

# Python 自动化运维 技术与最佳实践

刘天斯 著

---

Automation Operations with Python  
Technique and Best Practices

---

- 中国运维领域偶像级专家、腾讯高级系统工程师在天涯社区和腾讯近10年运维实践的经验和智慧结晶
- 不仅详尽介绍了服务监控、数据报表、系统安全等基础模块，而且深入讲解了自动化操作、系统管理、配置管理、集群管理及大数据应用等高级功能，包含4个完整的综合案例



机械工业出版社  
China Machine Press

# Python 自动化运维 技术与最佳实践

---

Automation Operations with Python  
Technique and Best Practices

---

刘天斯 著



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

Python 自动化运维：技术与最佳实践 / 刘天斯著. —北京：机械工业出版社，2014.11  
(2014.12 重印)

(Linux/Unix 技术丛书)

ISBN 978-7-111-48306-9

I. P… II. 刘… III. 软件工具—程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字 (2014) 第 242462 号

## Python 自动化运维：技术与最佳实践

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：姜 影

责任校对：殷 虹

印 刷：三河市宏图印务有限公司

版 次：2014 年 12 月第 1 版第 2 次印刷

开 本：186mm×240mm 1/16

印 张：19.5

书 号：ISBN 978-7-111-48306-9

定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

## Praise 本书赞誉

市面上介绍互动的、面向对象的 Python 编程语言的书有很多，其强大而又灵活的特性，使其成为很多企图通过工具来实现工作（半）自动化的运营同学的首选。更难得的是，本书作者以其在腾讯游戏运营的工作经验，辅以大量实际的案例来讲述了他是如何使用 Python 来解决诸如监控、安全、订制报表和大数据应用等问题，以及构建一个自动化运维的平台来提升运维工作效率，值得一看。

腾讯互动娱乐运营部副总经理 崔晓春

《Python 自动化运维：技术与最佳实践》是结合刘天斯先生超过十年，在互联网行业“天涯在线”及“腾讯”等的工作经验，实际贴近工作应用场景所撰写的书籍，没有浮夸的文藻修饰，只有实际的落地执行和动手操做，可以作为大家在工作中的工具书。

全书以系统信息的了解、采集、监控，以及信息良好地输出为开头，以提升个人工作效率的基础运维工具为承接，再深入介绍集中化管理海量机器、系统的方案，并且搭配实际的例子进行介绍，相信能够覆盖读者的大部分应用场景需求，也能够给予读者相关领域的入门指引。

刘天斯先生的精神也是很值得推广和赞赏的，在繁忙的工作之余，能够思考、总结，并且能够以文字的方式与更多的人分享和传承，是除了书籍本身之外，我学习到的重要收获。

腾讯互动娱乐运营部数据中心总监 孙龙君

在移动互联和大数据时代，无论是出于对效率的追逐，还是应对海量规模运维，自动化运维都是企业的必然选择。Python 因为具有简单、灵活、功能强大和适合脚本处理等优点，在运维领域被广泛使用，让很多运维工程师从烦琐的日常工作中解放出来。

天斯是运维领域的资深专家，在互联网行业工作多年，不仅具备解决各种运维难题的强大能力，拥有多项专利，还开发过多个运维利器，非常受欢迎。本书是国内第一本讲述

Python 如何应用在自动化运维领域的著作，是基于天斯对 Python 自动化运维的深入研究，以及在海量互联网实战经验中总结提炼而来，具有高度可读性和实战价值。

腾讯架构平台部运维服务中心总监 孙雷

刘天斯和我相识于腾讯，期间我正在负责腾讯云平台相关工作。腾讯有一个优良的新员工培养体系，那就是导师制度。有幸作为天斯的导师，让我接触并逐渐深入了解天斯。所以当天斯找到我为本书写推荐语时，我欣然应允，因为共事期间天斯给我留下了深刻的印象。时至今日，在中国的互联网企业里，我认为天斯都是最优秀的架构师之一。

天斯来腾讯工作之前，在中国著名的天涯社区负责整个社区的运维工作，经历了天涯社区从 Windows 平台到开源架构的大改造，因此对 B/S 相关产品的技术架构和细节非常熟悉；而天斯又是一个在技术输出领域非常活跃的人，自己维护的技术博客荣获 2010 年度十大杰出 IT 博客，在中国互联网技术领域小有名气。

记得来腾讯不到两个星期，天斯就向我提交了一份关于腾讯业务自动化运维的技术文档，从业务的部署到监控再到容灾等，都理解得较为深刻。这份输出文档让我眼前一亮，当时第一感觉是这个典型的在生活中不善言辞的 IT 男，一定对云计算中的自动运维管理有独到的思维和沉淀。

Python 语言作为获得 2010 年度编程大奖的语言，具备诸多优点：简单、开源、速度快、可移植性强、可扩展性强、面对对象、具备丰富的库等；更可贵的是，作为“胶水语言”，可以把 Python 嵌入 C/C++ 程序等，从而向程序用户提供脚本功能。

本书从互联网业务自动运维的场景出发，以 Python 语言为基础，总结了大量的实战案例，这些都是作者在十余年的大型互联网运维工作中的宝贵经验，相信会给读者带来不少的启发。

更难能可贵的是，作者能从通俗易懂的角度出发，由浅入深地剖析 Python 自动运维管理之道。因此，目前 Python 水平处于各种层次的读者均能有效地阅读和吸收，各取所需。

最后，感谢天斯能给中国互联网从业者带来这么好的分享，感谢我们的老东家——中国互联网的黄埔军校——腾讯培养了一批又一批的杰出架构师。

开卷有益，我想应该就是指此类书籍吧。

微赢宝创始人 许明

“Operation”，运维在互联网时代一直有着举足轻重的地位，而近两年运维本身这个群体也变得强大起来，最为显著的特征就是运维人员所出的书越来越多，而都以“专”、“精”为卖点。这也是作为一名运维人员值得骄傲的地方。

伴随着“云时代”、“物联网”的到来，无论数据，还是服务器规模都达到了空前的庞大，

企业对运维工作人员的要求也由之前的运维维护转为“DevOps”，即研发型运维；在这个充满挑战的时代，任何一个岗位都需要保持持续学习的状态，而运维也不例外。

“Python”，运维的标配语言，比起 Bash、Perl、PHP 等，它在系统管理上有着强大的开发能力和完整的工具链。易读易写，兼具面向对象和函数式风格，还有元编程能力都是它的优势所在。最关键的地方在于，可以利用 Python 系统化地将各个工具进行整合，对运维常用工具进行二次开发，形成一套完整的运维体系。“一套完整的产品生命周期”，这才是运维需要做的事情。

运维“三板斧”：系统安装、命令执行、配置管理，再加上监控与日志分析等这些都是我们最常用的工具，而它们都有 Python 的版本，例如：Fabric、Ansible、Saltstack、Func 等，这些都将在本书《Python 自动化运维：技术与最佳实践》中向大家一一呈现，安装、用法、技巧、特别是大量实例一网打尽。为了让读者更好地系统学习，天斯又写了前端以及从“0”开始打造一个运维平台，可谓用心良苦。

未来，中小型企业将精减运维，不会开发的运维，竞争力将显得更加单薄，相信天斯多年运维开发经验的结晶能帮到大家。

西山居架构师，《Puppet 实战》作者 刘宇

初识刘天斯先生是邀其参加我在 ChinaUnix 举办的活动——“千万级 pv 高性能高并发网站架构与设计交流”，刘天斯先生提出的架构方案，堪称成熟、缜密、灵动，足见其在系统运维领域的功力。纵观《Python 自动化运维：技术与最佳实践》一书，都是出自于刘天斯先生在天涯及腾讯工作的一线宝贵经验，相信无论是开发人员还是系统管理员们均能从中学学习到新的知识点，使自己的职业生涯更上一个新的台阶。

——融贯资讯系统架构师 余洪春

## 前言 *Preface*

### 为什么要写这本书

随着信息时代的迅速发展，尤其是互联网日益融入大众生活，作为这一切背后的 IT 服务支撑，运维角色的作用越来越大，传统的人工运维方式已经无法满足业务的发展需求，需要从流程化、标准化、自动化去构建运维体系，其中流程化与标准化是自动化的前提条件，自动化的最终目的是提高工作效率、释放人力资源、节约运营成本、提升业务服务质量等。我们该如何达成这个目标呢？运维自动化工具的建设是最重要的途径，具体包括监控、部署变更、安全保障、故障处理、运营数据报表等。本书介绍如何使用 Python 语言来实现这些功能点，以及 Python 在我们的自动化运维之路上发挥作用，解决了哪些运维问题等。

为什么是 Python？Python 是一种面向对象、解释型计算机程序设计语言，由 Guido van Rossum 于 1989 年年底发明，具有简单易学、开发效率高、运行速度快、跨平台等特点，尤其是具有大量第三方模块的支持，其中不乏优秀的运维相关组件，例如 Saltstack、Ansible、Func、Fabric 等。大部分运维人员为非专业开发人士，对他们而言，选择一门上手快、技术门槛低的开发语言非常重要。由于 Python 具有脚本语言的特点，学习资源多，社区非常活跃，且在 Linux 平台默认已安装等优势。Python 已经是当今运维领域最流行的开发语言之一。

2003 年毕业后，我的第一份工作是当 PHP 程序员，人力紧张时还要兼顾美工的工作。时常回想，其实也只有在小公司才能修炼出“十八般武艺”。在“非典”肆虐的岁月，大部分公司都闭门不招聘，一个毕业生能有这样的机会锻炼也显得尤为珍贵。工作中一次偶然的机会看到导师诗成兄在黑漆漆的界面中输入不同指令，第一感觉非常震撼，很酷，联想到《黑客帝国》电影中的画面，与之前接触到的 Windows 系统完全不一样，后来才晓得是 Redhat 9（红帽 9）。此后很长的一段时间里，整个人完全沉醉在 Linux 的世界里，处于一种痴迷的状态，那时我还是一个程序员。

到了 2005 年 10 月，看到隔壁公司招聘一名 Linux 系统工程师，抱着试一试的心态去面试，结果出乎意料，我被录用了，这样我就找到了第二个东家——天涯社区。人生的第一

个转折点在此酝酿，由于赶上了公司快速发展的阶段，接触到了很多开源技术，包括 LVS、Squid、Haproxy、MongoDB、MySQL、Cfengine 等，并且不断在生产环境中应用所学的技术，取得了非常不错的效果，重点业务的高可用持续保持在 99.99%。期间新的问题也陆续出现，包括如何更好整合各类开源组件，发挥其最大效能，以及如何高效运营。不可否认，具有开发背景的运维人员有着先天优势，可以在不同角色之间进行思考，扩大视野。期间我参与了推动大量标准化、规范化的建设，以此为前提，开发了“SDR1.0-Linux 主机集中管理”、“天涯 LVS 管理系统”、“天涯服务器管理系统（C/S 与 B/S 版）”、“服务器机柜模拟图平台”、“Varnish 缓存推送平台 V1.0”等平台，这些平台在很大程度上改变了运维人员手工作坊式的工作模式。在释放人力的同时，我看到国内其他公司的同仁也在做同样的事情，突然间有一个想法，就是开源。此时已经是 2009 年，这个想法也得到系统部经理小军认可，同年 12 月陆续在 code.google.com 平台托管，让业界更深入地了解天涯社区的技术架构。凭着这些作品及分享的技术文章，我的博客“运维进行时”（<http://blog.liuts.com/>）荣获了“2010 年度十大杰出 IT 博客”的殊荣。我还先后参与了 51CTO、IT168、CU 等门户网站以架构、运维为主题的专访，在运维圈得到越来越多同仁的认同。

再谈谈如何与 Python 结缘。接触 Python 是从《简明 Python 教程》开始，由于我有 Perl 与 PHP 的基础，学习 Python 没有太大压力。事实上，Python 的简洁、容易上手以及大量第三方模块等特点，深深吸引了我，让我第二次沉醉于知识的海洋。我很快深入学习了 Func、Django 框架、SQLAlchemy、BeautifulSoup、Pys60、wxPython、Pygame、wmi 等经典模块，同时将所学知识应用到运维体系中，解决在工作中碰到的问题。例如，开发的“多节点应用延时监控平台”解决了多运营商网络环境下的业务服务质量监控问题；开发的“Varnish&Squid 缓存推送平台”解决了快速刷新缓存对象的问题。再例如，删除敏感帖子的时效性要求非常高，需要在后台触发删除后立即生效，与缓存推送平台对接后很好地解决了这一问题；天涯服务器管理系统（C/S、B/S、移动版）实现自助、智能、多维度接入，提高了运维效率，减少了人工误操作，释放了人力资源，同时标准化与流程化得到技术保障与实施落地。

天涯社区是我个人职业生涯的培育期，让我重新审视自我，明确了未来的规划与定位。2011 年 9 月是我职业生涯的成长期的开始，加盟了腾讯，负责静态图片、大游戏下载业务 CDN 的运维工作，接触到庞大的用户群、海量的资源（设备、带宽、存储）、世界级的平台、人性化的工作氛围以及大量优秀的同事。所有的这些都深深地吸引着我，也让我的视野与工作能力得到前所未有的提升。分工细化产生运维工作模式的差异，从“单兵作战”转向“集团军作战”。我继续保持着对新技术的狂热，思考如何使用 Python 在运维工作中发挥作用。工作期间研究了大量高级组件，包括 Paramiko、Fabric、Saltstack、Ansible、Func 等，这些组件有了更高级的封装，强大且灵活，贴近各类业务场景。我个人也基于 Python 开发了集群自动化操作工具——yorauto，在公司各大事业群广泛使用，同时入选公司精品推荐组件。我的部分个人发明专利使用 Python 作为技术实现。目前我也关注大数据发展趋势，研究 Python 在大数据领域所扮演的角色。

回到主题“为什么要写这本书”，这一点可以从 51CTO 对我的专访中找到答案。当时的场景是这样的：

51CTO：您对开源是如何理解的？天涯社区在过去两年间陆续开源了包含 LVS 管理系统、Varnish 缓存推送平台、高性能数据引擎 memlink 等好几个项目，业内人士对此都十分关注，您认为这给整个产业带来了哪些好处？身为天涯社区的一位运维人员，您认为在这个过程中自己的价值在哪里？

刘天斯：开源就是分享，让更多人受益的同时自己也在提高。经常看到很多朋友都在做监控平台、运维工具。事实上功能惊人相似，大家都在做重复的工作，为什么不能由一个人开源出来，大家一起来使用、完善呢。这样对整个行业来讲，这块的投入成本都会降低，对个体来讲也是资源的整合。如果形成良性循环，行业的生态环境将会有很大程度的改善。本人热衷于开源技术，同样也愿意为开源贡献自己一分微薄之力，希望更多的人能支持开源、参考开源。

这就是我的初衷，也是答案。写书的意义在于将 10 年的工作沉淀、经验、思路方法做个梳理与总结，同时与大家分享。最终目的是为每个渴望学习、进步、提升的运营人员提供指导。

## 读者对象

- ☐ 系统架构师、运维人员
- ☐ 运营开发人员
- ☐ Python 程序员
- ☐ 系统管理员或企业网管
- ☐ 大专院校的计算机专业学生

## 如何阅读本书

本书分为三大部分。

第一部分为基础篇（第 1 ~ 4 章），介绍 Python 在运维领域中的常用基础模块，覆盖了系统基础信息、服务监控、数据报表、系统安全等内容。

第二部分为高级篇（第 5 ~ 12 章），着重讲解 Python 在系统运维生命周期中的高级应用功能，包括相关自动化操作、系统管理、配置管理、集群管理及大数据应用等内容。

第三部分为案例篇（第 13 ~ 16 章），通过讲解 4 个不同功能运维平台案例，让读者了解平台的完整架构及开发流程。

说明：

- 书中的代码以“【 路径 】”方式引用，测试路径为“/home/test/ 模块”、“/data/www/ 项目”。
- 书中涉及的所有示例及源码的 Github 地址为 <https://github.com/yorkoliu/pyauto>，以章节名称作为目录层次结构，模块及项目代码分别存放在对应的章节目录中。

其中第三部分以接近实战的案例来讲解，相比于前两部分更独立。如果你是一名经验丰富 Linux 管理员且具有 Python 基础，可以直接切入高级篇。但如果你是一名初学者，请一定从基础篇开始学习。本书不涉及 Python 基础知识，推荐新手在线学习手册：《简明 Python 教程》与《深入 Python: Dive Into Python 中文版》。

## 勘误和支持

由于笔者的水平有限，且编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为此，特意创建一个在线支持与应急方案问答站点：<http://qa.liuts.com>。你可以将书中的错误发布到“错误反馈”分类中，同时如果你遇到任何问题或有任何建议，也可以在问答站点中发表，我将尽量在线上提供最满意的解答。我也会将及时更新相应的功能更新。如果你有更多的宝贵意见，欢迎发送邮件至邮箱 [liutiansi@gmail.com](mailto:liutiansi@gmail.com)，期待能够得到你们的真挚反馈。

## 致谢

首先要感谢 Guido 大神，是他创立了 Python 语言，同时也要感谢提供 Python 优秀第三方模块的所有作者，开源的精神与力量在他们身上体现得淋漓尽致。

感谢钟总、王工、诗成兄，是他们给予我第一份工作，也为个人此后的成长提供了非常多的指导。感谢天涯社区的邢总（968）、王总（建科）、小军，是他们提供了这么优秀的平台，让我有机会可以尽情施展才能，体现个人价值。感谢腾讯的 Willim（崔晓春）、Tomxiao（肖志立）、Thundersun（孙雷）、Stanleysun（孙龙君）、Trackynong（农益辉）、Chanceli（李飞宏）、Blue（许明）导师，以及接入运维组（TEG）、数据管理组（IEG）所有兄弟姐妹在工作中给予的帮助、指导与支持，让我可以在新的环境继续突破自我，实现自我价值。感谢洪春兄（抚琴煮酒）的引荐，在他的努力下才促成了这本书的合作与出版。

感谢机械工业出版社的编辑杨福川和姜影，在这一年多的时间中始终支持我的写作，他们的鼓励和帮助引导我能顺利完成全部书稿。

感谢已经过世的爷爷，是他深深影响着我的人生观与价值观，他的教导我会永远铭记在心。感谢我的爸爸、妈妈，感谢他们将我培养成人，在成长的过程中不断鼓励、激励我继续

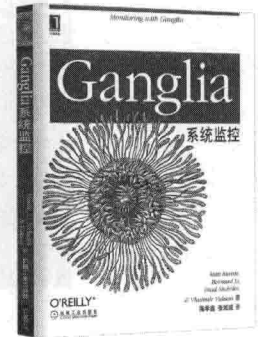
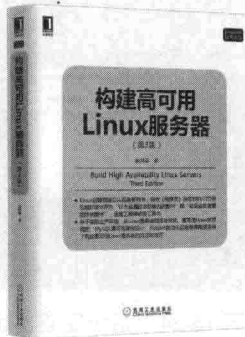
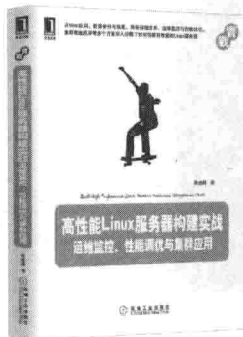
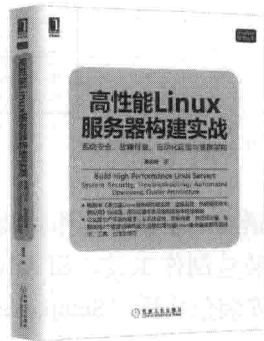
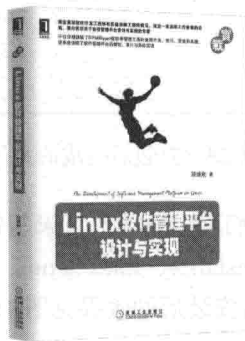
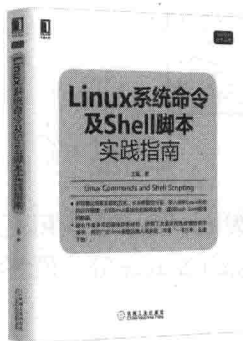
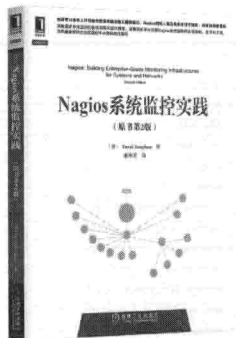
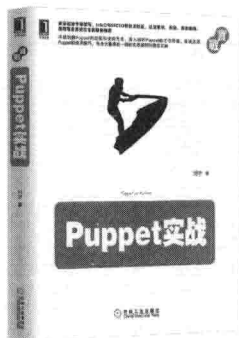
前进。感谢姐姐、弟弟，他们是我成长过程中最好的挚友与伙伴。

最后感谢我的爱人杜海英，没有你就没有我们幸福的小家和可爱的宝宝。感谢她支持我做的所有决定，没有她背后默默的支持与鼓励，也没有我今天的成就，更也不会有这本书。我想说：谢谢你！有你真好。

谨以此书献给我最亲爱的家人与我自己，以及众多热爱开源技术的朋友们！

刘天斯 (Yorkoliu)

## 推荐阅读



## Contents 目录

本书赞誉  
前言

### 第一部分 基础篇

第 1 章 系统基础信息模块详解	2
1.1 系统性能信息模块 psutil	2
1.1.1 获取系统性能信息	3
1.1.2 系统进程管理方法	6
1.2 实用的 IP 地址处理模块 IPy	7
1.2.1 IP 地址、网段的基本处理	8
1.2.2 多网络计算方法详解	9
1.3 DNS 处理模块 dnspython	11
1.3.1 模块域名解析方法详解	11
1.3.2 常见解析类型示例说明	12
1.3.3 实践：DNS 域名轮循业务监控	14
第 2 章 业务服务监控详解	17
2.1 文件内容差异对比方法	17
2.1.1 示例 1：两个字符串的差异对比	17
2.1.2 生成美观的对比 HTML 格式文档	19

2.1.3 示例 2: 对比 Nginx 配置文件差异 .....	19
2.2 文件与目录差异对比方法 .....	21
2.2.1 模块常用方法说明 .....	21
2.2.2 实践: 校验源与备份目录差异 .....	25
2.3 发送电子邮件模块 smtplib .....	27
2.3.1 smtplib 模块的常用类与方法 .....	27
2.3.2 定制个性化的邮件格式方法 .....	28
2.3.3 定制常用邮件格式示例详解 .....	29
2.4 探测 Web 服务质量方法 .....	34
2.4.1 模块常用方法说明 .....	35
2.4.2 实践: 实现探测 Web 服务质量 .....	36
<b>第 3 章 定制业务质量报表详解 .....</b>	<b>39</b>
3.1 数据报表之 Excel 操作模块 .....	39
3.1.1 模块常用方法说明 .....	41
3.1.2 实践: 定制自动化业务流量报表周报 .....	48
3.2 Python 与 rrdtool 的结合模块 .....	50
3.2.1 rrdtool 模块常用方法说明 .....	51
3.2.2 实践: 实现网卡流量图表绘制 .....	53
3.3 生成动态路由轨迹图 .....	56
3.3.1 模块常用方法说明 .....	56
3.3.2 实践: 实现 TCP 探测目标服务路由轨迹 .....	57
<b>第 4 章 Python 与系统安全 .....</b>	<b>60</b>
4.1 构建集中式的病毒扫描机制 .....	60
4.1.1 模块常用方法说明 .....	61
4.1.2 实践: 实现集中式的病毒扫描 .....	61
4.2 实现高效的端口扫描器 .....	64
4.2.1 模块常用方法说明 .....	64
4.2.2 实践: 实现高效的端口扫描 .....	66

## 第二部分 高级篇

<b>第 5 章 系统批量运维管理器 pexpect 详解</b>	70
5.1 pexpect 的安装	70
5.2 pexpect 的核心组件	71
5.2.1 spawn 类	71
5.2.2 run 函数	74
5.2.3 pxssh 类	75
5.3 pexpect 应用示例	76
5.3.1 实现一个自动化 FTP 操作	76
5.3.2 远程文件自动打包并下载	77
<b>第 6 章 系统批量运维管理器 paramiko 详解</b>	79
6.1 paramiko 的安装	79
6.2 paramiko 的核心组件	81
6.2.1 SSHClient 类	81
6.2.2 SFTPClient 类	82
6.3 paramiko 应用示例	85
6.3.1 实现密钥方式登录远程主机	85
6.3.2 实现堡垒机模式下的远程命令执行	85
6.3.3 实现堡垒机模式下的远程文件上传	88
<b>第 7 章 系统批量运维管理器 Fabric 详解</b>	91
7.1 Fabric 的安装	91
7.2 fab 的常用参数	92
7.3 fabfile 的编写	93
7.3.1 全局属性设定	93
7.3.2 常用 API	94
7.3.3 示例 1: 查看本地与远程主机信息	95
7.3.4 示例 2: 动态获取远程目录列表	96

7.3.5 示例 3: 网关模式文件上传与执行 .....	97
7.4 Fabric 应用示例 .....	98
7.4.1 示例 1: 文件打包、上传与校验 .....	98
7.4.2 示例 2: 部署 LNMP 业务服务环境 .....	99
7.4.3 示例 3: 生产环境代码包发布管理 .....	101
<b>第 8 章 从“零”开发一个轻量级 WebServer .....</b>	<b>104</b>
8.1 Yorserver 介绍 .....	104
8.1.1 功能特点 .....	104
8.1.2 配置文件 .....	105
8.2 功能实现方法 .....	106
8.2.1 HTTP 缓存功能 .....	107
8.2.2 HTTP 压缩功能 .....	111
8.2.3 HTTP SSL 功能 .....	111
8.2.4 目录列表功能 .....	114
8.2.5 动态 CGI 功能 .....	114
<b>第 9 章 集中化管理平台 Ansible 详解 .....</b>	<b>118</b>
9.1 YAML 语言 .....	119
9.1.1 块序列描述 .....	120
9.1.2 块映射描述 .....	120
9.2 Ansible 的安装 .....	121
9.2.1 业务环境说明 .....	121
9.2.2 安装 EPEL .....	122
9.2.3 安装 Ansible .....	122
9.2.4 Ansible 配置及测试 .....	122
9.2.5 配置 Linux 主机 SSH 无密码访问 .....	123
9.3 定义主机与组规则 .....	124
9.3.1 定义主机与组 .....	124
9.3.2 定义主机变量 .....	125
9.3.3 定义组变量 .....	125

9.3.4 分离主机与组特定数据 .....	126
9.4 匹配目标 .....	127
9.5 Ansible 常用模块及 API .....	127
9.6 playbook 介绍 .....	132
9.6.1 定义主机与用户 .....	132
9.6.2 任务列表 .....	133
9.6.3 执行 playbook .....	134
9.7 playbook 角色与包含声明 .....	135
9.7.1 包含文件, 鼓励复用 .....	135
9.7.2 角色 .....	136
9.8 获取远程主机系统信息: Facts .....	141
9.9 变量 .....	142
9.9.1 Jinja2 过滤器 .....	143
9.9.2 本地 Facts .....	143
9.9.3 注册变量 .....	144
9.10 条件语句 .....	145
9.11 循环 .....	146
9.12 示例讲解 .....	147
<b>第 10 章 集中化管理平台 Saltstack 详解 .....</b>	<b>155</b>
10.1 Saltstack 的安装 .....	156
10.1.1 业务环境说明 .....	156
10.1.2 安装 EPEL .....	156
10.1.3 安装 Saltstack .....	156
10.1.4 Saltstack 防火墙配置 .....	157
10.1.5 更新 Saltstack 配置及安装校验 .....	157
10.2 利用 Saltstack 远程执行命令 .....	158
10.3 Saltstack 常用模块及 API .....	161
10.4 grains 组件 .....	166
10.4.1 grains 常用操作命令 .....	167
10.4.2 定义 grains 数据 .....	167

10.5	pillar 组件	170
10.5.1	pillar 的定义	171
10.5.2	pillar 的使用	173
10.6	state 介绍	174
10.6.1	state 的定义	174
10.6.2	state 的使用	175
10.7	示例：基于 Saltstack 实现的配置集中化管理	177
10.7.1	环境说明	177
10.7.2	主控端配置说明	177
10.7.3	配置 pillar	179
10.7.4	配置 state	180
10.7.5	校验结果	183
<b>第 11 章</b>	<b>统一网络控制器 Func 详解</b>	<b>185</b>
11.1	Func 的安装	186
11.1.1	业务环境说明	186
11.1.2	安装 Func	186
11.2	Func 常用模块及 API	189
11.2.1	选择目标主机	190
11.2.2	常用模块详解	190
11.3	自定义 Func 模块	194
11.4	非 Python API 接口支持	198
11.5	Func 的 Facts 支持	199
<b>第 12 章</b>	<b>Python 大数据应用详解</b>	<b>202</b>
12.1	环境说明	202
12.2	Hadoop 部署	203
12.3	使用 Python 编写 MapReduce	207
12.3.1	用原生 Python 编写 MapReduce 详解	208
12.3.2	用 Mrjob 框架编写 MapReduce 详解	212
12.4	实战分析	216

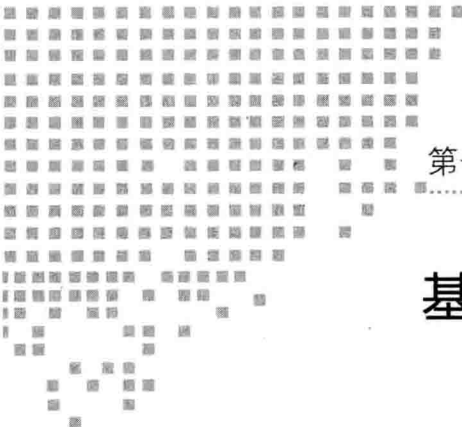
12.4.1 示例场景 .....	216
12.4.2 网站访问流量统计 .....	217
12.4.3 网站 HTTP 状态码统计 .....	219
12.4.4 网站分钟级请求数统计 .....	220
12.4.5 网站访问来源 IP 统计 .....	221
12.4.6 网站文件访问统计 .....	222

### 第三部分 案例篇

<b>第 13 章 从零开始打造 B/S 自动化运维平台 .....</b>	<b>226</b>
13.1 平台功能介绍 .....	226
13.2 系统构架设计 .....	227
13.3 数据库结构设计 .....	228
13.3.1 数据库分析 .....	228
13.3.2 数据字典 .....	228
13.3.3 数据库模型 .....	229
13.4 系统环境部署 .....	230
13.4.1 系统环境说明 .....	230
13.4.2 系统平台搭建 .....	230
13.4.3 开发环境优化 .....	233
13.5 系统功能模块设计 .....	235
13.5.1 前端数据加载模块 .....	235
13.5.2 数据传输模块设计 .....	237
13.5.3 平台功能模块扩展 .....	240
<b>第 14 章 打造 Linux 系统安全审计功能 .....</b>	<b>245</b>
14.1 平台功能介绍 .....	245
14.2 系统构架设计 .....	246
14.3 数据库结构设计 .....	247

14.3.1	数据库分析 .....	247
14.3.2	数据字典 .....	247
14.4	系统环境部署 .....	248
14.4.1	系统环境说明 .....	248
14.4.2	上报主机配置 .....	248
14.5	服务器端功能设计 .....	252
14.5.1	Django 配置 .....	252
14.5.2	功能实现方法 .....	253
<b>第 15 章</b>	<b>构建分布式质量监控平台 .....</b>	<b>256</b>
15.1	平台功能介绍 .....	256
15.2	系统构架设计 .....	257
15.3	数据库结构设计 .....	258
15.3.1	数据库分析 .....	258
15.3.2	数据字典 .....	258
15.3.3	数据库模型 .....	259
15.4	系统环境部署 .....	260
15.4.1	系统环境说明 .....	260
15.4.2	数据采集角色 .....	260
15.4.3	rrdtool 作业 .....	261
15.5	服务器端功能设计 .....	263
15.5.1	Django 配置 .....	263
15.5.2	业务增加功能 .....	264
15.5.3	业务报表功能 .....	266
<b>第 16 章</b>	<b>构建桌面版 C/S 自动化运维平台 .....</b>	<b>269</b>
16.1	平台功能介绍 .....	269
16.2	系统构架设计 .....	270
16.3	数据库结构设计 .....	271
16.3.1	数据库分析 .....	271
16.3.2	数据字典 .....	272

- 16.3.3 数据库模型 .....272
- 16.4 系统环境部署.....273
  - 16.4.1 系统环境说明 .....273
  - 16.4.2 系统环境搭建 .....273
- 16.5 系统功能模块设计.....274
  - 16.5.1 用户登录模块 .....274
  - 16.5.2 系统配置功能 .....275
  - 16.5.3 服务器分类模块 .....277
  - 16.5.4 系统升级功能 .....280
  - 16.5.5 客户端模块编写 .....284
  - 16.5.6 执行功能模块 .....287
  - 16.5.7 平台程序发布 .....289



## 第一部分 *Part 1*

# 基础篇

- 第 1 章 系统基础信息模块详解
- 第 2 章 业务服务监控详解
- 第 3 章 定制业务质量报表详解
- 第 4 章 Python 与系统安全

# 系统基础信息模块详解

系统基础信息采集模块作为监控模块的重要组成部分，能够帮助运维人员了解当前系统的健康程度，同时也是衡量业务的服务质量的依据，比如系统资源吃紧，会直接影响业务的服务质量及用户体验，另外获取设备的流量信息，也可以让运维人员更好地评估带宽、设备资源是否应该扩容。本章通过运用 Python 第三方系统基础模块，可以轻松获取服务关键运营指标数据，包括 Linux 基本性能、块设备、网卡接口、系统信息、网络地址库等信息。在采集到这些数据后，我们就可以全方位了解系统服务的状态，再结合告警机制，可以在第一时间响应，将异常出现在苗头时就得以处理。

本章通过具体的示例来帮助读者学习、理解并掌握。在本章接下来的内容当中，我们的示例将在一个连续的 Python 交互环境中进行。

进入 Python 终端，执行 python 命令进入交互式的 Python 环境，像这样：

```
# python
Python 2.6.6 (r266:84292, Nov 22 2013, 12:16:22)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

## 1.1 系统性能信息模块 psutil

psutil 是一个跨平台库 (<http://code.google.com/p/psutil/>)，能够轻松实现获取系统运行的

进程和系统利用率（包括 CPU、内存、磁盘、网络等）信息。它主要应用于系统监控，分析和限制系统资源及进程的管理。它实现了同等命令行工具提供的功能，如 ps、top、lsof、netstat、ifconfig、who、df、kill、free、nice、ionice、iostat、iotop、uptime、pidof、tty、taskset、pmap 等。目前支持 32 位和 64 位的 Linux、Windows、OS X、FreeBSD 和 Sun Solaris 等操作系统，支持从 2.4 到 3.4 的 Python 版本，目前最新版本为 2.0.0。通常我们获取操作系统信息往往采用编写 shell 来实现，如获取当前物理内存总大小及已使用大小，shell 命令如下：

```
物理内存 total 值：free -m | grep Mem | awk '{print $2}'
物理内存 used 值：free -m | grep Mem | awk '{print $3}'
```

相比较而言，使用 psutil 库实现则更加简单明了。psutil 大小单位一般都采用字节，如下：

```
>>> import psutil
>>> mem = psutil.virtual_memory()
>>> mem.total, mem.used
(506277888L, 500367360L)
```

psutil 的源码安装步骤如下：

```
# wget https://pypi.python.org/packages/source/p/psutil/psutil-2.0.0.tar.gz
--no-check-certificate
# tar -xzf psutil-2.0.0.tar.gz
# cd psutil-2.0.0
# python setup.py install
```

### 1.1.1 获取系统性能信息

采集系统的基本性能信息包括 CPU、内存、磁盘、网络等，可以完整描述当前系统的运行状态及质量。psutil 模块已经封装了这些方法，用户可以根据自身的应用场景，调用相应的方法来满足需求，非常简单实用。

#### (1) CPU 信息

Linux 操作系统的 CPU 利用率有以下几个部分：

- ❑ User Time，执行用户进程的时间百分比；
- ❑ System Time，执行内核进程和中断的时间百分比；
- ❑ Wait IO，由于 IO 等待而使 CPU 处于 idle（空闲）状态的时间百分比；
- ❑ Idle，CPU 处于 idle 状态的时间百分比。

我们使用 Python 的 psutil.cpu\_times() 方法可以非常简单地得到这些信息，同时也可以获取 CPU 的硬件相关信息，比如 CPU 的物理个数与逻辑个数，具体见下面的操作例子：

```
>>> import psutil
>>> psutil.cpu_times() # 使用 cpu_times 方法获取 CPU 完整信息, 需要显示所有逻辑 CPU 信息,
>>> # 指定方法变量 percpu=True 即可, 如 psutil.cpu_times(percpu=True)
scputimes(user=38.039999999999999, nice=0.01, system=110.88, idle=177062.59,
iowait=53.399999999999999, irq=2.9100000000000001, softirq=79.579999999999998,
steal=0.0, guest=0.0)
>>> psutil.cpu_times().user      # 获取单项数据信息, 如用户 user 的 CPU 时间比
38.0
>>> psutil.cpu_count()          # 获取 CPU 的逻辑个数, 默认 logical=True
4
>>> psutil.cpu_count(logical=False) # 获取 CPU 的物理个数
2
>>>
```

## (2) 内存信息

Linux 系统的内存利用率信息涉及 total (内存总数)、used (已使用的内存数)、free (空闲内存数)、buffers (缓冲使用数)、cache (缓存使用数)、swap (交换分区使用数) 等, 分别使用 psutil.virtual\_memory() 与 psutil.swap\_memory() 方法获取这些信息, 具体见下面的操作例子:

```
>>> import psutil
>>> mem = psutil.virtual_memory() # 使用 psutil.virtual_memory 方法获取内存完整信息
>>> mem
svmem(total=506277888L, available=204951552L, percent=59.5, used=499867648L,
free=6410240L, active=245858304, inactive=163733504, buffers=117035008L,
cached=81506304)
>>> mem.total      # 获取内存总数
506277888L
>>> mem.free       # 获取空闲内存数
6410240L
>>> psutil.swap_memory() # 获取 SWAP 分区信息
sswap(total=1073733632L, used=0L,
free=1073733632L, percent=0.0, sin=0, sout=0)
>>>
```

## (3) 磁盘信息

在系统的所有磁盘信息中, 我们更加关注磁盘的利用率及 IO 信息, 其中磁盘利用率使用 psutil.disk\_usage 方法获取。磁盘 IO 信息包括 read\_count (读 IO 数)、write\_count (写 IO 数)、read\_bytes (IO 读字节数)、write\_bytes (IO 写字节数)、read\_time (磁盘读时间)、write\_time (磁盘写时间) 等。这些 IO 信息可以使用 psutil.disk\_io\_counters() 获取, 具体见下面的操作例子:

```
>>> psutil.disk_partitions() # 使用 psutil.disk_partitions 方法获取磁盘完整信息
[sdiskpart(device='/dev/sda1', mountpoint='/', fstype='ext4', opts='rw'),
sdiskpart(device='/dev/sda3', mountpoint='/data', fstype='ext4', opts='rw')]
>>>
>>> psutil.disk_usage('/') # 使用 psutil.disk_usage 方法获取分区 (参数) 的使用情况
```

```

sdiskusage(total=15481577472, used=4008087552, free=10687057920,
percent=25.899999999999999)
>>>
>>>psutil.disk_io_counters()      # 使用 psutil.disk_io_counters 获取硬盘总的 IO 个数、
                                # 读写信息
sdiskio(read_count=9424, write_count=35824, read_bytes=128006144, write_
bytes=204312576, read_time=72266, write_time=182485)
>>>
>>>psutil.disk_io_counters(perdisk=True) # “perdisk=True” 参数获取单个分区 IO 个数、
                                # 读写信息
{'sda2': sdiskio(read_count=322, write_count=0, read_bytes=1445888, write_
bytes=0, read_time=445, write_time=0), 'sda3': sdiskio(read_count=618, write_
count=3, read_bytes=2855936, write_bytes=12288, read_time=871, write_time=155),
'sda1': sdiskio(read_count=8484, write_count=35821, read_bytes=123704320,
write_bytes=204300288, read_time=70950, write_time=182330)}

```

#### (4) 网络信息

系统的网络信息与磁盘 IO 类似，涉及几个关键点，包括 bytes\_sent（发送字节数）、bytes\_recv=28220119（接收字节数）、packets\_sent=200978（发送数据包数）、packets\_recv=212672（接收数据包数）等。这些网络信息使用 psutil.net\_io\_counters() 方法获取，具体见下面的操作例子：

```

>>>psutil.net_io_counters()      # 使用 psutil.net_io_counters 获取网络总的 IO 信息，默
                                # 认 pernic=False
snetio(bytes_sent=27098178, bytes_recv=28220119, packets_sent=200978, packets_
recv=212672, errin=0, errout=0, dropin=0, dropout=0)
>>>psutil.net_io_counters(pernic=True) # pernic=True 输出每个网络接口的 IO 信息
{'lo': snetio(bytes_sent=26406824, bytes_recv=26406824, packets_sent=198526,
packets_recv=198526, errin=0, errout=0, dropin=0, dropout=0), 'eth0':
snetio(bytes_sent=694750, bytes_recv=1816743, packets_sent=2478, packets_
recv=14175, errin=0, errout=0, dropin=0, dropout=0)}
>>>

```

#### (5) 其他系统信息

除了前面介绍的几个获取系统基本信息的方法，psutil 模块还支持获取用户登录、开机时间等信息，具体见下面的操作例子：

```

>>>psutil.users()      # 使用 psutil.users 方法返回当前登录系统的用户信息
[user(name='root', terminal='pts/0', host='192.168.1.103',
started=1394638720.0), user(name='root', terminal='pts/1',
host='192.168.1.103', started=1394723840.0)]
>>> import psutil, datetime
>>>psutil.boot_time()      # 使用 psutil.boot_time 方法获取开机时间，以 Linux 时间戳格式返回
1389563460.0
>>>datetime.datetime.fromtimestamp(psutil.boot_time()).strftime("%Y-%m-%d
%H:%M:%S")

```

```
'2014-01-12 22:51:00' # 转换成自然时间格式
```

### 1.1.2 系统进程管理方法

获得当前系统的进程信息，可以让运维人员得知应用程序的运行状态，包括进程的启动时间、查看或设置 CPU 亲和度、内存使用率、IO 信息、socket 连接、线程数等，这些信息可以呈现出指定进程是否存活、资源利用情况，为开发人员的代码优化、问题定位提供很好的数据参考。

#### (1) 进程信息

psutil 模块在获取进程信息方面也提供了很好的支持，包括使用 psutil.pids() 方法获取所有进程 PID，使用 psutil.Process() 方法获取单个进程的名称、路径、状态、系统资源利用率等信息，具体见下面的操作例子：

```
>>> import psutil
>>> psutil.pids() # 列出所有进程 PID
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19.....]
>>> p = psutil.Process(2424) # 实例化一个 Process 对象，参数为一进程 PID
>>> p.name() # 进程名
'java'
>>> p.exe() # 进程 bin 路径
'/usr/java/jdk1.6.0_45/bin/java'
>>> p.cwd() # 进程工作目录绝对路径
'/usr/local/hadoop-1.2.1'
>>> p.status() # 进程状态
'sleeping'
>>> p.create_time() # 进程创建时间，时间戳格式
1394852592.6900001
>>> p.uids() # 进程 uid 信息
puids(real=0, effective=0, saved=0)
>>> p.gids() # 进程 gid 信息
pgids(real=0, effective=0, saved=0)
>>> p.cpu_times() # 进程 CPU 时间信息，包括 user、system 两个 CPU 时间
pcputimes(user=9.050000000000007, system=20.25)
>>> p.cpu_affinity() # get 进程 CPU 亲和度，如要设置进程 CPU 亲和度，将 CPU 号作为参数即可
[0, 1]
>>> p.memory_percent() # 进程内存利用率
14.147714861289776
>>> p.memory_info() # 进程内存 rss、vms 信息
pmem(rss=71626752, vms=1575665664)
>>> p.io_counters() # 进程 IO 信息，包括读写 IO 数及字节数
pio(read_count=41133, write_count=16811, read_bytes=37023744, write_bytes=4722688)
>>> p.connections() # 返回打开进程 socket 的 namedutples 列表，包括 fs、family、laddr 等信息
[pconn(fd=65, family=10, type=1, laddr=('::ffff:192.168.1.20', 9000),
```

```

raddr=(),.....]
>>>p.num_threads()    # 进程开启的线程数
33

```

## (2) popen 类的使用

psutil 提供的 popen 类的作用是获取用户启动的应用程序进程信息，以便跟踪程序进程的运行状态。具体实现方法如下：

```

>>> import psutil
>>> from subprocess import PIPE
# 通过 psutil 的 Popen 方法启动的应用程序，可以跟踪该程序运行的所有相关信息
>>> p = psutil.Popen(["/usr/bin/python", "-c", "print('hello')"], stdout=PIPE)
>>> p.name()
'python'
>>> p.username()
'root'
>>> p.communicate()
('hello\n', None)
>>> p.cpu_times()    # 得到进程运行的 CPU 时间，更多方法见上一小节
pcputimes(user=0.01, system=0.040000000000000001)

```



参考提示

□ 1.1.1 节示例参考 <https://github.com/giampaolo/psutil>。

□ 1.1.1 节模块说明参考官网 <http://psutil.readthedocs.org/en/latest/>。

## 1.2 实用的 IP 地址处理模块 IPy

IP 地址规划是网络设计中非常重要的一个环节，规划的好坏会直接影响路由协议算法的效率，包括网络性能、可扩展性等方面，在这个过程当中，免不了要计算大量的 IP 地址，包括网段、网络掩码、广播地址、子网数、IP 类型等。Python 提供了一个强大的第三方模块 IPy (<https://github.com/haypo/python-ipy/>)，最新版本为 V0.81。IPy 模块可以很好地辅助我们高效完成 IP 的规划工作，下面进行详细介绍。

以下是 IPy 模块的安装，这里采用源码的安装方式：

```

# wget https://pypi.python.org/packages/source/I/IPy/IPy-0.81.tar.gz --no-check-certificate
# tar -zxvf IPy-0.81.tar.gz
# cd IPy-0.81
# python setup.py install

```

## 1.2.1 IP 地址、网段的基本处理

IPy 模块包含 IP 类，使用它可以方便处理绝大部分格式为 IPv6 及 IPv4 的网络和地址。比如通过 version 方法就可以区分出 IPv4 与 IPv6，如：

```
>>>IP('10.0.0.0/8').version()
4      #4 代表 IPv4 类型
>>>IP('::1').version()
6      #6 代表 IPv6 类型
```

通过指定的网段输出该网段的 IP 个数及所有 IP 地址清单，代码如下：

```
from IPy import IP
ip = IP('192.168.0.0/16')
print ip.len()      # 输出 192.168.0.0/16 网段的 IP 个数
for x in ip:        # 输出 192.168.0.0/16 网段的所有 IP 清单
    print(x)
```

执行结果如下：

```
65536
192.168.0.0
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4
192.168.0.5
192.168.0.6
192.168.0.7
192.168.0.8
.....
```

下面介绍 IP 类几个常见的方法，包括反向解析名称、IP 类型、IP 转换等。

```
>>>from IPy import IP
>>>ip = IP('192.168.1.20')
>>>ip.reverseNames()      # 反向解析地址格式
['20.1.168.192.in-addr.arpa.']
>>>ip.iptype()             #192.168.1.20 为私网类型 'PRIVATE'
>>> IP('8.8.8.8').iptype()  #8.8.8.8 为公网类型
'PUBLIC'
>>> IP("8.8.8.8").int()    # 转换成整型格式
134744072
>>> IP('8.8.8.8').strHex()  # 转换成十六进制格式
'0x8080808'
>>> IP('8.8.8.8').strBin()  # 转换成二进制格式
'0000100000000100000000100000001000'
>>> print(IP(0x8080808))    # 十六进制转成 IP 格式
8.8.8.8
```

IP 方法也支持网络地址的转换，例如根据 IP 与掩码生产网段格式，如下：

```
>>>from IPy import IP
>>>print(IP('192.168.1.0').make_net('255.255.255.0'))
192.168.1.0/24
>>>print(IP('192.168.1.0/255.255.255.0', make_net=True))
192.168.1.0/24
>>>print(IP('192.168.1.0-192.168.1.255', make_net=True))
192.168.1.0/24
```

也可以通过 `strNormal` 方法指定不同 `wantprefixlen` 参数值以定制不同输出类型的网段。输出类型为字符串，如下：

```
>>>IP('192.168.1.0/24').strNormal(0)
'192.168.1.0'
>>>IP('192.168.1.0/24').strNormal(1)
'192.168.1.0/24'
>>>IP('192.168.1.0/24').strNormal(2)
'192.168.1.0/255.255.255.0'
>>>IP('192.168.1.0/24').strNormal(3)
'192.168.1.0-192.168.1.255'
```

`wantprefixlen` 的取值及含义：

- ❑ `wantprefixlen = 0`，无返回，如 192.168.1.0；
- ❑ `wantprefixlen = 1`，prefix 格式，如 192.168.1.0/24；
- ❑ `wantprefixlen = 2`，decimalnetmask 格式，如 192.168.1.0/255.255.255.0；
- ❑ `wantprefixlen = 3`，lastIP 格式，如 192.168.1.0-192.168.1.255。

## 1.2.2 多网络计算方法详解

有时候我们想比较两个网段是否存在包含、重叠等关系，比如同网络但不同 `prefixlen` 会认为是不相等的网段，如 10.0.0.0/16 不等于 10.0.0.0/24，另外即使具有相同的 `prefixlen` 但处于不同的网络地址，同样也视为不相等，如 10.0.0.0/16 不等于 192.0.0.0/16。IPy 支持类似于数值型数据的比较，以帮助 IP 对象进行比较，如：

```
>>>IP('10.0.0.0/24') < IP('12.0.0.0/24')
True
```

判断 IP 地址和网段是否包含于另一个网段中，如下：

```
>>>'192.168.1.100' in IP('192.168.1.0/24')
True
>>>IP('192.168.1.0/24') in IP('192.168.0.0/16')
True
```

判断两个网段是否存在重叠，采用 IPy 提供的 `overlaps` 方法，如：

```
>>>IP('192.168.0.0/23').overlaps('192.168.1.0/24')
1      # 返回 1 代表存在重叠
>>>IP('192.168.1.0/24').overlaps('192.168.2.0')
0      # 返回 0 代表不存在重叠
```

**示例** 根据输入的 IP 或子网返回网络、掩码、广播、反向解析、子网数、IP 类型等信息。

```
#!/usr/bin/env python
from IPy import IP

ip_s = raw_input('Please input an IP or net-range: ')      # 接收用户输入，参数为 IP
地址或网段地址
ips = IP(ip_s)
if len(ips) > 1:      # 为一个网络地址
    print('net: %s' % ips.net())      # 输出网络地址
    print('netmask: %s' % ips.netmask())      # 输出网络掩码地址
    print('broadcast: %s' % ips.broadcast())      # 输出网络广播地址
    print('reverse address: %s' % ips.reverseNames()[0])      # 输出地址反向解析
    print('subnet: %s' % len(ips))      # 输出网络子网数
else:      # 为单个 IP 地址
    print('reverse address: %s' % ips.reverseNames()[0])      # 输出 IP 反向解析

print('hexadecimal: %s' % ips.strHex())      # 输出十六进制地址
print('binary ip: %s' % ips.strBin())      # 输出二进制地址
print('iptype: %s' % ips.iptype())      # 输出地址类型，如 PRIVATE、PUBLIC、LOOPBACK 等
```

分别输入网段、IP 地址的运行返回结果如下：

```
# python simple1.py
Please input an IP or net-range: 192.168.1.0/24
net: 192.168.1.0
netmask: 255.255.255.0
broadcast: 192.168.1.255
reverse address: 1.168.192.in-addr.arpa.
subnet: 256
hexadecimal: 0xc0a80100
binaryip: 11000000101010000000000100000000
iptype: PRIVATE

# python simple1.py
Please input an IP or net-range: 192.168.1.20
reverse address: 20.1.168.192.in-addr.arpa.
hexadecimal: 0xc0a80114
binaryip: 11000000101010000000000100010100
iptype: PRIVATE
```



参考提示

- 1.2.1 节官网文档与示例参考 <https://github.com/haypo/python-ipy/>。
- 1.2.2 节 示例 1 参考 <http://blog.philippklaus.de/2012/12/ip-address-analysis-using-python/> 和 [http://www.sourcecodebrowser.com/ipy/0.62/class\\_ipy\\_1\\_1\\_ipint.html](http://www.sourcecodebrowser.com/ipy/0.62/class_ipy_1_1_ipint.html) 等文章的 IPy 类说明。
- 书中涉及的所有示例及源码的 Github 地址为 <https://github.com/yorkoliu/pyauto>。

## 1.3 DNS 处理模块 dnspython

dnspython (<http://www.dnspython.org/>) 是 Python 实现的一个 DNS 工具包, 它支持几乎所有的记录类型, 可以用于查询、传输并动态更新 ZONE 信息, 同时支持 TSIG (事务签名) 验证消息和 EDNS0 (扩展 DNS)。在系统管理方面, 我们可以利用其查询功能来实现 DNS 服务监控以及解析结果的校验, 可以代替 nslookup 及 dig 等工具, 轻松做到与现有平台的整合, 下面进行详细介绍。

首先介绍 dnspython 模块的安装, 这里采用源码的安装方式, 最新版本为 1.9.4, 如下:

```
# http://www.dnspython.org/kits/1.9.4/dnspython-1.9.4.tar.gz
# tar -zxvf dnspython-1.9.4.tar.gz
# cd dnspython-1.9.4
# python setup.py install
```

### 1.3.1 模块域名解析方法详解

dnspython 模块提供了大量的 DNS 处理方法, 最常用的方法是域名查询。dnspython 提供了一个 DNS 解析器类——resolver, 使用它的 query 方法来实现域名的查询功能。query 方法的定义如下:

```
query(self, qname, rdtype=1, rdclass=1, tcp=False, source=None, raise_on_no_answer=True, source_port=0)
```

其中, qname 参数为查询的域名。rdtype 参数用来指定 RR 资源的类型, 常用的有以下几种:

- A 记录, 将主机名转换成 IP 地址;
- MX 记录, 邮件交换记录, 定义邮件服务器的域名;
- CNAME 记录, 指别名记录, 实现域名间的映射;
- NS 记录, 标记区域的域名服务器及授权子域;
- PTR 记录, 反向解析, 与 A 记录相反, 将 IP 转换成主机名;
- SOA 记录, SOA 标记, 一个起始授权区的定义。

rdclass 参数用于指定网络类型，可选的值有 IN、CH 与 HS，其中 IN 为默认，使用最广泛。tcp 参数用于指定查询是否启用 TCP 协议，默认为 False（不启用）。source 与 source\_port 参数作为指定查询源地址与端口，默认值为查询设备 IP 地址和 0。raise\_on\_no\_answer 参数用于指定当查询无应答时是否触发异常，默认为 True。

### 1.3.2 常见解析类型示例说明

常见的 DNS 解析类型包括 A、MX、NS、CNAME 等。利用 dnspython 的 dns.resolver.query 方法可以简单实现这些 DNS 类型的查询，为后面要实现的功能提供数据来源，比如对一个使用 DNS 轮循业务的域名进行可用性监控，需要得到当前的解析结果。下面一一进行介绍。

#### (1) A 记录

实现 A 记录查询方法源码。

【 /home/test/dnspython/simple1.py 】

```
#!/usr/bin/env python
import dns.resolver

domain = raw_input('Please input an domain: ')    # 输入域名地址
A = dns.resolver.query(domain, 'A')             # 指定查询类型为 A 记录
for i in A.response.answer:                     # 通过 response.answer 方法获取查询响应信息
    for j in i.items:                           # 遍历响应信息
        printj.address
```

运行代码查看结果，这里以 www.google.com 域名为例：

```
# python simple1.py
Please input an domain: www.google.com
173.194.127.180
173.194.127.178
173.194.127.176
173.194.127.179
173.194.127.177
```

#### (2) MX 记录

实现 MX 记录查询方法源码。

【 /home/test/dnspython/ simple2.py 】

```
#!/usr/bin/env python
import dns.resolver
```

```

domain = raw_input('Please input an domain: ')
MX = dns.resolver.query(domain, 'MX')    # 指定查询类型为 MX 记录
for i in MX:    # 遍历响应结果, 输出 MX 记录的 preference 及 exchanger 信息
    print 'MX preference =', i.preference, 'mail exchanger =', i.exchange

```

运行代码查看结果, 这里以 163.com 域名为例:

```

# python simple2.py
Please input an domain: 163.com
MX preference = 10 mail exchanger = 163mx03.mxmail.netease.com.
MX preference = 50 mail exchanger = 163mx00.mxmail.netease.com.
MX preference = 10 mail exchanger = 163mx01.mxmail.netease.com.
MX preference = 10 mail exchanger = 163mx02.mxmail.netease.com.

```

### (3) NS 记录

实现 NS 记录查询方法源码。

【 /home/test/dnspython/ simple3.py 】

```

#!/usr/bin/env python
import dns.resolver

domain = raw_input('Please input an domain: ')
ns = dns.resolver.query(domain, 'NS')    # 指定查询类型为 NS 记录
for i in ns.response.answer:
    for j in i.items:
        print j.to_text()

```

只限输入一级域名, 如 baidu.com。如果输入二级或多级域名, 如 www.baidu.com, 则是错误的。

```

# python simple3.py
Please input an domain: baidu.com
ns4.baidu.com.
dns.baidu.com.
ns2.baidu.com.
ns7.baidu.com.
ns3.baidu.com.

```

### (4) CNAME 记录

实现 CNAME 记录查询方法源码。

【 /home/test/dnspython/ simple4.py 】

```

#!/usr/bin/env python
import dns.resolver

```

```

domain = raw_input('Please input an domain: ')
cname = dns.resolver.query(domain, 'CNAME') # 指定查询类型为 CNAME 记录
for i in cname.response.answer: # 结果将返回 cname 后的目标域名
    for j in i.items:
        print j.to_text()

```

结果将返回 cname 后的目标域名。

### 1.3.3 实践：DNS 域名轮循业务监控

大部分的 DNS 解析都是一个域名对应一个 IP 地址，但是通过 DNS 轮循技术可以做到一个域名对应多个 IP，从而实现最简单且高效的负载均衡，不过此方案最大的弊端是目标主机不可用时无法被自动剔除，因此做好业务主机的服务可用监控至关重要。本示例通过分析当前域名的解析 IP，再结合服务端口探测来实现自动监控，在域名解析中添加、删除 IP 时，无须对监控脚本进行更改。实现架构图如图 1-1 所示。

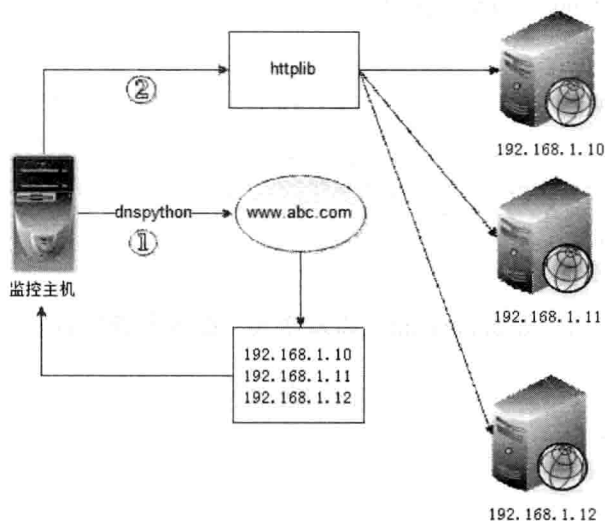


图 1-1 DNS 多域名业务服务监控架构图

#### 1. 步骤

- 1) 实现域名的解析，获取域名所有的 A 记录解析 IP 列表；
- 2) 对 IP 列表进行 HTTP 级别的探测。

#### 2. 代码解析

本示例第一步通过 `dns.resolver.query()` 方法获取业务域名 A 记录信息，查询出所有 IP 地

址列表，再使用 `httplib` 模块的 `request()` 方法以 GET 方式请求监控页面，监控业务所有服务的 IP 是否服务正常。

【 /home/test/dnspython/simple5.py 】

```
#!/usr/bin/python
import dns.resolver
import os
import httplib

iplist=[]      # 定义域名 IP 列表变量
appdomain="www.google.com.hk"    # 定义业务域名

def get_iplist(domain=""):      # 域名解析函数，解析成功 IP 将被追加到 iplist
    try:
        A = dns.resolver.query(domain, 'A')      # 解析 A 记录类型
    except Exception,e:
        print "dns resolver error:"+str(e)
        return
    for i in A.response.answer:
        for j in i.items:
            iplist.append(j.address)      # 追加到 iplist
    return True

def checkip(ip):
    checkurl=ip+":80"
    getcontent=""
    httplib.socket.setdefaulttimeout(5)      # 定义 http 连接超时时间 (5 秒)
    conn=httplib.HTTPConnection(checkurl)      # 创建 http 连接对象

    try:
        conn.request("GET", "/",headers = {"Host": appdomain})      # 发起 URL 请求，添
                                                                    # 加 host 主机头

        r=conn.getresponse()
        getcontent =r.read(15)      # 获取 URL 页面前 15 个字符，以便做可用性校验
    finally:
        if getcontent=="<!doctype html>":      # 监控 URL 页的内容一般是事先定义好的，比如
                                                                    # "HTTP200" 等
            print ip+" [OK]"
        else:
            print ip+" [Error]"      # 此处可放告警程序，可以是邮件、短信通知

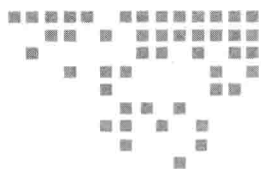
if __name__=="__main__":
    if get_iplist(appdomain) and len(iplist)>0:      # 条件：域名解析正确且至少返回一个 IP
        for ip in iplist:
            checkip(ip)
    else:
        print "dns resolver error."
```

我们可以将此脚本放到 `crontab` 中定时运行，再结合告警程序，这样一个基于域名轮循

的业务监控已完成。运行程序，显示结果如下：

```
# python simple5.py
74.125.31.94 [OK]
74.125.128.199 [OK]
173.194.72.94 [OK]
```

从结果可以看出，域名 `www.google.com.hk` 解析出 3 个 IP 地址，并且服务都是正常的。



## 业务服务监控详解

业务服务监控是运维体系中最重要的一环，是保证业务服务质量的关键手段。如何更有效地实现业务服务，是每个运维人员应该思考的问题，不同业务场景需定制不同的监控策略。Python 在监控方面提供了大量的第三方工具，可以帮助我们快速、有效地开发企业级服务监控平台，为我们的业务保驾护航。本章涉及文件与目录差异对比方法、HTTP 质量监控、邮件告警等内容。

### 2.1 文件内容差异对比方法

本节介绍如何通过 `difflib` 模块实现文件内容差异对比。`difflib` 作为 Python 的标准库模块，无需安装，作用是对比文本之间的差异，且支持输出可读性比较强的 HTML 文档，与 Linux 下的 `diff` 命令相似。我们可以使用 `difflib` 对比代码、配置文件的差别，在版本控制方面是非常有用。Python 2.3 或更高版本默认自带 `difflib` 模块，无需额外安装，我们先通过一个简单的示例进行了解。

#### 2.1.1 示例 1：两个字符串的差异对比

本示例通过使用 `difflib` 模块实现两个字符串的差异对比，然后以版本控制风格进行输出。

【 /home/test/diffib/simple1.py 】

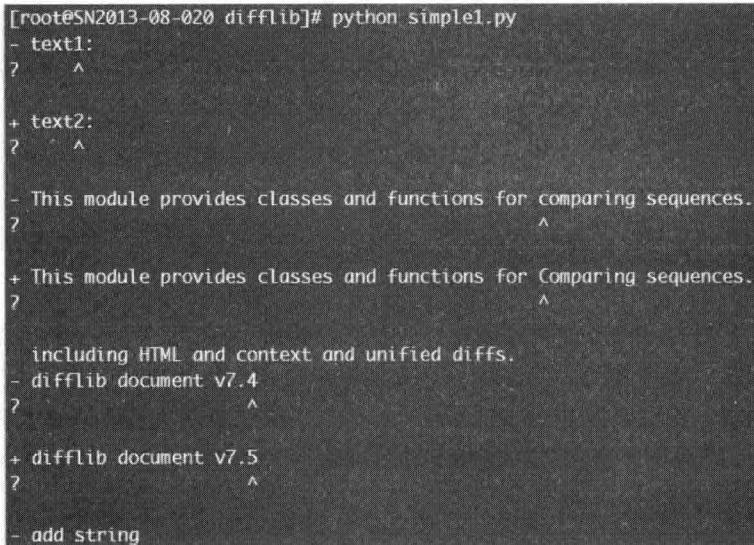
```
#!/usr/bin/python
```

```

import difflib
text1 = """text1:    # 定义字符串 1
This module provides classes and functions for comparing sequences.
including HTML and context and unified diffs.
difflib document v7.4
add string
"""
text1_lines = text1.splitlines()    # 以行进行分隔，以便进行对比
text2 = """text2:    # 定义字符串 2
This module provides classes and functions for Comparing sequences.
including HTML and context and unified diffs.
difflib document v7.5"""
text2_lines = text2.splitlines()
d = difflib.Differ()    # 创建 Differ() 对象
diff = d.compare(text1_lines, text2_lines)    # 采用 compare 方法对字符串进行比较
print '\n'.join(list(diff))

```

本示例采用 Differ() 类对两个字符串进行比较，另外 difflib 的 SequenceMatcher() 类支持任意类型序列的比较，HtmlDiff() 类支持将比较结果输出为 HTML 格式，示例运行结果如图 2-1 所示。



```

[root@SN2013-08-020 difflib]# python simple1.py
- text1:
?   ^

+ text2:
?   ^

- This module provides classes and functions for comparing sequences.
?                                     ^

+ This module provides classes and functions for Comparing sequences.
?                                     ^

- including HTML and context and unified diffs.
- difflib document v7.4
?   ^

+ difflib document v7.5
?   ^

- add string

```

图 2-1 示例运行结果

为方便大家理解差异关系符号，表 2-1 对各符号含义进行说明。

表 2-1 符号含义说明

符号	含义
' '	包含在第一个序列行中, 但不包含在第二个序列行
'+'	包含在第二个序列行中, 但不包含在第一个序列行
' '	两个序列行一致
'?'	标志两个序列行存在增量差异
'^'	标志出两个序列行存在的差异字符

### 2.1.2 生成美观的对比 HTML 格式文档

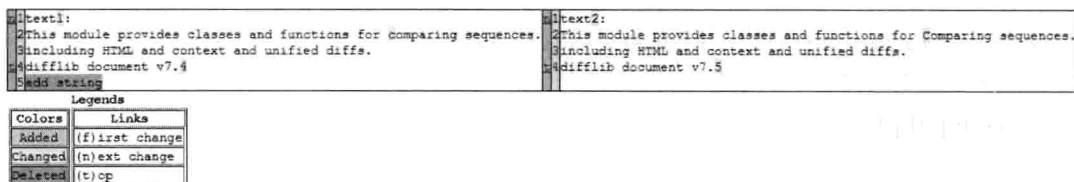
采用 `HtmlDiff()` 类的 `make_file()` 方法就可以生成美观的 HTML 文档, 对示例 1 中代码按以下进行修改:

```
d = difflib.Differ()
diff = d.compare(text1_lines, text2_lines)
print '\n'.join(list(diff))
```

替换成:

```
d = difflib.HtmlDiff()
print d.make_file(text1_lines, text2_lines)
```

将新文件命名为 `simple2.py`, 运行 `# python simple2.py > diff.html`, 再使用浏览器打开 `diff.html` 文件, 结果如图示 2-2 所示, HTML 文档包括了行号、差异标志、图例等信息, 可读性增强了许多。

图 2-2 在浏览器中打开 `diff.html` 文件

### 2.1.3 示例 2: 对比 Nginx 配置文件差异

当我们维护多个 Nginx 配置时, 时常会对比不同版本配置文件的差异, 使运维人员更加清晰地了解不同版本迭代后的更新项, 实现的思路是读取两个需对比的配置文件, 再以换行符作为分隔符, 调用 `difflib.HtmlDiff()` 生成 HTML 格式的差异文档。实现代码如下:

【 /home/test/difflib/simple3.py 】

```
#!/usr/bin/python
import difflib
import sys

try:
    textfile1=sys.argv[1]    # 第一个配置文件路径参数
    textfile2=sys.argv[2]    # 第二个配置文件路径参数
except Exception,e:
    print "Error:"+str(e)
    print "Usage: simple3.py filename1 filename2"
    sys.exit()

def readfile(filename):    # 文件读取分隔函数
    try:
        fileHandle = open (filename, 'rb' )
        text=fileHandle.read().splitlines()    # 读取后以行进行分隔
        fileHandle.close()
        return text
    except IOError as error:
        print('Read file Error:'+str(error))
        sys.exit()

if textfile1=="" or textfile2=="":
    print "Usage: simple3.py filename1 filename2"
    sys.exit()

text1_lines = readfile(textfile1)    # 调用 readfile 函数，获取分隔后的字符串
text2_lines = readfile(textfile2)

d = difflib.HtmlDiff()    # 创建 HtmlDiff() 类对象
print d.make_file(text1_lines, text2_lines)    # 通过 make_file 方法输出 HTML 格式的比对结果
```

运行如下代码：

```
# python simple3.py nginx.conf.v1 nginx.conf.v2 > diff.html
```

从图 2-3 中可以看出 nginx.conf.v1 与 nginx.conf.v2 配置文件存在的差异。



参考  
提示

2.1 节示例参考官网文档 <http://docs.python.org/2/library/difflib.html>。

1# For more information on configuration, see: 2# * Official English Documentation: http://nginx.org/en/docs/ 3# * Official Russian Documentation: http://nginx.org/ru/docs/ 4 5user nginx; 6worker_processes 1; 7 8error_log /var/log/nginx/error.log; 9error_log /var/log/nginx/error.log notice; 10error_log /var/log/nginx/error.log info; 11 12pid /var/run/nginx.pid; 13 14 15events { 16 worker_connections 1024; 17} 18 19 20http { 21 include /etc/nginx/mime.types; 22 default_type application/octet-stream; 23 24 log_format main '\$remote_addr - \$remote_user [\$time_local] "\$request" 25 "\$status \$body_bytes_sent "\$http_referer" 26 "\$http_user_agent" "\$http_x_forwarded_for"; 27 28 access_log /var/log/nginx/access.log main; 29 30 sendfile on; 31 #tcp_nopush on; 32 33 #keepalive_timeout 0; 34 keepalive_timeout 65; 35 36 #gzip on; 37 38 # Load config files from the /etc/nginx/conf.d directory 39 # The default server is in conf.d/default.conf 40 include /etc/nginx/conf.d/*.conf; 41 42	1# For more information on configuration, see: 2# * Official English Documentation: http://nginx.org/en/docs/ 3# * Official Russian Documentation: http://nginx.org/ru/docs/ 4 5user nginx; 6worker_processes 4; 7 8error_log /var/log/nginx/error.log; 9error_log /data/log/nginx/error.log notice; 10error_log /data/log/nginx/error.log info; 11 12pid /var/run/nginx.pid; 13 14 15events { 16 worker_connections 51200; 17} 18 19 20http { 21 include /etc/nginx/mime.types; 22 default_type application/octet-stream; 23 24 log_format main '\$remote_addr - \$remote_user [\$time_local] "\$request" 25 "\$status \$body_bytes_sent "\$http_referer" 26 "\$http_user_agent" "\$http_x_forwarded_for"; 27 28 access_log /data/log/nginx/access.log main; 29 30 sendfile on; 31 #tcp_nopush on; 32 33 #keepalive_timeout 0; 34 keepalive_timeout 65; 35 36 gzip on; 37 38 # Load config files from the /etc/nginx/conf.d directory 39 # The default server is in conf.d/default.conf 40 include /etc/nginx/conf.d/*.conf; 41 42
---	--

图 2-3 nginx.conf.v1 与 nginx.conf.v2 配置文件对比结果

## 2.2 文件与目录差异对比方法

当我们进行代码审计或校验备份结果时，往往需要检查原始与目标目录的文件一致性，Python 的标准库已经自带了满足此需求的模块 `filecmp`。`filecmp` 可以实现文件、目录、遍历子目录的差异对比功能。比如报告中输出目标目录比原始多出的文件或子目录，即使文件同名也会判断是否为同一个文件（内容级对比）等，Python 2.3 或更高版本默认自带 `filecmp` 模块，无需额外安装，下面进行详细介绍。

### 2.2.1 模块常用方法说明

`filecmp` 提供了三个操作方法，分别为 `cmp`（单文件对比）、`cmpfiles`（多文件对比）、`dircmp`（目录对比），下面逐一进行介绍：

- 单文件对比，采用 `filecmp.cmp(f1, f2[, shallow])` 方法，比较文件名为 `f1` 和 `f2` 的文件，相同返回 `True`，不相同返回 `False`，`shallow` 默认为 `True`，意思是只根据 `os.stat()` 方法返回的文件基本信息进行对比，比如最后访问时间、修改时间、状态改变时间等，会

忽略文件内容的对比。当 `shallow` 为 `False` 时，则 `os.stat()` 与文件内容同时进行校验。

**示例：**比较单文件的差异。

```
>>> filecmp.cmp("/home/test/filecmp/f1", "/home/test/filecmp/f3")
True
>>> filecmp.cmp("/home/test/filecmp/f1", "/home/test/filecmp/f2")
False
```

❑ **多文件对比**，采用 `filecmp.cmpfiles(dir1, dir2, common[, shallow])` 方法，对比 `dir1` 与 `dir2` 目录给定的文件清单。该方法返回文件名的三个列表，分别为匹配、不匹配、错误。匹配为包含匹配的文件列表，不匹配反之，错误列表包括了目录不存在文件、不具备读权限或其他原因导致的不能比较的文件清单。

**示例：**`dir1` 与 `dir2` 目录中指定文件清单对比。

两目录下文件的 `md5` 信息如下，其中 `f1`、`f2` 文件匹配；`f3` 不匹配；`f4`、`f5` 对应目录中不存在，无法比较。

```
[root@SN2013-08-020 dir2]# md5sum *
d9dfc198c249bb4ac341198a752b9458 f1
aa9aa0cac0ffc655ce9232e720bf1b9f f2
33d2119b71f717ef4b981e9364530a39 f3
d9dfc198c249bb4ac341198a752b9458 f5
[root@SN2013-08-020 dir1]# md5sum *
d9dfc198c249bb4ac341198a752b9458 f1
aa9aa0cac0ffc655ce9232e720bf1b9f f2
d9dfc198c249bb4ac341198a752b9458 f3
410d6a485bcf5d2d223f2ada9b9c52 f4
```

使用 `cmpfiles` 对比的结果如下，符合我们的预期。

```
>>>filecmp.cmpfiles("/home/test/filecmp/dir1", "/home/test/filecmp/dir2", ['f1', 'f2',
'f3', 'f4', 'f5'])
(['f1', 'f2'], ['f3'], ['f4', 'f5'])
```

❑ **目录对比**，通过 `dircmp(a, b[, ignore[, hide]])` 类创建一个目录比较对象，其中 `a` 和 `b` 是参加比较的目录名。`ignore` 代表文件名忽略的列表，并默认为 `['RCS', 'CVS', 'tags']`；`hide` 代表隐藏的列表，默认为 `[os.curdir, os.pardir]`。`dircmp` 类可以获得目录比较的详细信息，如只有在 `a` 目录中包括的文件、`a` 与 `b` 都存在的子目录、匹配的文件等，同时支持递归。

`dircmp` 提供了三个输出报告的方法：

- ❑ `report()`，比较当前指定目录中的内容；
- ❑ `report_partial_closure()`，比较当前指定目录及第一级子目录中的内容；

❑ `report_full_closure()`，递归比较所有指定目录的内容。

为输出更加详细的比较结果，`dircmp` 类还提供了以下属性：

- ❑ `left`，左目录，如类定义中的 `a`；
- ❑ `right`，右目录，如类定义中的 `b`；
- ❑ `left_list`，左目录中的文件及目录列表；
- ❑ `right_list`，右目录中的文件及目录列表；
- ❑ `common`，两边目录共同存在的文件或目录；
- ❑ `left_only`，只在左目录中的文件或目录；
- ❑ `right_only`，只在右目录中的文件或目录；
- ❑ `common_dirs`，两边目录都存在的子目录；
- ❑ `common_files`，两边目录都存在的子文件；
- ❑ `common_funny`，两边目录都存在的子目录（不同目录类型或 `os.stat()` 记录的错误）；
- ❑ `same_files`，匹配相同的文件；
- ❑ `diff_files`，不匹配的文件；
- ❑ `funny_files`，两边目录中都存在，但无法比较的文件；
- ❑ `subdirs`，将 `common_dirs` 目录名映射到新的 `dircmp` 对象，格式为字典类型。

示例：对比 `dir1` 与 `dir2` 的目录差异。

通过调用 `dircmp()` 方法实现目录差异对比功能，同时输出目录对比对象所有属性信息。

【 `/home/test/filecmp/ simple1.py` 】

```
import filecmp
a="/home/test/filecmp/dir1"      # 定义左目录
b="/home/test/filecmp/dir2"      # 定义右目录
dirobj=filecmp.dircmp(a,b,['test.py'])  # 目录比较，忽略 test.py 文件
# 输出对比结果数据报表，详细说明请参考 filecmp 类方法及属性信息
dirobj.report()
dirobj.report_partial_closure()
dirobj.report_full_closure()
print "left_list:" + str(dirobj.left_list)
print "right_list:" + str(dirobj.right_list)
print "common:" + str(dirobj.common)
print "left_only:" + str(dirobj.left_only)
print "right_only:" + str(dirobj.right_only)
print "common_dirs:" + str(dirobj.common_dirs)
print "common_files:" + str(dirobj.common_files)
print "common_funny:" + str(dirobj.common_funny)
print "same_file:" + str(dirobj.same_files)
print "diff_files:" + str(dirobj.diff_files)
print "funny_files:" + str(dirobj.funny_files)
```

为方便理解，通过 `tree` 命令输出两个目录的树结构，如图 2-4 所示。

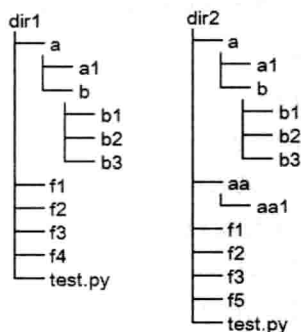


图 2-4 通过 `tree` 命令输出的两个目录

运行前面的代码并输出，结果如下：

```

# python simple1.py
-----report-----
diff /home/test/filecmp/dir1 /home/test/filecmp/dir2
Only in /home/test/filecmp/dir1 : ['f4']
Only in /home/test/filecmp/dir2 : ['aa', 'f5']
Identical files : ['f1', 'f2']
Differing files : ['f3']
Common subdirectories : ['a']
-----report_partial_closure-----
diff /home/test/filecmp/dir1 /home/test/filecmp/dir2
Only in /home/test/filecmp/dir1 : ['f4']
Only in /home/test/filecmp/dir2 : ['aa', 'f5']
Identical files : ['f1', 'f2']
Differing files : ['f3']
Common subdirectories : ['a']
diff /home/test/filecmp/dir1/a /home/test/filecmp/dir2/a
Identical files : ['a1']
Common subdirectories : ['b']
-----report_full_closure-----
diff /home/test/filecmp/dir1 /home/test/filecmp/dir2
Only in /home/test/filecmp/dir1 : ['f4']
Only in /home/test/filecmp/dir2 : ['aa', 'f5']
Identical files : ['f1', 'f2']
Differing files : ['f3']
Common subdirectories : ['a']
diff /home/test/filecmp/dir1/a /home/test/filecmp/dir2/a
Identical files : ['a1']
Common subdirectories : ['b']
diff /home/test/filecmp/dir1/a/b /home/test/filecmp/dir2/a/b
Identical files : ['b1', 'b2', 'b3']
left_list:['a', 'f1', 'f2', 'f3', 'f4']
  
```

```

right_list:['a', 'aa', 'f1', 'f2', 'f3', 'f5']
common:['a', 'f1', 'f2', 'f3']
left_only:['f4']
right_only:['aa', 'f5']
common_dirs:['a']
common_files:['f1', 'f2', 'f3']
common_funny:[]
same_file:['f1', 'f2']
diff_files:['f3']
funny_files:[]

```

## 2.2.2 实践：校验源与备份目录差异

有时候我们无法确认备份目录与源目录文件是否保持一致，包括源目录中的新文件或目录、更新文件或目录有无成功同步，定期进行校验，没有成功则希望有针对性地进行补备份。本示例使用了 filecmp 模块的 left\_only、diff\_files 方法递归获取源目录的更新项，再通过 shutil.copyfile、os.makedirs 方法对更新项进行复制，最终保持一致状态。详细源码如下：

【 /home/test/filecmp/simple2.py 】

```

#!/usr/bin/env python
import os, sys
import filecmp
import re
import shutil
holderlist=[]

def compareme(dir1, dir2):    # 递归获取更新项函数
    dircomp=filecmp.dircmp(dir1,dir2)
    only_in_one=dircomp.left_only    # 源目录新文件或目录
    diff_in_one=dircomp.diff_files    # 不匹配文件，源目录文件已发生变化
    dirpath=os.path.abspath(dir1)    # 定义源目录绝对路径
    # 将更新文件名或目录追加到 holderlist
    [holderlist.append(os.path.abspath(os.path.join(dir1,x))) for x in only_in_one]
    [holderlist.append(os.path.abspath(os.path.join(dir1,x))) for x in diff_in_one]
    if len(dircomp.common_dirs) > 0:    # 判断是否存在相同子目录，以便递归
        for item in dircomp.common_dirs:    # 递归子目录
            compareme(os.path.abspath(os.path.join(dir1,item)), \
                os.path.abspath(os.path.join(dir2,item)))
        return holderlist

def main():
    if len(sys.argv) > 2:    # 要求输入源目录与备份目录
        dir1=sys.argv[1]
        dir2=sys.argv[2]
    else:
        print "Usage: ", sys.argv[0], "datadir backupdir"
        sys.exit()

```

```

source_files=compareme(dir1,dir2)    # 对比源目录与备份目录
dir1=os.path.abspath(dir1)

if not dir2.endswith('/'): dir2=dir2+'/'    # 备份目录路径加“/”符
dir2=os.path.abspath(dir2)
destination_files=[]
createdir_bool=False

for item in source_files:    # 遍历返回的差异文件或目录清单
    destination_dir=re.sub(dir1, dir2, item)    # 将源目录差异路径清单对应替换成
                                                # 备份目录
    destination_files.append(destination_dir)
    if os.path.isdir(item):    # 如果差异路径为目录且不存在，则在备份目录中创建
        if not os.path.exists(destination_dir):
            os.makedirs(destination_dir)
            createdir_bool=True    # 再次调用 compareme 函数标记

if createdir_bool:    # 重新调用 compareme 函数，重新遍历新创建目录的内容
    destination_files=[]
    source_files=[]
    source_files=compareme(dir1,dir2)    # 调用 compareme 函数
    for item in source_files:    # 获取源目录差异路径清单，对应替换成备份目录
        destination_dir=re.sub(dir1, dir2, item)
        destination_files.append(destination_dir)

print "update item:"
print source_files    # 输出更新项列表清单
copy_pair=zip(source_files,destination_files)    # 将源目录与备份目录文件清单拆分成元组
for item in copy_pair:
    if os.path.isfile(item[0]):    # 判断是否为文件，是则进行复制操作
        shutil.copyfile(item[0], item[1])

if __name__ == '__main__':
    main()

```

更新源目录 dir1 中的 f4、code/f3 文件后，运行程序结果如下：

```

# python simple2.py /home/test/filecmp/dir1 /home/test/filecmp/dir2
update item:
['/home/test/filecmp/dir1/f4', '/home/test/filecmp/dir1/code/f3']
# python simple2.py /home/test/filecmp/dir1 /home/test/filecmp/dir2
update item:
[]    # 再次运行时已经没有更新项了

```



参考  
提示

□ 2.2.1 节模块方法说明参考 <http://docs.python.org/2/library/filecmp.html>。

□ 2.2.2 节示例参考 <http://linuxfreelancer.com/how-do-you-compare-two-folders-and-copy-the-difference-to-a-third-folder>。

## 2.3 发送电子邮件模块 smtplib

电子邮件是最流行的互联网应用之一。在系统管理领域，我们常常使用邮件来发送告警信息、业务质量报表等，方便运维人员第一时间了解业务的服务状态。本节通过 Python 的 smtplib 模块来实现邮件的发送功能，模拟一个 smtp 客户端，通过与 smtp 服务器交互来实现邮件发送的功能，这可以理解成 Foxmail 的发邮件功能，在第一次使用之前我们需要配置 smtp 主机地址、邮箱账号及密码等信息，Python 2.3 或更高版本默认自带 smtplib 模块，无需额外安装。下面详细进行介绍。

### 2.3.1 smtplib 模块的常用类与方法

SMTP 类定义：smtplib.SMTP([host[, port[, local\_hostname[, timeout]]]]), 作为 SMTP 的构造函数，功能是与 smtp 服务器建立连接，在连接成功后，就可以向服务器发送相关请求，比如登录、校验、发送、退出等。host 参数为远程 smtp 主机地址，比如 smtp.163.com；port 为连接端口，默认为 25；local\_hostname 的作用是在本地主机的 FQDN（完整的域名）发送 HELO/EHLO（标识用户身份）指令，timeout 为连接或尝试在多少秒超时。SMTP 类具有如下方法：

- ❑ SMTP.connect([host[, port]]) 方法，连接远程 smtp 主机方法，host 为远程主机地址，port 为远程主机 smtp 端口，默认 25，也可以直接使用 host:port 形式来表示，例如：SMTP.connect (“smtp.163.com”, “25”)。
- ❑ SMTP.login(user, password) 方法，远程 smtp 主机的校验方法，参数为用户名与密码，如 SMTP.login (“python\_2014@163.com”, “sdjkg358”)。
- ❑ SMTP.sendmail(from\_addr, to\_addrs, msg[, mail\_options, rcpt\_options]) 方法，实现邮件的发送功能，参数依次为是发件人、收件人、邮件内容，例如：SMTP.sendmail (“python\_2014@163.com”, “demo@domail.com”, body)，其中 body 内容定义如下：

```
"""From: python_2014@163.com
To: demo@domail.com
Subject: test mail
```

```
test mail body"""
```

- ❑ SMTP.starttls([keyfile[, certfile]]) 方法，启用 TLS（安全传输）模式，所有 SMTP 指令都将加密传输，例如使用 gmail 的 smtp 服务时需要启动此项才能正常发送邮件，如 SMTP.starttls()。
- ❑ SMTP.quit() 方法，断开 smtp 服务器的连接。

下面通过一个简单示例帮助大家理解，目的是使用 gmail 向 QQ 邮箱发送测试邮件，代

码如下:

```
#!/usr/bin/python
import smtplib
import string

HOST = "smtp.gmail.com"      # 定义 smtp 主机
SUBJECT = "Test email from Python"    # 定义邮件主题
TO = "testmail@qq.com"      # 定义邮件收件人
FROM = "mymail@gmail.com"    # 定义邮件发件人
text = "Python rules them all!"    # 邮件内容
BODY = string.join((        # 组装 sendmail 方法的邮件主体内容, 各段以 "\r\n" 进行分隔
    "From: %s" % FROM,
    "To: %s" % TO,
    "Subject: %s" % SUBJECT ,
    "",
    text
), "\r\n")

server = smtplib.SMTP()      # 创建一个 SMTP() 对象
server.connect(HOST, "25")    # 通过 connect 方法连接 smtp 主机
server.starttls()            # 启动安全传输模式
server.login("mymail@gmail.com", "mypassword")    # 邮箱账号登录校验
server.sendmail(FROM, [TO], BODY)    # 邮件发送
server.quit()                # 断开 smtp 连接
```

我们将收到一封这样的邮件, 如图 2-5 所示。

---

#### Test email from Python ☆

发件人: <mymail@gmail.com> 

时 间: 2014年3月27日(星期四) 上午7:42 (UTC-07:00 休斯顿、底特律时间)

收件人: <testmail@qq.com>

---

Python rules them all!

图 2-5 收到的邮件

## 2.3.2 定制个性化的邮件格式方法

通过邮件传输简单的文本已经无法满足我们的需求, 比如我们时常会定制业务质量报表, 在邮件主体中会包含 HTML、图像、声音以及附件格式等, MIME (Multipurpose Internet Mail Extensions, 多用途互联网邮件扩展) 作为一种新的扩展邮件格式很好地补充了这一点, 更多 MIME 知识见 <http://zh.wikipedia.org/wiki/MIME>。下面介绍几个 Python 中常用的 MIME 实现类:

- `email.mime.multipart.MIMEMultipart([_subtype[, boundary[, _subparts[, _params]]])`, 作用是生成包含多个部分的邮件体的 MIME 对象, 参数 `_subtype` 指定要添加到 "Content-type: multipart/subtype" 报头的可选的三种子类型, 分别为 `mixed`、`related`、

alternative, 默认值为 mixed。定义 mixed 实现构建一个带附件的邮件体; 定义 related 实现构建内嵌资源的邮件体; 定义 alternative 则实现构建纯文本与超文本共存的邮件体。

- ❑ email.mime.audio.MIMEAudio(\_audiodata[, \_subtype[, \_encoder[, \*\*\_params]]]), 创建包含音频数据的邮件体, \_audiodata 包含原始二进制音频数据的字节字符串。
- ❑ email.mime.image.MIMEImage(\_imagedata[, \_subtype[, \_encoder[, \*\*\_params]]]), 创建包含图片数据的邮件体, \_imagedata 是包含原始图片数据的字节字符串。
- ❑ email.mime.text.MIMEText(\_text[, \_subtype[, \_charset]]), 创建包含文本数据的邮件体, \_text 是包含消息负载的字符串, \_subtype 指定文本类型, 支持 plain (默认值) 或 html 类型的字符串。

### 2.3.3 定制常用邮件格式示例详解

前面两小节介绍了 Python 的 smtplib 及 email 模块的常用方法, 那么两者在邮件定制到发送过程中是如何分工的? 我们可以将 email.mime 理解成 smtplib 模块邮件内容主体的扩展, 从原先默认只支持纯文本格式扩展到 HTML, 同时支持附件、音频、图像等格式, smtplib 只负责邮件的投递即可。下面介绍在日常运营工作中邮件应用的几个示例。

**示例 1: 实现 HTML 格式的数据报表邮件。**

纯文本的邮件内容已经不能满足我们多样化的需求, 本示例通过引入 email.mime 的 MIMEText 类来实现支持 HTML 格式的邮件, 支持所有 HTML 元素, 包含表格、图片、动画、CSS 样式、表单等。本示例使用 HTML 的表格定制美观的业务流量报表, 实现代码如下:

【 /home/test/smtplib/simple2.py 】

```
#coding: utf-8
import smtplib
from email.mime.text import MIMEText      # 导入 MIMEText 类

HOST = "smtp.gmail.com"      # 定义 smtp 主机
SUBJECT = u" 官网流量数据报表 "      # 定义邮件主题
TO = "testmail@qq.com"      # 定义邮件收件人
FROM = "mymail@gmail.com"      # 定义邮件发件人
msg = MIMEText("")      # 创建一个 MIMEText 对象, 分别指定 HTML 内容、类型 (文本或 html)、字符编码

<table width="800" border="0" cellspacing="0" cellpadding="4">
  <tr>
    <td bgcolor="#CECFAD" height="20" style="font-size:14px">* 官网数据 <a
href="monitor.domain.com">更多 >></a></td>
  </tr>
  <tr>
    <td bgcolor="#EFEFDE" height="100" style="font-size:13px">
```



【 /home/test/smtplib/simple3.py 】

```
#coding: utf-8
import smtplib
from email.mime.multipart import MIMEMultipart    # 导入 MIMEMultipart 类
from email.mime.text import MIMEText            # 导入 MIMEText 类
from email.mime.image import MIMEImage          # 导入 MIMEImage 类

HOST = "smtp.gmail.com"    # 定义 smtp 主机
SUBJECT = u" 业务性能数据报表 "    # 定义邮件主题
TO = "testmail@qq.com"    # 定义邮件收件人
FROM = "mymail@gmail.com"    # 定义邮件发件人

def addimg(src, imgid):    # 添加图片函数，参数 1：图片路径，参数 2：图片 id
    fp = open(src, 'rb')    # 打开文件
    msgImage = MIMEImage(fp.read())    # 创建 MIMEImage 对象，读取图片内容并作为参数
    fp.close()    # 关闭文件
    msgImage.add_header('Content-ID', imgid)    # 指定图片文件的 Content-ID, <img>
                                                # 标签 src 用到

    return msgImage    # 返回 msgImage 对象

msg = MIMEMultipart('related')    # 创建 MIMEMultipart 对象，采用 related 定义内嵌资源
                                    # 的邮件体
msgtext = MIMEText("""    # 创建一个 MIMEText 对象，HTML 元素包括表格 <table> 及图片 <img>
<table width="600" border="0" cellpadding="4">
  <tr bgcolor="#CECFAD" height="20" style="font-size:14px">
    <td colspan=2>* 官网性能数据  <a href="monitor.domain.com"> 更多 >></a></td>
  </tr>
  <tr bgcolor="#EFEFDE" height="100" style="font-size:13px">
    <td>
      </td><td>
      </td>
    </tr>
  <tr bgcolor="#EFEFDE" height="100" style="font-size:13px">
    <td>
      </td><td>
      </td>
    </tr>
</table>""", "html", "utf-8")    # <img> 标签的 src 属性是通过 Content-ID 来引用的

msg.attach(msgtext)    # MIMEMultipart 对象附加 MIMEText 的内容
msg.attach(addimg("img/bytes_io.png", "io"))    # 使用 MIMEMultipart 对象附加 MIMEImage
                                                # 的内容
msg.attach(addimg("img/myisam_key_hit.png", "key_hit"))
msg.attach(addimg("img/os_mem.png", "men"))
msg.attach(addimg("img/os_swap.png", "swap"))
msg['Subject'] = SUBJECT    # 邮件主题
msg['From'] = FROM    # 邮件发件人，邮件头部可见
msg['To'] = TO    # 邮件收件人，邮件头部可见
try:
```

```

server = smtplib.SMTP()      # 创建一个 SMTP() 对象
server.connect(HOST, "25")    # 通过 connect 方法连接 smtp 主机
server.starttls()            # 启动安全传输模式
server.login("mymail@gmail.com", "mypassword")    # 邮箱账号登录校验
server.sendmail(FROM, TO, msg.as_string())        # 邮件发送
server.quit()                # 断开 smtp 连接
print "邮件发送成功! "
except Exception, e:
    print "失败: "+str(e)

```

代码运行结果如图 2-7 所示, 我们将业务服务器性能数据定期推送给管理员, 以方便管理员了解业务的服务情况。

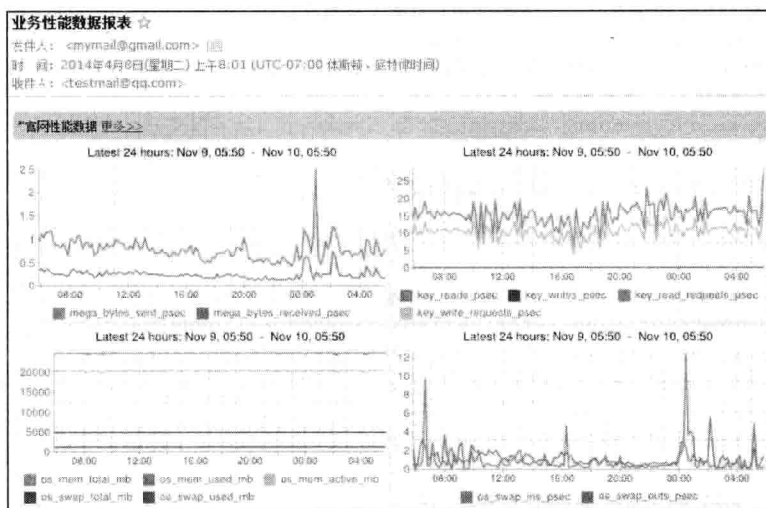


图 2-7 示例 2 运行结果

示例 3: 实现带附件格式的业务服务质量周报邮件。

本示例通过 MIMEText 与 MIMEImage 类的组合, 实现图文邮件格式。另通过 MIMEText 类再定义 Content-Disposition 属性来实现带附件的邮件。我们可以利用这些丰富的特性来定制周报邮件, 如业务服务质量周报。实现代码如下:

【/home/test/smtplib/simple4.py】

```

#coding: utf-8
import smtplib
from email.mime.multipart import MIMEMultipart    # 导入 MIMEMultipart 类
from email.mime.text import MIMEText            # 导入 MIMEText 类
from email.mime.image import MIMEImage          # 导入 MIMEImage 类
HOST = "smtp.gmail.com"                        # 定义 smtp 主机

```

```

SUBJECT = u" 官网业务服务质量周报 "      # 定义邮件主题
TO = "testmail@qq.com"      # 定义邮件接收人
FROM = "mymail@gmail.com"    # 定义邮件发件人

def adding(src, imgid):      # 添加图片函数, 参数 1: 图片路径, 参数 2: 图片 id
    fp = open(src, 'rb')     # 打开文件
    msgImage = MIMEImage(fp.read())    # 创建 MIMEImage 对象, 读取图片内容作为参数
    fp.close()               # 关闭文件
    msgImage.add_header('Content-ID', imgid)    # 指定图片文件的 Content-ID, <img>
                                                # 标签 src 用到

    return msgImage          # 返回 msgImage 对象

msg = MIMEMultipart('related')    # 创建 MIMEMultipart 对象, 采用 related 定义内嵌资源
                                    # 的邮件体
# 创建一个 MIMEText 对象, HTML 元素包括文字与图片 <img>
msgtext = MIMEText("<font color=red> 官网业务周平均延时图表 :<br><img src=\"cid:weekly\" \"
border=\"1\"><br> 详细内容见附件。</font>", "html", "utf-8")
msg.attach(msgtext)              # MIMEMultipart 对象附加 MIMEText 的内容
msg.attach(adding("img/weekly.png", "weekly"))    # 使用 MIMEMultipart 对象附加
                                                # MIMEImage 的内容

# 创建一个 MIMEText 对象, 附加 week_report.xlsx 文档
attach = MIMEText(open("doc/week_report.xlsx", "rb").read(), "base64", "utf-8")
attach["Content-Type"] = "application/octet-stream"    # 指定文件格式类型
# 指定 Content-Disposition 值为 attachment 则出现下载保存对话框, 保存的默认文件名使用
# filename 指定
# 由于 qqmail 使用 gb18030 页面编码, 为保证中文文件名不出现乱码, 对文件名进行编码转换
attach["Content-Disposition"] = "attachment; filename=\" 业务服务质量周报 (12 周).xlsx\""
decode("utf-8").encode("gb18030")

msg.attach(attach)              # MIMEMultipart 对象附加 MIMEText 附件内容
msg['Subject'] = SUBJECT        # 邮件主题
msg['From'] = FROM              # 邮件发件人, 邮件头部可见
msg['To'] = TO                  # 邮件收件人, 邮件头部可见
try:
    server = smtplib.SMTP()     # 创建一个 SMTP() 对象
    server.connect(HOST, "25")  # 通过 connect 方法连接 smtp 主机
    server.starttls()           # 启动安全传输模式
    server.login("mymail@gmail.com", "mypassword")    # 邮箱账号登录校验
    server.sendmail(FROM, TO, msg.as_string())        # 邮件发送
    server.quit()               # 断开 smtp 连接
    print " 邮件发送成功! "
except Exception, e:
    print " 失败: " + str(e)

```

代码运行结果如图 2-8 所示, 实现了发送业务服务质量周报的邮件功能。

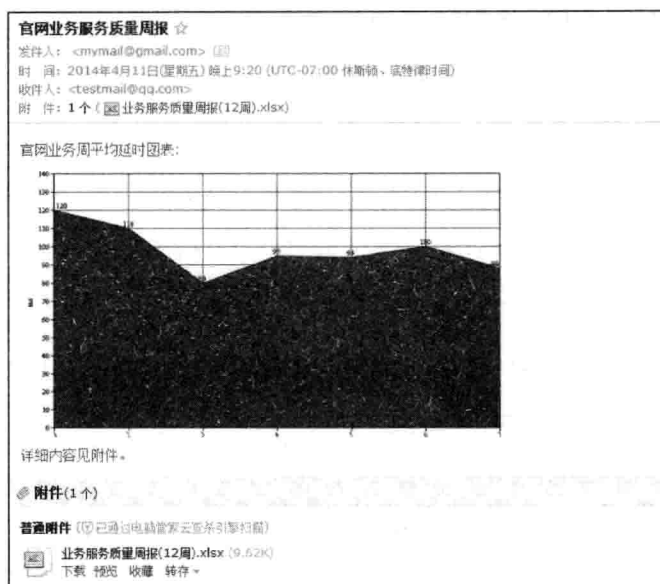


图 2-8 示例 3 的运行结果



参考提示

- 2.3.1 节 smtpplib 模块的常用类与方法内容参考 <https://docs.python.org/2.7/library/smtplib.html>。
- 2.3.2 节 email.mime 常用类定义内容参考 <https://docs.python.org/2.7/library/email.mime.html>。

## 2.4 探测 Web 服务质量方法

pycurl (<http://pycurl.sourceforge.net>) 是一个用 C 语言写的 libcurl Python 实现, 功能非常强大, 支持的操作协议有 FTP、HTTP、HTTPS、TELNET 等, 可以理解成 Linux 下 curl 命令功能的 Python 封装, 简单易用。本节通过调用 pycurl 提供的方法, 实现探测 Web 服务质量的情况, 比如响应的 HTTP 状态码、请求延时、HTTP 头信息、下载速度等, 利用这些信息可以定位服务响应慢的具体环节, 下面详细进行说明。

pycurl 模块的安装方法如下:

```
easy_install pycurl    #easy_install 安装方法
pip install pycurl     #pip 安装方法
```

```

# 源码安装方法
# 要求 curl-config 包支持, 需要源码方式重新安装 curl
# wget http://curl.haxx.se/download/curl-7.36.0.tar.gz
# tar -zxvf curl-7.36.0.tar.gz
# cd curl-7.36.0
# ./configure
# make && make install
# export LD_LIBRARY_PATH=/usr/local/lib
#
# wget https://pypi.python.org/packages/source/p/pycurl/pycurl-7.19.3.1.tar.gz
--no-check-certificate
# tar -zxvf pycurl-7.19.3.1.tar.gz
# cd pycurl-7.19.3.1
# python setup.py install --curl-config=/usr/local/bin/curl-config

```

校验安装结果如下:

```

>>> import pycurl
>>> pycurl.version
'PycURL/7.19.3.1 libcurl/7.36.0 OpenSSL/1.0.1e zlib/1.2.3'

```

### 2.4.1 模块常用方法说明

`pycurl.Curl()` 类实现创建一个 `libcurl` 包的 `Curl` 句柄对象, 无参数。更多关于 `libcurl` 包的介绍见 <http://curl.haxx.se/libcurl/c/libcurl-tutorial.html>。下面介绍 `Curl` 对象几个常用的方法。

- ❑ `close()` 方法, 对应 `libcurl` 包中的 `curl_easy_cleanup` 方法, 无参数, 实现关闭、回收 `Curl` 对象。
- ❑ `perform()` 方法, 对应 `libcurl` 包中的 `curl_easy_perform` 方法, 无参数, 实现 `Curl` 对象请求的提交。
- ❑ `setopt(option, value)` 方法, 对应 `libcurl` 包中的 `curl_easy_setopt` 方法, 参数 `option` 是通过 `libcurl` 的常量来指定的, 参数 `value` 的值会依赖 `option`, 可以是一个字符串、整型、长整型、文件对象、列表或函数等。下面列举常用的常量列表:

```

c = pycurl.Curl()      # 创建一个 curl 对象
c.setopt(pycurl.CONNECTTIMEOUT, 5)  # 连接的等待时间, 设置为 0 则不等待
c.setopt(pycurl.TIMEOUT, 5)        # 请求超时时间
c.setopt(pycurl.NOPROGRESS, 0)     # 是否屏蔽下载进度条, 非 0 则屏蔽
c.setopt(pycurl.MAXREDIRS, 5)      # 指定 HTTP 重定向的最大数
c.setopt(pycurl.FORBID_REUSE, 1)   # 完成交互后强制断开连接, 不重用
c.setopt(pycurl.FRESH_CONNECT, 1)  # 强制获取新的连接, 即替代缓存中的连接
c.setopt(pycurl.DNS_CACHE_TIMEOUT, 60) # 设置保存 DNS 信息的时间, 默认为 120 秒
c.setopt(pycurl.URL, "http://www.baidu.com") # 指定请求的 URL
c.setopt(pycurl.USERAGENT, "Mozilla/5.2 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50324)") # 配置请求 HTTP 头的 User-Agent
c.setopt(pycurl.HEADERFUNCTION, getheader) # 将返回的 HTTP HEADER 定向到回调函数 getheader

```

```
c.setopt(pycurl.WRITEFUNCTION, getbody)      # 将返回的内容定向到回调函数 getbody
c.setopt(pycurl.WRITEHEADER, fileobj)        # 将返回的 HTTP HEADER 定向到 fileobj 文件对象
c.setopt(pycurl.WRITEDATA, fileobj)          # 将返回的 HTML 内容定向到 fileobj 文件对象
```

□ `getinfo(option)` 方法，对应 `libcurl` 包中的 `curl_easy_getinfo` 方法，参数 `option` 是通过 `libcurl` 的常量来指定的。下面列举常用的常量列表：

```
c = pycurl.Curl()      # 创建一个 curl 对象
c.getinfo(pycurl.HTTP_CODE)      # 返回的 HTTP 状态码
c.getinfo(pycurl.TOTAL_TIME)     # 传输结束所消耗的总时间
c.getinfo(pycurl.NAMELOOKUP_TIME) # DNS 解析所消耗的时间
c.getinfo(pycurl.CONNECT_TIME)   # 建立连接所消耗的时间
c.getinfo(pycurl.PRETRANSFER_TIME) # 从建立连接到准备传输所消耗的时间
c.getinfo(pycurl.STARTTRANSFER_TIME) # 从建立连接到传输开始消耗的时间
c.getinfo(pycurl.REDIRECT_TIME)  # 重定向所消耗的时间
c.getinfo(pycurl.SIZE_UPLOAD)    # 上传数据包大小
c.getinfo(pycurl.SIZE_DOWNLOAD)  # 下载数据包大小
c.getinfo(pycurl.SPEED_DOWNLOAD) # 平均下载速度
c.getinfo(pycurl.SPEED_UPLOAD)   # 平均上传速度
c.getinfo(pycurl.HEADER_SIZE)    # HTTP 头部大小
```

我们利用 `libcurl` 包提供的这些常量值来达到探测 Web 服务质量的目的。

## 2.4.2 实践：实现探测 Web 服务质量

HTTP 服务是最流行的互联网应用之一，服务质量的好坏关系到用户体验以及网站的运营服务水平，最常用的有两个标准，一为服务的可用性，比如是否处于正常提供服务状态，而不是出现 404 页面未找到或 500 页面错误等；二为服务的响应速度，比如静态类文件下载时间都控制在毫秒级，动态 CGI 为秒级。本示例使用 `pycurl` 的 `setopt` 与 `getinfo` 方法实现 HTTP 服务质量的探测，获取监控 URL 返回的 HTTP 状态码，HTTP 状态码采用 `pycurl.HTTP_CODE` 常量得到，以及从 HTTP 请求到完成下载期间各环节的响应时间，通过 `pycurl.NAMELOOKUP_TIME`、`pycurl.CONNECT_TIME`、`pycurl.PRETRANSFER_TIME`、`pycurl.R` 等常量来实现。另外通过 `pycurl.WRITEHEADER`、`pycurl.WRITEDATA` 常量得到目标 URL 的 HTTP 响应头部及页面内容。实现源码如下：

【 /home/test/pycurl/simple1.py 】

```
# -*- coding: utf-8 -*-
import os,sys
import time
import sys
import pycurl

URL="http://www.google.com.hk"      # 探测的目标 URL
c = pycurl.Curl()      # 创建一个 Curl 对象
```

```

c.setopt(pycurl.URL, URL)      # 定义请求的 URL 常量
c.setopt(pycurl.CONNECTTIMEOUT, 5)  # 定义请求连接的等待时间
c.setopt(pycurl.TIMEOUT, 5)      # 定义请求超时时间
c.setopt(pycurl.NOPROGRESS, 1)    # 屏蔽下载进度条
c.setopt(pycurl.FORBID_REUSE, 1)  # 完成交互后强制断开连接, 不重用
c.setopt(pycurl.MAXREDIRS, 1)    # 指定 HTTP 重定向的最大数为 1
c.setopt(pycurl.DNS_CACHE_TIMEOUT, 30)  # 设置保存 DNS 信息的时间为 30 秒
# 创建一个文件对象, 以 "wb" 方式打开, 用来存储返回的 http 头部及页面内容
indexfile = open(os.path.dirname(os.path.realpath(__file__))+"/content.txt",
"wb")
c.setopt(pycurl.WRITEHEADER, indexfile)  # 将返回的 HTTP HEADER 定向到 indexfile 文件
对象
c.setopt(pycurl.WRITEDATA, indexfile)    # 将返回的 HTML 内容定向到 indexfile 文件对象
try:
    c.perform()      # 提交请求
except Exception,e:
    print "conneccion error:"+str(e)
    indexfile.close()
    c.close()
    sys.exit()

NAMELOOKUP_TIME = c.getinfo(c.NAMELOOKUP_TIME)  # 获取 DNS 解析时间
CONNECT_TIME = c.getinfo(c.CONNECT_TIME)        # 获取建立连接时间
PRETRANSFER_TIME = c.getinfo(c.PRETRANSFER_TIME)  # 获取从建立连接到准备传输所消
                                                    # 耗的时间
STARTTRANSFER_TIME = c.getinfo(c.STARTTRANSFER_TIME)  # 获取从建立连接到传输开始消
                                                    # 耗的时间

TOTAL_TIME = c.getinfo(c.TOTAL_TIME)            # 获取传输的总时间
HTTP_CODE = c.getinfo(c.HTTP_CODE)              # 获取 HTTP 状态码
SIZE_DOWNLOAD = c.getinfo(c.SIZE_DOWNLOAD)      # 获取下载数据包大小
HEADER_SIZE = c.getinfo(c.HEADER_SIZE)          # 获取 HTTP 头部大小
SPEED_DOWNLOAD=c.getinfo(c.SPEED_DOWNLOAD)      # 获取平均下载速度
# 打印输出相关数据
print "HTTP 状态码: %s" %(HTTP_CODE)
print "DNS 解析时间: %.2f ms"%(NAMELOOKUP_TIME*1000)
print "建立连接时间: %.2f ms" %(CONNECT_TIME*1000)
print "准备传输时间: %.2f ms" %(PRETRANSFER_TIME*1000)
print "传输开始时间: %.2f ms" %(STARTTRANSFER_TIME*1000)
print "传输结束总时间: %.2f ms" %(TOTAL_TIME*1000)
print "下载数据包大小: %d bytes/s" %(SIZE_DOWNLOAD)
print "HTTP 头部大小: %d byte" %(HEADER_SIZE)
print "平均下载速度: %d bytes/s" %(SPEED_DOWNLOAD)
# 关闭文件及 Curl 对象
indexfile.close()
c.close()

```

代码的执行结果如图 2-9 所示。

```
[root@SN2013-08-020 pycurl]# python simple1.py
HTTP状态码: 200
DNS解析时间: 113.18 ms
建立连接时间: 300.70 ms
准备传输时间: 301.06 ms
传输开始时间: 507.36 ms
传输结束总时间: 507.52 ms
下载数据包大小: 12006 bytes/s
HTTP头部大小: 798 byte
平均下载速度: 23656 bytes/s
```

图 2-9 探测到的 Web 服务质量

查看获取的 HTTP 文件头部及页面内容文件 content.txt, 如图 2-10 所示。

```
HTTP/1.1 200 OK^M
Date: Wed, 23 Apr 2014 15:19:04 GMT^M
Expires: -1^M
Cache-Control: private, max-age=0^M
Content-Type: text/html; charset=Big5^M
Set-Cookie: PREF-ID=e2c1021e3b4d36e8:FF-0:NW-1:TM-1398266344:LM-1398266344:S-X1ev4S^M
Set-Cookie: NID=67-P9W3T5im7sfXTfZVXP9m0SQq9SB3MlQqtA6SnLxh_4bbdN6lY3Q2vK0ciXtmhaG7^M
5:19:04 GMT; path=/; domain=google.com.hk; HttpOnly^M
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts/bin/ar^M
Server: gws^M
X-XSS-Protection: 1; mode=block^M
X-Frame-Options: SAMEORIGIN^M
Alternate-Protocol: 80:quic^M
Transfer-Encoding: chunked^M

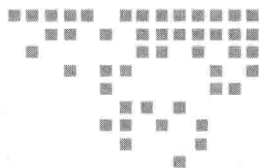
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="zh-HK">
<script>(function(){
window.google={kEI:"6NLXU-TQEMy6kAXeu4CoCA",getEI:function(a){for(var b;a&&(!a.getA
{return"https://"+window.location.protocol},kEXPI:"17259,4000116,4007661,4007830,400
,4012373,4012504,4013374,4013414,4013591,4013723,4013758,4013787,4013823,4013967,40
9,4015155,4015234,4015260,4015342,4015519,4015550,4015635,4015638,4015639,4015772,40
25,4016466,4016479,4016487,4016623,4016703,4016730,4016767,4016800,4016824,4016851,
177,4017201,4017205,4017261,4017336,8300015,8300017,8500165,8500223,8500240,8500252
```

图 2-10 content.txt 截图



参考  
提示

□ 2.4.1 节 pycurl 模块的常用类与方法说明参考官网 <http://pycurl.sourceforge.net/doc/index.html>。



## 定制业务质量报表详解

在日常运维工作当中，会涉及大量不同来源的数据，比如每天的服务器性能数据、平台监控数据、自定义业务上报数据等，需要根据不同时段，周期性地输出数据报表，以方便管理员更加清晰、及时地了解业务的运营情况。在业务监控过程中，也需要更加直观地展示报表，以便快速定位问题。本章介绍 Excel 操作模块、rrdtool 数据报表、scapy 包处理等，相关知识点运用到运营平台中将起到增色添彩的作用。

### 3.1 数据报表之 Excel 操作模块

Excel 是当今最流行的电子表格处理软件，支持丰富的计算函数及图表，在系统运营方面广泛用于运营数据报表，比如业务质量、资源利用、安全扫描等报表，同时也是应用系统常见的文件导出格式，以便数据使用人员做进一步加工处理。本节主要讲述利用 Python 操作 Excel 的模块 XlsxWriter (<https://xlsxwriter.readthedocs.org>)，可以操作多个工作表的文字、数字、公式、图表等。XlsxWriter 模块具有以下功能：

- ❑ 100% 兼容的 Excel XLSX 文件，支持 Excel 2003、Excel 2007 等版本；
- ❑ 支持所有 Excel 单元格数据格式；
- ❑ 单元格合并、批注、自动筛选、丰富多格式字符串等；
- ❑ 支持工作表 PNG、JPEG 图像，自定义图表；
- ❑ 内存优化模式支持写入大文件。

XlsxWriter 模块的安装方法如下：

```
# pip install XlsxWriter      #pip 安装方法
# easy_install XlsxWriter    #easy_install 安装方法

# 源码安装方法
# curl -O -L http://github.com/jmcnamara/XlsxWriter/archive/master.tar.gz
# tar zxvf master.tar.gz
# cd XlsxWriter-master/
# sudo python setup.py install
```

下面通过一个简单的功能演示示例，实现插入文字（中英字符）、数字（求和计算）、图片、单元格格式等，代码如下：

【 /home/test/XlsxWriter/simple1.py 】

```
#coding: utf-8
import xlsxwriter

workbook = xlsxwriter.Workbook('demo1.xlsx')    # 创建一个 Excel 文件
worksheet = workbook.add_worksheet()           # 创建一个工作表对象

worksheet.set_column('A:A', 20)                # 设定第一列 (A) 宽度为 20 像素
bold = workbook.add_format({'bold': True})      # 定义一个加粗的格式对象

worksheet.write('A1', 'Hello')                 # A1 单元格写入 'Hello'
worksheet.write('A2', 'World', bold)           # A2 单元格写入 'World' 并引用加粗格式对象 bold
worksheet.write('B2', u'中文测试', bold)       # B2 单元格写入中文并引用加粗格式对象 bold

worksheet.write(2, 0, 32)                      # 用行列表示法写入数字 '32' 与 '35.5'
worksheet.write(3, 0, 35.5)                    # 行列表示法的单元格下标以 0 作为起始值, '3,0' 等价于 'A3'
worksheet.write(4, 0, '=SUM(A3:A4)')           # 求 A3:A4 的和, 并将结果写入 '4, 0', 即 'A5'

worksheet.insert_image('B5', 'img/python-logo.png') # 在 B5 单元格插入图片
workbook.close()                                # 关闭 Excel 文件
```

程序生成的 demo1.xlsx 文档截图如图 3-1 所示。



图 3-1 demo1.xlsx 文档截图

### 3.1.1 模块常用方法说明

#### 1. Workbook 类

Workbook 类定义: `Workbook(filename[, options])`, 该类实现创建一个 XlsxWriter 的 Workbook 对象。Workbook 类代表整个电子表格文件, 并且存储在磁盘上。参数 filename (String 类型) 为创建的 Excel 文件存储路径; 参数 options (Dict 类型) 为可选的 Workbook 参数, 一般作为初始化工作表内容格式, 例如值为 `{'strings_to_numbers': True}` 表示使用 `worksheet.write()` 方法时激活字符串转换数字。

□ `add_worksheet([sheetname])` 方法, 作用是添加一个新的工作表, 参数 sheetname (String 类型) 为可选的工作表名称, 默认为 Sheet1。例如, 下面的代码对应的效果图如图 3-2 所示。

```
worksheet1 = workbook.add_worksheet()           # Sheet1
worksheet2 = workbook.add_worksheet('Foglio2')  # Foglio2
worksheet3 = workbook.add_worksheet('Data')     # Data
worksheet4 = workbook.add_worksheet()           # Sheet4
```

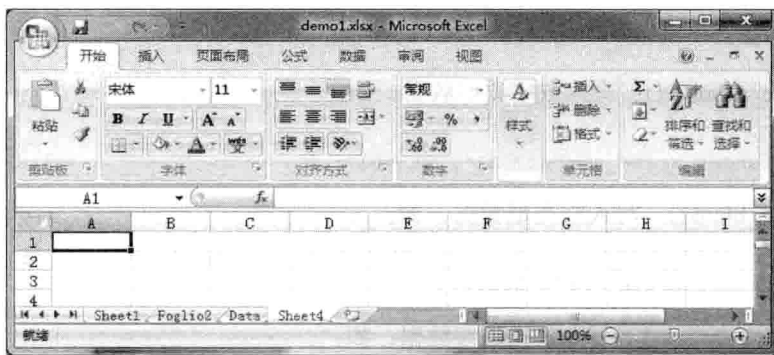


图 3-2 添加新工作表

□ `add_format([properties])` 方法, 作用是在工作表中创建一个新的格式对象来格式化单元格。参数 properties (dict 类型) 为指定一个格式属性的字典, 例如设置一个加粗的格式对象, `workbook.add_format({'bold': True})`。通过 Format methods (格式化方法) 也可以实现格式的设置, 等价的设置加粗格式代码如下:

```
bold = workbook.add_format()
bold.set_bold()
```

更多格式化方法见 [http://xlsxwriter.readthedocs.org/working\\_with\\_formats.html](http://xlsxwriter.readthedocs.org/working_with_formats.html)。

- ❑ `add_chart(options)` 方法，作用是在工作表中创建一个图表对象，内部是通过 `insert_chart()` 方法来实现，参数 `options` (dict 类型) 为图表指定一个字典属性，例如设置一个线条类型的图表对象，代码为 `chart = workbook.add_chart({'type': 'line'})`。
- ❑ `close()` 方法，作用是关闭工作表文件，如 `workbook.close()`。

## 2. Worksheet 类

`Worksheet` 类代表了一个 Excel 工作表，是 `XlsxWriter` 模块操作 Excel 内容最核心的一个类，例如将数据写入单元格或工作表格式布局等。`Worksheet` 对象不能直接实例化，取而代之的是通过 `Workbook` 对象调用 `add_worksheet()` 方法来创建。`Worksheet` 类提供了非常丰富的操作 Excel 内容的方法，其中几个常用的方法如下：

- ❑ `write(row, col, *args)` 方法，作用是写普通数据到工作表的单元格，参数 `row` 为行坐标，`col` 为列坐标，坐标索引起始值为 0；`*args` 无名字参数为数据内容，可以为数字、公式、字符串或格式对象。为了简化不同数据类型的写入过程，`write` 方法已经作为其他更加具体数据类型方法的别名，包括：

- `write_string()` 写入字符串类型数据，如：

```
worksheet.write_string(0, 0, 'Your text here');
```

- `write_number()` 写入数字类型数据，如：

```
worksheet.write_number('A2', 2.3451);
```

- `write_blank()` 写入空类型数据，如：

```
worksheet.write('A2', None);
```

- `write_formula()` 写入公式类型数据，如：

```
worksheet.write_formula(2, 0, '=SUM(B1:B5)');
```

- `write_datetime()` 写入日期类型数据，如：

```
worksheet.write_datetime(7, 0, datetime.datetime.strptime('2013-01-23', '%Y-%m-%d'), workbook.add_format({'num_format': 'yyyy-mm-dd'}));
```

- `write_boolean()` 写入逻辑类型数据，如：

```
worksheet.write_boolean(0, 0, True);
```

- `write_url()` 写入超链接类型数据，如：

```
worksheet.write_url('A1', 'ftp://www.python.org/')
```

下列通过具体的示例来观察别名 `write` 方法与数据类型方法的对应关系，代码如下：

```
worksheet.write(0, 0, 'Hello')           # write_string()
worksheet.write(1, 0, 'World')          # write_string()
worksheet.write(2, 0, 2)                 # write_number()
worksheet.write(3, 0, 3.00001)           # write_number()
worksheet.write(4, 0, '=SIN(PI()/4)')    # write_formula()
worksheet.write(5, 0, '')                # write_blank()
worksheet.write(6, 0, None)              # write_blank()
```

上述示例将创建一个如图 3-3 所示的工作表。

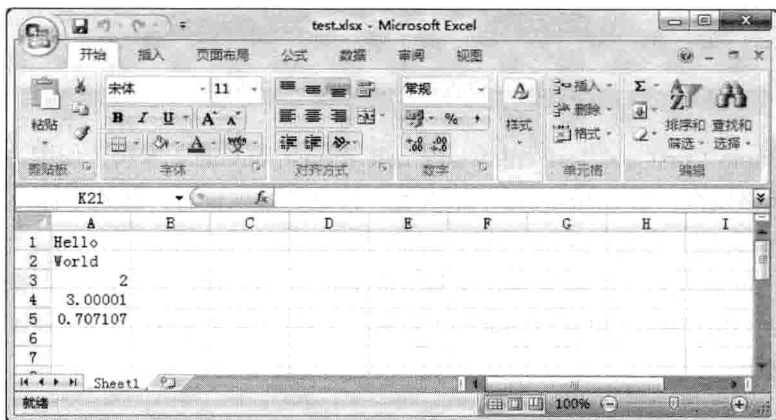


图 3-3 创建单元格并写入数据的工作表

□ `set_row (row, height, cell_format, options)` 方法，作用是设置行单元格的属性。参数 `row` (`int` 类型) 指定行位置，起始下标为 0；参数 `height` (`float` 类型) 设置行高，单位像素；参数 `cell_format` (`format` 类型) 指定格式对象；参数 `options` (`dict` 类型) 设置行 `hidden` (隐藏)、`level` (组合分级)、`collapsed` (折叠)。操作示例如下：

```
worksheet.write('A1', 'Hello')           # 在 A1 单元格写入 'Hello' 字符串
cell_format = workbook.add_format({'bold': True}) # 定义一个加粗的格式对象
worksheet.set_row(0, 40, cell_format)     # 设置第 1 行单元格高度为 40 像素，且引用加粗
                                           # 格式对象
worksheet.set_row(1, None, None, {'hidden': True}) # 隐藏第 2 行单元格
```

上述示例将创建一个如图 3-4 所示的工作表。

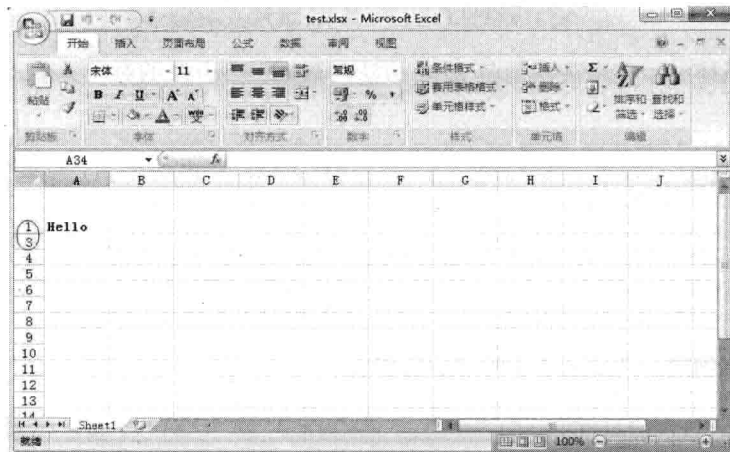


图 3-4 设置行单元格属性后的效果

❑ `set_column (first_col, last_col, width, cell_format, options)` 方法，作用为设置一列或多列单元格属性。参数 `first_col` (int 类型) 指定开始列位置，起始下标为 0；参数 `last_col` (int 类型) 指定结束列位置，起始下标为 0，可以设置成与 `first_col` 一样；参数 `width` (float 类型) 设置列宽；参数 `cell_format` (Format 类型) 指定格式对象；参数 `options` (dict 类型) 设置行 `hidden` (隐藏)、`level` (组合分级)、`collapsed` (折叠)。操作示例如下：

```
worksheet.write('A1', 'Hello')      # 在 A1 单元格写入 'Hello' 字符串
worksheet.write('B1', 'World')     # 在 B1 单元格写入 'World' 字符串
cell_format = workbook.add_format({'bold': True})  # 定义一个加粗的格式对象
                                                    # 设置 0 到 1 即 (A 到 B) 列单元格宽度为 10 像素，
                                                    # 且引用加粗格式对象
worksheet.set_column(0,1, 10,cell_format)
worksheet.set_column('C:D', 20)    # 设置 C 到 D 列单元格宽度为 20 像素
worksheet.set_column('E:G', None, None, {'hidden': 1})  # 隐藏 E 到 G 列单元格
```

上述示例将创建一个如图 3-5 所示的工作表。

❑ `insert_image(row, col, image[, options])` 方法，作用是插入图片到指定单元格，支持 PNG、JPEG、BMP 等图片格式。参数 `row` 为行坐标，`col` 为列坐标，坐标索引起始值为 0；参数 `image` (string 类型) 为图片路径；参数 `options` (dict 类型) 为可选参数，作用是指定图片的位置、比例、链接 URL 等信息。操作示例如下：

```
# 在 B5 单元格插入 python-logo.png 图片，图片超级链接为 http://python.org
worksheet.insert_image('B5', 'img/python-logo.png', {'url': 'http://python.org'})
```

上述示例将创建一个如图 3-6 所示的工作表。

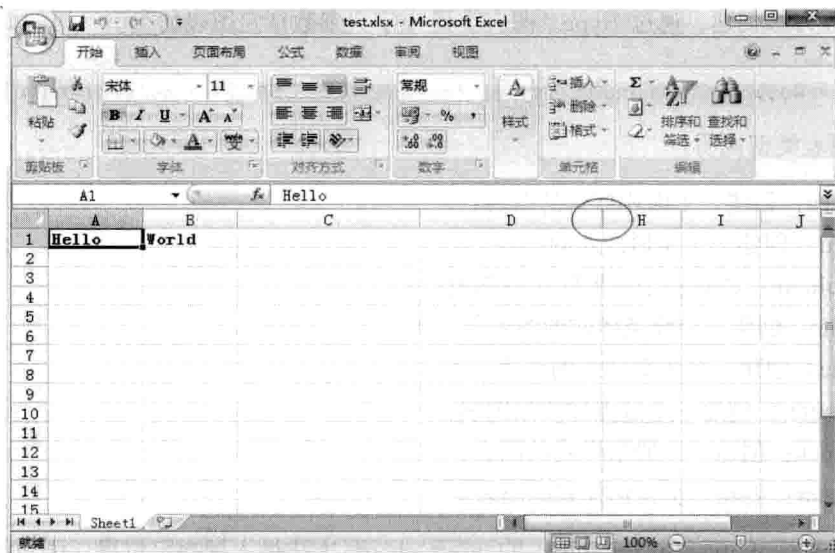


图 3-5 设置列单元格属性后的效果

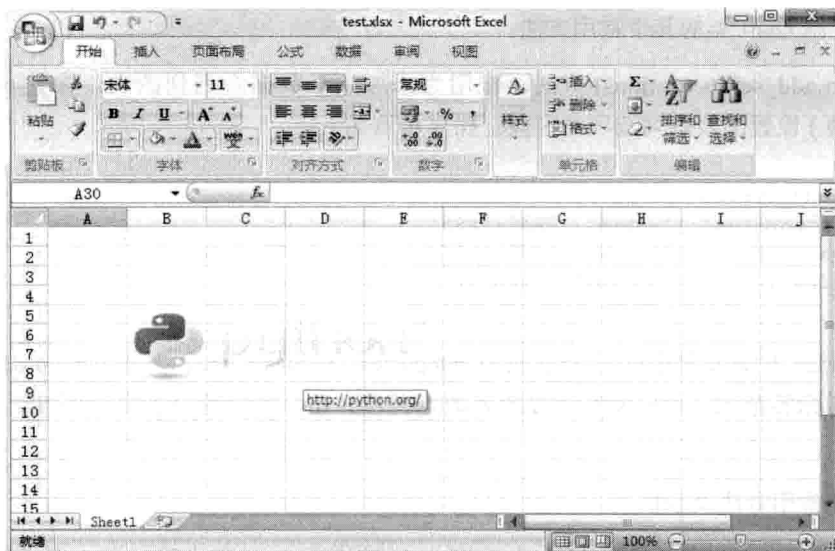


图 3-6 插入图片到单元格的效果

### 3. Chart 类

Chart 类实现在 XlsxWriter 模块中图表组件的基类，支持的图表类型包括面积、条形图、柱形图、折线图、饼图、散点图、股票和雷达等，一个图表对象是通过 Workbook（工作簿）

的 `add_chart` 方法创建, 通过 `{type, '图表类型'}` 字典参数指定图表的类型, 语句如下:

```
chart = workbook.add_chart({'type', 'column'})    # 创建一个 column(柱形) 图表
```

更多图表类型说明:

- `area`: 创建一个面积样式的图表;
- `bar`: 创建一个条形样式的图表;
- `column`: 创建一个柱形样式的图表;
- `line`: 创建一个线条样式的图表;
- `pie`: 创建一个饼图样式的图表;
- `scatter`: 创建一个散点样式的图表;
- `stock`: 创建一个股票样式的图表;
- `radar`: 创建一个雷达样式的图表。

然后再通过 `Worksheet` (工作表) 的 `insert_chart()` 方法插入到指定位置, 语句如下:

```
worksheet.insert_chart('A7', chart)    # 在 A7 单元格插入图表
```

下面介绍 `chart` 类的几个常用方法。

□ `chart.add_series(options)` 方法, 作用为添加一个数据系列到图表, 参数 `options` (dict 类型) 设置图表系列选项的字典, 操作示例如下:

```
chart.add_series({
    'categories': '=Sheet1!$A$1:$A$5',
    'values':      '=Sheet1!$B$1:$B$5',
    'line':        {'color': 'red'},
})
```

`add_series` 方法最常用的三个选项为 `categories`、`values`、`line`, 其中 `categories` 作为是设置图表类别标签范围; `values` 为设置图表数据范围; `line` 为设置图表线条属性, 包括颜色、宽度等。

□ 其他常用方法及示例。

○ `set_x_axis(options)` 方法, 设置图表 X 轴选项, 示例代码如下, 效果图如图 3-7 所示。

```
chart.set_x_axis({
    'name': 'Earnings per Quarter',    # 设置 X 轴标题名称
    'name_font': {'size': 14, 'bold': True},    # 设置 X 轴标题字体属性
    'num_font':  {'italic': True },    # 设置 X 轴数字字体属性
})
```

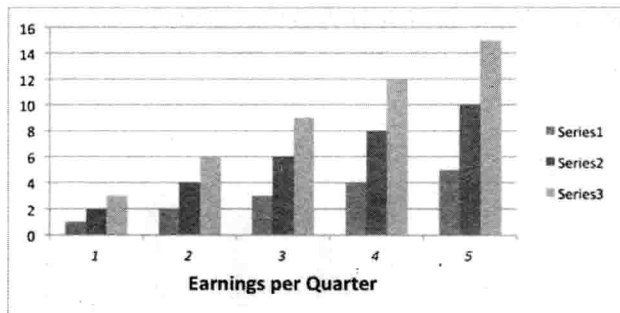


图 3-7 设置图表 X 轴选项

- `set_size(options)` 方法，设置图表大小，如 `chart.set_size({'width': 720, 'height': 576})`，其中 `width` 为宽度，`height` 为高度。
- `set_title(options)` 方法，设置图表标题，如 `chart.set_title({'name': 'Year End Results'})`，效果图如图 3-8 所示。

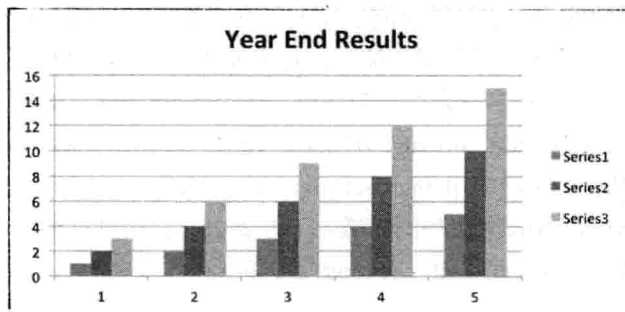


图 3-8 设置图表标题

- `set_style(style_id)` 方法，设置图表样式，`style_id` 为不同数字则代表不同样式，如 `chart.set_style(37)`，效果图如图 3-9 所示。

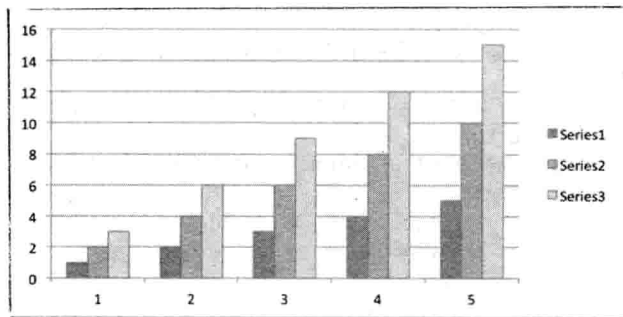


图 3-9 设置图表样式

○ `set_table(options)` 方法，设置 X 轴为数据表格形式，如 `chart.set_table()`，效果图如图 3-10 所示。

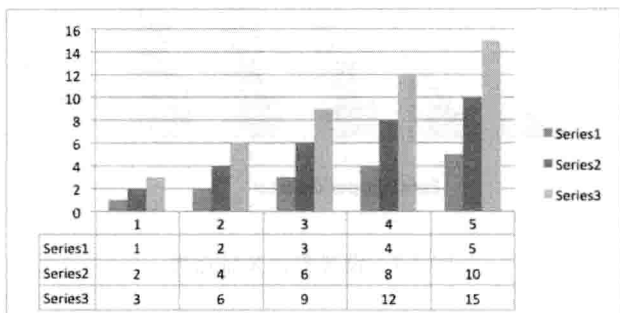


图 3-10 设置 X 轴为数据表格形式

### 3.1.2 实践：定制自动化业务流量报表周报

本次实践通过定制网站 5 个频道的流量报表周报，通过 `XlsxWriter` 模块将流量数据写入 Excel 文档，同时自动计算各频道周平均流量，再生成数据图表。具体是通过 `workbook.add_chart({'type': 'column'})` 方法指定图表类型为柱形，使用 `write_row`、`write_column` 方法分别以行、列方式写入数据，使用 `add_format()` 方法定制表头、表体的显示风格，使用 `add_series()` 方法将数据添加到图表，同时使用 `chart.set_size`、`set_title`、`set_y_axis` 设置图表的大小及标题属性，最后通过 `insert_chart` 方法将图表插入工作表中。我们可以结合 2.3 节的内容来实现周报的邮件推送，本示例略去此功能。实现的代码如下：

【 /home/test/XlsxWriter/simple2.py 】

```
#coding: utf-8
import xlsxwriter

workbook = xlsxwriter.Workbook('chart.xlsx')    # 创建一个 Excel 文件
worksheet = workbook.add_worksheet()           # 创建一个工作表对象
chart = workbook.add_chart({'type': 'column'})  # 创建一个图表对象
# 定义数据表头列表
title = ['u' 业务名称 'u' 星期一 'u' 星期二 'u' 星期三 'u' 星期四 'u' 星期五 'u' 星期六 'u' 星期日 'u' 平均流量]
buname= ['u' 业务官网 'u' 新闻中心 'u' 购物频道 'u' 体育频道 'u' 亲子频道]    # 定义频道名称
# 定义 5 频道一周 7 天流量数据列表
data = [
    [150,152,158,149,155,145,148],
    [89,88,95,93,98,100,99],
    [201,200,198,175,170,198,195],
    [75,77,78,78,74,70,79],
```

```

[88,85,87,90,93,88,84],
]
format=workbook.add_format()    # 定义 format 格式对象
format.set_border(1)           # 定义 format 对象单元格边框加粗 (1 像素) 的格式

format_title=workbook.add_format()    # 定义 format_title 格式对象
format_title.set_border(1)           # 定义 format_title 对象单元格边框加粗 (1 像素) 的格式
format_title.set_bg_color('#cccccc')  # 定义 format_title 对象单元格背景颜色为
                                      # '#cccccc' 的格式
format_title.set_align('center')     # 定义 format_title 对象单元格居中对齐的格式
format_title.set_bold()              # 定义 format_title 对象单元格内容加粗的格式

format_ave=workbook.add_format()    # 定义 format_ave 格式对象
format_ave.set_border(1)           # 定义 format_ave 对象单元格边框加粗 (1 像素) 的格式
format_ave.set_num_format('0.00')   # 定义 format_ave 对象单元格数字类别显示格式

# 下面分别以行或列写入方式将标题、业务名称、流量数据写入起初单元格, 同时引用不同格式对象
worksheet.write_row('A1', title, format_title)
worksheet.write_column('A2', buname, format)
worksheet.write_row('B2', data[0], format)
worksheet.write_row('B3', data[1], format)
worksheet.write_row('B4', data[2], format)
worksheet.write_row('B5', data[3], format)
worksheet.write_row('B6', data[4], format)

# 定义图表数据系列函数
def chart_series(cur_row):
    worksheet.write_formula('I'+cur_row, \
        '=AVERAGE(B'+cur_row+':H'+cur_row+')', format_ave)    # 计算 (AVERAGE 函数) 频
                                                                # 道周平均流量

    chart.add_series({
        'categories': '=Sheet1!$B$1:$H$1',    # 将“星期一至星期日”作为图表数据标签 (X 轴)
        'values':      '=Sheet1!$B$'+cur_row+':$H$'+cur_row,    # 频道一周所有数据作
                                                                # 为数据区域
        'line':        ({'color': 'black'},    # 线条颜色定义为 black (黑色)
        'name':         '=Sheet1!$A$'+cur_row,    # 引用业务名称为图例项
    })

for row in range(2, 7):    # 数据域以第 2 ~ 6 行进行图表数据系列函数调用
    chart_series(str(row))

# chart.set_table()    # 设置 X 轴表格格式, 本示例不启用
# chart.set_style(30)    # 设置图表样式, 本示例不启用
chart.set_size({'width': 577, 'height': 287})    # 设置图表大小
chart.set_title ({'name': u'业务流量周报图表'})    # 设置图表 (上方) 大标题
chart.set_y_axis ({'name': 'Mb/s'})    # 设置 y 轴 (左侧) 小标题

worksheet.insert_chart('A8', chart)    # 在 A8 单元格插入图表
workbook.close()    # 关闭 Excel 文档

```

上述示例将创建一个如图 3-11 所示的工作表。

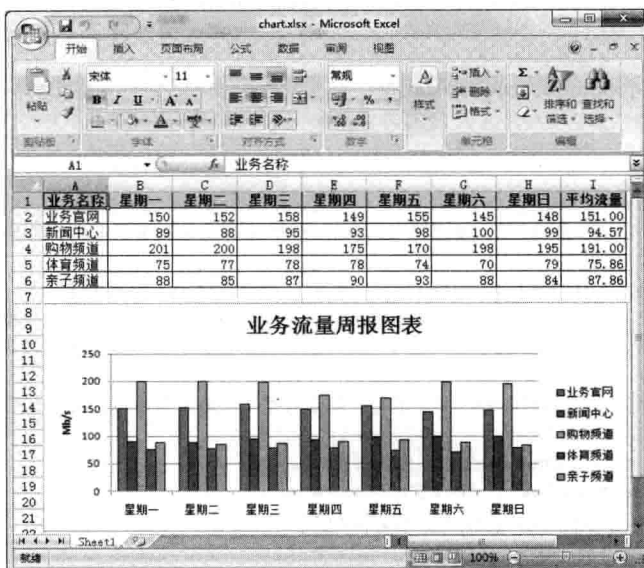


图 3-11 业务流量周报图表工作表



参考  
提示

3.4.1 节 XlsxWrite 模块的常用类与方法说明参考官网 <http://xlsxwriter.readthedocs.org>。

## 3.2 Python 与 rrdtool 的结合模块

rrdtool (round robin database) 工具为环状数据库的存储格式, round robin 是一种处理定量数据以及当前元素指针的技术。rrdtool 主要用来跟踪对象的变化情况, 生成这些变化的走势图, 比如业务的访问流量、系统性能、磁盘利用率等趋势图, 很多流行监控平台都使用到 rrdtool, 比较有名的为 Cacti、Ganglia、Monitorix 等。更多 rrdtool 介绍见官网 <http://oss.oetiker.ch/rrdtool/>。rrdtool 是一个复杂的工具, 涉及较多参数概念, 本节主要通过 Python 的 rrdtool 模块对 rrdtool 的几个常用方法进行封装, 包括 create、fetch、graph、info、update 等方法, 本节对 rrdtool 的基本知识不展开说明, 重点放在 Python rrdtool 模块的常用方法使用介绍上。

rrdtool 模块的安装方法如下:

```
easy_install python-rrdtool    #pip 安装方法
pip install python-rrdtool     #easy_install 安装方法
```

```
# 需要 rrdtool 工具及其他类包支持, CentOS 环境推荐使用 yum 安装方法
# yum install rrdtool-python
```

### 3.2.1 rrdtool 模块常用方法说明

下面介绍 rrdtool 模块常用的几个方法, 包括 create (创建 rrd)、update (更新 rrd)、graph (绘图)、fetch (查询 rrd) 等。

#### 1. Create 方法

create filename [--start|-b start time] [--step|-s step] [DS:ds-name:DST:heartbeat:min:max] [RRA:CF:xff:steps:rows] 方法, 创建一个后缀为 rrd 的 rrdtool 数据库, 参数说明如下:

- ❑ filename 创建的 rrdtool 数据库文件名, 默认后缀为 .rrd;
- ❑ --start 指定 rrdtool 第一条记录的起始时间, 必须是 timestamp 的格式;
- ❑ --step 指定 rrdtool 每隔多长时间就收到一个值, 默认为 5 分钟;
- ❑ DS 用于定义数据源, 用于存放脚本的结果的变量;
- ❑ DST 用于定义数据源类型, rrdtool 支持 COUNTER (递增类型)、DERIVE (可递增可递减类型)、ABSOLUTE (假定前一个时间间隔的值为 0, 再计算平均值)、GUAGE (收到值后直接存入 RRA)、COMPUTE (定义一个表达式, 引用 DS 并自动计算出某个值) 5 种, 比如网卡流量属于计数器型, 应该选择 COUNTER;
- ❑ RRA 用于指定数据如何存放, 我们可以把一个 RRA 看成一个表, 保存不同间隔的统计结果数据, 为 CF 做数据合并提供依据, 定义格式为: [RRA:CF:xff:steps:rows];
- ❑ CF 统计合并数据, 支持 AVERAGE (平均值)、MAX (最大值)、MIN (最小值)、LAST (最新值) 4 种方式。

#### 2. update 方法

update filename [--template|-t ds-name[:ds-name]...] N|timestamp:value[:value...] [timestamp:value[:value...] ...] 方法, 存储一个新值到 rrdtool 数据库, updatev 和 update 类似, 区别是每次插入后会返回一个状态码, 以便了解是否成功 (updatev 用 0 表示成功, -1 表示失败)。参数说明如下:

- ❑ filename 指定存储数据到的目标 rrd 文件名;
- ❑ -t ds-name[:ds-name] 指定需要更新的 DS 名称;
- ❑ N|Timestamp 表示数据采集的时间戳, N 表示当前时间戳;
- ❑ value[:value...] 更新的数据值, 多个 DS 则多个值。

### 3. graph 方法

graph filename [-s|--start seconds] [-e|--end seconds] [-x|--x-grid x-axis grid and label] [-y|--y-grid y-axis grid and label] [--alt-y-grid] [--alt-y-mrtg] [--alt-autoscale] [--alt-autoscale-max] [--units-exponent] value [-v|--vertical-label text] [-w|--width pixels] [-h|--height pixels] [-i|--interlaced] [-f|--imginfo formatstring] [-a|--imgformat GIF|PNG|GD] [-B|--background value] [-O|--overlay value] [-U|--unit value] [-z|--lazy] [-o|--logarithmic] [-u|--upper-limit value] [-l|--lower-limit value] [-g|--no-legend] [-r|--rigid] [--step value] [-b|--base value] [-c|--color COLORTAG#rrggb] [-t|--title title] [DEF:vname=rrd:ds-name:CF] [CDEF:vname=rpn-expression] [PRINT:vname:CF:format] [GPRINT:vname:CF:format] [COMMENT:text] [HRULE:value#rrggb[:legend]] [VRULE:time#rrggb[:legend]] [LINE{1|2|3}:vname[#rrggb[:legend]]] [AREA:vname[#rrggb[:legend]]] [STACK:vname[#rrggb[:legend]]] 方法，根据指定的 rrdtool 数据库进行绘图，关键参数说明如下：

- ❑ filename 指定输出图像的文件名，默认是 PNG 格式；
- ❑ --start 指定起始时间；
- ❑ --end 指定结束时间；
- ❑ --x-grid 控制 X 轴网格线刻度、标签的位置；
- ❑ --y-grid 控制 Y 轴网格线刻度、标签的位置；
- ❑ --vertical-label 指定 Y 轴的说明文字；
- ❑ --width pixels 指定图表宽度（像素）；
- ❑ --height pixels 指定图表高度（像素）；
- ❑ --imgformat 指定图像格式（GIF|PNG|GD）；
- ❑ --background 指定图像背景颜色，支持 #rrggb 表示法；
- ❑ --upper-limit 指定 Y 轴数据值上限；
- ❑ --lower-limit 指定 Y 轴数据值下限；
- ❑ --no-legend 取消图表下方的图例；
- ❑ --rigid 严格按照 upper-limit 与 lower-limit 来绘制；
- ❑ --title 图表顶部的标题；
- ❑ DEF:vname=rrd:ds-name:CF 指定绘图用到的数据源；
- ❑ CDEF:vname=rpn-expression 合并多个值；
- ❑ GPRINT:vname:CF:format 图表的下方输出最大值、最小值、平均值等；
- ❑ COMMENT:text 指定图表中输出的一些字符串；
- ❑ HRULE:value#rrggb 用于在图表上面绘制水平线；
- ❑ VRULE:time#rrggb 用于在图表上面绘制垂直线；
- ❑ LINE{1|2|3}:vname 使用线条来绘制数据图表，{1|2|3} 表示线条的粗细；

❑ AREA:vname 使用面积图来绘制数据图表。

#### 4. fetch 方法

fetch filename CF [--resolution|-r resolution] [--start|-s start] [--end|-e end] 方法，根据指定的 rrdtool 数据库进行查询，关键参数说明如下：

- ❑ filename 指定要查询的 rrd 文件名；
- ❑ CF 包括 AVERAGE、MAX、MIN、LAST，要求必须是建库时 RRA 中定义的类型，否则会报错；
- ❑ --start --end 指定查询记录的开始与结束时间，默认可省略。

### 3.2.2 实践：实现网卡流量图表绘制

在日常运营工作当中，观察数据的变化趋势有利于了解我们的服务质量，比如在系统监控方面，网络流量趋势图直接展现了当前网络的吞吐。CPU、内存、磁盘空间利用率趋势则反映了服务器运行健康状态。通过这些数据图表管理员可以提前做好应急预案，对可能存在的风险点做好防范。本次实践通过 rrdtool 模块实现服务器网卡流量趋势图的绘制，即先通过 create 方法创建一个 rrd 数据库，再通过 update 方法实现数据的写入，最后可以通过 graph 方法实现图表的绘制，以及提供 last、first、info、fetch 方法的查询。图 3-12 为 rrd 创建到输出图表的过程。

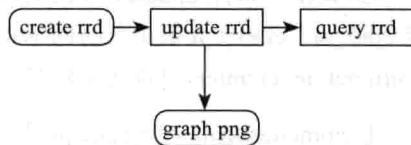


图 3-12 创建、更新 rrd 及输出图表流程

**第一步** 采用 create 方法创建 rrd 数据库，参数指定了一个 rrd 文件、更新频率 step、起始时间 --start、数据源 DS、数据源类型 DST、数据周期定义 RRA 等，详细源码如下：

【 /home/test/rrdtool/create.py 】

```

# -*- coding: utf-8 -*-
#!/usr/bin/python
import rrdtool
import time

cur_time=str(int(time.time())) # 获取当前 Linux 时间戳作为 rrd 起始时间
# 数据写频率 --step 为 300 秒（即 5 分钟一个数据点）
rrd=rrdtool.create('Flow.rrd', '--step', '300', '--start', cur_time,
# 定义数据源 eth0_in（入流量）、eth0_out（出流量）；类型都为 COUNTER（递增）；600 秒为心跳值，
# 其含义是 600 秒没有收到值，则会用 UNKNOWN 代替；0 为最小值；最大值用 U 代替，表示不确定
    'DS:eth0_in:COUNTER:600:0:U',
    'DS:eth0_out:COUNTER:600:0:U',

```

```

#RRA 定义格式为 [RRA:CF:xff:steps:rows], CF 定义了 AVERAGE、MAX、MIN 三种数据合并方式
#xff 定义为 0.5, 表示一个 CDP 中的 PDP 值如超过一半值为 UNKNOWN, 则该 CDP 的值就被标为 UNKNOWN
# 下列前 4 个 RRA 的定义说明如下, 其他定义与 AVERAGE 方式相似, 区别是存最大值与最小值
# 每隔 5 分钟 (1*300 秒) 存一次数据的平均值, 存 600 笔, 即 2.08 天
# 每隔 30 分钟 (6*300 秒) 存一次数据的平均值, 存 700 笔, 即 14.58 天 (2 周)
# 每隔 2 小时 (24*300 秒) 存一次数据的平均值, 存 775 笔, 即 64.58 天 (2 个月)
# 每隔 24 小时 (288*300 秒) 存一次数据的平均值, 存 797 笔, 即 797 天 (2 年)
'RRA:AVERAGE:0.5:1:600',
'RRA:AVERAGE:0.5:6:700',
'RRA:AVERAGE:0.5:24:775',
'RRA:AVERAGE:0.5:288:797',
'RRA:MAX:0.5:1:600',
'RRA:MAX:0.5:6:700',
'RRA:MAX:0.5:24:775',
'RRA:MAX:0.5:444:797',
'RRA:MIN:0.5:1:600',
'RRA:MIN:0.5:6:700',
'RRA:MIN:0.5:24:775',
'RRA:MIN:0.5:444:797')
if rrd:
    print rrdtool.error()

```

**第二步** 采用 `updatev` 方法更新 `rrd` 数据库, 参数指定了当前的 Linux 时间戳, 以及指定 `eth0_in`、`eth0_out` 值 (当前网卡的出入流量), 网卡流量我们通过 `psutil` 模块来获取, 如 `psutil.net_io_counters()[1]` 为入流量, 关于 `psutil` 模块的介绍见第 1.1。详细源码如下:

【 /home/test/rrdtool/update.py 】

```

# -*- coding: utf-8 -*-
#!/usr/bin/python
import rrdtool
import time, psutil

total_input_traffic = psutil.net_io_counters()[1] # 获取网卡入流量
total_output_traffic = psutil.net_io_counters()[0] # 获取网卡出流量
starttime=int(time.time()) # 获取当前 Linux 时间戳
# 将获取到的三个数据作为 updatev 的参数, 返回 {'return_value': 0L} 则说明更新成功, 反之失败
update=rrdtool.updatev('/home/test/rrdtool/Flow.rrd', '%s:%s:%s' %
(str(starttime), str(total_input_traffic), str(total_output_traffic)))
print update

```

将代码加入 `crontab`, 并配置 5 分钟作为采集频率, `crontab` 配置如下:

```
*/5 * * * * /usr/bin/python /home/test/rrdtool/update.py > /dev/null 2>&1
```

**第三步** 采用 `graph` 方法绘制图表, 此示例中关键参数使用了 `--x-grid` 定义 X 轴网格刻度; `DEF` 指定数据源; 使用 `CDEF` 合并数据; `HRULE` 绘制水平线 (告警线); `GPRINT` 输出最大值、最小值、平均值等。详细源码如下:

**[ /home/test/rrdtool/graph.py ]**

```
# -*- coding: utf-8 -*-
#!/usr/bin/python
import rrdtool
import time
# 定义图表上方大标题
title="Server network traffic flow (" + time.strftime('%Y-%m-%d', \
time.localtime(time.time())) + ")"
# 重点解释 "--x-grid", "MINUTE:12:HOUR:1:HOUR:1:0:%H" 参数的作用 (从左往右进行分解)
"MINUTE:12" 表示控制每隔 12 分钟放置一根次要格线
"HOUR:1" 表示控制每隔 1 小时放置一根主要格线
"HOUR:1" 表示控制 1 个小时输出一个 label 标签
"0:%H" 0 表示数字对齐格线, %H 表示标签以小时显示
rrdtool.graph( "Flow.png", "--start", "-1d", "--vertical-label=Bytes/s", \
"--x-grid", "MINUTE:12:HOUR:1:HOUR:1:0:%H", \
"--width", "650", "--height", "230", "--title", title,
"DEF:inoctets=Flow.rrd:eth0_in:AVERAGE",      # 指定网卡入流量数据源 DS 及 CF
"DEF:outoctets=Flow.rrd:eth0_out:AVERAGE",      # 指定网卡出流量数据源 DS 及 CF
"CDEF:total=inoctets,outoctets,+",      # 通过 CDEF 合并网卡出入流量, 得出总流量 total

"LINE1:total#FF8833:Total traffic",      # 以线条方式绘制总流量
"AREA:inoctets#00FF00:In traffic",      # 以面积方式绘制入流量
"LINE1:outoctets#0000FF:Out traffic",    # 以线条方式绘制出流量
"HRULE:6144#FF0000:Alarm value\\r",     # 绘制水平线, 作为告警线, 阈值为 6.1k
"CDEF:inbits=inoctets,8,*",             # 将入流量换算成 bit, 即 *8, 计算结果给 inbits
"CDEF:outbits=outoctets,8,*",           # 将出流量换算成 bit, 即 *8, 计算结果给 outbits
"COMMENT:\\r",                          # 在网格下方输出一个换行符
"COMMENT:\\r",
"GPRINT:inbits:AVERAGE:Avg In traffic\\: %6.2lf %Sbps",      # 绘制入流量平均值
"COMMENT: ",
"GPRINT:inbits:MAX:Max In traffic\\: %6.2lf %Sbps",           # 绘制入流量最大值
"COMMENT: ",
"GPRINT:inbits:MIN:MIN In traffic\\: %6.2lf %Sbps\\r",        # 绘制入流量最小值
"COMMENT: ",
"GPRINT:outbits:AVERAGE:Avg Out traffic\\: %6.2lf %Sbps",    # 绘制出流量平均值
"COMMENT: ",
"GPRINT:outbits:MAX:Max Out traffic\\: %6.2lf %Sbps",          # 绘制出流量最大值
"COMMENT: ",
"GPRINT:outbits:MIN:MIN Out traffic\\: %6.2lf %Sbps\\r")      # 绘制出流量最小值
```

以上代码将生成一个 Flow.png 文件, 如图 3-13 所示。



**提示**

查看 rrd 文件内容有利于观察数据的结构、更新等情况, rrdtool 提供几个常用命令:

- ☐ info 查看 rrd 文件的结构信息, 如 rrdtool info Flow.rrd;
- ☐ first 查看 rrd 文件第一个数据的更新时间, 如 rrdtool first Flow.rrd;
- ☐ last 查看 rrd 文件最近一次更新的时间, 如 rrdtool last Flow.rrd;
- ☐ fetch 根据指定时间、CF 查询 rrd 文件, 如 rrdtool fetch Flow.rrd AVERAGE。

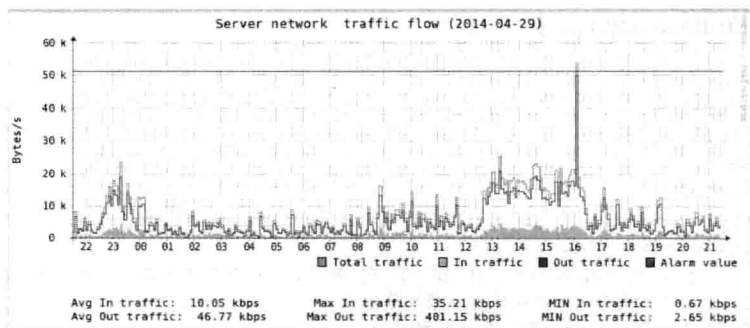


图 3-13 graph.py 执行输出图表



参考提示

3.2.1rrdtool 参数说明参考 <http://bbs.chinaunix.net/thread-2150417-1-1.html> 和 <http://oss.oetiker.ch/rrdtool/doc/index.en.html>。

### 3.3 生成动态路由轨迹图

scapy (<http://www.secdev.org/projects/scapy/>) 是一个强大的交互式数据包处理程序，它能够对数据包进行伪造或解包，包括发送数据包、包嗅探、应答和反馈匹配等功能。可以用在处理网络扫描、路由跟踪、服务探测、单元测试等方面，本节主要针对 scapy 的路由跟踪功能，实现 TCP 协议方式对服务可用性的探测，比如常用的 80 (HTTP) 与 443 (HTTPS) 服务，并生成美观的路由线路图报表，让管理员清晰了解探测点到目标主机的服务状态、骨干路由节点所处的 IDC 位置、经过的运营商路由节点等信息。下面详细介绍。

scapy 模块的安装方法如下：

```
# scapy 模板需要 tcpdump 程序支持，生成报表需要 graphviz、ImageMagick 图像处理包支持
# yum -y install tcpdump graphviz ImageMagick

# 源码安装
# wget http://www.secdev.org/projects/scapy/files/scapy-2.2.0.tar.gz
# tar -zxvf scapy-2.2.0.tar.gz
# cd scapy-2.2.0
# python setup.py install
```

#### 3.3.1 模块常用方法说明

scapy 模块提供了众多网络数据包操作的方法，包括发包 send()、SYN\ACK 扫描、嗅探

sniff()、抓包 wrpcap()、TCP 路由跟踪 traceroute() 等，本节主要关注服务监控内容接下来详细介绍 traceroute() 方法，其具体定义如下：

```
traceroute(target, dport=80, minttl=1, maxttl=30, sport=<RandShort>, l4=None, filter=None,
timeout=2, verbose=None, **kargs)
```

该方法实现 TCP 跟踪路由功能，关键参数说明如下：

- ❑ target：跟踪的目标对象，可以是域名或 IP，类型为列表，支持同时指定多个目标，如 ["www.qq.com", "www.baidu.com", "www.google.com.hk"]；
- ❑ dport：目标端口，类型为列表，支持同时指定多个端口，如 [80,443]；
- ❑ minttl：指定路由跟踪的最小跳数（节点数）；
- ❑ maxttl：指定路由跟踪的最大跳数（节点数）。

### 3.3.2 实践：实现 TCP 探测目标服务路由轨迹

在此次实践中，通过 scapy 的 traceroute() 方法实现探测机到目标服务器的路由轨迹，整个过程的原理见图 3-14，首先通过探测机以 SYN 方式进行 TCP 服务扫描，同时启动 tcpdump 进行抓包，捕获扫描过程经过的所有路由点，再通过 graph() 方法进行路由 IP 轨迹绘制，中间调用 ASN 映射查询 IP 地理信息并生成 svg 流程文档，最后使用 ImageMagick 工具将 svg 格式转换成 png，流程结束。

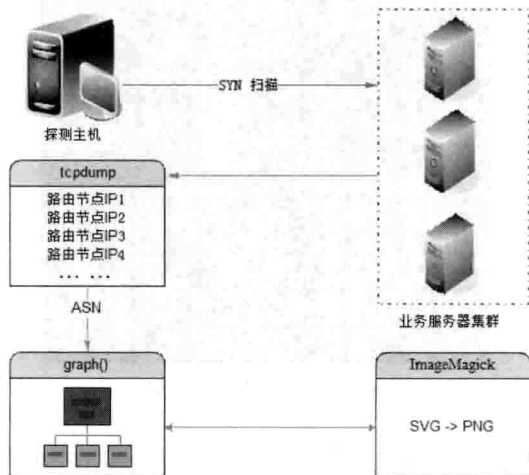


图 3-14 TCP 探测目标服务路由轨迹原理图

本次实践通过 traceroute() 方法实现路由的跟踪，跟踪结果动态生成图片格式。功能实现源码如下：

```
【 /home/test/scapy/simple1.py 】
```

```
# -*- coding: utf-8 -*-
import os,sys,time,subprocess
import warnings,logging
warnings.filterwarnings("ignore", category=DeprecationWarning) # 屏蔽 scapy 无用告警信息
logging.getLogger("scapy.runtime").setLevel(logging.ERROR) # 屏蔽模块 IPv6 多余告警
from scapy.all import traceroute

domains = raw_input('Please input one or more IP/domain: ') # 接受输入的域名或 IP
target = domains.split(' ')
dport = [80] # 扫描的端口列表

if len(target) >= 1 and target[0]!='':
    res,unans = traceroute(target,dport=dport,retry=-2) # 启动路由跟踪
    res.graph(target=> test.svg") # 生成 svg 矢量图形
    time.sleep(1)
    subprocess.Popen("/usr/bin/convert test.svg test.png", shell=True) #svg 转 png 格式
else:
    print "IP/domain number of errors,exit"
```

代码运行结果见图 3-15,“-”表示路由节点无回应或超时;“11”表示扫描的指定服务无回应;“SA”表示扫描的指定服务有回应,一般是最后一个主机 IP。

```
Received 73 packets, got 39 answers, remaining 21 packets
113.108.238.121:tcp80 180.96.12.11:tcp80
1 192.168.1.1 11 192.168.1.1 11
2 114.116.64.1 11 114.116.64.1 11
3 10.145.209.26 11 10.145.209.26 11
4 10.144.12.74 11 10.145.209.25 11
5 10.144.10.66 11 10.144.10.66 11
6 10.144.10.206 11 10.144.10.206 11
7 10.144.12.153 11 10.144.12.153 11
8 10.83.64.1 11 10.83.64.1 11
11 10.252.253.1 11 10.252.253.1 11
18 202.97.49.221 11
20 113.108.238.121 SA 202.102.69.22 11
21 113.108.238.121 SA -
22 113.108.238.121 SA -
23 113.108.238.121 SA 180.96.12.11 SA
24 113.108.238.121 SA 180.96.12.11 SA
25 113.108.238.121 SA 180.96.12.11 SA
26 113.108.238.121 SA 180.96.12.11 SA
27 113.108.238.121 SA 180.96.12.11 SA
28 113.108.238.121 SA 180.96.12.11 SA
29 113.108.238.121 SA 180.96.12.11 SA
30 113.108.238.121 SA 180.96.12.11 SA
```

图 3-15 代码运行结果

生成的路由轨迹图见图 3-16 (仅局部),“-”将使用 unk\* 单元代替,重点路由节点将通过 ASN 获取所处的运营商或 IDC 位置,如 IP “202.102.69.210”为“CHINANET-JS-AS-AP AS Number for CHINANET jiangsu province backbone,CN”意思为该 IP 所处中国电信江苏省骨干网。

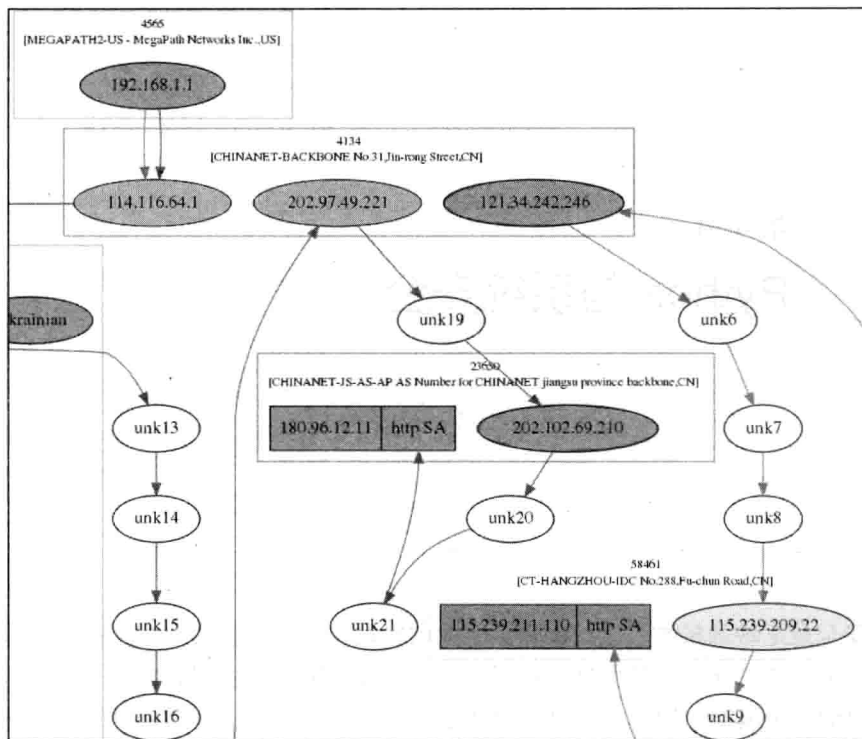


图 3-16 路由轨迹图

通过路由轨迹图，我们可以非常清晰地看到探测点到目标节点的路由走向，运营商时常会做路由节点分流，不排除会造成选择的路由线路不是最优的，该视图可以帮助我们了解到这个信息。另外 IE8 以上及 chrome 浏览器都已支持 SVG 格式文件，可以直接浏览，无需转换成 png 或其他格式，可以轻松整合到我们的运营平台当中。



参考提示

3.3.1 节 scapy 方法参数说明参考 <http://www.secdev.org/projects/scapy/doc/usage.html>。

# Python 与系统安全

信息安全是运维的根本，直接关系到企业的安危，稍有不慎会造成灾难性的后果。比如近年发生的多个知名网站会员数据库外泄事件，另外，国内知名漏洞报告平台乌云也频频爆出各大门户的安全漏洞。因此，信息安全体系建设已经被提到了前所未有的高度。如何提升企业的安全防范水准是目前普遍面临的问题，大体上主要分以下几个方面，包括安全设备防护、提高人员安全意识、实施系统平台安全加固、安全规范融合到 ITIL 体系、关注最新安全发展动向等，通过上述几个方面可以在很大程度上避免出现安全事故。本章主要讲述如何通过 Python 来实现系统级的安全防范策略，包括构建集中式的病毒扫描机制、端口安全扫描、安全密码生成等。

## 4.1 构建集中式的病毒扫描机制

Clam AntiVirus (ClamAV) 是一款免费而且开放源代码的防毒软件，软件与病毒库的更新皆由社区免费发布，官网地址：<http://www.clamav.net/lang/en/>。目前 ClamAV 主要为 Linux、Unix 系统提供病毒扫描、查杀等服务。pyClamad (<http://xael.org/norman/python/pyclamd/>) 是一个 Python 第三方模块，可让 Python 直接使用 ClamAV 病毒扫描守护进程 clamd，来实现一个高效的病毒检测功能，另外，pyClamad 模块也非常容易整合到我们已有的平台当中。下面详细进行说明。

pyClamad 模块的安装方法如下：

# 1、客户端（病毒扫描源）安装步骤

```
# yum install -y clamav clamd clamav-update      # 安装 clamavp 相关程序包
# chkconfig --levels 235 clamd on                # 添加扫描守护进程 clamd 系统服务
# /usr/bin/freshclam                             # 更新病毒库, 建议配置到 crontab 中定期更新
# setenforce 0                                    # 关闭 SELinux, 避免远程扫描时提示无权限的问题

# 更新守护进程监听 IP 配置文件, 根据不同环境自行修改监听的 IP, "0.0.0.0" 为监听所有主机 IP
# sed -i -e '/^TCPAddr/{ s/127.0.0.1/0.0.0.0/; }' /etc/clamd.conf
# /etc/init.d/clamd start                        # 启动扫描守护进程

# 2、主控端部署 pyClamad 环境步骤
# wget http://xael.org/norman/python/pyclamd/pyClamd-0.3.4.tar.gz
# tar -zxvf pyClamd-0.3.4.tar.gz
# cd pyClamd-0.3.4
# python setup.py install
```

### 4.1.1 模块常用方法说明

pyClamad 提供了两个关键类, 一个为 ClamdNetworkSocket() 类, 实现使用网络套接字操作 clamd; 另一个为 ClamdUnixSocket() 类, 实现使用 Unix 套接字类操作 clamd。两个类定义的方法完全一样, 本节以 ClamdNetworkSocket() 类进行说明。

- ❑ `__init__(self, host='127.0.0.1', port=3310, timeout=None)` 方法, 是 ClamdNetworkSocket 类的初始化方法, 参数 `host` 为连接主机 IP; 参数 `port` 为连接的端口, 默认为 3310, 与 `/etc/clamd.conf` 配置文件中的 `TCPsocket` 参数要保持一致; `timeout` 为连接的超时时间。
- ❑ `contscan_file(self, file)` 方法, 实现扫描指定的文件或目录, 在扫描时发生错误或发现病毒将不终止, 参数 `file` (string 类型) 为指定的文件或目录的绝对路径。
- ❑ `multiscan_file(self, file)` 方法, 实现多线程扫描指定的文件或目录, 多核环境速度更快, 在扫描时发生错误或发现病毒将不终止, 参数 `file` (string 类型) 为指定的文件或目录的绝对路径。
- ❑ `scan_file(self, file)` 方法, 实现扫描指定的文件或目录, 在扫描时发生错误或发现病毒将终止, 参数 `file` (string 类型) 为指定的文件或目录的绝对路径。
- ❑ `shutdown(self)` 方法, 实现强制关闭 clamd 进程并退出。
- ❑ `stats(self)` 方法, 获取 Clamscan 的当前状态。
- ❑ `reload(self)` 方法, 强制重载 clamd 病毒特征库, 扫描前建议做 reload 操作。
- ❑ `EICAR(self)` 方法, 返回 EICAR 测试字符串, 即生成具有病毒特征的字符串, 便于测试。

### 4.1.2 实践：实现集中式的病毒扫描

本次实践实现了一个集中式的病毒扫描管理, 可以针对不同业务环境定制扫描策略, 比如扫描对象、描述模式、扫描路径、调度频率等。示例实现的架构见图 4-1, 首先业务服务

器开启 clamd 服务 (监听 3310 端口), 管理服务器启用多线程对指定的服务集群进行扫描, 扫描模式、扫描路径会传递到 clamd, 最后返回扫描结果给管理服务器端。

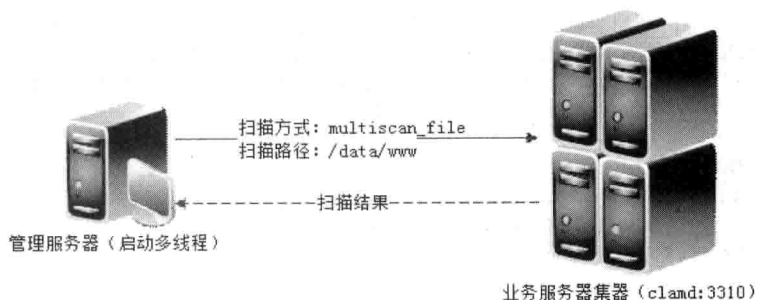


图 4-1 集群病毒扫描架构图

本次实践通过 ClamdNetworkSocket() 方法实现与业务服务器建立扫描 socket 连接, 再通过启动不同扫描方式实施病毒扫描并返回结果。实现代码如下:

【 /home/test/pyClamad/simple1.py 】

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
import time
import pyclamd
from threading import Thread

class Scan(Thread):
    def __init__(self, IP, scan_type, file):
        """ 构造方法, 参数初始化 """
        Thread.__init__(self)
        self.IP = IP
        self.scan_type = scan_type
        self.file = file
        self.connstr = ""
        self.scanresult = ""

    def run(self):
        """ 多进程 run 方法 """
        try:
            cd = pyclamd.ClamdNetworkSocket(self.IP, 3310)  # 创建网络套接字连接对象
            if cd.ping():  # 探测连通性
                self.connstr = self.IP + " connection [OK]"
                cd.reload()  # 重载 clamd 病毒特征库, 建议更新病毒库后做 reload() 操作
                if self.scan_type == "contscan_file":  # 选择不同的扫描模式
                    self.scanresult = "{0}\n".format(cd.contscan_file(self.file))
                elif self.scan_type == "multiscan_file":
                    self.scanresult = "{0}\n".format(cd.multiscan_file(self.file))

```

```

        elif self.scan_type=="scan_file":
            self.scanresult="{0}\n".format(cd.scan_file(self.file))
            time.sleep(1)    # 线程挂起 1 秒
        else:
            self.connstr=self.IP+" ping error,exit"
            return
    except Exception,e:
        self.connstr=self.IP+" "+str(e)

IPs=['192.168.1.21','192.168.1.22']    # 扫描主机列表
scantype="multiscan_file"    # 指定扫描模式, 支持 multiscan_file、contscan_file、scan_file
scanfile="/data/www"    # 指定扫描路径
i=1

threadnum=2    # 指定启动的线程数
scanlist = []    # 存储扫描 Scan 类线程对象列表

for ip in IPs:

    currp = Scan(ip,scantype,scanfile)    # 创建扫描 Scan 类对象, 参数 (IP, 扫描模式, 扫描路径)
    scanlist.append(currp)    # 追加对象到列表

    if i%threadnum==0 or i==len(IPs):    # 当达到指定的线程数或 IP 列表数后启动、退出线程
        for task in scanlist:
            task.start()    # 启动线程

            for task in scanlist:
                task.join()    # 等待所有子线程退出, 并输出扫描结果
                print task.connstr    # 打印服务器连接信息
                print task.scanresult    # 打印扫描结果
            scanlist = []
        i+=1

```

通过 EICAR() 方法生成一个带有病毒特征的文件 /tmp/EICAR, 代码如下:

```
void = open('/tmp/EICAR','w').write(cd.EICAR())
```

生成带有病毒特征的字符串内容如下, 复制文件 /tmp/EICAR 到目标主机的扫描目录当中, 以便进行测试。

```

#cat /tmp/EICAR
u'X5O!P%@AP[4\\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*'

```

最后, 启动扫描程序, 在本次实践过程中启用两个线程, 可以根据目标主机数量随意修改, 代码运行结果如图 4-2, 其中 192.168.1.21 主机没有发现病毒, 192.168.1.22 主机发现了病毒测试文件 EICAR。

```
[root@SN2013-08-020 pyClamad]# python simple1.py
192.168.1.21 connection [OK]
None
192.168.1.22 connection [OK]
{'u'/data/www/lwebadmin/EICAR': ('FOUND', 'Eicar-Test-Signature')}
```

图 4-2 集中式病毒扫描程序运行结果



参考提示

4.1.1 节 pyClamad 模块方法说明参考 <http://xael.org/norman/python/pyclamd/pyclamd.html>。

## 4.2 实现高效的端口扫描器

如今互联网安全形势日趋严峻，给系统管理员带来很大的挑战，网络的开放性以及黑客的攻击是造成网络不安全的主因。稍有疏忽将给黑客带来可乘之机，给企业带来无法弥补的损失。比如由于系统管理员误操作，导致核心业务服务器的 22、21、3389、3306 等高危端口暴露在互联网上，大大提高了被入侵的风险。因此，定制一种规避此安全事故的机制已经迫在眉睫。本节主要讲述通过 Python 的第三方模块 python-nmap 来实现高效的端口扫描，达到发现异常时可以在第一时间发现并处理，将安全风险降到最低的目的。python-nmap 模块作为 nmap 命令的 Python 封装，可以让 Python 很方便地操作 nmap 扫描器，它可以帮助管理员完成自动扫描任务和生成报告。

python-nmap 模块的安装方法如下：

```
# yum -y install nmap      # 安装 nmap 工具
# 模块源码安装
# wget http://xael.org/norman/python/python-nmap/python-nmap-0.1.4.tar.gz
# tar -zxvf python-nmap-0.1.4.tar.gz
# cd python-nmap-0.1.4
# python setup.py install
```

### 4.2.1 模块常用方法说明

本节介绍 python-nmap 模块的两个常用类，一个为 PortScanner() 类，实现一个 nmap 工具的端口扫描功能封装；另一个为 PortScannerHostDict() 类，实现存储与访问主机的扫描结果，下面介绍 PortScanner() 类的一些常用方法。

- ❑ scan(self, hosts='127.0.0.1', ports=None, arguments='-sV') 方法，实现指定主机、端口、nmap 命令行参数的扫描。参数 hosts 为字符串类型，表示扫描的主机地址，格式可以用 “scanme.nmap.org”、“198.116.0-255.1-127”、“216.163.128.20/20” 表示；参数

ports 为字符串类型，表示扫描的端口，可以用“22,53,110,143-4564”来表示；参数 arguments 为字符串类型，表示 nmap 命令行参数，格式为“-sU -sX -sC”，例如：

```
nm = nmap.PortScanner()
nm.scan('192.168.1.21-22', '22,80')
```

❑ `command_line(self)` 方法，返回的扫描方法映射到具体 nmap 命令行，如：

```
>>> nm.command_line()
u'nmap -oX - -p 22,80 -sV 192.168.1.21-22'
```

❑ `scaninfo(self)` 方法，返回 nmap 扫描信息，格式为字典类型，如：

```
>>> nm.scaninfo()
{'u'tcp': {'services': u'22,80', 'method': u'syn'}}
```

❑ `all_hosts(self)` 方法，返回 nmap 扫描的主机清单，格式为列表类型，如：

```
[u'192.168.1.21', u'192.168.1.22']
```

以下介绍 PortScannerHostDict () 类的一些常用方法。

❑ `hostname(self)` 方法，返回扫描对象的主机名，如：

```
>>> nm['192.168.1.22'].hostname()
u'SN2013-08-022'
```

❑ `state(self)` 方法，返回扫描对象的状态，包括4种状态（up、down、unknown、skipped），如：

```
>>> nm['192.168.1.22'].state()
u'up'
```

❑ `all_protocols(self)` 方法，返回扫描的协议，如：

```
>>> nm['192.168.1.22'].all_protocols()
[u'tcp']
```

❑ `all_tcp()` (self) 方法，返回 TCP 协议扫描的端口，如：

```
>>> nm['192.168.1.22'].all_tcp()
[22, 80]
```

❑ `tcp(self, port)` 方法，返回扫描 TCP 协议 port（端口）的信息，如：

```
>>> nm['192.168.1.22'].tcp(22)
{'state': u'open', 'reason': u'syn-ack', 'name': u'ssh'}
```

## 4.2.2 实践：实现高效的端口扫描

本次实践通过 python-nmap 实现一个高效的端口扫描工具，与定时作业 crontab 及邮件告警结合，可以很好地帮助我们及时发现异常开放的高危端口。当然，该工具也可以作为业务服务端口的可用性探测，例如扫描 192.168.1.20-25 网段 Web 服务端口 80 是否处于 open 状态。实践所采用的 scan() 方法的 arguments 参数指定为 “-v -PE -p '+' 端口”，-v 表示启用细节模式，可以返回非 up 状态主机清单；-PE 表示采用 TCP 同步扫描 (TCP SYN) 方式；-p 指定扫描端口范围。程序输出部分采用了三个 for 循环体，第一层遍历扫描主机，第二层为遍历协议，第三层为遍历端口，最后输出主机状态。具体实现代码如下：

【 /home/test/python-nmap/simple1.py 】

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import sys
import nmap

scan_row=[]
input_data = raw_input('Please input hosts and port: ')
scan_row = input_data.split(" ")
if len(scan_row)!=2:
    print "Input errors,example \"192.168.1.0/24 80,443,22\""
    sys.exit(0)
hosts=scan_row[0]    # 接收用户输入的主机
port=scan_row[1]    # 接收用户输入的端口

try:
    nm = nmap.PortScanner()    # 创建端口扫描对象
except nmap.PortScannerError:
    print('Nmap not found', sys.exc_info()[0])
    sys.exit(0)
except:
    print("Unexpected error:", sys.exc_info()[0])
    sys.exit(0)

try:
    # 调用扫描方法，参数指定扫描主机 hosts，nmap 扫描命令行参数 arguments
    nm.scan(hosts=hosts, arguments='-v -sS -p '+'port)
except Exception,e:
    print "Scan error:"+str(e)

for host in nm.all_hosts():    # 遍历扫描主机
    print('-----')
    print('Host : %s (%s)' % (host, nm[host].hostname()))    # 输出主机及主机名
    print('State : %s' % nm[host].state())    # 输出主机状态，如 up、down
```

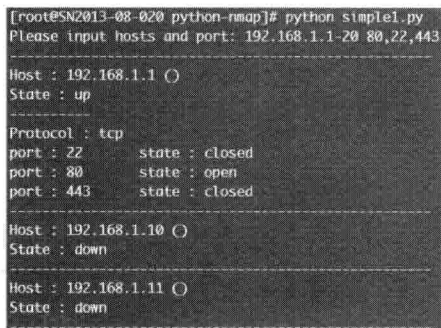
```

for proto in nm[host].all_protocols():    # 遍历扫描协议, 如 tcp、udp
    print('-----')
    print('Protocol : %s' % proto)        # 输入协议名

    lport = nm[host][proto].keys()        # 获取协议的所有扫描端口
    lport.sort()                          # 端口列表排序
    for port in lport:                    # 遍历端口及输出端口与状态
        print('port : %s\tstate : %s' % (port, nm[host][proto][port]['state']))

```

其中主机输入支持所有表达方式, 如 `www.qq.com`、`192.168.1.*`、`192.168.1.1-20`、`192.168.1.0/24` 等, 端口输入格式也非常灵活, 如 `80,443,22`、`80,22-443`。代码运行结果如图 4-3 所示。



```

[root@SN2013-08-020 python-nmap]# python simple1.py
Please input hosts and port: 192.168.1.1-20 80,22,443

Host : 192.168.1.1 ()
State : up

Protocol : tcp
port : 22      state : closed
port : 80      state : open
port : 443     state : closed

Host : 192.168.1.10 ()
State : down

Host : 192.168.1.11 ()
State : down

```

图 4-3 指定 IP 段与端口的扫描结果



参考  
提示

4.2.1 节 Python-nmap 模块方法与参数说明参考 <http://xael.org/norman/python/python-nmap/>。示例源码参考官方源码包中的 `example.py`。

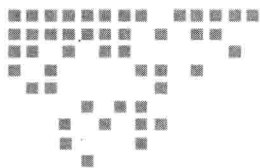




## 第二部分 *Part 2*

# 高级篇

- 第 5 章 系统批量运维管理器 pexpect 详解
  - 第 6 章 系统批量运维管理器 paramiko 详解
  - 第 7 章 系统批量运维管理器 Fabric 详解
  - 第 8 章 从“零”开发一个轻量级 WebServer
  - 第 9 章 集中化管理平台 Ansible 详解
  - 第 10 章 集中化管理平台 Saltstack 详解
  - 第 11 章 统一网络控制器 Func 详解
  - 第 12 章 Python 大数据应用详解
- .....



Chapter 3

## 第 5 章

# 系统批量运维管理器 pexpect 详解

pexpect 可以理解成 Linux 下的 expect 的 Python 封装，通过 pexpect 我们可以实现对 ssh、ftp、passwd、telnet 等命令行进行自动交互，而无需人工干涉来达到自动化的目的。比如我们可以模拟一个 FTP 登录时的所有交互，包括输入主机地址、用户名、密码、上传文件等，待出现异常我们还可以进行尝试自动处理。pexpect 的官网地址：<http://pexpect.readthedocs.org/en/latest/>，目前最高版本为 3.0。

## 5.1 pexpect 的安装

pexpect 作为 Python 的一个普通模块，支持 pip、easy\_install 或源码安装方式，具体安装命令如下（根据用户环境，自行选择 pip 或 easy\_install）：

```
pip install pexpect
easy_install pexpect
```

关于源码安装，笔者采用了 GitHub 平台的项目托管源，安装步骤如下：

```
#wget https://github.com/pexpect/pexpect/releases/download/3.0/pexpect-3.0.tar.gz
#tar -zxvf pexpect-3.0.tar.gz
#cd pexpect-3.0
#python setup.py install
```

校验安装结果，导入模块没有提示异常则说明安装成功：

```
# python
Python 2.6.6 (r266:84292, Jul 10 2013, 22:48:45)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pexpect
>>>
```

一个简单实现 SSH 自动登录的示例如下：

```
import pexpect
child = pexpect.spawn('scp foo user@example.com:..') #spawn 启动 scp 程序
child.expect('Password:') #expect 方法等待子程序产生的输出，判断是否匹配定义的字符串
                           #'Password:'
child.sendline(mypassword) # 匹配后则发送密码串进行回应
```

## 5.2 pexpect 的核心组件

下面介绍 pexpect 的几个核心组件包括 spawn 类、run 函数及派生类 pssh 等的定义及使用方法。

### 5.2.1 spawn 类

spawn 是 pexpect 的主要类接口，功能是启动和控制子应用程序，以下是它的构造函数定义：

```
class pexpect.spawn(command, args=[], timeout=30, maxread=2000,
searchwindowsize=None, logfile=None, cwd=None, env=None, ignore_sighup=True)
```

其中 command 参数可以是任意已知的系统命令，比如：

```
child = pexpect.spawn('/usr/bin/ftp') # 启动 ftp 客户端命令
child = pexpect.spawn('/usr/bin/ssh user@example.com') # 启动 ssh 远程连接命令
child = pexpect.spawn('ls -latr /tmp') # 运行 ls 显示 /tmp 目录内容命令
```

当子程序需要参数时，还可以使用 Python 列表来代替参数项，如：

```
child = pexpect.spawn ('/usr/bin/ftp', [])
child = pexpect.spawn ('/usr/bin/ssh', ['user@example.com'])
child = pexpect.spawn ('ls', ['-latr', '/tmp'])
```

参数 timeout 为等待结果的超时时间；参数 maxread 为 pexpect 从终端控制台一次读取的最大字节数，searchwindowsize 参数为匹配缓冲区字符串的位置，默认是从开始位置匹配。

需要注意的是，pexpect 不会解析 shell 命令当中的元字符，包括重定向“>”、管道“|”

或通配符“\*”，当然，我们可以通过一个技巧来解决这个问题，将存在这三个特殊元字符的命令作为 `/bin/bash` 的参数进行调用，例如：

```
child = pexpect.spawn('/bin/bash -c "ls -l | grep LOG > logs.txt"')
child.expect(pexpect.EOF)
```

我们可以通过将命令的参数以 Python 列表的形式进行替换，从而使我们的语法变成更加清晰，下面的代码等价于上面的。

```
shell_cmd = 'ls -l | grep LOG > logs.txt'
child = pexpect.spawn('/bin/bash', ['-c', shell_cmd])
child.expect(pexpect.EOF)
```

有时候调试代码时，希望获取 `pexpect` 的输入与输出信息，以便了解匹配的情况。`pexpect` 提供了两种途径，一种为写到日志文件，另一种为输出到标准输出。写到日志文件的实现方法如下：

```
child = pexpect.spawn('some_command')
fout = file('mylog.txt', 'w')
child.logfile = fout
```

输出到标准输出的方法如下：

```
child = pexpect.spawn('some_command')
child.logfile = sys.stdout
```

下面为一个完整的示例，实现远程 SSH 登录，登录成功后显示 `/home` 目录文件清单，并通过日志文件记录所有的输入与输出。

```
import pexpect
import sys

child = pexpect.spawn('ssh root@192.168.1.21')
fout = file('mylog.txt', 'w')
child.logfile = fout
#child.logfile = sys.stdout

child.expect("password:")
child.sendline("U3497DT32t")
child.expect('#')
child.sendline('ls /home')
child.expect('#')
```

以下为 `mylog.txt` 日志内容，可以看到 `pexpect` 产生的全部输入与输出信息。

```
# cat mylog.txt
```

```

root@192.168.1.21's password: U3497DT32t

Last login: Tue Jan  7 23:05:30 2014 from 192.168.1.20
[root@SN2013-08-021 ~]# ls /home
ls /home
cc.py          poster-0.8.1          tarfile.tar.gz  zipfile.zip
default.tar.gz poster-0.8.1.tar.gz    test.sh
dev            pypa-setuptools-c508be8585ab  zipfile1.zip

```

### (1) expect 方法

expect 定义了一个子程序输出的匹配规则。

方法定义：expect(pattern, timeout=1, searchwindowsize=1)

其中，参数 pattern 表示字符串、pexpect.EOF（指向缓冲区尾部，无匹配项）、pexpect.TIMEOUT（匹配等待超时）、正则表达式或者前面四种类型组成的列表（List），当 pattern 为一个列表时，且不止一个表列元素被匹配，则返回的结果是子程序输出最先出现的那个元素，或者是列表最左边的元素（最小索引 ID），如：

```

import pexpect
child = pexpect.spawn("echo 'foobar'")
print child.expect(['bar', 'foo', 'foobar'])
输出:1, 即 'foo' 被匹配

```

参数 timeout 指定等待匹配结果的超时时间，单位为秒。当超时被触发时，expect 将匹配到 pexpect.TIMEOUT；参数 searchwindowsize 为匹配缓冲区字符串的位置，默认是从开始位置匹配。

当 pexpect.EOF、pexpect.TIMEOUT 作为 expect 的列表参数时，匹配时将返回所处列表中的索引 ID，例如：

```

index = p.expect(['good', 'bad', pexpect.EOF, pexpect.TIMEOUT])
if index == 0:
    do_something()
elif index == 1:
    do_something_else()
elif index == 2:
    do_some_other_thing()
elif index == 3:
    do_something_completely_different()

```

以上代码等价于

```

try:
    index = p.expect(['good', 'bad'])
    if index == 0:

```

```

        do_something()
    elif index == 1:
        do_something_else()
except EOF:
    do_some_other_thing()
except TIMEOUT:
    do_something_completely_different()

```

**expect** 方法有两个非常棒的成员：**before** 与 **after**。**before** 成员保存了最近匹配成功之前的内容，**after** 成员保存了最近匹配成功之后的内容。例如：

```

import pexpect
import sys

child = pexpect.spawn('ssh root@192.168.1.21')
fout = file('mylog.txt', 'w')
child.logfile = fout

child.expect(["password:"])
child.sendline("980405")
print "before:" + child.before
print "after:" + child.after

```

运行结果如下：

```

before:root@192.168.1.21's
after:password:

```

## (2) read 相关方法

下面这些输入方法的作用都是向子程序发送响应命令，可以理解成代替了我们的标准输入键盘。

```

send(self, s) 发送命令，不回车
sendline(self, s='') 发送命令，回车
sendcontrol(self, char) 发送控制字符，如 child.sendcontrol('c') 等价于" ctrl+c"
sendeof() 发送 eof

```

## 5.2.2 run 函数

**run** 是使用 **pexpect** 进行封装的调用外部命令的函数，类似于 **os.system** 或 **os.popen** 方法，不同的是，使用 **run()** 可以同时获得命令的输出结果及命令的退出状态，函数定义：**pexpect.run(command, timeout=1, withexitstatus=False, events=None, extra\_args=None, logfile=None, cwd=None, env=None)**。

参数 **command** 可以是系统已知的任意命令，如没有写绝对路径时将会尝试搜索命令的

路径, events 是一个字典, 定义了 expect 及 sendline 方法的对应关系, spawn 方式的例子如下:

```
from pexpect import *
child = spawn('scp foo user@example.com:..')
child.expect('(?!i)password')
child.sendline(mypassword)
```

使用 run 函数实现如下, 是不是更加简洁、精炼了?

```
from pexpect import *
run('scp foo user@example.com:..', events={'(?!i)password': mypassword})
```

### 5.2.3 pxssh 类

pxssh 是 pexpect 的派生类, 针对在 ssh 会话操作上再做一层封装, 提供与基类更加直接的操作方法。

pxssh 类定义:

```
class pexpect.pxssh.pxssh(timeout=30, maxread=2000, searchwindowsize=None,
logfile=None, cwd=None, env=None)
```

pxssh 常用的三个方法如下:

- ❑ login() 建立 ssh 连接;
- ❑ logout() 断开连接;
- ❑ prompt() 等待系统提示符, 用于等待命令执行结束。

下面使用 pxssh 类实现一个 ssh 连接远程主机并执行命令的示例。首先使用 login() 方法与远程主机建立连接, 再通过 sendline() 方法发送执行的命令, prompt() 方法等待命令执行结束且出现系统提示符, 最后使用 logout() 方法断开连接。

【 /home/test/pexpect/ simple1.py 】

```
import pxssh
import getpass
try:
    s = pxssh.pxssh()      # 创建 pxssh 对象 s
    hostname = raw_input('hostname: ')
    username = raw_input('username: ')
    password = getpass.getpass('please input password: ') # 接收密码输入
    s.login(hostname, username, password) # 建立 ssh 连接
    s.sendline('uptime') # 运行 uptime 命令
    s.prompt()           # 匹配系统提示符
    print s.before        # 打印出现系统提示符前的命令输出
```

```

s.sendline('ls -l')
s.prompt()
print s.before
s.sendline('df')
s.prompt()
print s.before
s.logout() # 断开 ssh 连接
except pxssh.ExceptionPxssh, e:
    print "pxssh failed on login."
    print str(e)

```

## 5.3 pexpect 应用示例

下面介绍两个通过 pexpect 实现自动化操作的示例，其中一个实现 FTP 协议的自动交互，另一个为 SSH 协议自动化操作，这些都是日常运维中经常遇到的场景。

### 5.3.1 实现一个自动化 FTP 操作

我们常用 FTP 协议实现自动化、集中式的文件备份，要求做到账号登录、文件上传与下载、退出等实现自动化操作，本示例使用 pexpect 模块的 spawnu() 方法执行 FTP 命令，通过 expect() 方法定义匹配的输出版则，sendline() 方法执行相关 FTP 交互命令等，详细源码如下：

【/home/test/pexpect/simple2.py】

```

from __future__ import unicode_literals # 使用 unicode 编码

import pexpect
import sys

child = pexpect.spawnu('ftp ftp.openbsd.org') # 运行 ftp 命令
child.expect('(?(i)name .*: ') # (?(i) 表示后面的字符串正则匹配忽略大小写
child.sendline('anonymous') # 输入 ftp 账号信息
child.expect('(?(i)password') # 匹配密码输入提示
child.sendline('pexpect@sourceforge.net') # 输入 ftp 密码
child.expect('ftp> ')
child.sendline('bin') # 启用二进制传输模式
child.expect('ftp> ')
child.sendline('get robots.txt') # 下载 robots.txt 文件
child.expect('ftp> ')
sys.stdout.write(child.before) # 输出匹配 "ftp> " 之前的输入与输出
print("Escape character is '^]'.\n")
sys.stdout.write(child.after)
sys.stdout.flush()
# 调用 interact() 让出控制权，用户可以继续当前的会话手工控制子程序，默认输入 "^]" 字符跳出
child.interact()

```

```
child.sendline('bye')
child.close()
```

运行结果如下:

```
get robots.txt
local: robots.txt remote: robots.txt
227 Entering Passive Mode (129,128,5,191,197,243)
150 Opening BINARY mode data connection for 'robots.txt' (26 bytes).
226 Transfer complete.
26 bytes received in 3.29 secs (0.01 Kbytes/sec)
Escape character is '^]'.
```

ftp> # 调用 interact() 控制项让出, 用户可以手工进行交互

### 5.3.2 远程文件自动打包并下载

在 Linux 系统集群运营当中, 时常需要批量远程执行 Linux 命令, 并且双向同步文件的操作。本示例通过使用 spawn() 方法执行 ssh、scp 命令的思路来实现, 具体实现源码如下:

**【 /home/test/pexpect/ simple3.py 】**

```
import pexpect
import sys

ip="192.168.1.21" # 定义目标主机
user="root" # 目标主机用户
passwd="H6DSY#*$df32" # 目标主机密码
target_file="/data/logs/nginx_access.log" # 目标主机 nginx 日志文件

child = pexpect.spawn('/usr/bin/ssh', [user+'@'+ip]) # 运行 ssh 命令
fout = file('mylog.txt', 'w') # 输入、输出日志写入 mylog.txt 文件
child.logfile = fout

try:
    child.expect('(?!i)password') # 匹配 password 字符串, (?!i) 表示不区别大小写
    child.sendline(passwd)
    child.expect('#')
    child.sendline('tar -czf /data/nginx_access.tar.gz '+target_file) # 打包 nginx
                                                                    # 日志文件

    child.expect('#')
    print child.before
    child.sendline('exit')
    fout.close()
except EOF: # 定义 EOF 异常处理
    print "expect EOF"
except TIMEOUT: # 定义 TIMEOUT 异常处理
    print "expect TIMEOUT"
```

```
child = pexpect.spawn('/usr/bin/scp', [user+'@'+ip+':/data/nginx_access.tar.gz', '/home']) # 启动 scp 远程拷贝命令, 实现将打包好的 nginx 日志复制至本地 /home 目录
fout = file('mylog.txt', 'a')
child.logfile = fout
try:
    child.expect('(?i)password')
    child.sendline(passwd)
    child.expect(pexpect.EOF) # 匹配缓冲区 EOF (结尾), 保证文件复制正常完成
except EOF:
    print "expect EOF"
except TIMEOUT:
    print "expect TIMEOUT"
```



参考  
提示

5.2 节和 5.3 节常用类说明与应用案例参考 <http://pexpect.readthedocs.org/en/latest/>。

## 系统批量运维管理器 paramiko 详解

paramiko 是基于 Python 实现的 SSH2 远程安全连接，支持认证及密钥方式。可以实现远程命令执行、文件传输、中间 SSH 代理等功能，相对于 Pexpect，封装的层次更高，更贴近 SSH 协议的功能，官网地址：<http://www.paramiko.org>，目前最高版本为 1.13。

### 6.1 paramiko 的安装

paramiko 支持 pip、easy\_install 或源码安装方式，很方便解决包依赖的问题，具体安装命令如下（根据用户环境，自行选择 pip 或 easy\_install）：

```
pip install paramiko
easy_install paramiko
```

paramiko 依赖第三方的 Crypto、Ecdsa 包及 Python 开发包 python-devel 的支持，源码安装步骤如下：

```
# yum -y install python-devel
# wget http://ftp.dlitz.net/pub/dlitz/crypto/pycrypto/pycrypto-2.6.tar.gz
# tar -zxvf pycrypto-2.6.tar.gz
# cd pycrypto-2.6
# python setup.py install
# cd ..
# wget https://pypi.python.org/packages/source/e/ecdsa/ecdsa-0.10.tar.gz --no-check-certificate
# tar -zxvf ecdsa-0.10.tar.gz
```

```
# cd ecdsa-0.10
# python setup.py install
# cd ..
# wget https://github.com/paramiko/paramiko/archive/v1.12.2.tar.gz
# tar -zxvf v1.12.2.tar.gz
# cd paramiko-1.12.2/
# python setup.py install
```

校验安装结果，导入模块没有提示异常则说明安装成功：

```
# python
Python 2.6.6 (r266:84292, Jul 10 2013, 22:48:45)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>>
```

下面介绍一个简单实现远程 SSH 运行命令的示例。该示例使用密码认证方式，通过 `exec_command()` 方法执行命令，详细源码如下：

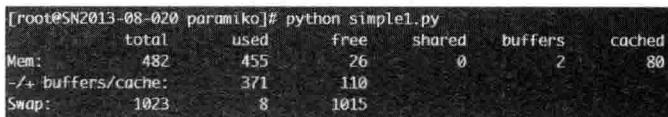
【 /home/test/paramiko/simple1.py 】

```
#!/usr/bin/env python
import paramiko

hostname='192.168.1.21'
username='root'
password='SKJh935yft#'
paramiko.util.log_to_file('syslogin.log') # 发送 paramiko 日志到 syslog.in.log 文件

ssh=paramiko.SSHClient() # 创建一个 ssh 客户端 client 对象
ssh.load_system_host_keys() # 获取客户端 host_keys, 默认 ~/.ssh/known_hosts, 非默认路
                             # 径需指定
ssh.connect(hostname=hostname,username=username,password=password) # 创建 ssh 连接
stdin,stdout,stderr=ssh.exec_command('free -m') # 调用远程执行命令方法 exec_command()
print stdout.read() # 打印命令执行结果, 得到 Python 列表形式, 可以使用 stdout.readlines()
ssh.close() # 关闭 ssh 连接
```

程序的运行结果截图如图 6-1 所示。



```
[root@SN2013-08-020 paramiko]# python simple1.py
total      used      free      shared    buffers     cached
Mem:       482        455         26          0          2         80
-/+ buffers/cache:      371       110
Swap:      1023          8       1015
```

图 6-1 程序运行结果

## 6.2 paramiko 的核心组件

paramiko 包含两个核心组件，一个为 SSHClient 类，另一个为 SFTPClient 类，下面详细介绍。

### 6.2.1 SSHClient 类

SSHClient 类是 SSH 服务会话的高级表示，该类封装了传输（transport）、通道（channel）及 SFTPClient 的校验、建立的方法，通常用于执行远程命令，下面是一个简单的例子：

```
client = SSHClient()
client.load_system_host_keys()
client.connect('ssh.example.com')
stdin, stdout, stderr = client.exec_command('ls -l')
```

下面介绍 SSHClient 常用的几个方法。

#### 1. connect 方法

connect 方法实现了远程 SSH 连接并校验。

方法定义：

```
connect(self, hostname, port=22, username=None, password=None, pkey=None, key_
filename=None, timeout=None, allow_agent=True, look_for_keys=True, compress=False)
```

参数说明：

- ❑ hostname (str 类型)，连接的目标主机地址；
- ❑ port (int 类型)，连接目标主机的端口，默认为 22；
- ❑ username (str 类型)，校验的用户名（默认为当前的本地用户名）；
- ❑ password (str 类型)，密码用于身份校验或解锁私钥；
- ❑ pkey (PKey 类型)，私钥方式用于身份验证；
- ❑ key\_filename(str or list(str) 类型)，一个文件名或文件名的列表，用于私钥的身份验证；
- ❑ timeout (float 类型)，一个可选的超时时间（以秒为单位）的 TCP 连接；
- ❑ allow\_agent (bool 类型)，设置为 False 时用于禁用连接到 SSH 代理；
- ❑ look\_for\_keys (bool 类型)，设置为 False 时用来禁用在本机 ~/.ssh 中搜索私钥文件；
- ❑ compress (bool 类型)，设置为 True 时打开压缩。

#### 2. exec\_command 方法

远程命令执行方法，该命令的输入与输出流为标准输入（stdin）、输出（stdout）、错误

(stderr) 的 Python 文件对象, 方法定义:

```
exec_command(self, command, bufsize=-1)
```

参数说明:

- ❑ `command` (str 类型), 执行的命令串;
- ❑ `bufsize` (int 类型), 文件缓冲区大小, 默认为 -1 (不限制)。

### 3. load\_system\_host\_keys 方法

加载本地公钥校验文件, 默认为 `~/.ssh/known_hosts`, 非默认路径需要手工指定, 方法定义:

```
load_system_host_keys(self, filename=None)
```

参数说明:

`filename` (str 类型), 指定远程主机公钥记录文件。

### 4. set\_missing\_host\_key\_policy 方法

设置连接的远程主机没有本地主机密钥或 `HostKeys` 对象时的策略, 目前支持三种, 分别是 `AutoAddPolicy`、`RejectPolicy` (默认)、`WarningPolicy`, 仅限用于 `SSHClient` 类, 分别代表的含义如下:

- ❑ `AutoAddPolicy`, 自动添加主机名及主机密钥到本地 `HostKeys` 对象, 并将其保存, 不依赖 `load_system_host_keys()` 的配置, 即使 `~/.ssh/known_hosts` 不存在也不产生影响;
- ❑ `RejectPolicy`, 自动拒绝未知的主机名和密钥, 依赖 `load_system_host_keys()` 的配置;
- ❑ `WarningPolicy`, 用于记录一个未知的主机密钥的 Python 警告, 并接受它, 功能上与 `AutoAddPolicy` 相似, 但未知主机会有告警。

使用方法如下:

```
ssh=paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

## 6.2.2 SFTPClient 类

`SFTPClient` 作为一个 SFTP 客户端对象, 根据 SSH 传输协议的 sftp 会话, 实现远程文件操作, 比如文件上传、下载、权限、状态等操作, 下面介绍 `SFTPClient` 类的常用方法。

### 1. from\_transport 方法

创建一个已连通的 SFTP 客户端通道，方法定义：

```
from_transport(cls, t)
```

参数说明：

t (Transport)，一个已通过验证的传输对象。

例子说明：

```
t = paramiko.Transport(("192.168.1.22",22))
t.connect(username="root", password="KJSdj348g")
sftp =paramiko.SFTPClient.from_transport(t)
```

### 2. put 方法

上传本地文件到远程 SFTP 服务端，方法定义：

```
put(self, localpath, remotepath, callback=None, confirm=True)
```

参数说明：

- ❑ localpath (str 类型)，需上传的本地文件（源）；
- ❑ remotepath (str 类型)，远程路径（目标）；
- ❑ callback(function(int, int))，获取已接收的字节数及总传输字节数，以便回调函数调用，默认为 None；
- ❑ confirm (bool 类型)，文件上传完毕后是否调用 stat() 方法，以便确认文件的大小。

例子说明：

```
localpath='/home/access.log'
remotepath='/data/logs/access.log'
sftp.put(localpath,remotepath)
```

### 3. get 方法

从远程 SFTP 服务端下载文件到本地，方法定义：

```
get(self, remotepath, localpath, callback=None)
```

参数说明：

- ❑ remotepath (str 类型)，需下载的远程文件（源）；

- ❑ `localpath` (str 类型), 本地路径 (目标);
- ❑ `callback(function(int, int))`, 获取已接收的字节数及总传输字节数, 以便回调函数调用, 默认为 `None`。

例子说明:

```
remotepath='/data/logs/access.log'
localpath='/home/access.log'
sftp.get(remotepath, localpath)
```

#### 4. 其他方法

SFTPClient 类其他常用方法说明:

- ❑ `Mkdir`, 在 SFTP 服务器端创建目录, 如 `sftp.mkdir("/home/userdir",0755)`。
- ❑ `remove`, 删除 SFTP 服务器端指定目录, 如 `sftp.remove("/home/userdir")`。
- ❑ `rename`, 重命名 SFTP 服务器端文件或目录, 如 `sftp.rename("/home/test.sh", "/home/testfile.sh")`。
- ❑ `stat`, 获取远程 SFTP 服务器端指定文件信息, 如 `sftp.stat("/home/testfile.sh")`。
- ❑ `listdir`, 获取远程 SFTP 服务器端指定目录列表, 以 Python 的列表 (List) 形式返回, 如 `sftp.listdir("/home")`。

#### 5. SFTPClient 类应用示例

下面为 SFTPClient 类的一个完整示例, 实现了文件上传、下载、创建与删除目录等, 需要注意的是, `put` 和 `get` 方法需要指定文件名, 不能省略。详细源码如下:

```
#!/usr/bin/env python
import paramiko

username = "root"
password = "KJsd8t34d"
hostname = "192.168.1.21"
port = 22

try:
    t = paramiko.Transport((hostname, port))
    t.connect(username=username, password=password)
    sftp = paramiko.SFTPClient.from_transport(t)

    sftp.put("/home/user/info.db", "/data/user/info.db") # 上传文件
    sftp.get("/data/user/info_1.db", "/home/user/info_1.db") # 下载文件
    sftp.mkdir("/home/userdir", 0755) # 创建目录
    sftp.rmdir("/home/userdir") # 删除目录
```

```
sftp.rename("/home/test.sh","/home/testfile.sh") # 文件重命名
print sftp.stat("/home/testfile.sh") # 打印文件信息
print sftp.listdir("/home") # 打印目录列表
t.close();
except Exception, e:
    print str(e)
```

## 6.3 paramiko 应用示例

### 6.3.1 实现密钥方式登录远程主机

实现自动密钥登录方式，第一步需要配置与目标设备的密钥认证支持，具体见 9.2.5 节，私钥文件可以存放在默认路径“~/.ssh/id\_rsa”，当然也可以自定义，如本例的“/home/key/id\_rsa”，通过 paramiko.RSAKey.from\_private\_key\_file() 方法引用，详细代码如下：

【 /home/test/paramiko/simple2.py 】

```
#!/usr/bin/env python
import paramiko
import os

hostname='192.168.1.21'
username='root'
paramiko.util.log_to_file('syslogin.log')

ssh=paramiko.SSHClient()
ssh.load_system_host_keys()
privatekey = os.path.expanduser('/home/key/id_rsa') # 定义私钥存放路径
key = paramiko.RSAKey.from_private_key_file(privatekey) # 创建私钥对象 key

ssh.connect(hostname=hostname,username=username,pkey = key)
stdin,stdout,stderr=ssh.exec_command('free -m')
print stdout.read()
ssh.close()
```

程序执行结果见图 6-1。

### 6.3.2 实现堡垒机模式下的远程命令执行

堡垒机环境在一定程度上提升了运营安全级别，但同时也提高了日常运营成本，作为管理的中转设备，任何针对业务服务器的管理请求都会经过此节点，比如 SSH 协议，首先运维人员在办公电脑通过 SSH 协议登录堡垒机，再通过堡垒机 SSH 跳转到所有的业务服务器进行维护操作，如图 6-2 所示。

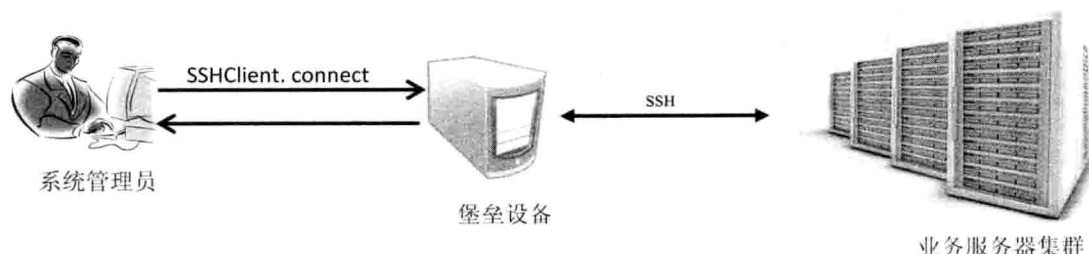


图 6-2 堡垒机模式下的远程命令执行

我们可以利用 paramiko 的 `invoke_shell` 机制来实现通过堡垒机实现服务器操作，原理是 `SSHClient.connect` 到堡垒机后开启一个新的 SSH 会话 (session)，通过新的会话运行 “ssh user@IP” 去实现远程执行命令的操作。实现代码如下：

【 /home/test/paramiko/simple3.py 】

```
#!/usr/bin/env python
import paramiko
import os,sys,time

blip="192.168.1.23"      # 定义堡垒机信息
bluser="root"
blpasswd="KJsdiug45"

hostname="192.168.1.21"  # 定义业务服务器信息
username="root"
password="IS8t5jgrie"

port=22
passinfo='\s password: '    # 输入服务器密码的前标志串
paramiko.util.log_to_file('syslogin.log')

ssh=paramiko.SSHClient()    #ssh 登录堡垒机
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect(hostname=blip,username=bluser,password=blpasswd)

channel=ssh.invoke_shell()  # 创建会话，开启命令调用
channel.settimeout(10)      # 会话命令执行超时时间，单位为秒

buff = ''
resp = ''
channel.send('ssh '+username+'@'+hostname+'\n')    # 执行 ssh 登录业务主机
while not buff.endswith(passinfo):    #ssh 登录的提示信息判断，输出串尾含有 "\s password:" 时
    try:                                # 退出 while 循环
        resp = channel.recv(9999)
    except Exception,e:
        print 'Error info:%s connection time.' % (str(e))
```

```

channel.close()
ssh.close()
sys.exit()
buff += resp
if not buff.find('yes/no')== -1:    # 输出串尾含有 "yes/no" 时发送 "yes" 并回车
    channel.send('yes\n')
    buff=''

channel.send(password+'\n')    # 发送业务主机密码

buff=''
while not buff.endswith('# '):    # 输出串尾为 "# " 时说明校验通过并退出 while 循环
    resp = channel.recv(9999)
    if not resp.find('password:') == -1:    # 输出串尾含有 "\'s password: " 时说明密码不正确,
        # 要求重新输入
        print 'Error info: Authentication failed.'
        channel.close()    # 关闭连接对象后退出
        ssh.close()
        sys.exit()
    buff += resp

channel.send('ifconfig\n')    # 认证通过后发送 ifconfig 命令来查看结果
buff=''
try:
    while buff.find('# ') == -1:
        resp = channel.recv(9999)
        buff += resp
except Exception, e:
    print "error info:" + str(e)

print buff    # 打印输出串
channel.close()
ssh.close()

```

运行结果如下:

```

# python /home/test/paramiko/simple3.py
ifconfig
eth0      Link encap:Ethernet  HWaddr 00:50:56:28:63:2D
          inet addr:192.168.1.21  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::250:56ff:fe28:632d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3523007 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6777657 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:606078157 (578.0 MiB)  TX bytes:1428493484 (1.3 GiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0

```

... ..

显示 “inet addr:192.168.1.21” 说明命令已经成功执行。

### 6.3.3 实现堡垒机模式下的远程文件上传

实现堡垒机模式下的文件上传，原理是通过 paramiko 的 SFTPClient 将文件从办公设备上传至堡垒机指定的临时目录，如 /tmp，再通过 SSHClient 的 invoke\_shell 方法开启 ssh 会话，执行 scp 命令，将 /tmp 下的指定文件复制到目标业务服务器上，如图 6-3 所示。

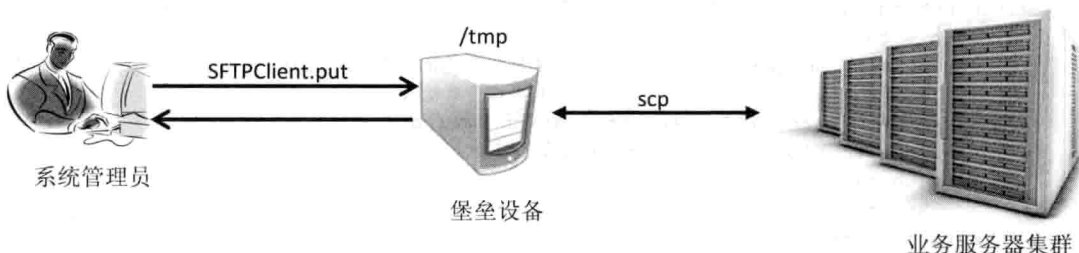


图 6-3 堡垒机模式下的文件上传

本示例具体使用 sftp.put() 方法上传文件至堡垒机临时目录，再通过 send() 方法执行 scp 命令，将堡垒机临时目录下的文件复制到目标主机，详细的实现源码如下：

【 /home/test/paramiko/simple4.py 】

```
#!/usr/bin/env python
import paramiko
import os,sys,time

blip="192.168.1.23"      # 定义堡垒机信息
bluser="root"
blpasswd=" IS8t5jgrie"
hostname="192.168.1.21"  # 定义业务服务器信息
username="root"
password=" KJsdiug45"

tmpdir="/tmp"
remotedir="/data"
localpath="/home/nginx_access.tar.gz"    # 本地源文件路径
tmppath=tmpdir+"/nginx_access.tar.gz"    # 堡垒机临时路径
remotepath=remotedir+"/nginx_access_hd.tar.gz"    # 业务主机目标路径
port=22
passinfo='\s password: '
paramiko.util.log_to_file('syslogin.log')
```

```

t = paramiko.Transport((blip, port))
t.connect(username=bluser, password=blpasswd)
sftp = paramiko.SFTPClient.from_transport(t)
sftp.put(localpath, tmpopath) # 上传本地源文件到堡垒机临时路径
sftp.close()

ssh=paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect(hostname=blip, username=bluser, password=blpasswd)

channel=ssh.invoke_shell()
channel.settimeout(10)

buff = ''
resp = ''
#scp 中转目录文件到目标主机
channel.send('scp '+tmpopath+' '+username+'@'+hostname+':'+remotepath+'\n')
while not buff.endswith(passinfo):
    try:
        resp = channel.recv(9999)
    except Exception,e:
        print 'Error info:%s connection time.' % (str(e))
        channel.close()
        ssh.close()
        sys.exit()
    buff += resp
    if not buff.find('yes/no')== -1:
        channel.send('yes\n')
        buff=''

channel.send(password+'\n')

buff=''
while not buff.endswith('# '):
    resp = channel.recv(9999)
    if not resp.find(passinfo)== -1:
        print 'Error info: Authentication failed.'
        channel.close()
        ssh.close()
        sys.exit()

    buff += resp
print buff
channel.close()
ssh.close()

```

运行结果如下，如目标主机 /data/nginx\_access\_hd.tar.gz 存在，则说明文件已成功上传。

```

# python /home/test/paramiko/simple4.py
nginx_access.tar.gz          100% 1590KB   1.6MB/s   00:00

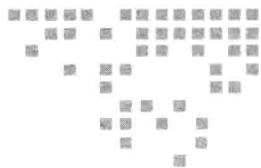
```

当然，整合以上两个示例，再引入主机清单及功能配置文件，可以实现更加灵活、强大的功能，大家可以自己动手，在实践中学习，打造适合自身业务环境的自动化运营平台。



参考  
提示

6.2 节和 6.3 节常用类说明与应用案例参考 <http://docs.paramiko.org/en/1.13/> 官网文档。



## 系统批量运维管理器 Fabric 详解

Fabric 是基于 Python (2.5 及以上版本) 实现的 SSH 命令行工具, 简化了 SSH 的应用程序部署及系统管理任务, 它提供了系统基础的操作组件, 可以实现本地或远程 shell 命令, 包括命令执行、文件上传、下载及完整执行日志输出等功能。Fabric 在 paramiko 的基础上做了更高一层的封装, 操作起来会更加简单。Fabric 官网地址为: <http://www.fabfile.org>, 目前最高版本为 1.8。

### 7.1 Fabric 的安装

Fabric 支持 pip、easy\_install 或源码安装方式, 很方便解决包依赖的问题, 具体安装命令如下 (根据用户环境, 自行选择 pip 或 easy\_install):

```
pip install fabric
easy_install fabric
```

Fabric 依赖第三方的 setuptools、Crypto、paramiko 包的支持, 源码安装步骤如下:

```
# yum -y install python-setuptools
# wget https://pypi.python.org/packages/source/F/Fabric/Fabric-1.8.2.tar.gz
--no-check-certificate
# tar -zxvf Fabric-1.8.2.tar.gz
# cd Fabric-1.8.2
# python setup.py install
```

校验安装结果, 如果导入模块没有提示异常, 则说明安装成功:

```
# python
Python 2.6.6 (r266:84292, Jul 10 2013, 22:48:45)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import fabric
>>>
```

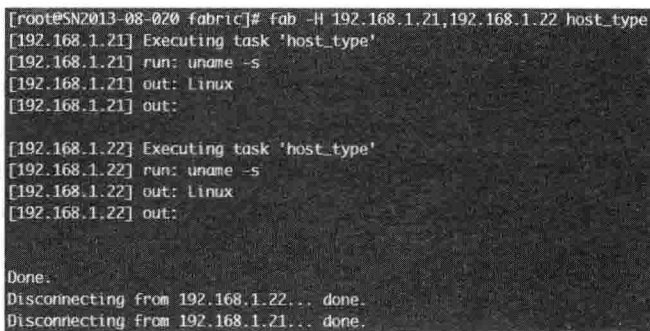
官网提供了一个简单的入门示例：

**[ /home/test/fabric/fabfile.py ]**

```
#!/usr/bin/env python
from fabric.api import run
```

```
def host_type():      # 定义一个任务函数，通过 run 方法实现远程执行 'uname -s' 命令
    run('uname -s')
```

运行结果如图 7-1 所示。



```
[root@SN2013-08-020 fabric]# fab -H 192.168.1.21,192.168.1.22 host_type
[192.168.1.21] Executing task 'host_type'
[192.168.1.21] run: uname -s
[192.168.1.21] out: Linux
[192.168.1.21] out:
[192.168.1.22] Executing task 'host_type'
[192.168.1.22] run: uname -s
[192.168.1.22] out: Linux
[192.168.1.22] out:
Done.
Disconnecting from 192.168.1.22... done.
Disconnecting from 192.168.1.21... done.
```

图 7-1 程序执行结果

其中，fab 命令引用默认文件名为 fabfile.py，如果使用非默认文件名称，则需通过“-f”来指定，如：fab -H SN2013-08-021,SN2013-08-022 -f host\_type.py host\_type。如果管理机与目标主机未配置密钥认证信任，将会提示输入目标主机对应账号登录密码。

## 7.2 fab 的常用参数

fab 作为 Fabric 程序的命令行入口，提供了丰富的参数调用，命令格式如下：

```
fab [options] <command>[:arg1,arg2=val2,host=foo,hosts='h1;h2',...] ...
```

下面列举了常用的几个参数，更多参数可使用 fab -help 查看。

□ -l, 显示定义好的任务函数名；

- ❑ -f, 指定 fab 入口文件, 默认入口文件名为 fabfile.py;
- ❑ -g, 指定网关 (中转) 设备, 比如堡垒机环境, 填写堡垒机 IP 即可;
- ❑ -H, 指定目标主机, 多台主机用 “,” 号分隔;
- ❑ -P, 以异步并行方式运行多主机任务, 默认为串行运行;
- ❑ -R, 指定 role (角色), 以角色名区分不同业务组设备;
- ❑ -t, 设置设备连接超时时间 (秒);
- ❑ -T, 设置远程主机命令执行超时时间 (秒);
- ❑ -w, 当命令执行失败, 发出告警, 而非默认中止任务。

有时候我们甚至不需要写一行 Python 代码也可以完成远程操作, 直接使用命令行的形式, 例如:

```
# fab -p Ksdh3458d(密码) -H 192.168.1.21,192.168.1.22 -- 'uname -s'
```

命令运行结果见图 7-1。

## 7.3 fabfile 的编写

fab 命令是结合我们编写的 fabfile.py (其他文件名须添加 -f filename 引用) 来搭配使用的, 部分命令行参数可以通过相应的方法来代替, 使之更加灵活, 例如 “-H 192.168.1.21,192.168.1.22”, 我们可以通过定义 env.hosts 来实现, 如 “env.hosts = ['192.168.1.21','192.168.1.22']”。fabfile 的主体由多个自定义的任务函数组成, 不同任务函数实现不同的操作逻辑, 下面详细介绍。

### 7.3.1 全局属性设定

env 对象的作用是定义 fabfile 的全局设定, 支持多个属性, 包括目标主机、用户、密码、角色等, 各属性说明如下:

- ❑ env.host, 定义目标主机, 可以用 IP 或主机名表示, 以 Python 的列表形式定义, 如 env.hosts=['192.168.1.21','192.168.1.22']。
- ❑ env.exclude\_hosts, 排除指定主机, 如 env.exclude\_hosts=['192.168.1.22']。
- ❑ env.user, 定义用户名, 如 env.user="root"。
- ❑ env.port, 定义目标主机端口, 默认为 22, 如 env.port="22"。
- ❑ env.password, 定义密码, 如 env.password='KSJ3548t7d'。
- ❑ env.passwords, 与 password 功能一样, 区别在于不同主机不同密码的应用场景, 需要注意的是, 配置 passwords 时需配置用户、主机、端口等信息, 如:

```
env.passwords = {
    'root@192.168.1.21:22': 'SJk348ygd',
    'root@192.168.1.22:22': 'KSh458j4f',
    'root@192.168.1.23:22': 'KSdu43598'
}
```

❑ env.gateway, 定义网关(中转、堡垒机) IP, 如 env.gateway = '192.168.1.23'。

❑ env.deploy\_release\_dir, 自定义全局变量, 格式: env.+ “变量名称”, 如 env.deploy\_release\_dir、env.age、env.sex 等。

❑ env.roledefs, 定义角色分组, 比如 web 组与 db 组主机区分开来, 定义如下:

```
env.roledefs = {
    'webservers': ['192.168.1.21', '192.168.1.22', '192.168.1.23', '192.168.1.24'],
    'dbservers': ['192.168.1.25', '192.168.1.26']
}
```

引用时使用 Python 修饰符的形式进行, 角色修饰符下面的任务函数为其作用域, 下面来看一个示例:

```
@roles('webservers')
def webtask():
    run('/etc/init.d/nginx start')

@roles('dbservers')
def dbtask():
    run('/etc/init.d/mysql start')

@roles('webservers', 'dbservers')
def pubclitask():
    run('uptime')

def deploy():
    execute(webtask)
    execute(dbtask)
    execute(pubclitask)
```

在命令行执行 #fab deploy 就可以实现不同角色执行不同的任务函数了。

### 7.3.2 常用 API

Fabric 提供了一组简单但功能强大的 fabric.api 命令集, 简单地调用这些 API 就能完成大部分应用场景需求。Fabric 支持常用的方法及说明如下:

- ❑ local, 执行本地命令, 如: local('uname -s');
- ❑ lcd, 切换本地目录, 如: lcd('/home');
- ❑ cd, 切换远程目录, 如: cd('/data/logs');
- ❑ run, 执行远程命令, 如: run('free -m');

- sudo, sudo 方式执行远程命令, 如: sudo('/etc/init.d/httpd start');
- put, 上传本地文件到远程主机, 如: put('/home/user.info', '/data/user.info');
- get, 从远程主机下载文件到本地, 如: get('/data/user.info', '/home/root.info');
- prompt, 获得用户输入信息, 如: prompt('please input user password:');
- confirm, 获得提示信息确认, 如: confirm("Tests failed. Continue[Y/N]?");
- reboot, 重启远程主机, 如: reboot();
- @task, 函数修饰符, 标识的函数为 fab 可调用的, 非标记对 fab 不可见, 纯业务逻辑;
- @runs\_once, 函数修饰符, 标识的函数只会执行一次, 不受多台主机影响。

下面结合一些示例来帮助大家理解以上常用的 API。

### 7.3.3 示例 1：查看本地与远程主机信息

本示例调用 local() 方法执行本地（主控端）命令，添加 “@runs\_once” 修饰符保证该任务函数只执行一次。调用 run() 方法执行远程命令。详细源码如下：

【 /home/test/fabric/simple1.py 】

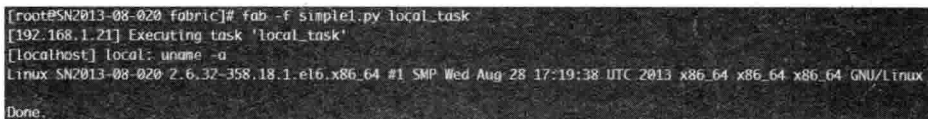
```
#!/usr/bin/env python
from fabric.api import *

env.user='root'
env.hosts=['192.168.1.21','192.168.1.22']
env.password='LKs934j3h3'

@runs_once      # 查看本地系统信息, 当有多台主机时只运行一次
def local_task():    # 本地任务函数
    local("uname -a")

def remote_task():
    with cd("/data/logs"):    # “with” 的作用是让后面的表达式的语句继承当前状态, 实现
        run("ls -l")         # “cd /data/logs && ls -l” 的效果
```

通过 fab 命令分别调用 local\_task 任务函数运行结果如图 7-2 所示。



```
[root@SN2013-08-020 fabric]# fab -f simple1.py local_task
[192.168.1.21] Executing task 'local_task'
[localhost] local: uname -a
Linux SN2013-08-020 2.6.32-358.18.1.el6.x86_64 #1 SMP Wed Aug 28 17:19:38 UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
Done.
```

图 7-2 调用 local\_task 任务函数运行结果

结果中显示了 “[192.168.1.21] Executing task 'local\_task'”，但事实上并非在主机 192.168.1.21 上执行任务，而是返回 Fabric 主机本地 “uname -a” 的执行结果。

调用 remote\_task 任务函数的执行结果如图 7-3 所示。

```

[root@SN2013-08-020 fabric]# fab -f simple1.py remote_task
[192.168.1.21] Executing task 'remote_task'
[192.168.1.21] run: ls -l
[192.168.1.21] out: 总用量 8076
[192.168.1.21] out: -rw-r--r-- 1 root root 8266998 3月  9 11:20 access.tar.gz
[192.168.1.21] out:

[192.168.1.22] Executing task 'remote_task'
[192.168.1.22] run: ls -l
[192.168.1.22] out: total 8076
[192.168.1.22] out: -rw-r--r-- 1 root root 8266998 Mar  9 11:38 access.tar.gz
[192.168.1.22] out:

Done.
Disconnecting from 192.168.1.22... done.
Disconnecting from 192.168.1.21... done.

```

图 7-3 调用 remote\_task 任务函数运行结果

### 7.3.4 示例 2：动态获取远程目录列表

本示例使用 “@task” 修饰符标志入口函数 go() 对外部可见，配合 “@runs\_once” 修饰符接收用户输入，最后调用 worktask() 任务函数实现远程命令执行，详细源码如下：

【 /home/test/fabric/simple2.py 】

```

#!/usr/bin/env python
from fabric.api import *

env.user='root'
env.hosts=['192.168.1.21','192.168.1.22']
env.password='LKs934jh3'

@runs_once      # 主机遍历过程中，只有第一台触发此函数
def input_raw():
    return prompt("please input directory name:",default="/home")

def worktask(dirname):
    run("ls -l "+dirname)

@task          # 限定只有 go 函数对 fab 命令可见
def go():
    getdirname = input_raw()
    worktask(getdirname)

```

该示例实现了一个动态输入远程目录名称，再获取目录列表的功能，由于我们只要求输入一次，再显示所有主机上该目录的列表信息，调用了一个子函数 input\_raw() 同时配置 @runs\_once 修饰符来达到此目的。

执行结果如图 7-4 所示。

```

[root@SN2013-08-020 fabric]# fab -f simple2.py go
[192.168.1.21] Executing task 'go'
please input directory name: [/home] /root
[192.168.1.21] run: ls -l /root
[192.168.1.21] out: 总用量 28
[192.168.1.21] out: drwxr-xr-x. 2 root root 4096 2月 15 21:23 ]
[192.168.1.21] out: -rw-----. 1 root root 964 8月 23 2013 anaconda-ks.cfg
[192.168.1.21] out: -rw-r--r--. 1 root root 13720 8月 23 2013 install.log
[192.168.1.21] out: -rw-r--r--. 1 root root 3857 8月 23 2013 install.log.syslog
[192.168.1.21] out:
[192.168.1.22] Executing task 'go'
[192.168.1.22] run: ls -l /root
[192.168.1.22] out: total 24
[192.168.1.22] out: -rw-----. 1 root root 964 Aug 23 2013 anaconda-ks.cfg
[192.168.1.22] out: -rw-r--r--. 1 root root 13720 Aug 23 2013 install.log
[192.168.1.22] out: -rw-r--r--. 1 root root 3857 Aug 23 2013 install.log.syslog
[192.168.1.22] out: -rw-----. 1 root root 0 Aug 23 2013 yum.log
[192.168.1.22] out:
Done.
Disconnecting from 192.168.1.22... done.
Disconnecting from 192.168.1.21... done.

```

图 7-4 程序运行结果

### 7.3.5 示例 3：网关模式文件上传与执行

本示例通过 Fabric 的 env 对象定义网关模式，即俗称的中转、堡垒机环境。定义格式为“env.gateway='192.168.1.23'”，其中 IP “192.168.1.23” 为堡垒机 IP，再结合任务函数实现目标主机文件上传与执行的操作，详细源码如下：

【/home/test/fabric/simple3.py】

```

#!/usr/bin/env python
from fabric.api import *
from fabric.context_managers import *
from fabric.contrib.console import confirm

env.user='root'
env.gateway='192.168.1.23'    # 定义堡垒机 IP，作为文件上传、执行的中转设备
env.hosts=['192.168.1.21','192.168.1.22']
# 假如所有主机密码都不一样，可以通过 env.passwords 字典变量——指定
env.passwords = {
    'root@192.168.1.21:22': 'LKs934jh3',
    'root@192.168.1.22:22': 'LKs934jh3',
    'root@192.168.1.23:22': 'UI7384hg6'    # 堡垒机账号信息
}

lpackpath="/home/install/lnmp0.9.tar.gz"    # 本地安装包路径
rpackpath="/tmp/install"    # 远程安装包路径

@task
def put_task():
    run("mkdir -p /tmp/install")

```

```

with settings(warn_only=True):
    result = put(lpackpath, rpackpath)    # 上传安装包
    if result.failed and not confirm("put file failed, Continue[Y/N]?"):
        abort("Aborting file put task!")

@task
def run_task():    # 执行远程命令, 安装 lnmp 环境
    with cd("/tmp/install"):
        run("tar -zxvf lnmp0.9.tar.gz")
        with cd("lnmp0.9/"):    # 使用 with 继续继承 /tmp/install 目录位置状态
            run("./centos.sh")

@task
def go():    # 上传、安装组合
    put_task()
    run_task()

```

示例通过简单的配置 `env.gateway='192.168.1.23'` 就可以轻松实现堡垒机环境的文件上传及执行, 相比 `paramiko` 的实现方法简洁了很多, 编写的任务函数完全不用考虑堡垒机环境, 配置 `env.gateway` 即可。

## 7.4 Fabric 应用示例

下面介绍三个比较典型的应用 Fabric 的示例, 涉及文件上传与校验、环境部署、代码发布的功能, 读者可以在此基础进行功能扩展, 写出更加贴近业务场景的工具平台。

### 7.4.1 示例 1: 文件打包、上传与校验

我们时常做一些文件包分发的的工作, 实施步骤一般是先压缩打包, 再批量上传至目标服务器, 最后做一致性校验。本案例通过 `put()` 方法实现文件的上传, 通过对比本地与远程主机文件的 `md5`, 最终实现文件一致性校验。详细源码如下:

【`/home/test/fabric/simple4.py`】

```

#!/usr/bin/env python
from fabric.api import *
from fabric.context_managers import *
from fabric.contrib.console import confirm

env.user='root'
env.hosts=['192.168.1.21','192.168.1.22']
env.password='LKs934jh3'

@task

```

```

@runs_once
def tar_task():    # 本地打包任务函数，只限执行一次
    with lcd("/data/logs"):
        local("tar -czf access.tar.gz access.log")

@task
def put_task():    # 上传文件任务函数
    run("mkdir -p /data/logs")
    with cd("/data/logs"):
        with settings(warn_only=True):    # put (上传) 出现异常时继续执行，非终止
            result = put("/data/logs/access.tar.gz", "/data/logs/access.tar.gz")
            if result.failed and not confirm("put file failed, Continue[Y/N]?"):
                abort("Aborting file put task!")    # 出现异常时，确认用户是否继续，(Y 继续)

@task
def check_task():    # 校验文件任务函数
    with settings(warn_only=True):
        # 本地 local 命令需要配置 capture=True 才能捕获返回值
        lmd5=local("md5sum /data/logs/access.tar.gz",capture=True).split(' ')[0]
        rmd5=run("md5sum /data/logs/access.tar.gz").split(' ')[0]
        if lmd5==rmd5:    # 对比本地及远程文件 md5 信息
            print "OK"
        else:
            print "ERROR"

```

本示例通过定义三个功能任务函数，分别实现文件的打包、上传、校验功能，且三个功能相互独立，可分开运行，如：

```

fab -f simple4.py tar_task    # 文件打包
fab -f simple4.py put_task    # 文件上传
fab -f simple4.py check_task  # 文件校验

```

当然，我们也可以组合在一起运行，再添加一个任务函数 go，代码如下：

```

@task
def go():
    tar_task()
    put_task()
    check_task()

```

运行 `fab -f simple4.py go` 就可以实现文件打包、上传、校验全程自动化。

## 7.4.2 示例 2：部署 LNMP 业务服务环境

业务上线之前最关键的一项任务便是环境部署，往往一个业务涉及多种应用环境，比如 Web、DB、PROXY、CACHE 等，本示例通过 `env.roledefs` 定义不同主机角色，再使用 “`@roles('webservers')`” 修饰符绑定到对应的任务函数，实现不同角色主机的部署差异，详细源

码如下:

**[ /home/test/fabric/simple5.py ]**

```
#!/usr/bin/env python
from fabric.colors import *
from fabric.api import *
env.user='root'
env.roldefs = {      # 定义业务角色分组
    'webservers': ['192.168.1.21', '192.168.1.22'],
    'dbservers': ['192.168.1.23']
}
env.passwords = {
    'root@192.168.1.21:22': 'SJK348ygd',
    'root@192.168.1.22:22': 'KSh458j4f',
    'root@192.168.1.23:22': 'KSdu43598'
}

@roles('webservers')    # webtask 任务函数引用 'webservers' 角色修饰符
def webtask():          # 部署 nginx php php-fpm 等环境
    print yellow("Install nginx php php-fpm...")
    with settings(warn_only=True):
        run("yum -y install nginx")
        run("yum -y install php-fpm php-mysql php-mbstring php-xml php-mcrypt php-gd")
        run("chkconfig --levels 235 php-fpm on")
        run("chkconfig --levels 235 nginx on")

@roles('dbservers')     # dbtask 任务函数引用 'dbservers' 角色修饰符
def dbtask():           # 部署 mysql 环境
    print yellow("Install Mysql...")
    with settings(warn_only=True):
        run("yum -y install mysql mysql-server")
        run("chkconfig --levels 235 mysqld on")

@roles('webservers', 'dbservers') # publictask 任务函数同时引用两个角色修饰符
def publictask():       # 部署公共类环境, 如 epel、ntp 等
    print yellow("Install epel ntp...")
    with settings(warn_only=True):
        run("rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm")
        run("yum -y install ntp")

def deploy():
    execute(publictask)
    execute(webtask)
    execute(dbtask)
```

本示例通过角色来区别不同业务服务环境, 分别部署不同的程序包。我们只需要一个 Python 脚本就可以完成不同业务环境的定制。

### 7.4.3 示例3：生产环境代码包发布管理

程序生产环境的发布是业务上线最后一个环节，要求具备源码打包、发布、切换、回滚、版本管理等功能，本示例实现了这一整套流程功能，其中版本切换与回滚使用了 Linux 下的软链接实现。详细源码如下：

【 /home/test/fabric/simple6.py 】

```
#!/usr/bin/env python
from fabric.api import *
from fabric.colors import *
from fabric.context_managers import *
from fabric.contrib.console import confirm
import time

env.user='root'
env.hosts=['192.168.1.21','192.168.1.22']
env.password='LKs934jh3'

env.project_dev_source = '/data/dev/Lwebadmin/'      # 开发机项目主目录
env.project_tar_source = '/data/dev/releases/'      # 开发机项目压缩包存储目录
env.project_pack_name = 'release'                  # 项目压缩包名前缀，文件名为 release.tar.gz

env.deploy_project_root = '/data/www/Lwebadmin/'    # 项目生产环境主目录
env.deploy_release_dir = 'releases'                # 项目发布目录，位于主目录下
env.deploy_current_dir = 'current'                  # 对外服务的当前版本软链接
env.deploy_version=time.strftime("%Y%m%d")+ "v2"    # 版本号

@runs_once
def input_versionid():      # 获得用户输入的版本号，以便做版本回滚操作
    return prompt("please input project rollback version ID:",default="")

@task
@runs_once
def tar_source():           # 打包本地项目主目录，并将压缩包存储到本地压缩包目录
    print yellow("Creating source package...")
    with lcd(env.project_dev_source):
        local("tar -czf %s.tar.gz ." % (env.project_tar_source + env.project_pack_name))
    print green("Creating source package success!")

@task
def put_package():          # 上传任务函数
    print yellow("Start put package...")
    with settings(warn_only=True):
        with cd(env.deploy_project_root+env.deploy_release_dir):
            run("mkdir %s" % (env.deploy_version))      # 创建版本目录
        env.deploy_full_path=env.deploy_project_root + env.deploy_release_dir +
        "/" +env.deploy_version
```

```

with settings(warn_only=True):    # 上传项目压缩包至此目录
    result = put(env.project_tar_source + env.project_pack_name + ".tar.gz",
env.deploy_full_path)
    if result.failed and no("put file failed, Continue[Y/N]?"):
        abort("Aborting file put task!")

with cd(env.deploy_full_path):    # 成功解压后删除压缩包
    run("tar -zxvf %s.tar.gz" % (env.project_pack_name))
    run("rm -rf %s.tar.gz" % (env.project_pack_name))

print green("Put & untar package success!")

@task
def make_symlink():    # 为当前版本目录做软链接
    print yellow("update current symlink")
    env.deploy_full_path=env.deploy_project_root + env.deploy_release_dir +
"/"+env.deploy_version
    with settings(warn_only=True):    # 删除软链接, 重新创建并指定软链源目录, 新版本生效
        run("rm -rf %s" % (env.deploy_project_root + env.deploy_current_dir))
        run("ln -s %s %s" % (env.deploy_full_path, env.deploy_project_root +
env.deploy_current_dir))
    print green("make symlink success!")

@task
def rollback():    # 版本回滚任务函数
    print yellow("rollback project version")
    versionid= input_versionid()    # 获得用户输入的回滚版本号
    if versionid=='':
        abort("Project version ID error,abort!")

    env.deploy_full_path=env.deploy_project_root + env.deploy_release_dir +
"/"+versionid
    run("rm -f %s" % env.deploy_project_root + env.deploy_current_dir)
    run("ln -s %s %s" % (env.deploy_full_path, env.deploy_project_root + env.
deploy_current_dir))    # 删除软链接, 重新创建并指定软链源目录, 新版本生效
    print green("rollback success!")

@task
def go():    # 自动化程序版本发布入口函数
    tar_source()
    put_package()
    make_symlink()

```

本示例实现了一个通用性很强的代码发布管理功能, 支持快速部署与回滚, 无论发布还是回滚, 都可以通过切换 `current` 的软链来实现, 非常灵活。该功能的流程图如图 7-5 所示。

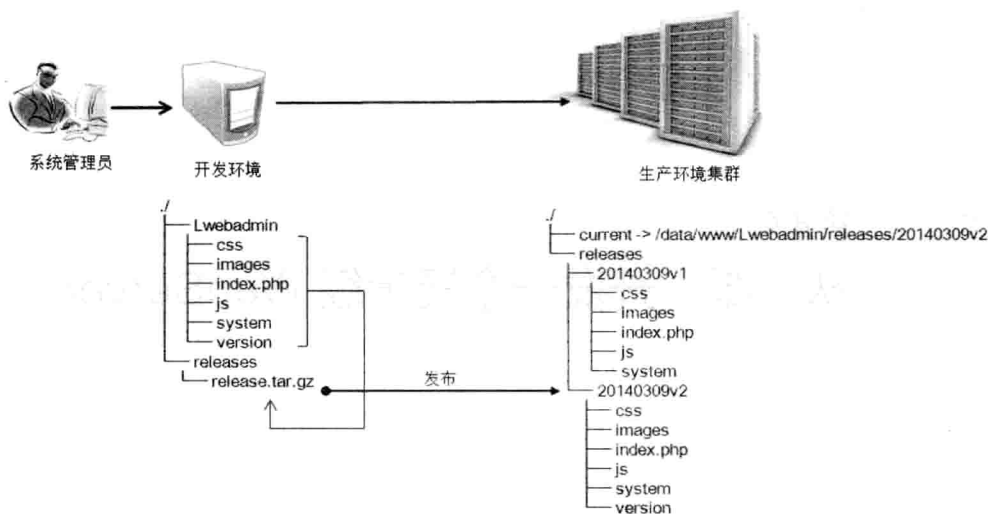


图 7-5 生产环境代码包发布管理流程图

在生产环境中 Nginx 的配置如下：

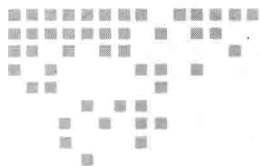
```
server_name domain.com
index index.html index.htm index.php;
root /data/www/Lwebadmin/current;
```

将站点根目录指向“/data/www/Lwebadmin/current”，由于使用 Linux 软链接做切换，管理员的版本发布、回滚操作用户无感知，同时也规范了我们业务上线的流程。



参考  
提示

7.2 节 fab 常用参数说明参考 <http://docs.fabfile.org/en/1.8/> 官网文档。



## 从“零”开发一个轻量级 WebServer

当今互联网行业中，Web 服务几乎覆盖所有业务，包括搜索、电商、社交、视频、游戏等。作为该行业的从业人员，尤其是一名运维人员，深入了解 HTTP 协议的工作原理及机制尤为重要，可以帮助运维人员对 Web 服务优化、运营提供理论指导。比如前端元素结构是否合理，HTTP 缓存配置是否与业务特性相符，HTTP 压缩比应该如何选择等，通过这些优化点可以提高业务服务质量，用户体验也会得到不少提升。本章节介绍作者开发的一轻量级 WebServer——Yorserver，从一个 WebServer 所具备的基本功能出发，详细介绍每个功能点的实现原理与方法。

### 8.1 Yorserver 介绍

#### 8.1.1 功能特点

Yorserver 是基于 Python 实现的轻量级 WebServer，具备一般 WebServer 的基本功能，支持 Linux i386 与 x86 系统。Yorserver 安装、配置都非常简单，其最新版本为 1.0.1，具备以下功能特点：

- ☐ 支持自定义 response 服务及协议版本；
- ☐ 支持 Expires 及 max-age 功能；
- ☐ 支持多进程或线程开启；
- ☐ 支持错误页及默认页配置；

- ❑ 支持 access\_log 及 error\_log 配置;
- ❑ 支持 gzip 压缩配置;
- ❑ 支持安全套连接服务 HTTPS;
- ❑ 支持 HTTP MIME 自定义配置;
- ❑ 支持 PHP、Perl、Python 脚本 cgi 访问;
- ❑ 支持配置文件。

Yorserver 程序目录结构及功能说明如图 8-1 所示,“可更改”表示支持配置文件定义,另外需要确保 cgi-bin 中的 CGI 文件具备可执行权限,具体操作命令: `chmod +x index.pl`。

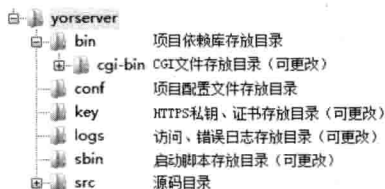


图 8-1 Yorserver 目录结构

运行: `sbin/server.sh start`, 启动 Yorserver 服务。

### 8.1.2 配置文件

Yorserver 采用 ConfigObj 读取配置文件, ConfigObj 是一个简单且功能强大的用于读写配置文件的 Python 应用接口。提供一个简单的编程接口和一个简单的语法配置文件。Yorserver 完整的配置文件内容如下:

`[/usr/local/yorserver/conf/yorserver.conf]`

```

# server_version: Add response HTTP header server version information.
server_version = "YorServer1.0"
# bind_ip: Allows you to bind yorserver to specific IP addresses.
bind_ip="0.0.0.0"
# port: Allows you to bind yorserver's port, http default 80 and Https 443.
port=80
# sys_version: Add response HTTP header python version information.
sys_version = ""
# protocol_version: Add response HTTP header protocol version.
protocol_version = "HTTP/1.0"
# Expires: Add response HTTP header Expires and Max-age version. format:d/h/m).
Expires="7d"

# Multiprocess: configure yorserver Multi process support(on/off).
Multiprocess="off"
# Multithreading: configure yorserver Multi threading support(on/off).

```

```

Multithreading="on"
# DocumentRoot: configure web server document root.
DocumentRoot="/usr/local/yorserver/www"
# page404: configure web server default 404 page.
page404="/404.html"
# Indexes: directory list (on/off).
Indexes="off"
# indexpage: configure web server default index page.
indexpage="/index.html"
# Logfile: configure web server log file path,disable logs Logfile="".
Logfile="/usr/local/yorserver/logs/access.log"
# errorfile: configure web server error file path.
errorfile="/usr/local/yorserver/logs/error.log"
[gzip]
# gzip: Enable(on) or Disable(off) gzip options.
gzip="on"
# configure compress level(1~9)
compresslevel=1
[ssl]
# ssl: Enable(on) or Disable(off) HTTPS options,port options must configure
"443".
ssl="off"
# configure privatekey and certificate pem.
privatekey="/usr/local/yorserver/key/server.key"
certificate="/usr/local/yorserver/key/server.crt"
[cgim]
# cgi_moudle: Enable(on) or Disable(off) cgi support.
cgi_moudle="on"
# cgi_path: configure cgi path,multiple cgi path use ',' delimited,cgi_path in
bin directory.
cgi_path='/cgi-bin',
# cgi_extensions: configure cgi file extension.
cgi_extensions="('.cgi','.py','.pl','.php')"
# contentType: configure file mime support.
[contentType]
css="text/css"
doc="application/msword"
gif="image/gif"
gz="application/x-gzip"
... ..

```

了解 Nginx 或 Apache 配置的人对 Yorserver 的配置并不会陌生，读者可以尝试通过修改不同参数值，来观察 Web 服务器与客户端表现出的差异，客户端可以使用 HttpWatch 工具来跟踪。下面介绍 Yorserver 各个功能点具体的实现原理及方法。

## 8.2 功能实现方法

Python 默认自带的模块已经可以实现简单的 HTTP 服务器，如 BaseHTTPServer 模块提

供基本的 Web 服务和处理器类；SimpleHTTPServer 模块包含 GET 与 HEAD 请求与处理支持；CGIHTTPServer 模块包含处理 POST 请求的支持。Yorserver 是基于 BaseHTTPServer 模块 Web 服务类 HTTPServer 扩展而来，同时也使用 CGIHTTPServer 模块提供 CGI 程序的接收与执行。下面详细介绍各个功能点。

## 8.2.1 HTTP 缓存功能

### (1) Expires 机制

在 HTTP/1.1 协议中，Expires 字段声明了一个网页或 URL 地址不再被浏览器缓存的时间，一旦超过了这个时间，浏览器会重新向原始服务器发起新请求，在 Yorserver 中 Expires 字段的配置如下，指定“Expires=“7d””，表示文件在客户端缓存 7 天。

```
# Expires: Add response HTTP header Expires and Max-age version. format:d/h/
m(day/hour/minute).
Expires="7d"
```

访问 Yorserver 服务下的站点 URL “http://192.168.1.20/index2.html”，通过 HttpWatch 进行跟踪，跟踪结果见图 8-2，可见 Expires 字段显示“Tue, 22 Jul 2014 23:18:49 GMT”，请求原始服务器时间 Date 字段为“Tue, 15 Jul 2014 15:18:49 GMT”，由于 Date 描述的时间为世界标准时间，换算成本地时间需“+8”，即“Tue, 15 Jul 2014 23:18:49”，加上配置的 7 天（7d）过期值，结果等于 Expires 字段值。

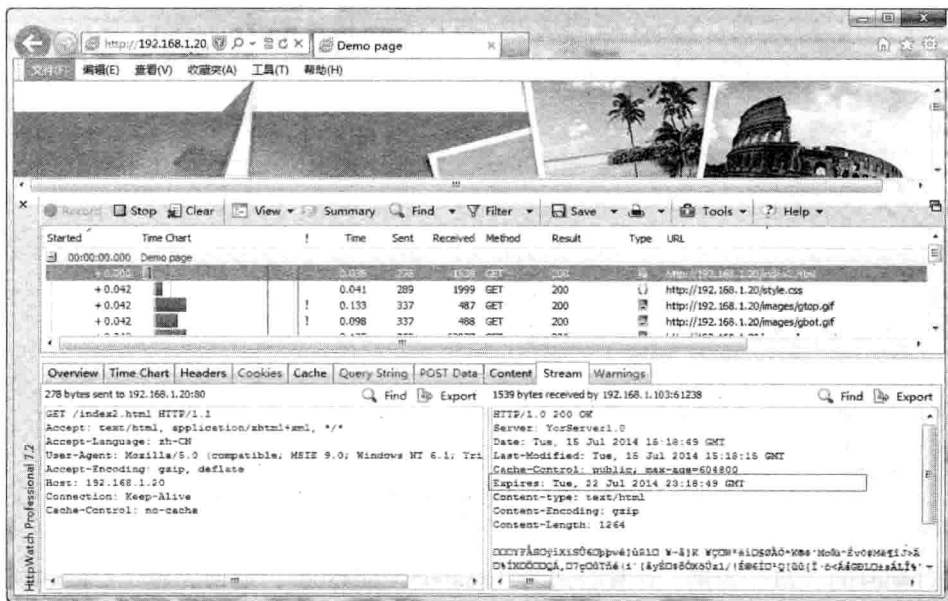


图 8-2 返回的 Expires 字段信息

关于 Yorserver 实现文件过期 Expires 的方法，实现原理为返回“当前时间”+“配置过期时间”，“过期时间”是通过 `datetime.timedelta()` 方法转换不同单位时间后，再与“当前时间”累加，“过期时间”支持通过 days（日）、hours（小时）、minutes（分钟）等单位来表示，以下为 Yorserver 文件过期 Expires 的实现方法：

```
# 文件过期 Expires 实现方法
def get_http_expiry(_Expirestype, _num):
    if _Expirestype=="d":      # 当前时间 + 过期时间（日、小时、分钟）
        expire_date = datetime.datetime.now() + datetime.timedelta(days=_num)
    elif _Expirestype=="h":
        expire_date = datetime.datetime.now() + datetime.timedelta(hours=_num)
    else:
        expire_date = datetime.datetime.now() + datetime.timedelta(minutes=_num)
    return expire_date.strftime('%a, %d %b %Y %H:%M:%S GMT')    # 格式化时间为
                                                                # Expires 格式
```

## （2）max-age 机制

客户端另一缓存机制则是利用 HTTP 消息头中的“cache-control”来控制，其中 max-age 字段实现在原始服务器返回的 max-age 配置的秒数内，浏览器将不会发送相关请求到服务器，而是由缓存直接提供，超过这一时间段后才向原始服务器发起请求，由服务器决定返回新数据还是仍由缓存提供。与 Expires 不同，max-age 是通过指定相对时间秒数来实现缓存过期，当与 Expires 同时存在时，max-age 会覆盖 Expires。下面详细介绍 max-age 的实现原理，由于 max-age 与 Expires 的时间结果是等价的，只是表现形式不同，因此只要得到其中一个值都可以计算出另一个值。Yorserver 是通过已知 Expires 值计算出 max-age，实现源码如下：

```
# 定义过期时间类型，统一成“秒”单位
ExpiresTypes = {
    "d"    : 86400,
    "h"    : 3600,
    "m"    : 60,
}

# 返回 max-age 方法，通过不同时间单位秒 * 数量得到
def secs_from_days(_seconds, _num):
    return _seconds * _num

# 定义“cache_control”返回内容
Expirestype="d"
Expirenum=7
CACHE_MAX_AGE=pubutil.secs_from_days(ExpiresTypes[Expirestype],int(Expirenum))
cache_control = 'public; max-age=%d' % (CACHE_MAX_AGE, )
```

以过期时间“7d”为例，计算公式为“86400\*7=604800”，返回完整“Cache-Control”内容为“Cache-Control: public; max-age=604800”，效果如图 8-3 所示。

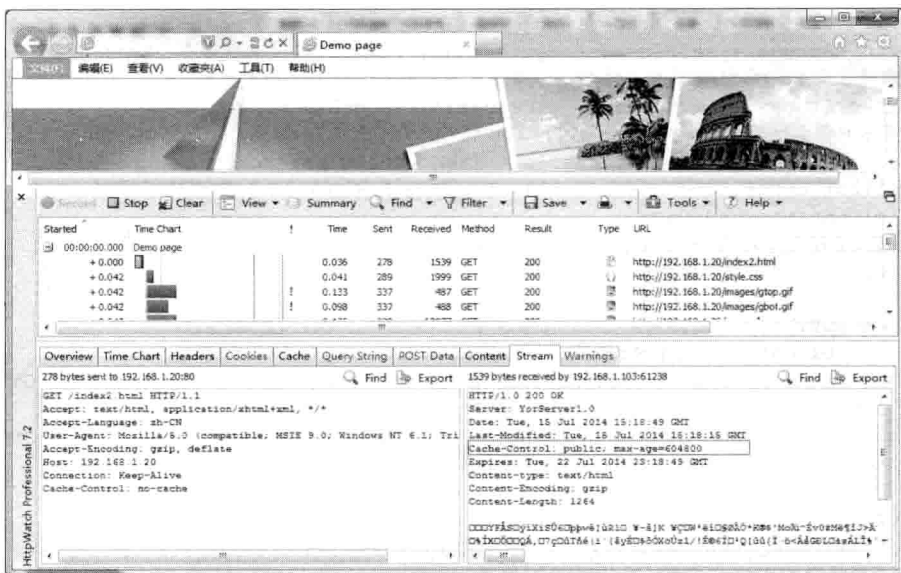


图 8-3 返回 max-age 字段信息

### (3) Last-Modified 机制

最后一种浏览缓存机制为 Last-Modified，其原理是客户端通过 If-Modified-Since 请求头将先前接收到服务器端文件的 Last-Modified 时间戳信息进行发送，目的是让服务器端进行比对验证，通过这个时间戳判断客户端的文件是否是最新，如不是最新的，则返回新的内容（HTTP 200），如果是最新的，则返回 HTTP 304 告诉客户端其本地缓存的文件是最新的，无需重启下载。于是客户端就可以直接从本地加载文件了。具体流程图如图 8-4 所示。

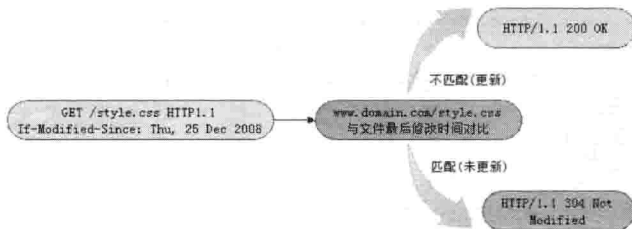


图 8-4 Last-Modified 机制流程图

Yorserver 实现 Last-Modified 缓存机制的原理，首先获取请求头是否包含 Pragma、Cache-Control 字段，检查其值是否为 no-cache，表示客户端要求不缓存，通常是用户主动强制刷新页面，如“Ctrl+F5”组合键，将返回 HTTP 200 状态，否则，将请求头部 If-Modified-Since 字段与服务器端文件 mtime（最后更新时间）进行比较，相匹配则说明文件没有更新，将返回“HTTP/1.0 304 Not Modified”，不匹配则返回“HTTP 200”，实现源码如下：

```

client_cache_cc = self.headers.getheader('Cache-Control') # 获取请求头 Cache-Control 值
client_cache_p = self.headers.getheader('Pragma') # 获取请求头 Pragma 值
# 获取请求头 If-Modified-Since 值, 以便与服务器端文件 mtime 进行比较
Modified_Since = self.headers.getheader('If-Modified-Since')
# 过滤用户强制刷新的场景, 将返回 HTTP 200 状态, 否则获取 If-Modified-Since 值
if client_cache_cc=='no-cache' or client_cache_p=='no-cache' or \
    (client_cache_cc==None and client_cache_p==None and Modified_Since==None):
    client_modified=None
else:
    try: # 兼容不同浏览器请求异常
        client_modified = Modified_Since.split(';')[0]
    except:
        client_modified=None
# 将文件 mtime 时间格式转为 Last-Modified 格式, 如 "Mon, 29 Dec 2008 16:51:22 GMT"
file_last_modified=self.date_time_string(fs.st_mtime)
if client_modified==file_last_modified: # 比较 If-Modified-Since 与文件 mtime 值
    self.send_response(304) # 匹配则返回 304 状态
    self.end_headers()
else:
    self.send_response(200) # 不匹配则返回 200 状态
    # 将文件 mtime 作为 Last-Modified 返回
    self.send_header('Last-Modified', file_last_modified)
    self.send_header('Cache-Control', cache_control)
    self.send_header('Expires', expiration)
    self.send_header('Content-type',content_type)

```

客户端请求及响应效果如图 8-5 所示, 当文件没有发生更新时返回 “HTTP/1.0 304 Not Modified” 状态, 当手工修改文件, 使文件 mtime 发生改变时, 将返回 “HTTP 200” 状态。

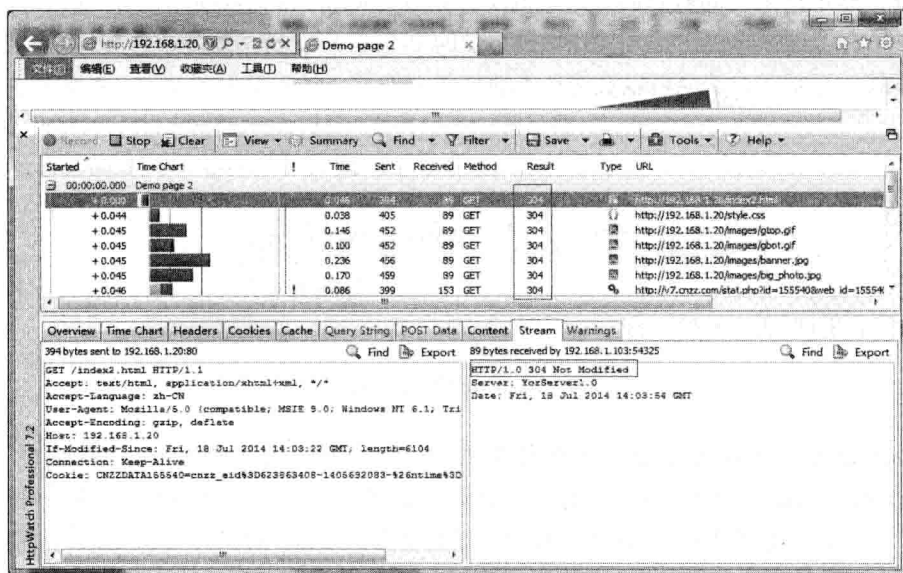


图 8-5 返回 304 状态信息

## 8.2.2 HTTP 压缩功能

启用 HTTP 内容压缩, 可为我们节省不少带宽成本, 并且也可以加快网页访问速度, 提升用户体验。目前主流的浏览器都支持客户端解压功能, Yorserver 服务器端采用 gzip 压缩机制, 其原理是在文件传输之前, 先使用 gzip 压缩后再传输给客户端, 客户端接收之后再由浏览器解压显示, 这样虽然稍微占用了一些服务器和客户端的 CPU 资源, 但是换来的是更高的带宽利用率。对于纯文本 (html、css、js 等) 来讲, 效果非常显著。Yorserver 压缩配置选项如下, 其中 compresslevel 为压缩比, 其值为 1~9, “1” 压缩比最小处理速度最快, “9” 压缩比最大但处理速度最慢, 损耗 CPU 资源。

```
[gzip]
# gzip: Enable(on) or Disable(off) gzip options.
gzip="on"
# configure compress level(1~9)
compresslevel=9
```

关于实现 HTTP 内容压缩的方法, 需要加载 gzip、cStringIO 两个模块, gzip 实现内容的压缩功能, cStringIO 的作用是操作内存文件, 读取磁盘文件内容写入内存文件, 再做压缩处理, 最后输出压缩后的内容返回给客户端, 详细源码如下:

```
#HTTP 内容压缩方法, 参数 buf 为文件内容, _compresslevel 为压缩比
def compressBuf(buf, _compresslevel):
    import gzip, cStringIO
    zbuf = cStringIO.StringIO()    # 创建一个内存流文件对象

    # 创建一个 gzip 文件对象
    zfile = gzip.GzipFile(mode = 'wb', fileobj = zbuf, compresslevel = _compresslevel)
    zfile.write(buf)    # 写入文件压缩内容
    zfile.close()
    return zbuf.getvalue()    # 返回压缩内容

f = open(DocumentRoot + sep + self.path)
if gzip=="on":    # 开启 gzip 选项则调用压缩方法 compressBuf(), 否则直接读取文件内容
    compressed_content =compressBuf(f.read(),compresslevel)
else:
    compressed_content = f.read()
```

HTTP 内容压缩效果如图 8-6 所示, index2.html 文件原始大小为 6104 字节, gzip 压缩后为 1158 个字节, 压缩了 81% 的内容, 效果很理想。

## 8.2.3 HTTP SSL 功能

HTTPS (Hyper Text Transfer Protocol over Secure Socket Layer) 是以安全为目标的 HTTP 通道, 可以理解成 HTTP 的安全版, 即 HTTP 协议下加入 SSL 层, HTTPS 的安全基础是

SSL, 因此加密的详细内容就需要 SSL (Secure Sockets Layer, 安全套接层)。目前 HTTPS 广泛用于互联网上安全敏感的通信, 例如电商在线交易支付方面。

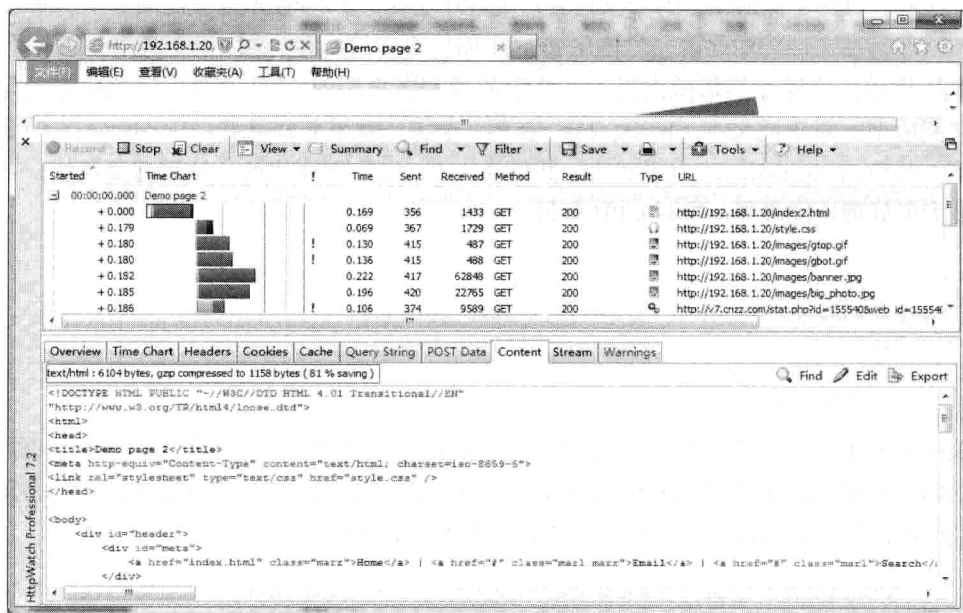


图 8-6 HTTP 压缩效果图

关于 Yorserver 配置 SSL 的选项, 需要修改监听端口为 443, 在启用 SSL 同时需要指定私钥 privatekey 及证书 certificate 两个选项, 具体配置如下:

```

# port: Allows you to bind yorserver's port, http default 80 and Https 443.
port=443

[ssl]
# ssl: Enable(on) or Disable(off) HTTPS options, port options must configure "443".
ssl="on"
# configure privatekey and certificate pem.
privatekey="/usr/local/yorserver/key/app.key"
certificate="/usr/local/yorserver/key/server.crt"

```

具体的功能实现使用了 OpenSSL、SocketServer 两个模块, 其中 OpenSSL 负责 SSL 的功能, SocketServer 负责基础通信。详细源码如下:

```

class SecureHTTPServer(HTTPServer):
    def __init__(self, server_address, HandlerClass):
        BaseServer.__init__(self, server_address, HandlerClass)
        ctx = SSL.Context(SSL.SSLv23_METHOD) # 定义一个 SSL 连接

```

```

ctx.use_privatekey_file(privatekey)    # 指定私钥文件
ctx.use_certificate_file(certificate)  # 指定证书文件
self.socket = SSL.Connection(ctx, socket.socket(self.address_family,\
self.socket_type))    # 创建一个连接对象, 参数使用给定的 OpenSSL.SSL.Context 实例和 Socket
self.server_bind()     # 服务绑定并激活
self.server_activate()

```

生成密钥与证书可以参考以下步骤:

```

# 生成 RSA 密钥 server.key
# openssl genrsa -des3 -out server.key 1024

# 复制一个密钥文件 app.key (无需输入密码)
# openssl rsa -in server.key -out app.key

# 生成一个证书请求 server.csr
# openssl req -new -key server.key -out server.csr

# 签发证书 server.crt
# openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt

```

下一步将生成的密钥文件 app.key、证书文件 server.crt 复制到 yorserver.conf 配置指定路径即可, 如 /usr/local/yorserver/key/app.key 与 /usr/local/yorserver/key/server.crt, 最后重启 Yorserver 服务, 效果如图 8-7 所示。



图 8-7 SSL 证书信息

## 8.2.4 目录列表功能

Web 目录列表很直观地展示了站点目录的结构，普遍应用在文档及下载服务中，当然，对安全级别要求较高的站点，建议还是关闭此功能。Yorserver 支持目录列表功能，在配置中开启 / 关闭的方法如下：

```
# Indexes: directory list (on/off).
Indexes="on"
```

实现的方法是通过 `os.listdir()` 方法获取站点目录（系统绝对路径）列表，通过前端“<li>”、“<a>” HTML 标签格式化输出，具体实现源码如下：

```
def list_directory(self, path):
    try:
        list = os.listdir(path)      # 获取当前目录系统绝对路径列表
    except os.error:
        self.send_error(404, "No permission to list directory");
        return None
    list.sort(lambda a, b: cmp(a.lower(), b.lower()))    # 不区分大小写对目录列表做排序
    f = StringIO()    # 创建内存文件对象
    f.write("<h2>Directory listing for %s</h2>\n" % self.path) #self.path为当前URL路径
    f.write("<hr>\n<ul>\n")
    # 输出上一级目录URL链接
    f.write('<li><a href="%s">Parent Directory</a>\n' % (pubutil.parent_dir(self.path)))
    for name in list:    # 遍历输出目录文件列表
        fullname = os.path.join(path, name)
        displayname = name = cgi.escape(name)    #HTML 字符转义
        if os.path.islink(fullname):
            displayname = name + "@"
        elif os.path.isdir(fullname):
            displayname = name + "/"
            name = name + os.sep
        f.write('<li><a href="%s">%s</a>\n' % (name, displayname))
    f.write("</ul>\n<hr>\n")
    f.seek(0)
    return f
```

目录列表效果如图 8-8 所示。

## 8.2.5 动态 CGI 功能

CGI (Common Gateway Interface, 通用网关接口) 实现让一个客户端从网页浏览器向在网络服务器上的程序请求数据。CGI 描述了客户端和服务器程序之间传输数据的一种标准。编写 CGI 程序的语言有 Shell、Perl、Python、Ruby、PHP、TCL、C/C++ 等。Yorserver 支持这些 CGI 程序的调用，需要修改相关配置，`cgi_path` 参数指定 CGI 程序的存放目录，默认为

yorserver/bin/cgi-bin 目录, 指定多个目录使用 “,” 号分隔; cgi\_extensions 参数指定 CGI 程序扩展名支持, 详细见下面的配置:

```
[cgim]
# cgi_moudle: Enable(on) or Disable(off) cgi support.
cgi_moudle="on"

# cgi_path: configure cgi path,multiple cgi path use ',' delimited,cgi_path in
bin directory.
cgi_path='/cgi-bin',

# cgi_extensions: configure cgi file extension.
cgi_extensions="('.cgi','.py','.pl','.php')"
```

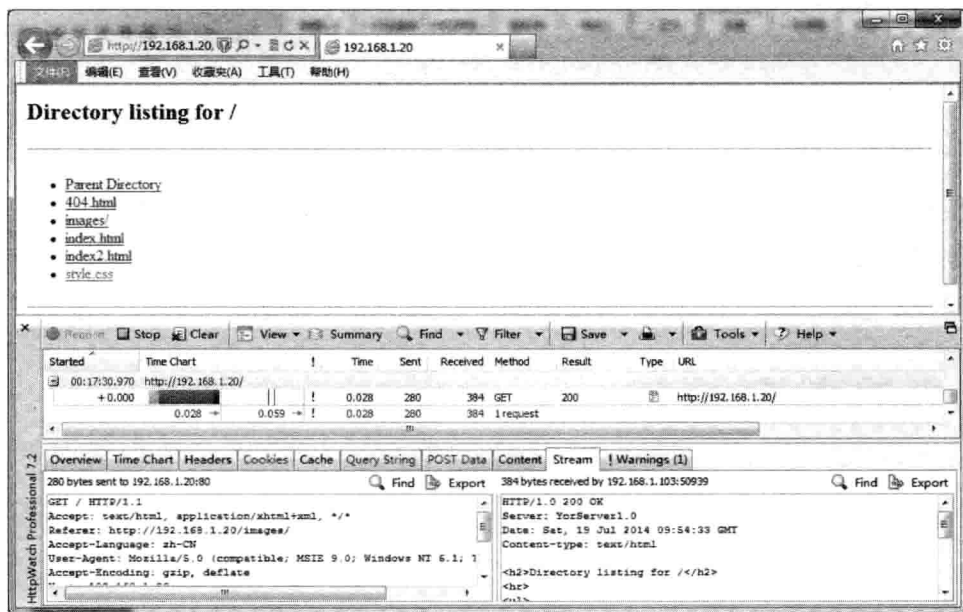


图 8-8 目录列表

Yorserver 采用 CGIHTTPServer 模块来实现 CGI 支持, 其 CGIHTTPRequestHandler 类继承了 SimpleHTTPRequestHandler 类, 因此, 该类除了可以执行 CGI 程序外还支持静态文件服务。另外在主服务类中需要继承 CGIHTTPRequestHandler 基类, 例如: class ServerHandler (CGIHTTPRequestHandler), 其他实现源码如下:

```
CGIHTTPRequestHandler.cgi_directories = cgi_path      # 指定 CGI 路径
if cgi_moudle=="on" and self.path.endswith(cgi_extensions):
    # 开启 CGI 且在配置
    # 扩展名列表中
    return CGIHTTPRequestHandler.do_GET(self)         # 调用 cgi do_GET() 方法, 返回执行结果
```

下面列举 Python 与 PHP CGI 实现冒泡排序法的示例。代码如下：

【 bin/cgi-bin/index.py 】

```
#!/usr/bin/env python
#coding=utf-8
print "Content-type: text/html\n\n";
print "<html><head><title>Python 冒泡排序测试</title></head><body>"
my_list = [23,45,67,3,56,82,24,23,5,77,19,33,51,99]

def bubble(bad_list):
    length = len(bad_list) - 1
    sorted = False
    while not sorted:
        sorted = True
        for i in range(length):
            if bad_list[i] > bad_list[i+1]:
                sorted = False
                bad_list[i], bad_list[i+1] = bad_list[i+1], bad_list[i]
bubble(my_list)
print my_list
print "</body></html>"
```

执行结果如图 8-9 所示。



图 8-9 Python CGI 运行结果图

【 bin/cgi-bin/index.php 】

```
#!/usr/bin/env php
<?php
echo "Content-type: text/html\n\n";
echo "<html><head><title>PHP 冒泡排序测试</title></head><body><pre>";
function bubble(array $array){
    for($i=0, $len=count($array)-1; $i<$len; ++$i){
        for($j=$len; $j>$i; --$j){
            if($array[$j] < $array[$j-1])
            {
                $temp = $array[$j];
                $array[$j] = $array[$j-1];
                $array[$j-1] = $temp;
            }
        }
    }
    return $array;
}
print_r(bubble(array(23,45,67,3,56,82,24,23,5,77,19,33,51,99)));
echo "</pre></body></html>";
?>
```

执行结果如图 8-10 所示。

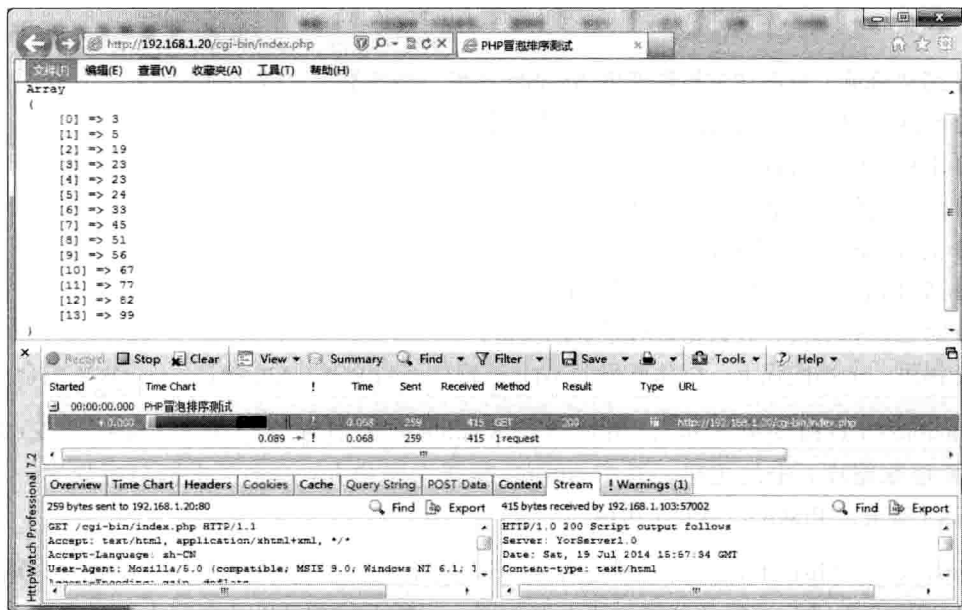


图 8-10 PHP CGI 运行结果图

## 集中化管理平台 Ansible 详解

Ansible (<http://www.ansibleworks.com/>) 一种集成 IT 系统的配置管理、应用部署、执行特定任务的开源平台，是 AnsibleWorks 公司名下的项目，该公司由 Cobbler 及 Func 的作者于 2012 年创建成立。Ansible 基于 Python 语言实现，由 Paramiko 和 PyYAML 两个关键模块构建。Ansible 具有如下特点：

- 部署简单，只需在主控端部署 Ansible 环境，被控端无需做任何操作；
- 默认使用 SSH (Secure SHell) 协议对设备进行管理；
- 主从集中化管理；
- 配置简单、功能强大、扩展性强；
- 支持 API 及自定义模块，可通过 Python 轻松扩展；
- 通过 Playbooks 来定制强大的配置、状态管理；
- 对云计算平台、大数据都有很好的支持；
- 提供一个功能强大、操作性强的 Web 管理界面和 REST API 接口——AWX 平台。

Ansible 的架构图见图 9-1，用户通过 Ansible 编排引擎操作公共 / 私有云或 CMDB (配置管理数据库) 中的主机，其中 Ansible 编排引擎由 Inventory (主机与组规则)、API、Modules (模块)、Plugins (插件) 组成。

Ansible 与 Saltstack 最大的区别是 Ansible 无需在被控主机部署任何客户端代理，默认直接通过 SSH 通道进行远程命令执行或下发配置；相同点是都具备功能强大、灵活的系统管理、状态配置，都使用 YAML 格式来描述配置，两者都提供丰富的模板及 API，对云计算平台、大数据都有很好的支持。Ansible 在 GitHub 上的地址为 <https://github.com/ansible/>，其中

提供了不少配置例子供参考，本文测试的版本为 1.3.2。

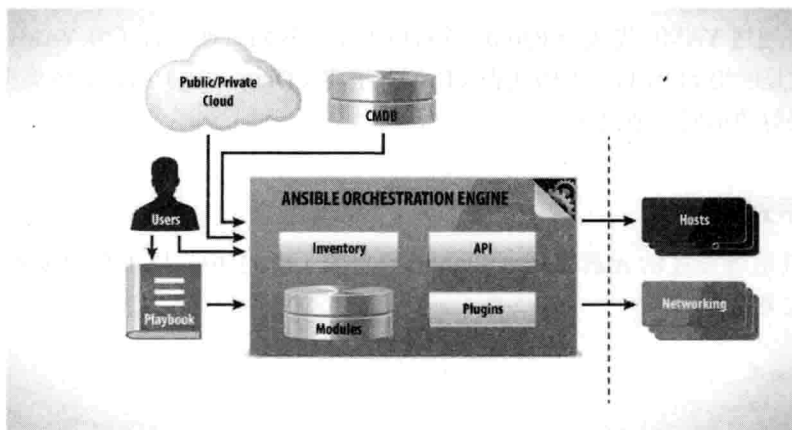


图 9-1 Ansible 架构图



Ansible 提供了一个在线 Playbook 分享平台，地址：<https://galaxy.ansibleworks.com>，该平台汇聚了各类常用功能的角色，找到适合自己的 Role（角色）后，只需要运行“ansible-galaxy install 作者 id. 角色包名称”就可以安装到本地，比如想安装 bennojoy 提供的 Nginx 安装与配置的角色，直接运行“ansible-galaxy install bennojoy.nginx”即可安装到本地，该角色的详细地址为：<https://galaxy.ansibleworks.com/list#/roles/2>。

为了方便读者更系统化地了解 Ansible 的技术点，本章将针对相关技术点进行详细展开介绍。

## 9.1 YAML 语言

YAML 是一种用来表达数据序列的编程语言，它的主要特点包括：可读性强、语法简单明了、支持丰富的语言解析库、通用性强等。Ansible 与 Saltstack 环境中配置文件都以 YAML 格式存在，熟悉 YAML 结构及语法对我们理解两环境的相关配置至关重要。下面的示例定义了 master 的不同业务环境下文件根路径的描述：

```
file_roots:
  base:
    - /srv/salt/
  dev:
    - /srv/salt/dev
```

```
prod:
  - /srv/salt/prod
```

本节主要通过 YAML 描述与 Python 的对应关系，从而方便读者了解 YAML 的层次及结构，最常见的是映射到 Python 中的列表（List）、字典（Dictionary）两种对象类型。下面通过块序列与块映射的示例详细说明。

### 9.1.1 块序列描述

块序列就是将描述的元素序列到 Python 的列表（List）中。以下代码演示了 YAML 与 Python 的对应关系：

```
import yaml
obj=yaml.load(
"""
- Hesperidae
- Papilionidae
- Apatelodidae
- Epiplemididae
""")
print obj
```

本例中引用“-”来分隔列表中的每个元素，运行结果如下：

```
['Hesperidae', 'Papilionidae', 'Apatelodidae', 'Epiplemididae']
```

YAML 也存在类似于 Python 块的概念，例如：

```
-
- Hesperidae
- Papilionidae
- Apatelodidae
- Epiplemididae
-
- China
- USA
- Japan
```

对应的 Python 结果为：

```
[['Hesperidae', 'Papilionidae', 'Apatelodidae', 'Epiplemididae'], ['China', 'USA', 'Japan']]
```

### 9.1.2 块映射描述

块映射就是将描述的元素序列到 Python 的字典（Dictionary）中，格式为“键（key）：值

(value)”，以下为 YAML 例子：

```
hero:
  hp: 34
  sp: 8
  level: 4
orc:
  hp: 12
  sp: 0
  level: 2
```

对应的 Python 结果为：

```
{'hero': {'hp': 34, 'sp': 8, 'level': 4}, 'orc': {'hp': 12, 'sp': 0, 'level': 2}}
```

当然，YAML 块序列与块映射是可以自由组合在一起的，它们之间可以相互嵌套，通过非常灵活的组合，可以帮助我们描述更加复杂的对象属性，例如：

```
- hero:
  hp: 34
  sp: 8
  level: 4
- orc:
  hp:
    - 12
    - 30
  sp: 0
  level: 2
```

对应的 Python 结果为：

```
[{'hero': {'hp': 34, 'sp': 8, 'level': 4}}, {'orc': {'hp': [12, 30], 'sp': 0, 'level': 2}}]
```

## 9.2 Ansible 的安装

Ansible 只需在管理端部署环境即可，建议读者采用 yum 源方式来实现部署，下面介绍具体步骤。

### 9.2.1 业务环境说明

为了方便读者理解，笔者通过虚拟化环境部署了两组业务功能服务器来进行演示。笔者的操作系统版本为 CentOS release 6.4，自带 Python 2.6.6。相关服务器信息如表 9-1 所示（CPU 核数及 Nginx 根目录的差异化是为方便演示生成动态配置需要）：

表 9-1 业务环境表

角色	主机名	IP	组名	Cpus (核数)	Web Root (Nginx 根目录)
Master	SN2013-08-020	192.168.1.20	—	—	—
minion	SN2013-08-021	192.168.1.21	webservers	2	/data
minion	SN2013-08-022	192.168.1.22	webservers	2	/data

### 9.2.2 安装 EPEL

由于目前 RHEL 官网的 yum 源还没有得到 Ansible 的安装包支持, 因此先安装 EPEL 作为部署 Ansible 的默认 yum 源。

❑ RHEL(CentOS)5 版本: `rpm -Uvh http://mirror.pnl.gov/epel/5/i386/epel-release-5-4.noarch.rpm`

❑ RHEL(CentOS) 6 版本: `rpm -Uvh http://ftp.linux.ncsu.edu/pub/epel/6/i386/epel-release-6-8.noarch.rpm`

### 9.2.3 安装 Ansible

主服务器安装 (主控端), 代码如下:

```
#yum install ansible -y
```

### 9.2.4 Ansible 配置及测试

第一步是修改主机与组配置, 文件位置 `/etc/ansible/hosts`, 格式为 ini, 添加两台主机 IP, 同时定义两个 IP 到 `webservers` 组, 更新的内容如下:

【 `/etc/ansible/hosts` 】

```
#green.example.com
#blue.example.com
192.168.1.21
192.168.1.22

[webservers]
#alpha.example.org
#beta.example.org
192.168.1.21
192.168.1.22
```

通过 `ping` 模块测试主机的连通性, 分别对单主机及组进行 `ping` 操作, 出现如图 9-2 所示的结果表示安装、测试成功。

```
[root@SN2013-08-020 ~]# ansible 192.168.1.21 -m ping -k
SSH password:
192.168.1.21 | success => {
  "changed": false,
  "ping": "pong"
}

[root@SN2013-08-020 ~]# ansible webservers -m ping -k
SSH password:
192.168.1.21 | success => {
  "changed": false,
  "ping": "pong"
}

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

图 9-2 测试主机连通性



**提示** 由于主控端与被控主机未配置 SSH 证书信任，需要在执行 `ansible` 命令时添加 `-k` 参数，要求提供 `root`（默认）账号密码，即在提示“SSH password:”时输入。很多人更倾向于使用 Linux 普通用户账户进行连接并使用 `sudo` 命令实现 `root` 权限，格式为：`ansible webservers -m ping -u ansible -sudo`。

## 9.2.5 配置 Linux 主机 SSH 无密码访问

为了避免 Ansible 下发指令时输入目标主机密码，通过证书签名达到 SSH 无密码是一个好的方案，推荐使用 `ssh-keygen` 与 `ssh-copy-id` 来实现快速证书的生成及公钥下发，其中 `ssh-keygen` 生成一对密钥，使用 `ssh-copy-id` 来下发生成的公钥。具体操作如下。

在主机端主机（SN2013-08-020）创建密钥，执行：`ssh-keygen -t rsa`，有询问直接按回车键即可，将在 `/root/.ssh/` 下生成一对密钥，其中 `id_rsa` 为私钥，`id_rsa.pub` 为公钥（需要下发到被控主机用户 `.ssh` 目录，同时要求重命名成 `authorized_keys` 文件）。

```
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa): (回车)
Enter passphrase (empty for no passphrase): (回车)
Enter same passphrase again: (回车)
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
8d:f0:47:c6:b9:55:5b:c0:0e:04:ec:e2:9c:38:f6:84 root@SN2013-08-020
The key's randomart image is:
+--[ RSA 2048 ]-----+
|          ..O..O..|
```

```

|          .....o |
|          .  .  =  .o. |
|          o.=.o  .  |
|          =So+      |
|          E =.      |
|          .  +      |
|          .          |
|          +-----+

```

接下来同步公钥文件 `id_rsa.pub` 到目标主机，推荐使用 `ssh-copy-id` 公钥拷贝工具，命令格式：`/usr/bin/ssh-copy-id [-i [identity_file]] [user@]machine`。本示例中我们输入以下命令同步公钥至 `192.168.1.21` 和 `192.168.1.22` 主机。

```
#ssh-copy-id -i /root/.ssh/id_rsa.pub root@192.168.1.21
#ssh-copy-id -i /root/.ssh/id_rsa.pub root@192.168.1.22
```

校验 SSH 无密码配置是否成功，运行 `ssh root@192.168.1.21`，如直接进入目标 `root` 账号提示符，则说明配置成功。

## 9.3 定义主机与组规则

Ansible 通过定义好的主机与组规则（Inventory）对匹配的目标主机进行远程操作，配置规则文件默认是 `/etc/ansible/hosts`。

### 9.3.1 定义主机与组

所有定义的主机与组规则都在 `/etc/Ansible/hosts` 文件中，为 `ini` 文件格式，主机可以用域名、IP、别名进行标识，其中 `webservers`、`dbservers` 为组名，紧跟着的主机为其成员。格式如下：

```
mail.example.com
192.168.1.21:2135

[webservers]
foo.example.com
bar.example.com
192.168.1.22

[dbservers]
one.example.com
two.example.com
three.example.com
192.168.1.23
```

其中, 192.168.1.21:2135 的意思是定义一个 SSH 服务端口为 2135 的主机, 当然我们也可以使用别名来描述一台主机, 如:

```
jumper ansible_ssh_port=22 ansible_ssh_host=192.168.1.50
```

jumper 为定义的一个别名, ansible\_ssh\_port 为主机 SSH 服务端口, ansible\_ssh\_host 为目标主机, 更多保留主机变量如下:

- ❑ ansible\_ssh\_host, 连接目标主机的地址。
- ❑ ansible\_ssh\_port, 连接目标主机 SSH 端口, 端口 22 无需指定。
- ❑ ansible\_ssh\_user, 连接目标主机默认用户。
- ❑ ansible\_ssh\_pass, 连接目标主机默认用户密码。
- ❑ ansible\_connection, 目标主机连接类型, 可以是 local、ssh 或 paramiko。
- ❑ ansible\_ssh\_private\_key\_file 连接目标主机的 ssh 私钥。
- ❑ ansible\_\*\_interpreter, 指定采用非 Python 的其他脚本语言, 如 Ruby、Perl 或其他类似 ansible\_python\_interpreter 解释器。

组成员主机名称支持正则描述, 示例如下:

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

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

### 9.3.2 定义主机变量

主机可以指定变量, 以便后面供 Playbooks 配置使用, 比如定义主机 hosts1 及 hosts2 上 Apache 参数 http\_port 及 maxRequestsPerChild, 目的是让两台主机产生 Apache 配置文件 httpd.conf 差异化, 定义格式如下:

```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

### 9.3.3 定义组变量

组变量的作用域是覆盖组所有成员, 通过定义一个新块, 块名由组名 + “:vars” 组成, 定义格式如下:

```
[atlanta]
host1
```

```

host2

[atlanta:vars]
ntp_server=ntp.atlanta.example.com
proxy=proxy.atlanta.example.com

```

同时 Ansible 支持组嵌套组，通过定义一个新块，块名由组名 + “:children” 组成，格式如下：

```

[atlanta]
host1
host2

[raleigh]
host2
host3

[southeast:children]
atlanta
raleigh

[southeast:vars]
some_server=foo.southeast.example.com
halon_system_timeout=30
self_destruct_countdown=60
escape_pods=2

[usa:children]
southeast
northeast
southwest
southeast

```



**提示** 嵌套组只能使用在 `/usr/bin/ansible-playbook` 中，在 `/usr/bin/ansible` 中不起作用。

### 9.3.4 分离主机与组特定数据

为了更好地规范定义的主机与组变量，Ansible 支持将 `/etc/ansible/hosts` 定义的主机名与组变量单独剥离出来存放指定的文件中，将采用 YAML 格式存放，存放位置规定：“`/etc/ansible/group_vars/+ 组名`”和“`/etc/ansible/host_vars/+ 主机名`”分别存放指定组名或主机名定义的变量，例如：

```

/etc/ansible/group_vars/dbservers


```

```
/etc/ansible/group_vars/webrowsers
/etc/ansible/host_vars/foosball
```

定义的 dbrowsers 变量格式为：

```
[ /etc/ansible/group_vars/dbrowsers ]
```

```
---
ntp_server: acme.example.org
database_server: storage.example.org
```

 **提示** 在 Ansible 1.2 及以后版本中，group\_vars/ 和 host\_vars/ 目录可以保存在 playbook 目录或 inventory 目录，如同时存在，inventory 目录的优先级高于 playbook 目录的。

## 9.4 匹配目标

在 9.3 节中已经完成主机与组的定义，本节将讲解如何进行目标（Patterns）匹配，格式为：ansible <pattern\_goes\_here> -m <module\_name> -a <arguments>。举例说明：重启 webrowsers 组的所有 Apache 服务。

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

本节将重点介绍 <pattern\_goes\_here> 参数的使用方法，详细规则及含义见表 9-2。

表 9-2 匹配目标主机规则表

规 则	含 义
192.198.1.2 或 one.example.com	匹配目标 IP 地址或主机名，多个 IP 或主机名使用 “:” 号分隔
webrowsers	匹配目标组为 webrowsers，多个组使用 “:” 号分隔
All 或 ‘*’	匹配目标所有主机
~(web db).*\.example\.com 或 192.168.1.*	支持正则表达式匹配主机或 IP 地址
webrowsers:!192.168.1.22	匹配 webrowsers 组且排除 192.168.1.22 主机 IP
webrowsers:&dbrowsers	匹配 webrowsers 与 dbrowsers 两个群组的交集
webrowsers:!{{excluded}}:&{{required}}	支持变量匹配方式

## 9.5 Ansible 常用模块及 API

Ansible 提供了非常丰富的功能模块，包括 Cloud（云计算）、Commands（命令行）、

Database (数据库)、Files (文件管理)、Internal (内置功能)、Inventory (资产管理)、Messaging (消息队列)、Monitoring (监控管理)、Net Infrastructure (网络基础服务)、Network (网络管理)、Notification (通知管理)、Packaging (包管理)、Source Control (版本控制)、System (系统服务)、Utilities (公共服务)、Web Infrastructure (Web 基础服务), 等等, 更多模块介绍见官网模块介绍 (网址: <http://ansibleworks.com/docs/modules.html>)。模块默认存储目录为 /usr/share/ansible/, 存储结构以模块分类名作为目录名, 模块文件按分类存放在不同类别目录中。命令行调用模块格式: `ansible <pattern_goes_here (操作目标)> -m <module_name (模块名)> -a <module_args (模块参数)>`, 其中默认模块名为 `command`, 即 “-m command” 可省略。获取远程 `webservers` 组主机的 `uptime` 信息格式如图 9-3 所示。

```
[root@SN2013-08-020 ~]# ansible webservers -m command -a "uptime"
192.168.1.22 | success | rc=0 >>
07:33:20 up 31 min, 1 user, load average: 0.00, 0.00, 0.00

192.168.1.21 | success | rc=0 >>
10:11:02 up 12:41, 1 user, load average: 0.01, 0.01, 0.00
```

图 9-3 获取主机 “uptime” 信息

以上命令等价于 `ansible webservers -a "uptime"`, 获得模块的帮助说明信息格式: `ansible-doc <模块名>`, 得到 `ping` 模块的帮助说明信息如图 9-4 所示。

```
[root@SN2013-08-020 ~]# ansible-doc ping
> PING

A trivial test module, this module always returns 'pong' on
successful contact. It does not make sense in playbooks, but it is
useful from '/usr/bin/ansible'

# Test 'webservers' status
ansible webservers -m ping
```

图 9-4 ping 模块帮助信息

在 playbooks 中运行远程命令格式如下:

```
- name: reboot the servers
  action: command /sbin/reboot -t now
```

Ansible 0.8 或以上版本支持以下格式:

```
- name: reboot the servers
  command: /sbin/reboot -t now
```

Ansible 提供了非常丰富的模块, 涉及日常运维工作的方方面面。下面介绍 Ansible 的常用模块, 更多模块介绍见官方说明。

## 1. 远程命令模块

### (1) 功能

模块包括 `command`、`script`、`shell`，都可以实现远程 shell 命令运行。`command` 作为 Ansible 的默认模块，可以运行远程权限范围所有的 shell 命令；`script` 功能是在远程主机执行主控端存储的 shell 脚本文件，相当于 `scp+shell` 组合；`shell` 功能是执行远程主机的 shell 脚本文件。

### (2) 例子

```
ansible webserver -m command -a "free -m"
ansible webserver -m script -a "/home/test.sh 12 34"
ansible webserver -m shell -a "/home/test.sh"
```

## 2. copy 模块

### (1) 功能

实现主控端向目标主机拷贝文件，类似于 `scp` 的功能。

### (2) 例子

以下示例实现拷贝 `/home/test.sh` 文件至 `webserver` 组目标主机 `/tmp/` 目录下，并更新文件属主及权限（可以单独使用 `file` 模块实现权限的修改，格式为：`path=/etc/foo.conf owner=foo group=foo mode=0644`）。

```
#
ansible webserver -m copy -a "src=/home/test.sh dest=/tmp/ owner=root
group=root mode=0755"
```

## 3. stat 模块

### (1) 功能

获取远程文件状态信息，包括 `atime`、`ctime`、`mtime`、`md5`、`uid`、`gid` 等信息。

### (2) 例子

```
ansible webserver -m stat -a "path=/etc/sysctl.conf"
```

## 4. get\_url 模块

### (1) 功能

实现在远程主机下载指定 URL 到本地，支持 `sha256sum` 文件校验。

## (2) 例子

```
ansible webservers -m get_url -a "url=http://www.baidu.com dest=/tmp/index.html
mode=0440 force=yes"
```

## 5. yum 模块

### (1) 功能

Linux 平台软件包管理操作，常见有 yum、apt 管理方式。

### (2) 例子

```
ansible webservers -m apt -a "pkg=curl state=latest"
ansible webservers -m yum -a "name=curl state=latest"
```

## 6. cron 模块

### (1) 功能

远程主机 crontab 配置。

### (2) 例子

```
ansible webservers -m cron -a "name='check dirs' hour='5,2' job='ls -alh > /dev/null'"
```

效果如下：

```
#Ansible: check dirs
* 5,2 * * * ls -alh > /dev/nullsalt '*' file.chown /etc/passwd root root
```

## 7. mount 模块

### (1) 功能

远程主机分区挂载。

### (2) 例子

```
ansible webservers -m mount -a "name=/mnt/data src=/dev/sd0 fstype=ext3 opts=ro
state=present"
```

## 8. service 模块

### (1) 功能

远程主机系统服务管理。

### (2) 例子

```
ansible webserver -m service -a "name=nginx state=stopped"
ansible webserver -m service -a "name=nginx state=restarted"
ansible webserver -m service -a "name=nginx state=reloaded"
```

## 9. sysctl 包管理模块

### (1) 功能

远程 Linux 主机 sysctl 配置。

### (2) 例子

```
sysctl: name=kernel.panic value=3 sysctl_file=/etc/sysctl.conf checks=before
reload=yessalt '*' pkg.upgrade
```

## 10. user 服务模块

### (1) 功能

远程主机系统用户管理。

### (2) 例子

```
# 添加用户 johnd;
ansible webserver -m user -a "name=johnd comment='John Doe'"
# 删除用户 johnd;
ansible webserver -m user -a "name=johnd state=absent remove=yes"
```



playbooks 模块调用格式如下，以 command 模块为例（0.8 或更新版本格式）：

- name: reboot the servers

command: /sbin/reboot -t now

## 9.6 playbook 介绍

playbook 是一个不同于使用 Ansible 命令行执行方式的模式，其功能更强大灵活。简单来说，playbook 是一个非常简单的配置管理和多主机部署系统，不同于任何已经存在的模式，可作为一个适合部署复杂应用程序的基础。playbook 可以定制配置，可以按指定的操作步骤有序执行，支持同步及异步方式。官方提供了大量的例子，可以在 <https://github.com/ansible/ansible-examples> 找到。playbook 是通过 YAML 格式来进行描述定义的，可以实现多台主机应用的部署，定义在 webservers 及 dbservers 组上执行特定指令步骤。下面为读者介绍一个基本的 playbook 示例：

**[ /home/test/ansible/playbooks/nginx.yml ]**

```
---
- hosts: webservers
  vars:
    worker_processes: 4
    num_cpus: 4
    max_open_file: 65506
    root: /data
    remote_user: root
  tasks:
    - name: ensure nginx is at the latest version
      yum: pkg=nginx state=latest
    - name: write the nginx config file
      template: src=/home/test/ansible/nginx/nginx2.conf dest=/etc/nginx/nginx.conf
      notify:
        - restart nginx
    - name: ensure nginx is running
      service: name=nginx state=started
  handlers:
    - name: restart nginx
      service: name=nginx state=restarted
```

以上 playbook 定制了一个简单的 Nginx 软件包管理，内容包括安装、配置模板、状态管理等。下面详细对该示例进行说明。

### 9.6.1 定义主机与用户

在 playbook 执行时，可以为主机或组定义变量，比如指定远程登录用户。以下为 webservers 组定义的相关变量，变量的作用域只限于 webservers 组下的主机。

```
- hosts: webservers
  vars:
    worker_processes: 4
    num_cpus: 4
```

```
max_open_file: 65506
root: /data
remote_user: root
```

hosts 参数的作用为定义操作的对象，可以是主机或组，具体定义规则见 9.3.1 节内容。本示例定义操作主机为 webserver 组，同时通过 vars 参数定义了 4 个变量（配置模板用到），其中 remote\_user 为指定远程操作的用户名，默认为 root 账号，支持 sudo 方式运行，通过添加 sudo: yes 即可。注意，remote\_user 参数在 Ansible 1.4 或更高版本才引入。

## 9.6.2 任务列表

所有定义的任务列表（tasks list），playbook 将按定义的配置文件自上而下的顺序执行，定义的主机都将得到相同的任务，但执行的返回结果不一定保持一致，取决于主机的环境及程序包状态。建议每个任务事件都要定义一个 name 标签，好处是增强可读性，也便于观察结果输出时了解运行的位置，默认使用 action（具体的执行动作）来替换 name 作为输出。下面是一个简单的任务定义示例：

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

功能是检测 Nginx 服务是否为运行状态，如没有则启动。其中 name 标签对下面的 action（动作）进行描述；action（动作）部分可以是 Ansible 的任意模块，具体见 9.5 节，本例为 services 模块，参数使用 key=value 的格式，如 “name=httpd”，在定义任务时也可以引用变量，格式如下：

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

在 playbook 可通过 template 模块对本地配置模板文件进行渲染并同步到目标主机。以 nginx 配置文件为例，定义如下：

```
- name: write the nginx config file
  template: src=/home/test/ansible/nginx/nginx2.conf dest=/etc/nginx/nginx.conf
  notify:
  - restart nginx
```

其中，“src=/home/test/ansible/nginx/nginx2.conf”为管理端模板文件存放位置，“dest=/etc/nginx/nginx.conf”为目标主机 nginx 配置文件位置，通过下面 nginx 模板文件可以让大家对模板的定义有个基本的概念。

```
【 /home/test/ansible/nginx/nginx2.conf 】
```

```

user          nginx;
worker_processes {{ worker_processes }};
{% if num_cpus == 2 %}
worker_cpu_affinity 01 10;
{% elif num_cpus == 4 %}
worker_cpu_affinity 1000 0100 0010 0001;
{% elif num_cpus >= 8 %}
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000 00100000
01000000 10000000;
{% else %}
worker_cpu_affinity 1000 0100 0010 0001;
{% endif %}
worker_rlimit_nofile {{ max_open_file }};
... ..

```

Ansible 会根据定义好的模板渲染成真实的配置文件，模板使用 YAML 语法，详细见 9.1 节，最终生成的 nginx.conf 配置如下：

```

user          nginx;
worker_processes 4;
worker_cpu_affinity 1000 0100 0010 0001;
worker_rlimit_nofile 65506;
... ..

```

当目标主机配置文件发生变化后，通知处理程序（Handlers）来触发后续的动作，比如重启 nginx 服务。Handlers 中定义的处理程序在没有通知触发时是不会执行的，触发后也只会运行一次。触发是通过 Handlers 定义的 name 标签来识别的，比如下面 notify 中的“restart nginx”与 handlers 中的“name: restart nginx”保持一致。

```

notify:
  - restart nginx
handlers:
  - name: restart nginx
    service: name=nginx state=restarted

```

### 9.6.3 执行 playbook

执行 playbook，可以通过 ansible-playbook 命令实现，格式：ansible-playbook playbook file (.yaml) [ 参数 ]，如启用 10 个并行进程数执行 playbook：

```
#ansible-playbook /home/test/ansible/playbooks/nginx.yml -f 10,
```

其他常用参数说明：

- ❑ -u REMOTE\_USER：手工指定远程执行 playbook 的系统用户；
- ❑ --syntax-check：检查 playbook 的语法；

- ❑ `--list-hosts` playbooks: 匹配到的主机列表;
- ❑ `-T TIMEOUT`: 定义 playbook 执行超时时间;
- ❑ `--step`: 以单任务分步骤运行, 方便做每一步的确认工作。

更多参数说明运行 `ansible-playbook -help` 来获得。

## 9.7 playbook 角色与包含声明

当我们写一个非常大的 playbook 时, 想要复用些功能显得有些吃力, 还好 Ansible 支持写 playbook 时拆分成多个文件, 通过包含 (include) 的形式进行引用, 我们可以根据多种维度进行“封装”, 比如定义变量、任务、处理程序等。

角色建立在包含文件之上, 抽象后更加清晰、可复用。运维人员可以更专注于整体, 只有在需要时才关注具体细节。Ansible 官方在 GitHub 上提供了大量的示例供大家参考借鉴, 访问地址 <https://github.com/ansible/ansible-examples> 即可获相应的学习资料。

### 9.7.1 包含文件, 鼓励复用

当多个 playbook 涉及复用的任务列表时, 可以将复用的内容剥离出, 写到独立的文件当中, 最后在需要的地方 include 进来即可, 示例如下:

【tasks/foo.yml】

```
---
# possibly saved as tasks/foo.yml
- name: placeholder foo
  command: /bin/foo
- name: placeholder bar
  command: /bin/bar
```

然后就可以在使用的 playbook 中 include 进来, 如:

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

当然, 也可以将变量传递到包含文件当中, 这称为“参数包含”。

如在部署多个 WordPress 的情况下, 可以根据不同用户单独部署 WordPress 的任务, 且引用单个 wordpress.yml 文件, 可以这样写:

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

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

注意，1.4 或更高版本可支持以 Python 的字典、列表的传递参数形式，如：

```
tasks:
- { include: wordpress.yml, user: timmy, ssh_keys: [ 'keys/one.txt', 'keys/
two.txt' ] }
```

使用这两种方法都进行变量传递，然后在包含文件中通过使用 `{{ user }}` 进行变量引用。

将处理程序（handlers）放到包含文件中是一个好的做法，比如重启 Apache 的任务，如下：

【 handlers/handlers.yml 】

```
---
# this might be in a file like handlers/handlers.yml
- name: restart apache
  service: name=apache state=restarted
```

需要时可以进行引用，像这样：

```
handlers:
- include: handlers/handlers.yml
```

## 9.7.2 角色

现在我们已经了解了变量、任务、处理程序的定义，有什么方法更好地进行组织或抽象，让其复用性更强、功能更具模块化？答案就是角色。角色是 Ansible 定制好的一种标准规范，以不同级别目录层次及文件对角色、变量、任务、处理程序等进行拆分，为后续功能扩展、可维护性打下基础。一个典型角色目录结构的示例如下：

```
site.yml
webserver.yml
fooservers.yml
roles/
  common/
    files/
    templates/
    tasks/
    handlers/
    vars/
    meta/
  webserver/
    files/
```

```

templates/
tasks/
handlers/
vars/
meta/

```

在 playbook 是这样引用的：

【 site.yml 】

```

---
- hosts: webservers
  roles:
    - common
    - webservers

```

角色定制以下规范，其中 x 为角色名。

- ❑ 如 roles/x/tasks/main.yml 文件存在，其中列出的任务将被添加到执行队列；
- ❑ 如 roles/x/handlers/main.yml 文件存在，其中所列的处理程序将被添加到执行队列；
- ❑ 如 roles/x/vars/main.yml 文件存在，其中列出的变量将被添加到执行队列；
- ❑ 如 roles/x/meta/main.yml 文件存在，所列任何作用的依赖关系将被添加到角色的列表（1.3 及更高版本）；
- ❑ 任何副本任务可以引用 roles/x/files/ 无需写路径，默认相对或绝对引用；
- ❑ 任何脚本任务可以引用 roles/x/files/ 无需写路径，默认相对或绝对引用；
- ❑ 任何模板任务可以引用文件中的 roles/x/templates/ 无需写路径，默认相对或绝对引用。

为了便于大家更好地理解和使用角色（role），对 9.6 节中的 nginx 软件包管理的 playbook（独立文件）修改成角色的形式，同时添加了一个公共类角色 common，从角色全局作用域中抽取出公共的部分，一般为系统的基础服务，比如 ntp、iptables、selinux、sysctl 等。本示例是针对 ntp 服务的管理。

### （1）playbook 目录结构

playbook 目录包括变量定义目录 group\_vars、主机组定义文件 hosts、全局配置文件 site.yml、角色功能目录，playbook 目录结构可参考图 9-5。

【 /home/test/ansible/playbooks/nginx 】

### （2）定义主机组

以下定义了一个业务组 webservers，成员为两台主机。

【 nginx/hosts 】

```
[webservers]
192.168.1.21
192.168.1.22
```

非必选配置，默认将引用 /etc/ansible/hosts 的参数，角色中自定义组与主机文件将通过“-i file”命令行参数调用，如 `ansible-playbook -i hosts` 来调用。

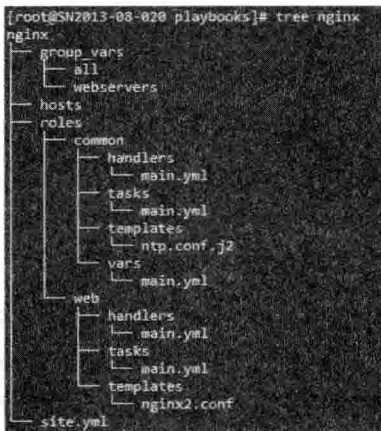


图 9-5 playbook 主目录结构

### (3) 定义主机或组变量

定义规则见 9.3 节所述，`group_vars` 为定义组变量目录，目录当中的文件名要与组名保持一致，组变量文件定义的变量作为域只受限于该组，`all` 代表所有主机。

#### 【nginx/group\_vars/all】

```
---
# Variables listed here are applicable to all host groups
ntpserver: ntp.sjtu.edu.cn
```

#### 【nginx/group\_vars/webservers】

```
---
worker_processes: 4
num_cpus: 4
max_open_file: 65536
root: /data
```

### (4) 全局配置文件 site.yml

下面的全局配置文件引用了两个角色块，角色的应用范围及实现功能都不一样：

**【 nginx/site.yml 】**

```

---
- name: apply common configuration to all nodes
  hosts: all
  roles:
    - common

- name: configure and deploy the webserver and application code
  hosts: webserver
  roles:
    - web

```

全局配置文件 site.yml 引用了两个角色，一个为公共类的 common，另一个为 web 类，分别对应 nginx/common、nginx/web 目录。以此类推，可以引用更多的角色，如 db、nosql、hadoop 等，前提是我们先要进行定义，通常情况下一个角色对应着一个特定功能服务。通过 hosts 参数来绑定角色对应的主机或组。

**(5) 角色 common 的定义**

角色 common 定义了 handlers、tasks、templates、vars 4 个功能类，分别存放处理程序、任务列表、模板、变量的配置文件 main.yml，需要注意的是，vars/main.yml 中定义的变量优先级高于 /nginx/group\_vars/all，可以从 ansible-playbook 的执行结果中得到验证。各功能块配置文件定义如下：

**【 handlers/main.yml 】**

```

- name: restart ntp
  service: name=ntpd state=restarted

```

**【 tasks/main.yml 】**

```

- name: Install ntp
  yum: name=ntp state=present

- name: Configure ntp file
  template: src=ntp.conf.j2 dest=/etc/ntp.conf
  notify: restart ntp

- name: Start the ntp service
  service: name=ntpd state=started enabled=true

- name: test to see if selinux is running
  command: getenforce
  register: sestatus
  changed_when: false

```

其中 `template: src=ntp.conf.j2` 引用模板时无需写路径，默认在上级的 `templates` 目录中查找。

#### 【 templates/ntp.conf.j2 】

```
driftfile /var/lib/ntp/drift
restrict 127.0.0.1
restrict -6 ::1

server {{ ntpserver }}

includefile /etc/ntp/crypto/pw
keys /etc/ntp/keys
```

此处 `{{ ntpserver }}` 将引用 `vars/main.yml` 定义的 `ntpserver` 变量。

#### 【 vars/main.yml 】

```
---
# Variables listed here are applicable to all host groups
ntpserver: 210.72.145.44
```

### (6) 角色 web 的定义

角色 `web` 定义了 `handlers`、`tasks`、`templates` 三个功能类，基本上是 9.6 节中的 `nginx` 管理 `playbook` 对应定义功能段打散后的内容。具体功能块配置文件定义如下：

#### 【 handlers/main.yml 】

```
- name: restart nginx
  service: name=nginx state=restarted
```

#### 【 tasks/main.yml 】

```
- name: ensure nginx is at the latest version
  yum: pkg=nginx state=latest
- name: write the nginx config file
  template: src=nginx2.conf dest=/etc/nginx/nginx.conf
  notify:
    - restart nginx
- name: ensure nginx is running
  service: name=nginx state=started
```

#### 【 templates/nginx2.conf 】

```
user          nginx;
worker_processes  {{ worker_processes }};
{% if num_cpus == 2 %}
```

```

worker_cpu_affinity 01 10;
{% elif num_cpus == 4 %}
worker_cpu_affinity 1000 0100 0010 0001;
{% elif num_cpus >= 8 %}
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000 00100000
01000000 10000000;
{% else %}
worker_cpu_affinity 1000 0100 0010 0001;
{% endif %}
worker_rlimit_nofile {{ max_open_file }};
.....

```

具体 web 角色定义细节将不展开描述, 可参考 9.6 节及 common 角色的说明。

### (7) 运行角色

```

#cd /home/test/ansible/playbooks/nginx
#ansible-playbook -i hosts site.yml -f 10

```

运行结果如图 9-6 与图 9-7 所示。

```

TASK: [Install ntp] *****
ok: [192.168.1.21]
ok: [192.168.1.22]

TASK: [Configure ntp file] *****
changed: [192.168.1.21]
changed: [192.168.1.22]

TASK: [Start the ntp service] *****
ok: [192.168.1.22]
ok: [192.168.1.21]

```

图 9-6 ntp 部署片段

```

TASK: [ensure nginx is at the latest version] *****
ok: [192.168.1.22]
ok: [192.168.1.21]

TASK: [write the nginx config file] *****
ok: [192.168.1.22]
ok: [192.168.1.21]

TASK: [ensure nginx is running] *****
ok: [192.168.1.22]
ok: [192.168.1.21]

PLAY RECAP *****
192.168.1.21      : ok=9    changed=2    unreachable=0    failed=0
192.168.1.22      : ok=9    changed=2    unreachable=0    failed=0

```

图 9-7 nginx 部署片段

## 9.8 获取远程主机系统信息: Facts

Facts 是一个非常有用的组件, 类似于 Saltstack 的 Grains 功能, 实现获取远程主机的

系统信息，包括主机名、IP 地址、操作系统、分区信息、硬件信息等，可以配合 `playbook` 实现更加个性化、灵活的功能需求，比如在 `httpd.conf` 模板中引用 Facts 的主机名信息作为 `ServerName` 参数的值。通过运行 `ansible hostname -m setup` 可获取 Facts 信息，例如，获取 192.168.1.21 的 Facts 信息需运行：`ansible 192.168.1.21 -m setup`，结果如下：

```
192.168.1.21 | success >> {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
      "192.168.1.21"
    ],
    "ansible_all_ipv6_addresses": [
      "fe80::250:56ff:fe28:632d"
    ],
    "ansible_architecture": "x86_64",
    "ansible_bios_date": "07/02/2012",
    "ansible_bios_version": "6.00",
    "ansible_cmdline": {
      "KEYBOARDTYPE": "pc",
      "KEYTABLE": "us",
      "LANG": "en_US.UTF-8",
      "SYSFONT": "latarcyrheb-sun16",
      "quiet": true,
      "rd_NO_DM": true,
      "rd_NO_LUKS": true,
      "rd_NO_LVM": true,
      "rd_NO_MD": true,
      "rhgb": true,
      "ro": true,
      "root": "UUID=b8d29324-57b2-4949-8402-7fd9ad64ac5a"
    },
    .....
  },
  .....
```

在模板文件中这样引用 Facts 信息：

```
{{ ansible_devices.sda.model }}
{{ ansible_hostname }}
```

## 9.9 变量

在实际应用场景中，我们希望一些任务、配置根据设备性能的不同而产生差异，比如使用本机 CPU 核数动态配置 Nginx 的 `worker_processes` 参数，可能有一组主机的应用配置文件几乎相同，但略有不同的配置项可以引用变量。在 Ansible 中使用变量的目的是方便处理系统之间的差异。

变量名的命名规则由字母、数字和下划线组合而成，变量必须以字母开头，如 “`foo_`”

port”是一个合法的变量，“foo5”也是可以的，“foo-port”、“foo port”、“foo.port”和“12”都是非法的变量命名。在 Inventory 中定义变量见 9.3.2 节和 9.3.3 节，在 playbook 定义变量见 9.6 节，建议回顾一下，加深记忆。

### 9.9.1 Jinja2 过滤器

Jinja2 是 Python 下一个广泛应用的模板引擎，它的设计思想类似于 Django 的模板引擎，并扩展了其语法和一系列强大的功能，官网地址：<http://jinja.pocoo.org/>。下面介绍一下 Ansible 使用 Jinja2 强大的过滤器 (Filters) 功能。

使用格式：{{ 变量名 | 过滤方法 }}。

下面是实现获取一个文件路径变量过滤出文件名的一个示例：

```
{{ path | basename }}
```

获取文件所处的目录名：

```
{{ path | dirname }}
```

下面为一个完整的示例，实现从 “/etc/profile” 中过滤出文件名 “profile”，并输出重定向到 /tmp/testshell 文件中。

```
---
- hosts: 192.168.1.21
  vars:
    filename: /etc/profile
  tasks:
    - name: "shell1"
      shell: echo {{ filename | basename }} >> /tmp/testshell
```

更多的过滤方法见 <http://jinja.pocoo.org/docs/templates/#builtin-filters>。

### 9.9.2 本地 Facts

我们可以通过 Facts 来获取目标主机的系统信息，当这些信息还不能满足我们的功能需求时，可以通过编写自定义的 Facts 模块来实现。当然，还有一个更简单的实现方法，就是通过本地 Facts 来实现。只需在目标设备 /etc/ansible/facts.d 目录定义 JSON、INI 或可执行文件的 JSON 输出，文件扩展名要求使用 “.fact”，这些文件都可以作为 Ansible 的本地 Facts，例如，在目标设备 192.168.1.21 定义三个变量，供以后 playbook 进行引用。

```
【 /etc/ansible/facts.d/preferences.fact 】
```

```
[general]
```

```
max_memory_size=32
max_user_processes=3730
open_files=65535
```

在主控端运行 `ansible 192.168.1.21 -m setup -a "filter=ansible_local"` 可看到定义的结果，返回结果如下：

```
192.168.1.21 | success >> {
  "ansible_facts": {
    "ansible_local": {
      "preferences": {
        "general": {
          "max_memory_size": "32",
          "max_user_processes": "3730",
          "open_files": "65535"
        }
      }
    }
  },
  "changed": false
}
```

注意返回 JSON 的层次结构，`preferences`（facts 文件名前缀）→ `general`（INI 的节名）→ `key:value`（INI 的键与值），最后就可以在我们的模板或 `playbook` 中通过以下方式进行调用：

```
{{ ansible_local.preferences.general.open_files }}
```

### 9.9.3 注册变量

变量的另一个用途是将一条命令的运行结果保存到变量中，供后面的 `playbook` 使用。下面是一个简单的示例：

```
- hosts: web_servers
  tasks:
    - shell: /usr/bin/foo
      register: foo_result
      ignore_errors: True

    - shell: /usr/bin/bar
      when: foo_result.rc == 5
```

上述示例注册了一个 `foo_result` 变量，变量值为 `shell: /usr/bin/foo` 的运行结果，`ignore_errors: True` 为忽略错误。变量注册完成后，就可以在后面 `playbook` 中使用了，当条件语句 `when: foo_result.rc == 5` 成立时，`shell: /usr/bin/bar` 命令才会运行，其中 `foo_result.rc` 为返回 / `usr/bin/foo` 的 `resultcode`（返回码）。图 9-8 返回“rc=0”的返回码。

```
[root@SN2013-08-020 ~]# ansible 192.168.1.21 -m command -a "echo 'This certainly is epic'"
192.168.1.21 | success | rc=0 >>
This certainly is epic
```

图 9-8 命令执行结果

## 9.10 条件语句

有时候一个 `playbook` 的结果取决于一个变量，或者取决于上一个任务（`task`）的执行结果值，在某些情况下，一个变量的值可以依赖于其他变量的值，当然也会影响 Ansible 的执行过程。

下面主要介绍 `When` 声明。

有时候我们想跳过某些主机的执行步骤，比如符合特定版本的操作系统将不安装某个软件包，或者磁盘空间爆满了将进行清理的步骤。在 Ansible 中很容易做到这一点，通过 `When` 子句实现，其中将引用 Jinja2 表达式。下面是一个简单的示例：

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

通过定义任务的 Facts 本地变量 `ansible_os_family`（操作系统版本名称）是否为 Debian，结果将返回 `BOOL` 类型值，为 `True` 时将执行上一条语句 `command: /sbin/shutdown -t now`，为 `False` 时该条语句都不会触发。我们再看一个示例，通过判断一条命令执行结果做不同分支的二级处理。

```
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
```

“`when: result|success`”的意思为当变量 `result` 执行结果为成功状态时，将执行 `/bin/something_else` 命令，其他同理，其中 `success` 为 Ansible 内部过滤器方法，返回 `Ture` 代表命令运行成功。

## 9.11 循环

通常一个任务会做很多事情，如创建大量的用户、安装很多包，或重复轮询特定的步骤，直到某种结果条件为止，Ansible 为我们提供了此支持。下面是一个简单的示例：

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

这个示例实现了一个批量创建系统用户的功能，with\_items 会自动循环执行上面的语句“user: name={{ item }} state=present groups=wheel”，循环的次数为 with\_items 的元素个数，这里有 2 个元素，分别为 testuser1、testuser2，会分别替换 {{ item }} 项。这个示例与下面的示例是等价的：

```
- name: add user testuser1
  user: name=testuser1 state=present groups=wheel
- name: add user testuser2
  user: name=testuser2 state=present groups=wheel
```

当然，元素也支持字典的形式，如下：

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

循环也支持列表 (List) 的形式，不过是通过 with\_flattened 语句来实现的，例如：

```
----
# file: roles/foo/vars/main.yml
packages_base:
  - [ 'foo-package', 'bar-package' ]
packages_apps:
  - [ ['one-package', 'two-package' ] ]
  - [ ['red-package'], ['blue-package'] ]
```

以上定义了两个列表变量，分别是需要安装的软件包名，以便后面进行如下引用：

```
- name: flattened loop demo
  yum: name={{ item }} state=installed
  with_flattened:
    - packages_base
    - packages_apps
```

通过使用 `with_flattened` 语句循环引用定义好的列表变量。

## 9.12 示例讲解

官网提供的 Haproxy+LAMP+Nagios 经典示例，也是目前国内最常用的技术架构，此案例访问地址为：[https://github.com/ansible/ansible-examples/tree/master/lamp\\_haproxy](https://github.com/ansible/ansible-examples/tree/master/lamp_haproxy)。下面将对该示例进行详细说明，内容覆盖前面涉及的几乎所有知识点，起到温故的作用，同时作为对 Ansible 的总结内容。

下面介绍 playbook 的基本信息。

### 1. 目录结构

示例 playbook 目录结构见图 9-9。

```
root@SN2813-08-020 ansible# tree lamp_haproxy/
lamp_haproxy/
├── group_vars
│   ├── all
│   ├── dbservers
│   ├── lbservers
│   └── webservers
├── hosts
├── roles
│   ├── base-apache
│   │   ├── tasks
│   │   └── main.yml
│   ├── common
│   │   ├── files
│   │   │   ├── epel.repo
│   │   │   └── RPM-GPG-KEY-EPEL-6
│   │   ├── handlers
│   │   │   └── main.yml
│   │   ├── tasks
│   │   │   └── main.yml
│   │   ├── templates
│   │   │   ├── iptables.j2
│   │   │   └── ntp.conf.j2
│   ├── db
│   │   ├── handlers
│   │   │   └── main.yml
│   │   ├── tasks
│   │   │   └── main.yml
│   │   ├── templates
│   │   │   └── my.cnf.j2
│   ├── haproxy
│   │   ├── handlers
│   │   │   └── main.yml
│   │   ├── tasks
│   │   │   └── main.yml
│   │   ├── templates
│   │   │   └── haproxy.cfg.j2
│   ├── nagios
│   │   ├── files
│   │   │   ├── ansible-managed-services.cfg
│   │   │   ├── localhost.cfg
│   │   │   └── nagios.cfg
│   │   ├── handlers
│   │   │   └── main.yml
│   │   ├── tasks
│   │   │   └── main.yml
│   │   ├── templates
│   │   │   ├── dbservers.cfg.j2
│   │   │   ├── lbservers.cfg.j2
│   │   │   └── webservers.cfg.j2
│   └── web
│       └── tasks
│           └── main.yml
└── site.yml
```

图 9-9 示例目录结构

## 2. 设备环境说明

两台 Web 主机、1 台数据库主机、1 台负载均衡器主机、1 台监控主机，hosts 配置如下：

### 【 hosts 】

```
[webservers]
web1
web2
[dbservers]
db1
[lbservers]
lb1
[monitoring]
nagios
```

## 3. palybook 入口文件 site.yml

需要注意的是 base-apache 角色，由于 webservers 及 monitoring 都需要部署 Apache 环境，为提高复用性，将部署 Apache 独立成 base-apache 角色。

### 【 Site.yml 】

```
---
- hosts: all
  roles:
    - common
- hosts: dbservers
  user: root
  roles:
    - db
- hosts: webservers
  user: root
  roles:
    - base-apache
    - web
- hosts: lbservers
  user: root
  roles:
    - haproxy
- hosts: monitoring
  user: root
  roles:
    - base-apache
    - nagios
```

#### 4. 定义组变量

下面定义 playbook 全局变量，变量作用域为所有主机。

【 group\_vars/all 】

```
---
# Variables here are applicable to all host groups

httpd_port: 80
ntpserver: 192.168.1.2
```

all 文件定义了匹配所有主机作用域的变量，一般为系统公共类基础配置，如 ntpserver 地址、sysctl 变量、iptables 配置等。

下面为定义 webservers 组的变量，变量作用域为 webservers 组主机。

【 group\_vars/webservers 】

```
---
# Variables for the web server configuration

# Ethernet interface on which the web server should listen.
# Defaults to the first interface. Change this to:
#
#   iface: eth1
#
# ...to override.
#
iface: '{{ ansible_default_ipv4.interface }}'

# this is the repository that holds our sample webapp
repository: https://github.com/bennojoy/mywebapp.git

# this is the shalsum of V5 of the test webapp.
webapp_version: 351e47276cc66b018f4890a04709d4cc3d3edb0d
```

webservers 文件定义了 webservers 组作用域的变量。本示例涉及 Apache 相关配置，其中 “iface: '{{ ansible\_default\_ipv4.interface }}'” 引用了 Facts 获取的本地网卡接口名信息，另外定义了一个 GitHub 的 repository，方便下载 Web 测试文件，如内部搭建 git 版本控制环境，此处也可以修改成本地的服务地址。

下面为定义 dbservers 组的变量，变量作用域为 dbservers 组主机。

【 group\_vars/dbservers 】

```
---
# The variables file used by the playbooks in the dbservers group.
```

```
# These don't have to be explicitly imported by vars_files: they are
autopopulated.
```

```
mysqlservice: mysqld
mysql_port: 3306
dbuser: root
dbname: foodb
upassword: abc
```

dbservers 文件定义了 dbservers 组作用域变量，本示例涉及 MySQL 数据库的基本应用信息。

下面为定义 lbservers 组作用域变量文件，本示例主要涉及 haproxy 环境涉及的配置参数值。

### 【 group\_vars/lbservers 】

```
---
# Variables for the HAProxy configuration

# HAProxy supports "http" and "tcp". For SSL, SMTP, etc, use "tcp".
mode: http

# Port on which HAProxy should listen
listenport: 8888

# A name for the proxy daemon, this will be the suffix in the logs.
daemonname: myapplb

# Balancing Algorithm. Available options:
# roundrobin, source, leastconn, source, uri
# (if persistence is required use, "source")
balance: roundrobin

# Ethernet interface on which the load balancer should listen
# Defaults to the first interface. Change this to:
#
# iface: eth1
#
# ...to override.
#
iface: '{{ ansible_default_ipv4.interface }}'
```

## 5. playbook 角色详解

本示例划分了 6 个角色，包括 base-apache、common、db、haproxy、nagios、web，分别对应 6 个功能环境部署，根据不同业务场景的需求，可以随意加、减角色，如将 base-apache 更换成 nginx，然后在 site.yml 中引用。

### (1) common 角色

common 的主要功能是部署、配置系统基础服务，包括 yum 源、安装 nagios 插件、NTP 服务、iptables、SELinux 等，任务 (tasks) 的定义如下：

【 roles/common/tasks/main.yml 】

```
---
# This role contains common plays that will run on all nodes.
- name: Create the repository for EPEL
  copy: src=epel.repo dest=/etc/yum.repos.d/epel.repo

- name: Create the GPG key for EPEL
  copy: src=RPM-GPG-KEY-EPEL-6 dest=/etc/pki/rpm-gpg

- name: install some useful nagios plugins
  yum: name={{ item }} state=present
  with_items:
    - nagios-nrpe
    - nagios-plugins-swap
    - nagios-plugins-users
    - nagios-plugins-procs
    - nagios-plugins-load
    - nagios-plugins-disk

- name: Install ntp
  yum: name=ntp state=present
  tags: ntp

- name: Configure ntp file
  template: src=ntp.conf.j2 dest=/etc/ntp.conf
  tags: ntp
  notify: restart ntp

- name: Start the ntp service
  service: name=ntpd state=started enabled=true
  tags: ntp

- name: insert iptables template
  template: src=iptables.j2 dest=/etc/sysconfig/iptables
  notify: restart iptables

- name: test to see if selinux is running
  command: getenforce
  register: sestatus
  changed_when: false
```

上述代码定义了两个远程文件复制 copy，其中 src (源文件) 的默认位置在 roles/common/files，使用 with\_item 标签实现循环安装 nagios 插件，同时安装 ntp 服务，引用模

块文件 `roles/common/templatesntp.conf.j2`，且同步到目标主机 `/etc/ntp.conf` 位置。配置系统 `iptables`，引用 `roles/common/templates/iptables.j2` 模板，“`notify: restart iptables`”，状态或模板发生变化时将通知处理程序（handlers）来处理。“`command: getenforce`”运行 `getenforce` 来检测 `selinux` 是否在运行状态，“`changed_when: false`”作用为不记录命令运行结果的 `changed` 状态，即 `changed` 为 `False`。

下面定义 `common` 角色的处理程序。

【 `roles/common/handlers/main.yml` 】

```
---
# Handlers for common notifications
- name: restart ntp
  service: name=ntpd state=restarted

- name: restart iptables
  service: name=iptables state=restarted
```

上述代码定义了两个处理程序，功能分别为重启 `ntp`、`iptables` 服务，其中“`name: restart ntp`”与任务（tasks）定义中的“`notify: restart ntp`”是一一对应的，“`name: restart iptables`”同理。

下面定义了 `common` 角色 `iptables` 的配置模板：

【 `roles/common/templates/iptables.j2` 】

```
{% if (inventory_hostname in groups['webservers']) or (inventory_hostname in
groups['monitoring']) %}
-A INPUT -p tcp --dport 80 -j ACCEPT
{% endif %}
... ..
{% for host in groups['monitoring'] %}
-A INPUT -p tcp -s {{ hostvars[host].ansible_default_ipv4.address }} --dport
5666 -j ACCEPT
{% endfor %}
```

“`inventory_hostname`”作为存放在 Ansible 的 `inventory` 文件中的主机名或 IP，好处是可以不依靠 `Facts` 的主机名参数 `ansible_hostname` 或其他原因，一般情况下 `inventory_hostname` 等于 `ansible_hostname`，但有时候我们习惯在 Ansible 的 `inventory` 中使用 IP 地址，而 `ansible_hostname` 则返回主机名。模板使用了 `jinja2` 的语法，本例 `if...endif` 语句判断当前的 `inventory_hostname` 是否在 `webservers` 及 `monitoring` 组中（定义具体在 `hosts` 文件中），条件成立则添加 80 端口访问权限（`-A INPUT -p tcp --dport 80 -j ACCEPT`）。`For...endfor` 语句实现了循环开通允许 `monitoring` 组主机访问 5666 端口，使用 `hostvars[host]` 得到主机对象，可以获得主机的 `Facts` 信息，如 `hostvars[host].ansible_default_ipv4.address` 获取主机 IP。

## (2) haproxy 角色

haproxy 角色主要实现了 haproxy 平台的部署、配置功能，任务 (tasks) 的定义：

### 【 roles/haproxy/tasks 】

```
---
# This role installs HAProxy and configures it.

- name: Download and install haproxy and socat
  yum: name={{ item }} state=present
  with_items:
    - haproxy
    - socat

- name: Configure the haproxy cnf file with hosts
  template: src=haproxy.cfg.j2 dest=/etc/haproxy/haproxy.cfg
  notify: restart haproxy
```

任务 (tasks) 定义了两个功能，一为安装，二为同步配置文件，安装使用了 yum 模块，循环安装 haproxy、socat 两个工具，同时根据配置参数渲染 roles/haproxy/templates/haproxy.cfg.j2 模板文件，完成后同步到目标主机 /etc/haproxy/haproxy.cfg 位置，状态发生变化时重启 haproxy 服务，使之生效。

下面定义了 haproxy 角色 haproxy.cfg 的配置模板：

### 【 roles/haproxy/templates/haproxy.cfg.j2 】

```
... ..
backend app
    {% for host in groups['lb_servers'] %}
        listen {{ daemonname }} {{ hostvars[host]['ansible_' + iface].ipv4.
address }}:{{ listenport }}
    {% endfor %}
    balance {{ balance }}
    {% for host in groups['web_servers'] %}
        server {{ hostvars[host].ansible_hostname }} {{ hostvars[host]
['ansible_' + iface].ipv4.address }}:{{ httpd_port }}
    {% endfor %}
```

{{ hostvars[host]['ansible\_' + iface].ipv4.address }} 实现了获取网卡名变量 iface (group\_vars/lb\_servers 中定义) 的 IPv4 IP 地址。

## (3) web 角色

web 角色主要实现了 php、php-mysql、git 平台部署及 SELinux 的配置功能，任务 (tasks) 的定义如下：

## 【 roles/web/tasks/main.yml 】

```

---
# httpd is handled by the base-apache role upstream
- name: Install php and git
  yum: name={{ item }} state=present
  with_items:
    - php
    - php-mysql
    - git
- name: Configure SELinux to allow httpd to connect to remote database
  seboolean: name=httpd_can_network_connect_db state=true persistent=yes
  when: sestatus.rc != 0
- name: Copy the code from repository
  git: repo={{ repository }} version={{ webapp_version }} dest=/var/www/html/

```

判断 `sestatus` 变量（roles/common/tasks/main.yml 中定义）返回的 `rc`（运行代码）不等于 0（失败）则配置 `selinux` `httpd` 访问远程数据库的权限，使用的是 `Ansible` 的 `seboolean` 模块，该条语句等价于命令行 “`setsebool httpd_can_network_connect_db 1`”，其中 “`persistent=yes`” 表示开机自启动。

## （4）nagios 角色

`nagios` 角色主要实现了 `nagios` 监控平台的部署，重点介绍任务（tasks）的定义：

## 【 roles/nagios/tasks/ main.yml 】

```

... ..
- name: create the nagios object files
  template: src={{ item + ".j2" }}
            dest=/etc/nagios/ansible-managed/{{ item }}
  with_items:
    - webservers.cfg
    - dbservers.cfg
    - lbservers.cfg
  notify: restart nagios

```

`template` 分发多个模板文件时可以使用 `with_items` 来循环同步，变量与字符使用 “+” 号连接（具体见 `jinja2` 语法）。

理解以上 4 个角色的定义后，再理解 `ansible-examples` 其他 `playbook` 的内容已经没有太大的困难，本书将不一一说明。



参考提示

❑ 9.1 节 YAML 语法介绍参考 <http://zh.wikipedia.org/zh-cn/YAML>。

❑ 9.2 节 ~ 9.11 节 Ansible 介绍及示例参考 <http://docs.ansible.com> 官网文档。

## 集中化管理平台 Saltstack 详解

Saltstack (<http://www.saltstack.com/>) 是一个服务器基础架构集中化管理平台，开始于 2011 年的一个项目，具备配置管理、远程执行、监控等功能，一般可以理解成简化版的 puppet (<http://puppetlabs.com/>) 和加强版的 func (<https://fedorahosted.org/func/>)。Saltstack 基于 Python 语言实现，结合轻量级消息队列 (ZeroMQ) 与 Python 第三方模块 (Pyzmq、PyCrypto、Pyjinja2、python-msgpack 和 PyYAML 等) 构建。Saltstack 具备如下特点。

- ❑ 部署简单、方便。
- ❑ 支持大部分 UNIX/Linux 及 Windows 环境。
- ❑ 主从集中化管理。
- ❑ 配置简单、功能强大、扩展性强。
- ❑ 主控端 (master) 和被控制端 (minion) 基于证书认证，安全可靠。
- ❑ 支持 API 及自定义模块，可通过 Python 轻松扩展。

通过部署 Saltstack 环境，我们可以在成千上万台服务器上做到批量执行命令，根据不同业务特性进行配置集中化管理、分发文件、采集服务器数据、操作系统基础及软件包管理等，因此，Saltstack 是运维人员提高工作效率、规范业务配置与操作的利器。目前 Saltstack 已经趋向成熟，用户群及社区活跃度都不错，同时官方也开放了不少子项目，具体可访问 <https://github.com/saltstack> 获得。

为了方便读者更系统化地了解 Saltstack 的技术点，本章将针对相关技术点详细展开介绍。

## 10.1 Saltstack 的安装

Saltstack 的不同角色服务安装非常简单，建议读者采用 yum 源方式来实现部署，下面介绍具体步骤。

### 10.1.1 业务环境说明

为了方便读者理解，笔者通过虚拟化环境部署了两组业务功能服务器来进行演示，操作系统版本为 CentOS release 6.4，自带 Python 2.6.6。相关服务器信息如表 10-1 所示（CPU 核数及 Nginx 根目录的差异化是为方便演示生成动态配置的需要）。

表 10-1 环境说明表

角色	Id (minion id)	IP	Groupsnode (组名)	Cpus (核数)	Web Root(Nginx 根目录)
Master	SN2013-08-020	192.168.1.20	—	—	—
minion	SN2012-07-010	192.168.1.10	web1group	2	/www
minion	SN2012-07-011	192.168.1.11	web1group	4	/www
minion	SN2012-07-012	192.168.1.12	web1group	2	/www
minion	SN2013-08-021	192.168.1.21	web2group	2	/data
minion	SN2013-08-022	192.168.1.22	web2group	2	/data

### 10.1.2 安装 EPEL

由于目前 RHEL 官网 yum 源还没有 Saltstack 的安装包支持，因此先安装 EPEL 作为部署 Saltstack 的默认 yum 源。

- ❑ RHEL(CentOS) 5 版本: rpm -Uvh 下载地址: <http://mirror.pnl.gov/epel/5/i386/epel-release-5-4.noarch.rpm>
- ❑ RHEL(CentOS) 6 版本: rpm -Uvh 下载地址: <http://ftp.linux.ncsu.edu/pub/epel/6/i386/epel-release-6-8.noarch.rpm>

### 10.1.3 安装 Saltstack

#### (1) 主服务器安装（主控端）

```
#yum install salt-master -y
#chkconfig salt-master on
#service salt-master start
```

#### (2) 从服务器安装（被控端）

```
#yum install salt-minion -y
```

```
#chkconfig salt-minion on
#service salt-minion start
```

### 10.1.4 Saltstack 防火墙配置

在主控端添加 TCP 4505、TCP 4506 的规则，而在被控端无须配置防火墙，原理是被控端直接与主控端的 zeromq 建立长链接，接收广播到的任务信息并执行，具体操作是添加两条 iptables 规则：

```
iptables -I INPUT -m state --state new -m tcp -p tcp --dport 4505 -j ACCEPT
iptables -I INPUT -m state --state new -m tcp -p tcp --dport 4506 -j ACCEPT
```

### 10.1.5 更新 Saltstack 配置及安装校验

Saltstack 分两种角色，一种为 master（主控端），另一种为 minion（被控端），安装完毕后需要对两种角色的配置文件进行修改，下面具体说明。

#### （1）master 主控端配置

##### 1) 更新主控端关键项配置：

##### 【 /etc/salt/master 】

```
# 绑定 Master 通信 IP;
interface: 192.168.1.20
# 自动认证，避免手动运行 salt-key 来确认证书信任;
auto_accept: True
# 指定 Saltstack 文件根目录位置
file_roots:
  base:
    - /srv/salt
```

##### 2) 重启 saltstack salt-master 服务使新配置生效，具体执行以下命令：

```
#service salt-master restart
```

#### （2）minion 被控端配置

##### 1) 更新被控端关键项配置：

##### 【 /etc/salt/minion 】

```
# 指定 master 主机 IP 地址
master: 192.168.1.20
# 修改被控端主机识别 id，建议使用操作系统主机名来配置
id: SN2013-08-021
```

2) 重启 saltstack salt-minion 服务使新配置生效, 具体执行以下命令:

```
service salt-minion restart
```

(3) 校验安装结果

通过 test 模块的 ping 方法, 可以确认指定被控端设备与主控端是否建立信任关系及连通性是否正常, 探测所有被控端采用 '\*' 来代替 'SN2013-08-021' 即可, 具体如图 10-1 所示。

```
[root@SN2013-08-020 ~]# salt 'SN2013-08-021' test.ping
SN2013-08-021:
True
```

图 10-1 测试安装主机的连通性



**提示** 当 /etc/salt/master 没有配置 auto\_accept: True 时, 需要通过 salt-key 命令来进行证书认证操作, 具体操作如下:

- ☐ salt-key -L, 显示已经或未认证的被控端 id, Accepted Keys 为已认证清单, Unaccepted Keys 为未认证清单;
- ☐ salt-key -D, 删除所有认证主机 id 证书;
- ☐ salt-key -d id, 删除单个 id 证书;
- ☐ salt-key -A, 接受所有 id 证书请求;
- ☐ salt-key -a id, 接受单个 id 证书请求。

## 10.2 利用 Saltstack 远程执行命令

Saltstack 的一个比较突出优势是具备执行远程命令的功能, 操作及方法与 func (<https://fedorahosted.org/func/>) 相似, 可以帮助运维人员完成集中化的操作平台。

命令格式: salt '<操作目标>' <方法> [参数]

示例: 查看被控主机的内存使用情况, 如图 10-2 所示。

```
[root@SN2013-08-020 ~]# salt 'SN2013-08-021' cmd.run 'free -m'
SN2013-08-021:

```

	total	used	free	shared	buffers	cached
Mem:	482	446	35	0	47	20
-/+ buffers/cache:		378	104			
Swap:	1023	27	996			

图 10-2 查看“SN2013-08-021”主机内存使用

其中针对 < 操作目标 >，Saltstack 提供了多种方法对被控端主机 (id) 进行过滤。下面列举常用的具体参数。

1) -E, --pcr，通过正则表达式进行匹配。示例：控测 SN2013 字符开头的主机 id 名是否连通，命令：salt -E '^SN2013.\*' test.ping，运行结果如图 10-3 所示。

```
[root@SN2013-08-020 ~]# salt -E '^SN2013.*' test.ping
SN2013-08-021:
  True
SN2013-08-022:
  True
```

图 10-3 正则匹配主机的连通性

2) -L, --list，以主机 id 名列表的形式进行过滤，格式与 Python 的列表相似，即不同主机 id 名称使用逗号分隔。示例：获取主机 id 名为 SN2013-08-021、SN2013-08-022；获取完整操作系统发行版名称，命令：salt -L 'SN2013-08-021,SN2013-08-022' grains.item osfullname，运行结果如图 10-4 所示。

```
[root@SN2013-08-020 ~]# salt -L 'SN2013-08-021,SN2013-08-022' grains.item osfullname
SN2013-08-021:
  osfullname: CentOS
SN2013-08-022:
  osfullname: CentOS
```

图 10-4 列表形式匹配主机的操作系统类型

3) -G, --grain，根据被控主机的 grains (10.4 节详解) 信息进行匹配过滤，格式为 '<grain value>:<glob expression>'，例如，过滤内核为 Linux 的主机可以写成 'kernel:Linux'，如果同时需要正则表达式的支持可切换成 --grain-pcr 参数来执行。示例：获取主机发行版本号为 6.4 的 Python 版本号，命令：salt -G 'osrelease:6.4' cmd.run 'python -V'，运行结果如图 10-5 所示。

```
[root@SN2013-08-020 ~]# salt -G 'osrelease:6.4' cmd.run 'python -V'
SN2013-08-021:
  Python 2.6.6
SN2013-08-022:
  Python 2.6.6
```

图 10-5 grain 形式匹配主机的 Python 版本

4) -I, --pillar，根据被控主机的 pillar (10.5 节详解) 信息进行匹配过滤，格式为“对象名称:对象值”，例如，过滤所有具备 'apache:htpd' pillar 值的主机。示例：探测具有“nginx:root: /data”信息的主机连通性，命令：salt -I 'nginx:root:/data' test.ping，运行结果如图 10-6 所示。

```
[root@SN2013-08-020 ~]# salt -I 'nginx:root:/data' test.ping
SN2013-08-021:
  True
SN2013-08-022:
  True
```

图 10-6 pillar 形式匹配主机的连通性

其中 pillar 属性配置文件如下（关于 pillar 后面 10.5 单独进行说明）：

```
nginx:
  root: /data
```

5) -N, --nodegroup, 根据主控端 master 配置文件中的分组名称进行过滤。以笔者定义的组为例（主机信息支持正则表达式、grain、条件运算符等），通常根据业务类型划分，不同业务具备相同的特点，包括部署环境、应用平台、配置文件等。举例分组配置信息如下：

【 /etc/salt/master 】

```
nodegroups:
  web1group: 'L@SN2012-07-010,SN2012-07-011,SN2012-07-012'
  web2group: 'L@SN2013-08-021,SN2013-08-022'
```

其中，L@ 表示后面的主机 id 格式为列表，即主机 id 以逗号分隔；G@ 表示以 grain 格式描述；S@ 表示以 IP 子网或地址格式描述。

示例：探测 web2group 被控主机的连通性，其命令为：salt -N web2group test.ping，运行结果如图 10-7 所示。

```
[root@SN2013-08-020 ~]# salt -N web2group test.ping
SN2013-08-022:
  True
SN2013-08-021:
  True
```

图 10-7 分组形式 (nodegroup) 匹配主机的连通性

6) -C, --compound, 根据条件运算符 not、and、or 去匹配不同规则的主机信息。示例：探测 SN2013 开头并且操作系统版本为 CentOS 的主机连通性，命令如下：

```
salt -C 'E@^SN2013.* and G@os:Centos' test.ping
```

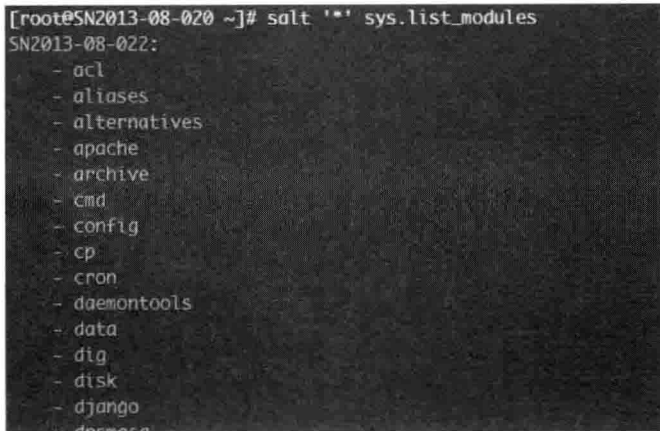
其中，not 语句不能作为第一个条件执行，不过可以通过以下方法来规避，示例：探测非 SN2013 开头的主机连通性，其命令为：salt -C '\* and not E@^SN2013.\*' test.ping。

7) -S, --ipcidr, 根据被控主机的 IP 地址或 IP 子网进行匹配，示例如下：

```
salt -S 192.168.0.0/16 test.ping
salt -S 192.168.1.10 test.ping
```

### 10.3 Saltstack 常用模块及 API

Saltstack 提供了非常丰富的功能模块，涉及操作系统的基础功能、常用工具支持等，更多模块信息见官网模块介绍：<http://docs.saltstack.com/ref/modules/all/index.html>。当然，也可以通过 sys 模块列出当前版本支持的模块，如图 10-8 所示。



```
[root@SN2013-08-020 ~]# salt '*' sys.list_modules
SN2013-08-022:
- acl
- aliases
- alternatives
- apache
- archive
- cmd
- config
- cp
- cron
- daemontools
- data
- dig
- disk
- django
- dracut
```

图 10-8 所有主机 Saltstack 支持的模块清单（部分截图）

接下来抽取常见的模块进行介绍，同时也会列举模块 API 使用方法。API 的原理是通过调用 master client 模块，实例化一个 LocalClient 对象，再调用 cmd() 方法来实现的。以下是 API 实现 test.ping 的示例：

```
import salt.client
client = salt.client.LocalClient()
ret = client.cmd('*', 'test.ping')
print ret
```

结果以一个标准的 Python 字典形式的字符串返回，可以通过 eval() 函数转换成 Python 的字典类型，方便后续的业务逻辑处理，程序运行结果如下：

```
{'SN2013-08-022': True, 'SN2013-08-021': True}
```



**提示** 将字典字符串转换成 Python 的字典类型，推荐使用 ast 模块的 literal\_eval() 方法，可以过滤表达式中的恶意函数。

### (1) Archive 模块

1) 功能: 实现系统层面的压缩包调用, 支持 gunzip、gzip、rar、tar、unrar、unzip 等。

2) 示例:

```
# 采用 gunzip 解压 /tmp/sourcefile.txt.gz 包
salt '*' archive.gunzip /tmp/sourcefile.txt.gz
```

```
# 采用 gzip 压缩 /tmp/sourcefile.txt 文件
salt '*' archive.gzip /tmp/sourcefile.txt
```

3) API 调用:

```
client.cmd('*', ' archive.gunzip', ['/tmp/sourcefile.txt.gz '])
```

### (2) cmd 模块

1) 功能: 实现远程的命令行调用执行 (默认具备 root 操作权限, 使用时需评估风险)。

2) 示例:

```
# 获取所有被控主机的内存使用情况
salt '*' cmd.run "free -m"
# 在 SN2013-08-021 主机运行 test.sh 脚本, 其中 script/test.sh 存放在 file_roots 指定的目录,
# 该命令会做两个动作: 首先同步 test.sh 到 minion 的 cache 目录 (如同步到 /var/cache/salt/
# minion/files/base/script/test.sh); 其次运行该脚本
'SN2013-08-021' cmd.script salt://script/test.sh
```

3) API 调用:

```
client.cmd('SN2013-08-021', 'cmd.run', ['free -m'])
```

### (3) cp 模块

1) 功能: 实现远程文件、目录的复制, 以及下载 URL 文件等操作。

2) 示例:

```
# 将指定被控主机的 /etc/hosts 文件复制到被控主机本地的 salt cache 目录 (/var/cache/salt/
minion/localfiles/);
```

```
salt '*' cp.cache_local_file /etc/hosts
```

```
# 将主服务器 file_roots 指定位置下的目录复制到被控主机
```

```
salt '*' cp.get_dir salt://path/to/dir/ /minion/dest
```

```
# 将主服务器 file_roots 指定位置下的文件复制到被控主机
```

```
salt '*' cp.get_file salt://path/to/file /minion/dest
```

# 下载 URL 内容到被控主机指定位置

```
salt '*' cp.get_url http://www.slashdot.org /tmp/index.html
```

### 3) API 调用:

```
client.cmd('SN2013-08-021', 'cp.get_file', ['salt://path/to/file', '/minion/dest'])
```

## (4) cron 模块

1) 功能: 实现被控主机的 crontab 操作。

2) 示例:

# 查看指定被控主机、root 用户的 crontab 清单

```
salt 'SN2013-08-022' cron.raw_cron root
```

# 为指定的被控主机、root 用户添加 /usr/local/weekly 任务作业

```
salt 'SN2013-08-022' cron.set_job root '*' '*' '*' '*' 1 /usr/local/weekly
```

# 删除指定的被控主机、root 用户 crontab 的 /usr/local/weekly 任务作业

```
salt 'SN2013-08-022' cron.rm_job root /usr/local/weekly
```

### 3) API 调用:

```
client.cmd('SN2013-08-021', 'cron.set_job', ['root', '*', '*', '*', '*', '*', '/usr/echo'])
```

## (5) dnsutil 模块

1) 功能: 实现被控主机通用 DNS 相关操作。

2) 示例:

# 添加指定被控主机 hosts 的主机配置项

```
salt '*' dnsutil.hosts_append /etc/hosts 127.0.0.1 ad1.yuk.com,ad2.yuk.com
```

# 删除指定被控主机 hosts 的主机配置项

```
salt '*' dnsutil.hosts_remove /etc/hosts ad1.yuk.com
```

### 3) API 调用:

```
client.cmd('*', 'dnsutil.hosts_append', ['/etc/hosts', '127.0.0.1', 'ad1.yuk.co'])
```

## (6) file 模块

1) 功能: 被控主机文件常见操作, 包括文件读写、权限、查找、校验等。

2) 示例:

# 校验所有被控主机 /etc/fstab 文件的 md5 是否为 6254e84e2f6ffa54e0c8d9cb230f5505, 一致则

```

返回 True
salt '*' file.check_hash /etc/fstab md5=6254e84e2f6ffa54e0c8d9cb230f5505

# 校验所有被控主机文件的加密信息、支持 md5、sha1、sha224、sha256、sha384、sha512 加密算法
salt '*' file.get_sum /etc/passwd md5

# 修改所有被控主机 /etc/passwd 文件的属组、用户权限，等价于 chown root:root /etc/passwd
salt '*' file.chown /etc/passwd root root

# 复制所有被控主机本地 /path/to/src 文件到本地的 /path/to/dst 文件
salt '*' file.copy /path/to/src /path/to/dst
# 检查所有被控主机 /etc 目录是否存在，存在则返回 True，检查文件是否存在使用 file.file_exists 方法
salt '*' file.directory_exists /etc

# 获取所有被控主机 /etc/passwd 的 stats 信息
salt '*' file.stats /etc/passwd

# 获取所有被控主机 /etc/passwd 的权限 mode，如 755、644
salt '*' file.get_mode /etc/passwd

# 修改所有被控主机 /etc/passwd 的权限 mode 为 0644
salt '*' file.set_mode /etc/passwd 0644

# 在所有被控主机创建 /opt/test 目录
salt '*' file.mkdir /opt/test

# 将所有被控主机 /etc/httpd/httpd.conf 文件的 LogLevel 参数的 warn 值修改成 info
salt '*' file.sed /etc/httpd/httpd.conf 'LogLevel warn' 'LogLevel info'

# 给所有被控主机的 /tmp/test/test.conf 文件追加内容 "maxclient 100"
salt '*' file.append /tmp/test/test.conf "maxclient 100"

# 删除所有被控主机的 /tmp/foo 文件
salt '*' file.remove /tmp/foo

```

### 3) API 调用:

```
client.cmd('*', 'file.remove ', ['/tmp/foo'])
```

## (7) iptables 模块

### 1) 功能：被控主机 iptables 支持。

### 2) 示例:

```

# 在所有被控端主机追加 (append)、插入 (insert) iptables 规则，其中 INPUT 为输入链
salt '*' iptables.append filter INPUT rule='-m state --state RELATED,ESTABLISHED
-j ACCEPT'
salt '*' iptables.insert filter INPUT position=3 rule='-m state --state
RELATED,ESTABLISHED -j ACCEPT'

```

```
# 在所有被控端主机删除指定链编号为 3 (position=3) 或指定存在的规则
salt '*' iptables.delete filter INPUT position=3
salt '*' iptables.delete filter INPUT rule='-m state --state RELATED,ESTABLISHED
-j ACCEPT'

# 保存所有被控端主机规则到本地硬盘 (/etc/sysconfig/iptables)
salt '*' iptables.save /etc/sysconfig/iptables
```

### 3) API 调用:

```
client.cmd('SN2013-08-022', 'iptables.append',['filter','INPUT','rule='\-p tcp
--sport 80 -j ACCEPT\'''])
```

## (8) netwrok 模块

1) 功能: 返回被控主机网络信息。

2) 示例:

```
# 在指定被控主机 'SN2013-08-022' 获取 dig、ping、traceroute 目录域名信息
salt 'SN2013-08-022' network.dig www.qq.com
salt 'SN2013-08-022' network.ping www.qq.com
salt 'SN2013-08-022' network.traceroute www.qq.com

# 获取指定被控主机 'SN2013-08-022' 的 MAC 地址
salt 'SN2013-08-022' network.hwaddr eth0

# 检测指定被控主机 'SN2013-08-022' 是否属于 10.0.0.0/16 子网范围, 属于则返回 True
salt 'SN2013-08-022' network.in_subnet 10.0.0.0/16

# 获取指定被控主机 'SN2013-08-022' 的网卡配置信息
salt 'SN2013-08-022' network.interfaces

# 获取指定被控主机 'SN2013-08-022' 的 IP 地址配置信息
salt 'SN2013-08-022' network.ip_addrs

# 获取指定被控主机 'SN2013-08-022' 的子网信息
salt 'SN2013-08-022' network.subnets
```

### 3) API 调用:

```
client.cmd('SN2013-08-022', 'network.ip_addrs')
```

## (9) pkg 包管理模块

1) 功能: 被控主机程序包管理, 如 yum、apt-get 等。

2) 示例:

```
# 为所有被控主机安装 PHP 环境, 根据不同系统发行版调用不同安装工具进行部署, 如 redhat 平台的 yum,
等价于 yum -y install php
```

```
salt '*' pkg.install php
```

```
# 卸载所有被控主机的 PHP 环境
```

```
salt '*' pkg.remove php
```

```
# 升级所有被控主机的软件包
```

```
salt '*' pkg.upgrade
```

### 3) API 调用:

```
client.cmd('SN2013-08-022', 'pkg.remove',['php'])
```

## (10) Service 服务模块

1) 功能: 被控主机程序包服务管理。

2) 示例:

```
# 开启 (enable)、禁用 (disable) nginx 开机自启动服务
```

```
salt '*' service.enable nginx
```

```
salt '*' service.disable nginx
```

```
# 针对 nginx 服务的 reload、restart、start、stop、status 操作
```

```
salt '*' service.reload nginx
```

```
salt '*' service.restart nginx
```

```
salt '*' service.start nginx
```

```
salt '*' service.stop nginx
```

```
salt '*' service.status nginx
```

### 3) API 调用:

```
client.cmd('SN2013-08-022', 'service.stop',['nginx'])
```

## (11) 其他模块

通过上面介绍的 10 个常用模块,基本上已经覆盖日常运维操作。Saltstack 还提供了 user (系统用户模块)、group (系统组模块)、partition (系统分区模块)、puppet (puppet 管理模块)、system (系统重启、关机模块)、timezone (时区管理模块)、nginx (Nginx 管理模块)、mount (文件系统挂载模块),等等,更多内容见官网介绍: <http://docs.saltstack.com/ref/modules/all/index.html#all-salt-modules>。当然,我们也可以通过 Python 扩展功能模块来满足需求。

## 10.4 grains 组件

grains 是 Saltstack 最重要的组件之一,grains 的作用是收集被控主机的基本信息,这些信息通常都是一些静态类的数据,包括 CPU、内核、操作系统、虚拟化等,在服务器端可以

根据这些信息进行灵活定制，管理员可以利用这些信息对不同业务进行个性化配置。官网提供的用来区分不同操作系统的示例如下（采用 jinja 模板）：

```
{% if grains['os'] == 'Ubuntu' %}
host: {{ grains['host'] }}
{% elif grains['os'] == 'CentOS' %}
host: {{ grains['fqdn'] }}
{% endif %}
```

示例中 CentOS 发行版主机将被 “host: {{ grains['fqdn'] }}” 匹配，以主机 SN2013-08-022（centOS 6.4）为例，最终得到 “host: SN2013-08-022”。同时，命令行的匹配操作系统发行版本为 CentOS 的被控端可以通过 -G 参数来过滤，如 salt -G 'os:CentOS' test.ping。

### 10.4.1 grains 常用操作命令

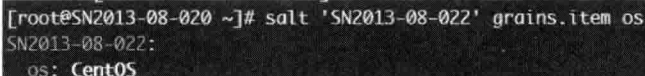
匹配内核版本为 2.6.32-358.14.1.el6.x86\_64 的主机：

```
salt -G 'kernelrelease:2.6.32-358.14.1.el6.x86_64' cmd.run 'uname -a'
```

获取所有主机的 grains 项信息：

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

当然，也可以获取主机单项 grains 数据，如获取操作系统发行版本，执行命令：salt 'SN2013-08-022' grains.item os，结果如图 10-9 所示。



```
[root@SN2013-08-020 ~]# salt 'SN2013-08-022' grains.item os
SN2013-08-022:
os: CentOS
```

图 10-9 根据 grains 获取主机操作系统发行版本信息

获取主机 id 为 “SN2013-08-022” 的所有 grains 键及值信息，执行命令如图 10-10 所示。

### 10.4.2 定义 grains 数据

定义 grains 数据的方法有两种，其中一种为在被控主机定制配置文件，另一种是通过主控端扩展模块 API 实现，区别是模块更灵活，可以通过 Python 编程动态定义，而配置文件只适合相对固定的键与值。下面分别举例说明。

```
[root@SN2013-08-020 ~]# salt 'SN2013-08-022' grains.items
SN2013-08-022:
  a: 1024
  biosreleasedate: 07/02/2012
  biosversion: 6.00
  cabinet: 13
  cpu_flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
  sxtopology tsc_reliable nonstop_tsc aperfmperf unfair_spinlock pni pclmulqdq ssse3 cx16 pcid
  s fsgsbase smep
  cpu_model: Intel(R) Pentium(R) CPU G2030 @ 3.00GHz
  cpuarch: x86_64
  defaultencoding: UTF8
  defaultlanguage: en_US
  deployment: datacenter4
  domain:
  fqdn: SN2013-08-022
  gpus:
    {'model': 'SVGA II Adapter', 'vendor': 'unknown'}
  host: SN2013-08-022
  id: SN2013-08-022
  ip_interfaces: {'lo': ['127.0.0.1'], 'eth0': ['192.168.1.22']}
  ipv4:
    127.0.0.1
    192.168.1.22
  kernel: Linux
  kernelrelease: 2.6.32-358.18.1.el6.x86_64
  localhost: SN2013-08-022
  manufacturer: VMware, Inc.
  master: 192.168.1.20
  max_open_file: 65535
```

图 10-10 获取主机所有 grains 信息（部分截图）

### 1. 被控端主机定制 grains 数据

SSH 登录一台被控主机，如 SN2013-08-022，配置文件定制的路径为 /etc/salt/minion，参数为 default\_include: minion.d/\*.conf，具体操作如下：

**[ /etc/salt/minion.d/hostinfo.conf ]**

```
grains:
  roles:
    - webserver
    - memcache
  deployment: datacenter4
  cabinet: 13
```

重启被控主机 salt-minion 服务，使之生效：service salt-minion restart。验证结果在主控端主机运行：salt 'SN2013-08-022' grains.item roles deployment cabinet，观察配置的键与值，如图 10-11 所示。

```
[root@SN2013-08-020 ~]# salt 'SN2013-08-022' grains.item roles deployment cabinet
SN2013-08-022:
  cabinet: 13
  deployment: datacenter4
  roles:
    webservers
    memcache
```

图 10-11 定制 grains 数据信息

## 2. 主控端扩展模块定制 grains 数据

首先在主控端编写 Python 代码，然后将该 Python 文件同步到被控主机，最后刷新生效（即编译 Python 源码文件成字节码 pyc）。在主控端 bash 目录（见 /etc/salt/master 配置文件的 file\_roots 项，默认的 base 配置在 /srv/salt）下生成 \_grains 目录，执行 `install -d /srv/salt/_grains` 开始编写代码，实现获取被控主机系统允许最大打开文件数（`ulimit -n`）的 grains 数据。

【 /srv/salt/\_grains/sysprocess.py 】

```
import os,sys,commands
def Grains_openfile():
    '''
        return os max open file of grains value
    '''
    grains = {}

    #init default value
    _open_file=65536

    try:
        getulimit=commands.getstatusoutput('source /etc/profile;ulimit -n')
    except Exception,e:
        pass
    if getulimit[0]==0:
        _open_file=int(getulimit[1])
    grains['max_open_file'] = _open_file
    return grains
```

上面代码的说明如下。

- `grains_openfile()` 定义一个获取最大打开文件数的函数，函数名称没有要求，符合 Python 的函数命名规则即可；
- `grains = {}` 初始化一个 grains 字典，变量名一定要用 grains，以便 Saltstack 识别；
- `grains['max_open_file'] = _open_file` 将获取的 Linux `ulimit -n` 的结果值赋予 grains['max\_open\_file']，其中“max\_open\_file”就是 grains 的项，\_open\_file 就是 grains 的值。

最后同步模块到指定被控端主机并刷新生效，因为 `grains` 比较适合采集静态类的数据，比如硬件、内核信息等。当有动态类的功能需求时，需要提行刷新，具体操作如下：

同步模块 `salt 'SN2013-08-022' saltutil.sync_all`，看看“SN2013-08-022”主机上发生了什么？文件已经同步到 `minion cache` 目录中，如下：

```
/var/cache/salt/minion/extmods/grains/grains_openfile.py
/var/cache/salt/minion/files/base/_grains/grains_openfile.py
```

`/var/cache/salt/minion/extmods/grains/` 为扩展模块文件最终存放位置，刷新模块后将在同路径下生成字节码 `pyc`；`/var/cache/salt/minion/files/base/_grains/` 为临时存放位置。

刷新模块 `salt 'SN2013-08-022' sys.reload_modules`，再看看主机发生了什么变化？在 `/var/cache/salt/minion/extmods/grains/` 位置多了一个编译后的字节码文件 `grains_openfile.pyc` 文件，为 Python 可执行的格式。

```
/var/cache/salt/minion/extmods/grains/grains_openfile.py
/var/cache/salt/minion/extmods/grains/grains_openfile.pyc
/var/cache/salt/minion/files/base/_grains/grains_openfile.py
```

校验结果为可以在主控端查看 `grains` 信息，执行 `salt 'SN2013-08-022' grains.item max_open_file`，结果显示“`max_open_file: 65535`”，这就是前面定制的主机 `grains` 信息。

```
SN2013-08-022:
  max_open_file: 65535
```

## 10.5 pillar 组件

`pillar` 也是 Saltstack 最重要的组件之一，其作用是定义与被控主机相关的任何数据，定义好的数据可以被其他组件使用，如模板、`state`、`API` 等。在 `pillar` 中定义的数据与不同业务特性的被控主机相关联，这样不同被控主机只能看到自己匹配的数据，因此 `pillar` 安全性很高，适用于一些比较敏感的数据，这也是区别于 `grains` 最关键的一点，如定义不同业务组主机的用户 `id`、组 `id`、读写权限、程序包等信息，定义的规范是采用 Python 字典形式，即键/值，最上层的键一般为主机的 `id` 或组名称。下面详细描述如何进行 `pillar` 的定义和使用。

## 10.5.1 pillar 的定义

### 1. 主配置文件定义

Saltstack 默认将主控端配置文件中的所有数据都定义到 pillar 中，而且对所有被控主机开放，可通过修改 /etc/salt/master 配置中的 pillar\_opts: Ture 或 False 来定义是否开启或禁用这项功能，修改后执行 salt '\*' pillar.data 来观察效果。图 10-12 为 pillar\_opts: Ture 的返回结果，以主机 “SN2013-08-022” 为例，执行 salt 'SN2013-08-022' pillar.data。

```
[root@SN2013-08-020 ~]# salt 'SN2013-08-022' pillar.data
SN2013-08-022:
-----
master:
-----
  auth_mode:
    1
  auto_accept:
    True
  cachedir:
    /var/cache/salt/master
  client_acl:
    -----
  client_acl_blacklist:
    -----
  cluster_masters:
  cluster_mode:
    paranoid
  conf_file:
    /etc/salt/master
  config_dir:
    /etc/salt
  cython_enable:
    False
  daemon:
    True
  default_include:
    master.d/*.conf
  enforce_mine_cache:
    False
```

图 10-12 主机所有 pillar 信息（部分截图）

### 2. SLS 文件定义

pillar 支持在 sls 文件中定义数据，格式须符合 YAML 规范，与 Saltstack 的 state 组件十分相似，新人容易将两者混淆，两者文件的配置格式、入口文件 top.sls 都是一致的。下面详细介绍 pillar 使用 sls 定义的配置过程。

### (1) 定义 pillar 的主目录

修改主配置文件 `/etc/salt/master` 的 `pillar_roots` 参数, 定义 pillar 的主目录, 格式如下:

```
pillar_roots:
  base:
    - /srv/pillar
```

同时创建 pillar 目录, 执行命令: `install -d /srv/pillar`。

### (2) 定义入口文件 top.sls

入口文件的作用一般是定义 pillar 的数据覆盖被控主机的有效域范围, “\*” 代表任意主机, 其中包括了一个 `data.sls` 文件, 具体内容如下:

【`/srv/pillar/top.sls`】

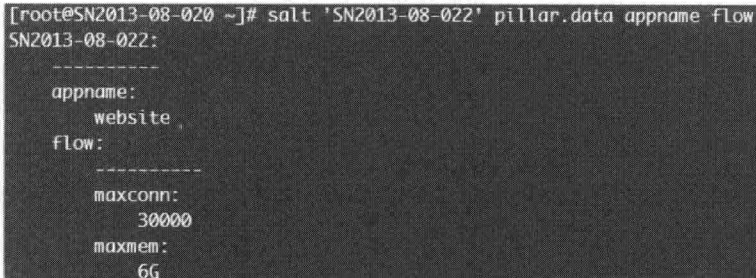
```
base:
  '*':
    - data
```

【`/srv/pillar/data.sls`】

```
appname: website
flow:
  maxconn: 30000
  maxmem: 6G
```

### (3) 校验 pillar

通过查看 “N2013-08-022” 主机的 pillar 数据, 可以看到多出了 `data.sls` 数据项, 原因是我们定义 `top.sls` 时使用 “\*” 覆盖了所有主机, 这样当查看 “SN2013-08-022” 的 pillar 数据时可以看到我们定义的数据, 如图 10-13 所示, 如果结果不符合预期, 可以尝试刷新被控主机 pillar 数据, 运行 `salt '*' saltutil.refresh_pillar` 即可。



```
[root@SN2013-08-020 ~]# salt 'SN2013-08-022' pillar.data appname flow
SN2013-08-022:
-----
appname:
  website
flow:
-----
  maxconn:
    30000
  maxmem:
    6G
```

图 10-13 返回主机 pillar 的信息

## 10.5.2 pillar 的使用

完成 pillar 配置后，接下来介绍使用方法。我们可以在 state、模板文件中引用，模板格式为 “{{ pillar 变量 }}”，例如：

```
{{ pillar['appname'] }} (一级字典)
{{ pillar['flow']['maxconn'] }} (二级字典) 或 {{ salt['pillar.get']('flow: 'maxconn', {}) }}
```

Python API 格式如下：

```
pillar['flow']['maxconn']
pillar.get(' flow:appname', {})
```

### 1. 操作目标主机

见 10.5.1 节，通过 -I 选项来使用 pillar 来匹配被控主机：

```
# salt -I 'appname:website' test.ping
SN2013-08-021:
    True
SN2013-08-022:
    True
```

### 2. 结合 grains 处理数据的差异性

首先通过结合 grains 的 id 信息来区分不同 id 的 maxcpu 的值，其次进行引用观察匹配的信息，延伸“10.5.1 pillar 的定义”的例子，将 data.sls 修改成如下形式，其中，“if … else … endif”为 jinja2 的模板语法，更多信息请访问 jinja2 官网语法介绍，网址为 <http://jinja.pocoo.org/docs/templates/>。

```
appname: website
flow:
    maxconn: 30000
    maxmem: 6G
    {% if grains['id'] == 'SN2013-08-022' %}
        maxcpu: 8
    {% else %}
        maxcpu: 4
    {% endif %}
```

通过查看被控主机的 pillar 数据，可以看到 maxcpu 的差异，如图 10-14 所示。

```

[root@SN2013-08-020 ~]# salt 'SN2013-08-021' pillar.data flow
SN2013-08-021:
-----
flow:
-----
maxconn:
    30000
maxcpu:
    4
maxmem:
    6G
[root@SN2013-08-020 ~]# salt 'SN2013-08-022' pillar.data flow
SN2013-08-022:
-----
flow:
-----
maxconn:
    30000
maxcpu:
    8
maxmem:
    6G

```

图 10-14 不同主机产生的 pillar 数据差异

## 10.6 state 介绍

state 是 Saltstack 最核心的功能，通过预先定制好的 sls (salt state file) 文件对被控主机进行状态管理，支持包括程序包 (pkg)、文件 (file)、网络配置 (network)、系统服务 (service)、系统用户 (user) 等，更多状态对象见 <http://docs.saltstack.com/ref/states/all/index.html>。

### 10.6.1 state 的定义

state 的定义是通过 sls 文件进行描述的，支持 YAML 语法，定义的规则如下：

```

$ID:
  $State:
    - $state: states

```

其中：

- \$ID，定义 state 的名称，通常采用与描述的对象保持一致的方法，如 apache、nginx 等；
- \$State，须管理对象的类型，详见 <http://docs.saltstack.com/ref/states/all/index.html>；
- \$state:states，定制对象的状态。

官网提供的示例如下：

```

1 apache:
2   pkg:
3     - installed
4   service:
5     - running
6     - require:
7       - pkg: apache

```

上述代码检查 apache 软件包是否已安装状态，如果未安装，将通过 yum 或 apt 进行安装；检查服务 apache 进程是否处于运行状态。下面详细进行说明：

第 1 行用于定义 state 的名称，此示例为 apache，当然也可以取其他相关的名称。

第 2 行和第 4 行表示 state 声明开始，使用了 pkg 和 service 这两个状态对象。pkg 使用系统本地的软件包管理器（yum 或 apt）管理将要安装的软件，service 管理系统守护进程。

第 3 行和第 5 行是要执行的方法。这些方法定义了 apache 软件包和服务目标状态，此示例要求软件包应当处于已安装状态，服务必须运行，如未安装将会被安装并启动。

第 6 行是关键字 require，它确保了 apache 服务只有在成功安装软件包后才会启动。



**注意** require：在运行此 state 前，先运行依赖的 state 关系检查，可配置多个 state 依赖对象；watch：在检查某个 state 发生变化时运行此模块。

## 10.6.2 state 的使用

state 的入口文件与 pillar 一样，文件名称都是 top.sls，但 state 要求 sls 文件必须存放在 saltstack base 定义的目录下，默认为 /srv/salt。state 描述配置 .sls 支持 jinja 模板、grains 及 pillar 引用等，在 state 的逻辑层次定义完成后，再通过 salt '\*' state.highstate 执行生效。下面扩展 10.5.1 节定义的范例，结合 grains 与 pillar，实现一个根据不同操作系统类型部署 apache 环境的任务。

### 1. 定义 pillar

【/srv/pillar/top.sls】

```

base:
  '*':
    - apache

```

在 top.sls 中引用二级配置有两种方式：一种是直接引用，如本示例中直接引用 apache.sls；另一种是创建 apache 目录，再引用目录中的 init.sls 文件，两者效果是一样的。为了规

范起见，笔者建议采用二级配置形式，同理，state 的 top.sls 也采用如此方式。

```
#mkdir /srv/pillar/apache # 创建 apache 目录
```

【 /srv/pillar/apache/init.sls 】

```
pkgs:
{% if grains['os_family'] == 'Debian' %}
    apache: apache2
{% elif grains['os_family'] == 'RedHat' %}
    apache: httpd
{% elif grains['os'] == 'Arch' %}
    apache: apache
{% endif %}
```

测试 pillar 数据，执行 salt '\*' pillar.data pkgs，结果返回以下信息，说明配置已生效。

```
SN2013-08-021:
```

```
-----
pkgs:
```

```
-----
apache:
```

```
    httpd
```

## 2. 定义 state

【 /srv/salt/top.sls 】

```
base:
  '*':
    - apache
```

【 /srv/salt/apache/init.sls 】

```
apache:
  pkg:
    - installed
    - name: {{ pillar['pkgs']['apache'] }}
  service.running:
    - name: {{ pillar['pkgs']['apache'] }}
    - require:
      - pkg: {{ pillar['pkgs']['apache'] }}
```

在配置中，{{ pillar['pkgs']['apache'] }} 将引用匹配到操作系统发行版对应的 pillar 数据，笔者的环境为 CentOS，故将匹配为 httpd，检查目标主机是否已经安装，没有则进行安装（yum -y install httpd），同时检查 apache 服务是否已经启动，没有则启动（/etc/init.d/httpd start）。

### 3. 执行 state

执行 state 及返回结果信息见图 10-15。

```
[root@SN2013-08-020 ~]# salt '*' state.highstate
SN2013-08-021:
-----
State: - pkg
Name:      httpd
Function:  installed
Result:    True
Comment:   The following packages were installed/updated: httpd.
Changes:   httpd: { new : 2.2.15-29.el6.centos
old :
}
-----
State: - service
Name:      httpd
Function:  running
Result:    True
Comment:   Started Service httpd
Changes:   httpd: True
```

图 10-15 执行 state 的结果信息

从图 10-15 中可以看出，结果返回两种对象类型结果，分别为 pkg 与 service，执行的结果是自动部署 apache 2.2.15 环境并启动服务。

## 10.7 示例：基于 Saltstack 实现的配置集中化管理

本示例实现一个集中化的 Nginx 配置管理，根据业务不同设备型号、分区、内核参数的差异化，动态产生适合本机环境的 Nginx 配置文件。本示例结合了 Saltstack 的 grains、grains\_module、pillar、state、jinja (template) 等组件。

### 10.7.1 环境说明

具体对照表 10-1 环境说明表，此处省略。

### 10.7.2 主控端配置说明

master 主配置文件的关键配置项如下：

【/etc/salt/master】(配置片段)

```
nodegroups:
  web1group: 'L@SN2012-07-010,SN2012-07-011,SN2012-07-012'
```

```

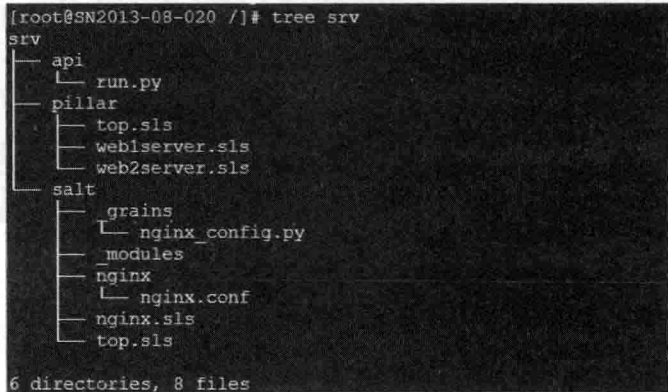
web2group: 'L@SN2013-08-021,SN2013-08-022'

file_roots:
  base:
    - /srv/salt

pillar_roots:
  base:
    - /srv/pillar

```

定义的 pillar、module api、state 目录结构，如图 10-16 所示。



```

[root@SN2013-08-020 /]# tree /srv
/srv
├── api
│   └── run.py
├── pillar
│   ├── top.sls
│   ├── web1server.sls
│   └── web2server.sls
└── salt
    ├── grains
    │   └── nginx_config.py
    ├── modules
    ├── nginx
    │   └── nginx.conf
    ├── nginx.sls
    └── top.sls
6 directories, 8 files

```

图 10-16 示例目录结构

使用 Python 编写 grains\_module，实现动态配置被控主机 grains 的 max\_open\_file 键，值为 ulimit -n 的结果，以便动态生成 Nginx.conf 中的 worker\_rlimit\_nofile、worker\_connections 参数的值，具体代码如下：

```

import os,sys,commands

def NginxGrains():
    '''
        return Nginx config grains value
    '''
    grains = {}
    max_open_file=65536
    try:
        getulimit=commands.getstatusoutput('source /etc/profile;ulimit -n')
    except Exception,e:
        pass
    if getulimit[0]==0:
        max_open_file=int(getulimit[1])
    grains['max_open_file'] = max_open_file

```

```
return grains
```

代码说明见“10.4.2 定义 Grains 数据”得“主控端扩展模块定制 Grains 数据”

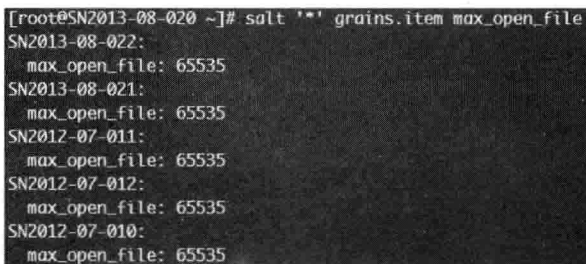
同步 grains 模块，运行：

```
# salt '*' saltutil.sync_all
```

刷新模块（让 minion 编译模块），运行：

```
# salt '*' sys.reload_modules
```

验证 max\_open\_file key 的 key 操作命令见图 10-17。



```
[root@SN2013-08-020 ~]# salt '*' grains.item max_open_file
SN2013-08-022:
  max_open_file: 65535
SN2013-08-021:
  max_open_file: 65535
SN2012-07-011:
  max_open_file: 65535
SN2012-07-012:
  max_open_file: 65535
SN2012-07-010:
  max_open_file: 65535
```

图 10-17 校验 max\_open\_file key 的 key 信息

### 10.7.3 配置 pillar

本示例使用分组规则定义 pillar，即不同分组引用各自的 sls 属性，使用 match 属性值进行区分，除了属性值为 nodegroup 外，还支持 grain、pillar 等形式。以下是使用 grain 作为区分条件例子：

```
dev:
  'os:Debian':
    - match: grain
    - servers
```

本示例通过 /etc/salt/master 中定义好的组信息，如 web1group 与 web2group 与业务组，分别引用 web1server.sls 与 web1server.sls，详见 /srv/pillar/top.sls 中的内容：

【 /srv/pillar/top.sls 】

```
base:
  web1group:
    - match: nodegroup
    - web1server
```

```
web2group:
  - match: nodegroup
  - web2server
```

定义私有配置。本示例通过 pillar 来配置 web\_root 的数据，当然，也可以根据不同需求进行定制，格式为 python 字典形式，即 "key:value"。

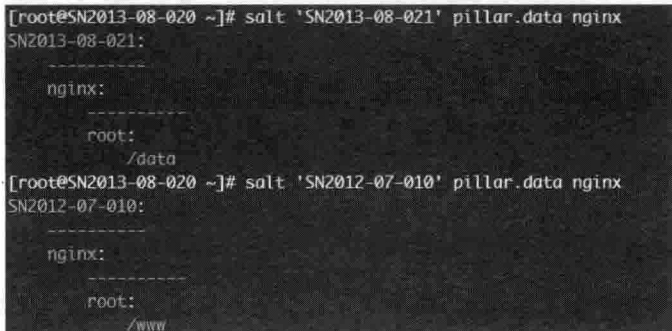
【/srv/pillar/web1server.sls】

```
nginx:
  root: /www
```

【/srv/pillar/web2server.sls】

```
nginx:
  root: /data
```

通过查看不同分组主机的 pillar 信息来验证配置结果，如图 10-18 所示。



```
[root@SN2013-08-020 ~]# salt 'SN2013-08-021' pillar.data nginx
SN2013-08-021:
-----
nginx:
-----
  root:
    /data
[root@SN2013-08-020 ~]# salt 'SN2012-07-010' pillar.data nginx
SN2012-07-010:
-----
nginx:
-----
  root:
    /www
```

图 10-18 不同分组的 pillar 差异信息

#### 10.7.4 配置 state

定义入口 top.sls:

【/srv/salt/top.sls】

```
base:
  '*':
    - nginx
```

下面定义 nginx 包、服务状态管理配置 sls，其中，salt://nginx/nginx.conf 为配置模板文件位置，-enable: True 检查服务是否在开机自启动服务队列中，如果不在则加上，等价于 chkconfig nginx on 命令“reload: True”，表示服务支持 reload 操作，不加则会默认执行 restart

操作。watch 一则检测 /etc/nginx/nginx.conf 是否发生变化，二则确保 nginx 已安装成功。

【 /srv/salt/nginx.sls 】

```
nginx:
  pkg:
    - installed
  file.managed:
    - source: salt://nginx/nginx.conf
    - name: /etc/nginx/nginx.conf
    - user: root
    - group: root
    - mode: 644
    - template: jinja

  service.running:
    - enable: True
    - reload: True
    - watch:
      - file: /etc/nginx/nginx.conf
      - pkg: nginx
```

定制 Nginx 配置文件 jinja 模板，各参数的引用规则如下：

- ❑ worker\_processes 参数采用 grains['num\_cpus'] 上报值（与设备 CPU 核数一致）；
- ❑ worker\_cpu\_affinity 分配多核 CPU，根据当前设备核数进行匹配，分别为 2、4、8、核或其他；
- ❑ worker\_rlimit\_nofile、worker\_connections 参数理论上为 grains['max\_open\_file']；
- ❑ root 参数为定制的 pillar['nginx']['root'] 值。

【 /srv/salt/nginx/nginx.conf 】

```
# For more information on configuration, see:
user      nginx;
worker_processes {{ grains['num_cpus'] }};
{% if grains['num_cpus'] == 2 %}
worker_cpu_affinity 01 10;
{% elif grains['num_cpus'] == 4 %}
worker_cpu_affinity 1000 0100 0010 0001;
{% elif grains['num_cpus'] >= 8 %}
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000 00100000
01000000 10000000;
{% else %}
worker_cpu_affinity 1000 0100 0010 0001;
{% endif %}
worker_rlimit_nofile {{ grains['max_open_file'] }};

error_log /var/log/nginx/error.log;
```

```

#error_log /var/log/nginx/error.log notice;
#error_log /var/log/nginx/error.log info;

pid /var/run/nginx.pid;

events {
    worker_connections {{ grains['max_open_file'] }};
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    access_log /var/log/nginx/access.log main;

    sendfile on;
    #tcp_nopush on;

    #keepalive_timeout 0;
    keepalive_timeout 65;

    #gzip on;

    # Load config files from the /etc/nginx/conf.d directory
    # The default server is in conf.d/default.conf
    #include /etc/nginx/conf.d/*.conf;
    server {
        listen 80 default_server;
        server_name _;

        #charset koi8-r;

        #access_log logs/host.access.log main;

        location / {
            root {{ pillar['nginx']['root'] }};
            index index.html index.htm;
        }

        error_page 404 /404.html;
        location = /404.html {
            root /usr/share/nginx/html;
        }

        # redirect server error pages to the static page /50x.html
        #
        error_page 500 502 503 504 /50x.html;
    }
}

```

```

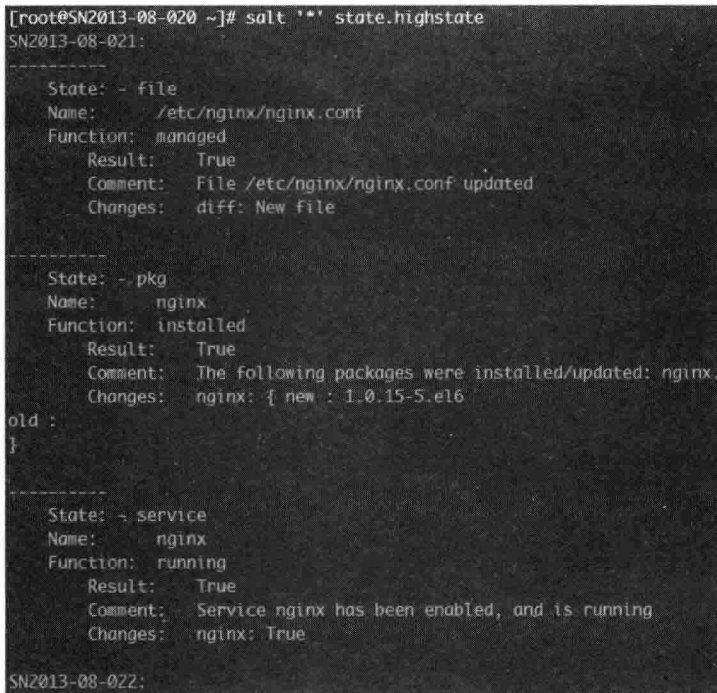
location = /50x.html {
    root /usr/share/nginx/html;
}

}

}

```

执行刷新 state 配置，结果如图 10-19 所示。



```

[root@SN2013-08-020 ~]# salt '*' state.highstate
SN2013-08-021:
-----
State: - file
Name: /etc/nginx/nginx.conf
Function: managed
Result: True
Comment: File /etc/nginx/nginx.conf updated
Changes: diff: New file

-----
State: - pkg
Name: nginx
Function: installed
Result: True
Comment: The following packages were installed/updated: nginx.
Changes: nginx: { new : 1.0.15-5.el6

old :
}

-----
State: - service
Name: nginx
Function: running
Result: True
Comment: Service nginx has been enabled, and is running
Changes: nginx: True

SN2013-08-022:

```

图 10-19 刷新 state 返回结果（部分截图）

### 10.7.5 校验结果

登录 web1group 组的一台服务器，检查 Nginx 的配置，尤其是变量部分的参数值，配置片段如下：

【 /etc/nginx/nginx.conf 】

```

user          nginx;
worker_processes 2;
worker_cpu_affinity 01 10;

```

```

worker_rlimit_nofile 65535;
error_log /var/log/nginx/error.log;
#error_log /var/log/nginx/error.log notice;
#error_log /var/log/nginx/error.log info;
pid /var/run/nginx.pid;
events {
    worker_connections 65535;
}
.....
location / {
    root /www;
    index index.html index.htm;
}

```

再登录 web2group 组的一台服务器，检查 Nginx 的配置，对比 web1group 组的服务器差异化，包括不同硬件配置、内核参数等，配置片段如下：

【 /etc/nginx/nginx.conf 】

```

user nginx;
worker_processes 4;
worker_cpu_affinity 1000 0100 0010 0001;
worker_rlimit_nofile 65535;
error_log /var/log/nginx/error.log;
#error_log /var/log/nginx/error.log notice;
#error_log /var/log/nginx/error.log info;
pid /var/run/nginx.pid;
events {
    worker_connections 65535;
}
.....
location / {
    root /data;
    index index.html index.htm;
}

```

至此，一个模拟生产环境 Web 服务集群的配置集中化管理平台已经搭建完成，大家可以利用这个思路扩展到其他功能平台。



参考  
提示

10.1 至 10.6 节的 Saltstack 介绍可参考官网文档 <http://docs.saltstack.com/en/latest/>。

## 统一网络控制器 Func 详解

Func (Fedora Unified Network Controller) 是由红帽子公司以 Fedora 平台构建的统一网络控制器,是为解决集群管理、监控问题而设计开发的系统管理基础框架,官网地址为 <https://fedorahosted.org/func>。它是一个能有效简化多服务器系统管理工作的工具,它易于学习、使用和扩展,功能强大,只需要极少的配置和维护操作。Func 分为 master 和 slave 两部分, master 为主控端, slave 为被控端。Func 具有以下特点。

- ❑ 支持在主机上管理任意多台服务器,或任意多个服务器组。
- ❑ 支持命令行方式发送远程命令或者远程获取数据。
- ❑ Func 通信基于 XMLRPC 和 SSL 标准协议,具有模块化的可扩展的特点。与 Saltstack 认证方式一致。
- ❑ 可以通过 Kickstart 预安装 Func 到系统中,自动注册到主控服务器端。
- ❑ 任何人都可以通过 Func 提供的 Python API 轻松编写自己的模块,以实现具体功能扩展。而且任何 Func 命令行能完成的工作,都能通过 API 编程实现。
- ❑ 提供封装大量通用的服务器管理命令模块。
- ❑ Func 平台没有与数据库关联,不需要复杂的安装与配置,服务器间安全证书的分发都是自动完成的。

Func 与 Saltstack 在主、被控端建立信任机制是一样的,都采用了证书+签名的方式。相比 Saltstack 或 Ansible, Func 在文件配置、状态管理方面还是空白,但在远程命令执行、API 支持、配置简单等方面还是能体现出其优势,适合中小型服务集群的远程命令执行、文件分发的任务,同时 API 支持跨语言,可以与现有运营平台打通,实现交互式更强、体验更

好的自动化运营平台。

## 11.1 Func 的安装

Func 需要在主控端、被控端部署环境，建议读者采用 yum 的方式实现部署。目前 Func 最新版本为 0.28，由 func、certmaster、pyOpenSSL 三个组件组成。下面详细讲解 Func 的安装步骤。

### 11.1.1 业务环境说明

为了方便读者理解，笔者通过虚拟化环境部署功能服务器来进行演示，操作系统版本为 CentOS release 6.4，自带 Python 2.6.6。相关服务器信息如表 11-1 所示。

表 11-1 业务环境表说明

角色	主机名	IP
Master	SN2013-08-020	192.168.1.20
minion	SN2013-08-021	192.168.1.21
minion	SN2013-08-022	192.168.1.22

### 11.1.2 安装 Func

#### 1. 主控端服务器安装

主控端部署在主机名为 SN2013-08-020 的设备上，通过 yum 方式安装，如下：

```
# yum install func -y
# /sbin/chkconfig --level 345 certmaster on
```

在设备通信上 Func 要求使用主机名来识别，在没有内部域名解析服务的情况下，可通过配置主机 hosts 来解决主机名的问题。主控端 hosts 配置如下：

【 /etc/hosts 】

```
192.168.1.21    SN2013-08-021
192.168.1.22    SN2013-08-022
192.168.1.20    func.master.server.com
```

修改 /etc/certmaster/minion.conf 的 certmaster 参数，指向证书服务器，即主控端服务器，func 命令用到此配置，如：

【 /etc/certmaster/minion.conf 】

```
# configuration for minions
[main]
certmaster = func.master.server.com
certmaster_port = 51235
log_level = DEBUG
cert_dir = /etc/pki/certmaster
```

启动证书服务：

```
# /sbin/service certmaster start
```

配置 iptables，开通 192.168.1.0/24 网段访问证书服务 51235 (certmaster 服务) 端口。

```
# iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 51235 -j ACCEPT
```

至此，主控端配置完毕。

## 2. 被控端服务器安装

被控端部署在主机名为 SN2013-08-021、SN2013-08-022 的设备上，同样通过 yum 方式安装，如下：

```
# yum install func -y
# /sbin/chkconfig --level 345 funcd on
```

配置 hosts 信息：

```
192.168.1.20    func.master.server.com
```

修改 /etc/certmaster/minion.conf 的 certmaster 参数，以便指向证书服务器发出签名请求，建立信任关系，如：

【 /etc/certmaster/minion.conf 】

```
# configuration for minions
[main]
certmaster = func.master.server.com
certmaster_port = 51235
log_level = DEBUG
cert_dir = /etc/pki/certmaster
```

修改 /etc/func/minion.conf 的 minion\_name 参数，作为被控主机的唯一标识，一般使用主机名，以 SN2013-08-021 主机为例，配置如下：

```
# configuration for minions
[main]
log_level = INFO
acl_dir = /etc/func/minion-acl.d

listen_addr =
listen_port = 51234
minion_name = SN2013-08-021
method_log_dir = /var/log/func/methods/
```

启动 func 服务：

```
# /sbin/service funcd start
```

配置 iptables，开通 192.168.1.20 主控端主机访问本机 51234（func 服务）端口。

```
# iptables -I INPUT -s 192.168.1.20 -p tcp --dport 51234 -j ACCEPT
```

至此，被控端配置完毕。

### 3. 证书签名

在主控端运行 `certmaster-ca --list` 获取当前请求证书签名的主机清单，如：

```
# certmaster-ca --list
sn2013-08-021
sn2013-08-022
```

证书签名通过 `certmaster-ca --sign hostname` 命令来完成，如：

```
# certmaster-ca --sign sn2013-08-021
```

当然，也可以结合 `--list`、`--sign` 参数实现一键完成所有主机的签名操作，如：

```
# certmaster-ca --sign `certmaster-ca --list`
```

Func 也提供了类似 Saltstack 自动签名的机制，通过修改 `/etc/certmaster/certmaster.conf` 的参数 `autosign = no` 为 `autosign = yes` 即可。

使用 `func "*" list_minions` 查看已经完成签名的主机名，如：

```
# func '*' list_minions
sn2013-08-021
sn2013-08-022
```

删除（注销）签名主机使用 `certmaster-ca -c hostname`，如：

```
# certmaster-ca -c sn2013-08-021
```

校验安装、任务签名是否正确，通过 `func "*" ping` 命令来测试，如图 11-1 所示。

```
[root@SN2013-08-020 func]# func "*" ping
[ ok ... ] sn2013-08-022
[ ok ... ] sn2013-08-021
```

图 11-1 测试认证主机的连通性



对已注销的被控服务器，要重新注册，先删除被控主机端 `/etc/pki/certmaster/` 下的证书文件，再运 3 行 `certmaster-request` 进行证书请求，具体操作步骤如下：

```
# rm -rf /etc/pki/certmaster/ 主机名 .*
# /usr/bin/certmaster-request
```

## 11.2 Func 常用模块及 API

Func 提供了非常丰富的功能模块，包括 `CommandModule`（执行命令）、`CopyFileModule`（拷贝文件）、`CpuModule`（CPU 信息）、`DiskModule`（磁盘信息）、`FileTrackerModule`（文件跟踪）、`IPtablesModule`（iptables 管理）、`MountModule`（Mount 挂载）、`NagiosServerModule`（Nagios 管理）、`NetworkTest`（网络测试）、`ProcessModule`（进程管理）、`SysctlModule`（sysctl 管理）、`SNMPModule`（SNMP 信息），等等，更多模块介绍见官网模块介绍：<https://fedorahosted.org/func/wiki/ModulesList>。命令行调用模块格式：

```
func <目标主机> call <module_name (模块名)> <method_name (方法名)> <module_
args (模块参数)>
```

模块命令行执行结果都以 Python 的元组字符串返回（API 以字典形式返回），这对后续进行结果集的解析工作非常有利，例如，远程运行“`df -m`”命令的运行结果如图 11-2 所示。

```
[root@SN2013-08-020 ~]# func "SN2013-08-022" call command run "df -m"
('sn2013-08-022',
 [0,
  'Filesystem          1M-blocks      Used Available Use% Mounted on\n/dev/sda1          14765      2730
11286 20% /ntmpfs          242          0      242  0% /dev/shm\n/dev/sda3          85
159      4004  4% /data\n',
  ''])
```

图 11-2 返回主机内存使用信息

在所有模块中，`CommandModule` 模块最常用，可以在目标被控主机执行任意命令。笔者建议使用 API 方式对应用场景的逻辑进行封装，将权限放到一个预先定制好的方框中，实

现收敛操作。下面对 Func 常用的模块一一进行讲解。

### 11.2.1 选择目标主机

Func 选择目标主机操作对象支持 “\*” 与 “?” 方式匹配，其中 “\*” 代表任意多个字符，“?” 代表单个任意字符，例如：

```
# func "SN2013-*-02?" call command run "uptime"
```

“SN2013-\*-02?” 在本文环境中将匹配到 SN2013-08-021、SN2013-08-022 两台主机，可以根据实际应用场景随意组合。例如，我们定义的多台 Web 业务服务器主机名分别为：web1、web2、web3、…、webn.webapp.com，要查看所有 Web 应用的 uptime 信息可以运行：

```
# func "web*.webapp.com" call command run "uptime"
```

多个目标主机名使用分号分隔，如：

```
# func "web.example.org;mailserver.example.org;db.example.org" call command run  
"df -m"
```

### 11.2.2 常用模块详解

#### 1. 执行命令模块

##### (1) 功能

CommandModule 实现 Linux 远程命令调用执行。

##### (2) 命令行模式

```
# func "*" call command run "ulimit -a"  
# func "SN2013-08-022" call command run "free -m"
```

##### (3) API 模式

```
import func.overlord.client as func  
client = func.Client("SN2013-08-022")  
print client.command.run("free -m")
```

#### 2. 文件拷贝模块

##### (1) 功能

CopyFileModule 实现主控端向目标主机拷贝文件，类似于 scp 的功能。

## (2) 命令行模式

```
# func "SN2013-08-022" copyfile -f /etc/sysctl.conf --remotepath /etc/sysctl.conf
```

## (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
client.local.copyfile.send("/etc/sysctl.conf", "/tmp/sysctl.conf")
```

# 3. CPU 信息模块

## (1) 功能

CpuModule 获取远程主机 CPU 信息，支持按时间（秒）采样平均值，如下面示例中的参数“10”。

## (2) 命令行模式

```
# func "SN2013-08-022" call cpu usage
# func "SN2013-08-022" call cpu usage 10
```

## (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
print client.cpu.usage(10)
```

# 4. 磁盘信息模块

## (1) 功能

DiskModule 实现获取远程主机的磁盘分区信息，参数为分区标签，如 /data 分区。

## (2) 命令行模式

```
# func "SN2013-08-022" call disk usage
# func "SN2013-08-022" call disk usage /data
```

## (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
print client.disk.usage("/dev/sda3")
```

## 5. 拷贝远程文件模块

### (1) 功能

GetFileModule 实现拉取远程 Linux 主机指定文件到主控端目录, 不支持命令行模式。

### (2) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
client.local.getfile.get("/etc/sysctl.conf", "/tmp/")
```

## 6. iptables 管理模块

### (1) 功能

IPTablesModule 实现远程主机 iptables 配置。

### (2) 命令行模式

```
# func "SN2013-08-022" call iptables.port drop_to 53 192.168.0.0/24 udp src
# func "SN2013-08-022" call iptables drop_from 192.168.0.10
```

### (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
client.iptables.port.drop_to(8080, "192.168.0.10", "tcp", "dst")
```

## 7. 系统硬件信息模块

### (1) 功能

HardwareModule 返回远程主机系统硬件信息。

### (2) 命令行模式

```
# func "SN2013-08-022" call hardware info
# func "SN2013-08-022" call hardware hal_info
```

### (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
print client.hardware.info(with_devices=True)
print client.hardware.hal_info()
```

## 8. 系统 Mount 管理模块

### (1) 功能

MountModule 实现远程主机 Linux 系统挂载、卸载分区管理。

### (2) 命令行模式

```
# func "SN2013-08-022" call mount list
# func "SN2013-08-022" call mount mount /dev/sda3 /data
# func "SN2013-08-022" call mount umount "/data"
```

### (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
print client.mount.list()
print client.mount.umount("/data")
print client.mount.mount("/dev/sda3", "/data")
```

## 9. 系统进程管理模块

### (1) 功能

ProcessModule 实现远程 Linux 主机进程管理。

### (2) 命令行模式

```
# func "SN2013-08-022" call process info "aux"
# func "SN2013-08-022" call process pkill nginx -9
# func "SN2013-08-022" call process kill nginx SIGHUP
```

### (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
print client.process.info("aux")
print client.process.pkill("nginx", "-9")
print client.process.kill("nginx", "SIGHUP")
```

## 10. 系统服务管理模块

### (1) 功能

ServiceModule 实现远程 Linux 主机系统服务管理。

### (2) 命令行模式

```
# func "SN2013-08-022" call service start nginx
```

### (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
print client.service.start("nginx")
```

## 11. 系统内核参数管理模块

### (1) 功能

SysctlModule 实现远程 Linux 主机系统内核参数管理。

### (2) 命令行模式

```
# func "SN2013-08-022" call sysctl list
# func "SN2013-08-022" call sysctl get net.nf_conntrack_max
# func "SN2013-08-022" call sysctl set net.nf_conntrack_max 15449
```

### (3) API 模式

```
import func.overlord.client as func
client = func.Client("SN2013-08-022")
print client.sysctl.list()
print client.sysctl.get('net.ipv4.icmp_echo_ignore_broadcasts')
print client.sysctl.set('net.ipv4.tcp_syncookies', 1)
```

**func 命令功能参数举例：**

1) 查看所有主机 uptime，开启 5 个线程异步运行，超时时间为 3 秒，命令如下：

```
# func -t 3 "*" call --forks="5" --async command run "/usr/bin/uptime"
```

2) 格式化输出结果，默认格式为 Python 的元组，分别添加 --json 或 --xml 来输出 JSON 及 XML 格式，命令如下：

```
# func -t 3 "*" call --forks="5" --json --async command run "/usr/bin/uptime"
```

## 11.3 自定义 Func 模块

Func 自带的模块已经非常丰富，但在日常系统运维当中，尤其是面对大规模的服务器集群、不同类别的业务平台，此时 Func 自带的模块或许已经不能满足我们的需求，所以有必要通过自定模块来填补这块的不足。本节介绍一个简单的 Func 自定义模块的，通过采用 Func 自带的建模块工具 func-create-module 来现实。

### (1) 自定义模块步骤

如图 11-3 所示, 自定义模块分为四个步骤进行, 第一步生成模块, 即通过 `func-create-module` 命令创建模块初始模板; 第二步编写逻辑, 即填充我们的业务功能逻辑, 生成模块; 第三步分发模块, 将编写完成的模块分发到所有被控主机; 第四步执行已经分发完成的模块, 调用方法与 Func 自带模块无差异。详细过程见图 11-3。



图 11-3 自定义模块发布流程

### (2) 生成模块

切换到 Func 安装包 `minion` 模块存储目录。笔者使用的是系统自带的 Python 2.6, 具体路径为 `/usr/lib/python2.6/site-packages/func/minion/modules`。

```
# cd /usr/lib/python2.6/site-packages/func/minion/modules
```

运行创建模块命令 `func-create-module`, 根据图 11-14 填写相关信息。

```
[root@SN2013-08-020 modules]# func-create-module
Module Name: MyModule
Description: My module for func.
Author: liutiansi
Email: liutiansi@gmail.com

Leave blank to finish.
Method: echo
Method:
Your module is ready to be hacked on. Wrote out to mymodule.py.
```

图 11-4 创建模块时填写的信息

最终生成了一个初始化的模块代码文件 `mymodule.py`:

【`/usr/lib/python2.6/site-packages/func/minion/modules/mymodule.py`】

```
#
# Copyright 2014
# liutiansi <liutiansi@gmail.com>
#
# This software may be freely redistributed under the terms of the GNU
# general public license.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

```
import func_module

class Mymodule(func_module.FuncModule):
    # Update these if need be.
    version = "0.0.1"
    api_version = "0.0.1"
    description = "My module for func."
    def echo(self):
        """
        TODO: Document me ...
        """
        pass
```

### (3) 编写逻辑

这一步只需在上述模块基础上做修改即可，如模块实现一个根据指定的条数返回最新系统日志 (/var/log/messages) 信息，修改后的代码如下：

**【 /usr/lib/python2.6/site-packages/func/minion/modules/mymodule.py 】**

```
#
# Copyright 2010
# liutiansi <liutiansi@gmail.com>
#
# This software may be freely redistributed under the terms of the GNU
# general public license.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

import func_module
from func.minion import sub_process

class Mymodule(func_module.FuncModule):

    # Update these if need be.
    version = "0.0.1"
    api_version = "0.0.1"
    description = "My module for func."

    def echo(self, vcount):
        """
        TODO: response system messages info
        """
        command="/usr/bin/tail -n "+str(vcount)+" /var/log/messages"
        cmdref = sub_process.Popen(command, stdout=sub_process.PIPE,
                                   stderr=sub_process.PIPE, shell=True,
                                   close_fds=True)
        data = cmdref.communicate()
        return (cmdref.returncode, data[0], data[1])
```

#### (4) 分发模块

首先编写分发模块的功能,使用Func的copyfile模块来实现,原理比较简单,即读取主控端func minion包下的模块文件(参数传入),通过Func的copyfile模块同步到目标主机的同路径下。一次编写可持续使用,源码如下:

```
【 /home/test/func/RsyncModule.py 】
```

```
#!/usr/bin/python
import sys
import func.overlord.client as fc
import xmlrpclib

module = sys.argv[1]
pythonmodulepath="/usr/lib/python2.6/site-packages/func/minion/modules/"
client = fc.Client("")
fb = file(pythonmodulepath+module, "r").read()
data = xmlrpclib.Binary(fb)

# 分发模块
print client.copyfile.copyfile(pythonmodulepath+ module,data)

# 重启 Func 服务
print client.command.run("/etc/init.d/funcd restart")
```

分发模块的运行结果如图 11-5 所示。

```
[root@SN2013-08-020 func]# cd /home/test/func/
[root@SN2013-08-020 func]# cp /usr/lib/python2.6/site-packages/func/minion/modules/mymodule.py /home/test/func/
[root@SN2013-08-020 func]# python RsyncModule.py mymodule.py
{'sn2013-08-022': 0, 'sn2013-08-021': 0}
{'sn2013-08-022': [0, 'Stopping func daemon: [ OK ]\nStarting func daemon: [ OK ]\n', ''], 'sn2013-08-021': [0, 'Stopping func daemon: [ OK ]\nStarting func daemon: [ OK ]\n', '']}
[root@SN2013-08-020 func]#
```

图 11-5 模块分发结果

检查被控主机 /usr/lib/python2.6/site-packages/func/minion/modules 目录是否多了一个 mymodule.py 文件,是则说明模块已经成功分发。

#### (5) 执行模块

最后,执行模块及返回结果见图 11-6。

```
[root@SN2013-08-020 func]# func "SN2013-08-021" call mymodule echo 5
{'sn2013-08-021': [0,
'Jan 1 07:42:25 SN2013-08-021 ntpd[1040]: synchronized to 210.72.145.44, stratum 1\nJan 1 07:46:19 SN2013-08-021 ntpd[1040]: time reset +233.756480 s\nJan 1 07:46:19 SN2013-08-021 ntpd[1040]: kernel time sync status change 2001\nJan 4 03:48:44 SN2013-08-021 ntpd[1040]: synchronized to 210.72.145.44, stratum 1\nJan 4 03:48:44 SN2013-08-021 ntpd[1040]: no servers reachable\n',
''}]
```

图 11-6 执行模块结果

正常返回了 5 条 `/var/log/messages` 信息，完成了自定义模块的全过程。

## 11.4 非 Python API 接口支持

Func 通过非 Python API 实现远程调用，目的是为第三方工具提供调用及返回接口。Func 使用 `func-transmit` 命令来实现，支持 YAML 与 JSON 格式，实现了跨应用平台、语言、工具等，比如通过 Java 或 C 生成 JSON 格式的接口定义，通过 `fun-transmit` 命令进行调用，使用上非常简单，扩展性也非常强。

定义一个 `command` 模块的远程执行，分别采用 YAML 及 JSON 格式进行定义，如下：

【 `/home/test/func/run.yaml` 】

```
clients: "*"
async: False
nforks: 1
module: command
method: run
parameters: "/bin/echo Hello World"
```

【 `/home/test/func/run.json` 】

```
{
  "clients": "*",
  "async": "False",
  "nforks": 1,
  "module": "command",
  "method": "run",
  "parameters": "/bin/echo Hello World"
}
```

各参数详细说明如下。

- ❑ `clients`，目标主机，`"*"` 代表所有被控主机；
- ❑ `async`，是否异步，是一个布尔值，`True` 为使用异步，`False` 则不使用；
- ❑ `nforks`，启用的线程数，用数字表示；
- ❑ `module`，模块名称，如 `command`、`copyfile`、`process` 等；
- ❑ `method`，方法名称，如 `command` 模块下的 `run` 方法；
- ❑ `parameters`，参数，如 `"/usr/bin/tail -100 /var/log/messages"`。

通过 `func-transmit` 命令调用不同接口配置，将返回不同的格式串，如图 11-7 和图 11-8 所示。

```
[root@SN2013-08-020 func]# func-transmit --yaml < run.yaml
---
sn2013-08-021:
- 0
- |
  Hello World
- ''
sn2013-08-022:
- 0
- |
  Hello World
- ''
```

图 11-7 返回标准的 YAML 格式

```
[root@SN2013-08-020 func]# func-transmit --json < run.json
{"sn2013-08-022": [0, "Hello World\n", ""], "sn2013-08-021": [0, "Hello World\n", ""]}
```

图 11-8 返回标准的 JSON 格式

返回的两种格式都可以被绝大部分语言所解析，方便后续处理。

## 11.5 Func 的 Facts 支持

Facts 是一个非常有用的组件，其功能类似于 Saltstack 的 grains、Ansible 的 Facts，实现获取远程主机的系统信息，以便在对目标主机操作时作为条件进行过滤，产生差异。Func 的 Facts 支持通过 API 来扩展用户自己的属性。Facts 由两部分组成，一为模块（module），另为方法（method），可通过 `list_fact_modules`、`list_fact_methods` 方法来查看当前支持的模块与方法的清单，如图 11-9 所示。

```
[root@SN2013-08-020 func]# func "" call fact list_fact_modules
{'sn2013-08-021': ['hardware', 'fact_module'],
 'sn2013-08-022': ['hardware', 'fact_module']}
[root@SN2013-08-020 func]#
[root@SN2013-08-020 func]# func "" call fact list_fact_methods
{'sn2013-08-021': ['hardware.cpu_model',
                   'kernel',
                   'cpumodel',
                   'hardware.kernel_version',
                   'cpuvendor',
                   'hardware.run_level',
                   'hardware.cpu_vendor',
                   'hardware.os_name',
                   'runlevel',
                   'os'],
 'sn2013-08-022': ['hardware.cpu_model',
                   'kernel',
                   'cpumodel',
                   'hardware.kernel_version',
                   'cpuvendor',
                   'hardware.run_level',
                   'hardware.cpu_vendor',
                   'hardware.os_name',
                   'runlevel',
                   'os']}
```

图 11-9 查看主机支持模块及方法

在使用 Facts 时，我们关注它的方法（func "\*" call fact list\_fact\_methods 显示的清单）即可，可通过命令行调用 Facts 的 call\_fact 方法查看所有主机的操作系统信息，具体见图 11-10。

```
[root@SN2013-08-020 func]# func "*" call fact call_fact "os"
{'sn2013-08-021': 'CentOS release 6.4 (Final)',
 'sn2013-08-022': 'CentOS release 6.4 (Final)'}
```

图 11-10 查看主机操作系统信息

Fact 支持 and 与 or 作为条件表达式连接操作符，下面详细介绍。

#### （1）and 表达式 --filter

语法：

```
--filter "keyword[operator]value,keyword2[operator]value2"
--filter "value in keyword,value ini keyword"
```

示例：所有满足内核（kernel）版本大于或等于 2.6，并且操作系统信息包含 CentOS 的目标主机运行 uptime 命令，如图 11-11 所示。

```
[root@SN2013-08-020 func]# func "*" call --filter "kernel>=2.6,CentOS in os" command run "uptime"
{'sn2013-08-022':
 [0,
  ' 04:29:41 up 1 day, 21:27, 1 user, load average: 0.00, 0.00, 0.00\n',
  '']}
{'sn2013-08-021':
 [0,
  ' 11:46:32 up 2 days, 9:36, 1 user, load average: 0.00, 0.00, 0.00\n',
  '']}
```

图 11-11 根据 fact 条件 (and) 过滤主机

#### （2）or 表达式 --filteror

语法：

```
--filteror "keyword[operator]value,keyword2[operator]value2"
--filteror "value in keyword,value ini keyword"
```

示例：所有满足内核（kernel）版本大于或等于 2.6，或者运行级别等于 5 的目标主机运行 df -m 命令，如图 11-12 所示。

```
[root@SN2013-08-020 func]# func "" call --filteror "kernel<=2.6,runlevel=5" command run "df -m"
C'sn2013-08-022',
[0,
  'Filesystem          1M-blocks      Used Available Use% Mounted on\n/dev/sda1          14765
  2730      11286  20% /\ntmpfs              242         0      242   0% /dev/shm\n/dev/sda3
  4385         159  4004   4% /data\n',
  '']
C'sn2013-08-021',
[0,
  'Filesystem          1M-blocks      Used Available Use% Mounted on\n/dev/sda1          14765
  3091      10924  23% /\ntmpfs              242         0      242   0% /dev/shm\n/dev/sda3
  4385         160  4003   4% /data\n',
  '']
```

图 11-12 根据 fact 条件 (or) 过滤主机



11.1 节 ~ 11.5 节关于 Func 的介绍参考官网文档 <https://fedorahosted.org/func/>。

# Python 大数据应用详解

随着云时代的到来，大数据（big data）也越来越受大家的关注，比如互联网行业日常生成的运营、用户行为数据，随着时间及访问量的增长这一规模日益庞大，单位可达到日 TB 或 PB 级别。如何在如此庞大的数据中挖掘出对我们有用的信息？目前业界主流存储与分析平台是以 Hadoop 为主的开源生态圈，MapReduce 作为 Hadoop 的数据集的并行运算模型，除了提供 Java 编写 MapReduce 任务外，还兼容了 Streaming 方式，我们可以使用任意脚本语言来编写 MapReduce 任务，优点是开发简单且灵活。本章详细介绍如何使用 Python 语言来实现大数据应用，将分别通过原生 Python 与框架（Framework）方式进行说明。



因为 Hadoop 不作为本章的主体内容，所以将不对其架构、子项目、优化等进行说明。

## 12.1 环境说明

为了方便读者理解，笔者通过虚拟化环境部署了 Hadoop 平台来进行演示，操作系统版本为 CentOS release 6.4，以及 Python 2.6.6、hadoop-1.2.1、jdk1.6.0\_45、mrjob-0.4.2 等。相关服务器信息如表 12-1 所示。

表 12-1 环境说明表

角色	主机名	IP	功能	存储分区
Master	SN2013-08-020	192.168.1.20	NameNode   Secondarynamenode   JobTracker	/data
Slave	SN2012-07-010	192.168.1.21	DataNode   TaskTracker	/data
Slave	SN2012-07-011	192.168.1.22	DataNode   TaskTracker	/data

## 12.2 Hadoop 部署

由于部署 Hadoop 需要 Master 访问所有 Slave 主机实现无密码登录，即配置账号公钥认证，具体参考 9.2.5 节关于配置 Linux 主机 SSH 无密码访问的介绍，本节将不再陈述。

### (1) 安装

SSH 登录 Master 主机，这里使用 root 账号进行相关演示。安装 JDK 环境：

```
# mkdir -p /usr/java/ && cd /usr/java
# wget http://uni-smr.ac.ru/archive/dev/java/SDKs/sun/j2se/6/jdk-6u45-
linux-x64.bin
# chmod +x jdk-6u45-linux-x64.bin
# ./jdk-6u45-linux-x64.bin

# vi /etc/profile (配置 Java 环境变量，追加以下内容)
export JAVA_HOME=/usr/java/jdk1.6.0_45
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=.:$JAVA_HOME/jre/lib:$JAVA_HOME/lib:$JAVA_HOME/lib/tools.jar

# cd /etc (使环境变量生效)
# . profile
```

安装 Hadoop，版本为 1.2.1，安装路径为 /usr/local。

```
# cd /usr/local
# wget http://mirrors.cnnic.cn/apache/hadoop/common/hadoop-1.2.1/hadoop-
1.2.1.tar.gz
# tar -zxvf hadoop-1.2.1.tar.gz
# cd /usr/local/hadoop-1.2.1/conf
```

修改目录 (/usr/local/hadoop-1.2.1/conf) 中的四个 Hadoop 核心配置文件 hadoop-env.sh、core-site.xml、hdfs-site.xml、mapred-site.xml，具体内容如下：

□ hadoop-env.sh，Hadoop 环境变量配置文件，指定 JAVA\_HOME。

```
export JAVA_HOME=/usr/java/jdk1.6.0_45
```

□ core-site.xml，Hadoop core 的配置项，主要针对 Common 组件的属性配置。由于默认的 hadoop.tmp.dir 的路径为 /tmp/hadoop-\${user.name}，笔者的 Linux 系统的 /tmp 文件系统的类型是 Hadoop 不支持的，会报 “File /tmp/<user>/input/conf/slaves could only be replicated to 0 nodes, instead of 1” 异常，因此手工修改 hadoop.tmp.dir 指向 /data/tmp/hadoop-\${user.name}，作为 Hadoop 用户的临时存储目录，配置如下：

```
<configuration>
<property>
```

```

    <name>hadoop.tmp.dir</name>
    <value>/data/tmp/hadoop-${user.name}</value>
  </property>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://192.168.1.20:9000</value> //master 主机 IP:9000 端口
  </property>
</configuration>

```

□ **hdfs-site.xml**, Hadoop 的 HDFS 组件的配置项, 包括 Namenode、Secondarynamenode 和 Datanode 等, 配置如下:

```

<configuration>
  <property>
    <name>dfs.name.dir</name>
    <value>/data/hdfs/name</value> //Namenode 持久存储名字空间、事务日志路径
  </property>

  <property>
    <name>dfs.data.dir</name>
    <value>/data/hdfs/data</value> //Datanode 数据存储路径
  </property>

  <property>
    <name>dfs.datanode.max.xcievers</name>
    <value>4096</value> //Datanode 所允许同时执行的发送和接受任务数量, 默认为 256
  </property>

  <property>
    <name>dfs.replication</name>
    <value>2</value> // 数据备份的个数, 默认为 3
  </property>
</configuration>

```

□ **mapred-site.xml**, 配置 map-reduce 组件的属性, 包括 jobtracker 和 tasktracker, 配置如下:

```

<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>192.168.1.20:9001</value>
  </property>
</configuration>

```

□ **masters**, 配置 Secondarynamenode 项, 环境使用主设备 192.168.1.20 同时承担 Secondarynamenode 的角色, 生产环境要求使用独立服务器, 起到 HDFS 文件系统元数据 (metadata) 信息的备份作用, 当 NameNode 发生故障后可以快速还原数据, 配置内容如下:

```
192.168.1.20
```

❑ slaves，配置所有 Slave 主机信息，填写 IP 地址即可。本示例中 Slave 的信息如下：

```
192.168.1.21
```

```
192.168.1.22
```

接下来，从主节点（Master）复制 jdk 及 Hadoop 环境到所有 Slave，目标路径要与 Master 保持一致，切记！执行以下命令进行复制：

```
# ssh root@192.168.1.21 '[ -d /usr/java ] || mkdir -p /usr/java ]'
# ssh root@192.168.1.22 '[ -d /usr/java ] || mkdir -p /usr/java ]'
# scp -r /usr/java/jdk1.6.0_45 root@192.168.1.21:/usr/java
# scp -r /usr/java/jdk1.6.0_45 root@192.168.1.22:/usr/java

# scp -r /usr/local/hadoop-1.2.1 root@192.168.1.21:/usr/local
# scp -r /usr/local/hadoop-1.2.1 root@192.168.1.22:/usr/local
```

Hadoop 部分功能是通过主机名来寻址的，因此需要配置主机名 hosts 信息（生产环境建议直接搭建内网 DNS 服务），保证 Hadoop 环境所有主机的 /etc/hosts 文件配置如下：

```
192.168.1.20    SN2013-08-020
192.168.1.21    SN2013-08-021
192.168.1.22    SN2013-08-022
```

管理员通过浏览器查看 datanode 信息，需要配置本地 hosts，如 Windows 7 系统 hosts 文件路径为 C:\Windows\System32\drivers\etc，添加所有 datanode 主机信息，如下：

```
192.168.1.21    SN2013-08-021
192.168.1.22    SN2013-08-022
```

如设备启用了 iptables 防火墙，需要对主节点（Master）及 Slave 主机添加以下规则：

Master:

```
iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 50030 -j ACCEPT
iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 50070 -j ACCEPT
iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 9000 -j ACCEPT
iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 9001 -j ACCEPT
```

Slaves:

```
iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 50075 -j ACCEPT
iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 50060 -j ACCEPT
iptables -I INPUT -s 192.168.1.20 -p tcp --dport 50010 -j ACCEPT
```

配置完成后在主节点（Master）上格式化文件系统的 namenode，执行：

```
# cd /usr/local/hadoop-1.2.1
```

```
# bin/hadoop namenode -format
```

最后，在主节点（Master）上执行启动命令，如下：

```
# bin/start-all.sh
```

## （2）检验安装结果

Hadoop 官方提供的一个测试 MapReduce 的示例，执行：

```
# bin/hadoop jar hadoop-examples-1.2.1.jar pi 10 100
```

如果返回如图 12-1 所示结果，则说明配置成功。

```
[root@SN2013-08-020 hadoop-1.2.1]# bin/hadoop jar hadoop-examples-1.2.1.jar pi 10 100
Number of Maps = 10
Samples per Map = 100
Wrote input for Map #0
Wrote input for Map #1
Wrote input for Map #2
Wrote input for Map #3
Wrote input for Map #4
Wrote input for Map #5
Wrote input for Map #6
Wrote input for Map #7
Wrote input for Map #8
Wrote input for Map #9
Starting Job
14/08/10 19:51:55 INFO mapred.FileInputFormat: Total input paths to process : 10
14/08/10 19:51:57 INFO mapred.JobClient: Running job: job_201408101951_0001
14/08/10 19:51:58 INFO mapred.JobClient: map 0% reduce 0%
14/08/10 19:54:13 INFO mapred.JobClient: map 20% reduce 0%
14/08/10 19:54:47 INFO mapred.JobClient: map 30% reduce 0%
14/08/10 19:54:54 INFO mapred.JobClient: map 50% reduce 0%
14/08/10 19:54:56 INFO mapred.JobClient: map 60% reduce 0%
14/08/10 19:55:12 INFO mapred.JobClient: map 60% reduce 20%
14/08/10 19:55:26 INFO mapred.JobClient: map 80% reduce 20%
14/08/10 19:55:29 INFO mapred.JobClient: map 90% reduce 20%
14/08/10 19:55:30 INFO mapred.JobClient: map 100% reduce 20%
14/08/10 19:55:31 INFO mapred.JobClient: map 100% reduce 26%
14/08/10 19:55:40 INFO mapred.JobClient: map 100% reduce 100%
```

图 12-1 计算 pi 的测试结果（部分截图）

访问 Hadoop 提供的管理页面，Map/Reduce 管理地址：<http://192.168.1.20:50030/>，如图 12-2 所示。

SN2013-08-020 Hadoop Map/Reduce Administration									
<b>State:</b> RUNNING <b>Started:</b> Fri Aug 22 22:15:42 CST 2014 <b>Version:</b> 1.2.1, r1503152 <b>Compiled:</b> Mon Jul 22 15:23:09 PDT 2013 by mattf <b>Identifier:</b> 201408222215 <b>SafeMode:</b> OFF									
Cluster Summary (Heap Size is 7.31 MB/966.69 MB)									
Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity
0	0	0	2	0	0	0	0	4	4

图 12-2 Map/Reduce 管理界面 (部分截图)

HDFS 存储管理地址: <http://192.168.1.20:50070/>, 如图 12-3 所示。

NameNode 'SN2013-08-020:9000'	
<b>Started:</b>	Fri Aug 22 22:14:01 CST 2014
<b>Version:</b>	1.2.1, r1503152
<b>Compiled:</b>	Mon Jul 22 15:23:09 PDT 2013 by mattf
<b>Upgrades:</b>	There are no upgrades in progress.
<a href="#">Browse the filesystem</a> <a href="#">Namenode Logs</a>	
Cluster Summary	
31 files and directories, 9 blocks = 40 total. Heap Size is 15.38 MB / 966.69 MB (1%)	
Configured Capacity	: 8.56 GB
DFS Used	: 456 KB
Non DFS Used	: 764.23 MB
DFS Remaining	: 7.82 GB
DFS Used%	: 0.01 %
DFS Remaining%	: 91.28 %
<a href="#">Live Nodes</a>	: 2
<a href="#">Dead Nodes</a>	: 0

图 12-3 HDFS 管理界面 (部分截图)

## 12.3 使用 Python 编写 MapReduce

Map 与 Reduce 为两个独立函数, 为了加快各节点的处理速度, 使用并行的计算方式, map 运算的结果再由 reduce 继续进行合并。例如, 要统计图书馆有多少本书籍, 首先一人一排进行统计 (map), 其次将每个人的统计结果进行汇总 (reduce), 最终得出总数。Hadoop 除

了提供原生态的 Java 来编写 MapReduce 任务，还提供了其他语言操作的 API——Hadoop Streaming，它通过使用标准的输入与输出来实现 map 与 reduce 之前传递数据，映射到 Python 中便是 sys.stdin 输入数据、sys.stdout 输出数据。其他业务逻辑也直接在 Python 中编写。

下面实现一个统计文本文件（/home/test/hadoop/input.txt）中所有单词出现的词频功能，分别使用原生 Python 与框架方式来编写 mapreduce。文本文件内容如下：

【 /home/test/hadoop/input.txt 】

```
foo foo quux labs foo bar quux abc bar see you by test welcome test
abc labs foo me python hadoop ab ac bc bec python
```

### 12.3.1 用原生 Python 编写 MapReduce 详解

#### （1）编写 Map 代码

见下面的 mapper.py 代码，它会从标准输入（stdin）读取数据，默认以空格分割单词，然后按行输出单词及其词频到标准输出（stdout），不过整个 Map 处理过程并不会统计每个单词出现的总次数，而是直接输出“word 1”，以便作为 Reduce 的输入进行统计，要求 mapper.py 具备可执行权限，执行 chmod +x /home/test/hadoop/mapper.py。

【 /home/test/hadoop/mapper.py 】

```
#!/usr/bin/env python
import sys
# 输入为标准输入 stdin;
for line in sys.stdin:
    # 删除开头和结尾的空格;
    line = line.strip()
    # 以默认空格分隔行单词到 words 列表;
    words = line.split()
    for word in words:
        # 输出所有单词，格式为“单词,1”以便作为 Reduce 的输入;
        print '%s\t%s' % (word, 1)
```

#### （2）编写 Reduce 代码

见下面的 reducer.py 代码，它会从标准输入（stdin）读取 mapper.py 的结果，然后统计每个单词出现的总次数并输出到标准输出（stdout），要求 reducer.py 同样具备可执行权限，执行 chmod +x /home/test/hadoop/reducer.py。

【 /home/test/hadoop/reducer.py 】

```
#!/usr/bin/env python
```

```

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# 获取标准输入，即 mapper.py 的输出；
for line in sys.stdin:
    # 删除开头和结尾的空格；
    line = line.strip()

    # 解析 mapper.py 输出作为程序的输入，以 tab 作为分隔符；
    word, count = line.split('\t', 1)

    # 转换 count 从字符型成整型；
    try:
        count = int(count)
    except ValueError:
        # count 非数字时，忽略此行；
        continue

    # 要求 mapper.py 的输出做排序 (sort) 操作，以便对连续的 word 做判断；
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # 输出当前 word 统计结果到标准输出
            print '%s\t%s' % (current_word, current_count)
            current_count = count
            current_word = word

# 输出最后一个 word 统计
if current_word == word:
    print '%s\t%s' % (current_word, current_count)

```

### (3) 测试代码

我们可以在 Hadoop 平台运行之前在本地进行测试，校验 mapper.py 与 reducer.py 运行的结果是否正确，测试结果如图 12-4 所示。

测试 reducer.py 时需要对 mapper.py 的输出做排序 (sort) 操作，当然，Hadoop 环境会自动实现排序，如图 12-5 所示。

### (4) 在 Hadoop 平台运行代码

首先在 HDFS 上创建文本文件存储目录，本示例中为 /user/root/word，运行命令：

```
# /usr/local/hadoop-1.2.1/bin/hadoop dfs -mkdir /user/root/word
```

上传文件至 HDFS，本示例中为 /home/test/hadoop/input.txt，如果有多个文件，可采用以下方法进行操作，因为 Hadoop 分析目标默认针对目录，目录下的文件都在运算范围中。

```
# /usr/local/hadoop-1.2.1/bin/hadoop fs -put /home/test/hadoop/input.txt /user/
root/word/
# /usr/local/hadoop-1.2.1/bin/hadoop dfs -ls /user/root/word/
Found 1 items
-rw-r--r--  2 root supergroup  118 2014-02-10 09:49 /user/root/word/input.txt
```



```
[root@SN2013-08-020 hadoop]# cat input.txt | ./mapper.py
foo 1
foo 1
quux 1
labs 1
foo 1
bar 1
quux 1
abc 1
bar 1
see 1
you 1
by 1
test 1
welcome 1
test 1
abc 1
labs 1
foo 1
```

图 12-4 mapper 执行结果（部分截图）



```
[root@SN2013-08-020 hadoop]# cat input.txt | ./mapper.py | sort -k1,1 | ./reducer.py
ab 1
abc 2
ac 1
bar 2
bc 1
bec 1
by 1
foo 4
hadoop 1
labs 2
me 1
python 2
quux 2
see 1
test 2
welcome 1
you 1
```

图 12-5 reducer 执行结果

下一步便是关键的执行 MapReduce 任务了，输出结果文件指定 /output/word，执行以下命令：

```
# /usr/local/hadoop-1.2.1/bin/hadoop jar /usr/local/hadoop-1.2.1/contrib/
```

```
streaming/hadoop-streaming-1.2.1.jar -file ./mapper.py -mapper ./mapper.py -file
./reducer.py -reducer ./reducer.py -input /user/root/word -output /output/word
```

图 12-6 为返回的执行结果，可以看到 map 及 reduce 计算的百分比进度。

```
[root@SN2013-08-020 ~]# /usr/local/hadoop-1.2.1/bin/hadoop jar /usr/local/hadoop-1.2.1/contrib/streaming/hadoop-
streaming-1.2.1.jar -file ./mapper.py -mapper ./mapper.py -file ./reducer.py -reducer ./reducer.py -input /user/root/
word -output /output/word
packageJobJar: [/mapper.py, ./reducer.py, /data/tmp/hadoop-root/hadoop-unjar5365959309140058570/] [] /tmp/streamjob3
1424838332804003.jar tmpDir=null
14/08/10 22:50:09 INFO util.NativeCodeLoader: Loaded the native-hadoop library
14/08/10 22:50:09 WARN SnappyLoadSnappy: Snappy native library not loaded
14/08/10 22:50:09 INFO mapred.FileInputFormat: Total input paths to process : 1
14/08/10 22:50:11 INFO streaming.StreamJob: getLocalDirs(): [/data/tmp/hadoop-root/mapred/local]
14/08/10 22:50:11 INFO streaming.StreamJob: Running job: job_201408101951_0002
14/08/10 22:50:11 INFO streaming.StreamJob: To kill this job, run:
14/08/10 22:50:11 INFO streaming.StreamJob: /usr/local/hadoop-1.2.1/libexec/./bin/hadoop job -Dmapred.job.tracker-1
92.168.1.20:9001 -kill job_201408101951_0002
14/08/10 22:50:11 INFO streaming.StreamJob: Tracking URL: http://SN2013-08-020:50030/jobdetails.jsp?jobid=job_2014081
01951_0002
14/08/10 22:50:12 INFO streaming.StreamJob: map 0% reduce 0%
14/08/10 22:51:40 INFO streaming.StreamJob: map 50% reduce 0%
14/08/10 22:52:34 INFO streaming.StreamJob: map 100% reduce 0%
14/08/10 22:52:38 INFO streaming.StreamJob: map 100% reduce 17%
14/08/10 22:52:46 INFO streaming.StreamJob: map 100% reduce 100%
14/08/10 22:52:50 INFO streaming.StreamJob: Job complete: job_201408101951_0002
14/08/10 22:52:50 INFO streaming.StreamJob: Output: /output/word
```

图 12-6 执行 MapReduce 任务结果

访问 <http://192.168.1.20:50030/jobtracker.jsp>，点击生成的 Jobid，查看 mapreduce job 信息，如图 12-7 所示。

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	2	0	0	2	0	0 / 1
reduce	100.00%	1	0	0	1	0	0 / 0

	Counter	Map	Reduce	Total
File Input Format Counters	Bytes Read	0	0	177
Job Counters	SLOTS_MILLIS_MAPS	0	0	134,819
	Launched reduce tasks	0	0	1
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Launched map tasks	0	0	3
	Data-local map tasks	0	0	3

图 12-7 Web 查看 mapreduce job 信息（部分截图）

查看生成的分析结果文件清单，其中 /output/word/part-00000 为分析结果文件，如

图 12-8 所示。

```
[root@SN2013-08-020 hadoop]# /usr/local/hadoop-1.2.1/bin/hadoop dfs -ls /output/word
Found 3 items
-rw-r--r-- 2 root supergroup 0 2014-08-10 22:52 /output/word/_SUCCESS
drwxr-xr-x - root supergroup 0 2014-08-10 22:50 /output/word/_logs
-rw-r--r-- 2 root supergroup 110 2014-08-10 22:52 /output/word/part-00000
[root@SN2013-08-020 hadoop]#
```

图 12-8 任务输出文件清单

最后查看结果数据，图 12-9 显示了单词个数统计的结果，整个分析过程结束。

```
[root@SN2013-08-020 hadoop]# /usr/local/hadoop-1.2.1/bin/hadoop dfs -cat /output/word/part-00000
ab 1
abc 2
ac 1
bar 2
bc 1
bec 1
by 1
foo 4
hadoop 1
labs 2
me 1
python 2
quux 2
see 1
test 2
welcome 1
you 1
```

图 12-9 查看结果文件 part-00000 内容



**提示** HDFS 常用操作命令有：

- 1) 创建目录，示例：bin/hadoop dfs -mkdir /data/root/test。
- 2) 列出目录清单，示例：bin/hadoop dfs -ls /data/root。
- 3) 删除文件或目录，示例：bin/hadoop fs -rmr /data/root/test。
- 4) 上传文件，示例：bin/hadoop fs -put /home/test/hadoop/\*.txt /data/root/test。
- 5) 查看文件内容，示例：bin/hadoop dfs -cat /output/word/part-00000。

### 12.3.2 用 Mrjob 框架编写 MapReduce 详解

Mrjob (<http://pythonhosted.org/mrjob/index.html>) 是一个编写 MapReduce 任务的开源 Python 框架，它实际上对 Hadoop Streaming 的命令进行了封装，因此接触不到 Hadoop 的数据流命令行，使我们可以更轻松、快速编写 MapReduce 任务。Mrjob 具有如下特点。

- 1) 代码简洁，map 及 reduce 函数通过一个 Python 文件就可以搞定；
- 2) 支持多步骤的 MapReduce 任务工作流；

- 3) 支持多种运行方式, 包括内嵌方式、本地环境、Hadoop、远程亚马逊;
- 4) 支持亚马逊网络数据分析服务 Elastic MapReduce (EMR);
- 5) 调试方便, 无须任何环境支持。

安装 Mrjob 要求环境为 Python 2.5 及以上版本, 源码下载地址: <https://github.com/yelp/mrjob>。

```
# pip install mrjob      #PyPI 安装方式
# python setup.py install  # 源码安装方式
```

回到实现一个统计文本文件 (/home/test/hadoop/input.txt) 中所有单词出现的词频功能, Mrjob 通过 mapper() 与 reducer() 方法实现了 MR 操作, 实现代码如下:

【 /home/test/hadoop/word\_count.py 】

```
from mrjob.job import MRJob
class MRWordCounter(MRJob):
    def mapper(self, key, line):
        for word in line.split():
            yield word, 1
    def reducer(self, word, occurrences):
        yield word, sum(occurrences)

if __name__ == '__main__':
    MRWordCounter.run()
```

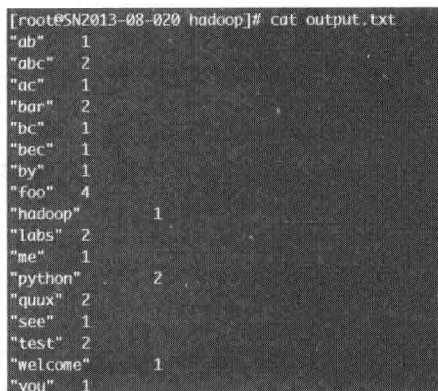
可以看出代码行数只是原生 Python 的 1/3, 逻辑也比较清晰, 代码中包含了 mapper、reducer 函数。mapper 函数接收每一行的输入数据, 处理后返回一对 key:value, 初始化 value 为数据 1; reducer 接收 mapper 输出的 key-value 对进行整合, 把相同 key 的 value 作累加 (sum) 操作后输出。Mrjob 利用 Python 的 yield 机制将函数变成一个 Generators (生成器), 通过不断调用 next() 去实现 key-value 的初始化或运算操作。前面介绍 Mrjob 支持四种运行方式, 包括内嵌 (-r inline)、本地 (-r local)、Hadoop (-r hadoop)、Amazon EMR (-r emr), 下面主要介绍前三者的运行方式。

#### (1) 内嵌 (-r inline) 方式

特点是调试方便, 启动单一进程模拟任务执行状态及结果, 默认 (-r inline) 可以省略, 输出文件使用 “> output-file” 或 “-o output-file”。下面两条命令是等价的:

```
# python word_count.py -r inline input.txt >output.txt
# python word_count.py input.txt -o output.txt
```

输出文件 output.txt 内容见图 12-10。



```
[root@SN2013-08-020 hadoop]# cat output.txt
"ab" 1
"abc" 2
"ac" 1
"bar" 2
"bc" 1
"bec" 1
"by" 1
"foo" 4
"hadoop" 1
"labs" 2
"me" 1
"python" 2
"quux" 2
"see" 1
"test" 2
"welcome" 1
"you" 1
```

图 12-10 查看输出 output.txt 文件内容

## (2) 本地 (-r local) 方式

用于本地模拟 Hadoop 调试，与内嵌 (inline) 方式的区别是启动了多进程执行每一个任务，如：

```
# python word_count.py -r local input.txt >output.txt
```

执行的结果与 inline 一样，只是运行过程存在差异。

## (3) Hadoop (-r hadoop) 方式

用于 Hadoop 环境，支持 Hadoop 运行调度控制参数，如：

- ❑ 指定 Hadoop 任务调度优先级 (VERY\_HIGH|HIGH)，如，`--jobconf mapreduce.job.priority=VERY_HIGH`。
- ❑ Map 及 Reduce 任务个数限制，如，`--jobconf mapred.map.tasks=10 --jobconf mapred.reduce.tasks=5`。

注意，执行之前需要指定 Hadoop 环境变量，执行结果见图 12-11。

访问 <http://192.168.1.20:50030/jobtracker.jsp>，显示的最后一行便是任务执行的信息，从中可以看到任务的优先级、map 及 reduce 的总数，如图 12-12 所示。

查看 Hadoop 分析结果文件，内容见图 12-13。

Mrjob 框架的介绍告一段落，下一节重点以实际案例进行说明。

```
[root@SN2013-08-020 hadoop]# export HADOOP_HOME=/usr/local/hadoop-1.2.1
[root@SN2013-08-020 hadoop]# python word_count.py -r hadoop --jobconf mapreduce.job.priority-VERY_HIGH --jobconf mapred.map.tasks-2 --jobconf mapred.reduce.tasks-1 -o hdfs:///output/hadoop_word hdfs:///user/root/word
no configs found; falling back on auto-configuration
no configs found; falling back on auto-configuration
creating tmp directory /tmp/word_count.root.20140810.150905.175991
writing wrapper script to /tmp/word_count.root.20140810.150905.175991/setup-wrapper.sh
Copying local files into hdfs:///user/root/tmp/mrjob/word_count.root.20140810.150905.175991/Files/
Using Hadoop version 1.2.1
Detected hadoop configuration property names that do not match hadoop version 1.2.1:
The have been translated as follows
  mapreduce.job.priority: mapred.job.priority
HADOOP: Warning: $HADOOP_HOME is deprecated.
HADOOP:
HADOOP: packageJobJar: [/data/tmp/hadoop-root/hadoop-unjar8783376954064499738/] □ /tmp/streamjob4692395595666690487.
jar tmpDir=null
HADOOP: Loaded the native-hadoop library
HADOOP: Snappy native library not loaded
HADOOP: Total input paths to process : 1
HADOOP: getLocalDirs(): [/data/tmp/hadoop-root/hadoop-root/mapred/local]
HADOOP: Running job: job_201408101951_0003
HADOOP: To kill this job, run:
HADOOP: /usr/local/hadoop-1.2.1/libexec/./bin/hadoop job -Dmapred.job.tracker=192.168.1.20:9001 -kill job_201408101951_0003
HADOOP: Tracking URL: http://SN2013-08-020:50030/jobdetails.jsp?jobid=job_201408101951_0003
HADOOP: map 0% reduce 0%
HADOOP: map 50% reduce 0%
HADOOP: map 100% reduce 0%
```

图 12-11 任务执行结果（部分截图）

Completed Jobs

Jobid	Started	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
job_201408222215_0001	Fri Aug 22 22:28:11 CST 2014	NORMAL	root	streamjob4197546037759856201.jar	100.00%	2	2	100.00%
job_201408222215_0002	Fri Aug 22 22:36:58 CST 2014	VERY_HIGH	root	streamjob6340565803234573854.jar	100.00%	2	2	100.00%

图 12-12 已完成任务清单（部分截图）

```
[root@SN2013-08-020 hadoop]# /usr/local/hadoop-1.2.1/bin/hadoop dfs -cat /output/hadoop_word/part-00000
"ab" 1
"abc" 2
"ac" 1
"bar" 2
"bc" 1
"bec" 1
"by" 1
"foo" 4
"hadoop" 1
"labs" 2
"me" 1
"python" 2
"quux" 2
"see" 1
"test" 2
"welcome" 1
"you" 1
```

图 12-13 查看任务结果文件内容

## 12.4 实战分析

在互联网企业中，随着业务量、访问量的不断增长，用户产生的数据也越来越大，如何处理大数据的存储与分析问题呢？比如 Web 服务器的访问 log，当日志只有 GB 单位大小时，我们还可以勉强通过 shell、awk 进行分析，当达到上百 GB，甚至上 PB 级别时，通过脚本的方式已经力不从心了。另外一个待解决的问题就是数据存储。Hadoop 很好地解决了这两个问题，即分布式存储与计算。下面将通过示例介绍如何从 Web 日志中快速获取访问流量、HTTP 状态信息、用户 IP 信息、连接数/分钟统计等。

### 12.4.1 示例场景

站点 www.website.com 共有 5 台 Web 设备，日志文件存放位置：/data/logs/ 日期 (20140215)/access.log，日志为默认的 Apache 定义格式，如：

```
125.26.28.8 - - [01/Aug/2010:09:56:53 +0700] "GET /teacher/jitra/image/pen.gif HTTP/1.1" 200 12014 "http://www.kpsw.ac.th/teacher/jitra/page4.htm"
"Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; GTB6.5; InfoPath.1; .NET CLR 2.0.50727; yie8)"
125.26.28.8 - - [01/Aug/2010:09:56:53 +0700] "GET /favicon.ico HTTP/1.1" 200 1187 "-" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; GTB6.5; InfoPath.1; .NETCLR 2.0.50727; yie8)"
66.249.65.37 - - [01/Aug/2010:09:57:59 +0700] "GET /picture/49-02/DSC02630.jpg HTTP/1.1" 200 79220 "-" "Googlebot-Image/1.0"
66.249.65.37 - - [01/Aug/2010:09:59:19 +0700] "GET /elearning/index.php?cal_m=2&cal_y=2011 HTTP/1.1" 200 9232 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)"
```

共有 12 列数据（空格分隔），分别为：①客户端 IP；②空白（远程登录名称）；③空白（认证的远程用户）；④请求时间；⑤ UTC 时差；⑥方法；⑦资源；⑧协议；⑨状态码；⑩发送字节数；⑪访问来源；⑫客户浏览器信息（不具体拆分）。

接下来在 5 台 Web 服务器部署 HDFS 的客户端，以便定期上传 Web 日志到 HDFS 存储平台，最终实现分布式计算。需要安装（JDK 配置环境变量）、Hadoop（原版 tar 包解析即可），详细见 12.2 相关内容。添加上传日志功能作业到 crontab，内容如下：

```
55 23 * * * /usr/bin/python /home/test/hadoop/hdfsput.py >> /dev/null 2>&1
```

通过 subprocess.Popen() 方法调用 Hadoop HDFS 相关外部命令，实现创建 HDFS 目录及客户端文件上传，详细代码如下：

```
【 /home/test/hadoop/hdfsput.py 】
```

```
import subprocess
```

```

import sys
import datetime
webid="web1" #HDFS 存储日志标志, 其他 Web 服务器分别为 web2、web3、web4、web5
currdate=datetime.datetime.now().strftime('%Y%m%d')
logspath="/data/logs/"+currdate+"/access.log" # 日志本地路径
logname="access.log."+webid #HDFS 存储日志名

try:
    subprocess.Popen(["/usr/local/hadoop-1.2.1/bin/hadoop", "dfs", "-mkdir",
"hd fs://192.168.1.20:9000/user/root/website.com/"+currdate], stdout=subprocess.PIPE)
# 创建 HDFS 目录, 目录格式: website.com/20140205
except Exception,e:
    pass
putinfo=subprocess.Popen(["/usr/local/hadoop-1.2.1/bin/hadoop",
"dfs", "-put", logspath, "hd fs://192.168.1.20:9000/user/root/website.
com/"+currdate+"/"+logname], stdout=subprocess.PIPE) # 上传本地日志到 HDFS

for line in putinfo.stdout:
    print line

```

在 crontab 定时作业运行后, 5 台 Web 服务器的日志在 HDFS 上的信息如下:

```

# /usr/local/hadoop-1.2.1/bin/hadoop dfs -ls /user/root/website.com/20140215
Found 5 items
-rw-r--r--   3 root supergroup  156541746 2014-02-15 23:55 /user/root/website.
com/20140215/access.log.web1
-rw-r--r--   3 root supergroup  251245315 2014-02-15 23:53 /user/root/website.
com/20140215/access.log.web2
-rw-r--r--   3 root supergroup  134256412 2014-02-15 23:55 /user/root/website.
com/20140215/access.log.web3
-rw-r--r--   3 root supergroup  192314554 2014-02-15 23:54 /user/root/website.
com/20140215/access.log.web4
-rw-r--r--   3 root supergroup  183267834 2014-02-15 23:55 /user/root/website.
com/20140215/access.log.web5

```

截至目前, 数据的分析源已经准备就绪, 接下来的工作便是分析了。

### 12.4.2 网站访问流量统计

网站访问流量作为衡量一个站点的价值、热度的重要标准, 另外在 CDN 服务中流量会涉及计费, 如何快速准确分析当前站点的流量数据至关重要, 当然, 使用 Mrjob 可以很轻松实现此类需求。下面实现精确到分钟统计网站访问流量, 原理是在 mapper 操作时将 Web 日志中小时的每分钟作为 key, 将对应的行发送字节数作为 value, 在 reducer 操作时对相同 key 作累加 (sum) 统计, 详细源码如下:

【 /home/test/hadoop/httpflow.py 】

```

from mrjob.job import MRJob
import re

class MRCounter(MRJob):

    def mapper(self, key, line):
        i=0
        for flow in line.split():
            if i==3: # 获取时间字段, 位于日志的第4列, 内容如 "[06/Aug/2010:03:19:44"
                timerow= flow.split(":")
                hm=timerow[1]+":"+timerow[2] # 获取 "小时:分钟", 作为 key
            if i==9 and re.match(r"\d{1,}", flow): # 获取日志第10列 - 发送的字节数,
                作为 value
                yield hm, int(flow) # 初始化 key:value
            i+=1

    def reducer(self, key, occurrences):
        yield key, sum(occurrences) # 相同 key "小时:分钟" 的 value 作累加操作

if __name__ == '__main__':
    MRCounter.run()

```

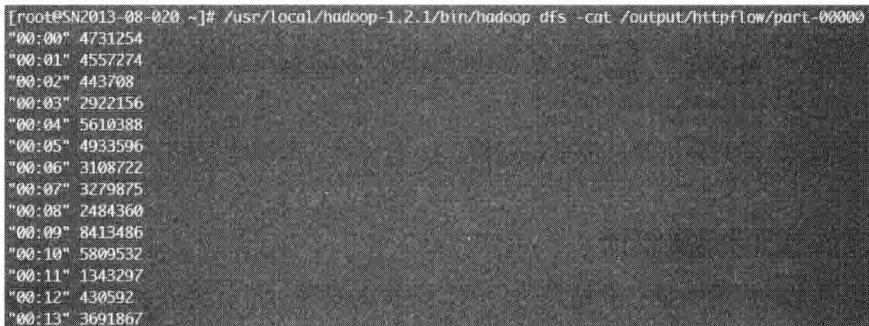
生成 Hadoop 任务, 运行:

```

# python /home/test/hadoop/httpflow.py -r hadoop --jobconf mapreduce.job.
priority=VERY_HIGH -o hdfs:///output/httpflow hdfs:///user/root/website.
com/20140215

```

分析结果见图 12-14。



```

[root@SN2013-08-020 ~]# /usr/local/hadoop-1.2.1/bin/hadoop dfs -cat /output/httpflow/part-00000
"00:00" 4731254
"00:01" 4557274
"00:02" 443708
"00:03" 2922156
"00:04" 5610388
"00:05" 4933596
"00:06" 3108722
"00:07" 3279875
"00:08" 2484360
"00:09" 8413486
"00:10" 5809532
"00:11" 1343297
"00:12" 430592
"00:13" 3691867

```

图 12-14 任务分析结果 (部分截图)

建议将分析结果数据定期入库 MySQL, 利用 MySQL 灵活、丰富的 SQL 支持, 可以很方便地对数据进行加工, 轻松输出比较美观的数据报表。图 12-15 为网站一天的流量趋势图。

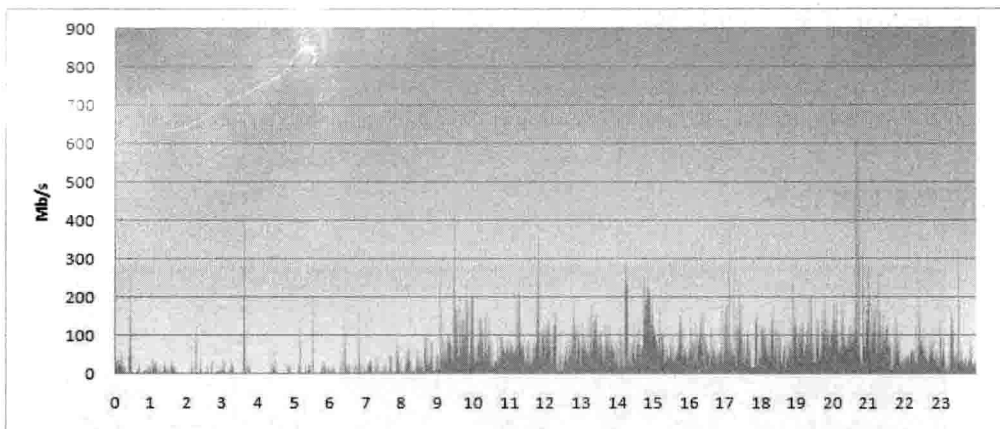


图 12-15 业务流量趋势图

### 12.4.3 网站 HTTP 状态码统计

统计一个网站的 HTTP 状态码比例数据,可以帮助我们了解网站的可用度及健康状态,比如我们关注的 200、404、5xx 状态等。在此示例中我们利用 Mrjob 的多步调用的形式来实现,除了基本的 mapper、reducer 方法外,还可以添加自定义处理方法,在 steps 中添加调用即可,详细源码如下:

【 /home/test/hadoop/httpstatus.py 】

```
from mrjob.job import MRJob
import re
class MRCounter(MRJob):
    def mapper(self, key, line):
        i=0
        for httpcode in line.split():
            if i==8 and re.match(r"\d{1,3}", httpcode):# 获取日志中 HTTP 状态码段, 作为 key
                yield httpcode, 1 # 初始化 key:value, value 计数为 1, 方便 reducer 作累加
            i+=1

    def reducer(self, httpcode, occurrences):
        yield httpcode, sum(occurrences) # 对排序后的 key 对应的 value 作 sum 累加

    def steps(self):
        return [self.mr(mapper=self.mapper), # 在 steps 方法中添加调用队列
                self.mr(reducer=self.reducer)]

if __name__ == '__main__':
    MRCounter.run()
```

生成 hadoop 任务，分析数据源保持不变，输出目录改成 /output/httpstatus，执行：

```
# python /home/test/hadoop/httpstatus.py -r hadoop --jobconf mapreduce.job.
priority=VERY_HIGH -o hdfs:///output/httpstatus hdfs:///user/root/website.
com/20140215
```

分析结果见图 12-16。

```
[root@SN2013-08-020 ~]# /usr/local/hadoop-1.2.1/bin/hadoop dfs -cat /output/httpstatus/part-000000
"200" 412334
"206" 19365
"301" 1594
"302" 44
"303" 412
"304" 127633
"400" 1
"404" 15499
"405" 72
"416" 12
"501" 2
"503" 31
```

图 12-16 任务分析结果

我们可以根据结果数据输出比例饼图，如图 12-17 所示。

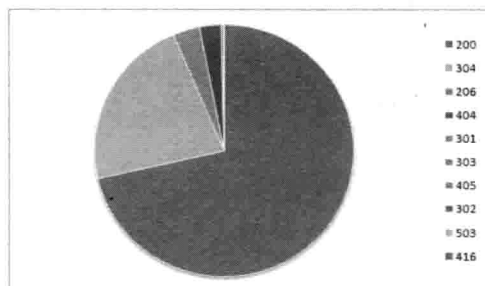


图 12-17 生成 HTTP 状态码饼图

#### 12.4.4 网站分钟级请求数统计

一个网站的请求量大小，直接关系到网站的访问质量，非常有必要对该数据进行分析且关注。本示例以分钟为单位对网站的访问数进行统计，原理与 12.4.2 类似，区别是 value 初始为 1，以便作累加统计，详细源码如下：

```
【 /home/test/hadoop/http_minute_conn.py 】
```

```
from mrjob.job import MRJob
import re
```

```

class MRCounter(MRJob):

    def mapper(self, key, line):
        i=0
        for dt in line.split():
            if i==3: # 获取时间字段，位于日志的第4列，内容如 "[06/Aug/2010:03:19:44"
                timerow= dt.split(":")
                hm=timerow[1]+":"+timerow[2] # 获取“小时:分钟”，作为 key
                yield hm, 1 # 初始化 key:value, value 计数为 1, 方便 reducer 作累加
            i+=1

    def reducer(self, key, occurrences):
        yield key, sum(occurrences)

if __name__ == '__main__':
    MRCounter.run()

```

生成 Hadoop 任务，输出目录 /output/ http\_minute\_conn，执行：

```

# python /home/test/hadoop/http_minute_conn.py -r hadoop --jobconf mapreduce.
job.priority=VERY_HIGH -o hdfs:///output/http_minute_conn hdfs:///user/root/
website.com/20140215

```

分析结果见图 12-18。



```

[root@SN2013-08-020 ~]# /usr/local/hadoop-1.2.1/bin/hadoop dfs -cat /output/http_minute_conn/part-00000
"00:00" 58
"00:01" 168
"00:02" 43
"00:03" 166
"00:04" 105
"00:05" 208
"00:06" 126
"00:07" 223
"00:08" 242
"00:09" 50
"00:10" 134
"00:11" 45
"00:12" 21
"00:13" 32

```

图 12-18 任务分析结果（部分截图）

#### 12.4.5 网站访问来源 IP 统计

统计用户的访问来源 IP 可以更好地了解网站的用户分布，同时也可以帮助安全人员捕捉攻击来源。实现原理是定义匹配 IP 正则字符串作为 key，将 value 初始化为 1，执行 reducer 操作时作累加（sum）统计，详细源码如下：

【 /home/test/hadoop/ipstat.py 】

```
from mrjob.job import MRJob
import re

IP_RE = re.compile(r"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}") # 定义 IP 正则匹配

class MRCounter(MRJob):

    def mapper(self, key, line):
        # 匹配 IP 正则后生成 key:value, 其中 key 为 IP 地址, value 初始值为 1
        for ip in IP_RE.findall(line):
            yield ip, 1

    def reducer(self, ip, occurrences):
        yield ip, sum(occurrences)

if __name__ == '__main__':
    MRCounter.run()
```

生成 Hadoop 任务, 输出目录 /output/ ipstat, 执行:

```
# python /home/test/hadoop/ipstat.py -r hadoop --jobconf mapreduce.job.
priority=VERY_HIGH -o hdfs:///output/ipstat hdfs:///user/root/website.
com/20140215
```

分析结果见图 12-19。



```
[root@SN2013-08-020 ~]# /usr/local/hadoop-1.2.1/bin/hadoop dfs -cat /output/ipstat/part-00000
"1.8.1.20"      1098
"1.8.1.6"       11
"1.8.1.7"       19
"1.9.0.1"       426
"1.9.0.10"      120
"1.9.0.11"      18
"1.9.0.13"      10
"1.9.0.14"      7
"1.9.0.6"       4
"1.9.0.7"       33
"1.9.0.8"       60
"1.9.1.1"       58
"1.9.1.10"      174
"1.9.1.11"      3203
"1.9.1.2"       322
"1.9.1.3"       492
```

图 12-19 任务分析结果 (部分截图)

## 12.4.6 网站文件访问统计

通过统计网站文件的访问次数可以帮助运维人员了解访问最集中的文件, 以便进行有针

对性的优化，比如调整静态文件过期策略、优化动态 cgi 的执行速度、拆分业务逻辑等。实现原理是将访问文件作为 key，初始化 value 为 1，执行 reducer 时作累加（sum）统计，详细源码如下：

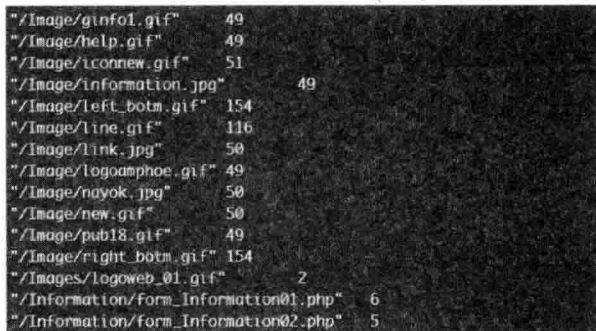
【 /home/test/hadoop/httpfile.py 】

```
from mrjob.job import MRJob
import re
class MRCounter(MRJob):
    def mapper(self, key, line):
        i=0
        for url in line.split():
            if i==6:      # 获取日志中 URL 文件资源字段，作为 key
                yield url, 1
            i+=1

    def reducer(self, url, occurrences):
        yield url, sum(occurrences)

if __name__ == '__main__':
    MRCounter.run()
```

执行结果如图 12-20 所示。



```
"/Image/ginfo1.gif"      49
"/Image/help.gif"       49
"/Image/iconnew.gif"    51
"/Image/information.jpg" 49
"/Image/left_botm.gif"  154
"/Image/line.gif"       116
"/Image/link.jpg"       50
"/Image/logoamphoe.gif" 49
"/Image/nayok.jpg"      50
"/Image/new.gif"        50
"/Image/pub18.gif"      49
"/Image/right_botm.gif" 154
"/Images/logoweb_01.gif" 2
"/Information/form_Information01.php" 6
"/Information/form_Information02.php" 5
```

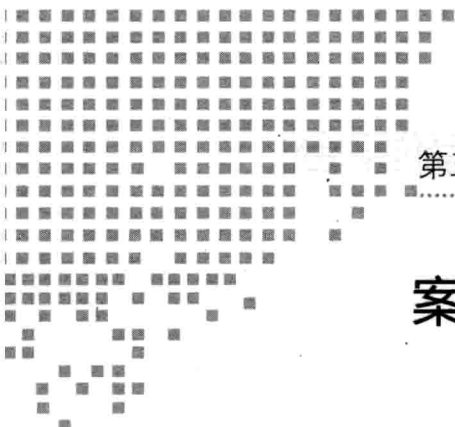
图 12-20 任务分析结果（部分截图）

同理，我们可以使用以上方法对 User-Agent 域进行分析，包括浏览器类型及版本、操作系统及版本、浏览器内核等信息，为更好地提升用户体验提供数据支持。



**参考提示** 12.2.1 小节原生 Python 编写 mapreduce 示例参考 <http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/>。

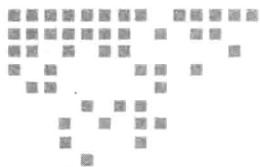




### 第三部分 *Part 3*

## 案例篇

- 第 13 章 从零开始打造 B/S 自动化运维平台
- 第 14 章 打造 Linux 系统安全审计功能
- 第 15 章 构建分布式质量监控平台
- 第 16 章 构建桌面版 C/S 自动化运维平台



## 从零开始打造 B/S 自动化运维平台

随着企业业务的不断发展，在运营方面，如何保障业务的高可用及服务质量，系统管理员将面临越来越多的挑战。目前，很多企业还处在传统的“半自动化”状态，一旦出现运维事故，技术部的每个人都会加入“救火”行列，最后弄得疲惫不堪。因此，构建高效的运营模式已迫在眉睫，可以从以下几个方面入手，包括定制符合企业特点的 IT 制度、流程规范、质量与成本管理、运营效率建设等。本章将介绍如何使用 Python 从零开始打造一个易用、扩展性强、安全、高效的自动化运维平台，从而提高运营人员的工作效率。（本章的源码也可从 <https://github.com/yorkoliu/pyauto> 下载。）

### 13.1 平台功能介绍

作为 ITIL 体系当中的一部分，本平台同样遵循 ITIL 标准设计规范。OMServer 是本平台的名称，后面的内容将使用它作为平台的称号。OMServer 实现了一个集中式的 Linux 集群管理基础平台，提供了模块扩展的支持，可以随意添加集群操作任务模块，服务器端模块支持前端 HTML 表单参数动态定制，可灵活实现日常运维远程操作、文件分发等任务；在安全方面，采用加密（RC4 算法）指令传输、操作日志记录、分离 Web Server 与主控设备等；在效率方面，管理员只需选择操作目标对象及操作模块即可完成一个现网变更任务。另外，在用户体验方面，采用前端异步请求，模拟 Linux 终端效果接收返回串。任何人都可以根据自身的业务特点对 OMServer 平台进行扩展，比如与现有资产平台进行对接，或整合到现有的运营平台当中。平台首页如图 13-1 所示。



图 13-1 平台首页界面

## 13.2 系统架构设计

OMServer 平台采用了三层设计模式，第一层为 Web 交互层，采用了 Django+prototype.js+MySQL 实现，服务器端采用了 Nginx+uwsgi 构建高效的 Web 服务；第二层为分布式计算层，采用 rpyc 分布式计算框架实现，作为第一层与第三层的数据交互及实现主控端物理分离，提高整体安全性，同时具备第三层的多机服务的能力；第三层为集群主控端服务层，支持 Saltstack、Ansible、Func 等平台。具体见如图 13-2 所示的系统架构图。

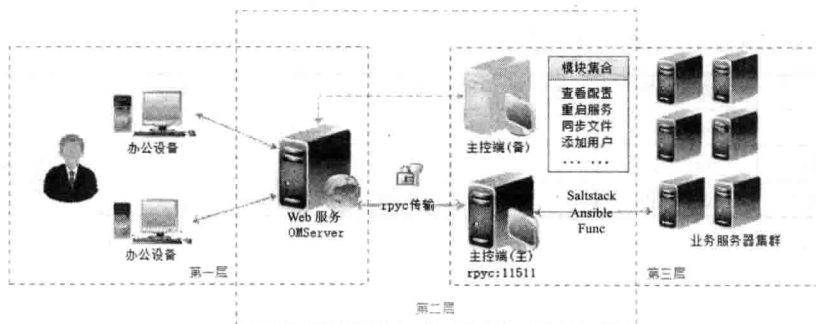


图 13-2 系统架构图

从图 13-2 可以看出系统的三个层次，首先管理员向 OMServer 平台所在 Web 服务器发起 HTTP 请求，OMServer 接收 HTTP POST 的数据并采用“RC4+b64encode+ 密钥 key”进行加密，再作为 rpyc 客户端向 rpyc 服务器发送加密指令串，rpyc 服务器端同时也是 Saltstack、Ansible、Func 等的主控端，主控端将接收到的数据通过“RC4+b64decode+ 密钥 key”进行解密，解析成 OMServer 调用的任务模块，结合 Saltstack、Ansible 或 Func 向目标业务服务器集群发送执行任务，执行完毕后，将返回的执行结果加解密处理，最后逐级返回给系统管理员，整个任务模块分发执行流程结束。

## 13.3 数据库结构设计

### 13.3.1 数据库分析

OMServer 平台采用了开源数据库 MySQL 作为数据存储，将数据库命名为 OMServer，该数据库总共有 4 张表，表信息说明如下。

- ❑ server\_fun\_categ: 服务功能分类表。
- ❑ server\_app\_categ: 服务应用分类表。
- ❑ server\_list: 服务器列表。
- ❑ module\_list: 模块列表。

### 13.3.2 数据字典

server\_fun\_categ 服务功能分类表。

字段名	数据类型	默认值	允许非空	自动递增	备注
ID	int(11)		NO	是	服务功能分类 ID
server_categ_name	char(20)		NO		服务功能分类名称

server\_app\_categ 服务应用分类表。

字段名	数据类型	默认值	允许非空	自动递增	备注
ID	int(11)		NO	是	服务应用分类 ID
server_categ_id	int(11)				服务功能分类 ID
app_categ_name	char(30)		NO		服务应用分类名称

server\_list 服务器列表。

字段名	数据类型	默认值	允许非空	自动递增	备注
server_name	char(13)		NO		主机名称
server_wip	char(15)		NO		主机外网 IP
server_lip	char(12)		NO		主机内网 IP
server_op	char(10)		NO		主机操作系统
server_app_id	int(11)		NO		服务应用分类 ID

module\_list 模块列表。

字段名	数据类型	默认值	允许非空	自动递增	备注
ID	int(11)		NO	是	模块 ID 号
module_name	char(20)		NO		模块名称
module_caption	char(255)		NO		模块功能描述
module_extend	varchar(2000)		NO		模块前端扩展

### 13.3.3 数据库模型

在 ITIL 体系中有一种比较典型的资产定义方法，即采用“功能分类”作为根类，其子类为“应用分类”，在最小单位的“服务器”中指定“应用分类”进行关联，完成其层次关系的定义，例如，Linux.Web（一级功能类别），bbs.domain.com（二级应用类别），10.11.100.10（服务器归 bbs.domain.com 类别），详见图 13-3 所示的数据库模型图。

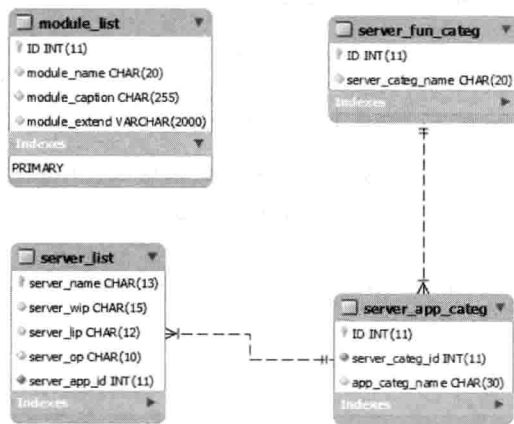


图 13-3 数据库模型

从模型关系图中可以看出，server\_list 表中的 server\_app\_id 字段被设置为外键，与 server\_app\_cate 表中的 ID 字段进行关联；server\_app\_cate 表中的 server\_cate\_id 字段被设置为外键，与 server\_fun\_cate 表中的 ID 字段进行关联。

## 13.4 系统环境部署

### 13.4.1 系统环境说明

OMServer 采用 Django-1.4.9、nginx-1.5.9、uwsgi-2.0.4、rpyc-3.2.3 等开源组件来构建。为了便于读者理解，下面对平台的运行环境、安装部署、开发环境优化等进行详细说明。环境设备角色表如表 13-1 所示。

表 13-1 系统环境说明表

角色	主机名	IP	环境说明
主控端	SN2013-08-020	192.168.1.20	Saltstack   Ansible   Func 主控端、rpyc 服务器端
Web Server	SN2012-07-010	192.168.1.10	Django+uwsgi、rpyc 客户端

### 13.4.2 系统平台搭建

OMServer 平台涉及两个角色，其中一个为 Web 服务端，运行 Django 及 rpyc 环境，另一角色为主控端，需要部署 Saltstack、Ansible 或 Func 主控端环境，可参与本书第 9 ~ 11 章内容，本节不予详细介绍。另外同样需要部署 rpyc 环境。

#### (1) Django 环境部署

本示例部署主机为 192.168.1.10 (SN2012-07-010)。

```
# cd /home
# mkdir -p /home/install/Django && cd /home/install/Django      # 创建安装包目录
# mkdir -p /data/logs/      # 创建 uwsgi 日志目录
```

1) 安装 pcre。pcre 是一个轻量级的正则表达式函数库，Nginx 的 HTTP Rewrite 模块会用到，最新版本为 8.34 (对于 OMServer 平台环境来说非必选项)。

```
# wget ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-8.34.tar.gz
# tar -zxvf pcre-8.34.tar.gz
# cd pcre-8.34
# ./configure
# make && make install
# cd ..
```

2) 安装 Nginx。Nginx 是最流行的高性能 HTTP 服务器，最新版本为 1.5.9。

```
# wget http://nginx.org/download/nginx-1.5.9.tar.gz
# tar -zxvf nginx-1.5.9.tar.gz
# cd nginx-1.5.9
```

```
# wget http://nginx.org/download/nginx-1.5.9.tar.gz
# tar -zxvf nginx-1.5.9.tar.gz
# cd nginx-1.5.9
# ./configure --user=nobody --group=nobody --prefix=/usr/local/nginx --with-http_
stub_status_module --with-cc-opt='-O3' --with-cpu-opt=opteron
# make && make install
# cd ..
```

3) 安装 MySQL-python。MySQL-python 是 Python 访问 MySQL 数据库的第三方模块库，最新版本为 1.2.3c1。

```
# yum install -y MySQL-python      #yum 安装方式
# wget http://nchc.dl.sourceforge.net/project/mysql-python/mysql-python/1.2.2/
# tar -zxvf MySQL-python-1.2.2.tar.gz      # 源码安装方式
# cd MySQL-python-1.2.2
# python setup.py install
# cd ..
```

4) 安装 uwsgi。uwsgi 是一个快速的、纯 C 语言开发的、自维护、对开发者友好的 WSGI 服务器，旨在提供专业的 Python Web 应用发布和开发功能，最新版本为 2.0.4。

```
# wget http://projects.unbit.it/downloads/uwsgi-2.0.4.tar.gz
# tar -zxvf uwsgi-2.0.4.tar.gz
# cd uwsgi-2.0.4
# make
# cp uwsgi /usr/bin
# cd ..
```

5) 安装 Django。Django 是一个 Python 最流行的开源 Web 开发框架，最新版本为 1.6.5。考虑到兼容与稳定性，本示例使用 1.4.9 版本进行开发。

```
# wget https://www.djangoproject.com/m/releases/1.4/Django-1.4.9.tar.gz
# tar -zxvf Django-1.4.9.tar.gz
# cd Django-1.4.9
# python setup.py install
```

6) 配置 Nginx。修改 /usr/local/nginx/conf/nginx.conf，添加以下 server 域配置：

```
server {
    listen 80;
    server_name omserver.domain.com;

    location / {
        uwsgi_pass 192.168.1.10:9000;
        include uwsgi_params;
        uwsgi_param UWSGI_CHDIR /data/www/OMserverweb;
        uwsgi_param UWSGI_SCRIPT django_wsgi;
        access_log off;
    }
}
```

```

    }

    location ~* ^.+\. (mpg|avi|mp3|swf|zip|tgz|gz|rar|bz2|doc|xls|exe|ppt|txt
|tar|mid|midi|wav|rtf|mpeg)$ {
        root /data/www/OMserverweb/static;
        access_log off;
    }
}

```

其中“omserver.domain.com”为平台访问域名，“/data/www/OMserverweb”为项目根目录，可以根据具体环境进行修改。

7) 配置 uwsgi。创建 uwsgi 配置文件 /usr/local/nginx/conf/uwsgi.ini，详细内容如下：

```

[uwsgi]
socket = 0.0.0.0:9000      # 监听的地址及端口
master = true              # 启动主进程
pidfile = /usr/local/nginx/uwsgi.pid
processes = 8              # uwsgi 开启的进程数
chdir = /data/www/OMserverweb # 项目主目录
pythonpath = /data/www
profiler=true
memory-report=true
enable-threads = true
logdate=true
limit-as=6048
daemonize=/data/logs/django.log

```

启动 uwsgi 与 nginx 服务，建议配置成服务自启动脚本，便于后续的日常维护。详细启动脚本这里不展开说明，有兴趣的读者可参阅互联网上已经存在的相关资源。

```

# /usr/bin/uwsgi --ini /usr/local/nginx/conf/uwsgi.ini
# /usr/local/nginx/sbin/nginx

```

访问 <http://omserver.domain.com>，出现如图 4-4 所示的页面说明 Django+uwsgi 环境部署成功！

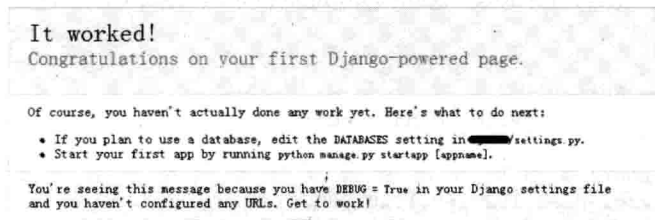


图 13-4 Django 默认首页

## (2) rpyc 模块安装。

rpyc (Remote Python Call) 是 Python 提供分布式计算的基础服务平台, 可以理解成封装程度更高的 Socket 编程, 最新版本为 3.3。本示例需要部署 rpyc 模块的主机为 192.168.1.20 (SN2013-08-020)、192.168.1.10 (SN2012-07-010)。

```
# wget https://pypi.python.org/packages/source/r/rpyc/rpyc-3.2.3.tar.gz --no-check-certificate
# tar -zxvf rpyc-3.2.3.tar.gz
# cd rpyc-3.2.3
# python setup.py install
```

### 13.4.3 开发环境优化

开发环境相对于生产环境更注重调试便捷性, 好的调试工具对软件开发将起到事半功倍的作用, 方便高效地定位问题。本节介绍 Django 必备调试工具 django-debug-toolbar 的安装与配置, 同时介绍如何实现一种 Django 代码自动刷新生效的方法。

#### (1) django-debug-toolbar 的安装

```
# wget https://github.com/robhudson/django-debug-toolbar/archive/master.zip
# unzip master
# cd django-debug-toolbar-master/
# python setup.py install
```

修改 Django 的 setting.py 配置, 关键参数如下:

```
INTERNAL_IPS = ('127.0.0.1', '192.168.1.101',) # 添加启动调试器的来源 IP
MIDDLEWARE_CLASSES = ( # MIDDLEWARE_CLASSES 添加以下行
    ...
    'debug_toolbar.middleware.DebugToolbarMiddleware',
)
INSTALLED_APPS = ( # INSTALLED_APPS 添加以下行
    ...
    'debug_toolbar',
)
TEMPLATE_DIRS = ( #TEMPLATE_DIRS 添加以下行, 注意与 python 的安装路径保持一致
    ...
    '/usr/lib/python2.6/site-packages/django_debug_toolbar-0.8.5-py2.6.egg/debug_toolbar/templates/',
)
```

务必要渲染一个模板, 这样 debug\_toolbar 才会自动附加调试信息到当前的页面, 否则看不到 debug\_tool 的界面。debug\_toolbar 在业务前端页面设计成可伸缩展示, 展开后的调试界面如图 13-5 所示。

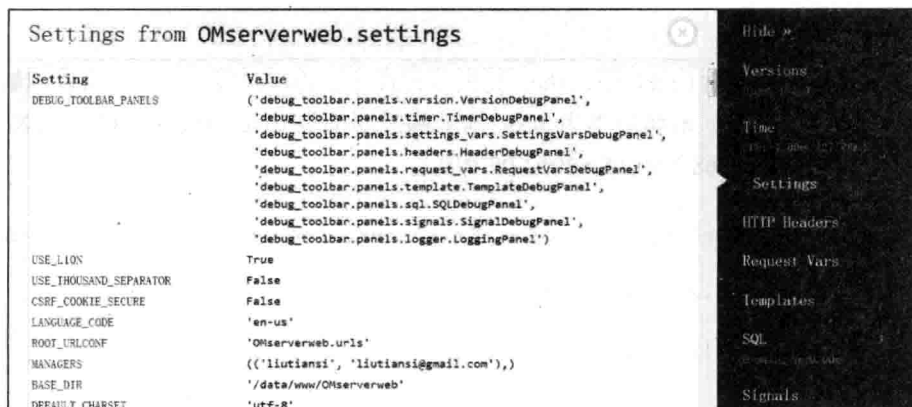


图 13-5 debug\_toolbar 界面

## (2) Django 源码自动重载 (reload) 方案

本方案结合 uwsgi 的 “--touch-reload” 参数来实现，参数格式：--touch-reload “文件”，即当该参数值指定的文件发生变化（修改或 touch 操作）时，uwsgi 进程将自动重载 (reload)，从而使我们的项目代码刷新生效。另外，如何保证一旦更新项目源码立即触发变更 --touch-reload 指定的文件？Linux 系统下的 inotify 可以做到这点，具体操作如下。

### 1) 在项目目录中创建一个监视文件：

```
# mkdir /data/www/OMserverweb/shell # 在项目目录中创建一个存放监视文件的目录 shell
# touch reload.set # 创建一个监视文件 reload.set
# yum -y install inotify-tools # 安装 inotify 程序包

# uwsgi 启动脚本添加 “--touch-reload” 项
# /usr/bin/uwsgi --ini "/usr/local/nginx/conf/*.ini" --touch-reload "/data/www/OMserverweb/shell/reload.set"
```

### 2) 编写监视脚本：

```
# vi /data/www/OMserverweb/shell/autoreload.sh

#!/bin/sh
objectdir="/data/www/OMserverweb"
# 启动 inotify 监视项目目录，参数 “--exclude” 为忽略的文件或目录正则
/usr/bin/inotifywait -mrq --exclude "(static|logs|shell|\.swp|\.swx|\.pyc|\.py|~)" --timefmt '%d/%m/%y %H:%M' --format '%T %w%f' --event modify,delete,move,create,attrib ${objectdir} | while read files
do
# 项目源码发生变化后，触发 touch reload.set 的操作，最终使 uwsgi 进程重载，达到刷新项目源码的目的
/bin/touch /data/www/OMserverweb/shell/reload.set
continue
```

```
done &
```

3) 启动脚本开启项目目录监视:

```
# /data/www/OMserverweb/shell/autoreload.sh
```

## 13.5 系统功能模块设计

### 13.5.1 前端数据加载模块

OMServer 平台的 Web 前端采用 prototype.js 作为默认 Ajax 框架, 通过 get 方式向定义好的 Django 视图发起请求, 功能视图通过 HttpResponse() 方法直接输出结果, 前端会将输出的结果做页面渲染。图 13-6 为应用 ID (app\_categId) 等于 1 的 HttpResponse() 输出结果, 前端会将这个结果串进行分割, 然后填充页面元素, 后端返回主机信息。

```
← → ↺ omserver.domain.com/autoadmin/server_list/?app_categId=1
192.168.1.10,192.168.1.20|192.168.1.10*sn2012-07-010,192.168.1.20*sn2013-08-020
```

图 13-6 后端返回主机信息

前端各区域对应的数据库表及视图方法见图 13-7。

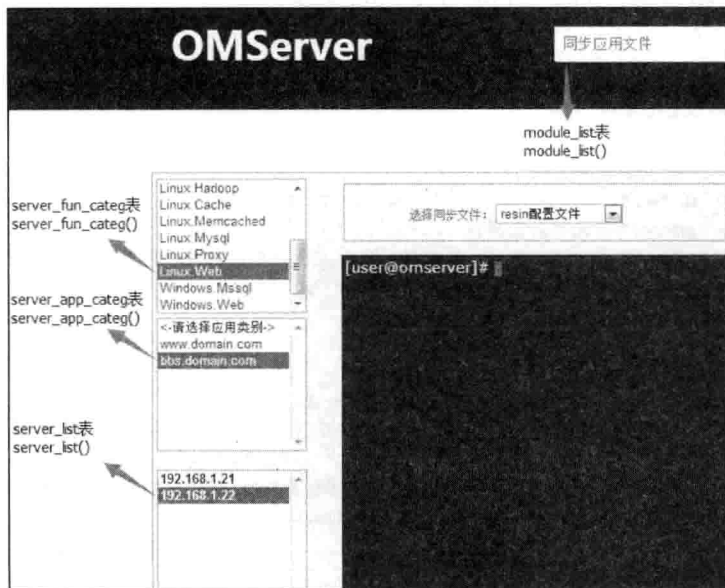


图 13-7 前端各区域对应后台方法及数据库表

局部方法代码如下：

【 /data/www/OMserverweb/autoadmin/views.py 】

```
"""
=Return server IP list
=返回服务器列表方法
"""
def server_list(request):
    ip=""
    ip_hostname=""

    if not 'app_categId' in request.GET:
        app_categId=""
    else:
        app_categId=request.GET['app_categId'] # 获取用户选择的应用分类 ID
        #ServerList 为 server_list 表模型对象，实现过滤获取的应用分类 ID 相匹配的主机列表
        ServerListObj = ServerList.objects.filter(server_app_id=app_categId)
        for e in ServerListObj:
            ip+=", "+e.server_lip
            ip_hostname+=", "+e.server_lip+"*"+e.server_name
        server_list_string=ip[1:]+ "|" +ip_hostname[1:]
        # 输出格式: 192.168.1.10,192.168.1.20|192.168.1.10*sn2012-07-010,\
        #192.168.1.20*sn2013-08-020, 其中 "|" 分隔符前部分为 IP 地址, 作为 HTML <option>
        # 下拉框显示项,
        # 分隔符后部分为 <option> 的 value, 以 "*" 号作为分隔符, 目的是为后端提供主机名及 IP 两种
        # 目标地址支持
        return HttpResponse(server_list_string)

"""
=Return module list
=返回功能模块列表方法
"""
def module_list(request):
    module_id="-1"
    module_name=u" 请选择功能模块 ..."
    # ModuleList 为 module_list 表模型对象，实现读取所有模块列表，以模块 id 做排序
    ModuleObj = ModuleList.objects.order_by('id')
    for e in ModuleObj:
        module_id+=", "+str(e.id)
        module_name+=", "+e.module_name
    module_list_string=module_name+"|" +module_id
    # 输出格式: "请选择功能模块...", 查看系统日志, 查看最新登录, 查看系统版本
    # -1,1001,1002,1003"
    # 其中 "|" 号分隔模块名称与模块 ID, Web 前端获取数据后通过 JavaScript 做拆分与组装
    return HttpResponse(module_list_string)
```

### 13.5.2 数据传输模块设计

传输模块采用 rpyc 分布式计算框架, 利用分布式特点可以实现多台主控设备的支持, 具备一定横向扩展及容灾能力。rpyc 分为两种角色, 一种为 Server 端, 另一种为 Client 端, 与传统的 Socket 工作方式一样, 区别是 rpyc 实现了更高级的封装, 支持同步与异步操作、回调和远程服务以及透明的对象代理, 可以轻松在 Server 与 Client 之间传递 Python 的任意对象, 在性能方面也非常高效。下面介绍的是 Django 的 module\_run() 视图方法, 实现接收功能模块的提交参数、加密、发送、接收功能模块运行结果等, 局部方法代码如下:

【/data/www/OMserverweb/autoadmin/views.py】

```
"""
= Run module
= 运行模块视图方法 (向 rpyc 服务器端发起任何请求)
"""
def module_run(request):
    import rpyc
    put_string=""

    if not 'ModuleID' in request.GET:      # 接收模块 ID、操作主机、模块扩展参数等 (更多源码已省略)
        Module_Id=""
    else:
        Module_Id=request.GET['ModuleID']
        put_string+=Module_Id+"@"
        .....

    try:
        conn=rpyc.connect('192.168.1.20',11511)    # 连接 rpyc 主控端主机, 端口: 11511
        # 调用 rpyc Server 的 login 方法实现账号、密码校验, 屏蔽恶意的连接
        conn.root.login('OMuser','KJS23o4ij09gHF734iuhdsfhkGYSihoiwhj38u4h')
    except Exception,e:
        logger.error('connect rpyc server error:'+str(e))
        return HttpResponse('connect rpyc server error:'+str(e))
    # 对请求数据串使用 tencode 方法进行加密, 密钥使用 Django 中 settings.SECRET_KEY 的值
    put_string=tencode(put_string,settings.SECRET_KEY)
    # 调用 rpyc Server 的 Runcommands 方法实现功能模块的任务下发, 返回的结果使用 tdecode 进行解密
    OPrsult=tddecode(conn.root.Runcommands(put_string),settings.SECRET_KEY)
    return HttpResponse(OPrsult)      # 输出结果供前端渲染
```

关于 rpyc 服务器端的实现原理, 首先接收 rpyc 客户端传递过来的信息, 通过解密方法还原出模块 ID、操作对象、模块扩展参数等信息, 再通过 exec 方法导入相应的功能模块 (要事先完成编写, 否则会提示找不到指定功能模块), 调用功能模块的相关方法, 实现操作任务向业务集群服务器下发与执行, 最后将任务执行结果串进行格式化、加密后返回给 Web 层。完整实现代码如下:

## 【 /home/test/OMServer/OMservermain.py 】

```

# -*- coding: utf-8 -*-
import time
import os,sys
import re
from cPickle import dumps
from rpyc import Service
from rpyc.utils.server import ThreadedServer
import logging
from libraries import *
from config import *
# 定义服务器端模块存放路径
sysdir=os.path.abspath(os.path.dirname(__file__))
sys.path.append(os.sep.join((sysdir,'modules/'+AUTO_PLATFORM)))

class ManagerService(Service):
    # 定义 login 认证方法, 对外开放调用的方法, rpyc 要求加上 “exposed_” 前缀, 调用时使用
    # login() 即可
    def exposed_login(self,user,passwd):
        if user=="OMuser" and passwd=="KJS23o4ij09gHF734iuhdsdfhkGYSihoiwhj38u4h":
            self.Checkout_pass=True      # 认证结果标记变量, 值为 “True” 则认证通过, 反之
                                         # 认证失败
        else:
            self.Checkout_pass=False

    def exposed_Runcommands(self,get_string):
        logging.basicConfig(level=logging.DEBUG,      # 启用系统日志记录
                            format='%(asctime)s [%(levelname)s] %(message)s',
                            filename=sys.path[0]+'logs/omsys.log',
                            filemode='a')
        # 判断是否通过认证
        try:
            if self.Checkout_pass!=True:
                return tencode("User verify failed!",SECRET_KEY)
        except:
            return tencode("Invalid Login!",SECRET_KEY)

        # 获取 rpyc Client 的请求串 get_string, 通过 tdecode 方法解密后再进行分隔, 分隔符为 “@@”
        self.get_string_array=tddecode(get_string,SECRET_KEY).split('@@')
        self.ModuleId=self.get_string_array[0]      # 获取功能模块 ID
        self.Hosts=self.get_string_array[1]         # 获取操作目标主机

        sys_param_array=[]      # 获取功能模块的扩展参数并追加到列表
        for i in range(2,len(self.get_string_array)-1):
            sys_param_array.append(self.get_string_array[i])
        # 加载模块 ID 应对的模块名, 格式为 “Mid_” + 模块 ID, 如 “Mid_1001.py”
        mid="Mid_"+self.ModuleId
        importstring = "from "+mid+" import Modulehandle"
        try:

```

```

        exec importstring
    except:
        return tencode(u"module\\"+mid+u"\ does not exist, Please add
it", SECRET_KEY)
    # 调用模块相关方法, 下发执行任务
    Runobj=Modulehandle(self.ModuleId,self.Hosts,sys_param_array)
    Runmessages=Runobj.run()
    # 根据不同主控端组件格式化输出, 支持 Func、Ansible、Saltstack
    if AUTO_PLATFORM=="func":
        if type(Runmessages) == dict:
            returnString = func_transform(Runmessages,self.Hosts)
        else:
            returnString = str(Runmessages).strip()

    elif AUTO_PLATFORM=="ansible":
        if type(Runmessages) == dict:
            returnString = ansible_transform(Runmessages,self.Hosts)
        else:
            returnString = str(Runmessages).strip()

    elif AUTO_PLATFORM=="saltstack":
        if type(Runmessages) == dict:
            returnString = saltstack_transform(Runmessages,self.Hosts)
        else:
            returnString = str(Runmessages).strip()
    # 对返回给 rpyc Client 的数据串进行加密
    return tencode(returnString, SECRET_KEY)
s=ThreadedServer(ManagerService,port=11511,auto_register=False)
s.start()    # 启动 rpyc 服务监听、接收、响应请求

```

数据传输的安全性关系到整个运营平台的生命线, 因此严格做好入侵安全防范至关重要。OMServer 平台采用 `base64.b64encode()`、`base64.b64decode()` 加上密钥混淆算法 (RC4) 实现数据的加密与解密。OMServer 平台遵循一个原则, 数据在传输之前调用 `tencode()` 方法进行加密, 在数据接收完毕后调用 `dencode()` 方法进行解密。解密的密钥采用项目 `settings.py` 中的 `SECRET_KEY` 变量值。同时在 `rpyc` 服务器端添加 `login()` 方法, 实现逻辑层的安全防护。

【 /home/test/OMServer/libraries.py 】

```

# -*- coding: utf-8 -*-
#!/usr/bin/env python
import random, base64
from hashlib import sha1
#RC4 加密算法
def crypt(data, key):
    x = 0
    box = range(256)
    for i in range(256):

```

```

        x = (x + box[i] + ord(key[i % len(key)])) % 256
        box[i], box[x] = box[x], box[i]
    x = y = 0
    out = []
    for char in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x]
        out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256]))
    return ''.join(out)
# 使用 RC4 算法加密编码后的数据, data 为加密的数据, key 为密钥
def tencode(data, key, encode=base64.b64encode, salt_length=16):
    """RC4 encryption with random salt and final encoding"""
    salt = ''
    for n in range(salt_length):
        salt += chr(random.randrange(256))
    data = salt + crypt(data, sha1(key + salt).digest())
    if encode:
        data = encode(data)
    return data
# 使用 RC4 算法解密编码后的数据, data 为加密的数据, key 为密钥
def tdecode(data, key, decode=base64.b64decode, salt_length=16):
    if decode:
        data = decode(data)
    salt = data[:salt_length]
    return crypt(data[salt_length:], sha1(key + salt).digest())

```

### 13.5.3 平台功能模块扩展

OMServer 平台模块的扩展需要完成两件事情，一是在前端添加模块基本信息，二是在服务器端编写对应的任务模块，下面对具体内容进行详细说明。

#### (1) 添加前端模块

添加前端模块包括指定模块名称、功能说明、模块扩展（HTML 表单作为模块参数）等，具体操作是点击首页的【添加模块】按钮，跳转到“添加模块”表单页面，其中最关键的是“模块扩展”输入框，支持所有 HTML 表单元素，后台通过 name 属性引用其值（value）。OMServer 目前支持最多两个扩展参数，name 属性要求使用“sys\_param\_1”、“sys\_param\_2”作为其定义值，当然，扩展更多参数的改造成本也非常低。在本示例中添加“重启进程服务”模块，具体操作如图 13-8 所示。

提交后将返回新增模块的 ID，该模块 ID 同时会作为服务器端任务模块的后缀名，如图 13-9 所示，记下模块 ID “1007”，前端模块添加完毕。



图 13-8 添加前端模块

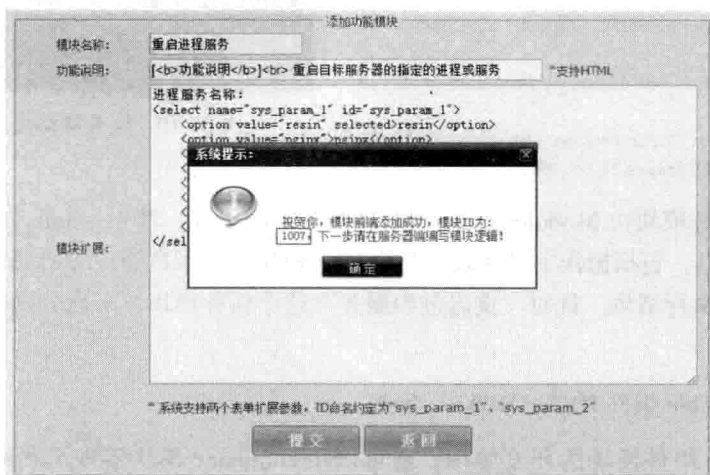


图 13-9 提交前端模块添加

## (2) 添加服务器端任务模块

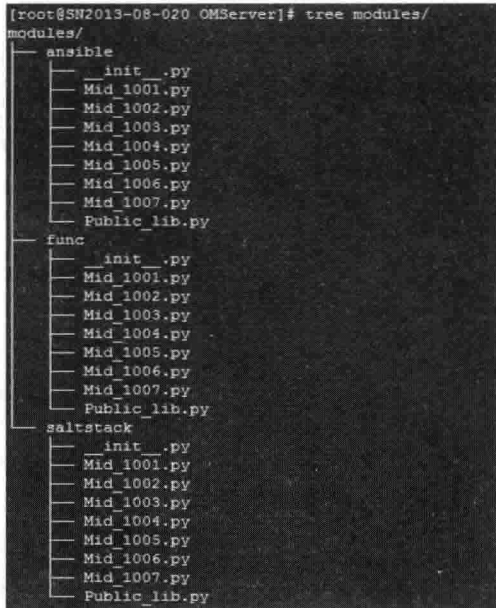
服务器端模块的作用是负责具体远程操作任务的功能封装, 支持 3 种 Python 自动化操作组件, 包括 Saltstack、Ansible、Func。不同组件的 API 语法及返回数据结构都不一样, 因此 OMServer 在设计时就将不同组件的模块进行隔离, 具体模块目录结构如图 13-10 所示, 在模块目录 (modules) 下组件名作为二级目录名, 二级目录下为具体的任务模块, 文件名称由 "Mid\_" + 模块 ID 组成, 与前端生成的模块 ID 进行关联。

关于任务模块的编写，不同组件的实现规范和方法都不一样，在编写任务模块之前需要更新配置文件 config.py 的两个选项，其中“`AUTO_PLATFORM`”为指定组件环境，可选项为“`ansible`”、“`saltstack`”、“`func`”，“`SECRET_KEY`”为指定加密、解密的密钥，与项目 settings.py 中的 `SECRET_KEY` 变量保持一致。另外 modules/(ansible|saltstack|func)/Public\_lib.py 文件的作用是导入、定义各组件的 API 模块包及全局参数，同时也增加代码的复用性。

【 /home/test/OMServer/config.py 】

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python
AUTO_PLATFORM = "saltstack"      # 指定组件环境，支持 Saltstack、Ansible、Func

# 密钥，与项目中 setting.py 的 SECRET_KEY 变量保持一致
SECRET_KEY = "ctmj#&8hrgow_`sj$ejt@9fzsmh_o)--(byt5jmg=e3#foya6u"
```



```
[root@SN2013-08-020 OMServer]# tree modules/
modules/
├── ansible
│   ├── init.py
│   ├── Mid_1001.py
│   ├── Mid_1002.py
│   ├── Mid_1003.py
│   ├── Mid_1004.py
│   ├── Mid_1005.py
│   ├── Mid_1006.py
│   ├── Mid_1007.py
│   └── Public_lib.py
├── func
│   ├── init.py
│   ├── Mid_1001.py
│   ├── Mid_1002.py
│   ├── Mid_1003.py
│   ├── Mid_1004.py
│   ├── Mid_1005.py
│   ├── Mid_1006.py
│   ├── Mid_1007.py
│   └── Public_lib.py
└── saltstack
    ├── init.py
    ├── Mid_1001.py
    ├── Mid_1002.py
    ├── Mid_1003.py
    ├── Mid_1004.py
    ├── Mid_1005.py
    ├── Mid_1006.py
    ├── Mid_1007.py
    └── Public_lib.py
```

图 13-10 服务器端模块目录结构

服务器端任务模块由 `Modulehandle` 类及其两个方法组成，其中 `__init__()` 方法作用为初始化模块基本信息，包括操作主机列表、模块 ID、模块扩展参数等；`run()` 方法实现组件 API 的调用以及返回执行结果。针对“重启进程服务”这个任务模块，下面分别介绍 3 个组件的不同实现方法。

#### 1) 编写 Ansible 组件 ID 为“1007”模块。

根据 Ansible 组件模块的开发原理，通过调用 `command` 模块实现远程命令执行，使用 `copy` 模块实现文件远程同步，详细源码如下：

【 /home/test/OMServer/modules/ansible/Mid\_1007.py 】

```
# -*- coding: utf-8 -*-
from Public_lib import *
# 重启应用模块进程服务 #
class Modulehandle():
    def __init__(self,moduleid,hosts,sys_param_row):      # 初始化方法无须改动
        self.hosts = ""
        self.Runresult = ""
        self.moduleid = moduleid      # 模块 ID
        self.sys_param_array= sys_param_row      # 模块扩展参数列表
        self.hosts=target_host(hosts,"IP")      # 格式化主机信息，参数“IP”为 IP 地址，“SN”为主机名
```

# 任务下发、执行方法

```
def run(self):
    try:
        # 根据模块扩展参数定义执行的不同命令集
        commonname=str(self.sys_param_array[0])
        if commonname=="resin":
            self.command="/etc/init.d/resin restart"
        elif commonname=="nginx":
            self.command="/etc/init.d/nginx restart"
        elif commonname=="haproxy":
            self.command="/etc/init.d/haproxy restart"
        elif commonname=="apache":
            self.command="/etc/init.d/httpd restart"
        elif commonname=="mysql":
            self.command="/etc/init.d/mysql restart"
        elif commonname=="lighttpd":
            self.command="/etc/init.d/lighttpd restart"
        # 调用 Ansible 提供的 API(command 模块), 执行远程命令
        self.Runresult = ansible.runner.Runner(
            pattern=self.hosts, forks=forks,
            module_name="command", module_args=self.command).run()
        if len(self.Runresult['dark']) == 0 and len(self.Runresult['contacted']) == 0:
            return "No hosts found, 请确认主机已经添加 ansible 环境! "
    except Exception,e:
        return str(e)
    return self.Runresult    # 返回执行结果
```

## 2) 编写 Saltstack 组件 ID 为 “1007” 模块。

根据 Saltstack 组件模块的开发原理, 通过调用 cmd() 方法配置 “cmd.run” 与 “cp.get\_file” 参数实现远程命令执行及文件远程同步, 详细源码 (部分) 如下:

【 /home/test/OMServer/modules/saltstack/Mid\_1007.py 】

```
def run(self):
    try:
        client = salt.client.LocalClient()
        .....
        # 调用 Saltstack 提供的 API(cmd.run 模块), 执行远程命令
        self.Runresult = client.cmd(self.hosts, 'cmd.run', [self.command], \
            expr_form='list')
        if len(self.Runresult) == 0:
            return "No hosts found, 请确认主机已经添加 saltstack 环境! "
    except Exception,e:
        return str(e)
    return self.Runresult    # 返回执行结果
```

## 3) 编写 Func 组件 ID 为 “1007” 模块。

根据 Func 组件模块的开发原理, 通过调用 client.command.run() 方法实现远程命令执行, 使用 client.copyfile.copyfile() 方法实现文件远程同步, 详细源码 (部分) 如下:

【 /home/test/OMServer/modules/func/Mid\_1007.py 】

```
def run(self):
    try:
        client = fc.Overlord(self.hosts)
        .....
        # 调用 Func 提供的 API (command.run 模块), 执行远程命令
        commonname=str(self.sys_param_array[0])
        self.Runresult=client.command.run(self.command)
    except Exception,e:
        return str(e)
    return self.Runresult      # 返回执行结果
```

任务模块编写完成后, 启动服务端服务, 运行以下命令:

```
# cd /home/test/OMServer
# python OMservermain.py &
```

最后, 打开浏览器访问 <http://omserver.domain.com>, 效果见图 13-11。



图 13-11 远程操作功能截图



参考  
提示

RC4 加密算法参考文章 <http://www.snip2code.com/Snippet/27937/Blockout-encryption-decryption-methods-p>。

## 打造 Linux 系统安全审计功能

随着互联网逐渐深入我们日常生活的方方面面，网络安全威胁也随之严重，比如服务器渗透、数据窃取、恶意攻击等。为了解决网络安全的问题，人们采取了各式各样的防护措施来保证网络或服务的正常运行，其中系统安全审计是记录入侵攻击主机的一个重要凭证：实时跟踪黑客的操作记录，可在第一时间监测到攻击者的行为，并让管理员采取相应的应对措施；同时也可作为日后攻击者的犯罪证据，为后续的审计工作提供数据依据，具有可靠性、完整性、不可抵赖等特点。本章将介绍在 OMServer 平台扩展 Linux 系统安全审计功能。

### 14.1 平台功能介绍

安全审计功能作为 OMServer 平台的一部分，扩展了 Linux 系统安全审计的功能，实现实时跟踪所有 Linux 服务器系统登录账号的操作记录，由于操作记录异地集中式存储，即使攻击者做了事后的操作痕迹清理也无济于事。该功能结合 Linux 系统的 history（命令行历史记录）工作机制实现，同时设置用户全局环境 `/etc/profile` 的 history 属性变量，实现定制系统用户实时触发事件，在该事件中加入 Python 编写的上报脚本，实现数据的实时跟踪，最后利用 OMServer 的前端作为实时输出展示，平台首页截图见图 14-1。

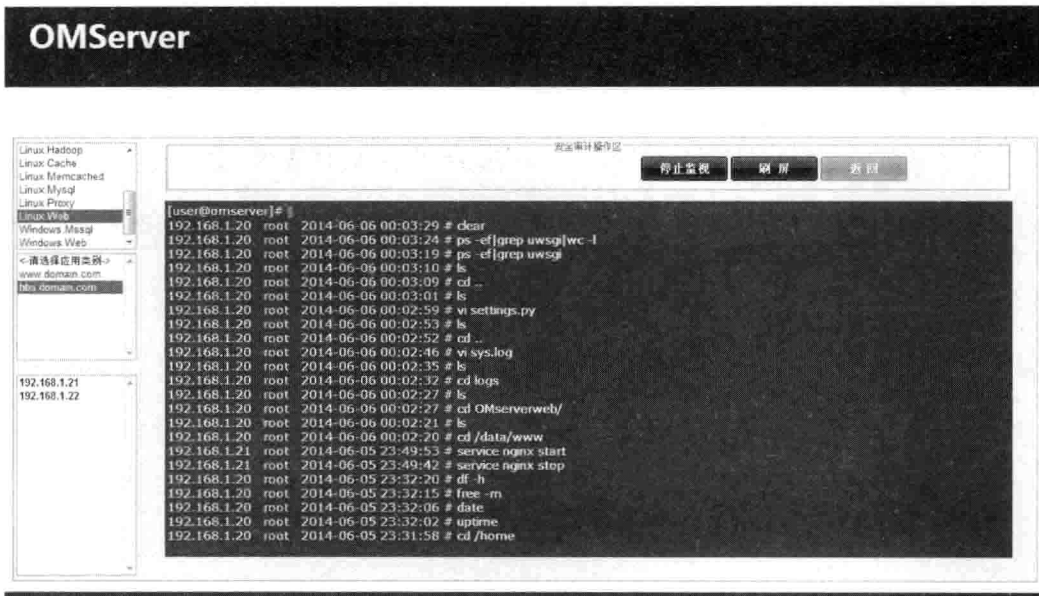


图 14-1 平台首页截图

## 14.2 系统架构设计

OMServer 平台安全审计的功能同样基于 B/S 结构，服务端的数据来源于业务集群 Agent 实时上报，使用 MySQL 数据库作为数据存储，客户端采用 prototype.js 前端架构实现数据同步展示，系统架构图见图 14-2。

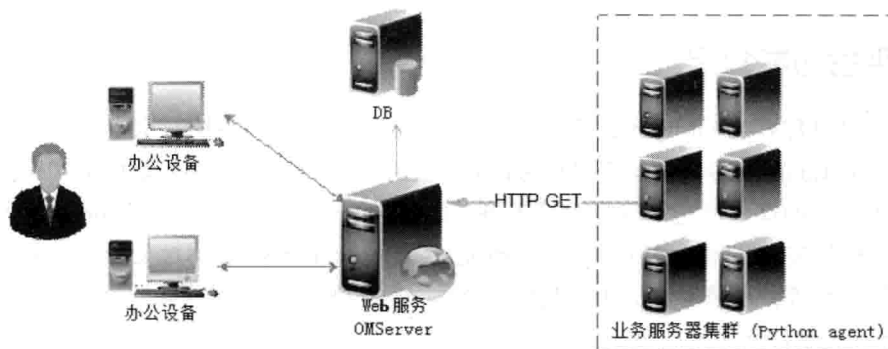


图 14-2 系统架构图

从图 14-2 中可以看出系统的整体架构，首先管理员在业务服务器集群部署 Python 数据

上报脚本，通过 OMServer 提供的 cgi 接口实现数据接收、入库，最后通过访问前端页面来查看、跟踪服务器上报的审计信息，整个流程结束。

## 14.3 数据库结构设计

### 14.3.1 数据库分析

安全审计服务器端功能是在 OMServer 平台上进行扩展，追加一张 server\_history 表，用于操作事件信息的存储，且与 server\_list 的 IP 字段配置外键关联。表信息说明如下。

□ server\_history：操作事件表。

□ server\_list：服务器列表。

### 14.3.2 数据字典

server\_list 服务器列表。

字段名	数据类型	默认值	允许非空	自动递增	备注
server_name	char(13)		NO		主机名称
server_wip	char(15)		NO		主机外网 IP
server_lip	char(12)		NO		主机内网 IP
server_op	char(10)		NO		主机操作系统
server_app_id	int(11)		NO		服务应用分类 ID

server\_history 操作事件表。

字段名	数据类型	默认值	允许非空	自动递增	备注
ID	int(11)		NO	是	主键 ID
history_id	int(11)		NO		事件 ID
history_ip	char(15)		NO		事件 IP 地址
history_user	char(15)		NO		事件用户名
history_datetime	datetime		NO		事件时间
db_datetime	timestamp	CURRENT_TIMESTAMP	NO		入库时间
history_command	char(255)		NO		事件命令

数据库模型功能沿用了 OMServer 系统中主机表 (server\_list) 的层次结构，且追加了操作事件表 (server\_history)，其中，将表 server\_history 的 history\_ip 字段设置成外键，与 server\_list 表中的 server\_lip 字段进行关联，详细见图 14-3 的数据库模型。

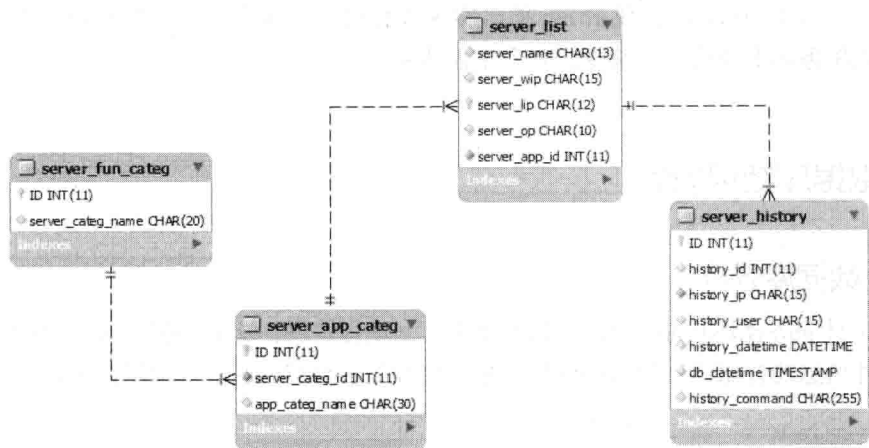


图 14-3 平台数据库模型

## 14.4 系统环境部署

### 14.4.1 系统环境说明

系统安全审计功能的服务器端作为 OMServer 项目的 App 存在，关于服务器端 Web Server 的环境搭建本节将不再说明。为了便于读者理解，下面对上报主机系统环境配置、Agent 与服务器端 Python 实现方法进行详细说明，环境设备角色如表 14-1 所示。

表 14-1 系统环境说明表

角色	主机名	IP	环境说明
WEBServer	SN2012-07-010	192.168.1.10	Django+uwsgi+MySQL 环境
AppServer	SN2012-07-021	192.168.1.21	Python 2.4 或以上
AppServer	SN2012-07-022	192.168.1.22	Python 2.4 或以上

### 14.4.2 上报主机配置

系统安全审计功能主机上报需要完成两个任务，一为配置用户 profile，二为编写上报 Python 脚本，下面一一进行说明。

#### （1）系统用户环境配置

通过配置 Linux profile 的 history 相关变量来实现与安全审计功能的对接，包括指定系统账号 history 存放路径、存储长度、扩展信息、PROMPT\_COMMAND 事件等，更多见以下配

置及含义说明。

```
# vi /etc/profile
# 追加以下配置
#add by OMAudit
export HISTFILE=$HOME/.bash_history      # 指定用户 history 日志存放路径
export HISTSIZE=1200                     # 指定 history 命令输出的记录数
export HISTFILESIZE=1200                 # 指定历史记录文件 .bash_history 的最大存储行数
export HISTCONTROL=ignoredups            # 不记录连续重复的命令
export HISTTIMEFORMAT="%F %T "           # history 命令显示当前记录的用户与时间，例如：
# "root 2014-06-05 23:32:16 free -m"

# PROMPT_COMMAND 变量最为核心，实现了指定内容在出现 bash 提示符前执行的功能；
# "history -a" 将目前新增的 history 命令写入 histfiles 中；"history -c" 删除记录的所有命令（仅内存）；
# "history -r" 将 histfiles 的内容读到内存中，即可以通过 history 查看；
# "/home/test/OMAudit/OMAudit_agent.py $(history 1)" 通过 $(history 1) 获取最后一条命令，且作为参数传递给 OMAuditmain.py 脚本，做后续的命令数据信息上报
export PROMPT_COMMAND="history -a; history -c; history -r;"/home/test/OMAudit/OMAudit_agent.py $(history 1)'
shopt -s histappend                      # 历史清单将以添加形式加入 HISTFILE 变量指定的文件，而不是覆盖

typeset -r PROMPT_COMMAND                # 设置环境变量只读，提高安全性
typeset -r HISTTIMEFORMAT
```

保存配置后使其生效，运行“source /etc/profile”命令，profile 环境配置完成。

## （2）客户端上报脚本

客户端上报脚本的作用是将接收的最新 Linux 命令“\$(history 1)”及服务器相关信息提交到 OMServer 主机，其中 config.py 为上报 agent 的配置文件，涉及三个选项，详细说明如下：

### 【 /home/test/OMAudit/config.py 】

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python

Net_driver = "eth0"      # 为便于记录上报来源主机，获取指定网卡驱动的 IP 地址
OMServer_address = "omserver.domain.com" #OMServer 服务器端地址，作为上报的目的
Connect_TimeOut = 3      # 指定上报超时时间，单位为秒
```

OMAudit\_agent.py 作为主上报 agent 程序，负责信息的上报，采用了 httplib 模块作为 HTTP 客户端，详细源码及说明如下：

### 【 /home/test/OMAudit/OMAudit\_agent.py 】

```
#!/usr/bin/env python
```

```

#coding:utf-8
import sys
import socket
import fcntl
import struct
import logging
from config import *
import urllib,httplib
socket.setdefaulttimeout(Connect_TimeOut)      # 设置全局 Socket 超时时间 (覆盖 HTTP 连接超时)

logging.basicConfig(level=logging.DEBUG,        # 启用日志记录
                    format='%(asctime)s [%(levelname)s] %(message)s',
                    filename=sys.path[0]+'/omsys.log',
                    filemode='a')

# 对 $(history 1) 信息进行合法校验, 少于 6 个参数则报错, 正确的格式为 "173 root 2014-06-07 22:05:56 ls"
if len(sys.argv)<6:
    logging.error('History not configured in /etc/profile!')
    sys.exit()

def get_local_ip(ethname):      # 获取本地 IP 地址函数, 用来确认数据来源
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        addr = fcntl.ioctl(sock.fileno(), 0x8915, struct.pack('256s', ethname))
        return socket.inet_ntoa( addr[20:24] )
    except Exception,e:
        logging.error('get localhost IP address error:'+str(e))
        return "127.0.0.1"

def pull_history(http_get_param=""):      # 数据上报函数
    try:
        # 与 OMServer 服务器建立 HTTP 连接, 指定超时时间
        http_client =httplib.HTTPConnection(OMServer_address, 80, timeout=Connect_TimeOut)
        http_client.request("GET", http_get_param)      # 发起 GET 请求
        response =http_client.getresponse()      # 获取 HTTP 返回对象

        if response.status != 200:      # 非 HTTP 200 状态则退出
            logging.error('response http status error:'+str(response.status))
            sys.exit()

        http_content=response.read().strip()      # 返回字符串非 "OK" 则退出
        if http_content != "OK":
            logging.error('response http content error:'+str(http_content))
            sys.exit()

    except Exception, e:
        logging.error('connection django-cgi server error:'+str(e))
        sys.exit()

```

```

finally:
    if http_client:
        http_client.close()
    else:
        logging.error('connection django-cgi server unknown error.')
        sys.exit()

Sysip = get_local_ip(Net_driver)      # 调用获取本地 IP 函数
SysUser = sys.argv[2]                 # 获取 history 信息中的系统用户
History_Id = sys.argv[1]              # 获取 history ID 信息
History_date = sys.argv[3]            # 获取 history 日期信息
History_time = sys.argv[4]            # 获取 history 时间信息
History_command = ""

for i in range(5, len(sys.argv)):      # 获取 history 的系统命令信息
    History_command+= sys.argv[i]+" "

# 合并所有信息的 HTTP GET 参数格式, 部分信息使用 urllib.quote 进行 URL 编码
s= "/omaudit/omaudit_pull/?history_id="+History_Id+"&history_ip="+Sysip+"&history_user="+SysUser+ \
"&history_datetime="+History_date+urllib.quote(" ") +History_time+"&history_command="+urllib.quote(History_command.strip())

pull_history(s)      # 调用数据上报函数

```

添加 “/home/test/OMAAudit/OMAAudit\_agent.py” 可执行权限, 执行以下 chmod 命令, 客户端上报 agent 部署完毕。接下来使用 SSH 工具登录 Linux 服务器, 输入的任何 shell 命令都会即时同步到服务器端, 见图 14-4。

```
# chmod +x/home/test/OMAAudit/OMAAudit_agent.py
```

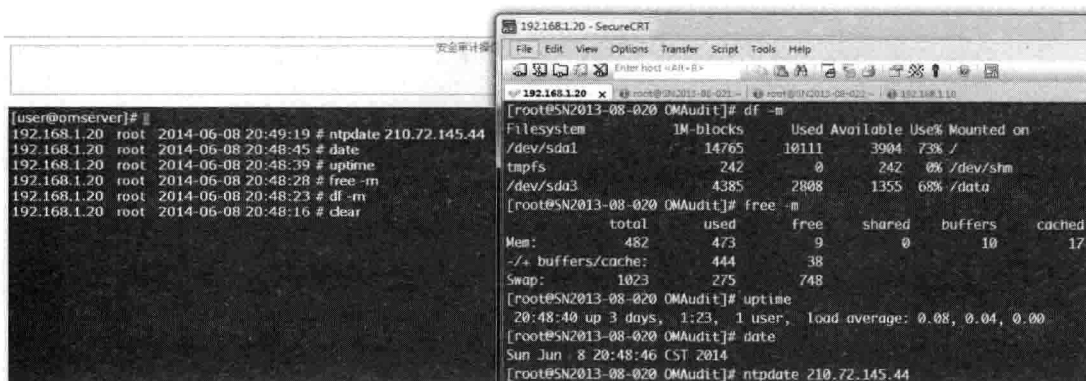


图 14-4 系统命令即时上报并展示

## 14.5 服务器端功能设计

### 14.5.1 Django 配置

安全审计功能作为 OMServer 的一个功能扩展，需要 Web 服务器端开发框架（Django）同样做些变更来支持新增的功能。由于该功能作为项目的一个 App，因此，第一步需要创建一个 App，操作如下：

```
# cd /data/www/OMserverweb
# python manage.py startapp omaudit
```

在创建的 omaudit 目录中修改 urls.py，添加 App 的 URL 映射规则，内容如下：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('omaudit.views',
    (r'^$', 'index'),
    (r'omaudit_pull/$', 'omaudit_pull'), # 映射到 omaudit_pull 方法，实现客户端数据接收
    (r'omaudit_run/$', 'omaudit_run'), # 映射到 omaudit_run 方法，实现前端实时查询
)
```

修改 App 的 models.py，实现与数据库的关系映射，内容如下：

```
from django.db import models

# Create your models here.
class ServerHistory(models.Model):
    id = models.IntegerField(primary_key=True, db_column='ID') # Field name
    made_lowercase.
    history_id = models.IntegerField()
    history_ip = models.CharField(max_length=45)
    history_user = models.CharField(max_length=45)
    history_datetime = models.DateTimeField()
    db_datetime = models.DateTimeField()
    history_command = models.CharField(max_length=765)
    class Meta:
        db_table = u'server_history'
```

如果数据库结构已经存在，可以通过 `python manage.py inspectdb` 命令来生成 models 代码。

最后修改项目 settings.py，注册该 App 名称，内容如下：

```
INSTALLED_APPS = (
    ...
    # 'django.contrib.admindocs',
    'public',
    'autoadmin',
)
```

```
'omaudit',      # 添加此行, 注册该 App
)
```

### 14.5.2 功能实现方法

服务器端提供了两个关键视图方法, 分别实现前端实时展示 (omaudit\_run) 及数据接收 (omaudit\_pull), 下面针对两个方法进行说明。

#### (1) 前端实时展示 (omaudit\_run) 方法

关于前端数据实时展示的实现原理, 通过前端 JavaScript 的 setInterval() 方法实现定时函数调用, 首次请求默认返回 ID 倒序最新 5 条记录, 并记录下 LastID (最新记录 ID), 后面的定时调用将传递 LastID 参数, 数据库查询条件是 “ID>LastID”, 从而达到实时获取最新记录的目的, 同时也支持选择主机来作为过滤条件, 功能实现流程图见图 14-5。

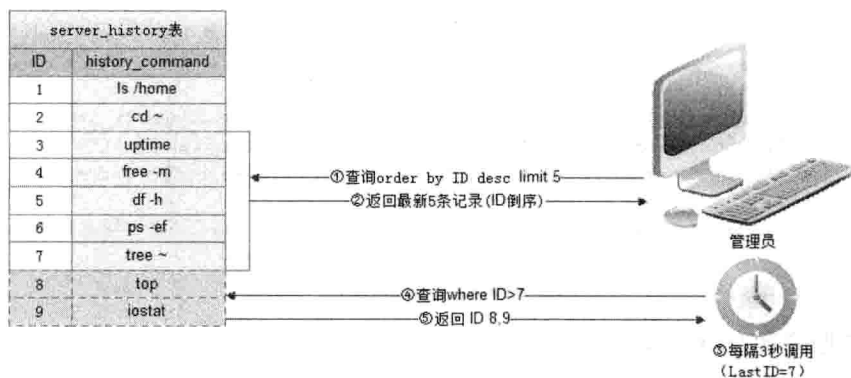


图 14-5 前端数据展示流程图

omaudit\_run() 方法实现源码如下:

```
"""
= 事件任务前端展示方法
"""
def omaudit_run(request):
    if not 'LastID' in request.GET:      # 获取上次查询到的最新记录 ID
        LastID=""
    else:
        LastID=request.GET['LastID']
    if not 'hosts' in request.GET:      # 获取选择的主机地址信息
        Hosts=""
    else:
        Hosts=request.GET['hosts']
    ServerHistory_string=""
```

```
host_array=target_host(Hosts,"IP").split(';') # 调用 target_host 方法过滤出 IP 地址

if LastID=="0": # 符合第一次提交条件，查询不加 "id>LastID" 条件，反之
    if Hosts=="": # 符合没有选择主机条件，查询不加 "history_ip in host_array"
        # 条件，反之
        ServerHistoryObj = ServerHistory.objects \
            .order_by('-id')[ :5]
    else:
        ServerHistoryObj = ServerHistory.objects \
            .filter(history_ip__in=host_array).order_by('-id')[ :5]
else:
    if Hosts=="":
        ServerHistoryObj = ServerHistory.objects \
            .filter(id__gt=LastID).order_by('-id')
    else:
        ServerHistoryObj = ServerHistory.objects \
            .filter(id__gt=LastID,history_ip__in=host_array).order_by('-id')

lastid=""
i=0
for e in ServerHistoryObj: # 遍历查询结果，返回给前端
    if i==0:
        lastid=e.id
    ServerHistory_string+=""+e.history_ip+ \
        "</font>&nbsp;&nbsp;&nbsp;\t"+ e.history_user+"&nbsp;&nbsp;&nbsp;\t"+ \
        str(e.db_datetime)+"\t # <font color=#ffffff>"+e.history_command+"</font>*"
    i+=1
ServerHistory_string+="@"+str(lastid) # 通过 “@@” 字符分隔事件记录与 lastid,
# 前端拆分

return HttpResponse(ServerHistory string)
```

## (2) 数据接收 (omaudit\_pull) 方法

数据接收方法相对比较简单，即将接收到的信息直接入库，实现的源码如下：

```
"""
= 事件任务 pull 方法
"""

def omaudit_pull(request):
    if request.method == 'GET':          # 校验 HTTP (GET) 请求参数合法性
        if not request.GET.get('history_id', ''):
            return HttpResponse("history_id null")
        if not request.GET.get('history_ip', ''):
            return HttpResponse("history_ip null")
        if not request.GET.get('history_user', ''):
            return HttpResponse("history_user null")
        if not request.GET.get('history_datetime', ''):
            return HttpResponse("history_datetime null")
        if not request.GET.get('history_command', ''):
            return HttpResponse("history_command null")
```

```

history_id=request.GET['history_id']    # 获取 HTTP 请求参数值
history_ip=request.GET['history_ip']
history_user=request.GET['history_user']
history_datetime=request.GET['history_datetime']
history_command=request.GET['history_command']

historyobj = ServerHistory(history_id=history_id, \    # 数据入库 (insert)
    history_ip=history_ip, \
    history_user=history_user, \
    history_datetime=history_datetime, \
    history_command=history_command)
try:
    historyobj.save()
except Exception,e:
    return HttpResponse(" 入库失败, 请与管理员联系! "+str(e))

Response_result="OK"    # 输出 "OK" 字符作为成功标志
return HttpResponse(Response_result)
else:
    return HttpResponse(" 非法提交! ")

```

当然, 如接入的集群过于庞大, 服务器端数据库会逐步形成瓶颈, 主机审计信息入库会出现一定延时, 影响 Linux 用户操作体验, 一个可行的方案是采用信息异步入库, 即先将信息写入本地, 再通过上报程序后台提交信息, 用户将无感知。

## 构建分布式质量监控平台

中国互联网呈多运营商并存的发展格局，一个注重用户体验的业务平台，在上线前就必须解决多运营商互联互通的问题，例如电信、联通、移动等网络的接入。目前有两个常见方案，一是将服务器资源放置不同运营商的 IDC，二是直接接入支持 BGP 协议的 IDC。在此之后，我们还要考虑监控不同运营网络访问业务平台的质量问题，例如骨干网络路由延时或调度不合理甚至网络故障，导致访问业务网络出现延时、丢包等现象，影响用户体验。因此，必须提供一种分布式（多运营商支持）的业务服务质量监控机制。本章通过实现一套分布式的质量监控平台整体进行说明。

### 15.1 平台功能介绍

分布式质量监控平台实现了多个数据采集点（不同运营商链路）对 Web 业务平台进行探测，采集的信息包括 DNS 解析时间、建立连接时间、准备传输时间、开始传输时间、传输总时间、HTTP 状态、下载数据包大小、下载速度等，覆盖了 HTTP 请求的整个生命周期。分析这些数据，可以帮助我们快速发现（异常告警）、定位访问业务延时过大问题。通过 RRDTOOL 做数据报表展示，报表类型包括请求响应时间、下载速度、可用性的自定义、日、月、年等，可以让管理员了解业务服务质量的整体趋势，平台截图见图 15-1。

## 业务质量监控

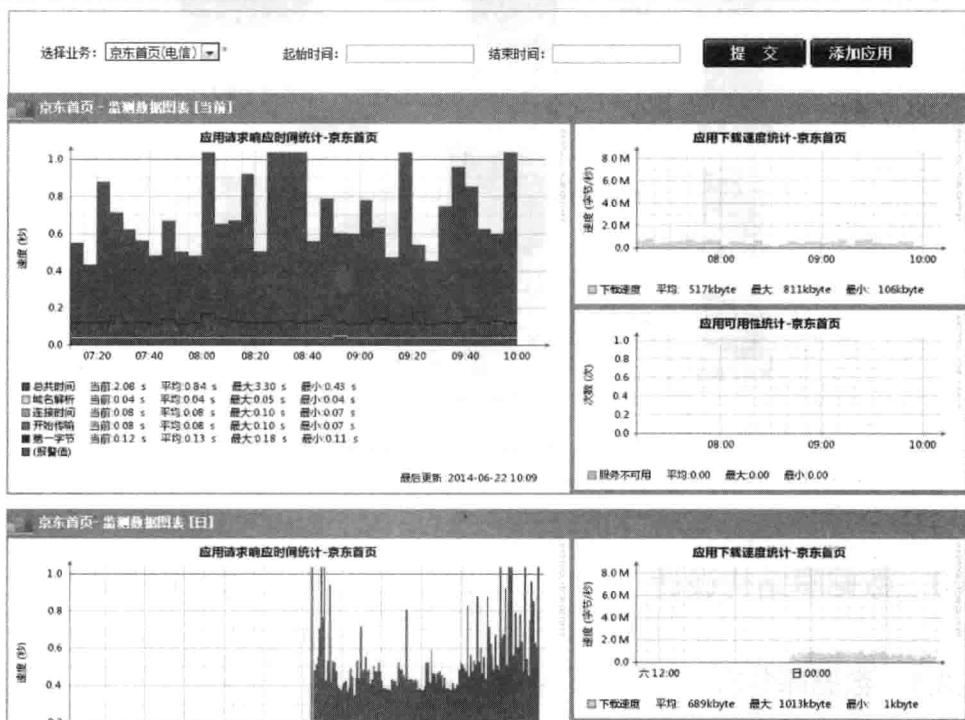


图 15-1 平台首页截图

## 15.2 系统构架设计

分布式质量监控平台由三种不同功能角色组成,第一种为数据采集探测功能,采用 Python+pycurl 模块实现数据的采集并入库 MySQL;第二种为后台定时 rrdtool 作业,实现 MySQL 数据导出并更新 RRDTOOL,采用了 Python+rrdtool 模块实现;第三种为 Web 报表展示,采用 Django+MySQL+rrdtool 模块实现,服务器端采用了 Nginx+uwsgi 构建高效的 Web 服务,根据管理员发起的请求条件输出不同类型的报表。系统架构图见图 15-2。

从图 15-2 中可以看出系统的整体架构,首先通过不同采集点向业务服务集群发起定时探测任务,将获取的响应数据入库 MySQL,异常返回信息将触发告警。功能模块定时从 MySQL 数据库拉取数据做 rrdtool update 操作,为后续的报表输出提供数据支持。最后管理员通过前端 Web 页面查询、定制输出报表,整个流程结束。

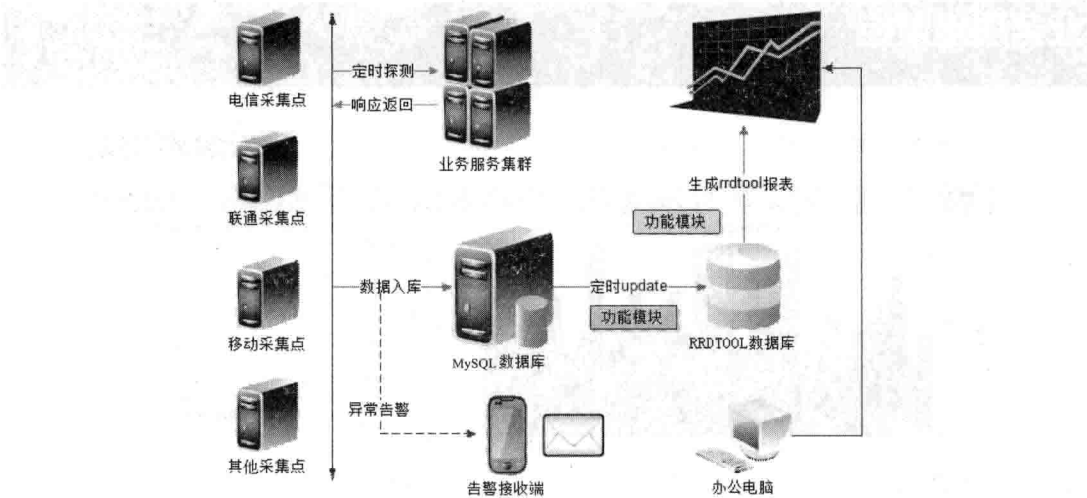


图 15-2 系统架构图

## 15.3 数据库结构设计

### 15.3.1 数据库分析

分布式质量监控平台有两张数据库表，分别为 webmonitor\_hostinfo 及 webmonitor\_monitordata 表，其中 webmonitor\_monitordata 的 FID 字段配置外键关联，表信息说明如下：

- ❑ webmonitor\_hostinfo：业务信息表
- ❑ webmonitor\_monitordata：采集数据信息表

### 15.3.2 数据字典

webmonitor\_hostinfo 业务信息表。

段名	数据类型	默认值	允许非空	自动递增	备注
ID	int(11)		NO	是	业务 ID
AppName	char(20)		NO		业务名称
URL	char(100)		NO		探测 URL
IDC	char(10)		NO		探测点
Alarmtype	char(10)		NO		告警类型
Alarmconditions	char(20)		NO		告警条件

webmonitor\_monitordata 采集数据信息表。

字段名	数据类型	默认值	允许非空	自动递增	备注
ID	int(11)		NO	是	探测结果 ID
FID	int(11)		NO		业务 ID
NAMELOOKUP_TIME	double		NO		DNS 解析时间
CONNECT_TIME	double		NO		建立连接时间
PRETRANSFER_TIME	double		NO		准备传输时间
STARTTRANSFER_TIME	double		NO		开始传输时间
TOTAL_TIME	double		NO		传输总时间
HTTP_CODE	char(80)		NO		HTTP 状态或异常信息
SIZE_DOWNLOAD	int(6)		NO		下载数据包大小
HEADER_SIZE	smallint(6)		NO		HTTP 头大小
REQUEST_SIZE	smallint(6)		NO		请求包大小
CONTENT_LENGTH_DOWNLOAD	smallint(6)		NO		下载内容长度
SPEED_DOWNLOAD	int(6)		NO		下载速度
DATETIME	int(11)		NO		探测时间
MARK	enum('0','1')		NO		更新 RRDTOOL 标记

### 15.3.3 数据库模型

从图 15-3 的数据库 EER 图可以看出, 表 webmonitor\_monitordata 的 FID 字段被设置为外键, 与 webmonitor\_hostinfo 表中的 ID 字段进行关联, 作为采集信息数据与业务信息表的唯一关联。

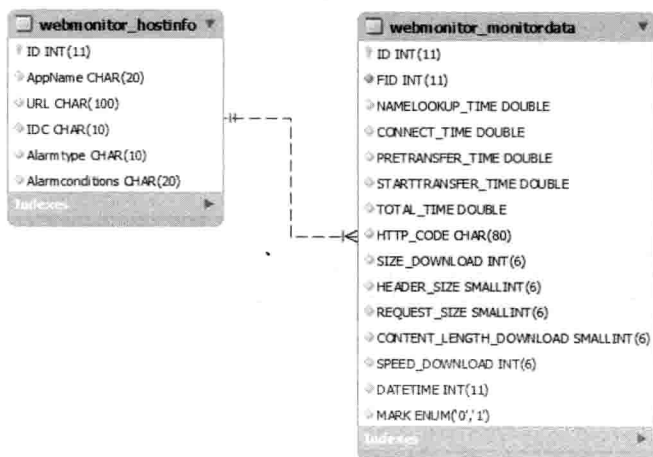


图 15-3 系统数据库模型

## 15.4 系统环境部署

### 15.4.1 系统环境说明

前面介绍了分布式质量监控平台的三种角色，为了便于读者理解，下面对不同角色的环境、实现方法进行详细说明，环境设备角色表如表 15-1 所示。

表 15-1 系统环境说明表

角色	主机名	IP	环境说明
Web Server	SN2012-07-010	192.168.1.10	Django+uwsgi+rrdtool+MySQL 环境
rrdtool 作业	SN2012-07-010	192.168.1.10	Python 2.6+rrdtool
数据采集 (电信)	SN2013-08-020	192.168.1.20	Python 2.6+pycurl
数据采集 (联通)	SN2013-08-021	192.168.1.21	Python 2.6+pycurl

### 15.4.2 数据采集角色

数据采集功能角色需要完成两个任务：一为采集远程业务服务集群 HTTP 响应数据，并将数据写入远程 MySQL 数据库；二为提供异常 HTTP 响应告警支持。本示例部署 192.168.1.20、192.168.1.21 主机，分别模拟电信与联通网络。下面详细说明。

数据采集端只有两个 Python 文件，一个为 config.py，其定义了数据库信息、运营商网络代码、连接超时时间等，内容如下：

【 /data/detector/config.py 】

```
# -*- coding: utf-8 -*-
# 定义 MySQL 数据信息
DBNAME='WebMonitor'
DBUSER='webmonitor_user'
DBPASSWORD='SKJDH3745tgDTS'
DBHOST='192.168.1.10'

# 修改成探测运营商网络代码 (重要)
# settings.py 中定义 IDC={'ct':'电信','cnc':'联通','cmcc':'移动'}
# “ct”代表电信探测点网络，联通网络修改成“cnc”，移动网络修改成“cmcc”，其他类似
IDC="ct"

# 连接的等待时间
CONNECTTIMEOUT = 5

# 请求超时时间
TIMEOUT = 10
```

```
# 告警邮件地址
MAILTO="user1@domain.com,user2@domain.com"
# 告警手机号
MOBILETO="136****3463"
```

另一个为提供业务服务质量采集功能的 `runmonitor.py`，采用了 `pycurl` 模块实现，通过定义 `setopt()` 方法定量参数，模拟一个 HTTP 请求器（request），也可以理解成一个简单的浏览器。再通过 `getinfo()` 定义的定量获取 HTTP 返回结果（response），采集的数据将即时入库，异常响应将触发告警，关键代码如下：

【 /data/detector/runmonitor.py 】

```
.....
Curlobj = pycurl.Curl()      # 创建 Curl 对象
Curlobj.setopt(Curlobj.URL, url)    # 定义请求的 URL
# 定义 setopt 请求器常量，各参数详细说明见 2.4 节
Curlobj.setopt(Curlobj.CONNECTTIMEOUT, CONNECTTIMEOUT)
Curlobj.setopt(Curlobj.TIMEOUT, TIMEOUT)
Curlobj.setopt(Curlobj.NOPROGRESS, 0)
Curlobj.setopt(Curlobj.FOLLOWLOCATION, 1)
Curlobj.setopt(Curlobj.MAXREDIRS, 5)
Curlobj.setopt(Curlobj.OPT_FILETIME, 1)
Curlobj.setopt(Curlobj.NOPROGRESS, 1)
bodyfile = open(os.path.dirname(os.path.realpath(__file__))+"/_body", "wb")
Curlobj.setopt(Curlobj.WRITEDATA, bodyfile)
Curlobj.perform()
bodyfile.close()
# 定义 getinfo 响应返回常量，各参数详细说明见 2.4 节
self.NAMELOOKUP_TIME=Decimal(str(round(Curlobj.getinfo(Curlobj.NAMELOOKUP_
TIME), 2)))
self.CONNECT_TIME=Decimal(str(round(Curlobj.getinfo(Curlobj.CONNECT_TIME), 2)))
self.PRETRANSFER_TIME=Decimal(str(round(Curlobj.getinfo(Curlobj.PRETRANSFER_
TIME), 2)))
self.STARTTRANSFER_TIME=Decimal(str(round(Curlobj.getinfo(Curlobj.
STARTTRANSFER_TIME), 2)))
self.TOTAL_TIME = Decimal(str(round(Curlobj.getinfo(Curlobj.TOTAL_TIME), 2)))
self.HTTP_CODE = Curlobj.getinfo(Curlobj.HTTP_CODE)
.....
```

最后，配置系统 `crontab`，5 分钟作一次数据采集，内容如下：

```
* /5 * * * * /usr/bin/python /data/detector/runmonitor.py > /dev/null 2>&1
```

### 15.4.3 rrdtool 作业

`rrdtool` 作业实现从 MySQL 导出数据并更新到 `rrdtool` 中，以便为后面的 `rrdtool` 报表功

能提供数据支持。具体方法是通过查询 webmonitor\_monitordata 表字段 MARK 为 '0' 的记录, 再将数据通过 rrdtool.updatev() 方法做 rrdtool 更新, 最后更新数据库标志 MARK 为 '1'。rrdtool 作业部署在任一台安装 rrdtool 模块的主机上即可, 本示例的 rrdtool 作业与 Web Server 部署在同一台主机上。部分关键源码如下:

【 /data/www/Servermonitor/webmonitor/updaterrd.py 】

```
def updateRRD(self, rowobj):          # 更新 rrd 文件方法
    if str(rowobj["HTTP_CODE"])=="200":    # 非 HTTP200 状态标志 "1"
        unavailablevalue=0
    else:
        unavailablevalue=1
    FID=rowobj["FID"]
    time_rrdpath=RRDPATH+'/'+str(self.getURL(FID))+'/'+str(FID)+'_'+\
    str(self.rrdtype[0])+'.rrd'          # 指定三个特性数据 rrdtool 文件位置
    download_rrdpath=RRDPATH+'/'+str(self.getURL(FID))+'/'+str(FID)+'_'+\
    str(self.rrdtype[1])+'.rrd'
    unavailable_rrdpath=RRDPATH+'/'+str(self.getURL(FID))+'/'+str(FID)+'_'+\
    str(self.rrdtype[2])+'.rrd'
    try:    # 将查询的 MySQL 记录更新到 rrd 文件
        rrdtool.updatev(time_rrdpath, '%s:%s:%s:%s:%s' % (str(rowobj["DATETIME"])\
        , str(rowobj["NAMELOOKUP_TIME"]), str(rowobj["CONNECT_
        TIME"]), str(rowobj["PRETRANSFER_TIME"]), str(rowobj["STARTTRANSFER_
        TIME"]), str(rowobj["TOTAL_TIME"])))
        rrdtool.updatev(download_rrdpath, '%s:%s' % (str(rowobj["DATETIME"]), \
        str(rowobj["SPEED_DOWNLOAD"])))
        rrdtool.updatev(unavailable_rrdpath, '%s:%s' % (str(rowobj
        ["DATETIME"])\
        , str(unavailablevalue)))
        self.setMARK(rowobj["ID"])      # 更新数据库标志
    except Exception, e:
        logging.error('Update rrd error:'+str(e))

def setMARK(self, _id):              # 更新已标志记录方法
    try:
        self.cursor.execute("update webmonitor_monitordata set \
        MARK='1' where ID='%s'%(_id)")
        self.conn.commit()
    except Exception, e:
        logging.error('SetMark database error:'+str(e))

def getNewdata(self):                # 获取未标志的新记录方法
    try:
        self.cursor.execute("select ID, FID, NAMELOOKUP_TIME, CONNECT_
        TIME, PRETRANSFER_TIME, STARTTRANSFER_TIME, TOTAL_TIME, HTTP_CODE, SPEED_
        DOWNLOAD, DATETIME from webmonitor_monitordata where MARK='0'")
```

```

        for row in self.cursor.fetchall():
            self.updateRRD(row)
    except Exception,e:
        logging.error('Get new database error:'+str(e))

```

同目录下的 config.py 为 rrdtool 作业配置文件，定义了数据库连接信息及项目路径等信息，可根据实际情况相应修改，最后配置系统 crontab，建议与采集同一执行频率，如每 5 分钟，内容如下：

```

*/5 * * * * /usr/bin/python /data/www/Servermonitor/webmonitor/updaterrrd.py > /dev/null 2>&1

```

## 15.5 服务器端功能设计

服务器端以 Web 的形式作为服务平台，以 Django 作为开发框架，结合 rrdtool 模块实现了业务添加（rrdtool create）、报表绘图（rrdtool graph）等功能。另外，项目改用直接操作 SQL 方式来代替 Django 的 ORM，为熟悉 SQL 的人员提供另一种选择。下面详细介绍项目配置及功能实现方法。

### 15.5.1 Django 配置

作为一个新 Django 项目，第一步需要创建一个项目，操作如下：

```

# cd /data/www
# django-admin.py startproject Servermonitor

```

在创建的 omaudit 目录中修改 urls.py，添加 App 的 URL 映射规则，内容如下：

```

from django.conf.urls.defaults import *

urlpatterns = patterns('webmonitor.views',
    (r'^$', 'index'),      # 映射到 index 方法，实现前端首页渲染
    (r'add_do/', 'adddo'),  # 映射到 adddo 方法，实现新增业务提交服务器端处理
    (r'add/', 'add'),       # 映射到 add 方法，实现新增业务页面渲染
    (r'monitorlist/', 'monitorlist'), # 映射到 monitorlist 方法，实现前端扫描记录列表展示
)

```

修改项目 settings.py，关键配置项如下：

```

import os
BASE_DIR = os.path.dirname(os.path.abspath(__file__))

SYSTEM_NAME=" 分布式质量监控平台 V1.0"    # 定义系统名称

```

```

IDC={'ct':'电信','cnc':'联通','cmcc':'移动'}    # 定义采集 IDC
RRDPATH=BASE_DIR+"/rrd"    #rrdtool rrd 文件存储路径
PNGPATH=BASE_DIR+"/site_media/rrdtool"    #rrdtool 生成 png 存储路径

# 定义 webmonitor app 路径, 调用 graphrrd.sh 用 rrd 绘图相关参数
MAINAPPPATH=BASE_DIR+"/webmonitor"
#
TIME_ALARM=1    # 定义“业务请求响应时间统计”图表告警线阈值 (HRULE), 单位为秒
TIME_YMAX=1    # 定义“业务请求响应时间统计”图表 Y 轴最大值, 单位为秒
DOWN_APEED_YMAX=8388608    # 定义“业务下载速度统计”图表 Y 轴最大值, 单位为字节

```

项目包括两个 App, 其中, webmonitor 为功能应用, publicclass 用于提供公共方法调用。以下结合具体功能对两个 App 进行介绍。

### 15.5.2 业务增加功能

业务增加模块后台实现了两个功能点: 一为将业务信息写入 MySQL 数据库, 包括业务名称、监控 URL、告警通知方式、探测点及规则等; 二为创建所选择的探测点三个图表对应的 rrdtool 文件, 图表包括业务请求响应时间统计、业务下载速度、业务可用性统计。前端功能截图如图 15-4 所示。

填写业务信息	
业务名称:	<input type="text"/>
监控URL:	<input type="text"/>
通知方式:	<input checked="" type="radio"/> 短信 <input type="radio"/> 邮件 <input type="radio"/> MSN/Yahoo
选择探测点:	<input type="checkbox"/> 移动 <input type="checkbox"/> 联通 <input type="checkbox"/> 电信
探测规则:	<input type="checkbox"/> 200状态码 <input type="checkbox"/> 自定义返回串: <input type="text"/>
<div> <input type="button" value="增加"/> <input type="button" value="返回"/> </div>	

图 15-4 前端功能截图

创建 rrd 文件功能利用了 rrdtool 模块的 create() 方法实现, 实现源码如下:

```

"""
= 创建 rrd
- create_rrd(url)
"""
def create_rrd(url):
    URL=url
    domain=GetURLdomain(url)    # 调用 GetURLdomain() 方法获取 URL 域名部分
    HID=[]
    cur_time=str(int(time.time()))    # 获取当前 Linux 时间戳, 作为 rrdool.create 方法的
    start 参数值

```

```

HID=getID(URL)      # 调用 getID() 方法获取 URL 对应的所有采集点 ID
for id in HID:      # 遍历采集点 ID
    try:
        # 参数指定 rrd 文件路径, 如 “项目根目录 /rrd/www.baidu.com/17_time.rrd”;
        # step 指定步长, 设置为 300 秒, 即每隔 5 分钟收到一个值; start 指定第一条记录的起
        # 始时间, 使用 cur_time 变量指定
        rrd_time=rrdtool.create(settings.RRDPATH+'/'+str(domain)+'/'+str(id)+ \
            '_time.rrd', '--step', '300', '--start', cur_time,
            'DS:NAMELOOKUP_TIME:GAUGE:600:0:U',      # 定义数据 5 个数据源 (DS)
            'DS:CONNECT_TIME:GAUGE:600:0:U',          # GAUGE 计量类型, 收到数据后
                                                    # 直接存入 RRD;
            'DS:PRETRANSFER_TIME:GAUGE:600:0:U',      # '600' 为心跳值, 即两
                                                    # 个刻度无效时, 使用
            'DS:STARTTRANSFER_TIME:GAUGE:600:0:U',    # UNKNOWN 填充; 0:U
                                                    # 输入数据的界限
            'DS:TOTAL_TIME:GAUGE:600:0:U',
            'RRA:AVERAGE:0.5:1:600',                # 定义数据存放格式 (RRA), 分别为平均、
                                                    # 最大、最小合并数据
            'RRA:AVERAGE:0.5:6:700',                # 方式, 其他参数说明可参考 3.2 节案例
                                                    # 源码说明
            'RRA:AVERAGE:0.5:24:775',
            'RRA:AVERAGE:0.5:288:797',
            'RRA:MAX:0.5:1:600',
            'RRA:MAX:0.5:6:700',
            'RRA:MAX:0.5:24:775',
            'RRA:MAX:0.5:444:797',
            'RRA:MIN:0.5:1:600',
            'RRA:MIN:0.5:6:700',
            'RRA:MIN:0.5:24:775',
            'RRA:MIN:0.5:444:797')

        if rrd_time:
            logging.error(rrdtool.error())
            (其他两张图表 rrdtool.create 方法类似, 此处省略)
    except Exception,e:
        logging.error('create rrd error!'+str(e))

```

针对 “www.baidu.com” 业务增加成功后, 执行 “#ll rrd/www.baidu.com”, 输出已生成的 rrd 文件清单, 见图 15-5。前缀 “17\_、18\_” 代表不同运营商探测点 ID。对采集端而言, 一个运营商被视为一个独立业务, 产生的数据也是独立的。比如, 在增加业务时选择了电信、联通两个运营商探测点, 对该业务做数据采集时, 将产生两份数据。

```

total 984
-rw-rw-rw- 1 root root 71800 Jun 28 16:30 17_download.rrd
-rw-rw-rw- 1 root root 352280 Jun 28 16:30 17_time.rrd
-rw-rw-rw- 1 root root 71800 Jun 28 16:30 17_unavailable.rrd
-rw-rw-rw- 1 root root 71800 Jun 28 16:30 18_download.rrd
-rw-rw-rw- 1 root root 352280 Jun 28 16:30 18_time.rrd
-rw-rw-rw- 1 root root 71800 Jun 28 16:30 18_unavailable.rrd

```

图 15-5 生成不同运营商的 rrd 数据文件

### 15.5.3 业务报表功能

分布式质量监控平台提供了非常丰富的报表功能, 常规报表包括最近 3 小时、当天、当前月、当前年; 自定义报表根据选择的时间范围定制。在页面中提交前保持起始时间与结束时间为空则为常规报表, 最新 3 小时报表具体见图 15-6。

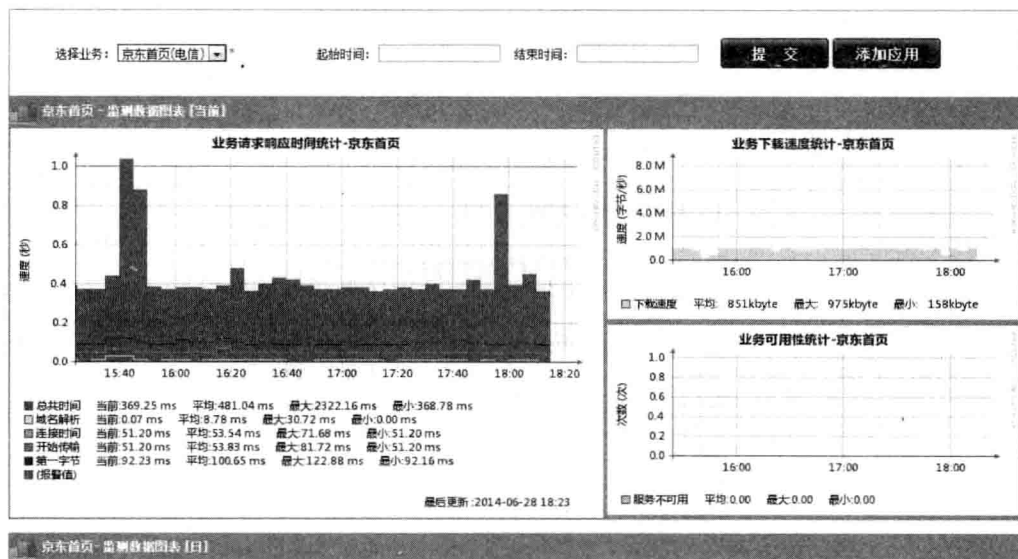


图 15-6 输出业务“当前(最新 3 小时)”报表

自定义报表根据选择的时间范围进行 rrdtool 查询, 结果如图 15-7 所示。

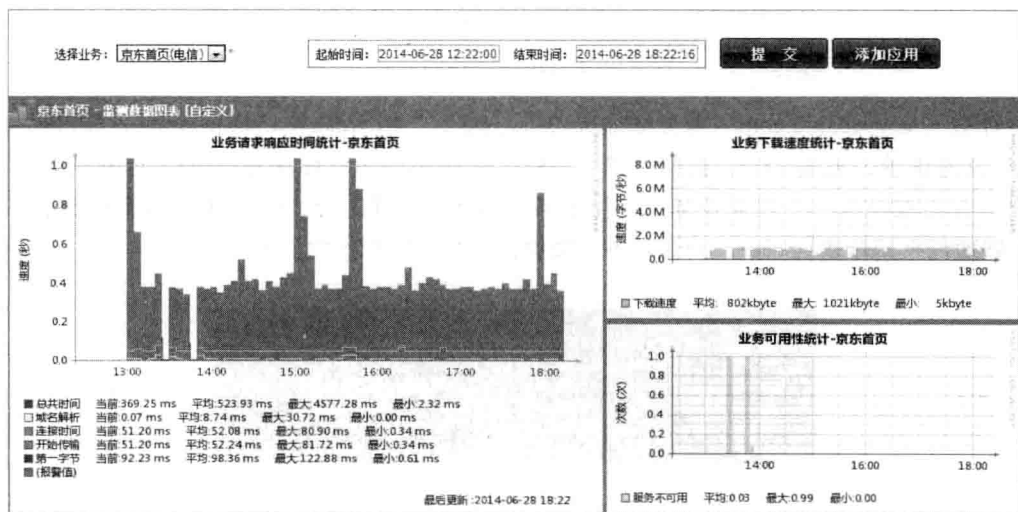


图 15-7 输出自定义时间报表

两种报表实现原理是通过定制 graph 方法的 “--start” 参数来实现，如常规报表中当前、日、月、年图表对应分别为 “-3h、-1day、-1month、-1year”，参数 “--end” 为当前时间；自定义报表采用提交的起始时间与结束时间来对应 “--start” 与 “--end” 参数。平台 rrdtool graph 生成图表时要求有中文支持，但 Python 的 rrdtool 模块没有封装 “--font” 参数，为了解决此问题，最终采用原生 rrdtool 命令行方案，在 Django 中视图中通过 os.system() 方法来调用，实现关键代码如下：

**[ /data/www/Servermonitor/webmonitor/graphrrd.sh ]**

```
#!/bin/sh
rrdfile=$1      # 接收 rrdtool 文件路径
pngfile=$2      # 接收生成 png 图片路径
rrdtype=$3      # 接收 rrd 类别，区分定义的三种图表
appname=$4      # 接收业务名称
GraphStart=$5   # 接收 rrdtool 起始时间
GraphEnd=$6     # 接收 rrdtool 结束时间
ymax=$7        # 接收 Y 轴最大值
Alarm=$8        # 接收告警红线值
# 定义两种字体
rrdtool_font_msyhbd="/data/www/Servermonitor/site_media/font/msyhbd.ttf"
rrdtool_font_msyh="/data/www/Servermonitor/site_media/font/msyh.ttf"

if [ "$rrdtype" == "time" ]; then
/usr/local/rrdtool/bin/rrdtool graph ${pngfile} -w 500 -h 207 \
-n TITLE:9:${rrdtool_font_msyhbd} \      # 定义标题字体
-n UNIT:8:${rrdtool_font_msyh} \        # 定义 Y 轴单位字体
-n LEGEND:8:${rrdtool_font_msyh} \      # 定义图例字体
-n AXIS:8:${rrdtool_font_msyh} \        # 定义坐标轴字体
-c SHADEA#808080 \      # 左上边框颜色
-c SHADEB#808080 \      # 右上边框颜色
-c FRAME#006600 \      # 数据标记说明边框颜色
-c ARROW#FF0000 \      # X、Y 轴箭头颜色
-c AXIS#000000 \      # X、Y 轴线颜色
-c FONT#000000 \      # 图形所有字体颜色
-c CANVAS#eeffff \      # 图形数据区域背景颜色
-c BACK#ffffff \      # 图形背景（不含数据区域）颜色
--title " 业务请求响应时间统计 -${appname}" -v " 速度 （ 秒 ）" \      # 图表标题
--start ${GraphStart} \      # 图表起始时间
--end ${GraphEnd} \      # 图表结束时间
--lower-limit=0 \      # 限制 Y 轴的下限
--base=1024 \      # 修改 1k 对应的刻度，默认为 1000
-u ${ymax} -r \      # 定义 Y 轴最大值
DEF:NAMELOOKUP_TIME=${rrdfile}:NAMELOOKUP_TIME:AVERAGE \      # 定义数据源及合并统计
                                     # 类型为 AVERAGE

DEF:CONNECT_TIME=${rrdfile}:CONNECT_TIME:AVERAGE \
DEF:PRETRANSFER_TIME=${rrdfile}:PRETRANSFER_TIME:AVERAGE \
DEF:STARTTRANSFER_TIME=${rrdfile}:STARTTRANSFER_TIME:AVERAGE \
DEF:TOTAL_TIME=${rrdfile}:TOTAL_TIME:AVERAGE \
```

```
COMMENT:" \n" \
AREA:TOTAL_TIME#0011ff: 总共时间 \      #用“方块”的形式来绘制“总共时间”数据
#GPRINT 定义图表下方的文字说明，参数 TOTAL_TIME 定义数据来源变量；LAST 定义合并（统计）类型，
#指显示当前值；
#其他部分为输出的文字及数值格式
GPRINT:TOTAL_TIME:LAST:" 当前\:%0.21f %Ss" \
GPRINT:TOTAL_TIME:AVERAGE:" 平均\:%0.21f %Ss" \
GPRINT:TOTAL_TIME:MAX:" 最大\:%0.21f %Ss" \
GPRINT:TOTAL_TIME:MIN:" 最小\:%0.21f %Ss" \
COMMENT:" \n" \
LINE1:NAMELOOKUP_TIME#eeee00: 域名解析 \      #用“线条”的形式来绘制“域名解析”数据
GPRINT:NAMELOOKUP_TIME:LAST:" 当前\:%0.21f %Ss" \
GPRINT:NAMELOOKUP_TIME:AVERAGE:" 平均\:%0.21f %Ss" \
GPRINT:NAMELOOKUP_TIME:MAX:" 最大\:%0.21f %Ss" \
GPRINT:NAMELOOKUP_TIME:MIN:" 最小\:%0.21f %Ss" \
COMMENT:" \n" \
(“连接时间”、“开始传输”、“第一字节”定义与“域名解析”，此处省略)

HRULE:${Alarm}#ff0000:(告警值)" \      #输出告警红线值
COMMENT:" \n" \
COMMENT:" \n" \
COMMENT:"\\t\\t\\t\\t\\t\\t\\t\\t最后更新 \\:$(date +%Y-%m-%d %H:%M')\n"
(其他两张图表 rrdtool graph 参数类似，此处省略)
```



## 构建桌面版 C/S 自动化运维平台

OManager 与 OMServer 平台实现了相同的功能,最大的区别是 OManager 是基于 C/S 结构(桌面版本)的,OMServer 是 B/S 结构(Web 版本)的。C/S 结构相对于 B/S 结构,具有交互性更强、存取模式更加安全、网络通信量低、响应速度更快、利于处理大量数据、可调用操作系统 API 等特点。当然,它也有局限性,比如要求相对统一的硬件、操作系统(版本、类型)等,由于在公司内部局域网使用且使用人群比较固定,这些条件基本都可以满足。OManager 是基于 Python 的 wxpython GUI(图形用户界面)开发,具备跨平台的能力,比如在 Linux 桌面环境,源码无须做任何改动即可直接兼容,平台支持的系统有 Windows XP、Windows 2000 或 Windows 2003、Windows 7 等;支持 Linux 2.6 或以上内核,如 Redhat、Ubuntu 等发行版。下面对平台进行全面介绍。

### 16.1 平台功能介绍

与 OMServer 一样,OManager 同样实现了一个集中式的 Linux 集群管理基础平台,支持模块扩展功能,管理员可以在 OManager 平台添加集群任务模块,其中客户端模块采用 XRC(XML Resource)方式动态定制,服务器端则与 OMServer 共享一套主控服务器端。OManager 实现日常运维远程操作、文件分发、在线升级等功能;安全方面,采用加密(RC4 算法)指令传输、操作日志记录、个性化配置等;效率方面,管理员只需选择操作目标对象及操作模块即可完成一个现网变更任务。另外在用户体验方面,模拟 Linux 终端效果,接收返回串,并使用 Psyco 模块对 Python 运行程序进行加速。任何人都可以根据自身的业务特点对

OManager 平台进行扩展, 现已支持 XML 与现有资产平台进行对接。平台登录、管理界面见图 16-1 和图 16-2。

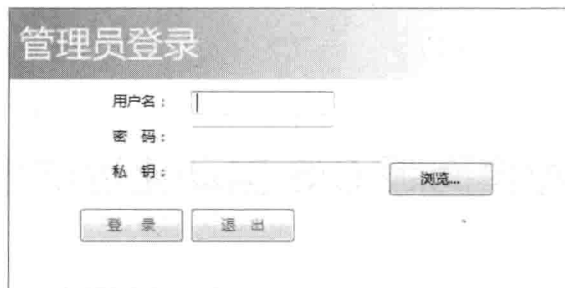


图 16-1 平台登录页面



图 16-2 平台主界面

## 16.2 系统构架设计

OManager 平台采用了两层设计模式。

第一层为客户端交互层，采用了 wxpython+xcrc+rpc+MySQL 等技术，实现了客户端与主控服务器端直连通信，rpc 分布式计算框架负责传输与计算，传输采用加密（RC4 算法）方式，保证平台整体安全性；

第二层为集群主控端服务层，支持 Saltstack、Ansible、Func 等平台，且具备多机服务的能力。系统架构图见图 16-3。

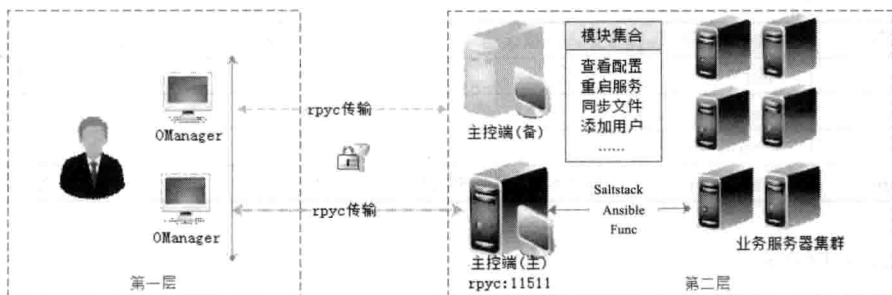


图 16-3 系统架构图

从图 16-3 中可以看出系统两个层次的结构，首先管理员在办公电脑安装 OManager 客户端软件包，作为 rpyc 客户端向 rpyc 服务器发送加密指令串，指令串通过“RC4+b64encode+密钥 key”进行加密，rpyc 服务器端同时也是 Saltstack、Ansible、Func 等的主控端，主控端将接收的数据通过“RC4+b64decode+密钥 key”进行解密，解析成 OManager 调用的任务模块，结合 Saltstack、Ansible 或 Func 向目标业务服务器集群发送执行任务，执行完毕后，对返回的执行结果做加密/解密处理，最后返回给客户端，整个任务模块分发执行流程结束。

## 16.3 数据库结构设计

### 16.3.1 数据库分析

OManager 平台采用了开源数据库 MySQL 以存储数据，数据库名为 OManager，数据库总共有 3 张表，表信息说明如下。

- upgrade：系统升级表；
- users：用户表；
- user\_logs：操作日志表。

### 16.3.2 数据字典

1) upgrade 系统升级表。

段名	数据类型	默认值	允许非空	自动递增	备注
version	char(5)		NO		最新版本号

2) user\_logs 操作日志表。

字段名	数据类型	默认值	允许非空	自动递增	备注
id	int(5)		NO	是	日志 ID
user	char(10)		NO		管理员账号
event	char(255)		NO		操作事件
Datetime	timestamp	CURRENT_TIMESTAMP	NO		操作日期

3) users 用户表。

字段名	数据类型	默认值	允许非空	自动递增	备注
admin	char(20)		NO		管理员账号
passwd	char(32)		NO		管理员密码
Privatekey	char(32)		NO		私钥 md5
privileges	char(62)		NO		权限角色

### 16.3.3 数据库模型

考虑到平台的通用性，OManager 的数据库结构设计得非常简单，只涉及账号及操作日志等基础表，平台中服务器分类及清单来源于企业资产库生成的 XML 文件。数据库中 users 表存储了管理员的账号信息；user\_logs 表存储了管理员的操作日志，表中字段“user”配置外键，与 users 表中的“admin”字段进行关联，upgrade 表存储 OManager 的版本号，系统数据库模型见图 16-4。

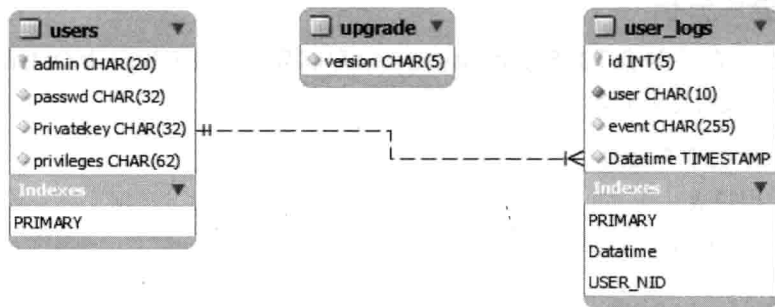


图 16-4 系统数据库模型

## 16.4 系统环境部署

### 16.4.1 系统环境说明

OManager 由 wxPython2.8、rpyc-3.2.3、psyco-1.6 等开源组件构建。为了便于读者理解，下面对平台的运行环境、安装部署、开发环境优化等进行详细说明，环境设备角色表如表 16-1 所示。

表 16-1 系统环境说明表

角色	主机名	IP	环境说明
主控端	SN2013-08-020	192.168.1.20	Saltstack   Ansible   Func 主控端、rpyc 服务器端
OManager	DELL-PC	192.168.1.101	wxPython、rpyc 客户端

### 16.4.2 系统环境搭建

OManager 平台基于多种 Python 第三方模块实现，包括 wxpython、rpyc、MySQL-python、psyco、pywin32 等，这些开源组件无论是在开发效率还是运行速度方面都赢得了很好的口碑，尤其容易上手，可以做到快速开发、快速实现。wxPython 官网提供了非常丰富的 demo 代码，可以帮助开发人员做到边学边用，下面介绍平台所需模块及功能说明。

- ❑ MySQL-python-1.2.4b4.win32-py2.7.exe: Python 访问 MySQL 的 API 模块。
- ❑ psyco-1.6.win32-py25.exe: Python 程序提速模块。
- ❑ pyinstaller-2.0.zip: Python 程序打包工具，安装包制作推荐使用 Smart Install Maker。
- ❑ pywin32-218.win32-py2.7.exe: Windows 系统 API 访问库。
- ❑ rpyc-3.2.3.win32.exe: 分布式计算框架。
- ❑ wxPython2.8-win32-docs-demos-2.8.12.1.exe: wxPython Demo (可选项)。
- ❑ wxPython2.8-win32-unicode-2.8.12.1-py27.exe: Python GUI 图形库。

平台重点文件及目录说明见图 16-5。

data	平台数据目录，存放配置文件、服务器信息XML文件等
img	平台图标目录
include	Python头文件存放目录，编译环境用到
Module	平台功能模块，XBC存放目录
numbers	平台帐号密码存放目录
tmp	平台临时目录，存放升级包描述XML文件
MD5sum.exe	文件MD5计算工具
OManager.exe	平台入口可执行文件

图 16-5 系统目录结构及说明

## 16.5 系统功能模块设计

### 16.5.1 用户登录模块

OManager 平台的登录采用了双重安全校验机制：一种为传统的用户名与密码匹配，另一种为密钥文件校验方式，实现的原理是在密钥文件中输入任意随机字符串，通过平台自带的 md5sum.exe 工具计算出该文件的 md5，将生成的 md5 字符串更新到 users（用户表）管理员账号对应的 Privatekey 字段，以 root 用户的密钥 numbers/root.pem 为例，使用方法见图 16-6 和图 16-7。

```
D:\python\OManager\OManager>MD5sum.exe numbers/root.pem
8115082536da7863426017e0248bf3a8 numbers/root.pem
```

图 16-6 查看密钥文件 md5

admin 管理员帐号	passwd 管理员密码	Privatekey 私钥MD5	privileges 权限角色
root	e10adc3949ba59abbe56e057f20f883e	8115082536da7863426017e0248bf3a8	root

图 16-7 数据库存储的密钥文件 md5 数据

管理员登录时首先获得选择密钥文件的 md5，再与数据库中的 Privatekey 字段进行匹配，建议由超级管理员提前开设好所有用户的账号信息，包括用户名、密码及密钥。再统一将密钥文件以人为单位进行发放。验证的实现方法源码如下：

```
def Check(self, name, password, Privatekey):
    import md5
    m = md5.new(password)      # 使用 md5 模块计算密码的 md5 串
    md5pass=m.hexdigest()
    myrow=DBclass()           # 创建数据库连接对象（自定义类）
    sql = "select admin,privileges from users where admin='%s' and
passwd='%s' \
and Privatekey='%s'" % (name, md5pass, Privatekey)    # 参照 MySQL 中的用户名、
                                                         # 密码、密钥进行校验

    result = myrow.fetchall(sql)
    return result            # 返回结果集
```

下面是计算密钥文件 md5 的实现方法，主要用到了 hashlib 模块：

```
# 计算文件 md5 值，参数 fileName 为实体文件路径；参数 excludeLine 为排除的文本行；
# 参数 includeLine 为额外包含的行
def md5(fileName, excludeLine="", includeLine=""):
    m = hashlib.md5()        # 使用 hashlib 模块生成一个 md5 hash 对象
    try:
```

```

        fd = open(fileName, "rb")    # 打开密钥文件
    except IOError:
        print "Unable to open the file in readmode:", filename
        return
    eachLine = fd.readline()
    while eachLine:    # 遍历密钥文件
        if excludeLine and eachLine.startswith(excludeLine):    # 排除指定的行
            continue
        m.update(eachLine)    # 用 update 方法对行字符串进行 md5 加密且不断做更新处理
        eachLine = fd.readline()
    m.update(includeLine)    # 对额外包含的行做更新和加密处理
    fd.close()
    return m.hexdigest()    # 返回十六进制结果

```

调用计算文件密钥 md5 方法:

```
md5(self.Privatekey.GetValue())    #self.Privatekey.GetValue() 为用户选择的密钥文件路径
```

## 16.5.2 系统配置功能

OManager 平台将常用的参数配置化, 包括连接数据库、主控端、传输密钥等信息, 当外部环境发生变化时无须做代码变更, 简单更新配置即可, 提高了平台的易用性, 降低使用门槛, 具体是通过 ConfigParser 模块操作 ini 文件实现, 效果见图 16-8。

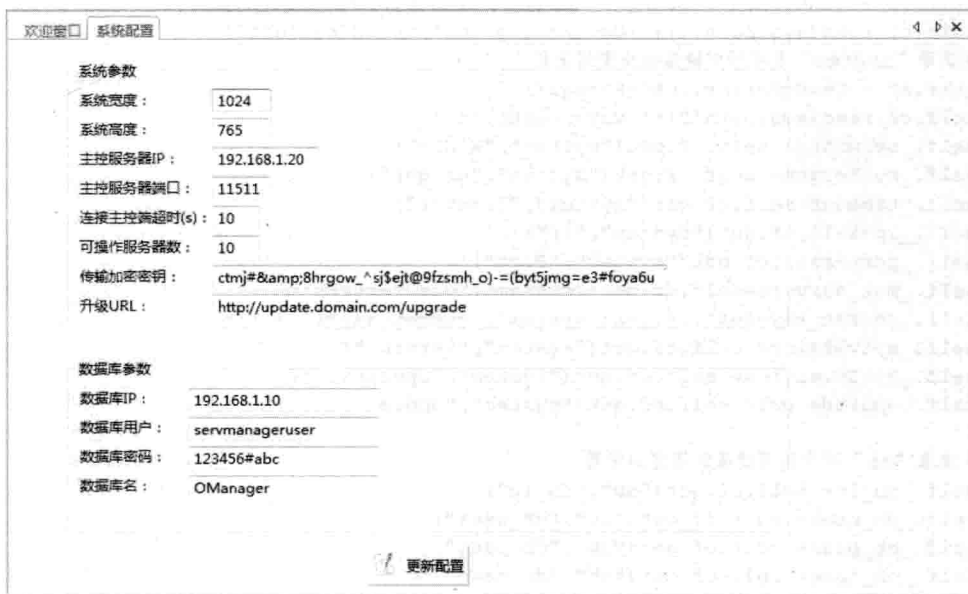


图 16-8 系统配置功能

手工修改 ini 文件与界面操作达到的效果是一样的, 平台 ini 文件格式如下:

**[ data/config.ini ]**

```
[system]
height = 765
width = 1024
version = v2014
upversion = 10026
ip = 192.168.1.20
port = 11511
timeout = 10
max_servers = 10
secret_key = ctmj#&8hrgow^sj$ejt@9fzsmh_o)-(byt5jmg=e3#foya6u
upgrade_url = http://update.domain.com/upgrade

[db]
db_ip = 192.168.1.10
db_user = servmanageruser
db_pass = 123456#abc
db_name = OManager
```

使用 ConfigParser 模块操作 ini 配置文件非常方便, 通过 get()、set() 方法来读取与更新配置文件。读配置源码如下:

```
self.cf = ConfigParser.ConfigParser()    # 创建 ConfigParser 对象
self.cf.read(sys.path[0]+'data/config.ini', encoding='utf8')    # 读取配置文件
# 读取 "system" 节中所有键值到指定的变量
self.cf = ConfigParser.ConfigParser()
self.cf.read(sys.path[0]+'data/config.ini')
self._syswidth= self.cf.get("system", "Width")
self._sysheight= self.cf.get("system", "Height")
self._timeout=self.cf.get("system", "Timeout")
self._ip=self.cf.get("system", "IP")
self._port=self.cf.get("system", "Port")
self._max_servers=self.cf.get("system", "max_servers")
self._secret_key=self.cf.get("system", "secret_key")
self._sysversion= self.cf.get("system", "Version")
self._sysUpversion= self.cf.get("system", "Upversion")
self._upgrade_url= self.cf.get("system", "upgrade_url")

# 读取 "db" 节中所有键值到指定的变量
self._db_ip= self.cf.get("db", "db_ip")
self._db_user= self.cf.get("db", "db_user")
self._db_pass= self.cf.get("db", "db_pass")
self._db_name= self.cf.get("db", "db_name")
```

更新配置也非常简单, 将 get() 方法更换成 set(), 再指定 ini 的节、键、值三个元素即可。下面是更新数据库 IP 参数的代码, 其中 self.DB\_ip.GetValue() 为输入框的内容。

```
self.cf.set("db", "db_ip", self.DB_ip.GetValue())
```

### 16.5.3 服务器分类模块

为了让 OManager 更具通用性, 平台的服务器信息依赖企业现有资产库数据, 通过平台规范好的格式生成 XML 文件, 结合 Tree 与 ListBox 控件实现功能分类与服务器联动, 效果见图 16-9。

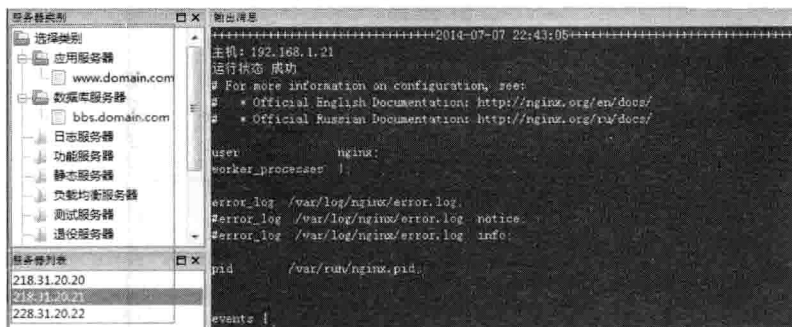


图 16-9 服务器分类选择

服务器分类的组织形式与 OMserver 保持一致, 即功能分类→业务分类→服务器。图 16-10 为服务器类别的 XML 数据文件, 用来描述服务器类别信息。

【data/ServerOptioninfo.xml】

```
<?xml version="1.0" encoding="UTF-8"?>
<wml>
  <AppClass id="1">
    <appname>应用服务器</appname>
  </AppClass>
  <AppClass id="2">
    <appname>数据库服务器</appname>
  </AppClass>
  <AppClass id="3">
    <appname>日志服务器</appname>
  </AppClass>
  <AppClass id="4">
  <AppClass id="5">
  <AppClass id="6">
  <AppClass id="7">
    <appname>测试服务器</appname>
  </AppClass>
  <AppClass id="8">
  <AppClass id="9">
  <AppClass id="10">
  <AppClass id="11">
  <AppClass id="12">
  <AppClass id="13">
    <appname>游戏服务器</appname>
  </AppClass>
</wml>
```

图 16-10 服务器分类的 XML 文件

其中，“<AppClass id="1">” 标签 id 属性值为功能分类 ID 号，“<appname> 应用服务器 </appname>” 使用 <appname> 子元素描述功能分类名称。服务器信息的 XML 数据文件用来描述服务器的详细属性，详细内容见图 16-11，属性与子元素说明见表 16-2。

【 data/Serverinfo.xml 】(部分内容)

```
- <wml>
-   <server ip="192.168.1.20">
        <serverserial>SN2013-08-020</serverserial>
        <wip>218.31.20.20</wip>
        <lip>192.168.1.20</lip>
        <os>Linux</os>
        <app>www.domain.com</app>
        <locate>05-02-10</locate>
        <option>1</option>
    </server>
-   <server ip="192.168.1.21">
        <serverserial>SN2013-08-021</serverserial>
        <wip>218.31.20.21</wip>
        <lip>192.168.1.21</lip>
        <os>Linux</os>
        <app>www.domain.com</app>
        <locate>05-05-01</locate>
        <option>1</option>
    </server>
```

图 16-11 服务器信息的 XML 文件

表 16-2 属性与子元素说明

属性与子元素	含 义
ip	IP 地址 (唯一标识), 内外网 IP 均可
serverserial	主机名
wip	外网 IP 地址
lip	内网 IP 地址
os	操作系统类别
app	应用名称, 一般为应用域名
locate	服务器所处机架位置
option	功能分类 ID, 与服务器功能分类的 XML 文件中 AppClass 标签的 id 属性关联

每个管理员所负责的服务器资源通常都不一样, 一般以服务器功能分类的维度划分。OManager 可为这种权限要求提供支持, 实现的思路是在 users 表的 privileges (权限角色) 字段定义服务器分类 ID, 其中“root”为特殊权限, 代表超级管理员, 所有服务器资源都可见。账号“demo”的权限配置, 以及在平台中展示的效果见图 16-12。

admin 管理员帐号	passwd 管理员密码	Privatekey 私钥MD5	privileges 权限角色
demo	e10adc3949ba59abbe56e057f20f883e	dced4ca98819e25a3ebcc939d5c21378	1, 2, 3, 5, 6
root	e10adc3949ba59abbe56e057f20f883e	8115082536da7863426017e0248bf3a8	root



图 16-12 用户权限配置及展示效果

当窗体 (wx.Frame) 初始化时, “服务器类别” 控件会自动加载数据, 实现的方法是通过遍历以上提到的两个 XML 数据, 将当前账号的权限 ID 列表与服务器类别 ID 进行关联, 获取所具备的权限, 即拥有的服务器类别 ID。“应用名称” 则通过服务器信息的 “<option>” 元素与服务器类别 ID 进行匹配, 实现源码如下:

```
import xml.etree.ElementTree as ET
import os
import sys

root_tree = ET.parse(sys.path[0]+'data/ServerOptioninfo.xml') # 打开服务器类别 XML 文档
class_tree = ET.parse(sys.path[0]+'data/Serverinfo.xml')      # 打开服务器信息 XML 文档

root_doc = root_tree.getroot()    # 获得服务器类别 XML 文档 root 节点
class_doc = class_tree.getroot()  # 获得服务器信息 XML 文档 root 节点

class ServerClassList():

    def Return_list(self, UserPrivileges):    # 返回服务器类别、应用方法
        ServerList_KEY=[]                  # 定义返回的服务器类别、应用信息列表对象
        serverclass=[]                     # 定义服务器类别列表对象
        serverapp=[]                       # 定义业务应用列表对象

        for root_child in root_doc:        # 遍历服务器类别节点

            if not root_child.get('id') in UserPrivileges and not
            UserPrivileges[0]=="root":
                continue    # 没有权限的服务器类别将被忽略
            serverclass.append(root_child[0].text.encode('gbk')) # 追加服务器类
                                                                    # 别名称 <appname>

            serverapp=[]
            for class_child in class_doc:    # 遍历服务器信息节点
                # 如与功能分类 ID 相匹配, 则追加 <app> 到 serverapp
```

```

# 通过 index() 方法产生的异常判断当前 <app> 是否已经存在于 serverapp 中
if class_child[6].text==root_child.get('id'):
    try:
        serverapp.index(class_child[4].text.encode('gbk'))
    except:
        serverapp.append(class_child[4].text.encode('gbk'))

serverclass.append(serverapp)
ServerList_KEY.append(serverclass)
serverclass=[]
# 返回结果串格式: [['应用服务器', ['www.a.com', 'www.b.com']], ['数据库服务器', ['www.
c.com']]...]
return ServerList_KEY

```

### 16.5.4 系统升级功能

相比 B/S 结构程序, C/S 结构的另一缺点是不方便升级, 部分软件甚至要求重新安装、重启计算机等操作。为了解决此问题, OManager 在系统升级方面结合了 B/S 的模式, 将升级包放在远端, 由管理员触发升级操作, 同时不影响当前的其他操作, 重启 OManager 程序后即可完成升级。OManager 系统升级流程图见图 16-13。

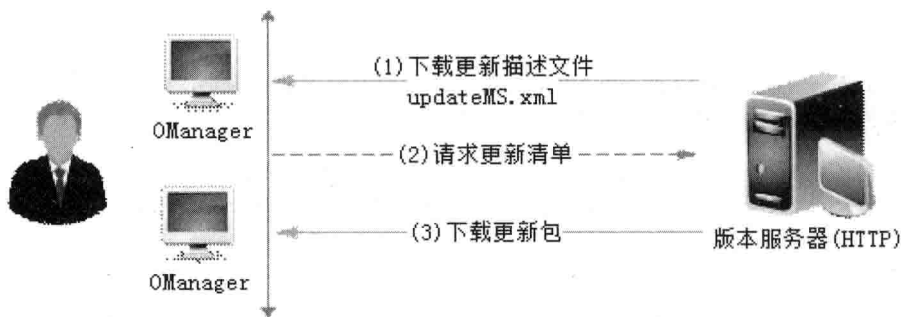


图 16-13 系统升级流程图

OManager 系统升级的原理: 首先将升级描述文件 (updateMS.xml)、升级包上传至版本服务器, 由管理员触发升级操作, 再通过 urllib 模块实现 HTTP 方式下载 updateMS.xml 文件并进行分析, 获取所有需要升级的程序包, 包括远程 URL 及下载本地存储地址, 最后遍历下载所有升级包到指定的位置, 完成整个升级过程。下面是升级描述文件 updateMS.xml 的示例:

【 tmp/updateMS.xml 】

```

<?xml version="1.0" encoding="UTF-8"?>
<wml>

```

```

<AppClass id="1">
  <localsrc>data/Serverinfo.xml</localsrc>
  <remotesrc>/data/Serverinfo.xml</remotesrc>
</AppClass>
<AppClass id="2">
  <localsrc>data/ServerOptioninfo.xml</localsrc>
  <remotesrc>/data/ServerOptioninfo.xml</remotesrc>
</AppClass>
<AppClass id="3">
  <localsrc>OManager.10026.exe</localsrc>
  <remotesrc>/OManager.exe</remotesrc>
</AppClass>
</wml>

```

在此 XML 文件中，localsrc 与 remotesrc 分别表示本地存储地址及远程 URL 路径，远程 URL 文件与本地路径建议保持一致，可以提高系统的可维护性，如本地的“data/Serverinfo.xml”与远程的“/data/Serverinfo.xml”，远程升级包目录结构见图 16-14。

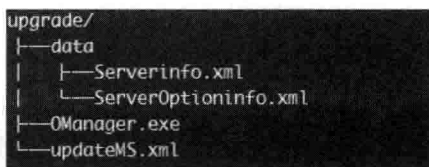


图 16-14 远程升级包存储路径

在此配置中，需要升级的程序包为 OManager.exe、Serverinfo.xml、ServerOptioninfo.xml 三个，变更项包括了添加主机信息、主程序优化等，下面介绍升级步骤。

- 1) 上传升级相关文件到版本服务器指定位置，具体见图 16-14。
- 2) 更新数据库中平台最新版本号，即更新 upgrade 表的 version 字段，如更新版本号为“10026”；用户会根据 data/config.ini 中的 upversion 键值与最新版本号进行匹配，当小于最新版本号时将触发升级。
- 3) 点击“系统升级”工具栏图标进行升级，升级成功后如图 16-15 所示。

升级结束后，平台根目录下多了一个“OManager.10026.exe”最新版本的程序包，见图 16-16。

运行“OManager.10026.exe”，在主程序左侧的服务器列表框中多了一台“218.31.20.11”主机，见图 16-17。再查看 data/config.ini 中的 upversion 键值，已经改为“10026”，说明系统已成功升级。

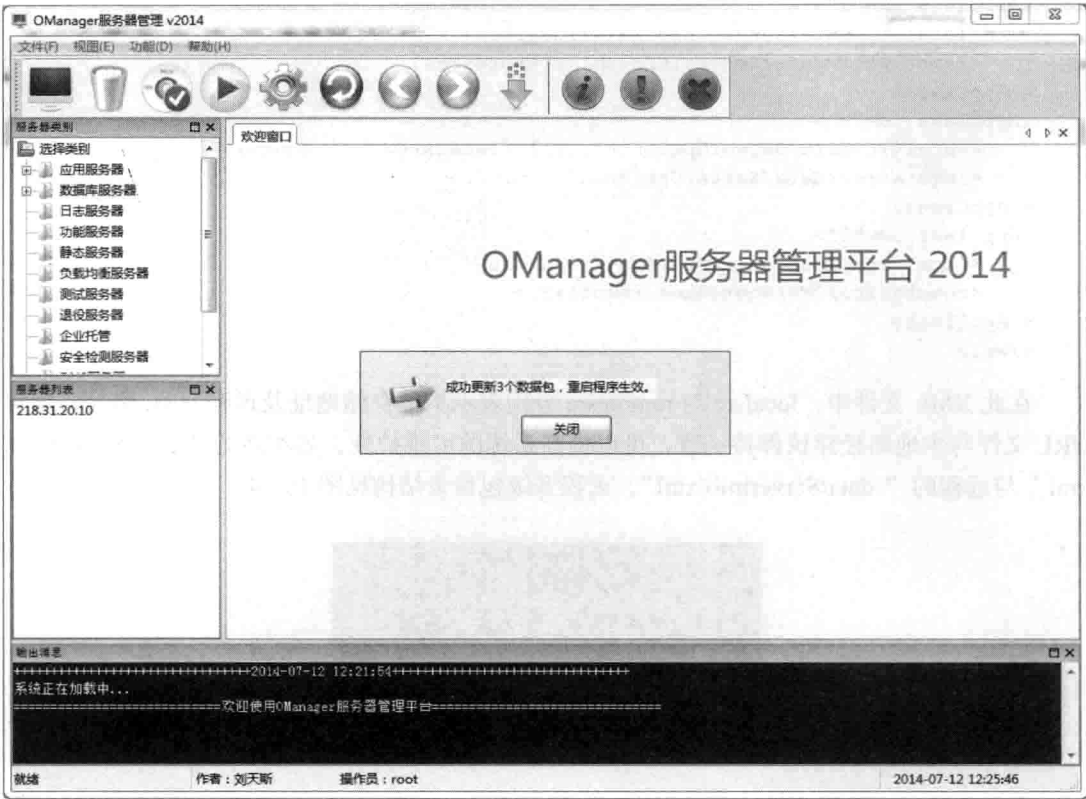


图 16-15 系统升级成功

MD5sum.exe	2013/7/21 21:07
OManager.10026.exe	2014/7/12 12:26
OManager.exe	2014/7/12 12:20
msvc90.dll	2013/8/4 15:37
msvc90.dll	2013/8/4 15:37
msvc90.dll	2013/8/4 15:37
python27.dll	2012/4/10 23:31
wxbase30u_net_vc90.dll	2013/12/28 2:21
wxbase30u_vc90.dll	2013/12/28 2:21

图 16-16 升级后的文件列表



图 16-17 更新后的主机列表

介绍完系统升级的操作过程，下面介绍 OManager 实现升级功能的源码分析，使用模块 `urllib.urlopen(url).read()` 方法实现 HTTP 协议文件下载，使用 `xml.etree.ElementTree` 模块实现 XML 文件的分析。

```
def load_data(self, event):    # 系统升级方法
    try:
        if self.button.GetLabel() == u" 关闭 ":
            self.Destroy()
        url = self.updateURL + "/updateMS.xml"    # 指定升级描述文件远程及本地路径
        localfile = sys.path[0] + '/tmp/updateMS.xml'
        if not self.download(url, localfile):    # 下载升级描述文件
            return
    except Exception, e:
        wx.MessageBox(u" 更新描述文件下载失败 " + str(e), u"OManager: ", style=wx.
OK|wx.ICON_ERROR)
        self.Destroy()
        return
    try:    # 打开升级描述文件，为下面的分析做好准备
        import xml.etree.ElementTree as ET
        update_tree = ET.parse(sys.path[0] + '/tmp/updateMS.xml')
        up_doc = update_tree.getroot()
    except Exception, e:
        wx.MessageBox(u" 导入更新包出错 ", u"OManager: ", style=wx.OK|wx.ICON_ERROR)
        self.Destroy()
        return

    try:    # 遍历描述文件，获取升级描述文件中所有程序包的远程及本地路径，调用
        # download() 方法实现下载
        upgrade_count = 0
        for cur_child in up_doc:
            upgrade_count += 1
            url = self.updateURL + cur_child[1].text
```

```

        localfile=sys.path[0]+'/'+'cur_child[0].text
        if self.download(url,localfile)==False:
            break
        self.cf.set("system", "Upversion", self.lastversion) # 更新 config.ini
                                                    # 最新版本号
        self.cf.write(open(sys.path[0]+'data/config.ini', "w"))
        self.ConnStaticText.SetLabel(u" 成功更新 "+str(upgrade_count)+" 个数据包 ...")
        self.button.SetLabel(u" 关闭 ")
    except Exception,e:
        wx.MessageBox(u" 系统文件下载失败 ", "OManager", style=wx.OK|wx.ICON_ERROR)
        self.Destroy()
        return
    finally:
        pass
    event.Skip()

```

### 16.5.5 客户端模块编写

OManager 提供客户端模块开发支持，与 OMserver 的实现思想一样，区别是 OMserver 基于 HTML 表单来定义，而 OManager 基于 XRC。XRC (XML Resource) 的设计来源于 wxWidgets，原理是将界面设计的工作从程序中独立出来，类似于 Django 开发框架中模板系统的角色，目的是将业务逻辑与界面进行分离，好处是代码的结构会更加清晰，可读性也会大大提高。具体做法是通过 XML 格式定义系统界面，当程序运行时再载入。XRC 的使用手册见 [http://wiki.wxwidgets.org/Using\\_XML\\_Resources\\_with\\_XRC](http://wiki.wxwidgets.org/Using_XML_Resources_with_XRC)。OManager 平台将功能模块采用 XRC 设计，在主程序中按功能分类导入，效果见图 16-18 和图 16-19。



图 16-18 功能模块菜单

OManager 平台提供了最多 2 个控件参数的定义，控件类别支持 wxSpinCtrl (微调控制器)、wxListBox (列表控件)、wxTextCtrl (文本输入控件) 等，当然，扩展更多的控件类型也非常简单，前提是需要了解各控件的属性及方法，其中控件值会被当成模块参数通过 rpyc

传输到服务器端。下面为“bas\_1001\_系统日志.xrc”功能模块的设计，包括一个容量控件 wxPanel 对象，wxPanel 对象包含了两个对象，一个文字标签控件 wxStaticText 对象，通过 <label> 元素定义该模块的功能文字说明；另一个对象为微调控制器 wxSpinCtrl，通过 <value> 元素定义默认值，<min> 与 <max> 定义控件的最小值及最大值。更多的控件介绍请参考：[http://wiki.wxwidgets.org/Using\\_XML\\_Resources\\_with\\_XRC](http://wiki.wxwidgets.org/Using_XML_Resources_with_XRC)。该功能模块的 XRC 定义内容如下：



图 16-19 功能模块窗口

【 Module/ bas\_1001\_系统日志.xrc 】

```
<?xml version="1.0" encoding="utf-8"?>
<resource>
    <object class="wxPanel" name="panel">
        <size>200,100</size>
        <object class="wxStaticText" name="label1">
            <label> 功能描述：显示服务器 Message 最新选择条数的记录。</label>
            <pos>30,20</pos>
        </object>
        <object class="wxSpinCtrl" name="Parameter1_object_id">
            <style>wxSP_ARROW_KEYS</style>
```

```

        <value>30</value>
        <min>1</min>
        <max>100</max>
        <pos>30,50</pos>
    </object>
</object>
</resource>

```

在主程序中,通过 `xrc.XmlResource()` 方法加载 XRC 模块文件,使用 `xrc.XRCCTRL()` 方法获取控件对象,使用对象的 `GetValue()` 或 `GetStringSelection()` 方法得到控件输入值,其中 `wxSpinCtrl`、`wxTextCtrl` 控件使用 `GetValue()` 方法, `wxListBox` 控件使用 `GetStringSelection()` 方法。在主程序中调用 XRC 的方法,源码(部分)如下:

```

from wx import xrc
self.res = xrc.XmlResource(sys.path[0]+'Module/bas_1001_系统日志.xrc')
# 加载模块资源文件
panel = self.res.LoadPanel(self, "panel")    # 加载 panel 面板控件
try:
    self.Parameter1 = xrc.XRCCTRL(panel, 'Parameter1_object_id') # 加载控件 1 对象名
except Exception,e:
    pass
# 获取不同控件的返回值,GetClassName() 方法返回控件类别名,用于定位不同控件获取 value 的方法
try:
    if self.Parameter1.GetClassName()=="wxSpinCtrl":
        self.Parameter1_value=self.Parameter1.GetValue()
    elif self.Parameter1.GetClassName()=="wxListBox":
        self.Parameter1_value=self.Parameter1.GetStringSelection()
except Exception,e:
    pass
...

```

平台功能模块 XRC 文件命名遵循一定的标准规范,即“模块功能类别\_模块 ID\_功能中文名称.xrc”,文件名将会以“\_”作为分隔符,拆分的数据将应用到系统功能中,比如文件名前缀“模块功能类别”会根据不同类别代号加载到不同功能菜单,实现源码(部分)如下:

```

bashmenu = wx.Menu()    # 定义 "基本功能" 二级菜单
appmenu = wx.Menu()     # 定义 "应用功能" 二级菜单
dbmenu = wx.Menu()      # 定义 "数据库功能" 二级菜单
servicemenu = wx.Menu() # 定义 "后台服务功能" 二级菜单
middlemenu = wx.Menu()  # 定义 "中间件功能" 二级菜单
# 根据不同 XRC 文件前缀,将三级菜单追加到对应的二级菜单中
for file_info in self.ModuleDetail:
    file_array=string.split(file_info, '_')
    if file_info[0:3]=="bas":
        bashmenu.Append(int(file_array[1]),file_array[2],file_array[2])
    elif file_info[0:3]=="app":

```

```

        appmenu.Append(int(file_array[1]),file_array[2],file_array[2])
    elif file_info[0:3]=="dba":
        dbmenu.Append(int(file_array[1]),file_array[2],file_array[2])
    elif file_info[0:3]=="ser":
        servicemenu.Append(int(file_array[1]),file_array[2],file_array[2])
    elif file_info[0:3]=="mid":
        middlemenu.Append(int(file_array[1]),file_array[2],file_array[2])

```

文件“模块 ID”段将作为该模块的唯一标识，与服务器端模块进行匹配。另外，要求模块 XRC 文件必须存放于平台 Module 目录。以下为客户端的所有模块清单，其中 ID 为“100\*”的模块，服务器端已完成对接，其他部分读者可以根据自身的需求自行开发或扩展，平台功能模块 XRC 文件列表见图 16-20。

名称	修改日期	类型	大小
app_1005_同步应用文件.xrc	2014/7/2 6:59	XRC 文件	1 KB
app_1006_查看应用配置.xrc	2014/7/2 23:47	XRC 文件	1 KB
app_3200_YUM安装.xrc	2013/7/20 18:30	XRC 文件	1 KB
app_3201_硬件检查.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_1001_系统日志.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_1002_最后登录.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_1003_系统版本.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_1004_内核模块.xrc	2014/7/3 19:10	XRC 文件	1 KB
bas_1007_重启进程服务.xrc	2014/7/2 23:47	XRC 文件	1 KB
bas_3100_可控服务器.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_3105_监听端口.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_3106_系统用户.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_3107_系统组.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_3109_计划任务.xrc	2013/7/20 18:30	XRC 文件	1 KB
bas_3110_活动用户.xrc	2013/7/20 18:30	XRC 文件	1 KB
dba_3300_更新配置.xrc	2013/7/20 18:30	XRC 文件	1 KB
dba_3302_重启MySQL.xrc	2013/7/20 18:30	XRC 文件	1 KB
dba_3303_锁进程.xrc	2013/7/20 18:30	XRC 文件	1 KB
dba_3304_写语句.xrc	2013/7/20 18:30	XRC 文件	1 KB
dba_3305_检查备份.xrc	2013/7/20 18:30	XRC 文件	1 KB
mid_3500_消息服务.xrc	2014/6/29 22:23	XRC 文件	1 KB
ser_3400_后台分析检查.xrc	2014/6/29 22:23	XRC 文件	1 KB

图 16-20 功能模块 XRC 文件列表

### 16.5.6 执行功能模块

由于 OManager 只有两层结构，与服务器端的通信就是一个交互过程，由客户端发起任务请求，服务器执行任务并返回操作结果，操作步骤见图 16-21。

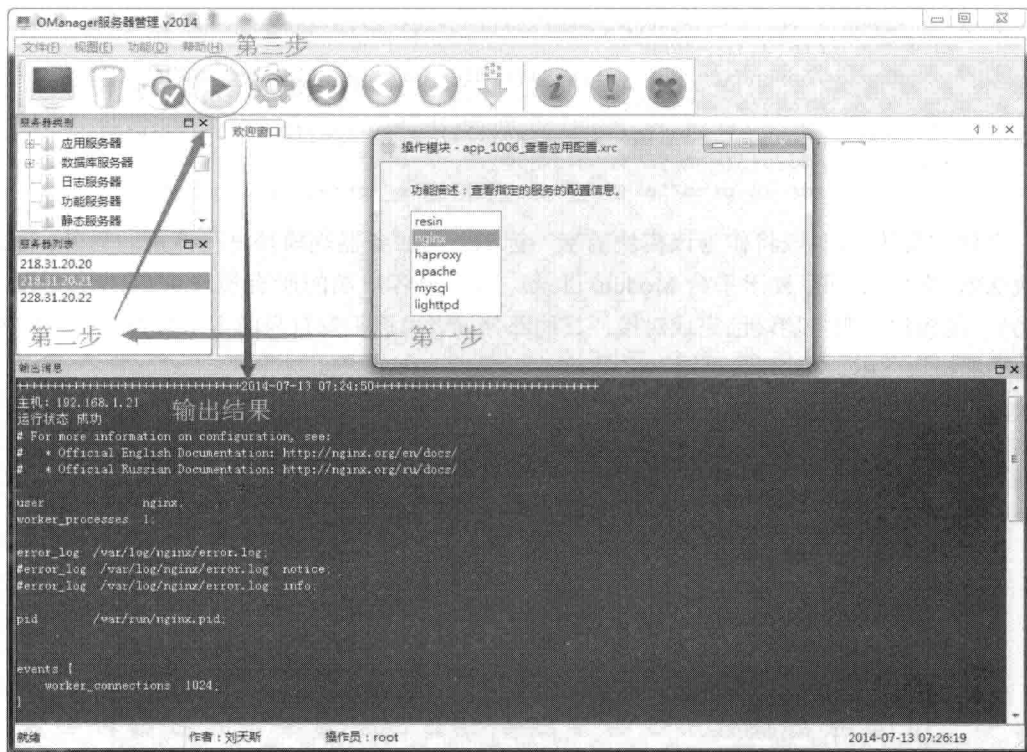


图 16-21 功能模块执行步骤

为提高平台的通用性及兼容度，OManager 的数据封装、传输、加密方式及服务器端与 OMserver 一致，即传输采用了 rpyc 框架、RC4 加密算法、服务器端同一监听服务。服务器端的实现本节不再做介绍，具体可参考 13.5.3 节。下面介绍基于 wxPython 实现的客户端提交任务的几个方法。

```
try:
    conn=rpyc.connect(self._ip,int(self._port))    # 连接 rpyc 服务器
    # 调用 login() 方法实现通信账号、密码校验
    conn.root.login('OMuser','KJS23o4ij09gHF734iuh sdfhkGYSihoiwhj38u4h')
except Exception,e:
    message=u" 系统提示: 连接远程服务器超时。"+str(e)
    wx.MessageBox(message,u"OManager 服务器管理平台: ",style=wx.OK|wx.ICON_ERROR)
    return

# 调用 OnGetSelectServerinfo 方法获取计算机名、字符串、服务器数量
_server_list=self.OnGetSelectServerinfo('serverserial_ip',1,int(self._max_servers))
```

```

# 判断用户是否选择了至少一台服务器, 不选择则直接返回
if not _server_list:
    return
# 操作记录调用了 Addsyslogs() 方法写入 user_logs 表, 用于操作记录追溯
Intologs.Addsyslogs(self.CurrentAdmin,u" 操作对象: "+\
self.OnGetSelectServerinfo('lip',1,20)+u"- 操作 MID:"+GetModelestrrow[0])

# 合并提交串, 格式: "模块 ID@@ 主机 IP* 主机名, N@@ 参数 1@@ 参数 2@@ "
例如: "1001@@192.168.1.21*SN2013-08-021@@30@@ "
put_string+=str(GetModelestrrow[0])+"@@"+_server_list+"@@"+Parameter_string
# 调用 tencode() 方法对提交串进行加密
put_string=FunApp.tencode(put_string,self._secret_key)

# # 调用 rpyc 的 Runcommands() 方法执行任务, 返回的结果通过 tdecode() 方法解密 OPresult=
FunApp.tdecode(conn.root.Runcommands(put_string),self._secret_key).
decode('utf8')
# 在“输出消息”框输出返回结果
self.OnWriteMessageBox(FunApp.format_str(OPresult))
conn.close()

```

下面为“输出消息”框输出消息方法, 使用 SetInsertionPoint(0) 获取消息插入点, 通过 WriteText() 方法写入消息, 代码如下:

```

def OnWriteMessageBox(self,message):
    t = time.localtime(time.time())
    st = time.strftime("%Y-%m-%d %H:%M:%S", t) # 获取当前系统时间
    self.SysMessaegText.SetInsertionPoint(0) # 设置消息框插入点, 参数 0 为开始位置
    # 将方法参数 message( 消息内容 ) 写入消息框
    self.SysMessaegText.WriteText("++++++"+str(st)+"++++++\n"+message+"\n")
    self.SysMessaegText.SetInsertionPoint(0)

```

执行任务返回的结果见图 16-22。另外 OManager 的窗体元素支持任意角度的组合、分离、拖动等, 管理员可以根据不同喜好进行调整。

### 16.5.7 平台程序发布

为了让平台在没有 Python 以及第三方模块包的环境中正常运行, 对源程序进行打包发布是项目最后一个环节, 对此 pyinstaller (<http://www.pyinstaller.org>) 提供了很好的解决方案, 其支持 Linux 与 Windows 平台可执行程序的制作, 简单易用。Pyinstaller 2.0 无须安装, 解压即可使用, 下面为平台打包的 bat 批处理脚本。



图 16-22 功能模块执行结果

### [ install.bat ]

```

cd D:\python\OManager\OManager
d:
rd /S /Q dist
rd /S /Q build
del logdict2.7.3.final.0-1.log
python d:/soft/pyinstaller-2.0/pyinstaller.py --onedir -w --icon=img/imac.ico
OManager.py
copy MD5sum.exe dist\OManager
xcopy /s data dist\OManager\data\
xcopy /s img dist\OManager\img\
xcopy /s Module dist\OManager\Module\
xcopy /s numbers dist\OManager\numbers\
xcopy /s tmp dist\OManager\tmp\
rd /S /Q build
rd /S /Q build
del logdict2.7.3.final.0-1.log

```

假设项目目录为“D:\python\OManager\OManager”，参数“--onedir”为创建的一个目录，包含 exe 文件以及相关依赖类包；“-w”表示制作视窗界面，无控制台（命令行）；“--icon”指定执行程序图标；“OManager.py”为平台入口源程序。通过 xcopy 复制平台相关目录到打包路径（如 dist\OManager）。打包后的目录结构见图 16-23。

名称	修改日期	类型	大小
tmp	2014/7/13 10:11	文件夹	
numbers	2014/7/13 10:11	文件夹	
Module	2014/7/13 10:11	文件夹	
include	2014/7/13 10:11	文件夹	
img	2014/7/13 10:11	文件夹	
data	2014/7/13 10:11	文件夹	
wxmsw30u_xrc_vc90.dll	2013/12/28 2:23	应用程序扩展	658 KB
wxmsw30u_html_vc90.dll	2013/12/28 2:23	应用程序扩展	587 KB
wxmsw30u_core_vc90.dll	2013/12/28 2:22	应用程序扩展	4,671 KB
wxmsw30u_aui_vc90.dll	2013/12/28 2:23	应用程序扩展	391 KB
wxmsw30u_adv_vc90.dll	2013/12/28 2:22	应用程序扩展	1,222 KB
wxbase30u_xml_vc90.dll	2013/12/28 2:23	应用程序扩展	133 KB
wxbase30u_vc90.dll	2013/12/28 2:21	应用程序扩展	1,978 KB
wxbase30u_net_vc90.dll	2013/12/28 2:21	应用程序扩展	152 KB
python27.dll	2012/4/10 23:31	应用程序扩展	2,250 KB
msvcr90.dll	2013/8/4 15:37	应用程序扩展	641 KB
msvcp90.dll	2013/8/4 15:37	应用程序扩展	556 KB
msvcm90.dll	2013/8/4 15:37	应用程序扩展	220 KB
OManager.exe	2014/7/13 10:11	应用程序	2,121 KB
MD5sum.exe	2013/7/21 21:07	应用程序	791 KB
wx_xrc.pyd	2014/6/29 11:50	PYD 文件	146 KB

图 16-23 打包后生成的文件列表

最后一步就是制作安装包，我们可以简单对目录制作压缩包发布，也可以使用更加专业的安装包制作工具，如 Advanced Installer、Inno Setup、Smart Install Maker 等，最终将生成一个安装包文件“Setup.exe”，单击安装后的效果见图 16-24。



图 16-24 系统安装界面



参考  
提示

RC4 加密算法参考文章 <http://www.snip2code.com/Snippet/27937/Blockout-encryption-decryption-methods-p>。

随着移动互联网的普及，拥有超大用户规模的应用和服务越来越多，服务器运维所面临的挑战也随之越来越大。当规模增长到一定程度时，手动管理的方式自然无法应对，于是自动化运维成为解决问题的银弹。在自动化运维方面，已经有大量优秀的开源工具和最佳实践，Python凭借其灵活性，在自动化运维方面具有先天优势，已经被广泛使用，而且基于Python编写了很多自动化的运维工具，这些工具能大大提高运维的效率，服务器集群的规模越大，优势越明显。即便不使用工具，很多运维工作也能通过几行简单的Python语句来实现自动化操作，简单、方便。

本书作者先后在国内著名的天涯社区和腾讯从事运维工作近10年，不仅是公司内部的技术核心人物之一，而且在中国整个运维技术圈子内都有很高的知名度，被视为偶像级运维专家。他对Python在运维领域的应用有非常深入的研究，而且在腾讯的生产环境中得到了应用和实践，无论是知识还是经验，都非常宝贵。



中国领先的IT技术学习服务提供商

51CTO.com  
技术成就梦想



投稿热线: (010) 88379604  
客服热线: (010) 88378991 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导: 计算机/Linux

ISBN 978-7-111-48306-9



9 787111 483069 >

定价: 69.00元