下来会是什么结果呢？如图 2-39 所示。

图 2-39 不收敛情况

我们发现很多轮后，全部为 0 了。我们分析一下过程，第一轮后，甲的钱就输光了，没有赢得一分钱。但是乙和丙各有输赢，他们一直赌到 2000 多轮时，乙的钱全部输光了，甲乙都没钱投进来赌了，导致丙再也赢不到钱了，最后所有人结果都变为 0 了。



这里，我们再分析一下输赢关系，甲的钱全部输给丙和乙后，丙跟乙赌，赢的多输的少，于是所有的钱慢慢都被丙赢走了，导致最后无法维持一个平衡的输赢结果。因此，如果我们要维持平衡和收敛，必须保证赢了钱的人不准走，必须又输给别人才行，让钱一直在三人圈里转而不流失。换句话说，如果存在某人只赢不输，那么这个游戏就玩不下去。

赌钱游戏讲完了，我们再看看 PageRank 算法的公式：

$$PR(A) = \left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \dots \right) q + 1 - q$$

上面的 $L(B)$ 代表页面 B 指向其他页面的连接数，我们举个例子：

假设有 A、B、C 三张网页，它们的链接关系如下：

- A 包含 B 和 C 的链接
- B 包含 C 的链接
- C 包含 A 的链接

根据上面的公式，得到各网页 PR 值如下：

$$\begin{aligned} PR(B) &= PR(A)/2; \\ PR(B) &= PR(A)/2 + PR(C); \\ PR(A) &= PR(C); \end{aligned}$$

可以回过头对照一下，把 A、B、C 改成甲、乙、丙就是上面举的赌钱游戏例子。

那么 q 是干嘛的？公式里的 q 叫做逃脱因子，名字很抽象，目的就是用于解决上面赌钱游戏中“只输不赢”不收敛的问题， $1-q$ 会保证其中一个 PR 值为 0 时计算下来不会全部为 0，那么加了这么一个 $(\dots)*q+1-q$ 的关系后，整体的 PR 值会变化吗？

当每个页面的初始 PR 值为 1 时， $0 \leq q \leq 1$ （计算时通常取值 0.8），我们把所有页面的 PR 值相加看看，假设有 n 张网页：

$$\begin{aligned} & PR(x_1) + PR(x_2) + \dots + PR(x_n) \\ &= ((PR(x_2)/L(x_2) + \dots) * q + 1 - q) + \dots + ((PR(x_1)/L(x_1) + \dots) * q + 1 - q) \\ &= (PR(x_1) * L(x_1)/L(x_1) + PR(x_2) * L(x_2)/L(x_2) + \dots + PR(x_n) * L(x_n)/L(x_n)) * q + n * (1 - q) \\ &= (PR(x_1) + PR(x_2) + \dots + PR(x_n)) * q + n - n * q \\ &= n * q + n - n * q \\ &= n \end{aligned}$$

由于初始 PR 值为 1，所以最后所有页面的 PR 值相加结果还是为 n ，保持不变，但是加上 $(\dots)*q+1-q$ 的关系后，就避免了 PR 值为 0 可以寻求收敛进行排序。

当然实际应用中，这个公式还可以设计得更复杂，我们这里只是为了理解原理，并不是为了做搜索算法，所以就不再深入下去了。



提示

世界的很多东西都是上面这样的游戏，就像炒股，股民赚的钱也就是机构亏的钱，机构赚的

钱也就是股民亏的钱，也许股民们应该研究一下 PageRank 算法，看看股票起起落落的背后是不是收敛了，收敛了说明炒下去永远别想解套，而且机构永远不会亏。

如何使用并行计算方式求 PR 值？从前面的并行计算例子中，我们可以通过 Fourinone 提供的各种并行计算模式去设计，思路方法可以有很多种。

思路一：可以采取工人互相合并的机制（工人互相合并及 receive 使用可参见 sayhello demo），每个工人分析当前网页链接，对每个链接进行一次 PR 值投票，通过 receive 直接投票到该链接对于网页所在的工人机器上，这样经过一轮工人的互相投票，然后再统计一下本机器各网页所得的投票数，这样就得到新的 PR 值。但是这种方式，对于每个链接投票都要调用一次 receive 到其他工人机器，比较耗用带宽，网页数量庞大链接众多时要调用很多次 receive，导致性能不高。

思路二：由于求 PR 值的特点是输入数据大，输出数据小，也就是网页成千上万占空间多，但是算出来的 PR 值占空间小，我们姑且用内存可以装下。因此我们优先考虑每个工人统计各自机器上的网页，计算各链接对应网页所得投票，然后返回工头统一合并得到各网页的 PR 值。可以采用最基本的“总一分一总”并行计算模式实现（请参考分布式计算上手 demo）。

并行计算的拆分和合并设计如图 2-40 所示。

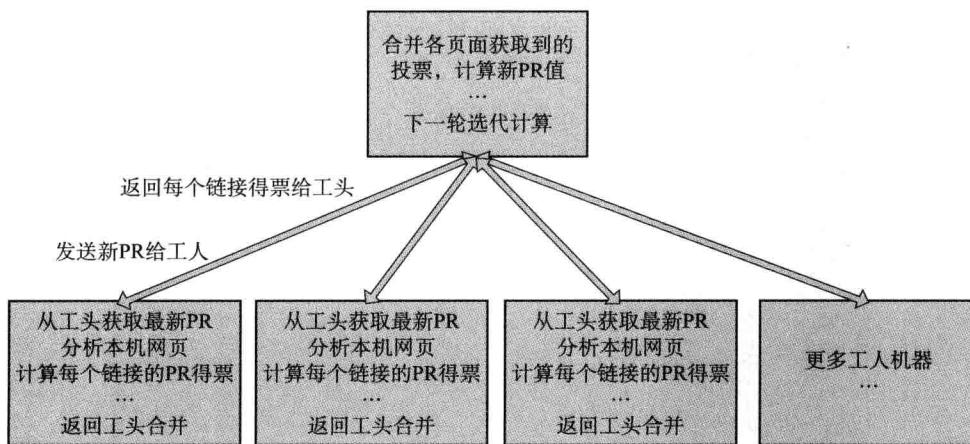


图 2-40 并行计算方式求 PR 值

可以看到如下结论：

- 工人负责统计各自机器上网页的各个链接的 PR 值。
- 工头负责合并累加得到各链接对应网页的新 PR 值，并迭代计算。

程序实现方法如下：

- **PageRankWorker:** 是一个 PageRank 工人实现，为了方便演示，它通过一个字符串数组代表包括的链接（实际上应该从本地网页文件里获取）：

```
links = new String[]{"B", "C"};
```

然后对链接集合中的每个链接进行 PR 投票：

```
for(String p:links)
    outhouse.setObj(p, pr/links.length);
```

- **PageRankCtor:** 是一个 PageRank 包工头实现，它将 A、B、C 三个网页的 PageRank 初始值设置为 1.00，然后通过 doTaskBatch 进行阶段计算，doTaskBatch 提供一个栅栏机制，等待每个工人计算完成才返回，工头将各工人返回的链接投票结果合并累加：

```
pagepr = pagepr+(Double)prwh.getObj(page);
```

得到各网页新的 PR 值（这里取 q 值为 1 进行计算），然后连续迭代 500 轮计算。

运行步骤如下：

- 1) 启动 ParkServerDemo（它的 IP 端口已经在配置文件指定，如图 2-41 所示）：

```
java -cp fourinone.jar; ParkServerDemo
```



图 2-41 ParkServerDemo

- 2) 运行 A、B、C 三个 PageRankWorker，传入不同的 IP 和端口号，如图 2-42 所示：

```
java -cp fourinone.jar; PageRankWorker localhost 2008 A
java -cp fourinone.jar; PageRankWorker localhost 2009 B
java -cp fourinone.jar; PageRankWorker localhost 2010 C
```

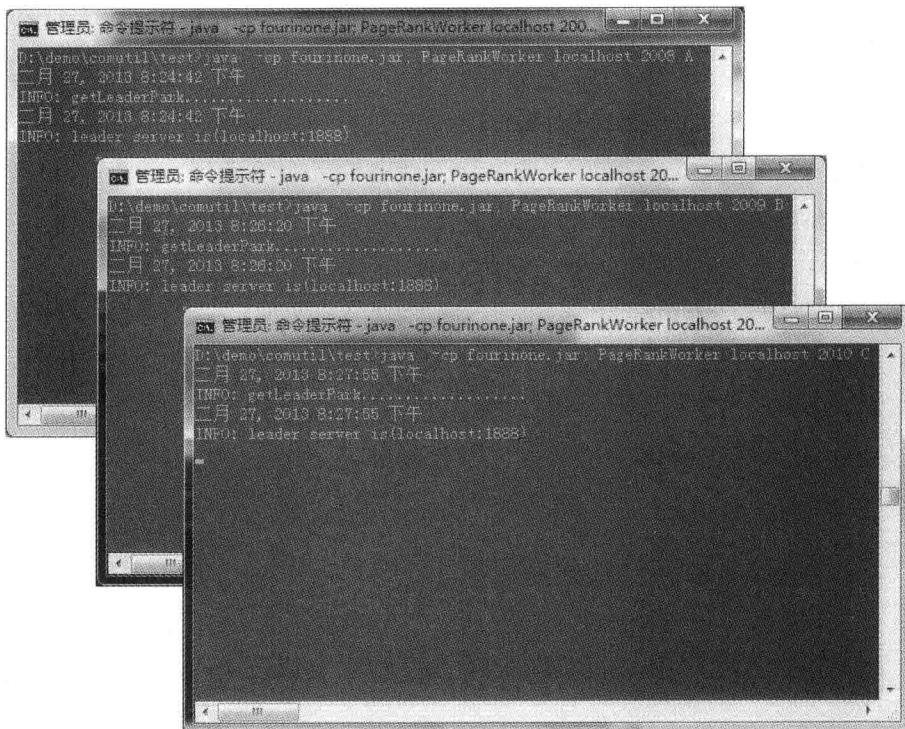


图 2-42 PageRankWorker

3) 运行 PageRankCtor, 结果如图 2-43 所示:

```
java -cp fourinone.jar; PageRankCtor
```

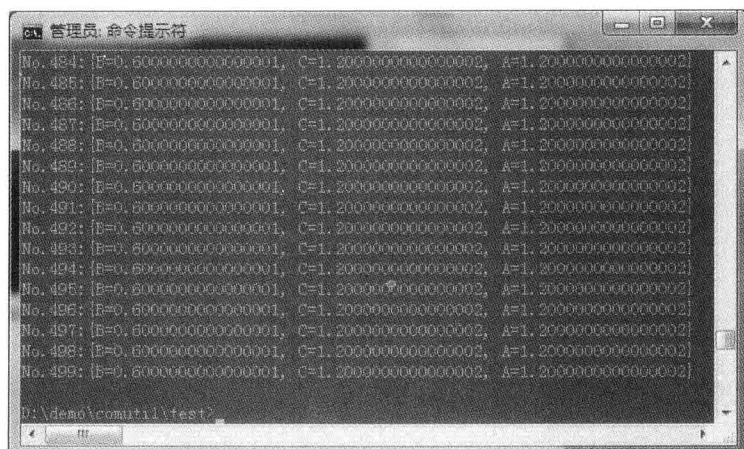


图 2-43 PageRankCtor

我们可以看到跟开始的单机程序结果是一样的，同时各工人窗口依次输出了各自的 PR 值，如图 2-44 所示。

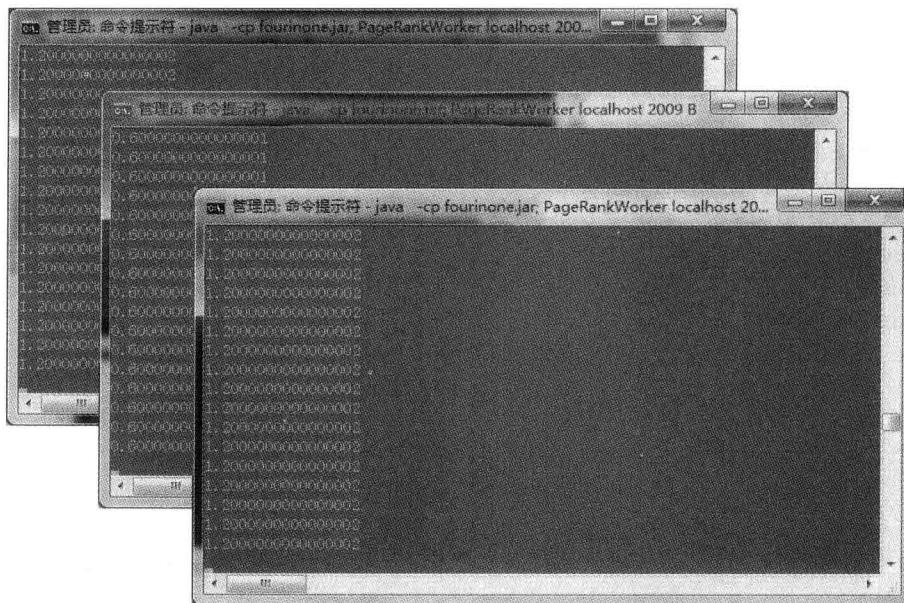


图 2-44 工人运行结果

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// PageRankWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
import com.fourinone.Workman;

public class PageRankWorker extends MigrantWorker
{
    public String page = null;
    public String[] links;

    public PageRankWorker(String page, String[] links){
        this.page = page;
        this.links = links;
    }
}
```

```

}

public WareHouse doTask(WareHouse inhouse)
{
    Double pr = (Double)inhouse.getObj(page);
    System.out.println(pr);

    WareHouse outhouse = new WareHouse();
    for(String p:links)
        outhouse.setObj(p, pr/links.length); // 对包括的链接 PR 投票

    return outhouse;
}

public static void main(String[] args)
{
    String[] links = null;
    if(args[2].equals("A"))
        links = new String[]{"B","C"}; // A 页面包括的链接
    else if(args[2].equals("B"))
        links = new String[]{"C"};
    else if(args[2].equals("C"))
        links = new String[]{"A"};

    PageRankWorker mw = new PageRankWorker(args[2],links);
    mw.waitWorking(args[0],Integer.parseInt(args[1]),"pagerankworker");
}
}

// PageRankCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.Iterator;

public class PageRankCtor extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("pagerankworker");
        System.out.println("wks.length:"+wks.length);

        for(int i=0;i<500;i++) { // 500 轮
            WareHouse[] hmarr = doTaskBatch(wks, inhouse);
            WareHouse prwh = new WareHouse();
            for(WareHouse result:hmarr){
                for(Iterator
                    iter=result.keySet().iterator();iter.hasNext();){
                    String page = (String)iter.next();
                    Double pagepr = (Double)result.getObj(page);
                    if(prwh.containsKey(page))
                        pagepr = pagepr+(Double)prwh.getObj(page);
                    prwh.setObj(page,pagepr);
                }
            }
        }
    }
}

```

```

        inhouse = prwh;//迭代
        System.out.println("No."+i+":"+inhouse);
    }
    return inhouse;
}

public static void main(String[] args)
{
    PageRankCtor a = new PageRankCtor();
    WareHouse inhouse = new WareHouse();
    inhouse.setObj("A",1.00d);//A 的 pr 初始值
    inhouse.setObj("B",1.00d);//B 的 pr 初始值
    inhouse.setObj("C",1.00d);//C 的 pr 初始值
    a.giveTask(inhouse);
    a.exit();
}
}

```

2.5.15 使用并行计算实现上亿排序

1. 为什么需要并行计算来排序

对于大型互联网应用中经常面临对上亿大数据的排序处理等需求，并且上亿大数据量的排序处理能力也是检验分布式系统的计算能力的经典指标之一。

通常在单台计算机上的排序算法有插入排序、快速排序、归并排序、冒泡排序、二叉树排序等等。但是如果是上亿规模的数据，也就是大概 G 以上数量级的排序。超出了单台计算机内存和 CUP 的能力，实现起来很困难。为了实现上亿的大数据量快速排序，需要考虑利用多台计算机的协同计算能力实现，将多台计算机的内存和 CUP 等资源利用起来，建立分布式的计算，通过分布式协作方式，由一台调度的主计算机命令各负责任务处理的计算机先将海量数据首先进行分组归类，然后合并归类，再用逐个排序的方式去完成。

如果有 10 台计算机，每台上有 5GB 数据文件，如何对这 50GB 的数据排序？

对于上亿的数据，数量很大，通常无法在一台计算机上保存，数据本身就分布在多台计算机硬盘上，它本身就是分布式存储的，这里为了方便理解整个排序原理，我们不采用分布式文件系统存储，而假设每台计算机上已经放置好了几 G 的数据文件。

2. 数据取值范围

假设这 50GB 的数据都是整数，而且取值范围属于 0 到 max，这个 max 我们可以自由指定，比如 max=10 万，这 50GB 的数据都是 10 万以内的数字。

3. 输入输出数据

输入数据是放置在每台机器上的 5GB 的无序数据，那么输出数据是怎么样放置的呢，将

这 50GB 排好序的数据放置到其中一台计算机上去吗，这也是不合理的，如果数据量再大一点，一台计算机的硬盘也不够放，实际上，输出数据也是分布式地放在每台计算机上，只不过是排好了顺序的。

4. 并行计算排序的详细过程

按照并行计算的思想，首先我们需要对问题进行拆分和合并，体现在首先对数据进行分类，每台计算机只管某个范围的数据，比如第一台计算机只管 10000 以下大小的数据，第二台计算机只管 10000 ~ 20000 大小的数据……第 10 台计算机只管 90000 ~ 100000 大小的数据。

然后合并，将属于其他计算机范围的数据发给对方，最后各自在本机中完成本范围内的数据。我们可以归纳为“分类、合并、排序”三个阶段。

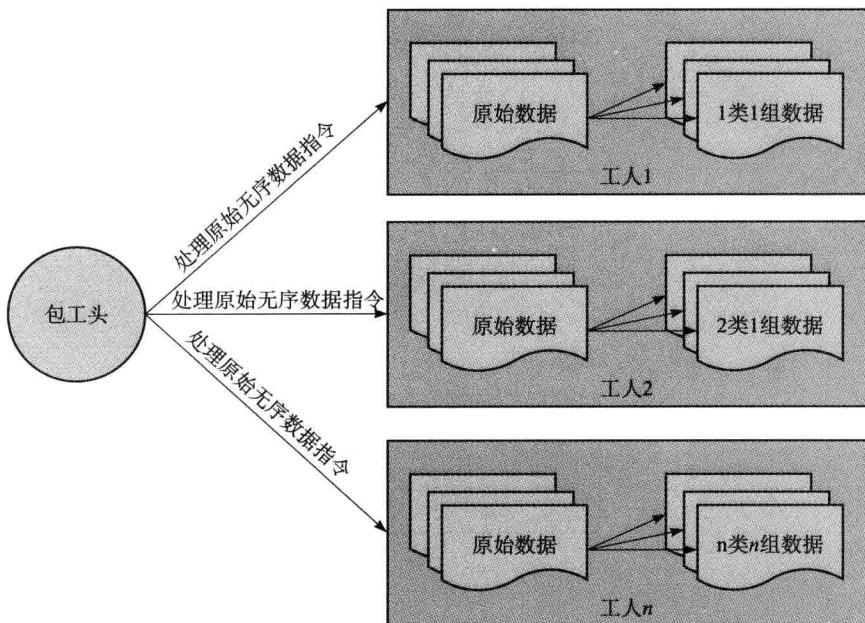


图 2-45 分类阶段

第一阶段：将原始数据分组分类（见图 2-45）

- 1) 杂乱无序的原始数据分散在各个工人计算机上，等待进行分组分类。
- 2) 工头计算机发出处理原始无序数据的命令，该命令是并发调度，每台工人计算机无须先后等待完成，同时进行。
- 3) 每台工人计算机同时对各自的原始数据进行分组分类处理。

分类规则

由于原始数据数量太大，需要分散到不同任务计算机存储和处理，因此需要进行两个维

度的划分，每台任务计算机将自己的原始数据共分为 n 组， n 为工人计算机的总数，同时每组的数据不能太大，还需要根据最小处理单元进一步分类为 s 份，比如每次最小处理十万条，那总数据量除于十万获得 s 份，最小处理单元可以根据计算机不同而设置。最后的结果就是将原始数据处理后得到一个 n 组，每组又有多类的数据格式存放。

那如何判断一个数字 x 是属于哪组哪类的呢？规则如下：

假设原始数据里的数字最大值为 m ，那么 $x*n/m$ 得到所属分组， $x*s/m$ 得到所在分类。

4) 完成后返回通知工头计算机进行下一阶段任务分配。

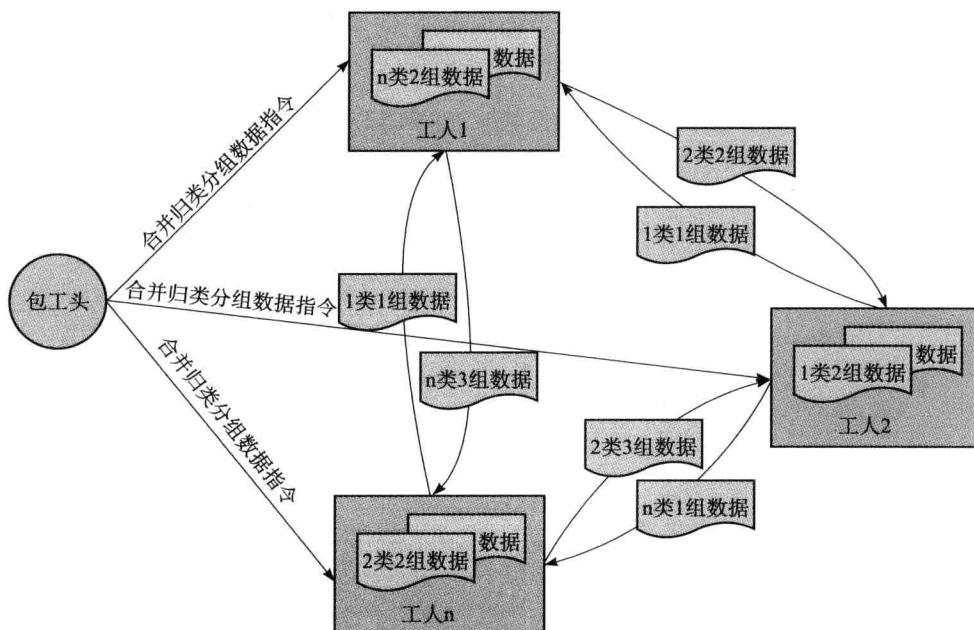


图 2-46 合并阶段

第二阶段：分组分类后的数据的合并处理（见图 2-46）

1) 工头计算机发出合并归类分组数据的命令，并通过命令将“工人计算机的总数”以及“被命令的计算机所在的位置序号”发给被命令的工人计算机（实际上这两个信息可以由工人自己获取，不一定需要工头发送）。

2) 每台工人计算机收到命令后，根据分组分类的标示信息，将属于哪台工人计算机的数据取出并发给该计算机，属于自己的继续保存。每次只发一类数据，每类数据包括“组信息、类信息、数据本身”内容。通过合并后，每台工人计算机上保存着属于自己范围的数据，虽然每份里的数据仍然是无序的，但是粗的分组分类使得这些范围是有序的。

3) 完成后返回通知工头计算机进行下一阶段任务分配。

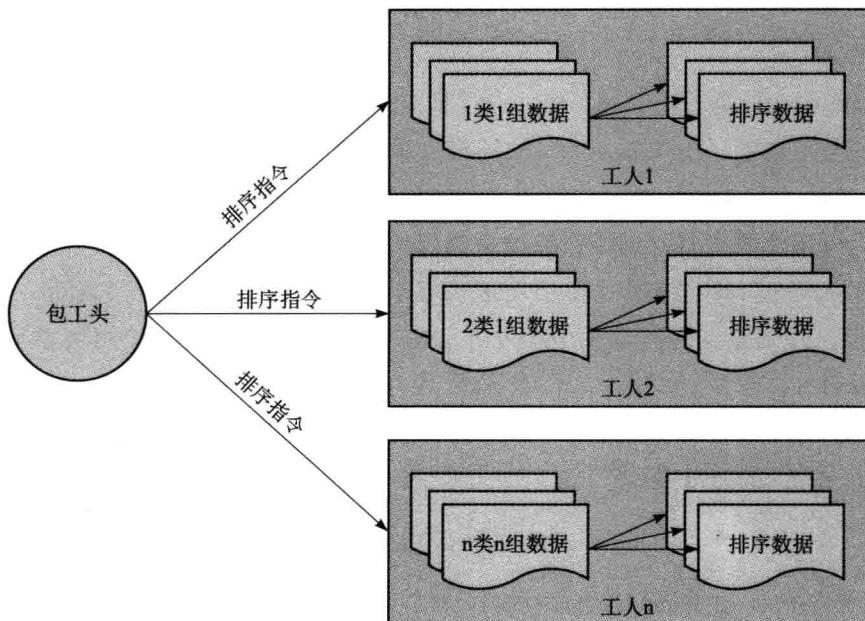


图 2-47 排序阶段

第三阶段：分组分类后的数据的排序处理（见图 2-47）

- 1) 工头计算机发出排序数据的命令。
- 2) 工人计算机对属于自己范围的分组分类数据进行排序，最后得到一个整体原始数据的排序结果，但是它是根据范围分散到不同任务计算机上存放的。
- 3) 完成后返回通知工头计算机完成排序。

下面是一个按照这个三阶段并行计算排序思想的 demo，由于这里旨在抛砖引玉，通过尽量简单的示例，将并行计算的排序思想阐述清楚，所以 demo 都是基于内存完成，并且以输入数据的方式进行了简化。由各工人机器生成一定数量的随机数字，中间分类结果的数据保存在 map 变量里，输出结果数据则直接打印出来。实际上，一个完整的上亿排序程序，使用内存保存输入数据和中间分类结果都是不够的，会导致内存溢出，需要参考上面第一、二阶段描述的分类规则，根据每台工人计算机的内存能力，设置最小处理单元，并且输入数据、中间结果数据、输出数据都使用文件方式存储，只有最小处理单元的数据才能通过内存排序。（详见第 6 章对分布式文件的简化操作。）

- SortCtor：是一个工头实现，可以看到它的程序结构很简单，总共分为三个步骤，每步通过发送一个 step 的命令通知工人去做，并且都是使用 doTaskBatch 批量完成，也

就是每一步必须全部工人做完才能进行下一步。前两个步骤是分类与合并，工头不关心结果，只关心是否做完。最后一步各自排序完成后，工头会匹对一下总数，并输出排序时间。

- **SortWorker**: 是一个工人实现，它对照工头的三步命令执行三阶段任务，第一阶段生成一定数量（由参数传入）的随机数据，并将数据分类后存放在一个 `HashMap` 里，`key` 为分类 id，由通过对工人总数取模获取，`value` 为属于这一分类的所有数字；第二个阶段将不属于自己的分类发给其他工人（通过对比自身的 `index`），所有工人通过 `receive` 同时互相发送，可以看到 `receive` 方法的实现里，将收到的数据合并到该工人所属的分类中（工人互相合并及 `receive` 使用可参见 `sayhello demo`）。

运行步骤：

- 1) 编译 demo 的 java 类：

```
Java -classpath fourinone.jar; *.java
```

- 2) 启动 `ParkServerDemo`（它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定）：

```
Java -classpath fourinone.jar; ParkServerDemo
```

- 3) 运行 `SortWorker`（依次传入 ip、端口号、生成数字数量、数字取值范围）：

```
Java -classpath fourinone.jar; SortWorker localhost 2008 1000 100000
Java -classpath fourinone.jar; SortWorker localhost 2009 1000 100000
Java -classpath fourinone.jar; SortWorker localhost 2010 1000 100000
Java -classpath fourinone.jar; SortWorker localhost 2011 1000 100000
```

上面表示运行 4 个工人，每个工人生成 1000 个取值范围为 100000 的数字，然后开始对这 4000 个数字进行排序。

- 4) 运行 `SortCtor`

```
Java -classpath fourinone.jar; SortCtor
```

可以看到工头窗口会输出总共完成排序的数量和时间，如图 2-48 所示。

各工人窗口会输出各自排好序的数字，发现各工人数字依次遵循 0 ~ 25000, 25000 ~ 50000, 50000 ~ 75000, 75000 ~ 100000 取值范围的分类，如图 2-49 所示。

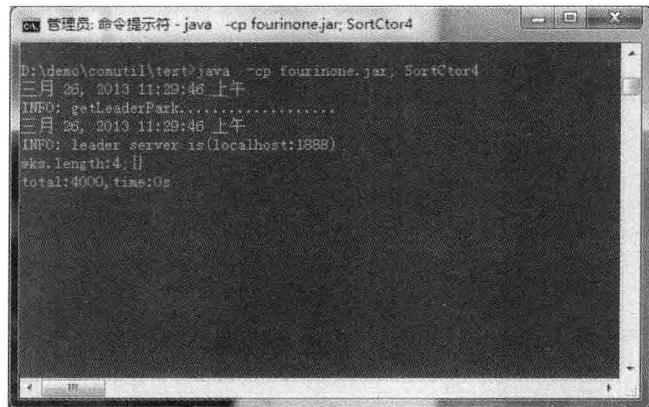


图 2-48 SortCtor



图 2-49 SortWorker

完整 demo 源码如下：

```

// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

```

```

}

//SortWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
import com.fourinone.Workman;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;
import java.util.Collections;

public class SortWorker extends MigrantWorker
{
    private final int rammax;//1000000 排序数据的取值范围最大值
    private int totalmax;//排序数据的总量(每个工人)
    private int total = 0;//记录当前生成随机数据的数量
    private HashMap<Integer,List<Integer>> wharr = new HashMap<Integer,List<Integer>>();
    //用来存放分类信息
    private Random rad = new Random();
    private int wknum=-1;
    private int index=-1;
    private Workman[] wms = null;

    public SortWorker4(int totalmax, int rammax)
    {
        this.totalmax = totalmax;
        this.rammax = rammax;
    }

    public Integer[] getNumber()
    {
        if(total++<totalmax){
            int thenum = rad.nextInt(rammax);
            int numi = (thenum*wknum)/rammax;//通过对工人总数取模获取分类
            return new Integer[]{numi,thenum};
        }
        else return new Integer[]{-1,-1};
    }

    public WareHouse doTask(WareHouse wh)
    {
        int step = (Integer)wh.getObj("step");
        if(wms==null){
            wms = getWorkerAll();
            wknum = wms.length;
        }
        index = getSelfIndex();
        System.out.println("wknum:"+wknum+";step:"+step);
        WareHouse resultWh = new WareHouse("ok",1);

        if(step==1){
            Integer[] num = null;
            while(true){
                num = getNumber();

```

```

        if(num[0]!=-1) {
            List<Integer> arr = wharr.get(num[0]); // 取出该分类的 list
            if(arr==null)
                arr = new ArrayList<Integer>();
            arr.add(num[1]);
            wharr.put(num[0], arr);
        }
        else break;
    }
}else if(step==2) {
    for(int i=0;i<wms.length;i++) {
        if(i!=index&&wharr.containsKey(i)){
            List<Integer> othernum = wharr.remove(i);
            Workman wm = wms[i];
            System.out.println(i+"-receive:"+wm.receive(new
                WareHouse(i, othernum))); // 将不属于自己的分类发给其他工人
        }
    }
}else if(step==3) {
    List<Integer> curlist = wharr.get(index);
    Collections.sort(curlist); // 对属于自己分类的数据进行内存内排序
    System.out.println(curlist);
    System.out.println(curlist.size());
    resultWh.setObj("total",curlist.size());
}
return resultWh;
}

protected boolean receive(WareHouse inhouse)
{
    List<Integer> thisnum = wharr.get(index);
    thisnum.addAll((List<Integer>)inhouse.get(index));
    return true;
}

public static void main(String[] args)
{
    SortWorker mw = new SortWorker(Integer.parseInt(args[2]),Integer.
        parseInt(args[3]));
    mw.waitWorking(args[0],Integer.parseInt(args[1]),"SortWorker");
}
}

// SortCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.Date;

public class SortCtor extends Contractor
{
    public WareHouse giveTask(WareHouse wh)
    {
        WorkerLocal[] wks = getWaitingWorkers("SortWorker");
        System.out.println("wks.length:"+wks.length+" ; "+wh);
    }
}

```

```

        wh.setObj("step", 1);//1:group;
        doTaskBatch(wks, wh);

        wh.setObj("step", 2);//2:merge;
        doTaskBatch(wks, wh);

        wh.setObj("step", 3);//3:sort
        WareHouse[] hmarr = doTaskBatch(wks, wh);
        int total=0;
        for(int i=0;i<hmarr.length;i++){
            Object num = hmarr[i].getObj("total");
            if(num!=null)
                total+=(Integer)num;
        }
        wh.setObj("total",total);
        return wh;
    }

    public static void main(String[] args)
    {
        Contractor a = new SortCtor();
        WareHouse wh = new WareHouse();
        long begin = (new Date()).getTime();
        a.doProject(wh);
        long end = (new Date()).getTime();
        System.out.println("total:"+wh.getObj("total")+",time:"+ (end-begin)/
        1000+"s");
    }
}

```

2.5.16 工人服务化模式应用示例

前面 2.1.8 节中，我们详细介绍了工人服务化模式，这里我们演示建立一个如下 demo：输入一个名字，返回一个 hello xx 的服务 demo，我们看到大部分服务化产品的上手程序都是类似这样的 sayHello。设计思路如下：

- CtorClient：建立了一个工头客户端，在服务化运用中，包工头已经失去了在并行计算里的名称含义，它变成了一个可以获取多种类型服务，调用多种类型服务，获取调用结果的综合性服务客户端。

我们可以看到，CtorClient 定义了一个 sayHello 的服务接口，这个服务接口是面向用户的，sayHello 服务的实现是调用了 giveTask，而在 giveTask 里面则是通过获取工人服务类型，调用 doTaskBatch 获得结果，然后将结果返回给 sayHello。

sayHello 的实现实际上是将用户的 String name 输入包装为工人通用服务 doTask 的输入，再请求远程工人的服务实现，最后将结果转换为 String 返回给用户。

- **ServiceWorker:** 建立了一个工人服务，需要在配置文件 config.xml 的工人模块部分，设置为 <SERVICE>true</SERVICE>，这样保证工人可以对外提供多个客户端的请求服务。在工人的 doTask 里面实现了 sayHello 服务的逻辑，就是获取输入参数后，加上 hello 返回。
- **ClientMain:** 为了模拟多个工头客户端请求服务，我们使用了前面介绍的 BeanContext.tryStart 启动两个进程去同时调用 sayHello 服务，并将各自的日志结果输出到相应文件。运行步骤如下：

1) 编译 demo 的 java 类：

```
Java -classpath fourinone.jar; *.java
```

2) 启动 ParkServerDemo (它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定)：

```
Java -classpath fourinone.jar; ParkServerDemo
```

3) 运行 ServiceWorker (传入端口号参数，这里启动一个工人即可，代表只有一个服务地址)：

```
Java -classpath fourinone.jar; ServiceWorker 2008
```

4) 运行 ClientMain (它内部会启动 2 个工头客户端访问服务)：

```
Java -classpath fourinone.jar; ClientMain
```

下面是 demo 源码：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
    public static void main(String[] args){
        BeanContext.startPark();
    }
}

// ServiceWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;

public class ServiceWorker extends MigrantWorker
{
    public WareHouse doTask(WareHouse inhouse)
    {
        // 取出参数的值
        String inputstring = inhouse.getString("InputString");
        // 收到服务请求后，返回 hello
        return new WareHouse("Result", inputstring+",hello");
    }
}
```

```

    }

    public static void main(String[] args)
    {
        ServiceWorker mw = new ServiceWorker();
        // 启动服务, <SERVICE>true</SERVICE>
        mw.waitWorking("localhost", Integer.parseInt(args[0]), "HelloService");
    }
}

// CtorClient
public class CtorClient extends Contractor
{
    public String sayHello(String name)
    {
        // 封装输入参数 InputString, 然后实际调用通用服务接口
        WareHouse result = giveTask(new WareHouse("InputString", name));
        // 返回服务请求的结果
        return result.getString("Result");
    }

    public WareHouse giveTask(WareHouse inhouse)
    {
        // 获取提供 Hello 服务的地址, 这里假设只有一个, 但可以是多个
        WorkerLocal[] wks = getWaitingWorkers("HelloService");
        System.out.println("wks.length:" + wks.length);

        // 请求服务并传入 InputString 参数, 然后等待结果完成后返回
        WareHouse[] result = doTaskBatch(wks, inhouse);

        return result[0];
    }

    public static void main(String[] args)
    {
        CtorClient a = new CtorClient();
        String serviceResult = a.sayHello(args[0]);
        System.out.println(serviceResult);
        a.exit();
    }
}

// ClientMain
import com.fourinone.StartResult;
import com.fourinone.BeanContext;

public class ClientMain
{
    public static void main(String[] args)
    {
        // 同时启动 2 个客户端调用 Hello 服务, 将结果输出到相应日志
        StartResult<Integer> ctor1 = BeanContext.tryStart("java", "-cp", "fourinone.
            jar;","CtorClient","client1");
        ctor1.print("log/ctor1.log");
        StartResult<Integer> ctor2 = BeanContext.tryStart("java", "-cp", "fourinone.

```

```

        jar;,"CtorClient","client2");
ctor2.print("log/ctor2.log");
}
}

```

2.6 实时流计算

实时流计算的场景：业务系统根据实时的操作，不断生成事件（消息 / 调用），然后引起一系列的处理分析，这个过程是分散在多台计算机上并行完成的，看上去就像事件连续不断地流经多个计算节点处理，形成一个实时流计算系统。

市场上流计算产品有很多，主要是通过消息中枢结合工人模式实现的，大致过程如下：

- 1) 开发者实现好流程输入输出节点逻辑，上传 job 到任务生产者。
- 2) 任务生产者将任务发送到 ZooKeeper，然后监控任务状态。
- 3) 任务消费者从 ZooKeeper 上获取任务。
- 4) 任务消费者启动多个工人进程，每个进程又启动多个线程执行任务。
- 5) 工人之间通过 zeroMQ 交互。

我们也可以做一个简单的流计算系统，做法跟上面有些不同：

- 1) 首先不过多依赖 ZooKeeper，任务的分配最好直接给到工人，并能直接监控工人完成状态，这样效率会更高。
- 2) 工人之间直接通信，不依赖 zeroMQ 转发。
- 3) 并行管理扁平化，多进程中再分多线程意义不大，增加管理成本，实际上一台机器 8 个进程，每个进程再开 8 个线程，总体跟 8 ~ 10 个进程或者线程的效果差不多（数量视机器性能不同）。
- 4) 做成一个流计算系统，而不是平台。

设计思路：用工头去做任务生产和分配，用工人去做任务执行，为了达到流的效果，需要在工人里面调用工头的方式，将多个工人节点串起来，形成一个计算拓扑图。

下面程序演示了连续多个消息先发到一个工人节点 A 处理，然后再发到两个工人节点 B 并行处理的流计算过程，并且获取到最后处理结果打印输出（如果不需要获取结果可以直接返回）。

- StreamCtorA：工头 A 实现，它获取到线上工人 A，然后将消息发给它处理，并轮循等待结果。工头 A 的 main 函数模拟了多个消息的连续调用。
- StreamWorkerA：工人 A 实现，它接收到工头 A 的消息进行处理，然后创建一个工头 B，通过工头 B 将结果同时发给两个工人 B 处理，然后将结果返回工头 A。

□ StreamCtorB：工头 B 实现，它获取到线上两个工人 B，调用 doTaskBatch 等待两个工人处理完成，然后返回结果给工人 A。

□ StreamWorkerB：工人 B 实现，它接收到任务消息后模拟处理后返回结果。

运行步骤（在本地模拟）如下：

1) 启动 ParkServerDemo（它的 IP 端口已经在配置文件指定）。

```
java -cp fourinone.jar; ParkServerDemo
```

2) 启动工人 A。

```
java -cp fourinone.jar; StreamWorkerA localhost 2008
```

3) 启动两个工人 B。

```
java -cp fourinone.jar; StreamWorkerB localhost 2009
```

```
java -cp fourinone.jar; StreamWorkerB localhost 2010
```

4) 启动工头 A。

```
java -cp fourinone.jar; StreamCtorA
```

多机部署说明：StreamCtorA 可以单独部署一台机器，StreamWorkerA 和 StreamCtorB 部署一台机器，两个 StreamWorkerB 可以部署两台机器。

总结：如何选择计算平台和计算系统。

如果我们只有几台机器，但是每天有人开发不同的流处理应用要在这几台机器上运行，我们需要一个计算平台来管理好 job，让开发者按照规范配置好流程和运行时节点申请，打包成 job 上传，然后平台根据每个 job 配置动态分配资源依次执行每个 job 内容。

如果我们的几台机器只为一个流处理业务服务，比如实时营销，我们需要一个流计算系统，按照业务流程部署好计算节点即可，不需要运行多个 job 和动态分配资源。按照计算平台的方式做只会增加复杂性，开发者也不清楚每台机器上到底运行了什么逻辑。

如果你想实现一个计算平台，可以参考动态部署和进程管理功能。

完整源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}
```

```

//StreamCtorA
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.ArrayList;

public class StreamCtorA extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("StreamWorkerA");
        System.out.println("wks.length:"+wks.length);

        WareHouse result = wks[0].doTask(inhouse);
        while(true){
            if(result.getStatus() != WareHouse.NOTREADY)
            {
                break;
            }
        }
        return result;
    }

    public static void main(String[] args)
    {
        StreamCtorA sc = new StreamCtorA();
        for(int i=0;i<10;i++){
            WareHouse msg = new WareHouse();
            msg.put("msg", "hello"+i);
            WareHouse wh = sc.giveTask(msg);
            System.out.println(wh);
        }
        sc.exit();
    }
}

//StreamWorkerA
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;

public class StreamWorkerA extends MigrantWorker
{
    public WareHouse doTask(WareHouse inhouse)
    {
        System.out.println(inhouse);
        //do something
        StreamCtorB sc = new StreamCtorB();
        WareHouse msg = new WareHouse();
        msg.put("msg", inhouse.getString("msg")+", from StreamWorkerA");
        WareHouse wh = sc.giveTask(msg);
        sc.exit();

        return wh;
    }

    public static void main(String[] args)
}

```

```

    {
        StreamWorkerA wd = new StreamWorkerA();
        wd.waitWorking(args[0],Integer.parseInt(args[1]),"StreamWorkerA");
    }
}

//StreamCtorB
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.ArrayList;

public class StreamCtorB extends Contractor
{
    public WareHouse giveTask(WareHouse inhouse)
    {
        WorkerLocal[] wks = getWaitingWorkers("StreamWorkerB");
        System.out.println("wks.length:"+wks.length);

        WareHouse[] hmarr = doTaskBatch(wks, inhouse);

        WareHouse result = new WareHouse();
        result.put("B1",hmarr[0]);
        result.put("B2",hmarr[1]);

        return result;
    }
}
//StreamWorkerB
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;

public class StreamWorkerB extends MigrantWorker
{
    public WareHouse doTask(WareHouse inhouse)
    {
        System.out.println(inhouse);
        //do something
        inhouse.put("msg",inhouse.getString("msg")+",from StreamWorkerB");
        return inhouse;
    }
}

public static void main(String[] args)
{
    StreamWorkerB wd = new StreamWorkerB();
    wd.waitWorking(args[0],Integer.parseInt(args[1]),"StreamWorkerB");
}
}

```

第3章

分布式协调的实现

分布式协调是分布式应用中不可缺少的，通常会设立专门的协调者角色，即将多机协调的职责从分布式应用中独立出来，以减少系统的耦合性和增强可扩展性。Apache 的 ZooKeeper、Google 的 Chubby 都是分布式协调的实现者。Fourinone 实际上可以单独当做 ZooKeeper 用，它使用最少的代码实现了 ZooKeeper 的所有功能，并且力图做到功能更强、使用更简洁。

本章会从设计角度讲述分布式协调系统的实现原理，包括归纳出的 API 介绍，权限机制介绍，并详细阐述了在领导者选举机制上和 Paxos 算法的区别，最后再结合实践中的统一配置和集群管理等应用讲解 demo，让读者有更直观的体会。

3.1 协调架构原理简介

Fourinone 对分布式协调的实现，是通过建立一个 domain/node 两层结构的节点信息去完成，domain 可以是分类或者包，node 可以是具体属性，domain 和 node 都是根据需求设计命名，比如可以将 domain 命名为 “a.b.c...” 表示一个树型类目。一个 domain 下可以有很多个 node，每个 node 只指定一个 domain，可以通过 domain 返回它下面所有的 node。domain 不需要单独建立，通常在建立 node 时，如果不存在 domain 会自动创建。如果 domain 下没有 node 了，该 domain 会自动删除。如果删除 domain，该 domain 下面 node 也都会删除。每个 node 下可以存放一个值，可以是任意对象。所有的节点信息存放在 ParkServer 里，ParkServer 提供协调者的功能。如图 3-1 所示。

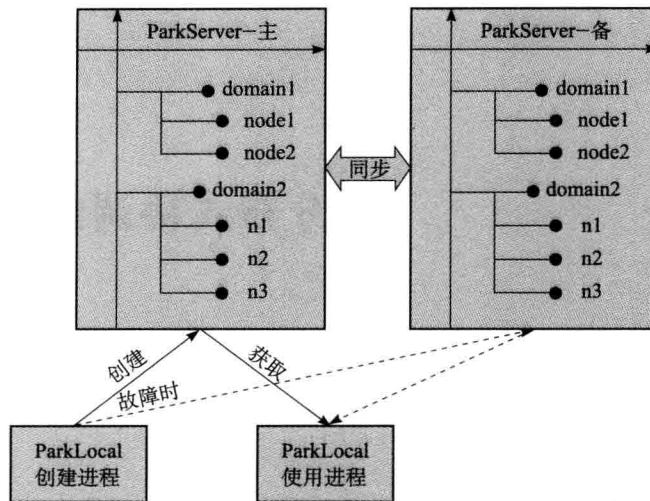


图 3-1 domain/node 结构图

从图 3-1 可以看到，其他分布式进程可以通过 ParkServer 的用户接口 ParkLocal，对节点进行增加、修改、删除、指定心跳、指定权限等操作，并且结合 ParkServer 提供同步备份、领导者选举、过期时间设置等功能，共同来实现众多分布式协调功能。

我们举个例子，说明两个分布式应用完成协调功能的流程：

1) 分布式应用 A 通过 ParkLocal 在 ParkServer 上创建一个 domain/node 的节点，并且在节点里存放相应的 value，这个节点以及它的 value 值代表分布式应用 A 的某种协调信息，它存放在 ParkServer 上用于向分布式应用 B 分享。

2) 分布式应用 B 通过 ParkLocal 操作 ParkServer，对它上面的分布式应用 A 建立的这个 domain/node 节点进行监听，如果节点 value 发生变化，那么分布式应用 B 可以获取到这个 value，并进行相应的业务处理，这样便将各自独立的分布式应用 A 和 B 协调了起来。

3) 由于 ParkServer 保存着用于协调的节点和信息，为了防止 ParkServer 崩机导致整体故障，ParkServer 配置为一主多备的关系，互相同步信息，在故障时可以进行领导者选举，切换到备用 ParkServer 上继续提供协调服务。

分布式协调的场景还有很多（我们在 3.6 节会详细讲解），例如：

1) 分布式配置，多个机器的应用公用一个配置信息，并且挂掉能够通过领导者选举进行恢复；

2) 分布式锁，多个机器竞争一个锁，当某个机器释放锁或者挂掉，其他机器可以竞争到锁，继续执行任务；

3) 集群管理，集群内机器可以互相感知并进行领导者选举；

4) 多个节点，每个节点具有读写权限不一样的操作。

3.2 核心 API

ParkLocal 核心 API 说明如下。

□ 创建 node，可以根据是否需要权限和心跳属性调用不同方法：

```
public ObjectBean create(String domain, Serializable obj); // 自动创建 node
public ObjectBean create(String domain, String node, Serializable obj);
public ObjectBean create(String domain, String node, Serializable obj,
AuthPolicy auth);
public ObjectBean create(String domain, String node, Serializable obj, boolean
heartbeat);
public ObjectBean create(String domain, String node, Serializable obj,
AuthPolicy auth, boolean heartbeat);
```

□ 更新 node：

```
public ObjectBean update(String domain, String node, Serializable obj);
```

□ 获取 node：

```
public ObjectBean get(String domain, String node);
```

□ 获取最新 node，需要传入旧 node 进行对照：

```
public ObjectBean getLastest(String domain, String node, ObjectBean ob);
```

□ 获取最新 domain：

```
public List<ObjectBean> get(String domain);
```

□ 获取最新 domain 下所有 node，需要传入旧的 node 集合对照：

```
public List<ObjectBean> getLastest(String domain, List<ObjectBean> oblist);
```

□ 删除 node：

```
public ObjectBean delete(String domain, String node);
```

□ 强行设置 domain 可删除：

```
public boolean setDeletable(String domain);
```

□ 删除 domain 及其下所有 node：

```
public List<ObjectBean> delete(String domain);
```

□ 添加 node 的事件监听：

```
public void addLastestListener(String domain, String node, ObjectBean ob,
LastestListener lisr);
```

□ 添加 domain 的事件监听：

```
public void addLastestListener(String domain, List<ObjectBean> oblist,
LastestListener lisr);
```

下面详细说明 ParkLocal 几个核心的 API 的使用，其他的 API 使用与此类似。

1. 创建 node

```
public ObjectBean create(String domain, String node, Serializable obj);
```

通过上面方法在 ParkServer 里建立一个节点，需要指定 domain 和 node 名称，如果第一次创建，domain 不存在，ParkServer 会自动创建，因此不需要单独创建 domain 的 API。如果 domain 和 node 都存在，重复创建会失败。

节点的值是一个 Serializable 对象，它必须是可序列化的，因为需要进行网络传输。

2. 创建心跳属性节点

```
public ObjectBean create(String domain, String node, Serializable obj, boolean
heartbeat);
```

有的情况下，我们需要创建节点后，创建进程跟 ParkServer 保持心跳连接，如果创建进程死掉了，那这个节点会自动被删除掉，心跳属性节点很适合在集群管理等场景上的应用。

3. 获取 node

```
public ObjectBean get(String domain, String node);
```

根据 domain 和 node 名称获取对应的对象，但是返回一个 ObjectBean 的封装对象，我们要拿到原始对象，可以通过 toObject() 获取，ObjectBean 也封装了 domain 和 node 的名称信息，可以通过 getDomain 和 getNode 获取。

4. 获取最新 node，需要传入旧 node 进行对照

```
public ObjectBean getLastest(String domain, String node, ObjectBean ob);
```

getLastest 是一个很有用的方法，可以获取到该节点的最新版本，它获取最新版本的方式是需要将旧版本的 ObjectBean 传进去，ParkServer 会进行对比，如果发现对象值更新了，就返回一个新的 ObjectBean，如果没有更新变化，就返回 null。

5. 添加 node 的事件监听

```
public void addLastestListener(String domain, String node, ObjectBean ob,
LastestListener lisr);
```

如果我们要检测 ParkServer 上节点是否变化，可以通过上面 getLastest 方法轮询最新值，也可以通过事件方式响应。

我们需要对一个节点进行监听，事件响应方式监控配置信息发生变化，需要实现一个 LastestListener 的事件接口并进行注册，当信息变化时，会产生事件并获取到变化后的对象进行处理，LastestListener 的 happenLastest 方法有个 boolean 返回值，如果返回 false，它会一直监控配置信息变化，继续有新的变化时还会进行事件调用；如果返回 true，它完成本次事件调用后就终止。

如果初次注册节点的事件监听，可以指定一个初始对比值 ObjectBean ob，代表事件是相对于该值的变化或持续变化；如果不需要初始对比值，可以传入 null，只要目前节点的值不为 null，就会产生事件。

domain 的事件监听注册跟 node 使用类似，只是初始对比值为 List<ObjectBean>。

3.3 权限机制

上面我们介绍了 domain/node 节点和相关的 API，实际上在创建节点时是可以指定读写权限的，本节将详细介绍一下节点的权限机制。下面是创建 node 节点的代码：

```
public ObjectBean create(String domain, String node, Serializable obj,
AuthPolicy auth);
```

此时，可以指定一个权限参数，有只读（AuthPolicy.OP_READ）、读写（AuthPolicy.OP_READ_WRITE）、所有（AuthPolicy.OP_ALL）三种属性，默认为 AuthPolicy.OP_ALL。



注意

这里的权限属性是指创建进程对其他使用进程的权限约束，而不包括它自己。也就是对 node 的创建进程来说，它拥有对该 node 和 domain 所有操作权限（读写删，只要它不退出或者中止）。

假设现在创建了一个 domain 为 d，node 为 n 的节点，对于其他使用进程来说，操作权限如下所示：

权限 \ 其他进程	读 (get) n	写 (update) n	删 (delete) n	删 (delete) d
AuthPolicy.OP_READ	Yes	No	No	No
AuthPolicy.OP_READ_WRITE	Yes	Yes	No	No
AuthPolicy.OP_ALL	Yes	Yes	Yes	No

从列表中可以发现，当创建进程指定 node 的权限为 AuthPolicy.OP_ALL 时，其他使用进程可以删除该 node，但是不能删除其 domain，这是为什么呢？

因为 domain 下通常还有其他 node，它们的权限并不都是 AuthPolicy.OP_ALL，比如还有一个 n1 的 node 权限为 AuthPolicy.OP_READ，按照正常操作，该使用进程无法删除 n1，假设它可以删除 domain，那么它最后间接删除了 n1，于是发生了悖论。因此，为了避免风险，所有的使用进程只能根据权限删除 node，但是无法删除 domain。

不过如果你允许承担这样的删除风险，也可以在创建进程里强行指定该 domain 可删除，通过在 domain 创建后，调用：

```
public boolean setDeletable(String domain);
```

该方法只能被 domain 的创建进程调用，其他使用进程没有权限调用。

强行指定可删除后，其他进程可以直接删除该 domain 及所含 node 并忽略后果。

关于权限操作的例子详细参见 3.6.4 节“多节点权限操作示例”。

3.4 相对于 ZooKeeper 的区别

ZooKeeper 无疑是一款成功的开源产品，并拥有广泛的信任者和应用场景，和以往一样，老外作者在 Apache 网站上发布了一款产品，我们的工程师马上会虚心地学习和忠心捍卫，而国产原创的产品往往会遭到百般质疑，因为我们的原创更多是抄袭和粗制滥造，我们的国产更多是框架集成而不是架构设计。这种情感上的倾向性不是一天能改变的。

做产品对比和列举优势往往容易引起激烈争论，会被认为是在宣传和引导产品使用，实际上在都能满足功能需求的情况下，选择使用哪款产品更多是政治因素决定的，而不是技术因素，领导意志及工程师本身的熟悉程度和爱好等都是决定因素。

这里我们仅仅从技术角度阐述几点优势，ZooKeeper 作为一个 Chubby 和 Paxos 模仿品，缺乏创新型的设计改进，它仍然存在以下缺点：

- 树型配置节点的繁琐复杂，性能低下。为了保证这种结构，ZooKeeper 需要维持一套虚拟文件结构的开销，对于目录结构深的树节点造成性能影响，而配置信息结构实际上往往不一定需要树结构。
- “观察”（watch）机制的僵化设计：ZooKeeper 没有获取最新版本信息的方法支持，它

只能粗暴地在每次写入更新等方法时注册一个 watch，当这些方法被调用后就回调，它不考虑信息内容是否变化，对于没有使信息内容发生改变的更新，ZooKeeper 仍然会回调，并且 ZooKeeper 的回调比较呆板，它只能用一次，如果信息持续变化，必须又重新注册 watch。而 Fourinone 的事件处理则可以自由控制是否持续响应信息变化。

- 领导者选举机制实现得太过局限，集群只有两个节点，ZooKeeper 无法进行领导者选举，ZooKeeper 的领导者选举必须要奇数节点的奇怪限制。另外，ZooKeeper 的领导者选举实现虽然比原始的 Paxos 要简化，但是它仍然存在领导者（Leader）、跟随者（Follower）、观察者（Observer）、学习者（Learner）等众多角色，以及跟随状态（Following）、寻找状态（Looking）、观察状态（Observing）、领导状态（Leading）等复杂状态。相对于 Fourinone 的领导者选举，ZooKeeper 仍然不够直观简洁，难以用较少配置和代码演示。
- Windows 系统上几乎不支持，需要安装 Linux 壳，并且仅建议用于学习研究。Fourinone 支持 Windows、Linux 集群混合使用。

Fourinone 提出一种新的分布式协调系统设计，在满足 ZooKeeper 所有功能下，克服了以上缺点，提出了新的配置结构、变化事件机制、简化的领导者选举实现，能更好地满足分布式协调需求。

3.5 与 Paxos 算法的区别

Paxos 在维持领导者选举或者变量修改一致性上，采取一种类似议会投票的过半同意机制，比如设定一个领导者，需要将此看做一个议案，征求过半同意，每个节点通过一个议案会有编号记录，再次收到此领导者的不同人选，发现已经有编号记录便驳回，最后以多数通过的结果为准。

我们举个简单的例子，来阐述一下 Paxos 的基本思想：

假设我们有 5 台计算机 A、B、C、D、E，每台计算机保存着公司 CEO 的信息，现在 CEO 任期到了，需要进行新一届选举了。

A 计算机发起一个选举议案，提议 CEO 为“张三”，如果没有其他候选人议案，也没有网络问题，只要其中半数以上计算机收到并通过议案，那么最终“张三”当选 CEO。

由于是分布式环境，并发请求、机器故障、网络故障等问题是常态，如果 A 和 E 同时提交选举议案，A 提名“张三”，E 提名“李四”，那么肯定会涉及多计算机的一致性问题了：

假设 A、B、C 先收到 A 的议案，D、E 先收到 E 的议案，那么 A 继续提交给 D 时，D 告诉它已经先收到 E 的议案了，因此驳回了 A 的请求。同样 E 继续提交给 A、B、C 时也碰

到相同的问题。我们可以通过“在每台计算机同时接受议案提交时设置一个编号，编号先的通过，编号后的驳回”的方式来实现。

议案提交上去后，发现 A、B、C 投票“张三”为 CEO，D、E 投票“李四”为 CEO，少数服从多数，因此最后结果为“张三”当选 CEO。

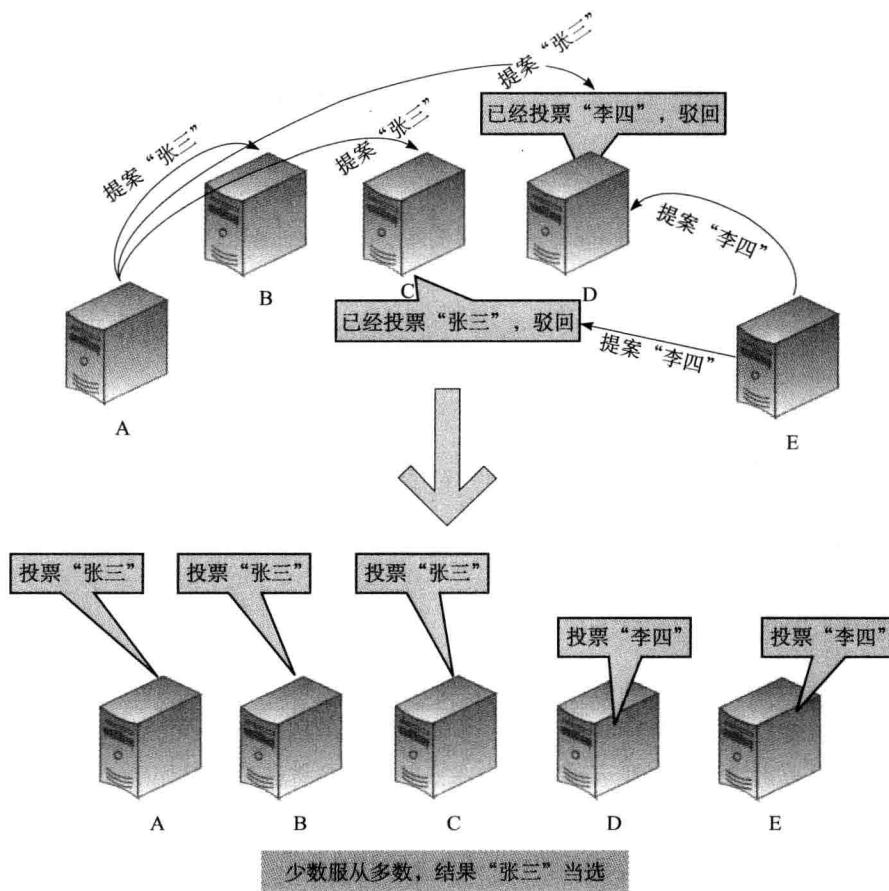


图 3-2 Paxos 算法示意图

如果是 C 计算机发生了网络问题或者故障，双方投票相同，那么选举无法完成。

如果 C 计算机发生了网络问题或者故障，A、B、D 投票“张三”，E 投票“李四”，那么结果为“张三”当选，而 C 对于这些情况一无所知，但是当 C 计算机恢复正常时，他会发起一个“询问谁是 CEO”的议案获取最新信息。

简言之，Paxos 对每个节点的并发修改采取编号记录的方式保持一致性，对多个节点的并发修改采取少数服从多数的方式保持一致性。Paxos 有点类似分布式二阶段提交方式，但是又不同，二阶段提交不能是多数节点同意，必须是全部同意。为了遵守过半节点同意的约

束，Paxos 算法往往要求节点总数为奇数。

Fourinone 选取领导者采取的是一种谦让方式，集群中节点会先询问其他节点是否愿意当领导者，没人愿意它才担任；如果已经有了领导了，那它就谦让；正因为大家都谦让，不互相争抢，领导者之间能避免冲突保持一致性。一旦确定了领导者，就只跟该领导者打交道，所有对变量的操作都是通过领导者进行，不会再去操作其他候选节点，操作结果由领导者统一同步到候选节点，跟上面 Paxos 算法保证一致性的方式是不一样的，Paxos 算法会去访问和操作所有节点征求同意，最后以多数节点的结果生效。

到此，我们可以想像一下基于 Paxos 方式的实现难度和工作量，Fourinone 在领导者选举和一致性上要更加简化和直接。

3.6 实践与应用

3.6.1 如何实现公共配置管理

在分布式多台机器环境下，维持统一的配置信息是最常见的需求，当配置信息改变时，所有的机器能实时获取并更新。

Fourinone 通过 Park 进行配置信息管理，Park 提供创建和修改信息的方法，并支持轮循和事件响应两种方式获取变化的对象，两种方式的效果一样。

下面是主要 API 的设计：

SetConfig: 在 ParKServer 上建立一个“domain=浙江、node=杭州、value=西湖”的配置信息，并且在 8 秒后把“西湖”改为“余杭”。

GetConfigA: 演示了以轮循方式监控配置信息的变化，它调用一个 getLastest 的方法，该方法可以传入一个旧版本的对象，并对比获取最新版本的对象，如果有就打印，如果没有最新版本，就返回 null。

GetConfigB: 演示了事件响应方式监控配置信息变化，它实现一个 LastestListener 的事件接口并进行注册，当信息变化时，会产生事件并获取到变化后的对象进行处理，LastestListener 的 happenLastest 方法有个 Boolean 返回值，如果返回 false，它会一直监控配置信息变化，继续有新的变化时还会进行事件调用；如果返回 true，它完成本次事件调用后就终止。

运行步骤：

- 1) 启动 ParkServerDemo（它的 IP 端口已经在配置文件中指定），结果如图 3-3 所示：

```
Java -classpath fourinone.jar; ParkServerDemo
```

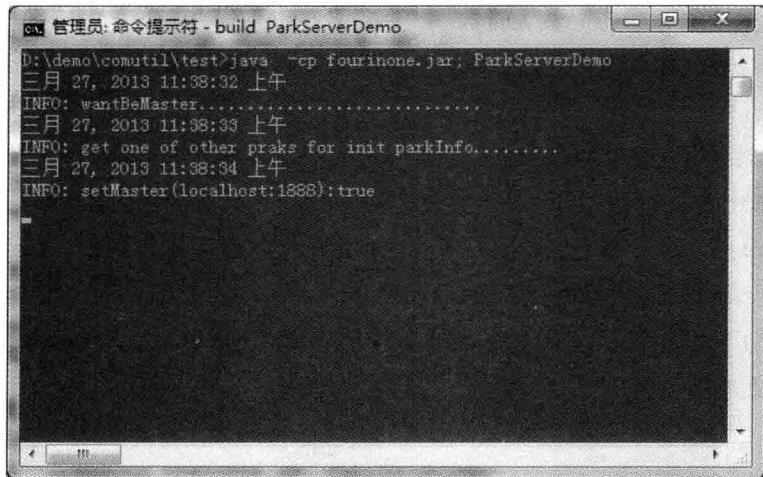


图 3-3 ParkServerDemo

2) 运行 GetConfigA, 结果如图 3-4 所示。

```
java -cp fourinone.jar; GetConfigA
```

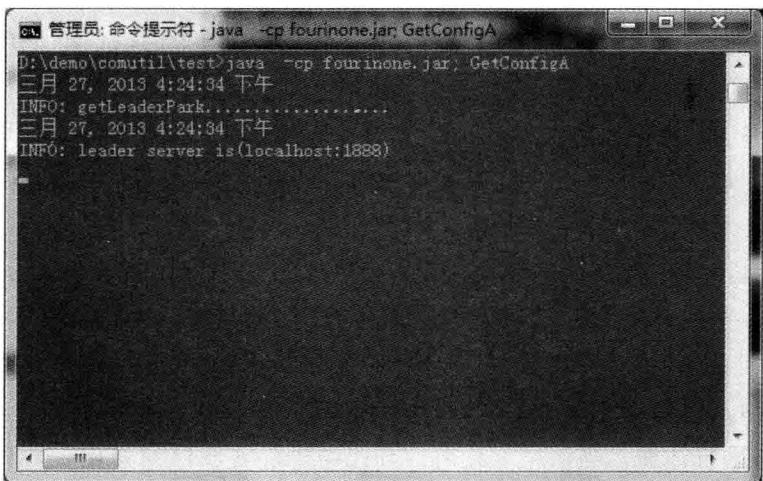


图 3-4 GetConfigA

3) 运行 GetConfigB, 结果如图 3-5 所示。

```
java -cp fourinone.jar; GetConfigB
```

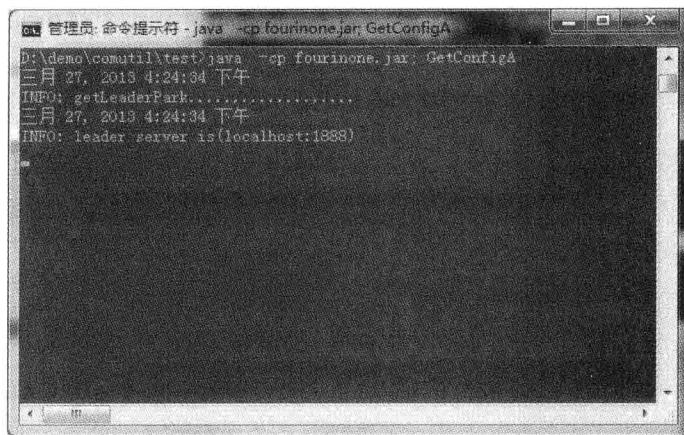


图 3-5 GetConfigB

4) 运行 SetConfig, 结果如图 3-6 所示:

```
java -cp fourinone.jar; SetConfig
```



图 3-6 SetConfig

可以看到 SetConfig 对节点 “domain= 浙江、node= 杭州” 做了更改后，GetConfigA 和 GetConfigB 都同时获取到了更新。

如果是线上环境，为避免 ParkServer 容机，ParkServer 可以配置 Master 和任意数量的 Slave，请使用 ParkMasterSlave 替换上面的 ParkServerDemo 即可，每次输入 M 或者 S 启动 Master 或者 Slave，运行过程关掉 Master，GetConfig 仍然可以从 Slave 获取配置信息。

我们打开 ParkMasterSlave 程序可以看到如下代码：

```
String[][] master = new String[][]{{"localhost", "1888"}, {"localhost", "1889"}};
String[][] slave = new String[][]{{"localhost", "1889"}, {"localhost", "1888"}};
```

注意

配置 Master/Slave 时请注意一下顺序，需要将自己的地址放在最前面，备用的地址放后面，比如：

对 Master 来说：{ "localhost", "1888" }，{ "localhost", "1889" }；

对 Slave 来说：{ "localhost", "1889" }，{ "localhost", "1888" }

下面我们演示一下 ParkServer 如果出现故障，通过配置 Master/Slave 来维持正常的服务，还是以刚才的 demo 为例。

1) 启动 ParkMasterSlave Master (它的 IP 端口已经在程序内指定)，结果如图 3-7 所示：

```
java -cp fourinone.jar; ParkMasterSlave M
```

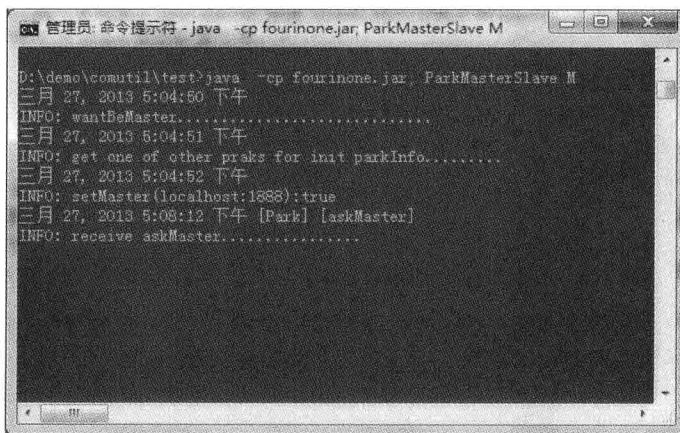


图 3-7 ParkMasterSlave M

2) 启动 ParkMasterSlave Slave (它的 IP 端口已经在程序内指定)，结果如图 3-8 所示：

```
java -cp fourinone.jar; ParkMasterSlave S
```

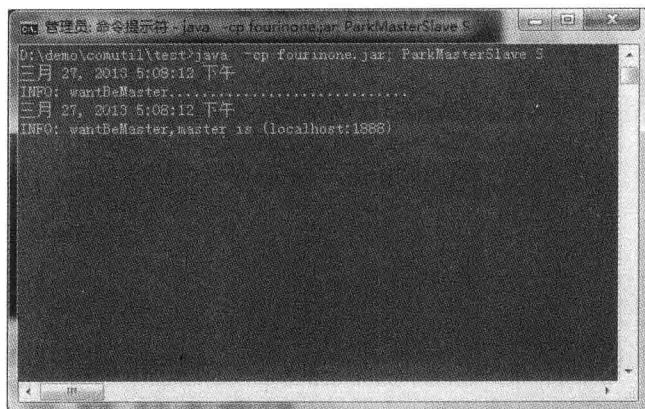


图 3-8 ParkMasterSlave S

可以看到, Slave 窗口输出: INFO: wantBeMaster,master is (localhost:1888), 表示集群中之前启动的 1888 是领导者。

3) 运行 GetConfigA 和 GetConfigB

```
java -cp fourinone.jar; GetConfigA
java -cp fourinone.jar; GetConfigB
```

4) 这个时候假设 ParkServer 的 Master 挂掉了, 我们把启动的 Master 关闭 (Ctrl+C), 可以看到如图 3-9 所示的结果。

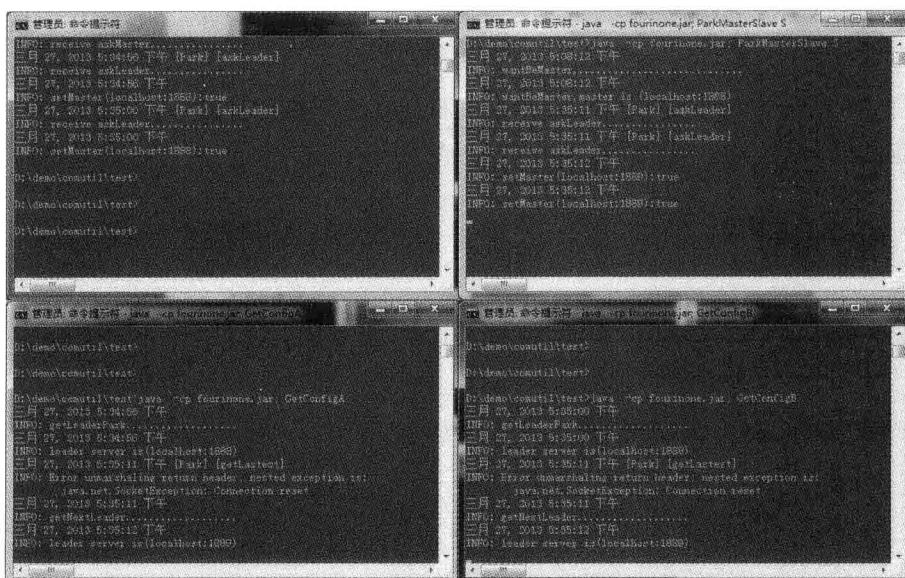


图 3-9 Master 挂掉

ParkServer Slave 和 GetConfigA 跟 GetConfigB 窗口都出现了提示，领导者切换到 Slave 上继续提供服务。

5) 运行 SetConfig，结果如图 3-10 所示：

```
java -cp fourinone.jar; SetConfig
```



图 3-10 SetConfig

可以看到 SetConfig 会切换到 Slave 的 localhost:1889 上设置配置信息，并且 GetConfigA 跟 GetConfigB 也都会从 localhost:1889 上获取信息和响应更新，而不会因为 master (localhost:1888) 故障受影响。

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// ParkMasterSlave
import com.fourinone.BeanContext;
```

```
public class ParkMasterSlave
{
    public static void main(String[] args)
    {
        String[][] master = new String[][]{{"localhost","1888"}, {"localhost",
        "1889"}};
        String[][] slave = new String[][]{{"localhost","1889"}, {"localhost",
        "1888"}};

        String[][] server = null;
        if(args[0].equals("M"))
            server = master;
        else if(args[0].equals("S"))
            server = slave;

        BeanContext.startPark(server[0][0],Integer.parseInt(server[0][1]),
        server);
    }
}

// GetConfigA
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;

public class GetConfigA
{
    public static void main(String[] args)
    {
        ParkLocal pl = BeanContext.getPark();
        ObjectBean oldob = null;
        while(true){
            ObjectBean newob = pl.getLastest("zhejiang", "hangzhou", oldob);
            if(newob!=null){
                System.out.println(newob);
                oldob = newob;
            }
        }
    }
}

// GetConfigB
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.LastestListener;
import com.fourinone.LastestEvent;
import com.fourinone.ObjectBean;

public class GetConfigB implements LastestListener
{
    public boolean happenLastest(LatestEvent le)
    {
        ObjectBean ob = (ObjectBean)le.getSource();
```

```

        System.out.println(ob);
        return false;
    }

    public static void main(String[] args)
    {
        ParkLocal pl = BeanContext.getPark();
        pl.addLastestListener("zhejiang", "hangzhou", null, new GetConfigB());
    }
}

// SetConfig
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;

public class SetConfig
{
    public static void main(String[] args)
    {
        ParkLocal pl = BeanContext.getPark();
        ObjectBean xihu = pl.create("zhejiang", "hangzhou", "xihu");
        System.out.println("Waiting...");
        try{Thread.sleep(8000);}catch(Exception e){}
        ObjectBean yuhang = pl.update("zhejiang", "hangzhou", "yuhang");
    }
}
}

```

3.6.2 如何实现分布式锁

分布式锁可以让多个分布式应用依照先后顺序保持一致性操作，因此在很多方面都有广泛的应用，比如我们在 2.1.7 计算中的故障处理也谈到 2 个工头通过竞争一个分布式锁去做任务调度。

LockDemo 是利用 Fourinone 进行分布式锁的实现，设计思路如下：

1) 可以启动多个 LockDemo 实例，每个实例在 Fourinone 上建立一个自己的 node，node 的 domain 为 lock，node 的值为 node（为方便演示，将 node 的名称和值设为一样）。

2) 然后再轮循判断 domain lock 的第一个元素是否是自己的 node，如果是，就执行，这里模拟线程执行 8 秒，执行完将自己的 node 删除，代表释放锁；如果不是，就继续等待。

运行步骤如下：

1) 启动 ParkServerDemo（它的 IP 端口已经在配置文件指定），结果如图 3-11 所示：

```
java -cp fourinone.jar; ParkServerDemo
```

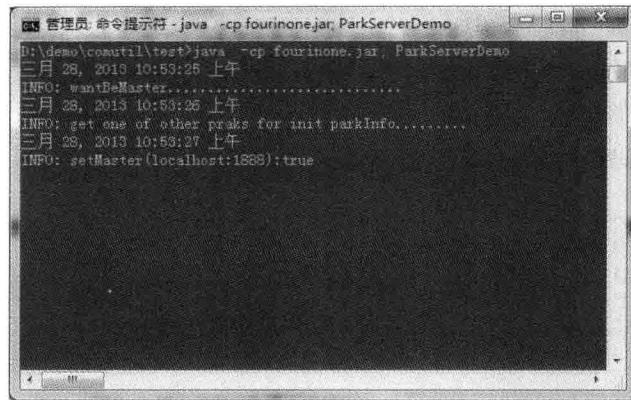


图 3-11 ParkServerDemo

2) 运行 LockDemo，node 名称动态传入参数：

```
java -cp fourinone.jar; LockDemo ccc
java -cp fourinone.jar; LockDemo bbb
java -cp fourinone.jar; LockDemo aaa
```

先后启动了多个 LockDemo 实例，观察它们按照预期的顺序先后获取到锁并进行 8 秒的业务操作。如图 3-12 所示。



图 3-12 LockDemo

可以看到，由于启动先后顺序为 ccc、bbb、aaa，ccc 首先获取到锁并开始执行，bbb 和 aaa 处于等待中；ccc 处理完成后，bbb 获取到锁，开始执行，aaa 继续处于等待中；bbb 处理完成后，最后 aaa 开始执行。

下面是 demo 源码：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
    public static void main(String[] args){
        BeanContext.startPark();
    }
}

// LockDemo
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import java.util.List;

public class LockDemo
{
    public void lockutil(String node)
    {
        ParkLocal pl = BeanContext.getPark();
        ObjectBean ob = pl.create("lock", node, node);

        System.out.print("try get lock.");
        while(true){
            List<ObjectBean> oblist = pl.get("lock");
            String curnode = (String)oblist.get(0).toObject();
            if(curnode.equals(node)){
                System.out.println("");
                System.out.println("ok, get lock and doing...");
                try{Thread.sleep(8000);}catch(Exception e){}
                pl.delete("lock", node);
                System.out.println("done.");
                break;
            }
            else
                System.out.print(".");
        }
    }

    public static void main(String[] args)
    {
        LockDemo ld = new LockDemo();
        ld.lockutil(args[0]);
    }
}
```

3.6.3 如何实现集群管理

对于像淘宝这样上万台服务器集群环境的大型互联网应用，通常我们面临这样一种需求：我们需要一个集群管理者管理集群里的服务器，同一个集群中任何一台服务器宕机，其他服务器都能感知。如果是集群管理者宕机，集群中所有的服务器不能受任何影响，能实时切换到备份管理者上提供服务。

这个 demo 演示了如何利用 Fourinone 用简单几行代码实现上述功能：

- **GroupManager**：是一个集群管理者，它有 master 和 slave 两个实例，实际上你可以建立任意多的 slave。
- **GroupServer**：代表一个集群中的 server，它启动后会注册自己的信息到集群管理者，然后监控集群中其他机器的状况并实时反馈。它使用一个 getLastest 的 API，这个 API 可以返回最新的集群状况，如果不是最新的就返回 null。

```
pl.create("group", args[0], args[0], AuthPolicy.OP_ALL, true);
```

- 上面的方法进行节点的注册，其中前 3 个参数分别是 domain、node、value，AuthPolicy.OP_ALL 表示该节点的权限为公共，也就是可以被其他进程修改删除，true 代表它是个保持连接节点，如果失去连接，该节点会被删除。

运行步骤：

1) 启动 GroupManager 进程，输入参数分别为 M，代表 master，结果如图 3-13 所示：

```
java -cp fourinone.jar; GroupManager M
```

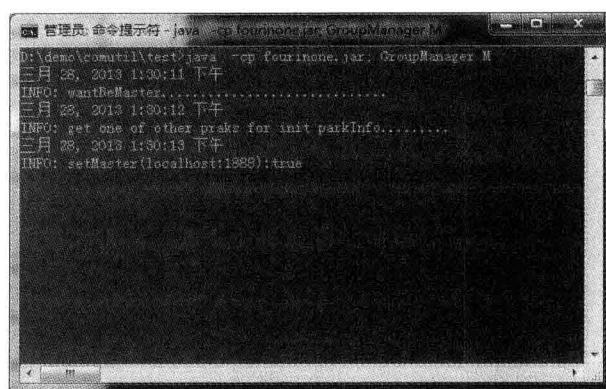


图 3-13 GroupManager M

2) 启动 GroupManager 进程，输入参数分别为 S，代表 slave，结果如图 3-14 所示：

```
java -cp fourinone.jar; GroupManager S
```



图 3-14 GroupManager S

3) 启动 3 个 GroupServer 进程，每次输入参数分别为“one,two,three”代表 3 台集群 server（它访问 master 的 IP 端口已经在配置文件指定），结果如图 3-15 所示。

```
java -cp fourinone.jar; GroupServer one
java -cp fourinone.jar; GroupServer two
java -cp fourinone.jar; GroupServer three
```

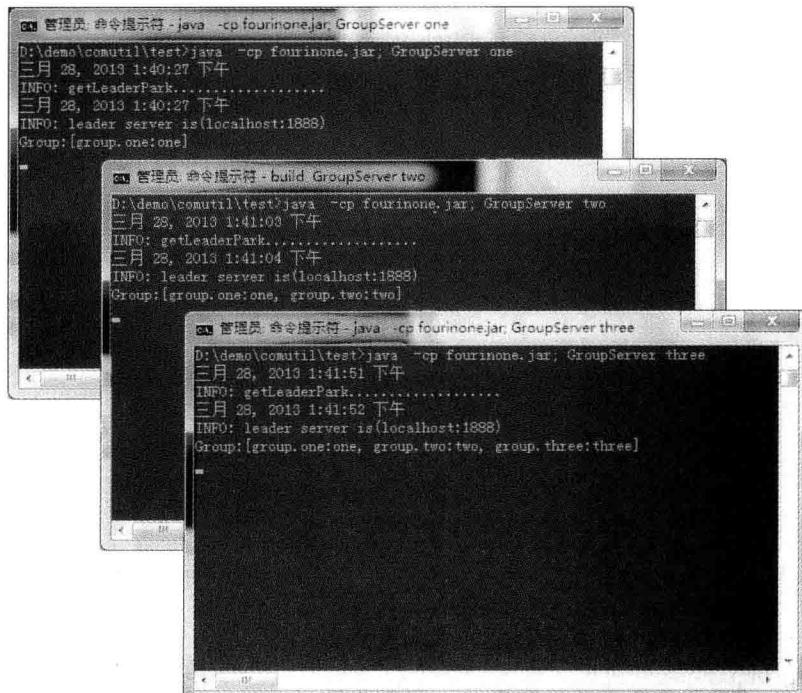


图 3-15 GroupServer

观察每个 GroupServer 进程里的输出，每个进程窗口都会有集群里所有服务器的信息输出，可以关掉其中一个进程模拟一个 GroupServer 宕机，此时其他两个 GroupServer 进程会实时输出集群更新信息，请再关掉 master 进程模拟 GroupManager 宕机，会发现两个 GroupServer 进程会即时选取 slave 为新的领导者，请把刚才关掉的一个 GroupServer 进程恢复，会发现所有 GroupServer 在 master 宕机情况下，也能实时得到集群的最新信息。

下面我们演示这一系列操作：

1) 关掉 GroupServer three，如图 3-16 所示。



图 3-16 关闭 GroupServer

可以看到，GroupServer three 关闭后，GroupServer one 和 GroupServer two 都输出了集群更新信息。

2) 再关掉 GroupManager master，如图 3-17 所示。

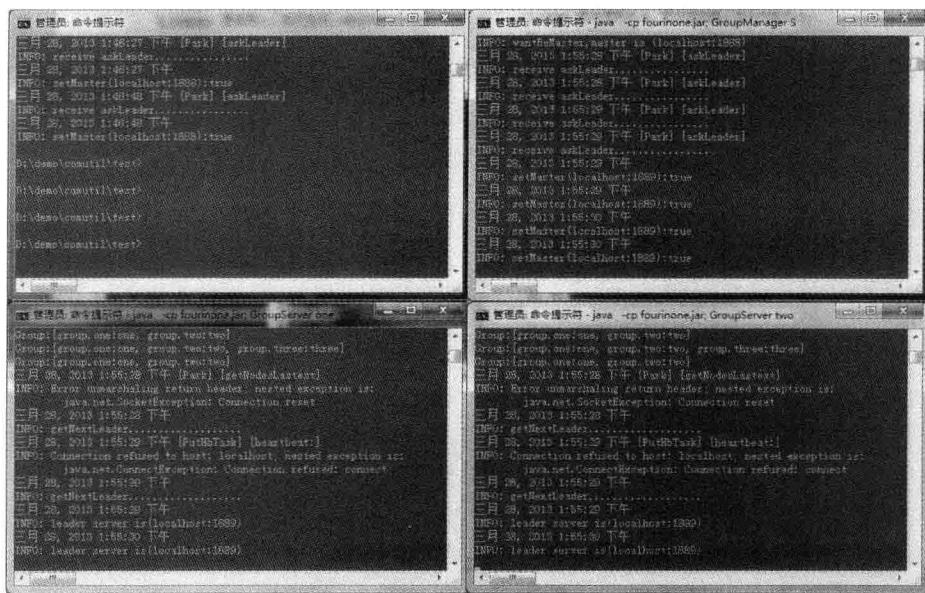


图 3-17 关闭 GroupManager

可以看到，当 GroupManager master 关闭时，GroupManager slave 成为集群管理的领导者，并且 GroupServer one 和 GroupServer two 都切换到 GroupManager slave 上。

3) 恢复 GroupServer three，如图 3-18 所示。

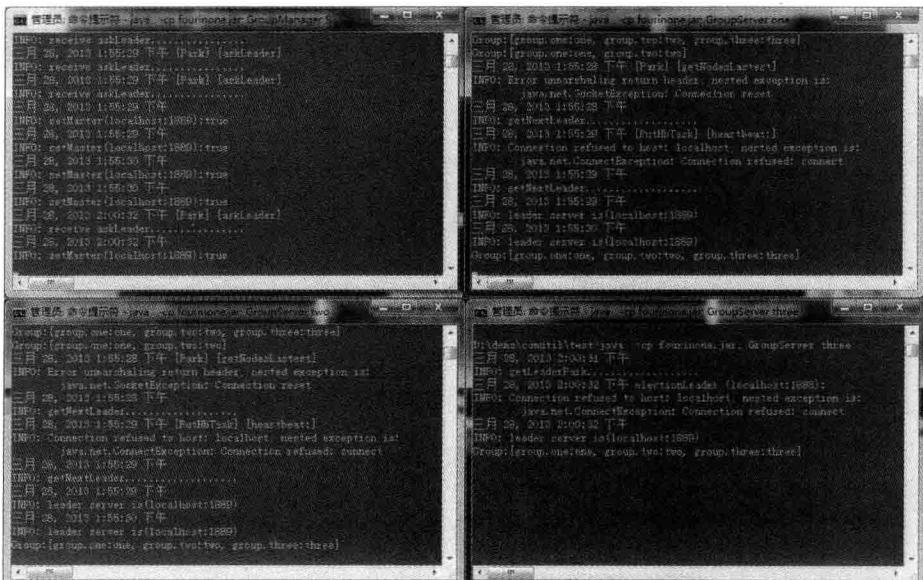


图 3-18 恢复 GroupServer

从上面可以发现，虽然 GroupManager master 挂了，但是 GroupServer three 重新加入集群，GroupServer one 和 GroupServer two 仍然可以通过 GroupManager slave 感知到集群的状况。

注意

在集群管理或者集群配置信息等应用中，对及时性要求较高，可以将心跳时间调整得快些，避免延迟。比如将默认的 3000 毫秒改为 1000 毫秒或者其他值，根据自己的服务器性能和网络质量等因素去定：

```
<HEARTBEAT>1000</HEARTBEAT> (配置文件中 Park 部分)
```

下面是 demo 源码：

```
// GroupManager
import com.fourinone.BeanContext;

public class GroupManager
{
    public static void main(String[] args)
    {
        String[][] master = new String[][]{{"localhost","1888"}, {"localhost",
                "1889"}};
        String[][] slave = new String[][]{{"localhost","1889"}, {"localhost",
                "1888"}};

        String[][] server = null;
        if(args[0].equals("M"))
            server = master;
        else if(args[0].equals("S"))
            server = slave;

        BeanContext.startPark(server[0][0], Integer.parseInt(server[0][1]),
                server);
    }
}

// GroupServer
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import com.fourinone.AuthPolicy;
import java.util.List;

public class GroupServer
{
    public static void main(String[] args)
    {
        ParkLocal pl = BeanContext.getPark();
        pl.create("group", args[0], args[0], AuthPolicy.OP_ALL, true);

        List<ObjectBean> oldls = null;
        while(true){
```

```
        List<ObjectBean> newls = pl.getLastest("group", oldls);
        if(newls!=null){
            System.out.println("Group:"+newls);
            oldls = newls;
        }
    }
}
}
```

3.6.4 多节点权限操作示例

根据 3.3 节权限机制介绍，下面是一个操作节点的演示 demo，请留意各自节点的权限范围，程序说明如下：

- 1) ParkServerDemo: 启动 parkserver (它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定)。
- 2) ParkSet: 往 parkserver 里创建了 d1n1、d2n2、d3n3、d4n4 共 4 个节点，分别对应只读、读写，所有，所有 + 强行删除权限。
- 3) ParkGet: 依次对 d1n1、d2n2、d3n3、d4n4 进行读、写、删除、删除 domain 操作，观察结果输出，如果没有权限操作，parkserver 会输出信息，并且操作返回的结果对象为空。

启动命令和顺序：

```
Java -classpath fourinone.jar; *.java
Java -classpath fourinone.jar; ParkServerDemo
Java -classpath fourinone.jar; ParkSet
Java -classpath fourinone.jar; ParkGet
```

下面是 demo 源码：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo{
    public static void main(String[] args){
        BeanContext.startPark();
    }
}

// ParkSet
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import com.fourinone.AuthPolicy;
public class ParkSet{
    public static void main(String[] args){
        // 获取 parkserver 用户接口
        ParkLocal pl = BeanContext.getPark();
```

```

// 在 domain d1 下创建节点 node n1, 指定权限为只读
ObjectBean d1n1 = pl.create("d1","n1","v1",AuthPolicy.OP_READ);
if(d1n1!=null)
    System.out.println("d1n1 with AuthPolicy.OP_READ create success!");

// 在 domain d2 下创建节点 node n2, 指定权限为读写
ObjectBean d2n2 = pl.create("d2","n2","v2",AuthPolicy.OP_READ_WRITE);
if(d2n2!=null)
    System.out.println("d2n2 with AuthPolicy.OP_READ_WRITE create success!");

// 在 domain d3 下创建节点 node n3, 指定权限为所有
ObjectBean d3n3 = pl.create("d3","n3","v3",AuthPolicy.OP_ALL);
if(d3n3!=null)
    System.out.println("d3n3 with AuthPolicy.OP_ALL create success!");

// 在 domain d4 下创建节点 node n4, 指定权限为所有, 并且创建完成强行设置为其他进程可删除
ObjectBean d4n4 = pl.create("d4","n4","v4",AuthPolicy.OP_ALL);
if(d4n4!=null)
    System.out.println("d4n4 with AuthPolicy.OP_ALL create success!");
boolean r = pl.setDeletable("d4");
if(r)
    System.out.println("set d4 deletable!");
}

}

// ParkGet
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import java.util.List;
public class ParkGet{
    public static void main(String[] args){
        // 获取 parkserver 用户接口
        ParkLocal pl = BeanContext.getPark();

        // 获取节点 d1n1, 节点权限为 AuthPolicy.OP_READ
        ObjectBean d1n1 = pl.get("d1","n1");// 获取节点
        System.out.println("get d1n1:"+ (String)d1n1.toObject());
        d1n1 = pl.update("d1","n1","v1-update");// 更新节点
        if(d1n1!=null)
            System.out.println("update node d1n1 success!");
        else
            System.out.println("update node d1n1 failure!");
        List<ObjectBean> d1 = pl.delete("d1");// 删除 domain
        if(d1!=null)
            System.out.println("delete domain d1 success!");
        else
            System.out.println("delete domain d1 failure!");
        d1n1 = pl.delete("d1","n1");// 删除节点
    }
}

```

```

if(d1n1!=null)
    System.out.println("delete node d1n1 success!");
else
    System.out.println("delete node d1n1 failure!");

// 获取节点 d2n2, 节点权限为 AuthPolicy.OP_READ_WRITE
ObjectBean d2n2 = pl.get("d2","n2");
System.out.println("get d2n2:"+ (String)d2n2.toObject());
d2n2 = pl.update("d2","n2","v2-update");
if(d2n2!=null)
    System.out.println("update node d2n2 success!");
else
    System.out.println("update node d2n2 failure!");
List<ObjectBean> d2 = pl.delete("d2");
if(d2!=null)
    System.out.println("delete domain d2 success!");
else
    System.out.println("delete domain d2 failure!");
d2n2 = pl.delete("d2","n2");
if(d2n2!=null)
    System.out.println("delete node d2n2 success!");
else
    System.out.println("delete node d2n2 failure!");

// 获取节点 d3n3, 节点权限为 AuthPolicy.OP_ALL
ObjectBean d3n3 = pl.get("d3","n3");
System.out.println("get d3n3:"+ (String)d3n3.toObject());
d3n3 = pl.update("d3","n3","v3-update");
if(d3n3!=null)
    System.out.println("update node d3n3 success!");
else
    System.out.println("update node d3n3 failure!");
List<ObjectBean> d3 = pl.delete("d3");
if(d3!=null)
    System.out.println("delete domain d3 success!");
else
    System.out.println("delete domain d3 failure!");
d3n3 = pl.delete("d3","n3");
if(d3n3!=null)
    System.out.println("delete node d3n3 success!");
else
    System.out.println("delete node d3n3 failure!");

// 获取节点 d4n4, 节点权限为 AuthPolicy.OP_ALL
ObjectBean d4n4 = pl.get("d4","n4");
System.out.println("get d4n4:"+ (String)d4n4.toObject());
d4n4 = pl.update("d4","n4","v4-update");
if(d4n4!=null)
    System.out.println("update node d4n4 success!");
else

```

```
        System.out.println("update node d4n4 failure!");
        // 由于创建进程已经强行指定该 domain 可删除 setDeletable(d4)，因此这里可以删除掉
        List<ObjectBean> d4 = pl.delete("d4");
        if(d4!=null)
            System.out.println("delete domain d4 success!");
        else
            System.out.println("delete domain d4 failure!");
        d4n4 = pl.delete("d4","n4");// 这里删除节点会失败，因为上面已经删除了该 domain 下所有节点
        if(d4n4!=null)
            System.out.println("delete node d4n4 success!");
        else
            System.out.println("delete node d4n4 failure!");
    }
}
```

3.6.5 领导者选举相关属性设置

我们在前面小节已经基本了解了领导者选举的过程和应用，并学会配置了一个领导者候选者的 Master/Slave 的主备关系，当 Master 崩机或者网络故障时，可以不受影响切换到 Slave 上提供服务，这里我们补充一些关于领导者选举的属性设置，可以应用在不同的情形下。在 config.xml 的 park 部分配置里，我们可以看到：

```
<SERVERS>localhost:1888,localhost:1889</SERVERS>
<ALWAYSTRYLEADER>false</ALWAYSTRYLEADER>
```

ALWAYSTRYLEADER 这个配置项默认值是 false，代表是否一直进行领导者寻找，如果在集群中没有领导者，也没有候选人替补，应用程序会根据配置值一直寻找并等待下去，直到最后找到，还是寻找一轮后放弃，我们可以看看下面的简单例子。

LeaderTest 是一个简单的 ParkServer 应用端，它只有一行代码“ParkLocal pl = BeanContext.getPark();”，如果我们不启动 ParkServer 的 Master/Slave 主备结构，而直接运行 LeaderTest，代码如下，结果如图 3-19 所示。

```
java -cp fourinone.jar; LeaderTest

import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;

public class LeaderTest
{
    public static void main(String[] args)
    {
        ParkLocal pl = BeanContext.getPark();
    }
}
```

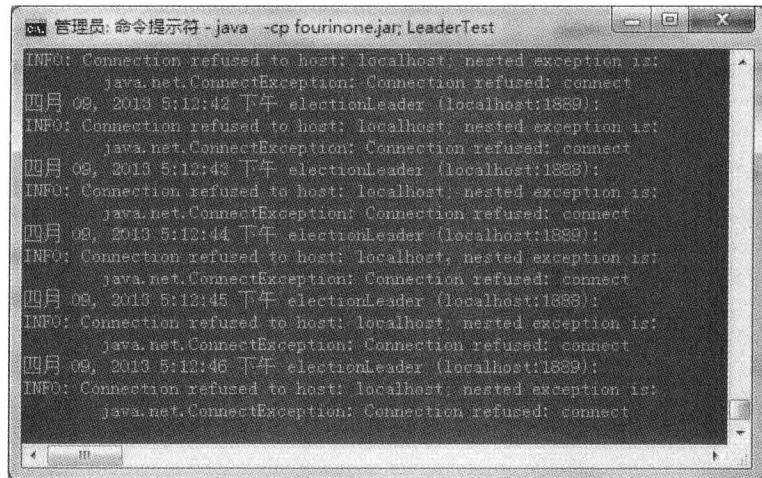


图 3-19 LeaderTest-1

我们发现 LeaderTest 会一直反复地请求 localhost:1888 和 localhost:1889，直到最后连接成功。如果我们不想这样无限制地寻找下去，可以将配置改为：

```
<ALWAYSTRYLEADER>true</ALWAYSTRYLEADER>
```

我们再启动 LeaderTest，如图 3-20 所示。

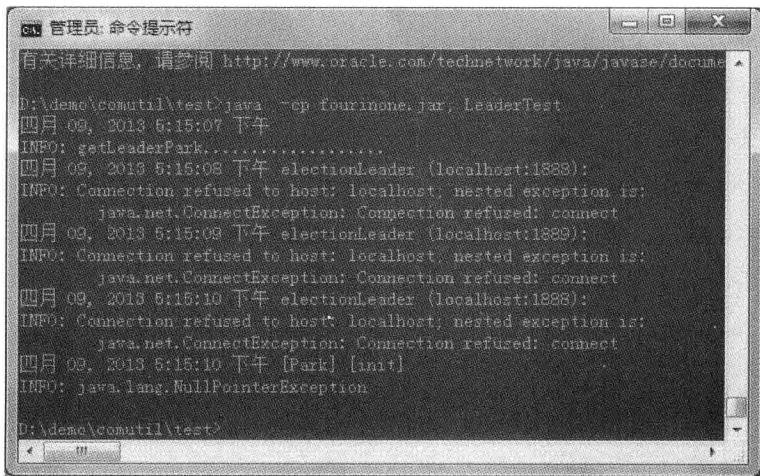


图 3-20 LeaderTest-2

我们发现 LeaderTest 按照“localhost:1888、localhost:1889、localhost:1888”寻找了一轮后，发现集群仍然没有可用的领导者，便停止寻找而退出。

第4章 分布式缓存的实现

本章讲述小型缓存、大型分布式缓存的原理和实现机制，并且讲述经典的一致哈希算法原理，以及改进的基于日期 key 取模和分组算法去做集群负载均衡和扩容，最后讲述一个分布式 Session 的实现案例，以及相关的配置属性。

4.1 小型网站或企业应用的缓存实现架构

缓存的概念来源于操作系统，我们知道 CPU 通过 load 和 store 的指令实现寄存器和内存的交互，将需要执行的指令缓存起来准备执行。我们的企业应用也面临同样的需要，对于频繁使用的用户信息或者数据查询，也需要放在一个缓存系统里，目的是提升效率和减少对底层资源的耗时操作。

为中小型的网站或者企业应用设计一个缓存系统很简单，设计一个面向内存使用的键 / 值存储的 CacheServer 即可，并且可以实现主备切换，以保证可靠性，如图 4-1 所示。

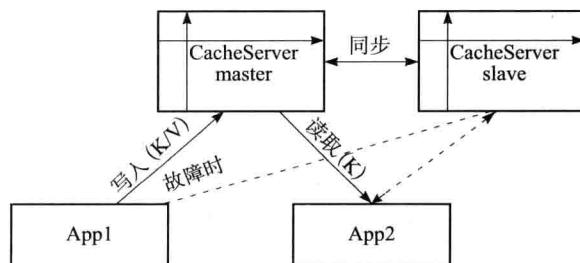


图 4-1 中小型的网站或者企业应用的缓存系统架构图

我们可以通过 ParkServer 去实现小型缓存，利用 domain/node 进行键 / 值的存储即可，因为 domain/node 都是内存操作而且读写锁分离，同时拥有复制备份领导者切换功能，完全

满足缓存的高性能与可靠性需求。直接启动一个 ParkServer，分别在两个 Java 进程中使用 ParkLocal 的 create 和 get 方法即可实现缓存的读写操作（详见 4.2 节中的 demo）。

但是对于大型互联网应用，高峰访问量上百万的并发读写吞吐量，会超出单台服务器的承受能力，我们需要改进上面的设计，考虑解决负载和扩容等问题。

4.2 大型分布式缓存系统实现过程

我们接着上一节的思路，如果是大型网站的缓存，单台 ParkServer 的压力不能承受，需要建立多台 CacheServer，并使用 CacheFacade 进行负载均衡。CacheFacade 会根据 key 自动寻找存储它的 CacheServer，数据在多台 CacheServer 上是均匀分布的，虽然每台 CacheServer 的数据都不一样，但是每台 CacheServer 都可以有自己的备份服务器，CacheServer 出现故障时，几乎实时就能切换到备份服务器处理请求，所以既能保证高性能又能保证高可靠。改进后的架构如图 4-2 所示。

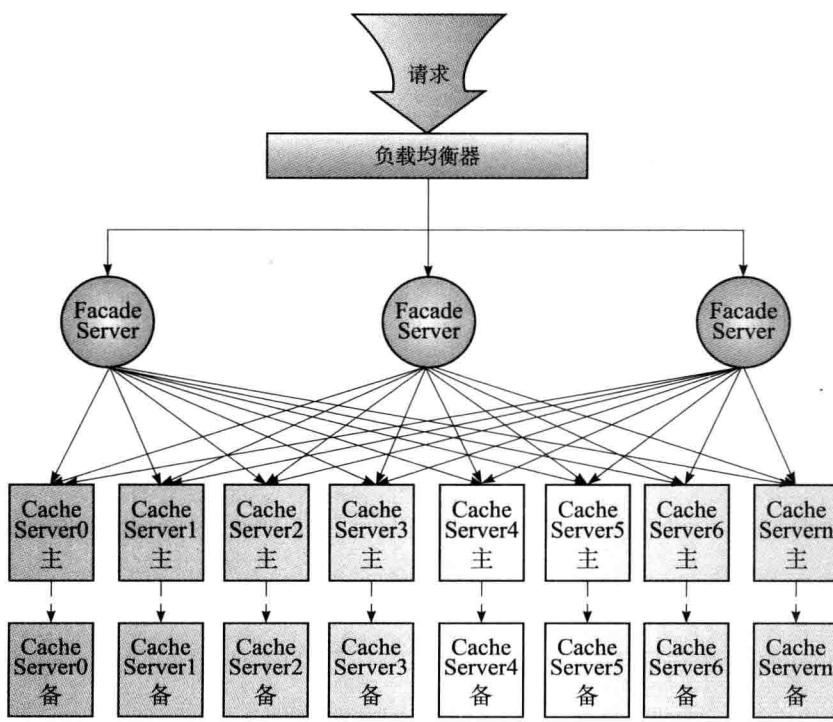


图 4-2 大型分布式缓存系统实现架构

Fourinone 提供了 FacadeServer 的解决方案去解决大集群的分布式缓存，利用硬件负载均衡路由到一组 Facade 服务器上，Facade 可以自动为缓存内容生成 key，并根据 key 准确找

到散落在背后的缓存集群的具体哪台服务器，当缓存服务器的容量到达限制时，可以自由扩容，不需要成倍扩容，因为 Facade 的算法会登记服务器扩容时间版本，并将 key 智能地跟这个时间匹配，这样在扩容后还能准确找到之前分配到的服务器。我们在 4.4 节还会详细介绍日期 key 取模算法，并介绍如何基于分布式缓存实现一个 Session 系统。

Facade 服务器部署在一系列的服务器上，对应图 4-2 所示的架构图里的 FacadeServer，缓存服务器对应图 4-2 中的 CacheServer。

FacadeServer 是一组相同的软件服务器，它们彼此可互相替换，也可以自由增加或减少，具体取决于请求读写量的高峰期规模，FacadeServer 的作用是根据读写请求自动判断数据落到后端哪台 CacheServer 上，它在完成这个匹配计算的过程中不需要借助其他任何服务器，因此它不会有瓶颈限制。另外，FacadeServer 并不限制只跟后端集群中的某几台 CacheServer 发生交互，它可以跟任何 CacheServer 交互，而且它有很强的故障处理能力和高可用性。当它准备交互的 CacheServer 主机出现故障时，它会转而请求备机，如果主机和备机都出现故障时，它会重新选择其他的 CacheServer 直到成功，并且当 CacheServer 扩充时 FacadeServer 也能准确判断新数据的写和旧数据的读归属哪台 CacheServer。在物理上，每个 FacadeServer 独立部署在一台计算机服务器上。

CacheServer 是一个可以水平扩充的 cache 集群，它可以根据缓存规模增大而增大，它的扩容是任意的，没有规则限制，可以每次扩容一台至多台。cache 集群由多个单元组成，缓存的数据被 FacadeServer 水平切割存储在不同的单元里，虽然每个单元保存的数据都不同，但是每个单元都是主备结构，主备之间的数据是相同并且同步的。在物理上，每个 CacheServer 主备独立部署一台计算机服务器。

接下来的 DEMO 同时演示了小型缓存和大型缓存的使用：

- CachePutDemo：先将 100 条数据分布式存储在 A、B、C、三台缓存 Server 中，然后再将这 100 条数据的 key 保存在 ParkServer 的小型缓存中。
- CacheGetDemo：先将 100 条数据的 key 从 ParkServer 中取出，再根据 key 从分布式缓存的 A、B、C 三台 Server 中取出。

运行步骤如下：

- 1) 启动 3 个 CacheServer 进程，每个输入的参数分别为 A、B、C：

```
java -cp fourinone.jar; CacheServer A
java -cp fourinone.jar; CacheServer B
java -cp fourinone.jar; CacheServer C
```

- 2) 启动完的结果如图 4-3 所示，三个 CacheServer 已经准备就绪。

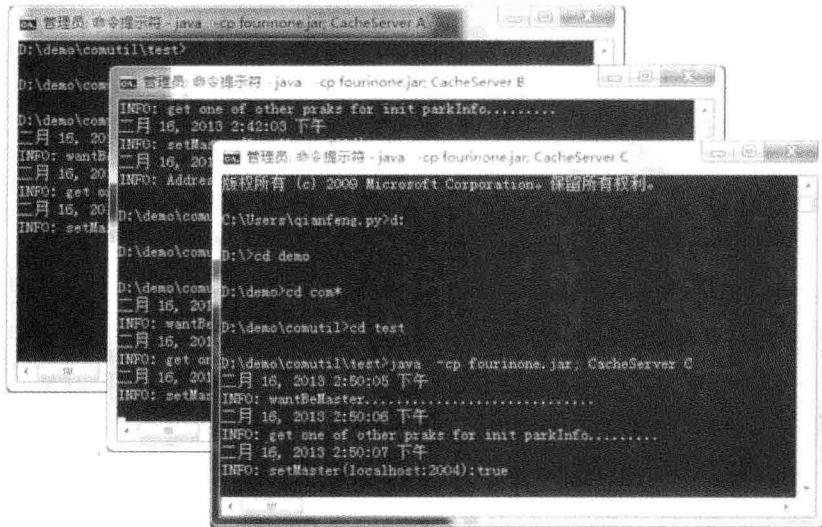


图 4-3 CacheServer

3) 启动 ParkServerDemo (它的 IP 端口已经在配置文件指定) :

```
java -cp fourinone.jar; ParkServerDemo
```

4) 启动好一个新的 ParkServer 进程, 如图 4-4 所示。

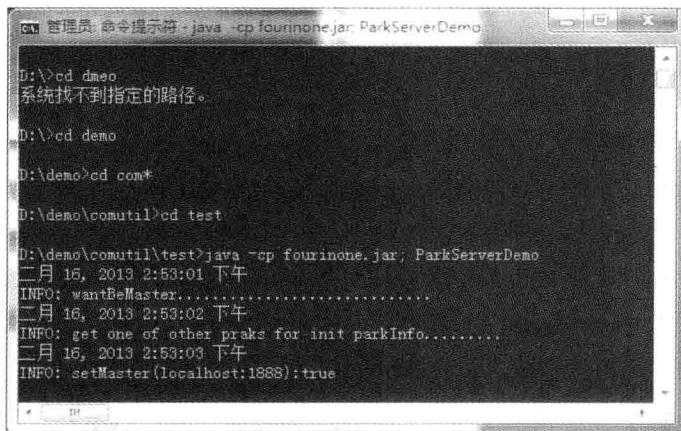


图 4-4 ParkServerDemo

5) 启动 CacheFacadeDemo (它的 IP 端口已经在配置文件指定) :

```
java -cp fourinone.jar; CacheFacadeDemo
```

6) 启动好一个 FacadeServer 如图 4-5 所示。

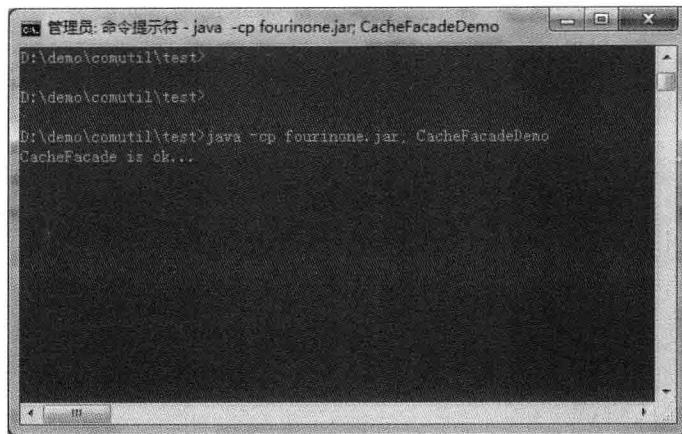


图 4-5 CacheFacadeDemo

在三个 CacheServer (用于分布式缓存)、1 个 ParkServer (用于小型缓存)、1 个 FacadeServer (用于负载均衡) 启动完成后，我们已经搭建好一个分布式缓存 DEMO，接下来我们就可以使用客户端写入和读取数据了。

7) 运行 CachePutDemo:

```
java -cp fourinone.jar; CachePutDemo
```

插入 100 条数据到 CacheServer 中，并将返回的 key 保存到 ParkServer，然后程序完成退出，如图 4-6 所示。

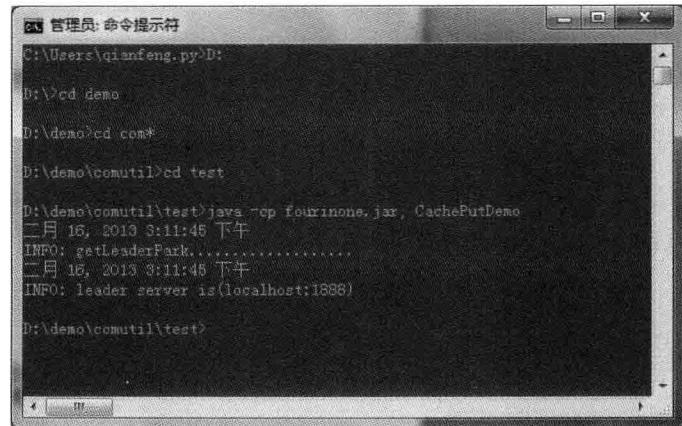


图 4-6 CachePutDemo

8) 运行 CacheGetDemo:

```
java -cp fourinone.jar; CacheGetDemo
```

可以看到，CacheGetDemo 先从 ParkServer 里取出所有的 key，再通过 key 到三个 CacheServer 上取出所有缓存的数据并显示出来，FacadeServer 负责完成 key 到 CacheServer 的路由。如图 4-7 所示。

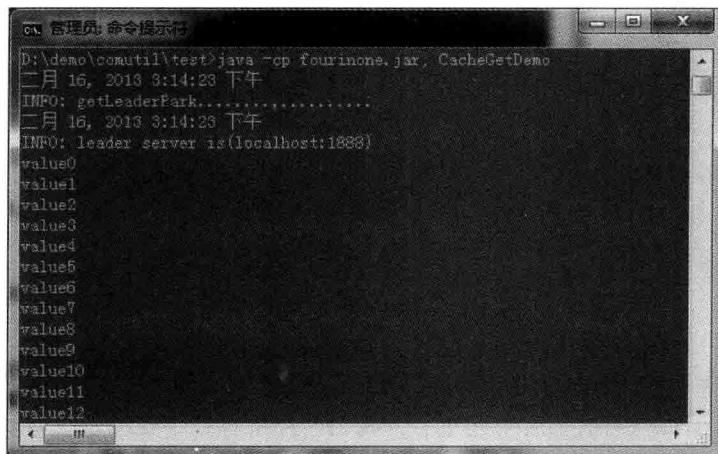


图 4-7 CacheGetDemo

注意

我们在 CacheServer 的代码里可以看到建立了三组主备关系的缓存服务器和端口（这里是在程序里指定了地址和端口，也可以在每个 CacheServer 各自 config 文件的 CacheService 部分配置）。

```
String[][] cacheServerA = new
String[][]{{"localhost","2000"}, {"localhost","2001"}};
String[][] cacheServerB = new
String[][]{{"localhost","2002"}, {"localhost","2003"}};
String[][] cacheServerC = new
String[][]{{"localhost","2004"}, {"localhost","2005"}};
```

除此之外，FacadeServer 还需要了解以上 CacheServer 加入集群的时间信息，我们找到 config.xml 配置里的 CACHEGROUP 部分：

```
<PROPSROW DESC="CACHEGROUP">
<STARTTIME>2010-01-01</STARTTIME>
<GROUP> localhost:2000...@2010-01-01;localhost:2002...;localhost:2004...</GROUP>
</PROPSROW>
<PROPSROW DESC="CACHEGROUP">
<STARTTIME>2018-05-01</STARTTIME>
<GROUP>...</GROUP>
</PROPSROW>
```

我们发现 localhost:2000、localhost:2002、localhost:2004 这三组 CacheServer 的加入集群

时间都是在 2010-01-01 后 2018-05-01 前，它们应该属于第一组。这个分组配置信息是很重要的，会影响它们的 key 寻址路由。在后面的章节会在对照一致性哈希讲日期取模算法原理时，详细讲解日期分组的配置方法。



我们这里为了演示只有一个 FacadeServer，如果有多个 FacadeServer，它们各自的 config.xml 里面的 CACHEGROUP 配置要保持一致。

DEMO 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// CacheFacadeDemo
import com.fourinone.BeanContext;
public class CacheFacadeDemo
{
    public static void main(String[] args)
    {
        BeanContext.startCacheFacade();
        System.out.println("CacheFacade is ok...");
    }
}

// CacheServer
import com.fourinone.BeanContext;
public class CacheServer
{
    public static void main(String[] args)
    {
        String[][] cacheServerA = new String[][]{{"localhost", "2000"}, {"localhost",
            "2001"}};
        String[][] cacheServerB = new String[][]{{"localhost", "2002"},
            {"localhost", "2003"}};
        String[][] cacheServerC = new String[][]{{"localhost", "2004"},
            {"localhost", "2005"}};

        String[][] cacheServer = null;
        if(args[0].equals("A"))
            cacheServer = cacheServerA;
        else if(args[0].equals("B"))
            cacheServer = cacheServerB;
        else if(args[0].equals("C"))
            cacheServer = cacheServerC;
```

```

        BeanContext.startCache(cacheServer[0][0], Integer.parseInt(cacheServer[0]
            [1]), cacheServer);
        // 如果使用配置文件配置地址端口，则使用 BeanContext.startCache(); 启动
    }
}

// CachePutDemo
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.CacheLocal;

public class CachePutDemo
{
    public static void putSmallCache(String[] keyArray)
    {
        ParkLocal pl = BeanContext.getPark();
        pl.create("cache", "keyArray", keyArray);
    }

    public static String[] putBigCache()
    {
        CacheLocal cc = BeanContext.getCache();
        String[] keyArray = new String[100];
        for(int i=0;i<100;i++)
            keyArray[i] = cc.add("key", "value"+i);
        return keyArray;
    }

    public static void main(String[] args){
        String[] keyArray = putBigCache();
        putSmallCache(keyArray);
    }
}

// CacheGetDemo
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.CacheLocal;

public class CacheGetDemo
{
    public static String[] getSmallCache()
    {
        ParkLocal pl = BeanContext.getPark();
        return (String[])pl.get("cache", "keyArray").toObject();
    }

    public static void getBigCache(String[] keyArray)
    {
        CacheLocal cc = BeanContext.getCache();
        for(String k:keyArray)
            System.out.println(cc.get(k, "key"));
    }

    public static void main(String[] args){
        String[] keyArray = getSmallCache();
        getBigCache(keyArray);
    }
}

```

4.3 一致性哈希算法的原理、改进和实现

在淘宝网的很多内部应用中，都涉及分布式的负载均衡，一个简陋的做法就是按机器数量取模，为了保证扩容后不出问题，往往采用成倍扩容的方式去解决，但是这样容易造成机器资源浪费。那有没有更好的方法呢？很多人都会想到一致性哈希算法，包括很多人也问及在 Fourinone 的分布式缓存的负载均衡设计中，为什么不采用一致性哈希算法。因此，我们有必要先看看一致性哈希算法的原理和实现。

一致性哈希算法最初由麻省理工学院在 1997 年提出，用于解决因特网中的热点 (Hot spot) 问题，后来广泛用于分布式应用的负载均衡。

一致性哈希算法提出了一个环的思想。很多人第一次看到出现一个环会觉得的奇怪，我们后面会试图猜测一下算法作者是如何想到这个环的设计。算法思想是：服务器节点通过哈希值 key 将环分成多个区域，存储的数据计算哈希值 key 后，按顺时针方向存储到离它的 key 最近的服务器节点上去。如图 4-8 所示。

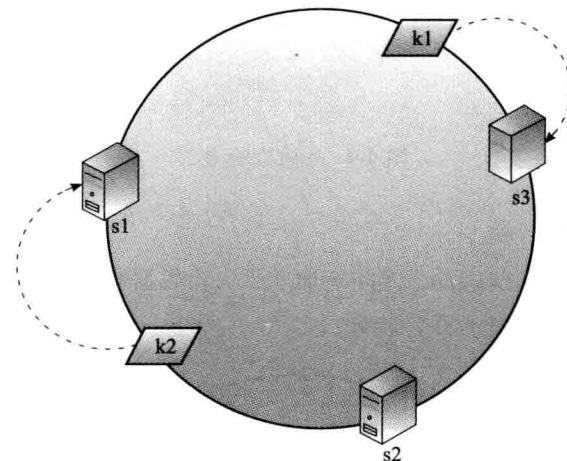


图 4-8 一致性哈希 1

图中 s_1 、 s_2 、 s_3 代表三台数据服务器的 key， k_1 、 k_2 代表两条数据的 key，这里 key 通常是“数据内容”、“服务器编号或者 IP”，通过哈希函数得到。

下面考虑两个问题：

- 1) 如何存放数据？我们通过计算发现 k_1 离顺时针方向的 s_3 最近，那么 k_1 代表的数据存到 s_3 数据服务器上，同理， k_2 数据存放到 s_1 服务器上。
- 2) 如何获取数据？跟存放的方式一样，通过计算 k_1 发现顺时针方向的 s_3 最近，那么到 s_3 上获取 k_1 对应的数据。

我们再看看如果集群发生故障、扩容、分布不均的时候，会受到什么样的影响？

1. 数据服务器发生故障的时候

如图 4-9 所示，假设 s3 服务器故障，那么按照算法 k1 被导向 s2，虽然 s2 并没有 k1 的数据，但是受到影响的只有 k1，k2 并不会受到影响。按照一致性哈希算法的说法，只是局部范围受影响，整体容错性提升了。

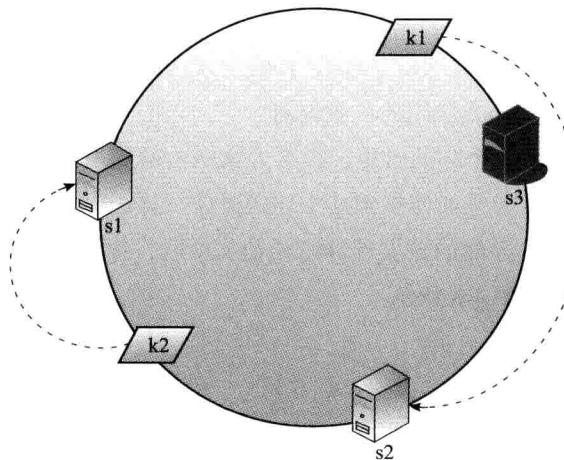


图 4-9 一致性哈希 2

2. 集群服务器扩容的时候

如图 4-10 所示，如果在 s1 和 s2 之间新加入了一台数据服务器 new，那么按照算法，k2 被导向 new，其他的（比如 k1）不受影响。

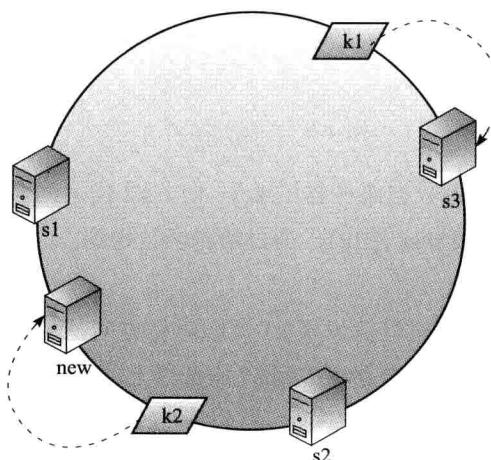


图 4-10 一致性哈希 3

3. 集群服务器分布不均的时候

如果数据服务器 key 都互相接近挤在一起，那么容易出现多个数据 key 都落到一台数据服务器上的情况，如图 4-11 所示，k1、k2、k3 都指向 s2 服务器，导致集群数据分布不均衡。

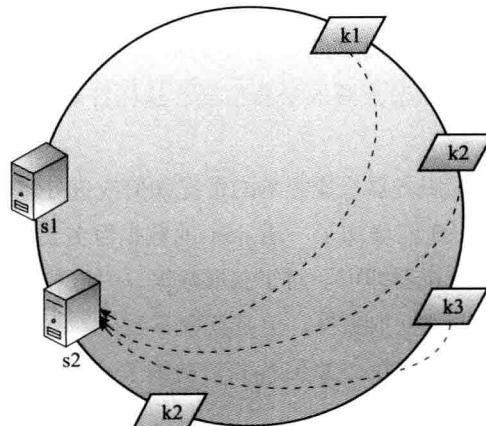


图 4-11 一致性哈希 4

4. 一致性哈希算法的改进

针对上面受影响的情况，一致性哈希算法并没有定义出标准的处理做法，为了改进一致性哈希算法，增加故障的容错能力，可以为每个服务器节点增加备份节点，改进后的一致性哈希算法如图 4-12 所示

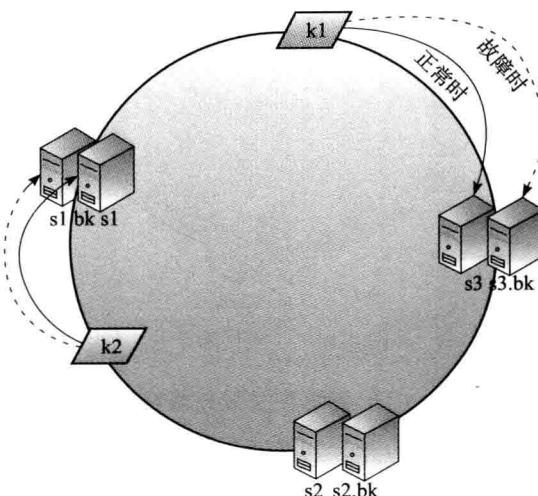


图 4-12 一致性哈希 5

数据服务器主备节点保持数据同步，故障时，算法切换到备份节点上获取数据。这样 s1、s2、s3 只作为数据存储节点，而不同时作为相邻节点的备份节点，这样职责更分明，容易管理和维护，减少数据迁移复杂度，只不过这样会牺牲更多的机器做备份。

我们再看看当 s3 故障时如何处理。我们先按照一致性哈希算法将 k1 导向 s3，发现 s3 故障既不能写也不能读，这个时候算法改进，会找到 s3 的备份节点进行写或者读。如果 s3 恢复，根据数据同步，它会从备份节点获取最新信息，这样针对数据服务器故障的情况便不会再受影响。

那么对于增加了新的数据服务器造成影响的情况如何改进处理呢？

如图 4-13 所示，如果在 s3 前增加了一组 new 的数据服务器主备节点，对于新增数据不存在问题，直接写入新的节点机器即可。对于获取数据，采取如下步骤：

1) k1 会首先导向 new 节点获取数据，但是获取不到；

2) 继续按顺时针方向向下寻找，如果有多个新增节点继续向下，直到在 s3 找到数据，获取到数据后从 s3 删除；

3) 将 k1 数据写入到自己的顺时针方向第一个节点 new 上。

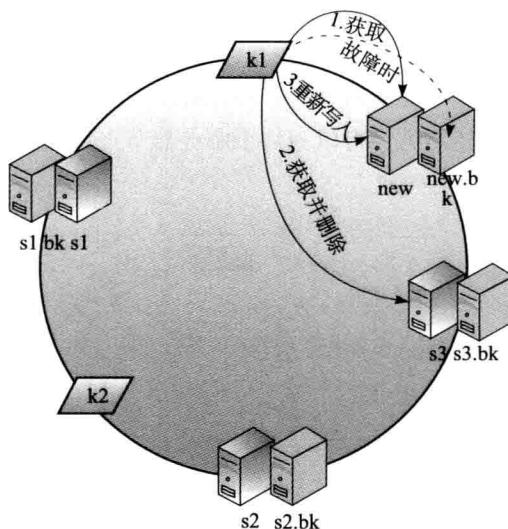


图 4-13 一致性哈希 6

通过以上步骤我们可以看到，由于新增数据服务器对集群位置的改变，k1 在新增节点上寻找不到数据，会顺时针往下轮询直到找到之前存放的服务器。并且在找到后将该数据从之前存放的服务器删除，而改写入到自己顺时针最近的服务器上，这样做的目的是为了避免下次获取 k1 数据时再次轮询多台服务器导致的延时。当然如果能承受延时，也可以不调整 k1 数据位置，每次采取顺时针逐个轮询获取。

另外，对于环上集群服务器分配不均情况的改进做法有以下几个：

- 计算服务器的 key 时，尽可能让 key 值平衡分布，不要靠得太近；
- 采用虚拟化手段将一台服务器物理机隔离成多个虚拟机，在形式上增加出多个服务器节点维持分布平衡。

图 4-14 所示，s1-1、s1-2、s1-3 都是 s1 的虚拟节点，s2-1、s2-2、s2-3 都是 s2 的虚拟节点，在增加了数据服务器虚拟节点后，达到一个相对平衡分布的效果。

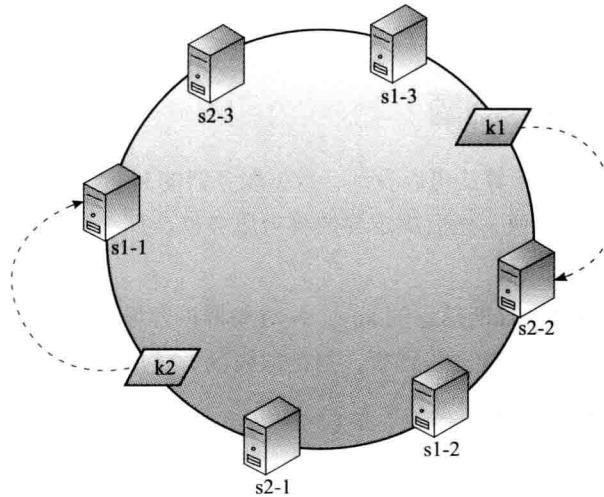


图 4-14 一致性哈希 7

接下来简单描述一下一致性哈希算法的实现。上面以环图方式表示算法原理，但是如果涉及实现，我们可以换一种方式理解该算法。我们用一个数组表示服务器节点的 key 值，如下所示：

```
s[] = {10, 20, 30}
```

代表 3 个服务器 key 节点分别是 $s1=10$, $s2=20$, $s3=30$ 。这时，我们有个数据 $key=15$ ，跟上面这个数组比对，发现在 $s1$ 和 $s2$ 之间，根据算法，它应该顺时针导向到 $s2$ 上。

假设集群扩容，新增加了一个 $s=17 \sim s=18$ 的服务器：

```
s[] = {10, 17, 18, 20, 30}
```

那么，数据 $key=15$ 会从 $s1$ 后向下找， $s2=17$, $s3=18$, 直到 $s4=20$ ，如果还未找到，会按顺时针方向一直向下，直到回到 $s1$ 形成一个循环。

于是我们不难理解为什么画出来是一个圆环的设计了，因为实现上多半是从头到尾又返回，轮询这个服务器 key 数组，轮询的方向是从左往右，是顺时针方向。

算法都是人创造的，都是灵活的，千万不要教条化，为了实现算法本身而实现。在某种算法上花费了学习成本，以后就抱着这种算法的步骤僵化套用是不好的。应该多观察我们需要解决的问题，然后思考合适的算法，如果思考过程中碰到问题，再回头参考前人已经总结出来的算法寻求借鉴，这样的才能有更深刻的体会。

在 Fourinone 的分布式缓存负载均衡实现中，我们并没有采用一致性哈希算法，而采用了一种基于日期 key 取模的方式。基于日期分组策略实现扩容后数据分布均匀，也能达到目的和解决问题。

4.4 解决任意扩容的问题

我们从上面的一致性哈希算法可以看到，数据服务器的 key 实际上定义了一个区间界值，然后数据 key 去比对这个界值，然后决定导向哪台服务器。那么定义这个区间界值的不一定是哈希值，也可以是其他信息。

Fourinone 通过生成含有日期信息的 key，并对集群扩容增加日期配置，通过 key 和集群配置的日期匹配计算出覆盖范围的机器数，再用取模的方式准确得到负载的计算机，对于集群的任意数量的扩容都不会受到影响。

原理如图 4-15 所示。

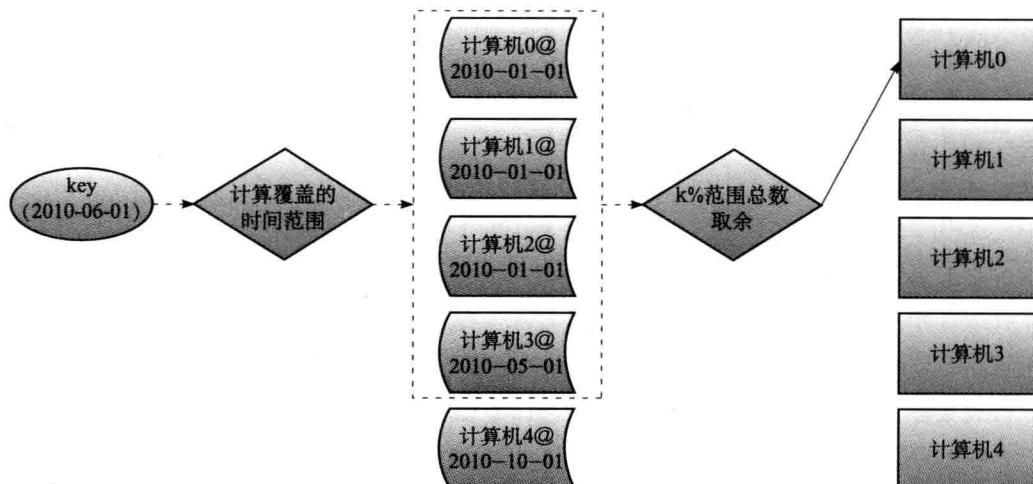


图 4-15 日期 key 取模

算法过程如下：

- 1) 首先生成的 key 是包含日期信息的，根据该 key 的时间点去计算覆盖集群计算机的范围。
- 2) 集群的计算机有个统一的配置表，它包括每台计算机的 IP、端口等信息，除此外，还

包括每台计算机加入集群的日期信息。当一个包括日期信息的 key 路由时，会计算集群中比 key 包含日期小的范围，然后得到这个范围的计算机数量。对于一个新生成的 key，它的时间是当前时间，也就是会覆盖集群中所有的计算机，因为计算机加入集群的时间是一个过去时。

在 Fourinone 的实现中，通过 config.xml 中的 cache 部分配置可以看到：

```
<GROUP>
localhost:2000,localhost:2001@2010-01-01;localhost:2002,localhost:2003@2010-05-01;
localhost:2004,localhost:2005@2010-05-01
</GROUP>
```

这里定义了 3 组 CacheServer，用 “;” 间隔分开：

```
localhost:2000,localhost:2001@2010-01-01;
localhost:2002,localhost:2003@2010-05-01;
localhost:2004,localhost:2005@2010-05-01
```

上面的 “localhost:2000,localhost:2001” 被 “@” 间隔开，代表一组主备两台的 CacheServer，“@” 后的 2010-01-01 代表这组 CacheServer 加入集群的时间，key 路由时会将自己生成的时间跟 CacheServer 加入集群的时间做对比，找到属于这个时间范围的所有 CacheServer。

然后 key 再对这个范围的计算机数量取余，就得到集群中对应序号的计算机，也就是路由目标。

4.5 解决扩容后数据均匀的问题

我们从上面基于日期 key 的取模可以解决扩容前后的读写路由问题，但是当集群一下子扩容了多台机器后，我们希望能保证数据均匀分布，让新的数据更地散落到新扩容的机器上存储，减少已有集群机器的存储压力，在 Fourinone 中是采用日期分组策略去完成的。

我们可以在 config.xml 配置文件中看到以下关于缓存分组的内容：

```
<PROPSROW DESC="CACHEGROUP">
<STARTTIME>2010-01-01</STARTTIME>
<GROUP>localhost:2000,localhost:2001@2010-01-01;localhost:2002,localhost:2003@2010-05-01;localhost:2004,localhost:2005@2010-05-01</GROUP>
</PROPSROW>
<PROPSROW DESC="CACHEGROUP">
<STARTTIME>2018-05-01</STARTTIME>
<GROUP>localhost:2008,localhost:2009@2018-05-01;localhost:2010,localhost:2011@2018-05-01</GROUP>
</PROPSROW>
.....
```

上面分成了两个组，“<STARTTIME>2010-01-01</STARTTIME>”代表这组 CacheServer 是 2010-01-01 后加入集群的，截至到 2018-05-01；“<STARTTIME>2018-05-01</STARTTIME>”代表另一组 CacheServer 是 2018-05-01 后加入集群的，这里时间发生在未来只是为了方便举例)，我们可以根据集群实际扩容时间进行分组记录。



提示

计算的思路是，在判断 key 时，会先找到 key 所在的分组，然后在分组内找到取模所在的服务器。我们再对比一下和一致性哈希算法的区别，一致性哈希算法实际上通过服务器的 key 划分出一个个区间，然后匹配数据 key 属于哪个区间；而这里通过记录了集群扩容的时间分组信息，再结合 key 的时间信息做匹配，看归属哪个时间组里的哪台服务器。

日期时间信息冲突怎么办？曾经有使用者问过，取时间信息可能会出现相同的情况。如果数据 key 的时间和服务器扩容的时间出现冲突怎么办？按日期时间取模方式会不会出问题？

实际上在集群负载的场景下有个特点，就是服务器的扩容时间是粗粒度的，它不需要也不会精确到秒，因为服务器不会每秒都扩容，一般一年、半年才扩容一次；而数据 key 的时间是个细粒度的，因此很准确就能匹配到属于哪个粗粒度的时间范围，而不会发生冲突。

4.6 分布式 Session 的架构设计和实现

如果建立一个网站，经常用到 Session，Web 中的 Session 指的就是用户在浏览某个网站时，从进入网站到浏览器关闭所经过的这段时间，也就是用户浏览这个网站所花费的时间。一个 Session 的概念需要包括特定的客户端、特定的服务器端以及不中断的操作时间。A 用户和 C 服务器建立连接时所处的 Session 同 B 用户和 C 服务器建立连接时所处的 Session 是两个不同的 Session。Session 的概念还包括在会话的这段时间内，用户可以往服务器缓存写入和读取信息，该信息只在 Session 有效时间内存在，失效就被删除。大型网站应用中为了减少数据库负载，提升访问速度，通常将属于某次用户会话的频繁操作的数据存放在 Session 中。因此，当网站服务器是跨多台服务器的分布式环境时，则相应需要分布式 Session 结构去解决。

目前市场上的 Tomcat 等 J2EE Web 服务器实现了分布式 Session 机制，但是 Tomcat 等 Web 服务器 Session 只是在单台服务器环境上有效，不能满足集群多服务器的分布式环境，需要借助分布式缓存系统实现，但是较多存在故障容错能力弱、易丢失内存数据、无法动态切换备份机等问题，并且集群部署上复杂，需要成倍扩容，缓存容易不命中。

我们接下来构思一下分布式 Session 的架构设计，能够基于该设计实现大型分布式 Session 系统，并可以自由扩容缓存容量，用来解决大型互联网等应用面临的分布式跨多计算

机的 Session 问题。



我们分析分布式 Session 系统的特点后发现，Session 系统其实就是一个基于分布式缓存系统应用，可以把缓存 key 当作 Cookie ID 的实现，它只不过是保存在浏览器客户端，每次浏览器访问网站服务器时会携带这个 Cookie ID 信息。

回顾一下我们前面讲述的分布式缓存技术，采用一组相同的 FacadeServer+不同的 CacheServer 的分布式设计模型，可以具备良好的负载均衡和可扩充性，并且有很好的故障容错能力。它对外通过 FacadeServer 分解负载，并通过生成含有日期信息的 key，对集群扩容增加日期配置，通过 key 和集群配置的日期匹配计算出覆盖范围的机器数，再以取模的方式准确得到负载的计算机，这样能准确将数据分散存放在不同的 CacheServer 上，它的规模和缓存能力可以随计算机的扩充而扩充。

我们基于上面的分布式缓存设计稍做调整后得到下面的分布式 Session 系统架构，如图 4-16 所示。

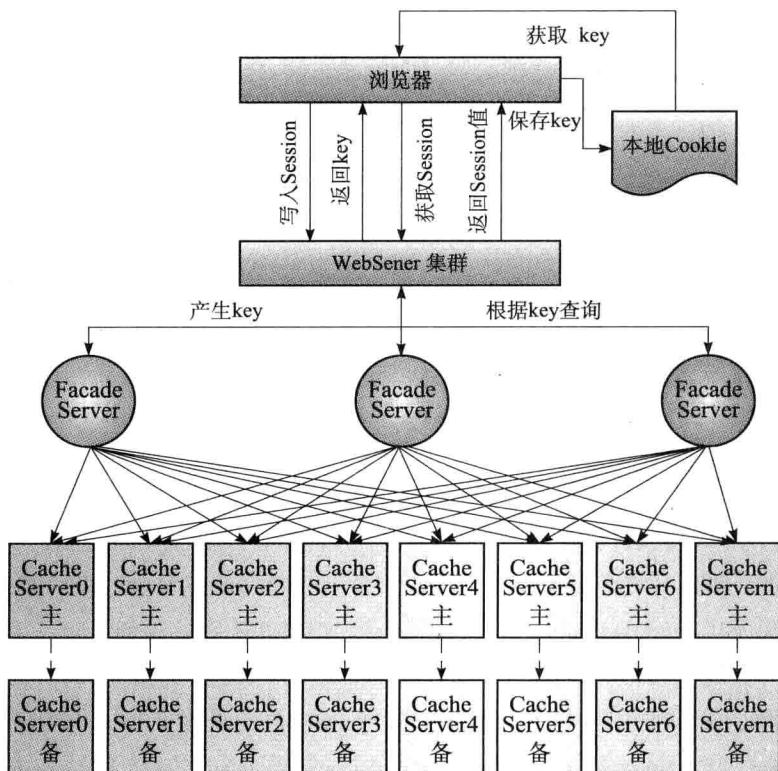


图 4-16 分布式 Session 系统架构

架构的详细实现过程如下：

- 1) 首先，浏览器提交 Session 的写入请求，比如用户登录，写入用户信息。
- 2) 当外部访问请求非常大时，首先通过 WebServer 集群进行请求疏导，物理上负载均衡器是部署在 WebServer 集群中，可以有一个到多个。WebServer 集群将写入请求负载到后端的 FacadeServer 上。
- 3) FacadeServer 和 CacheServer 的结构即前面讲述的分布式缓存结构，它由 Session 系统的底层 Cache 支撑。
- 4) 如果是第一次写入，请求中没有携带 key 信息，FacadeServer 会自动依据生成 key 并返回给浏览器，浏览器将 key 信息保存在本地 Cookie 中。
- 5) 当浏览器在下次发出 Session 的读请求时，会从本地 Cookie 将 key 信息携带并进行请求提交，FacadeServer 会根据 key 信息直接判断出属于哪台 CacheServer，并将读取的 Session 结果返回给客户端。
- 6) Session 系统的负载和扩容实现。当浏览器将 Cookie 里的 key 信息携带发出请求时，FacadeServer 并不需要根据 key 去一个配置服务器查询关联到哪台存储服务器，因为当请求量很大时，配置服务器的查询会成为瓶颈，并且传统的 key 取模方式有很大的缺陷，比如对于集群数量为 n，那么数字 ID 的 key，按照 $key \% n$ 得到路由的目标计算机，当集群数量扩充时，取模变得不准确，如果要维持准确，通常需成倍去扩容，这会造成成本增加和浪费。这里采用前面讲述的日期取模算法，将含有日期信息的 key 和集群配置的日期匹配计算出覆盖范围的机器数，再以取模的方式准确得到负载的计算机，对于集群的任意数量的扩容都不会受到影响。

4.7 缓存容量的相关属性设置

通过上面的缓存实例，我们了解了如何读写缓存，那么如果不不停地往缓存里写入数据，会发生什么情况呢？在 config.xml 的 Park 部分有个 SAFEMEMORYPER 配置项：

```
<SAFEMEMORYPER>0.95</SAFEMEMORYPER>
```

默认配置为 0.95，代表内存占用安全比率在 95% 左右，如果超出该比率，将无法正确写入，会返回 null，并且系统会提示写入错误，但是不会抛出系统异常。这个安全占用比率的设置是为了避免发生 OutOfMemory 造成系统崩溃。但是我们要认识到，由于 JVM 的自动回收和释放内存等其他因素影响，这个比率不一定能非常精确地反映真实内存占用比例，它只反映了某个时间点的大致占用状况。我们可以写个 DEMO 看看效果。

SetValue 是一个 main 函数类，它获取到 ParkLocal 后，不停地写入 domain 和 node 以及对应的 value 值，为了方便演示，domain/node/value 都为序号数字，如果写入成功，会返回

结果对象，否则返回 null。我们将其输出显示出来。

另外，为了尽快看到效果，我们将 SAFEMEMORYPER 调整得小一些：

```
<SAFEMEMORYPER>0.40</SAFEMEMORYPER>
```

运行步骤如下：

1) 启动 ParkServerDemo (它的 IP 端口已经在配置文件指定)，结果如图 4-17 所示：

```
java -cp fourinone.jar; ParkServerDemo
```

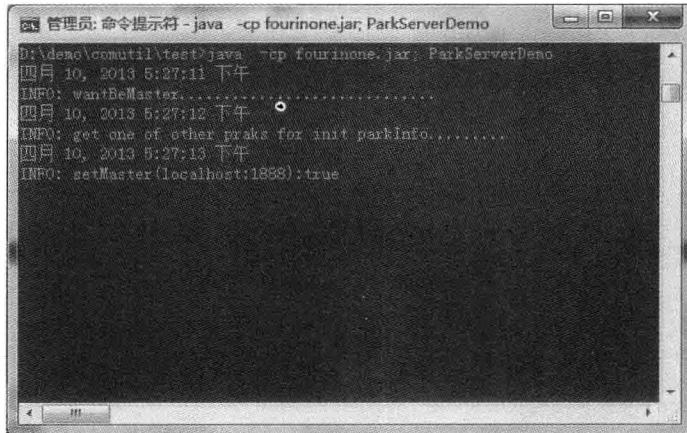


图 4-17 ParkServerDemo

2) 启动 SetValue，结果如图 4-18 所示。

```
java -cp fourinone.jar; SetValue
```

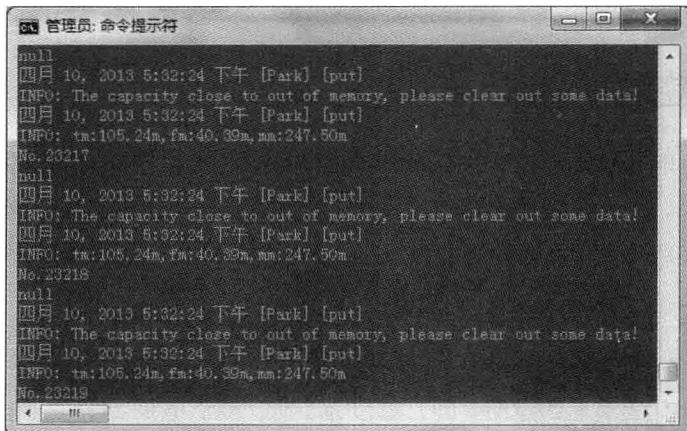


图 4-18 SetValue

我们运行了一会儿后可以看到，界面输出“`The capacity close to out of memory, please clear out some data!`”这意味着内存已经接近安全比率，不再允许写入数据，请及时清空缓存的数据，减少占用。这个时候写入数据不成功，返回 `null` 值。

DEMO 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// SetValue
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;

public class SetValue
{
    public static void main(String[] args)
    {
        ParkLocal pl = BeanContext.getPark();
        int i=0;
        while(true){
            ObjectBean ob = pl.create(i++, i++, i++);
            System.out.println("No." + i);
            System.out.println(ob);
            i=i+1;
        }
    }
}
```

4.8 缓存清空的相关属性设置

对于我们写入的 domain、node 和 value 的键值数据，可以匹配一个过期时间，过期后框架会自动进行清空，并且清空也可以设置一个周期，每隔一段时间清空一次过期的数据。在 config.xml 的 Park 部分有个 EXPIRATION 和 CLEARPERIOD 配置项用来完成该功能：

```
<EXPIRATION>24</EXPIRATION>
<CLEARPERIOD>0</CLEARPERIOD>
```

`EXPIRATION` 代表节点的过期时间，单位为小时，默认值是 24；

`CLEARPERIOD` 代表清空的周期时间，单位为小时，默认为 0（表示不清空）。

如果需要定期清空过期节点数据，可以根据需要设置上面两个配置项，比如为了方便我们接下来演示效果，修改为：

```
<EXPIRATION>0.02</EXPIRATION>
<CLEARPERIOD>0.01</CLEARPERIOD>
```

EXPIRATION : $0.02 \times 60 \times 60 = 72$ 秒，表示写入的节点数据在 72 秒后过期。

CLEARPERIOD: $0.01 \times 60 \times 60 = 36$ 秒，表示每隔 36 秒清空一次过期数据。

ClearTest 是一个 main 函数类，他获取到 ParkLocal 后，先创建一个

```
{domain="china", node="city1", value="beijing"}
```

的节点，并输出创建时间；然后等待 30 秒后，再创建 2 个节点：

```
{domain="china", node="city2", value="shanghai"}
{domain="china", node="city3", value="shenzhen", heartbeat="true"}
```



注意

这里创建的三个节点都在相同的 domain="china" 下，并且“shenzhen”这个节点是一个“心跳”节点，其他 2 个是普通节点。对于“心跳”节点，我们知道它是跟 ParkServer 一直保持联系，除非机器故障或者网络中断导致 ParkServer 删除该节点，否则它不存在随时间过期的概念，因此设置过期自动清空对于“心跳”节点来说是无效的。

在三个节点创建成功后，ClearTest 用一个 while 不断的轮询，获取 domain="china" 下面所有的节点，并且输出显示，然后停留 10 秒后又继续。我们运行该 demo 查看效果：

运行步骤：

1) 启动 ParkServerDemo (它的 IP 端口已经在配置文件指定)，结果如图 4-19 所示：

```
java -cp fourinone.jar; ParkServerDemo
```

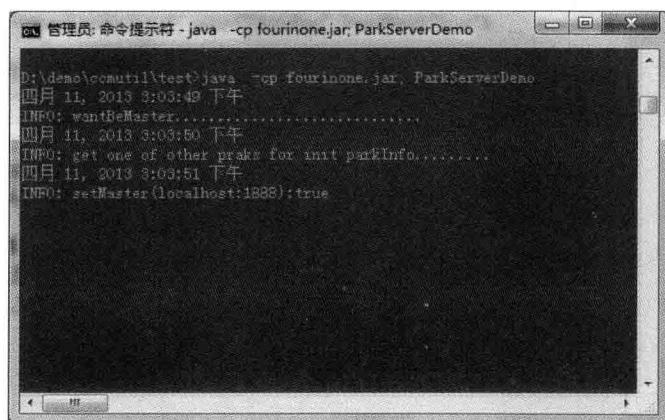


图 4-19 ParkServerDemo

2) 启动 ClearTest:

```
java -cp fourinone.jar; ClearTest
```

```
java -cp fourinone.jar; ClearTest
U:\demo\comotil\test>java -cp fourinone.jar; ClearTest
四月 11, 2013 3:09:21 下午
INFO: setLeaderPark.....
四月 11, 2013 3:09:21 下午
INFO: leader server islocalhost:1888
Thu Apr 11 15:09:21 CST 2013:
china.city1:beijing
Thu Apr 11 15:09:51 CST 2013:
china.city2:shanghai
Thu Apr 11 15:09:51 CST 2013:
china.city3:shenzhen
Thu Apr 11 15:09:51 CST 2013:
[china.city1:beijing, china.city2:shanghai, china.city3:shenzhen]
Thu Apr 11 15:10:01 CST 2013:
[china.city1:beijing, china.city2:shanghai, china.city3:shenzhen]
Thu Apr 11 15:10:11 CST 2013:
[china.city1:beijing, china.city2:shanghai, china.city3:shenzhen]
Thu Apr 11 15:10:21 CST 2013:
[china.city1:beijing, china.city2:shanghai, china.city3:shenzhen]
Thu Apr 11 15:10:31 CST 2013:
[china.city1:beijing, china.city2:shanghai, china.city3:shenzhen]
Thu Apr 11 15:10:41 CST 2013:
[china.city2:shanghai, china.city3:shenzhen]
Thu Apr 11 15:10:51 CST 2013:
[china.city2:shanghai, china.city3:shenzhen]
Thu Apr 11 15:11:01 CST 2013:
[china.city2:shanghai, china.city3:shenzhen]
Thu Apr 11 15:11:11 CST 2013:
[china.city3:shenzhen]
Thu Apr 11 15:11:21 CST 2013:
[china.city3:shenzhen]
Thu Apr 11 15:11:31 CST 2013:
[china.city3:shenzhen]
Thu Apr 11 15:11:41 CST 2013:
[china.city3:shenzhen]
```

图 4-20 ClearTest

出现图 4-20，我们可以看到，“Beijing” 节点创建成功后，等待了 30 秒连续创建了“shanghai” 和 “shenzhen” 节点，然后随着图里每隔 10 秒的时间输出当前 domain 里的节点状况。

开始有三个节点，“Beijing” 节点的创建时间为 “15: 09: 21”，大约在 “15: 10: 41” 时，“Beijing” 节点被自动清空掉了，因为我们设置的过期时间是 72 秒，“15: 10: 41” - “15: 09: 21” = 80 秒，已经超过了 72 秒。这里为什么不刚好等于 72 秒呢？这是因为节点虽然过期了，但是要等待每次清空周期执行的原因。

同样，再过 30 秒后，发现 “shanghai” 节点也过期并清空掉，只剩下 “shenzhen” 节点一直存在，这是因为 “shenzhen” 节点是一个 “心跳” 节点，它没有时间过期的概念而不会被框架自动清空。

读者可以自行调整 EXPIRATION 和 CLEARPERIOD 配置，并认真观察随着时间变化节

点被清空的状况，细细体会其中的功能机制。

DEMO 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// ClearTest
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import java.util.Date;
import java.util.List;

public class ClearTest
{
    public static void main(String[] args)
    {
        try{
            ParkLocal pl = BeanContext.getPark();

            ObjectBean ob1 = pl.create("china", "city1", "beijing");
            System.out.println(new Date() + ":");
            System.out.println(ob1);

            Thread.sleep(1000*30);

            ObjectBean ob2 = pl.create("china", "city2", "shanghai");
            System.out.println(new Date() + ":");
            System.out.println(ob2);

            ObjectBean ob3 = pl.create("china", "city3", "shenzhen", true);
            System.out.println(new Date() + ":");
            System.out.println(ob3);

            while(true){
                List<ObjectBean> oblist = pl.get("china");
                System.out.println(new Date() + ":");
                System.out.println(oblist);
                Thread.sleep(1000*10);
            }
        }catch(Exception e){
            System.out.println(e.toString());
        }
    }
}
```

第 5 章

消息队列的实现

本章讲述了中间件和消息队列（MQ）的发展史 JMS 规范定义的发送 / 接收、主题订阅两种经典消息服务模式的机制原理，并详细举例说明如何在 Fourinone 中实现这两种经典模式。

5.1 闲话中间件与 MQ

谈起 MQ，闲话几句中间件历史，大约 2000 年左右时，当时比较主流的分布式技术是 Corba 和微软阵营的 Com/Com+，当时在一个小企业实习接受过 Corba 技术培训，那会儿拿本厚厚的 Corba 技术书，虽然也不怎么懂“公共对象请求代理体系结构”这个抽象高深的名字，但看见别人听着后更加一脸茫然时，顿时感到一种自豪感，掌握了 Corba 先进技术就有一个更好的未来。当时费了很大劲学会了安装和开发 Helloworld，学会了配置一个最简单的 IDL，其实并不知道这个东西能用在什么地方，为什么要用，只是隐约听见公司投标的邮政系统好像会用。

后来 Corba 技术最终没有发展起来，它想提出一个公共的标准，但是同时期 J2EE 规范已经形成，整个 Java 阵营都在支持 EJB1.0 和 JMS，微软的 DCOM 也发展成完整的 .NET 技术体系，Corba 夹在这两大技术阵营中间，声音越来越小，投入越来越弱…所有大的商业软件公司中间件产品对 Corba 的支持都不怎么给力，最终随着 WebService 和标准化的 WSDL 逐渐普及，Corba 的声音完全被淹没了，作者通过 WebService 也终于明白了为什么要用 Corba。

MQ 活了过来，并且一直发展得很好，MQ 最初在银行前置终端机就有应用，这种屏蔽底层通信细节、异步松耦合并在异构应用之间的消息驱动方式逐渐受到欢迎，广泛应用到中国几乎每家银行业务系统中，并且 Java 阵营制定了 JMS 标准来规范化 MQ 产品。

“我发一个消息你来收，或者你订阅我的消息，我来通知你”的设计出现到越来越多的解决方案中，MQ 和 Webservice 成了后来构建 SOA 的重要基础技术，IBM 基于 MQ 推出了 IBM MB 的 ESB 产品，产品 License 售价超过百万，而且还是按 CPU 卖。

成王败寇，用在 IT 技术行业又何尝不恰当，一代代的创新技术就像起义者，推翻了旧技术后，又彼此竞争淘汰，Corba 倒下了，最终 MQ 和 Webservice 踏着失败者的尸体占据了市场，因此学习分布式技术，我们不能不去了解 MQ 的机制和原理。

Fourinone 可以当成简单的 MQ 来使用，但更多的是旨在揭露实现原理，告诉你如何设计 MQ，下面讲述了 MQ “队列和主题订阅两种核心模式”的实现并演示了相关 Demo。

5.2 JMS 的两种经典模式

在讲述实现原理之前，我们先了解一下官方定义的 Java 消息服务的两种经典模式。

- 点对点或队列模式？在点对点或队列模式下，一个生产者向一个特定的队列发布消息，一个消费者从该队列中读取消息。这里，生产者知道消费者的队列，并直接将消息发送到消费者的队列。这种模式概括为：只有一个消费者将获得消息。生产者不需要在接收者消费该消息期间处于运行状态，接收者也同样不需要在消息发送时处于运行状态。每一个成功处理的消息都由接收者签收。
- 发布者 / 订阅者模式？发布者 / 订阅者模型支持向一个特定的消息主题发布消息。0 或多个订阅者可能对接收来自特定消息主题的消息感兴趣。在这种模式下，发布者和订阅者彼此不知道对方。这种模式好比是匿名公告板。这种模式概括：多个消费者可以获得消息。在发布者和订阅者之间存在时间依赖性。发布者需要建立一个订阅（subscription），以便客户能够订阅。订阅者必须保持持续的活动状态以接收消息，除非订阅者建立了持久的订阅。在那种情况下，在订阅者未连接时发布的消息将在订阅者重新连接时重新发布。

在 JMS 规范中，JMS 提供了将应用与提供数据的传输层相分离的方式。同一组 Java 类可以通过 JNDI 中关于提供者的信息，连接不同的 JMS 提供者。这一组类首先使用一个连接工厂以连接到队列或主题，然后发送或发布消息。在接收端，客户接收或订阅这些消息。其中点对点消息往往与队列（javax.jms.Queue）相关联。而发布 / 订阅消息往往通过事件驱动模型实现，消息生产者和消费者都参与消息的传递，生产者发布事件，而使用者订阅感兴趣的事件，并使用事件。该类型消息一般与特定的主题（javax.jms.Topic）关联。JMS 规范定义的主要对象角色有以下几个：

- 1) 连接工厂（ConnectionFactory）是由管理员创建，并绑定到 JNDI 树中。客户端使用

JNDI 查找连接工厂，然后利用连接工厂创建一个 JMS 连接。

2) JMS 连接 (Connection) 表示 JMS 客户端和服务器端之间的一个活动的连接，是由客户端通过调用连接工厂的方法建立的。

3) JMS 会话 (Session) 表示 JMS 客户与 JMS 服务器之间的会话状态。JMS 会话建立在 JMS 连接上，表示客户与服务器之间的一个会话线程。

4) JMS 目的 (Destination)，又称为消息队列，是实际的消息源。

5) JMS 生产者和消费者。生产者 (Message Producer) 和消费者 (Message Consumer) 对象由 Session 对象创建，用于发送和接收消息。

5.3 如何实现发送接收的队列模式

我们可以将 Domain 视为 MQ 队列，每个 node 为一个队列消息，检查 Domain 的变化来获取队列消息。

□ Sender：是一个队列发送者，它发送消息的实现是在 queue 上创建一个匿名节点来存放消息

```
pl.create(queue, (Serializable) obj);
```

□ Receiver：是一个队列接收者，他轮循 queue 上有没有最新消息，有就取出，并删除该节点，注意它是每次获取第一个消息，这样保证消息读取的顺序。如图 5-1 所示。

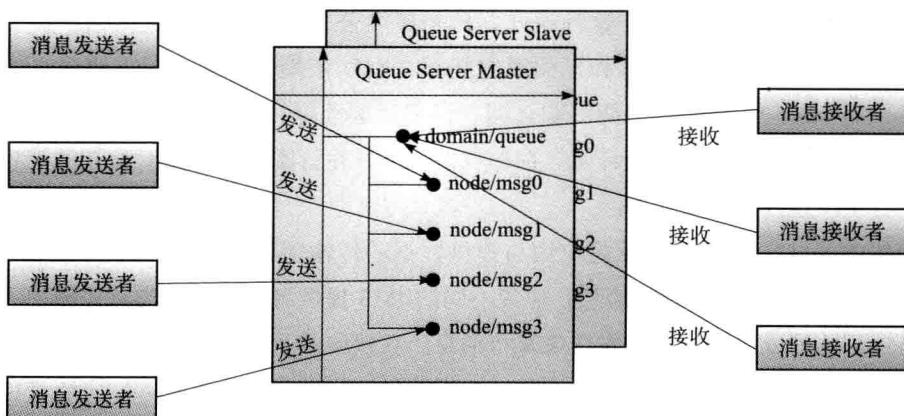


图 5-1 发送接收模式实现

运行步骤：

1) 启动 ParkServerDemo (它的 IP 端口已经在配置文件指定)，结果如图 5-2 所示：

```
java -cp fourinone.jar; ParkServerDemo
```

```
D:\>cd demo
系统找不到指定的路径。
D:\>cd demo
D:\demo>cd com*
D:\demo\comutil>cd test

D:\demo\comutil\test>java -cp fourinone.jar; ParkServerDemo
二月 16, 2013 2:53:01 下午
INFO: wantBeMaster.....
二月 16, 2013 2:53:02 下午
INFO: get one of other parks for init parkInfo.....
二月 16, 2013 2:53:03 下午
INFO: setMaster(localhost:1888):true
```

图 5-2 ParkServerDemo

2) 运行 Sender，结果如图 5-3 所示：

```
java -cp fourinone.jar; Sender
```

```
D:\>cd demo\comutil\test>
D:\>cd demo\comutil\test>

D:\demo\comutil\test>java -cp fourinone.jar; Sender
二月 16, 2013 7:50:34 下午
INFO: getLeaderPark.....
二月 16, 2013 7:50:35 下午
INFO: leader server is(localhost:1888)
```

图 5-3 Sender

在 Sender 的程序里，往“queue1”队列里发了“hello”、“world”、“mq”三个消息。

3) 运行 Receiver，结果如图 5-4 所示：

```
java -cp fourinone.jar; Receiver
```

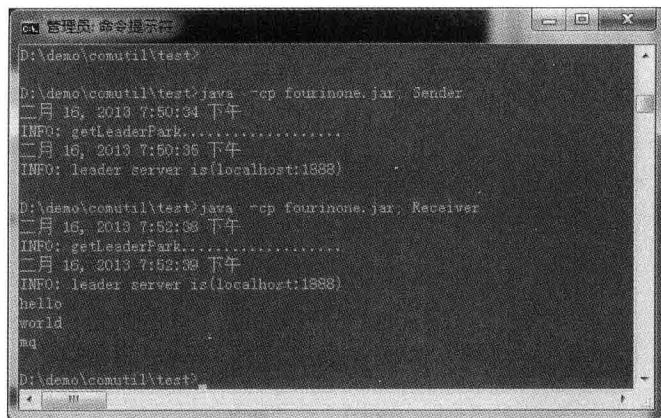


图 5-4 Receiver

可以看到，按照发送的先后顺序，依次收到“hello”、“world”、“mq”三个消息。我们打开 Receiver 程序可以看到：

```
while(true)
{
    oblist = pl.get(queue);
    if(oblist!=null)
    {
        ObjectBean ob = oblist.get(0);
        ...
        pl.delete...
        break;
    }
}
```

Receiver 实际上轮循检查队列是否有消息，有的话每次取出所有队列的消息 list，然后再取出第一个消息并删除，或许这样不是最高效，因为如果消息队列的消息太多，一次性取出队列所有消息会影响性能，但是这里旨在演示清楚 MQ 接收消息的实现，开发者明白了实现机制可以发挥自己的创造性，根据自己需求的消息特点针对性的设计 MQ 发送接收的实现，比如模仿商业 MQ 对每个消息队列的容量做一个限制等等。

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}
```

```
// Sender
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import java.io.Serializable;

public class Sender
{
    private static ParkLocal pl = BeanContext.getPark();

    public static void send(String queue, Object obj)
    {
        pl.create(queue, (Serializable)obj);
    }

    public static void main(String[] args)
    {
        send("queue1","hello");
        send("queue1","world");
        send("queue1","mq");
    }
}

// Receiver
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import java.util.List;

public class Receiver
{
    private static ParkLocal pl = BeanContext.getPark();

    public static Object receive(String queue)
    {
        Object obj=null;
        List<ObjectBean> oblist = null;
        while(true)
        {
            oblist = pl.get(queue);
            if(oblist!=null)
            {
                ObjectBean ob = oblist.get(0);
                obj = ob.toObject();
                pl.delete(ob.getDomain(), ob.getNode());
                break;
            }
        }
        return obj;
    }

    public static void main(String[] args)
    {
        System.out.println(receive("queue1"));
        System.out.println(receive("queue1"));
    }
}
```

```

        System.out.println(receive("queue1"));
    }
}
}

```

5.4 如何实现主题订阅模式

我们可以将 Domain 视为订阅主题，将每个订阅者注册到 Domain 的节点（Node）上，发布者将消息逐一更新每个节点，订阅者监控每个属于自己的节点的变化事件获取订阅消息，收到后清空内容等待下一个消息，多个消息用一个 arraylist 存放。

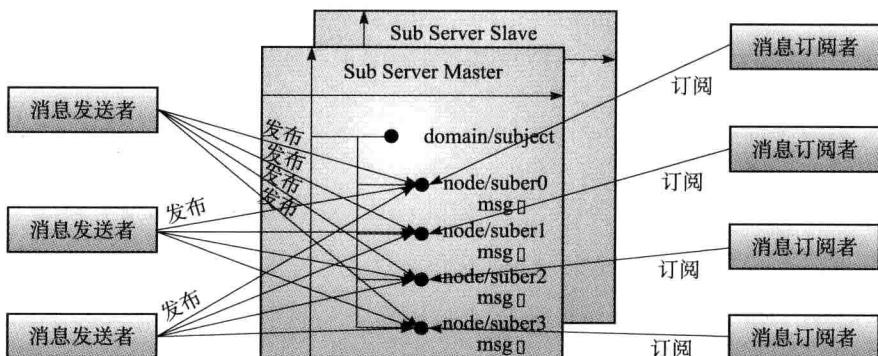


图 5-5 主题订阅模式实现

- Publisher：是一个主题发布者，它通过 `pl.get(topic)` 获取 topic（主题）的所有订阅者节点，并将消息更新到每个节点上，如果有多个追加到 arraylist 存放。
- Subscriber：是一个消息订阅者，他通过 `subscrib(String topic, String subscribeName, LastestListener lister)` 实现消息订阅，其中 3 个参数分别是主题名、订阅者名称、事件处理实现。Subscriber 实现了 LastestListener 事件处理接口 `happenLastest(LastestEvent le)`，这个接口会传入更新的节点内容对象，然后 Subscriber 用一个空的 arraylist 清空内容，等待下一次接收消息。`happenLastest` 有个 boolean 返回值，如果返回 false，它会一直监控变化，继续有新的变化时还会进行事件调用；如果返回 true，它完成本次事件调用后就终止。

运行步骤：

- 1) 启动 ParkServerDemo（它的 IP 端口已经在配置文件指定）：

```
java -cp fourinone.jar; ParkServerDemo
```

- 2) 运行 Subscriber，因为 Subscriber 可以有多个，传入不同的 `subscribeName` 参数代表不同的 Subscriber，如图 5-6 所示。

```
java -cp fourinone.jar; Subscriber aaa  
java -cp fourinone.jar; Subscriber bbb  
java -cp fourinone.jar; Subscriber ccc
```



图 5-6 Subscriber

我们启动了 3 个订阅者，名称依次为 aaa、bbb、ccc，订阅者启动好后处于事件监听状态，等待发布者投递消息，如图 5-6 所示。

3) 运行 Publisher，结果如图 5-7 所示。

```
java -cp fourinone.jar; Publisher
```

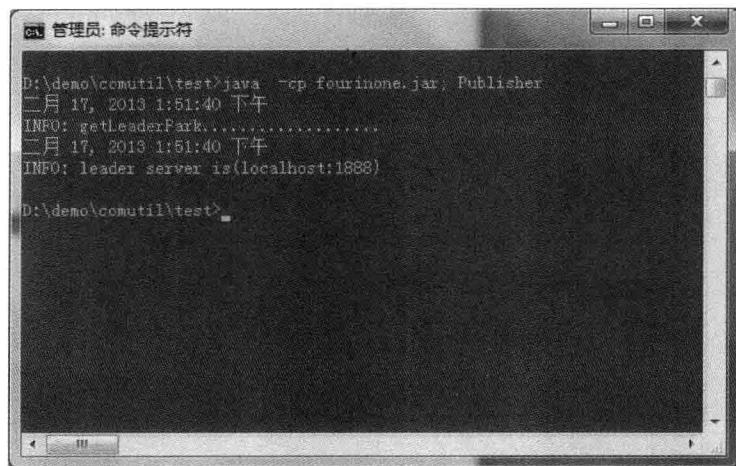


图 5-7 Publisher

运行 Publisher 开始投递消息，投递完成后 Publisher 退出，我们看看各个订阅者的窗口显示如图 5-8 所示。



图 5-8 订阅消息结果

可以看到，3个订阅者都显示出收到了发布者的“hello world”的消息。

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// Subscriber
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import com.fourinone.LastestEvent;
import com.fourinone.LastestListener;
import java.util.ArrayList;

public class Subscriber implements LastestListener
{
    private static ParkLocal pl = BeanContext.getPark();

    public boolean happenLastest(LatestEvent le)
    {
        ObjectBean ob = (ObjectBean)le.getSource();
        ArrayList arr = (ArrayList)ob.toObject();
        System.out.println("published message:"+arr);
        ObjectBean newob = pl.update(ob.getDomain(), ob.getNode(), new
            ArrayList());
        le.setSource(newob);
        return false;
    }

    public static void subscrib(String topic, String subscribeName,
        LatestListener lister)
    {
        ArrayList arr = new ArrayList();
        ObjectBean ob = pl.create(topic, subscribeName, arr);
        pl.addLatestListener(topic, subscribeName, ob, lister);
    }

    public static void main(String[] args)
    {
        subscrib("topic1", args[0], new Subscriber());
    }
}

// Publisher
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
```

```

import java.util.ArrayList;
import java.util.List;

public class Publisher
{
    private static ParkLocal pl = BeanContext.getPark();

    public static Object publish(String topic, Object obj)
    {
        List<ObjectBean> oblist = pl.get(topic);
        if(oblist!=null)
        {
            for(ObjectBean ob:oblist)
            {
                ArrayList arr = (ArrayList)ob.toObject();
                arr.add(obj);
                pl.update(ob.getDomain(), ob.getNode(), arr);
            }
        }else return null;
        return obj;
    }

    public static void main(String[] args)
    {
        publish("topic1", "helloworld");
    }
}

```

Fourinone 不实现 JMS 的规范，不提供 JMS 的消息确认和消息过滤等特殊功能，不过开发者可以基于 Fourinone 去扩充自己的这些功能，包括 MQ 集群。如果需要事务处理可以将多个消息封装在一个集合内进行发送，上面的队列接收者收到消息后删除实际上是一种消息确认方式，也可以将业务逻辑处理完后再进行删除。如果需要持久保存消息可以在封装一层消息发送者，发送前后根据需要进行数据库或者文件持久保存。利用一个独立的 Domain/Node 建立队列或者主题的 key 隐射，再仿照上面分布式缓存的智能根据 key 定位服务器的做法实现集群管理。

第 6 章

分布式文件系统的实现

本章讲述如何使用 FTTP 去实现一个分布式文件系统，包括 FTTP 的架构原理和远程文件各种方式的访问和操作，以及整型数据处理等，包含了每一步的具体操作，可帮助入门的读者快速上手。

在 FTTP 中通过 FtpAdapter 和 FileAdapter 实现文件 IO 的支持。其中，FtpAdapter 提供对远程文件的操作，FileAdapter 提供对本地文件的操作，两者的 API 和使用相似，这里主要说明 FtpAdapter。

FtpAdapter 提供了对分布式文件的便利操作，将集群中所有机器的硬盘资源利用起来，通过统一的 FTTP 文件路径访问，并且在 Windows 和 Linux 中都受到支持。操作系统上的任何目录文件都可以通过添加 FTTP 协议头和 IP 去访问 `ftp://IP 或域名 /Windows 或 Linux 原有目录文件名：`

访问 Windows 系统的 `d:/data/a.log` 文件：

Windows: `ftp://192.168.0.1/d:/data/a.log`

访问 Linux 系统的 `/home/user/a.log` 文件：

Linux: `ftp:// 192.168.0.1/home/user/a.log`

获取远程文件内容变得更简单，比如以这样的方式读取远程文件：

```
FtpAdapter fa = FtpAdapter("ftp://192.168.0.1/home/log/a.log");
fa.getFtpReader().readAll();
```

这里读取了 192.168.0.1 这台 Linux 服务器上的 `/home/log/a.log` 文件中的所有内容。

提供对集群文件的操作支持，包括：

1) 元数据访问，包括添加删除，按块拆分，高性能并行读写，排他读写（按文件部分内容锁定），随机读写，集群复制等。

- 2) 对集群文件的解析支持(包括按行、按分割符、按最后标识读取)。
- 3) 对整型数据的高性能读写(ArrayInt 比 ArrayList 存得更多更快)。
- 4) 两阶段提交和事务补偿处理。
- 5) 自带一个集群文件浏览器,可以查看集群所有硬盘上的文件(不同于 Hadoop 的 NameNode,没有单点问题和容量限制)。

总的来说,将集群看做一个操作系统,像操作本地文件一样操作远程文件。

但是 Fourinone 并不提供一个完整的分布式存储系统,比如文件数据的导入导出、拆分存储、负载均衡、备份容灾等存储功能,不过 Fourinone 会使完成这些工作变得简单,使开发人员可以利用 API 方便地设计和实现这些功能,用来满足自己的特定需求。

6.1 FTTT 架构原理解析

目前在云计算和大数据浪潮的推动下,分布式文件存储技术的发展已经如火如荼,不论大企业还是小创业公司都想推自己的云存储产品和服务,下面我们来关注一下这个领域背后的落地技术实现,从市场上已有的分布式文件系统实现技术入手分析,先做一个背景铺垫,再以此对照阐述 FTTT 的分布式文件技术架构。

目前市场上已有的分布式文件系统的元数据实现主要有 Google 提出的 GFS 的论文理论和按照该论文实现的 Hadoop 的 HDFS 分布式文件系统。在 HDFS 分布式文件系统中,NameNode 是分布式文件元数据管理的中心服务器,负责管理文件系统的目录命名和客户端对文件的访问。

在 HDFS 分布式文件系统中,要存储一个文件,其内容会被拆分成多个块,这些块数据散落存储在不同的计算机节点上,而该文件的路径目录名称等元数据存放在 NameNode 机器上,该文件被拆分的块位置信息等也存放在其上。当客户端读取该文件时,会访问 NameNode 查找该文件路径并获取拆分的块位置信息,然后直接到存放各块的计算机上读取块内容合并得到结果。

可见,NameNode 负责保存和管理所有的 HDFS 元数据,它维持着一个像操作系统文件资源管理器一样的树状目录结构,通过它可以访问、查询、获取文件的元数据信息。

但是 Hadoop 单一 NameNode 的设计会严重制约整个 Hadoop 的可扩展性和可靠性。首先,NameNode 是整个系统中明显的单点故障源,单一 NameNode 的内存容量有限,使得 Hadoop 集群的节点数量被限制到 2000 个左右,能支持的文件系统大小被限制在 10 ~ 50PB,最多能支持的文件数量大约为 1.5 亿 左右(注:实际数量取决于 NameNode 的内存大小)。同时,在集中式的 NameNode 造成数据块存储计算机的心跳报告也会对 NameNode 的性能造成严

重的影响。对于这样的系统，有 1800 个存储节点，每个存储节点有 3T 存储，整个集群大约有 1.8PB 有效存储 ($1800 \times 3T/3$ ，假设每个数据块有 3 份备份)，那么每个存储节点上有大约 50000 个左右的数据块（假设数据块大小是 64MB，然而有的数据块并没有达到 64MB 大小）。如果存储节点每小时会发送一次块信息的心跳报告，那么 Namenode 每两秒会收到一次报告信息，每个报告信息包含 50000 条数据，处理这些数据无疑会占用相当资源。实际上，集群的 NameNode 重启需要数小时，这大大降低了系统的可用性。

由此可见目前已有技术，如 Hadoop 的 NameNode 管理文件元数据的方式，面临单点故障、容量限制、内存限制、性能限制等问题。为此，Fourinone 尝试通过 FFTP 分布式文件技术屏蔽这些问题，并让设计具有高可扩展性和高可靠性。

FFTP 对于分布式文件系统的元数据管理设计不同于 Hadoop 的 NameNode，它自己不维持一个随着扩容而逐渐庞大的元数据信息的存储，而是在底层利用操作系统本身已有的文件元数据信息，因为操作系统本身已经实现了对所有文件的元数据管理和存储，所以像 NameNode 这样自身维持一个庞大的分布式元数据信息是一种重复建设，而且实现复杂，容易产生问题。如果利用操作系统的元数据信息，会大大减少目录节点的存储量，目录节点仅仅只需要维持一个存储计算机的集群信息，它背后联系着集群每台存储计算机的元数据信息，这样不存在节点数量限制，可以无限扩充。当客户端需要获取文件元数据信息时，会向目录节点提交一个集群路径（FFTP 路径），目录节点解析该路径，并找到背后关联的存储计算机上的文件元数据信息，然后返回给客户端节点。在整个过程中目录节点只是一

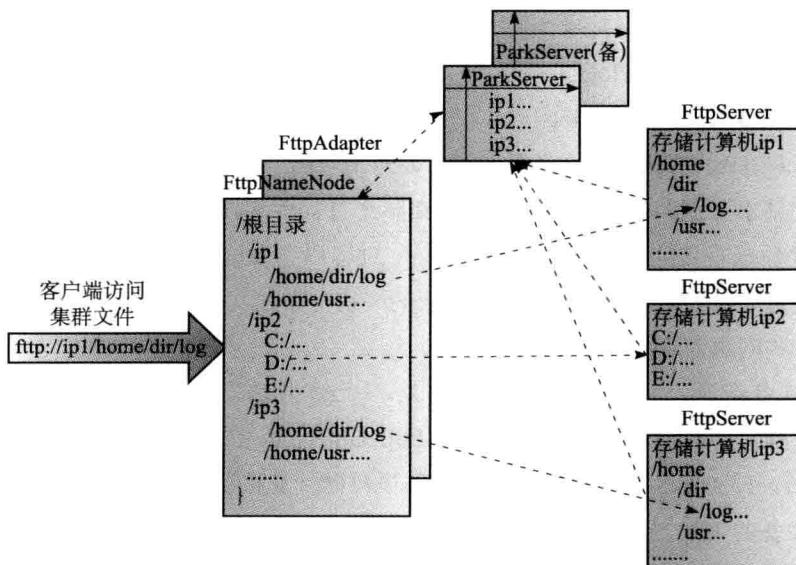


图 6-1 FFTP 架构

个解析和中转的角色，它本身不维持元数据存储，它是虚拟的，松耦合的，所以也不存在过多数量的数据块元数据心跳报告导致性能低下。FTTP 架构的设计结构和过程描述参见图 6-1。

从图 6-1 可以看到，FTTP 分布式文件系统的架构主要由 FtpAdapter、FtpServer、FtpNameNode、ParkServer 组成。

说明：

1) “ParkServer” 在这里主要提供协调服务，它可以部署在一台独立的计算机节点上，用来管理存储计算机节点的集群信息。存储计算机节点在加入集群时首先在“ParkServer”中进行登记，然后“ParkServer”不断检测各存储计算机节点的在线状况。

2) “FtpServer” 为一个存储计算机节点，可以部署在多个机器上，它主要操作所在计算机的操作系统的存储文件和获取文件元数据信息并返回给“FtpExploer”。

3) “FtpNameNode” 是集群存储文件浏览器，可以和客户端在同一台计算机上。通过“FtpNameNode”可以浏览获取集群中所有计算机的文件元数据。“FtpNameNode”看似提供一个完整的文件目录树结构，包括了整个集群文件系统，但实际上只是一个虚拟的、动态的结构，它先从“ParkServer”获取集群可用于存储的计算机节点，将它们按照 IP 进行显示，并且根据客户端提交的 FTTP 访问请求，通过“FtpAdapter”进行 FTTP 的协议解析，然后即时访问后端的“FtpServer”，获取操作系统的文件元数据，通过包装转换发回客户端。“FtpNameNode”本身不维护集群文件元数据的存储。

4) “FtpAdapter” 提供对远程文件的所有操作和协议转换，客户端通过统一的 FTTP 文件协议访问元数据，FTTP 保持操作系统文件目录的原始结构，比如：

- Windows: `ftp://ip/d:/log/`
- Linux: `ftp://ip/home/dir/log/`

客户端以 FTTP 的方式访问文件元数据，发送请求给“FtpAdapter”，之后“FtpAdapter”解析客户端的请求并返回它需要的元数据，在整个过程中客户端不需要与背后各存储计算机上的真实文件数据直接交互。“FtpNameNode”对客户端来说就是一个大的虚拟的集群文件目录，可以通过“FtpAdapter”获取到它需要的一切分布式文件元数据。

有以下一些问题需要考虑。

- **关于单点问题：**由于 FtpNameNode 是一个虚拟动态的实现，它是可以随时被复制和替换，因此不存在单点问题。FtpNameNode 获取集群文件元数据时会依赖 ParkServer，ParkServer 存储着最新集群里存储计算机节点的信息，由于 ParkServer 是一主多备的关系，当 ParkServer 出现故障，可以及时从它的备份获取节点信息。因此整体设计结构有效地避免了单点故障问题。

- **关于文件拆分：**如果将一个大的文件拆分后散落在不同 FtpServer 节点上保存，那么它们的元数据如何保存呢？对于 FtpNameNode 来说，它也仅仅增加一些文件拆分的信息存储，而每个拆分的块文件元数据也保存在 FtpServer 的操作系统上，FtpNameNode 本身不维持这些块文件的元数据存储，因此 FtpNameNode 没有一个容量的限制。
- **关于增容减容：**由于 FtpServer 和 FtpNameNode 之间是一个松散的结构关系，因此 FtpServer 可以自由增加或者减少，FtpNameNode 会及时获取到集群存储节点的改变。这些改变对元数据的管理没有影响，因为 FtpNameNode 本身不维持整个分布式文件系统的元数据的存储，所以 FtpServer 的增加或减少没有太大影响，FtpNameNode 会动态地获取到最新的元数据状况，FtpServer 所在计算机的操作系统维持着文件元数据的原始存储。
- **FTP 与 FTP/HTTP 的区别：**FTP 是一种文件上传下载的协议，有一套完整的命令规范，比如登录、put、get 等，但是它不能用于获取文件元数据信息，并且也只支持本地服务器两台计算机之间的文件传输，因此 FTP 不用于分布式文件系统。超文本传输协议（HTTP）是一种通信协议，它允许将超文本标记语言（HTML）文档从 Web 服务器传送到 Web 浏览器。HTTP 允许以二进制的方式上传下载文件，但是封装在 HTTP 报文中，并指定特定的内容格式的方式。不过 HTTP 也只支持浏览器和 Web 服务器两者之间的文件内容的上传下载，不能获取文件元数据和管理文件元数据，因此它也不用于分布式文件系统。

6.2 搭建配置 FtpAdapter 环境

FtpAdapter 的使用很简单，在每台计算机上启动一个 FtpServer 即可，另外需要启动一个 ParkServer 负责协调。ParkServer 也可以和其中一个 FtpServer 放在相同计算机上。

1) 启动 ParkServerDemo (它的 IP 端口已经在配置文件的 PARK 部分的 SERVERS 指定)，如图 6-7 所示。

```
java -cp fourinone.jar; ParkServerDemo
```

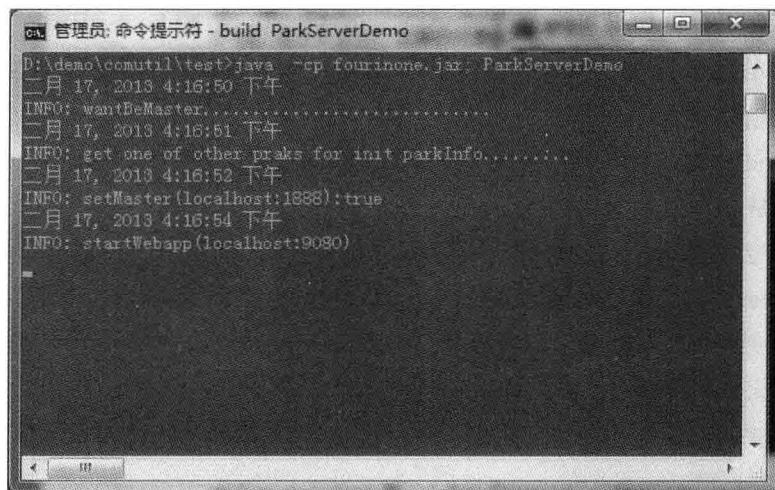


图 6-2 启动 ParkServerDemo

启动结束要检查界面命令行输出 Webapp 已经在 9080 端口监听。

2) 每台计算机启动 FtpServer，需要指定该计算机的 IP 为输入参数结果如图 6-3 所示：

```
java -cp fourinone.jar; FtpServer localhost
```

(这里是本机演示，如果是远程计算机要输入 IP)

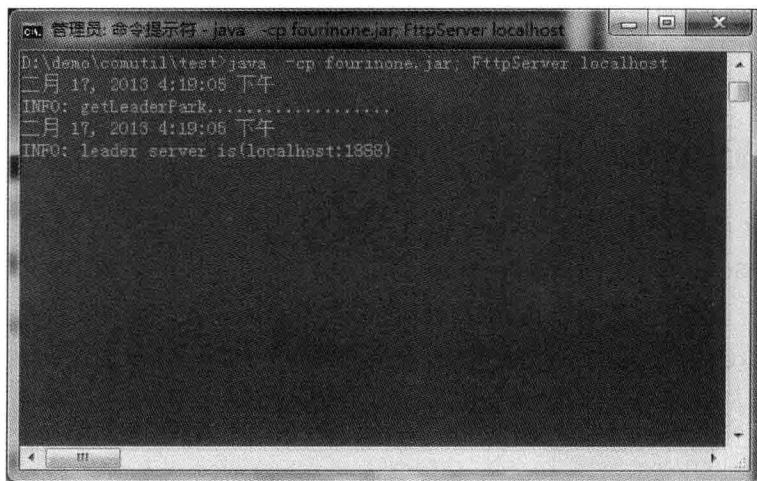


图 6-3 启动 FtpServer

启动好后打开浏览器访问 <http://localhost:9080/admin/ftp.jsp>，可以看到整个集群的文件系统（如图 6-4 所示），说明启动成功。这里的 localhost 是默认值，通常为 ParkServer 的 IP。

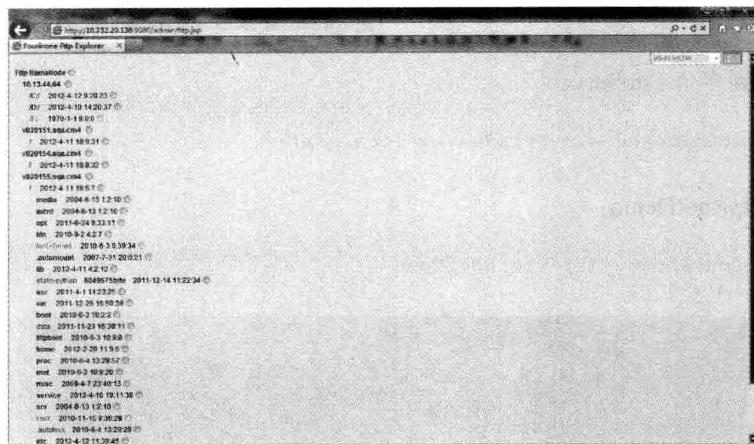


图 6-4 Ftp NameNode

**注意**

由于 FtpAdapter 实现了简单身份认证，因此初次打开集群文件浏览器会提示输入账号密码。打开 config.xml 配置文件，找到 WEBAPP 部分的配置项，SERVERS 可以配置 IP 和端口，USERS 可以配置多个账号密码，用逗号分隔。

```
<PROPSROW DESC="WEBAPP">
  <SERVERS>localhost:9080</SERVERS>
  <USERS>admin:admin,guest:123456,test:test</USERS> </PROPSROW>
```

可以动态增加或者关闭 FtpServer，然后刷新 ftp.jsp，查看及时更新的文件目录。

6.3 访问集群文件根目录

成功启动 FtpServer 后，可以使用 FtpAdapter 的 API 进行相关操作。FtpAdapter.ftpRoots() 是一个静态方法，可以得到集群文件系统根目录，它返回一个 String 数组，通常是 IP 字符对应每台计算机。

listRoots() 方法可以得到每台计算机上的硬盘目录，比如：

```
FtpAdapter fa = new FtpAdapter("ftp://" + ftproots[i]);
String[] roots = fa.listRoots();
```

FtpRootDemo 显示了获取集群文件的根目录和它们各自下面的硬盘目录，如图 6-5 所示。

下面介绍运行步骤：

1) 启动 ParkServerDemo：

```
java -cp fourinone.jar; ParkServerDemo
```

2) 启动一到多个 FtpServer:

```
java -cp fourinone.jar; FtpServer localhost
```

3) 运行 FtpRootDemo:

```
java -cp fourinone.jar; FtpRootDemo
```

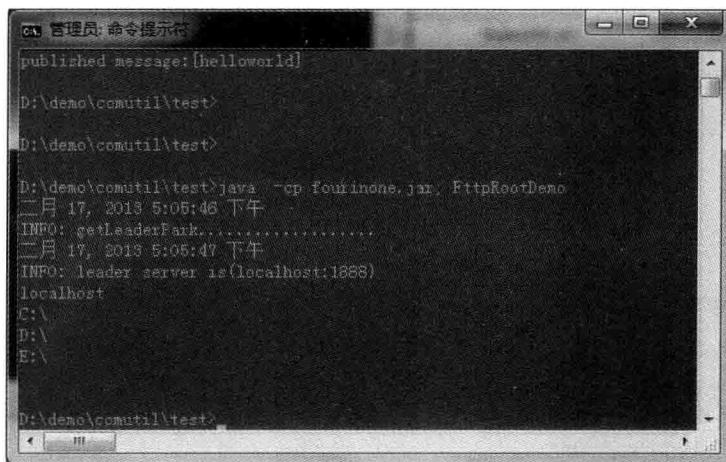


图 6-5 FtpRootDemo

由于我们只在本机启动了 FtpServer，因此从图 6-5 中可以看到将本机 IP 和硬盘根目录显示出来。

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// FtpServer
import com.fourinone.BeanContext;
public class FtpServer
{
    public static void main(String[] args)
    {
        BeanContext.startFtpServer(args[0]);
    }
}
```

```

        }

    }

// FtpRootDemo
import com.fourinone.FtpAdapter;
import com.fourinone.FtpException;

public class FtpRootDemo
{
    public static void main(String[] args){
        try{
            String[] ftproots = FtpAdapter.fttpRoots();
            for(int i=0;i<ftproots.length;i++){
                System.out.println(ftproots[i]);

                FtpAdapter fa = new FtpAdapter("ftp://"+ftproots[i]);
                String[] roots = fa.listRoots();
                for(int j=0;j<roots.length;j++){
                    System.out.println(roots[j]);
                }
                System.out.println("");
            }
        }catch(FtpException fe){
            fe.printStackTrace();
        }
    }
}

```

6.4 访问和操作远程文件

下面实例化一个 `FtpAdapter` 类，并输入一个 FTP 路径，可以是一个目录路径，也可以是个文件路径。

```
FtpAdapter fa = new FtpAdapter("ftp://192.168.0.8/home/qianfeng.py/ftp/
tmp/1.log");
```

`FtpAdapter` 类提供了一系列访问远程文件的方法：

- `createDirectory` 和 `createFile` 方法分别用于创建目录和文件。
- `rename` 方法用于进行重命名，比如将 `fa` 的文件名重命名为 `2.log`:

```
fa.rename("2.log")
```

`rename` 方法返回一个新的 `FtpAdapter` 对象，表示命名后的文件。

- `copyTo` 方法用于进行集群内复制，比如：

```
fa.copyTo("ftp:// 192.168.0.9/home/qianfeng.py/ftp/tmp/3.log")
```

表示将 fa 文件复制到 192.168.0.9 计算机上的指定目录下。

□ copyTo 方法返回一个新的 FtpAdapter 对象，表示复制后的文件。

□ delete 方法用于删除该文件，比如：

```
fa.delete()
```

注意

在某些情况下，delete 方法可能没有那么快及时生效，为了能高效读写文件，如果虚拟机内有缓存，导致不能马上删除，可以通过修改文件后缀设置一个删除标记或者移入一个回收站文件夹，再在其他时间统一删除。

getProperty 方法用于获取文件或者目录属性，它返回一个 FileProperty 对象，此对象中包含以下方法：

- exists() 文件或者目录存在
- isFile() 是文件
- isDirectory() 是目录
- isHidden() 是隐藏文件
- canRead() 可读
- canWrite() 可写
- lastModifiedDate() 最后修改时间
- length() 文件长度
- getParent() 父目录
- getName() 名称
- getPath() 路径
- list() 子文件列表

用于获取文件属性信息的方法如下：

□ getChildProperty 方法返回一个 FileProperty 数组，用于获取所有子文件的属性。

□ FtpOperateDemo 用于演示创建一个远程文件目录，并在该目录下创建、重命名、复制文件，打印文件属性，最后删除的基本操作功能。

运行步骤如下：

1) 启动 ParkServerDemo：

```
java -cp fourinone.jar; ParkServerDemo
```

2) 启动一到多个 FtpServer：

```
java -cp fourinone.jar; FtpServer localhost
```

3) 运行 FtpOperateDemo:

```
java -cp fourinone.jar; FtpOperateDemo
```

完整的 demo 源码如下：

```
// FtpOperateDemo
import com.fourinone.FtpAdapter;
import com.fourinone.FftpException;
import com.fourinone.FtpAdapter.FileProperty;

public class FtpOperateDemo{
    public static void printProp(FileProperty prop){
        System.out.println("exists:"+prop.exists());
        System.out.println("isFile:"+prop.isFile());
        System.out.println("isDirectory:"+prop.isDirectory());
        System.out.println("isHidden:"+prop.isHidden());
        System.out.println("canRead:"+prop.canRead());
        System.out.println("canWrite:"+prop.canWrite());
        System.out.println("lastModifiedDate:"+prop.lastModifiedDate());
        System.out.println("length:"+prop.length());
        System.out.println("getParent:"+prop.getParent());
        System.out.println("getName:"+prop.getName());
        System.out.println("getPath:"+prop.getPath());
        if(prop.isDirectory())
            System.out.println("fp.list():"+prop.list().length);
        System.out.println("");
    }

    public static void main(String[] args){
        try{
            FtpAdapter dir = new FtpAdapter("ftp://localhost/d:/ftp/tmp/");
            dir.createDirectory();
            FileProperty dirProp = dir.getProperty();
            printProp(dirProp);

            FtpAdapter f1 = new FtpAdapter(dirProp.getPath(),"1.log");
            FtpAdapter f2 = null;
            FtpAdapter f3 = null;

            if(dirProp.exists()){
                f1.createFile();
                f2 = f1.rename("2.log");
                f3 = f2.copyTo("ftp://localhost/d:/ftp/tmp/3.log");
            }

            FileProperty[] childProps = dir.getChildProperty();
            for(int i=0;i<childProps.length;i++){
                printProp(childProps[i]);
            }

            System.out.println(f1.delete());
        }
    }
}
```

```
        System.out.println(f2.delete());
        System.out.println(f3.delete());
        System.out.println(dir.delete());

        dir.close();
        f1.close();
        f2.close();
        f3.close();
    }catch(FtpException fe){
        fe.printStackTrace();
    }
}
}
```

6.5 集群内文件复制和并行复制

集群内文件复制是经常要应对的需求，比如备份容灾，文件迁移，同步数据等。

FtpAdapter 提供了简单高效的文件复制方法，支持远程文件的集群内复制：

```
FtpAdapter fromfile = new FtpAdapter("ftp://192.168.0.1/home/log/a.log");
FtpAdapter tofile = fromfile.copyTo("ftp://192.168.0.2/home/log/
a.log",FileAdapter.m(1));
```

上面代码代表将 a.log 文件复制到其他机器并得到相应的文件对象。

这里的 copyTo 方法的第二个参数，表示复制时，每次以 1M/S 的速度传输：

```
FileAdapter.m(1) 1M
FileAdapter.g(1) 1G
FileAdapter.k(1) 1k
```

数字 1 可以自由设置为其他数字，copyTo 的默认值是每次以 1M/S 的速度复制，可以根据网络情况调整这个参数，达到最优化。



注意

这里的复制方法名叫做 copyTo 而不是 copy，因为它们之间存在区别，copyTo 只是将前一文件内容复制到后一文件内容里，追加到末尾，但并不从头覆盖前一个文件的内容。通常在操作系统上覆盖文件，在用户不知情的情况下弹出警告框提示获得同意，因此 copyTo 能避免在未经许可下覆盖旧文件。如果你需要这样做，可以参考 6.6 节“读写远程文件”，设置读写位置为文件开始，从头覆盖文件内容。

FtpCopyDemo 演示了集群中两台机器间的复制功能。

运行步骤如下：

1) 启动 ParkServerDemo:

```
java -cp fourinone.jar; ParkServerDemo
```

2) 在 192.168.0.1 机器上启动 FtpServer:

```
java -cp fourinone.jar; FtpServer 192.168.0.1
```

3) 在 192.168.0.2 机器上启动 FtpServer:

```
java -cp fourinone.jar; FtpServer 192.168.0.2
```

4) 运行 FtpCopyDemo:

```
java -cp fourinone.jar; FtpCopyDemo
```

完整 demo 源码如下:

```
// FtpCopyDemo
import com.fourinone.FtpAdapter;
import com.fourinone.FtpException;
import java.util.Date;

public class FtpCopyDemo
{
    public static void main(String[] args){
        try{
            long begin = (new Date()).getTime();
            FtpAdapter fromfile = new FtpAdapter("ftp://192.168.0.1/home/
                someone/ftp/tmp/a.log");
            FtpAdapter tofile = fromfile.copyTo("ftp://192.168.0.2/home/someone/
                ftp/tmp/a.log");
            if(tofile!=null)
                System.out.println("copy ok.");
            long end = (new Date()).getTime();
            System.out.println("time:"+ (end-begin)/1000+"s");
        }catch(FtpException fe){
            fe.printStackTrace();
        }
    }
}
```

如果要进行并行复制，可以使用 tryCopyTo，它的使用和 copyTo 一样，只不过是立即返回一个 Result<FtpAdapter> 对象，需要检查 Result 的 getStatus 是否复制完成，状态显示就绪代表复制已完成，这时可以获取到复制后的文件对象。

FtpMulCopyDemo 演示了将一台计算机上的 a.log 文件并行复制到 4 台计算机上，并通过结果状态检查复制是否完成。

下面是内网环境下向 4 台机器复制 1GB 文件的测试结果（均为 4 核 4GB 内存配置）：

- 内网（并行复制）：完成工作的耗时为 39 秒，速度大约是 105M/s。
- 内网（串行复制）：完成工作的耗时为 60 秒，速度大约是 68M/s。
- 由此可以观察到内网并行复制的速度比串行要快很多。
- 局域网的传输数据的极限是 100M/s，传统的串行复制无法超越这个速度，但是并行的传送总量和花费时间算下来会优于串行传送。
- 外网环境受网络带宽局限，速度大约是 3.6M/s。

运行步骤如下：

1) 启动 ParkServerDemo：

```
java -cp fourinone.jar; ParkServerDemo
```

2) 在每台机器上启动 FtpServer：

```
java -cp fourinone.jar; FtpServer 192.168.0.1
java -cp fourinone.jar; FtpServer 192.168.0.2
java -cp fourinone.jar; FtpServer 192.168.0.3
java -cp fourinone.jar; FtpServer 192.168.0.4
java -cp fourinone.jar; FtpServer 192.168.0.5
```

可以通过访问 <http://localhost:9080/admin/ftp.jsp> 检查集群文件系统启动状况。

3) 运行 FtpMulCopyDemo：

```
java -cp fourinone.jar; FtpMulCopyDemo
```

完整 demo 源码如下：

```
// FtpMulCopyDemo
import com.fourinone.FtpAdapter;
import com.fourinone.FtpException;
import com.fourinone.FileAdapter;
import com.fourinone.Result;
import java.util.Date;

public class FtpMulCopyDemo
{
    public static void main(String[] args){
        long begin = (new Date()).getTime();
        try{
            Result<FtpAdapter>[] rs = new Result[4];
            String fromftp = "ftp://192.168.0.1/home/someone/ftp/tmp/a.log";
            FtpAdapter fa1 = new FtpAdapter(fromftp);
            FtpAdapter fa2 = new FtpAdapter(fromftp);
            FtpAdapter fa3 = new FtpAdapter(fromftp);
            FtpAdapter fa4 = new FtpAdapter(fromftp);

            rs[0]=fa1.tryCopyTo("ftp://192.168.0.2/home/someone/ftp/tmp/a.log",
            rs[1]=fa2.tryCopyTo("ftp://192.168.0.3/home/someone/ftp/tmp/a.log",
            rs[2]=fa3.tryCopyTo("ftp://192.168.0.4/home/someone/ftp/tmp/a.log",
            rs[3]=fa4.tryCopyTo("ftp://192.168.0.5/home/someone/ftp/tmp/a.log");
        }
    }
}
```

```
    FileAdapter.m(1));
rs[1]=fa2.tryCopyTo("ftp://192.168.0.3/home/someone/ftp/tmp/a.log",
    FileAdapter.m(1));
rs[2]=fa3.tryCopyTo("ftp://192.168.0.4/home/someone/ftp/tmp/a.log",
    FileAdapter.m(1));
rs[3]=fa4.tryCopyTo("ftp://192.168.0.5/home/someone/ftp/tmp/a.log",
    FileAdapter.m(1));

    int n=0;
    while(n<4){
        for(int i=0;i<rs.length;i++){
            if(rs[i]!=null&&rs[i].getStatus()!=Result.NOTREADY){

System.out.println(i+",getStatus:"+rs[i].getStatus()+",getResult:"+rs[i].
    getResult());
            rs[i]=null;
            n++;
        }
    }
}

fa1.close();
fa2.close();
fa3.close();
fa4.close();
}catch(FtpException fe){
    fe.printStackTrace();
}
long end = (new Date()).getTime();
System.out.println("time:"+ (end-begin)/1000+"s");
}
}
```

6.6 读写远程文件

FtpAdapter 是通过 FtpReadAdapter 来直接读取远程文件内容：

```
FtpAdapter fa = new FtpAdapter("ftp://192.168.0.1/home/log/1.log");
FtpReadAdapter reader = fa.getFtpReader();
byte[] bts = reader.readAll();
```

上面的代码是读取整个文件的内容，如果文件内容很大，每次只读取一部分内容，需要指定 FtpReadAdapter 的读取范围：

```
FtpReadAdapter reader = fa.getFtpReader(5,10);
byte[] bts = reader.readAll();
```

上面代码表示从第 5 个字节，往后读 10 个字节。读取方式的示例还有：

- fa.getFtpReader(5, FileAdapter.m(8)) 从第 5 个字节往后读 8M。

- fa.getFtpReader(5, FileAdapter.k(512)) 从第 5 个字节往后读 512K。

FtpAdapter 是通过 FtpWriteAdapter 来直接写入远程文件内容：

```
FtpAdapter fa = new FtpAdapter("ftp://192.168.0.1/home/log/1.log");
FtpWriteAdapter writer = fa.getFtpWriter();
int r = writer.write("hello world".getBytes());
```

上面的 FtpWriteAdapter 没有指定写入范围，默认追加在文件末尾，如果需要指定范围，则应使用如下代码：

```
FtpWriteAdapter writer = fa.getFtpWriter(5,10);
int r = writer.write("hello world".getBytes());
```

上面代码表示从第 5 个字节开始，往后写 10 个字节，写入内容为“hello world”，如果写入内容超出 10 个字节，则截断，不够则填补空位。

除 readAll 和 write 外，FtpAdapter 也提供 readAllSafety 和 writeSafety 方法，它们的用法一样，但是后两者代表排它读写，主要用于并发读写。

对于数字存储，FtpAdapter 也提供 getIntFtpReader 和 getIntFtpWriter 用于整型读写，操作与字节读写类似，只是写入或返回的是整数，比如：

- fa.getIntFtpReader(5,3) 表示从第 5 个整数开始，往后读 3 个整数。

- fa.getIntFtpWriter().writeInt(new int[]{1,2,3}) 表示将一个整数数组写入文件末尾。

同样，整型读写也都提供排它读写。FtpWriteReadDemo 演示了对远程文件的读写操作。

运行步骤如下：

1) 启动 ParkServerDemo：

```
java -cp fourinone.jar; ParkServerDemo
```

2) 在 192.168.0.1 机器上启动 FtpServer：

```
java -cp fourinone.jar; FtpServer 192.168.0.1
```

3) 在本地运行 FtpWriteReadDemo，远程读写 192.168.0.1 上的文件：

```
java -cp fourinone.jar; FtpWriteReadDemo
```

完整 demo 源码如下：

```
// FtpWriteReadDemo
import com.fourinone.FtpAdapter;
import com.fourinone.FtpException;
```

```

public class FtpWriteReadDemo
{
    public static void ftpWrite(){
        try{
            FtpAdapter fa = new FtpAdapter("ftp://192.168.0.1/home/log/1.log");
            fa.getFtpWriter().write("hello world".getBytes());
            fa.close();
        }catch(FtpException fe){
            fe.printStackTrace();
        }
    }

    public static void ftpRead(){
        try{
            FtpAdapter fa = new FtpAdapter("ftp://192.168.0.1/home/log/1.log");
            byte[] bts = fa.getFtpReader().readAll();
            System.out.println("logstr:"+new String(bts));

            byte[] hellobts = fa.getFtpReader(0,5).readAll();
            System.out.println("hellostr:"+new String(hellobts));

            fa.close();
        }catch(FtpException fe){
            fe.printStackTrace();
        }
    }

    public static void main(String[] args){
        ftpWrite();
        ftpRead();
    }
}

```

6.7 解析远程文件

假设通过 `FtpAdapter` 已经读取到远程文件中一部分数据，如下：

```

FtpAdapter fa = new FtpAdapter("ftp://192.168.0.1/home/log/1.log");
FtpReadAdapter reader = fa.getFtpReader();
byte[] bts = reader.readAll();

```

上面代码得到一个 `byte` 数组，那么如何解析这个数组呢，可以通过 `byte` 初始化得到一个 `ByteReadParser`：

```
ByteReadParser brp = FileAdapter.getByteReadParser(bts);
```

`ByteReadParser` 提供了按数量、按行、按分割符、按结束符解析等很便利的方法：

(1) 按数量

```
public byte[] read(int totalnum);
```

(2) 按行如: brp.read(100); 表示读取前 100 个字符

```
public byte[] readLine();
```

如: new String(brp.readLine()) 表示读取一行字符, 多次调用直到末尾。注意, readLine 是以 “\r\n” 为分割符号。

(3) 按分割符

```
public byte[] read(byte[] split);
```

如: brp.read("@#\$.getBytes()); 表示读取以 “@#\$” 做分割符号的前面的字符段, 多次调用直到末尾。

(4) 按结束符

```
public byte[] readLast(byte[] split);
```

如: brp.readLast (“。” .getBytes()); 表示读取最后一个以句号结尾的前面的字符段。

按块读取后解析如何处理截断数据? 根据内存大小一次读取 8M ~ 64M 的数据, 通过 readLast 找到最后一行前面的所有数据, 根据 byte[] 大小记录好位置, 然后再通过 readLine 按行解析这个 byte[]。下一次往前读的时候从记录位置开始。

FtpParseDemo 演示使用 ByteReadParser 的基本解析操作。

运行步骤如下:

1) 启动 ParkServerDemo:

```
java -cp fourinone.jar; ParkServerDemo
```

2) 在 192.168.0.1 机器上启动 FtpServer:

```
java -cp fourinone.jar; FtpServer 192.168.0.1
```

3) 在本地运行 ByteReadParser, 远程读写 192.168.0.1 上的 /home/log/b.log 文件的内容并解析, b.log 的内容可以自动更改:

```
java -cp fourinone.jar; ByteReadParser
```

完整 demo 源码如下:

```
// FtpParseDemo
```

```

import com.fourinone.FtpAdapter;
import com.fourinone.FtpException;
import com.fourinone.FileAdapter;
import com.fourinone.FileAdapter.ByteReadParser;

public class FtpParseDemo
{
    public static void main(String[] args){
        try{
            FtpAdapter fa = new FtpAdapter("ftp://192.168.0.1/home/log/b.log");
            byte[] bts = fa.getFtpReader(0,100).readAll();
            System.out.println(bts.length);
            ByteReadParser brp = FileAdapter.getByteReadParser(bts);
            byte[] splitbts = brp.read(" ".getBytes());
            System.out.println(new String(splitbts));
            byte[] linebts = brp.readLine();
            System.out.println(new String(linebts));
            byte[] lastbts = brp.readLast("googl".getBytes());
            System.out.println(new String(lastbts));
        }catch(FtpException fe){
            fe.printStackTrace();
        }
    }
}

```

6.8 并行读写远程文件

FtpAdapter 是通过 FtpReadAdapter 的 tryReadAll 方法进行并行读：

```

FtpAdapter fa = new FtpAdapter("ftp://192.168.0.1/home/log/1.log");
Result<byte[]> rs = fa.getFtpReader().tryReadAll();

```

调用 tryReadAll 会立即返回一个 Result<byte[]>, 但是不能马上获取到结果值, 需要轮循检查它的状态是否就绪。

rs.getStatus() 有三种状态：

- Result.NOTREADY 未就绪
- Result.READY 就绪
- Result.EXCEPTION 异常

轮循直到状态准备就绪：

```
while(rs.getStatus()==Result.NOTREADY);
```

状态就绪就可以通过 getResult() 获取到读取结果：

```
byte[] bts = rs.getResult();
```

可以对一个远程文件的不同部分同时并行读写，也可以对多个远程文件同时并行读写，比如：

```
String ftppath = "ftp:// 192.168.0.1/home/log/1.log";
FtpAdapter fa0 = new FtpAdapter(ftppath);
FtpAdapter fa1 = new FtpAdapter(ftppath);
FtpAdapter fa2 = new FtpAdapter(ftppath);
Result<byte[]> rs0 = fa0.getFtpReader(0,5).tryReadAll();
Result<byte[]> rs1 = fa1.getFtpReader(5,5).tryReadAll();
Result<byte[]> rs2 = fa2.getFtpReader(10,5).tryReadAll();
```

上面是 3 个同时并行的读取，分别从一个文件的 0、5、10 位置向后读取 5 个字节。

如果是并行写，则是：

```
Result<Integer>[] rs0 = fa0.getFtpWriter(0,5).tryWrite("hello".getBytes());
Result<Integer>[] rs1 = fa1.getFtpWriter(5,5).tryWrite("world".getBytes());
Result<Integer>[] rs2 = fa2.getFtpWriter(10,5).tryWrite("ftp!".getBytes());
```

注意

上面代码中的 fa0、fa1、fa2 是 3 个不同的 FtpAdapter，而不是同一个 FtpAdapter，否则后面的 getFtpReader 指定的读取范围会覆盖前面的范围，导致意外错乱。由于是对同一个远程文件并行读写，因此 fa0、fa1、fa2 的 ftppath 相同，如果对多个远程文件并行读写，则 ftppath 不同。

FtpMulWriteReadDemo 演示了并行对三个远程文件进行写，然后再并行进行读，在轮循状态时，如果读取到结果后，将 rs[i] 设置为 null 表示不再重复检查。

运行步骤如下：

1) 启动 ParkServerDemo：

```
java -cp fourinone.jar; ParkServerDemo
```

2) 在 192.168.0.1 机器上启动 FtpServer：

```
java -cp fourinone.jar; FtpServer 192.168.0.1
```

3) 在本地运行 FtpMulWriteReadDemo，远程并行读写 192.168.0.1 上的 /home/log/1.log 文件的内容：

```
java -cp fourinone.jar; FtpMulWriteReadDemo
```

完整 demo 源码如下：

```
// FtpMulWriteReadDemo
```

```
import com.fourinone.FtpAdapter;
import com.fourinone.FftpException;
import com.fourinone.Result;

public class FtpMulWriteReadDemo
{
    public static void ftpMulWrite(){
        try{
            String ftppath = "ftp://192.168.0.1/home/log/1.log";
            Result<Integer>[] rs = new Result[3];
            FtpAdapter fa0 = new FtpAdapter(ftppath);
            rs[0]=fa0.getFtpWriter(0,5).tryWrite("hello".getBytes());

            FtpAdapter fa1 = new FtpAdapter(ftppath);
            rs[1]=fa1.getFtpWriter(5,5).tryWrite("world".getBytes());
            FtpAdapter fa2 = new FtpAdapter(ftppath);
            rs[2]=fa2.getFtpWriter(10,5).tryWrite("ftp!".getBytes());

            int n=0;
            while(n<3){
                for(int i=0;i<rs.length;i++){
                    if(rs[i]!=null&&rs[i].getStatus() !=Result.NOTREADY){
                        System.out.println(rs[i].getResult());
                        rs[i]=null;
                        n++;
                    }
                }
            }

            fa0.close();
            fa1.close();
            fa2.close();
        }catch(FftpException fe){
            fe.printStackTrace();
        }
    }

    public static void ftpMulRead(){
        try{
            Result<byte[][]>[] rs = new Result[3];
            String ftppath = "ftp://192.168.0.1/home/log/1.log";

            FtpAdapter fa0 = new FtpAdapter(ftppath);
            rs[0]=fa0.getFtpReader(0,5).tryReadAll();
            FtpAdapter fa1 = new FtpAdapter(ftppath);
            rs[1]=fa1.getFtpReader(5,5).tryReadAll();
            FtpAdapter fa2 = new FtpAdapter(ftppath);
            rs[2]=fa2.getFtpReader(10,5).tryReadAll();

            int n=0;
            while(n<3){
                for(int i=0;i<rs.length;i++){

```

```
        if(rs[i] != null && rs[i].getStatus() != Result.NOTREADY) {
            System.out.println(new String(rs[i].getResult()));
            rs[i] = null;
            n++;
        }
    }

    fa0.close();
    fa1.close();
    fa2.close();
} catch(FtpException fe) {
    fe.printStackTrace();
}
}

public static void main(String[] args) {
    ftpMulWrite();
    ftpMulRead();
}
}
```

6.9 批量并行读写远程文件和事务补偿处理

FileBatch 类支持批量并行读写操作，包括对 FtpAdapter 和 FileAdapter 的支持，它跟并行读写的区别是不需要检查结果，会等到所有并行读写任务全部完成才返回，并在发生异常时提供事务补偿支持。

1. 批量并行读

```
public Result<byte[][]>[] readAllBatch(TryByteReadAdapter[] fras)
```

实现对多个 FtpReadAdapter 任务的批量读，输入一个 FtpReadAdapter 数组，并行进行它们的读取，直到每个 FtpReadAdapter 读完后，以数组的方式批量输出它们对应的结果，比如：

```
FtpReadAdapter[] fras = new FtpReadAdapter[3];
fras[0]=new FtpAdapter(ftppath).getFtpReader(0,5);
fras[1]=new FtpAdapter(ftppath).getFtpReader(5,5);
fras[2]=new FtpAdapter(ftppath).getFtpReader(10,5);
Result<byte[][]>[] rs = new FileBatch().readAllBatch(fras);
```

上面代码表示并行从 3 个位置读一个文件内容，等全部读完后，将对应的结果放在一个数组中返回。

2. 批量并行写

```
FtpWriteAdapter[] fwas = new FtpWriteAdapter[3];
fwas[0]=new FtpAdapter(fttppath).getFtpWriter(0,5);
fwas[1]=new FtpAdapter(fttppath).getFtpWriter(5,5);
fwas[2]=new FtpAdapter(fttppath).getFtpWriter(10,5);
Result<Integer>[] rs = new FileBatch().writeBatch(fwas, "abcde".getBytes());
```

上面代码表示并行对一个文件的 3 个位置写入“abcde”字符，等全部写完后，返回对应的结果数组。



注意

这里与并行读写一样，3 个 FtpReadAdapter 或者 FtpWriteAdapter 是由 3 个不同的 FtpAdapter 生成，而不是同一个生成。

3. 批量并行读写

```
Result<Integer>[] rs = new FileBatch().readWriteBatch(fras,fwas);
```

上面的代码表示将前面的批量读和批量写在一个过程中完成，从 fras 中每个 FtpReadAdapter 读，然后通过 fwas 中对应的每 FtpWriteAdapter 写入，所有读写完成后返回写入结果数组。

4. 事务补偿处理

在批量并行读写过程中，如果其中一个 FtpReadAdapter 或 FtpWriteAdapter 发生错误，那么框架会进行分布式事务处理，进行两阶段提交，然后调用 undo 操作进行事务补偿处理，撤销已经产生的改动和影响。

FileBatch 类提供了对 undo 方法的定义：

```
public Result[] undo(Result[] rtarr)
```

rtarr 是传入的结果，然后返回执行 undo 撤销处理后的结果。

比如调用 readAllBatch 发生错误，FileBatch 会将结果传入 undo 进行撤销操作，然后才返回结果。

因此开发者需要自己实现 undo 方法的内容，继承 FileBatch 类覆盖 undo 方法：

```
public Result[] undo(Result[] rtarr){
    for(int i=0;i<rtarr.length;i++){
        if(rtarr[i].getStatus()==Result.EXCEPTION)
            System.out.println("Result index"+i+" Error");
    }
}
```

```

        return rtarr;
    }
}

```

上面的 undo 方法将发生异常的结果的序号输出显示。

所有的批量读写方法都可以以排它的方式进行，只需指定 boolean locked 参数即可。

另外，除了支持 byte 批量并行读写外，也支持所有的整型批量并行读写，提供的 API 和操作几乎类似。

FtpBatchWriteReadDemo 演示了一个批量并行读、批量并行写、批量并行读写操作和事务补偿操作。

运行步骤如下：

1) 启动 ParkServerDemo：

```
java -cp fourinone.jar; ParkServerDemo
```

2) 在 192.168.0.1 机器上启动 FtpServer：

```
java -cp fourinone.jar; FtpServer 192.168.0.1
```

3) 在本地运行 FtpBatchWriteReadDemo：

```
java -cp fourinone.jar; FtpMulWriteReadDemo
```

完整 demo 源码如下：

```

// FtpBatchWriteReadDemo
import com.fourinone.FtpAdapter;
import com.fourinone.FtpException;
import com.fourinone.Result;
import com.fourinone.FtpAdapter.FtpReadAdapter;
import com.fourinone.FtpAdapter.FtpWriteAdapter;
import com.fourinone.FileBatch;

public class FtpBatchWriteReadDemo extends FileBatch
{
    public void ftpBatchWrite() {
        try{
            String ftppath = "ftp://192.168.0.1/home/log/1.log";
            FtpWriteAdapter[] fwas = new FtpWriteAdapter[3];

            FtpAdapter fa0 = new FtpAdapter(ftppath);
            fwas[0]=fa0.getWriter(0,5);

            FtpAdapter fa1 = new FtpAdapter(ftppath);
            fwas[1]=fa1.getWriter(5,5);

            FtpAdapter fa2 = new FtpAdapter(ftppath);

```

```
    fwas[2]=fa2.getFtpWriter(10,5);

    Result<Integer>[] rs = this.writeBatch(fwas, "abcde".getBytes());

    System.out.println(rs[0].getResult());
    System.out.println(rs[1].getResult());
    System.out.println(rs[2].getResult());

    fa0.close();
    fa1.close();
    fa2.close();
}catch(FtpException fe){
    fe.printStackTrace();
}
}

public void ftpBatchRead(){
try{
    String ftppath = "ftp://192.168.0.1/home/log/1.log";

    FtpReadAdapter[] fras = new FtpReadAdapter[3];

    FtpAdapter fa0 = new FtpAdapter(ftppath);
    fras[0]=fa0.getFtpReader(0,5);

    FtpAdapter fa1 = new FtpAdapter(ftppath);
    fras[1]=fa1.getFtpReader(5,5);

    FtpAdapter fa2 = new FtpAdapter(ftppath);
    fras[2]=fa2.getFtpReader(10,5);

    Result<byte[]>[] rs = this.readAllBatch(fras);

    System.out.println(new String(rs[0].getResult()));
    System.out.println(new String(rs[1].getResult()));
    System.out.println(new String(rs[2].getResult()));

    fa0.close();
    fa1.close();
    fa2.close();
}catch(FtpException fe){
    fe.printStackTrace();
}
}

public void ftpBatchReadWrite(){
try{
    String readpath = "ftp://192.168.0.1/home/log/1.log";
    FtpReadAdapter[] fras = new FtpReadAdapter[3];
    FtpAdapter fa0 = new FtpAdapter(readpath);
    fras[0]=fa0.getFtpReader(0,5);
    FtpAdapter fa1 = new FtpAdapter(readpath);
    fras[1]=fa1.getFtpReader(5,5);
    FtpAdapter fa2 = new FtpAdapter(readpath);
    fras[2]=fa2.getFtpReader(10,5);
}
```

```

        String writepath = "ftp://192.168.0.1/home/log/2.log";
        FtpWriteAdapter[] fwas = new FtpWriteAdapter[3];
        FtpAdapter fav0 = new FtpAdapter(writepath);
        fwas[0]=fav0.getFtpWriter(0,5);
        FtpAdapter fav1 = new FtpAdapter(writepath);
        fwas[1]=fav1.getFtpWriter(5,5);
        FtpAdapter fav2 = new FtpAdapter(writepath);
        fwas[2]=fav2.getFtpWriter(10,5);

        Result<Integer>[] rs = this.readWriteBatch(fras,fwas);

        System.out.println(rs[0].getResult());
        System.out.println(rs[1].getResult());
        System.out.println(rs[2].getResult());

        fa0.close();
        fal.close();
        fa2.close();
        fav0.close();
        fav1.close();
        fav2.close();
    }catch(FtpException fe){
        fe.printStackTrace();
    }
}

public Result[] undo(Result[] rtarr){
    System.out.println("undo.....");
    for(int i=0;i<rtarr.length;i++){
        if(rtarr[i].getStatus()==Result.EXCEPTION)
            System.out.println("Result index"+i+" Error");
    }
    return rtarr;
}

public static void main(String[] args){
    FtpBatchWriteReadDemo fwrd = new FtpBatchWriteReadDemo ();
    fwrd.ftpBatchWrite();
    fwrd.ftpBatchRead();
    fwrd.ftpBatchReadWrite();
}
}

```

6.10 如何进行整型读写

我们在前面的小节里面也谈到了对整型的读取，接下来我们通过 `FileAdapter` 来介绍对本地整型数据的操作，以及整型列表对象的使用，并与 `java.util.ArrayList` 的性能进行对比。

与读取字节类似，在构建一个 FileAdapter 对象后，可以通过 getIntWriter 方法获取 IntWriteAdapter 对象进行整数写操作：

```
FileAdapter fa = new FileAdapter(path);
IntWriteAdapter wa = fa.getIntWriter();
```

或者指定开始位置，向后写入 intNum 个整数：

```
IntWriteAdapter wa = getIntWriter(long beginIndex, long intNum)
```

如果不指定起始位置，默认是追加到最后。注意，在使用时容易出现不追加到最后，而反复覆盖已有位置的情况，比如：

```
IntWriteAdapter wa = fa.getIntWriter();
int[] nums = new int[5];
wa.writeInt(nums);
wa.writeInt(nums);
wa.writeInt(nums);
```

上面代码的结果是 nums 没有追加到文件最后，而是反复覆盖相同位置，如果修改为：

```
IntWriteAdapter wa = fa.getIntWriter();
int[] nums = new int[5];
wa.writeInt(nums);
wa = fa.getIntWriter();
wa.writeInt(nums);
wa = fa.getIntWriter();
wa.writeInt(nums);
```

结果就每次追加到文件最后了，这是因为每次调用 fa.getIntWriter()，它会自动调整文件大小，计算文件的最后位置，如果沿用旧的 IntWriteAdapter，则没有获取到文件更新后的最后位置，导致反复覆盖相同位置。

同样，构建一个 FileAdapter 对象后，可以通过 getIntReader 方法获取 IntReadAdapter 对象进行整数读操作：

```
FileAdapter fa = new FileAdapter(path);
IntReadAdapter ra = fa.getIntReader();
```

或者指定开始位置读取 intNum 个整数：

```
IntReadAdapter ra = fa.getIntReader(long beginIndex, long intNum);
```

跟上面谈到的 FFTP 使用类似，操作结束时，需要进行关闭：

```
fa.close();
```

除了提供对整型数据的便利操作外，框架还提供了 ListInt 数组用于对整型数据的存放，ListInt 在容量和排序的性能上都比 java.util.ArrayList 要高，接下来我们以一个综合的 demo：ListIntTest 来演示对整型数据的读写、装载、排序，分别在运行时输入参数 0、1、2 进行区分。

下面介绍运行步骤。

1) 读写演示，代码如下：

```
java -cp fourinone.jar; ListIntTest 0
```

如果命令参数是 0，ListIntTest 调用了 writeReadTest，首先在同一运行目录下建立一个“data.txt”的数据文件，写入了 100 个整数（每个整数是随机生成的，取值范围在 1 千万内），然后再每 10 个一批地将所有整数读出并输出显示，如图 6-6 所示。

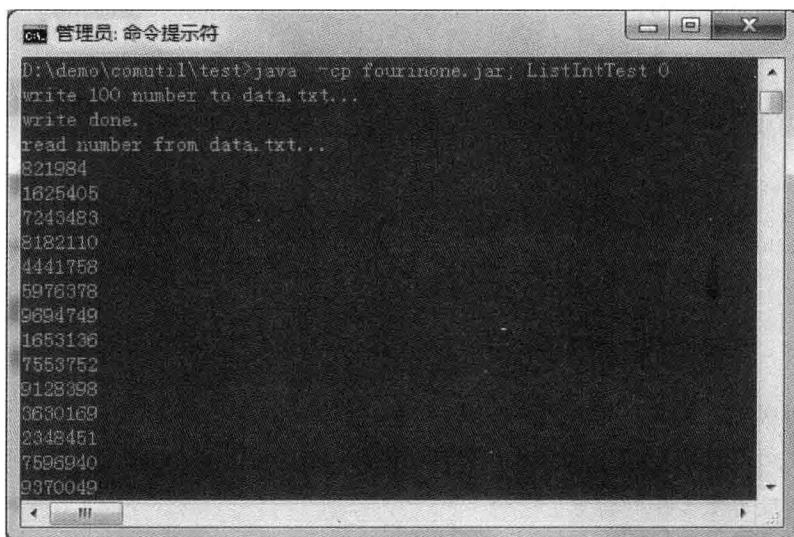


图 6-6 ListIntTest 0

2) 装载演示，代码如下：

```
java -cp fourinone.jar; ListIntTest 1
```

如果命令参数是 1，ListIntTest 调用了 capacityTest，它按先后顺序运行了 ListInt 和 ArrayList 的装载容量测试进行对比。

首先通过 ArrayAdapter 获取到一个 ListInt 实例：

```
ListInt ai = ArrayAdapter.getListInt();
```

然后往里面写入 2 千万个整数（取值范围为 1 千万内）。接着，再建立一个 ArrayList 的实例，同样往里面写入 2 千万个整数，为了能看清楚过程，每写入 1 千万后会输出一条信息。运行结果如图 6-7 所示。

```

Administrator: 命令提示符
D:\demo\comutil\test>java -cp fourinone.jar; ListIntTest 1
insert int into ListInt...
20000000 number be inserted.
insert int into ArrayList...
0 number be inserted.
10000000 number be inserted.
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.lang.Integer.valueOf(Unknown Source)
    at ListIntTest.arraylistCapacity(ListIntTest.java:85)
    at ListIntTest.capacityTest(ListIntTest.java:123)
    at ListIntTest.main(ListIntTest.java:142)

D:\demo\comutil\test>

```

图 6-7 ListIntTest 1

我们发现，向 ListInt 写入 2 千万个整数是可以顺利完成的，但是向 ArrayList 写入 2 千万个整数则导致内存溢出，查看内容输出可以看到，向 ArrayList 写入 1 千万个整数是正常完成的，但是再大就出现问题了（这里用的是一台普通的内存为 4GB 的 PC，JVM 的内存使用会小于 4GB）。

3) 排序演示，代码如下：

```
java -cp fourinone.jar; ListIntTest 2
```

如果命令参数是 2，ListIntTest 调用了 sortTest，此方法可以输入一个参数，用来表示要测试排序多少个整数，这里测试了 5 百万个数字排序。首先程序建立了一个“data.txt”文件并生成了 5 百万个无序整数（取值范围为 1 千万以内），然后先后使用 ListInt 和 ArrayList 进行排序。

对于 ListInt 排序，我们可以看到先从文件中读出所有整数到一个数组中，然后使用 ListInt 的 sort 方法，输入该数组进行排序，最后再输出前 100 个数字进行验证。对于 ArrayList 来说，先通过 readListIntAll 直接从文件读取所有整数到一个 ArrayList 中，然后再利用 Collections.sort 对该 ArrayList 进行排序，最后同样读取前 100 个数字进行验证。结果如图 6-8 所示。

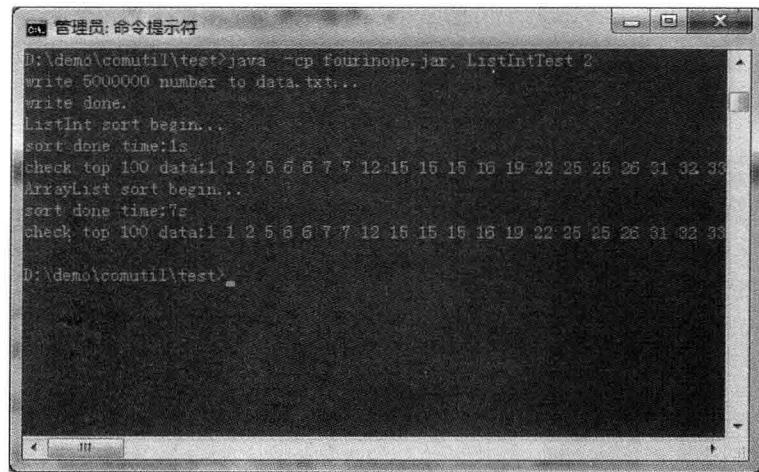


图 6-8 ListIntTest 2

我们可以看到，在同台（2.4GHz，4G 内存）机器上，利用 ListInt 完成 5 百万数字排序时间是 1 秒，而 ArrayList 则用了 7 秒。我们大体可以看到 ListInt 相对于 ArrayList，对于整数可以存储更多，排序更快，使用起来更高效。

完整 demo 源码如下：

```

// ListIntTest
import com.fourinone.FileAdapter;
import com.fourinone.FileAdapter.IntReadAdapter;
import com.fourinone.FileAdapter.IntWriteAdapter;
import com.fourinone.ArrayAdapter;
import com.fourinone.ArrayAdapter.ListInt;
import java.util.Random;
import java.util.ArrayList;
import java.util.Collections;

public class ListIntTest
{
    public void intWrite(String path, int num)
    {
        System.out.println("write "+num+" number to "+path+"...");
        FileAdapter fa = new FileAdapter(path);
        fa.delete();
        IntWriteAdapter wa = fa.getIntWriter();
        Random rad = new Random();
        int[] nums = new int[num];
        for(int i=0;i<nums.length;i++){
            int thenum = rad.nextInt(10000000);
            nums[i]=thenum;
        }
        wa.writeInt(nums);
        System.out.println("write done.");
    }
}

```

```
        fa.close();
    }

    public void intRead(String path)
    {
        System.out.println("read number from "+path+"...");
        FileAdapter fa = new FileAdapter(path);
        IntReadAdapter ra = null;
        int total=0;
        for(int n=0;n<10;n++){
            ra = fa.getIntReader(n*10,10);
            int[] its = ra.readIntAll();
            for(int i:its){
                System.out.println(i);
                total++;
            }
        }
        System.out.println("total:"+total);
        fa.close();
    }

    public void listintCapacity()
    {
        System.out.println("insert int into ListInt...");
        ListInt ai = ArrayAdapter.getListInt();
        Random rad = new Random();
        for(int i=0;i<20000000;i++)
            ai.add(rad.nextInt(10000000));

        System.out.println(ai.size()+" number be inserted.");
    }

    public void arraylistCapacity()
    {
        System.out.println("insert int into ArrayList...");
        ArrayList<Integer> al = new ArrayList<Integer>();
        Random rad = new Random();
        int i=0;
        while(i<20000001){
            al.add(rad.nextInt(10000000));
            if(i%10000000==0){
                System.out.println(i+" number be inserted.");
            }
            i++;
        }
    }

    public void listintSort()
    {
        FileAdapter fa = new FileAdapter("data.txt");
        int[] rls = fa.getIntReader().readIntAll();
        ListInt is = ArrayAdapter.getListInt();

        System.out.println("ListInt sort begin...");
        long begin = (new java.util.Date()).getTime();
```

```
    is.sort(rls);
    long end = (new java.util.Date()).getTime();
    System.out.println("sort done time:"+ (end-begin)/1000+"s");

    System.out.print("check top 100 data:");
    for(int i=0;i<100;i++)
    {
        System.out.print(rls[i]+" ");
    }
    System.out.println("...");
}

public void arrayListSort()
{
    FileAdapter fa = new FileAdapter("data.txt");
    ArrayList<Integer> rls = (ArrayList)fa.getIntReader().readListIntAll();

    System.out.println("ArrayList sort begin...");
    long begin = (new java.util.Date()).getTime();
    Collections.sort(rls);
    long end = (new java.util.Date()).getTime();
    System.out.println("sort done time:"+ (end-begin)/1000+"s");

    System.out.print("check top 100 data:");
    for(int i=0;i<100;i++)
    {
        System.out.print(rls.get(i)+" ");
    }
    System.out.println("...");
}

public static void writeReadTest()
{
    ListIntTest test = new ListIntTest();
    test.intWrite("data.txt",100);
    test.intRead("data.txt");
}

public static void capacityTest()
{
    ListIntTest test = new ListIntTest();
    test.listintCapacity();
    test.arraylistCapacity();
}

public static void sortTest(int num)
{
    ListIntTest test = new ListIntTest();
    test.intWrite("data.txt",num);
    test.listintSort();
    test.arraylistSort();
}

public static void main(String[] args)
{
```

```

        if(args[0].equals("0"))
        {
            writeReadTest();
        }
        else if(args[0].equals("1"))
        {
            capacityTest();
        }
        else if(args[0].equals("2"))
        {
            sortTest(5000000);
        }
    }
}

```

6.11 基于整型读写的上亿排序

在第2章我们讲述了一个上亿排序的例子，那是基于内存方式的，那么在真实的场景中，数据到达一定的量级后，仅使用内存方式是很难完成的，本节我们基于本章学习的文件和整型数据操作，来实现一个更接近真实场景的上亿数据排序的例子。

根据第2章上亿排序的例子中的分类规则，我们知道，每台计算机的内存有限，我们必须设置好一个最小处理单元，按照最小处理单元来分类、合并，最后在最小处理单元内排序，我们下面举例说明。

假设有一台计算机，上面有2500万数据，要将这2500万数据直接导入内存排序是不行的，因为内存太小，每次最多只能处理625万的数据排序，那么625万就是一个最小处理单元。我们需要将2500万数据按照取值范围分成4类，每类大约625万，然后逐个排序。如果数据的取值范围均匀地分布在0~10万内，那么可以按如下进行分类。

- 0~25000内的数据：大约625万，导入内存排序
- 25000~50000内的数据：大约625万，导入内存排序
- 50000~75000内的数据：大约625万，导入内存排序
- 75000~100000内的数据：大约625万，导入内存排序

虽然我们的内存有限，但是只要划分好数据分类，最后还是能利用有限的内存得到整体排序的结果。

上面的例子只是针对单台计算机的，如果我们有更多的数据，这些数据分布存储在多台计算机上，那么分类和排序过程就更加复杂了，比如有1亿数据分布存储在4台计算机上，每台计算机存储2500万，要求这1亿数据整体排序，那么按照第2章上亿排序的思路，需要进行如下分类拆分和合并：

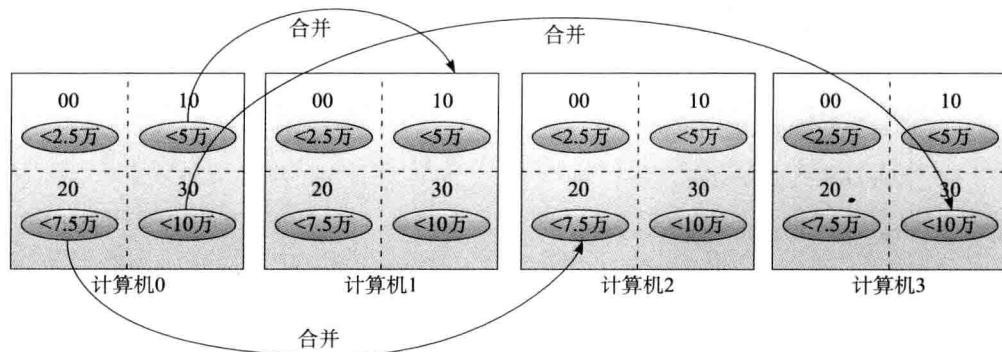


图 6-9 整型文件排序 1

从图 6-9 我们可以看到每台计算机将各自的 2500 万数据分成了 4 类，在实现上，分别将数据分到 4 个文件中去，依次是：

- 00 文件，装 0 ~ 25000 内的数据。
- 10 文件，装 25000 ~ 50000 内的数据。
- 20 文件，装 50000 ~ 75000 内的数据。
- 30 文件，装 75000 ~ 100000 内的数据。

然后我们可以看到，对于 00 文件，每台计算机都有一个，我们可以指定每台计算机负责收集一个分类，比如第 0 台计算机负责收集 00 文件（25000 内的数据），第 1 台计算机负责收集 10 文件（50000 内的数据）……为了达到收集目的，各计算机之间需要进行互相合并，将属于其他计算机的分类发给对方，过程如下：

发送 / 接收	计算机 0	计算机 1	计算机 2	计算机 3
计算机 0		发送 10 文件	发送 20 文件	发送 30 文件
计算机 1	发送 00 文件		发送 20 文件	发送 30 文件
计算机 2	发送 00 文件	发送 10 文件		发送 30 文件
计算机 3	发送 00 文件	发送 10 文件	发送 20 文件	

也就是每台计算机发送分类文件给相应的其他计算机时，同时也接收其他计算机发给属于自己的分类文件，并将文件追加到本地该分类文件中。该过程完成后，形成以下结果：

计算机 0	计算机 1	计算机 2	计算机 3
00 文件	10 文件	20 文件	30 文件

在完成合并后，每台计算机只剩下属于自己分类的文件，由于每个文件内的数据范围都是划分好的，此时再对每个文件进行内存排序就可以得到整体的排序数据了。

这里会有个问题，我们从最开始的每台计算机将数据分类成 4 个文件，到最后合并为 1

个文件，如果数据均匀分布，文件还是大约有 2500 万数据，按照我们之前的假设，每台计算机的内存最多只能处理 625 万的数据排序，那么对于 2500 万的大数据文件，仍然无法将其加载到内存中一次性完成。

这个时候又涉及之前我们提到的最小处理单元的问题了，现在的最小处理单元还是 2500 万，如果需要降低到 625 万，我们要重新对数据进行分类，如图 6-10 所示。

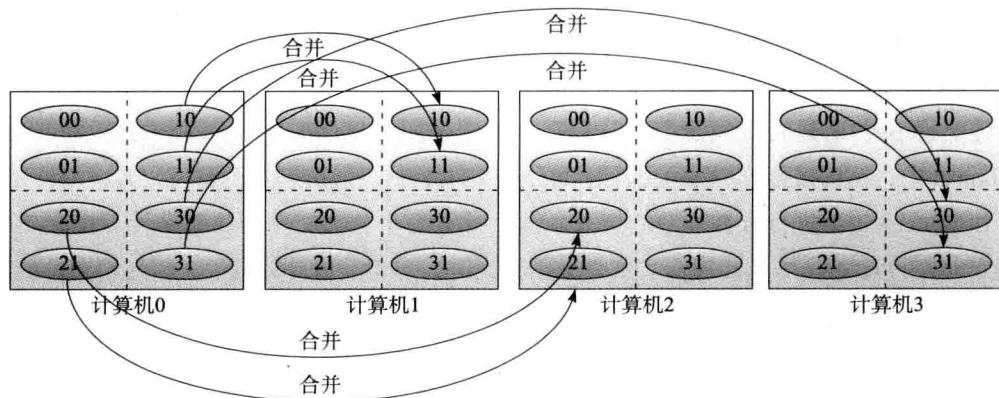


图 6-10 整型文件排序 2

如果我们将每台计算机上 2500 万数据分成 8 类，计算机 0 负责收集 00、01 分类文件、计算机 1 负责收集 10、11 文件……然后互相合并完成，最后每台计算机上剩下 2 个分类文件，每类文件有 1250 万数据，降低了一倍最小处理单元。按照该思路，如果将每台计算机上 2500 万数据分类 16 类，那最后每台计算机就剩下 4 个分类文件，每个文件有 625 万数据，就刚好达到我们的内存能接受的大小了。

但是我们要注意，分类越多意味着合并的过程越长，耗用网络发送文件越多，对排序性能会有影响，因此要在内存处理最小单元和分类数量之间权衡尝试，以达到性能最优化。

下面的 demo 完整演示了以上介绍的基于文件方式排序的思路。

- SortFileData：负责原始数据的生成和排序结果的验证，creatData 方法按照每 50 万一批生成 10 000 000 范围内的随机整数到指定数据文件中，累计生成 4 个各自包含 2500 万整数的数据文件，用于模拟排序计算。在完成了排序后，可以通过 checkData 抽取前 100 个数字进行结果验证。
- SortFileCtor：是一个工头实现，它先后发出“分类”、“合并”、“排序”三个环节的调度命令，指挥工人完成整个排序过程，在每个环节都使用 doTaskBatch 的栅栏机制，等待该环节结束再进行下一步，最后输出整体结果和耗用时间。
- SortFileWorker：是一个工人实现，相应于工头的三个环节调度命令进行实现，

SortFileWorker 的构造函数有 4 个重要参数：

```
SortFileWorker(int n, int max, int every, String path)
```

其中 4 个参数依次代表 "分成多少类、随机数据范围、每次处理多少数据、输入数据文件"，这 4 个参数会影响到排序过程中的相关计算。除此外，当前计算机在集群中的位置也是重要的计算参数，在程序中是通过 index = getSelfIndex(); 获取的。

在 doTask 实现中首先获取 int step = (Integer)wh.getObj("step");，得到工头调度命令，然后判断 step 的值进行每个环节的任务处理。

对于 step==1，进行分类环节。从数据文件中按照每批读取 every 数量的整数，先放到 warehouse 中，然后根据分类写入到不同的分类文件中去，完成分类环节。

对于 step==2，进行合并环节。合并环节会计算每个分类文件所属哪台计算机，然后调用对方工人的 receive 方法传递分类文件数据，同时，对于其他计算机工人传递给自己的文件数据，在 receive 方法中进行实现，将接收到的数据追加写入到本地文件中去。

对于 step==3，进行排序环节。将本地的分类文件读到 ListInt 中进行内存排序，然后再写回结果到分类文件中。最后统计当前工人完成的排序总数并返回给工头。

下面介绍具体的运行步骤。

1) 启动 ParkServerDemo：

```
java -cp fourinone.jar; ParkServerDemo
```

2) 启动 SortFileDialog，结果如图 6-11 所示。

```
java -cp fourinone.jar; SortFileDialog
```

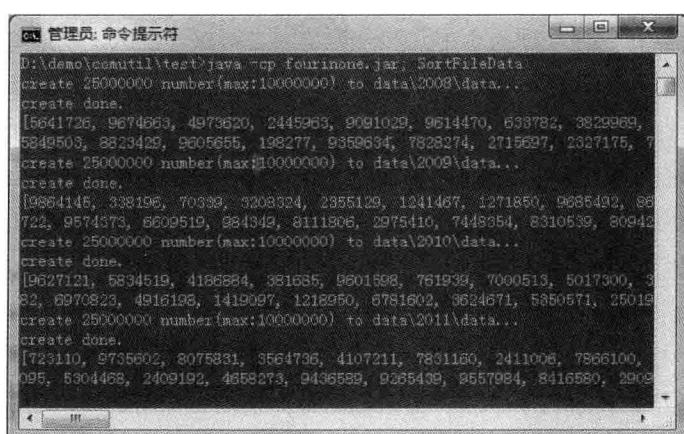


图 6-11 启动 SortFileDialog

SortFileData 在运行 data 的目录下建立了 4 个文件夹 2008、2009、2010、2011，每个文件夹下面创建了一个 data 文件，写入 2500 万整型数据。然后验证这些数据是随机无序的。

3) 启动 SortFileWorker:

```
java -cp fourinone.jar; SortFileWorker localhost 2008 4 10000000 500000
      data\\2008\\data
java -cp fourinone.jar; SortFileWorker localhost 2009 4 10000000 500000
      data\\2009\\data
java -cp fourinone.jar; SortFileWorker localhost 2010 4 10000000 500000
      data\\2010\\data
java -cp fourinone.jar; SortFileWorker localhost 2011 4 10000000 500000
      data\\2011\\data
```

我们启动了 4 个工人实例，为了方便在本地演示，使用 localhost 和不同端口号区分，分别占用 2008、2009、2010、2011 端口，跟上面建立的各数据文件对应，除此之外，“4 10000000 500000”这三个运行传入的参数，分别代表了“分类数、数据取值范围、每次处理多少数据”，用来构造 SortFileWorker。启动完成后如图 6-12 所示。



图 6-12 启动 SortFileWorker

4) 启动 SortFileCtor:

```
java -cp fourinone.jar; SortFileCtor
```

工头运行后，调度每个工人进行并行计算，可以看到当完成三个环节步骤之后，各工人窗口输出了排序的过程，如图 6-13 所示。



图 6-13 SortFileWorker 结果

同时，工头窗口也输出了最后的统计和整体时间，1亿个整数排序时间大约在 37 秒，用了单机 4 个工人实例，普通 pc 配置（2.4GHz CPU，4G 内存），如图 6-14 所示。

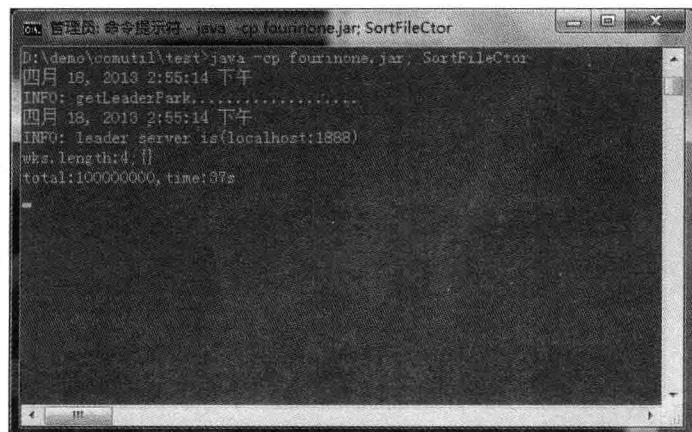


图 6-14 运行 SortFileCtor

我们再打开数据存放目录，发现每个工人目录的 output 目录下都有一个排序结果的数据文件，我们抽取 4 个工人目录下排序结果的前 100 条数据验证，如图 6-15 所示

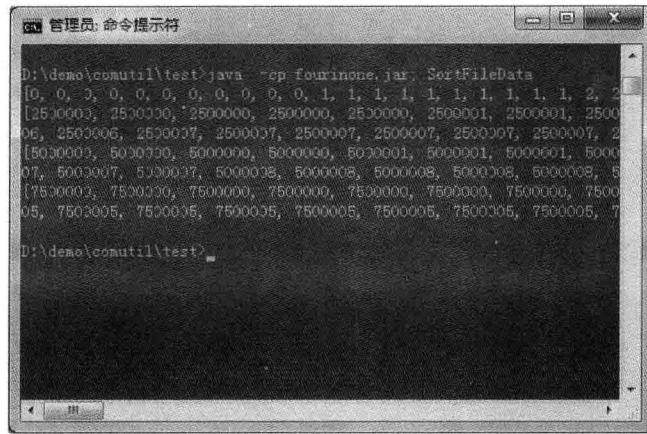


图 6-15 排序结果验证

可以看到，原始 1 亿条整型数据，按照 $(0, 2500000, 5000000, 7500000, 10000000)$ 的区间被拆分成 4 个文件 $(00, 10, 20, 30)$ 存放，每个文件内部都是区间内排序好的，4 个文件合起来，就是完整的 1 亿条整型数据排序结果。

前面我们分析过，如果内存小，可以将分类分的更细降低最小处理单元大小，下面我们将看看分类数为 8 的情况。

1) 启动 SortFileWorker:

```
java -cp fourinone.jar; SortFileWorker localhost 2008 8 10000000 500000  
data\\2008\\data

java -cp fourinone.jar; SortFileWorker localhost 2009 8 10000000 500000  
data\\2009\\data

java -cp fourinone.jar; SortFileWorker localhost 2010 8 10000000 500000  
data\\2010\\data

java -cp fourinone.jar; SortFileWorker localhost 2011 8 10000000 500000  
data\\2011\\data
```

除了第3个参数从“4”改为“8”外，其他参数不变，然后我们再运行一下工头执行计算。

2) 启动 SortFileCtor:

```
java -cp fourinone.jar; SortFileCtor
```

可以查看到各工人窗口输出的排序过程，如图 6-16 所示。



图 6-16 8个分类工人运行结果

我们发现分类数为8后，工人之间合并的环节耗时更多，因为生成了更多分类文件，需要更多的文件合并传送。

同时我们查看工头的窗口输出，如图 6-17 所示。

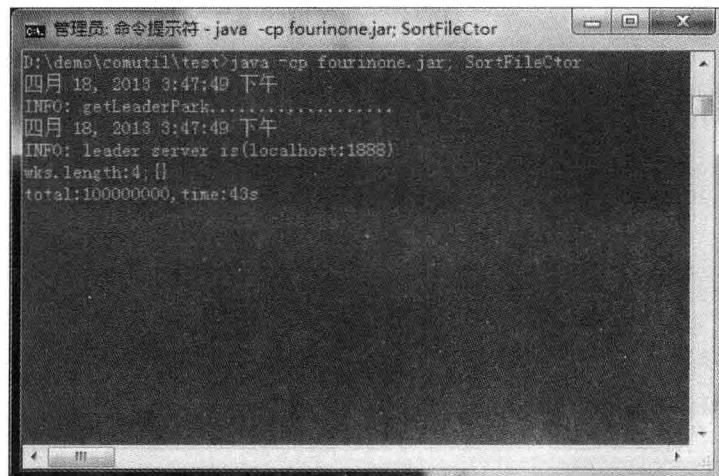


图 6-17 8个分类工头运行结果

发现整体的排序时间为 43 秒，比起之前 4 个分类的 37 秒要长，主要是合并环节更加耗用时间。因此，虽然分类多会减轻内存的负担，降低内存最小排序处理单元大小，但是也会耗用更多的时间在文件网络传送和文件 IO 读写上。

我们打开 output 结果目录，发现每个目录下有两个分类的结果文件，我们抽取各个文件夹下面两个分类文件的前 100 条验证结果，如图 6-18 所示。

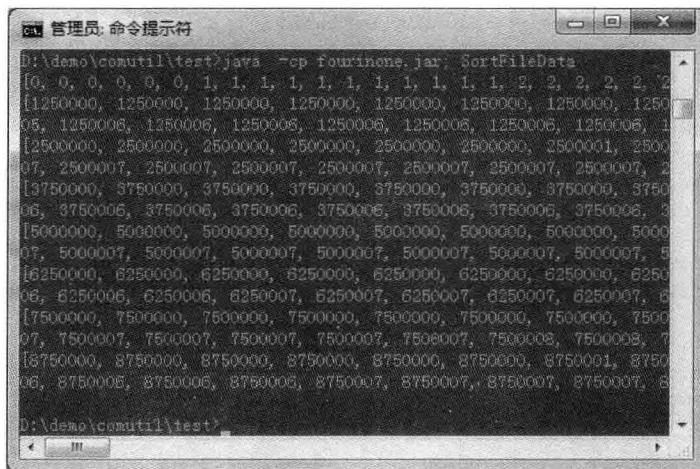


图 6-18 8 个分类排序的结果验证

我们发现分类从 4 个区间变成了 8 个区间：(0, 1250000, 2500000, 3750000, 5000000, 6250000, 7500000, 8750000, 10000000)，每个区间的数据量也减少了将近一倍，这样更有利于我们做分布式存储和计算。

不仅是分类数可以根据内存大小调整，也可以调整工人数量，采取多机多实例 $n \times n$ 的方式调整并观察性能效果，最后得到一个最优的排序计算参数。

完整 demo 源码如下：

```
// ParkServerDemo
import com.fourinone.BeanContext;
public class ParkServerDemo
{
    public static void main(String[] args)
    {
        BeanContext.startPark();
    }
}

// SortFileData
import com.fourinone.FileAdapter;
import com.fourinone.FileAdapter.IntReadAdapter;
import com.fourinone.FileAdapter.IntWriteAdapter;
```

```

import java.util.Random;
import java.util.List;

public class SortFileData
{
    public static void creatData(int total, int max, String path)
    {
        System.out.println("create "+total+" number(max:"+max+") to "+path+"...");
        int every = 500000;
        FileAdapter fa = new FileAdapter(path);
        fa.delete();
        Random rad = new Random();
        while(total>0){
            IntWriteAdapter wa = fa.getIntWriter();
            int[] nums = new int[total-every<0?total:every];
            for(int i=0;i<nums.length;i++){
                nums[i]=rad.nextInt(max);
            }
            wa.writeInt(nums);
            total-=nums.length;
        }
        System.out.println("create done.");
        fa.close();
    }

    public static void checkData(String path)
    {
        FileAdapter fa = new FileAdapter(path);
        List<Integer> rls = fa.getIntReader(0,100).readListIntAll();
        System.out.println(rls+"...");
    }

    public static void main(String[] args)
    {
        creatData(25000000,10000000,"data\\2008\\data");
        checkData("data\\2008\\data");
        creatData(25000000,10000000,"data\\2009\\data");
        checkData("data\\2009\\data");
        creatData(25000000,10000000,"data\\2010\\data");
        checkData("data\\2010\\data");
        creatData(25000000,10000000,"data\\2011\\data");
        checkData("data\\2011\\data");
    }
}

// SortFileWorker
import com.fourinone.MigrantWorker;
import com.fourinone.WareHouse;
import com.fourinone.Workman;
import com.fourinone.FileAdapter;
import com.fourinone.FileAdapter.ReadAdapter;
import com.fourinone.FileAdapter.WriteAdapter;
import com.fourinone.FileAdapter.IntReadAdapter;

```

```
import com.fourinone.FileAdapter.IntWriteAdapter;
import com.fourinone.ArrayAdapter;
import com.fourinone.ArrayAdapter.ListInt;
import java.io.File;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;

public class SortFileWorker extends MigrantWorker
{
    private int n=-1,max=-1,every=-1,m=-1,index=-1;
    private String path;
    private HashMap<Integer,List<Integer>> wharr=new HashMap<Integer,List
        <Integer>>();
    private Random rad = new Random();
    private Workman[] wms = null;
    private FileAdapter[] faws = null;

    public SortFileWorker(int n, int max, int every, String path){
        this.n = n;
        this.max = max;
        this.every = every;
        this.path = path;
        faws = new FileAdapter[n];
    }

    public WareHouse doTask(WareHouse wh){
        try{
            int step = (Integer)wh.getObj("step");
            if(wms==null){
                wms = getWorkerAll();
                m = wms.length;
                for(int l=0;l<n;l++){
                    faws[l] = new FileAdapter(new File(path).getParent()+"\\\
                        output\\\"+l*m/n+l%(n/m)); // "output/" + l*m/n+l%(n/m)
                    faws[l].delete();
                }
            }
            index = getSelfIndex();
            System.out.println("wknum:"+m+";step:"+step);
            WareHouse resultWh = new WareHouse("ok",1);

            if(step==1){
                FileAdapter fa = new FileAdapter(path);
                IntReadAdapter ira = null;
                int begin = 0;
                int[] rls = null;
                while(true){
                    ira = fa.getIntReader(begin,every);
                    rls = ira.readIntAll();
                    if(rls!=null){
                        for(int i:rls){
                            Integer numi = (int)(new Long(i)*new Long(n)/new Long(max));
                            wharr.put(i,numi);
                        }
                    }
                }
            }
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

```

        List<Integer> arr = wharr.get(numi);
        if(arr==null)
            arr = new ArrayList<Integer>();
        arr.add(i);
        wharr.put(numi, arr);
    }
    for(int j=0;j<n;j++){
        if(wharr.containsKey(j))
            faws[j].getIntWriter().writeListInt(wharr.remove(j));
    }
    begin+=rls.length;
}else break;
}
fa.close();
}else if(step==2){
    for(int j=0;j<n/m;j++) {
        for(int i=0;i<m;i++) {
            if(i!=index){
                if(faws[i*n/m+j].exists()){
                    int[] itsn = faws[i*n/m+j].getIntReader().readIntAll();
                    Workman wm = wms[i];
                    WareHouse whij = new WareHouse();
                    whij.put("i",i);
                    whij.put("j",j);
                    whij.put("v",itsn);
                    System.out.println(i+"-receive:"+wm.receive(whij));
                    faws[i*n/m+j].close();
                }
            }
        }
    }
}else if(step==3){
    int[] arrl = null;
    int total = 0;
    for(int j=0;j<n/m;j++) {
        if(faws[index*n/m+j].exists()){
            arrl = faws[index*n/m+j].getIntReader().readIntAll();
            ListInt is = ArrayAdapter.getListInt();
            is.sort(arrl);
            total+=arrl.length;
            faws[index*n/m+j].getIntWriter(0,arrl.length).writeInt(arrl);
            faws[index*n/m+j].close();
        }
        for(int i=0;i<m;i++)
            if(i!=index&&faws[i*n/m+j].exists())
                faws[i*n/m+j].delete();
    }
    resultWh.setObj("total",total);
    System.out.println("over.");
}
return resultWh;
}catch(Exception ex) {

```

```
        System.out.println(ex);
        return null;
    }
}

protected boolean receive(WareHouse inhouse)
{
    Integer i = (Integer)inhouse.get("i");
    Integer j = (Integer)inhouse.get("j");
    int[] v = (int[])inhouse.get("v");
    faws[i*n/m+j].getIntWriter().writeInt(v);
    return true;
}

public static void main(String[] args)
{
    SortFileWorker mw = new SortFileWorker(Integer.parseInt(args[2]),Integer.
    parseInt(args[3]),Integer.parseInt(args[4]),args[5]);
    mw.waitWorking(args[0],Integer.parseInt(args[1]),"SortWorker");
}
}

// SortFileCtor
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.WorkerLocal;
import java.util.Date;

public class SortFileCtor extends Contractor
{
    public WareHouse giveTask(WareHouse wh)
    {
        WorkerLocal[] wks = getWaitingWorkers("SortWorker");
        System.out.println("wks.length:"+wks.length+";"+wh);
        int total=0;

        wh.setObj("step", 1);//1:group;
        doTaskBatch(wks, wh);

        wh.setObj("step", 2);//2:merge;
        doTaskBatch(wks, wh);

        wh.setObj("step", 3);//3:sort
        WareHouse[] hmarr = doTaskBatch(wks, wh);
        for(int i=0;i<hmarr.length;i++){
            Object num = hmarr[i].getObj("total");
            if(num!=null)
                total+=(Integer)num;
        }

        wh.setObj("total",total);
        return wh;
    }
}
```

```
public static void main(String[] args)
{
    Contractor a = new SortFileCtor();
    WareHouse wh = new WareHouse();
    long begin = (new Date()).getTime();
    a.doProject(wh);
    long end = (new Date()).getTime();

    System.out.println("total:"+wh.getObj("total")+",time:"+ (end-
        begin)/1000+"s");
}
}
```

第 7 章

分布式作业调度平台的实现

在前面第 2 章介绍的分布式并行计算原理中，我们看到可以将工头工人程序部署到不同的机器上运行，然后完成计算。但是，当这样的并行计算应用非常多，而我们的机器数量有限时，我们需要排队依次来使用机器，第一个并行计算应用跑完了，再跑第二个应用，如果第一个并行计算应用只占部分机器，那么可以考虑同时跑第二个应用……这就涉及建立一个作业调度平台去完成。

我们知道 Hadoop 本身包含了作业调度的部分，按照 Hadoop 开发规范，开发好一个作业，打包后，Hadoop 会分发到相应的机器上去运行，但是实际上这仅仅是一种任务调度层面的实现，我们还需要资源层面的调度，比如对 CPU、内存、带宽等的分配和管理。

本章会讲述调度平台的设计和实现，包括任务调度和资源调度的实现机制，各种资源调度算法，并以 MPI 调度器为例讲述一个完整 Demo，最后再讲述市场上常用的调度实现和框架，比如 Torque、Mesos、Yarn 等。

7.1 调度平台的设计与实现

如果要设计实现一个简单的作业调度器，应该怎么做呢？

我们需要这个调度器提供机器资源的管理功能，可以支持上传任务包，任务包里包含并行计算的程序和资源请求配置，比如需要多少台机器来运行，然后用调度器来判断是否有足够的机器资源来启动，当然还需要一个存放任务包的队列，等等。

我们初步想了想，得到一个调度平台的架构，如图 7-1 所示。

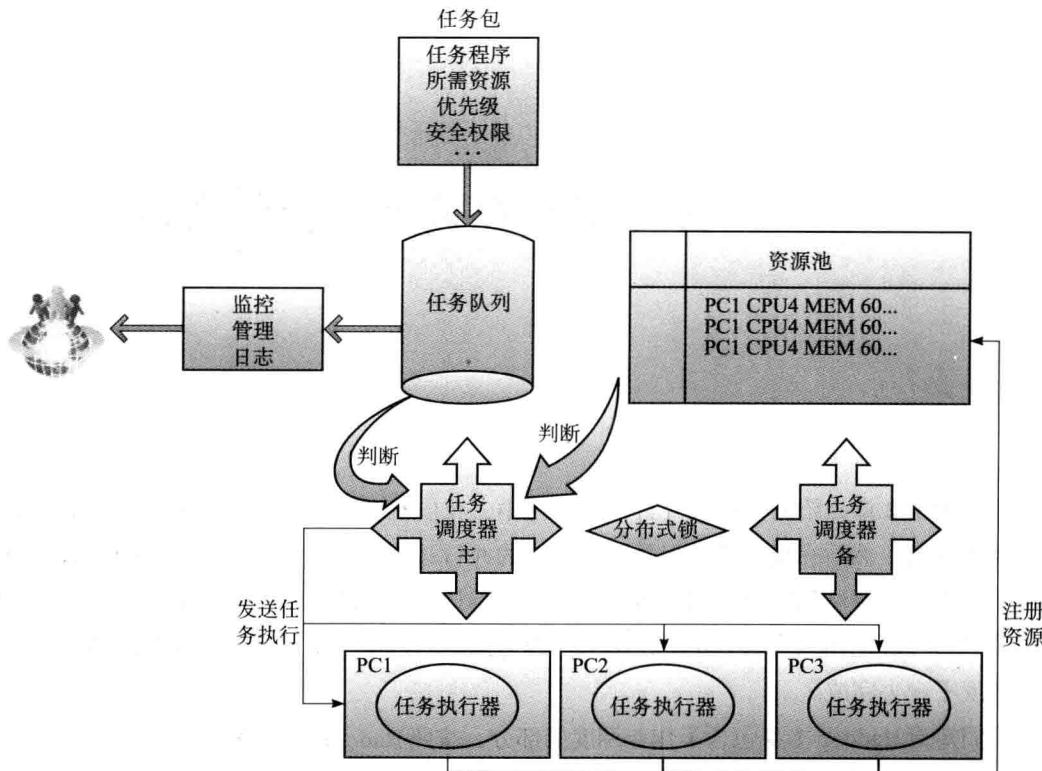


图 7-1 MPI 调度器

我们设想这个设计结构大致由任务包、任务队列、资源池、任务调度器、任务执行器、分布式锁、监控管理等几个核心组件组成，能基本解决任务调度和资源调度问题，同时它不仅支持命令方式操作，也是可编程的，这样更灵活。

组件说明如下：

1) “任务包”是一个任务运行压缩包，包括了完整的任务运行程序、配置属性、优先级设置、安全权限等运行时需要的信息，可以定义任务运行所需要的资源，比如启动该任务程序，最少要占用多少 CPU 和内存。由开发者按照“任务包”的规范开发打包好，然后提交给“任务队列”等待执行，提交的方式可以是命令行也可以是管理界面。

2) “任务队列”是一个分布式消息队列的实现，它主要用于存放任务包，以排队的方式等待获取资源进行执行，而不是全部任务同时运行争抢资源，任务队列一般是先进先出，也可以根据优先级配置信息调整策略。

3) “监控管理”是建立在任务队列基础之上的一套界面系统，可以通过它查看队列里任务的数量、等待状况、执行状况，并且可以取消任务和查看任务输出日志，支持命令行和界

面操作，同时对任务执行的关键环节（如完成或者取消中止）可以发出事件响应。

4) “资源池”是一个记录集群计算机资源实时信息的组件，它记录了集群可用计算机的CPU，内存的可用清单，并根据使用情况更新最新信息，资源池可以放在内存里，也可以持久保存在文件数据里。

5) “任务调度器”是一个一直运行并轮循检测是否未执行任务并有足够的资源运行。它会判断任务队列和资源池两个条件都满足的情况下，获取任务队列里的任务包发送到指定资源的机器上运行，并且根据任务包里的运行属性配置，比如超时检测，如果正常运行完成或者超时，都会释放所占有的资源，并且更新资源池，并标记任务队列里的队列属性状态（已完成或者中止等）或者删除。“任务调度器”从“资源池”获取到的资源有可能是多台机器的，也就是将任务包发送到多台机器并行执行，并且实时检查并行执行状况，这和Torque的方式完全不同。

6) “分布式锁”，由于“任务调度器”是一个不停止运行的重要组件，所以为了避免单点故障，它是一个主备关系，同时有“任务调度器-主”和“任务调度器-备”同时竞争一个分布式锁，竞争到者为主，未竞争到者为备进行替补等待，一旦“任务调度器-主”出故障，“任务调度器-备”马上抢到分布式锁并继续进行调度检测工作。

7) “任务执行器”负责PC机器资源的注册和资源隔离，以及在隔离的资源内执行任务。它首先会获取所在计算机的CPU可用核数、内存可用数等资源，发送到资源池进行注册，初始化资源池里的信息。当“任务调度器”发送任务到“任务执行器”执行时，并告诉所需要资源，“任务执行器”使用操作系统或者虚拟机的资源隔离技术，比如Linux的lxc和cgroup，进行资源的隔离，并且在该分配的资源下执行任务，同时将任务的执行状况实时汇报给“任务调度器”。当任务开始执行时，它会更新资源池的该资源占用信息，当任务结束释放资源时，它也会更新最新的资源信息。

设计完成后，我们再考虑一下如何落地实现，根据我们前面掌握的分布式实现技术，借助Fourinone大部分都可以实现了，例如：

- 任务队列使用消息队列实现，资源池使用缓存实现。
- 调度器根据任务队列和资源池条件，根据调度算法进行调度。
- 分布式锁采用分布式协调功能实现，任务执行采用自动部署实现。

不清楚的地方可以回过头查看前面章节相应的技术内容和Demo。

我们在前面的2.3节里面介绍过MPI，接下来我们基于上面的调度平台设计动手做一个MPI调度器，代码全部在MpiScheduler.java里。

实现说明：由于MPI的运行都是通过mpirun脚本命令进行的，一般会封装成一个sh的脚本文件，用户将sh脚本提交到MpiScheduler上，由MpiScheduler通过判断“机器列表资

源池”配置和“任务队列”配置两个条件，计算出是否进行该计算单元的调度，如果机器资源不足或者没有任务则不调度，否则调用 sh 脚本执行，根据脚本返回值获取调度是否成功，完成后释放机器资源信息，MpiScheduler 更新“机器列表资源池”。如果 MpiScheduler 接到新的提交作业的命令，没有足够的机器资源，则放入“任务队列”中等待。如果执行任务超时，则调用杀死 MPI 进程的脚本进行停止，并返回释放机器资源更新“机器列表”。

这样初步实现了作业提交，队列管理，机器资源分配和管理，MPI 的 sh 脚本调度，调度结果检查等功能，代码里有详细注释，所用到统一配置，队列，分布式锁等分布式基础功能，前面有相关功能指南和 demo。

运行步骤如下：

1) 启动配置和队列服务：

```
java -cp fourinone.jar: ParkServerDemo
```

2) 启动调度器或提交：

```
job: java -cp fourinone.jar: MpiScheduler 0 /home/mpi/ mpi.sh param 40 60
```

整体代码如下：

```
//MpiScheduler.java
import com.fourinone.Contractor;
import com.fourinone.WareHouse;
import com.fourinone.BeanContext;
import com.fourinone.ParkLocal;
import com.fourinone.ObjectBean;
import com.fourinone.StartResult;
import com.fourinone.FileAdapter;
import java.util.List;
import java.util.ArrayList;
import java.io.Serializable;

public class MpiScheduler extends Contractor
{
    private static boolean runFlag = false;
    private static ParkLocal pl = BeanContext.getPark(); // 配置和队列管理

    public WareHouse giveTask(WareHouse inhouse)
    {
        // 初始化 mpi 机器列表配置
        pl.create("mpi", "0", new Integer(50));
        pl.create("mpi", "1", new Integer(100));
        pl.create("mpi", "2", new Integer(50));

        ArrayList<StartResult<Integer>> rsarr = new ArrayList<StartResult
<Integer>>(); // 调度结果数组
        while(true){
            // 根据队列是否存在作业和是否有足够机器判断调度
        }
    }
}
```

```

for(int i=0;i<3;i++){
    ObjectBean job = receive(i+"");// 不阻塞接收队列，获取到作业对象
    if(job!=null){
        WareHouse jobobj = (WareHouse)job.toObject();
        int num = jobobj.getStringInt("computnum");// 获取申请机器数
        Integer iob = (Integer)(pl.get("mpi",i+"").toObject());// 获取配置列表里
        剩下的机器数量
        if(iob>=num){
            System.out.print(i+"type spent "+num+" computer to doTask");
            pl.delete(job.getDomain(), job.getNode());// 删除队列
            pl.update("mpi", i+"", new Integer(iob-num));// 更新机器列表
            System.out.println(", and remain "+(iob-num));
            //FileAdapter.createTempFile
            // 调度 job 对象里的 sh 脚本
            StartResult<Integer> res = BeanContext.tryStart(new FileAdapter
                (jobobj.getString("shdir"),"sh", jobobj.getString("sh"),
                 jobobj.getString("param"), jobobj.getString("computnum"),
                 ">>log/"+i+".log", "2>&l");
            res.setObj("job",jobobj);
            rsarr.add(res);// 将调度结果添加到结果数组
        } //else System.out.println(i+"type remain "+iob+" is not enough
          for "+num);
    }
}

// 检查结果数组是否完成调度
ArrayList<StartResult<Integer>> rmvrs = new ArrayList<StartResult<Integer
>>();// 记录完成的结果
for(StartResult<Integer> rs:rsarr){
    WareHouse jobwh = (WareHouse)rs.getObj("job");
    int timeout = jobwh.getStringInt("timeout");

    if(rs!=null&&rs.getStatus(StartResult.s(timeout))!=StartResult.
        NOTREADY){// 检查结果是否完成或超时
        System.out.print("Result:"+rs.getResult());
        Integer iob = (Integer)(pl.get("mpi",jobwh.getString("mpiType")).
            toObject());
        Integer newiob = iob+jobwh.getStringInt("computnum");
        pl.update("mpi", jobwh.getString("mpiType"), newiob);
        System.out.println(", and remain "+newiob+" "+jobwh);
        rmvrs.add(rs);
    }
}
rsarr.removeAll(rmvrs); // 完成从结果数组删除
}

// 发送到队列
public static void send(String queue, Object obj){
    pl.create(queue, (Serializable)obj);
}

// 从队列接收
public static ObjectBean receive(String queue){
    ObjectBean ob=null;
    List<ObjectBean> oblist = pl.get(queue);
}

```

```

        if(oblist!=null)
            ob = oblist.get(0);
        return ob;
    }

    public static void main(String[] args)
    {
        WareHouse jobwh = new WareHouse();
        jobwh.setString("mpiType",args[0]); //mpi 计算类型
        jobwh.setString("shdir",args[1]); //sh 脚本运行目录
        jobwh.setString("sh",args[2]); //sh 脚本名称
        jobwh.setString("param",args[3]); //sh 脚本参数
        jobwh.setString("computnum",args[4]); //申请计算机数量
        jobwh.setString("timeout",args[5]); //超时时间，单位为秒
        send(args[0],jobwh); //提交到队列
        System.out.println("submit job to queue:"+jobwh);

        // 判断锁是否作为调度服务，不作为调度服务提交 job 后直接退出
        ObjectBean oblock = pl.getBean("mpi","lock");
        if(oblock==null){
            pl.create("mpi", "lock", true, true);
            MpiScheduler a = new MpiScheduler();
            a.giveTask(null);
        }
    }
}

```

关于 MPI 调度在 7.4.1 节“其他 MPI 作业资源调度技术”我们还会介绍市场上基于 Torque 等任务管理工具方式的做法。

7.2 资源隔离的实现

我们如何让计算任务的进程能受限制的使用机器资源呢，这里不仅指占用某台机器的粗粒度使用，也指对某台机器的 CPU/ 内存 / 带宽 / 硬盘的细粒度限制。

资源限制和资源控制看上去差不多，但是我们接下来会发现他们的区别。

资源调度有两种解决方案：限制方式和控制方式。

所谓限制方式，也就是有个虚拟的容器限制，进程无法使用更多的资源。

Lxc、Cgroup 的实现是一种限制方式，通过虚拟化或者轻量级虚拟化限制资源的使用，后面 7.4.2 谈到的 Mesos 也是这种方式。

Cgroup 是 Linux 内核提供的一个操作系统层面的资源控制组，通过对进程组（process groups）的资源进行分配和使用限制达到目的。该项技术最初也是 Google 工程师发明，目前成为 lxc(轻量级虚拟化容器) 的重要组成技术。

Cgroup 提供了对进程组可以使用的资源数量限制、优先级控制、进程组使用的资源数量记录（进程组使用的 CPU 时间）、进程组隔离、进程组挂起和恢复等功能。

Cgroup 的一些主要子系统如下：

- ❑ Blkio：这个子系统为块设备设定输入 / 输出限制，比如物理设备（磁盘、固态硬盘、USB 等等）。
- ❑ Cpu：这个子系统使用调度程序提供对 CPU 的 Cgroup 任务访问。
- ❑ Cpuacct：这个子系统自动生成 Cgroup 中任务所使用的 CPU 报告。
- ❑ Cpuset：这个子系统为 Cgroup 中的任务分配独立 CPU（在多核系统）和内存节点。
- ❑ Devices：这个子系统可允许或者拒绝 Cgroup 中的任务访问设备。
- ❑ Freezer：这个子系统挂起或者恢复 Cgroup 中的任务。
- ❑ memory：这个子系统设定 Cgroup 中任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。
- ❑ net_cls：这个子系统使用等级识别符（Classid）标记网络数据包，可允许 Linux 流量控制程序（tc）识别从具体 Cgroup 中生成的数据包。
- ❑ Ns：名称空间子系统。
- ❑ Cgroup 启动是通过 Cgconfig 完成的：

```
#/etc/init.d/cgconfig start
#/etc/init.d/cgconfig stop
```

显示已经存在的子系统：

```
cpu /cgroup/cpu
```

创建一个新的 Cgroup：

```
cgcreate -g cpu,net_cls:/mygroup
```

通过 cgclassify 命令将进程移到 Cgroup 中：

```
cgclassify -g cpu,memory:group1 1701
```

或者通过 cgexec 命令在 Cgroup 中启动进程：

```
cgexec -g cpu:group1 lynx
```

Cgroup 详细的使用可参见 Linux 官方文档 <http://www.redhat.com>。

另一种是控制方式，进程实际上可以用更多的资源，但是我们通过监控的手段，根据资源使用状况，控制住进程的启动。

通过 JVM 或者其他系统监控 API，对 CPU 内存使用情况进行获取并控制，早期的 Hadoop 对内存的控制也是该方式。

Java 的 MemoryPoolMXBean 机制可以很好地设置使用内存的阀值，并在内存不足时进行事件通知，并将任务分配到其他 Java 虚拟机，然后停止接受任务，内存足够后再重新恢复接收任务。这个机制是可以用来实现内存隔离的（不能像 Cgroup 那样隔离，但是可以做到控制）。另外，OperatingSystemMXBean 机制可以支持线程和 Java 虚拟机进行 CPU 时间检测（包括用户 CPU 时间和系统 CPU 时间），通过 CPU 检测可以对占用率高的时刻停止接收任务。

遗憾的是，Java 本身对系统层面的监控力度支持还是不太理想，没有太好的做法。很多老外推荐使用开源软件 Sigar（System Information Gatherer And Reporter）来弥补 Java 本身在这块的不足，Sigar 提供了跨平台的系统信息收集的 API，包括：

- CPU 信息，包括基本信息（vendor、model、mhz、cacheSize）和统计信息（user、sys、idle、nice、wait）。
- 文件系统信息，包括 Filesystem、Size、Used、Avail、Use%、Type。
- 事件信息，类似 Service Control Manager。
- 内存信息，物理内存和交换内存的总数、使用数、剩余数；RAM 的大小。
- 网络信息，包括网络接口信息和网络路由信息。
- 进程信息，包括每个进程的内存、CPU 占用数、状态、参数、句柄。
- IO 信息，包括 IO 的状态，读写大小等。
- 服务状态信息。
- 系统信息，包括操作系统版本，系统资源限制情况，系统运行时间以及负载，Java 的版本信息等。

7.3 资源调度算法

一个调度平台，可以根据业务需要选择不同的调度算法，这里的作业资源调度算法跟操作系统的进程资源调度算法有相似性，但是不存在操作系统的系统进程用户进程调度划分，这里按照通俗的理解，例举一些常用的作业资源调度算法。

一种方式是先来后到的方式，先来的先被调用，先分配 CPU、内存等资源，后来的在队列等待，这种方式适合平均计算时间、耗用资源情况差不多的作业，为了让后来的作业有机会提前运行，通常还会匹配优先级，即优先级高的先运行，优先级一样的按先来后到方式运行。

但是实际操作的时候，优先级容易碰到问题，如果用户都认为自己的作业优先，把自己提交的作业优先级都设置的最高，这样排在后面的作业还是要等很久才被调度，特别是前面有一个耗用资源特别久的作业，比如占用几个小时乃至几天的大部分机器的 CPU 和内存的训练算法作业，导致排在后面的大量很短时间运行完、耗用资源比较少的作业很久才被调度，

实际上他们优先调度更适合。

另一种方式，也就是根据上面问题产生的最短时间或者最少资源耗用优先方式。但是这样也是有问题的，如果一个用户像买彩票多买概率就高一样，一次提交了大量作业（也许还有相同的作业），就为了优先得到执行，如果他提交的作业时间都比较短，那永远都是这个用户在占用计算平台集群资源，其他用户永远在等。

因此，还有一种公平方式去保证资源调度，也就是每个用户分配一个资源池，不能多用，如果要多提交作业任务，把自己的资源池占光了，就只能等了，大家都公平的使用资源。这样看似解决了公平问题，但是又是有问题的，跟大锅饭问题相似，有的用户的作业多任务重，分配到的资源少，有的用户作业稀少也占用一样的资源池，反而不公平了，这里按用户平均分配资源池并不能真正公平。

所以，又出现了按容量调度方式，通俗的说，每个用户的资源池里如果有多余的资源容量，要共享出来，拿出来给其他有需要的用户使用，调度平台去保障这种容量的管理和分配。

还只有更多的调度算法，这里不再一一列举。

我们可以看到，作业资源调度算法随着使用问题和解决会不断发展新的内容，同时在不同的业务场景中还会面临不同的权重，比如作业性质的权衡，生产作业要优先，训练作业要往后，生产作业中止后恢复要继续优先排，还有资源类别的权衡和分配策略，CPU 占用型和内存占用型谁更优先，CPU、内存、硬盘、带宽几种资源之间的调度搭配，不同硬件性能服务器的资源调度侧重，等等，是一个很复杂的课题，跟业务场景也很相关。

7.4 其他作业调度平台简介

在 7.1 节我们构思了一个调度平台的设计，并且利用之前学习的分布式技术来实现一个 MPI 的调度平台。本节我们看看市场上其他 MPI 调度技术，了解到其他互联网公司怎么做 MPI 调度的，再介绍一下 Mesos 和 Yarn 调度框架。

7.4.1 其他 MPI 作业资源调度技术

目前大型互联网公司的 MPI 集群的资源调度的主要做法是，基于 Toroue 的 PBS 方式进行任务和资源调度。PBS(Portable Batch System) 最初由 NASA 的 Ames 研究中心开发，主要为了提供一个能满足异构计算网络需要的软件包，用于灵活的批处理，特别是满足高性能计算的需要，如集群系统、超级计算机和大规模并行系统。

PBS 的主要特点有：代码开放，免费获取；支持批处理、交互式作业和串行、多种并行作业，如 MPI、PVM、HPF、MPL。PBS 是功能最为齐全，历史最悠久，支持最广泛的本地

集群调度器之一，PBS 目前包括 openPBS、PBS Pro 和 Torque 三个主要分支。其中 OpenPBS 是最早的 PBS 系统，目前已经没有太多后续开发，PBS Pro 是 PBS 的商业版本，功能最为丰富。Torque 是 Clustering 公司接过了 OpenPBS，并给与后续支持的一个开源版本。

由于 MPI 任务的运行需要执行 MPIRUN 命令，但是 MPIRUN 只能保证在一台机器上进行 MPI 的任务执行，而不能多台计算机 MPI 任务并行执行，并且，如果同时 100 个 MPI 任务执行 MPIRUN，那么会同时启动 100 个进程争夺资源，这样 CPU 时间片会轮流分配给各个人的任务，从而影响所有人的正常作业。所以，为了能实现多机并行启动 MPIRUN，并且针对有限的机器资源，能让多个 MPI 任务排队等待，形成任务队列，依次执行，有效分配资源，避免资源竞争。一种简单的做法就是利用 PBS 的任务调度方式来实现。

比如用 Torque 做任务管理系统，当多个用户使用同一个计算资源时，每个用户用 Torque 脚本提交自己的任务，由 Torque 对这些任务进行管理和资源的分配，提供对批处理作业和分散的计算节点的控制。

通过 Torque 命令提供任务的提交、队列管理、启动、超时中止、任务运行状态查看等。

由于 Torque 是一个类似于 Windows 任务管理器的作业管理系统，它不是一个可编程的专业的计算调度框架，它需要以人工输入命令的方式提交任务，然后以进程启动的方式执行，如果任务超时便杀死，它更像一个操作系统的任务管理工具，针对任务的启动和查看状态的进程管理，而不是重点在任务调度和资源调度上。

任务调度上，它没有一个调度器角色的概念，在启动 MPI 任务后，能够实时检测各任务的运行状态，并针对异常和任务完成状态作出处理和响应。当有异常产生时，基于 Torque 的任务工具无法容错，只能全盘中止，再重新计算。而且也无法检测到任务总体完成状况。

资源调度上，它无法进行计算机的 CPU、内存、网络等资源的隔离和分配，根据 MPI 任务需要的资源，比如一个任务需要 2 个 CPU，80G 内存，它能够隔离出一台机器上符合要求的资源用于完成该任务，并且将剩余的资源用于其他任务。如果无法做到资源隔离和分配，那么不能达到高的资源利用率，容易造成浪费。

另外，Torque 通过命令提交 MPI 任务，但是只当成普通的操作系统进程任务，并没有一个 MPI 任务包的概念，包括 MPI 的运行程序，所需资源和配置等。由于需要大量的命令参数来输入这些运行时的必要条件，造成移植性扩充性差，操作复杂。

7.4.2 Mesos 和 Yarn 简介

1. Mesos 介绍

我们在第 2 章谈到 Spark 的时候提及过 Mesos，Mesos 最初是加州伯克利大学的一个研究项目，后来加入到 Apache 孵化器成为开源产品。Mesos 的目标是提供一个分布式应用的资

源隔离框架，能在它上面运行 Hadoop、MPI、Hypertable、Spark 以及其他分布式计算产品，Mesos 通过提供一个编程 API 框架，让其他产品扩充实现这个框架接口内容去完成这一目标，如图 7-2 所示。

Mesos 通过 Zookeeper 提供 Master 的故障恢复。Mesos 对于资源隔离的实现是采用 Linux Containers 完成，也就是我们前面介绍的 Cgroup 进程组方式隔离。Mesos 目前主要是提供 CPU 和内存的资源管理，其他资源如网络带宽和硬盘使用不支持。

Mesos 对 MPI 的默认支持是 mpich2 版本，也表示能支持 Torque 方式启动，但是它修改了部分 Torque 代码，在淘宝内部的 MPI 应用中，多数是 Openmpi 版本，而且 Torque 的版本使用官方的，经过我们尝试，发现存在不兼容。Openmpi 实现的 MPI 应用最好去扩充 Mesos 提供的资源调用接口（Scheduler 和 Executor），不要直接采用 Mesos 默认的 MPI 支持方式。

下面我们讲述一下 Mesos 的架构和完成一个资源分配的流程，如图 7-3 所示。

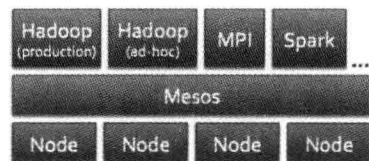


图 7-2 Mesos 产品图

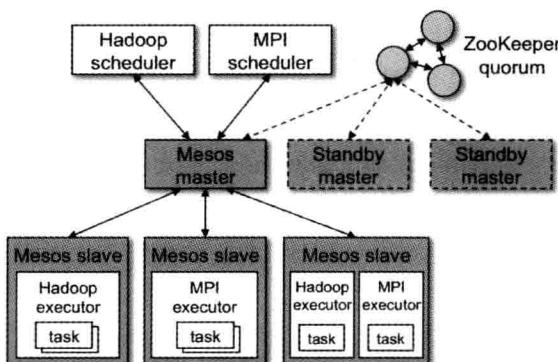


图 7-3 Mesos 架构

通过图 7-3，我们可以看到 Mesos 是一个 Master/Slave 的分布式结构的延伸设计，Slave 负责资源的注册，Master 负责资源的分配（offer）。另外为了让分布式应用获取资源跑起来，Mesos 还设计了 Scheduler 和 Executor 两个组件，Scheduler 向 Master 注册并接受资源的分配，Executor 用于在 Slave 上执行任务。

我们看看一个资源分配的流程：

- 1) Slave 向 Master 注册资源，这里是细粒度的，包括 CPU 和内存；
- 2) Master 发送资源 Offer 给 Framework；
- 3) Framework 判断资源是否合适，并反馈给 Master 需要运行的任务及其资源；
- 4) Master 发送任务到 Slave 上的 Executor 上执行。

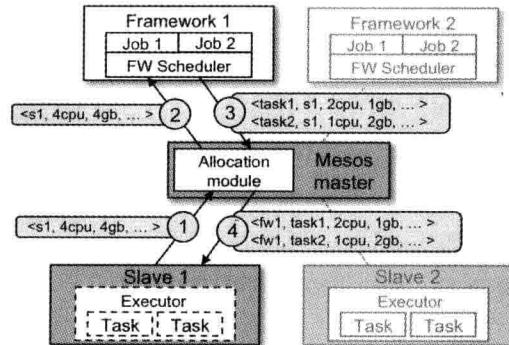


图 7-4 Mesos 流程



通过这个完整流程，我们可以发现，“资源的分配策略”和“运行任务逻辑”实际上由开发者自己实现 Scheduler 和 Executor。

尽管 Mesos 在其官网上醒目提示：本产品仍然还不稳定。但是 Mesos 的论文还是值得参考的，目前资源隔离领域并没有太好的做法（资源隔离方案实现上跟操作系统也存在很大的绑定性），Mesos 是这块技术领域探索上的先行者。

（请参考 http://www.mesosproject.org/papers/nsdi_mesos.pdf）

目前运行在 Mesos 上的主要是他们同出一门的产品 Spark，虽然 Mesos 很想让 Hadoop 运行在其上面，但是显然 Hadoop 社区在情感上难以接受这样，它们自己搞了一个 Yarn 的框架。

2.Yarn 介绍

也许大家都注意到了，Hadoop2.0 跟 1.0 比有了巨大的变化，或许为了弥补某些方面的不足，Hadoop2.0 痛下决心，将原来根据 Google Map/Reduce 和 GFS 论文实现的 Hadoop1.0 版，翻天覆地改装成了一个资源和任务调度的框架，如果不是还继续姓 Hadoop，看上去真的不像一个父母生的孩子了。2.0 版的 Hadoop 看上去像如图 7-5 所示。

Hadoop 官方说 1.0 版里 JobTracker

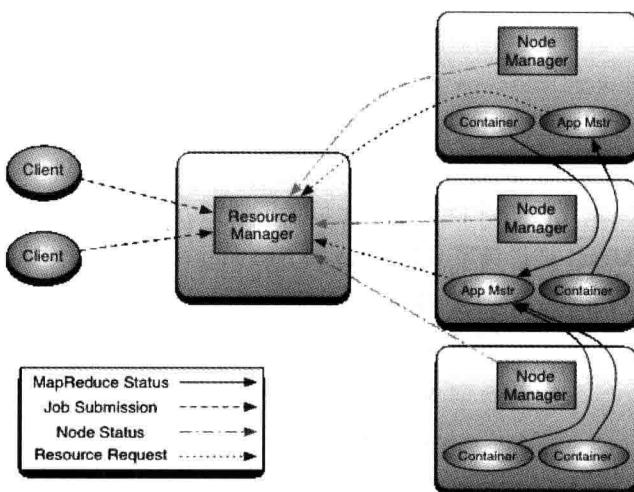


图 7-5 Yarn 架构

设计的不好，需要一分为二，资源分配和任务分配功能不能合在一起。他们想做成一个资源请求和分配，再加上容器隔离执行任务的一套框架，这个框架不仅可以运行适合 Map/Reduce 按行分析的计算，也可以运行其他各种计算框架（这样就没人再攻击 Hadoop1.0 的 Map/Reduce 缺点了）。

这个新框架的名字叫做 Yarn，它包括下面核心的组件：

- ResourceManager：负责资源管理和调度。
- NodeManager：负责各节点的资源汇报。
- ApplicationMaster：负责资源请求和判断。
- Container：隔离资源的单元，用于运行任务。

我们再结合上面的架构图描述一个资源请求的流程：

- 1) NodeManager 向 ResourceManager 注册各机器资源。
- 2) 客户端向 ResourceManager 提交作业。
- 3) ApplicationMaster 向 ResourceManager 请求资源，并判断是否满足需要。
- 4) ResourceManager 以 Container 的形式将资源反馈给 ApplicationMaster。
- 5) Container 做为资源单元保证作业隔离运行。

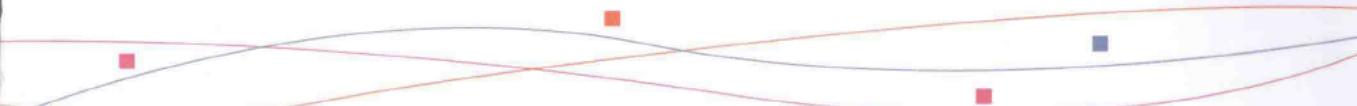
到这里，我们是否感觉这些流程与上面的 Mesos 有点像呢：

<input type="checkbox"/> Mesos	有点像	Yarn
<input type="checkbox"/> master	有点像	ResourceManager
<input type="checkbox"/> slave	有点像	NodeManager
<input type="checkbox"/> framework	有点像	ApplicationMaster
<input type="checkbox"/> executor	有点像	Container

当然，它们是不同的产品，很多功能并不等同，只是相似而已。不过我们因此更容易理解一个资源调度框架的主要结构和职责。

Yarn 官网里介绍有这么一句话：“In addition it can optionally specify location preferences such as specific machine or rack; this allows for first class support of the MapReduce model of performing the computation close to the data being processed…” 表示 ApplicationMaster 在进行资源请求时，可以指定一些位置的属性，比如特定的机器和机架，为了让计算靠近数据存放的机器。对于 MPI 等迭代计算的应用，计算初始时需要下载大量的数据到本地，非常耗费时间和带宽，这个功能或许能有所改观。

Yarn 目前还没有最稳定版本发布，当前也只支持内存隔离，还不支持 CPU、网络等其他资源隔离，不过 Yarn 社区正准备尝试用 Linux 上的 Cgroup 进程组方式改进。



本书从作者的实战经验出发，深入浅出地讲解了如何建立一个Hadoop那样的分布式系统，实现对多台计算机CPU、内存、硬盘的统一利用，从而获取强大计算能力去解决复杂问题。一般互联网企业的分布式存储计算系统都是个大平台，系统复杂、代码庞大，而且只适合公司的业务，工程师很难下载安装到自己的电脑里学习和吃透。本书对分布式核心技术进行了大量归纳和总结，并从中抽取出一套简化的框架和编程API进行讲解，方便工程师了解分布式系统的主要技术实现。这不是一本空谈概念、四处摘抄的书，这本书包含了大量精炼示例，手把手教你掌握分布式核心技术。

本书主要内容

- 分布式并行计算的基本原理解剖；
- 分布式协调的实现，包括如何实现公共配置管理，如何实现分布式锁，如何实现集群管理等；
- 分布式缓存的实现，包括如何提供完整的分布式缓存来利用多机内存能力；
- 消息队列的实现，包括如何实现发送和接收模式；
- 分布式文件系统的实现，包括如何像操作本地文件一样操作远程文件，并利用多机硬盘存储能力；
- 分布式作业调度平台的实现，包括资源隔离、资源调度等。



CD-ROM

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604

数字阅读：www.hzmedia.com.cn
华章网站：www.hzbook.com
网上购书：www.china-pub.com



上架指导：计算机/程序设计/大数据

ISBN 978-7-111-45503-5



9 787111 455035 >

定价：59.00元（附光盘）