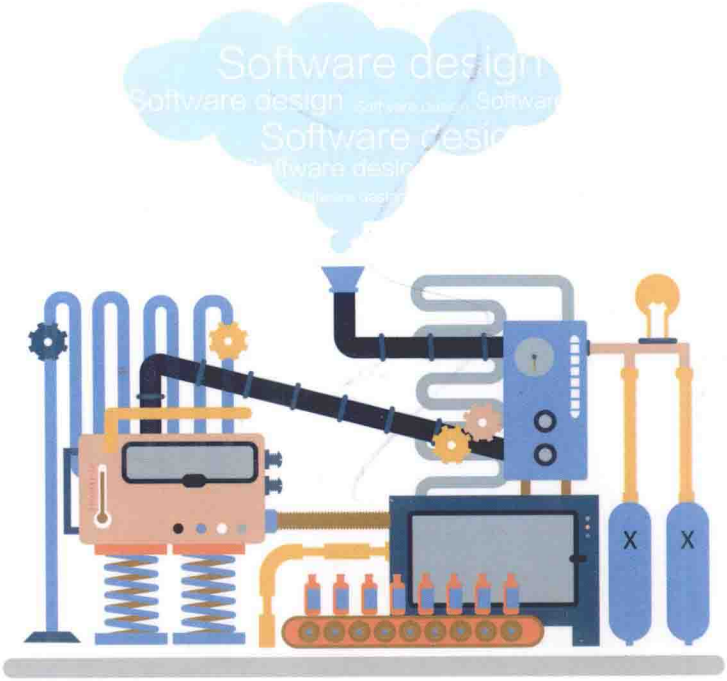
Broadview®
www.broadview.com.cn

自动化运维
软件设计实战

面对众多的设备类型如何实现自动化运维？
如果Ansible、Puppet、SaltStack都无法满足你对自动化运维的需求，那么本书将会带给你一种全新的思路！

吴文豪 著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

自动化运维软件设计实战

吴文豪 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书主要讲解采用 OSGi 技术来设计一款可插拔式的运维软件的方法与思想，为读者提供一种不一样的运维软件设计与自动化运维解决方案。

本书分三部分，第一部分讲解开源社区中比较流行的三款集中化运维软件，第二部分与读者一起分享为什么要采用 OSGi 的技术来设计集中化运维软件，第三部分介绍设计这款运维软件所涉及的技术和一些设计思想。

本书适合从事系统运维及运维开发的人员阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

自动化运维软件设计实战/吴文豪著. —北京：电子工业出版社，2015.7

ISBN 978-7-121-26468-9

Ⅰ. ①自… Ⅱ. ①吴… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 142285 号

责任编辑：陈晓猛
印　　刷：北京京师印务有限公司
装　　订：北京京师印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：18.25　字数：408.8 千字
版　　次：2015 年 7 月第 1 版
印　　次：2015 年 7 月第 1 次印刷
印　　数：3000 册　定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前 言

不知不觉也与各种开发语言做了挺久的小伙伴，依稀记得当初很好奇用 C 语言这种在黑框框里面跑的程序怎么才能写出一个界面，后来在兴趣的驱使下不断地接触各种各样的技术，让我感慨时间过得还是挺快的。

我曾经接触过一个与自动化运维相关的项目，整个项目对我来说挑战是非常大的，而且开发过程也非常坎坷。当做到运维部分的时候，出现了非常大的挑战，我们不得不面对技术水平参差不齐的维护团队，各种各样的操作系统，限制条件非常多的网络环境，当然，还有项目进度的步步紧逼。对于运维的功能，Ansible、Puppet、SaltStack 都是非常不错的选择，但是偏偏在我们所要面对的环境下用起来实在太困难。在和同事讨论之后，偶然发现 OSGi 这种技术能够解决我们的问题，所以也就自己重新造了一个轮子来解决我们所面临的问题。

项目进行得差不多的时候，我觉得 Apache Karaf 与 Apache ActiveMQ 这两种技术整合起来所设计的运维框架也有它的一些优点，于是在 OSChina 上写了一篇博客与大家分享。非常巧的是，电子工业出版社的编辑通过我的博客联系上了我，希望我可以写一本关于采用 Apache Karaf 与 Apache ActiveMQ 整合所设计的运维软件的书籍，这让我感到非常荣幸。

我把当初设计这套软件的思路以及一些需要注意的要点写在了这本书中，希望我所分享的内容能够对运维的小伙伴们有所帮助。

本书面向的读者

本书面向的读者是从事系统运维的开发人员，希望能够给读者在设计运维软件的时候提供一种不同的思路。书中的内容以思路分享居多，因为笔者认为如今的互联网非常发达，某个功能如何实现，我们搜索一下就会找到很多方案，而思路的分享更能引起读者与笔者在思想上的碰撞，在碰撞中让读者发现一些其他的方法。

内容介绍

本书共 12 章。

第 1 章与读者一起探讨什么是自动化运维。

第 2 到第 4 章简单介绍目前比较热门的集中化运维软件 Ansible、Puppet 和 SaltStack。

第 5 章介绍为什么在有这么多集中化运维软件的情况下我们还需要重复造一个轮子。

第 6 章和第 7 章介绍重复制作轮子所需要的一些技术——Apache Karaf 和 Apache ActiveMQ。

第 8 章到第 10 章介绍如何使用 Apache Karaf 和 Apache ActiveMQ 制作出一个可插拔式的集中化运维框架。

第 11 章介绍如何与 Zabbix 进行整合。

第 12 章与读者分享了一个小故事，希望通过这个小故事让读者能够更加了解这款运维软件所要解决的问题。

致谢

- 感谢我的同事陈自欣，我非常佩服他在技术知识面上的广度，采用 OSGi 的技术来开发运维软件的思路就是他提出来的。
- 感谢我的同事崔威，在我刚工作的时候教会了我许多软件开发的技术，让我在后续的技术发展道路上少走了许多弯路。
- 感谢我的家人，给了我这么多的时间让我可以专心地完成本书的写作。

吴文豪

2015 年 5 月

目 录

第 1 章 什么是自动化运维 / 1
1.1 硬件运维和软件运维 / 1
1.1.1 小故事之一——电脑专家 / 1
1.1.2 小故事之二——你居然不会修电脑 / 2
1.1.3 硬件运维与软件运维 / 2
1.2 软件运维的主要问题 / 3
1.2.1 设备数量多 / 3
1.2.2 系统异构性大 / 3
1.2.3 虚拟化的成熟带来更大的困难 / 4
1.3 运维常用工具 / 4
1.3.1 Puppet / 6
1.3.2 SaltStack / 6
1.3.3 Ansible / 7
1.4 自动化运维 / 7
1.5 小结 / 9

第 2 章 集中化运维利器——Ansible / 11
2.1 环境准备 / 11
2.2 安装 Ansible / 12
2.2.1 使用 CentOS 的 EPEL 源进行安装 / 12
2.2.2 使用 Easy_Install 安装 Ansible / 14

2.3 Ansible 基础 / 14
2.3.1 资产配置 / 14
2.3.2 执行命令 / 17
2.3.3 指定目标主机 / 18
2.3.4 常用命令示例 / 19
2.4 Ansible 常用模块 / 21
2.4.1 文件管理模块 / 21
2.4.2 命令执行模块 / 25
2.4.3 网络相关模块 / 28
2.4.4 源码管理模块 / 30
2.4.5 包管理模块 / 32
2.4.6 系统管理模块 / 33
2.5 PlayBook / 37
2.5.1 PlayBook 简介 / 38
2.5.2 Include 语法 / 41
2.5.3 变量 / 41
2.5.4 条件 / 43
2.5.5 循环 / 44
2.5.6 PlayBook 使用实例——集中化日常巡检 / 46
2.6 使用 Ansible 的 API / 49
2.7 小结 / 50
2.7.1 Ansible 的优点 / 50
2.7.2 Ansible 的缺点 / 51

第 3 章 集中化运维利器——Puppet / 52
3.1 Puppet 与 Ansible / 52
3.2 Puppet 基础 / 56
3.2.1 安装 Puppet / 57
3.2.2 Puppet 主要配置文件 / 58
3.2.3 颁发证书 / 61
3.2.4 第一个 Puppet 示例 / 62
3.3 Puppet 的常用资源 / 64
3.3.1 定时任务——cron / 64

3.3.2 命令执行——exec / 65
3.3.3 文件管理——file / 67
3.3.4 包管理——packag / 69
3.3.5 服务管理——service / 70
3.4 Puppet 语法基础 / 71
3.4.1 资源 / 72
3.4.2 类 / 73
3.4.3 变量 / 73
3.5 小结 / 76
3.5.1 Puppet 的优点 / 76
3.5.2 Puppet 的缺点 / 76

第 4 章 集中化运维利器——SaltStack / 77
4.1 SaltStack、Puppet、Ansible / 77
4.2 无 Agent 模式——SaltSSH / 79
4.3 SaltStack 的基本组成 / 81
4.4 Salt State 概述 / 82
4.4.1 top.sls / 82
4.4.2 state 文件 / 83
4.4.3 配置主机 / 83
4.4.4 SaltState 之 Requires / 84
4.4.5 Template、Extends、Includes / 85
4.5 无主服务器模式运行 / 88
4.6 使用 SaltStack 的定时作业 / 89
4.7 实时执行命令 / 89
4.7.1 target / 89
4.7.2 function / 93
4.7.3 arguments / 93
4.8 Pillar / 93
4.8.1 使用 Pillar / 94
4.8.2 Pillar 的一些操作方法 / 95
4.9 小结 / 96
4.9.1 SaltStack 的优点 / 96

4.9.2 SaltStack 的缺点 / 96

第 5 章 重复造一个轮子 / 97
5.1 从一个自动化运维软件说起 / 97
5.2 困难重重 / 100
5.2.1 多样的设备类型 / 100
5.2.2 运维设备的总量大 / 100
5.2.3 艰难的环境 / 100
5.2.4 多变的客户需求 / 101
5.3 轮子需要的特性 / 102
5.4 ActiveMQ 基础 / 104
5.4.1 配置 ActiveMQ / 105
5.4.2 部署 ActiveMQ / 114
5.4.3 第一个 ActiveMQ 例子 / 117
5.5 Apache Karaf / 123
5.5.1 OSGi 简介 / 123
5.5.2 为什么选择 Karaf / 124
5.5.3 基础架构设计 / 124
5.5.4 启动 Apache Karaf / 126
5.5.5 制作第一个 OSGi 包 / 127

第 6 章 ActiveMQ 概览 / 136
6.1 消息发送 / 136
6.1.1 TextMessage / 136
6.1.2 MapMessage / 138
6.1.3 BytesMessage / 140
6.1.4 StreamMessage / 144
6.1.5 BlobMessage / 145
6.2 断线重连机制 FailOver / 158
6.2.1 配置 FailOver / 158
6.2.2 FailOver 的常用参数 / 159
6.3 消息生命周期 / 160

6.3.1 为什么消息需要生命周期 / 160
6.3.2 使用消息超时机制 / 162
6.4 清空不常用的队列 / 163
6.5 使用 JMX 获取队列信息 / 164
6.5.1 启用 ActiveMQ 的 JMX 功能 / 165
6.5.2 获取 ActiveMQ 的队列信息 / 167
6.6 ActiveMQ 的 HA 方案 / 173
6.6.1 配置 NFS 服务器 / 173
6.6.2 配置 NFS 客户端 / 173
6.6.3 调整消息中间件的配置文件 / 174
6.6.4 将 Failover 作为连接串 / 174
6.6.5 原理 / 175

第 7 章 Apache Karaf 概览 / 176
7.1 理解 Import 和 Export / 176
7.2 Service Wraper / 180
7.2.1 支持的平台 / 180
7.2.2 使用 Service Wrapper / 181
7.2.3 Karaf Wrapper 的配置文件 / 184
7.3 使用控制台 / 187
7.3.1 Shell 模块 / 187
7.3.2 OSGi 模块 / 190
7.3.3 LOG 模块 / 191
7.3.4 SSHD 模块 / 192
7.4 Karaf 的日志 / 194
7.4.1 Karaf.Out / 194
7.4.2 Karaf.log / 195
7.4.3 Application log4j 日志 / 196
7.5 Karaf 子实例 / 197
7.5.1 使用 Karaf 子实例 / 197
7.5.2 为什么需要使用子实例 / 201
7.6 扩展 Karaf 控制台 / 203
7.6.1 使用 Maven 创建项目 / 204

7.6.2 编写控制台插件包 / 206
7.6.3 部署插件包 / 207
7.7 使用 Web 控制台 / 207
7.8 使用 Feature——JDBC 数据源 / 209

第 8 章 核心框架 / 213
8.1 核心层概述 / 213
8.2 核心框架 / 214
8.2.1 服务端消息处理 / 216
8.2.2 客户端消息处理 / 217
8.2.3 插件状态汇报 / 218
8.3 消息分发服务端 / 219
8.4 插件状态服务端 / 220
8.5 PlayBook 服务端 / 221
8.5.1 PlayBook 服务端设计目的 / 221
8.5.2 PlayBook 设计示意图 / 223
8.6 结果处理服务端 / 226
8.6.1 结果处理服务端设计目的 / 226
8.6.2 结果处理服务端处理流程 / 226

第 9 章 通用插件包 / 228
9.1 插件包概览 / 228
9.2 作业调度模块——Cron4J / 230
9.2.1 Cron4J 基本使用方式 / 231
9.2.2 作业调度参数 / 232
9.2.3 重新调度作业 / 233
9.2.4 调度系统进程 / 233
9.3 数据访问模块——MidaoProject / 234
9.3.1 为什么选择 Midao / 235
9.3.2 使用 Midao / 236
9.4 序列化模块——Gson / 237
9.5 交互式命令执行模块——JavaExpect / 242

9.6 小结 / 249

第10章 常用插件 / 250

10.1 文件下发插件 / 250

10.1.1 文件下发插件设计 / 250

10.1.2 使用 Apache Common IO / 251

10.2 文件抓取插件 / 254

10.2.1 文件抓取插件整体设计 / 254

10.2.2 文件抓取插件设计要点 / 256

10.3 命令执行插件 / 257

10.4 目录结构查询插件 / 258

第11章 整合 Zabbix / 261

11.1 编译安装 Zabbix / 261

11.1.1 部署 MySQL / 261

11.1.2 编译部署 Apache+PHP / 263

11.1.3 安装 Zabbix / 267

11.2 强大的触发规则 / 268

11.2.1 触发规则概览 / 268

11.2.2 特色的触发规则 / 270

11.3 Zabbix 调用 OSGi 运维功能 / 271

第12章 案例 / 275

第1章 什么是自动化运维

1.1 硬件运维和软件运维

运维，指的是对一些已经搭建好的网络软 / 硬件进行维护。

在笔者实际从事运维开发这一工作之前，只要提到运维工程师，脑海里面就会出现这么一个场景：在一个有几百台设备发出“嗡嗡”声的机房里面，一个身穿白衬衫、带着眼镜的帅气小伙正在对一台电源出故障的设备进行故障排查（曾经简单地认为这就是运维的全部内容）。

从事运维开发工作之后，发现运维也是有细分领域的，有负责业务运维的、数据库运维的、主机运维的，等等。虽然运维的工作被细分成这么多种，但是粗略地看，运维主要分为硬件运维和软件运维两大块。无论是哪一种运维，它的最终目的都是为了让应用系统能够稳定地运行，更好地为企业提供服务。

1.1.1 小故事之一——电脑专家

小明对电脑的硬件非常感兴趣，家里的电脑都不知道被他拆了又装、装了又拆多少遍了，年纪轻轻的他就已经懂得了各种电脑问题的维修技巧。这不，在大妈圈里面，小明可是一个被传得神乎其神的电脑专家。按照大妈们的话来说，无论你的电脑出了哪方面的问题，只要找到小明，问题都能迎刃而解。

“咚咚咚”，小明听到三声敲门声之后打开了门，看到一脸焦急的小菜。还没等小明开

口，小菜就说到："小明哥啊，赶紧帮忙救个火啊，我在学校负责维护的一套选课系统今天在选课的时候突然变得很慢了，整个系统慢到几乎是不可用的状态了，平时我也就只会装个杀毒软件杀个毒，每周重启一下电脑，但这次不管怎么弄就是没效果。我听我妈妈说你在电脑方面很厉害，你能不能帮我检查一下，给我提供点建议？"小明一听就傻眼了……

1.1.2 小故事之二——你居然不会修电脑

小菜是某大学计算机专业的学生，在班里可是出了名的计算机高手，不论是 C 语言、Java 程序设计，还是数据库这些科目，小菜在班里的考试成绩都是排名第一。而且在学长的指导下，小菜在系统运维方面的能力也十分了得，学校的教务系统、班上的点名系统都是小菜在负责维护，只要他发现系统出现运行缓慢或者是无法打开的现象，不出半天他就能漂亮地把问题解决。

在班里计算机成绩这么出众，带来的结果必定只有一个，那就是有女同学会叫你帮忙修电脑。最近小菜就碰到这么个烦心事，有个师妹的电脑出问题了，不知道为什么就是开不了机，于是就找到了小菜，希望小菜能帮她把电脑修好。小菜去师妹宿舍之后捣鼓了半天，却没定位出电脑开不了机的问题是因为主板损坏。于是师妹忍不住在旁边小声嘀咕："还说是什么计算机高手，连修个电脑都不会"。

1.1.3 硬件运维与软件运维

硬件运维可以理解为对基础设施的运维，包括机房的设备和电脑设备。比如对机房的备用电源、空调，主机上的硬盘、内存这些物理设备的维护，都可以看作硬件运维。

软件运维包括系统运维和应用运维。系统运维指的是对操作系统、数据库、中间件等的监控和维护，这些系统是介于设备和应用之间的。应用运维指的是对业务系统的运维，例如企业内部的 OA、ERP 等业务系统。

前面我们讲到小明作为一个对硬件知识了解得比较深入的中国好邻居，却解决不了小菜的燃眉之急。第二个小故事里面的小菜虽然对软件维护方面的知识有很深了解，但是却解决不了师妹的电脑开不了机的硬件故障。运维的知识很广阔，硬件层面的运维与软件层面的运维所需要了解的知识点差别还是比较大的。

所以本书后续所描述的自动化运维中的"运维"，指的都是软件运维，也就是系统层面的运维和应用层面的运维。

1.2 软件运维的主要问题

1.2.1 设备数量多

在虚拟化发展起来之前，企业的 IT 建设普遍是把单个应用部署在一套甚至是多套基础设施上进行建设的。直白点讲，就是一个应用会部署在一套服务器上。随着公司内部的 IT 系统一天一天地增大，运维工程师需要运维的主机数量也随之慢慢地增加。刚开始的时候，运维工程师可能只需要运维十多台设备，一段时间过后，他们需要运维的设备数可能就会翻倍了。给十台主机打补丁、升级软件，估计还受得了，但是给五十台主机打补丁、升级软件，这种重复性的劳动估计就没多少人能受得了了。

有小伙伴可能会问，为什么不把多个应用系统部署在一台主机上呢？这样一方面可以减少由于主机数量的增加而给运维带来更大的负担，另一方面也可以为企业节省成本，何乐而不为呢？我们可以从以下几个角度来看待这个问题。

首先，为了避免被某个软件厂商垄断了企业的系统，一般情况下企业的系统都会交给不同的厂商进行开发。既然是不同的厂商，当多个应用系统被部署到了一起的时候，很容易就会出现这样一个场景：A 公司和 B 公司的系统在同一台主机上，A 公司于本周五上了一个新功能，周末的时候出现了主机的 I/O 负载很高，导致该主机上的业务系统都出现了反应迟钝的现象。于是 B 公司的开发人员就开骂了，指责 A 公司的新功能导致主机负载高，这个得扣他们的款。A 公司的开发人员也不认输，认为我们的功能绝对没有问题，是你们的系统周末出账用了太多的 I/O 才导致系统的速度变缓慢的。于是一场扯皮大战就这样开始了。

然后，从保证业务系统的可用性来看，不同的应用系统之间需要在系统层进行隔离。不然一旦操作系统出现问题，部署在操作系统上的所有应用系统都会出现故障，这肯定是达不到企业在业务上的要求的。而且，多套业务系统部署在一台主机上，也会为性能优化、故障排查等后续处理带来许多干扰。

1.2.2 系统异构性大

给运维工程师带来困扰的第二个问题就是系统的异构性。

由于应用系统是由不同开发商开发的，不同开发商内部的技术栈会有差异，所以很容

易会出现 A 开发商开发出的应用系统需要跑在 Redhat 上，Web 服务器需要用 Tomcat，数据库需要用 MySQL，而 B 开发商开发出的应用系统需要用 Windows Server 来承载，Web 服务器需要用 IIS，数据库需要用 SQLServer 的情况。

这个时候运维工程师就傻眼了，这怎么运维？系统出故障了之后还得想想究竟是用 SSH 去维护还是用远程桌面去维护。公司规定每月要有一次常规性的主机重启，究竟哪个 IP 的设备需要用 SSH 去重启，哪个 IP 的设备需要用远程桌面去重启？

1.2.3 虚拟化的成熟带来更大的困难

可能有那么一些运维工程师通过自己多年积累下的脚本度过了一些难关，本以为可以歇一歇了，没想到近年来，随着虚拟化的日渐成熟，又一个挑战来了。

以前企业的设备数量虽然会增长，但是毕竟需要走过一个漫长的企业内部流程才能完成设备的采购和上线。但是随着虚拟化的成熟，企业的 IT 建设再也不需要像以前一样，上一个新的业务系统就经历一个漫长的采购流程了，也不需要再费心思去找个机房放主机了。IT 管理人员只需要申请一台虚拟机，然后再在 CMDB 里面填一下这台主机的信息就可以了。

> CMDB 存储管理企业 IT 架构中设备的各种配置信息，它与所有服务支持和服务交付流程都紧密关联，支持这些流程的运转，发挥配置信息的价值，同时依赖于相关流程以保证数据的准确性。

在这个虚拟化如此“任性”的年代，IT 建设的成本在不断降低，IT 建设的速度也在不断提升，需要运维的设备数量从原来的几百台增加到几千台甚至上万台，而且很有可能这些设备也仅仅是这家企业的一个部门的设备数而已，这给运维工程师带来了更大的挑战。

1.3 运维常用工具

当设备数量很多、设备存在很大的异构性时，我们能不能有一个更加轻松愉快的方式来管理这些主机呢？从竖井式 IT 建设到层次性 IT 建设过程中对运维工程师来说，最大的问题是设备数量的爆炸性增长，原有靠堆人力或者积累脚本的做法已经显得不太可行了。这个时候，我们就需要思考这样一个问题，能不能有一个既可以减少人力成本，又可以可

靠地批量完成集中化运维任务的方法呢？

在设备的架构比较一致的情况下，实现集中化运维并不是什么难事。对于 Windows 的设备，假如我们需要对它们进行集中化的重启，就可以采用 PowerShell 的方式来对设备进行集中化的操作。例如，我们需要对主机进行重启的时候，就可以通过 PowerShell 去调用 WMI 的 API 来完成这个功能，而像一些 msi 程序的发布和安装，就可以通过 AD 域控的方式去完成，总体来说还是挺方便的。

在 Linux 和 UNIX 操作系统下，我们可以通过 SSH+Expect 的方式或者是双机互信后采用 SSH 的方式完成集中化的运维，难度也不大。

但是，由于不同业务系统对可用性和鲁棒性的要求不一样，通常会出现既有 Windows 主机，也会有 Linux 和 UNIX 主机的场景。而这种操作系统的多样性，也给集中化的运维带来不少的麻烦。无论是 AD 域控+WMI 还是 SSH 的方式，都只是解决了一类主机运维的问题。试想一下，现在需要对运维的所有主机进行常规的集中化重启操作，我们还得先去看一下究竟哪些设备是 Windows 操作系统，哪些设备是 Linux 操作系统，然后再去做相应的操作。这种费时费力的操作方式并不是我们想要的，我们更希望有一个统一的入口，我们可以通过这个入口所提供的对外封装好的一些接口去对异构的设备进行集中化运维，如图 1.1 所示。

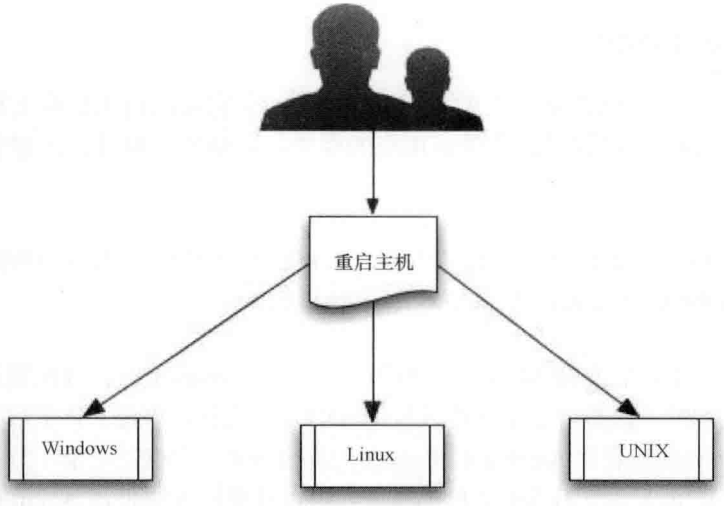


图 1.1　对异构设备进行集中化运维

随着开源软件的不断发展，现在已经有许多不错的开源软件可供选择了，可以通过较低的成本完成集中化运维的目标。目前比较流行的开源集中化运维软件有 Puppet、SaltStack 和 Ansible。

1.3.1 Puppet

Puppet 是一款用 Ruby 语言实现的 C/S 类型的集中化配置软件，它是由 Luke Kanies 带领他的 Puppet Labs 团队开发的。

Puppet 既可以采用“客户端 / 服务端”的模式运行，也可以单独运行。它是采用 C/S 星型结构设计的，所有的客户端都会和一个或者几个服务器交互。每个客户端都会周期性地向服务器发送请求，获取最新的配置，保证配置信息的同步。默认情况下每半个小时会连接一次服务器，下载最新的配置文件，并且严格按照配置文件来配置服务器。

在可扩展性方面，Puppet 有一套自己的 DSL 语言，可以很容易地在它原有模块的基础上进行扩展，并且内置了一些可以管理配置文件、用户、软件包的模块。

命令式的集中化运维软件描述的是一个运维操作应该怎么去做，而声明式的方法只关心一个运维操作最后是否到达我们所期望的状态，而具体它是怎么做到的，我们则不关心。Puppet 就是一个采用声明式的方法来定义应该怎么对系统做配置的软件。

1.3.2 SaltStack

SaltStack 是一个用 Python 开发的集中化运维软件，它可以简化运维工程师对设备的批量运维操作。SaltStack 内置了许多现成可用的模块，包括安装软件、配置参数、启停服务等功能。

从支持操作系统的丰富度这个角度来看，SaltStack 支持的操作系统种类也十分丰富，它支持 Linux、UNIX、Solaris、Windows 等多种操作系统。

在软件的设计上，它支持 Master 主动推送配置和 Minion 定时拉取配置的方式，这点与 Puppet 是十分类似的。同时，它还支持远程命令的并行执行，自带了许多日常执行模块，所以我们可以把 SaltStack 看作 Ansible 和 Puppet 的混合版本。它也是一个十分不错的集中化运维软件，并且 SaltStack 还支持 SaltSSH 的方式，可以让我们无须使用 Agent 就能够对主机进行轻易的批量操作。当我们希望 SaltStack 能够具备更好的扩展性，以及更好地使用 SaltStack 本身提供的模块时，我们可以在客户机安装 Salt Minion 来进行主机的集中化运维。

1.3.3 Ansible

Ansible 是一个用 Python 设计的通过 SSH 的方式对主机信息进行集中化运维的软件。没看错，用纯 SSH 的方式，也就是说我们的主机上是不需要安装任何 Agent 端的。它与 SaltStack 非常类似，都是一种命令式的集中化运维工具。

虽然说不需要安装任何 Agent 就可以用，但是它对被操作的主机其实还是有一定要求的。例如，有一些模块在调用的时候会提示客户机需要安装某个 Python 的扩展。当然，对于 Windows 主机来说，使用 Ansible 就有点麻烦了。因为我们需要在 Windows 上配置好 OpenSSH，这样才可以让 Ansible 正常地运转。当然，Ansible 在 Windows 上的表现并不是那么出色，但是换个角度看，Ansible 这种不需要装 Agent 的做法也为我们提供了不少的便利（注：Ansible 较新的版本已经能够对 Windows 进行操作了，采用的是 WMI 的方式）。

1.4 自动化运维

有了像 Puppet、Ansible 和 SaltStack 这样的工具，我们已经可以轻松地做到集中化运维了，一些集中化的部署、集中化重启的动作都可以轻易完成。但是我们的运维过程却还没有达到自动化的水平。例如，我们现在维护了 200 台设备，A 主机上跑了一个业务系统，由于程序设计的原因，总是会不定期地出现应用服务器崩溃的问题，但是开发商由于技术问题一时半会解决不了，这个时候运维工程师就只能盯着自己的手机短信，一旦出现业务系统故障了，就得熟练地打开 SecurityCRT 登录主机，然后再熟练地敲下 restart 命令。没几天后，有用户报障说业务系统 B 无法上传文件了，再次熟练地登录那台主机，发现本来就不大的磁盘被系统写日志文件给写满了，于是工程师又熟练地敲下了 rm 命令删掉一些日志文件。一段时间后，又有用户报障……

于是，我们可怜的运维工程师虽然有那么多集中化运维工具可以选择，但是却没有一款能帮助他们减轻负担。因为这些故障往往是非集中化的操作，解决的方法都是靠运维工程师在日常运维的过程中对某些场景积累下来的特定的经验，所以在这种情况下，无论使用哪一种集中化运维工具，和我们直接通过 SSH 或者远程桌面去操作其实没有多大的区别。问题出现一两次，我们可以登录出现故障的主机进行处理，但是次数多了呢？难道我们就只有在每次故障出现之后都重复做这些运维操作吗？

这个时候，我们就需要把运维从集中化提升到自动化的水平了，而运维自动化，是一

个对单点发生的故障运维知识沉淀的过程。

为什么说是单点发生？假如是多点发生并且重复的故障，我们通过集中化运维工具就可以很好地解决问题，但是面对单点发生的故障，集中化运维工具并不能产生很好的效果。

为什么说是知识的沉淀？假设出现我们上面所说的 B 主机磁盘写满导致业务系统无法上传文件这样的故障，在排查后已经把问题定位清楚并且有解决方法了。对于运维来说，这个过程就是一个经验沉淀的过程，然后就可以把这个知识积累下来，去寻找一些方法使这个过程自动化。

对于这种由监控所驱动的自动化运维，一般由四个步骤组成，分别是了解、决策、执行，还有记录与反馈，如图 1.2 所示。

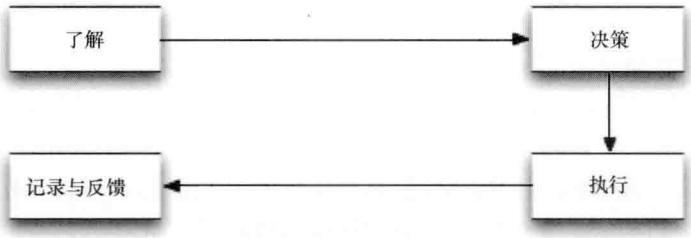


图 1.2　由监控驱动的自动化运维

第一步是要对设备、业务等我们所关注的对象进行监控。我们可以把监控看作一个了解信息的过程，只有当我们了解了实际情况之后，我们才能进行后续的操作。

第二步是决策。决策这个过程定义了在什么情况下我们应该执行什么操作的规则。例如，哪些主机磁盘满了需要做删除日志文件的操作，哪些主机磁盘满了需要做归档日志文件的操作，这些都是需要在决策过程中去定义的。

第三步是执行。我们根据之前的规则做出相应的决策之后，然后就可以开始执行相应的运维操作了。这个过程我们需要关注的是如何可以屏蔽异构的操作系统给我们带来的不便，也就是怎么样能让异构操作系统的差异对执行的动作来说是透明的。

最后一步是记录和反馈。在完成了具体的维护操作以后，我们需要有一套记录的机制，用于查询究竟在什么时候做出了什么样的运维操作，并且在执行完成之后能够通过短信、邮件等方式给运维人员发出通知，让运维人员能够在执行完之后马上知道系统究竟执行了

哪些运维操作，结果是怎么样的。

对于了解、决策、记录和反馈这几个过程来说，目前开源软件中做得最好的就算 Zabbix 了，它有丰富的告警策略、多种设备监控、强大的动作管理的功能，并且还具备了十分强大的扩展能力。而对于执行这个动作，虽然 Zabbix 也可以做到，但是 Zabbix 没有对异构操作系统进行集中管理的能力，并且也没有人为它封装相应的运维模块，所以这个环节一般情况下更倾向于选择像 SaltStack、Ansible 或者 Puppet 这些集中化运维工具。各环节选用的开源软件如图 1.3 所示。

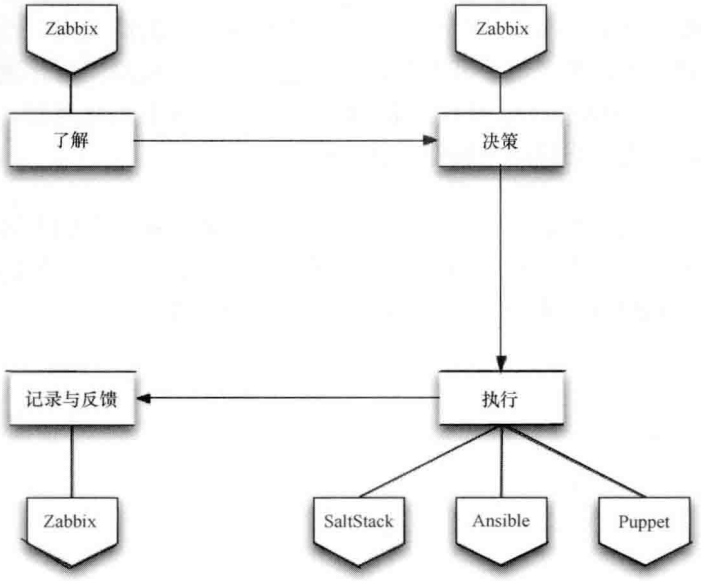


图 1.3　各环节选用的开源软件

接下来的章节会介绍如何对设备进行集中化运维，如何整合 Zabbix 实现自动化运维；最后讲解在极端恶劣的环境下如何通过 OSGi+MQ 的架构去设计一个易部署、易开发、跨平台性强的集中化运维框架并分享相应的案例。

1.5　小结

没有一个非常好的定义能说明什么样的运维程度才能算得上是自动化运维，有时候我

们会认为能够批量地操作主机就算是自动化运维，能够定期地出具巡检报告就算是自动化运维，或者说能够根据监控的动作去触发某些指定的动作就是自动化运维。笔者认为这些都没错，其实一切能让我们把人工的运维操作交给计算机去完成的运维工具，都算得上是自动化运维工具。不同的人对同一件事情会有相距甚远的看法，主要是因为各自接触的业务领域不同罢了。

笔者在做一款业务运维的自动化运维软件之前，认为 Puppet、SaltStack、Ansible 这样的运维软件就是自动化运维软件，使用这些软件就能够让我们的运维达到自动化的程度。当和同事讨论这款软件怎么和监控软件整合在一起的时候，突然又觉得自动化运维重要的一点是要和监控结合起来才叫自动化运维，让主机根据自身的情况为自己解决问题，这才叫自动化运维。到后来，笔者着手开发基础业务运维系统（按照同事开玩笑的话就是“搞机的”）的时候，又觉得能够让流程去驱动运维工作，那才叫自动化运维，例如能够让服务器上下架等工作具备更多的自动化操作，更少的人工干预。

所以说，自动化运维的范围非常广，一本书也无法涵盖如此多的内容。本书后续章节主要与读者分享为什么需要重新造一个轮子，如何设计一套跨平台、易部署的自动化运维工具，以及如何与 Zabbix 进行整合，让监控去触发运维作业。

第2章 集中化运维利器——Ansible

Ansible 是集中化运维工具的代表之一，它提供了系统配置、软件部署和流程化 IT 运维任务等功能。它是一款命令式的集中化运维软件。前面我们也讨论过，命令式的集中化运维软件所描述的是一个运维操作应该怎么去做，它更关心的是应该如何去完成一个任务。

与 Puppet 这类具备 Agent 端的运维软件相比，Ansible 具备了无 Agent 端的优势，采用 SSH 协议进行通信，也就是说目标主机上只要有 SSH 就可以用了。这种设计方式一方面避免了 Agent 升级的问题，另一方面也避免了第一次安装时需要集中部署客户端所带来的巨大工作量的问题。

与其他运维软件相比，Ansible 的学习曲线没那么陡峭，十分容易上手。下面讲解 Ansible 的安装、基础使用方式、常用模块、PlayBook 以及 AnsibleAPI 的使用，让读者能够快速地对 Ansible 有一个基础性的了解。

2.1 环境准备

首先，我们需要对 Ansible 的基础功能有一个大概的了解，知道它究竟能做一些什么样的运维操作。我们会进行一系列的实践，所以请读者准备两台虚拟机以方便后续的学习。

实验环境配置如表 2.1 所示。

表 2.1　Ansible 的实验环境

操作系统	IP	作　用
CentOS6.4	192.168.41.136	Ansible Server
CentOS6.4	192.168.41.139	运维目标主机

整个实验的环境结构如图 2.1 所示。

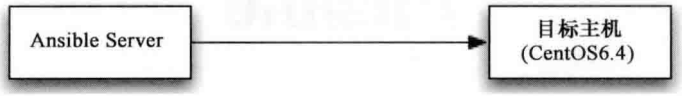


图 2.1　Ansible 实验环境

为了能让读者更好地进行实验，了解 Ansible 的用途以及所提供的功能，上面提供的是一个简化了的实践环境。由于 Ansible 是不需要 Agent 端的集中化运维工具，它是通过 SSH 方式与远端主机交互的。我们只要确保运维目标主机上的 SSH 服务是开启的，并且 22 端口是对外放通的就可以了。在实际运维环境中，目标主机的数量不会只有几台这么少，往往都是成百上千台，这个实验环境只是一个大环境下的缩影。笔者曾经在实际生产环境所接触到的一个安全系统背后就有 800 多台设备在做支持。

2.2　安装 Ansible

安装 Ansible 的方式有很多种，下面介绍使用 CentOS 的 EPEL 源进行安装和采用源码进行编译这两种方式。

2.2.1　使用 CentOS 的 EPEL 源进行安装

这是一种能够最快速接触并感受到 Ansible 方便的安装方法，我们无须烦恼 Ansible 安装时的依赖包问题。

安装 EPEL 源：

```
rpm-ivh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
```

Enterprise Linux 额外软件包（Extra Packages for Enterprise Linux，EPEL）是由来自 Fedora Project 的志愿者发起的社区力量，为了创建由高质量的附加软件组成的、用于补足 RHEL 和其他兼容版本的软件仓库。

通过 Yum 源安装 Ansible：

```
yum install ansible
```

测试 Ansible 是否安装成功：

```
ansible all -m ping
```

当 Ansible 安装成功后，执行上述命令就会打印出类似下面的结果：

```
alpha.example.org | FAILED => FAILED: [Errno -2] Name or service not known
blue.example.com | FAILED => FAILED: [Errno -2] Name or service not known
beta.example.org | FAILED => FAILED: [Errno -2] Name or service not known
green.example.com | FAILED => FAILED: [Errno -2] Name or service not known
192.168.100.1 | FAILED => FAILED: timed out
192.168.100.10 | FAILED => FAILED: timed out
www001.example.com | FAILED => FAILED: [Errno -2] Name or service not known
192.168.1.100 | FAILED => FAILED: timed out
www002.example.com | FAILED => FAILED: [Errno -2] Name or service not known
www003.example.com | FAILED => FAILED: [Errno -2] Name or service not known
192.168.1.110 | FAILED => FAILED: timed out
www006.example.com | FAILED => FAILED: [Errno -2] Name or service not known
www005.example.com | FAILED => FAILED: [Errno -2] Name or service not known
www004.example.com | FAILED => FAILED: [Errno -2] Name or service not known
db-99-node.example.com | FAILED => FAILED: [Errno -2] Name or service not known
db-100-node.example.com | FAILED => FAILED: [Errno -2] Name or service not known
10.25.1.56 | FAILED => FAILED: timed out
db02.intranet.mydomain.net | FAILED => FAILED: timed out
db01.intranet.mydomain.net | FAILED => FAILED: timed out
db-101-node.example.com | FAILED => FAILED: [Errno -2] Name or service not known
10.25.1.57 | FAILED => FAILED: timed out
```

上面这些 IP 是从哪里来的呢？我们暂且先不管，我们只需要知道 Ansible 管理节点已经装好，可以开始下面的学习就够了。

2.2.2 使用 Easy_Install 安装 Ansible

Ansible 其实可以看作一个 Python 的模块，所以当然也能够通过 Easy_Install 进行安装，下面的安装过程使用的环境依然是 CentOS。

> easy_install 是由 PEAK（Python Enterprise Application Kit）开发的 setuptools 包里的一个命令，所以使用 easy_install 实际上是在调用 setuptools 来完成安装模块的工作。Perl 用户比较熟悉 CPAN，而 Ruby 用户则比较熟悉 Gems；引导 setuptools 的 ez_setup 工具和随之而生的扩展后的 easy_install 与 "Cheeseshop"（Python Package Index，也称为 PyPI）一起工作来实现相同的功能。它可以很方便地让用户自动下载、编译、安装和管理 Python 包。

安装 python-setuptools：

```
yum install python-setuptools
```

安装 Ansible：

```
easy_install ansible
```

假如在安装过程中出现了如下错误，请检查一下是否安装了 python-devel：

```
src/MD2.c:31:20: fatal error: Python.h: No such file or directory
#include "Python.h"
```

2.3 Ansible 基础

2.3.1 资产配置

我们在执行完"ansible all -m ping"这一条命令之后，发现出现了许多主机信息，这个时候我们就会有这样一个疑问，这些主机信息都是从哪里来的呢？

Ansible 默认会把所管理的目标主机信息存放于/etc/ansible/hosts 中，打开 hosts 文件，就可以看到如下所示的信息了。

```
green.example.com
blue.example.com
192.168.100.1
192.168.100.10

[webservers]
alpha.example.org
beta.example.org
192.168.1.100
192.168.1.110

[dbservers]
db01.intranet.mydomain.net
db02.intranet.mydomain.net
10.25.1.56
10.25.1.57


db-[99:101]-node.example.com
```

这是 Ansible 的资产配置文件，定义了究竟 Ansible 需要管理哪些主机，第 1～4 行声明了我们需要管理的主机有四台，可以看到配置 IP 和域名都是可行的。

```
green.example.com
blue.example.com
192.168.100.1
192.168.100.10
```

接下来的这段配置信息有点不一样，它们都有[webservers]和[dbservers]这样的标签，这表明这些主机分别属于[webservers]组和[dbservers]组。这样配置目标设备分类的主要目的在于为服务器进行分类，把属于数据库的服务器分为一类，把属于中间件的服务器分为另一类。当我们需要对某类服务器进行操作的时候，就可以直接指定这一类主机所在的主机组就可以了，例如要对 dbservers 的主机进行集中化重启，那么只需要把组名交给 Ansible 让它去执行就可以了。

```
[webservers]
alpha.example.org
beta.example.org
```

```
192.168.1.100
192.168.1.110

[dbservers]
db01.intranet.mydomain.net
db02.intranet.mydomain.net
10.25.1.56
10.25.1.57
```

最后一段配置信息比较特别，它表明从 db-99-node.example.com 到 db-101-node.example.com 的这一批主机也是受 Ansible 管理的。

```
db-[99:101]-node.example.com
```

这种配置非常适合于主机 IP 或者域名比较有规律的设备，例如我们有一批位于 192.168.1.100 到 192.168.1.200 这个 IP 端的设备需要使用 Ansible 进行集中运维，这个时候只需要在配置文件里面配置 192.168.1.[100:200]就可以了。

理解 Ansible 的配置文件的设置方式之后，把 hosts 文件清空，然后把实践所用的主机 192.168.41.139 加入到 hosts 文件中。

1. 常用的资产配置方式

Ansible 的资产配置有许多种方法，我们挑一些比较常用的配置方法来进行学习。

（1）指定连接设备所使用的 SSH 端口。

有些时候目标客户机的 SSH 端口没有开在 22 端口上，这个时候就需要使用指定 SSH 端口的方式去连接目标主机。下面的配置说明了 Ansible 在操作 192.168.41.139 这台服务器时需要使用 5309 这个端口。

```
192.168.41.139:5309
```

（2）指定设备的别名。

指定设备别名的原因是为了减少在设备数量很多时的记忆成本，可以在资产配置信息里面指定设备的别名。下面的配置表示目标主机为 192.168.1.50，使用的端口是 5555，这台主机的别名是 oracleServer。

```
oracleServer ansible_ssh_port=5555 ansible_ssh_host=192.168.1.50
```

（3）采用通配符进行配置。

通配符配置的方式可以帮助我们减少连续的主机名或 IP 的配置工作量。下面的例子分别声明了运维的目标主机为从 www.01.example.com 到 www.50.example.com 的主机。

```
www.[01:50].example.com
```

除了支持以数字作为范围之外，还支持用英文进行范围的声明。下面的例子声明了运维的目标主机是从 db-a.example.com 到 db-f.example.com 的主机。

```
db-[a:f].example.com
```

2. 资产配置中的常用变量

Ansible 所提供的资产配置常用的变量如表 2.2 所示，值得留意的是 ansible_ssh_pass 这个变量，由于 Ansible 是采用 SSH 与目标主机进行通信的，当没有指定 SSH 密码的时候，Ansible 会从认证的 Key 里面找，通过双机互信的方式进行 SSH 的连接。当指定了 ansible_ssh_user 和 ansible_ssh_pass 这两个参数之后，Ansible 就会采用 sshpass 进行 SSH 的认证了。

表 2.2 Ansible 所提供的资产配置常用变量

变　量	作　用
ansible_ssh_host	指定远程主机
ansible_ssh_port	指定 SSH 连接端口
ansible_ssh_user	指定 SSH 连接用户
ansible_ssh_pass	指定 SSH 连接密码
ansible_python_interpreter	指定目标主机 Python 路径

OpenSSH 自带的 SSH 客户端程序（也就是 'ssh' 命令）默认不允许以非交互的方式传递密码，sshpass 的出现解决了这一问题。它允许我们用 -p 参数指定明文密码，然后直接登录远程服务器。例如：sshpass -p 'ssh_password' ssh www.iredmail.org。

2.3.2 执行命令

在了解了主机资产的配置方法之后，我们先把/etc/ansible/hosts 中的内容清空，加入如下信息，表明我们需要用用户名为 root、密码为 1q2w3e4r 的账号去对 IP 为 192.168.41.139 的主机进行操作：

```
192.168.41.139 ansible_ssh_user=root ansible_ssh_pass=1q2w3e4r
```

配置完成后，输出第一条命令：

```
ansible all -a "echo HelloWorld"
```

我们期望目标主机会返回 HelloWorld 的结果，但是实际上 Ansible 的执行却卡在了询问是否记录 SSH 密钥的位置：

```
paramiko: The authenticity of host '192.168.41.139' can't be established.
The ssh-rsa key fingerprint is 75c4e8af474a9ecbae1e32d3c2550306.
Are you sure you want to continue connecting (yes/no)?
```

这是由于 Ansible 检查服务器存放的 SSH-KEY 造成的，因为没有添加双机互信，所以 Ansible 就卡在了询问是否记录 SSH 密钥的位置了。可以通过修改配置文件的方式来解决这个问题，不要让 Ansible 对 SSH-KEY 进行检查就可以了。修改配置文件/etc/ansible/ansible.cfg，把#host_key_checking = False 这一句的注释去掉，然后再执行一次命令，就能得到正确的结果了。

```
192.168.41.139 | success | rc=0 >>
HelloWorld
```

2.3.3 指定目标主机

Ansible 命令的基本语法格式如下所示。pattern 参数声明了需要操作的目标主机，module_name 声明了需要使用的模块是哪一个，最后一个参数 arguments 用于传递参数给模块。

```
ansible <pattern > -m <module_name> -a <arguments>
```

指定所有的主机：

```
ansible all -m shell -a "uptime"
```

指定特定的主机组：

```
ansible webservers:dbservers -m shell -a "uptime"
```

排除指定的主机组：

```
ansible webservers:!phoenix -m shell -a "uptime"
```

指定同时存在于两个组的主机：

```
ansible webservers:&staging -m shell -a "uptime"
```

复合条件：

```
ansible webservers:dbservers:&staging:!phoenix -m shell -a "uptime"
```

采用正则表达式指定主机：

```
ansible one*.com:dbservers -m shell -a "uptime"
```

2.3.4 常用命令示例

以下的示例主要目的是为了让读者对 Ansible 的常用模块有一个大概的了解，读者可以在自己的环境上亲自做一下，有一个大致的印象就可以了。

1. 命令执行

重启主机：

```
ansible all -a "/sbin/reboot" -f 10
```

Shell 模块：

```
ansible all -m shell -a 'echo $TERM'
```

底层 SSH 模块：

```
ansible all -m raw -a "hostname --fqdn"
```

2. 文件操作

下发文件：

```
ansible all -m copy -a "src=/etc/hosts dest=/tmp/hosts"
```

为文件赋予指定的权限：

```
ansible all -m file -a "dest=b.txt mode=600 owner=demo group=demo"
```

创建文件夹：

```
ansible all -m file -a "dest=/path/to/c mode=644 owner=mdehaan
group=mdehaan state=directory"
```

删除文件：

```
ansible all -m file -a "dest=/path/to/c state=absent"
```

3. 包管理模块

使用 Yum 源进行安装：

```
ansible webservers -m yum -a "name=acme state=installed"
```

安装指定版本的包：

```
ansible webservers -m yum -a "name=acme-1.5 state=installed"
```

安装最新版本的安装包：

```
ansible webservers -m yum -a "name=acme state=latest"
```

卸载安装包：

```
ansible webservers -m yum -a "name=acme state=removed"
```

4. 用户管理

新增用户：

```
ansible all -m user -a "name=demo password=1q2w3e4r"
```

删除用户：

```
ansible all -m user -a "name=foo state=absent"
```

5. 版本管理

使用 Git 拉取文件：

```
ansible all -m git -a "repo=git://demo/repo.git dest=/srv/myapp
version=HEAD"
```

6. 服务管理

启动服务：

```
ansible webservers -m service -a "name=httpd state=started"
```

重启服务：

```
ansible webservers -m service -a "name=httpd state=restarted"
```

停止服务：

```
ansible webservers -m service -a "name=httpd state=stopped"
```

7. 后台管理

启动一个执行 360 秒的后台作业：

```
ansible all -B 360 -a "/usr/bin/long_running_operation --do-stuff"
```

检查作业状态：

```
ansible all -m async_status -a "jid=1311"
```

后台运行 1800 秒，每 60 秒检查一次作业状态：

```
ansible all -B 1800 -P 60 -a "/usr/bin/long_running_operation --do-stuff"
```

8. 设备信息查询

获取设备的信息列表：

```
ansible all -m setup
```

2.4 Ansible 常用模块

Ansible 有许多现成的模块可供选用，我们可以使用“ansible-doc –l”这个命令来查看 Ansible 里面内置的模块。下面仅对几个常用的模块进行介绍并对每个模块的使用场景做一些简要的说明。剩余的模块，在工作的时候用到了，查一下就可以。

2.4.1 文件管理模块

1. 文件组装模块——assemble

assemble 模块用于把多份配置文件片段组装成一份配置文件，当需要对不同的主机分配不同的配置文件时，可以考虑使用此模块。组装的方式如图 2.2 所示。

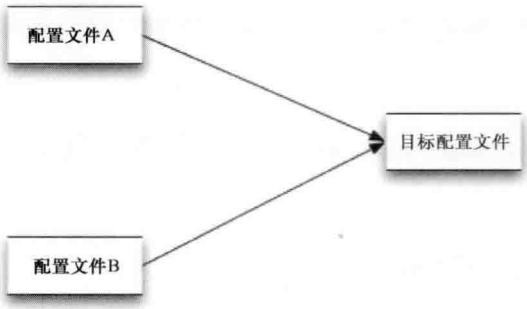


图 2.2　组装的方式

例：将/root/demo 下的片段文件组装后放到/root/target 目录下。

```
ansible all -m assemble -a 'dest=/root/target src=/root/demo'
```

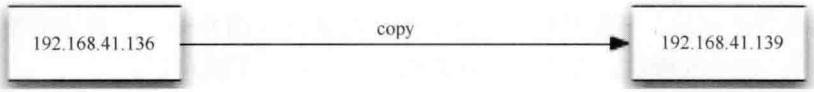
assemble 模块参数及说明如表 2.3 所示。

表 2.3　assemble 模块参数及说明

参　数	必　填	默 认 值	选　项	说　明
backup	否	no	yes/no	是否创建一份备份文件
delimiter	否	—	—	配置文件片段之间的分隔符
dest	是	—	—	生成路径
others	否	—	—	文件模块参数
src	是	—	—	片段文件夹路径

2．文件复制模块——copy

文件复制的模块常用于做集中下发的动作，假如远端主机装有 SELinux，那么还需要在目标主机上安装 libselinux-python 模块才能使用 copy 模块。



SELinux 是一种基于域-类型模型（domain-type）的强制访问控制（MAC）安全系统，它由 NSA 编写并设计成内核模块包含到内核中，相应的某些安全相关的应用也被打了 SELinux 的补丁。最后还有一个相应的安全策略，任何程序对其资源享有完全的控制权。假设某个程序打算把含有潜在重要信息的文件放到/tmp 目录下，那么在 DAC 情况下没人能阻止它。SELinux 提供了比传统的 UNIX 权限更好的访问控制。

例：将/root/demo/copydemo.txt 复制到所有机器的/root 目录下。

```
ansible all -m copy -a 'dest=/root src=/root/demo/copydemo.txt'
```

copy 模块参数及说明如表 2.4 所示。

表 2.4 copy 模块参数及说明

参　数	必　填	默认值	选　项	说　明
backup	否	no	yes/no	是否备份原始文件
content	否	—	—	当用 content 代替 src 参数时，可以把文档的内容设置为特定的值
dest	是	—	—	文件复制的目的地
force	否	no	yes-/no	是否覆盖
others	否	—	—	文件模块参数
src	否	—	—	复制的源文件
validate	否	—	—	复制前是否检验需要复制目的地的路径

3. 文件拉取模块——fetch

fetch 模块和 copy 模块很类似，都是对文件进行复制，但是 fetch 模块的作用是把被管理节点的文件批量地复制到服务器上，可以看作一个文件上传的动作。

例：将远端机器的/etc/salt/minion 文件收集回服务器的/root/demo 目录下。

```
ansible all -m fetch -a 'dest=/root/demo src=/etc/salt/minion'
```

fetch 模块参数及说明如表 2.5 所示。

表 2.5 fetch 模块参数及说明

参　数	必　填	默认值	选　项	说　明
dest	是	—	—	文件存放路径，假如存放路径是/backup，复制的源文件为/etc/profile，目标主机名是 host，那么就会被存为/backup/host/etc/profile 下
fail_on_missing	否	no	yse/no	假如找不到目标文件则标记为失败

（续表）

参　数	必　填	默认值	选　项	说　明
flat	否	—	—	用于覆写原有的 dest 存放规则
validate_md5	否	no	yse/no	是否用 md5 进行文件的校验
src	是	—	—	目标文件路径

4．文件管理模块——file

文件自身有许多属性，例如修改文件所属的用户组、文件所属的用户、是否需要删除文件，这些都是我们平常需要使用的功能，而 file 模块就是为完成上述这些功能而准备的。

例：删除所有服务器下的/root/copydemo.txt 文件。

```
ansible all -m file -a 'path=/root/copydemo.txt state=absent'
```

file 模块参数及说明如表 2.6 所示。

表 2.6　file 模块参数及说明

参　数	必　填	默认值	选　项	说　明
force	否	no	yse/no	是否覆盖原有文件
group	否	—	—	文件属于的用户组
mode	否	—	—	文件的读/写权限
owner	否	—	—	文件的属于的用户
path	是	—	—	文件路径
recurse	否	no	yse/no	是否递归设置属性
selevel	否	s0	—	selinux 的级别
serole	否	—	—	selinux 角色
setype	否	—	—	selinux 类型
seuser	否	—	—	selinux 用户
src	否	—	—	文件链接路径
state	否	file	file link directory hard touch absent	如果是 directory，那么则会创建文件夹 如果是 file，则会创建文件 如果是 link，则会创建链接 如果是是 hard，则创建硬链接 如果是 touch，则会创建一份文件 如果是 absent，则会删除文件

5．ini 文件管理模块

ini 文件是十分常见的一种配置文件，Ansible 内置了 ini 配置文件的管理模块，用于对

ini 文件进行配置项的管理。

> 🔊 ini 文件由节（section）、参数（name=value）和注释所组成，格式如下：
>
> [section]
>
> name=value ;this is comment

例：修改/root/demo.ini 配置文件，selection 为 cron 的选项组的 crontime 选项，把 cron 的值修改为 10。

```
ansible all -m ini_file -a 'dest=/root/demo.ini section=cron
option=crontime value=10'
```

ini 模块参数及说明如表 2.7 所示。

表 2.7 ini 模块参数及说明

参 数	必 填	默 认 值	选 项	说 明
backup	否	no	yse/no	是否创建备份文件
dest	是	—	—	ini 文件路径
option	否	—	—	ini 文件的键选项
others	否	—	—	文件模块的其他参数
section	是	—	—	选中 ini 的变量名
value	否	—	—	ini 变量的值

2.4.2 命令执行模块

1. 命令执行模块——command

command 模块用于在给定的主机上执行命令。值得留意的是，command 模块执行的命令是获取不到$HOME 这样的环境变量的，一些运算符，例如“>”、“<”在 command 模块上也是不能使用的。

command 模块参数及说明如表 2.8 所示。

表 2.8 command 模块参数及说明

参 数	必 填	默 认 值	选 项	说 明
chdir	否	—	—	执行命令前先进入到某个目录
creates	否	—	—	一个文件名，假如文件名已经存在，则不会运行此步骤

（续表）

参　数	必　填	默认值	选　项	说　明
executable	否	—	—	改变执行命令所用的 shell
free_form	是	—	—	需要执行的指令
removes	否	—	—	一个文件名，假如不存在该文件，则此步骤不会被执行

2. command 模块的增强版——shell

command 模块是不支持运算符的，同时也不支持管道这样的操作。我们使用 command 模块可以尝试执行一下检查 MySQL 进程是否存在的命令。

```
ansible all -m command -a 'ps -ef|grep mysql'
```

执行完后，Ansible 会发出不支持操作符的提示，而 shell 模块是专门解决这种问题的，它支持管道操作符，可以看作 command 模块的增强版，可以使用如下方式对 MySQL 进程进行检查就可以得到正确的结果。

```
ansible all -m shell -a 'ps -ef|grep mysql'
```

shell 模块参数及说明如表 2.9 所示。

表 2.9　shell 模块参数及说明

参　数	必　填	默认值	选　项	说　明
chdir	否	—	—	执行命令前先进入到某个目录
creates	否	—	—	一个文件名，假如文件名已经存在，则不会运行此步骤
free_form	是	—	—	需要执行的指令
removes	否	—	—	一个文件名，假如不存在该文件，则此步骤不会被执行

3. 脚本执行模块——script

很多时候执行单条命令并不能满足我们的需求，我们需要在目标主机上执行一系列的命令。这种情况下可以考虑把多条命令写成脚本，然后通过 Ansible 的文件管理模块把脚本下发到目标机器上；接着再用 script 模块进行脚本的执行，得到所期望的结果。需要注意的是执行的脚本是在管理主机上存在的脚本。

例如我们有一份巡检脚本，已经通过 copy 模块把脚本下发到了主机上，并且用 file 模块完成了脚本文件的授权。脚本 inspection.sh 内容如下：

```
#!/bin/bash
phy_cpu='cat /proc/cpuinfo |grep "physical id"|sort |uniq|wc -l'
logic_cpu_num='cat /proc/cpuinfo |grep "processor"|wc -l'
cpu_core_num='cat /proc/cpuinfo |grep "cores"|uniq|awk -F: '{print $2}''
cpu_freq='cat /proc/cpuinfo |grep MHz |uniq |awk -F: '{print $2}''
system_core='uname -r'
system_version='head -n 1 /etc/issue'
system_hostname='hostname'
system_envirement_variables='env |grep PATH'
mem_free='grep MemFree /proc/meminfo'
disk_usage='df -h'
system_uptime='uptime'
system_load='cat /proc/loadavg'
system_ip='ifconfig |grep "inet addr"|grep -v "127.0.0.1" |awk -F:
'{print $2}' |awk '{print $1}''
mem_info=`/usr/sbin/dmidecode | grep -A 16 "Memory Device" | grep -E
"Size|Locator" | grep -v Bank'
mem_total='grep MemTotal /proc/meminfo'

day01='date +%Y'
day02='date +%m'
day03='date +%d'

path=inspection.txt
echo -e ""> $path
echo -e    $day01年$day02月$day03系统巡检报告 >> $path
echo -e 服务器IP: "\t"$system_ip >> $path
echo -e 主机名: "\t"$system_hostname >> $path
echo -e 系统内核: "\t" $system_core >> $path
echo -e 操作系统版本:"\t" $system_version >> $path
echo -e 磁盘使用用情况: "\t""\t" $disk_usage >> $path
echo -e CPU核数: "\t" $cpu_core_num >> $path
echo -e 物理CPU个数: "\t" $phy_cpu >> $path
echo -e 逻辑CPU个数: "\t" $logic_cpu_num >> $path
echo -e CPU的主频:"\t" $cpu_freq >> $path
echo -e 系统环境变量:"\t" $system_envirement_var >> $path
echo -e 内存简要信息: "\t" $mem_info >> $path
echo -e 内存总大小: "\t" $mem_total >> $path
echo -e 内存空闲: "\t" $mem_free >> $path
echo -e 时间/系统运行时间/当前登陆用户/系统过去1分钟/5分钟/15分钟内平均负载/"\t"
```

```
$system_uptime >> $path
echo -e  1分钟/5分钟/15分钟平均负载/在采样时刻，运行任务的数目/系统活跃任务的个数/最大的pid值线程/ "\t" $system_load >> $path
```

可以看到整份脚本的命令数量是比较多的，采用 shell 模块或者 command 模块都无法很好地完成这个巡检任务。这时可以用 script 模块完成主机的批量巡检。

```
ansible all -M script -a '/root/demo/inspection.sh'
```

script 模块参数及说明如表 2.10 所示。

表 2.10　script 模块参数及说明

参　数	必　填	默认值	选　项	说　明
free_form	是	—	—	需要执行的脚本

4. SSH 命令执行模块——raw

相信大家都还记得前面所提到的 Ansible，虽然不需要安装客户端，但是内置的模块大多需要客户机上有 Python 环境或者具备某些 Python 扩展才能够执行。假设我们管理的设备上没有 Python 环境，Ansible 的很多模块都用不了，但是又想执行一些简单的命令，怎么办呢？这个时候就可以使用 raw 模块了，这个模块是直接通过 SSH 的方式而不是通过 Python 的方式去对目标主机进行操作的。

例如，可以通过 raw 模块直接进行一些简单的命令：

```
ansible all -m raw -a 'ip a'
```

raw 模块的参数及说明如表 2.11 所示。

表 2.11　raw 模块的参数及说明

参　数	必　填	默认值	选　项	说　明
executable	否	—	—	改变执行命令所用的 shell
free_form	是	—	—	需要执行的指令

2.4.3　网络相关模块

1. 下载模块——get_url

get_url 模块用于下载网络上的文件。

例：下载百度的首页。

```
ansible all -m get_url -a 'dest=/root url=http://www.baidu.com'
```

get_url 模块参数及说明如表 2.12 所示。

表 2.12 get_url 模块参数及说明

参 数	必 填	默 认 值	选 项	说 明
dest	是	—	—	文件下载路径
force	否	no	yes/no	是否覆盖
others	否	—	—	文件模块的其他参数
sha256sum	否	—	—	是否采用 SHA-256 校验和
url	是	—	—	下载文件的目标路径
use_proxy	否	no	yes/no	是否使用代理

2. Web 请求模块——uri

uri 模块主要用于发送 HTTP 协议，通过使用 uri 模块，可以让目标主机向指定的网站发送如 Get、Post 这样的 HTTP 请求，并且能得到返回的状态码。uri 模块参数及其说明如表 2.13 所示。

表 2.13 uri 模块的参数及说明

参 数	必 填	默 认 值	选 项	说 明
HEADER_	否	—	—	HTTP 头
Body	否	—	—	HTTP 消息体
Creates	否	—	—	文件名称
Dest	否	—	—	文件下载路径
follow_redirects	否	no	yes/no	URI 模块是否应该遵循所有的重定向
force_basic_auth	否	no	yes/no	强制在发送请求前发送身份验证
method	否	GET	GET POST PUT HEAD DELETE OPTIONS	HTTP 方法

（续表）

参　数	必　填	默认值	选　项	说　明
others	否	—	—	文件模块参数
password	否	—	—	密码
removes	否	—	—	需要删除的文件名称
return_content	否	no	yes/no	返回内容
status_code	否	200	—	状态码
timeout	否	30	—	超时限制
url	是	—	—	url 地址
user	否	—	—	用户名

2.4.4　源码管理模块

1. Git

当我们需要做文件集中下发的时候，除了可以用 copy 这样的方式之外，其实还可以用源码管理模块来实现，Git 是目前比较流行的版本管理工具之一。

> Git 是一款免费、开源的分布式版本控制系统。它是 Linus Torvalds 为了帮助管理 Linux 内核而开发的一个开放源码的版本控制软件。
>
> Torvalds 开始着手开发 Git 是为了作为一种过渡方案来替代 BitKeeper，后者之前一直是 Linux 内核开发人员在全球使用的主要源代码工具。开放源码社区中的有些人觉得 BitKeeper 的许可证并不适合开放源码社区的工作，因此 Torvalds 决定着手研究许可证更为灵活的版本控制系统。

在了解 Git 模块所提供的功能之前，我们先快速地了解一下 Git 这款工具，它与传统的集中式版本管理工具不一样，它有本地仓库这个概念，如图 2.3 所示。

图 2.3　Git 的本地仓库与远端仓库

每当我们做资料的提交的时候，都会先提交到本地仓库，然后再提交到远端仓库。正是因为有了本地仓库这一概念，使得 Git 可以脱离服务器进行版本的控制，等到需要提交

了，或者网络能连接上服务器的时候，再从本地仓库进行一次批量提交。提交到本地仓库的动作叫作 Commit，从本地仓库提交到服务器上的动作叫作 Push，而从服务器上拉取最新版本的动作叫作 Pull。理解了这几个名词之后，我们再看看图 2.4。

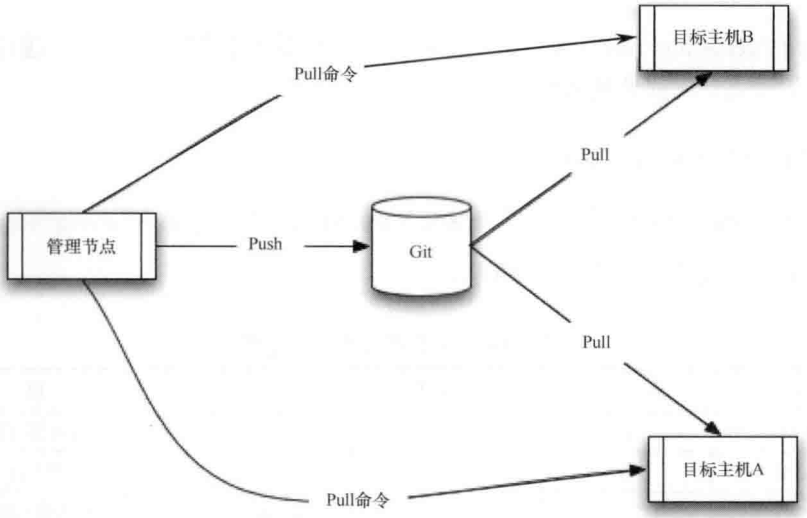


图 2.4　使用 Git 模块完成文件管理

首先，我们在管理节点上把需要下发的文件 commit 到本地，然后通过 Push 操作将文件传送到远端的 Git，接着通过 Git 模块命令目标主机执行 Pull 命令。通过这样一系列的操作，完成文件从管理节点到目标主机的传输过程。Git 模块参数及说明如表 2.14 所示。

表 2.14　Git 模块参数及说明

参　数	必　填	默 认 值	选　项	说　明
depth	否	—	—	创建一个浅克隆历史阶段到指定的版本
dest	是	—	—	代码克隆的位置
executable	否	—	—	Git 的可执行路径
force	否	no	yes/no	是否强制更新
remote	否	origin	—	远端的版本
repo	是	—	—	仓库地址
update	否	no	yes/no	是否更新远端仓库
version	否	HEAD	—	迁出的版本

2.4.5 包管理模块

Ansible 支持多种包管理模块，此处以 APT 和 YUM 为例。

1. APT

Advanced Packaging Tool（APT）是 Linux 下的一款安装包管理工具。通过使用它，我们可以非常容易地完成软件的安装。

例：使用 APT 模块安装 JDK

```
apt: name=openjdk-6-jdk state=latest install_recommends=no
```

APT 模块参数及说明如表 2.15 所示。

表 2.15 APT 模块参数及说明

参　数	必　填	默认值	选　项	说　明
cache_valid_time	否	—	—	校验缓存是否过期
default_release	否	—	—	等同于 apt –t
force	否	no	yes/no	是否强制更新或删除
install_recommends	否	true	yes/no	等同于 apt –no-install-recommends
pkg	否	—	—	包名
purge	否	—	yes/no	卸载后是否清除配置文件
state	否	present	latest absent present	更新、删除、安装包
update_cache	否	—	—	安装前是否更新缓存
upgrade	否	yes	yes safe full dist	safe 等同于 safe-upgrade full 等同于 full-upgrade dist 等同于 dist-upgrade

2. YUM

YUM（全称为 Yellow dog Updater，Modified）是一个在 Fedora 和 RedHat 以及 CentOS 中的 shell 前端软件包管理器。基于 RPM 包管理，能够从指定的服务器自动下载 RPM 包并安装，可以自动处理依赖性关系，并且一次安装所有依赖的软件包，无须烦琐地一次次下载、安装。

例：使用 YUM 模块安装最新的 httpd。

```
yum: name=httpd state=latest
```

YUM 模块参数及说明如表 2.16 所示。

表 2.16 YUM 模块参数及说明

参 数	必 填	默 认 值	选 项	说 明
conf_file	否	—	—	YUM 配置文件
disable_gpg_check	否	no	yes/no	是否开启 GPG 检查
disablerepo	否	—	—	禁用的仓库
enablerepo	否	—	—	启用的仓库
list	否	—	—	非幂等性命令
name	是	—	—	包名
state	否	present	present latest absent	安装、更新、卸载操作

2.4.6 系统管理模块

Ansible 带有许多常用的系统管理模块，我们只讲解几个最常用的。

1. 作业管理模块——cron

在 Linux 服务器中，定时任务一般由 cron 来承担，我们只需要按照 cron 所要求的格式配置好相关的参数，系统就会帮我们自动对作业进行定期的调度。

在用户所建立的 crontab 文件中，每一行都代表一项任务，每行的每个字段代表一项设置，它的格式共分为六个字段，前五段是时间设定段，第六段是要执行的命令段，格式如下：

```
minute   hour   day   month   week   command
```

其中：

- minute 表示分钟，可以是从 0 到 59 之间的任何整数；
- hour 表示小时，可以是从 0 到 23 之间的任何整数；
- day 表示日期，可以是从 1 到 31 之间的任何整数；
- month 表示月份，可以是从 1 到 12 之间的任何整数；

- week 表示星期几，可以是从 0 到 7 之间的任何整数，这里的 0 或 7 代表星期日；
- command 是要执行的命令，可以是系统命令，也可以是自己编写的脚本文件。

例如我们需要每分钟执行一次命令，那么就可以在 Crontab 中加入如下配置：

```
* * * * * command
```

当管理的主机数目比较多的时候，人工到每台主机上进行 Crontab 的配置效率显然不高效，这时就可以使用 Ansible 的 cron 模块了。

例：每天 8 点进行 MySQL 数据库的备份。

```
ansible all -m cron -a 'name=demo hour=8 job= mysqldump -uroot -pxxxx
demo>demo.sql'
```

cron 模块参数及说明如表 2.17 所示。

表 2.17　cron 模块参数及说明

参　数	必　填	默 认 值	选　项	说　明
backup	否	—	—	修改前是否备份
cron_file	否	—	—	cron 定义文件，指定了则读这份，不会读用户的 cron.d 文件
day	否	*	—	天
hour	否	*	—	小时
minute	否	*	—	分钟
month	否	*	—	月
weekday	否	*	—	星期几
job	否	—	—	执行的命令
name	否	—	—	作业的描述
reboot	否	no	yes/no	重启后是否需要执行
special_time	否	—	reboot yearly annually monthly weekly daily hourly	特定的执行时间

（续表）

参　数	必　填	默认值	选　项	说　明
state	否	present	present absent	启用或停用作业
user	否	root	—	执行作业的用户

2. 用户组管理模块——group

使用 Group 可以对主机进行批量的用户组添加或者删除。

例：为主机批量添加 Zabbix 用户组。

```
ansible all -m group -a 'name=zabbix state=present'
```

group 模块参数及说明如表 2.18 所示。

表 2.18　group 模块参数及说明

参　数	必　填	默认值	选　项	说　明
gid	否	—	—	用户组的 GID
name	是	—	—	用户组的名字
state	否	present	present absent	新增/删除
system	否	no	yes/no	是否是系统组

3. 服务管理模块——service

service 模块可以帮助我们批量对服务进行操作。

例：批量重启 httpd 服务。

```
ansible all -m group -a 'name=httpd state=restarted'
```

service 模块参数及相关参数如表 2.19 所示。

表 2.19　service 模块及相关参数

参　数	必　填	默认值	选　项	说　明
arguments	否	—	—	参数
enabled	否	—	yes/no	开机自启动
name	是	—	—	服务名称

（续表）

参　数	必　填	默认值	选　项	说　明
pattern	否	—	—	如果服务没响应，则 ps 查看是否具有指定参数的进程，有则认为服务已经启动
runlevel	否	default	—	OpenRC init 脚本
sleep	否	—	—	如果服务被重新启动，则睡眠很多秒后再执行停止和启动命令
state	否	—	started stopped restarted reloaded	服务的状态

4．用户管理模块——user

user 模块可以用于对目标主机进行批量的用户管理。

例：移除所有主机上的 zabbix 用户。

```
ansible all -m group -a 'name=zabbix state=absent remove=yes'
```

user 模块参数及相应说明如表 2.20 所示。

表 2.20　user 模块参数及相应说明

参　数	必　填	默认值	选　项	说　明
append	否	—	—	增加到组
comment	否	—	—	可选设置用户账户的描述
createhome	否	no	yes/no	是否创建 home 目录
force	否	no	yes/no	是否强制操作
generate_ssh_key	否	no	yes/no	是否生成 SSH 密钥
group	否	—	—	用户组
groups	否	—	—	以逗号分割的用户组
home	否	—	—	home 目录
login_class	否	—	—	可以设置用户的登录类 FreeBSD、OpenBSD 和 NetBSD 系统
name	是			用户名
non_unique	否	no	yes/no	相当于 useradd –u
password	否	—	—	密码
remove	否	no	yes/no	相当于 userdel –remove

（续表）

参　数	必　填	默认值	选　项	说　明
shell	否	—	—	该用户的 shell
ssh_key_bits	否	2048	—	密钥的位数
ssh_key_comment	否	ansible-generated	—	密钥的说明
ssh_key_file	否	$HOME/.ssh/id_rsa	—	密钥的文件名
ssh_key_passphrase	否	—	—	SSH 密钥的密码
ssh_key_type	否	rsa	—	SSH 密钥的类型
state	否	present	present absent	新增/删除
system	否	no	yes/no	设置为系统账号
uid	否	—	—	用户的 UID
update_password	否	always	always on_create	是否需要更新密码

5. 系统信息模块——setup

setup 模块可以获取主机上的许多信息，比如这台主机的 IP 是多少，主机上有一些什么样的环境变量，它是承载在什么样的虚拟化平台之上的。我们在后续编写 PlayBook 的时候就会经常使用 setup 模块进行主机信息的查询。

例：获取所有的信息。

```
ansible all -m setup
```

setup 模块参数及说明如表 2.21 所示。

表 2.21　setup 模块参数及说明

参　数	必　填	默认值	选　项	说　明
fact_path	否	/etc/ansible/facts.d	—	fact 的路径
filter	否	*	—	过滤窜

2.5 PlayBook

2.4 节中我们抽取了 Ansible 的一些常用模块进行讲解，相信读者已经对 Ansible 有了

初步的了解。但是模块毕竟是独立运行的，例如有如下场景：我们需要先批量创建用户，然后为其中的三台主机部署 MySQL，另外的三台主机部署 Apache。这种情况下再用单条的命令去实现就显得不那么方便了。带条件的运维操作如图 2.5 所示。

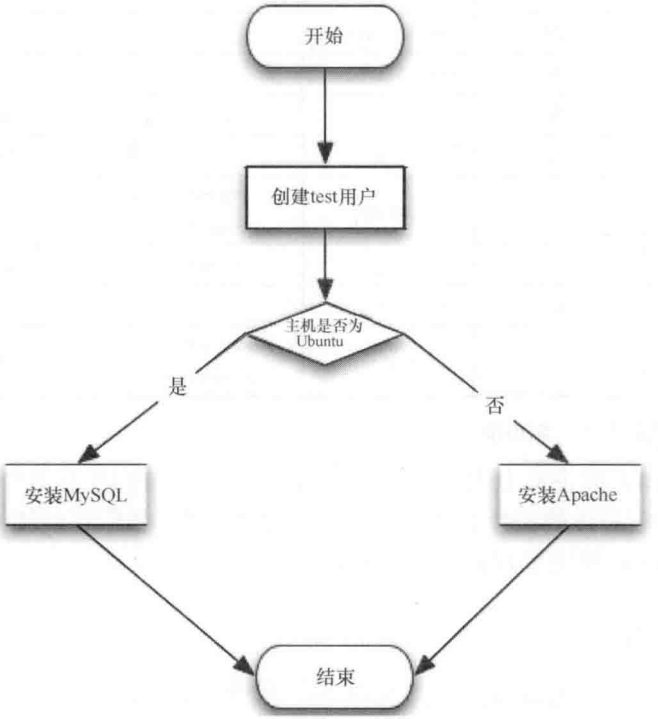


图 2.5　带条件的运维操作

可以看到这个简单场景具备了下面所说的一些特点：

- 流程化——先创建用户，再部署程序；
- 具备条件——某些设备需要部署 MySQL，某些设备需要部署 Apache。为了解决这种流程化的运维操作，Ansible 有了 PlayBook 这个概念。

2.5.1　PlayBook 简介

PlayBook 是 Ansible 的另外一种执行模式，这种执行模式比直接执行单条命令具有更多的特性。可以说假如 Ansible 的模块是设备的零件，那么 PlayBook 就是整个设备的设计蓝图。下面是一个 PlayBook 的例子。

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
   yum: pkg=httpd state=latest
  - name: write the apache config file
   template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
    - name: ensure apache is running
    service: name=httpd state=started
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

1. hosts 和 users

每份 PlayBook 都需要指定针对哪些主机进行运维，而 hosts 变量则说明了这个问题，users 则说明了采用什么用户执行这条命令。

针对 WebServer 主机组，采用 Root 用户执行命令：

```
---
- hosts: webservers
  remote_user: root
```

采用 sudo 模式执行：

```
---
- hosts: webservers
  remote_user: yourname
  sudo: yes
```

针对特定的任务采用 sudo：

```
---
- hosts: webservers
```

```
    remote_user: yourname
    tasks:
      - service: name=nginx state=started
  sudo: yes
```

采用自己的账户登录，采用其他账户执行 sudo：

```
---
- hosts: webservers
  remote_user: yourname
  sudo: yes
  sudo_user: postgres
```

2. Tasks list

每一个 PlayBook 都会有一份作业列表，说明究竟要按照什么样的顺序去执行这些命令。

使用服务模块：

```
tasks:
  - name: make sure apache is running
    service: name=httpd state=running
```

使用 command 模块：

```
tasks:
  - name: disable selinux
    command: /sbin/setenforce 0
```

使用 shell 模块：

```
tasks:
  - name: run this command and ignore the result
    shell: /usr/bin/somecommand || /bin/true
```

使用文件模块：

```
tasks:
  - name: Copy ansible inventory file to client
    copy: src=/etc/ansible/hosts dest=/etc/ansible/hosts
          owner=root group=root mode=0644
```

使用模板模块：

```
tasks:
  - name: create a virtual host file for {{ vhost }}
    template: src=somefile.j2 dest=/etc/httpd/conf.d/{{ vhost }}
```

3. Handlers

可以把 Handlers 看作观察者模式，一旦某个动作有反应了，就会回调给定的方法。

配置完模板之后重启服务：

对一个正在运行的 Apache 服务完成配置文件的更新之后，需要重启 Apache 服务，这时就需要有一个通知的动作。即当配置文件下发完成后，需要通知 Apache 服务进行重启。在这种情况下就可以使用 Ansible 的 Handlers 功能。

```
- name: template configuration file
  template: src=template.j2 dest=/etc/httpd.conf
  notify:
    - restart apache
```

2.5.2 Include 语法

可以写一个很长的 PlayBook 来完成一些运维工作，但是一份很大的 PlayBook 很难达到重用的目标，这时可以采用 include。总的来说，PlayBook 的 Include 就是为了解决重用而设计的。

简单的 include：

```
tasks:
 - include: tasks/foo.yml
```

引用的同时传入变量：

```
tasks:
  - include: wordpress.yml user=timmy
  - include: wordpress.yml user=alice
  - include: wordpress.yml user=bob
```

2.5.3 变量

在开发的时候，我们有变量这个概念，有了变量，也就意味着函数的可重用性得到了

保证。而在 Ansible 的 PlayBook 里面，也是有变量这个概念的。这样使得 PlayBook 的可重用性得到了提高。

在 PlayBook 中定义变量：

```
- hosts: webservers
  vars:
    http_port: 80
```

使用变量：

```
My amp goes to {{ max_amp_value }}
```

变量格式化过滤器：

```
{{ some_variable | to_nice_json }}
{{ some_variable | to_nice_yaml }}
```

条件过滤器：

条件过滤器是一个类似 Swtich 语句的功能。

```
tasks:
  - shell: /usr/bin/foo
    register: result
    ignore_errors: True

  - debug: msg="it failed"
    when: result|failed
  - debug: msg="it changed"
    when: result|changed
  - debug: msg="it succeeded"
    when: result|success
  - debug: msg="it was skipped"
    when: result|skipped
```

默认变量：

```
{{ variable | mandatory }}
```

路径过滤器：

```
{{ path | basename }}   #给变量加上绝对路径
{{ path | dirname }}    #给路径加上文件夹名称
```

为变量编码或解码：

```
{{ encoded | b64decode }}
{{ decoded | b64encode }}
```

为变量做 md5 运算：

```
{{ filename | md5 }}
```

If 判断：

```
{% if 'webserver' in group_names %}
......
{% endif %}
```

循环变量：

```
{% for host in groups['app_servers'] %}
  {{ hostvars[host]['ansible_eth0']['ipv4']['address'] }}
{% endfor %}
```

从命令行传递变量：

```
ansible-playbook release.yml --extra-vars "version=1.23.45
other_variable=foo"
```

2.5.4 条件

对主机进行批量操作时，我们需要根据不同的情况进行不同的操作。这时我们就需要用到 PlayBook 的条件判断功能。例如，当运行的前置命令成功了，才执行后续的命令。又或者是针对不同的操作系统、不同的 IP 段执行不同的操作。为了达到这些目的，需要具备一些条件判断的功能，而 Ansible 也提供了非常强大的条件功能供我们使用。

当操作系统是 Debian 的时候关机：

```
tasks:
  - name: "shutdown Debian flavored systems"
    command: /sbin/shutdown -t now
```

```
    when: ansible_os_family == "Debian"
```

带管道的 when 语句：

```
tasks:
  - command: /bin/false
    register: result
    ignore_errors: True
  - command: /bin/something
    when: result|failed
  - command: /bin/something_else
    when: result|success
  - command: /bin/still/something_else
    when: result|skipped
```

判断变量是否已经定义：

```
tasks:
   - shell: echo "I've got '{{ foo }}' and am not afraid to use it!"
     when: foo is defined

   - fail: msg="Bailing out: this play requires 'bar'"
     when: bar is not defined
```

2.5.5 循环

常用循环的写法：

```
- name: add several users
  user: name={{ item }} state=present groups=wheel
  with_items:
     - testuser1
     - testuser2
```

用哈希表做循环变量：

```
- name: add several users
  user: name={{ item.name }} state=present groups={{ item.groups }}
  with_items:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

把文件名称作为变量循环：

```
---
- hosts: all
  tasks:
    - file: dest=/etc/fooapp state=directory
    - copy: src={{ item }} dest=/etc/fooapp/ owner=root mode=600
      with_fileglob:
        - /playbooks/files/fooapp/*
```

复合变量循环：

假如变量的格式如下所示。

```
---
alpha: [ 'a', 'b', 'c', 'd' ]
numbers:  [ 1, 2, 3, 4 ]
```

循环的时候期望传入的变量为(a,1) (b,2)这样的组合，则可以采用如下的方法：

```
tasks:
    - debug: msg="{{ item.0 }} and {{ item.1 }}"
      with_together:
        - alpha
        - numbers
```

步进变量循环：

循环的变量为整形，如 i=0;i<100;i++这种情况。

```
---
- hosts: all
  tasks:
    - group: name=evens state=present
    - group: name=odds state=present

    - user: name={{ item }} state=present groups=evens
      with_sequence: start=0 end=32 format=testuser%02x

    - file: dest=/var/stuff/{{ item }} state=directory
      with_sequence: start=4 end=16 stride=2

    - group: name=group{{ item }} state=present
      with_sequence: count=4
```

随机变量循环：

```
    - debug: msg={{ item }}
      with_random_choice:
        - "go through the door"
        - "drink from the goblet"
        - "press the red button"
    - "do nothing"
```

Do-Until 类型的循环：

```
    - action: shell /usr/bin/foo
      register: result
      until: result.stdout.find("all systems go") != -1
      retries: 5
      delay: 10
```

2.5.6 PlayBook 使用实例——集中化日常巡检

日常巡检是运维人员经常需要做的一个工作，通过使用 Ansible，我们可以轻易地完成大批量主机集中化巡检的任务。对主机进行巡检一共有以下几个步骤（过程见图 2.6）：

（1）准备巡检脚本文件。
（2）下发巡检脚本到目标主机。
（3）执行脚本文件并生成相应的报告。
（4）抓取主机上所生成的巡检报告。
（5）清理主机上的巡检报告。

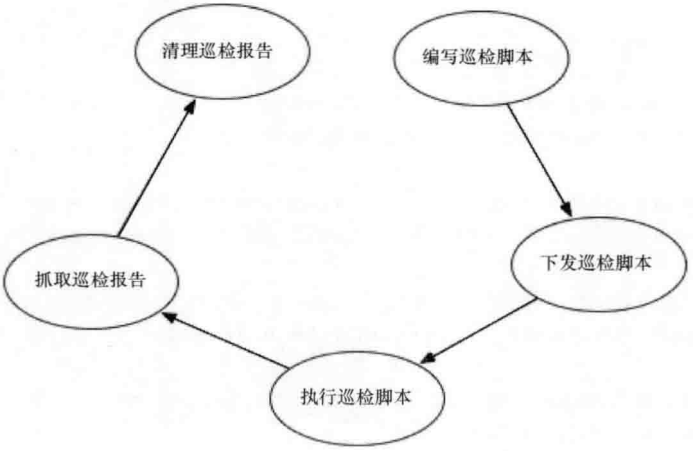


图 2.6　主机常规巡检过程

为了能更好地看出效果，我们多准备一台 CentOS 的主机作为我们巡检的目标主机，IP 为 192.168.41.138，并在 Ansible 的资产文件中对该主机进行相应的资产信息配置。配置完成后，执行 ansible all -m ping。检查一下是否能够正常地连接到目标设备。若返回结果都为 success，则连接正常。

```
192.168.41.139 | success >> {
    "changed": false,
    "ping": "pong"
}

192.168.41.138 | success >> {
    "changed": false,
    "ping": "pong"
}
```

然后，我们需要做的就是准备一份巡检脚本，此处以 2.4 节的 inspection.sh 为例。

接着，开始编写日常巡检的 PlayBook。

指定主机：

我们这次的巡检是对所有主机进行巡检，没有附带一些变量信息，所以我们只要指定需要执行巡检的主机为 all 即可。

```
hosts: all
```

使用 script 模块执行脚本：

```
  - name: 执行脚本，生成报告
   script: /home/inspection.sh
```

使用 fetch 模块进行报告的抓取：

```
  - name: 抓取生成的报告
   fetch: dest=/home/demo/report validate_md5=yes  src=/home/inspection.txt
```

清除目标主机上生成的报告：

```
  - name: 清理主机上生成的报告
   shell: rm -f /home/inspection.txt
```

最终的 yml 文件如下所示。

```
- hosts: all
  tasks:
  - name: 执行脚本，生成报告
    script: /home/inspection.sh
  - name: 抓取生成的报告
    fetch: dest=/home/demo/report validate_md5=yes src=/home/inspection.txt
  - name: 清理主机上生成的报告
    shell: rm -f /home/inspection.txt
```

然后调用 ansible-playbook 进行巡检：

```
ansible-playbook inspection.yml
```

执行结果如下所示。

```
PLAY [all]
*******************************************************************

GATHERING FACTS
*******************************************************************
ok: [192.168.41.138]
ok: [192.168.41.139]

TASK: [执行脚本，生成报告] ******************************************
changed: [192.168.41.138]
changed: [192.168.41.139]

TASK: [抓取生成的报告]
***********************************************
changed: [192.168.41.139]
changed: [192.168.41.138]

TASK: [清理主机上生成的报告] ****************************************
changed: [192.168.41.139]
changed: [192.168.41.138]

PLAY RECAP
*******************************************************************
192.168.41.138          : ok=4   changed=3   unreachable=0   failed=0
192.168.41.139          : ok=4   changed=3   unreachable=0   failed=0
```

然后我们可以看到管理节点上的/home/demo 目录下会产生一个 report 目录：

```
[root@centos demo]#  ls report/
192.168.41.138  192.168.41.139
```

我们可以查看到相应的巡检报告：

```
2014年12月24系统巡检报告
服务器IP:        192.168.41.138
主机名:         centos
系统内核:        2.6.32-358.el6.i686
操作系统版本:    CentOS release 6.4 (Final)
磁盘使用情况:          Filesystem Size Used Avail Use% Mounted on / dev/mapper/
vg_centos-lv_root 18G 877M 16G 6% / tmpfs 504M 0 504M 0% /dev/shm /dev/sda1
485M 30M 430M 7% /boot
CPU核数:
物理CPU个数:     0
逻辑CPU个数:     1
CPU的主频:       2494.025
系统环境变量:
内存简要信息:
内存总大小:      MemTotal: 1030680 KB
内存空闲:        MemFree: 476172 KB
时间/系统运行时间/当前登录用户/系统过去1分钟/5分钟/15分钟内平均负载/
13:08:30 up
14:36, 3 users, load average: 0.08, 0.04, 0.00
1分钟/5分钟/15分钟平均负载/在采样时刻,运行任务的数目/系统活跃任务的个数/最大的pid
值线程/0.08 0.04 0.00 1/140 21335
```

2.6 使用 Ansible 的 API

Ansible 除了有直接命令调用、PlayBook 调用的方式，还支持直接通过 API 调用的方式，这种调用方式主要是针对开发者而设计的。

简单调用：

```
import ansible.runner
```

```
runner = ansible.runner.Runner(
   module_name='ping',
   module_args='',
   pattern='web*',
   forks=10
)
datastructure = runner.run()
```

调用成功后，Ansible 会返回 JSON 格式的字符串。

获取所有主机的启停信息：

```
#!/usr/bin/python
import ansible.runner
import sys

results = ansible.runner.Runner(
   pattern='*', forks=10,
   module_name='command', module_args='/usr/bin/uptime',
).run()
if results is None:
   print "No hosts found"
   sys.exit(1)
for (hostname, result) in results['contacted'].items():
   if not 'failed' in result:
      print "%s >>> %s" % (hostname, result['stdout'])
print "FAILED *******"
for (hostname, result) in results['contacted'].items():
   if 'failed' in result:
      print "%s >>> %s" % (hostname, result['msg'])
print "DOWN *********"
for (hostname, result) in results['dark'].items():
   print "%s >>> %s" % (hostname, result)
```

2.7 小结

2.7.1 Ansible 的优点

（1）部署成本极低：不需要对被管理的目标主机安装 Agent 是一件十分惬意的事情。

（2）没有 Agent 更新的问题：正因为 Ansible 是无 Agnet 的集中化运维软件，所以它也就没有 Agent 更新的问题，极大地简化了 Agent 维护成本。

（3）学习成本低：Ansible 的操作方式中大量现成的模块减少了我们不少的工作量，命令式的操作思路与常规的命令行操作思路十分类似，让使用者非常容易接受。

（4）完备的模块：拥有许多现成的模块，涵盖了许多日常运维所需要的功能。

2.7.2 Ansible 的缺点

事情总是有两面性的，Ansible 通过 SSH 的方式进行运维操作，给自身带来了部署成本低、无 Agent 更新问题等优点。但恰恰是这些优点带给了 Ansible 一些缺点。

（1）对 Windows 的兼容性很差：假如需要通过 Ansible 对 Windows 主机进行运维，那么需要先在 Windows 上安装 OpenSSH Server。新版本的 Ansible 虽然支持 Windows 主机的管理，但是是通过 WMI 进行管理的，对一些低版本的 Windows 操作系统的支持并不是很好。

（2）目标主机需要 Python 模块：Ansible 之所以会有 raw 这个 SSH 模块，就是为了解决使用 Ansible 操作目标主机时，目标主机缺少相应 Python 模块的问题。很多时候，我们运维的 AIX 主机上并没有安装 Python 环境，更别说其他 Python 模块了。

第3章 集中化运维利器——Puppet

3.1 Puppet 与 Ansible

在第 2 章对 Ansible 有了了解之后，接下来我们要接触的是一款叫作 Puppet 的集中化运维软件，它与 Ansible 做的事情很类似，都是对主机进行集中化的运维。Ansible 能够通过 PlayBook 让运维操作做到流程化。对技术的投入是需要花费我们许多时间与精力的。现在又来了一个类似的工具，当然先要看看它和 Ansible 的差异，看看到底它能够为我们做哪些与 Ansible 不一样的事情，确定合适了之后，我们才投入更多的精力去对它进行深入的学习与了解。这样一方面能够让我们对不同场景的技术选型做判断提供帮助，另一方面，也可以帮我们节省一些学习技术的成本。

笔者认为 Puppet 与 Ansible 几点比较主要的差异如下所述。

1. 开发语言上的差异

虽然说是差异，但是也就是语言使用上的不一样，两款软件都有各自的问题。Puppet 是使用 Ruby 编写的一款集中化运维软件，而 Ansible 是基于 Python 编写的自动化运维软件。从使用人群上来看，无论是 Ruby 还是 Python 在国内的使用者都不多，所以当需要进行一些定制性的开发时无论是哪款软件都会碰到人才的问题。从安装上来看，无论是 Ansible 还是 Puppet 都会有依赖包的问题，都会碰到装了这个依赖又少了那个依赖的烦人问题。

2. 使用的思路不一样

针对 Ansible 这种命令式的集中化运维工具，我们使用的思路是告诉它这台主机要通过什么方式会有一个 MySQL，另外一台主机要通过什么方式才会有一个 Apache。而针对 Puppet 这类声明式的集中化运维软件，我们的使用思路则是告诉它，我在这台主机上要有一个 Apache，在另外一台主机上要有一个 MySQL。用什么方式去装？那不是使用者需要关心的事情。

我们通过为 Ubuntu 和 CentOS 的服务器批量安装 MySQL 这样一个简单的例子来了解 Ansible 和 Puppet 在使用思路上的差别。

使用 Ansible 的时候，我们的思路是写一个 PlayBook，然后根据主机的信息来判断是使用 APT 的方式还是通过 YUM 源的方式去安装，如图 3.1 所示。

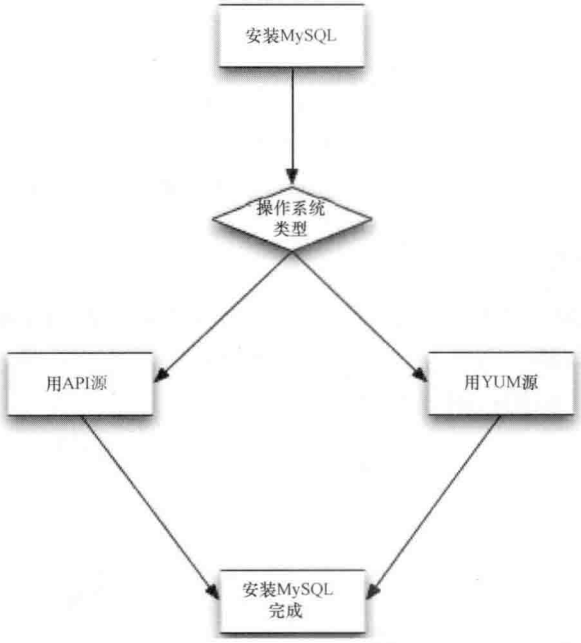


图 3.1　Ansible 命令式的使用思路

可以看到整体的思路与开发软件是很类似的。我们会先判断目标主机究竟是 CentOS 还是 Ubuntu。假如目标主机是 CentOS，那么我们就使用 YUM 进行安装，假如目标主机是 Ubuntu，那么我们就使用 APT-get 去安装。整个使用过程中，我们的思路都是非常流程化的、命令式的。

但是 Puppet 的思路则和 Ansible 有比较大的区别，在使用 Puppet 的过程中，我们所要做的只是告诉 PuppetMaster 我要在服务器上部署 MySQL，具体什么操作系统，应该要怎么装，那不

是我关心的事情，我只关心结果。Puppet 的使用方式是不是很有大老板的思路，如图 3.2 所示。

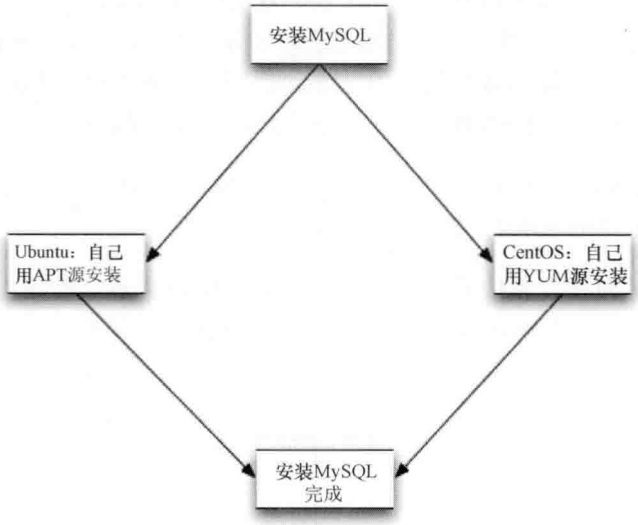


图 3.2　Puppet 的使用思路

3．触发机制的差别

Ansible 的触发机制是由服务器进行的一个主动触发，假如 Ansible 的管理节点是经理，受控节点是员工，那么到了需要干活的时候，Ansible 的管理节点要主动地喊大家干活，不然的话员工们是不会干活的，如图 3.3 所示。

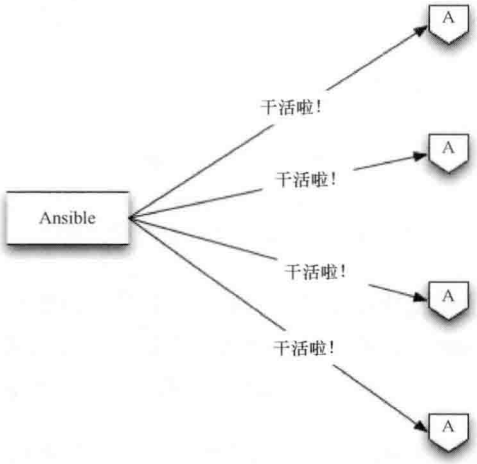


图 3.3　Ansible 的触发机制

Puppet 的触发机制则有明显的差别，Puppet 的管理节点需要做的是把任务放在一个约定好的位置，然后 Puppet 的受控节点就会主动上来找活干了，如图 3.4 所示。

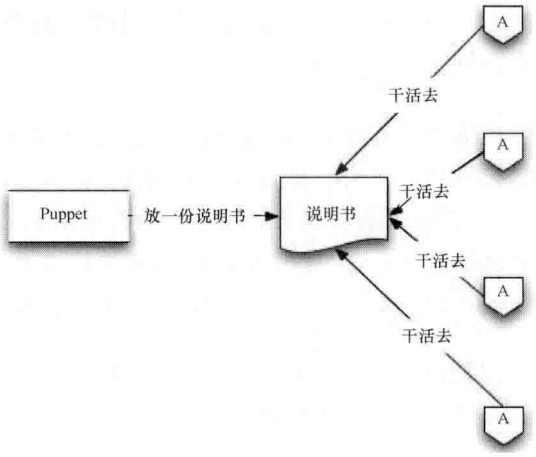


图 3.4　Puppet 的触发机制

也正是因为 Puppet 的这种机制，当目标主机数量较多，请求的频率较相近的时候，容易出现一些性能上的瓶颈，这方面读者可以在需要深入学习 Puppet 的时候再做了解。

虽然 Puppet 的触发机制是被动式的，但是偶尔也会碰到一些紧急的时候需要受控节点马上干活，所以 Puppet 还提供了 Kick 这个方式，主动让受控节点马上干活，但是 Kick 这个方式依旧不会提供任何的说明去告诉受控节点应该怎么干活，它只会告诉受控节点“你的任务很急，赶紧去干”，如图 3.5 所示。

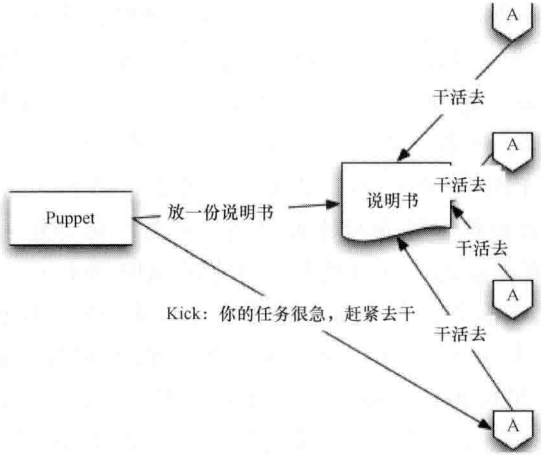


图 3.5　Puppet Kick 的触发机制

4. 不一样的通信方式

Ansible 通过 SSH 去对受控节点进行控制，Puppet 通过 SSL 进行受控节点与控制节点之间的通信。所以在设计上，Ansible 是不需要安装客户端的，而 Puppet 则有 Agent 端这么一个概念，也就是需要在目标主机上安装一个客户端。

> SSL 协议位于 TCP/IP 协议与各种应用层协议之间，为数据通信提供安全支持。SSL 协议可分为两层：SSL 记录协议（SSL Record Protocol）——它建立在可靠的传输协议（如 TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持；SSL 握手协议（SSL Handshake Protocol）——它建立在 SSL 记录协议之上，用于在实际的数据传输开始前，通信双方进行身份认证、协商加密算法、交换加密密钥等。

由于需要在客户机上安装自己的客户端，那么 Puppet 所能操作的设备种类也比 Ansible 要多，特别是对 Windows 的管理上面，Puppet 表现得更为出色。

3.2 Puppet 基础

和了解 Ansible 的过程一样，我们还是先对 Puppet 进行安装，然后一边学习 Puppet 的常用功能，一边与 Ansible 进行对照，让读者能够快速地对 Puppet 有一个入门级别的了解。

> 读者可能会纳闷，怎么讲来讲去都是入门，就不能讲一些深入的吗？
>
> 还记得在学校读书的时候，上了一节关于 Java 的课，老师拿着那本《Java 面向对象》的书和我们讲："我告诉你们，这本书学完了，你们还是什么都做不了，你们应该自己去学习 Struct、JSP、Spring 等技术"。那时我才知道，原来 Java 的技术有这么多。于是我就跑到了图书馆，一本一本地借。到后来参加了工作，我用过.NET，搞过 Java，因为 Zabbix 的原因和 PHP 还有 C 扯上了关系，顺带还把 Python 补习了一下，还接触了许多操作系统相关的知识。随着对技术的接触越久，就会发现技术的水很深。人的精力是有限的，我没有办法把所有的技术、所有的 API 都记得很牢，我也没有办法记得 Spring MVC 应该怎么配置，甚至自己搞开发这么多年了，JDBC 连接数据库的方法我都还得从自己的博客上面复制粘贴下来。但是假如有人想做工作流，我会推荐 Activiti，它比 JBPM5 用起来爽一些。选 Java 的 MVC 框架，我会推荐 Spring MVC，因为它可以基于注解，可以少写很多配置文件，而且做的也还算不错。技术的领域很

广，既然这么宽广，笔者比较倾向快速地对一些技术做一个入门级的了解，然后再根据自己实际的需要去判断究竟哪个技术更适合自己，这样可以为我们节省许多学习的成本。所以这就是为什么一直都对技术做入门级别介绍的原因，当读者发现对 Puppet 感兴趣了，可以去买一本详细介绍 Puppet 的书，读一读官方的文档，这样就可以有更深入的了解了。

3.2.1 安装 Puppet

1. 在 Windows 下安装 Puppet

在 Windows 下安装 Puppet 是非常容易的，首先在 Puppet 官网上下载 Puppet 的安装包，然后在 Windows 下安装即可，如图 3.6 所示。

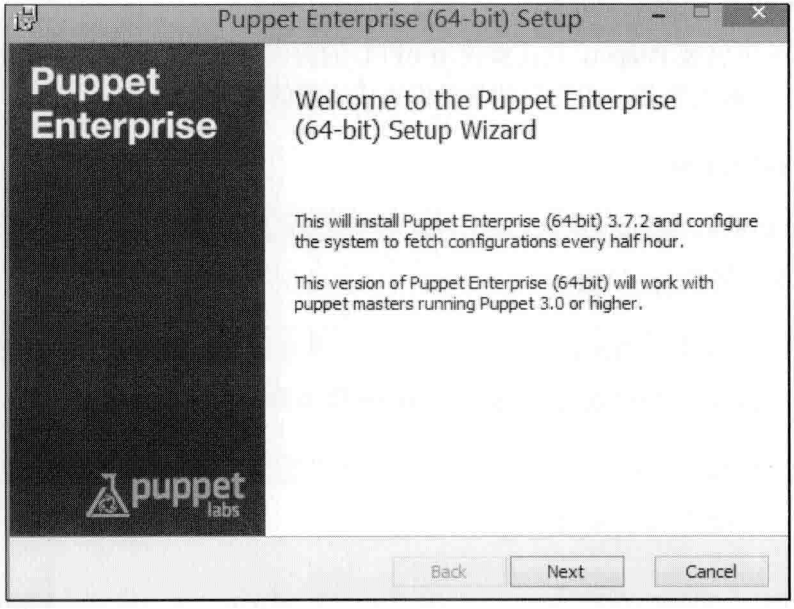


图 3.6 开始进行 Puppet 的安装

勾选完同意协议，继续进行下一步的安装，如图 3.7 所示。

接下来设置 Puppet 的安装路径和 Puppet Master 的域名，完成后继续单击下一步，就可以完成 Puppet Agent 的安装了，如图 3.8 所示。

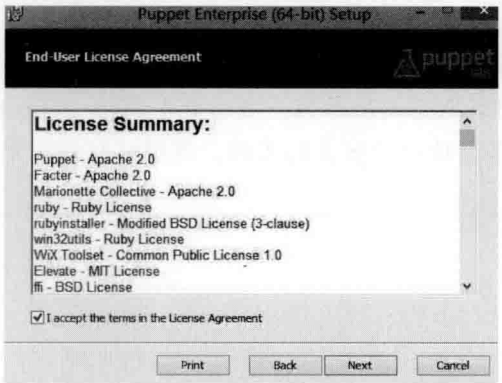


图 3.7　勾选同意协议

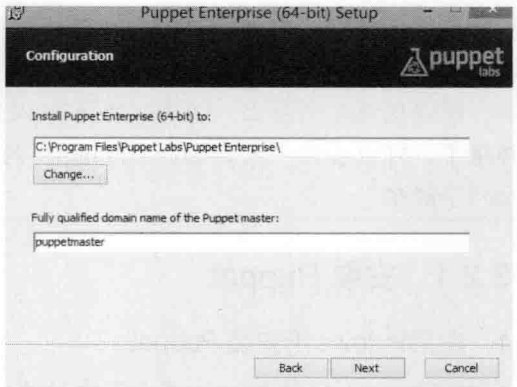


图 3.8　设置 Puppet 的安装路径和 PuppetMaster 的域名

2. 在 CentOS 下安装 Puppet

在 CentOS 下安装 Puppet，只需要使用 EPEL 的源进行安装就可以了，使用源进行安装可以不用管 Puppet 的依赖，源会自动帮我们完成依赖的安装。

安装 Puppet Master：

```
yum install puppet-server
```

CentOS 下安装 Puppet Agent：

```
yum install puppet
```

安装完毕之后，我们可以进入 Puppet Master 服务器，执行

```
puppet cert -all
```

就可以看到所有正在等待认证的主机了。

```
"ppagent.localdomain"     (04:75:49:27:A7:BE:93:5F:C6:08:6A:83:6E:F4:3F:BA)
```

3.2.2　Puppet 主要配置文件

Puppet 比较主要的配置文件有两份，一份是 Auth.conf，另外一份是 Puppet.conf。Auth.conf 是配置 Puppet RestAPI 调用权限的配置文件，而 puppet.conf 是 Puppet 的基础配置文件。

1. 认证配置文件 Auth.conf

Auth.conf 是认证配置文件，主要用于控制 Puppet 的 Rest API 是否允许被调用，Puppet

的 RestAPI 调用方式如下：

```
https://{server}:{port}/{environment}/{resource}/{key}
```

当调用 RestAPI 的时候，服务器就会根据 Auth.conf 定义的规则进行权限的判断，Auth.conf 的格式 ACL 格式如下：

```
path [~] {/path/to/resource|regex}
[environment {list of environments}]
[method {list of methods}]
[auth[enthicated] {yes|no|on|off|any}]
[allow {hostname|certname|*}]
```

ACL 的 Path 是一个正则表达式或路径前缀，表明了允许在什么目录下进行操作，Environment 则声明了环境配置，method 表明了允许在这个路径下操作的方法，auth 则用于进行授权配置，最后的 allow 声明了允许哪些主机进行 RestApi 的调用。

Auth.conf 配置文件如代码清单 3.1 所示。

代码清单 3.1——auth.conf 配置文件

```
path ~ ^/catalog/([^/]+)$
method find
allow $1

path ~ ^/node/([^/]+)$
method find
allow $1

path /certificate_revocation_list/ca
method find
allow *

path ~ ^/report/([^/]+)$
method save
allow $1

path /file
allow *
```

```
path /certificate/ca
auth any
method find
allow *

path /certificate/
auth any
method find
allow *

path /certificate_request
auth any
method find, save
allow *

path /
auth any
```

2. Puppet 的主要配置文件 Puppet.conf

Puppet.conf 是 Puppet 的主要配置文件。它配置所有的 Puppet 命令和服务，包括 Puppet Agent、Puppet Master、Puppet Apply 和 Puppet Cert。Puppet.conf 文件类似于一个标准的 INI 文件，有一些语法扩展。

Puppet.conf 主要的配置部分有以下 4 个：

- main 是 Puppet 的全局配置段，它可以被其他段重写；
- master 用于配置 Puppet Master 和 Puppet cert 命令；
- agent 用于配置 Puppet Agent；
- user 主要用于 Puppet apply 命令。

下面为 puppet.conf 的配置文件样例，如代码清单 3.2 所示。

代码清单 3.2——puppet.conf 配置文件

```
[main]
  certname = puppetmaster01.example.com
  logdir = /var/log/pe-puppet
  rundir = /var/run/pe-puppet
  basemodulepath=
```

```
        /etc/puppetlabs/puppet/environments/production/modules:/opt/puppet/s
hare/puppet/modules
           server = puppet.example.com
           user  = pe-puppet
           group = pe-puppet
           archive_files = true
           archive_file_server = puppet.example.com

         [master]
           certname = puppetmaster01.example.com
           dns_alt_names                                                    =
puppetmaster01,puppetmaster01.example.com,puppet,puppet.example.com
           ca_name = 'Puppet CA generated on puppetmaster01.example.com at
2013-08-09 19:11:11 +0000'
           reports = http,puppetdb
           reporturl = https://localhost:443/reports/upload
           node_terminus = exec
           external_nodes = /etc/puppetlabs/puppet-dashboard/external_node
           ssl_client_header = SSL_CLIENT_S_DN
           ssl_client_verify_header = SSL_CLIENT_VERIFY
           storeconfigs_backend = puppetdb
           storeconfigs = true
           autosign = true

         [agent]
           report = true
           classfile = $vardir/classes.txt
           localconfig = $vardir/localconfig
           graph = true
           pluginsync = true
        environment = production
```

3.2.3 颁发证书

Puppet 出于安全的原因，采用 SSL 隧道的方式进行通信，所以就出现了前面看到的颁发证书这么一个环节。假如 Master 没有为 Agent 颁发证书，Agent 会持续等待证书的颁发，并且会每隔两分钟去检查证书是否已经颁发了。

在 Master 端，我们可以通过“puppet cert –all”命令查看所有已经颁发和未颁发证书的设备，可以通过“puppet cert sign –all”命令完成证书的颁发。有一个地方是需要注意的，我们需要保证 Master 和 Agent 的时间是同步的，不然会导致证书的失效从而造成连接的失效，建议

在设备上做好 ntp 的操作。理解了颁发证书的用意之后，我们就可以开始对设备颁发证书了。

> 在计算机的世界里，时间是非常重要的，例如对于火箭发射这种科研活动，对时间的统一性和准确性要求就非常高。是按照 A 这台计算机的时间？还是按照 B 这台计算机的时间？NTP（Network Time Protocol，网络时间协议）就是用来解决这个问题的，NTP 是用来使网络中的各个计算机时间同步的一种协议。它的用途是把计算机的时钟同步到世界协调时（UTC），其精度在局域网内可达 0.1 ms，在互联网上绝大多数的地方其精度可以达到 1 ~ 50 ms。

查看待签发设备：

```
puppet cert list
```

颁发证书：

```
puppet cert sign -all
```

再次查看 Agent 的证书状态时，可以看到设备列表前会多了一个加号，结果如下：

```
+"ppagent.localdomain" (04:75:49:27:A7:BE:93:5F:C6:08:6A:83:6E:F4:3F:BA)
```

3.2.4 第一个 Puppet 示例

下面我们会做一个 Puppet 进行文件操作的示例，读者可以先不管为什么要这样做，稍后会进行讲解。

步骤 1：创建模块。

```
mkdir -p /etc/puppet/modules/linuxfiletest/{manifests,templates,files}
```

步骤 2：在/etc/puppet/modules/ linuxfiletest /manifests/目录下创建 init.pp 文件，加入以下内容。

```
class linuxfiletest {
file{"/tmp/filetest.txt":content=>"Hello Puppet";}
}
```

步骤 3：创建节点目录，在/etc/puppet/manifests 目录下创建目录 nodes。

```
mkdir /etc/puppet/manifests/nodes
```

步骤 4：创建节点文件 hostgroup.pp 并加入以下内容。

```
node 'ppagent.localdomain'{
include linuxfiletest
}
```

步骤 5：载入受控节点，编辑/etc/puppet/manifests/site.pp，加入以下内容

```
import "nodes/*.pp"
```

步骤 6：启动 CentOS 上的 Puppet Agent。

```
puppet agent --no-daemonize --verbose -server centos.localdomain
```

经过上述 6 个步骤，我们可以看到 192.168.41.136 这台 CentOS 服务器上的 tmp 目录下出现了一个叫 filetext.txt 的文件，内容为“Hello Puppet”。

文件已经出现了，接下来回过头来看看为什么需要做这样的操作，这些操作都有一些什么用处。

首先我们做的一个操作是创建模块，我们通过命令在/etc/puppet/modules /目录下创建了 linuxfiletest 这个文件夹，而在 Puppet 的世界里面，linuxfiletest 被称为一个模块，一个模块是包含代码和数据包的。

在创建 linuxfiletest 模块的时候，我们还创建了 manifests、templates 和 files 目录。这三个目录分别用于存放 puppet 代码、ERB 模板以及被 Agent 端拉取的文件。从目录结构来看，Puppet 的目录结构存放是具有一定约束的，而我们之前所接触的 Ansible 则对目录结构没有那么强的约束。Puppet 模块的基本组成如图 3.9 所示。

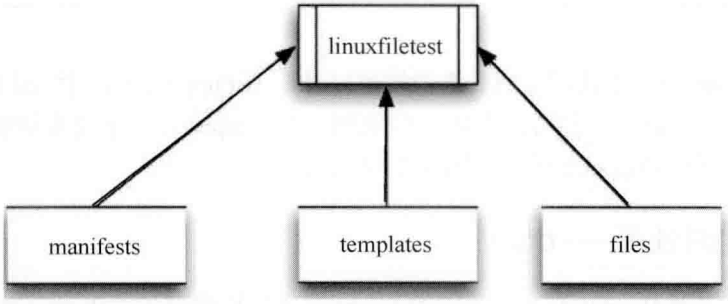


图 3.9　Puppet 模块的基本组成

在第 2 步的时候，我们编写了一份 init.pp 文件，并且在里面告诉了 Puppet Agent 需要在/tmp/filetest.txt 目录下创建一份文件，文件的内容是 Hello Puppet。

第 3 和第 4 步完成了节点的配置和载入，它声明了什么节点需要引入一些什么模块，完成一些什么样的配置操作。

第 5 步的时候我们编辑了/etc/puppet/manifests/site.pp 文件，告诉了 Puppet 哪些节点是需要接受 Puppet 管辖的。

当完成了前面几步的配置之后，我们用“puppet agent --no-daemonize --verbose –server centos.localdomain”命令启动了 Agent，这是一种让 Agent 在前台运行的方式，可以让我们很方便地看到 Puppet Agent 的执行信息。

最后，我们就可以看到一份内容为 Hello Puppet 的文件在/tmp/filetest.txt 目录下生成了。

了解完 Ansible 之后再来使用 Puppet，真是会忍不住说一句，真不是一般麻烦啊。不就是想下发一份文件嘛，还要编这编那的。其实 Puppet 虽然有目录结构放置的约束，让我们感觉做起事情来受到了限制，但从另外一个方面来看，正式因为这些约束让我们编写脚本的时候有了一定的规范性。比如同样是开发一个 B/S 架构的系统，我们可以用最原始的模式开发，也可以用 MVC 的模式进行开发，同样都能达到效果。但是有了一定的约束之后，当文件数量越来越多的时候，我们还是能很容易地找到我们所需要找的内容。当有新的同事来接手的时候，也能很容易地熟悉这样一套约定的规则。

3.3 Puppet 的常用资源

学会了 Puppet 的基本使用方式，我们就可以看一下 Puppet 的常用资源了，Puppet 的资源很多，我们挑几个最常用的资源来做一些案例，对 Puppet 有一个较为初步的认识，更多的资源，读者可以到 Puppet 的官方网站再深入地了解。

3.3.1 定时任务——cron

cron 资源用于对 Puppet 的 Agent 中的 Crontab 进行批量配置，Puppet 在 Puppet Master 上获取了配置信息之后，会把响应的 cron 规则写入 Crontab，或者从 Crontab 中移除。

例 1：每分钟运行一次 NTP。

```
cron{'ntpdate':
      command => "ntpdate pool.ntp.org  ",
      hour=>0,
     minute=>1
     }
```

例 2：每三小时执行一次指定的脚本，执行脚本的用户为 root。

```
cron{'ntpdate':
      command => "/home/puppet/demo.sh",
      hour=>"*/3",
     minute=>0
    }
```

cron 资源的参数及说明如表 3.1 所示。

表 3.1　cron 资源的参数及说明

属　性	说　明
Command	Crontab 中需要执行的命令
Ensure	设置为 present 则追加 cron 作业，设置为 absent 则废除该作业
Environment	环境变量，例如 PATH=/bin:/usr/bin:/usr/sbin.
Hour	Crontab 表达式的小时
Minute	Crontab 表达式的分钟
Month	Crontab 表达式的月份
Monthday	Crontab 表达式的天
Name	作业名称
Provider	Crontab 的提供者，默认为 crontab
Special	可以指定一些特殊的时刻让作业执行，例如 reboot 或者 annually
Target	Cronjob 存放的位置
User	使用什么用户去执行命令
Weekday	Crontab 表达式的星期几

3.3.2　命令执行——exec

exec 资源用于执行命令，最好和 creates 或者 onlyif 属性一起使用，假设不带条件的话，每次当 Puppet Agent 去服务器检查配置之后，都会执行相应的命令。

例 1：当/var/htdocs/www 下的 index.php 文件不存在的时候，进入到/var/htdocs/www 目录下，解压 Web 项目。

```
exec { "tar -xf /var/htdocs/www/deploy.tar":
      cwd   => " /var/htdocs/www/",
      creates => " /var/htdocs/www/index.php",
      path   => ["/usr/bin", "/usr/sbin"]
}
```

例 2：当 nagios 的配置文件有更新的时候，重新编译 nagios。

```
file { "/var/nagios/configuration":
     source  => "puppet://$pupptserver/nagios/",
     recurse => true,
     before => Exec["nagios-rebuid"]
 }
 exec { "nagios-rebuild":
     command => "/usr/bin/make",
     cwd => "/var/nagios/configuration"
 }
```

exce 模块参数及说明如表 3.2 所示。

表 3.2　ecex 模块参数及说明

属　性	说　明
command	需要执行的命令
creates	当一个文件提供了这个参数的时候， 只有当指定的文件不存在时才会执行命令
cwd	进入某个目录执行命令
environment	提供环境变量
group	使用什么用户组去执行该命令
logoutput	执行命令的时候是否在客户端输出日志， 可选项为 true、false、on_failure
onlyif	当此参数给定的条件为 true 时，才会执行相关的命令
path	用于搜索被执行命令的路径，多个路径的时候用“：”分隔
provider	使用哪一个 provider 进行命令的执行， 可选项为 posix、shell、windows
refresh	刷新命令时所触发的命令

（续表）

属　性	说　明
refreshonly	当某个资源被更新时触发此命令
returns	执行返回结果，默认为 0
timeout	命令执行超时时间
tries	最大执行重试次数
try_sleep	每次尝试执行的间隔
unless	假如被设置了，该命令会直到 unless 设置的命令执行完才会开始执行
user	执行此命令的用户

3.3.3 文件管理——file

file 资源主要用于对文件进行管理，包括对文件名称、文件所有者、文件的内容等。而且 file 资源可以创建文件、目录以及链接，是一个经常会使用到的资源类型。

例 1：在所有节点上创建/etc/inspection 目录，权限为 700，所有者为 demo，要求目录下只允许有 shell 目录，并且 shell 目录及下一级目录的权限属性保持不变。

```
file{ "/etc/inspection":
    owner   => "demo",
    group   => "demo",
    mode    => 0700,
    ensure  => directory,
    recurse => true,
    purge   => true,
    force   => true,
    ignore  => "shell",
  }
```

例 2：同步服务器文件到本地。

```
file{'demo':
    name=>'/tmp/demo',
    source=>presend,
    source=>'puppet://$fileserver/modules/demo/demo'
}
```

file 模块参数及说明如表 3.3 所示。

表 3.3　file 模块参数及说明

属　　性	说　　明
path	需要管理的文件的路径
ensure	present：创建文件 absent：删除文件 file：确保该文件是一个普通文件， 并且可以使用 content 和 source 属性 directory：确保该内容是一个文件夹，并且启动了 source、recurse、recurselimit、ignore、purge 的属性
backup	替换文件之前是否需要备份
checksum	是否使用校验和，默认为 md5，可选的有 md5lite、 sha256、sha256lite、mtime、ctime、none
content	文件的内容
ctime	文件的修改时间
force	是否强制执行文件操作，可选的值有 yes、no、true、false
group	文件所属的用户组
ignore	是否忽略转义字符
links	创建链接，可选的值有 follow、manage， follow 为硬链接、manage 为软链接
mode	文件/文件夹的读/写权限
mtime	文件的修改时间，一个只读的属性
owner	文件/文件夹的所属用户
provider	指定 file 资源的提供者，Puppet 会自己去判断 agent 是什么 操作系统，一般不用设置，可选项为 windows 和 posix
purge	这个属性只有在 ensure=>direction 并且 recurse=>true 的情况下才生效，说明未受管理的文件是否应该清除
recurse	是否递归操作目录，可选项为 true、false、inf、remote
recurselimit	递归层级
replace	当一份文件存在但是内容不匹配的时候，是否需要替换
selinux_ignore_default	是否忽略 Selinux
selrole	Selinux 所使用的角色
seltype	Selinux 的组件
seluser	Selinux 所使用的用户
show_diff	文档由于差异而受 Puppet 更改的时候，是否展示文件差异
source	用于复制到本地系统的 URL 路径

（续表）

属　性	说　明
source_permissions	指定 Puppet 是否需要使用文件所属的用户、用户组、文件/文件夹权限
sourceselect	复制所有符合条件的文件还是只复制一个
target	创建一个符号链接
type	文件类型
validate_cmd	在替换文件之前的检验命令
validate_replacement	validate_cmd 替换所用的参数

3.3.4 包管理——packag

package 类型主要用于管理软件包，它实现了对软件包的安装、卸载、升级等功能，对于类似 Windows 这类没有包管理的系统，package 资源还可以通过 source 指定一个 URL 供系统提取包文件。与 Ansible 相比，Puppet 的包管理不需要知道究竟是用 YUM 源管理的还是 APT 源管理的，但 Ansible 是需要的，这一点倒是挺方便。

例 1：批量部署 MySQL。

```
package { "mysql":
   name   => $operatingsystem ?
     {
        RedHat  => ["mysql-server","mysql"],
         SLES    => ["mysql","mysql-client"],
         default => ["mysql-server","mysql"],
     },
    ensure => installed,
     }
```

例 2：使用 Source 为 Windows 安装 MySQL。

```
package{'mysql':
   ensure=>installed,
   provider=>'msi',
   source=>'D:/mysql-5.5.16-winx64.msi',
   install_options=>{'INSTALLDIR'=>'C:\mysql'},
}
```

package 模块参数及说明如表 3.4 所示。

表 3.4　package 模块参数及说明

属　性	说　明
name	包名称
ensure	包是否应该存在，present 表示应该存在，absent 表示应该被卸载，purged 表示连同配置文件一并卸载，latest 表示需要更新到最新的版本
adminfile	只用于 Solaris，一个包含默认安装包的文件
allow_virtual	表明是否可以用虚拟的包名去进行包的安装和卸载，可选项为 true、false
allowcdrom	指明 APT 源是否可以使用光盘作为源
category	包的只读参数
configfiles	配置文件是否需要被替换，replace 为替换、keep 则表示不修改，默认为 replace
description	包的只读参数
flavor	仅支持 OpenBSD，表明你需要哪种类型的包
install_options	安装参数
instance	包的只读参数
package_settings	包的参数
platform	包的只读参数
provider	安装提供者，Puppet 会自动选择
responsefile	目前只用于 Solari 和 Debian，指向用于应答的文件
root	包的只读参数
source	安装包的路径
status	包的只读参数
uninstall_options	卸载时所用的参数
vendor	包的只读参数

3.3.5　服务管理——service

serivce 模块主要是用于对服务进行管理，它支持对服务的启动、停止、重启等操作。

例 1：设置服务器启动之后，SSHServer 自动启动，服务器重启之后 SSHServer 能够自动启动，配置文件更改后自动执行 restart 的动作。

```
service { $ssh::params::ssh_service_name:
          ensure => stopped,
          hasstatus => false,
```

```
                hasrestart => true,
                enable => true,
                subscribe => Class["ssh::config"],
        }
```

例 2：启动 MySQL 服务。

```
service{'mysql':
    ensure=>"running"
}
```

service 模块参数及说明如表 3.5 所示。

表 3.5 service 模块参数及说明

属　性	说　明
name	服务名称
ensure	指定服务的状态，可选的值为 stopped、running
binary	守护进程的路径
control	用于管理服务的控制变量
enable	设置服务是否开机启动
flags	指定一个标志通过启动脚本
hasrestart	指定服务是否具有 restart 命令，可选值为 true/false
hasstatus	指定服务是否具有 status 命令，可选值为 true/false
manifest	提供一个命令用于配置服务
path	提供一个用于搜索 init 脚本的路径
pattern	用于正则匹配进程信息以便停止服务
provider	服务提供者，Puppet 会自动选择
restart	提供重启服务的命令，如未提供则使用默认的重启服务命令
start	提供启动服务的命令，如未提供则使用默认的启动服务命令
status	提供查看服务状态的命令，如未提供则使用默认的查看服务状态命令
stop	提供停止服务的命令，如未提供则使用默认的停止服务命令

3.4 Puppet 语法基础

由于 Puppet 是用 Ruby 编写的，因此 Puppet 的语法也和 Ruby 非常类似，都是较为简

单的面对对象的高级语言。

值得一提的是 Puppet 把需要管理的内容抽象成为了资源，每个资源拥有不同的属性，Puppet 的语法主要用于描述资源的属性以及资源与资源之间的关系。

3.4.1 资源

首先看这样一个例子：

```
file {" /root/demo" :
        name => "root/demo" ,
        owner => root ,
        group => root ,
        mode => 755;
    }
```

上面的代码让/root/demo 这份文件属于 root 用户以及 root 用户组，并且把/root/demo 文件的权限配置为 755。

文件的第一行指定了资源类型以及资源的名称，这里声明了资源类型是 file 类型，这个资源的标题为/root/demo，标题用做资源的唯一标识。

```
file {" /root/demo" :
```

从第二行开始主要是配置了 file 资源的一些参数：

```
name => "root/demo" ,
owner => root ,
group => root ,
mode => 755;
```

这里指定了要对/root/demo 文件进行操作，并且把它的用户和用户组都设置为 root，设置这份文件的权限为 755。

有些时候，我们会在一份代码里用到许多相同的资源，这时可以使用“[]”把所有资源的标题都写到里面，这样就可以减少重复劳动。

```
file {
[ " / etc /passwd" , " / etc / hosts " ] :
        owner => root ,
```

```
            group => root ,
            mode => 644;
    }
```

3.4.2 类

类的作用是把一组资源归为一类，使得我们可以一起使用这些资源，例如我们可以把sshd 和配置文件做成一个名称为 ssh 的类，其他模块需要用的时候只要直接引用 ssh 这一个类就可以了，这样可以写出更简洁的代码，便于维护。类可以继承，下面看一个具体的例子：

```
class httpd {
    package {
        " httpd" :
            ensure => installed ;
    }
    service {
        " httpd" :
            ensure => running ;
    }
}
```

可以看到 httpd 类把 package 和 service 这两个资源组合起来了，用于完成 Httpd 服务的安装和启动。对照 Ansible 的 PlayBook 来看，其实两者是有点类似的，Ansible 的 PlayBook 把不同的命令组合在了一起，而 Puppet 则是通过类把不同的资源组合在了一起。

3.4.3 变量

大多数语言都支持变量，Puppet 的语法也支持变量，在 Puppet 的语法里面，使用$来定义变量，如下：

```
$content=" Hello World,Puppet"
file {
    "/root/demo" :
    content => $content;
}
```

上面的代码定义了一个 content 变量，变量的内容为“Hello World,Puppet”，然后在资源内使用了这个变量。除了使用自己定义的变量，也可以使用 Facter 收集到的变量。

Facter 可以收集各个受控节点的主机信息，这些信息包括服务器的操作系统、内存大

小、虚拟化平台等信息，我们可以通过 facter 命令收集主机信息，下面是 facter 默认采集的一些指标，我们可以在代码中直接使用。例如我们需要对目标操作系统变量进行使用，那么我们直接使用$operatingsystem 就可以了。

```
dnsdomainname: Unknown host
architecture => i386
augeasversion => 1.0.0
boardmanufacturer => Intel Corporation
boardproductname => 440BX Desktop Reference Platform
boardserialnumber => None
domain => localdomain
facterversion => 1.6.18
fqdn => centos.localdomain
hardwareisa => i686
hardwaremodel => i686
hostname => centos
id => root
interfaces => eth1,eth2,lo
ipaddress => 192.168.41.136
ipaddress_eth1 => 192.168.41.136
ipaddress_eth2 => 192.168.41.137
ipaddress_lo => 127.0.0.1
is_virtual => true
kernel => Linux
kernelmajversion => 2.6
kernelrelease => 2.6.32-358.el6.i686
kernelversion => 2.6.32
mac address => 00:0C:29:35:B2:53
mac address_eth1 => 00:0C:29:35:B2:53
macaddress_eth2 => 00:0C:29:35:B2:5D
manufacturer => VMware, Inc.
memoryfree => 828.09 MB
memorysize => 1006.52 MB
memorytotal => 1006.52 MB
mtu_eth1 => 1500
mtu_eth2 => 1500
mtu_lo => 16436
netmask => 255.255.255.0
netmask_eth1 => 255.255.255.0
```

```
        netmask_eth2 => 255.255.255.0
        netmask_lo => 255.0.0.0
        network_eth1 => 192.168.41.0
        network_eth2 => 192.168.41.0
        network_lo => 127.0.0.0
        operatingsystem => CentOS
        operatingsystemrelease => 6.4
        osfamily => RedHat
path => /usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
        physicalprocessorcount => 1
        processor0 => Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
        processorcount => 1
        productname => VMware Virtual Platform
        ps => ps -ef
        puppetversion => 2.7.26
        rubysitedir => /usr/lib/ruby/site_ruby/1.8
        rubyversion => 1.8.7
        selinux => false
    serialnumber => VMware-56 4d 42 61 df 6e a7 a5-2f f3 99 7c f3 35 b2 53
        sshdsakey                                                          =>
    AAAAB3NzaC1kc3MAAACBAK1zGGn8urDMzAg5ZvJGINWPzm/r/gbGjfzI9cI6MUowNC/9qV
XcGHR5nDMd988uC9CURKYvWC+uKtlyiPegMu52yrMHirHFs/rz+BYYYAB6eM0z+dOW8aM2Fc13
P28PugCQsWoAlb8UcWB39g44pVHfVVlUU9Oh7hRCTweabPIPAAAAFQCBgR06j2NqzGdz6SQa+Q
F5aXmwbwAAAIBM8EZ6Sh2BfAVuiq8VjacsnjUGN24T9c1s3obMghZiLa8lQJVjnJ2rvMy2CAqX
LNwLPfX3gFwYZzUFKmmXJ96h5gzAyWoi1uq3g2QxvUGGF+fzCxRi2cV7kxvN5gAR038PQ/ioPr
+dWgiEOYD9eEgd9/m7R2t27ZuD+jJ+4AWrUgAAAIEAlU1G17lrNJ8G9ZPZwCrZVDAD6X8935LC
Qffn+UNaC8c9RH9vm9JMHVdcwktkiHKnnE3sXK2W2gnNBHJafoHu03DyR4Qt2M8uQxcF7bQy4A
MfKgl7d9Ao5QWDXIMgC0Vzcq52vsApr2y4MHiIpK0xtETgqfQHgKwO6cI+jCpv8M0=
            sshrsakey                                                      =>
    AAAAB3NzaC1yc2EAAAABIwAAAQEAl1oCZQmL7Tu+z0fNv5CFy80Np1YappzE9VqTmNP0ny
CB7JTidbKMZ/L/kHavtxXOAlFRhJ9Rly2wtmb7Fw6jva67D1+sfttcm6kCyR7xdujbOdW0aqS4
zjuD/b8dwiF+1KezG8UHzCxXe5qkSYyvKV2/Omvs4iC9iqyIexdGhm9/SvMrMZ9Rf5Kvxu001I
9z/QZCDn6aoCVM4yBzWSiZSWx3dC/FgrZKRSsLHitYps0R1bM+JSUciq3YpWZ5yXkKxCxR5/F4
ujkhdZ3Rr5B/D5oSA41NB9ZZumWvm3AemuwpVV6twvvJ914/dfeSY08CuEVPiZfPxqjICnmYBH
J3rw==
            swapfree => 1.97 GB
            swapsize => 1.97 GB
            timezone => CST
            type => Other
            uniqueid => 00000000
```

```
uptime => 4:30 hours
uptime_days => 0
uptime_hours => 4
uptime_seconds => 16259
virtual => vmware
```

3.5 小结

3.5.1 Puppet 的优点

（1）丰富的模块：我们日常工作所需要使用的大多数模块都不需要自己写，直接使用“puppet module search” 命令搜索就可以了，一旦理解了 Puppet 的基本使用方法之后，我们就可以从开源软件中获得许多现成可用的资源。

（2）多操作系统兼容：Puppet 所支持的操作系统十分全面，如 AIX、HP-UX、Linux、Windows，覆盖面很广。

3.5.2 Puppet 的缺点

（1）采用 Ruby 开发：操作系统需要安装 Ruby 等环境，一旦主机没有 Ruby 环境，甚至连 GCC 环境都没有的时候部署人员就得头疼了。

（2）基于证书：Puppet 的通信是需要证书的，这其实是 Puppet 的优点（安全），但同样也带来了缺点。一旦服务器上的时间与客户机上的时间不匹配时，证书就报废了，又得重新颁发一次。笔者就碰到过这么一个让人头疼的问题，一台 RedHat 服务器的时间会突然快一个小时，有时候又突然慢一个小时，虚拟机的维护团队已经帮我们设置成每分钟 NTP 一次了，但还是会出现这个情况。

（3）命令式触发功能较弱：服务器希望让客户机马上触发一个特定的指定对于 Puppet 是比较难实现的，虽然有 Kick 这么一个动作，但依然无法很好地达到我们期望的效果。对于配置来说是十分合适的，但对于差异化即时处理的话优势则没那么明显。

第4章 集中化运维利器——SaltStack

4.1 SaltStack、Puppet、Ansible

使用 Ansible 的时候，我们的思路是命令式的，我们会直接给服务器下达指令，让服务端告诉客户端它们需要干什么、怎么干，它是由服务端主动触发的一款集中化运维工具。

而在使用 Puppet 的时候，我们的使用思路则更倾向于声明式的，我们会告诉 Puppet，我们需要 Agent 达到一个什么样的状态，哪些设备需要配置成什么样，而具体怎么配置的过程则是透明的，大多数情况下，使用者无须关心整个配置过程究竟是怎么样的。

那有没一款软件既有声明式的特性，又具备命令式特性；既具备了对主机集中配置的能力，又具备实时执行命令的能力呢？有的，这款软件就是 SaltStack。

SaltStack 是一款使用 Python 编写的既具备声明式特性，又具备命令式特性的集中化运维软件。命令式的特性旨在实时地对设备进行批量的操作，而声明式的特性旨在对设备进行一个集中化的配置，SaltStack 由 Master 和 Minion 组成，其实 Minion 就是我们前面所讲的 Agent，只是换了一个名字罢了。Puppet 与 Saltstack 对应关系如图 4.1 所示。

在部署模式上面，SaltStack 具备 Puppet 的安装客户端的模式，如图 4.2 所示。

SaltStack 也具备像 Ansible 那种直接通过 SSH 协议进行管理的模式。这就让我们在不需要 SaltStack 提供的模块功能的前提下通过 SSH 协议对主机进行批量的操作。虽然使用

SaltSSH 的模式会大大削弱 SaltStack 的能力，但是它具备了不需要安装 Agent 端的便利性，并且这种模式非常适合第一次部署 SaltStack Minion 的时候使用，如图 4.3 所示。

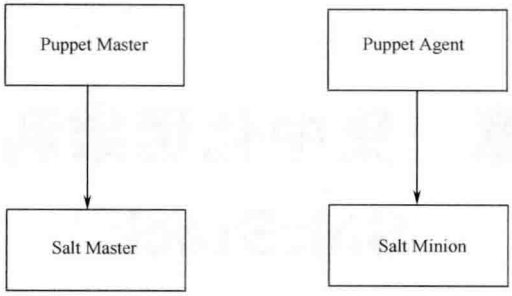


图 4.1　Puppet 与 SaltStack 对应关系图

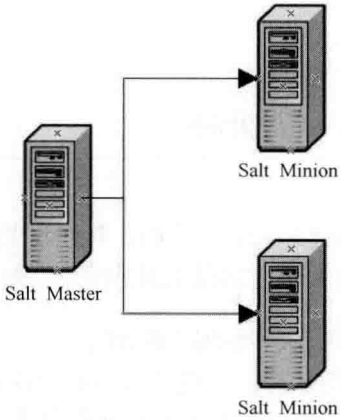


图 4.2　使用 Salt Minion 的部署模式

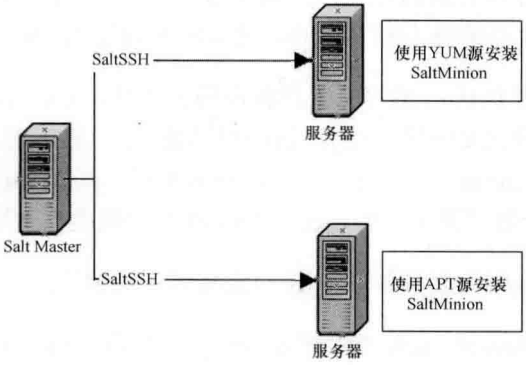


图 4.3　Salt SSH 直接命令调用安装 SaltMinion

SaltStack 有一个叫作 SLS 的文件，它是一份 YAML 格式的文件，用于描述客户机应该达到一个怎么样的配置状态。

> YAML 是一种直观的能够被计算机识别的数据序列化格式，是一个可读性高并且容易被人类阅读，容易和脚本语言交互，用来表达资料序列的编程语言。
>
> 它是类似于标准通用标记语言的子集 XML 的数据描述语言，语法比 XML 简单得多。
>
> 以下为 YAML 的一个简单例子：
>
> ```
> user:
>  name: demo
>  age:18
> ```

服务器端会根据用户配置好的 SLS 文件去配置具体的 Minion。SaltStack 对于 Minion 的配置与 Puppet 是有差别的。PuppetAgent 会定时主动地下载 PuppetMaster 上的 pp 文件，然后使用 pp 文件来配置自己。虽然 Puppet 可以使用 Kick 命令让客户机马上下载 PuppetMaster 上的配置文件，然后主动配置自己的动作，但是大多数情况下 Puppet 的配置机制还是在 Agent 端上触发的。而 SaltStack 则主要靠 SaltMaster 去主动通知 SaltMinion，由 SaltMinion 对 SaltMaster 上的文件进行下载，然后进行 SLS 文件的解析，最后完成整个配置。与 Puppet 的差别在于，Puppet 的触发是客户机定时去下载的，而 SaltStack 是由 Master 通知 Minion 去下载的，它们的触发源是有差别的。SaltStack 的触发模式如图 4.4 所示。

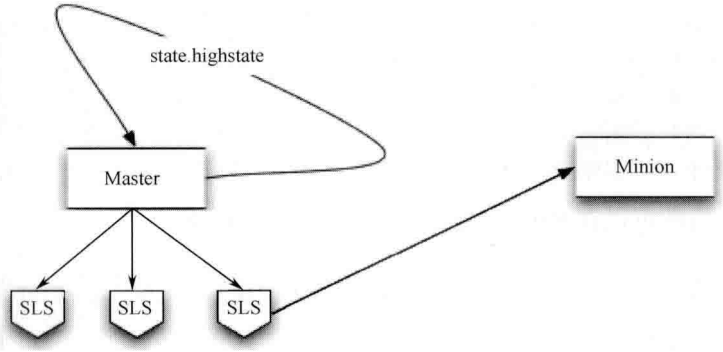


图 4.4 SaltStack 的触发模式

4.2 无 Agent 模式——SaltSSH

SaltStack 和 Puppet 的结构有点类似，都需要在客户机上安装程序。但是 SaltStack 与

Puppet 有一点小差别，它除了支持安装 Minion 的方式，还支持无 Agnet 的方式对主机进行批量的管理。

从 0.17.0 版本开始，Salt 开始支持通过 SSH 的方式进行批量管理节点，这个特性使得 Salt 可以通过 SSH 的方式进行主机之间的管理，而且不需要在客户机上装 Salt Minion。在 CentOS 下，直接用 “yum install salt-ssh” 命令安装就可以了。

安装 SaltSSH 模块之前，我们需要先配置一份 SSH 管理的清单，这份配置文件默认为 /etc/salt/roster，然后就可以进行相应的配置了。Roster 的常用配置方式如下：

```
<Salt ID>:   # 设备 id
     host:   # 目标设备的 IP 或 DNS
     user:   # 用户
   passwd:   # 密码
     port:   # ssh 端口
     sudo:   # 是否可以执行 sudo
     priv:   # ssh 私钥路径
  timeout:   # ssh 连接超时时间
```

以下为配置实例：

```
  web1:
  host: 192.168.41.138
  user: root
passwd: 1q2w3e4r
  sudo: True
```

配置完成之后就可以在 SaltMaster 端执行执行如下命令：

```
salt-ssh -i '*' -r 'vmstat'
```

服务器返回结果如下：

```
web1:
    ----------
    retcode:
        0
    stderr:

    stdout:
```

```
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0  94600 141028 384820    0    0     0    11  108   45  2  1 97  0  0
```

SaltStack 除了适合用于 SSH 批量操作之外，它其实更多地用于第一次对 Minion 进行批量安装的时候，减少我们对设备进行 SaltMinion 的批量部署的工作量。

```
salt-ssh '*' -r 'yum -y install salt-minion'
```

4.3 SaltStack 的基本组成

SaltStack 的节点类型一共有三种，它们分别是 Master、Minion 还有 Syndic。

Master 负责的事情是调用命令，让 Minion 完成相应的动作，从而对 Minion 进行集中化的操作。

Minion 负责的事情就是执行 Master 所下发的指令，完成实际的运维任务。

Syndic 专门负责解决网络拓扑中的二级代理的问题，它需要在服务器上运行 Master，并需要让子网下的 Minion 指向自己，也就是让自己伪装成 Master。

而 Master、Minion、Syndic 之间的通信则是使用 ZeroMQ 这个消息中间件完成的。

> ZeroMQ 是一个类似于 Socket 的一系列接口，它跟 Socket 的区别是：普通的 Socket 是端到端的（1：1 的关系），而 ZeroMQ 是 N：M 的关系。人们对 BSD 套接字的了解较多的是点对点的连接，点对点连接需要显式地建立连接、销毁连接、选择协议（TCP/UDP）和处理错误等，而 ZeroMQ 屏蔽了这些细节，让网络编程更为简单。ZeroMQ 用于 node 与 node 间的通信，node 可以是主机或者是进程。

SaltStack 的基础结构如图 4.5 所示。

ZeroMQ 需要使用 4505 和 4506 两个端口，所以我们需要打开这两个端口的防火墙，才可以使各个组件之间正常通信。下面以 CentOS 为例：

```
lokkit -p 22:tcp -p 4505:tcp -p 4506:tcp
```

SaltStack 与 Puppet 类似，也有一个认证的过程。当 SaltStack 的 Minion 配置完成之后，我们就可以在 SaltStack 的 Master 上执行 salt-key 进行待认证的设备的查看操作了。接着，我们可以执行 salt-key –A，对所有的主机进行认证。认证完成之后，我们可以在服务器上执行“salt ‘*’ test.ping”命令来查看是否能 ping 通客户机，当检查完毕后，我们就可以开始对 Minion 进行批量管理了。

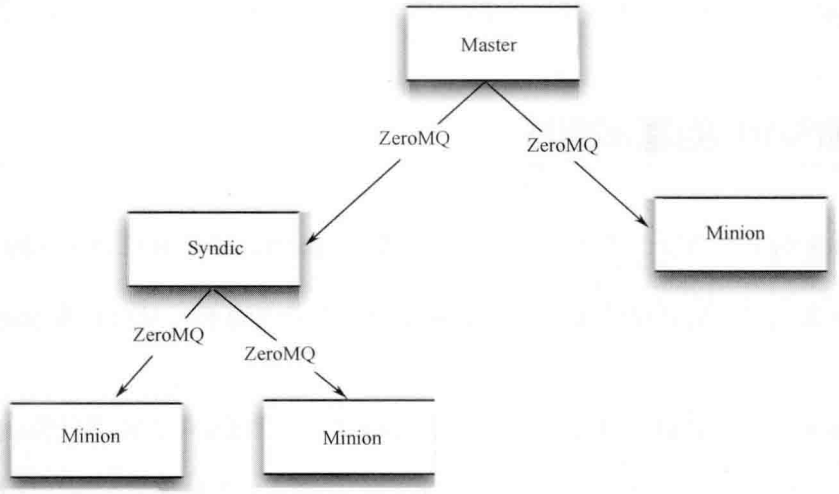


图 4.5　SaltStack 的基础结构

4.4　Salt State 概述

SLS（代表 Salt State 文件）是 Salt State 系统的核心。SLS 描述了系统的目标状态，由格式简单的数据构成。它与 Puppet 的 pp 文件的作用是一样的，都是为了描述客户机最后应该到达一个什么样的状态。SLS 文件由 top.sls 和对应的 sls 文件组成。

4.4.1　top.sls

top.sls 是配置管理的入口文件，一切都是从这里开始的，在 master 主机上，top.sls 默认存放在/srv/salt/目录。

top.sls 默认从 base 标签开始解析执行，下一级是操作的目标，可以通过正则、grain

模块或分组名来进行匹配，再下一级是要执行的 state 文件，不包括扩展名。

通过正则进行匹配的示例：

```
base:
  '*':
- webserver
```

通过分组名进行匹配的示例：

```
base:
  group1:
    - match: nodegroup
- webserver
```

通过 grain 模块匹配的示例：

```
base:
  'os:Fedora':
    - match: grain
- webserver
```

4.4.2 state 文件

准备好 top.sls 文件后，编写一个 state 文件/srv/salt/webserver.sls，文件内容如下：

```
apache:
  pkg:
- installed
```

第一行被称为标签定义，在这里被定义为安装包的名称。注意：不同发行版软件包命名不同，比如 Fedora 中叫 httpd 的包在 Debian/Ubuntu 中叫 apache2。

第二行被称为状态定义，在这里定义使用 pkg state module。

第三行被称为函数定义，在这里定义使用 pkg state module 调用 installed 函数。

4.4.3 配置主机

当我们把 SLS 文件准备好了之后，就可以在服务端执行

```
salt '*' state.highstate
```

或附加上 test=True 参数，测试执行

```
salt '*' state.highstate -v test=True
```

SaltStackMaster 端对目标主机发出指令并运行 state.highstate 模块，目标主机首先会将 top.sls 下载、解析，然后 top.sls 会根据相应的匹配规则对内容进行解析、执行，然后将结果反馈给 Master，如图 4.6 所示。

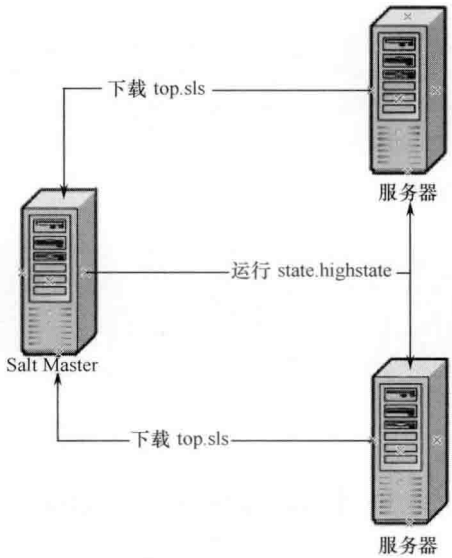


图 4.6　SaltStack 的配置过程

4.4.4　SaltState 之 Requires

当我们的配置具备先后顺序的时候，就可以考虑使用 SaltState 中的 Requires。Requires 是专门用于解决配置的先后顺序而设计的，考虑如下场景，我们需要先完成 httpd 的安装，然后再下发 index.html 文件到指定的主机上。

编写 init.sls：

```
httpd:
  pkg.installed: []
  service.running:
```

```
    - require:
      - pkg: httpd
/var/www/index.html:
  file:
    - managed
    - source: salt://apache/index.html
    - require:
      - pkg: httpd
```

执行 salt '*' state.highstate，然后我们就可以看到客户机上的 HTTP 服务安装好了，同时 index.html 也会被传输到 Minion 的指定目录。

4.4.5 Template、Extends、Includes

为了让使用者能够编写出更加简洁、可复用的 SLS 文件，SaltStack 提供了 Template、Extend、Includes 的机制。合理使用这些特性有助于我们更加高效地编写出可维护的、复用性更高的 SLS 文件。

1. Template

合理地使用 Template 可以编写出可维护性更高的 SLS 文件，与 Ansible 一样，SaltStack 的 Template 技术也是采用 Jinja2 实现的，下面看一下实例。

使用数组生成 SLS：

```
{% for usr in ['moe','larry','curly'] %}
{{ usr }}:
  user.present
{% endfor %}
```

生成结果如下：

```
moe:
  user.present
larry:
  user.present
curly:
  user.present
```

在 SLS 中使用 Grains：

当我们需要针对不同的操作系统进行不同的配置时，就可以在模板中使用 Grains。

```
apache:
  pkg.installed:
{% if grains['os'] == 'RedHat' %}
    - name: httpd
{% elif grains['os'] == 'Ubuntu' %}
    - name: apache2
{% endif %}
```

Grains 是 Minion 第一次启动的时候采集的静态数据，可以用在 Salt 模块和其他组件中。Grains 在每次 Minion 启动（重启）时都会采集一次数据并向 Master 汇报，采集的都是一些系统的基础信息。

2. 使用 Include 语法引用其他 SLS 文件

使用 Include 语法主要是为了提高 SIS 的重用性，让我们写过的 SLS 可以得到重复的利用，如图 4.7 所示。

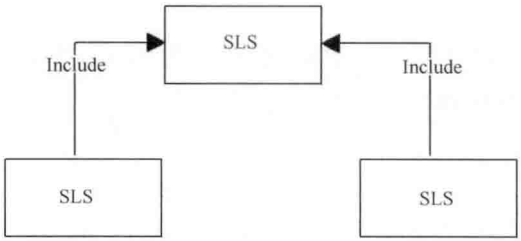


图 4.7　SIS 文件 Include 示例

以下为代码实例：

```
include:
  - python.python-libs

django:
  pkg.installed:
- require:
    - pkg: python-dateutil
```

3. 使用 Extend 扩展引用的 SLS

当继承下来的 SLS 文件需要扩展的时候，我们会面临这样一个问题，究竟是改动引用

的 SLS 文件还是新做一份 SLS 文件被引用呢？SaltStack 为我们提供了 Extend 的功能，让我们可以在“include”一份 SLS 文件的基础上，同时还能扩展它，解决了子配置文件需要扩展父配置文件的问题，如图 4.8 所示。

```
include:
  - apache.apache

extend:
apache:
service:
- running
- watch:
        - file: /etc/httpd/extra/httpd-vhosts.conf

/etc/httpd/extra/httpd-vhosts.conf:
  file.managed:
    - source: salt://apache/httpd-vhosts.conf
```

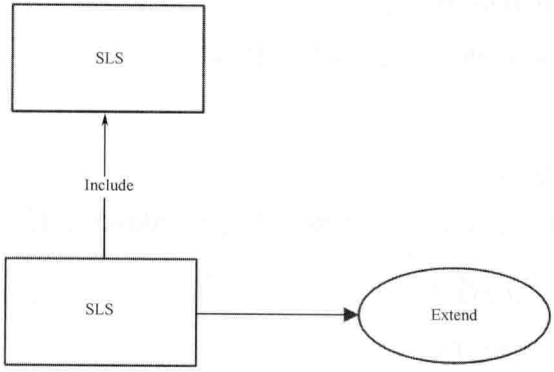


图 4.8　SLS 文件 Extend 示例

4. 使用 Watch 覆写 SLS

除了能够对父 SLS 文件的内容进行扩展，子 SLS 文件还具备了覆写父 SLS 文件的功能，达到能够灵活地对配置进行引用，最大限度地复用 SLS 文件的目的。

```
include:
  - apache.apache

extend:
```

```
    apache:
      service:
        - running
        - watch:
          - file: mywebsite
  mywebsite:
    file.managed:
      - name: /etc/httpd/extra/httpd-vhosts.conf
      - source: salt://apache/httpd-vhosts.conf
```

4.5 无主服务器模式运行

Salt-Minion 是可以脱离 SaltMaster 器运行的，脱离 SaltMaster 的 Minion 可以完成以下事情：

- 脱离 Master 使用 salt-call 命令；
- 无 Master 的 States，通过本地 States 文件执行。

配置方式如下所述。

步骤一：配置 SaltMinion。

编辑/etc/salt/minion 文件，告诉 minion 是脱离 SaltMaster 运行的：

```
file_client: local
```

步骤二：创建本地 States Tree。

创建/srv/salt/top.sls 文件：

```
base:
  '*':
    - webserver
```

创建/srv/salt/webserver.sls 文件：

```
apache:
  pkg:
    - installed
```

步骤三：本地调用 salt-call。

```
salt-call -local state.highstate -l debug
```

4.6 使用 SaltStack 的定时作业

由于 SaltStack 的 Minion 是可以自己独立跑的，所以当我们编写好模块后，可以调用 salt-call 让 Minion 自己执行命令。

```
salt-call cmd.run 'ls'
```

假如我们需要让 salt-call 定期跑呢？使用操作系统的 crontab 就可以了，在 crontab 里加入如下命令：

```
* * * * salt-call cmd.run 'ls'
```

4.7 实时执行命令

SaltStack 具备配置主机的功能，也具备直接执行命令的功能，可以让运维人员通过执行远程命令实时地对主机进行批量的操作。远程执行命令的语法如下：

```
salt '<target>' <function> [arguments]
```

下面对每个参数进行简单的介绍。

4.7.1 target

target 用于指定命令在哪些机器上执行，这里可以使用多种匹配 Minion 的语法，如使用通配符进行匹配：

```
salt '*' test.ping
```

1. 使用通配符进行匹配

匹配所有的 Minion：

```
salt '*' test.ping
```

通配符匹配：

```
salt '*.example.net' test.ping
salt '*.example.*' test.opping
```

数字的通配符匹配：

```
salt 'web?.example.net' test.ping
```

2. 范围匹配

匹配 web1~web5 的主机：

```
salt 'web[1-5]' test.ping
```

匹配 web1 和 web3 的主机：

```
salt 'web[1,3]' test.ping
```

匹配 web-x、 web-y、 web-z：

```
salt 'web-[x-z]' test.ping
```

3. 正则式匹配

匹配 web1-prod 和 web1-devel:

```
salt -E 'web1-(prod|devel)' test.ping
```

在 state 文件中使用正则匹配：

```
base:
  'web1-(prod|devel)':
  - match: pcre
  - webserver
```

4. 列表匹配

匹配 web1、web2、web3 的主机：

```
salt -L 'web1,web2,web3' test.ping
```

5. 使用 Grains 进行匹配

根据操作系统信息匹配：

```
salt -G 'os:CentOS' test.ping
```

列出可以使用的系统信息：

```
salt '*' grains.ls
```

列出详细信息：

```
salt '*' grains.items
```

配置 Grains，修改 Minion 的配置文件即可：

```
grains:
  roles:
    - webserver
    - memcache
  deployment: datacenter4
  cabinet: 13
  cab_u: 14-15
```

也可以在/etc/salt/grains 下配置：

```
roles:
  - webserver
  - memcache
deployment: datacenter4
cabinet: 13
cab_u: 14-15
```

在 top 文件中使用 Grain 进行匹配：

```
'node_type:web':
  - match: grain
  - webserver

'node_type:postgres':
  - match: grain
  - database

'node_type:redis':
```

```
    - match: grain
    - redis

  'node_type:lb':
    - match: grain
    - lb
```

6. 其他匹配方式

使用 IP 进行匹配：

```
salt -S 192.168.40.20 test.ping
salt -S 10.0.0.0/24 test.ping
```

在命令行中使用复合条件：

```
salt -C 'webserv* and G@os:Debian or E@web-dc1-srv.*' test.ping
```

在 SLS 文件中使用复合条件：

```
base:
  'webserv* and G@os:Debian or E@web-dc1-srv.*':
    - match: compound
    - webserver
```

根据节点组匹配（在 Master 配置文件中加入分组配置信息）：

```
nodegroups:
  group1: 'L@foo.domain.com,bar.domain.com,baz.domain.com or bl*.domain.com'
  group2: 'G@os:Debian and foo.domain.com'
```

命令行中使用分组匹配：

```
salt -N group1 test.ping
```

SLS 文件中使用分组匹配：

```
base:
  group1:
    - match: nodegroup
    - webserver
```

4.7.2 function

function 是 SaltStack 自带的函数，例如 cmd.run 就是在客户机上运行指定的命令：

```
salt '*' cmd.run 'uname -a'
```

和 Puppet 以及 Ansible 一样，SaltStack 也提供了许多可使用的模块，以下列举一些例子。

（1）查询所有设备 root 用户的 Crontab 信息。

```
salt '*' cron.list_tab root
```

（2）在 root 用户的 crontab 下添加定时任务。

```
salt '*' cron.set_job root '*' '*' '*' '*' 1 /usr/local/weekly
```

（3）重启所有设备。

```
salt '*' system.reboot
```

4.7.3 arguments

arguments 是调用函数时传入的参数，很多模块都需要使用者提供相应的参数才能正常运作，可以直接提供参数：

```
salt '*' cmd.exec_code python 'import sys; print sys.version'
```

也可以提供 key-value 类型的参数：

```
salt '*' pip.install salt timeout=5 upgrade=True
```

4.8 Pillar

Pillar 用于定义一些常用变量，Grains 虽然提供了操作系统信息的变量，但是当我们需要定义一些自定义变量的时候，就需要使用 Pillar。

Pillar 的存放位置是在 Master 上的配置文件中声明的，设置存放 Pillar 的路径的变量名

称为 pillar_roots。

虽然都能提供一些变量信息，但 Pillar 与 Grains 也有一些差异，如表 4.1 所示。

表 4.1　Pillar 和 Grains 的区别

参数	Pillar	Grains
存储的内容类型	动态的、变化的	静态的、不常变化的
存储位置	Master 本地	Minion 本地
操作权限	Minion 只允许查看自己的 Pillar	Minion 可以对自己的 Grains 值进行增加和删除

4.8.1　使用 Pillar

Pillar 的定义方法和 SLS 是一样的，首先要定义 top.sls 文件，然后再定义相应的模块文件。我们可以让不同操作系统的同一个变量为不同的变量值。

1. 定义 top.sls

让所有的设备都需要使用 packages 模块：

```
base:
  '*':
  packages
```

2. 编写 packages.sls

通过 Grains 获取系统的变量信息，假如系统是 RedHat 类型，则定义 apache 变量的值为 httpd，git 变量的值为 git；假如系统是 Debian 类型，则定义 apache 的变量为 apache2，git 变量的值为 git-core。

```
{% if grains['os'] == 'RedHat' %}
    apache: httpd
    git: git
{% elif grains['os'] == 'Debian' %}
    apache: apache2
    git: git-core
{% endif %}
```

3. 调用变量

变量定义完毕之后，就可以使用变量了，在 SLS 文件中，我们使用{{变量名称}}就可以获取具体的变量值了：

```
apache:
  pkg.installed:
  name: {{ pillar['apache'] }}
```

4.8.2 Pillar 的一些操作方法

查看 Minion 的 Pillar：

```
salt '*' pillar.items
```

使用 Pillar 的 Get 函数：当变量定义出现多层级的时候

```
foo:
  bar:
    baz: qux
```

可以通过常规的模式去获取 Pillar 的值：

```
{{ pillar['foo']['bar']['baz'] }}
```

也可以用 get 函数获取：

```
{{ salt['pillar.get']('foo:bar:baz', 'qux') }}
```

刷新 Minion 的 Pillar：

Pillar 的定义是在 Master 端的，定义完 Pillar 之后，我们需要把 Pillar 的值刷新到 Minion 中。

```
salt '*' saltutil.refresh_pillar
```

使用 Pillar 筛选 Minion：

```
salt -I 'somekey:specialvalue' test.ping
```

使用 HighState 设置 Pillar：

```
salt '*' state.highstate pillar='{"cheese": "spam"}'
```

4.9 小结

4.9.1 SaltStack 的优点

（1）SaltStack 具备了 Ansible 与 Puppet 两者的特性，既可以主动触发命令进行实时的主机批量的管理，又可以通过 SLS 文件对主机需要达到的状态进行描述，达到对主机进行集中化配置的目的，功能相当全面。

（2）既具备了安装 Minion 进行集中化管理的模式，又具备了纯 SSH 的无 Minion 的管理模式，可以让使用者根据自己的实际情况选择部署模式。

（3）能够集中化运维的操作系统类型相当全面，支持 CentOS、Windows、OSX 等操作系统。

4.9.2 SaltStack 的缺点

（1）集中化操作时返回结果不够友好，很多时候不成功的设备没有返回结果，导致运维人员需要一个个查找才能知道哪些成功，哪些没成功。

（2）没有源的情况下易部署性不高，安装较为不便。曾经有同事大面积部署 SaltStack 的时候，在其中两台不能联网的 SUSE 上安装 SaltMinion，那真是装到他“泪流满面”啊。

第5章 重复造一个轮子

5.1 从一个自动化运维软件说起

笔者曾经参与过一款自动化运维软件的设计，这款自动化运维软件的目标是对软件、操作系统进行一些自动化的运维，目的是帮助运维人员降低日常运维的工作量。

既然要做自动化运维，当然就需要有一个切入点，我们的切入点是监控，以监控作为驱动点来实现日常运维的自动化。不同的厂商都在做监控系统，从头做起并不是一个明智的选择，而作为一款成熟的监控软件，Zabbix 是非常不错的选择。于是我们在 Zabbix 的基础上进行了二次开发，实现了一套具备监控能力的系统，但是系统最终的目标不仅仅只是监控，还需要通过监控来驱动运维的自动化，具备对主机进行集中化运维以及自动化运维的功能。刚开始的时候，运维的功能需求并不多，也就三个，做的都是一些集中化的操作，包括对文件的集中下发、周期性地抓取一些文件、能够调用主机上面的一些命令，我们分别把这三个功能简称为文件下发、文件抓取、远程执行命令。这是一个自动化运维的系统需要具备的最基础的功能。

接到这个需求之后，几个负责这块开发的同事就开始对现有的开源软件进行调研了。一番调研之后，他们发现 Puppet 能够满足这些需求，并且在公司测试环境上也测试成功了。本以为凭借着 Puppet 所提供的现成的模块就能够很容易地满足这几个需求，结果跑去客户现场折腾了几天后，垂头丧气地回来了。

第二个星期，他们又开始尝试了 SaltStack，在公司测试环境测试通过之后，他们又跑

到客户现场开始折腾了。但是这次的试验结果和上一次一样，他们再次垂头丧气地回来了。

在这之后没多久，笔者就接手了这个运维平台功能的研发工作。当然，前面所讲到的文件下发、文件抓取、远程执行命令这三个需求也作为优先级最高的需求排上了日程。了解了这几个功能点具体希望完成的目标之后，第一感觉是这几个功能应该是非常简单并且容易实现的，而且作为基础功能，任何一款集中化运维软件应该都支持这几个功能。

文件下发的目标就是对用户所选择的设备进行批量的文件传输，把已经上传到服务器的文件传输到客户机上，常用于对配置文件、脚本文件或者安装包做集中管理，如图 5.1 所示。

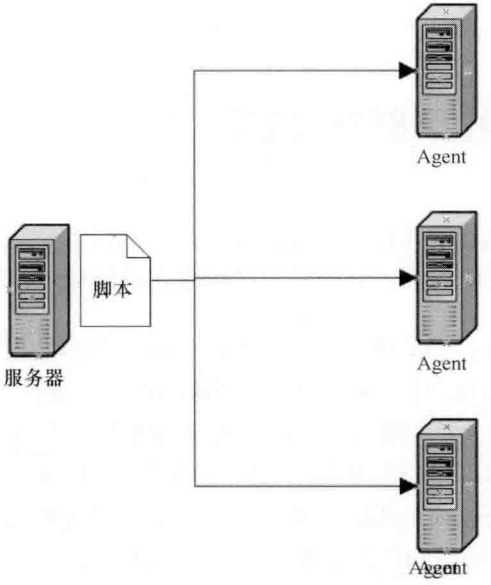


图 5.1　文件下发功能示意图

文件抓取的目标是将在客户机上生成的或者已经存在的文件抓取到服务器的指定目录下，如图 5.2 所示。比较常见的情况是把在服务器上生成好的巡检报告收取回服务器端，又或者是把一些配置文件收取回服务器端，用于做配置文件的基线对比。

远程命令执行的目标是对所有的客户机进行批量的命令执行，如图 5.3 所示。例如批量地执行主机上的巡检报告，批量地安装文件。主要是让运维人员感觉自己操作一台设备和操作多台设备是一样的，而且后续还能由告警进行驱动，进行特定命令的执行。

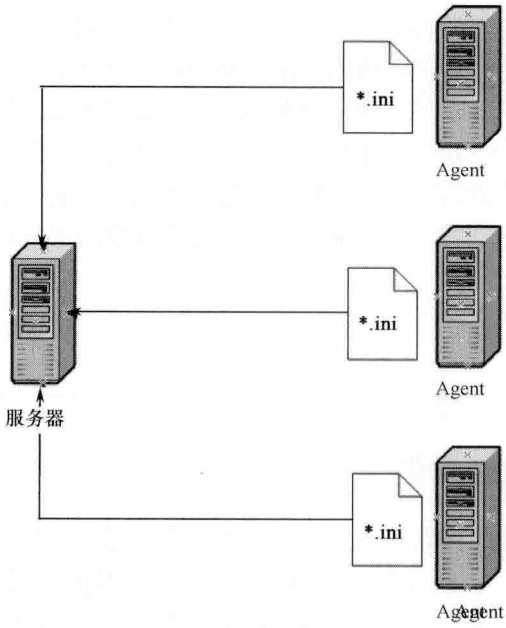


图 5.2　文件抓取示意图

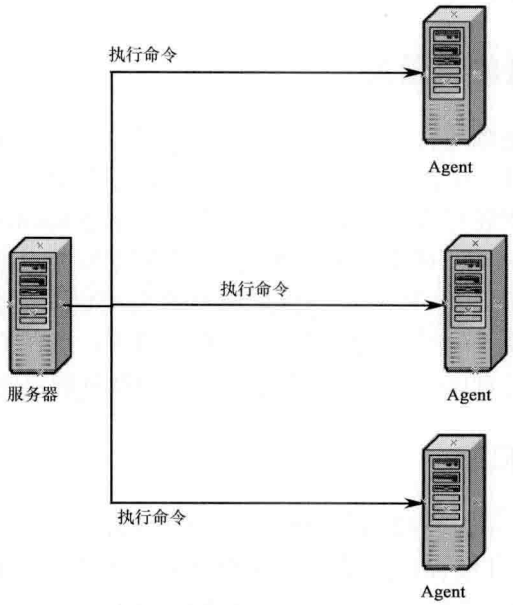


图 5.3　远程命令执行示意图

相信看完这几个需求点的功能描述之后，读者都会觉得非常简单，一下子就能实现，但接下来又或者会觉得疑惑，这些功能无论是 Puppet、SaltStack 还是 Ansible 都是支持的，而且都是最基础的功能，为什么笔者的同事到现场试验完之后，会觉得 Puppet 和 SaltStack 都不适用呢？且不说用一些成熟的开源软件，就算是自己写个简单的 Socket，应该也不是一件困难的事情才对。笔者当时也有这样的疑惑。于是，笔者带着疑问与去现场做试验的同事进行了讨论。讨论完之后，发现事情并没有笔者所想的那么简单。

5.2 困难重重

5.2.1 多样的设备类型

我们碰到的第一棘手的问题就是设备类型的问题，这个运维平台客户端所需要兼容的设备类型非常多，包括了 Solaris、Windows Server、RedHat、AIX、HP-UX 等，不同的操作系统又同时存在许多不同的版本，例如 Windows 的版本就从 2000 到 2012 各个版本都有，在这样的环境前提下，Ansible 首先就不可用了，因为除了要集中化管理 Linux 和 UNIX 主机之外，还需要对 Windows 主机进行管理。

5.2.2 运维设备的总量大

系统上线后，需要运维的目标环境将近一万台设备，在这个环境中，除了操作系统的种类繁多之外，还有另外一个问题——同一个种类的操作系统又具备很多个不同的版本，同一版本的操作系统上所装有的软件又不一样，例如同样是 Solaris 的操作系统，一部分是装了 GCC，一部分则连 GCC 都没有安装好。这给我们的安装带来了极大困难。在这种情况下，使用 Ruby 编写的 Puppet 和使用 Python 编写的 SaltStack 都不适合。需要根据每台设备做不同的操作而无法统一进行部署或者快速简单的部署，这种工作量实在太大了，我们的团队没有那么多的人员对这样规模的设备量进行一个批量的实施。

5.2.3 艰难的环境

由于环境上的约束，在实际部署的时候，这些具体需要运维的设备是无法联网的，而且有些时候甚至还不允许架设内部源，这个难题让本来打算用源来进行部分主机（如 Redhat、SUCE 等发现版）批量安装 Puppet 或者 SaltStack 的同事再次陷入了困境，不能使用源就意味着需要自己动手在设备上进行源码的编译，又或者寻找对应版本的安装包并且

把需要的依赖包都准备好，这也给现场部署人员带来了极大的工作量。其次，由于涉及的地市数量比较多，地市运维人员的技术水平参差不齐，让地市运维人员自己动手编译的可能性也不大，这又给我们在客户端部署上带来了更大的挑战。

5.2.4 多变的客户需求

客户的需求是十分多变的。现场的环境分为内部网络和外部网络，地市的设备要连接服务器就需要通过外部网络与服务器进行连接。

就拿文件抓取这个例子来说，最开始的需求是把所有的文件都抓取回客户端。没过多久后，需求就变了。文件应该分为两类，一类是普通配置文件，另外一类是日志类型的文件。对于配置文件，由于文件的内容比较小，所以需要被全部收取回主服务器，需求变化前的示意图如图 5.4 所示。

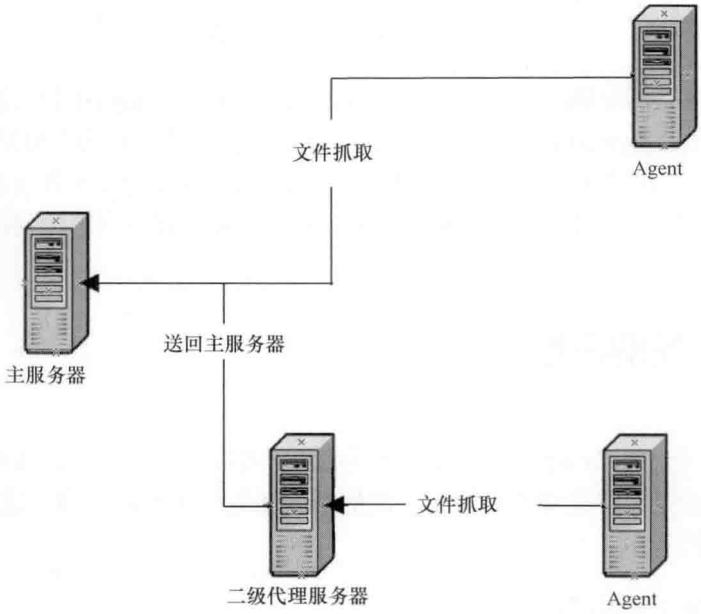


图 5.4 需求变化前的示意图

而对于较大的日志类型的文件，却只有在内部网络的二级代理才允许把文件送回应用服务器以供下载，而地市代理这种需要使用骨干网资源的二级代理，则不允许把文件直接送到主服务器，只有当需要的时候才把指定的文件送回主服务器。需求变化后的示意图如图 5.5 所示。

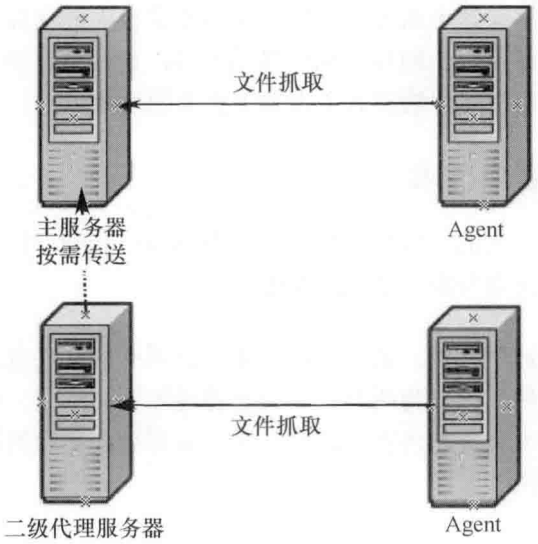


图 5.5　需求变化后的示意图

随着个性化场景的不断出现，我们不得不对 Puppet 或者 Ruby 的模块进行扩展。这就需要我们的开发人员对 Python 或者 Ruby 有一定的了解。但是当我们去求职网站上面找相应人才的时候，我们再一次陷入了困境，会 Python 并且会 SaltStack 的求职者或者是会 Ruby 并且会 Puppet 的求职者实在是太少了。即使进行内部培训也难以在短期内达到期望的效果。

5.3　轮子需要的特性

受制于上面所说的这些条件，我们不得不重新思考应该如何去设计这几个运维的功能。自己重新造一个轮子已经是势在必行了。既然需要新造一个产品，那么这个产品就必然有自己所需要的特性。

1．客户端/服务器端模式

针对我们前面所碰到的问题，我们提出了这个运维软件的基础架构是客户端/服务器端模式的。因为对于 Linux/UNIX 设备来说，我们可以使用 SSH 这种无 Agent 的模式进行集中化的运维，但是对于 Windows 来说，我们则需要用 WMI 的模式进行管理，采用无 Agent 的模式会给我们带来异构操作系统的问题。而且一旦具备了客户端，对于后续软件的开发也有极大的便利，我们可以更加容易地调用操作系统上的一些资源。

2. 易部署

这款运维软件应该是易于部署的，我们面对的环境实在太复杂了，这款运维软件一定是一款能够通过简单的命令就启动起来的软件。最重要的是，它必须是能够跨平台运行的。而跨平台最好的语言也就只有 Java 了，虽然 Java 的程序启动起来需要的内存比许多开发语言需要的内存都要多，但是它却能为我们解决部署以及跨平台的问题，从此不再需要考虑目标操作系统上究竟有没有 GCC 等问题了。但是我们还是会碰到有机器上是没有 JRE 的情况，所以我们的这个运维软件应该是和 JRE 打包在一起的。

3. 热更新

由于多方的压力，我们开发这款运维软件的时间不能太长，并且我们也没有一些历史的积累，所以要保证这款软件具备热更新的能力。因为水平有限，我们没有办法保证自己写出来的程序一定没有问题。所以笔者期望当发现程序有问题时，可以通过简单的替换文件就可以完成运维软件的升级。不然一旦发现程序出现问题，更新的工作量简直不堪设想。

4. 插件化

后期的运维功能可能会越来越多，笔者期望每个运维功能都是一个插件，每个插件都具备自己独特的运维功能。一方面当客户端需要新增运维功能的时候，只需要把插件放到客户机上，就可以完成客户端运维功能的扩充。另一方面，由于插件彼此独立运行，可以保证即使某一插件出现问题，也不影响整体的运维功能。

5. 通信上具备自动重连的机制

由于网络可能会出现不稳定或者中间件服务器停机的情况，所以在架构上我们的客户端和服务器端都需要具备自动重连的机制。一旦出现连接中断了，依然可以通过这种机制进行重新连接，而不是需要对软件进行重新启动的操作。

6. 尽量使用成熟稳定的开源软件进行组合

我们应该选用成熟稳定的开源软件进行组合来完成这款运维软件，而不是从头开始写所有的功能，不然实在是太费时费力了，而且写出来的程序稳定性也不高。

7. 服务端需要支持分布式

我们的目标运维服务器的最大接入量为 10000 台，一台服务器对这样一个数量的客户机进行管控，到了一定数量级的时候很有可能会出现性能上的瓶颈，所以我们需要让服务

端支持分布式的特性，而且最好是通过简单的部署就可以以服务器的水平扩展。

5.4 ActiveMQ 基础

决定好这款集中化运维软件需要的特性之后，就需要开始对其实现技术进行技术选型了。由于这款运维软件的基础架构是采用客户端/服务器端模式进行设计的，我们需要考虑的第一个问题就是通信。

一讲到通信，我们首先会想到的就是采用 Socket，但是使用 Socket 会给我们带来很多额外的问题。比较典型的问题就是当网络拓扑出现二级代理时，我们需要额外考虑如何做网络包的转发，如图 5.6 所示。

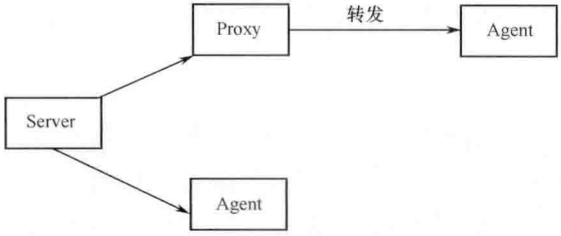


图 5.6　二级代理情况下网络包的转发

当我们需要实现文件收取的需求时，我们又会碰到网络并发的问题，程序没有花一定的时间进行测试，我们不能确定二级代理是不是能够承受一个较大规模的并发量，如图 5.7 所示。

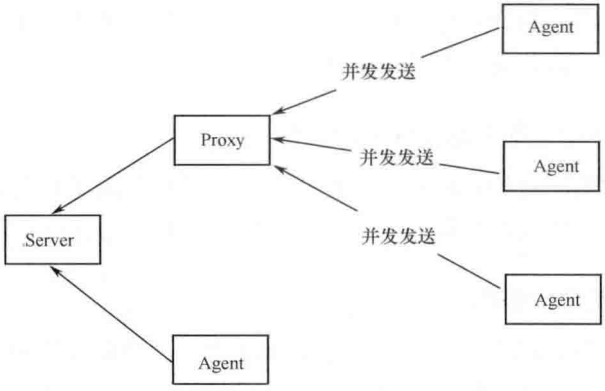


图 5.7　并发传输的问题

所以为了能够快速、可靠地完成客户端和服务器端的通信，采用底层 Socket 进行通信的方式是不明智的，所以我们采用了消息中间件技术来实现客户端和服务器端的通信。因为采用消息中间件可以为我们屏蔽许多技术难点，为软件开发争取了不少时间，并且能够保证客户端和服务器端通信的稳定性，而且往往成熟的消息中间件都有 HA 方案以及负载均衡的方案，我们直接拿来用就可以了。

开源的 MQ 技术有许多，究竟使用哪一款呢？经过我们的一番研究，最后我们选择的消息中间件技术为 ActiveMQ。ActiveMQ 是一个用 Java 编写的消息中间件，提供的功能与我们的上面所描述的需要支持的特性是比较贴切的。

从开发的角度来看，首先，ActiveMQ 除了提供标准 JMS 所提供的功能之外，而且还在 JMS 的基础上实现了 BlobMessage 的传输，大大减少了对文件传输的开发量。其次，ActiveMQ 提供的 Failover 的机制可以解决断线重连的问题，一旦客户端或者服务器端出现断线的情况，具备这种机制的客户端或服务器端都可以非常稳定地运行。

从部署和维护的角度来看，首先，采用基于 Java 开发的 ActiveMQ 具备了很强的易部署性，部署人员只需要通过简单的脚本就可以完成 ActiveMQ 的部署，大大地降低了部署的成本；再者，ActiveMQ 还具备了友好的管理界面，可以让维护人员能够轻松地掌握 ActiveMQ 的运行状态。

5.4.1 配置 ActiveMQ

ActiveMQ 有几个常用的配置文件，我们会经常需要调整它们，在实际部署 ActiveMQ 之前，我们先学习 ActiveMQ 中两份比较重要的配置文件。

1. ActiveMQ 的配置文件

activemq.xml 是 ActiveMQ 的配置文件，这份文件里面描述了 ActiveMQ 所需要侦听的端口、端口的最大连接数等内容。以下是 ActiveMQ 的一份配置清单：

```
<beans

  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
        http://activemq.apache.org/schema/core
        http://activemq.apache.org/schema/core/activemq-core.xsd">
          <bean
          class="org.springframework.beans.factory.config.PropertyPlaceholde
rConfigurer">
        <property name="locations">
        <value>file: ${activemq.conf}/credentials.properties</value>
        </property>
        </bean>

        <bean id="logQuery" class="org.fusesource.insight.log.log4j.Log4jLogQuery"
                lazy-init="false" scope="singleton"
                init-method="start" destroy-method="stop">
        </bean>

        <broker
            xmlns="http://activemq.apache.org/schema/core"
            brokerName="localhost"
            dataDirectory="${activemq.data}">

        <destinationPolicy>
        <policyMap>
        <policyEntries>
        <policyEntry topic=">">
        <pendingMessageLimitStrategy>
        <constantPendingMessageLimitStrategy limit="1000"/>
        </pendingMessageLimitStrategy>
        </policyEntry>
        </policyEntries>
        </policyMap>
        </destinationPolicy>
        <managementContext>
        <managementContext createConnector="false"/>
        </managementContext>

        <persistenceAdapter>
        <kahaDB directory="${activemq.data}/kahadb"/>
```

```
        </persistenceAdapter>

        <systemUsage>
        <systemUsage>
        <memoryUsage>
        <memoryUsage percentOfJvmHeap="70" />
        </memoryUsage>
        <storeUsage>
        <storeUsage limit="100 gb"/>
        </storeUsage>
        <tempUsage>
        <tempUsage limit="50 gb"/>
        </tempUsage>
        </systemUsage>
        </systemUsage>

        <transportConnectors>
            <transportConnector
            name="openwire"
            uri="tcp://0.0.0.0:61616?maximumConnections=1000&amp;wireFormat.maxF
rameSize=104857600"/>
            <transportConnector
            name="amqp"
            uri="amqp://0.0.0.0:5672?maximumConnections=1000&amp;wireFormat.maxFr
ameSize=104857600"/>
            <transportConnector
            name="stomp"
            uri="stomp://0.0.0.0:61613?maximumConnections=1000&amp;wireForm
at.maxFrameSize=10
            4857600"/>
            <transportConnector
            name="mqtt"
            uri="mqtt://0.0.0.0:1883?maximumConnections=1000&amp;wireFormat.maxFr
ameSize=104857600"/>
            <transportConnector
            name="ws"
            uri="ws://0.0.0.0:61614?maximumConnections=1000&amp;wireFormat.maxFr
ameSize=104857600"/>
```

```
        </transportConnectors>

        <shutdownHooks>
                    <bean
                    xmlns=http://www.springframework.org/schema/beans
         class="org.apache.activemq.hooks.SpringContextHook" />
        </shutdownHooks>
        </broker>
        <import resource="jetty.xml"/>
        </beans>
```

首先是 constantPendingMessageLimitStrategy 这个节点的配置，这个配置表明了如果一个消费者消费消息的速度比较慢，就会对非持久化主题产生影响，因为消息会在 broker 的内存中大量堆积。由于开启了 producer-flow-control，将导致所有的生产者生产消息的速度放慢，这样即使某些消费消息速度非常快的消费者也将放慢其消费消息的速度。

第二个常用的配置项是 systemUsage 节点下的配置项，这个配置项下的 memoryUsage 表示的是非持久化消息占用的内存大小，storeUsage 表示的是持久化消息所占用的磁盘大小，tempUsage 表示的是临时消息占用的磁盘大小。假如设置的磁盘存储最大值大于磁盘的实际可以存储值，ActiveMQ 会对磁盘使用的最大值进行重新分配。

最后一个常用的配置项是通信协议的配置项，transportConnectors 节点下声明了 ActiveMQ 支持的通信协议以及协议所侦听的端口。默认情况下，ActiveMQ 支持 OpenWire 协议、AMQP 协议、STOMP 协议、MQTT 协议以及 WS 协议，每个协议中的 maximumConnections 表示 ActiveMQ 的最大连接数，当我们把 ActiveMQ 投入生产的时候，这个值需要根据实际的情况进行对应的配置。

```
    <transportConnectors>
        <transportConnector
        name="openwire"
        uri="tcp://0.0.0.0:61616?maximumConnections=1000&amp;wireFormat.maxFr
ameSize=104857600"/>
        <transportConnector
        name="amqp"
        uri="amqp://0.0.0.0:5672?maximumConnections=1000&amp;wireFormat.maxFra
meSize=104857600"/>
        <transportConnector
        name="stomp"
```

```
        uri="stomp://0.0.0.0:61613?maximumConnections=1000&amp;wireFormat.maxF
rameSize=104857600"/>
        <transportConnector
        name="mqtt"
        uri="mqtt://0.0.0.0:1883?maximumConnections=1000&amp;wireFormat.maxFr
ameSize=104857600"/>
        <transportConnector
        name="ws"
        uri="ws://0.0.0.0:61614?maximumConnections=1000&amp;wireFormat.maxFra
meSize=104857600"/>
    </transportConnectors>
```

2. Jetty 的配置文件

读者可能会感觉到奇怪，为什么 ActiveMQ 里面会有 Jetty 的配置文件呢？这是由于 ActiveMQ 自身内嵌了一个 Jetty 的服务器，里面承载了管理界面的容器以及用于对文件进行传输的服务 FileServer。

```
        <beans
            xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

        <bean
        id="securityLoginService"
class="org.eclipse.jetty.security.HashLoginService">
        <property name="name" value="ActiveMQRealm" />
        <property                                          name="config"
value="${activemq.conf}/jetty-realm.properties" />
        </bean>

        <bean                                      id="securityConstraint"
class="org.eclipse.jetty.util.security.Constraint">
        <property name="name" value="BASIC" />
        <property name="roles" value="user,admin" />
        <!-- set authenticate=false to disable login -->
        <property name="authenticate" value="true" />
        </bean>
```

```
        <bean
        id="adminSecurityConstraint"
class="org.eclipse.jetty.util.security.Constraint">
        <property name="name" value="BASIC" />
        <property name="roles" value="admin" />
        <!-- set authenticate=false to disable login -->
        <property name="authenticate" value="true" />
        </bean>
        <bean
        id="securityConstraintMapping"
    class="org.eclipse.jetty.security.ConstraintMapping">
        <property name="constraint" ref="securityConstraint" />
        <property name="pathSpec" value="/admin/*,*.jsp" />
        </bean>
        <bean
        id="adminSecurityConstraintMapping"
    class="org.eclipse.jetty.security.ConstraintMapping">
        <property name="constraint" ref="adminSecurityConstraint" />
        <property name="pathSpec" value="*.action" />
        </bean>
        <bean
        id="securityHandler"
class="org.eclipse.jetty.security.ConstraintSecurityHandler">
        <property name="loginService" ref="securityLoginService" />
        <property name="authenticator">
        <bean
class="org.eclipse.jetty.security.authentication.BasicAuthenticator" />
        </property>
        <property name="constraintMappings">
        <list>
        <ref bean="adminSecurityConstraintMapping" />
        <ref bean="securityConstraintMapping" />
        </list>
        </property>
        <property name="handler">
        <bean
        id="sec" class="org.eclipse.jetty.server.handler.HandlerCollection">
```

```
<property name="handlers">
<list>
<bean class="org.eclipse.jetty.webapp.WebAppContext">
<property name="contextPath" value="/hawtio" />
<property
name="war"
value="${activemq.home}/webapps/hawtio" />
<property name="logUrlOnStart" value="true" />
</bean>
<bean class="org.eclipse.jetty.webapp.WebAppContext">
<property name="contextPath" value="/admin" />
<property
name="resourceBase" value="${activemq.home}/webapps/admin" />
<property name="logUrlOnStart" value="true" />
</bean>
<bean class="org.eclipse.jetty.webapp.WebAppContext">
<property name="contextPath" value="/fileserver" />
<property
name="resourceBase"
value="${activemq.home}/webapps/fileserver" />
<property name="logUrlOnStart" value="true" />
<property
name="parentLoaderPriority" value="true" />
        </bean>
<bean class="org.eclipse.jetty.webapp.WebAppContext">
<property name="contextPath" value="/api" />
<property
name="resourceBase" value="${activemq.home}/webapps/api" />
<property name="logUrlOnStart" value="true" />
</bean>
<bean class="org.eclipse.jetty.server.handler.ResourceHandler">
<property name="directoriesListed" value="false" />
<property name="welcomeFiles">
<list>
<value>index.html</value>
</list>
</property>
```

```
        <property
        name="resourceBase" value="${activemq.home}/webapps/" />
        </bean>
        <bean
        id="defaultHandler"
class="org.eclipse.jetty.server.handler.DefaultHandler">
        <property name="serveIcon" value="false" />
        </bean>
        </list>
        </property>
        </bean>
        </property>
        </bean>

        <bean id="rewrite" class="org.eclipse.jetty.rewrite.handler.RewriteHandler">
        <property name="rules">
        <set>
        <bean class="org.eclipse.jetty.rewrite.handler.RedirectRegexRule">
        <property name="regex" value="/api/jolokia(.*)"/>
        <property name="replacement" value="/hawtio/jolokia$1"/>
        </bean>
        </set>
        </property>
        </bean>

        <bean
        id="contexts"
class="org.eclipse.jetty.server.handler.ContextHandlerCollection">
        </bean>

        <bean  id="jettyPort"  class="org.apache.activemq.web.WebConsolePort"
init-method="start">
        <!-- the default port number for the web console -->
        <property name="port" value="8161"/>
        </bean>
```

```
        <bean                id="Server"                depends-on="jettyPort"
class="org.eclipse.jetty.server.Server" init-method="start"
              destroy-method="stop">

        <property name="connectors">
        <list>
        <bean
        id="Connector"
class="org.eclipse.jetty.server.nio.SelectChannelConnector">
        <!-- see the jettyPort bean -->
        <property
        name="port" value="#{systemProperties['jetty.port']}" />
        </bean>
        <!--
        <bean id="SecureConnector" class="org.eclipse.jetty.server.ssl.SslSelect
ChannelConnector">
        <property name="port" value="8162" />
        <property name="keystore" value="file: ${activemq.conf}/broker.ks" />
        <property name="password" value="password" />
        </bean>
                     -->
        </list>
        </property>

        <property name="handler">
        <bean
        id="handlers"
class="org.eclipse.jetty.server.handler.HandlerCollection">
        <property name="handlers">
        <list>
        <ref bean="rewrite"/>
        <ref bean="contexts" />
        <ref bean="securityHandler" />
        </list>
        </property>
        </bean>
        </property>
```

```
    </bean>
    </beans>
```

这份配置文件中我们经常需要调整的配置项为 jettyPort 节点，这个节点声明了 Jetty 容器启动时所侦听的端口，与后续我们使用 ActiveMQ 的管理界面以及发送 BlobMessage 时所用的端口息息相关。至于文件是如何进行传输的，我们后面再讲。

```
    <bean id="jettyPort"
    class="org.apache.activemq.web.WebConsolePort" init-method="start">
    <property name="port" value="8161"/>
    </bean>
```

5.4.2 部署 ActiveMQ

ActiveMQ 的部署非常简单，只需要下载 ActiveMQ 的部署包，通过以下命令就可以完成中间件的管理了。

启动 ActiveMQ：

```
./apache-activemq-5.9.0/bin/activemq start
```

停止 ActiveMQ：

```
./apache-activemq-5.9.0/bin/activemq stop
```

获取指定队列的信息：

```
./apache-activemq-5.9.0/bin/activemq query Demo
```

获取 ActiveMQ 队列的性能信息：

```
./apache-activemq-5.9.0/bin/activemq dstat
```

启动完成后，我们就可以进入 ActiveMQ 的管理界面了，输入 ip:8161 就可以进入 ActiveMQ 的管理界面。假如没有成功打开，可以查看一下./data 目录下的 activemq.log 日志文件。

Home 和 Queues 是我们使用频率很高的两个界面。

Home 页面主要提供了 ActiveMQ 的版本信息、启动时间信息、持久化消息磁盘使用率、内存使用率、临时消息使用率等信息，如图 5.8 所示。

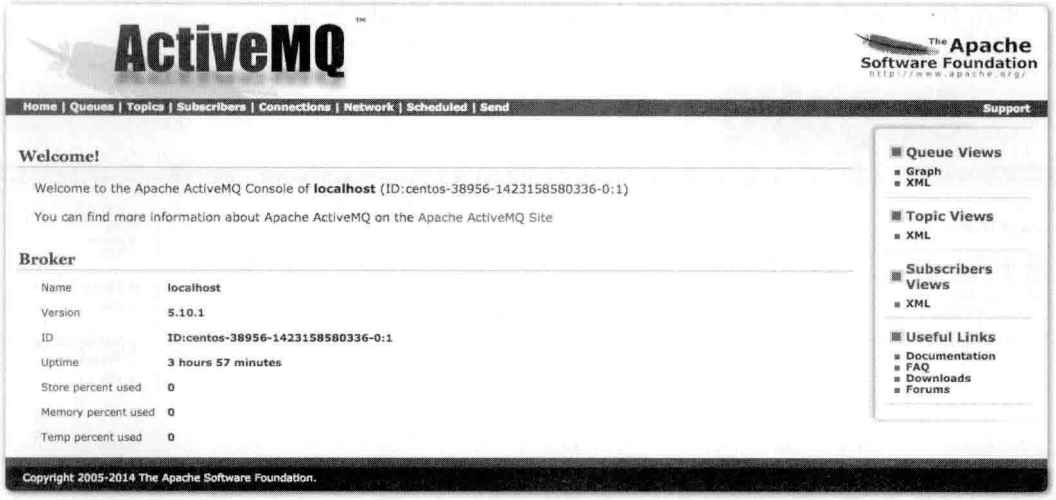


图 5.8 ActiveMQ 的管理界面

Queues 界面主要用于对队列信息进行管理，可以在这个界面上看到队列的名称（Name）、待消费的消息数（Number Of Pending Messages）、消费者数量（Number Of Consumera）、消息入队数量（Message Enqueued），以及消息出队数量（Message Dequeued），如图 5.9 所示。

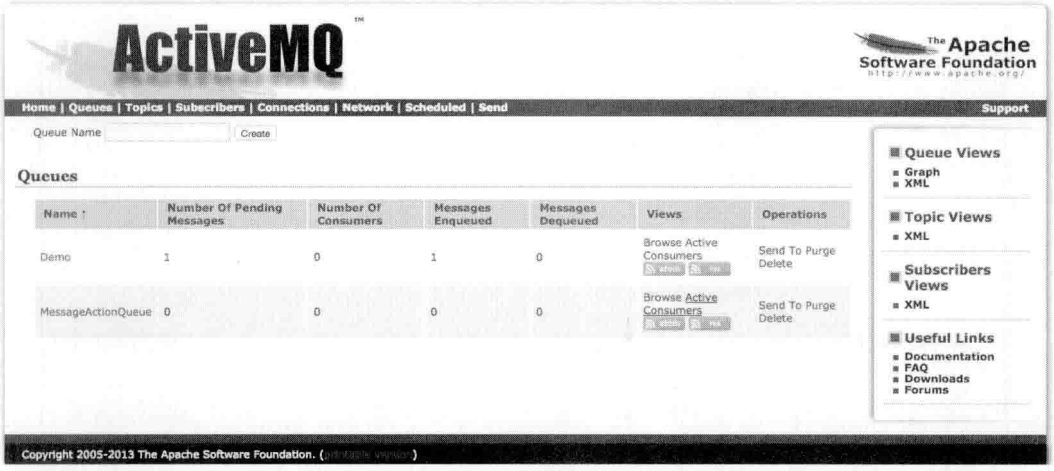


图 5.9 ActiveMQ 的队列查看界面

单击 Browse 可以查看具体队列上的消息量，如图 5.10 所示。

图 5.10　ActiveMQ 的队列信息界面

再次单击后还能看到具体的消息内容，如图 5.11 所示。

图 5.11　ActiveMQ 的消息详情页面

这在我们做 Agent 和 Server 之间的消息调试时是非常有用的，在程序调试的时候，我们可以先停下 Agent，让 Server 发送消息到消息中间件，然后我们就可以通过 Queues 去查看发送的消息是否有误。接着我们可以再启动 Agent，把消息从队列上收下来，然后继续对程序进行调试。

5.4.3 第一个 ActiveMQ 例子

1. 消息生产者

```
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.ActiveMQSession;

public class Producter {

    public static void main(String[] args) throws JMSException {
        String url = "tcp://192.168.41.136:61616?"
                + "jms.blobTransferPolicy.defaultUploadUrl="
                + "http://192.168.41.136:8161/fileserver/";
        ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(
                "admin", "admin", url);
        ActiveMQConnection connection;
        connection = (ActiveMQConnection)
connectionFactory.createConnection();
        connection.start();
        ActiveMQSession session = (ActiveMQSession)
connection.createSession(
                false, Session.AUTO_ACKNOWLEDGE);
        Destination destination = session.createQueue("demo");
        MessageProducer messageProducer =
session.createProducer(destination);
        TextMessage message = session.createTextMessage();
```

```
            message.setText("This is Message");
            messageProducer.send(message);
            messageProducer.close();
            session.close();
            connection.close();
        }
    }
```

首先，我们创建了一个连接，URL 里面的 61616 是 ActiveMQ 侦听的端口，而 8161 则是 Jetty 侦听的端口，在发送文件消息的时候使用。

接着我们创建了一个 ActiveMQ 的工厂连接，最后启动这个连接，让程序连接上 ActiveMQ。

```
String url = "tcp://192.168.41.136:61616?"
                + "jms.blobTransferPolicy.defaultUploadUrl="
                + "http://192.168.41.136:8161/fileserver/";
ActiveMQConnectionFactory connectionFactory
                = new ActiveMQConnectionFactory("admin", "admin", url);
ActiveMQConnection connection;
connection = (ActiveMQConnection) connectionFactory.createConnection();
connection.start();
```

接着需要创建一个会话，这个会话说明了使用什么样的机制去发送消息。第一个参数声明了要使用什么方式进行消息的签收，true 表示带事务的 session，一旦事务提交成功，消息就会被自动签收，如果事务回滚了，消息会重新发送；false 则代表 session 的签收方式是不带事务的，这种签收方式取决于第二个参数，第二个参数所支持的模式有三种，如表 5.1 所示。

表 5.1　ActiveMQ 的签收信息模式

模　式	说　明
Session.AUTO_ACKNOWLEDGE	消息自动签收
Session.CLIENT_ACKNOWLEDGE	客户端调用 acknowledge 方法手动签收
Session.DUPS_OK_ACKNOWLEDGE	不是必须签收，消息可能会重复发送。在第二次重新传送消息的时候，消息头的 JmsDelivered 会被置为 true，表示当前消息已经传送过一次，客户端需要进行消息的重复处理控制

```
ActiveMQSession session = (ActiveMQSession) connection.createSession(
                false, Session.AUTO_ACKNOWLEDGE);
```

然后，我们需要告诉程序我们使用的队列是哪一个，这里所使用的队列名称为 demo，使用 session 创建了一个 TextMessage 的生产者。然后设置了 TextMessage 消息的消息内容，接着再使用生产者把消息送到队列里面。

```
Destination destination = session.createQueue("demo");
MessageProducer messageProducer = session.createProducer(destination);
TextMessage message = session.createTextMessage();
message.setText("This is Message");
messageProducer.send(message);
```

最后，我们把生产者、会话以及 ActiveMQ 连接都关闭。

```
messageProducer.close();
session.close();
connection.close();
```

这样就完成了整个消息的发送，我们打开 Queue 界面就可以看到，队列里面生成了一个 demo 队列，并且里面有一条文本消息，消息的内容是 This is Message，如图 5.12、图 5.13 所示。

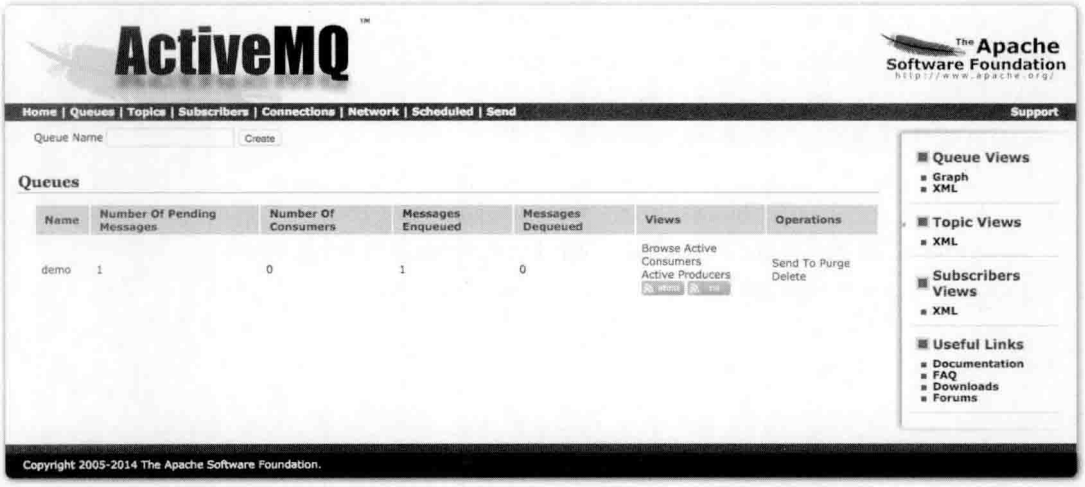


图 5.12　查看消息是否成功发送

图 5.13　查看消息发送的详细内容

2. 消息消费者

```
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.ActiveMQSession;

public class Comsumer {

    public static void main(String[] args) throws JMSException {
        String url = "tcp://192.168.41.136:61616?"
                + "jms.blobTransferPolicy.defaultUploadUrl="
```

```
                    + "http://192.168.41.136:8161/fileserver/";
            ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(
                    "admin", "admin", url);
            ActiveMQConnection connection;
            connection = (ActiveMQConnection)
connectionFactory.createConnection();
            connection.start();
            ActiveMQSession session = (ActiveMQSession)
connection.createSession(
                    false, Session.AUTO_ACKNOWLEDGE);
            Destination destination = session.createQueue("demo");
            MessageConsumer messageConsumer =
session.createConsumer(destination);
            while (true) {
                TextMessage message = ((TextMessage)
messageConsumer.receive());
                System.out.println(message.getText());
            }
        }
    }
```

消息的消费者与消息的生产者在初始化的部分是一样的，当会话和使用队列都初始化完毕时，我们使用 session 创建一个消息的消费者。

```
    MessageConsumer messageConsumer = session.createConsumer(destination);
```

接着，我们写一个循环，不断地从 demo 队列里面收取消息，然后转换成 TextMessage，再把消息的内容打印出来。

```
    while (true) {
        TextMessage message = ((TextMessage) messageConsumer.receive());
            System.out.println(message.getText());
        }
```

可以看到控制台里面打印了刚才发送到队列里面的信息，如图 5.14 所示。

```
Problems  Javadoc  Declaration  Console  Progress
Comsumer [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java (2015年2月7日 下午9:27:33)
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
This is Message
```

图 5.14　消息消费内容

再打开 ActiveMQ 的管理界面，可以看到刚才在队列里面的消息已经不见了，并且 Number Of Consumers 的数量从 0 增加到了 1，表明有一个消费者正在对这个队列进行侦听，如图 5.15 所示。

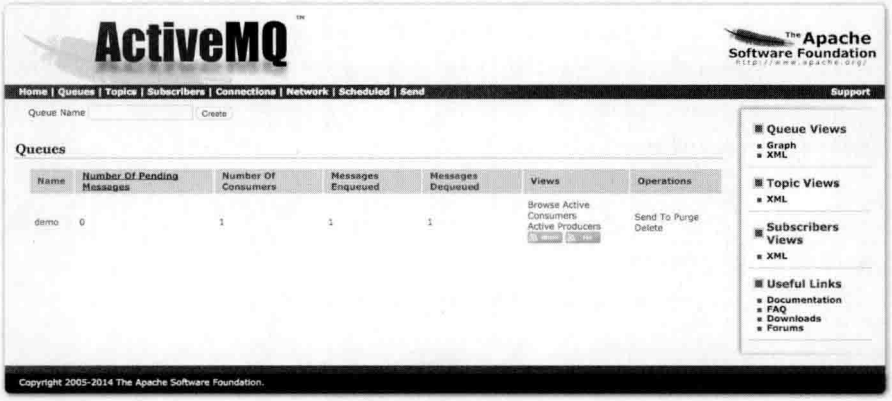


图 5.15　队列中的消息情况

单击 Active Consumers，查看正在对这个队列进行侦听的队列，如图 5.16、图 5.17 所示。

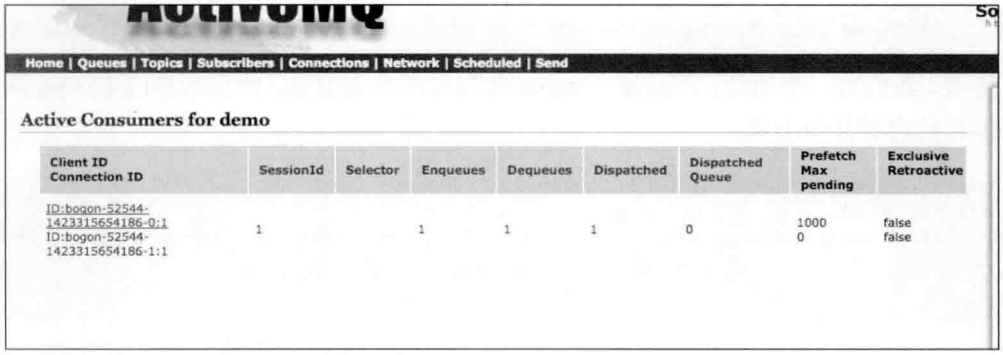


图 5.16　队列消费情况

Connection ID:bogon-52544-1423315654186-0:1

Connection ID	ID:bogon-52544-1423315654186-0:1
Remote Address	tcp://192.168.41.1:52545
Active	true
Connected	true
Blocked	false
Slow	false
Dispatch Queue Size	0

Consumers

Destination ?	SessionId	Selector	Enqueues	Dequeues	Dispatched	Dispatched Queue	Prefetch Max pending	Exclusive Retroactive
Queue demo	1		1	1	1	0	1000 0	false false
Topic ActiveMQ.Advisory.TempQueue,ActiveMQ.Advisory.TempTopic	-1		0	0	0	0	1000 0	false false

图 5.17　队列侦听情况

可以看到是 192.168.41.1 这个 IP 的消费者正在对这个队列的内容进行消费。当我们发现队列有 2 个消费者同时进行侦听的时候，使用 ActiveMQ 的这几个管理界面能够帮助我们很好地确定究竟是哪个设备正在对队列进行侦听。

5.5　Apache Karaf

5.5.1　OSGi 简介

OSGi 技术是面向 Java 的动态模型系统。Java 为其提供了在多个平台支持产品的可移植性的特性。OSGi 服务平台为开发者提供了程序热更新的特性，可以让应用程序（在 OSGi 的世界里面，应用程序指的是 OSGi 的 Bundle）不需要重启就完成代码的热更新。为了让各个插件达到高内聚、低耦合的目的，OSGi 技术还提供了一种面向服务的架构，它能使这些组件动态地发现其他插件。

OSGi 规范的核心组件是 OSGi 框架。这个框架为应用程序提供了一个标准环境。整个框架可以划分为 4 个层次：

- L0——运行环境；
- L1——模块；
- L2——生命周期管理；

- L3——服务注册。

还有一个无处不在的安全系统渗透到所有层，如图 5.18 所示。

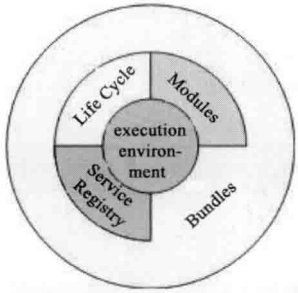


图 5.18　OSGi 的框架层次

5.5.2　为什么选择 Karaf

因为 OSGi 的特性，使得整个程序在设计上就是插件化的，但是 OSGi 容器有许多，我们当时挑选 OSGi 容器的原则有两点：第一点是容器要足够小，第二点是一定要具备热更新的特性。在权重上第二点会比第一点更大。因为不能自动完成插件包的热更新，批量更新插件包时会给我们带来非常大的工作量。在选型的时候，Apache Felix 和 Apache Karaf 都在备选名单之中。在资源的占用上，Apache Felix 和 Apache Karaf 所占用的资源是差不多的。在文件大小上，Apache Felix 的文件大小小于 10 MB，而 Apache Karaf 所占用的磁盘空间则将近 30 MB，所以在文件大小方面，Apache Felix 是具备优势的。但是 Apache Felix 需要进行手动执行脚本才能完成热部署的过程，而 Apache Karaf 可以自动完成插件包的热部署过程，所以，我们最后选择的 OSGi 容器是 Apache Karaf。Apache Felix 与 Apache Karaf 选项评分表如表 5.2 所示。

表 5.2　选项评分表

参　数	Apache Felix	Apache Karaf
资源占用	—	S
文件大小	—	-
自动热部署特性	—	+

5.5.3　基础架构设计

当 OSGi 容器挑选完毕之后，结合我们之前所选择的 ActiveMQ，整个运维软件的基础

架构也就定下来了，如图 5.19 所示。

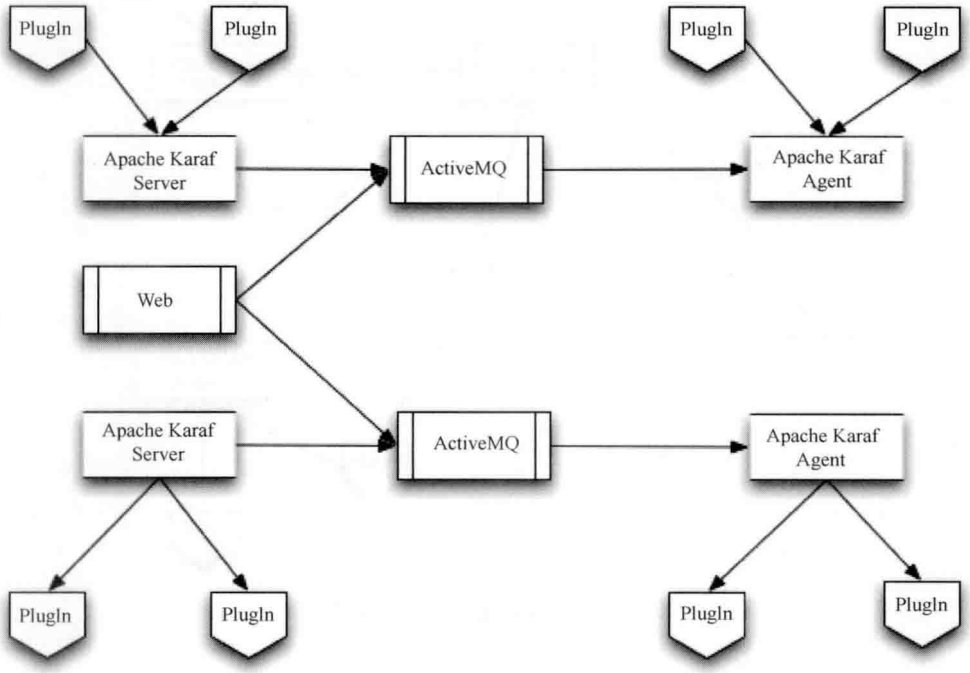


图 5.19 运维框架基础架构

我们在客户端和服务端都部署 Karaf，在客户机上放 Agnet 端专用的插件包，在 Server 端放 Server 端的专用插件包。Agent 和 Server 端的插件包则通过 MQ 进行消息的收/发。也就是说，服务端会有这么一个插件包，它会接收界面传过来的消息，然后针对消息进行处理或者转发，一旦客户端收到则会做出相应的动作，然后再进行动作的汇报。

当需要对插件包进行更新的时候，我们可以通过文件传输的方式，直接把 Karaf Agent 端的插件包替换掉。由于 Karaf 的热部署的特性，使得插件包会被自动重新加载，达到客户端热更新的效果。

基于 Apache ActiveMQ 和 Apache Karaf 这两项技术，我们就可以很轻易地开发出一套具备热插拔特性的运维基础框架了，如图 5.20 所示。

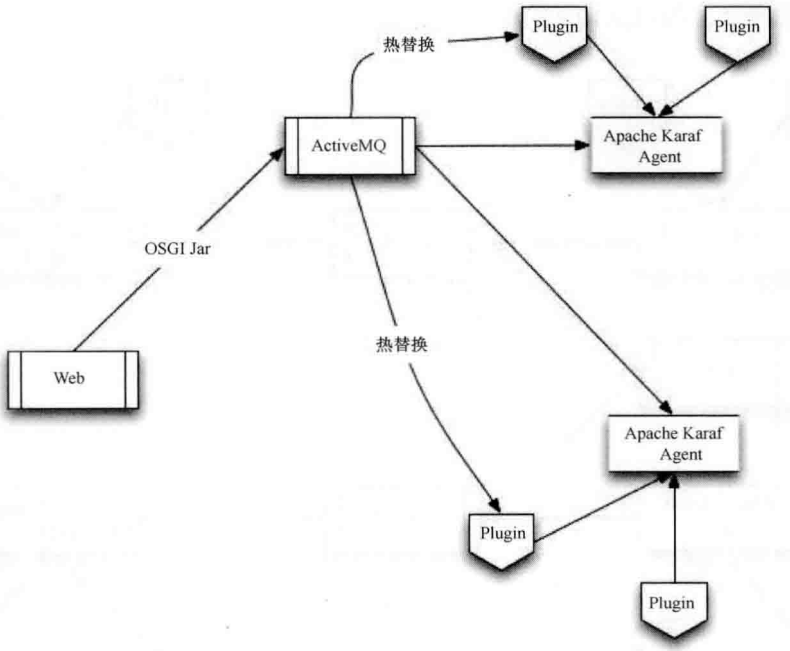


图 5.20　插件化运维框架设计

5.5.4　启动 Apache Karaf

解压完 Apache Karaf 之后，我们先看一下它的基本目录结构：

- bin；
- data；
- demos；
- deploy；
- etc；
- karaf-manual-2.3.4.html；
- karaf-manual-2.3.4.pdf；
- lib；
- LICENSE；
- NOTICE；

- README；
- RELEASE-NOTES；
- system。

bin 目录存放了 Karaf 的启停命令。data 目录为 Karaf 启动后生成的缓存文件。deploy 目录为热部署的目录，任何 OSGi 的程序放到这个目录下之后都会被启动起来。etc 目录为配置文件的目录。

接下来可以进入到 bin 目录下，我们会看到以下几份文件，如表 5.3 所示。

表 5.3 Apache Karaf bin 目录下的文件说明

admin	Karaf 管理工具，用于管理多实例时使用
client	连接到 karaf 容器，对 Karaf 进行操作
karaf	以前台运行模式进入 Karaf
setenv	配置常用环境变量
Shell	启动 Karaf，仅启动 shell 模块
start	以后台模式启动 Karaf
status	查看 Karaf 的状态
stop	停止 Apache Karaf

使用前台运行的模式运行 Karaf，执行./bin/karaf：

```
[root@centos bin]# ./karaf
        __ __                  ____
      / //_/____ _____ ____ _ / __/
     / ,<  / __ `/ __/ __ `/ /_
    / /| |/ /_/ / /  / /_/ / __/
   /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (2.3.4)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'osgi:shutdown' or 'logout' to shutdown Karaf.
```

至此，就完成 Karaf 的启动了。

5.5.5 制作第一个 OSGi 包

为了能够更好地理解热更新的特性，我们接下来会制作第一个 OSGi 包，然后将其部署到 Apache Karaf 容器中。

我们所使用的开发环境是 Eclipse，单击 New->Other，选择 Plug-in Project，如图 5.21 所示。接着填写项目的基本信息，如项目名称。这里需要注意的是 an OSGi FrameWork 的选

项，我们需要选择 standard，表示我们用的是标准的 OSGi 框架，如图 5.22 所示。

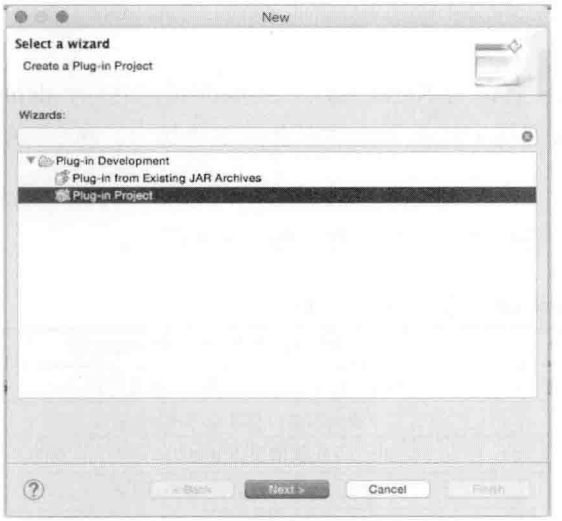


图 5.21　创建 OSGi Bundle

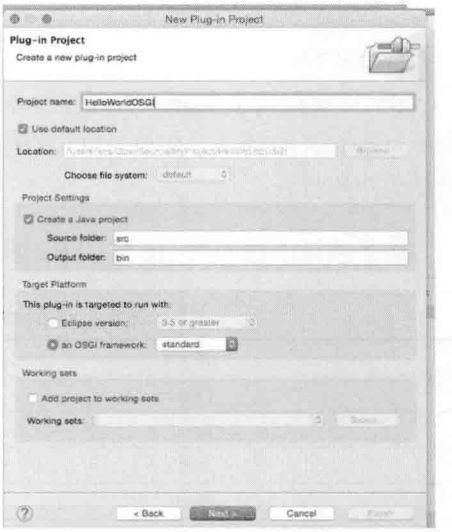


图 5.22　设置项目名称及使用的 OSGi 框架

接着我们需要设置一下这个 OSGi 包的版本，如图 5.23 所示。

使用 HelloWorld 的模板进行创建，如图 5.24 所示。

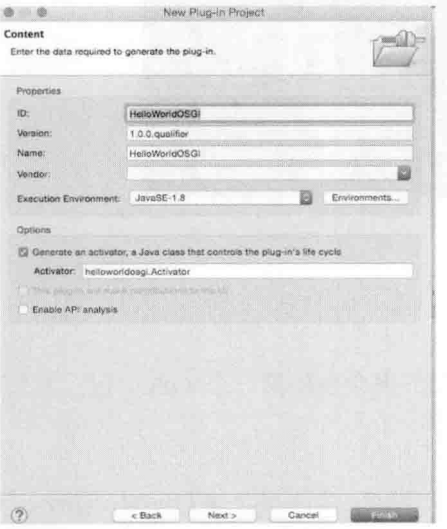


图 5.23　设置 OSGi 插件包的基础信息

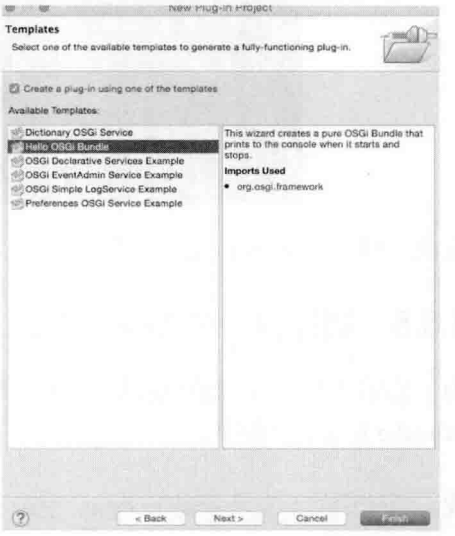


图 5.24　设置使用 HelloWorld 模板进行创建

设置模板中启动和停止所打印的消息内容，如图 5.25 所示。

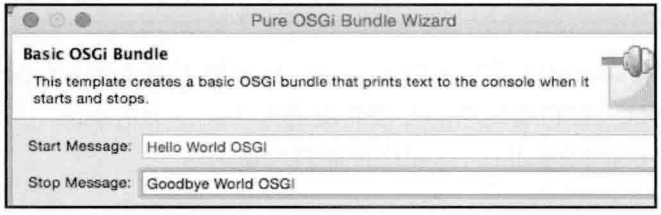


图 5.25　设置模板输出信息的内容

至此，一个最简单的 OSGi 包就创建完成了，接着我们来看一下 OSGi 项目的目录结构，如图 5.26 所示。

图 5.26　OSGi 的基本程序结构

标准的 OSGi 目录结构非常简单，它最主要的文件是 Java 代码和 MANIFEST.MF 文件。

```
package helloworldosgi;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

    public void start(BundleContext context) throws Exception {
        System.out.println("Hello World OSGI");
    }

    public void stop(BundleContext context) throws Exception {
```

```
            System.out.println("Goodbye World OSGI");
        }
    }
```

这是一个示例代码，可以看到这个 Activator 类实现了 BundleActivator 接口，表明它是一个 Bundle，然后这个接口有两个需要被实现的方法：一个是 start，另外一个是 stop。这两个方法分别会在 OSGi Bundle 启动和停止的时候调用。

MANIFEST.MF 这个文件是 OSGi 包种很重要的一份文件，打开这份文件，我们首先可以看到这样 Overview 的界面，如图 5.27 所示。

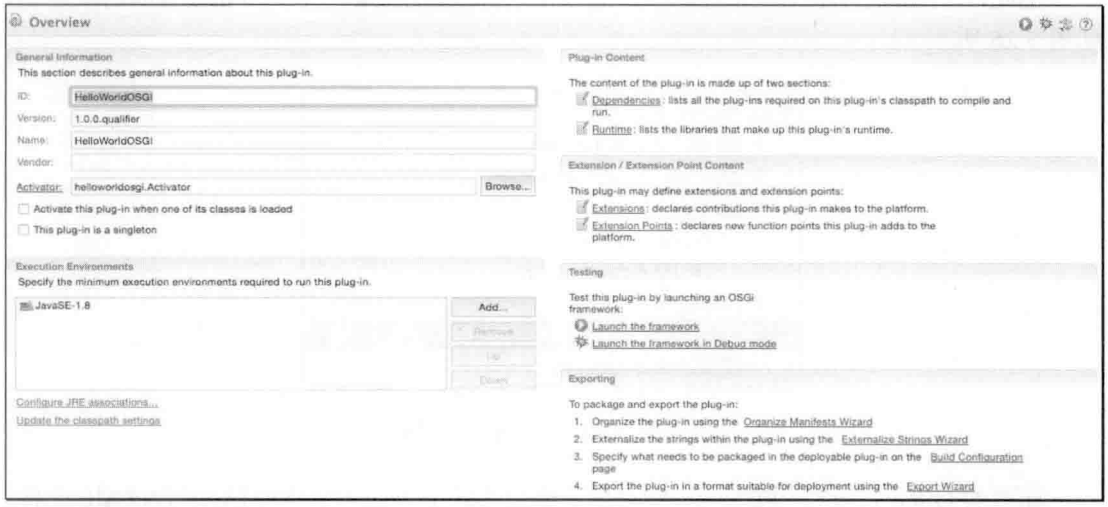


图 5.27　MANIFEST.MF 配置文件

Version 声明了这个插件包的版本信息，Name 声明了这个包在容器中展示的名称，而 Execution Environment 则声明了这个包使用什么版本的 JDK 进行编译。我们会发现 OSGi 的程序没有 Main 函数，那一个 OSGi 是怎么知道启动的程序包究竟是哪个呢？Activator 选项中就说明了究竟需要使用哪个 Java 文件作为 OSGi 项目的启动文件。

第二个选项卡是 Dependencies，这个选项卡用于配置这一个 OSGi 包需要依赖哪一些外部导入的包，如图 5.28 所示。导入的包必须是在 OSGi 容器中导出的包，否则在实际运行时会报找不到需要导入的依赖包的错误，导致 OSGi 程序无法正常进入 Active 状态。

图 5.28　OSGi 依赖包配置界面

第三个选项卡是 Runtime，这个选项卡是用于配置这一个 OSGi 包中的哪些包是导出到容器并可供其他插件包所使用的，如图 5.29 所示。这里的导入和导出我们可以这样理解，OSGi 是一个大容器，每一个插件包都可以把自己所拥有的东西导出到容器中，分享给其他需要使用的插件包。而其他插件包可以导入这些被分享到容器的资源。

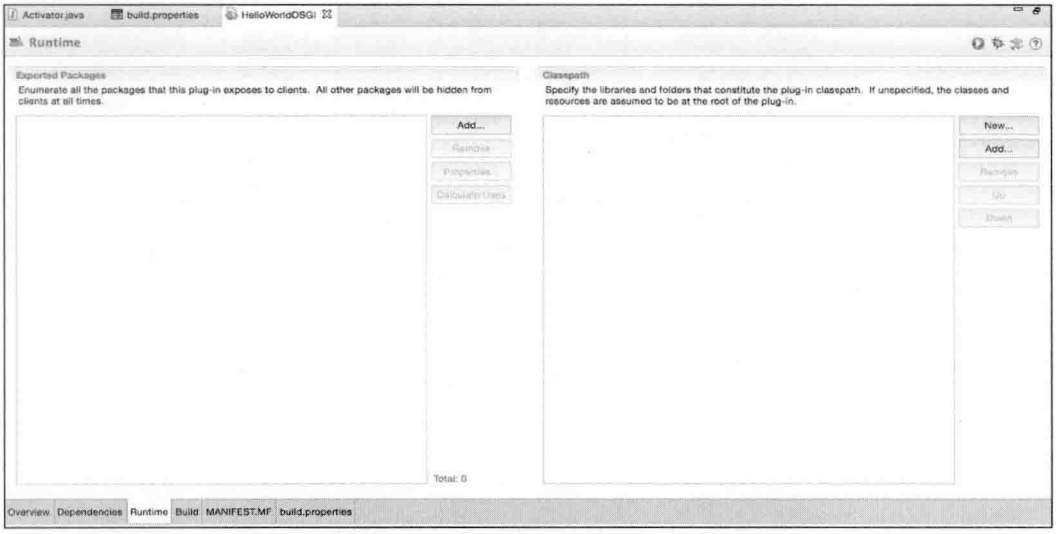


图 5.29　OSGi 导出配置界面

接下来的这个选项卡是 Build，Build 选项卡主要用于配置软件设计时需要依赖哪些包

进行编译，是一个设计时需要用到的选项卡。外部依赖包的信息在 Extra Classes Entries 里面进行配置，如图 5.30 所示。

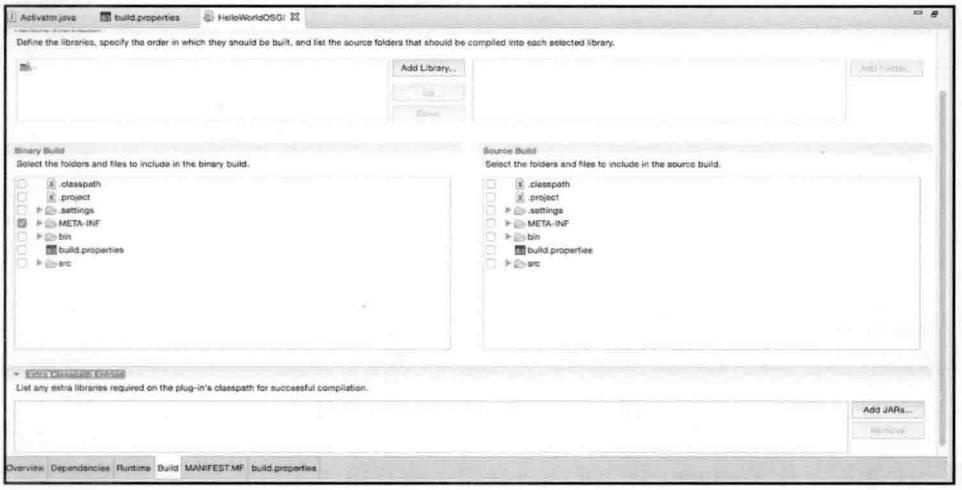


图 5.30　OSGi 构建设置

使用模板生成程序之后，我们不需要对程序做任何修改，接下来我们会完成插件包的打包部署的过程。打开项目，选择 Export，如图 5.31 所示。

图 5.31　导出 OSGi 包

选择 Deployable Plug-ins and fragments，然后设置导出插件包的路径，如图 5.32 所示。

图 5.32　设置导出选项

单击 Options 选项卡，勾选 Use class files compiled in the workspace，不然导出来的 jar 包有可能会出现中文乱码，如图 5.33 所示。

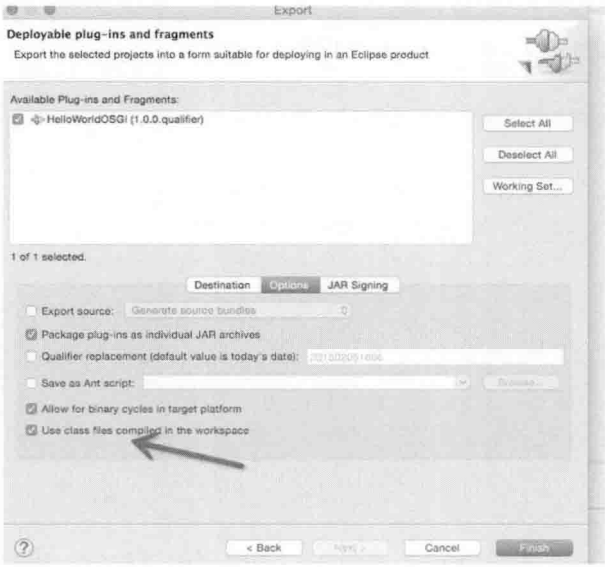


图 5.33　勾选 Use class files compiled in the workspace

导出 jar 包完毕之后，我们先用前台模式把 Karaf 启动起来，然后把这个 jar 包部署到 Karaf 的 Deploy 目录下：

```
[root@centos apache-karaf-2.3.4]# ./bin/karaf
        __ __                  ____
      / //_/___ _________ _/ __/
     / ,< / __ `/ ___/ __ `/ /_
    / /| \/ /_/ / /  / /_/ / __/
   /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (2.3.4)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'osgi:shutdown' or 'logout' to shutdown Karaf.

karaf@root> Hello World OSGI
```

可以看到插件包一放到 Karaf 里面的时候，程序就被执行了。然后我们改动一下程序的代码，代码的调整如下：

```
package helloworldosgi;

import java.util.Date;

import org.osgi.framework.BundleActivator;

import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

public void start(BundleContext context) throws Exception {
System.out.println("Hello World OSGI");
System.out.println("Current Time is" + new Date());
}

public void stop(BundleContext context) throws Exception {
System.out.println("Goodbye World OSGI");
System.out.println("Current Time is" + new Date());
}
}
```

然后重新打包，再部署到 Deploy 目录下。在部署的过程中，我们并没有重启 Karaf 容

器，而是直接把 OSGi 包替换到 Deploy 目录下，模拟了热更新的过程。

```
Apache Karaf (2.3.4)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'osgi:shutdown' or 'logout' to shutdown Karaf.

karaf@root> Hello World OSGI
Goodbye World OSGI。
Hello World OSGI
Current Time isFri Feb 06 02:29:46 CST 2015
```

我们会发现上一个程序包在热替换的过程中，会先执行 Stop 里面的方法，打印出“Goodbye World OSGI”字样，然后再重启调用了新的 OSGi 包里面的 Start 方法，完成一次热更新的过程。

第 6 章 ActiveMQ 概览

使用 ActiveMQ 的主要原因是 ActiveMQ 能够为我们降低客户端和服务器端通信开发的工作量，为了更好地使用 ActiveMQ 来完成整套运维框架的开发，我们首先对 ActiveMQ 的几个比较有用的特性进行讲解。

6.1 消息发送

JMS 定义了 5 种消息类型，分别为 TextMessage、MapMessage、BytesMessage、StreamMessage 和 ObjectMessage。ActiveMQ 在 JMS 定义的消息基础上还增加了对 BlobMessage 的支持，BlobMessage 主要用于对大型的文件进行传输，这也是为什么我们会在这么多种 MQ 产品的挑选过程中选中了 ActiveMQ 的原因。

6.1.1 TextMessage

使用 TextMessage，我们可以发送文本类型的数据。既然可以发送文本数据，那我们也可以通过采用序列化和反序列化的技术进行实体数据的发送。

1. 发送普通文本消息

```
MessageProducermessageProducer = session.createProducer(destination);
TextMessage message = session.createTextMessage();
message.setText("This is Message");
```

2. 实体序列化后发送

当我们需要对对象进行传输的时候，采用非格式化的字符串进行传输会为后续的解析带来诸多不便。这时我们可以考虑把对象进行序列化，接收端通过反序列化的方式得到实体的信息，然后再进行处理。

> 序列化是将对象的状态信息转换为可以存储或传输的形式的过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后可以通过从存储区中读取或反序列化对象的状态来重新创建该对象。

实体信息：

```
public class Entity {
    private String messageHeader;
    private String messageContent;

    public String getMessageHeader() {
        returnmessageHeader;
    }

public void setMessageHeader(String messageHeader) {
        this.messageHeader = messageHeader;
    }

public String getMessageContent() {
        returnmessageContent;
    }

public void setMessageContent(String messageContent) {
        this.messageContent = messageContent;
    }
}
```

发送实体消息：

```
TextMessage message = session.createTextMessage();
Entity entity = new Entity();
entity.setMessageContent("content");
entity.setMessageHeader("header");
```

```
Gsongson = new Gson();
String jsonString = gson.toJson(entity);
message.setText(jsonString);
messageProducer.send(message);
```

序列化后的消息如图 6.1 所示。

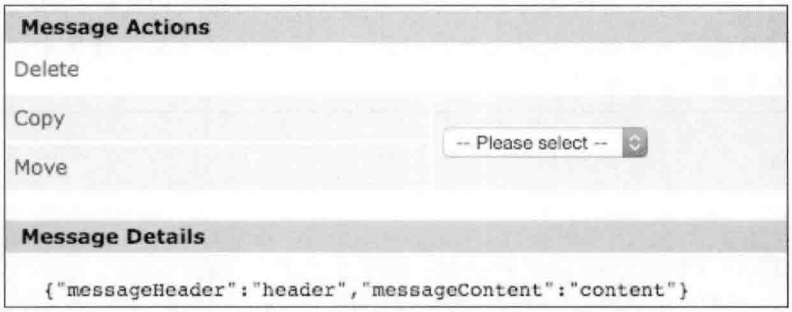


图 6.1　序列化后的消息

可以看到序列化后的消息已经被存放到了消息中间件上，接下来，我们只需要把消息从队列上收下来，再进行一次 JSON 的反序列化就可以把 JSON 变成对象，然后在客户端就可以进行操作了。

```
TextMessage message=(TextMessage) messageConsumer.receive();
Gsongson=new Gson();
String json=message.getText();
Entity entity=gson.fromJson(json,Entity.class);
```

6.1.2　MapMessage

MapMessage 可以让我们发送的消息是 HashMap 类型，它使用一张哈希表来存放其主体内容。

1. 发送普通的 MapMessage

```
MapMessage message = session.createMapMessage();
message.setObject("message", "message content");
message.setBoolean("boolenvalue", true);
messageProducer.send(message);
```

MapMessage 消息体如图 6.2 所示。

Message Details

{message=message content, boolenvalue=true}

图 6.2 MapMessage 消息体

2. 发送实体对象

既然能够在 MapMessage 里面存放 Object，那我们能不能将实体对象存放进去呢？

```
Entity entity = new Entity();
entity.setMessageContent("content");
entity.setMessageHeader("header");
MapMessage message = session.createMapMessage();
message.setObject("message", "message content");
message.setBoolean("boolenvalue", true);
message.setObject("entity", entity);
messageProducer.send(message);
```

当发送消息的时候，我们发现控制台报了这样一个错误：

```
Exception in thread "main" javax.jms.MessageFormatException: Only
objectified primitive objects,String, Map and List types are allowed but was:
Entity@50cbc42f type: class Entity
```

看来 MapMessage 只能发送 String、Map 和 List 类型，那能不能用 List 来包装着 Entity 送出去呢？伴随着这个问题，我们再次对程序进行了改造：

```
Entity entity = new Entity();
entity.setMessageContent("content");
entity.setMessageHeader("header");
List<Entity> entities=new ArrayList<Entity>();
entities.add(entity);
MapMessage message = session.createMapMessage();
message.setObject("message", "message content");
message.setBoolean("boolenvalue", true);
message.setObject("entity", entities);
messageProducer.send(message);
```

结果还是出现了如下报错信息：

```
Exception in thread "main" java.lang.RuntimeException: java.io.IOExce
```

```
ption: Object is not aprimitive: Entity@64c64813
```

看来 MapMessage 只能发送原生的对象，在序列化之后才能用 MapMessage 发送出去。最后，我们发送原生的 List 和 Map 对象，看队列中是如何展示的：

```
MapMessage message = session.createMapMessage();
message.setObject("message", "message content");
message.setBoolean("boolenvalue", true);

Map<String, String> map = new HashMap<String, String>();
map.put("key", "value");
message.setObject("map", map);

List<String> list = new ArrayList<String>();
list.add("str1");
list.add("str2");
message.setObject("list", list);
messageProducer.send(message);
```

MapMessage 与 List 消息体如图 6.3 所示。

Message Details

```
{message=message content, boolenvalue=true, map={key=value}, list=[str1, str2]}
```

图 6.3　MapMessage 与 List 消息体

6.1.3　BytesMessage

ByteMessage 可以把字节流存放在消息主体中，适合于必须压缩发送的大量数据，需要与现有消息格式保持一致等情况。

1. 发送小文件消息

```
BytesMessage message = session.createBytesMessage();

byte[] bytes = FileUtils.readFileToByteArray(new File(
        "/Users/kira/Downloads/AmazeUI-2.2.1.zip"));
message.writeBytes(bytes);
messageProducer.send(message);
```

在 ActiveMQ 管理界面上查看 ByteMessage 消息内容，如图 6.4 所示。

图 6.4　在 ActiveMQ 管理界面上查看 BytesMessage 消息内容

在 ActiveMQ 的管理页面上查看队列信息，发现队列的内容全是乱码信息，这是因为管理页面直接把二进制的内容变成 String 了，所以就出现了一堆乱码，但这仅仅是展示上不太友好，不影响我们后续的操作。

2. 接收文件

```
BytesMessage message = ((BytesMessage) messageConsumer.receive());
byte[] bs=new byte[message.readByte()];
message.readBytes(bs);
FileUtils.writeByteArrayToFile(
new File("/Users/kira/OpenSource/MyProject/ActiveMQDemo/File.zip"), bs);
```

成功接收文件后如图 6.5 所示。

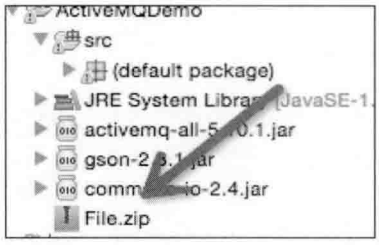


图 6.5　成功接收文件

既然 1 MB 的文件能够通过 BytesMessage 进行收/发，那么更大的文件是否也能通过这种消息进行发送呢？带着这个疑问，我们用一个 400 MB 的文件进行试验。结果出现了如下的报错信息：

```
Exception in thread "main" javax.jms.JMSException: Broken pipe
        atorg.apache.activemq.util.JMSExceptionSupport.create(JMSExceptionSu
pport.java:72)
        atorg.apache.activemq.ActiveMQConnection.syncSendPacket(ActiveMQConn
ection.java:1435)
        atorg.apache.activemq.ActiveMQConnection.syncSendPacket(ActiveMQConn
ection.java:1345)
        atorg.apache.activemq.ActiveMQSession.send(ActiveMQSession.java:1904)
        atorg.apache.activemq.ActiveMQMessageProducer.send(ActiveMQMessagePr
oducer.java:288)
        atorg.apache.activemq.ActiveMQMessageProducer.send(ActiveMQMessagePr
oducer.java:223)
        atorg.apache.activemq.ActiveMQMessageProducerSupport.send(ActiveMQMe
ssageProducerSupport.java:241)
        atProducter.main(Producter.java:64)
        Caused by: java.net.SocketException: Broken pipe
        atjava.net.SocketOutputStream.socketWrite0(Native Method)
        atjava.net.SocketOutputStream.socketWrite(SocketOutputStream.java:109)
        atjava.net.SocketOutputStream.write(SocketOutputStream.java:153)
        atorg.apache.activemq.transport.tcp.TcpBufferedOutputStream.write(Tcp
BufferedOutputStreamjava:96)
        atjava.io.DataOutputStream.write(DataOutputStream.java:107)
        atorg.apache.activemq.openwire.v10.BaseDataStreamMarshaller.tightMarshal
ByteSequence2(BaseDataStreamMarshaller.java:431)
        atorg.apache.activemq.openwire.v10.MessageMarshaller.tightMarshal2(Mes
sageMarshaller.java:182)
        atorg.apache.activemq.openwire.v10.ActiveMQMessageMarshaller.tightMa
rshal2(ActiveMQMessageMarshaller.java:89)
        atorg.apache.activemq.openwire.v10.ActiveMQBytesMessageMarshaller.tightM
arshal2(ActiveMQBytesMessageMarshaller.java:89)
        atorg.apache.activemq.openwire.OpenWireFormat.marshal(OpenWireFormat.
java:226)
```

```
        atorg.apache.activemq.transport.tcp.TcpTransport.oneway(TcpTransport.
java:175)
        atorg.apache.activemq.transport.AbstractInactivityMonitor.doOnewaySe
nd(AbstractInactivityMonitor.java:304)
        atorg.apache.activemq.transport.AbstractInactivityMonitor.oneway(Abs
tractInactivityMonitor.java:286)
        atorg.apache.activemq.transport.TransportFilter.oneway(TransportFilt
er.java:85)
        atorg.apache.activemq.transport.WireFormatNegotiator.oneway(WireForma
tNegotiator.java:104)
        atorg.apache.activemq.transport.MutexTransport.oneway(MutexTransport
.java:68)
        atorg.apache.activemq.transport.ResponseCorrelator.asyncRequest(Resp
onseCorrelator.java:81)
        atorg.apache.activemq.transport.ResponseCorrelator.request(ResponseC
orrelator.java:86)
        atorg.apache.activemq.ActiveMQConnection.syncSendPacket(ActiveMQConn
ection.java:1406)
        ... 6 more
```

ActiveMQ 的客户端出现了 BrokenPipe 的错误，看来大文件是不能直接整个送出去的，需要切分成一个个的小文件进行发送，如图 6.6 所示。

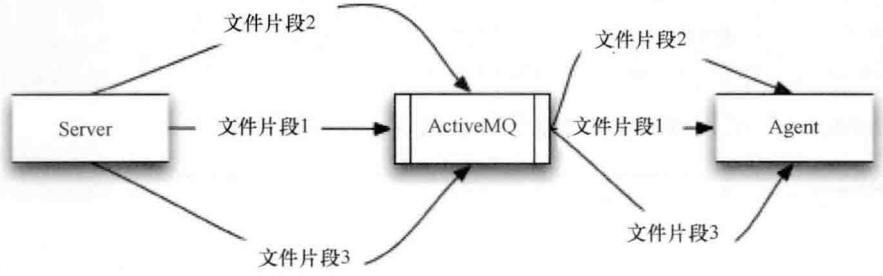


图 6.6　切割大文件进行文件传输

但是一旦切开的动作需要开发人员自己做，那么整个文件传输过程的工作量就会变大，我们需要为文件片段的顺序做处理，还需要考虑在发送文件片段时是否会出现其他文件片段的情

况。假设会出现，我们还需要在客户端对这种不同文件的文件片段进行处理，如图 6.7 所示。

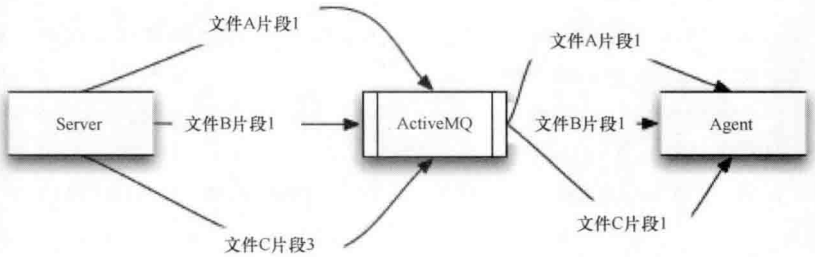


图 6.7　多文件片段干扰

所以为了降低开发工作量，在传输大文件时我们还是使用另外一种叫作 BlobMessage 的消息进行发送。

6.1.4　StreamMessage

StreamMessage 从字面上看是用于传输流信息的，但是实际上它是以队列这种先进先出的模式进行信息传递的。例如，在发送的时候，StreamMessage 的消息顺序为 A→B，那么接收的时候，客户端读出消息的顺序也为 A→B。它也支持属性字段和 MapMessage 所支持的数据类型。所以使用这种消息格式时，收/发双方需要事先协商好字段的顺序，以保证写/读顺序相同。

```
StreamMessage message=session.createStreamMessage();
message.writeBoolean(true);
message.writeString("message content");
messageProducer.send(message);
```

StreamMessage 在 ActiveMQ 管理界面上的消息内容如图 6.8 所示。

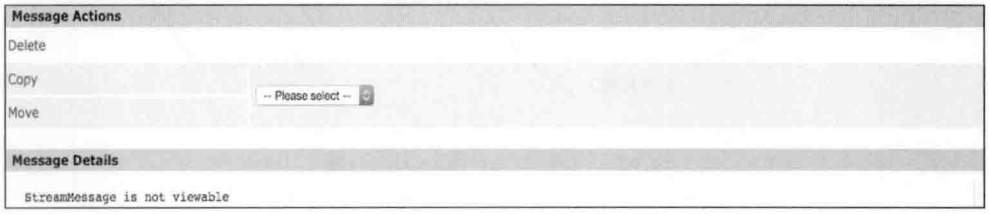


图 6.8　StreamMessage 在 ActiveMQ 管理界面上的消息内容

由于生产者发送消息的顺序是先 int 类型后 string 类型，消费者需要按照这个顺序进行读取。

```
StreamMessage message=session.createStreamMessage();
message.writeBoolean(true);
message.writeString("message content");
```

消费者可以正常读取消息的内容，假如我们把程序调整为如下的结构：

```
StreamMessage message = ((StreamMessage) messageConsumer.receive());
System.out.println(message.readString());
System.out.println(message.readBoolean());
```

也就是先读取 String 类型，再读取布尔类型，读取的结果如下：

```
true
false
```

由于第一个消息的内容为 true，readString 方法把 true 转换成 String 类型了，所以就输出了 true，message content 由于无法转换为布尔类型，所以就返回了 false。

再看看 StreamMessage 能否发送大文件：

```
byte[] bytes = FileUtils.readFileToByteArray(new File(
        "/Users/kira/Documents/大文件.mp4"));

StreamMessage message=session.createStreamMessage();
message.writeBoolean(true);
message.writeString("message content");
message.writeBytes(bytes);
messageProducer.send(message);
```

会得到和 BytesMessage 发送文件时一样的异常。

6.1.5 BlobMessage

ActiveMQ 在 JMS 的协议之上做了一些扩展，BlobMessage 就是 ActiveMQ，使用 BlobMessage 能让开发人员花费最少的精力来完成大文件的传输。BlobMessage 的实现原理如图 6.9 所示。

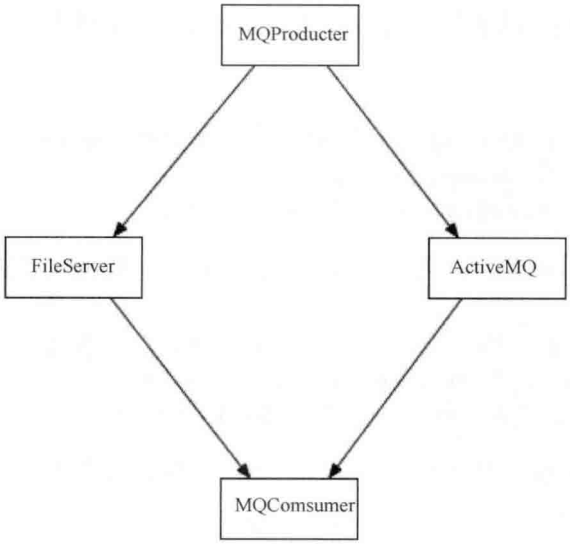


图 6.9　BlogMessage 消息传输图

首先，生产者会把文件传输到 FileServer 上，FileServer 是一个内嵌在 ActiveMQ 内的应用，部署在内嵌的 Jetty 里面。FileServer 的代码非常简单，我们可以快速地浏览一下文件是如何上传到服务器的。

1. FilenameGuardFilter

```
packageorg.apache.activemq.util;

importjava.io.IOException;
importjavax.servlet.Filter;
importjavax.servlet.FilterChain;
importjavax.servlet.FilterConfig;
importjavax.servlet.ServletException;
importjavax.servlet.ServletRequest;
importjavax.servlet.ServletResponse;
importjavax.servlet.http.HttpServletRequest;
importjavax.servlet.http.HttpServletRequestWrapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class FilenameGuardFilter
```

```
        implements Filter
        {
        private static final Logger LOG =
        LoggerFactory.getLogger(FilenameGuardFilter.class);

        public void destroy()
          {
          }

        public void init(FilterConfigconfig) throws ServletException
          {
          }

        public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws
        IOException, ServletException {
        if ((request instanceofHttpServletRequest)) {
        HttpServletRequesthttpRequest = (HttpServletRequest)request;
            GuardedHttpServletRequestguardedRequest =
            newGuardedHttpServletRequest(httpRequest);
        chain.doFilter(guardedRequest, response);
           } else {
        chain.doFilter(request, response);
           }
          }

        private    static    class    GuardedHttpServletRequest    extends
HttpServletRequestWrapper
          {
        publicGuardedHttpServletRequest(HttpServletRequesthttpRequest) {
        super();
           }

        private String guard(String filename) {
             String guarded = filename.replace(":", "_");
        if (FilenameGuardFilter.LOG.isDebugEnabled()) {
        FilenameGuardFilter.LOG.debug("guarded " + filename + " to " + guarded);
             }
        return guarded;
           }
```

```
    public String getParameter(String name)
        {
    if (name.equals("Destination")) {
    return guard(super.getParameter(name));
          }
    returnsuper.getParameter(name);
        }

    public String getPathInfo()
        {
    return guard(super.getPathInfo());
        }

    public String getPathTranslated()
        {
    return guard(super.getPathTranslated());
        }

    public String getRequestURI()
        {
    return guard(super.getRequestURI());
        }
      }
    }
```

FilenameGuardFilter 是一个过滤器，主要用于重命名上传的文件，因为上传上来的文件名称有可能会重复，所以在保存之前，FilenameGuardFilter 这个过滤器会先把文件重新命名。

2. IOHelper

```
    packageorg.apache.activemq.util;

    importjava.io.File;
    importjava.io.FileInputStream;
    importjava.io.FileOutputStream;
    importjava.io.IOException;
    importjava.io.InputStream;
    importjava.io.OutputStream;
```

```
public final class IOHelper
{
protected static final int MAX_DIR_NAME_LENGTH =
Integer.valueOf(System.getProperty("MaximumDirNameLength",
"200")).intValue();
protected static final int MAX_FILE_NAME_LENGTH =
Integer.valueOf(System.getProperty("MaximumFileNameLength",
"64")).intValue();
private static final int DEFAULT_BUFFER_SIZE = 4096;

public static String getDefaultDataDirectory()
  {
returngetDefaultDirectoryPrefix() + "activemq-data";
  }

public static String getDefaultStoreDirectory() {
returngetDefaultDirectoryPrefix() + "amqstore";
  }

public static String getDefaultDirectoryPrefix()
  {
try
    {
returnSystem.getProperty("org.apache.activemq.default.directory.pref
ix", ""); } catch (Exception e) {
    }
return "";
  }

public static booleandeleteFile(File fileToDelete)
  {
if ((fileToDelete == null) || (!fileToDelete.exists())) {
return true;
    }
boolean result = deleteChildren(fileToDelete);
result&= fileToDelete.delete();
return result;
  }

public static booleandeleteChildren(File parent) {
```

```
        if ((parent == null) || (!parent.exists())) {
        return false;
            }
        boolean result = true;
        if (parent.isDirectory()) {
        File[] files = parent.listFiles();
        if (files == null)
        result = false;
        else {
        for (inti = 0; i<files.length; i++) {
                File file = files[i];
        if ((!file.getName().equals(".")) && (!file.getName().equals("..")))
                {
        if (file.isDirectory())
        result&= deleteFile(file);
        else {
        result&= file.delete();
                  }
                }
              }
            }
          }
        return result;
          }

        public static void moveFile(File src, File targetDirectory) throws
IOException
          {
        if (!src.renameTo(new File(targetDirectory, src.getName())))
        throw new IOException("Failed to move " + src + " to " + targetDirectory);
          }

        public static void copyFile(File src, File dest) throws IOException
          {
        FileInputStreamfileSrc = new FileInputStream(src);
        FileOutputStreamfileDest = new FileOutputStream(dest);
        copyInputStream(fileSrc, fileDest);
          }

        public static void copyInputStream(InputStream in, OutputStream out)
```

```
throws IOException {
        byte[] buffer = new byte[4096];
        intlen = in.read(buffer);
        while (len>= 0) {
        out.write(buffer, 0, len);
        len = in.read(buffer);
            }
        in.close();
        out.close();
          }

        public static void mkdirs(File dir)
        throwsIOException
          {
        if (dir.exists()) {
        if (!dir.isDirectory()) {
        throw new IOException("Failed to create directory '" + dir + "', regular file
already existed with that name");
              }

            }
        else if (!dir.mkdirs())
        throw new IOException("Failed to create directory '" + dir + "'");
          }
        }
```

第二个类是 IOHelper，这是一个工具类，主要供上层类进行调用，进行文件夹的创建、文件的移动以及文件的写入等操作。很奇怪为什么 ActiveMQ 的 FileServer 应用不使用 Apache Common IO，这个 Apache 包封装了不少 I/O 操作的工具类，可以大大简化 IOHelper 的代码量。

3. RestFilter

```
        packageorg.apache.activemq.util;

        importjava.io.File;
        importjava.io.FileOutputStream;
        importjava.io.IOException;
        import java.net.URL;
        importjavax.servlet.Filter;
        importjavax.servlet.FilterChain;
        importjavax.servlet.FilterConfig;
```

```
importjavax.servlet.ServletContext;
importjavax.servlet.ServletException;
importjavax.servlet.ServletRequest;
importjavax.servlet.ServletResponse;
importjavax.servlet.UnavailableException;
importjavax.servlet.http.HttpServletRequest;
importjavax.servlet.http.HttpServletResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class RestFilter
implements Filter
{
 private static final Logger LOG = LoggerFactory.getLogger(RestFilter.
class);
private static final String HTTP_HEADER_DESTINATION = "Destination";
private static final String HTTP_METHOD_MOVE = "MOVE";
private static final String HTTP_METHOD_PUT = "PUT";
private static final String HTTP_METHOD_GET = "GET";
private static final String HTTP_METHOD_DELETE = "DELETE";
private String readPermissionRole;
private String writePermissionRole;
privateFilterConfigfilterConfig;

public void init(FilterConfigfilterConfig)
throwsUnavailableException
  {
    this.filterConfig = filterConfig;
    this.readPermissionRole  =  filterConfig.getInitParameter("read-
permission-role");
    this.writePermissionRole = filterConfig.getInitParameter("write-
permission-role");
  }

private File locateFile(HttpServletRequest request) {
return new

File(this.filterConfig.getServletContext().getRealPath(request.getServletPat
h()),
request.getPathInfo());
  }

public void doFilter(ServletRequest request, ServletResponse response,
```

```
FilterChain chain) throws
        IOException, ServletException {
        if   ((!(request   instanceofHttpServletRequest))   ||   (!(response
instanceofHttpServletResponse))) {
                if (LOG.isDebugEnabled()) {
                    LOG.debug("request not HTTP, can not understand: " +
request.toString());
                }
            chain.doFilter(request, response);
            return;
            }

            HttpServletRequesthttpRequest = (HttpServletRequest)request;
            HttpServletResponsehttpResponse = (HttpServletResponse)response;

        if (httpRequest.getMethod().equals("MOVE"))
            doMove(httpRequest, httpResponse);
        else if (httpRequest.getMethod().equals("PUT"))
            doPut(httpRequest, httpResponse);
        else if (httpRequest.getMethod().equals("GET")) {
            if (checkGet(httpRequest, httpResponse)) {
                chain.doFilter(httpRequest, httpResponse);
            }
            }
        else if (httpRequest.getMethod().equals("DELETE"))
            doDelete(httpRequest, httpResponse);
        else
            chain.doFilter(httpRequest, httpResponse);
          }

        protected void doMove(HttpServletRequest request, HttpServletResponse
response) throws
        ServletException, IOException
          {
            if (LOG.isDebugEnabled()) {
   LOG.debug("RESTful file access: MOVE request for " + request.getRequestURI());
          }

   if ((this.writePermissionRole != null) && (!request.isUserInRole(this.write
PermissionRole))) {
            response.sendError(403);
            return;
```

```
            }

            File file = locateFile(request);
            String destination = request.getHeader("Destination");

        if (destination == null) {
            response.sendError(400, "Destination header not found");
            return;
            }
        try
            {
                URL destinationUrl = new URL(destination);
                IOHelper.copyFile(file, new File(destinationUrl.getFile()));
                IOHelper.deleteFile(file);
            } catch (IOException e) {
                response.sendError(500);
                return;
            }

                response.setStatus(204);
          }

        protectedbooleancheckGet(HttpServletRequest                request,
HttpServletResponse response) throws ServletException, IOException
          {
            if (LOG.isDebugEnabled()) {
                LOG.debug("RESTful file access: GET request for " +
request.getRequestURI());
            }

   if ((this.readPermissionRole != null) && (!request.isUserInRole(this.read
PermissionRole))) {
                response.sendError(403);
                return false;
            }
            return true;
          }

        protected void doPut(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
          {
            if (LOG.isDebugEnabled()) {
                LOG.debug("RESTful file access: PUT request for " +
```

```
request.getRequestURI());
            }

            if ((this.writePermissionRole != null)
            && (!request.isUserInRole(this.writePermissionRole))) {
                response.sendError(403);
                return;
            }

            File file = locateFile(request);

        if (file.exists()) {
            boolean success = file.delete();
        if (!success) {
            response.sendError(500);

            return;
              }
            }

        FileOutputStream out = new FileOutputStream(file);
        try {
                IOHelper.copyInputStream(request.getInputStream(), out);
            } catch (IOException e) {
                LOG.warn("Exception occured", e);
                throw e;
            } finally {
                out.close();
            }

                response.setStatus(204);
          }

    protected void doDelete(HttpServletRequest request, HttpServletResponse
response) throws
        ServletException, IOException
          {
                if (LOG.isDebugEnabled()) {
                LOG.debug("RESTful file access: DELETE request for " +
                request.getRequestURI());
            }

            if ((this.writePermissionRole != null)
```

```
        && (!request.isUserInRole(this.writePermissionRole))) {
                response.sendError(403);
                return;
        }

        File file = locateFile(request);

    if (!file.exists()) {
    response.sendError(404);

    return;
        }

    boolean success = IOHelper.deleteFile(file);

    if (success) {
    response.setStatus(204);
        }
    else
        {
    response.sendError(500);
        }
      }

    public void destroy()
      {
      }
    }
```

最后一个是 RestFilter，RestFilter 主要用于相应客户端的请求，它采用 RestFul 风格的 API 进行设计，而客户端在上传文件时所调用的就是 Put 的 API。RestFilter 接收到了 Put 请求之后，会把文件从流写入磁盘，完成文件上传的过程。

当文件上传完毕之后，MQ 的生产者会再发送一条普通消息到队列中，MQ 的消费者发现队列上有消息后，就会把消息收下来，并且会采用 Get 的方法把 MQ 中的文件也一起收取下来。

4. 文件消息发送

```
    BlobMessage message = session.createBlobMessage(new File(
                "/User/kira/file.txt"));
    messageProducer.send(message);
```

发送文件消息时最需要注意的地方是创建连接，FileServer 的 url 一定要设置正确，例如，下面这个 url 的意思是将消息的内容发送到 127.0.0.1 这台主机上，文件上传到 127.0.0.1 这个 IP 所在的主机上。

```
String url = "tcp://127.0.0.1:61616?"
            + "jms.blobTransferPolicy.defaultUploadUrl="
            + "http://127.0.0.1:8161/fileserver/";
    ActiveMQConnectionFactoryconnectionFactory =
newActiveMQConnectionFactory(
            "admin", "admin", url);
```

传输 BlobMessage 如图 6.10 所示。

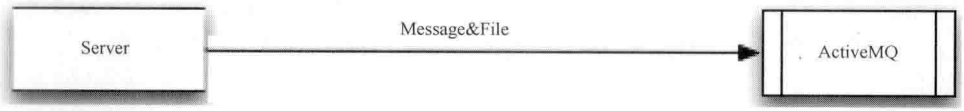


图 6.10　传输 BlobMessage

但是，这两个 IP 其实可以设置成不一样，如图 6.11 所示。

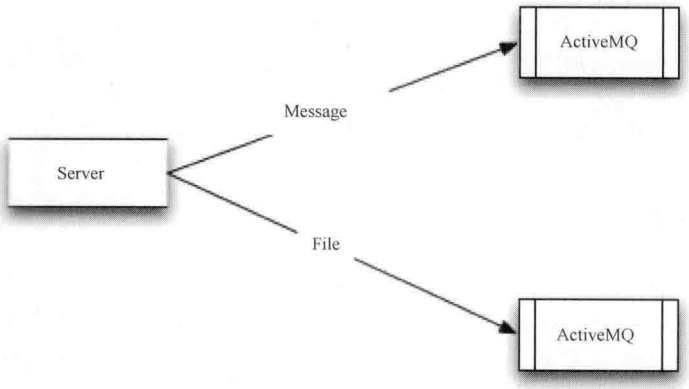


图 6.11　消息发送的 IP 与文件发送的 IP 不一致的情况

```
String url = "tcp://127.0.0.1:61616?"
            + "jms.blobTransferPolicy.defaultUploadUrl="
            + "http://192.168.41.2:8161/fileserver/";
    ActiveMQConnectionFactoryconnectionFactory =
newActiveMQConnectionFactory(
```

```
                "admin", "admin", url);
```

上面的这个连接设置的是将消息内容发送到 127.0.0.1 这个 IP 所在的主机上，文件则是上传到 192.168.41.2 这个 IP 所在的主机上。这个地方的设置是需要注意的，一旦客户端和服务器端的 ActiveMQUrl 配置不正确，就会出现消息收下来了，但是文件收取不下来的情况。

6.2 断线重连机制 FailOver

6.2.1 配置 FailOver

由于消费者是采用订阅者的模式进行设置的，也就是我们的消费者会一直侦听 MQ 上的队列，一旦有消息，就会触发函数里面的方法，在未收到消息时则会阻塞。

```
MessageConsumermessageConsumer = session.createConsumer(destination);
    messageConsumer.setMessageListener(new MessageListener() {

        @Override
        public void onMessage(Message message) {
            try {
                BlobMessageblobMessage = (BlobMessage) message;
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
```

上面是一个比较典型的例子，但是由于网络环境的不稳定，又或者是中间件由于故障而停机，消费者的连接会由于种种情况而断开连接，但是作为一个 Agent 端，它应该能够在故障恢复之后自动进行重连，否则会出现大规模的客户端不可用而需要重启的情况。

这个时候，我们就需要使用 ActiveMQ 的 FailOver 机制了。

开启 ActiveMQ 会话的 FailOver 机制非常简单，只需要在连接上进行简单的配置就可以了，我们把上面的 url 连接进行一些调整。

调整前：

```
String url = "tcp://127.0.0.1:61616?"
            + "jms.blobTransferPolicy.defaultUploadUrl="
            + "http://127.0.0.1:8161/fileserver/";
```

调整后：

```
String url = "failover:(tcp://127.0.0.1:61616)?"
            + "jms.blobTransferPolicy.defaultUploadUrl="
            + "http://127.0.0.1:8161/fileserver/";
```

经过简单的 url 调整，ActiveMQ 的会话已经具备了断线重连的能力了。

我们使用调整前的 url 进行 ActiveMQ 的连接，当消费者与 MQ 的连接建立好了之后，我们模拟 ActiveMQ 出现了故障，直接把 ActiveMQ 的进程杀死，消费者这边就开始不断报下面的错误了：

```
javax.jms.IllegalStateException: The Consumer is closed
    atorg.apache.activemq.ActiveMQMessageConsumer.checkClosed(ActiveMQMess
ageConsumer.java:861)
    atorg.apache.activemq.ActiveMQMessageConsumer.receive(ActiveMQMessag
eConsumer.java:552)
    atComsumer.main(Comsumer.java:37)
```

然后再次启动 ActiveMQ，它依旧没有重新连接上 MQ 服务器。

但是使用调整后的 url 进行 ActiveMQ 连接，当我们把 MQ 服务器杀死之后，程序并没有出现错误信息，只是消费者无法继续消费队列上的信息了。我们重新把 ActiveMQ 启动起来，程序又会继续对队列的内容进行消费了。

6.2.2 FailOver 的常用参数

FailOver 常用配置参数及说明如表 6.1 所示。

表 6.1 FailOver 常用配置参数及说明

参 数	默 认 值	描 述
initialReconnectDelay	10	第一次连接失败到第二次连接直接的时间间隔（毫秒）
maxReconnectDelay	30000	每次重连的最大时间间隔（毫秒）
useExponentialBackOff	true	每次重连的间隔时间是否越来越短

（续表）

参　数	默 认 值	描　述
reconnectDelayExponent	2.0	重连加速比
maxReconnectAttempts	−1 | 0	最大的重连次数。从 5.6 版本开始，−1 代表无限次的重连。而 5.6 版本之前，0 代表无限次的重连
randomize	true	是否从 url 列表里面随机地选择连接的服务器
backup	false	是否启动快速重连的机制
timeout	−1	为发送的操作设置超时的时间，而不打断重连进程的重连操作

6.3 消息生命周期

6.3.1 为什么消息需要生命周期

一旦消费者出现了故障，消息就会一直留在队列中，然后等消息启动之后，消息就会被全量接收。对于某些消息来说，这种做法是合适的，但是对于某些操作，这种做法就不太合适了。一个比较典型的场景是服务器端会每天晚上发送一个消息，通知客户端进行常规的巡检，但是客户端由于挂死了而无法进行消息的收取，假如消息是没有生命周期的，那么一旦客户端启动起来之后，就会不断地收取队列里面的消息。

在 Agent 端运转正常的情况下，数据的流向如图 6.12 所示。

图 6.12　正常情况下的巡检过程

但是一旦 Agent 端出现异常，命令的消息就会堵塞在消息队列里面，产生消息队列的消息堆积，如图 6.13 所示。

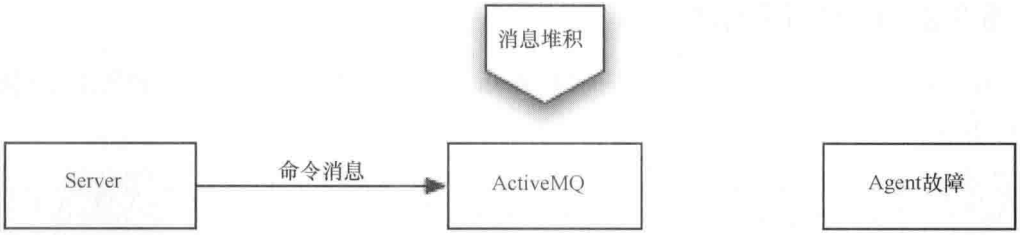


图 6.13　Agent 出现故障后消息堆积

等到 Agent 端从故障中恢复过来时，就会从消息队列把所有的消息都收下来，并且不断地执行，如图 6.14 所示。

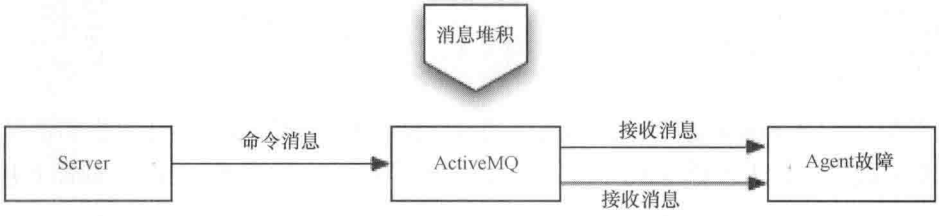


图 6.14　Agent 恢复正常后，出现不正常的巡检操作

在这种情况下，一旦客户端启动起来之后，就会不断地执行巡检，造成主机的 CPU 使用率异常。在这种消息具有时效性的情况下，我们需要让在队列里面的消息具备生命周期。例如触发巡检的消息，假如在队列里面放了 30 分钟依然没有客户端接收，那么这条消息就需要被自动消除，避免客户端连续接收过期的消息，导致运行异常，如图 6.15 所示。

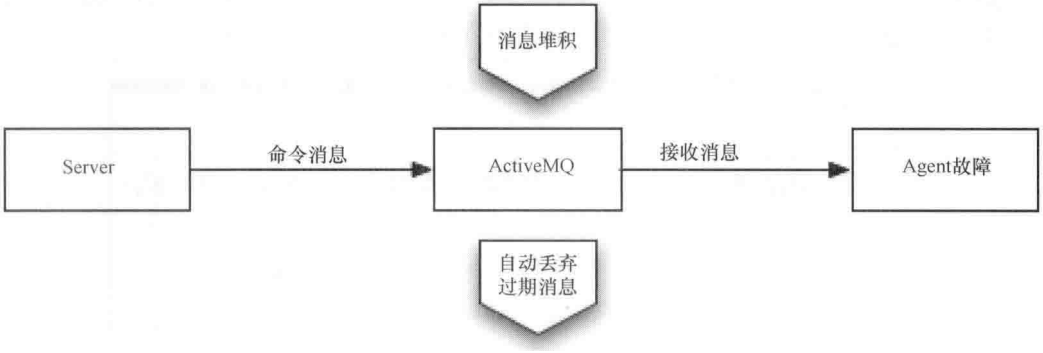


图 6.15　使用自动丢弃过期消息的功能以避免消息堆积

6.3.2 使用消息超时机制

为了应对消息是有时效性的这个问题，ActiveMQ 已经为我们准备好了设置消息超时的 API，使用方式如下：

```
Entity entity = new Entity();
entity.setMessageContent("content");
entity.setMessageHeader("header");
List<Entity> entities = new ArrayList<Entity>();
entities.add(entity);
TextMessage message = session.createTextMessage();
Gsongson = new Gson();
String json = gson.toJson(entities);
message.setText(json);
messageProducer.setTimeToLive(5000);
messageProducer.send(message);
```

关键的代码是 messageProducer.setTimeToLive(5000)，它设置了发送出去的消息是有生命周期的，假如超过了 5 秒依旧没被接收，那么这条消息应该需要被销毁，如图 6.16 所示。

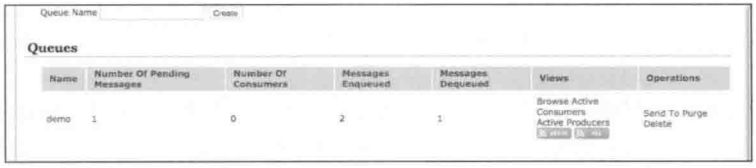


图 6.16　发送超时时间为 5 秒的消息

我们先发送一条消息，然后打开 ActiveMQ 的管理界面，可以看到队列里面有一条消息，但是没有消费者收取队列里面的内容，如图 6.17 所示。

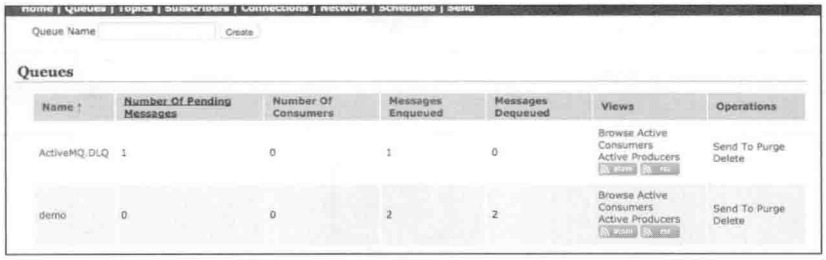


图 6.17　消息自动超时

隔了 5 秒之后，我们会发现 demo 队列里面的队列数已经变为 0 了，但是多了一个叫作

ActiveMQ.DLQ 的队列。打开 ActiveMQ.DLQ 队列之后，我们可以看到如图 6.18 所示的界面。

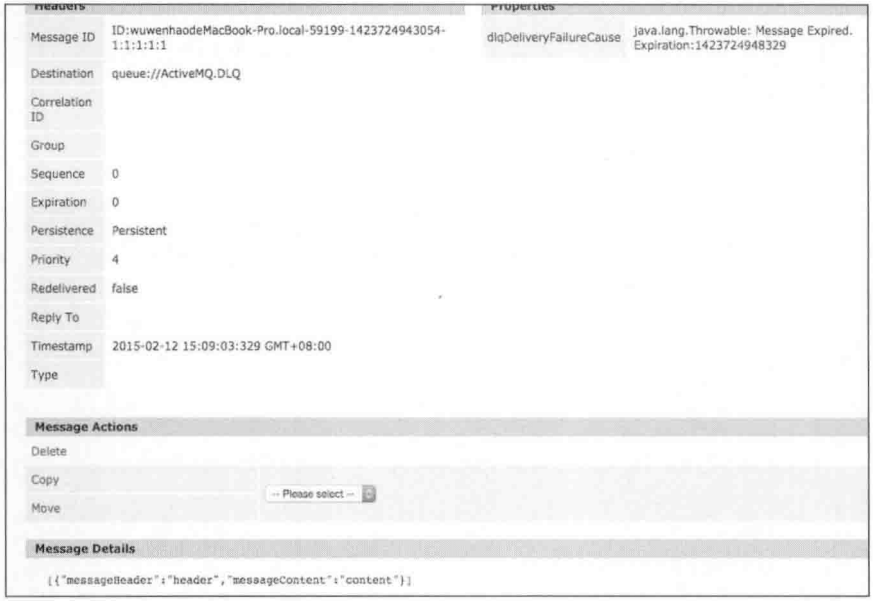


图 6.18　ActiveMQ.DLQ 队列

这个内容是我们刚才发送出去的，但是没有被收取的那条消息。这说明 ActiveMQ 已经把超时没有客户端接收的队列移走了。使用这个特性，我们就可以避免消息堆积且没有客户端接收的情况下，造成了大量队列阻塞的问题。

6.4　清空不常用的队列

我们在架构设计时，也应设计每个客户端在队列上注册队列的规则。举个例子，客户端的 IP 为 192.168.1.44 ，这个客户端上部署了 3 个不同的功能插件，那么这个队列上会注册 3 个队列，队列的名字分别为：

- A_192.168.1.44；
- B_192.168.1.44；
- C_192.168.1.44。

上面就是按照功能名称_主机 IP 这样的规则进行队列的注册，但是当客户端的数量增

加的时候，会出现这样一个问题，那就是队列上出现了太多的队列，给我们在使用 ActiveMQ 的管理界面进行管理时带来了不便。一方面是无法一眼就看到我们需要查找的队列的使用情况，另一方面是由于队列的注册量太多，导致打开 ActiveMQ 的管理页面的速度非常慢，这时我们就可以配置 ActiveMQ，自动清理一段时间没有使用的队列。

修改配置文件 activemq.xml：

```
            <broker
            mlns="http://activemq.apache.org/schema/core"
schedulePeriodForDestinationPurge="10000">
            destinationPolicy>
        <policyMap>
        <policyEntries>
            <policyEntry queue=">"
            gcInactiveDestinations="true" inactiveTimoutBeforeGC="30000"/>
        </policyEntries>
        </policyMap>
        </destinationPolicy>
        </broker>
```

- schedulePeriodForDestinationPurge=10000：每 10 秒检查一次，默认为 0，此功能关闭；
- gcInactiveDestinations= true：删除不活动队列，默认为 false；
- inactiveTimoutBeforeGC=30000：不活动则 30 秒后删除，默认为 60 秒。

等待 30 秒之后，ActiveMQ 就会把 30 秒内没有消息传输并且待收消息为 0 的队列清空。这样就可以减少非常多不活动的队列，一方面可以降低 ActiveMQ 队列的内存使用情况，另一方面，也非常便于我们在 ActiveMQ 的管理页面上进行故障的排查。

6.5 使用 JMX 获取队列信息

JMX（Java Management Extensions，Java 管理扩展）是一个为应用程序、设备、系统等植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活地开发无缝集成的系统、网络和服务管理应用。

通过使用 JMX 协议，我们可以获取 ActiveMQ 的许多信息，其中对开发过程比较有用的信息是队列的信息，包括在 ActiveMQ 上注册的所有队列的名称、队列的生产者消费者

数量等，通过收集这些信息，可以为后续服务端作业调度提供非常有用的信息。

6.5.1 启用 ActiveMQ 的 JMX 功能

使用 JMX 获取 ActiveMQ 的信息之前，我们需要对 ActiveMQ 进行配置，让 ActiveMQ 打开 JMX 的支持。

此时需要修改 ActiveMQ.xml 的两处地方，一个是在 broker 位置加上 useJmx="true"，第二个需要修改的地方是 managementContext 节点的 createConnector，这个属性默认是 false，修改为 true 即可。修改后配置如下，然后重启 ActiveMQ。

```
        <broker
        useJmx="true"xmlns="http://activemq.apache.org/schema/core"
brokerName="localhost"
        dataDirectory="${activemq.data}">

        <destinationPolicy>
        <policyMap>
        <policyEntries>
        <policyEntry topic=">"
        gcInactiveDestinations="true" inactiveTimoutBeforeGC="30000">
        <pendingMessageLimitStrategy>
        <constantPendingMessageLimitStrategy limit="1000"/>
        </pendingMessageLimitStrategy>
        </policyEntry>
        </policyEntries>
        </policyMap>
        </destinationPolicy>

        <managementContext>
        <managementContext createConnector="true"/>
        </managementContext>
```

重启完成之后，我们先使用 JConsole 进行实验，看是否真的把 ActiveMQ 的 JMX 协议打开了，在控制台输入 jconsole，如图 6.19 所示。

```
kira — bash — 80×24
wuwenhaodeMacBook-Pro:~ kira$ jconsole
```

图 6.19 使用命令打开 JConsole

使用远程进程进行连接，如图 6.20 所示。

图 6.20　连接到 ActiveMQ

连接成功后会进入到监控页面，如图 6.21 所示。

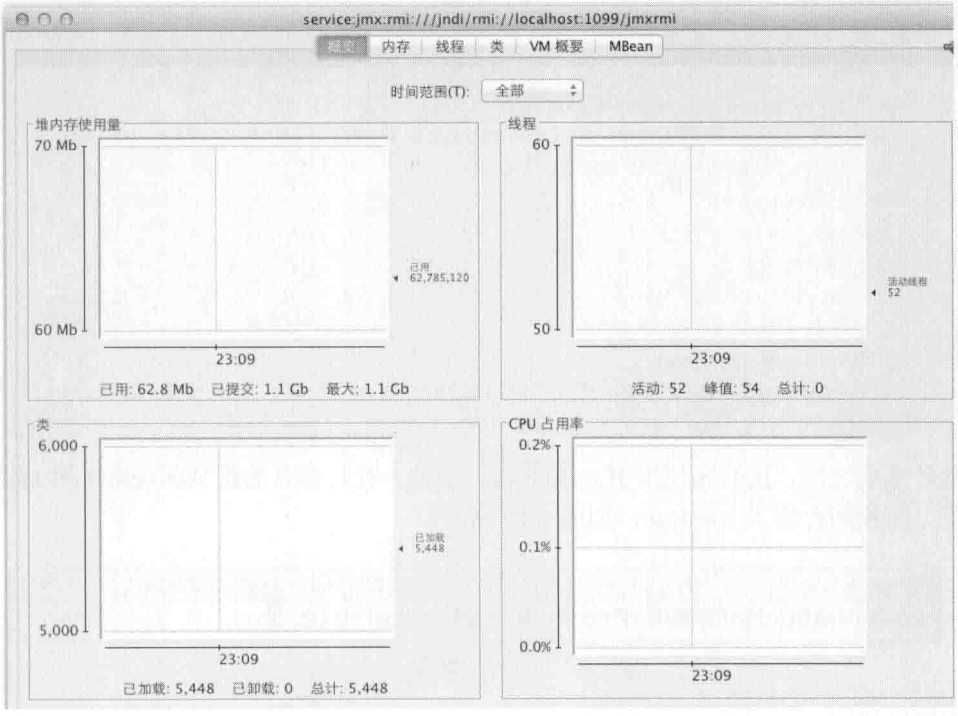


图 6.21　成功连接到 ActiveMQ

单击 MBean 的选项，可以看到展示页面的左侧有许多可以选择的项，这些项就是我们所说的 MBean，如图 6.22 所示。

图 6.22　查看 MBean

打开后我们可以看到每个 Bean 的属性以及相对应的 ObjectName，后续在 Java 程序当中会使用 ObjectName 去获取这个对象的值，如图 6.23 所示。

图 6.23　查看 MBean 详细信息

6.5.2　获取 ActiveMQ 的队列信息

获取 ActiveMQ 的 mbean 属性及操作：

```
importjava.io.IOException;
importjava.net.MalformedURLException;
```

```
importjava.util.Iterator;
importjava.util.Set;

importjavax.management.InstanceNotFoundException;
importjavax.management.IntrospectionException;
importjavax.management.MBeanAttributeInfo;
importjavax.management.MBeanInfo;
importjavax.management.MBeanOperationInfo;
importjavax.management.MBeanServerConnection;
importjavax.management.MBeanServerInvocationHandler;
importjavax.management.MalformedObjectNameException;
importjavax.management.ObjectInstance;
importjavax.management.ObjectName;
importjavax.management.ReflectionException;
importjavax.management.remote.JMXConnector;
importjavax.management.remote.JMXConnectorFactory;
importjavax.management.remote.JMXServiceURL;
importorg.apache.activemq.broker.jmx.BrokerViewMBean;
importorg.apache.activemq.broker.jmx.QueueViewMBean;

public class JMXGetter {

    public static void main(String[] args) throws IOException,
            MalformedObjectNameException, InstanceNotFoundException,
            IntrospectionException, ReflectionException {

        String                          surl                          =
"service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi";
        JMXServiceURLurl = new JMXServiceURL(surl);
        JMXConnectorjmxc = JMXConnectorFactory.connect(url, null);
        MBeanServerConnectionmbsc = jmxc.getMBeanServerConnection();
        ObjectNamembeanName = new ObjectName(

  "org.apache.activemq:type=Broker,brokerName=localhost");
        MBeanInfo info = mbsc.getMBeanInfo(mbeanName);
        if (info.getAttributes().length > 0) {
            for (MBeanAttributeInfo m : info.getAttributes()) {
                System.out.println(m.getName());
            }
        }
        if (info.getOperations().length > 0) {
            for (MBeanOperationInfo m : info.getOperations()) {
                System.out.println(m.getName());
```

```
                }
            }

            jmxc.close();
        }
    }
```

获取 ActiceMQ 的队列信息：

```
    importjava.io.IOException;
    importjava.net.MalformedURLException;
    importjava.util.Iterator;
    importjava.util.Set;
    importjavax.management.InstanceNotFoundException;
    importjavax.management.IntrospectionException;
    importjavax.management.MBeanAttributeInfo;
    importjavax.management.MBeanInfo;
    importjavax.management.MBeanOperationInfo;
    importjavax.management.MBeanServerConnection;
    importjavax.management.MBeanServerInvocationHandler;
    importjavax.management.MalformedObjectNameException;
    importjavax.management.ObjectInstance;
    importjavax.management.ObjectName;
    importjavax.management.ReflectionException;
    importjavax.management.remote.JMXConnector;
    importjavax.management.remote.JMXConnectorFactory;
    importjavax.management.remote.JMXServiceURL;

    importorg.apache.activemq.broker.jmx.BrokerViewMBean;
    importorg.apache.activemq.broker.jmx.QueueViewMBean;

    public class JMXMonitor {

        public static void main(String[] args) throws IOException,
                MalformedObjectNameException, InstanceNotFoundException,
                IntrospectionException, ReflectionException {
            JMXServiceURLurl = new JMXServiceURL(

"service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi");
            JMXConnector connector = JMXConnectorFactory.connect(url, null);
            connector.connect();
            MBeanServerConnection connection =
    connector.getMBeanServerConnection();
```

```
                ObjectName name = new ObjectName(

    "org.apache.activemq:type=Broker,brokerName=localhost");
                BrokerViewMBeanmBean =
        (BrokerViewMBean) MBeanServerInvocationHandler
                        .newProxyInstance(connection, name, BrokerViewMBean.
class, true);
                for (ObjectNamequeueName : mBean.getQueues()) {
                    QueueViewMBeanqueueMBean =
        (QueueViewMBean) MBeanServerInvocationHandler
                            .newProxyInstance(connection, queueName,
                                    QueueViewMBean.class, true);

    System.out.println("\n-----------------------------\n");

                    // 消息队列名称
                    System.out.println("States for queue --- " + queueMBean.
getName());

                    // 队列中剩余的消息数
                    System.out.println("Size --- " + queueMBean.getQueueSize
());
                    // 消费者数
                    System.out.println("Number of consumers --- "
                            + queueMBean.getConsumerCount());

                    // 出队数
                    System.out.println("Number of dequeue ---"
                            + queueMBean.getDequeueCount());
                }
            }
        }
```

可能读者会觉得奇怪，我们使用 ActiveMQ 的主要目的是为了减少我们在客户端和服务器端的通信代码的开发量，为什么会突然要获取队列上注册的信息呢？

有那么一些场景需要做服务端的定时作业的调度，由于我们客户端在 MQ 队列上的注册规则是“功能名称_IP 地址”这样的格式，当某个功能需要对注册到 MQ 队列的客户机进行作业调度的时候，就可以使用 JMX 先从队列上获取注册了这个功能的客户机，然后向这些客户机所注册的队列发送调度信息，如图 6.24 所示。

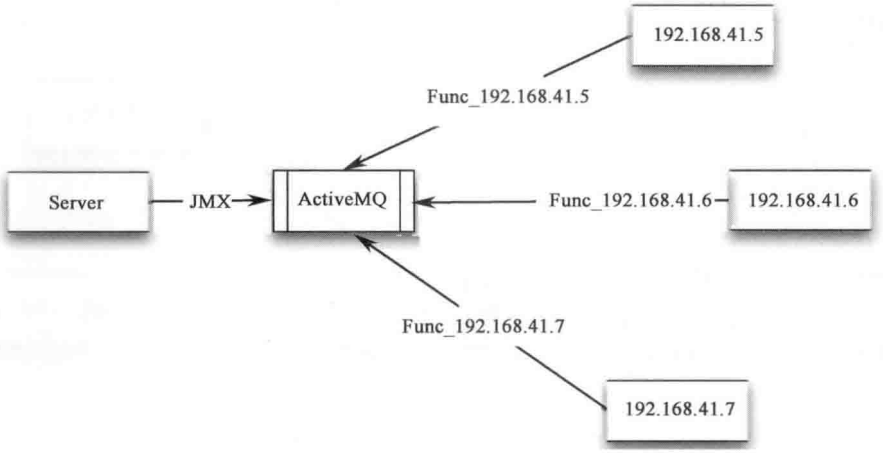


图 6.24　使用 JMX 获取 Agent 注册在队列上的名称

首先，客户机会在 MQ 上按照规则注册上各个功能性插件的信息。然后，Func 功能的 Server 端就会通过 JMX 协议去队列中获取已经注册好的队列名称，获取到的结果为 192.168.41.5、192.168.41.6、192.168.41.7。最后，Func 功能的 Server 端就会发送作业调度的消息，让各个客户端完成作业的调度，如图 6.25 所示。

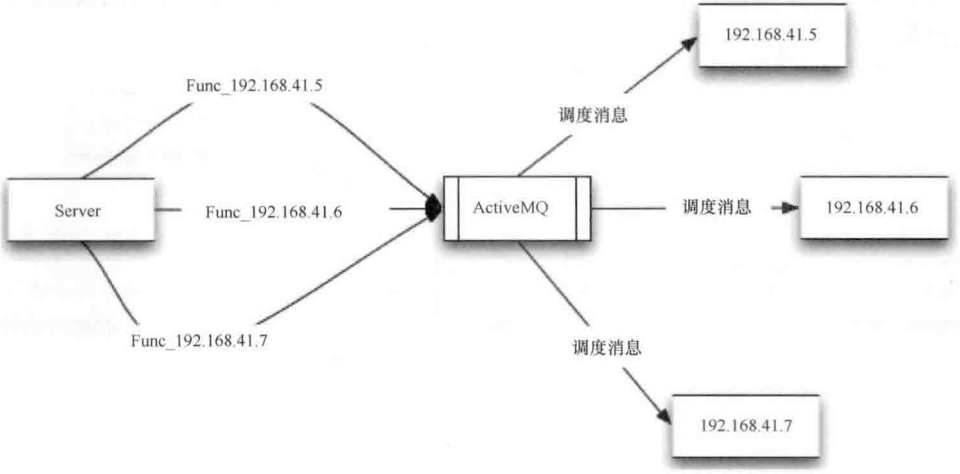


图 6.25　根据队列上注册的名称进行客户端的调度

除了完成作业的调度功能之外，我们还可以通过 JMX 获取 ActiveMQ 上注册的所有队列的消息数量。这样就可以有一个队列监控程序，定期检查是否出现队列大量堵塞的情况，

如图 6.26 所示。

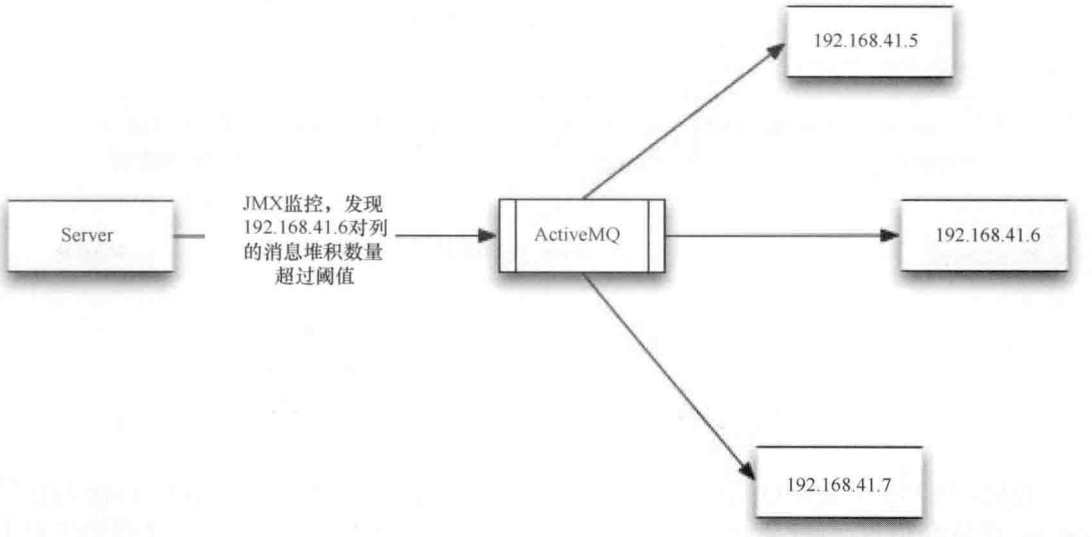


图 6.26　使用 JMX 监控队列堆积情况

一旦队列发生了堵塞，就可以清空该队列的信息，避免客户端出现无法接收后续消息的异常情况，如图 6.27 所示。

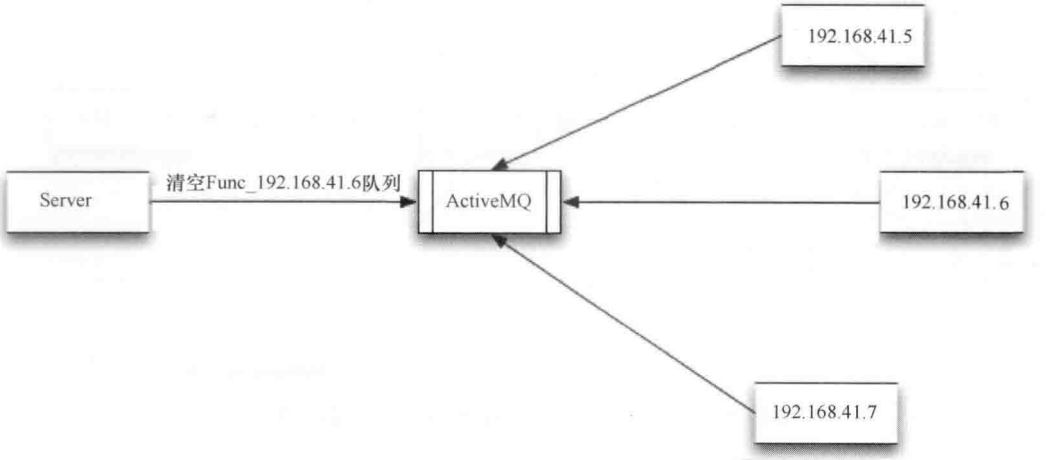


图 6.27　使用 JMX 清理 ActiveMQ 上的队列

6.6 ActiveMQ 的 HA 方案

为了避免由于消息中间件出现故障，导致系统不可用的情况，为消息中间件做高可用是有必要的。ActiveMQ 的 HA 方案有许多种，以下仅介绍其中一种。随着客户端大量地连接到消息中间件，我们的消息中间件也需要做一定的冗余。客户端的 FailOver 机制保证了客户端到消息中间件不间断，接着我们需要为 ActiveMQ 做冗余，保证一个 ActiveMQ 进程停止了不会对整体有影响。下面以 CentOS 为例，采用 NFS 的方案为 ActiveMQ 做高可用。

6.6.1 配置 NFS 服务器

NFS（Network File System，即网络文件系统）是 FreeBSD 支持的文件系统中的一种，它允许网络中的计算机之间通过 TCP/IP 网络共享资源。在 NFS 的应用中，本地 NFS 的客户端应用可以透明地读/写位于远端 NFS 服务器上的文件，就像访问本地文件一样。

安装 NFS 服务器：

```
yum install nfs-utils rpcbind
```

设置共享目录，编辑/etc/exports：

```
/home/mqsharedata 192.168.41.199(rw,sync,no_root_squash)
/home/mqsharedata 192.168.41.199(rw,sync,no_root_squash)
```

启动 NFS 服务器：

```
service rpcbind start
chkconfig rpcbind on
service nfs start
```

6.6.2 配置 NFS 客户端

安装 NFS 的客户端：

```
yum install nfs-utils portmap（适用 centos 5）
yum install nfs-utils rpcbind（适用 centos 6）
```

检查可挂载文件系统：

```
showmount -e 192.168.188.143
```

挂载文件系统：

```
mount -t nfs 192.168.188.143:/home/mqsharedata /home/mqsharedata -o nolock
```

6.6.3 调整消息中间件的配置文件

ActiveMQ 默认会把消息持久化到 KahaDB 里面，我们修改 activemq.xml 的 kahadb 路径。

修改 activemq.xml：

```
<persistenceAdapter>
    <kahaDB directory="/home/mqsharedata"/>
</persistenceAdapter>
```

6.6.4 将 Failover 作为连接串

由于 ActiveMQ 具备 Failover 的机制，我们在使用 ActiveMQ 客户端的时候，只需要在 Failover 的连接里面提供多个 ActiveMQ 的地址，就可以让 ActiveMQ 的客户端连接在出现问题之后自动地连接到另外一台 ActiveMQ 服务器，做到故障的快速切换。

```
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
        ActiveMQConnection.DEFAULT_USER,
ActiveMQConnection.DEFAULT_PASSWORD,
"failover:(tcp://192.168.41.197:61616,tcp://192.168.41.198:61616)");
```

故障前的示意图如图 6.28 所示。

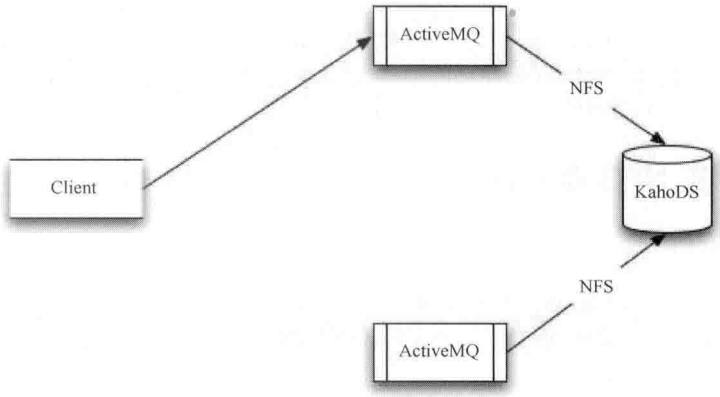


图 6.28　故障前客户端与 MQ 的连接情况

当 ActiveMQ 出现故障后，由于我们的客户端具备 Failover 机制，所以会自动连向另外一台 ActiveMQ，如图 6.29 所示。

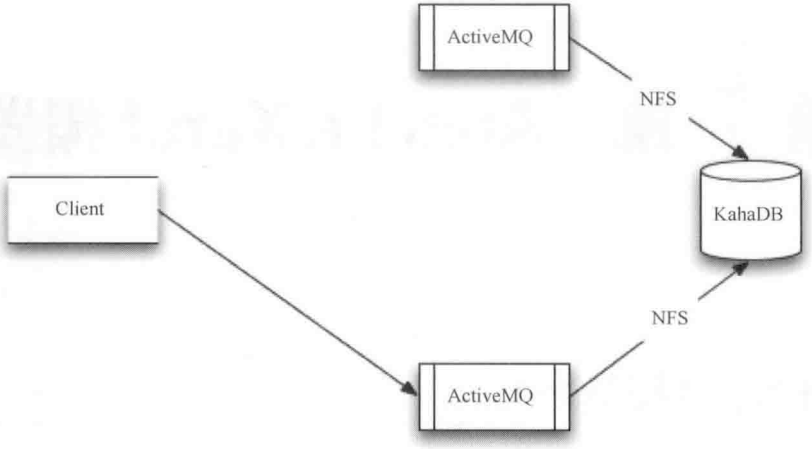


图 6.29　客户端自动连向另外一台 ActiveMQ

如果 NFS 存在单点，可以采用 DRBD 和 KeepAlive 进行 HA，这里就不再展开，有兴趣的读者可以自行了解。

6.6.5　原理

基于共享文件系统的原理还是比较简单的，因为 activemq 在启动时会锁住 lock 文件，当 Master 启动之后，作为 Slaver 的 ActiveMQ 就只能等到 lock 文件的锁释放了。

当 Master 出现故障之后之后，lock 文件的锁会被释放，然后其中一个 Slaver 就锁定了 lock 文件而作为 Master 存在。

当出现故障的 ActiveMQMaster 重新启动之后，由于 lock 文件已经被锁定，所以原来的 Master 就作为 Slaver 存在了，因为 ActiveMQ 的数据存储在 KahaDB 里面，而 KahaDB 又在 NFS 那头，所以数据自然就是一样的了。

而使用了 Failover 的客户端连接，当发现 ActiveMQ 的 Master 节点连接出现问题时，就会主动连接 Failover 里面提供的其他连接。

第7章 Apache Karaf 概览

7.1 理解 Import 和 Export

在制作 OSGi 插件包时，我们看到两个选项卡分别标识着 Import 和 Export。我们需要对 Import 和 Export 这两个概念有比较好的理解之后，在实际开发中才会比较容易接受。

Apache Karaf 是一个实现了 OSGi 的容器，在这个容器里面不同的插件包被称为 Bundle。由于 OSGi 开发过程都是模块化的，每个模块只负责自己需要完成的事情。这样就会出现一个问题，那就是一些功能是公用的，这些功能会被几个不同的 Bundle 所引用。Import 示例如图 7.1 所示。

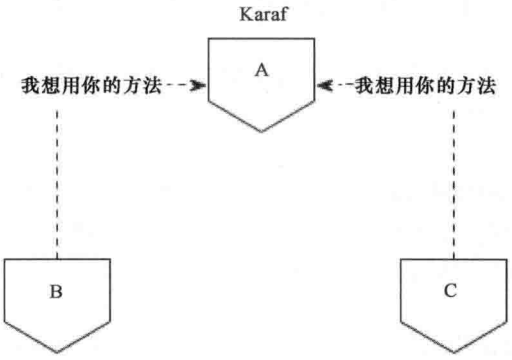


图 7.1 Import 示例

但是在容器内部就要遵循容器的规则，不是 BundleA 里面的所有方法都可以给 BundleB

和 BundleC 直接调用，首先需要做的是 BundleA 要把能供别人调用的方法或者能供别人使用的实体类等信息暴露出来，这个就是我们所说的 Export 了。我们可以把 Export 看作 BundleA 把自己愿意共享的方法贡献出来的动作。假设 Bundle 没有把方法暴露出来，就说明 Bundle 不愿意把 Bundle 信息共享出来，自然，B 和 C 都会出现无法调用的情况，如图 7.2 所示。

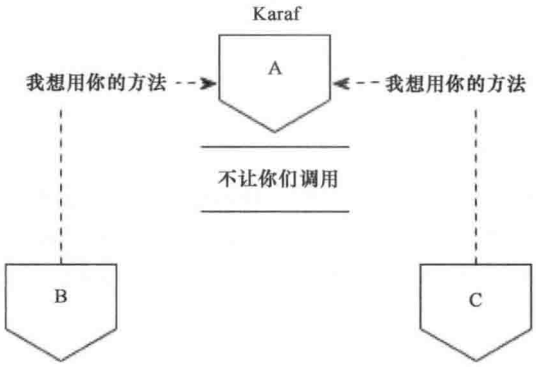


图 7.2　Bundle 不允许其他 Bundle 调用特定的方法

例如图 7.3 就是 ActiveMQBundel 把 ActiveMQ 的方法暴露出来的设定方式。

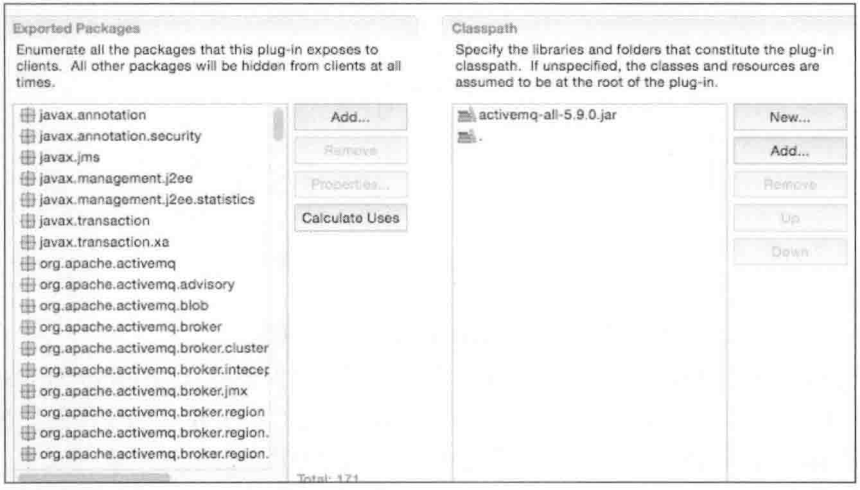


图 7.3　ActiveMQ 暴漏指定的包

Bundle 导出特定的包到容器中的示意图如图 7.4 所示。

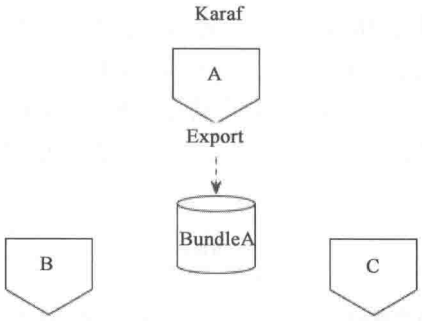


图 7.4　Bundle 导出特定的包到容器中

当 Bundle 的信息被暴露出来之后，其他 Bundle 就可以对这个 Bundle Export 的信息进行“Import”了，只有在“Import”了之后，才可以进行 Bundle 信息的调用。我们可以把 Import 的动作看作从共享出来的 Bundle 池中获取需要的 Bundle 信息，如图 7.5 所示。

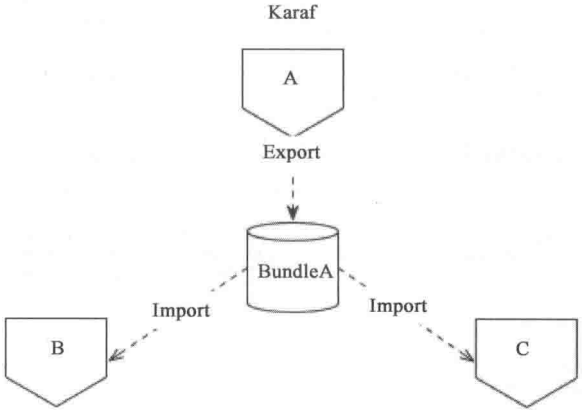


图 7.5　Import 暴漏出来的包

图 7.6 是一个“Import”了 ActiveMQBundle 的设定方式。

理解了 Import 和 Export 的具体作用之后，我们回过头来看 ActiveMQ 的 Bundle。作为一个基础插件包，是不是就只有 Export，没有 Import 呢？其实并不是这样的，我们平常使用第三方的 jar 包的时候，一般情况下都是直接把 jar 包引入到项目中就可以了，但是在制作 OSGi 插件包的时候，我们还需要关注这个第三方插件包是否依赖了其他程序包，或者说是 Java 本身的基础包。例如我们后续选用的一款叫作 Cron4J 的定时调度插件包，它就不需要任何依赖，因为它所需要的依赖都在 OSGi 容器里面默认存在，所以我们可以看到这个插件包的 Import 是非常干净的，如图 7.7 所示。

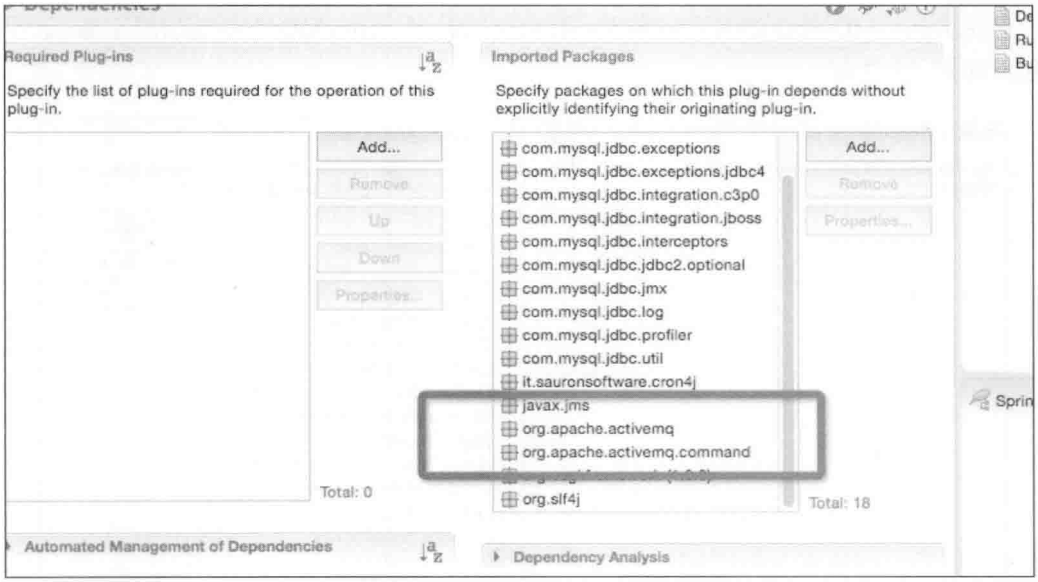


图 7.6　Import ActiveMQ 的插件包

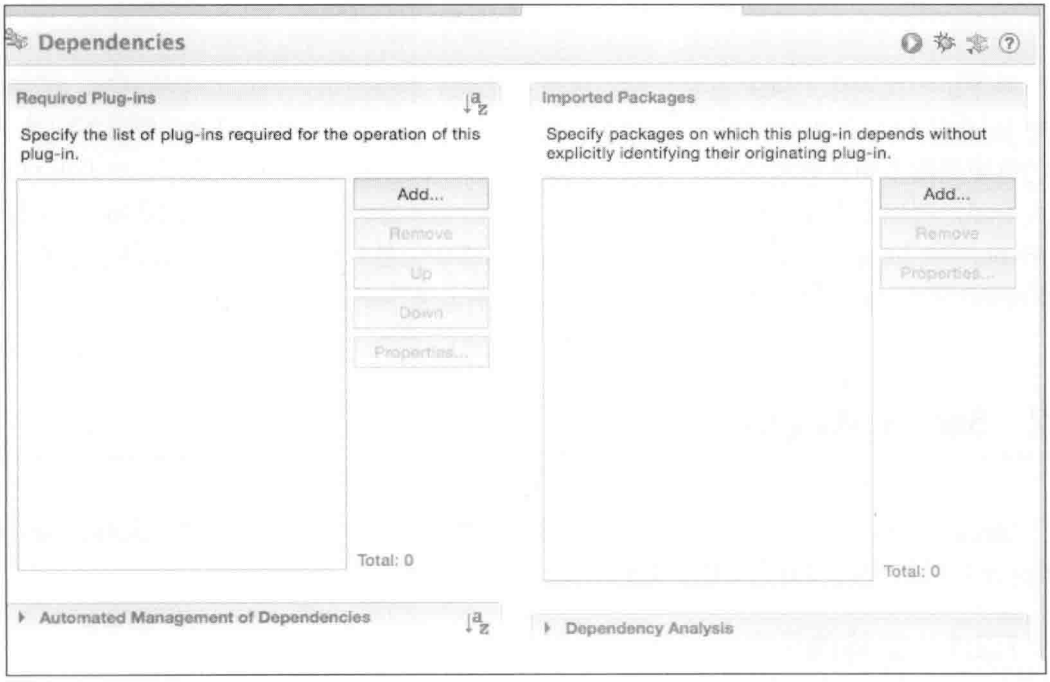


图 7.7　Cron4J Import 的插件包

但是 ActiveMQ 的依赖就比较多了，我们可以看到 ActiveMQ 大量地引用了 javax 里面的包，如图 7.8 所示。

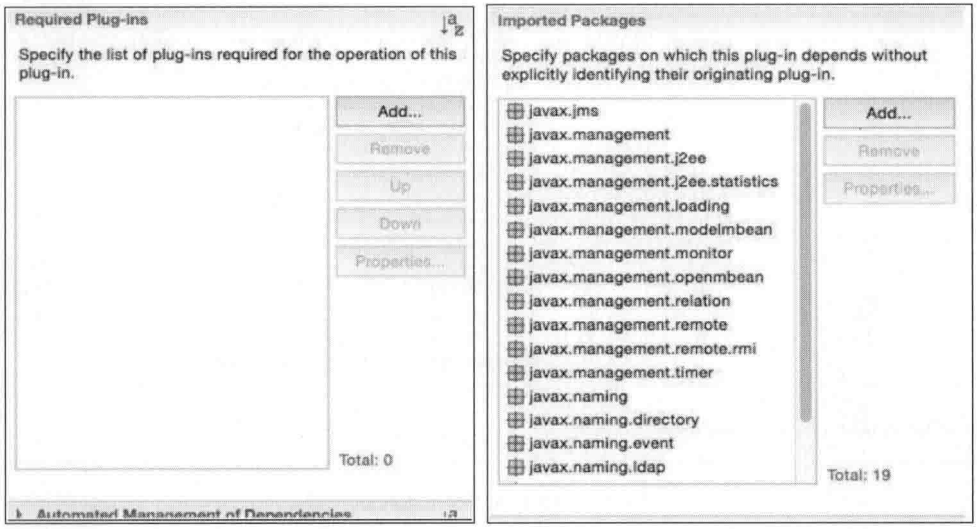


图 7.8　ActiveMQ 需要 Import 的 Package

笔者第一次制作 OSGi 包的时候在这个位置踩了不少的坑，当时制作的时候，笔者首先把 jar 包放到项目里面，发现不起效果，然后尝试把 jar 包解压后放进去，依旧不起效果，报的错误是找不到依赖包写入，当时可是纳闷了好几天。网上的资料翻遍了，由于用 OSGi 的人不多，所以没有找到合适的答案。直到周末的时候想起了.NET 技术中的 MEF 也有这种 Export 和 Import 的概念，然后随着 Karaf 的错误信息一点一点地排错，终于把第一个 ActiveMQ 的 Bundle 做好了，现在回想起来当初真是太不容易了。

7.2　Service Wraper

Service Wrapper 是一个可以让 Karaf 作为 Windows 服务运行的服务，同样，Service Wrapper 也可以让我们很容易地把 Karaf 安装成守护进程。

7.2.1　支持的平台

Karaf Wrapper 支持的平台是非常全面的，下面是它所支持的平台：

- AIX；
- FreeBSD；
- HP-UX，32-bit and 64-bit versions；
- SGI Irix；
- Linux kernels 2.2.x，2.4.x，2.6.x，例如 Debian、Vbuntu、RedHat，也支持其他发现版；
- Macintosh OS X；
- Sun OS，Solaris 9 和 10，目前支持 32 位和 64 位 sparc 以及 x86 系统；
- Windows 2000、XP、2003、Vista、2008 和 Windows 7。

7.2.2 使用 Service Wrapper

由于 Karaf 在 UNIX 和 Linux 下的 Wrapper 使用方式比较简单，此处以 Windows 主机为例来说明。

在控制台中启动 Karaf，然后运行：

```
karaf@root>features:install wrapper
```

安装完之后，我们就可以使用 Wrapper 的命令了，我们可以执行 wrapper:install–help 来查看具体的帮助信息。

```
karaf@root>wrapper:install --help
DESCRIPTION
wrapper:install
        Install the container as a system service in the OS.
SYNTAX
wrapper:install [options]

OPTIONS
        -s, --start-type
Mode in which the service is installed. AUTO_START or DEMAND_START
(Default: AUTO_START)
                (defaults to AUTO_START)
        --help
                Display this help message
        -n, --name
The service name that will be used when installing the service. (Default: karaf)
```

```
                    (defaults to karaf)
            -d, --display
      The display name of the service.
            -D, --description
      The description of the service.
                    (defaults to )
```

接下来运行 wrapper:install 命令就可以把 Karaf 安装为服务了，如果我们还希望 Karaf 能够开机启动，我们可以执行如下命令进行服务安装的预配置。

```
karaf@root>wrapper:install -s AUTO_START -n KARAF -d Karaf -D "Karaf Service"
```

执行完毕之后会出现如下提示：

```
karaf@root>wrapper:install -s AUTO_START -n KARAF -d Karaf -D "Karaf
Service"

Creating file: C:\Users\kira\Downloads\apache-karaf-2.4.1\bin\KARAF-
wrapper\.exe
Creating file: C:\Users\kira\Downloads\apache-karaf-2.4.1\bin\..\etc\
KARAF-wrapp
er.conf
Creating file: C:\Users\kira\Downloads\apache-karaf-2.4.1\bin\KARAF-
service.bat
Creating file: C:\Users\kira\Downloads\apache-karaf-2.4.1\lib\wrapper.dll
Creating file: C:\Users\kira\Downloads\apache-karaf-2.4.1\lib\karaf-
wrapper.jar
Creating file: C:\Users\kira\Downloads\apache-karaf-2.4.1\lib\karaf-
wrapper-main
.jar

Setup complete.  You may wish to tweak the JVM properties in the wrapper
configu
ration file:
C:\Users\kira\Downloads\apache-karaf-2.4.1\bin\..\etc\KARAF-wrapper.conf

before installing and starting the service.

To install the service, run:
  C:>  C:\Users\kira\Downloads\apache-karaf-2.4.1\bin\KARAF-service.
bat install
```

```
        Once installed, to start the service run:
          C:> net start "KARAF"

        Once running, to stop the service run:
          C:> net stop "KARAF"

        Once stopped, to remove the installed the service run:
          C:>   C:\Users\kira\Downloads\apache-karaf-2.4.1\bin\KARAF-service.
bat remove
```

可以看到 Karaf Service 在安装的时候生成了 Karaf Wrapper 的配置文件，并且给出了配置路径，我们后续会对这份文件进行配置，同时还给出了启停以及安装和卸载 Karaf 服务的示例，这时 Apache Karaf 的服务并没有安装，Wrapper 只是帮我们生成了用于服务安装的 exe 文件。

接下来，我们先退出 Karaf 程序，进入控制台，按照上面的提示执行 Karaf 服务的安装命令：

```
        C:\Users\kira\Downloads\apache-karaf-2.4.1\bin\KARAF-service.bat install
```

当界面上出现

```
        wrapper  | Karaf installed.
```

这样的提示之后，就说明 Karaf 的服务已经安装成功了，我们可以进入 Windows 的服务里查看 Karaf 服务，如图 7.9 所示。

名称	描述	状态	启动类型	登录为
KARAF	Kara...		自动	本地系统
KtmRm for Distri...	协调...		手动	网络服务
Link-Layer Topol...	创建...		手动	本地服务
Media Center Ex...	允许 ...		禁用	本地服务
Microsoft .NET F...	Micr...		手动	本地系统
Microsoft .NET F...	Micr...		手动	本地系统
Microsoft iSCSI I...	管理...		手动	本地系统
Microsoft Softw...	管理...		手动	本地系统
Multimedia Clas...	基于...	已启动	自动	本地系统
Net.Tcp Port Sh...	提供...		禁用	本地服务
Netlogon	为用...		手动	本地系统

图 7.9　Windows 下的 Karaf 服务

7.2.3 Karaf Wrapper 的配置文件

其实 Karaf Wrapper 安装了一次之后，就可以直接复制到同类型的不同主机上进行运行，没必要每次都跑到主机上生成一次，而为了实现能直接复制到不同的主机上就能运行的目的，我们需要调整 Karaf Wrapper 的配置文件。因为里面设定了使用的 JDK、Karaf 的目录等信息，假如两台设备的 JDK 和 Karaf_Home 目录是一样的，则无须修改这份配置文件；如果不一样，则需要把路径设置正确，否则服务是没办法正常运行的。

还有一点需要注意的是 wrapper.logfile 这个配置项，我们会发现采用 Wrapper 启动的 Karaf 输出日志的位置和使用控制台启动的 Karaf 输出日志的位置不一样，就是因为这个配置项导致的，我们可以根据自己的需要把日志输出到不同的地方。

```
#*********************************************************************
# Wrapper 的基础配置
#*********************************************************************
set.default.JAVA_HOME=C:\\Program Files\\Java\\jre1.8.0_31
set.default.KARAF_HOME=C:\Users\kira\Downloads\apache-karaf-2.4.1
set.default.KARAF_BASE=C:\Users\kira\Downloads\apache-karaf-2.4.1
set.default.KARAF_DATA=C:\Users\kira\Downloads\apache-karaf-2.4.1\data
set.default.KARAF_ETC=C:\Users\kira\Downloads\apache-karaf-2.4.1\bin\\etc

# Java 目录
wrapper.working.dir=%KARAF_BASE%
wrapper.java.command=%JAVA_HOME%/bin/java
wrapper.java.mainclass=org.apache.karaf.shell.wrapper.Main
wrapper.java.classpath.1=%KARAF_BASE%/lib/karaf-wrapper.jar
wrapper.java.classpath.2=%KARAF_HOME%/lib/karaf.jar
wrapper.java.classpath.3=%KARAF_HOME%/lib/karaf-jaas-boot.jar
wrapper.java.classpath.4=%KARAF_BASE%/lib/karaf-wrapper-main.jar
wrapper.java.library.path.1=%KARAF_BASE%/lib/

# 应用程序参数，从 1 开始
#wrapper.app.parameter.1=

# JVM 参数，从 1 开始
wrapper.java.additional.1=-Dkaraf.home="%KARAF_HOME%"
wrapper.java.additional.2=-Dkaraf.base="%KARAF_BASE%"
```

```
wrapper.java.additional.3=-Dkaraf.data="%KARAF_DATA%"
wrapper.java.additional.4=-Dkaraf.etc="%KARAF_ETC%"
wrapper.java.additional.5=-Dcom.sun.management.jmxremote
wrapper.java.additional.6=-Dkaraf.startLocalConsole=false
wrapper.java.additional.7=-Dkaraf.startRemoteShell=true
wrapper.java.additional.8=-Djava.endorsed.dirs="%JAVA_HOME%/jre/lib/
endorsed;%JAVA_HOME%/lib/endorsed;%KARAF_HOME%/lib/endorsed"
wrapper.java.additional.9=-Djava.ext.dirs="%JAVA_HOME%/jre/lib/ext;%
JAVA_HOME%/lib/ext;%KARAF_HOME%/lib/ext"

# 取消注释以启用 JMX
# wrapper.java.additional.n=-Dcom.sun.management.jmxremote.port=1616
# wrapper.java.additional.n=-Dcom.sun.management.jmxremote.authenticate=false
# wrapper.java.additional.n=-Dcom.sun.management.jmxremote.ssl=false

# 取消注释以加载自己的插件
# wrapper.java.additional.n=-Xrunyjpagent

# 取消注释以启动远程调试
# wrapper.java.additional.n=-Xdebug -Xnoagent -Djava.compiler=NONE
# wrapper.java.additional.n=-Xrunjdwp:transport=dt_socket,server=y,
suspend=n,address=5005

# Java Heap 启动大小 (MB)
#wrapper.java.initmemory=3

# Java Heap 最大值 (MB)
wrapper.java.maxmemory=512


#*******************************************************************
# Wrapper 的日志属性
#*******************************************************************
# 控制台上日志的输入格式化
wrapper.console.format=PM

# 输出的日志级别
wrapper.console.loglevel=INFO
```

```
# 日志文件输出路径
wrapper.logfile=%KARAF_DATA%/log/wrapper.log

# 日志文件输出格式
wrapper.logfile.format=LPTM

# 输出的日志文件级别
wrapper.logfile.loglevel=INFO

# 单个日志文件最大容量
wrapper.logfile.maxsize=10m

# 日志文件最大数
wrapper.logfile.maxfiles=5

# 是否记录时间日志输出
wrapper.syslog.loglevel=NONE

#*********************************************************************
# Windows Wraper 属性
#*********************************************************************
# Title to use when running as a console
wrapper.console.title=KARAF

#*********************************************************************
# Windows NT/2000/XP Wrapper 属性
#*********************************************************************
# WARNING - Do not modify any of these properties when an application
# using this configuration file has been installed as a service.
#Please uninstall the service before modifying this section.  The
#service can then be reinstalled.

# 服务的名称
wrapper.ntservice.name=KARAF

# 服务展示名称
wrapper.ntservice.displayname=Karaf

# 服务描述
```

```
wrapper.ntservice.description=Karaf Service

# 服务依赖，从 1 开始
wrapper.ntservice.dependency.1=

# 服务的启动模式
wrapper.ntservice.starttype=AUTO_START

# 允许服务在桌面呈现
wrapper.ntservice.interactive=false
```

7.3 使用控制台

Apache Karaf 的控制台是非常有用的功能模块，后续开发插件包排错都需要用到这个控制台，因为 Eclipse 对 OSGi 的调试支持得不是很好，而且 OSGi 的程序由于模块化的问题，所以插件包都比较小，代码量也很小，所以我们往往会直接把程序放到 Karaf 里面，在控制台里面排错。

Apache Karaf 除了具备热部署的功能，控制台内部还具备很多实用的功能。

7.3.1 Shell 模块

Shell 模块可以看作在 Karaf 内部自己实现的一个迷你的 Shell 环境，它提供了一些很基础的功能。

1. 在 Karaf 内部执行系统命令——shell:exec

使用 shell:exec 可以在 Karaf 内部执行命令。它可以在 Windows 和 Linux 上执行命令并返回结果。例如，在 Windows 上执行 dir 的命令：

```
shell:execcmd /c dir
```

可以得到如下结果：

```
2015/02/17  00:19<DIR>          .
2015/02/17  00:19<DIR>          ..
```

```
2015/02/17  00:19<DIR>          bin
2015/02/17  00:19<DIR>          data
2014/12/30  13:00<DIR>          demos
2014/12/30  13:00<DIR>          deploy
2014/12/30  13:00<DIR>etc
2015/02/17  00:19<DIR>          instances
2014/12/30  13:00          285,541 karaf-manual-2.4.1.html
2014/12/30  13:00          516,102 karaf-manual-2.4.1.pdf
2014/12/30  13:00<DIR>          lib
2014/12/30  13:00           28,228 LICENSE
2015/02/17  00:19                0 lock
2014/12/30  13:00            1,915 NOTICE
2014/12/30  13:00            3,933 README
2014/12/30  13:00          180,855 RELEASE-NOTES
2015/02/17  00:19<DIR>          system
             7 个文件      1,016,574 字节
10 个目录      52,517,269,504 可用字节
```

在 Linux 上执行列出目录的命令：

```
shell:execls
```

也可以得到磁盘上的目录结构：

```
LICENSE
NOTICE
README
RELEASE-NOTES
bin
data
demos
deploy
etc
instances
karaf-manual-2.3.4.html
karaf-manual-2.3.4.pdf
lib
lock
system
```

看到这里，读者可能会觉得，其实也就执行了一下命令，为什么非得跑到 Apache Karaf

中执行，在操作系统层面上执行不就可以了吗？单纯看这里，会觉得执行命令的这个功能不实用，但是结合 Karaf 内置的 SSHD 服务，就能带来很多便利。

2. 获取 Karaf 基础信息——shell:info

在控制台上执行 shell:info，可以得到如下输出结果：

```
Karaf
Karaf version              2.3.4
Karaf home                 /Users/kira/apache-karaf-2.3.4
Karaf base                 /Users/kira/apache-karaf-2.3.4
OSGi Framework             org.apache.felix.framework - 4.0.3

JVM
Java Virtual MachineJava HotSpot(TM) 64-Bit Server VM version 25.25-b02
  Version                  1.8.0_25
  Vendor                   Oracle Corporation
Pid                       2269
  Uptime                   3 minutes
  Total compile time         7.587 seconds
Threads
  Live threads              32
  Daemon threads            27
  Peak                     37
  Total started             46
Memory
  Current heap size          28,396 kbytes
  Maximum heap size          466,432 kbytes
  Committed heap size        125,952 kbytes
  Pending objects            0
  Garbage collector Name = 'PS Scavenge', Collections = 5, Time = 0.042 seconds
  Garbage collectorName = 'PS MarkSweep', Collections = 1, Time = 0.058 seconds
Classes
  Current classes loaded      3,811
  Total classes loaded        3,811
  Total classes unloaded      0
Operating system
```

```
Name                    Mac OS X version 10.10.2
Architecture             x86_64
Processors               4
```

可以看到 shell:info 的方法为我们提供了以下几大块的信息：

- Karaf 的基础信息，这部分信息在后续开发中可以帮助我们定位 Karaf 都装到什么位置去了；
- JVM 的基础信息，包括使用的 JDK 版本、当前使用的线程数、当前使用的内存数量；
- 操作系统信息，包括系统的名称、CPU 的核心数以及 CPU 的架构。

7.3.2 OSGi 模块

当 OSGi 包打包完成并且部署到了 Apache Karaf 的 deploy 目录下面之后，我们需要确认这些 OSGi 包是否都正常启动了，这时我们需要用到 OSGi 模块里面的 list 功能，命令如下：

```
osgi:list
```

执行完后我们可以看到插件的状态列表：

```
START   LEVEL 100 , List Threshold: 50
  ID    State       Blueprint   Level  Name
[58]    [Active]    [Failure]   [80]    mvn:snc/ shellextra/ 0.0.1- SNAPSHOT
[60]    [Active]    [Created]   [80]    Apache Karaf :: Demos :: Extend Console
Command (2.3.4)
[61]    [Active]    [       ]   [80]    osgi-activemqall-bundle (2.0)
[62]    [Resolved]  [       ]   [80]    snc_topsql_agent (2.0.0)
[63]    [Resolved]  [       ]   [80]    snc_playbook_server (2.0)
[64]    [Resolved]  [       ]   [80]    snc_session_tosql_server (2.0.0)
[65]    [Active]    [       ]   [80]    osgi-cron4j-bundle (2.0)
[66]    [Resolved]  [       ]   [80]    snc_topsql_server (2.0.0)
[67]    [Resolved]  [       ]   [80]    snc_download_server (2.0)
[68]    [Active]    [       ]   [80]    osgi-db-bundle (2.0)
[69]    [Resolved]  [       ]   [80]    Snc_result_server (2.0.0)
[70]    [Resolved]  [       ]   [80]    snc_upload_agent (2.0)
[71]    [Active]    [       ]   [80]    osgi-log4j-bundle (2.0)
[72]    [Resolved]  [       ]   [80]    snc_download_agent (2.0)
[73]    [Starting]  [       ]   [80]    Snc_pluginstatus_server (1.0.0)
```

```
[74]    [Active]    [          ][80]    osgi-gson-bundle (2.0)
[75]    [Active]    [          ][80]    snc_framework_common (2.0)
[76]    [Resolved]  [          ][80]    snc_md5_agent (2.0)
[77]    [Resolved]  [          ][80]    snc_schedule_server (2.0)
[78]    [Resolved]  [          ][80]    snc_sql_agent (2.0)
[79]    [Active]    [          ][80]    osgi-dom4jAndPoi (1.0.0)
[80]    [Resolved]  [          ][80]    snc_inspection_server (2.0)
[81]    [Resolved]  [          ][80]    snc_structure_agent (2.0)
[82]    [Resolved]  [          ][80]    snc_session_topsql_agent (2.0.0)
[83]    [Resolved]  [          ][80]    snc_statuscenter_server (2.0)
[84]    [Resolved]  [          ][80]    snc_command_agent (2.0)
```

State 列表示 Active 的插件包启动是正常的，而所有不是 Active 的插件包都是没有正常启动的，需要我们进一步查看为什么没有启动成功。

7.3.3 LOG 模块

在完成插件包的部署之后，我们会先运行 osgi:list，检查插件包是否正常启动。假设我们检查到存在没有 Active 的插件包，那我们就需要检查插件包的出错日志，这个时候 Log 模块就派上用场了。

在控制台上打印所有的日志：

最简单的做法是在控制台上把所有的日志信息都打印出来，这个时候我们就可以使用 Log:Display 了。

```
        2015-02-17 20:01:57,941 | ERROR | raf-2.3.4/deploy | MessageController
| ork.controller.MessageController  195 | 75 - snc_framework_common - 2.0.0 |
[异常]:[TOPSQL 服务端_2.0]:[SERVER_INIT_PROPERTIES]:[服务端初始化参数异
常]:java.io.FileNotFoundException: mqServer.properties (No such file or
directory)
        2015-02-17  20:01:57,943  |  WARN  |  raf-2.3.4/deploy  |
fileinstall| ?                                                  ? | 6 -
org.apache.felix.fileinstall - 3.2.8 | Error while starting bundle:
file:/Users/kira/apache-karaf-2.3.4/deploy/snc_topsql_server_2.0.0.jar
        org.osgi.framework.BundleException: Activator start error in bundle
snc_topsql_server [66].
            atorg.apache.felix.framework.Felix.activateBundle(Felix.java:202
```

```
7)[org.apache.felix.framework-4.0.3.jar:]
            atorg.apache.felix.framework.Felix.startBundle(Felix.java:1895)[
org.apache.felix.framework-4.0.3.jar:]
            atorg.apache.felix.framework.BundleImpl.start(BundleImpl.java:94
4)[org.apache.felix.framework-4.0.3.jar:]
            at
org.apache.felix.fileinstall.internal.DirectoryWatcher.startBundle(Directory
Watcher.java:1263)[6:org.apache.felix.fileinstall:3.2.8]
            at
org.apache.felix.fileinstall.internal.DirectoryWatcher.startBundles(Director
yWatcher.java:1235)[6:org.apache.felix.fileinstall:3.2.8]
            atorg.apache.felix.fileinstall.internal.DirectoryWatcher.process
(DirectoryWatcher.java:524)[6:org.apache.felix.fileinstall:3.2.8]
            atorg.apache.felix.fileinstall.internal.DirectoryWatcher.run(Dir
ectoryWatcher.java:308)[6:org.apache.felix.fileinstall:3.2.8]
        Caused by: java.lang.NullPointerException
            atjava.net.URI$Parser.parse(URI.java:3042)[:1.8.0_25]
            atjava.net.URI.<init>(URI.java:588)[:1.8.0_25]
            atorg.apache.activemq.ActiveMQConnectionFactory.createURI(Active
MQConnectionFactory.java:175)[61:osgi-activemqall-bundle:2.0]
            atorg.apache.activemq.ActiveMQConnectionFactory.setBrokerURL(Act
iveMQConnectionFactory.java:375)[61:osgi-activemqall-bundle:2.0]
            atorg.apache.activemq.ActiveMQConnectionFactory.<init>(ActiveMQC
onnectionFactory.java:154)[61:osgi-activemqall-bundle:2.0]
            atframework.JMSConsumer.start(JMSConsumer.java:60)[75:snc_framew
ork_common:2.0]
            atsnc_topsql_server.Activator.start(Activator.java:105)[66:snc_t
opsql_server:2.0.0]
            atorg.apache.felix.framework.util.SecureAction.startActivator(Se
cureAction.java:645)[org.apache.felix.framework-4.0.3.jar:]
            atorg.apache.felix.framework.Felix.activateBundle(Felix.java:197
7)[org.apache.felix.framework-4.0.3.jar:]
            ... 6 more
```

通过查看日志信息，我们可以看到插件包没启动成功的原因是由于参数初始化异常导致的，这时我们就可以有针对性地进行程序的调整了。

7.3.4 SSHD 模块

Apache Karaf 内嵌了 SSHD 的服务，我们可以通过一个 SSH 容器跳到另外一个 Apache Karaf 容器，如图 7.10 所示。

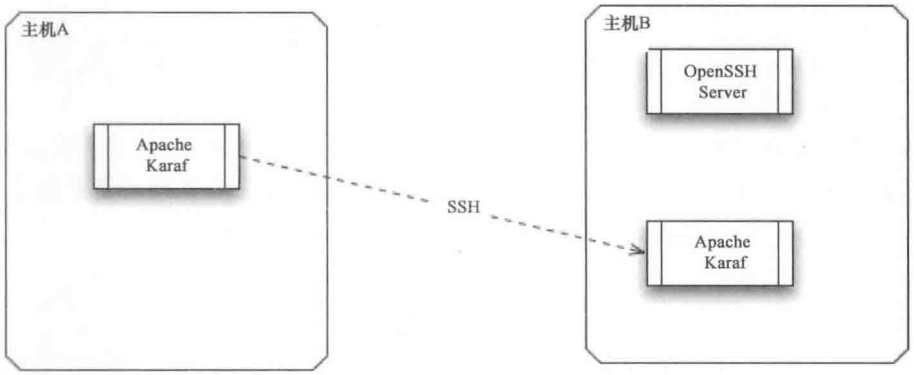


图 7.10　Karaf 内置的 SSHD 模块

也可以通过 Karaf 的 SSH 客户端跳到一个开启了 SSH 服务的主机上，由于我们的客户机部署到了许多不同的设备上，当需要手工检查某个插件包是否出现异常，或者要跳转到某台设备上进行插件包的重新加载，再或者是移除插件包时，我们就可以用到这个功能了。

下面我们先在 UNIX 系统上开启 Apache Karaf，然后再在 Windows 系统上开启 Apache Karaf 服务，接着使用 SSH 模块跳转到 Windows 主机上，移除 Karaf 的 Deploy 目录上的一份文件。

1．跳转到 Windows 主机

首先要跳转到 Windows 主机，执行如下命令：

```
ssh -p 8101 karaf@10.211.55.65
```

交互式的提示会要求输入密码，这时输入密码 karaf，然后就可以跳转到 10.211.55.65 这台 Windows 主机上进行操作了。这个时候，我们并没有进入到 Windows 的系统层面，我们仅仅是在 Apache Karaf 的容器内部，因为 Windows 本身没有开启 SSH 服务，我们使用的是 Apache Karaf 的。

2．使用 Shell 模块

进入到 Karaf 容器内部之后，前面讲到过的 Shell 模块在这时就发挥作用了。我们先执行一下 shell:info：

```
Operating system
```

```
Name                    Windows 7 version 6.1
Architecture                amd64
Processors                  1
```

可以看到我们的确进入到启动 Apache Karaf 的 Windows 内部了，接着我们需要把部署到 Deploy 目录下的 osgi-db-bundle_2.0.0.jar 包移除。我们首先要做的事情是确定 Apache Karaf 到底安装到了哪个目录，shell:info 为我们提供了非常有用的信息。

```
Karaf
Karaf version          2.4.1
Karaf home             C:\Users\kira\Downloads\apache-karaf-2.4.1
Karaf base             C:\Users\kira\Downloads\apache-karaf-2.4.1
OSGi Framework         org.apache.felix.framework - 4.4.1
```

通过 shell:info，我们知道 Apache Karaf 安装在了 C:\Users\kira\Downloads\ apache - karaf-2.4.1 目录下，根据我们已知的 karaf 目录结构，我们需要移除的文件是 C:\Users\kira\Downloads\apache-karaf-2.4.1\deploy\osgi-db-bundle_2.0.0.jar。

这时我们再次调用 Shell 模块完成移除这个动作：

```
shell:execcmd /c del /s
C:\\Users\\kira\\Downloads\\apache-karaf-2.4.1\\deploy\\-bundle_2.0.0.jar
```

需要注意的是，Windows 操作系统的路径需要加转义符号，不然会识别不出来。通过这样一系列的操作，我们就可以完成移除一个坏掉的插件包的任务了，但是每台主机都这样操作还是不方便，所以我们后续会增加文件下发的功能，实现“自己更新自己”。当由于某些功能异常而导致 Karaf 插件包无法正常启动时，这种做法还是非常实用的。

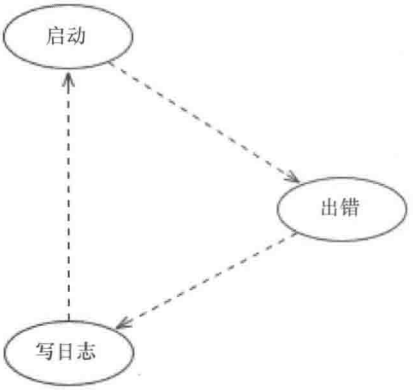
7.4 Karaf 的日志

使用 Log 模块可以监控 Karaf 的日志输出，我们还可以通过监控 Karaf 产生的其他几种日志对 Karaf 进行观察。

7.4.1 Karaf.Out

首先是{KARAF_BASE}/data/Karaf.out 这份日志，这份日志不受任何配置文件控制，只是标准输出的重定向，在最新的版本里已经改进了，可以配置标准输出到/dev/null，

如果是之前的版本，打开 bin/start 文件，就会看到 exec “$KARAF_HOME”/bin/karaf server “$@” >> “$KARAF_DATA/karaf.out” 2>&1 &。也可以手动更改重定向的位置，并且通过配置 Syslog-ng 来实现日志的 rotation，或者手写一个 Java 小程序，通过 log4j 来管理。OSGi 的程序有个特点，就是一旦出错，就会重新启动，但是一旦重新启动，又会再次出错，而每次出错的日志都会在这份重定向生成的日志文件中记录，如图 7.11 所示。



7.11 由于启动出错导致不断写日志

这样就会造成一个烦人的问题，一旦某个插件包由于依赖包的导入缺失而导致出错，并且长期没有人发现，就会出现这一份 karaf.out 的日志文件容量暴增的情况。笔者就曾经碰到过这样的问题，由于开发了一个新的插件包，需要更新到所有的客户机上，所以我们从管理服务器上把插件包批量地下发了下去，下发下去之后没有进行插件包状态的检查，大意地以为插件包都正常启动了。一周以后，维护团队开始抱怨我们的运维客户端产生了很大的日志文件，日志文件的大小居然有 30 GB，这可把我们吓坏了，赶紧进行问题的排查，最终定位到问题就出现在这里。因为这份 karaf.out 的文件是重定向生成的，由于我们的客户端没有导入一个必须的依赖包，导致 Apache Karaf 不断地输出错误信息，于是就出现了这样一个 30 GB 大小的日志文件。

7.4.2 Karaf.log

{KARAF_BASE}/data/log/karaf.log 这个日志在配置标准的 Karaf 日志文件（etc/org.ops4j.pax.logging.cfg）之前就存在了，而且之后不管如何配置 loglevel，它都存在。这是因为这个文件是日志管理组件（karaf logging service）启动之前的日志输出，当然不能受之后加载的配置管理。

需要重新定位这份日志文件的输出可以修改 etc/java.util.logging.properties，其实这个文件用的是默认的 log appender.out，所以只要覆盖配置就行了。例如，更改输出位置，那就加一条：

```
log4j.appender.out.file=logs/karaf.log
```

karaf.log 这份文件输出的信息太少，对我们的排错没有太大的作用，所以在实际工作中，我们很少会使用这份日志文件。

7.4.3 Application log4j 日志

根据 Karaf 的 guide，可以参照 log4j 的配置信息来配置日志。问题是现在有一部分应用使用的是 logback，怎么办？

第一要改的是日志服务（logging service），既然要用 logback 的服务，默认的日志服务就不能用了，同时也得加上 logback 依赖的包。

修改 etc/startup.properties：

```
mvn\:org.ops4j.pax.logging/pax-logging-service/1.7.2 = 8
# added for logback
mvn\:org.codehaus.groovy/groovy-all/2.2.1 = 8
mvn\:org.codehaus.janino/com.springsource.org.codehaus.commons.compi
ler/2.6.1 = 8
mvn\:org.apache.servicemix.bundles/org.apache.servicemix.bundles.gro
ovy/1.6.3_2 = 8
mvn\:org.apache.servicemix.bundles/org.apache.servicemix.bundles.ant
lr/2.7.7_5 = 8
mvn\:org.apache.servicemix.bundles/org.apache.servicemix.bundles.asm
/2.2.3_5 = 8
mvn\:org.ops4j.pax.logging/pax-logging-api/1.7.2 = 8
mvn\:org.ops4j.pax.logging/pax-logging-logback/1.7.2 = 8
# end for logback
```

修改配置文件 etc/org.ops4j.pax.logging.cfg：

```
org.ops4j.pax.logging.logback.config.file=${karaf.base}/config/logback.xml
```

7.5 Karaf 子实例

7.5.1 使用 Karaf 子实例

Karaf 可以以多实例的模式运行，一个子实例其实就是一个 Karaf 的拷贝，但是一个子实例不会把整个 Karaf 都复制，它只是把 Karaf 的配置文件以及 bin 目录的信息复制到子实例的目录中。

控制台的 admin 命令可以让我们对子实例进行创建和管理，每个新的子实例都是通过 admin 命令创建的。

1. 准备环境

在开始创建子实例之前，我们先在 deploy 下放置一些 OSGi 的 Bundle 包，如图 7.12 所示。

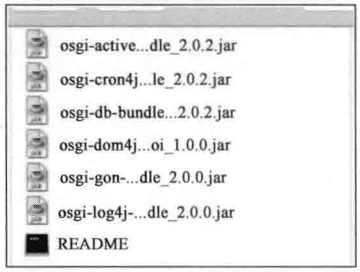


图 7.12 预先准备的 Bundle 包

然后我们启动 Karaf 容器，使用 osgi:list 命令查看 bundle 包是否正常启动了：

```
START LEVEL 100 , List Threshold: 50
   ID   State          Blueprint   Level   Name
[  54]  [Active     ]  [          ][  80]  osgi-log4j-bundle (2.0)
[  55]  [Active     ]  [          ][  80]  osgi-activemqall- bundle (2.0)
[  56]  [Active     ]  [          ][  80]  osgi-db-bundle (2.0)
[  57]  [Active     ]  [          ][  80]  osgi-gson-bundle (2.0)
[  58]  [Active     ]  [          ][  80]  osgi-dom4jAndPoi (1.0.0)
[  59]  [Active     ]  [          ][  80]  osgi-cron4j-bundle (2.0)
```

可以看到所有的 Bundle 包都是 Active 状态，说明所有的 Bundle 都正常启动了。

2. 创建子实例

在控制台内部使用 admin:create 命令就可以创建子实例了，例如我们可以运行 admin:create demo 创建一个子实例，执行完毕后出现如下输出：

```
        Creating dir: /Users/kira/apache-karaf-2.3.4/instances/demo/bin
        Creating dir: /Users/kira/apache-karaf-2.3.4/instances/demo/etc
        Creating dir: /Users/kira/apache-karaf-2.3.4/instances/demo/system
        Creating dir: /Users/kira/apache-karaf-2.3.4/instances/demo/deploy
        Creating dir: /Users/kira/apache-karaf-2.3.4/instances/demo/data
        Creating   file   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
config.properties
        Creating   file   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
jre.properties
        Creating   file   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
custom.properties
        Creating file /Users/kira/apache-karaf-2.3.4/instances/demo/etc/java.
util.logging.properties
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
org.apache.felix.fileinstall-deploy.cfg
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
org.apache.karaf.log.cfg
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
org.apache.karaf.features.cfg
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
org.ops4j.pax.logging.cfg
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
org.ops4j.pax.url.mvn.cfg
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
startup.properties
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
users.properties
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
keys.properties
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
system.properties
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
org.apache.karaf.shell.cfg
        Creating   file:   /Users/kira/apache-karaf-2.3.4/instances/demo/etc/
org.apache.karaf.management.cfg
        Creating file: /Users/kira/apache-karaf-2.3.4/instances/demo/bin/karaf
        Creating file: /Users/kira/apache-karaf-2.3.4/instances/demo/bin/ start
```

```
Creating file: /Users/kira/apache-karaf-2.3.4/instances/demo/bin/stop
```

可以看到创建了一个子实例，Karaf 会帮助我们在 instance 目录下创建一个名为 demo 的文件夹，然后创建 bin、etc、system、deploy 和 data 文件夹。我们可以看到，它并没有把原始实例的 deploy 目录下的文件复制到子实例中，仅仅是把 bin 目录和 etc 目录的内容复制到了子实例当中。

3. 管理子实例

子实例创建之后，我们需要启动这个实例，使用 admin:start demo 命令进行子实例的启动。

启动完成之后，我们可以查看所有子实例的运行状态：

```
admin:list
```

可以看到当前的 Karaf 启动了 2 个实例，一个实例的 pid 是 root，另外一个实例的 pid 是 demo，并且它们的 SSH 端口也是不一样的：

```
karaf@root>admin:list
  SSH Port   RMI Ports        State       Pid     Name
[    8101] [1099/44444     ] [Started ] [ 6408] root
[    8102] [1100/44445     ] [Started ] [ 6938] demo
```

当我们不需要其中一个实例的时候，就可以使用 stop 命令停止指定的实例。

```
admin:stop demo
```

执行完之后，我们再使用 list 的命令进行查看，就会发现 demo 的实例已经停止运行了。

```
karaf@root>admin:list
  SSH Port   RMI Ports        State       Pid     Name
[    8101] [1099/44444     ] [Started ] [ 6408] root
[    8102] [1100/44445     ] [Stopped ] [    0] demo
```

我们可以发现，创建子实例只是初始化了一个 Karaf 的最小容器，但是很多时候我们希望把已经部署在 deploy 下的插件包也一起克隆过来，这时我们就可以采用 admin 模块的 clone 命令：

```
admin:clone root newdemo
```

执行完之后，我们回到 Karaf 的 instance 目录，会发现 newdemo 这个实例除了把 root 实例的基础配置复制过来之外，还把 deploy 目录下部署的插件包也一并复制过来了，如图 7.13 所示。

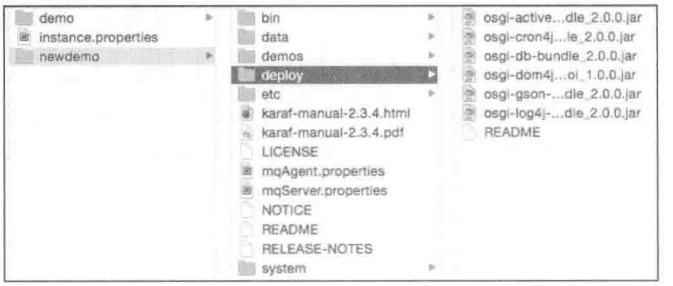


图 7.13　newdemo 子实例文件夹

当我们不再需要子实例的时候，我们就可以使用 admin 模块的 destory 命令把子实例删掉：

```
admin:destroynewdemo
```

执行完毕后，我们通过 admin:list 进行查询，会发现 newdemo 这个子实例已经不见了：

```
admin:destroynewdemokaraf@root>admin:list
  SSH Port   RMI Ports        State       Pid  Name
[    8101] [1099/44444     ] [Started ] [ 6408] root
[    8102] [1100/44445     ] [Started ] [ 7037] demo
```

再到文件系统里面进行查看，会发现 instance 目录下的 newdemo 目录也一并消失了，如图 7.14 所示。

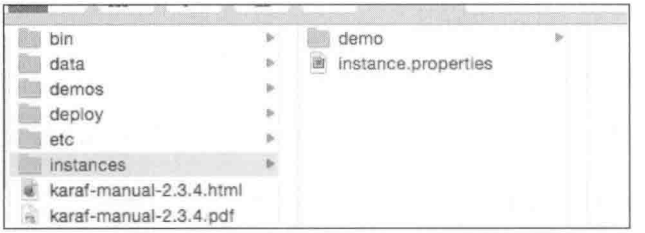


图 7.14　删除子实例

4．连接子实例

当我们能够对子实例进行管理之后，我们就可以连接上子实例，对子实例的 Bundle 包进行管理了：

```
admin:connect demo
```

进入到 demo 这一个子实例之后，我们可以执行 shell:info 查看一下当前实例的信息：

```
karaf@demo>shell:info
```

```
Karaf
Karaf version          2.3.4
Karaf home             /Users/kira/apache-karaf-2.3.4
Karaf base             /Users/kira/apache-karaf-2.3.4/instances/demo
OSGi Framework         org.apache.felix.framework - 4.0.3

JVM
  Java Virtual Machine   Java HotSpot(TM) 64-Bit Server VM version 25.25-b02
  Version                1.8.0_25
  Vendor                 Oracle Corporation
Pid                      7037
  Uptime                 3 minutes
  Total compile time     7.126 seconds
Threads
  Live threads           33
  Daemon threads         25
  Peak                   33
  Total started          45
Memory
  Current heap size      34,805 kbytes
  Maximum heap size      466,432 kbytes
  Committed heap size    251,392 kbytes
  Pending objects        0
  Garbage collector Name = 'PS Scavenge', Collections = 3, Time = 0.039 seconds
  Garbage collector Name = 'PS MarkSweep', Collections = 1, Time = 0.050 seconds
Classes
  Current classes loaded    4,043
  Total classes loaded      4,043
  Total classes unloaded    0
Operating system
  Name                   Mac OS X version 10.10.2
  Architecture           x86_64
  Processors             4
```

可以看到子实例和主实例的信息基本上是一致的，主要的差别在于 Karaf base 目录位置和主实例不一样。

进入到子实例之后，所有的操作和在主实例中是一样的，我们同样可以对子实例进行 OSGi 插件包的部署、日志的查看等任务。

7.5.2 为什么需要使用子实例

子实例的功能看起来和 Docker 的思想非常类似，每个实例都是一个箱子，我们可以通

过一些创建好的实例快速地进行其他实例的部署，如图 7.15 所示。

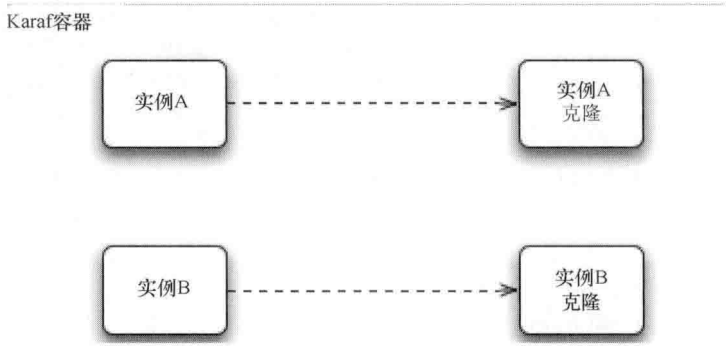


图 7.15　子实例的原理

那作为一款运维软件，为什么会用到子实例的功能呢？原因是这样的，我们在实际环境中，碰到了二级代理的情况，既然有二级代理，那么 ActiveMQ 的部署也是多个的。

这样就会有一个问题了，OSGi Server 只有一个。虽然可以通过 ActiveMQ 的特性，在一个 OSGi Server 中完成对所有 ActiveMQ 消息的接收，但是这样会造成管理上和使用上的许多不便，如图 7.16 所示。

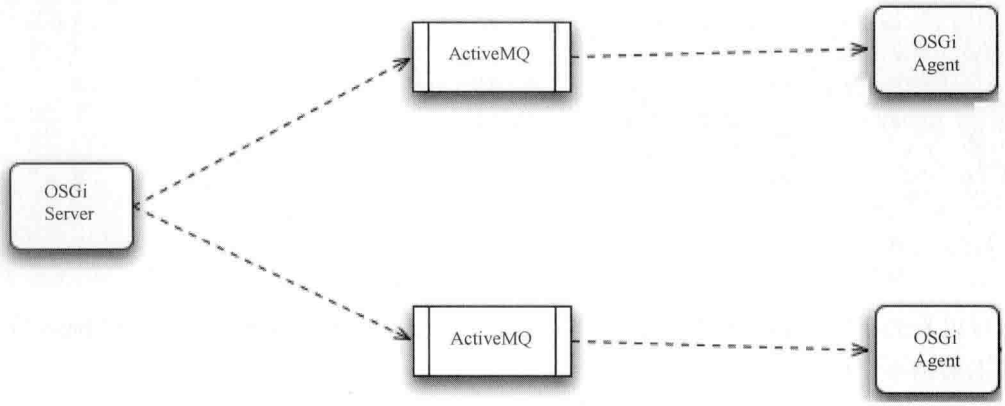


图 7.16　单实例存在性能上的瓶颈

第一点是启动给我们带来的问题，ActiveMQ 的 Failover 特性只有在第一次成功连接上服务器时才会起作用，假设其中一台 ActiveMQ 由于故障停机了，这个时候我们进行 OSGi Server 的重启，就会发现 OSGi Server 启动不了，因为给定的 ActiveMQ 连接中有一台由于故障的原因连接不上。

第二点是在程序排错上也带来了许多不便，假设我们需要对二级代理 A 下面的运维功能进行排错，由于只有一个 OSGi Server，这就使得我们在排错的时候影响到了其他正常运作的服务器。

所以当出现多个代理时，我们需要对 Karaf 进行多实例的使用，方便运维人员的管理，也方便开发人员排错，如图 7.17 所示。

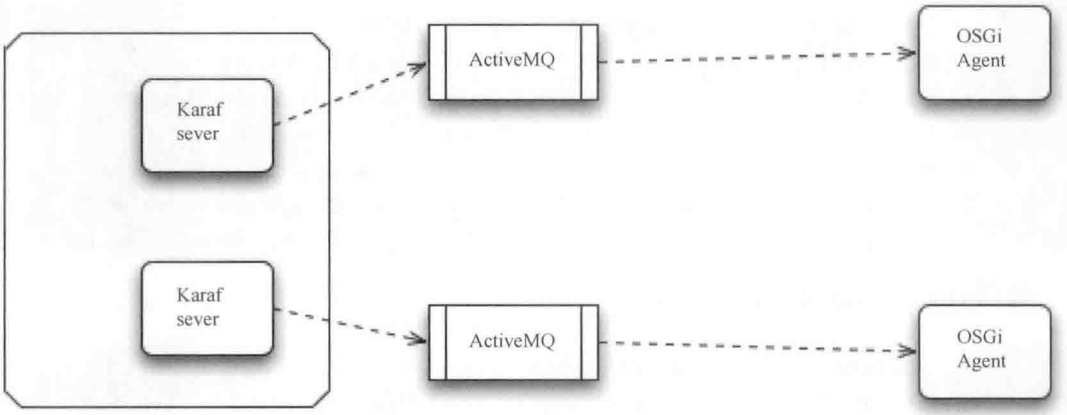


图 7.17　使用子实例轻松做到水平扩展

7.6　扩展 Karaf 控制台

执行命令是运维软件必备的一个基础功能，由于可以执行远程主机上的命令，那么也可以通过命令来执行 Karaf 内部的命令了，如图 7.18 所示。

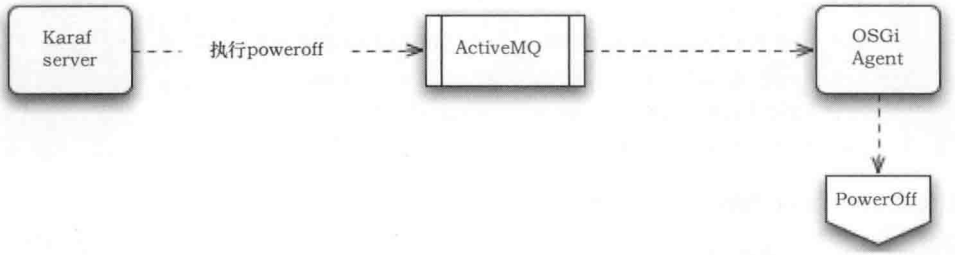


图 7.18　使用 Karaf 执行远程命令

既然可以在远端的操作系统上执行命令，那我们同样也可以通过OSGi Agent调用Karaf的命令。例如，我们通过Karaf的client查看当前容器插件包运行是否正常：

```
./client osgi:list
```

可以得到如下结果：

```
START LEVEL 100 , List Threshold: 50
   ID   State         Blueprint    Level  Name
[  54] [Active     ] [          ] [   80] osgi-log4j-bundle (2.0)
[  55] [Active     ] [          ] [   80] osgi-activemqall-bundle (2.0)
[  56] [Active     ] [          ] [   80] osgi-db-bundle (2.0)
[  57] [Active     ] [          ] [   80] osgi-gson-bundle (2.0)
[  58] [Active     ] [          ] [   80] osgi-dom4jAndPoi (1.0.0)
[  59] [Active     ] [          ] [   80] osgi-cron4j-bundle (2.0)
```

使用client在Karaf内部执行命令：

```
$ ./client shell:exec uptime
17:36  up 2 days, 17:07, 2 users, load averages: 1.97 1.96 2.05
```

也就是说我们可以通过执行命令来调用Karaf内部的功能，当我们需要开发某个功能，但发现客户端又已经具备这个功能的时候，就可以直接进行命令的调用了。

同样地，对于一些简单、常用的功能，我们也可以考虑将其包装在Karaf的控制台中，这样可以让这些功能插件具备主动触发的功能。

7.6.1 使用Maven创建项目

使用命令创建Maven项目：

```
mvnarchetype:create \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DgroupId=org.apache.karaf.shell.samples \
  -DartifactId=shell-sample-commands \
  -Dversion=1.0-SNAPSHOT
```

在命令行中配置maven项目：

```
mvnarchetype:generate
```

然后输入项目的包名等信息，编写 POM 文件，导入相应的依赖包：

```
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.apache.karaf.shell.samples</groupId>
<artifactId>shell-sample-commands<artifactId>
<packaging>bundle</packaging>
<version>1.0-SNAPSHOT</version>
<name>shell-sample-commmands</name>


<dependencies>
<dependency>
<groupId>org.apache.karaf.shell</groupId>
<artifactId>org.apache.karaf.shell.console</artifactId>
<version>2.3.7-SNAPSHOT</version>
</dependency>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<version>2.4.0</version>
<configuration>
<instructions>
```

```
<Import-Package>
org.apache.felix.service.command,
org.apache.felix.gogo.commands,
org.apache.karaf.shell.console,
            *
</Import-Package>
</instructions>
</configuration>
</plugin>
</plugins>
</build>

</project>
```

配置 Java5 编译支持：

```
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
<target>1.5</target>
<source>1.5</source>
</configuration>
</plugin>
</plugins>
</build>
```

使项目成为 Eclipse 项目：

```
mvneclipse:eclipse
```

7.6.2 编写控制台插件包

创建 HelloShellCommand.java 文件：

```
packageorg.apache.karaf.shell.samples;
importorg.apache.felix.gogo.commands.Command;
importorg.apache.karaf.shell.console.OsgiCommandSupport;
```

```
@Command(scope = "test", name = "hello", description="Says hello")
public class HelloShellCommand extends OsgiCommandSupport {

    @Override
protected Object doExecute() throws Exception {
System.out.println("Executing Hello command");
return null;
    }
}
```

HelloShellCommand 继承了 OsgiCommandSupport 类，doExecute 方法就是我们在控制台里执行命令时会被触发的方法。Command 注释声明了 HelloShellCommand 属于 test 模块，调用的方法为 hello。

7.6.3 部署插件包

为插件项目在 src/main/目录下创建 resources 目录，在 resources 目录下，继续创建 OSGI-INF/blueprint 目录，然后在里面编写如下配置文件：

```
<blueprintxmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
    <command-bundle xmlns="http://karaf.apache.org/xmlns/shell/v1.0.0">
     <command name="test/hello">
    <action class="org.apache.karaf.shell.samples.HelloShellCommand"/>
     </command>
     </command-bundle>
</blueprint>
```

插件包编写完成之后，就可以放到 deploy 目录下了，然后可以在控制台上执行 test:hello 命令进行扩展包的调用了。

7.7 使用 Web 控制台

Web 控制台提供了一个图形化的界面让我们控制 Karaf，我们可以用 Web 控制台完成子实例的创建、OSGi 插件包的启停、特性的安装和卸载等动作。

安装 Web 控制台：

```
root@karaf>features:installwebconsole
```

安装完毕之后，我们就可以访问 Karaf 的 Web 控制台了，打开浏览器，输入

```
http://localhost:8181/system/console
```

就可以看到控制台的界面了，如图 7.19、图 7.20 所示。

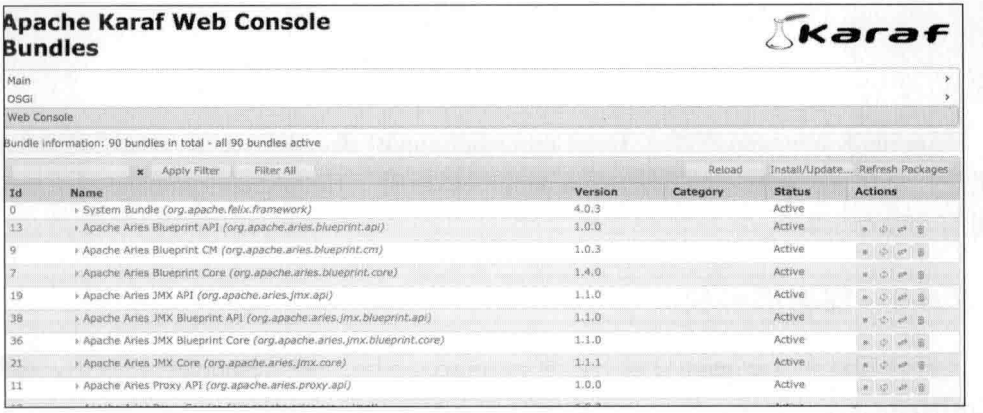


图 7.19　Apache Karaf Web 控制台

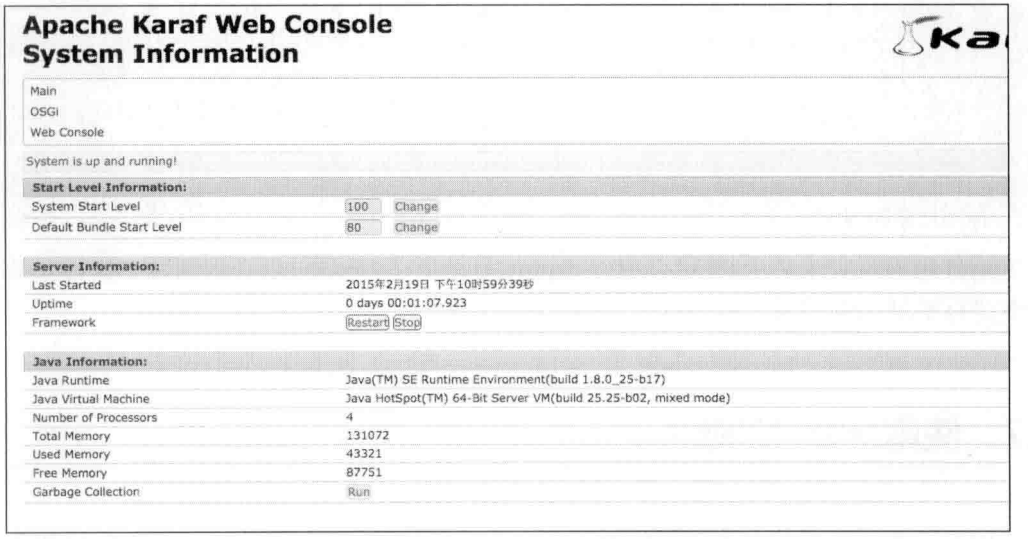


图 7.20　Apache Karaf 信息页面

Web 控制台提供了 Karaf 操作的功能，由于控制台的特点，我们一般只会在服务器端

使用这个特性。

7.8 使用 Feature——JDBC 数据源

Web 控制台是 Karaf 比较常用的一个 Feature，另外一个比较常用的 Feature 是 JDBC 数据源，当我们安装好这个 Feature 之后，就可以在 Karaf 内部进行数据库的操作了。

安装 JDBCFeature：

```
karaf@root()>feature:installjdbc
```

安装完毕之后，我们就可以使用 JDBC 模块了。

```
karaf@root()>jdbc
jdbcjdbc:createjdbc:datasourcesjdbc:delete
jdbc:executejdbc:infojdbc:queryjdbc:tables
```

常用命令：

jdbc:create 命令可以帮助我们完成数据源的创建，我们可以输入 jdbc:create–help 来查看帮助，jdbc:create 作为创建数据源的模块，是整个 jdbc:feature 最重要的一个步骤。

```
karaf@root()>jdbc:create --help
DESCRIPTION
jdbc:create

    Create a JDBC datasource

SYNTAX
jdbc:create [options] name

ARGUMENTS
name
            The JDBC datasource name

OPTIONS
      -t, --type
```

```
                The JDBC datasource type (generic, MySQL, Oracle, Postgres, H2,
                  HSQL, Derby, MSSQL)
          -v, --version
                  The version of the driver to use
          -url
                  The JDBC URL to use
          --help
                  Display this help message
          -p, --password
                  The database password
          -i, --install-bundles
                  Try to install the bundles providing the JDBC driver
          -d, --driver
     The classname of the JDBC driver to use. NB: this option is used
     only the type generic
          -u, --username
                  The database username
```

下面以 MySQL 数据源为例：

```
      jdbc:create      -t      MySQL      -urljdbc:mysql://127.0.0.1:3306/
blog?characterEncoding=utf8 -p 8565 -u root -imysqldemo
```

jdbc:datasources：

查看可用的所有数据源。当我们完成数据源的安装之后，执行 jdbc:datasources 命令就可以查看当前可用的所有数据源了，后续需要删除数据源的时候也可以先执行这个命令进行数据源的检查操作。

```
Name                    |Product    |Version    |URL        | Status
-------------------------------------------------------------------------
jdbc/mysqldemo, 694     | MySQL     | 5.6.14    | jdbc:mysql://127.0.0.1:3306/
blog? characterEncoding=utf8                                | OK
```

jdbc:delete：

当我们不需要使用某个数据源的时候，可以使用 jdbc:delete 删除某个数据源。

```
      karaf@root()>jdbc:deletemysqldemo
```

jdbc:execute：

jdbc:execute 命令可以让我们执行没有结果返回的 SQL 语句，我们可以用它进行表的创建、数据的插入等操作。

```
karaf@root()>jdbc:executejdbc/mysqldemo "create table person(name varchar(100),
nick varchar(100))"
```

执行完毕后，可以看到数据库中已经多了一张 person 表了，如图 7.21 所示。

netdisk_info	123	48 KB	InnoDB	2015-02-11
person	0	16 KB	InnoDB	2015-02-19
sp_ad	0	0 KB	MyISAM	2015-02-10

图 7.21　使用 Feature 操作数据库

然后再使用数据源进行数据的插入：

```
    karaf@root()>jdbc:executejdbc/mysqldemo "insert into person(name, nick)
values('test','test')"
```

执行完毕后再检查一下数据库，发现多了一条记录，SQL 已经被成功执行了，如图 7.22 所示。

Objects　person @blog (127.0.0.1)

name	nick
test	test

图 7.22　使用 Feature 进行数据库的插入

jdbc:query：

jdbc:query 是一个用于执行有结果返回的 SQL 语句，通常在查询的时候使用。

使用 jdbc:query 执行查询语句：

```
    karaf@root()>jdbc:queryjdbc/mysqldemo "select * from person"
    nick | name
    -----------
    test | test
```

jdbc:tables：

jdbc:tables 用于查看数据库中所有表的信息。

```
jdbc:tablesjdbc/mysqldemo
TABLE_NAME      |REMARKS     | TABLE_TYPE     | TABLE_SCHEM    | TABLE_CAT
-----------------------------------------------------------------------
netdisk_info    |            | TABLE          |                | blog
person          |            | TABLE          |                | blog
sp_ad           |            | TABLE          |                | blog
sp_asset        |            | TABLE          |                | blog
sp_auth_access  |            | TABLE          |                | blog
```

第 8 章　核心框架

8.1　核心层概述

运维框架结构图如图 8.1 所示。

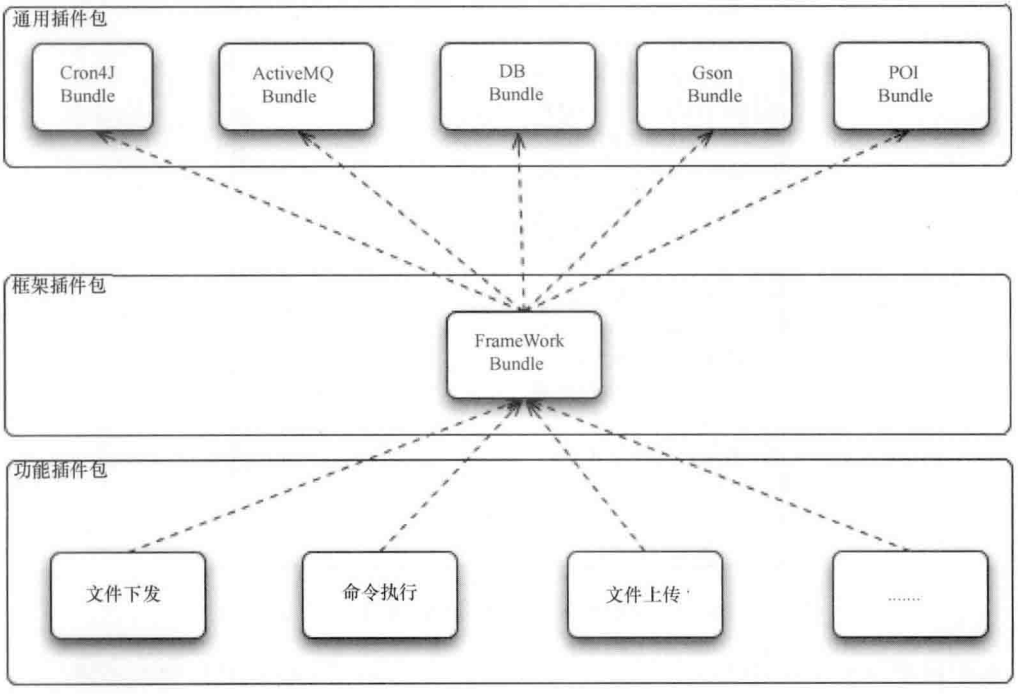


图 8.1　运维框架结构图

框架插件包的设计基于通用插件包，它们会在通用插件包的基础上完成一些后续运维功能所要用到的基础功能，方便后续实际运维功能的开发。整个运维框架的核心层包括负责消息接收调用的核心框架，负责对核心框架中的状态报告消息入库的插件状态服务端，屏蔽开发人员在发送消息时所需要了解的复杂的消息发送规则的消息转发服务端，支持步骤流程化的 PlayBook 服务端插件，对返回结果进行统一处理的结果处理服务端。由框架包和这些基础插件包共同组成核心框架层，一方面是为了减少代码的重复量，充分发挥 OSGi 插件化开发所设计的特性，另一方面又能够屏蔽功能开发人员对整体消息发送规则的了解，起到专心开发具体功能的良好作用。

8.2 核心框架

核心框架主要负责对一些通用的功能进行包装，避免这些通用的功能散落在不同的模块之中，核心框架在制作的时候，会依赖 Midao、ActiveMQ 这几个通用插件包，如图 8.2、图 8.3 所示。

图 8.2　核心框架中需要的 Import 第三方包

由于框架包要负责对消息的收/发进行处理，所以 ActiveMQ 的依赖包是必需的。由于每个客户端都需要报告自己的运行状态，所以这时作业调度的插件包也是必需的。无论在

客户端还是服务器端，都会涉及对数据库访问的需求，所以框架包也依赖了数据库操作的插件包，如图 8.4 所示。

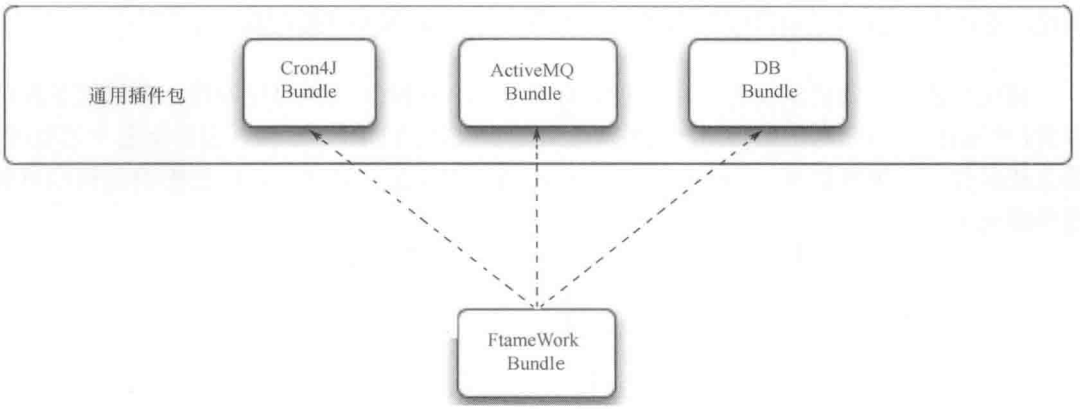


图 8.3　核心框架的依赖关系

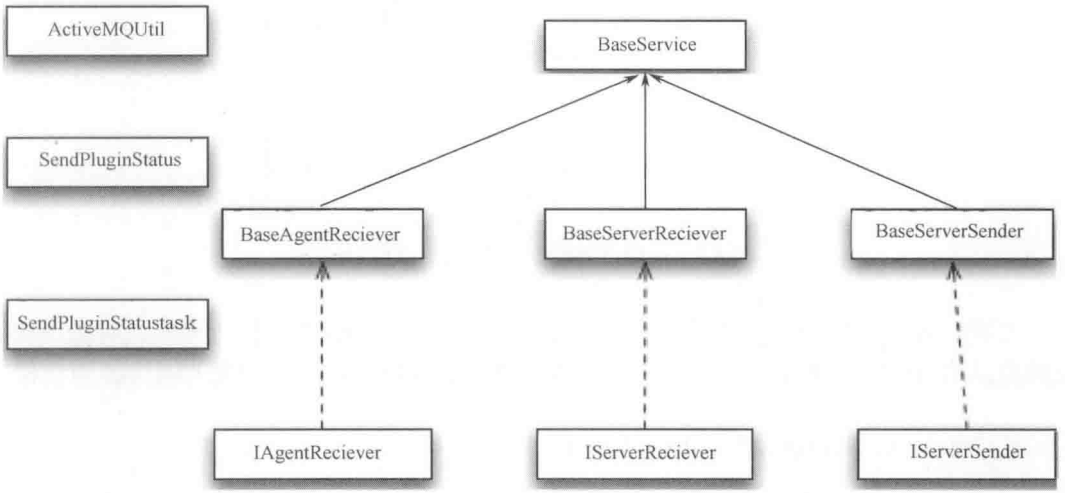


图 8.4　核心框架主要类关系图

在核心框架设计的过程中，为了屏蔽开发人员对消息收/发细节的了解，主要使用接口回调的方式进行设计。根据需要运行的环境（客户端、服务端）进行接口的实现，然后再调用基类中的方法进行收/发即可。而具体的收/发规则屏蔽在基类中，调用者只需要关心自己收到和发送的消息就可以了。

8.2.1 服务端消息处理

框架包完成的一件最重要的事情是对消息收/发规则的包装，为了后续功能的开发更加简便，不需要了解消息的具体收/发规则，框架包需要把规则屏蔽起来。

首先从消息接收的角度看，由于我们有多个 ActiveMQ 的消息中间件，所以服务端或者客户端的消费者启动的时候，会先循环读取配置文件的 URL，然后再实例化多个消费者，最后让具体的实现类把接口实现一下，这样每当框架收到消息之后，就会把消息转给具体的功能类了，如图 8.5 所示。

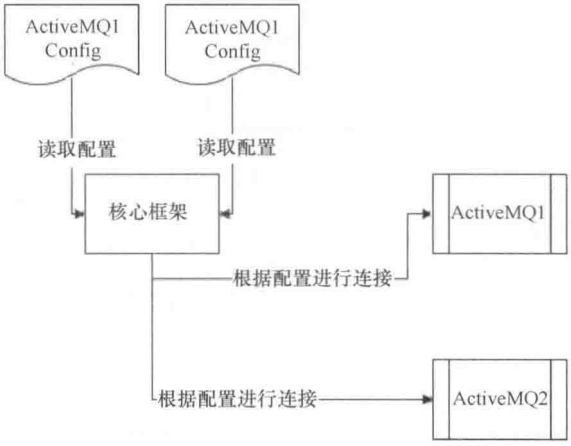


图 8.5　循环读取配置文件进行 MQ 的连接

当我们成功获取到 MQ 的连接之后，就可以采用 ActiveMQ 的侦听模式，每当消费者收到消息的时候，就调用传入的接口，完成消息的收取以及传递给指定的调用者的过程。

循环传入 ActiveMQ 对象代码如下所示。

```
        public void recieve(String queueName, final IServerReciever server
Reciever)
                    throws JMSException {
   String[] paramsStrings = properties.getProperty ("mq.url").trim().split(",");

                for (int i = 0; i < paramsStrings.length; i++) {
                    final ActiveMQUtil activeMQUtil = new ActiveMQUtil();
                    activeMQUtil.startConnection(paramsStrings[i],  username,
```

```
password);
            final ActiveMQSession session = (ActiveMQSession) active MQUtil
.createSession();
                final MessageConsumer consumer = activeMQUtil.create Consumer
(session, 1, queueName);

                    consumer.setMessageListener(new MessageListener() {

                        @Override
                        public void onMessage(Message rs) {
                            try {
                            serverReciever.recieve(rs, session, ip, proxyip);
                            } catch (Exception e) {
                                e.printStackTrace();
                            }
                        }
                    });
                }
            }
```

8.2.2 客户端消息处理

接下来需要封装起来的是消息的消费者，封装的目的是让客户端在接收消息时不用知道具体的消息消费规则，它只负责收就好了。由于我们消息发送的规则是“队列名_IP”这样的模式，所以具体的功能消费者也需要按照这样的模式进行消息的消费。

实现的方式依然是使用回调，由于一个客户端插件只会连接唯一的 ActiveMQ，所以就不需要对消费者进行循环的初始化，我们只需要按照客户端配置的 MQUrl 进行消费者的初始化就可以了。

客户端插件消息的收取封装如下所示。

```
   public void recieve(String queueName, final IAgentReciever agentReciever)
                    throws JMSException {

                final ActiveMQSession session = (ActiveMQSession) activeMQUtil
                        .createSession();

                session.createQueue(queueName + ip);
```

```
            final MessageConsumer consumer = activeMQUtil.createConsumer
(session,1, queueName + ip);

            consumer.setMessageListener(new MessageListener() {

                @Override
                public void onMessage(Message rs) {
                    try {
            agentReciever.recieveMessage(rs, session, ip, proxyip);
                    } catch (JMSException e) {
                        e.printStackTrace();
                    }
                }
            });
        }
```

8.2.3 插件状态汇报

每一个插件都需要汇报自己的状态，当插件的数量和部署的设备数越来越多的时候，我们就需要对插件进行集中管理。例如，我们需要知道哪些主机上具备某些插件，当主机不具备下发插件的时候，就不允许对该主机进行下发的操作。每个插件需要定时汇报该插件的状态，插件还需要汇报自己的版本号，方便我们对插件进行类似批量升级等操作。

由于每一个客户端和服务器端插件都会继承 BaseServerReviever、BaseAgentReciever 以及 BaseServerSender 这三个基类的其中一个，而这三个基类都继承于 BaseService 这个基类，所以当 OSGi 插件启动时，基类就会进行状态的周期性发送。

插件状态的周期性汇报如下所示。

```
    @Override
        public void run() {
            MessageProducer pluginProducer = null;

            try {
                pluginProducer = activeMQUtil.createProducer(session,
                        1, "StatusCenter", Session.AUTO_ACKNOWLEDGE);
```

```
                TextMessage statusMsg = session.createTextMessage(proxy +
"," + ip+ "," + functionName);

                pluginProducer.send(statusMsg);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (pluginProducer != null) {
                    try {
                        pluginProducer.close();
                    } catch (JMSException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
```

8.3 消息分发服务端

由于运维功能是需要触发的，而触发的本质就是把消息放到不同的队列上，让每个收到消息的客户端或者服务端完成相应的操作，在引入消息分发服务端这个概念之前，运维功能的调用者需要知道究竟要把消息发送到哪个二级代理的哪个队列上，如图 8.6 所示。

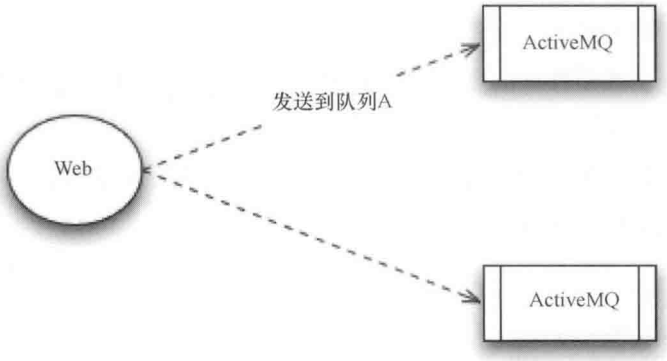


图 8.6 调用者需要了解具体的发送规则

在设计上会出现一个问题就是一旦消息的传输协议发生了改变，调用者也需要做一轮较大的改动，并且调用者对于运维框架的事情知道的太多了，我们希望调用者只有一个入口，由这个入口对后续的消息进行转发，避免调用者对后端有过多的了解。这时就可以引入一个消息的转发端，这个转发端不负责具体功能的处理，只负责把消息转送到不同的队列上，如图 8.7 所示。

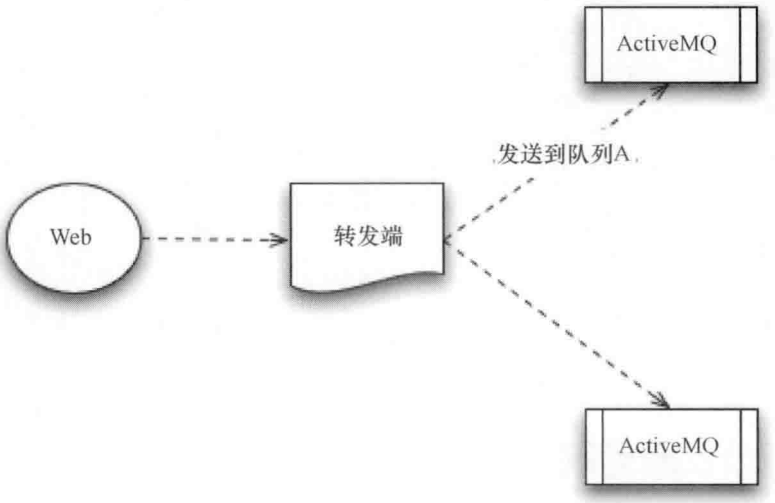


图 8.7　引入转发端后的调用方式

转发端程序流程：

（1）启动转发端并注册转发端名称。

（2）开始侦听队列消息，有消息则进入下一步。

（3）接收消息，开始进行消息的二次处理（把与外部通信的消息收/发协议转换成内部通信用的消息收/发协议）。

（4）根据协议指定的二级代理、主机 IP、功能名称进行消息的转发。

（5）完成消息的转发，重新回到步骤 2。

8.4　插件状态服务端

在设计插件状态服务端时，我们思考的一个问题是如何判定一个插件是正常的。思考了

一轮之后，我们决定利用时间来判定插件究竟是否异常。例如，我们规定的心跳周期是 3 分钟，假如超过 3 分种还是没有心跳消息发送到队列上，那么就认为这个插件出现了问题。

思路定下来之后，我们就可以进行表的设计了，如表 8.1 所示。

表 8.1 设计表

字段名	说明	字段名	说明
Id	Id	Lasttime	最后通信时间
Proxyip	二级代理 IP	Pluginname	插件名称
Ip	主机 IP	Version	插件版本

这里可以看到有一个字段是二级代理的 IP，二级代理的 IP 用于区分不同子网下拥有同一个 IP 地址的主机。要获取二级代理的 IP 其实是很麻烦的，所以我们在 Agent 端的 MQ 配置文件里面把二级代理的 IP 在安装的时候就配置上去，这样就可以在发送状态时把二级代理的 IP 一并带回来了。

插件状态服务端处理流程：

（1）启动并注册插件状态管理服务端。

（2）开始侦听插件状态队列中的消息，若有消息输入则进入下一步。

（3）从连接池获取服务端数据库连接的数据源。

（4）检查是否已经存在同样的二级代理、IP 的主机，若无，则把数据插入到表中；若有，则对该二级代理、IP 的主机的最后通信时间进行更新。

8.5 PlayBook 服务端

8.5.1 PlayBook 服务端设计目的

PlayBook 的概念是从 Ansible 那里借鉴过来的，由于在设计插件的时候，每个插件功能的独立性导致每个插件都只专心完成一件事情，并且这些事情的差异都是很大的。例如，文件下发的客户端插件就只负责下发文件，执行命令的客户端插件只负责完成命令的执行，如图 8.8 所示。

但是在实际运维过程中，我们需要完成的功能绝对不仅仅是下发一份文件，或者执行一条

命令那么简单，所以我们需要有一个能整合客户端功能的服务端，把这些功能都聚合到一起。

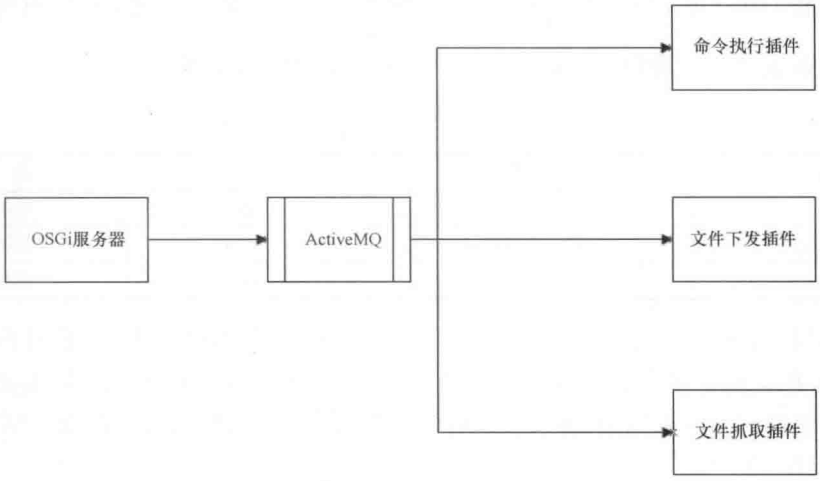


图 8.8　各插件功能相对独立，难以完成混合性的操作

例如，我们需要执行应用程序的部署，就需要先对文件进行传输，然后再对文件进行解压，接着启动相应的服务，最后可能会执行一些脚本来确保本次部署是成功的，如图 8.9 所示。

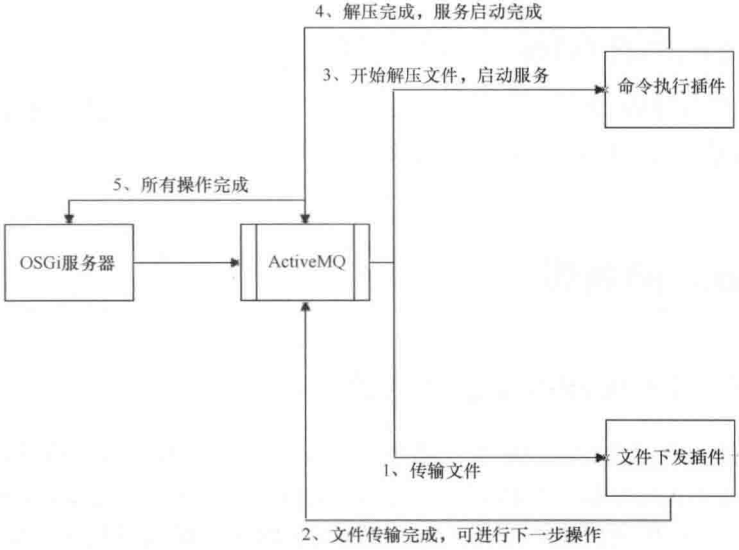


图 8.9　让插件组合具备流程化的功能，组合起来使用

单靠独立的插件是没办法完成类似这样的需求的，虽然我们可以把这样的需求按照步

骤做成一个客户端的插件包，然后每当需要使用这样的功能时就让这个插件包进行操作。但这样一方面会出现大量的代码重复，另一方面是每当操作的步骤有一些变化时，都需要更新插件包。所以基于这个原因，我们设计了 PlayBook 服务端，简单来说，PlayBook 是为了能够按照一定的流程去对服务端插件进行流程化操作而设计的。

PlayBook 作为流程化处理的消息转发端，与常规的消息分发服务端有一定的差异之处。

（1）它是具备流程性的，也就是说 PlayBook 服务端会根据我们指定的流程来完成消息一步一步的转发。

（2）它是具备上下文的，一旦前面的步骤执行完毕，PlayBook 会尝试把前面步骤执行的结果保存下来，当后续的步骤需要使用到前面步骤的结果时，就把结果替换到具体的参数里面去。

（3）它是具备逻辑性的，PlayBook 还需要根据每步执行成功与失败的情况，判断下一步应该执行哪种操作。

8.5.2 PlayBook 设计示意图

PlayBook 作为流程化处理的消息转发端，它需要处理的事情是把消息按照步骤送到具体的队列中，把前面步骤的结果作为参数传递给后续的步骤，判断是否需要继续往下执行，如图 8.10 所示。

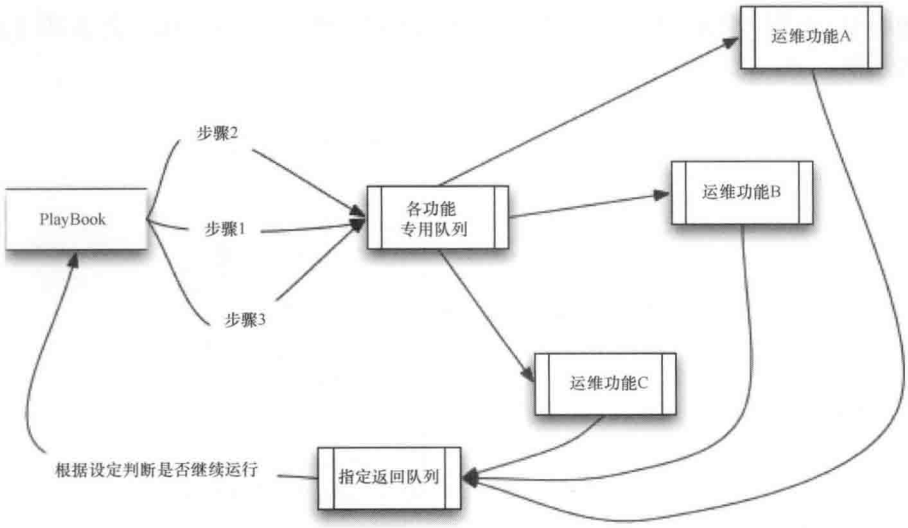


图 8.10 PlayBook 整体流程示意图

下面从 PlayBook 的具体执行流程入手进行分析，看看设计的时候需要留意的要点。

首先，PlayBook 需要接收调用端所发送的消息，前面我们讲过，为了让调用端尽可能地少了解运维框架内部所发生的事情，所以运维框架整体对外只会公开消息分发服务端的调用方法。消息最开始会从消息分发服务端流入，经过一次消息分发转发到 PlayBook 服务端，如图 8.11 所示。

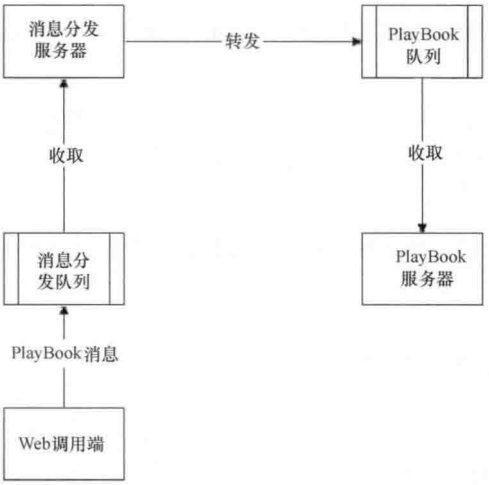


图 8.11　PlayBook 的转发操作

当 PlayBook 服务端插件收到消息后，就要开始对实体进行指定临时会话 ID 的操作了，如图 8.12 所示。

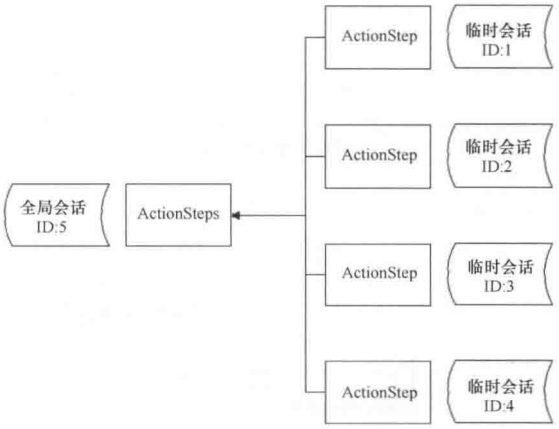


图 8.12　临时会话 ID 与全局会话 ID

这里有两个 ID 需要留意一下，一个是全局会话 ID，另一个是各个 ActionStep 中的临时会话 ID。由于所有 ActionStep 组成的是一个完整的运维动作，这个完整的运维动作在每次进行操作的时候需要把结果反馈给调用端。对于调用端来说，它并不关心 PlayBook 内部究竟是如何进行运维操作执行的，它关心的是整体运维操作的结果。所以全局会话 ID 用于供调用端完成整体运维功能消息的收取，如图 8.13 所示。

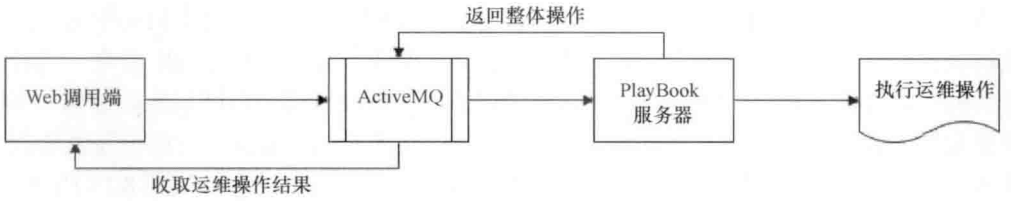


图 8.13　全局会话 ID

而临时会话 ID 则是供 PlayBook 内部使用的，由于每个运维操作都有先后顺序，所以当我们能够从临时会话中收取到消息时，则说明该步骤已经执行完成，可以判断是否需要进行下一步的执行了，如图 8.14 所示。

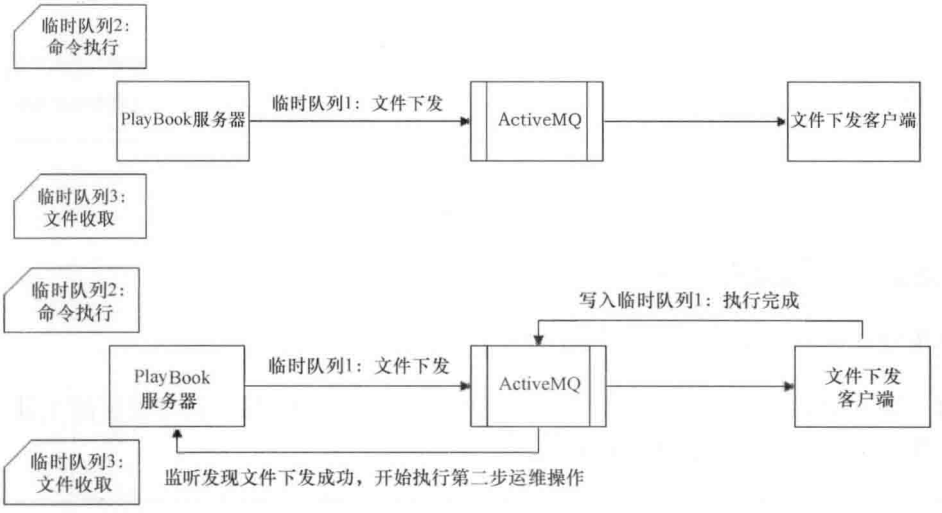


图 8.14　临时会话 ID

最后一个需要解决的问题是如何让整体运维操作具备上下文。当 PlayBook 接收到运维操作之后，需要开启一个线程，每当收到一个临时队列里面的消息之后，就把消息内容暂存到列表中，在消息发送之前，把变量替换到发送的消息体中就可以了。

8.6 结果处理服务端

8.6.1 结果处理服务端设计目的

由于客户端在进行相应操作时，很多时候是需要返回结果的，并且我们期望这些结果能够被持久化。例如，我们使用执行命令的插件在客户端上执行了一句 Shell 命令，或者使用操作数据库的插件执行了一句 SQL。大多数情况下，这些操作都会返回相应的结果。所以在通用框架层，我们可以设计一个结果处理的服务端，将所有需要被持久化的消息发送到这个插件所监听的队列上，让结果处理服务端插件完成结果的持久化过程，如图 8.15 所示。

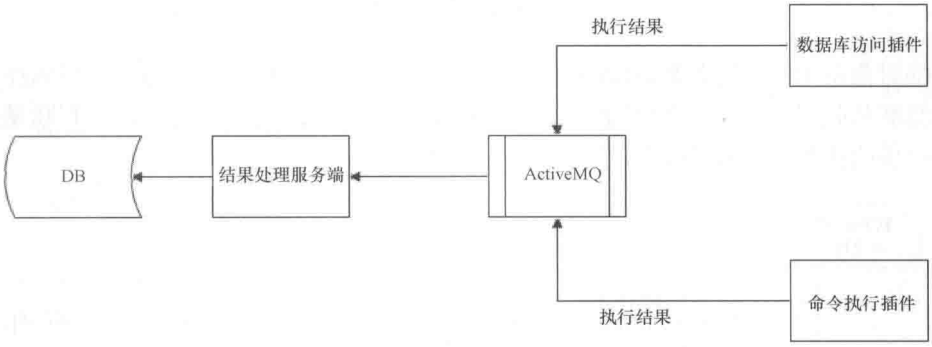


图 8.15　结果处理服务端插件

8.6.2 结果处理服务端处理流程

结果处理服务端插件运作流程如图 8.16 所示。

（1）Web 界面把需要发送的 Shell 命令插入到数据库的表中，获取相应的主键 ID，这时数据库里应该就会有一条这样的记录了：

ID	Command	Result
1	ps -ef|grep mysql	

可以看出，这样做的目的是为了让结果处理服务端插件去更新这条记录而不是新增一条记录。为什么要采取这样的设计方式呢？第一点，从数据存储的角度来看，执行命令的结果不会单独存在，往往会有其他数据表与之关联，所以一开始就把没有执行结果的数据

插入到数据库中是有一定必要性的。第二点，客户端可能出现了无法预知的错误，导致执行结果无法被送回结果处理服务端所侦听的队列中，导致本次执行记录的丢失。所以在设计结果处理服务端的时候，所采用的方式是先把没有执行结果的记录插入到数据库中，然后再让结果处理服务端进行记录的更新。

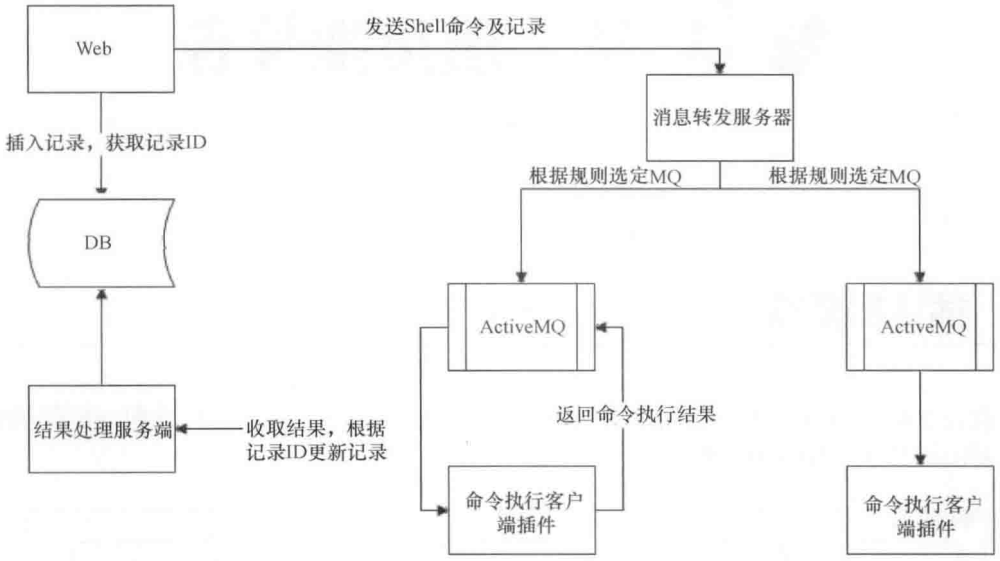


图 8.16　结果处理服务端插件运作流程

（2）数据成功入库之后，我们把该记录的 ID 以及对应的命令交给消息转发服务端，让消息转发服务端帮我们把消息分发到具体的 MQ 队列中。

（3）当消息被分发到指定的 MQ 队列之后，特定的命令执行客户端就会把命令收下，开始执行命令。

（4）命令执行完毕后，客户端插件把结果以及从服务端送来的 ID 一起交给 ActiveMQ 的结果处理服务端队列，由结果处理服务端进行消息的收取。

（5）结果处理客户端收取到消息后，根据记录 ID 去更新对应的记录。

ID	Command	Result
1	ps -ef|grep mysql	<更新此字段>

第 9 章　通用插件包

9.1　插件包概览

在设计插件包的时候，我们把插件包分为三个层次，分别是通用插件包、框架插件包以及功能插件包，如图 9.1 所示。

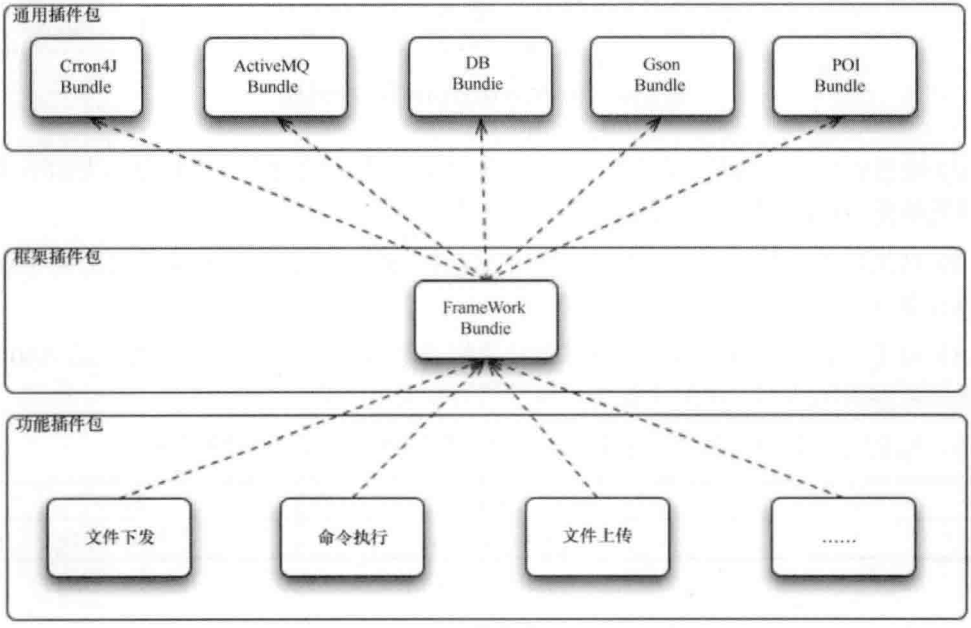


图 9.1　插件包的层次

可以看到我们规划的插件包都是各自比较独立的，首先是通用插件包，通用插件包以第三方库为主，它是后续两个插件包的基石。由于插件包制作有 Import 和 Export 的问题，所以我们选用第三方库的时候都会更倾向于挑选一些依赖很少的第三方库。

在最开始设计通用插件包的时候有两种思路，一种是没有基础框架这个概念，也就是把上一章中讲到的通用功能全部合并到具体的通用插件包和功能插件包里面。这样做的好处是插件包与插件包之间几乎不存在依赖关系，更新插件包时不需要考虑是否会对其他功能产生影响。缺点是大量重复的代码散落在不同的插件包上，一旦基础代码需要调整，那么就要进行全量插件包的更新。

第二种思路是有框架包的做法，但是使用框架包也会带来一个问题，就是一旦框架包没有处理好，就会导致所有的功能插件包功能异常，这时就需要把所有的客户端重新用人工的方式传一遍，后果也是不堪设想的。考虑再三，我们还是可以在测试环境对框架包进行控制的，所以就采用了框架插件包的设计，如图 9.2 所示。

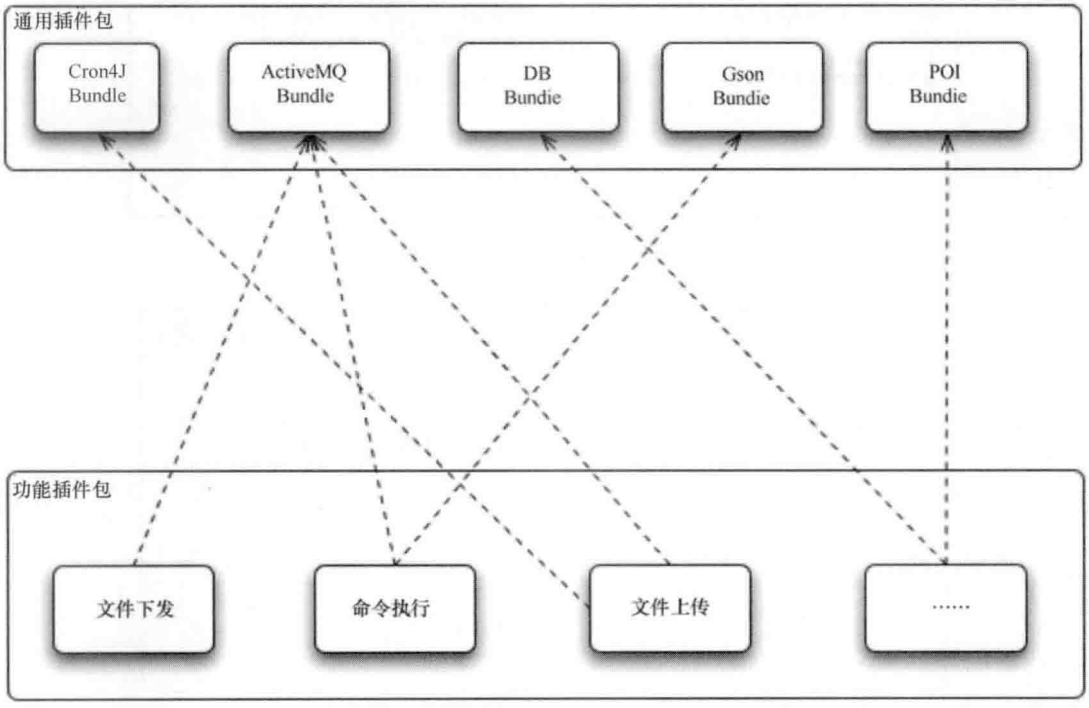


图 9.2　使用 OSGi 的 Import 和 Export 的特性进行插件的设计

定下了三层插件包的结构之后，我们需要选一些经常使用到的第三方库作为通用插件包，和 Apache Karaf 一同部署到客户端。

9.2 作业调度模块——Cron4J

Cron4J 是一个作业调度的第三方包，作为一个运维框架，作业调度的功能是必不可少的，因为我们会碰到许多需要定时调度的情况，比如定时生成巡检报告、定时执行文件的收取、定时的检查数据库的健康状态，等等。作业调度模块会选择 Cron4J 的原因是它没有任何依赖，它是一个很轻量级的作业调度包，却能够很好地满足我们的作业调度的需求，如图 9.3、图 9.4 所示。

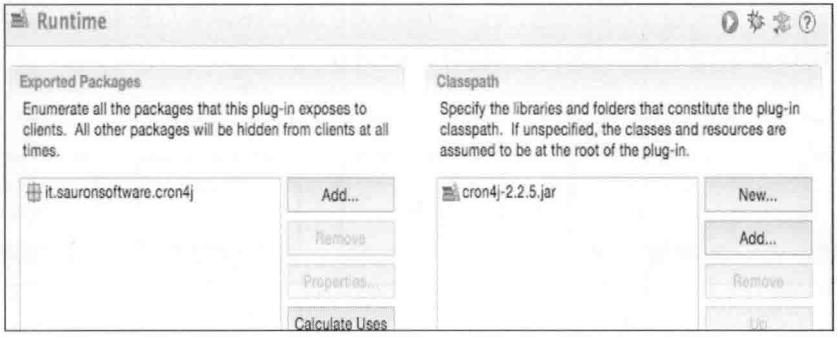


图 9.3　Cron4J 需要 Export 的包

图 9.4　Cron4J 需要 Import 的包

9.2.1 Cron4J 基本使用方式

Cron4J 的使用方式非常简单，我们先看一个例子：

```
import it.sauronsoftware.cron4j.Scheduler;
public class Quickstart {
    public static void main(String[] args) {
        Scheduler s = new Scheduler();
        s.schedule("* * * * *", new Runnable() {
            public void run() {
                System.out.println("Another minute ticked away...");
            }
        });
        s.start();
        try {
            Thread.sleep(1000L * 60L * 10L);
        } catch (InterruptedException e) {
            ;
        }
        s.stop();
    }
}
```

首先创建了一个作业，然后给定了一个作业的调度周期和作业的调度任务，最后当我们不需要使用这个作业时，我们就把它停止。

Cron4J 的使用非常简单，但是它提供了许多有用的功能：

- 可以调度多个任务；
- 可以随时手工调度一个已经被调度了的任务；
- 可以为一个正在运行中的作业重新指定调度周期；
- 可以移除一个作业，即使那个作业正在调度；
- 可以多次调度一个作业；
- 可以从多种数据源处完成作业的调度。

可以看到 Cron4J 这个包“麻雀虽小但五脏俱全”，非常适合作为作业调度用的通用插件包。

9.2.2 作业调度参数

Cron4J 的调度参数是和 UNIX 的 crontab 一样的参数，类似* * * * *这种格式。各个参数所代表的含义如下所述。

- 参数 1：指定每多少分钟调度一次，可选范围从 0 到 59；
- 参数 2：指定每多少小时调度一次，可选范围从 0 到 23；
- 参数 3：指定每月中的那一天调度一次，可选范围从 1 到 31，可以用 L 指定每月的最后一天；
- 参数 4：指定每月中的那一天调度一次，可选范围从 1 到 12，可以用别名代替"jan"、"feb"、"mar"、"apr"、"may"、"jun"、"jul"、"aug"、"sep"、"oct"、"nov" 和 "dec"；.
- 参数 5：指定每周的那一天调度一次，可选范围从 0 到 6，也可以用别名"sun"、"mon"、"tue"、 "wed"、 "thu"、 "fri" 和 "sat"。.

每当分钟数等于 5 的时候调度一次：

```
5 * * * *
```

每分钟调度一次：

```
* * * * *
```

周一的 12 点时，每分钟调度一次：

```
12 * * Mon
```

每月的第 16 天并且这天是周一的 12 点时，每分钟调度一次：

```
12 16 * Mon
```

周一到周五的 11 点 59 分调度一次：

```
59 11 * * 1,2,3,4,5
59 11 * * 1-5
```

每 5 分钟调度一次：

```
*/5 * * * *
```

每小时的 3~18 分这个区间的每 5 分钟调度一次：

```
3-18/5 * * * *
```

每天 9 点到 17 点区间内，每隔 15 分钟调度一次：

```
*/15 9-17 * * *
```

9.2.3 重新调度作业

重新调度作业是作业框架必需的功能，Cron4J 也提供了这样的功能，每当 Cron4J 启动了一个 Schedule 之后，都会生成一个唯一的 ID，当我们需要对这个作业进行重新调度时，我们只需要提供重新调度的 ID 并且给定调度的频率就可以了。

```
package test;
import it.sauronsoftware.cron4j.Scheduler;
public class cron4jtest {
    public static void main(String[] args) {
        Scheduler scheduler = new Scheduler();
        String scheduleID = scheduler.schedule("* * * * *", new Runnable() {
            @Override
            public void run() {
                System.out.println("schedule");
            }
        });
        scheduler.start();
        System.out.println(scheduleID);
        scheduler.reschedule(scheduleID, "*/5 * * * *");
    }
}
```

9.2.4 调度系统进程

Cron4J 可以通过 ProcessTask 类非常容易地调度起一个系统进程：

```
ProcessTask task = new ProcessTask("C:\\Windows\\System32\\notepad.exe");
Scheduler scheduler = new Scheduler();
scheduler.schedule("* * * * *", task);
```

```
scheduler.start();
```

调度进程并传入参数：

```
String[] command = { "C:\\Windows\\System32\\notepad.exe", "C:\\File.txt" };
ProcessTask task = new ProcessTask(command);
// ...
```

也可以传入环境变量：

```
String[] command = { "C:\\tomcat\\bin\\catalina.bat", "start" };
String[] envs = { "CATALINA_HOME=C:\\tomcat", "JAVA_HOME=C:\\jdks\\jdk5" };
ProcessTask task = new ProcessTask(command, envs);
// ...
```

通过使用 Cron4J 的系统进程调度的功能，可以让我们非常容易地在运维系统里面控制中间件、数据库的定时重启，以及故障后的自动重启等工作。

9.3 数据访问模块——MidaoProject

Midao 是一个轻量级的数据访问模块，它的特点也是非常简单实用的，并且依赖包比较少。由于 Midao 是数据访问的模块，在制作 OSGi 插件包的时候我们会把 Midao 的 jar 包和其他数据访问的驱动包会放在一起，作为一个 DBBundle 包预先放置在 Apache Karaf 中，方便我们后续开发插件时对数据库进行访问，如图 9.4 至图 9.6 所示。

图 9.4　把数据库驱动和 Midao 作为一个插件包

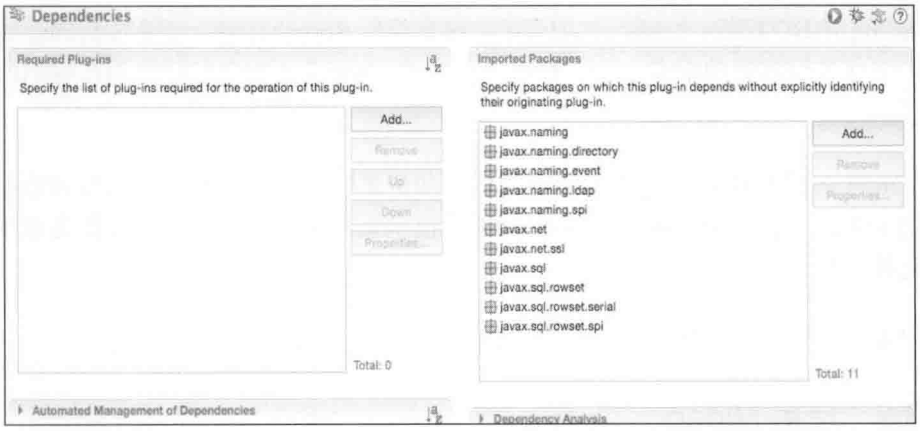


图 9.5　数据访问插件包需要 Import 的包

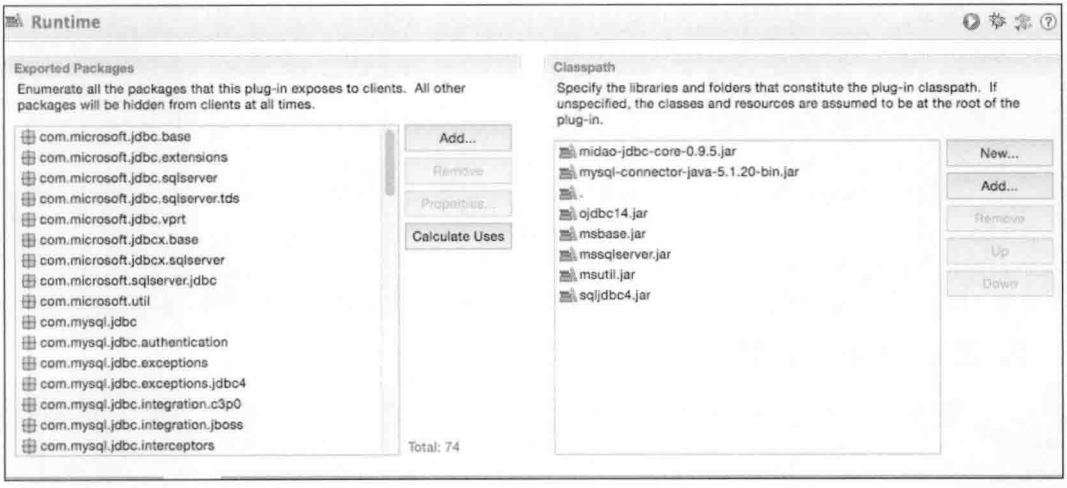


图 9.6　数据访问插件包需要 Export 的包

Midao 虽然是一个轻量级的 JDBC 包，但是它提供的功能却非常实用，并且支持的数据源也非常多，Derby、MySQL、MSSQL、Oracle 等数据库都是支持的。

9.3.1　为什么选择 Midao

有很多开源的 JDBC 数据源访问层的项目可以选择，比较流行的有 Spring JdbcTemplate、Apache DBUtils，以及 Hibernate 这种 ORM 的第三方库。

与 Sping JDBC 相比，Midao 更加简单，没有类似 Spring JDBC 这种对 Spring 容器的依赖，简化了许多数据源的配置。从依赖这一点来看，Spring JDBC 就已经不在我们的选择范围之内了。

与 ORM 技术相比，因为我们开发的不是业务系统，所以我们并不需要对数据库做一次完整的 ORM，并且 Hibernate 这种 ORM 的技术依赖包太多，也不适合作为运维框架的数据源连接来使用。

所以在数据访问这一块，我们的要求是越轻量越好，依赖包越少越好。

9.3.2 使用 Midao

1. 简单数据查询

```
        QueryRunnerService runner = MjdbcFactory.getQueryRunner(this.dataSource);

        Map<String, Object>queryParameters = new HashMap<String, Object>();
        queryParameters.put("id", 1);

        MapInputHandler input = new MapInputHandler("SELECT name FROM students WHERE
id = :id",
        queryParameters);
        Map<String, Object> result = runner.query(input, new MapOutputHandler());
```

2. 使用 Midao 完成数据的插入

```
        String INSERT_STUDENT_TABLE = "INSERT INTO students (name, address)
VALUES ('Not me',
        'unknown')";
        QueryRunnerService runner = MjdbcFactory.getQueryRunner(this.dataSource);
        MapOutputHandler handler = new MapOutputHandler();
        Map<String, Object> result = runner.update(INSERT_STUDENT_TABLE,
handler, new Object[0])
        Map<String, Object> result =
            runner.overrideOnce(MjdbcConstants.OVERRIDE_GENERATED_COLUMN_NAMES,
        new String [] {"ID"}).update(INSERT_STUDENT_TABLE, handler, new Object[0])
```

3. 调用存储过程

```
        QueryRunnerService runner = MjdbcFactory.getQueryRunner(this.dataSource);
```

```
        BeanInputHandler<Student> input = null;
        Student student = new Student();
        student.setId(2);

        input     =     new     BeanInputHandler<Student>("{call     TEST_NAMED
(:id, :name, :address)}", student);


        Student result = runner.call(input);
```

4. 使用 XML 作为数据源

```
        String xmlContent = "<?xml version=\"1.0\"?>" +
           "<root>" +
              "<query id='findStudent' outputHandler='MapOutputHandler'>" +
                 "SELECT name FROM students WHERE id = #{id,jdbcType=INTEGER,
mode=in}" +
              "</query>" +
           "</root>";

        // xml should be added to Repository before it can be executed
        XmlRepositoryFactory.addAll(
        XmlRepositoryFactory.getDocument(new ByteArrayInputStream(
        xmlContent.getBytes()
           )
        ));

        XmlInputOutputHandler handler = new XmlInputOutputHandler("findStudent", 1);
        // XML query execution
        Map<String, Object> student = (Map<String, Object>) runner.execute(handler);
```

9.4 序列化模块——Gson

其实 ActiveMQ 所发送的消息体已经足够方便了，绝大部分的消息我们都可以用 MapMessage 和 TextMessage 来完成传输，大文件我们也可以通过 BlobMessaege 进行传输，这样看来，作为序列化反序列化的 Gson 库应该派不上什么用场，但我们的运维框架有一个

PlayBook 的功能，它是前端用于告诉运维服务端如何让运维客户端运作的一个服务端插件包。由于传输的类型是复杂属性，并且需要和前端结合起来使用，所以这时我们就更倾向于定义实体，然后通过序列化和反序列化的技术将实体在前端和服务端之间进行传输。

1. 实体对象序列化传输

```
public class Student {
privateint id;
private String name;
private Date birthDay;

publicintgetId() {
return id;
    }

public void setId(int id) {
this.id = id;
    }

public String getName() {
return name;
    }

public void setName(String name) {
this.name = name;
    }

public Date getBirthDay() {
returnbirthDay;
    }

public void setBirthDay(Date birthDay) {
this.birthDay = birthDay;
    }
}

importjava.util.ArrayList;
importjava.util.Date;
importjava.util.List;
```

```
importcom.google.gson.Gson;
importcom.google.gson.reflect.TypeToken;

public class GsonTest1 {

public static void main(String[] args) {
Gsongson = new Gson();

Student student1 = new Student();
student1.setId(1);
student1.setName("Demo");
student1.setBirthDay(new Date());

//序列化操作
String s1 = gson.toJson(student1);
System.out.println("简单Bean转化为Json===" + s1);

//反序列化操作
Student student = gson.fromJson(s1, Student.class);
System.out.println("Json转为简单Bean===" + student);


Student student2 = new Student();
student2.setId(2);
student2.setName("Demo2");
student2.setBirthDay(new Date());

Student student3 = new Student();
student3.setId(3);
student3.setName("Demo3");
student3.setBirthDay(new Date());

List<Student> list = new ArrayList<Student>();
list.add(student1);
list.add(student2);
list.add(student3);

//泛型序列化
String s2 = gson.toJson(list);
```

```
//泛型反序列化
List<Student>retList = gson.fromJson(s2,
newTypeToken<List<Student>>() {
}.getType());
    }
}
```

当我们只需要导出部分实体的时候，就可以使用注解来进行控制。例如，界面的实体由于有自己特殊的要求，和服务器端的实体有出入，但是这部分差异的实体信息并不需要被传输，这时就可以使用下面介绍的方法了。

2. 使用 Gson 注解

```
importjava.util.Date;

importcom.google.gson.annotations.Expose;
importcom.google.gson.annotations.SerializedName;

public class Student {
privateint id;
  .
@Expose
private String name;

@Expose
@SerializedName("bir")
private Date birthDay;

publicintgetId() {
return id;
    }

public void setId(int id) {
this.id = id;
    }

public String getName() {
return name;
    }

public void setName(String name) {
this.name = name;
    }
```

```
        public Date getBirthDay() {
        returnbirthDay;
            }

        public void setBirthDay(Date birthDay) {
        this.birthDay = birthDay;
            }
        }

        importjava.util.ArrayList;
        importjava.util.Date;
        importjava.util.List;

        importcom.google.gson.FieldNamingPolicy;
        importcom.google.gson.Gson;
        importcom.google.gson.GsonBuilder;
        importcom.google.gson.reflect.TypeToken;

        public class GsonTest2 {

        public static void main(String[] args) {
        Gsongson = new GsonBuilder()
                .excludeFieldsWithoutExposeAnnotation() //不导出实体中没有用
@Expose注解的属性
                .enableComplexMapKeySerialization() //支持Map的key为复杂对象的形式
                .serializeNulls().setDateFormat("yyyy-MM-ddHH:mm:ss:SSS")// 时
间转化为特定格式
                .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE)//会把
字段首字母大写
        注：对于实体上使用了@SerializedName注解的不会生效
                .setPrettyPrinting() //对JSON结果格式化
                .setVersion(1.0)    //有的字段不是一开始就有的，会随着版本的升级而添加
进来，那么在进行序列化和返序列化的时候就会根据版本号来选择是否要序列化
                                    //@Since(版本号)能完美地实现这个功能，还有的字段可能
随着版本的升级而删除
                //@Until(版本号)也能实现这个功能，GsonBuilder.setVersion(double)
方法需要调用
        .create();

        Student student1 = new Student();
        student1.setId(1);
        student1.setName("Demo1");
```

```
        student1.setBirthDay(new Date());

        //实体序列化
        String s1 = gson.toJson(student1);

        //实体反序列化
        Student student = gson.fromJson(s1, Student.class);

        Student student2 = new Student();
        student2.setId(2);
        student2.setName("Demo2");
        student2.setBirthDay(new Date());

        Student student3 = new Student();
        student3.setId(3);
        student3.setName("Demo3");
        student3.setBirthDay(new Date());

        List<Student> list = new ArrayList<Student>();
        list.add(student1);
        list.add(student2);
        list.add(student3);

        String s2 = gson.toJson(list);
        List<Student>retList = gson.fromJson(s2,
        newTypeToken<List<Student>>() {
        }.getType());
            }
        }
```

9.5 交互式命令执行模块——JavaExpect

最后要介绍的一个常用的模块就是 JavaExpect 了，ExpectShell 有 Expect，Python 有 PExpect，它们在交互式的操作上都做得非常好，但是 Java 的 Expect 就没有那么好了。先说我们为什么会使用 JavaExpect 这个东西，我们当时做功能的时候，碰到这样一个需求，这个需求说起来也简单，由于我们运维监控所用的账号是普通账号，管理 Oracle 的账号是普通账号，重启主机所需要用到的账号是 root 账号，这样就会碰到切换账号的问题，如图 9.7 所示。

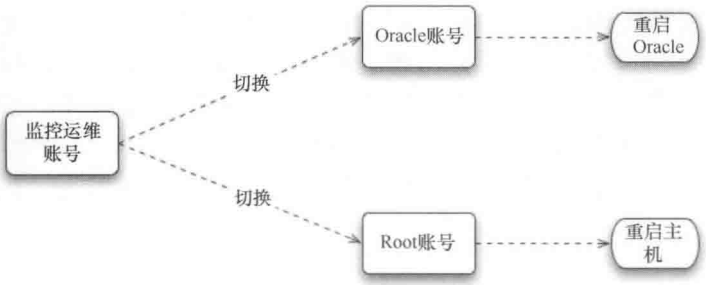


图 9.7 账号切换

而在 Linux 或者 UNIX 操作系统上进行账号的切换，就一定会碰到交互式输入密码的问题，所以我们需要用到 Expect 的功能。

但是 JavaExpect 还是比较烦人的，我们碰到最棘手的问题就是本地切换账号，特别是在本地切换 root 账号的时候，总是无法切换过去，命令非常简单，就执行了一下“su– root”这样的命令，但是 JDK 却报错了。这时我们开始怀疑，难道 Java 有安全沙箱的限制，不让我们进行本地账号的切换？正当我们一筹莫展的时候，突然想到了一个方法，本地账号不能切换，那我们通过本地 SSH 能不能进行切换呢？虽然这些设备之间不能进行 SSH 的跳转，但是它们本机的 SSH 服务还是有开启的。然后，我们尝试了 JavaExpect 这个第三方包。这个第三方包实现思路很清晰，它通过调用 JSCH 的 SSH 功能，再通过 SSH 执行命令，然后我们发现使用 SSH 连接上主机之后，就可以进行账号的切换了，如图 9.8 所示。

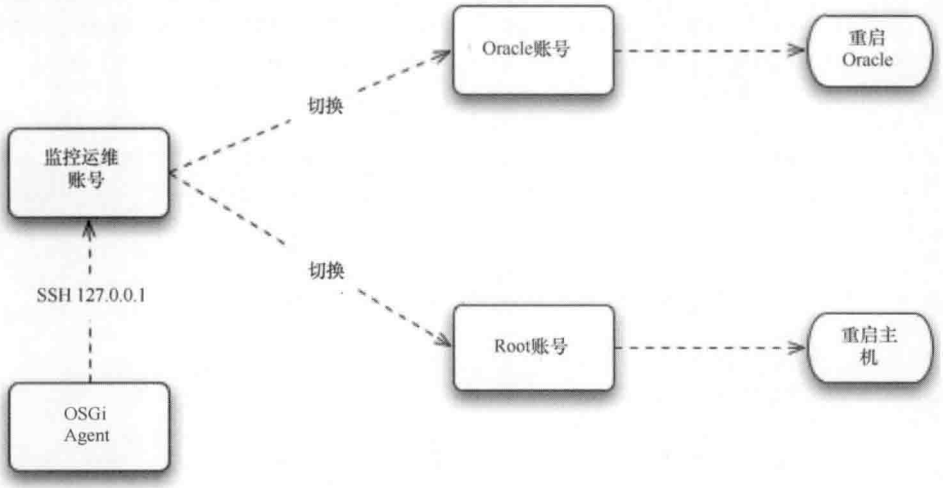


图 9.8 通过 SSH 进行账号切换

也就是说，虽然我们没办法用本地账号执行账号切换的命令，但是我们可以用本地进程 SSH 到本机的运维监控账号，然后再通过 SSH 发送命令结合 Expect 的方式进行账号的切换。而 Expect 这步是怎么实现的呢？我们可以通过不断读取 JSCH 返回的字符，然后用正则表达式匹配，例如账号切换的时候，一旦碰到“*assword:”，我们就认为下一个环境我们需要输入密码了，然后就开始发送账号的密码，完成账号的切换。下面看一下这个包的关键实现部分。

整个方法里面，最值得我们学习的是名字为_expect 的这个方法，这个方法先定义了我们需要正则匹配的内容，然后不断地读取输入的字符，假如到达了超时的时间依然没有一个是能匹配到的，那么就抛出一个超时的异常，假如匹配到了内容，则返回一个匹配结果的实体。当使用_expect 这个方法的上层方法发现能够从输出流中匹配到我们期望的信息之后，就可以使用 JSCH 的功能进行命令的发送，完成命令行交互的过程了。

```
packagecom.nemo.javaexpect.shell;

importjava.io.IOException;
importjava.io.InputStream;
importjava.util.regex.Matcher;
importjava.util.regex.Pattern;

importcom.nemo.javaexpect.shell.driver.Connection;
importcom.nemo.javaexpect.shell.exception.CommandTimeoutException;
importcom.nemo.javaexpect.shell.exception.ConnectionException;
importcom.nemo.javaexpect.shell.exception.MatchFailedException;
importcom.nemo.javaexpect.shell.logger.DefaultShellLogger;
importcom.nemo.javaexpect.shell.logger.ShellLogger;

public abstract class DefaultShell implements Shell, ShellLogable {
    public final static String SOURCE_PACKAGE_PARENT =
"com.nemo.javaexpect";

    private String lastCommand;

    /**
     * 用于做 expect 操作的流大小
     */
    private byte[] b = new byte[4096];

    /**
```

```
 * expect 超时时间
 */
private final int EXPECT_SLEEP_TIME_WHEN_NOT_FOUND = 10;

privateStringBuffer buff = new StringBuffer();
private Connection driver;
private String waitForPrompt;
privateintcommandTimeout;

publicDefaultShell(Connection driver, String waitForPrompt,
        intcommandTimeout) {
    this.driver = driver;
    this.waitForPrompt = waitForPrompt;
    this.commandTimeout = commandTimeout;
}

private void _close() {
    driver.close();
}

@Override
public void close() {
    log("close()", "\nConnection Closed!!!");
    _close();
}

private void _send(String command) {
    if (command == null)
        command = "";
    try {
        driver.getOutputStream().write(command.getBytes());
        driver.getOutputStream().write('\r');
        driver.getOutputStream().flush();
    } catch (IOException e) {
        throw new ConnectionException(e.getMessage());
    }
}

@Override
public void send(String command) {
    log("send(\"" + command + "\")", "");
```

```
            _send(command);
        }

        @Override
        publicCommandResult execute(String command, String waitForPattern) {
            lastCommand = "execute\"(" + command + "\",\"" + waitForPattern
+ "\")";
            return execute(command, waitForPattern, commandTimeout);
        }

        privateCommandResult     _execute(String     command,     String
waitForPattern,
                int timeout) {
            _send(command);
            return _expect(waitForPattern, timeout);
        }

        @Override
        publicCommandResult execute(String command) {
            lastCommand = "execute(\"" + command + "\")";
            return _execute(command, waitForPrompt, commandTimeout);
        }

        @Override
        publicCommandResult expect(String waitForPattern) {
            lastCommand = "expect(\"" + waitForPattern + "\")";
            return _expect(waitForPattern, commandTimeout);
        }

        privateCommandResult _expect(String waitForPattern, int timeout) {

            long start = System.currentTimeMillis();
            long end = timeout * 1000 + start;

            Pattern p = Pattern.compile(waitForPattern);
            while (true) {
                InputStream in = driver.getInputStream();
                try {

                    while (in != null &&in.available() > 0) {
                        int length = in.read(b);
```

```
                        buff.append(new String(b, 0, length));
                    }
                } catch (IOException e) {

                }
                Matcher m = p.matcher(buff);
                if (m.find()) {
                    int length = m.end();
                    String tmp = buff.substring(0, length);
                    log(lastCommand, tmp);
                    CommandResult r = new DefaultCommandResult(lastCommand,
tmp);
                    buff.delete(0, length);
                    return r;
                }
                if (System.currentTimeMillis() > end) {
                    String tmp = buff.toString();
                    log(lastCommand, tmp);
                    throw   new   CommandTimeoutException("Expect:   "   +
waitForPattern
                            + "\nCurrent buffer: " + tmp.toString());
                } else {
                    try {
                        Thread.sleep(EXPECT_SLEEP_TIME_WHEN_NOT_FOUND);
                    } catch (InterruptedException e) {
                    }
                }
            }
        }

        @Override
        publicCommandResult execute(String command, String waitForPattern,
                int timeout) {
            lastCommand = "execute(\"" + command + "\",\"" + waitForPattern
+ "\","
                    + String.valueOf(timeout) + ")";
            return _execute(command, waitForPattern, timeout);
        }

        @Override
        publicCommandResult execute(String command, int timeout) {
```

```
        lastCommand = "execute(\"" + command + "\"," +
String.valueOf(timeout)
                + ")";
        return _execute(command, waitForPrompt, timeout);
    }

    @Override
    publicCommandResult expect(String waitForPattern, int timeout) {
        lastCommand = "expect(\"" + waitForPattern + "\","
                + String.valueOf(timeout) + ")";
        return _expect(waitForPattern, timeout);
    }

    privateCommandResult _getLastExitCode() {
        _send("echo XYZ$?ZYX");
        return _expect("XYZ[0-9]*ZYX", commandTimeout);
    }

    @Override
    publicCommandResultgetLastExitCode() {
        lastCommand = "\"getLastExitCode()\"";

        CommandResult r = _getLastExitCode();
        Pattern pattern = Pattern.compile("XYZ($?)ZYX");
        Matcher matcher = pattern.matcher(r.getCommandResult());
        if (matcher.find()) {
            int ret = Integer.parseInt(matcher.group(1));
            return new DefaultCommandResult("getLastExitCode()", ret);
        } else {
            throw new MatchFailedException(
                    "Expect the pattern XYZ($?)ZYX, but actual value is "
                            + r.getCommandResult());
        }
    }

    public static String getStackTrace() {
        Throwable e = new Throwable();
        for (StackTraceElement el : e.getStackTrace()) {
            String str = el.toString();
            if (!str.contains(SOURCE_PACKAGE_PARENT))
```

```
                returnstr;
            }
            return "";

        }

        @Override
        public void log(String command, String result) {
            ShellLogger logger = this.getLogger();
            if (logger != null) {
                String id = this.getShellId();
                String trace = getStackTrace();
                logger.log(trace, id, command, result);
            }
        }
    }
```

9.6 小结

通用插件包其实可以看作常规开发过程中的第三方基础包，只是由于在 OSGi 容器内一些模块都插件化了，所以才把各个第三方包提取出来作为独立的通用插件包而存在罢了。通用插件包的选项十分重要，同样的功能，A 框架和 B 框架虽然都可以实现，但是这个基础包所需要的依赖包对选型起到了十分重要的判断作用。还记得当时需要实现一个出巡检报告的功能，最终是需要生成一份 Word 文档的。当时选择了 Apache POI 这个框架进行实现，写功能实现代码的时间并不长，但是制作 POI 这个通用插件包却花费了相当多的时间。POI 这个插件包需要非常多的 XML 的基础包，每次制作完之后，放到 OSGi 容器里面一运行就抛出“在容器内找不到 xxx 依赖包”这样的错误，折腾了好一阵子才把这个 POI 的通用插件包做好。所以在选择第三方插件包制作通用插件包的时候，有这样一个建议，功能相差不太大的时候，尽量优先选择依赖包少的框架包。

第10章 常用插件

基于前面所设计的框架插件包和通用插件包，我们可以完成许多运维操作。下面以文件下发、文件抓取、命令执行以及操作系统的目录结构获取这四个最常用的功能为例来讲解常用插件。当需要更多运维插件的时候，可以举一反三制作出符合实际情况的插件来进行使用。

10.1 文件下发插件

10.1.1 文件下发插件设计

集中化的资料下发是运维过程中很常用的功能之一，文件下发功能主要用于把文件批量下发到指定的目录下。常用于对主机配置文件进行批量化的更新，巡检脚本的集中化下发，为后续的巡检功能做准备，等等。

文件下发功能的制作相对简单，首先调用端会把需要下发的文件的路径发送到服务端的消息分发插件上，让消息分发插件进行消息的分发，送到指定的 ActiveMQ 的指定队列上，然后客户端把消息和文件收下来，写到指定的位置上，如图 10.1 所示。

1. 文件下发功能处理流程

（1）由 Web 页面提供文件路径、文件名称、下发目标路径、下发目标 IP、下发目标所属二级代理等信息，发送到服务器端的消息分发插件中。

（2）消息分发插件根据接收到的消息，初始化指定 ActiveMQ 的连接，然后根据“功能名称_IP”的规则，将消息发送到该二级代理指定的队列中。

（3）文件下发客户端侦听到队列上有消息，把消息从队列上收取下来，开始做写入文件前的校验，包括目录是否可写、给定的路径是否存在等。

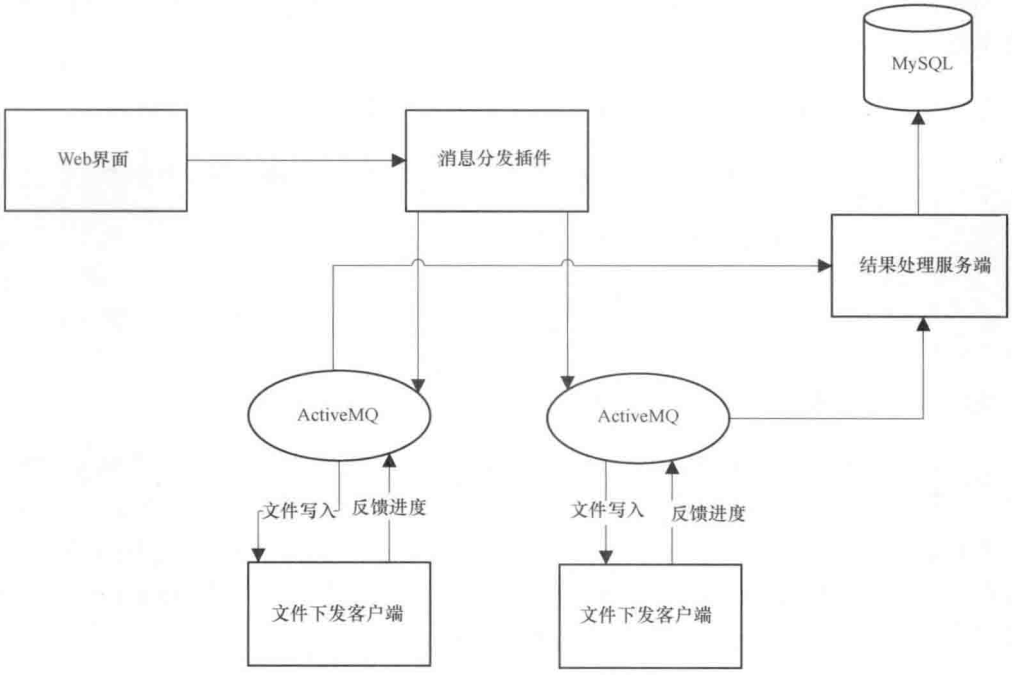


图 10.1　文件下发插件整体运作流程

（4）完成文件写入前校验之后，我们就可以开始对文件进行写入了。写入完成之后，就可以把信息返回到通用入库消息端，由通用层的结果入库服务端进行结果的入库。

2．设计时的注意点

（1）需要校验文件能否写入，磁盘空间是否足够。

（2）要区分不同的操作系统，Windows 操作系统下存在下发的文件没有权限的问题，而在 Linux 操作系统上，下发的文件是有权限这一说法的，所以需要在程序上做一些处理。

10.1.2　使用 Apache Common IO

文件下发的过程其实就是文件写入的过程，ActiveMQ 已经为传输的过程提供了保障，我们不需要过多地关心传输过程中会出现的问题，我们需要处理的事情就是文件的写入了。常规的 Java 代码实现文件写入的方式相信读者都不陌生，但是对于这种常规的操作，会不会有一些现成的第三方库能够为我们提供帮助，减少代码编写过程中出现的错误呢？这时，

我们就可以考虑使用 Apache Common IO。

Apache Common 项目里面有许多优秀通用包，Common IO 就是其中一个用于处理 I/O 的通用包。

在不使用 Common IO 的时候，我们读取文件的代码如下：

```
File file = new File("E:\\pratics\\demo-mybatis\\db.properties");
InputStream in = null;
in = new FileInputStream(file);
byte[] bytes = new byte[in.available()];
in.read(bytes);
in.close();
```

使用 Common IO 之后，读取文件的代码得到了大幅度的简化：

```
File file = new File("E:\\pratics\\demo-mybatis\\db.properties");
byte[] bytes= FileUtils.readFileToByteArray(file);
```

并且 CommonIO 为我们做了更多的逻辑判断，为 I/O 的操作提供了更多的保障，我们可以看一下读取文件时 Apache Common IO 使用的几个主要方法，看看 Apache Common IO 都做了哪些操作。

```
public static byte[] readFileToByteArray(File file) throws IOException {
        FileInputStream in = null;

        byte[] var2;
        try {
            in = openInputStream(file);
            var2 = IOUtils.toByteArray(in, file.length());
        } finally {
            IOUtils.closeQuietly(in);
        }

        return var2;
    }

public static FileInputStream openInputStream(File file) throws IOException {
        if(file.exists()) {
            if(file.isDirectory()) {
                throw new IOException("File \'" + file + "\' exists but
is a directory");
            } else if(!file.canRead()) {
                throw new IOException("File \'" + file + "\' cannot be read");
```

```
                } else {
                    return new FileInputStream(file);
                }
            } else {
                throw new FileNotFoundException("File \'" + file + "\' does
not exist");
            }
        }

        public static byte[] toByteArray(InputStream input, long size)
throws IOException {
            if(size > 2147483647L) {
                throw new IllegalArgumentException("Size cannot be greater than
Integer max value: " + size);
            } else {
                return toByteArray(input, (int)size);
            }
        }

        public static byte[] toByteArray(InputStream input, int size) throws
IOException {
            if(size < 0) {
                throw new IllegalArgumentException("Size must be equal or
greater than zero: " + size);
            } else if(size == 0) {
                return new byte[0];
            } else {
                byte[] data = new byte[size];

                int offset;
                int readed;
                for(offset = 0; offset < size && (readed = input.read(data,
offset, size - offset)) != -1; offset += readed) {
                    ;
                }

                if(offset != size) {
                    throw new IOException("Unexpected readed size. current:
" + offset + ", excepted: " + size);
                } else {
                    return data;
                }
            }
        }
```

可以看到，在读取文件的时候，Apache Common IO 做了很多的判断和检查，它不仅在代码上为我们做了使用上的简化，而且还在代码的执行过程中为我们提供了许多保障。

10.2 文件抓取插件

10.2.1 文件抓取插件整体设计

文件抓取也是一个比较常用的运维功能，它主要用于完成文件从客户机到服务器之间的传输。常见的需求是把日志文件或者配置文件收取回服务器，又或者是文件变更情况，例如进行文件的采集，也就是说当配置文件发生变更之后才把文件收回服务器。

但是对于一些规模比较大的环境来说，可能还会出现这样一种情况，仅把文件传输到二级代理服务器上，当需要的时候才把文件传输到服务器上，避免带宽被过度使用。

文件抓取整体设计流程如图 10.2 所示。

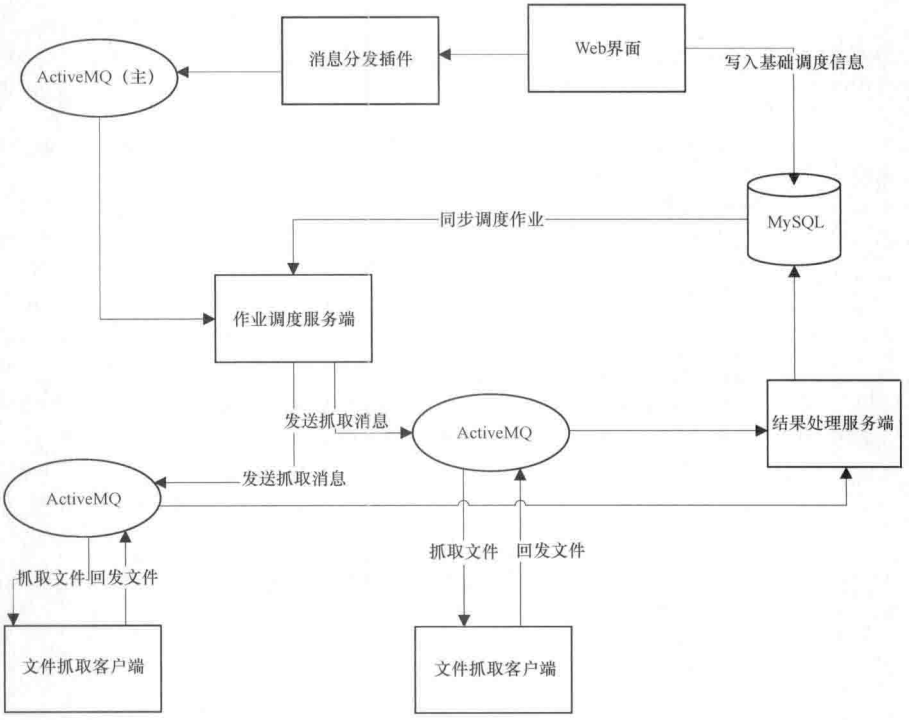


图 10.2　文件抓取整体设计流程

文件抓取插件的设计涉及文件抓取的启动、文件抓取频率的调度、文件抓取的停止三个主要功能点。

1. 文件抓取启动设计流程

（1）Web 界面根据选定的采集方式（周期采集/差异采集）、采集频率、采集路径、主机 IP、二级代理 IP 等信息把消息组装起来，然后把消息传输给消息分发服务端并对作业调度的消息进行持久化。

（2）消息分发服务端根据消息将指定的内容传输到指定的 ActiveMQ 上。

（3）作业调度服务端收取作业调度的消息，启动调度作业，开始周期性地向指定队列发送文件抓取的消息。

（4）判断抓取的类型，若为周期性抓取，则每到指定的周期就开始对文件进行抓取，传送到 ActiveMQ 的队列上。若为文件对比模式的抓取，则在传输文件到 ActiveMQ 到队列之前，先对指定文件进行 md5 的校验，若文件存在差异，才把文件传输到 ActiveMQ 服务器中。

（5）客户端根据队列上指定的消息对内容进行收取，然后根据消息的内容对目标文件开始读取并把文件发送到文件抓取服务端所侦听的消息队列上。

（6）为了方便后续对调度作业的管理，把启动后的作业 ID 传递给 ActiveMQ，交给结果入库服务端进行作业 ID 的入库。

2. 文件抓取重新调度设计流程

（1）Web 界面根据选定的采集方式（周期采集/差异采集）、采集频率、采集路径、主机 IP、二级代理 IP 等信息对消息进行组装，把消息传输给消息分发服务端并对作业调度的消息进行持久化，给定作业调度的类型为重新调度。

（2）消息分发服务端根据消息将指定的内容传输到指定的 ActiveMQ 上。

（3）作业调度服务端收取作业重新调度的消息，从作业列表中获取该作业的实例，进行作业的重新调度。

（4）更新数据库中的调度周期。

3. 文件抓取停止调度设计流程

（1）Web 界面根据选定的采集方式（周期采集/差异采集）、采集频率、采集路径、主机 IP、二级代理 IP 等信息对消息进行组装，把消息传输给消息分发服务端并且对作业调度的消息进行持久化，并给定作业调度的类型为停止调度。

（2）消息分发服务端根据消息将指定的内容传输到指定的 ActiveMQ 上。

（3）作业调度服务端收取作业重新调度的消息，从作业列表中获取该作业的实例，进行作业的停止调度。

（4）更新数据库中的调度周期。

10.2.2 文件抓取插件设计要点

第一个问题是如何处理文件抓取后的存放位置。

由于在 OSGi Server 中，插件是相对独立的，所以我们可以通过部署层面解决文件存放的问题。假如我们需要把所有抓取到的文件都存放到服务器中，那么我们只需要把插件都放到服务器上就可以了，如图 10.3 所示。

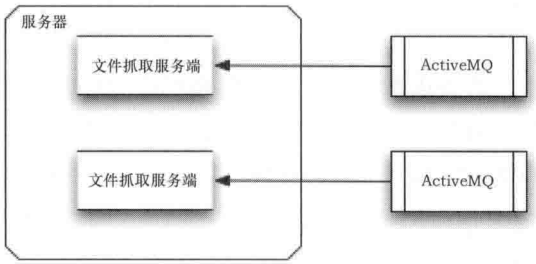


图 10.3　文件抓取后存放位置的问题

当碰到二级代理的情况，我们只需要把特定二级代理的文件抓取服务端插件部署到二级代理上面，就可以让文件都收取到二级代理服务器上面了，如图 10.4 所示。

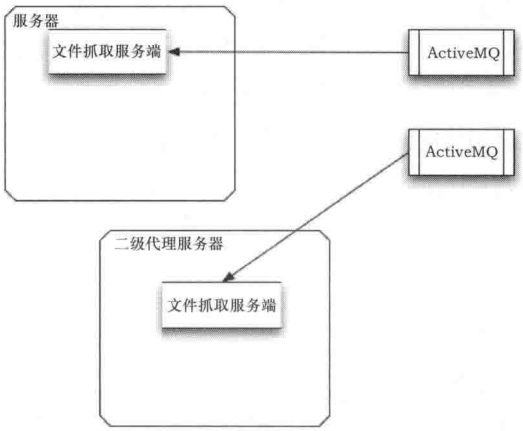


图 10.4　让文件存放到二级代理上

第二个问题是如何判断文件抓取作业已经成功操作了（启动、暂停、重新调度），因为不可以认为在 Web 页面上把消息发送过去，作业就一定操作成功了。作业调度服务端可能会因为异常而没有成功对作业进行调度，所以在判断作业成功启动方面要交给作业调度服务端来进行判断，当真正成功启动了之后，作业调度服务端需要向通用消息处理服务端发送抓取成

功的消息，让通用结果入库端进行作业 ID 的入库操作，这样才算真正完成了作业的启动。

10.3 命令执行插件

能够批量对主机进行具体命令的操作是集中化运维的基础，我们经常会希望对主机进行集中化的命令操作，由于 Java 在 Linux 上执行命令的方式和在 Windows 上执行命令的方式不同，所以我们在执行的时候需要区别对待。

命令执行插件的设计流程如图 10.5 所示。

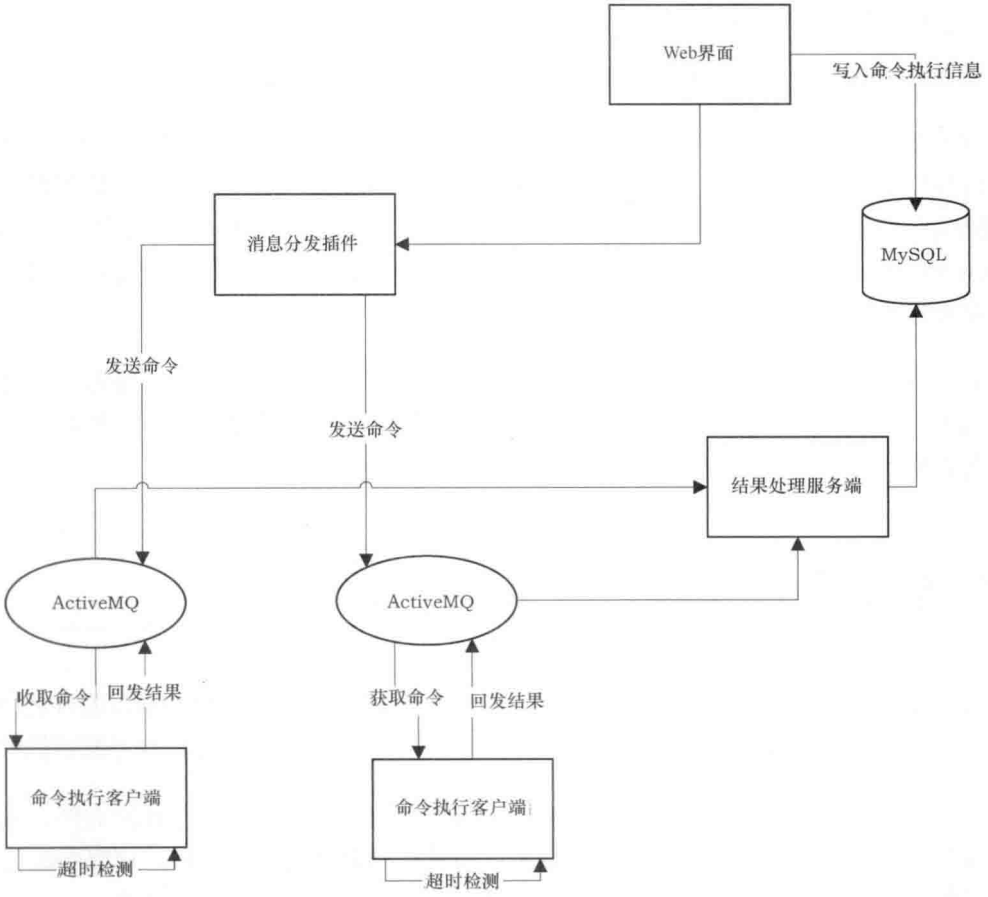


图 10.5 命令执行插件设计流程

1. 命令执行客户端运行流程

（1）Web 界面根据选定的主机 IP、二级代理 IP、需要执行的命令等信息把消息进行组装，然后把消息传输给消息分发服务端。

（2）消息分发服务端根据消息将指定的内容传输到指定的 ActiveMQ 上，传输给指定的命令执行客户端所侦听的队列。

（3）客户端根据队列上指定的消息对内容进行收取，然后通过命令执行客户端对命令进行执行。

（4）命令执行客户端把执行的结果回发到结果处理服务端队列，由结果处理服务端进行结果的持久化处理。

2. 命令执行客户端的设计要点

在命令执行插件设计时有以下几点需要注意。

首先要区分 Windows 和 Linux 操作系统执行命令的方式。这里我们可以通过 Karaf 的命令进行判断，假如客户机是 Linux 操作系统，我们直接执行命令就可以，假如客户机是 Windows 操作系统，就使用 cmd /c 命令进行执行。当然，在 Windows 上还会碰到 vbscript 和 powershell 类型的脚本，这时我们需要根据消息协议中的命令类型的设定，来判断究竟使用哪种技术进行脚本的处理。

然后要具备命令执行的超时机制，一旦客户端没有执行超时的机制，很容易就出现运维目标主机 CPU 使用率异常升高的问题。笔者曾经碰到非常有意思的问题，就是执行了一个不会退出的命令，导致客户机的 CPU 使用率异常升高了。

10.4 目录结构查询插件

目录结果查询插件主要用于获取客户端的系统目录结构。例如客户端发出的目录获取路径为“/”，则获取“/”目录下的所有目录名称——因为我们需要对文件进行下发和抓取。

目录结构获取插件算不上是一个实际的运维功能，它只是一个辅助性的插件。通过使用目录结果来获取插件，我们可以在前端实时地获取到具体的目录结构。让前端进行实时目录结构的展示，可以很方便地对文件抓取、文件下发这样的功能进行辅助。

1. 目录结构获取插件设计思路

（1）Web 界面根据选定的主机 IP、二级代理 IP、需要获取的目录等信息把消息组装成消息实体，然后把消息传输给消息分发服务端。

（2）消息分发服务端根据消息将指定的内容传输到指定的 ActiveMQ 上，传输给指定的客户端。

（3）客户端根据给定的路径开始获取路径，把路径序列化为 JSON 数据，返回到调用端指定的队列上。

（4）Web 端收取到消息之后，对 JSON 数据进行解析，绑定在界面上。

（5）用户根据自己的需求进行目录的选取或者对指定目录的子目录进行获取。

设计流程图如图 10.6 所示。

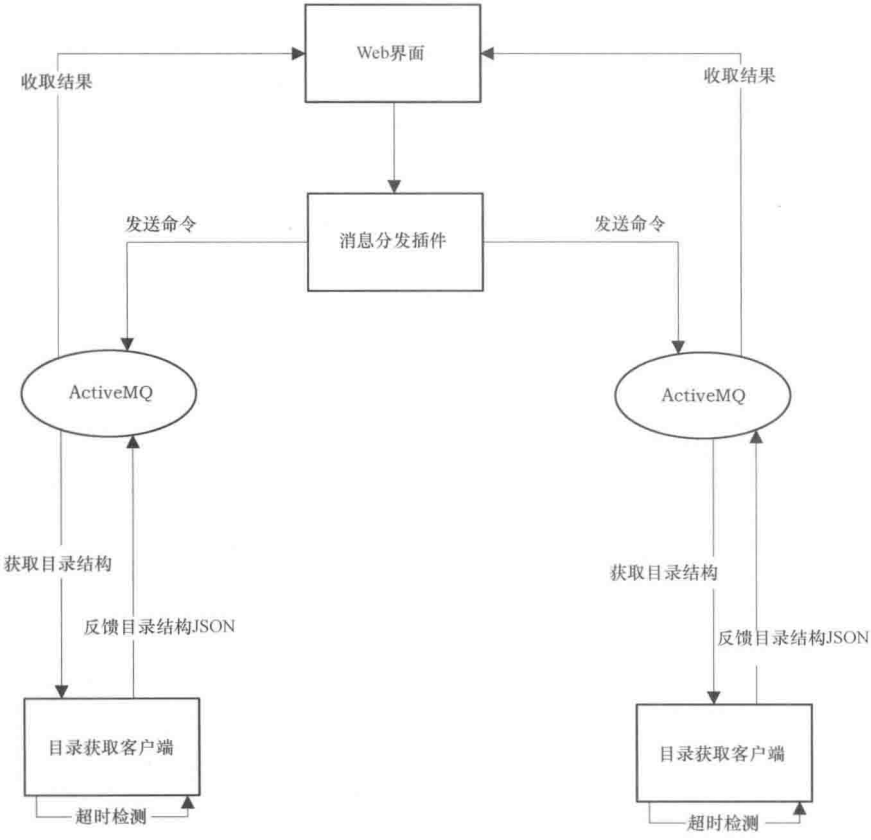


图 10.6　目录结构获取客户端插件设计

2. 目录结构获取插件设计的注意点

在设计目录结构获取插件时，首先，需要注意的是调用端需要有超时的机制，调用端在超时后（如客户端 30 秒仍未返回结果），应该给用户友好的提示，说明本次调用已经超时。

然后，目录获取插件是需要对消息进行窜行收取的，也就是第一个消息收取完之后，才允许第二次消息的收取。否则当调用端进行多次连续的调度时，会造成客户端同时获取设备的目录结构的情况发生，导致设备的 CPU 使用率异常提高。

第11章 整合 Zabbix

11.1 编译安装 Zabbix

Zabbix 是一个非常优秀的开源监控系统，相信读者对 Zabbix 都已经有一定的了解了。安装 Zabbix 有许多种方式，Zabbix Server 可以安装在 Linux 上，Windows 上是不行的，至于 UNIX 呢？有同事曾经外出部署，出去前对方说好给一台 RedHat 作为 Zabbix Server 的部署设备，但是去到现场后对方给了一台 AIX，这位同事当时就“跪”了，至此后，我们再也没有人在 UNIX 系统上面部署过 Zabbix Server，所以效果如何也就不知道了。

以下为 CentOS 下部署 Zabbix Server 的例子，部署的过程全部采用源码编译的方式进行安装。使用源码编译安装的原因是很多情况下生产环境是不能联网的，所以采用源码编译安装是一个不错的方法。

11.1.1 部署 MySQL

安装 MySQL 之前要先安装 CMake，这样才能继续进行 MySQL 的编译。

> CMake 是一个跨平台的安装（编译）工具，可以用简单的语句来描述所有平台的安装（编译过程）。它能够输出各种各样的 makefile 或者 project 文件，能测试编译器所支持的 C++特性，类似 UNIX 下的 automake。只是 CMake 的组态档取名为 CmakeLists.txt。Cmake 并不直接建构出最终的软件，而是产生标准的建构档（如 UNIX 的 Makefile 或 Windows Visual C++ 的 projects/workspaces），然后再以一般的建构方式使用。这使得熟

> 悉某个集成开发环境（IDE）的开发者可以用标准的方式建构他的软件，这种可以使用各平台的原生建构系统的能力是 CMake 与 SCons 等其他类似系统的区别之处。

```
tar xvf cmake-2.6.4.tar.gz
cd  cmake-2.6.4
./configure --prefix=/usr/local/cmake
gmake
make install
```

安装完毕后更新环境变量：

```
vi ~/.bash_profile
```

进入 bash_profile 之后修改 PATH：

```
PATH=$PATH:$HOME/bin:/usr/local/cmake/bin
```

让修改立即生效：

```
source ~/.bash_profile
```

接下来就可以对 MySQL 进行编译安装了，编译过程会有点久，读者需要耐心等待一下。

```
tar -xvf mysql-5.6.12.tar.gz
cd mysql-5.6.12

cmake -DCMAKE_INSTALL_PREFIX=/usr/local/mysql \
-DMYSQL_USER=kira \
-DEXTRA_CHARSETS=all \
-DDEFAULT_CHARSET=utf8 \
-DMYSQL_UNIX_ADDR=/usr/local/mysql/mysql.sock \
-DDEFAULT_COLLATION=utf8_general_ci \
-DWITH_EXTRA_CHARSETS:STRING=utf8,gbk \
-DWITH_MYISAM_STORAGE_ENGINE=1 \
-DWITH_INNOBASE_STORAGE_ENGINE=1 \
-DWITH_MEMORY_STORAGE_ENGINE=1 \
-DWITH_READLINE=1 \
-DENABLED_LOCAL_INFILE=1 \
-DMYSQL_DATADIR=/usr/local/mysql/data \
-DMYSQL_TCP_PORT=3306

make
```

```
make install
```

编译完成之后我们需要把 MySQL 的配置文件复制到 etc 目录下：

```
chmod -Rf 777 /usr/local/mysql/*
chmod -Rf 777 /home/software/mysql-5.6.12
cp /home/software/mysql-5.6.12/support-files/my-default.cnf /etc/my.cnf
```

初始化数据库：

```
cd /home/software/mysql-5.6.12/scripts
./mysql_install_db --user=root --basedir=/usr/local/mysql/ --datadir=
/usr/local/mysql/data/
```

新增 MySQL 用户组及 MySQL 用户，用于启动 MySQL：

```
groupadd mysql
useradd -r -g mysql mysql
```

启动 MySQL：

```
/usr/local/mysql/bin/mysqld_safe
```

修改 MySQL 用户的密码：

```
/usr/local/mysql/bin/mysqladmin -u root password 1234
```

为 MySQL 创建软连接：

```
ln -s /usr/local/mysql/bin/mysql /usr/sbin/
```

开放 MySQL 远程访问的权限：

```
mysql -uroot -p1234
#GRANT  ALL  PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY '8565' WITH
GRANT OPTION;
#flush privileges;
```

11.1.2 编译部署 Apache+PHP

首先需要安装一些编译 PHP 所依赖的组件。

安装 libxml：

```
tar zxvf libxml2-2.7.7.tar.gz
cd libxml2-2.7.7
./configure --prefix=/usr/local/libxml2
make
make install
```

libxml 是一个用来解析 XML 文档的函数库。它用 C 语言写成，并且能为多种语言所调用，例如 C 语言、C++、XSH、C#、 Python、Kylix/Delphi、Ruby 和 PHP 等。Perl 中也可以使用 XML::LibXML 模块。它最初是为 GNOME 开发的项目，但现在可以用在各个方面。libXML 代码的可移植性非常好，因为它基于标准的 ANSI C 库，并采用 MIT 许可证。

安装 zlib：

```
tar -zxf zlib-1.2.5.tar.gz
cd zlib-1.2.5
./configure
make
make install
```

zlib 是提供数据压缩用的函式库，由 Jean-loup Gailly 与 Mark Adler 开发，初版 0.9 版在 1995 年 5 月 1 日发布。zlib 使用 DEFLATE 算法，最初是为 libpng 函式库所写的，后来普遍为许多软件所使用。此函式库为自由软件，使用 zlib 授权。截至 2007 年 3 月，zlib 是包含在 Coverity 的美国国土安全部赞助者选择继续审查的开源项目。

安装 jpeg：

```
tar -xvf jpegsrc.v8c.tar.gz
cd jpeg-8c/
./configure --prefix=/usr/local/jpeg
make
make install
```

安装 libpng：

```
tar -xvf libpng-1.5.7.tar.gz
cd libpng-1.5.7
./configure --prefix=/usr/local/libpng
make
```

```
make install
```

安装 freetype：

```
tar -xvf freetype-2.4.8.tar.gz
cd freetype-2.4.8
./configure --prefix=/usr/local/freetype
make
make install
```

安装 gettext：

```
tar -xvf gettext-0.18.1.1.tar.gz
cd gettext-0.18.1.1
./configure
make
make install
cp /usr/lib64/gettext/* /usr/lib64/   （64 位的系统才要）
```

安装 gd：

```
tar -xvf gd-2.0.35.tar.gz
cd gd/2.0.35
./configure --prefix=/usr/local/gd --with-jpeg=/usr/local/jpeg \
--with-png=/usr/local/libpng --with-freetype=/usr/local/freetype \
--enable-m4_pattern_allow
make
make install
```

安装 pcre：

```
unzip pcre-8.10.zip
cd pcre-8.10
./configure --prefix=/usr/local/pcre
make
make install
```

安装 apr：

```
tar -xvf apr-1.4.5.tar.gz
cd apr-1.4.5
./configure --prefix=/usr/local/apr
```

```
make
make install
```

安装 apr-util：

```
tar -xvf apr-util-1.3.12.tar.gz
cd apr-util-1.3.12
./configure --prefix=/usr/local/apr-util --with-apr=/home/software/apr-1.4.5
make
make install
```

安装 Apache：

```
tar -xvf httpd-2.4.6
cd httpd-2.4.6
./configure --prefix=/usr/local/apache --enable-module=so --with-apr=
/home/software/apr-1.4.5
--with-apr-util=/home/software/apr-util-1.3.12 --with-pcre=/usr/local/pcre/
make
make install
```

启动 Apache：

```
/usr/local/apache/bin/httpd -k start
```

安装 PHP：

```
tar -xvf php-5.4.17.tar.gz
cd php-5.4.17

./configure --prefix=/usr/local/php\
 --with-apxs2=/usr/local/apache/bin/apxs\
 --with-jpeg-dir=/usr/local/jpeg\
 --with-png-dir=/usr/local/libpng\
 --with-libxml-dir=/usr/local/libxml2\
 --with-gd=/usr/local/gd\
 --with-freetype-dir=/usr/local/freetype\
 --with-zlib=/usr/local/zlib --with-gettext\
 --enable-bcmath --enable-ftp --enable-mbstring --enable-sockets\
 --with-iconv-dir=/usr/lib --with-mysql=/usr/local/mysql
-with-mysqli=/usr/local/mysql/bin/mysql_config

 make
```

```
make install
```

修改 PHP 的配置文件：

```
cp php.ini-production /usr/local/php/lib//php.ini
vi  /usr/local/php/php.ini

#date.timezone = Asia/shanghai
#memory_limit = 256M
#max_input_time = 600
#max_execution_time = 600
#post_max_size = 32M
#upload_max_filesize = 16M
```

开启 Apache 的 PHP 支持：

```
vi /usr/local/apache/conf/httpd.conf
#AddType application/x-httpd-php .php
#DirectoryIndex index.html index.php
```

重新启动 Apache：

```
/usr/local/apache/bin/httpd -k stop
/usr/local/apache/bin/httpd -k start
```

11.1.3 安装 Zabbix

安装 Zabbix Server：

```
tar -xvf zabbix-2.2.0rc2.tar.gz
cd zabbix-2.2.0rc2
./configure --prefix=/usr/local/zabbix \
--enable-server --with-mysql=/usr/local/mysql/bin/mysql_config \
--enable-agent --enable-ipv6 --with-net-snmp --with-jabber=/usr/local
/jabber --with-libcurl
```

启动 Zabbix Server：

```
cp /usr/local/mysql/lib/libmysqlclient.so.18 /usr/lib
cp /usr/local/jabber/lib/libiksemel.so.3 /usr/lib
```

```
groupadd zabbix
useradd -g zabbix kira
passwd kira
su - kira
/usr/local/zabbix/sbin/zabbix_server -c /usr/local/zabbix/etc/zabbix_server.conf start
```

部署 Zabbix Server 页面：

```
cp -Rf /home/software/zabbix-2.2.0rc2/frontends/ /usr/local/apache/htdocs/
mv /usr/local/apache/htdocs/frontends/ /usr/local/apache/htdocs/zabbixZ
```

访问 Zabbix，如图 11.1 所示。

图 11.1　Zabbix DashBoard

至此，Zabbix Server 就启动起来了。

11.2　强大的触发规则

11.2.1　触发规则概览

自动化运维的前提其中的一个规则是在监控时发现了问题，然后对事件进行相应的动作，那么触发动作的前提就是软件具备强大的告警能力，而这点正是 Zabbix 所擅长的，如图 11.2、图 11.3 所示。

监控项 组名 Zabbix servers 主机 Zabbix server

名称	Key	类型	数据类型	状态
Buffers memory	vm.memory.size[buffers]	Zabbix 客户端	Numeric (unsigned)	活跃
Cached memory	vm.memory.size[cached]	Zabbix 客户端	Numeric (unsigned)	活跃
Checksum of /etc/inetd.conf	vfs.file.cksum[/etc/inetd.conf]	Zabbix 客户端	Numeric (unsigned)	不支持
Checksum of /etc/passwd	vfs.file.cksum[/etc/passwd]	Zabbix 客户端	Numeric (unsigned)	活跃
Checksum of /usr/sbin/sshd	vfs.file.cksum[/usr/sbin/sshd]	Zabbix 客户端	Numeric (unsigned)	活跃
Checksum of /boot/vmlinuz	vfs.file.cksum[/boot/vmlinuz]	Zabbix 客户端	Numeric (unsigned)	不支持
Checksum of /etc/services	vfs.file.cksum[/etc/services]	Zabbix 客户端	Numeric (unsigned)	活跃
Checksum of /usr/bin/ssh	vfs.file.cksum[/usr/bin/ssh]	Zabbix 客户端	Numeric (unsigned)	活跃
CPU idle time (avg1)	system.cpu.util[,idle,avg1]	Zabbix 客户端	Numeric (float)	活跃
CPU iowait time (avg1)	system.cpu.util[,iowait,avg1]	Zabbix 客户端	Numeric (float)	活跃
CPU system time (avg1)	system.cpu.util[,system,avg1]	Zabbix 客户端	Numeric (float)	活跃
CPU nice time (avg1)	system.cpu.util[,nice,avg1]	Zabbix 客户端	Numeric (float)	活跃
CPU user time (avg1)	system.cpu.util[,user,avg1]	Zabbix 客户端	Numeric (float)	活跃
Email (SMTP) server is running	net.tcp.service[smtp]	Zabbix 客户端	Numeric (unsigned)	活跃
Free disk space on /opt	vfs.fs.size[/opt,free]	Zabbix 客户端	Numeric (unsigned)	活跃
Free disk space on /	vfs.fs.size[/,free]	Zabbix 客户端	Numeric (unsigned)	活跃

图 11.2 监控项

由于 Zabbix 还能支持非常丰富的告警表达式，所以 Zabbix 的界面还为我们提供了表达式构建器，方便我们对表达式进行构造。

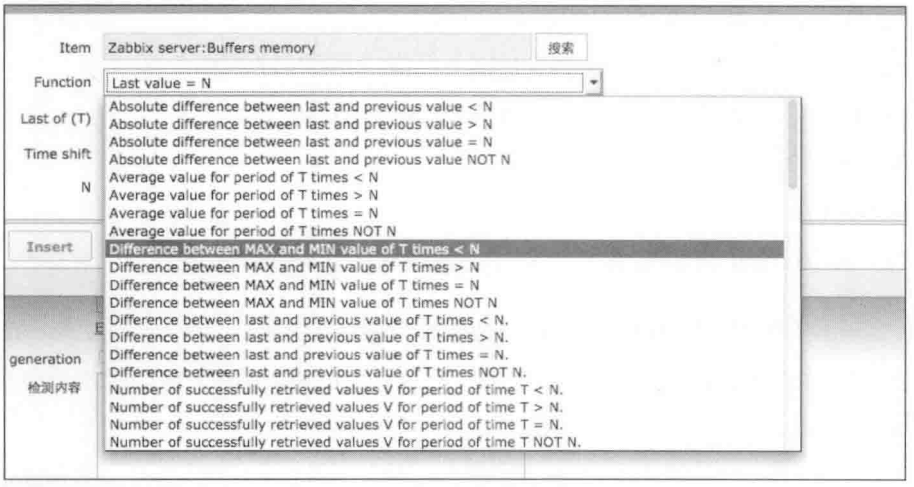


图 11.3 强大的触发规则

11.2.2 特色的触发规则

Zabbix 的触发规则非常强大，我们可以创建许多复杂的触发规则。表达式的语法如下：

```
{<server>:<key>.<function>(<parameter>)}<operator><constant>
```

- server：表示这个触发规则使用哪一台主机；
- key：表示触发规则针对哪一个监控项；
- function：由于每次监控采集的值都是单个点的，当我们需要监控某个时间段的最大值是大于多少的时候，就需要使用 function 来帮助我们实现了，函数里面是可以传递一些参数的，例如 sum 是统计所有监控点的函数，sum(600)则代表统计最近 600 秒的所有值，sum(#5)代表统计最后 5 次的采样值；
- operators：触发器表达式的一些操作符，用来描述某个表达式是大于某个值还是小于某个值才需要告警。

符号语法如下所述。

- /：除；
- *：乘；
- -：减；
- +：加；
- <：小于，A<B 与 A<=B-0.000001 等价；
- >：大于，A>B 与 A>=B+0.000001 等价；
- #：不等于，A#B 与(A<=B-0.00001)|(A>=B+0.000001)等价；
- =：相等，与(A>B-0.0000001)&(A<B+0.000001)等价；
- &：逻辑与；
- |：逻辑或。

例 1：当主机 host 的 CPU 最后一次平均负载大于 5 的时候告警。

```
{host:system.cpu.load[all,avg1].last(0)}>5
```

例 2：当 CPU 的最后一次平均负载大于 5 并且 10 分钟内的平均负载最小值大于 2 的时候告警。

```
{host:system.cpu.load[all,avg1].last(0)}>5|{host:system.cpu.load
[all,avg1].min(10m)}>2
```

例 3：当/etc/passwd 文件被修改后告警。

```
{host:vfs.file.cksum[/etc/passwd].diff(0)}>0
```

例 4：当下行网络带宽大于 100 KB 的时候告警。

```
{host:net.if.in[eth0,bytes].min(5m)}>100K
```

例 5：当两个节点的 SMTP 服务都 down 了才告警。

```
{smtp1.zabbix.com:net.tcp.service[smtp].last(0)}=0&{smtp2.zabbix
.com:net.tcp.service[smtp].last(0)}=0
```

例 6：当 ZabbixAgent 的版本为 beta8 的时候告警。

```
{host:agent.version.str("beta8")}=1
```

例 7：当服务器在最后 30 分钟大于 5 分钟无法 ping 通的时候告警。

```
{host:icmpping.count(30m,0)}>5
```

例 8：当某个监控项大于 3 分钟没有数据采集的时候告警。

```
{zabbix.zabbix.com:tick.nodata(3m)}=1
```

例 9：在晚上 12 点到早上 6 点这段期间内，CPU 平均负载最小值大于 2 的时候告警。

```
{host:system.cpu.load[all,avg1].min(5m)}>2&{host:system.cpu.load
[all,avg1].time(0)}>000000&{host:system.cpu.load[all,avg1].time(0)}<060000
```

例 10：当服务器与客户机的时间差大于 10 秒的时候告警。

```
{MySQL_DB:system.localtime.fuzzytime(10)}=0
```

例 11：当今天的平均负载比昨天同一时间的平均负载大于 2 的时候告警。

```
{server:system.cpu.load.avg(1h)}/{server:system.cpu.load.avg(1h,1d)}>2
```

11.3 Zabbix 调用 OSGi 运维功能

可以看到 Zabbix 在告警功能上做的是非常灵活的，有了这么灵活的告警策略，我们就可以在告警之后触发某些动作，让这个动作帮助我们完成自动化的恢复，实现自动化运维目的。

整体设计的数据流如图 11.4 所示。

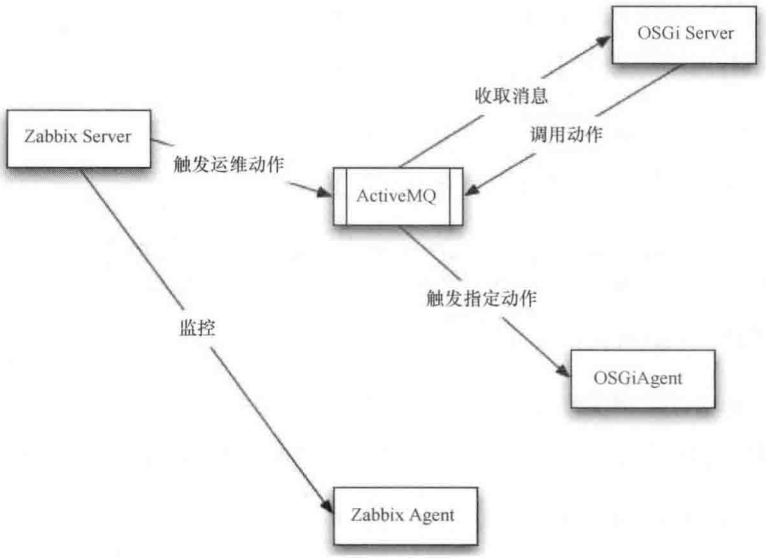


图 11.4　整合 Zabbix 与 OSGi 运维框架的思路

我们使用 Zabbix 的监控功能时，当发现客户端有告警出现之后，调用脚本向 ActiveMQ 发送消息。为了支持不同的平台，Java 是最好的选择，所以我们会让 Zabbix 调用一个叫作 JMSSender 的包，把消息送到消息队列上，后续就是常规的 OSGi 集中运维框架所做的事情了。

配置告警触发动作，如图 11.5、图 11.6 所示。

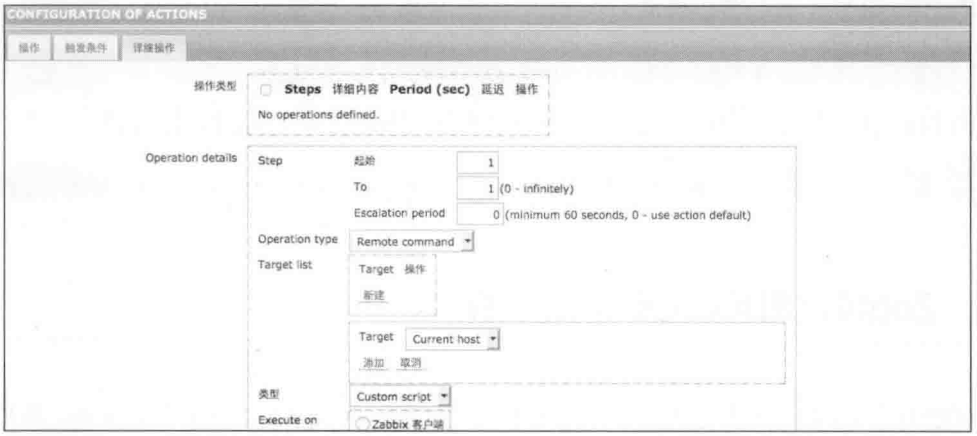


图 11.5　配置触发动作（1）

图 11.6 配置触发动作（2）

配置完成报警触发动作之后，我们就可以通过 JMSSender 把运维动作发送下去了，下面为 JMSSender 的代码。

```
public class main {

    public static void main(String[] args) throws FileNotFoundException,
            IOException {
        String queuename = args[0];
        String content = args[1];

        String fileName = (new File(System.getProperty("java.class.path")
                .replace("jmssender.jar", "mqAgent.properties")))
                .getAbsolutePath();
        Properties properties = new Properties();
        properties.load(new FileInputStream(new File(fileName)));

        ConnectionFactory connectionFactory;
        Connection connection = null;
        Session session;
```

```
                Destination destination;
                MessageProducer producer;
                connectionFactory = new ActiveMQConnectionFactory(
                        properties.getProperty("client.mq.username"),
                        properties.getProperty("client.mq.password"),
                        properties.getProperty("client.mq.url"));

                try {
                    connection = connectionFactory.createConnection();
                    connection.start();
                    session = connection.createSession(Boolean.TRUE,
                            Session.AUTO_ACKNOWLEDGE);
                    destination  =  session.createQueue(queuename+properties.
getProperty("client.ip"));
        //top_sql_
                    producer = session.createProducer(destination);
                    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);


                    TextMessage message = session.createTextMessage(content);

                    producer.send(message);

                    session.commit();
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    try {
                        if (null != connection)
                            connection.close();
                    } catch (Throwable ignore) {
                    }
                }
            }
        }
```

第12章 案例

周末的晚上，小菜约大鸟在外面吃饭。两人聊的正投入，小菜手机突然响了起来，大鸟瞄了小菜的手机一眼，看到来电显示上面写着 K 总，然后在旁边偷偷地笑："哈哈，这个时候打电话过来，准不是什么好事"。小菜推了一下大鸟，"去去去，说不定人家 K 总看我最近表现不错，要让我出任 CEO，迎娶白富美，准备给我大涨工资呢。"说完，小菜按下了接听的按钮。电话那头传来了 K 总的声音："小菜啊，问你个问题，你对 C 语言熟不熟悉？"小菜一听，笑嘻嘻地说："熟悉，当然熟悉啊，想当年我学习的第一门语言就是 C 语言，还用它写过一些……"还没等小菜说下去，K 总就说："熟悉就好，这样，下周有个项目，我希望你能接手一下，详细的情况下周告诉你。"没等小菜反应过来，K 总就把电话挂了。小菜把电话放回口袋，用求助的眼神看了看大鸟。大鸟笑嘻嘻地说："说不定下周一就让你出任 CEO，迎娶白富美啦，哈哈。"

周一，小菜回到公司，K 总就把小菜叫到了办公室，"来，这边坐，我们有一个项目，是基于 Zabbix 做的监控，希望能在这个基础上做一个运维软件的补充。简单来说就是我们希望能够在监控的基础上加入运维的功能，通过监控来驱动运维的操作，完成一些常规运维操作的自动化。"小菜听了之后，信心满满地告诉 K 总："这个问题简单，就交给我负责吧，我已经有思路了。"K 总拍了拍小菜的肩膀："年轻人，别让我失望啊，这样吧，下周给我一个解决方案。细节方面你可以问一下大涛"。

出了 K 总的办公室，小菜心里想："这事情听起来不难啊，Zabbix 是现成的工具，再补充一个运维的功能，就这样？所谓的运维的功能不就操作一下主机嘛，用 Python 写个 SSH 的小程序不就完了？自己写还费时费力，用 Ansible 和 Zabbix 整合起来不就可以了嘛，看来这回涨薪有望了。"

小菜心中设想的系统结构设计如图 12.1 所示。

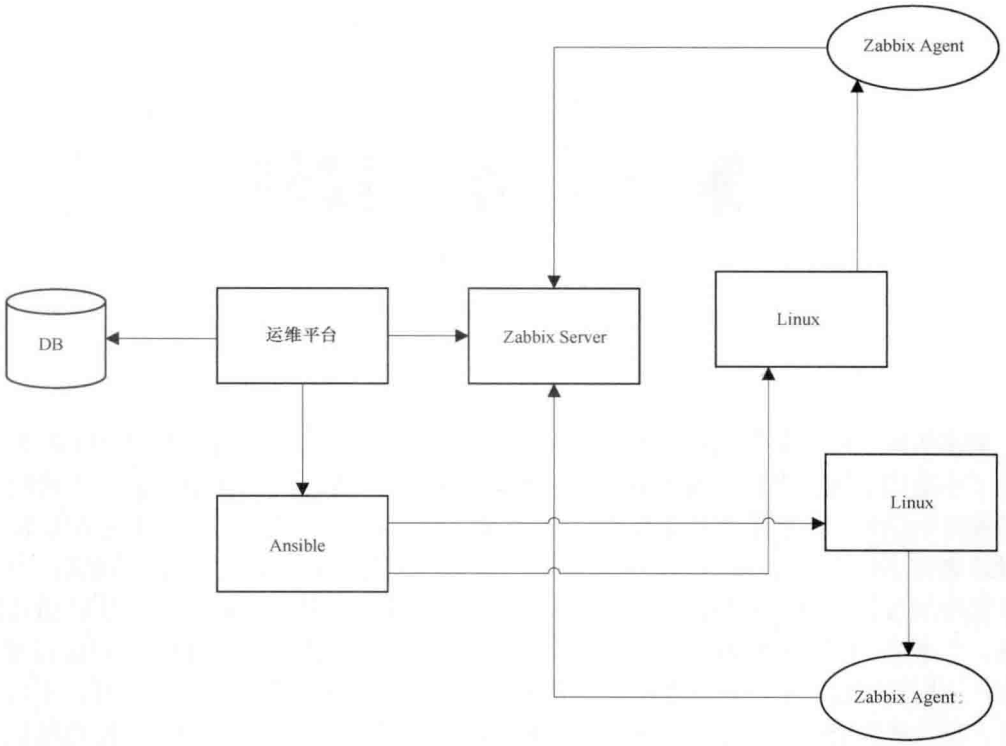


图 12.1　小菜设想中的系统结构设计

正好大涛这时从小菜面前走过，小菜赶紧叫住大涛："涛哥，K 总让我想一下怎么样给我们的产品加上一些运维的功能，你给我说说现在是什么样一个情况呗。"大涛一听，拍了拍小菜的肩膀："K 总让你负责这件事情啊，好事好事，来来，我这就和你讲一讲"。当大涛把事情的来龙去脉给小菜讲了一遍之后，信心满满的小菜开始不淡定了："这…这和 K 总给我讲的完全不一样啊"。大涛笑着说："后续的事情就靠你啦，加油干吧小伙子。"这个时候，小菜终于清楚地意识到事情和他想象中差了十万八千里。他从大涛那里听来的情况是这样的：

（1）需要运维的目标设备数量有一万多台，这些设备上的操作系统种类非常多，有 Windows Server、Redhat、AIX 等，不同种类的操作系统还存在许多不同的版本。

（2）假设需要部署客户端，那么只允许进行一次大规模部署，大规模部署之后，后续的更新希望是比较方便的，不用每次都让维护团队的人手动更新客户端。

（3）由于环境的问题，不能访问外部网络。

（4）由于部署的范围比较广，会涉及不同地市的部署，网络环境上也会有多级网络的情况。

（5）维护团队的水平参差不齐，希望能够尽量简化部署过程的难度。

（6）需求后续会不断地变化。

（7）Zabbix 具有非常强大的告警和触发机制，所以我们需要做的事情就是把运维的功能和 Zabbix 的触发功能对接起来。

这回小菜汗都冒出来了，心想："这回掉到坑里面去了"。

为了应对这种艰难的环境，小菜回想起大鸟曾经教过自己 Puppet 和 SaltStack 的一些知识，于是晚上回家补习了一下 Puppet 和 SaltStack 的知识，觉得这两款开源软件比较靠谱，两款软件的程序结构如图 12.2、图 12.3 所示。

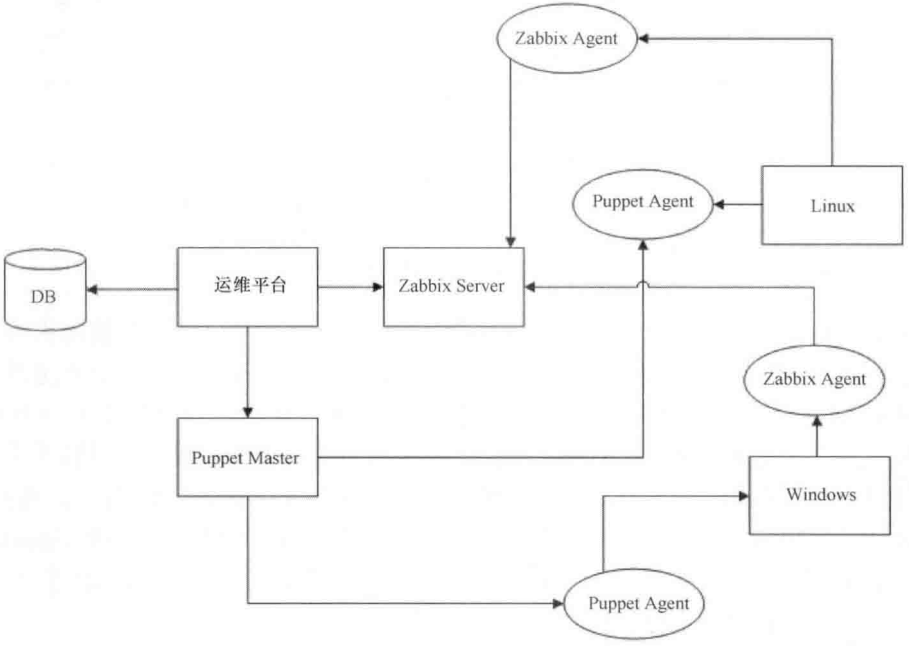


图 12.2　使用 Puppet 的程序结构

第二天，小菜在公司尝试了 SaltStack 和 Puppet 的部署，发现没有什么太大的问题，然后就考虑去客户现场的测试环境上尝试部署。在客户现场的测试环境上开始尝试部署之后，小菜发现自己太小看现场环境了。各种依赖包的缺失，甚至有些机器连 GCC 都没装好。小菜废了好大的力气，总算是把 SaltStack 和 Puppet 的客户端安装上去了。"这可不行啊，我自己部署都这么困难，假如交给维护团队去部署，到时候岂不是会被投诉到头都大了"。

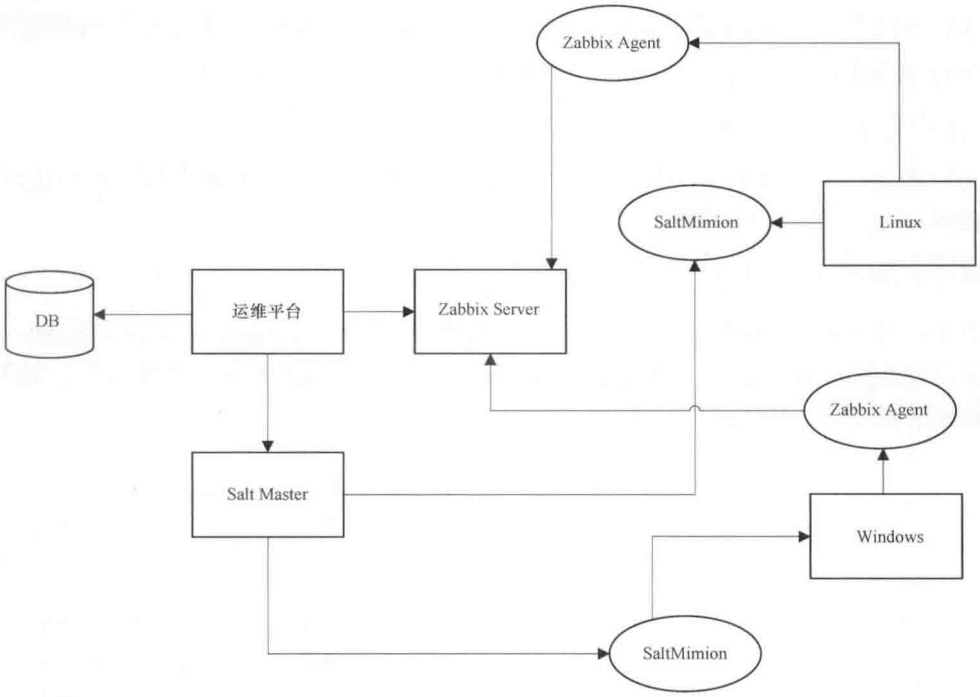


图 12.3　使用 SaltStack 的程序结构

晚上十二点，小菜实在想不出什么好点子，拿起电话打给了大鸟，把情况告诉了大鸟，大鸟听了，想了一想，和小菜说："你们这个项目的需求也真是奇葩，不过也不是没有办法，你听说过 OSGi 吗？它应该能够解决你的问题"。小菜听完之后，赶紧搜索了一下 OSGi 的资源，看完之后觉得无从下手，首先 OSGi 这种技术用的人本来就很少，资料不多。然后，网上所讲到的 OSGi 的应用大多都是一些模块化的开发方式。小菜心里想："大鸟究竟有没详细了解过这项技术呢，不会是在忽悠我吧。"无意间，小菜看到了一个叫 Apache Karaf 的项目，小菜怀着好奇的心看了一下它的相关文档。看着看着，小菜突然眼前一亮，哈，就是这个，我就是想要一个这样的容器。

今天是与 K 总约定需要汇报方案的日子，小菜信心满满地打开了昨晚通宵做好的 PPT。

"基于我们这个项目的一些约束条件，我尝试采用了 Ansible、Puppet 和 SaltStack 这三个方案与 Zabbix 同我们现有的平台进行整合，但是都出现了一些问题。Ansible 具备不需要安装客户端的优点，对 UNIX 和 Linux 的支持非常不错，但是对 Windows 的支持偏弱。而 SaltStack 和 Puppet 虽然非常不错，但要在我们这个项目的约束条件下运用起来，成本太高。"小菜说道。K 总听了，疑惑地问道："那你有什么好的解决方案？"小菜笑了笑，说道："Java

里面有一种叫 OSGi 的技术。我所提议的方案是采用 OSGi 的技术进行实现的。简单来说，就是客户端我们全新开发，毕竟我们需要的功能并不会很多，但是定制性会比较大。然后，这种技术是 Java 实现的，我们可以在部署客户端的时候把 JRE 一起打包到安装包里面，这样就可以降低客户端部署的难度，而且 Java 的跨平台特性也是非常不错的。最后，由于是我们自己开发的，难免会出现比较多的 BUG，但是我们的客户端数量又如此之多，这种情况下，OSGi 的特性能够帮助我们很容易地解决这个问题。当然 OSGi 技术只是解决了 Agent 和 Server 的问题，那 Agent 和 Server 之间的通信，我们可以选择现成的一些消息中间件，我选择的是 Apache ActiveMQ，用来降低开发难度，加快整体的开发进度。而与 Zabbix 整合的部分，我们只需要通过 Zabbix 的动作管理去配置，让它去触发一个 Agent 的动作就可以了”。

具体的运维系统方案如图 12.4 所示。

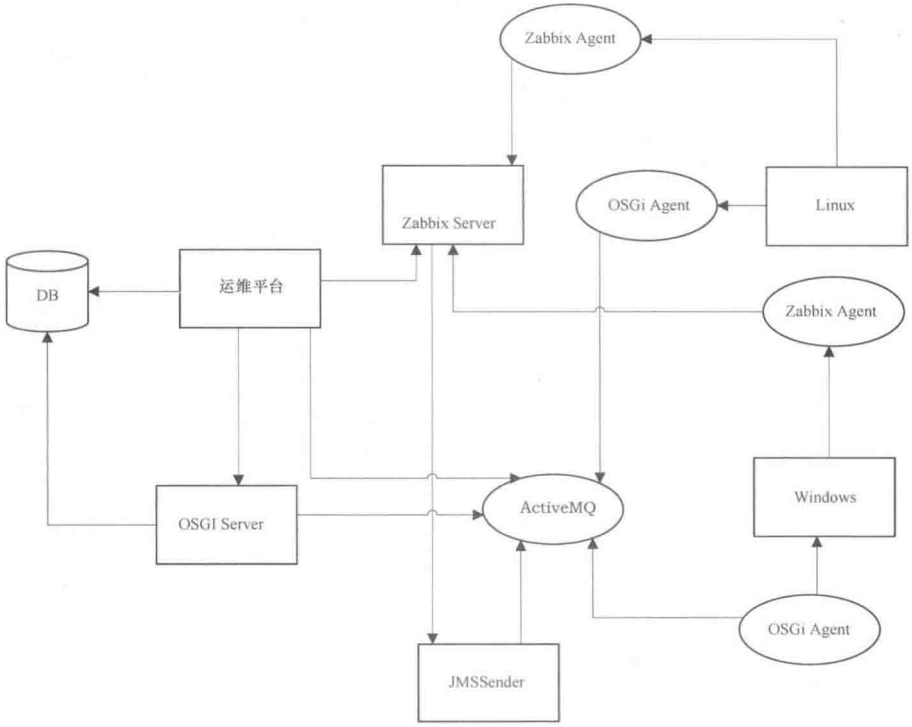


图 12.4　采用 OSGi+ActiveMQ 的运维系统方案

K 总听了之后，感觉有点道理，说道：“好吧，就按照你这个方案，我希望一个月之内能看到成果。”小菜听了，心想总算过了一关，这回思路是有了，具体实现呢？于是打开了购物网站，买了几本书进行补习，开始进行他这个项目的下一步计划。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

自动化运维
软件设计实战

业内专家力荐

随着云计算的兴起，大量采用廉价的x86服务器结合Linux操作系统来作为IT基础架构已经是非常普遍的情况，但是在传统行业中，除了x86+Linux，还混杂着大量如版本不一、缺少升级的AIX、Solaris、HP-UX、Windows 组成的历史遗留系统；同时传统企业在稳定生产的需求上，也难以提供市场上大部分自动化运维软件运行的基础依赖。如何在这种复杂的环境中，实现自动化运维，降低运维成本，成为了一个巨大的技术挑战。

本书从实践出发，详细地描述了解决这个技术挑战的思路、过程，以及具体的落地方案，而且过程中尽可能地使用成熟可靠的开源软件，使得整个方案复制性强，为广大拼杀在传统行业的广大运维人员及运维开发人员提供了很好的参考。

上海新炬网络高级产品经理

陈自欣

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：陈晓猛
封面设计：李　玲

上架建议：运维开发

ISBN 978-7-121-26468-9

9 787121 264689 >

定价：69.00元

[General Information]
书名=自动化运维软件设计实战
作者=吴文豪著
页数=279
SS号=13868202
DX号=
出版日期=2015.07
出版社=北京电子工业出版社