

奔跑吧 Ansible

Ansible: Up and Running 让自动化配置管理与部署变得简单

[加] Lorin Hochstein 著
陈尔冬 译



中国工信出版集团

电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

O'REILLY®

奔跑吧Ansible

Ansible: Up and Running

[加] Lorin Hochstein 著

陈尔冬 译

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

内 容 简 介

Ansible是近年用户量急速蹿升的开源配置管理工具。在Ansible之前，行业中已经有很多开源配置管理工具了，特别是鼎鼎大名的Puppet，堪称配置管理界的超级巨星。然而，Ansible依靠它的简单易用、“零依赖”及弱抽象还是获得了无数开发者和运维工程师的青睐。遗憾的是，由于Ansible还很年轻，除了官方文档外，其他相关的优秀文档可谓凤毛麟角。而本书恰恰就是为了缓解这一现状而写的。作者在本书中演示了如何使用Ansible在接近真实的生产环境进行管理的案例，这既演示了Ansible的强大功能，又能够帮助读者快速入门与上手，非常适合作为官方文档的扩展资料来阅读。

© 2015 by Lorin Hochstein

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2016. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由O'Reilly Media, Inc. 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2015-5385

图书在版编目（CIP）数据

奔跑吧Ansible / (加) 霍克斯坦 (Hochstein,L.) 著；陈尔冬译. —北京：电子工业出版社，2016.1

书名原文：Ansible: Up and Running

ISBN 978-7-121-27507-4

I . ①奔… II . ①霍… ②陈… III . ①程序开发工具 IV . ①TP311.52

中国版本图书馆CIP数据核字(2015)第263674号

策划编辑：张春雨

责任编辑：付 睿

封面设计：Ellie Volkhausen 张 健

印 刷：北京天宇星印刷厂

装 订：北京天宇星印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：787×980 1/16 印张：21.75 字数：476千字

版 次：2016年1月第1版

印 次：2016年1月第1次印刷

定 价：79.00元

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

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

服务热线：(010) 88258888。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

推荐语

大规模集群的配置管理工具一直都是提升效率的利器。新浪在 2003 年前后开始使用 Cfengine；在 2010 年前后，为了解决 Cfengine 的一些问题，我们逐渐在一些业务中用 SSH 进行批量命令执行和配置文件拉取，这种组合一直用到现在，管理着数万台服务器，但现在基本上已经是 SSH 为主了。本文的译者在新浪负责了很多配置管理系统的开发，以我们的经验来看，基于 SSH 进行配置管理有很多好处，这也是我对 Ansible 很感兴趣的地方。充分利用系统现成的机制，这不仅省去了 Agent 的安装，在数千台规模的大型数据中心，少一些中心化服务还意味着少了一些路由和安全策略的配置，省去了很多不必要的麻烦。当然，Ansible 还有很多吸引人的地方，非常值得你去学习和发现。

——童剑
新浪研发中心总经理

Ansible 作为自动化系统运维的一大利器，在构建整个体系过程中有着举足轻重的地位。DSL、幂等性、playbook、大量的模板等都是它的魅力所在，再加上易封装、接口调用方便，Ansible 正在被越来越多的大公司采用，包括独立使用以及与其他工具（如 Puppet）结合使用。

本书通过简单易用的大量实例帮忙读者快速上手，通读全书会让你对 Ansible 有一个全面的了解，动手操作书中的实例，快速掌握 Ansible，剩下的就交给你的灵感吧。

——刘宇
@守住每一天，金山西山居架构师
《Puppet 实战》作者，《Puppet 实战手册》译者之一

对于 DevOps，我自己的理解是运维即开发，人管理代码，代码管理机器，而不是人直接管理机器。Ansible 帮我们实现了运维人员开始向运维开发的转型，让繁杂、危险的运维工作变得简单、安全和可控。《奔跑吧 Ansible》秉承了 Ansible 这一工具简明的一贯特点，不是长篇累牍地讲解复杂技术原理，而是列举了大量简明的实例，拿来阅读半小时即可上手解决实际问题。

——程辉
UnitedStack 公司创始人兼 CEO

推荐序一

几年前，我和尔冬都作为技术团队的一员在一起开始新浪微博的研发，研发团队开发完第一个服务后就面临服务发布及管理的问题。最开始是通过工程师手动登录服务器的方式来发布与启动。当服务请求增大、服务单元增多后，工程师便将命令写成脚本，通过 SSH 在多台机器执行，半人工地解决了一段时间的问题。当服务规模进一步增大，手工运行 SSH 变得困难后，我们开始寻找一些工具来使这些工作自动化，当时能找到的工具都需要在机器上安装一个 agent，通过 agent 接收指令来执行对应的操作。随着服务集群规模变大、依赖变多，当时使用的工具需要具备指令安全、执行进度汇报、执行结果检测等功能。于是我们自己做了一个脚本，可以通过命令行的方式控制软件发布、服务启动与进度查看。随着机器规模的增加，命令行及脚本的方式显得不是很直观，也不利于新进入团队的人了解及使用。于是我们做了一个 Web 界面来中心化管理及执行脚本，并且可以很直观地查看进度，还可以选择发布任务执行的范围，较好地解决了当时服务发布遇到的问题。

几年之后，我们发现 Ansible 使用相同的理念更好地解决了以上问题。其设计方法也遵循“脚本 + 可视化管理”的思路，Ansible 本身是一种脚本控制的语言，在此之上，我们可以选择其商业化的 Tower 软件来可视化地进行管理。而其脚本在指令安全、执行进度汇报、执行结果检测方面，相对于直接运行操作系统脚本都具有更大的优势。

自动化管理是大势所趋，尔冬及时翻译的《奔跑吧 Ansible》一书可以给国内同行带来很多启发，希望大家能够利用 Ansible 工具及设计思想将应用的发布和配置管理提升到新的高度。

——Tim Yang
微博研发副总经理，“高可用架构”公号主

推荐序二

运维发展到今天，已经不是刀耕火种的年代，各种运维自动化工具层出不穷，运维人员也逐步摆脱了直接登录服务器进行操作所带来的繁琐、重复和高风险性。自动化工具也有一个演进过程，从 Puppet、Saltstack 到 Chef，运维人员在学习和使用的过程中也深受其复杂性而苦恼，特别是客户端机制。相比之下，Ansible 的使用简单很多，这也是它广受欢迎的主要原因。

运维自动化工具本来是用来简化运维工作的，但如果工具本身比较复杂，甚至需要一定的程序开发能力，就会增加使用和推广的难度。Ansible 有三个最吸引人的地方：无客户端、简单易用和日志集中存储。很多时候，运维人员对服务器仅具有临时权限，甚至没有权限，所以无法部署客户端程序。另外，客户端机制往往也是运维自动化容易出问题的地方，这涉及客户端的安装、配置修改和卸载，其中任何一步没有同步完成，都可能会带来隐患。Ansible 很简单，上手方便，不需要啃一本大部头书才能学会使用（从这一点来看，真可谓业界良心）。另外 Ansible 又很好地解决了 Shell 操作日志的集中存储问题——这一点在被管理服务器数量少时，貌似作用还不大，但在批量管理大量服务器时，就能显示出其便利性了。所有操作日志都存储在 Ansible 发起服务器，可以采用自定义的格式，这样可以很方便地知晓哪些服务器的操作有问题，哪些已成功，而且便于日后的追溯。当然，Ansible 的配置管理功能简单而强大，所有被管理的软件配置都集中存储，如果目标服务器想安装 MySQL，InnoDB 所占用内存需要从 10GB 调整为 12GB，那么在 Ansible 发起服务器简单修改即可。

我对 Ansible 的印象非常好。在 2014 年年底的时候，为了给公司采购公有云提供决策支持，我选定了国内八大公有云，对它们进行了长达四个月的测试，总共进行了上万次测试。每家公有云随机选择 3~5 台云主机和 RDS，每台云主机测试多轮，每轮测试 200 次。测试项目包括网络稳定性（探测节点涵盖全国共 400 多个）、CPU、内存、磁盘及云主机

整机性能测试。如果不借助于 Ansible，仅仅两个人是无论如何也不可能在短时间内完成一万多次测试的。我们基于 Ansible 编写了一个批量自动化测试工具，这个工具完成了对新入手云主机的一切工作，包括初始化、测试工具部署、测试数据装载及自动发起指定次数的测试流程。结果很完美，所有测试结果都汇聚在 Ansible 发起服务器，我们又开发了一个日志自动分析的工具，能从 Ansible log 中截取有用的信息并加以汇总（就只差用 Excel 出图了）。

同时，Ansible 虽然不如 Puppet 等复杂，但也还是需要一些方法和技巧，而且版本更新迭代较快。因此我也很高兴看到尔冬兄亲自翻译的《奔跑吧 Ansible》这本书，其英文原版由 Michael DeHaan (Ansible 软件的创作者，Ansible 公司前 CTO) 亲自作序，其受认可度可见一斑。本书内容编排由浅入深，理论与实践并重，作者特别提到了 SSH multiplexing，Ansible 与之配合，可以用来管理成千上万的服务器节点。

尔冬兄是运维行业资深人士，深度参与了新浪微博从小到大的发展过程。每次和尔冬兄交流，总能感觉到他对运维行业深深的感情，以及关于运维的真知灼见。这次尔冬兄亲自翻译此书，可以说是国内诸多 Ansible 使用者的福祉，可以帮助大家更好地学习、理解、掌握 Ansible，并融会贯通。

——萧田国

开放运维联盟联合主席，高效运维社区发起人

译者序

由于诸多原因，早在童年时代计算机就进入了我的生活。对计算机的迷恋最终将我带进了计算机工程领域。而让我真正从玩耍转向工程化地对待计算机的分水岭就是，对系统管理领域的接触。好吧，不管多么不乐意，这类工作在那个年代就是被称作网管。从我刚开始入门系统管理的时候，我就一直有一个疑问：如果一家公司足够大，有上百台计算机，该怎么管呢？总不能一个个远程桌面连上去吧？嗯，没错，那个时候我使用的还是 Windows，而上百台计算机对于当时的我来说已经是一个很夸张的数字了。

2007 年我加入了新浪平台架构部，这里的工作为我真正打开了新世界的大门。那个时候部门正在使用 CFEngine 管理上百台机器、为不同功能的服务器划分角色、为相同功能的服务器进行编号（像为公牛编号那样）、为配置文件编写模板来减少硬编码，所有这一切都用一种工程实践的方法解决了我之前的所有疑问。后来，我才知道这个实践性很强的方法的名字——配置管理。

转眼之间，我已经在新浪工作了七年之久。这七年间我的职位与工作内容有多次变化，但所围绕的工作核心从未改变：如何让数千台服务器按照我们想要的方式运转。为了更好地达到这一目的，我尝试过各种配置管理的方式：从 CFEngine 到 Puppet、SaltStack，甚至是自行开发配置管理工具。但是每一种方式都与我理想中那个遵循“KISS 原则”、易于学习，且在功能上具有无穷扩展空间的配置管理工具相差甚远——直到经同事文旭的推荐，我认识了 Ansible。Ansible 的轻量、最小化抽象层及轻松扩展与收缩一下子就吸引了我。目前为止，它是与我理想中的配置管理工具最接近的一个。

这样优秀的工具我当然不会自己独享。我曾经在各种场合向正在寻找合适配置管理工具的朋友与同事推荐 Ansible。我发现有一部分朋友虽然对于 Ansible 给予了正面的评价，却对缺少中文文档感觉略有不便。这让我意识到语言仍旧是部分技术人员学习技术的障碍之一。显然，我并不具备帮助技术人员提升英语阅读水平的能力，但至少我可以将这

本《奔跑吧 Ansible》的中文版带给大家，希望本书可以帮助一些读者快速上手 Ansible。

由于水平所限，本书中难免出现一些翻译错误。诚恳地欢迎大家向我或者出版社反馈本书中的各种错误。

最后，我想要感谢赵新宇、陈明杰和刘宇等朋友，他们在翻译本书过程中提供了无私的帮助和支持。还要感谢我的夫人张若金的支持与理解。没有你们就不会有本书的出版。

原书推荐序

在 2012 年 2 月创立的时候，Ansible 还是一个非常简单的项目，随后它的快速发展令我们倍感惊喜。现在，它已经是上千人参与开发的产品了（如果包括参与贡献想法的人，还会更多），并且广泛部署于几乎每个国家。在各种技术会议中总是能找到有（至少）几个人在使用它，这在计算机领域也是件很不寻常的事。

Ansible 的不平凡源自于它的平凡。Ansible 并不企图做盘古开天地般的创新，而是从那些聪明的家伙们已经提出的想法中提炼出精华，并将这些想法尽可能地落地。

Ansible 旨在探求某些学术的 IT 自动化方法（它们本身就是对大型繁杂的商业套件的一种反应）与简单粗暴解决问题的脚本之间的平衡点，另外，我们如何能将配置管理系统、部署发布系统、编配系统（orchestration project）以及千奇百怪但是非常重要的 Shell 脚本库替换为一个单一系统呢？这恰恰是 Ansible 要实现的理念。

我们可以从 IT 自动化技术栈中移除主要架构组件吗？去掉管理性守护进程，转而依赖于 OpenSSH，意味着系统转眼间就可以开始管理一台新的计算机，而不需要在被管理的机器上安装任何东西。更深一层来说，系统更趋于可靠和安全了。

我注意到，提前尝试使用自动化系统本该使事情变得简单，但实际上却变得更难了。并且编写以自动化在以前为目的的东西就好像个吸收时间的黑洞，使我无法在本应该更专注的事情上投入更多时间。况且我并不想在这种系统上投入数个月以成为这个领域的专家。

我个人尤其享受编写新的软件，而不喜欢在使其自动化方面花太多时间。简而言之，我希望自动化的事情尽快完成，这样我就能有更多时间投入在我更关注的事情上面。Ansible 并不是一个你需要整天和它打交道的系统。你可以很快把它拿起来，很快搞定，然后又很快回到你更关心的事情上面。

我希望这些也会成为你喜欢 Ansible 的原因。

尽管我花了大量时间来确保 Ansible 的文档易于理解和掌握，但是有不同形式的材料可以参考，并依此尝试实践应用总是大有裨益的。

在《奔跑吧 Ansible》一书中，Lorin 使用非常流畅的行文、适于逐步探索的顺序介绍了 Ansible。Lorin 几乎是从最开始就参与到了 Ansible 项目中，我真诚地感谢他做出的贡献。

我还要真诚地感谢今天项目中的每一位成员，以及未来的每一位成员。

最后，希望你们喜欢这本书，享受瞬间就可以管理你的计算机的愉悦感！啊，对了，别忘记安装 cowsay^{注1}！

——Michael DeHaan

Ansible 软件的创作者，Ansible, Inc. 公司前 CTO^{注2}

2015 年 4 月

注 1：Ansible 的创作者很有趣。从 0.5 版本开始他给 Ansible 留了个彩蛋：如果你的机器安装了 cowsay 的话，执行 playbook 的时候，终端上就会显示一只奶牛。最主要的是，他还在发布 0.5 版本的时候严肃地介绍了这么做的优点。——译者注

注 2：如 Michael DeHaan 自己所说，他其实更享受编写新软件。目前他已经从 Ansible 公司离职到 DataStax 工作。在 Ansible 公司的最后工作日，他写下了一篇博文：*Happy Trails, Ansible* (<http://michaeldehaan.net/post/109595670406/happy-trails-ansible>)。——译者注

前言

我为什么写这本书

多年前，我使用时下流行的 Python Web 框架 Django 编写了我人生中的第一个 Web 应用，当那个应用终于在我的台式机成功运行时，油然而生的成就感让我至今难忘。首先运行 `django manage.py runserver`，然后打开浏览器并访问 `http://localhost:8000`，最后是见证奇迹的时刻——我的 Web 应用闪亮登场！

然而，随后我就发现让该死的应用运行在 Linux 服务器上其实有非常多令人无奈的事情。除了把 Django 本身和我的应用安装在服务器之外，我还必须得安装 Apache 和 `mod_python` 模块。有了 `mod_python` 模块，Apache 才可以运行 Django 应用。这还不算完，我还必须弄明白天书一般的 Apache 配置文件，并把它配置为可以同时运行我的程序和其他所依赖的静态内容。

说实在的，每一步都不算难，但是想让所有的环节都配置正确也是个痛苦的过程。作为一名程序员，我才不想不停地摆弄配置文件，我只是想让我的应用运行而已。好在当我把一切都配好了之后，就不用再去动了。然而几个月后的一天，噩耗传来，在另一台服务器上还得再经历这些无奈，而且还是从头开始。

终于有一天，我发现痛苦的根源在于我用的方法不对。做这些事情的正确方法有个学名：配置管理。使用配置管理要注意的一个重要的事情是：它是一种获取那些始终保持最新的配置与信息的方法。你不再需要频繁地搜索正确的文档或者查找以前的笔记。

不久以前，一位同事出于兴趣尝试使用 Ansible 部署新项目，他请我帮忙推荐一些官方文档以外的 Ansible 应用实践方面的参考资料。那时我才突然发现，好像官方文档以外没有什么资料可以推荐的了。于是我决定填补这个空白，所以就有了这本书。可惜这本

书对于那位同事来说太迟了，但希望对于你来说它来得正是时候。

谁适合读这本书

这本书是写给需要管理 Linux 或者类 Unix 服务器的人的。如果你对下列术语如数家珍：系统管理、运维、部署、配置管理（看到这也许有的读者在叹气）或者 *DevOps*，那么看到本书你也许会觉得如获至宝。

尽管我也在管理自己的 Linux 服务器，但我的专业领域其实是软件开发。这意味着本书的范例将偏向于部署领域，尽管我同意 Andrew Clay Shafer（见附录 D 参考文献 webops）的部署与配置没有明确边界的观点。

本书引导

我对这样的提纲并不感冒：“第 1 章涵盖这个；第 2 章涵盖那个”，类似这样的内容不会有人真正仔细去看（反正我从没看过）。我认为目录反倒更易于浏览。

本书的内容编排是适于从前向后顺序阅读的，后面的章节会基于前面的内容。本书包含大量范例，建议你按照本书指导在自己的计算机上去试验这些范例。绝大部分范例都是面向 Web 应用的。

本书约定

本书使用的排版约定如下：

斜体 (*Italic*)

新术语、URL、电子邮件地址、文件名和文件扩展名以这种字体展示。

等宽字体 (*Constant width*)

程序代码或正文中包含的程序元素，如变量、函数名、数据库、数据类型、环境变量、语句和关键字等代码文本以这种字体展示。

等宽黑体字 (*Constant width bold*)

需要用户输入的命令或其他文本以这种字体展示。

等宽斜体字 (*Constant width italic*)

需要由用户替换为具体内容的文本，或需要遵照具体上下文确定具体内容的文本由这种字体展示。

这个图标表示小窍门、建议或是一般注意事项。



这个图标表示一般性的提示。



这个图标表示警告或提醒。



本书切口处  中的页码对应英文原书页码。

在线资源

本书中的样例代码都可以在本书的 GitHub 页面 (<http://github.com/lorin/ansiblebook>) 上获取到。Ansible 官方也提供了丰富的文档 (<http://docs.ansible.com>) 供参考。

我还在 GitHub 上维护了一个简明参考手册 (<http://github.com/lorin/ansible-quickref>)。

Ansible 的代码存放在 GitHub 上，分割成三部分分别存放在不同的代码仓库上：

- 主仓库 (<https://github.com/ansible/ansible>)。
- 核心模块 (<https://github.com/ansible/ansible-modules-core>)。
- 其他模块 (<https://github.com/ansible/ansible-modules-extras>)。

建议将 Ansible 的模块索引加到你的收藏夹中，在使用 Ansible 过程中，你将会时常需要去查看它，网址为：http://docs.ansible.com/modules_by_category.html。Ansible Galaxy 是一个由社区共同维护的 Ansible 角色仓库，网址为：<https://galaxy.ansible.com/>。

如果你有任何关于 Ansible 的问题，都可以到 Ansible 项目的 Google Group 中讨论：
<https://groups.google.com/forum/#!forum/ansible-project>。

如果你希望向 Ansible 开发组贡献代码，可以访问 Ansible 开发组的 Google Group：
<https://groups.google.com/forum/#!topic/ansible-devel/68x4MzXePC4>。

在 <irc.freenode.net> 上还有个非常活跃的 #ansible IRC 频道，在那里你可以得到更为实时的帮助。

本书旨在帮你解决实际问题。一般来说，除大批量使用代码之外，如果你需要在自己的程序或文档中使用本书提供的范例代码是不需要联系我们取得授权的。例如，使用本书的几段代码编写一个程序不需要向我们申请许可。但是销售或者分发 O'Reilly 图书随附的代码光盘则必须事先获得授权。引用书中的代码来回答问题也无须我们授权。将大段的示例代码整合到自己的产品文档中则必须经过我们的授权。

我们非常希望你能在引用本书时表明出处，但并不强求。出处一般包含有书名、作者、出版商和 ISBN。例如：“Ansible: Up and Running by Lorin Hochstein (O'Reilly). Copyright 2015 Lorin Hochstein, 978-1-491-91532-5”。

如果有关于使用代码的未尽事宜，可以随时与我们联系：permissions@oreilly.com。

Safari Books Online



Safari Books Online 是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

技术专家、软件开发者、Web 设计师、商务人士和创意精英都可以将 Safari 在线图书作为他们的调研、解决问题、学习和认证的主要资料来源。

Safari Books Online 对于组织团体、政府机构和个人提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sam、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 及其他数十家出版社的上千种图书、培训视频和正式出版前的书稿。要了解更多关于 Safari Books Online 的信息，请访问我们的网站。

联系我们

请将对本书的评价和发现的问题通过如下地址告知出版者。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

本书拥有一个专属网页，你可以在那找到本书的勘误表、范例代码和其他信息，网址为：
<http://bit.ly/ansible-up-and-running>。

对本书的意见和技术性问题还可以发送电子邮件到 bookquestions@oreilly.com。

想要了解更多关于 O'Reilly 图书、培训课程、会议和新闻等信息，请访问 O'Reilly 官方网站：<http://www.oreilly.com>。

Facebook 地址：<http://facebook.com/oreilly>。

Twitter 地址：<http://twitter.com/oreillymedia>。

YouTube 地址：<http://www.youtube.com/oreillymedia>。

致谢

感谢 Jan-Piet Mens、Matt Jaynes 和 John Jarvis 审阅本书的草稿并提出反馈意见。感谢在 SendGrid 工作的 Isaac Saldana 和 Mike Rowan 的大力支持。感谢 Michael Dehaan 创建了 Ansible 项目并引领着迅速崛起的 Ansible 社区。同时也感谢 Michael Dehaan 对本书的反馈，包括他对 Ansible 名字由来的讲解。感谢本书的编辑 Brian Anderson 超耐心地与我共同工作。

感谢我父母无私的支持；感谢我的哥哥 Eric，他是我们家族中的正牌作家；感谢我的两个儿子 Benjamin 和 Julian；最后，感谢我的妻子 Stacy 在各方面的付出与无条件的支持。

目录

原书推荐序	xxiii
前言	xxv
第 1 章 概述	1
关于版本的说明	2
Ansible 的优点	2
Ansible 如何运作	3
Ansible 的精妙设计有哪些	4
易读的语法	4
远程主机无须安装任何依赖	5
基于推送模式	5
Ansible 管理小规模集群	6
内置模块	6
非常轻量的抽象层	7
Ansible 太过于简单了吗	8
我需要具备哪些基础知识	9
哪些内容不会涉及	9
安装 Ansible	10
建立一台用于测试的服务器	11
使用 Vagrant 来创建测试服务器	11

将测试服务器的信息配置在 Ansible 中	15
使用 ansible.cfg 文件来简化配置	16
继续前进	20
第 2 章 playbook : 一切的开端.....	21
一些准备工作	21
一个简单的 playbook	22
指定一个 nginx 配置文件	24
创建一个定制的首页	25
创建一个 webservers 群组	25
运行这个 playbook	26
playbook 是 YAML 格式的	28
文件的起始	28
注释	28
字符串	28
布尔型	29
列表	29
字典	30
折行	30
剖析 playbook	31
play	32
task	33
模块	34
将它们整合在一起	35
执行 Ansible 后发生变化了吗？跟踪主机状态	36
来点更酷炫的：添加 TLS 支持	36
生成 TLS 证书	38
变量	38
生成 nginx 配置模板	40
handler	41
运行 playbook	43

第 3 章 inventory：描述你的服务器.....	45
inventory 文件.....	45
准备工作：创建多台 Vagrant 虚拟机	46
inventory 行为参数	49
ansible_connection	50
ansible_shell_type	50
ansible_python_interpreter.....	50
ansible_*_interpreter	50
改变行为参数的默认值	51
群组	51
范例：部署一个 Django 应用.....	52
别名和端口	55
群组嵌套	55
编号主机（宠物 vs. 公牛）	56
主机与群组变量：在 inventory 内部	56
主机和群组变量：在各自的文件中	58
动态 inventory.....	60
动态 inventory 脚本的接口	61
编写动态 inventory 脚本	62
预装的 inventory 脚本	65
将 inventory 分割到多个文件中.....	66
使用 add_host 和 group_by 在运行时添加条目	66
add_host	66
group_by	68
第 4 章 变量与 fact	71
在 playbook 中定义变量	71
查看变量的值	72
注册变量	72
fact	76
查看与某台服务器关联的所有 fact.....	77
查看 fact 子集.....	77
任何模块都可以返回 fact.....	79

本地 fact.....	79
使用 set_fact 定义新变量.....	80
内置变量.....	81
hostvars	81
inventory_hostname.....	82
groups.....	82
在命令行设置变量	83
优先级.....	84
第 5 章 初识 Mezzanine : 我们的测试应用	85
为什么向生产环境部署软件是一件复杂的事.....	85
PostgreSQL : 数据库.....	88
Gunicorn : 应用服务器.....	88
nginx : Web 服务器.....	89
Supervisor : 进程管理器	90
第 6 章 使用 Ansible 部署 Mezzanine.....	91
列出 playbook 中的 task.....	91
组织要部署的文件	92
变量与私密变量.....	92
使用迭代 (with_items) 安装多个软件包	94
向 task 中添加 sudo 语句	96
更新 apt 缓存	96
使用 Git 来 Check Out 项目源码.....	98
将 Mezzanine 和其他软件包安装到 virtualenv 中	99
task 中的复杂参数 : 稍微跑个题	102
创建数据库和数据库用户	104
从模板生成 local_settings.py 文件	105
运行 django-manage 命令	108
在应用环境中运行自定义的 Python 脚本	110
设置服务的配置文件.....	112
启用 nginx 配置文件	115
安装 TLS 证书	116

安装 Twitter 计划任务.....	117
playbook 全文	118
在 Vagrant 虚拟机上运行 playbook	121
将 Mezzanine 部署到多台主机	122
第 7 章 复杂 playbook	123
在控制主机上运行 task.....	123
在涉及的主机以外的机器上运行 task.....	124
手动采集 fact	124
逐台主机运行	125
只执行一次	126
处理不利行为命令 : changed_when 和 failed_when	127
从主机获取 IP 地址.....	131
使用 Vault 加密敏感数据	132
通过模式匹配指定主机.....	133
限定某些主机运行	134
过滤器	135
default 过滤器	135
用于注册变量的过滤器	135
应用于文件路径的过滤器	136
编写你自己的过滤器.....	137
lookup	138
file	139
pipe.....	140
env.....	140
password.....	141
template.....	141
csvfile.....	141
dnstxt.....	142
redis_kv.....	143
etcd.....	144
编写你自己的 lookup 插件	145
更复杂的循环	145

with_lines	146
with_fileglob	146
with_dict.....	147
将循环结构用作 lookup 插件	148
第 8 章 role：扩展你的 playbook	149
role 的基本构成	149
范例：Database 和 Mezzanine role	150
在你的 playbook 中使用 role	150
Pre-Task 和 Post-Task.....	152
用于部署数据库的“database” role	153
用于部署 Mezzanine 的“mezzanine” role.....	155
使用 ansible-galaxy 创建 role 文件与目录	160
从属 role	160
Ansible Galaxy.....	161
Web 界面	161
命令行工具	162
向 Galaxy 贡献你自己编写的 role.....	163
第 9 章 让 Ansible 快到飞起	165
SSH Multiplexing 与 ControlPersist	165
手动启用 SSH Multiplexing	166
Ansible 中的 SSH Multiplexing 选项.....	167
pipelining	169
启用 pipelining	169
将主机配置为支持 pipelining.....	169
fact 缓存.....	171
JSON 文件 fact 缓存后端	172
Redis fact 缓存后端	173
Memcached fact 缓存后端	173
并行性.....	174
加速模式.....	175
火球模式.....	175

第 10 章 自定义模块	177
范例：检测远程服务器是否可达	177
使用 script 模块替代编写自己的模块	177
模块形式的 can_reach	178
自定义模块该放到哪里	179
Ansible 如何调用模块	179
生成带有参数的独立 Python 脚本（仅限 Python 模块）	179
将模块复制到远程主机	179
在远程主机上创建参数文件（仅限非 Python 模块）	179
调用模块	180
预期的输出	181
Ansible 预期的输出变量	181
使用 Python 来实现模块	182
解析参数	183
访问参数	184
导入 AnsibleModule 辅助类	184
参数选项	185
AnsibleModule 的初始化参数	188
返回成功或失败	191
调用外部命令	192
检测模式（dry run）	193
文档化你的模块	194
调试你的模块	196
使用 Bash 实现模块	197
为 Bash 指定替代的位置	198
范例模块	199
第 11 章 Vagrant	201
便捷的 Vagrant 配置项	201
端口转发和私有 IP 地址	201
启用 agent forwarding	203
Ansible 置备器	203
置备器何时运行	204

由 Vagrant 生成 inventory	204
并行配置	205
指定群组	206
第 12 章 Amazon EC2.....	209
术语	211
实例.....	211
Amazon 系统镜像	211
标签.....	211
指定认证凭据	212
环境变量	212
配置文件	213
必要条件 : Boto Python 库	213
动态 inventory.....	214
inventory 缓存	216
其他配置项	217
自动生成群组.....	217
使用标签定义动态群组.....	217
把标签应用到现有资源	218
更好听的群组名	219
EC2 Virtual Private Cloud (VPC) 和 EC2 Classic	219
配置 ansible.cfg 支持使用 EC2	220
启动新的实例	221
EC2 密钥对	222
创建新的密钥.....	222
上传已有密钥.....	224
安全组.....	224
允许的 IP 地址	226
安全组端口	226
获取最新的 AMI	226
向群组中添加一个新的实例	227
等待服务器启动	230
创建实例的幂等性方法.....	231

全部加在一起	231
指定 Virtual Private Cloud	233
动态 inventory 和 VPC	237
构建 AMI	238
使用 ec2_ami 模块	238
使用 Packer	238
其他模块	242
第 13 章 Docker	243
Docker 与 Ansible 配合案例	244
Docker 应用的生命周期	244
容器化我们的 Mezzanine 应用	245
使用 Ansible 创建 Docker 镜像	247
Mezzanine	248
其他的容器镜像	253
Postgres	253
Memcached	253
Nginx	254
certs	255
构建镜像	256
部署 Docker 化的应用	256
启动数据库容器	257
获取数据库容器的 IP 地址和映射端口	257
等待数据库启动	261
初始化数据库	263
启动 Memcached 容器	264
启动 Mezzanine 容器	264
启动证书容器	265
启动 Nginx 容器	265
完整的 playbook	266
第 14 章 调试 Ansible playbook	269
调试 SSH 问题	269

debug 模块	271
assert 模块	271
在执行前检查你的 playbook	273
语法检查	273
列出主机	273
列出 task	274
检测模式	274
diff (显示文件差异)	275
限制特定的 task 运行	275
step	275
start-at-task	276
tags	276
继续向前	277
 附录 A SSH	279
 附录 B 默认设置	289
 附录 C 为 EC2 证书使用 IAM role	293
 术语	297
 参考文献	303
 索引	305

概述

在 IT 行业工作很有意思。向客户交付软件的时候，“只要在一台机器上装好软件就能搞定收工^{注1}”这种事情只能存在于理想中。实际上，我们都慢慢变成了系统工程师。

我们现在部署应用时需要把不同的服务组合起来。这些服务运行在一组分布式计算资源上，并且使用不同的网络协议相互通信。一个应用一般由 Web 服务、应用服务、内存缓存系统、任务队列、消息队列、SQL 数据库、NoSQL 数据库及负载均衡等几部分组成。

我们还得保证服务具有适当的冗余，这样出现异常（放心，异常一定会有的）时，我们的软件可以平滑地处理这些异常。配套辅助也是我们自己去部署和维护的。辅助服务一般包括日志系统、监控系统和数据分析系统。有时候辅助系统还包括我们所依赖的第三方服务，比如用于管理虚拟机实例^{注2} 的 IaaS 终端。

你完全可以手工配置这些服务：SSH 登录到一台服务器，安装软件包，编辑配置文件，然后换下一台继续。想想就痛苦！特别是在重复到三四次的时候，你会感到它出奇地费时费力、枯燥乏味，还特别容易出错。而且还有很多更复杂的任务，比如在你的应用中搭建 OpenStack 云，全手动来完成是个很疯狂的想法。幸运的是，我们有更好的方式。

既然你在读这本书，那么你应该热衷于配置管理理念，并且在考虑采用 Ansible 作为你的配置管理工具。不管你是准备向生产环境部署自己代码的开发者，还是寻求更好的自动化解决方案的系统管理员，我相信你都会觉得 Ansible 是解决你问题的完美方案。

注 1： 好吧，其实软件部署从来没这么简单过。

注 2： 可以参考 *The Practice of Cloud System Administration* 和 *Designing Data-Intensive Application*。这两本超棒的书就是描述构建和维护这种类型的分布式系统的。

关于版本的说明

本书的所有范例代码都在 Ansible 1.8.4 版本上经过了测试，这是写作本书时的最新版本。由于 Ansible 项目的主要目标之一就是向下兼容，所以这些范例应该不经修改就可以在之后的版本上正常运行。

溯源 “Ansible”

Ansible 出自科幻小说，是虚构的一种以超光速传递信息的通讯装置。Ursula K. Le Guin 在他的 *Rocannon's World* 一书中提出了这个概念。后来，有很多其他的科幻作家借鉴过这个概念。

实际上，Ansible 的作者 Michael DeHaan 是从 Orson Scott Card 撰写的 *Ender's Game* 一书中借鉴来的。在这本书中，Ansible 可以跨越任何距离同时控制无数飞船，就好像在我们的世界控制海量远程服务器一样。

Ansible 的优点

Ansible 被定义为配置管理工具，并且通常与 *Chef*、*Puppet* 及 *Salt* 相提并论。当我们提到配置管理的时候，通常都会联想到编写一个描述所有服务器状态的配置文件，并使用工具确保所有服务器都保持在那个状态之上。这些状态包括但不限于：确保所依赖的软件包已经被安装、配置文件包含正确的内容和正确的权限、相关服务被正确运行，等等。Ansible 引入一种特定领域语言（domain-specific language，缩写 DSL），你可以使用 DSL 来描述服务器的状态。

其实这些工具也可以用于部署。通常意义的部署指的是将自研软件编译成二进制文件、生成相关的静态资源、将所有必需的文件复制到服务器上并将服务启动起来这一完整过程。*Capistrano* 和 *Fabric* 就是两个开源部署工具的例子。Ansible 不仅是配置管理的利器，也是用于部署的神兵。对于负责运维的人来说，用一个简单的工具就可以同时完成配置管理和部署的工作，生活都瞬间变得美好了。

有些人会讨论部署中对编配（Orchestration）的需求。他们指的是如何解决各种远程服务器正确提供服务，以及保证各种操作以特定的顺序执行的问题。例如，你需要在启动 Web 服务器之前先启动数据库，或者为了实现零停机升级，你需要将 Web 服务器逐一从负载均衡上摘除并升级。实际上，这也是 Ansible 擅长的，Ansible 的设计初衷就是在若干服务器上从零开始执行所有必需的配置与操作。Ansible 利用极度简洁的模型来控制各种操作按照所需顺序执行。

最后，我们来聊聊新服务器的置备（provisioning）。在 Amazon EC2 这种公有云环境，置备就是指创建新的虚拟机实例。Ansible 也能覆盖这个领域。Ansible 具有大量模块可以和 EC2、Azure、Digital Ocean、Google Compute Engine、Linode、Rackspace 及其他支持 OpenStack API 的公有云通信。



本章马上会讨论到的工具 *Vagrant* 有一个特别容易引起混淆的地方，它使用术语“置备器（provisioner）”来指代用于配置管理的工具。因此，*Vagrant* 将 Ansible 看作一种置备器。而我认为 *Vagrant* 才是置备器，因为 *Vagrant* 负责创建虚拟机。

Ansible 如何运作

图 1-1 展示了一个简单的 Ansible 使用案例。有一位名为 Stacy 的用户使用 Ansible 来管理三台基于 Ubuntu 发行版并且运行 Nginx 的 Web 服务器。她编写了一个 Ansible 脚本：*webservers.yml*。Ansible 中将脚本（script）称作 *playbook*。*playbook* 描述了哪些主机（Ansible 将其称为远程服务器）需要配置，以及需要在那些主机上运行的有序任务列表。在这个范例中，主机就是 Web1、Web2 和 Web3，而任务则包括：

- 安装 nginx。
- 生成 nginx 配置文件。
- 复制 SSL 证书到相应位置。
- 启动 nginx 服务。

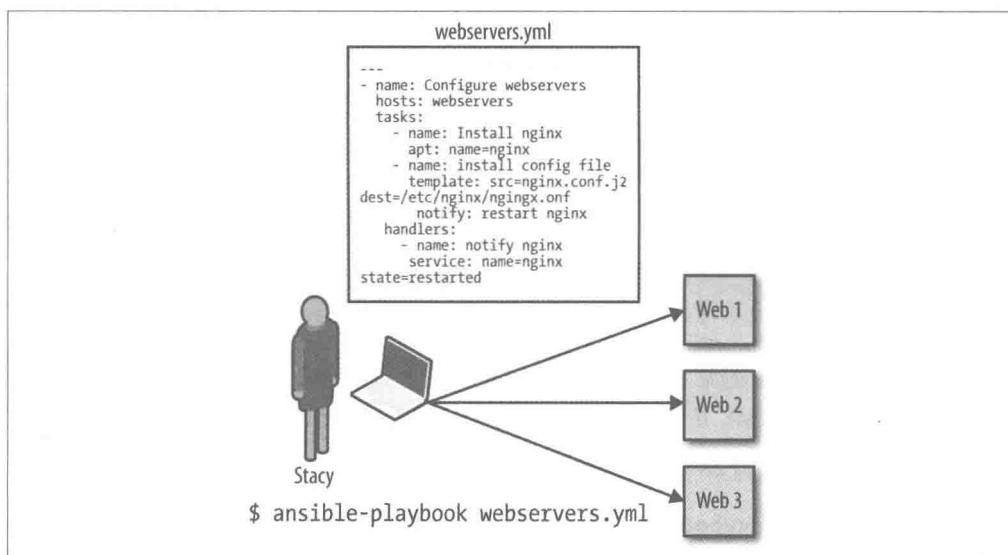


图1-1 运行一个Ansible playbook来配置三台Web服务器

我们将在下一章详细讨论 playbook。Stacy 使用命令 `ansible-playbook` 来执行 playbook。在本范例中，playbook 的名字是 `webservers.yml`，所以用以下命令执行：

```
$ ansible-playbook webservers.yml
```

4 ➤ Ansible 将会并行地建立连接到 Web1、Web2 和 Web3 的 SSH。接着，将会同时在三台主机上执行任务列表中的第一个任务。在这个范例中，第一个任务是安装 nginx 的 apt 软件包（因为 Ubuntu 使用 apt 作为包管理工具）。该任务在 playbook 中的描述如下：

```
- name: install nginx
  apt: name=nginx
```

根据上面的配置，Ansible 将会做如下操作：

1. 生成安装 nginx 软件包的 Python 程序。
2. 将此程序复制到 Web1、Web2 和 Web3。
3. 在 Web1、Web2 和 Web3 上执行此程序。
4. 等候该程序在所有主机上完成。

之后，Ansible 将会按照相同的步骤继续进行任务列表中的下一个任务。需要格外注意的是：

- 对于每一个任务，Ansible 都是在所有主机之间并行执行的。
- 在开始下一个任务之前，Ansible 会等待所有主机都完成上一个任务。
- Ansible 会按照你指定的顺序来运行任务。

5 ➤ Ansible 的精妙设计有哪些

现在有很多流行的开源配置管理工具可供选择。下面我们会聊一聊我偏爱 Ansible 的原因。

易读的语法

前面我们提到 Ansible 的配置管理脚本叫 `playbook`。Ansible 的 playbook 语法是基于 YAML 开发的。而 YAML 是一种以易于人类读写为设计理念的数据格式语言。某种程度上说，YAML 之于 JSON 就好像 Markdown 之于 HTML。

我喜欢把 Ansible 的 playbook 看作可执行的文档。这就像是一个 README 文件，它里面描述了那些为了部署你的软件所使用的命令。只不过这个命令永远不会因为忘记更新

而过期，因为它们就是直接执行的代码。

远程主机无须安装任何依赖

使用 Ansible 管理的服务器需要安装 ssh 和 Python 2.5（或更新版本），或者安装 *simplejson* 库的 Python 2.4。除此以外，不再需要预装任何 agent 程序或其他软件了。

控制主机（用于控制远程主机的那台主机）需要安装 Python 2.6 或者更高版本。



有些模块也许依赖 Python 2.5 或更新版本，也有些模块可能会需要额外的依赖。
请查阅相应模块的官方文档来确认具体依赖信息。

基于推送模式

以 Chef 和 Puppet 为代表的使用 agent 程序的配置管理系统默认使用“拉取模式”。安装在服务器上的 agent 程序定期向中心服务报备状态并拉取相应的配置信息。这种场景下，要让配置变更在服务器上生效需要如下步骤。

1. 你：对配置管理脚本进行变更。
2. 你：将变更内容更新到配置管理中心服务。
3. 服务器上的 agent 程序：当周期计时器到期时被唤醒。
4. 服务器上的 agent 程序：连接到配置管理中心服务。
5. 服务器上的 agent 程序：下载新的配置管理脚本。
6. 服务器上的 agent 程序：在本地执行那些改变服务器状态的配置管理脚本。

作为对比，我们来看一下 Ansible 默认采取的推送模式所需的变更步骤。

1. 你：在 playbook 中进行变更。
2. 你：运行新的 playbook。
3. Ansible：连接到服务器并执行那些改变服务器状态的模块

一旦你运行 `ansible-playbook` 命令，Ansible 马上连接到远程服务器并开始工作。

基于推送模式的方式最显著的优点是：直接由你来控制变更在服务器上发生的时间。你

不需要等着计时器过期。拉取模式的拥趸号称拉取模式在大规模服务器场景下具有较好的扩展性，并且更适合处理新服务器随时上下线的场景。然而，我们将在本书后面章节讨论到，Ansible 已经成功地在具有成千上万个节点的生产环境中工作，并且对于服务器动态上下线的场景可以完美支持。

如果你真的更喜欢拉取模式，Ansible 官方版本也可以支持。你可以使用一个名为 `ansible-pull` 的工具，它是在 Ansible 官方版本内一起发布的。我在本书中不会介绍拉取模式，你可以在官方文档 (http://docs.ansible.com/playbook_intro.html#ansible-pull) 中获取更多关于它的信息。

Ansible 管理小规模集群

没错，Ansible 可以轻松向上扩展至管理成百上千的节点。但是，它真正吸引我的地方是它向下收缩规模的能力。使用 Ansible 管理单一节点非常简单，你只需要编写一个 Ansible 脚本文件。Ansible 遵循了 Alan Kay 的格言“简单的事情应该保持简单，复杂的事情应该做到可能”。

内置模块

你可以使用 Ansible 在你管理的远程服务器上执行任意 Shell 命令，但是 Ansible 真正强大的地方在于内置的一系列模块。通过模块，你可以执行像安装软件包、重启服务或者复制配置文件这样的任务。

稍后我们就会看到，Ansible 模块是声明式的，可以使用它们来描述你希望服务器所处的状态。你可以像下面这样调用 `user` 模块来确保所有“web”组的服务器上都有“deploy”这个用户：

```
user: name=deploy group=web
```

另外，模块是幂等性的。如果用户“deploy”不存在，Ansible 就创建它。如果存在，Ansible 不会做任何事。幂等性是个非常赞的特性，因为它意味着向同一台服务器多次执行同一个 Ansible playbook 是安全的。相对于一般运维团队自己编写的 shell 脚本来说，这是一个非常大的优势。运维团队自己编写的脚本在第二次执行的时候很可能会带来不一样的（并且很可能是意外的）影响。

关于收敛性 (Convergence)

配置管理领域的书籍往往会展到收敛性的概念。配置管理中的收敛性与 Mark Burgess 以及他编写的配置管理系统 *CFEngine* 最紧密相关。

如果一个配置管理系统是收敛性的，那么这个系统也许需要多次运行才能将服务器置于期望的状态。而在这个过程中的每一次运行，都会使服务器更接近于那个状态。

收敛性的想法并没有被真正应用到 Ansible 中，Ansible 并没有需要多次运行来配置服务器的设计。与之相对，Ansible 的模块实现的行为是幂等性(idempotence)的，只需要运行一次 playbook 就可以将每台服务器都置为期望的状态。

如果你对 Ansible 作者对于收敛性理念的看法感兴趣，可以看看 Michael DeHaan 发布在 Ansible 项目新闻组 (<https://groups.google.com/forum/#!msg/ansible-project/WpRblldA2PQ/IYDpFjBXDlsJ>) 的文章，题目为：*Idempotence, convergence, and other silly fancy words we use to often*。

非常轻量的抽象层

某些配置管理工具提供一个抽象层，这样你就可以对运行不同操作系统的服务器使用相同的配置管理脚本来管理。例如，配置管理工具提供一个“package”抽象去替代 yum 或者 apt 等包管理工具，这样就无须处理包管理工具之间的差异了。

Ansible 的处理方式不太一样。在基于 apt 的系统上，你还是得使用“apt”模块安装软件包，在基于 yum 的系统上使用“yum”模块安装软件包。

虽然听起来这种设计是个缺点，但从实践角度来看，我认为它反倒使得 Ansible 更易用。Ansible 不需要我们去学习一套新的用于屏蔽不同操作系统差异的抽象。这使得 Ansible 更易上手，在你开始使用之前几乎不需要什么其他知识。

如果你愿意，可以在编写自己的 Ansible playbook 时，实现针对运行不同操作系统的远程服务器执行不同的操作。但是实际工作中我会尽量避免这么做。相反，我会更专注于编写用于某个特定操作系统（比如 Ubuntu）的 playbook。

在 Ansible 社区，复用的基本单元是模块。由于模块的功能范围非常小，并且可以只针对特定的操作系统，所以非常易于实现定义明确且易于分享的模块。Ansible 项目对于接受社区贡献的模块这点上非常开放。我也贡献过几个模块，因此很清楚这一点。

Ansible playbook 的设计并不是为了在不同的系统环境之间复用。在第 8 章我们将会讨论 *role*。*role* 是一种整合 playbook 的方法，所以 *role* 是可复用的。此外还有非常便利的 Ansible Galaxy，它是一个在线的 *role* 仓库。

在实际中，每家公司组建服务的方式都略有不同，针对自己公司的情况来编写 playbook 会比尝试复用别人的 playbook 更合适。但是我认为查阅其他人分享的 playbook 仍然具有重要价值，可以帮助理解工作原理。

Ansible 软件与 Ansible 公司是什么关系

Ansible 不仅是软件的名字，还是运作这个开源软件的公司的名字。Ansible 软件的创始人 Michael DeHaan 同时也是 Ansible 公司的前 CTO。为了防止混淆，我们将把软件称作 Ansible，而公司则使用“Ansible 公司”来称呼。

Ansible 公司主要经营围绕 Ansible 的培训和咨询服务。除此之外，还销售一个名为 *Ansible Tower* 的基于 Web 的管理工具。

Ansible 太过于简单了吗

在我撰写这本书的时候，我的编辑曾经向我提起有些使用“某某配置管理工具”的人说 Ansible 就是基于 SSH 的 for 循环脚本。如果你也在考虑是否要从其他配置管理工具切换到 Ansible，你可能也会担心 Ansible 的功能是否足够强大以满足你的需求。

你马上就会看到，Ansible 提供了大量 Shell 脚本难以实现的功能。除了我之前提到的提供幂等性的模块之外，Ansible 对于模板提供了完美的支持，而且在不同的范围内定义变量。任何将使用 Ansible 与使用 Shell 脚本画等号的人，大概都没有不得不维护一个由 Shell 编写的重要程序的经历。如果让我来选择用于完成配置管理任务的工具，我永远都会选择 Ansible 而不是 Shell 脚本。

9 担心 SSH 的扩展性？我们会在第 9 章将详细讨论这个问题。Ansible 使用 SSH multiplexing 来优化性能。并且，业界中已经有人使用 Ansible 来管理成千上万的节点^{注3}。



我对于其他配置管理工具并不是十分熟悉，所以无法在此比较它们的细节区别。如果你想了解针对配置管理工具的细致比较，我推荐你看看 Matt Jaynes 写的 *Taste Test: Puppet, Chef, Salt, Ansible*。十分巧合，Matt 也是 Ansible 党。

注 3： 可以参阅 Rackspace 的 Jesse Keating 的 *Using Ansible at Scale to Manage a Public Cloud* (<http://www.slideshare.net/JesseKeating/ansiblefest-rax>)。

我需要具备哪些基础知识

想要更高效地使用 Ansible 的话，你需要对基本的 Linux 系统管理非常熟悉。Ansible 可以帮你非常轻松地实现任务自动化，但它不是能将你自己都不知道如何做的工作做完的“自动化魔法”。

具体到本书，我会假定读者们至少熟悉某一个 Linux 发行版（例如 Ubuntu、RHEL、CentOS、SUSE），并且懂得如何完成如下工作：

- 使用 SSH 连接到远程服务器。
- 会处理 Bash 命令行的输入输出（管道和重定向）。
- 安装软件包。
- 使用 sudo 命令。
- 检查和设置文件权限。
- 启动和停止系统服务。
- 设置环境变量。
- 编写脚本（语言不限）。

如果你熟悉上面的所有概念，那么你就已经准备好学习和使用 Ansible 了。

我不会假定你对某一特定编程语言有所了解。例如，你根本不需要会写 Python 就可以使用 Ansible。但如果你希望能够编写自己的模块就另当别论了。

Ansible 使用 YAML 文件格式和 Jinja2 模板语言，所以要使用 Ansible 你需要学习一些 YAML 和 Jinja2 的知识，它们都很容易上手。

哪些内容不会涉及

10

本书并不是 Ansible 百科全书。本书编写的目的是为了帮你快速上手 Ansible，并演示一些执行任务的典型范例，这些范例都是无法直接在官方文档中查阅到的。

本书不会详细介绍所有 Ansible 官方模块。Ansible 有 200 多个官方模块，并且 Ansible 的模块的官方参考文档已经很棒了。

Ansible 使用 Jinja2 作为模板引擎，我仅会在本书介绍模板引擎的基本功能。这主要是因为在使用 Ansible 的过程中，我发现通常只需要用到 Jinja2 的基本功能。如果在你的模板中需要使用 Jinja2 的更多高级功能，建议查阅 Jinja2 的官方文档 (<http://jinja.pocoo.org/docs/dev/>)。

Ansible 的有些特性主要是为了更好地在旧版本 Linux 上运行，本书不会详细介绍这些特性。这些特性包括 *paramiko SSH* 客户端和 *accelerated mode*。对于这些特性相关的问题，我会给出官网文档的链接并简单略过。

在 1.7 版本中，Ansible 加入了支持管理 Windows 服务器的特性。我不会在本书中介绍关于 Windows 的内容，因为我并没有使用 Ansible 管理 Windows 服务器的经验。并且我始终认为这是一个细分场景，使用 Ansible 管理 Windows 主机领域可能适合单独写一本书。

我不会在本书讨论 Ansible Tower，它是由 Ansible 公司开发的，用于管理 Ansible 的商业 Web 工具。本书将聚焦在 Ansible 本身。而 Ansible 及其所附带的所有模块都是完全开源的。

最后，还有几个 Ansible 的特性被我忽略，以保持书的厚度可控。这些特性包括拉取模式、日志、使用非 SSH 协议连接到主机，以及交互式输入信息或密码。建议你查阅官方文档来了解这些特性的更多信息。

安装 Ansible

如果你在 Linux 上运行 Ansible，目前所有主流的 Linux 发行版都有 Ansible 的软件包，所以你可以使用原生包管理工具安装它。但是这种方法安装的 Ansible 可能是一个旧版本。如果你在 Mac OS X 上运行 Ansible，我强烈建议你使用非常赞的 Homebrew 包管理工具来安装 Ansible。

如果所有其他的方法都无法使用，你可以使用 Python 包管理工具 *pip* 来安装 Ansible。你可以用如下方法使用 root 来安装它：

```
$ sudo pip install ansible
```

11 如果你不使用 root 来安装 Ansible，则可以将 Ansible 安全地安装到一个 Python *virtualenv* 中。如果你对于 Python *virtualenv* 不熟悉，可以使用一个比较新的工具 *pipsi*，它可以帮助你自动将 Ansible 安装到 Python *virtualenv* 中：

```
$ wget https://raw.githubusercontent.com/mitsuhiko/pipsi/master/get-pipsi.py
$ python get-pipsi.py
$ pipsi install ansible
```

如果你选择使用 *pipsi* 来安装，就需要将 *~/.local/bin* 添加到你的 PATH 环境变量中。某些 Ansible 的插件或者模块可能会依赖额外的 Python 库。如果你已经安装了 *pipsi*，并且想要安装 *docker-py*（Ansible 的 Docker 模块需要此 Python 库）和 *boto*（Ansible 的 EC2

模块需要此 Python 库), 则需要进行如下操作 :

```
$ cd ~/.local/venvs/ansible  
$ source bin/activate  
$ pip install docker-py boto
```

如果你比较喜欢冒险, 希望使用最前沿版本的 Ansible, 你甚至可以直接从 GitHub 上获取开发版分支 :

```
$ git clone https://github.com/ansible/ansible.git --recursive
```

如果你选择使用开发版分支的 Ansible, 则每次都需要使用如下命令更新环境变量。这其中也包括 PATH 环境变量, 以便 shell 知道 `ansible` 和 `ansible-playbook` 程序的具体位置。

```
$ cd ./ansible  
$ source ./hacking/env-setup
```

想了解关于安装的更多细节, 可以查看如下资源。

- Ansible 官方安装文档 : http://docs.ansible.com/intro_installations.html。
- pip : <http://pip.readthedocs.org/>。
- virtualenv : <http://docs.python-guide.org/en/latest/dev/virtualenvs/>。
- pipsi : <https://github.com/mitsuhiko/pipsi>。

建立一台用于测试的服务器

如果想要跟着本书中的范例来做试验的话, 你需要拥有一台 Linux 服务器并具有 SSH 登录和 root 权限。幸运的是, 目前通过公有云服务登录一台低成本的 Linux 虚拟机非常简单便利, 例如 Amazon EC2、Google Compute Engine、Microsoft Azure^{注4}、Digital Ocean、Rackspace、SoftLayer、HP Public Cloud、Linode 等。

使用 Vagrant 来创建测试服务器

如果你不想花钱购买公有云服务, 我建议你在自己的服务器上安装 Vagrant。Vagrant 是一个优秀的开源虚拟机管理工具。你可以使用 Vagrant 在你的笔记本电脑上启动一台 Linux 虚拟机, 我们可以使用它作为测试服务器。

Vagrant 内置支持使用 Ansible 部署虚拟机, 但是我们将在第 11 章再详细讨论这个话题。现在, 我们仅仅是将 Vagrant 虚拟机当作一台普通的 Linux 服务器来管理。

12

注 4 : 是的, 你没有看错。Azure 提供 Linux 服务器。

要使用 Vagrant，需要保证你的机器上已经安装了 VirtualBox。VirtualBox 可以在 <http://www.virtualbox.org> 下载，而 Vagrant 则可以在 <http://www.vagrantup.com> 下载。

我建议你专门创建一个目录用于存储 Ansible playbook 和相关文件。在后面的范例中，我将这个目录命名为 *playbooks*。

运行下列命令将创建一个 64 位 Ubuntu 14.04 (Trusty Tahr)^{注5} 虚拟机镜像^{注6} 对应的 Vagrant 配置文件 (Vagrantfile)，并启动该虚拟机。

```
$ mkdir playbooks  
$ cd playbook  
$ vagrant init ubuntu/trusty64  
$ vagrant up
```



当你第一次运行 `vagrant up` 的时候，它将会自动下载虚拟机镜像文件。因此第一次运行 `vagrant up` 将会需要多一些的时间，具体时长取决于网络连接的速度。

如果一切正常，输出将会如下所示：

```
A `Vagrantfile` has been placed in this directory. You are now  
ready to `vagrant up` your first virtual environment! Please read  
the comments in the Vagrantfile as well as documentation on  
'vagrantup.com' for more information on using Vagrant.  
  
Bringing machine 'default' up with 'virtualbox' provider...  
==> default: Box 'ubuntu/trusty64' could not be found. Attempting to find  
and install...  
    default: Box Provider: virtualbox  
    default: Box Version: >= 0  
==> default: Loading metadata for box 'ubuntu/trusty64'  
    default: URL: https://vagrantcloud.com/ubuntu/trusty64  
==> default: Adding box 'ubuntu/trusty64' (v14.04) for provider: virtualbox  
    default: Downloading: https://vagrantcloud.com/ubuntu/trusty64/  
    version/1/provider/virtualbox.box  
==> default: Successfully added box 'ubuntu/trusty64' (v14.04) for  
    'virtualbox'!  
==> default: Importing base box 'ubuntu/trusty64'...  
==> default: Matching MAC address for NAT networking...
```

注 5：每个 Ubuntu 版本都会拥有一个由“形容词 + 动物”组成的代号。Trusty Tahr 是 Ubuntu 14.04 的代号。——译者注

注 6：Vagrant 使用术语 *machine* 代表虚拟机，术语 *box* 代表虚拟机镜像。

```
==> default: Checking if box'ubuntu/trusty64'is up to date...
==> default: Setting the name of the VM: vm_default_1409457679518_47647
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 => 2222 (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection timeout. Retrying...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Mounting shared folders...
    default: /vagrant => /Users/lorinhochstein/playbooks
```

你可以运行如下命令 SSH 到新出炉的 Ubuntu 10.04 虚拟机中：

```
$ vagrant ssh
```

如果一切顺利，你应当看到如下所示的登录信息：

```
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-35-generic x86_64)
```

```
* Documentation: https://help.ubuntu.com/
```

```
System information as of Sun Aug 31 04:07:21 UTC 2014
```

System load: 0.0	Processes: 73
Usage of /: 2.7% of 39.34GB	Users logged in: 0
Memory usage: 25%	IP address for eth0: 10.0.2.15
Swap usage: 0%	

```
Graph this data and manage this system at:
```

```
https://landscape.canonical.com/
```

```
Get cloud support with Ubuntu Advantage Cloud Guest:
```

```
http://www.ubuntu.com/business/services/cloud
```

```
0 packages can be updated.
```

```
0 updates are security updates.
```

```
Last login: Sun Aug 31 04:07:21 2014 from 10.0.2.2
```

输入 **exit** 退出 SSH 会话。

这种方法让我们直接与 Shell 交互，但是 Ansible 使用标准 ssh 客户端连接到虚拟机，而不是使用 vagrant ssh 命令。

如下操作可以告诉 Vagrant 输出 SSH 连接的详细信息：

```
$ vagrant ssh-config
```

在我的机器上，输出信息如下：

```
Host default
  HostName 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /Users/lorinhochstein/dev/ansiblebook/ch01/playbooks/.vagrant/
machines/default/virtualbox/private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

其中最重要的部分有：

```
HostName 127.0.0.1
User vagrant
Port 2222
IdentityFile /Users/lorinhochstein/dev/ansiblebook/ch01/playbooks/.vagrant/
machines/default/virtualbox/private_key
```



Vagrant 1.7 版本改变了它处理 SSH 私钥的行为。在 Vagrant 1.7 版本中，Vagrant 为每个机器都创建了一个私钥。之前的版本使用相同的私钥，该私钥存放在`~/.vagrant.d/insecure_private_key` 中。本书中的范例都使用 Vagrant 1.7。

在你的机器上，除了 IdentifyFile 的路径之外，其他每个部分都应该与范例中一样。

这些信息可以用来确认你可以从命令行发起 SSH 会话。在我的机器上，SSH 命令是这样的：

```
$ ssh vagrant@127.0.0.1 -p 2222 -i /Users/lorinhochstein/dev/ansiblebook/
ch01/
playbooks/.vagrant/machines/default/virtualbox/private_key
```

你应该能看到 Ubuntu 的登录界面。输入 **exit** 退出 SSH 会话。

将测试服务器的信息配置在 Ansible 中

Ansible 只能管理那些它明确了解的服务器。你只需要在 `inventory` 文件中指定服务器的信息就可以将这些信息提供给 Ansible。

每台服务器都需要一个用来让 Ansible 识别的名字。你可以使用服务器的主机名，或者也可以给服务器起一个别名，并传递一些附加参数来告诉 Ansible 如何连接到服务器的。我们将为 Vagrant 服务器起一个别名：`testserver`。

在 `playbooks` 目录下创建一个名叫 `hosts` 的文件。这个文件将作为 `inventory` 文件。如果你正在使用 vagrant 虚拟机作为测试服务器，你的 `hosts` 文件应该与例 1-1 所示内容很相像。我把文件内容拆分到多行是因为页面的限制，在你的文件中这些内容应该是一行。

例 1-1 `playbooks/hosts`

```
testserver ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222 \
ansible_ssh_user=vagrant \
ansible_ssh_private_key_file=.vagrant/machines/default/virtualbox/private_key
```

这里我们看到一个使用 Vagrant 的缺点。我们不得不明确地传入额外参数来告诉 Ansible 如何连接。在绝大部分情况下，我们不需要这些额外信息。

在本章稍后部分，我们将会看到如何通过使用 `ansible.cfg` 文件来避免冗长的 `inventory` 文件。在后面的章节，我们还将看到如何利用 Ansible 变量达到相近的效果。

如果你有一台运行在 Amazon EC2 的 Ubuntu 机器，主机名类似 `ec2-203-0-113-120.compute-1.amazonaws.com` 这样。那么你的 `inventory` 文件将类似如下所示（所有信息应该在一行）：

```
testserver ansible_ssh_host=ec2-203-0-113-120.compute-1.amazonaws.com \
ansible_ssh_user=ubuntu ansible_ssh_private_key_file=/path/to/keyfile.pem
```



Ansible 支持 `ssh-agent` 程序，所以你不需要在你的 `inventory` 文件中明确说明 SSH 文件。如果你以前没有使用过 `ssh-agent`，请查看在 279 页的“SSH agent”获取更详细信息。

我们将使用 `ansible` 命令行工具来验证我们是否可以使用 Ansible 连接到服务器。你不会经常用到 `ansible` 命令，它通常用于点对点的、一次性的事情上。

让我们告诉 Ansible 连接到名为 `testserver` 的服务器（在 `inventory` 的 `hosts` 文件中描述，

并且引入 ping 模块：

```
$ ansible testserver -i hosts -m ping
```

如果你的本地 SSH 客户端打开了 host key 验证功能，那么在 Ansible 第一次尝试连到主机时，你可能会看到如下信息：

```
The authenticity of host '[127.0.0.1]:2222 ([127.0.0.1]:2222)' can't be  
established. RSA key fingerprint is e8:0d:7d:ef:57:07:81:98:40:31:19:53:a8  
:d0:76:21. Are you sure you want to continue connecting (yes/no)?
```

直接输入 yes 就好。

如果执行成功，将会得到如下所示的输出：

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



如果 Ansible 的执行没有成功，可以添加 `-vvvv` 参数来查看这个错误的更多细节。

```
$ ansible testserver -i hosts -m ping -vvvv
```

我们可以看到模块执行成功了。输出中 `"changed": false` 部分告诉我们模块的执行没有改变服务器的状态。输出中 `"ping": "pong"` 是由 `ping` 模块定义的输出。

`ping` 模块除了检查 Ansible 是否可以打开到服务器的 SSH 会话之外，并不做任何其他的事情。它就是用来检测是否能连接到服务器的实用工具。

使用 `ansible.cfg` 文件来简化配置

在前面的范例中，我们不得不在 `inventory` 文件中输入很多内容来告诉 Ansible 关于测试服务器的信息。幸好 Ansible 有许多方法来让你定义各种变量。这样，我们就不需要把那些信息都堆在一个地方了。

我们马上就会使用一个这样的机制，`ansible.cfg` 文件。`ansible.cfg` 文件可以设定一些默认值，这样我们就不需要对同样的内容输入很多遍。

我应该把 `ansible.cfg` 文件放到哪里呢

Ansible 按照如下位置和顺序来查找 `ansible.cfg` 文件：

1. `ANSIBLE_CONFIG` 环境变量所指定的文件。
2. `./ansible.cfg` (当前目录下的 `ansible.cfg`)。
3. `~/.ansible.cfg` (主目录下的 `.ansible.cfg`)。
4. `/etc/ansible/ansible.cfg`。

我通常把 `ansible.cfg` 与我的 playbooks 一起放到当前目录。这样，我就可以把它和 playbooks 放到同一个版本控制仓库中。

例 1-2 展示的 `ansible.cfg` 文件指定了 inventory 文件的位置 (`hostfile`)、SSH 使用的用户名 (`remote_user`) 和 SSH 私钥 (`private_key_file`)。这个范例中假定你在使用 Vagrant。如果你在使用自己的服务器，你需要将 `remote_user` 和 `private_key_file` 设为相应的值。

例 1-2 `ansible.cfg`

```
[defaults]
hostfile = hosts
remote_user = vagrant
private_key_file = .vagrant/machines/default/virtualbox/private_key
host_key_checking = False
```

我们的范例配置还关闭了 SSH 的 host key 检查。这样当处理 Vagrant 机器时就会很方便，否则，每次我们销毁并重新创建一个 Vagrant 虚拟机之后，都需要编辑我们的 `~/.ssh/known_hosts` 文件。但是，关闭主机秘钥检查在通过网络连接其他主机时会成为安全风险。如果你对 host key 并不熟悉，请查看附录 A 中关于 host key 的更多细节。

Ansible 与版本控制

Ansible 使用 `/etc/ansible/hosts` 作为 inventory 文件的默认位置。不过，我从来不使用它，因为我希望我的 inventory 文件和 playbook 一起进行版本控制。

虽然本书没有覆盖到版本控制的内容，我仍然强烈建议你使用类似 Git 这样的版本控制系统来保管所有的 playbook。如果你是一位开发者，那么你应该对配置管理系统已经相当熟悉了。如果你是系统管理员并且还没有使用过版本控制系统，这恰好是个完美的上手时机。

有了我们设定的默认值，我们再也不需要在 *hosts* 文件中指定 SSH 用户和私钥文件了。它将简化为：

```
testserver ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
```

我们还可以在执行 *ansible* 命令时去掉 -i *hostname* 参数：

```
$ ansible testserver -m ping
```

我喜欢使用 *ansible* 命令行工具随心所欲地在远程机器上执行命令，就好像并行执行 SSH 脚本一样。你也可以使用 *command* 模块来随心所欲地执行命令。当使用这个模块的时候，你需要使用 -a 参数将需要执行的命令传入模块。

例如，想要查看服务器上的运行时间可以这么实现：

```
$ ansible testserver -m command -a uptime
```

输出应该类似如下内容：

```
testserver | success | rc=0 >>
17:14:07 up 1:16, 1 user, load average: 0.16, 0.05, 0.04
```

command 模块很常用，因而 *ansible* 命令将它设为了默认模块。所以我们也就可以使用更简单的方式执行：

```
$ ansible testserver -a uptime
```

如果我们的命令包含空格，我们需要使用引号将命令括起来，这样 shell 才会将整个字符串作为单一参数传给 Ansible。例如，要查看 */var/log/dmesg* 日志文件的最后几行：

```
$ ansible testserver -a "tail /var/log/dmesg"
```

我的 Vagrant 虚拟机上的输出类似下面这样：

```
testserver | success | rc=0 >>
[ 5.170544] type=1400 audit(1409500641.335:9): apparmor="STATUS"
operation="profile_replace" profile="unconfined" name="/usr/lib/NetworkManager/
nm-dhcp-client.act on" pid=888 comm="apparmor_parser"
[ 5.170547] type=1400 audit(1409500641.335:10): apparmor="STATUS"
operation="profile_replace" profile="unconfined" name="/usr/lib/connman/
scripts/dhclient-script" pid=888 comm="apparmor_parser"
[ 5.222366] vboxvideo: Unknown symbol drm_open (err 0)
[ 5.222370] vboxvideo: Unknown symbol drm_poll (err 0)
[ 5.222372] vboxvideo: Unknown symbol drm_pci_init (err 0)
[ 5.222375] vboxvideo: Unknown symbol drm_ioctl (err 0)
[ 5.222376] vboxvideo: Unknown symbol drm_vblank_init (err 0)
[ 5.222378] vboxvideo: Unknown symbol drm_mmap (err 0)
```

```
[ 5.222380] vboxvideo: Unknown symbol drm_pci_exit (err 0)
[ 5.222381] vboxvideo: Unknown symbol drm_release (err 0)
```

如果我们要使用root来执行，需要传入参数 -s 来告诉Ansible要 sudo 为 root 执行。例如，访问 /var/log/syslog 需要使用 root 权限：

```
ansible testserver -s -a "tail /var/log/syslog"
```

输出如下：

```
testserver | success | rc=0 >>
Aug 31 15:57:49 vagrant-ubuntu-trusty-64 ntpdate[1465]:/
adjust time server 91.189
94.4 offset -0.003191 sec
Aug 31 16:17:01 vagrant-ubuntu-trusty-64 CRON[1480]: (root) CMD (    cd / &&
run-p
rts --report /etc/cron.hourly)
Aug 31 17:04:18 vagrant-ubuntu-trusty-64 ansible-ping: Invoked with data=None
Aug 31 17:12:33 vagrant-ubuntu-trusty-64 ansible-ping: Invoked with data=None
Aug 31 17:14:07 vagrant-ubuntu-trusty-64 ansible-command: Invoked with
executable
None shell=False args=uptime removes=None creates=None chdir=None
Aug 31 17:16:01 vagrant-ubuntu-trusty-64 ansible-command: Invoked with
executable
None shell=False args=tail /var/log/messages removes=None creates=None
chdir=None
Aug 31 17:17:01 vagrant-ubuntu-trusty-64 CRON[2091]: (root) CMD (    cd / &&
run-pa
rts --report /etc/cron.hourly)
Aug 31 17:17:09 vagrant-ubuntu-trusty-64 ansible-command: Invoked with
executable=
None shell=False args=tail /var/log/dmesg removes=None creates=None
chdir=None
Aug 31 17:19:01 vagrant-ubuntu-trusty-64 ansible-command: Invoked with
executable=
None shell=False args=tail /var/log/messages removes=None creates=None
chdir=None
Aug 31 17:22:32 vagrant-ubuntu-trusty-64 ansible-command: Invoked with
executable=
None shell=False args=tail /var/log/syslog removes=None creates=None
chdir=None
```

我们可以从输出中看到，Ansible 在运行时会写入 syslog。

在使用 *ansible* 命令行工具的时候，不仅可以使用 *ping* 和 *command* 模块，还可以使用任何你喜欢的其他模块。例如，你可以像这样在 Ubuntu 上安装 nginx：

```
$ ansible testserver -s -m apt -a name=nginx
```



如果安装 nginx 失败，那么你可能需要更新一下软件包列表。为了告诉 Ansible 在安装软件包之前执行等同于 `apt-get update` 的操作，需要将参数从 `name=nginx` 变为 "`name=nginx update_cache=yes`"。

你可以使用如下操作重启 nginx：

```
$ ansible testserver -s -m service -a name=nginx  
state=restarted
```

20 因为只有 root 才可以安装 nginx 软件包和重启服务，所以我们需要使用 `-s` 参数来使用 sudo。

继续前进

简要回顾一下：在本章，我们简要介绍了 Ansible 的基本概念，包括它如何与远程服务器通信，以及它和其他配置管理工具的不同之处。我们还演示了如何使用 `ansible` 命令行工具对单台主机执行简单任务。

当然了，使用 `ansible` 对单台主机运行命令并不怎么有趣。在下一章，我们将讲解 `playbook`，这才是重头戏。

playbook：一切的开端

使用 Ansible 时，绝大部分时间将花费在编写 *playbook* 上。*playbook* 是一个 Ansible 术语，它指的是用于配置管理的脚本。让我们看一个实例：安装 Nginx Web 服务器并将其配置为可用于安全通信的状态。

如果你完全遵从本章的范例来操作，那么最后你会完成的文件列表如下：

- *playbooks/ansible.cfg*
- *playbooks/hosts*
- *playbooks/Vagrantfile*
- *playbooks/web-notls.yml*
- *playbooks/web-tls.yml*
- *playbooks/files/nginx.key*
- *playbooks/files/nginx.crt*
- *playbooks/files/nginx.conf*
- *playbooks/templates/index.html.j2*
- *playbooks/templates/nginx.conf.j2*

一些准备工作

在我们对 Vagrant 虚拟机运行这个 *playbook* 之前，需要先暴露 80 和 443 端口以便访问它们。如图 2-1 所示，我们将通过配置 Vagrant 实现对本地机器的 8080 端口和 8443 端口的请求转发到 Vagrant 虚拟机的 80 端口和 443 端口。这样我们就可以通过 `http://localhost:8080` 和 `http://localhost:8443` 访问到运行在 Vagrant 里的 Web 服务器。

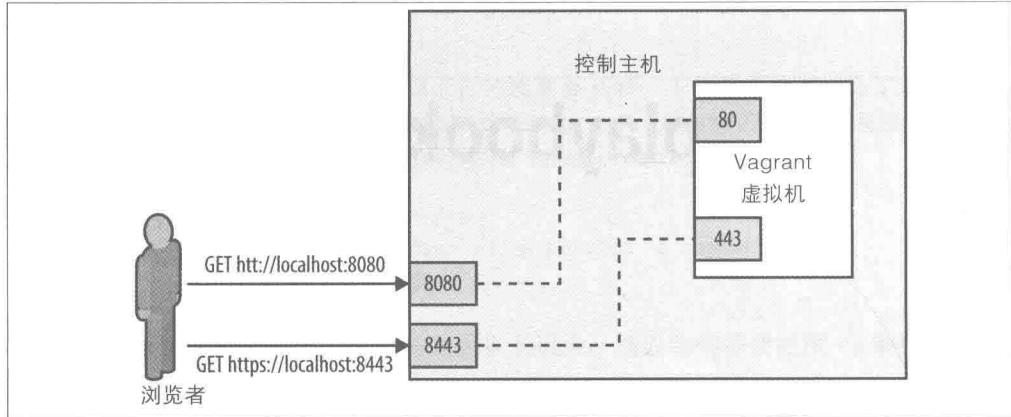


图2-1 暴露Vagrant虚拟机上的端口

如下所示修改你的 *Vagrantfile* :

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.network "forwarded_port", guest: 443, host: 8443
end
```

这会将本机的 8080 端口映射到 Vagrant 虚拟机的 80 端口，并将本机的 8443 端口映射到 Vagrant 虚拟机的 443 端口。完成变更后，需要执行如下操作来告诉 Vagrant 让新配置生效：

```
$ vagrant reload
```

你会看到输出包括如下内容：

```
==> default: Forwarding ports...
    default: 80 => 8080 (adapter 1)
    default: 443 => 8443 (adapter 1)
    default: 22 => 2222 (adapter 1)
```

一个简单的 playbook

在我们的第一个 playbook 范例中，我们将配置一台主机来运行 Nginx Web 服务器。在这个范例中，我们不会把 Web 服务器配置为支持 TLS 加密的，这将使得建立 Web 服务器更简单些。但是真正的网站应该启用 TLS 加密，我们将在本章稍后的部分介绍如何实现它。

23 首先，我们来看一下当运行例 2-1 中的代码后会发生什么，然后我们会详细分析这一段代码。

例2-1 web-notls.yml

```
- name: Configure webserver with nginx
  hosts: webservers
  sudo: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/nginx.conf dest=/etc/nginx/sites-available/default

    - name: enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link

    - name: copy index.html
      template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
      mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

为什么在有的地方使用“True”，而其他地方又使用“yes”

眼尖的读者可能已经注意到例 2-1 中，在 playbook 中的某处使用了 True（用来启用 sudo），又在 playbook 中的另一处使用了 yes（用来更新 apt 缓存）。

关于如何在 playbook 中表示“是”和“否”，Ansible 非常灵活。严格地说，模块参数（就像 update_cache=yes）对于值的处理和 playbook 中其他地方（就像 sudo:True）对于值的处理有所不同。这是因为其他地方由 YAML 解析器来处理值，所以只用 YAML 的真实性值的约定：

YAML“真”值

true, True, TRUE, yes, Yes, YES, on, On, ON, y, Y

YAML“否”值

false, False, FALSE, no, No, NO, off, Off, OFF, n, N

模块参数是作为字符串传递的，因此使用 Ansible 内置约定：

模块参数为真

yes, on, 1, true

模块参数为假

no, off, 0, false

我倾向于遵从 Ansible 官方文档的示例。向模块传递参数时候使用 yes 和 no (这是为了保持和模块官方文档一致), 而 playbook 中其他地方使用 True 和 False。

指定一个 nginx 配置文件

在这个 playbook 可以运行之前, 它还需要两个额外的文件。首先, 我们需要定义一个 nginx 的配置文件。

如果你只需要提供静态服务内容的话, nginx 随附的那个开箱即用的文件就挺好的。但是绝大多数情况下都还是需要自定义一部分配置, 所以我们还是需要用自己的配置文件覆盖默认的文件, 并将它作为 playbook 的一部分。就像我们后面部分将会看到的, 我们将需要修改 nginx 配置文件以添加对 TLS 的支持。例 2-2 是一个 nginx 的基本配置文件, 将它保存在 *playbooks/files/nginx.conf* 中^{注 1}, ^{注 2}。

例 2-2 files/nginx.conf

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    root /usr/share/nginx/html;
    index index.html index.htm;

    server_name localhost;

    location / {
        try_files $uri $uri/ =404;
    }
}
```



按照惯例, Ansible 会将一般文件放在名为 *files* 的子目录中, 将 Jinja2 模板文件放在名为 *templates* 的子目录中。我将在整本书中都遵循这个惯例。

^{注 1}：需要注意的是, 虽然我们叫这个文件 *nginx.conf*, 但实际上它替代的是 nginx 子配置文件。

^{注 2}：按 nginx 默认的配置文件分割习惯, 这个文件是配置虚拟主机的。在 nginx 中虚拟主机的术语是 Server Block。——译者注

创建一个定制的首页

让我们来添加一个定制的首页。我们将使用 Ansible 的模板功能，使 Ansible 从模板中生成这个配置文件。模板如例 2-3 所示，将它放到 `playbooks/templates/index.html.j2` 中。

例2-3 playbooks/template/index.html.j2

```
<html>
  <head>
    <title>Welcome to ansible</title>
  </head>
  <body>
    <h1>nginx, configured by Ansible</h1>
    <p>If you can see this, Ansible successfully installed nginx.</p>
    <p>{{ ansible_managed }}</p>
  </body>
</html>
```

这个模板引用了一个特别的 Ansible 变量，叫作 `ansible_managed`。当 Ansible 渲染这个模板的时候，它将会把这个变量替换为和这个模板文件生成时间相关的信息。图 2-2 展示了一张在浏览器中查看生成的 HTML 页面时的截图。



图2-2 渲染后的HTML

创建一个 webservers 群组

让我们在 inventory 文件中创建一个 webservers 群组，以便在 playbook 中引用这个群组。目前，这个群组只包含我们的测试服务器。

inventory 文件使用 `.ini` 文件格式。我们将在本书后面的章节深入研究这种格式。编辑你的 `playbooks/hosts` 文件，在 `testserver` 行上面添加一行 `[webservers]`，如例 2-4 所示。这表示这台 `testserver` 属于 `webservers` 群组。

例2-4 playbooks/hosts

```
[webservers]
```

```
testserver ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
```

现在你应该可以使用 ansible 命令行工具来 ping webservers 群组了。

```
$ ansible webservers -m ping
```

输出应该看起来像这样：

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

运行这个 playbook

执行 playbook 需要使用 ansible-playbook 命令。按照如下方法运行 playbook：

```
$ ansible-playbook web-notls.yml
```

输出应该如例 2-5 所示：

例2-5 ansible-playbook的输出

```
PLAY [Configure webserver with nginx] *****
GATHERING FACTS *****
ok: [testserver]

TASK: [install nginx] *****
changed: [testserver]

TASK: [copy nginx config file] *****
changed: [testserver]

TASK: [enable configuration] *****
ok: [testserver]

TASK: [copy index.html] *****
changed: [testserver]

TASK: [restart nginx] *****
changed: [testserver]

PLAY RECAP *****
testserver : ok=6    changed=4    unreachable=0    failed=0
```

Cowsay

如果你在本地机器安装了 *cowsay* 程序，Ansible 的输出将会变成如下所示：

```
< PLAY [Configure webserver with nginx] >
-----
 \   ^__^
  \  (oo)\_____
   (__)\       )\/\
    ||----w |
     ||     ||
```

如果你不想看到这头奶牛，你可以通过设置 `ANSIBLE_NOCOWS` 环境变量来关闭 *cowsay*。

```
$ export ANSIBLE_NOCOWS=1
```

你还可以通过在 `ansible.cfg` 中添加如下一行来关闭 *cowsay*。

```
[defaults]
nocows = 1
```

如果你没碰到任何报错^{注3}，你应能使用浏览器访问 `http://localhost:8080` 并看到自定义的 HTML 页面，输出结果如图 2-2 所示。



如果你将 playbook 文件权限设置为可执行，并且首行如下所示^{注4, 注5}：

```
#!/usr/bin/env ansible-playbook
```

这样你就可以通过直接调用它自己来执行，如下所示：

```
$ ./web-notls.yml
```

“Gathering Facts” 是做什么用的

你可能已经注意到在 Ansible 初次开始运行的时候有如下输出：

```
GATHERING FACTS *****
```

^{注3}：如果遇到了错误，你可能需要跳到第 14 章来帮助你调试。

^{注4}：这种语法特性叫作 shebang。

^{注5}：当类 Unix 操作系统中的文本文件第一个行前两个字符为 `#!` 时叫作 shebang。操作系统的程序载入器会分析 `#!` 后的内容，将这些内容作为解释器指令，调用该指令，并将这个含有 shebang 的文件及其路径作为该解释器的参数。——译者注

```
ok: [testserver]
```

当 Ansible 开始运行 playbook 的时候，它做的第一件事就是从它连接到的服务器上收集各种信息。这些信息包括：操作系统、主机名、所有网络接口的 IP 地址和 MAC 地址等。

这样，你就可以在之后的 playbook 中使用这些信息了。例如，你可能需要将主机的 IP 地址填充在配置文件中。

如果你不需要使用这些信息，你可以关闭 *fact gathering* 以节省一些时间。在后面的章节中我们会讨论如何使用这些信息以及如何关闭 fact gathering。

playbook 是 YAML 格式的

Ansible 的 playbook 是使用 YAML 语法编写的。YAML 是一种类似于 JSON 的文件格式，不过 YAML 更适合人来读写。在我们开始详细讨论 playbook 之前，让我们先了解一下编写 playbook 所必需的 YAML 语法。

文件的起始

YAML 文件以三个减号开头以标记文档的开始：

不过呢，如果你忘记在你的 playbook 文件开头敲着三个减号，并不会影响 Ansible 的运行。

注释

注释以井号开始一直到本行结束，这与 Shell、Python 及 Ruby 是一样的。

```
# This is a YAML comment
```

字符串

一般来说，YAML 字符串并不必使用引号引起，当然如果你喜欢用引号引起起来也没有问题。即便字符串中含有空格，你也完全不需要使用引号。例如，下面是一个 YAML 中的字符串：

```
this is a lovely sentence
```

其等价的 JSON 格式为：

```
"this is a lovely sentence"
```

在 Ansible 中，有几种情况比较例外，需要将字符串引起来。这些情况下，通常会引入 {{braces}} 用法，用于变量替代。我们稍后会接触到这种用法。

布尔型

YAML 具有内置的布尔类型，并且提供了多种多样的释义为“是”或“否”的字符串。我们在 22 页讨论过这个内容。

我个人一直在 Ansible playbook 中使用 True 和 False。

例如，这是 YAML 中的布尔型：

```
True
```

等价的 JSON 格式为：

```
true
```

列表

YAML 中的列表就好像 JSON 和 Ruby 中的数组或者 Python 中的列表。严格来说，在 YAML 中叫作数列。但是在这里，为了和 Ansible 官方文档保持一致，我还是叫它们列表。

列表使用减号 “-” 作为分割符，就像这样：

- My Fair Lady
- Oklahoma
- The Pirates of Penzance

等价的 JSON 格式为：

```
[  
  "My Fair Lady",  
  "Oklahoma",  
  "The Pirates of Penzance"  
]
```

(再次注意：在 YAML 中，即使字符串中含有空格，我们也不必把它们引起来。)

YAML 还为列表支持一种内联格式，就像这样：

```
[My Fair Lady, Oklahoma, The Pirates of Penzance]
```

字典

YAML 中的字典就像 JSON 中的对象、Python 中的字典或者 Ruby 中的哈希。严格来说，在 YAML 里应该叫映射。同样为了与 Ansible 官方文档保持一致，我还是叫它们字典。

字典范例如下：

```
address: 742 Evergreen Terrace
city: Springfield
state: North Takoma
```

等价的 JSON 格式：

```
{
  "address": "742 Evergreen Terrace",
  "city": "Springfield",
  "state": "North Takoma"
}
```

YAML 也为字典支持内联格式，就像下面这样：

```
{address: 742 Evergreen Terrace, city: Springfield, state: North Takoma}
```

折行

编写 playbook 的时候，你会经常碰到需要向模块传递许多参数的场景。为了美观起见，你或许希望将这些很长的段落拆分成多行，同时 Ansible 又能将其视为单行字符串。

你完全可以使用 YAML 语法来实现折行。YAML 中使用大于号 (>) 来标记折行，YAML 解释器将会把换行符替换为空格。例如：

```
address: >
  Department of Computer Science,
  A.V. Williams Building,
  University of Maryland
city: College Park
state: Maryland
```

等价的 JSON 格式为：

```
{
  "address": "Department of Computer Science, A.V. Williams Building,
             University of Maryland",
  "city": "College Park",
  "state": "Maryland"
}
```

剖析 playbook

31

让我们从一个 YAML 文件的角度重新审视我们的 playbook。例 2-6 再次列出了 web-notls.yml。

例2-6 web-notls.yml

```
- name: Configure webserver with nginx
  hosts: webservers
  sudo: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/nginx.conf dest=/etc/nginx/sites-available/default

    - name: enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link

    - name: copy index.html
      template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
      mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

例 2-7 展示了这个文件的 JSON 版本：

例2-7 web-notls.yml等价的JSON格式

```
[{"name": "Configure webserver with nginx",
  "hosts": "webservers",
  "sudo": true,
  "tasks": [
    {"name": "Install nginx",
     "apt": "name=nginx update_cache=yes"},

    {"name": "copy nginx config file",
     "template": "src=files/nginx.conf dest=/etc/nginx/
sites-available/default"}
```

```
32 >     },
    {
        "name": "enable configuration",
        "file": "dest=/etc/nginx/sites-enabled/default src=/etc/nginx/sites-
available/default state=link"
    },
    {
        "name": "copy index.html",
        "template" : "src=templates/index.html.j2 dest=/usr/share/nginx/html/
index.html mode=0644"
    },
    {
        "name": "restart nginx",
        "service": "name=nginx state=restarted"
    }
]
}
]
```



一个合法的 JSON 文件也一定是一个合法的 YAML 文件。这是因为 YAML 也允许字符串被引号引起起来，也将 `true` 和 `false` 都视为合法的布尔型，并且还支持与 JSON 数组和对象语法相同的内联列表与内联字典。但是千万别把你的 playbook 写成 JSON 版本，YAML 的亮点就是更易于人来阅读。

play

通过观察 YAML 或者 JSON 任意一种描述，可以清楚地发现 playbook 其实就是一个字典组成的列表。明确地讲，一个 playbook 就是一组 *play* 组成的列表。

这是我们范例中的 *play*^{注6}：

```
33 > - name: Configure webserver with nginx
  hosts: webservers
  sudo: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/nginx.conf dest=/etc/nginx/sites-available/default

    - name: enable configuration
```

注 6：实际上，这是只包含一个 play 的列表。

```
file: >
  dest=/etc/nginx/sites-enabled/default
  src=/etc/nginx/sites-available/default
  state=link

- name: copy index.html
  template: src=templates/index.html.j2
  dest=/usr/share/nginx/html/index.html mode=0644

- name: restart nginx
  service: name=nginx state=restarted
```

每个 play 必须包含下面两项。

- *host* : 需要配置的一组主机。
- *task* : 需要在这些主机上执行的任务。

play 就可以想象为连接到主机 (host) 上执行任务 (task) 的事物。

除了需要指定 host 和 task 之外，play 还支持一些可选配置。我们将稍后讨论这些配置，这里先介绍三个最为常见的配置。

name

一段注释，用来描述这个 play 是做什么的。Ansible 将会在 play 开始运行的时候将这段文字打印出来。

sudo

如果为真，Ansible 会在运行每个 task 的时候都使用 sudo 命令切换为（默认）root 用户。在管理 Ubuntu 服务器的时候这个配置会非常有用，因为 Ubuntu 默认不允许使用 root 用户进行 SSH 登录。

vars

变量与其值组成的列表。我们将会在本章中看到它的实例。

task

我们的范例 playbook 包含一个具有 5 个 task 的 play。下面是这个 play 的第一个 task :

```
- name: install nginx
  apt: name=nginx update_cache=yes
```

name 是可选的，所以像下面这样编写 task 是完全合法的：

```
- apt: name=nginx update_cache=yes
```

尽管 name 是可选的，我还是建议你配置它们。因为它们对于 task 的目的有非常好的提示作用。（name 配置对于其他人理解你的 playbook 非常有帮助。嗯，这个“其他人”也可能是几个月后的你自己。）就像我们已经看到过的，Ansible 会在 task 运行的时候将对应的 name 打印出来。最后，在第 14 章我们将会看到，可以使用 `--start-at-task <task name>` 参数告诉 ansible-playbook 从中间的 task 开始运行 playbook。当然，这需要你使用 name 来引用 task。

34 每个 task 必须包含由模块的名字组成的 key，以及由传到模块的参数组成的 value。在前面的范例中，模块名是 apt，参数是 `name=nginx update_cache=yes`。

这些参数告诉 apt 模块安装一个名为 `nginx` 的软件包，并在安装软件之前更新软件包缓存（与执行 `apt-get update` 命令等价）。

有一点非常重要：从 Ansible 前端所使用的 YAML 解释器角度来看，参数将被按照字符串处理，而不是字典。这意味着如果你想将参数分割为多行的话，你需要像如下这样使用折行语法：

```
- name: install nginx
  apt: >
    name=nginx
    update_cache=yes
```

Ansible 也支持能够将模块参数指定为 YAML 字典的 task 语法。当使用复杂参数的模块时，这个功能非常好用。我们将在第 102 页中的“Task 中的复杂参数：稍微跑个题”中讨论这种语法。

Ansible 还支持一个旧版本语法：它使用 action 作为 key，将模块的名字也放到 value 中。按照这个语法，前面的范例可以写成这样：

```
- name: install nginx
  action: apt name=nginx update_cache=yes
```

模块

模块是由 Ansible 包装后在主机上执行一系列操作的脚本^{注7}。这个描述看起来超级普通，但实际上 Ansible 模块千差万别。本章我们使用到的模块如下：

apt

使用 apt 包管理工具安装或删除软件包。

copy

将一个文件从本地复制到主机上。

注 7：随 Ansible 一起发布的模块都是由 Python 编写的，但实际上模块可以使用任何语言来编写。

file

设置文件、符号链接或者目录的属性。

service

启动、停止或者重启一个服务。

template

从模板生成一个文件并复制到主机上。

< 35

查看 Ansible 模块文档

Ansible 随附的 `ansible-doc` 命令行工具可以用于查看模块的文档。可以将它看作 Ansible 模块的 man 手册。例如，想要查看 `service` 模块的文档，可以运行：

```
$ ansible-doc service
```

如果你使用 Mac OS X 操作系统，有个极好的文档查看器叫作 Dash (<http://kapi.com/dash>)，它也支持 Ansible。Dash 为所有的 Ansible 模块创建了索引。Dash 是一个商业工具（在编写本书的时候售价为 19.99 美元），但是我发现它物超所值。

回顾一下第 1 章中，我们使用 Ansible 在主机上执行了一个 task，实现过程为：基于模块的名字和参数生成一个自定义脚本，然后将这个脚本复制到主机上并执行它。

Ansible 内置的模块有 200 多个，而且这个数字在每个更新的版本中都在增长。除此之外，你还可以在其他地方寻找第三方模块，或者自己编写模块。

将它们整合在一起

我们来总结一下，一个 playbook 包括一个或多个 play。一个 play 由 host 的无序集合与 task 的有序列表组成。每一个 task 仅由一个模块构成。

图 2-3 是一张实体关系图，它描述了 playbook、play、host、task 和模块（module）之间的关系。

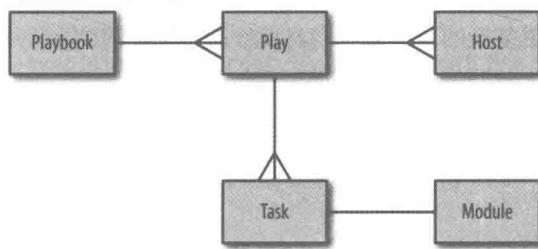


图2-3 实体关系图

36 执行 Ansible 后发生了变化了吗？跟踪主机状态

当运行 `ansible-playbook` 时，Ansible 便会输出它在 play 中执行的每一个 task 的状态信息。

我们回头看看例 2-5，需要注意的是有些任务的状态显示 `changed`，而其他的状态都是 `ok`。例如，`install nginx` 任务的状态就是 `changed`，在我的终端上呈现为黄色。

```
TASK: [install nginx] *****
changed: [testserver]
```

而另一方面，`enable configuration` 任务的状态是 `ok`，在我的终端上呈现为绿色：

```
TASK: [enable configuration] *****
ok: [testserver]
```

任何 Ansible 任务运行时都有可能以某种方式改变主机的状态。Ansible 模块会在采取任何行动之前先检查主机的状态是否需要改变。如果主机的状态与模块的参数相匹配，那么 Ansible 不会在这台主机上做任何操作，并直接响应 `ok`。

另一方面，如果主机的状态与模块的参数不同，那么 Ansible 将改变主机上的状态并返回 `changed`。

这个范例的输出表明，`install nginx` 任务的状态被改变了。这意味着在我运行这个 playbook 之前，这台主机上没有事先安装 `nginx` 软件包。而 `enable configuration` 任务的状态没有被改变，这意味着这台主机上已经有了一个配置文件，并且与我希望复制过去的相同。这是因为我在 playbook 中使用的 `nginx.conf` 文件与 Ubuntu 的 `nginx` 软件包安装的 `nginx.conf` 一模一样。

本章稍后我们将会讨论到，Ansible 检测状态变化的机制可以通过使用 `handlers` 来触发额外的操作。但是，即使不考虑使用 `handlers` 方面，这仍然是一个可以用于反馈在 playbook 运行中是否被更改状态的好办法。

来点更酷炫的：添加 TLS 支持

让我们继续尝试一个更复杂的范例：我们将要修改一下之前的 playbook，以使得我们的 Web 服务器支持 TLS。这里我们要用到的新特性包括：

- 变量
- Handlers

TLS vs. SSL

在关于 Web 服务器的安全连接的相关术语中，你可能对 SSL 比对 TLS 更熟悉一些。SSL 是用于浏览器与 Web 服务器之间安全连接的旧版协议，而它已经被名为 TLS 的新版协议所取代。

尽管很多人继续使用术语 SSL 指代当前版本的安全协议，但是我在本书中会使用更准确的叫法：TLS。

例 2-8 所示为添加了 TLS 支持配置的 playbook。

例2-8 web-tls.yml

```
- name: Configure webserver with nginx and tls
  hosts: webservers
  sudo: True
  vars:
    key_file: /etc/nginx/ssl/nginx.key
    cert_file: /etc/nginx/ssl/nginx.crt
    conf_file: /etc/nginx/sites-available/default
    server_name: localhost
  tasks:
    - name: Install nginx
      apt: name=nginx update_cache=yes cache_valid_time=3600

    - name: create directories for ssl certificates
      file: path=/etc/nginx/ssl state=directory

    - name: copy TLS key
      copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
      notify: restart nginx

    - name: copy TLS certificate
      copy: src=files/nginx.crt dest={{ cert_file }}
      notify: restart nginx

    - name: copy nginx config file
      template: src=templates/nginx.conf.j2 dest={{ conf_file }}
      notify: restart nginx

    - name: enable configuration
      file: dest=/etc/nginx/sites-enabled/default src={{ conf_file }} state=link
      notify: restart nginx

    - name: copy index.html
```

```
template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
          mode=0644
handlers:
  - name: restart nginx
    service: name=nginx state=restarted
```

生成 TLS 证书

我们需要手动生成 TLS 证书。在生产环境中，你需要从证书权威机构购买你的 TLS 证书。我们使用自签发证书主要是因为我们可以免费生成证书。

在你的 playbooks 目录下创建一个 files 子目录，然后在该目录下生成 TLS 证书和私钥：

```
$ mkdir files
$ openssl req -x509 -nodes -days 3650 -newkey rsa:2048 \
  -subj /CN=localhost \
  -keyout files/nginx.key -out files/nginx.crt
```

该命令会在 files 目录下生成文件 `nginx.key` 和 `nginx.crt`。证书的有效期是从创建之日起 10 年（3650 天）的时间。

变量

现在，在我们的 playbook 的 play 中多了一个名为 `vars` 的区段：

```
vars:
  key_file: /etc/nginx/ssl/nginx.key
  cert_file: /etc/nginx/ssl/nginx.crt
  conf_file: /etc/nginx/sites-available/default
  server_name: localhost
```

这部分中定义了四个变量并为每个变量进行了赋值。

在我们的范例中，所有变量的值都是字符串（例如：`/etc/nginx/ssl/nginx.crt`），但是任何合法的 YAML 对象都可以作为变量的值。除了字符串和布尔型之外，你还可以使用列表和字典。

变量不但可以在 tasks 中使用，而且可以在模板文件中使用。你可以使用 `{{braces}}` 的形式来引用变量。在 braces 的位置填写变量名，Ansible 将会使用变量的值替换它。

思考一下下面 playbook 中的 task：

```
- name: copy TLS key
  copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
```

当 Ansible 执行这个 task 的时候，将会使用 `/etc/nginx/ssl/nginx/nginx.key` 替换 `key_file` } }。

什么时候必须使用引号

如果你刚好在模块声明之后引用变量，YAML 解析器会将这个变量引用误解为内联字典。思考一下这个范例的运行结果：

```
- name: perform some task
  command: {{ myapp }} -a foo
```

Ansible 会尝试将 `{{ myapp }}` -a foo 的第一部分解析为字典而不是字符串，然后返回一个错误。这种情况下，你必须使用引号将参数引起起来：

```
- name: perform some task
  command: "{{ myapp }} -a foo"
```

在你的参数中含有冒号的时候也会产生类似的问题。例如：

```
- name: show a debug message
  debug: msg="The debug module will print a message: neat, eh?"
```

`msg` 参数中的冒号会误导 YAML 解析器。你需要使用引号将整个参数字符串引起起来，以避免这个问题。

不幸的是，仅仅将参数字符串引起来并不能解决这个问题。

```
- name: show a debug message
  debug: "msg=The debug module will print a message: neat, eh?"
```

嗯，这下 YAML 解析器开心了，但是输出可不是你想要的

```
TASK: [show a debug message] ****
ok: [localhost] => {
    "msg": "The"
}
```

`debug` 模块的 `msg` 参数需要将字符串使用引号引起来以捕获空格。在这个特定的范例中，整个参数字符串和 `msg` 参数都需要使用引号引起来。好在 Ansible 同时支持单引号和双引号，所以你可以这样做：

```
- name: show a debug message
  debug: "msg='The debug module will print a message: neat, eh?'"
```

这样就会得到我们期望的输出了：

```
TASK: [show a debug message] ****
ok: [localhost] => {
    "msg": "The debug module will print a message: neat, eh?"
}
```

如果你忘记在正确的地方使用引号，最终以一个非法的 YAML 文件执行，Ansible 会输出友好易读的错误信息。

40

生成 nginx 配置模板

如果你喜欢 Web 开发，那么你应该使用过模板系统来生成 HTML。不过有的读者可能没有使用过模板系统，所以在这里我还是举例介绍一下它的作用。模板就是一个文本文件，它具有一些特别的语法，可以指定变量，而这些变量最终会被替换成具体值。你可能收到过某些公司自动发送的邮件，那些邮件就很可能是由例 2-9 所示的邮件模板来生成的：

例2-9 一个邮件模板

```
Dear {{ name }},
```

```
You have {{ num_comments }} new comments on your blog: {{ blog_name }}.
```

Ansible 的使用场景并不是 HTML 或者邮件，而是配置文件。你肯定不想手动编辑配置文件，特别是你需要在多个配置文件中复用某些配置数据（例如，你队列服务器的 IP 地址或者数据库的授权信息）时，这种感觉就会更强烈了。更好的办法是：将这些具体到你的部署情况的信息记录在单一位置，然后通过模板生成所有需要这些信息的文件。

Ansible 的模板是使用 Jinja2 模板引擎来实现的。如果你以前使用过某个模板库，例如 Mustache、ERB 或者 Django 模板系统，你会感觉 Jinja2 似曾相识。

Nginx 的配置文件需要存放 TLS 证书和私钥的路径。我们将使用 Ansible 的模板功能来定义配置文件，以避免将那些可能经常变化的配置写死。

在你的 *playbooks* 目录下，创建一个名为 *templates* 的子目录，并如例 2-10 所示创建文件 *templates/nginx.conf.j2*：

例2-10 templates/nginx.conf.j2

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    listen 443 ssl;

    root /usr/share/nginx/html;
```

```
index index.html index.htm;  
  
server_name {{ server_name }};  
ssl_certificate {{ cert_file }};  
ssl_certificate_key {{ key_file }};  
  
location / {  
    try_files $uri $uri/ =404;  
}  
}
```

41

我们使用 .j2 文件名后缀来表示这个文件是一个 Jinja2 模板。不过，你可以使用其他的文件名后缀，Ansible 不会在意，只要你喜欢就好。

在我们的模板里，我们引用了下面三个变量。

`server_name`

Web 服务器的主机名（例如 `www.example.com`）。

`cert_file`

TLS 证书的路径。

`key_file`

私钥文件的路径。

我们在 playbook 中定义了这些变量。

Ansible 在 playbook 中还使用 Jinja2 模板引擎对变量取值。回忆一下我们之前在 playbook 中看到的 `{{ conf_file }}` 语法。



早期版本的 Ansible 在 playbook 中使用 \$（而非大括号）来引用变量。过去，你需要使用 `$foo` 来引用变量 `foo`，而现在则需要使用 `{{ foo }}`。`$` 已经被弃用，如果你在网上看到它了，那么说明你在看旧版本的 Ansible 代码。

你可以在你的模板中使用所有 Jinja2 特性，但是我们没办法详细讨论所有这些特性。你可以查阅 Jinja2 模板设计者文档 (<http://jinja2.pocoo.org/docs/dev/templates/>) 获取详细的信息。其实你很可能并不需要那些高级的模板功能。你最有可能在 Ansible 中用到的 Jinja2 特性是 filter，我们将在后面的章节详细讨论它。

handler

让我们回过头继续来看 `web-tls.yml` 这个 playbook。这里有两个 playbook 元素我们还没

有讨论过。其一就是 handlers 区段，就像这样：

42

```
handlers:  
  - name: restart nginx  
    service: name=nginx state=restarted
```

其二，有几个 task 中包含 notify 关键字。例如：

```
- name: copy TLS key  
  copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600  
  notify: restart nginx
```

handler 是 Ansible 提供的条件机制之一。handler 和 task 很相似，但是它只有在被 task 通知的时候才会运行。如果 Ansible 识别到 task 改变了系统的状态，task 就会触发通知机制。

task 将 handler 的名字作为参数传递，以此来通知 handler。前面的范例中，handler 的名字是 restart nginx。对于 nginx 服务器来说，我们需要在下列事件发生时重启它^{注8}：

- TLS 密钥发生变化。
- TLS 证书发生变化。
- 配置文件发生变化。
- *sites-enabled* 目录下的内容发生变化。

我们在每一个与上述事件相关的任务中都声明一个 notify，以确保这些条件触发的时候 Ansible 会重启 nginx。

关于 handler 的几件需要牢记于心的事

handler 只会在所有任务执行完后执行。而且即使被通知了多次，它也只会执行一次。handler 按照 play 中定义的顺序执行，而不是被通知的顺序。

Ansible 官方文档提到 handler 唯一的常见用途就是重启服务和重启服务器。就我个人而言，我仅用它重启服务。尽管这只是一个相当小的优化，因为我们总是可以在 playbook 的末尾无条件地重启服务，而不是在变化时候触发它，并且重启服务通常并不需要很长的时间，但是我仍然推荐使用 handler。

我曾经遇到的另一个关于 handler 的陷阱就是，在调试 playbook 的时候，handler 可能会引起一些小麻烦。事情是这样的：

1. 我运行了 playbook。

注 8： 我们也可以选择使用 state=reloaded 来重载配置文件，而不是重启服务。

2. 我的一个 task 改变了状态并发起 *notify*。
3. 后面的 task 发生了错误，Ansible 退出了。
4. 我修复了 playbook 中的错误。
5. 我再次运行 playbook。
6. 重新开始执行后并没有 task 汇报状态改变，所以 Ansible 并不会执行 handler。

43

运行 playbook

像之前一样，我们使用 `ansible-playbook` 命令来运行 playbook。

```
$ ansible-playbook web-tls.yml
```

输出应该类似下面这样：

```
PLAY [Configure webserver with nginx and tls] *****
GATHERING FACTS *****
ok: [testserver]

TASK: [Install nginx] *****
changed: [testserver]

TASK: [create directories for tls certificates] *****
changed: [testserver]

TASK: [copy TLS key] *****
changed: [testserver]

TASK: [copy TLS certificate] *****
changed: [testserver]

TASK: [copy nginx config file] *****
changed: [testserver]

TASK: [enable configuration] *****
ok: [testserver]

NOTIFIED: [restart nginx] *****
changed: [testserver]

PLAY RECAP *****
testserver : ok=8      changed=6      unreachable=0      failed=0
```

打开你的浏览器访问 `https://localhost:8443` (不要忘记在 http 后面加上 s)。如果你像我一样使用 Chrome，你会看到一个非常吓人的类似“Your Connection is not Private”(你的

连接并不私密)这样的信息(见图 2-4)。

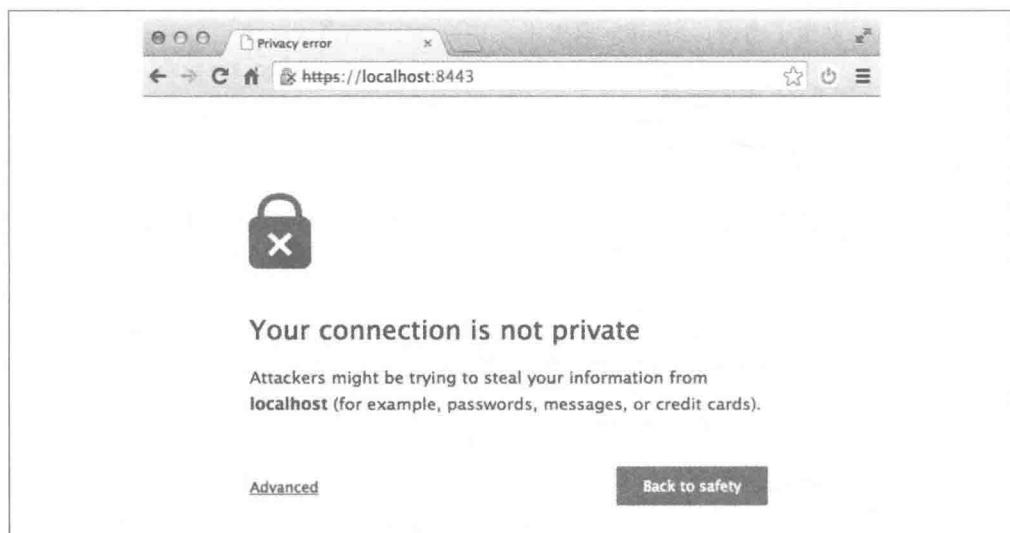


图2-4 以Chrome为代表的浏览器并不信任自签发的TLS证书

尽管报错信息很严重,但是别担心。这个错误是预料中的,因为我们生成了自签发的证书,而像 Chrome 这样的浏览器只信任正式的权威机构发行的证书。

在本章,我们讨论了很多关于 Ansible “做什么”的内容。这些内容描述了 Ansible 会在你的主机上做哪些事情。我们本节讨论的 handler 只是 Ansible 提供的流程控制机制之一。在后面的章节中,我们将会看到循环迭代和条件执行的范例。

在下一章,我们将要讨论关于“谁”的话题,具体来说就是如何定义与描述你希望 playbook 运行的目标主机。

inventory：描述你的服务器

到目前为止，我们都是对一台服务器（在 Ansible 中也称为主机）进行操作与管理。但真实情况中，你肯定要管理很多台主机。Ansible 可管理的主机集合叫作 *inventory*。

inventory 文件

在 Ansible 中，描述你主机的默认方法是将它们列在一个文本文件中，这个文本文件叫作 *inventory* 文件。一个简单的 *inventory* 文件可能只包含一组主机名的列表，就像下面例 3-1 所示：

例3-1 一个非常简单的*inventory*文件

```
ontario.example.com
newhampshire.example.com
maryland.example.com
virginia.example.com
newyork.example.com
quebec.example.com
rhodeisland.example.com
```



Ansible 默认使用你本地的 SSH 客户端，这意味着你在 SSH 配置文件中设置的别名都可以被识别到。但如果你将 Ansible 配置为使用 Paramiko 连接插件替代默认 SSH 插件的话，那些别名就不能被识别到了。

有一个主机会被 Ansible 默认地自动添加到 *inventory* 中，那就是 *localhost*。Ansible 认为 *localhost* 代表你的本机，所以在需要的时候它会直接在本机执行而不通过 SSH 连接。



尽管 Ansible 会自动将 localhost 添加到你的 inventory 文件中，你还是得在 inventory 文件中至少添加一台其他主机，否则 ansible-playbook 会报错退出：

```
ERROR: provided hosts list is empty
```

在这种 inventory 文件中没有其他主机的情况下，你可以显式地添加 localhost 记录如下：

```
localhost ansible_connection=local
```

准备工作：创建多台 Vagrant 虚拟机

为了能够讨论清楚 inventory 的特性，我们需要管理多台主机。我们需要配置 Vagrant，让它再启动三台主机。我们简单地把它们叫作 vagrant1、vagrant2 和 vagrant3。

在你更改已存在的 Vagrantfile 之前，确保你已经销毁了正在运行的虚拟机：

```
$ vagrant destroy --force
```

如果你没有使用 --force 参数，Vagrant 将会提示你确认销毁虚拟机的操作。

下一步，编辑你的 Vagrantfile，如例 3-2 所示：

例3-2 三台服务器版本的Vagrantfile

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # 为每台虚拟机使用相同的密钥
  config.ssh.insert_key = false

  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
    vagrant1.vm.network "forwarded_port", guest: 80, host: 8080
    vagrant1.vm.network "forwarded_port", guest: 443, host: 8443
  end

  config.vm.define "vagrant2" do |vagrant2|
    vagrant2.vm.box = "ubuntu/trusty64"
    vagrant2.vm.network "forwarded_port", guest: 80, host: 8081
    vagrant2.vm.network "forwarded_port", guest: 443, host: 8444
  end

  config.vm.define "vagrant3" do |vagrant3|
    vagrant3.vm.box = "ubuntu/trusty64"
    vagrant3.vm.network "forwarded_port", guest: 80, host: 8082
    vagrant3.vm.network "forwarded_port", guest: 443, host: 8445
  end
end
```

1.7 版本以上的 Vagrant 默认会为每台主机使用不同的 SSH 密钥。例 3-2 包含一行配置文件以恢复到以前版本的行为，所有主机都使用同样的 SSH 密钥：

```
config.ssh.insert_key = false
```

为所有主机使用相同密钥将简化我们的 Ansible 配置，因为我们只需要在 *ansible.cfg* 文件中指定一个 SSH 密钥。不过你需要在 *ansible.cfg* 中编辑 *host_key_checking* 的值。你的配置文件应该像例 3-3 所示：

例3-3 ansible.cfg

```
[defaults]
hostfile = inventory
remote_user = vagrant
private_key_file = ~/.vagrant.d/insecure_private_key
host_key_checking = False
```

现在，我们将假定这些服务器都可能用作 Web 服务器，所以例 3-2 将每台 Vagrant 虚拟机的 80 端口和 443 端口都映射到本机的某个端口上。

你可以通过运行下面的命令启动这三台虚拟机了：

```
$ vagrant up
```

如果所有配置都没有问题，输出应该类似下面这样：

```
Bringing machine 'vagrant1' up with 'virtualbox' provider...
Bringing machine 'vagrant2' up with 'virtualbox' provider...
Bringing machine 'vagrant3' up with 'virtualbox' provider...
...
vagrant3: 80 => 8082 (adapter 1)
vagrant3: 443 => 8445 (adapter 1)
vagrant3: 22 => 2201 (adapter 1)
==> vagrant3: Booting VM...
==> vagrant3: Waiting for machine to boot. This may take a few minutes...
vagrant3: SSH address: 127.0.0.1:2201
vagrant3: SSH username: vagrant
vagrant3: SSH auth method: private key
vagrant3: Warning: Connection timeout. Retrying...
==> vagrant3: Machine booted and ready!
==> vagrant3: Checking for guest additions in VM...
==> vagrant3: Mounting shared folders...
    vagrant3: /vagrant => /Users/lorinhochstein/dev/oreilly-ansible/playbooks
```

接下来，让我们创建一个包括这三台机器的 *inventory* 文件。

首先，我们需要知道本机的哪几个端口映射到了这三台虚拟机的 SSH 端口（22）。回忆一下，我们可以通过如下命令来得到该信息：

```
$ vagrant ssh-config
```

输出应该类似下面这样：

```
48 >  
Host vagrant1  
  HostName 127.0.0.1  
  User vagrant  
  Port 2222  
  UserKnownHostsFile /dev/null  
  StrictHostKeyChecking no  
  PasswordAuthentication no  
  IdentityFile /Users/lorinhochstein/.vagrant.d/insecure_private_key  
  IdentitiesOnly yes  
  LogLevel FATAL  
  
Host vagrant2  
  HostName 127.0.0.1  
  User vagrant  
  Port 2200  
  UserKnownHostsFile /dev/null  
  StrictHostKeyChecking no  
  PasswordAuthentication no  
  IdentityFile /Users/lorinhochstein/.vagrant.d/insecure_private_key  
  IdentitiesOnly yes  
  LogLevel FATAL  
  
Host vagrant3  
  HostName 127.0.0.1  
  User vagrant  
  Port 2201  
  UserKnownHostsFile /dev/null  
  StrictHostKeyChecking no  
  PasswordAuthentication no  
  IdentityFile /Users/lorinhochstein/.vagrant.d/insecure_private_key  
  IdentitiesOnly yes  
  LogLevel FATAL
```

我们可以看到：vagrant1 使用 2222 端口，vagrant2 使用 2200 端口，而 vagrant3 则使用 2201 端口。

像下面这样编辑你的 *hosts* 文件：

```
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
```

```
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200  
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

现在，请确认你可以顺利地登录到这些机器里。例如，要得到 vagrant2 的网卡配置信息可以运行：

```
$ ansible vagrant2 -a "ip addr show dev eth0"
```

在我的机器上运行的输出结果如下：

```
vagrant2 | success | rc=0 >>  
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP  
group default qlen 1000  
    link/ether 08:00:27:fe:1e:4d brd ff:ff:ff:ff:ff:ff  
    inet 10.0.2.15/24 brd 10.0.2.255 scope global eth0  
        valid_lft forever preferred_lft forever  
    inet6 fe80::a00:27ff:fe1e:4d/64 scope link  
        valid_lft forever preferred_lft forever
```

49

inventory 行为参数

要想在 Ansible inventory 文件中描述我们的 Vagrant 虚拟机，我们必须明确地指定相应的主机名（127.0.0.1），以及 Ansible 的 SSH 客户端可以连接到的端口（2222、2220 或 2201）。

Ansible 将这些变量命名为 *inventory 行为参数* (*behavioral inventory parameters*)。当你需要覆盖某台主机的 Ansible 默认配置的时候，你可以使用这些参数（见表 3-1）。

表3-1 inventory行为参数

名称	默认值	描述
ansible_ssh_host	主机的名字	SSH 目的主机名或 IP
ansible_ssh_port	22	SSH 目的端口
ansible_ssh_user	root	SSH 登录使用的用户名
ansible_ssh_pass	none	SSH 认证所使用的密码
ansible_connection	smart	Ansible 使用何种连接模式连接到主机（详细说明见下文）
ansible_ssh_private_key_file	none	SSH 认证所使用的私钥
ansible_shell_type	sh	命令所使用的 Shell（详细说明见下文）

续表

名称	默认值	描述
ansible_python_interpreter	/usr/bin/python	主机上的 Python 解释器（详细说明见下文）
ansible_*_interpreter	none	类似 Python 解析器配置的其他语言版（详细说明见下文）

大部分选项的名字都可以明显地表示它的意义，但是有几个需要额外解释一下。

ansible_connection

Ansible 支持多种 *transport* 机制。所谓 *transport*，就是 Ansible 使用来连接到主机的机制。默认的配置为 *smart*，智能传输模式。智能传输模式将会检测本地安装的 SSH 客户端是否支持一个名为 *ControlPersist* 的特性。如果本地 SSH 客户端支持 *ControlPersist*，Ansible 将使用本地 SSH 客户端；如果本地 SSH 客户端不支持 *ControlPersist*，那么智能传输模式将转为使用一个名为 *paramiko* 的 Python SSH 客户端库。

ansible_shell_type

Ansible 是通过 SSH 连接到远程主机，然后执行脚本来工作的。默认情况下，Ansible 假定远程 Shell 为位于 */bin/sh* 的 Bourne Shell，并会生成适用于 Bourne Shell 的命令行参数。

50

Ansible 还支持 *csh*、*fish* 和 *powershell*（在 Windows 上）作为该参数的合法值。我还从来没有遇到需要变更 Shell 类型的情况。

ansible_python_interpreter

由于 Ansible 随附的模块是使用 Python 2 实现的，Ansible 需要知道远程主机的 Python 解释器的路径。如果你的远程主机的 Python 2 解释器的路径不是 */usr/bin/python*，那么你可能需要变更这个参数。例如，你管理的主机的发行版是 Arch Linux，那么你就需要把这个参数的值变更为 */usr/bin/python2*。因为 Arch 将 Python 3 安装在 */usr/bin/python*，但是 Ansible 模块（目前）还不兼容 Python 3。

ansible_*_interpreter

如果你使用了不是由 Python 编写的自定义模块，你可以使用这个参数来制定解释器的路径（如 */usr/bin/ruby*）。我们将在 10 章讨论该内容。

改变行为参数的默认值

你可以在 `ansible.cfg` 文件（见表 3-2）的 `[defaults]` 部分更改一些行为参数的默认值。回忆一下，我们之前曾经使用这个功能改变过默认的 SSH 用户。

表3-2 可以在`ansible.cfg`中更改的默认值

inventory 行为参数	<code>ansible.cfg</code> 选项
<code>ansible_ssh_port</code>	<code>remote_port</code>
<code>ansible_ssh_user</code>	<code>remote_user</code>
<code>ansible_ssh_private_key_file</code>	<code>private_key_file</code>
<code>ansible_shell_type</code>	<code>executable</code> (参见下面一段正文)

`ansible.cfg` 中的 `executable` 配置项与 `ansible_shell_type` 行为参数并不完全一样。
executable 配置项用于指定远程主机上使用的 Shell 的全路径（例如：`/usr/local/bin/fish`）。Ansible 将会提取出此路径中的 base name（在 `/usr/local/bin/fish` 这个例子中，base name 就是 `fish`），并将其作为 `ansible_shell_type` 的默认值。

51

群组

在执行配置 task 的时候，我们更希望针对一组主机来执行操作，而不是对一个孤立的主机。

Ansible 自动定义了一个群组叫作 `all`（或者 `*`），它包括 inventory 中的所有主机。例如，我们可以通过执行如下命令来检测主机上的时钟是否大致同步：

```
$ ansible all -a "date"
```

或者

```
$ ansible '*' -a "date"
```

在我的系统中输出如下：

```
vagrant3 | success | rc=0 >>
Sun Sep  7 02:56:46 UTC 2014

vagrant2 | success | rc=0 >>
Sun Sep  7 03:03:46 UTC 2014

vagrant1 | success | rc=0 >>
Sun Sep  7 02:56:47 UTC 2014
```

我们可以在 inventory 文件中定义自己的群组。Ansible 的 inventory 文件是 .ini 格式的。在 .ini 格式中，同类的配置值归类在一起组成区段。

这里我们将演示如何将我们的 Vagrant 主机分配到名为 *vagrant* 的群组中。此外，本章开始部分提到的范例主机也将包含在这个文件中：

```
ontario.example.com
newhampshire.example.com
maryland.example.com
virginia.example.com
newyork.example.com
quebec.example.com
rhodeisland.example.com

[vagrant]
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

52 我们也可以在最上面列出 Vagrant 主机，然后再将它们加入群组，就像这样：

```
maryland.example.com
newhampshire.example.com
newyork.example.com
ontario.example.com
quebec.example.com
rhodeisland.example.com
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
virginia.example.com

[vagrant]
vagrant1
vagrant2
vagrant3
```

范例：部署一个 Django 应用

假设你负责部署一个处理长时间作业的 Django Web 应用。这个应用需要支持如下服务：

- 由 Gunicorn HTTP 服务器运行的 Django Web 应用本身。
- 架设于 Gunicorn 之前同时处理静态请求的 nginx Web 服务器。
- 为 Web 应用执行长时间作业的 Celery 任务队列。
- 作为 Celery 后端的 RabbitMQ 消息队列。

- 作为持久化存储的 PostgreSQL 数据库。



在后面的章节中，我们将从头到尾地演示一个部署这类 Django Web 应用的详细范例。不过我们在那个例子中不会用到 Celery 和 RabbitMQ。

我们需要将这个应用部署到不同类型的环境中：生产环境（最终应用工作的环境）、仿真环境（用于测试的环境，整个团队都可以共享地访问该环境中的主机），以及 Vagrant 环境（用于本地测试）。

部署到生产环境时，我们希望整个系统响应快速且稳定。所以我们：

- 为了获得更好的性能，在多台主机上运行 Web 应用并在这些主机前面设置负载均衡器。
- 为了获得更好的性能，在多台主机上运行任务队列。
- 让 Gunicorn、Celery、RabbitMQ 和 PostgreSQL 都运行在独立的服务器上。
- 用两台主机部署 PostgreSQL，一台作为主库，另一台作为从库。

< 53

假设我们有一台负载均衡器、三台 Web 服务器、三台任务队列、一台 RabbitMQ 服务器及两台数据库服务器，也就是总共有 10 台服务器需要我们来管理。

对于仿真环境，我们希望使用更少的服务器（相对生成环境）以节省成本，因为仿真环境比生产环境下的访问量少很多。比方说我们决定只使用两台主机用于仿真环境。我们将把 Web 服务器和任务队列放到一台测试环境的主机上，而把 RabbitMQ 和 PostgreSQL 放到另外一台上。

至于本地的 vagrant 环境，我们决定使用三台服务器：一台用于 Web 服务器，一台用于任务队列，而另一台将同时运行 RabbitMQ 和 PostgreSQL。

例 3-4 列出一个 inventory 文件范例。在这个范例中，服务器按照环境（生产、仿真、vagrant）和功能（Web 服务器、任务队列等）两种纬度分组。

例3-4 部署一个Django应用的inventory文件

```
[production]
delaware.example.com
georgia.example.com
maryland.example.com
newhampshire.example.com
newjersey.example.com
```

```
newyork.example.com
northcarolina.example.com
pennsylvania.example.com
rhodeisland.example.com
virginia.example.com

[staging]
ontario.example.com
quebec.example.com

[vagrant]
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201

[lb]
delaware.example.com

[web]
georgia.example.com
newhampshire.example.com
newjersey.example.com
ontario.example.com
vagrant1

[task]
newyork.example.com
northcarolina.example.com
maryland.example.com
ontario.example.com
vagrant2

[rabbitmq]
pennsylvania.example.com
quebec.example.com
vagrant3

[db]
rhodeisland.example.com
virginia.example.com
quebec.example.com
vagrant3
```

我们可以先将所有服务器列在 inventory 文件的最上面，而并不指定群组。但是这并不是必需的，而且会使得这个文件变得更长。

注意，我们仅需要为 Vagrant 实例指定一次行为参数。

别名和端口

我们按照如下所示的方式描述我们的 Vagrant 主机：

```
[vagrant]
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

在这里的 `vagrant1`、`vagrant2` 和 `vagrant3` 就是别名 (*alias*)。它们并不是真正的主机名，而是用于指代这些主机的易识别的名字而已。

Ansible 支持使用 `<hostname>:<port>` 语法来指代主机。所以我们可以使用 `127.0.0.1:2222` 来替代包含 `vagrant1` 的那一行。

但是，如例 3-5 所示的文件却并不能正常工作。

例3-5 并不能正常工作的文件

```
[vagrant]
127.0.0.1:2222
127.0.0.1:2200
127.0.0.1:2201
```

这是因为 Ansible 的 inventory 只能将一台主机与 `127.0.0.1` 相关联。所以上例中 `vagrant` 群组将只包含一台主机而不是三台。 ◀ 55

群组嵌套

Ansible 也允许你定义由群组组成的群组。例如，Web 服务器和任务队列服务器都需要安装 Django 及其依赖的软件包。如果定义一个包括 `web` 和 `task` 两个群组的 “`django`” 群组，那么解决这个问题将会非常简单。你可以在 inventory 文件中添加如下内容来达到这个目的：

```
[django:children]
web
task
```

需要注意的是，相对于主机的群组，指定群组的群组在语法上会有些变化。这是为了让 Ansible 知道应该将 `web` 和 `task` 解释为群组而不是主机。

编号主机（宠物 vs. 公牛）

例 3-4 所示的 inventory 文件看起来非常复杂。实际上，它只描述了 15 台主机——在这个充斥着海量服务器与云概念的世界里，15 台主机听起来并不是一个很大的数字。在 inventory 中，之所以定义 15 台主机就显得如此烦琐，是因为每台主机都使用了完全不一样的主机名。

微软的 Bill Baker 提出了“像对待宠物一样处理服务器”和“像对待公牛一样处理服务器”^{注 1} 的区别。我们会给宠物起有特色的名字，并且将它视作独立的个体来照顾。而另一方面，当我们聊到公牛的时候，我们会使用识别号来指代它们。

对待公牛的方法更具有扩展性，并且 Ansible 通过数字通配支持这种方法。例如，如果你的 20 台服务器名为 `web1.example.com`、`web2.example.com`，以此类推，你可以在 inventory 文件中像这样指代它们：

```
[web]
web[1:20].example.com
```

如果你习惯在首列加零（例如 `web01.example.com`），那么指定一个首列加零的范围就可以了，例如：

```
[web]
web[01:20].example.com
```

Ansible 也支持使用字母序的字符来指定范围。如果使用字母序规则的 20 台服务器名为：`web-a.example.com`、`web-b.example.com`，以此类推，那么你可以这样做：

```
[web]
web-[a-t].example.com
```

主机与群组变量：在 inventory 内部

回顾一下我们如何为 Vagrant 主机指定 inventory 行为参数的：

```
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

这些参数其实就是对于 Ansible 具有特殊意义的变量。我们也可以针对主机定义任意的变量名与相应的值。例如，我们可以定义一个名为 `color` 的变量并为每台服务器设定一

注 1：Cloudscaling 公司的 CTO Randy Bias 的演讲 (*Pets vs Cattle: The Elastic Cloud Story*, <http://www.slideshare.net/randybias/pets-vs-cattle-the-elastic-cloud-story>) 让这个比喻变得流行起来。

个值：

```
newhampshire.example.com color=red  
maryland.example.com color=green  
ontario.example.com color=blue  
quebec.example.com color=purple
```

这样，该变量就可以像其他变量一样在 playbook 中使用了。

就我个人来说，我并不经常针对主机设置变量，但是我经常针对群组设置变量。

回到我们的 Django 范例，Web 应用和任务队列服务需要与 RabbitMQ 和 PostgreSQL 通信。我们假设连接到 PostgreSQL 数据库的安全性使用网络层（只有 Web 应用和任务队列在网络可达数据库）和用户名密码两种方式来保障。而 RabbitMQ 则只在网络层上进行安全性保障。

我们需要做下列工作来把一切都搭建好。

- 为 Web 服务器配置 PostgreSQL 数据库主库的主机名、端口号、用户名、密码和数据库名。
- 为任务队列配置 PostgreSQL 数据库主库的主机名、端口号、用户名、密码和数据库名。
- 为 Web 服务器配置 RabbitMQ 服务器的主机名和端口号。
- 为任务队列配置 RabbitMQ 服务器的主机名和端口号。
- 为 PostgreSQL 数据库主库配置从库（仅限生产环境）的主机名、端口号、用户名和密码。

这些配置信息是根据环境不同而变化的，因此正好适合针对生成环境、仿真环境和 vagrant 环境的群组设置群组变量。

例 3-6 示范了一种在 inventory 文件中将这些信息定义为群组变量的方法。

例3-6 在inventory中指定群组变量

```
[all:vars]  
ntp_server=ntp.ubuntu.com  
  
[production:vars]  
db_primary_host=rhodeisland.example.com  
db_primary_port=5432  
db_replica_host=virginia.example.com  
db_name=widget_production  
db_user=widgetuser  
db_password=pFmMxcyD;Fc6)6
```

```
rabbitmq_host=pennsylvania.example.com  
rabbitmq_port=5672  
  
[staging:vars]  
db_primary_host=quebec.example.com  
db_name=widget_staging  
db_user=widgetuser  
db_password=L@4Ryz8cRUXedj  
rabbitmq_host=quebec.example.com  
rabbitmq_port=5672  
  
[vagrant:vars]  
db_primary_host=vagrant3  
db_primary_port=5432  
db_name=widget_vagrant  
db_user=widgetuser  
db_password=password  
rabbitmq_host=vagrant3  
rabbitmq_port=5672
```

需要留意的是，群组变量是如何通过 [`<group name>:vars`] 组织成若干区段的。

此外还要注意的是，如何利用 Ansible 自动创建的 `all` 群组来指定不会根据不同主机而变化的变量。

主机和群组变量：在各自的文件中

如果你所管理的主机不太多，将主机和群组变量放到 `inventory` 文件中是合理的。但随着 `inventory` 文件变得越来越大，仍然使用这种方式就会使变量越来越难以管理。

58 此外，尽管 Ansible 变量支持布尔型、字符串、列表和字典，但是在 `inventory` 文件中，你只能为变量指定布尔型或字符串。

Ansible 提供了扩展性更好的方法来玩转主机与群组变量：你可以为每个主机和群组创建独立的变量文件。Ansible 会使用 YAML 格式来解析这些变量文件。

Ansible 会在名为 `host_vars` 的目录中寻找主机变量文件，在名为 `group_vars` 的目录中寻找群组变量文件。Ansible 假设这些目录在包含 `playbook` 的目录下或者与 `inventory` 文件相邻的目录下。对我们目前的场景来说，这两个目录是同一个。

举个例子，如果我存放 `playbook` 的目录是 `/home/lorin/playbooks/`，`inventory` 文件的路径为 `/home/lorin/playbooks/hosts`，那么我将会把 `quebec.example.com` 主机的变量存放在 `/home/lorin/playbooks/host_vars/quebec.example.com` 文件中，而把 `production` 群组的

变量存放在 `/home/lorin/playbook/group_vars/production` 文件中。

例 3-7 所示为 `/home/lorin/playbook/group_vars/production` 文件的内容。

例3-7 group_vars/production

```
db_primary_host: rhodeisland.example.com
db_replica_host: virginia.example.com
db_name: widget_production
db_user: widgetuser
db_password: pFmMxxyD;Fc6)6
rabbitmq_host:pennsylvania.example.com
```

注意，我们也可以使用 YAML 字典来组织这些数值，如例 3-8 所示。

例3-8 group_vars/production 使用字典版本

```
db:
    user: widgetuser
    password: pFmMxxyD;Fc6)6
    name: widget_production
    primary:
        host: rhodeisland.example.com
        port: 5432
    replica:
        host: virginia.example.com
        port: 5432

rabbitmq:
    host: pennsylvania.example.com
    port: 5672
```

如果我们选择使用 YAML 字典，我们访问变量的方式也需要跟着改变，需要将

59

```
 {{ db_primary_host }}
```

变为

```
 {{ db.primary.host }}
```

如果你想更进一步地进行拆分，Ansible 还允许将 `group_vars/production` 定义为一个目录（而不是文件）。然后将多个包含变量定义的 YAML 文件存放在这个目录中。

例如，我们可以将数据库相关的变量放在一个文件中，把 Rabbit MQ 相关的变量放在另一文件中，如例 3-9 和例 3-10 所示：

```
例3-9 group_vars/production/db
db:
  user: widgetuser
  password: pFmMxxyD;Fc6)6
  name: widget_production
  primary:
    host: rhodeisland.example.com
    port: 5432
  replica:
    host: virginia.example.com
    port: 5432
```

```
例3-10 group_vars/production/rabbitmq
rabbitmq:
  host: pennsylvania.example.com
  port: 6379
```

我觉得将变量分割到太多的文件会带来管理复杂的问题，所以通常来说保持配置简单是更好的选择。

动态 inventory

到目前为止，我们已经在 `inventory` 文件中明确地定义了所有的主机。但是，你可能使用了 Ansible 以外的某个系统来记录你的主机。举例来说，如果你的主机运行在 Amazon EC2 上，那么 EC2 便会为你维护主机的信息，并且你可以通过 EC2 的 Web 接口、查询 API 或者 `awscli` 一类的命令行工具来随时获取这些信息，其他云计算服务商也有类似的接口；如果你是自己管理服务器，而且在使用类似 Cobbler 或者 Ubuntu MAAS 这样的自动化安装系统，那么自动化安装系统已经准备好记录了你的服务器的信息了；又或者，你也可以有一个酷炫的配置管理数据库（CMDBs），并由它管理所有的信息。
60

你肯定不希望手动将这些信息复制到你的主机文件里，因为这些文件中的信息最终很难与真正管理主机信息的外部系统相一致。利用 Ansible 支持的动态 `inventory` (*dynamic inventory*) 功能可以避免对这些信息进行复制。

如果 `inventory` 文件标记为可执行，那么 Ansible 会假设这是一个动态 `inventory` 脚本，并且会执行它，而不是读取它的内容。



想要将一个文件标记为可执行，使用 `chmod +x` 命令。例如：

```
$ chmod +x dynamic.py
```

动态 inventory 脚本的接口

一个动态 inventory 脚本必须支持如下两个命令行参数：

- `--host=<hostname>` 用来列出主机的详细信息
- `--list` 用来列出群组。

展示主机详细信息

Ansible 会按照如下形式调用 inventory 脚本来获取单台主机的详细信息：

```
$ ./dynamic.py --host=vagrant2
```

输出应该包含所有主机特定的变量，也包括行为参数，类似这样：

```
{ "ansible_ssh_host": "127.0.0.1", "ansible_ssh_port": 2200,  
  "ansible_ssh_user": "vagrant"}
```

输出是一个名字为变量名、值为变量值的 JSON 对象。

列出群组

动态 inventory 脚本需要能够列出所有的群组和每台主机的详细信息。例如，如果我们的脚本叫作 `dynamic.py`，Ansible 将按照如下方式调用它来列出所有的群组：

```
$ ./dynamic.py --list
```

输出应该类似这样：

```
{"production": ["delaware.example.com", "georgia.example.com",  
                 "maryland.example.com", "newhampshire.example.com",  
                 "newjersey.example.com", "newyork.example.com",  
                 "northcarolina.example.com", "pennsylvania.example.com",  
                 "rhodeisland.example.com", "virginia.example.com"],  
 "staging": ["ontario.example.com", "quebec.example.com"],  
 "vagrant": ["vagrant1", "vagrant2", "vagrant3"],  
 "lb": ["delaware.example.com"],  
 "web": ["georgia.example.com", "newhampshire.example.com",  
         "newjersey.example.com", "ontario.example.com", "vagrant1"],  
 "task": ["newyork.example.com", "northcarolina.example.com",  
          "ontario.example.com", "vagrant2"],  
 "rabbitmq": ["pennsylvania.example.com", "quebec.example.com", "vagrant3"],  
 "db": ["rhodeisland.example.com", "virginia.example.com", "vagrant3"]}
```

< 61

输出是一个 JSON 对象，该 JSON 对象的名字为群组名，值为由主机的名字组成的数组。

作为优化，`--list` 命令可以包含所有主机的变量值。这将会免去 Ansible 逐一调用 `--host` 去获取每个主机的变量的麻烦。

想要达到这个优化的效果，`--list` 命令应该返回包含每台主机的变量的键值对，该键值对的键名为 `_meta`，形式如下：

```
"_meta" :  
  { "hostvars" :  
      "vagrant1" : { "ansible_ssh_host": "127.0.0.1", "ansible_ssh_port": 2222,  
                     "ansible_ssh_user": "vagrant"},  
      "vagrant2": { "ansible_ssh_host": "127.0.0.1", "ansible_ssh_port": 2200,  
                     "ansible_ssh_user": "vagrant"},  
      ...  
  }
```

编写动态 inventory 脚本

使用 `vagrant status` 命令来查看哪些机器正处于运行状态是 Vagrant 的便利功能之一。假设我们的 Vagran file 如例 3-2 所示，如果我们执行 `vagrant status`，输出信息将如例 3-11 所示。

例3-11 vagrant status的输出

```
$ vagrant status  
Current machine states:  
  
vagrant1          running (virtualbox)  
vagrant2          running (virtualbox)  
vagrant3          running (virtualbox)  
  
62 ┶ This environment represents multiple VMs. The VMs are all listed  
       above with their current state. For more information about a specific  
       VM, run `vagrant status NAME`.
```

既然 Vagrant 已经替我们维护服务器信息了，我们就没有必要在 Ansible inventory 文件中再去罗列 Vagrant 虚拟机了。我们可以编写动态 inventory 脚本从 Vagrant 查询哪些虚拟机正在运行。

一旦我们为 Vagrant 创建相应的动态 inventory 脚本，即便我们通过变更 `Vagrantfile` 来改变 Vagrant 虚拟机的运行数量，我们也不再需要去编辑 Ansible inventory 文件了。

现在让我们从头到尾做一个从 Vagrant^{注2} 获取主机详细信息的动态 inventory 脚本的例子。

注 2：没错，Ansible 已经提供了支持 Vagrant 的动态 inventory 脚本，但是将它作为练习从头到尾做一次有利于理解动态 inventory 脚本。

我们的动态 inventory 脚本需要调用 `vagrant status` 命令。例 3-11 所示是其输出，该输出是为了便于用户阅读，而不是程序解析。我们可以利用 `--machine-readable` 参数，以一种便于程序解析的格式来列出运行中的主机，就像这样：

```
$ vagrant status --machine-readable
```

输出类似下面这样：

```
1410577818,vagrant1,provider-name,virtualbox
1410577818,vagrant1,state,running
1410577818,vagrant1,state-human-short,running
1410577818,vagrant1,state-human-long,The VM is running. To stop this
VM%!(VAGRANT_COMMA) you can run `vagrant halt` to\shut it down
forcefully%!(VAGRANT_COMMA)or you can run `vagrant suspend` to simply\
nsuspend the virtual machine. In either case%!(VAGRANT_COMMA) to restart it
again%!(VAGRANT_COMMA)\nsimply run`vagrant up`.
1410577818,vagrant2,provider-name,virtualbox
1410577818,vagrant2,state,running
1410577818,vagrant2,state-human-short,running
1410577818,vagrant2,state-human-long,The VM is running. To stop this
VM%!(VAGRANT_COMMA) you can run `vagrant halt` to\shut it down
forcefully%!(VAGRANT_COMMA)or you can run `vagrant suspend` to simply\
nsuspend the virtual machine. In either case%!(VAGRANT_COMMA) to restart it
again%!(VAGRANT_COMMA)\nsimply run`vagrant up`.
1410577818,vagrant3,provider-name,virtualbox
1410577818,vagrant3,state,running
1410577818,vagrant3,state-human-short,running
1410577818,vagrant3,state-human-long,The VM is running. To stop this
VM%!(VAGRANT_COMMA) you can run `vagrant halt` to\shut it down
forcefully%!(VAGRANT_COMMA)
or you can run `vagrant suspend` to simply\nsuspend the virtual machine. In
either case%!(VAGRANT_COMMA) to restart it again%!(VAGRANT_COMMA)\nsimply
run `vagrant up`.
```

63

如果想要获取特定 Vagrant 虚拟机的详细信息，例如 `vagrant2`，我们可以运行下面的程序：

```
$ vagrant ssh-config vagrant2
```

输出如下：

```
Host vagrant2
HostName 127.0.0.1
User vagrant
Port 2200
UserKnownHostsFile /dev/null
StrictHostKeyChecking no
```

```
PasswordAuthentication no
IdentityFile /Users/lorinhochstein/.vagrant.d/insecure_private_key
IdentitiesOnly yes
LogLevel FATAL
```

我们的动态 inventory 脚本将会调用这些命令，解析输出信息，最后输出恰当的 JSON。我们可以使用 Paramiko 库来解析 vagrant ssh-config 的输出。下面的 Python 交互式会话演示了如何使用 Paramiko 库：

```
>>> import subprocess
>>> import paramiko
>>> cmd = "vagrant ssh-config vagrant2"
>>> p = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE)
>>> config = paramiko.SSHConfig()
>>> config.parse(p.stdout)
>>> config.lookup("vagrant2")
{'identityfile': ['/Users/lorinhochstein/.vagrant.d/insecure_private_key'],
 'loglevel': 'FATAL', 'hostname': '127.0.0.1', 'passwordauthentication': 'no',
 'identitiesonly': 'yes', 'userknownhostsfile': '/dev/null', 'user':
 'vagrant',
 'stricthostkeychecking': 'no', 'port': '2200'}
```



在使用这个脚本之前你需要预先安装 Python 的 Paramiko 库。你可以使用如下命令利用 pip 来安装它：

```
$ sudo pip install paramiko
```

例 3-12 列出了完整的 vagrant.py 脚本

例3-12 vagrant.py

```
#!/usr/bin/env python
# 改编自 Mark Mandel 实现的版本
# https://github.com/ansible/ansible/blob/devel/plugins/inventory/vagrant.py
# License: GNU General Public License, Version 3 <http://www.gnu.org/licenses/>
import argparse
import json
import paramiko
import subprocess
import sys

def parse_args():
    parser = argparse.ArgumentParser(description="Vagrant inventory script")
    group = parser.add_mutually_exclusive_group(required=True)
```

```

group.add_argument('--list', action='store_true')
group.add_argument('--host')
return parser.parse_args()

def list_running_hosts():
    cmd = "vagrant status --machine-readable"
    status = subprocess.check_output(cmd.split()).rstrip()
    hosts = []
    for line in status.split('\n'):
        _, host, key, value = line.split(',')
        if key == 'state' and value == 'running':
            hosts.append(host)
    return hosts

def get_host_details(host):
    cmd = "vagrant ssh-config {}".format(host)
    p = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE)
    config = paramiko.SSHConfig()
    config.parse(p.stdout)
    c = config.lookup(host)
    return {'ansible_ssh_host': c['hostname'],
            'ansible_ssh_port': c['port'],
            'ansible_ssh_user': c['user'],
            'ansible_ssh_private_key_file': c['identityfile'][0]}

def main():
    args = parse_args()
    if args.list:
        hosts = list_running_hosts()
        json.dump({'vagrant': hosts}, sys.stdout)
    else:
        details = get_host_details(args.host)
        json.dump(details, sys.stdout)

if __name__ == '__main__':
    main()

```

预装的 inventory 脚本

◀ 65

Ansible 随附了几个动态 inventory 脚本，你可以按需选用。我并不会去费力寻找软件包管理工具将这些文件安装到哪去了，而是会直接从 GitHub 上获取我需要的那个。你也可以连接到 Ansible 的 GitHub 仓库 (<https://github.com/ansible/ansible>) 上，然后访问 `plugins/inventory` 目录来获取这些脚本。

很多 inventory 脚本有一个配套的配置文件。第 12 章将会详细讲述 Amazon EC2

inventory 脚本。

将 inventory 分割到多个文件中

如果你想要同时使用常规 inventory 文件和动态 inventory 脚本（或者说是静态与动态 inventory 文件的任意组合），只要将所有这些文件都放到同一个目录，并配置 Ansible，让它使用这个目录作为 inventory 即可。要做到这一点，可以通过 *ansible.cfg* 文件中的 `hostfile` 参数，也可以在命令行中使用 `-i` 参数。Ansible 将会处理所有的文件并将结果合并为一个完整的 inventory。

举例来说，假设我们的目录结构就像这样：*inventory/hosts* 和 *inventory/vagrant.py*。

我们的 *ansible.cfg* 文件将包含下面两行：

```
[defaults]
hostfile = inventory
```

使用 add_host 和 group_by 在运行时添加条目

Ansible 也可以让你在执行 playbook 的时候向 inventory 中添加主机和群组。

add_host

`add_host` 模块可以向 inventory 中添加主机。在使用 Ansible 来初始化 IaaS 云中的新虚拟机实例的时候，这个模块非常好用。

如果我已经使用了动态 inventory，为什么还需要 add_host

即便你在使用动态 inventory 脚本，在同一个 playbook 中创建新的虚拟机实例并配置这些实例的时候，`add_host` 模块仍然是非常有用的。

如果在 playbook 执行的时候，一个新的主机被创建了，动态 inventory 脚本是无法把这个新主机追加进来的。这是因为动态 inventory 脚本是在 playbook 开始的时候被执行的，所以如果有新的主机在 playbook 执行时被添加，Ansible 无法感知到。

我们将在第 12 章讨论一个使用 `add_host` 模块的云计算案例。

用如下方式调用该模块：

```
add_host name=hostname groups=web,staging myvar=myval
```

指定群组列表和额外的变量是可选项。

这里有一个使用 add_host 命令的实践案例，创建一个新的 vagrant 主机并配置它：

```
- name: Provision a vagrant machine
  hosts: localhost
  vars:
    box: trusty64
  tasks:
    - name: create a Vagrantfile
      command: vagrant init {{ box }} creates=Vagrantfile

    - name: Bring up a vagrant server
      command: vagrant up

    - name: add the Vagrant hosts to the inventory
      add_host: >
        name=vagrant
        ansible_ssh_host=127.0.0.1
        ansible_ssh_port=2222
        ansible_ssh_user=vagrant
        ansible_ssh_private_key_file=/Users/lorinhochstein/.vagrant.d/
        insecure_private_key

- name: Do something to the vagrant machine
  hosts: vagrant
  sudo: yes
  tasks:
    # 真正需要执行的 task 列在这里
    - ...
```



add_host 模块添加主机仅在本次 playbook 执行过程中生效。它并不会修改你的 inventory 文件。

67

当我使用 playbook 进行虚拟化环境初始化的时候，我喜欢将整个工作分成两个 play。第一个 play 对本机运行并创建主机，第二个 play 配置这台主机。

需要注意的是，我们在这个任务中使用了 creates=Vagrantfile 参数：

```
-name: create a Vagrantfile
  command: vagrant init {{ box }} creates=Vagrantfile
```

这会告诉 Ansible 如果 *Vagrantfile* 存在的话，则说明已经处于正确的状态，而且不需要

再次运行这个命令。这是在使用 command 模块时实现幂等性的一种方法，它可以确保（潜在非幂等性的）命令只被运行一次。

group_by

Ansible 还允许你在 playbook 执行的时候使用 *group_by* 模块来创建新群组。它让你可以基于已经为每台主机自动设定好的变量值来创建群组，Ansible 将这些变量叫作 fact^{注3}。

如果 Ansible 的 fact gathering 启用，那么 Ansible 会自动为主机设置一组变量。例如，*ansible_machine* 变量在 32 位 x86 机器上的值是 *i386*，而在 64 位 x86 机器上的值则为 *x86_64*。如果 Ansible 同时管理这两种主机，我们就可以在 task 中创建 *i386* 和 *x86_64* 群组。

亦或者我们想要按照 Linux 发行版（比如 Ubuntu、CentOS）来划分群组，我们就可以使用 *ansible_distribution* 这个 fact。

```
- name: create groups based on Linux distribution
  group_by: key={{ ansible_distribution }}
```

在例 3-13 中，我们使用 *group_by* 分别为我们的 Ubuntu 主机和 CentOS 主机创建单独的群组。然后，我们使用 *apt* 模块向 Ubuntu 中安装软件包，同时使用 *yum* 模块向 CentOS 中安装软件包。

例3-13 基于Linux发行版创建ad-hoc群组

```
68 >
- name: group hosts by distribution
  hosts: myhosts
  gather_facts: True
  tasks:
    - name: create groups based on distro
      group_by: key={{ ansible_distribution }}

- name: do something to Ubuntu hosts
  hosts: Ubuntu
  tasks:
    - name: install htop
      apt: name=htop
    # ...

- name: do something else to CentOS hosts
  hosts: CentOS
  tasks:
    - name: install htop
      yum: name=htop
    # ...
```

注 3： 我们将会在第 4 章详细讨论 fact。

尽管在 Ansible 中使用 `group_by` 是一种实现条件操作的方法，但我并没有发现它有什么太大的用处。在第 6 章，我们将会看到如何使用 `when` 语句基于变量执行不同操作的范例演示。

到这里我们就结束了关于 Ansible 中 `inventory` 的全部讨论了。在下一章，我们将会讨论如何使用变量。关于 *ControlPersist*（也被称作 SSH multiplexing）的更多细节可以查阅第 9 章。

变量与fact

Ansible 并不是一个十分成熟的编程语言，但是它有若干编程语言的特性。而其中最重要的特性就是变量替代。在这一章，我们将详细讨论 Ansible 对于变量的支持，其中包括一种被 Ansible 称为 fact 的特殊类型变量。

在 playbook 中定义变量

最简单的定义变量的方法是在你的 playbook 中添加 `vars` 区段，并在该区段中列出变量名与变量值。回忆一下在例 2-8 中，我们使用这个方法定义了一些配置相关的变量，如下所示：

```
vars:  
  key_file: /etc/nginx/ssl/nginx.key  
  cert_file: /etc/nginx/ssl/nginx.crt  
  conf_file: /etc/nginx/sites-available/default  
  server_name: localhost
```

Ansible 也允许你通过定义名为 `vars_files` 的区段把变量放到一个或者多个文件中。让我们把上面例子中的变量从 playbook 中挪出来，放到名为 `nginx.yml` 的文件中。我们需要将 `vars` 区段替换成 `vars_files`，就像这样：

```
vars_files:  
  - nginx.yml
```

`nginx.yml` 文件的内容将如例 4-1 所示。

例4-1 nginx.yml

```
key_file: /etc/nginx/ssl/nginx.key  
cert_file: /etc/nginx/ssl/nginx.crt
```

70 conf_file: /etc/nginx/sites-available/default
server_name: localhost

我们将在第 6 章中看到一个 `vars_files` 的实践案例，在那个案例中我们将使用这个特性将具有敏感信息的变量隔离出来。

就像我们在第 3 章中讨论过的，Ansible 也允许你定义与主机或者群组相关的变量，这些变量可以定义在 `inventory` 文件中，也可以定义在与 `inventory` 文件放在一起的独立文件中。

查看变量的值

为了便于调试，Ansible 通常可以很方便地查看变量的值。在第 2 章中，我们曾经看到如何使用 `debug` 模块来打印任意信息。我们也可以使用它来输出变量的值。用法就像这样：

```
- debug: var=myvarname
```

在本章我们将多次使用这种形式的 `debug` 模块。

注册变量

你会发现你经常需要基于 task 执行的结果来设置变量的值。想要实现这个操作，我们可以在调用模块的时候使用 `register` 语句来创建注册变量。例 4-2 示范了如何将 `whoami` 命令的输出捕获到名为 `login` 的变量中。

例4-2 将命令的输出捕获到变量中

```
- name: capture output of whoami command  
  command: whoami  
  register: login
```

为了以后使用 `login` 变量，我们需要了解该变量值的类型。使用 `register` 关键字设置的变量肯定是字典类型，但是字典的具体 key 是不同的，这取决于所调用的模块。

不幸的是，Ansible 官方模块文档并没有包含每个模块返回值相关的信息。模块文档一般都包含使用 `register` 关键字的范例，这些范例反倒是更有帮助。因为我发现确认模块返回形式的最简单的方法就是设置一个 `register` 变量，然后使用 `debug` 模块输出这个变量。

我们来运行如例 4-3 所示的 playbook。

71 例4-3 whomai.yml

```
- name: show return value of command module  
  hosts: server1  
  tasks:
```

```
- name: capture output of id command
  command: id -un
  register: login
- debug: var=login
```

debug 模块的输出如下：

```
TASK: [debug var=login] ****
ok: [server1] => {
  "login": {
    "changed": true, ❶
    "cmd": [ ❷
      "id",
      "-un"
    ],
    "delta": "0:00:00.002180",
    "end": "2015-01-11 15:57:19.193699",
    "invocation": {
      "module_args": "id -un",
      "module_name": "command"
    },
    "rc": 0, ❸
    "start": "2015-01-11 15:57:19.191519",
    "stderr": "", ❹
    "stdout": "vagrant", ❺
    "stdout_lines": [ ❻
      "vagrant"
    ],
    "warnings": []
  }
}
```

❶ 输出中名为 changed 的 key 出现在所有 Ansible 模块的返回值中，并且 Ansible 用它来判断是否发生了状态改变。对于 command 和 shell 模块来说，总是会将 changed 的 value 设置为 true，除非使用 changed_when 语句覆盖了 changed 的值。我们将在第 7 章详细讨论 changed_when 语句。

❷ 为名 cmd 的 key 对应的 value 是用字符串列表形式描述的被调用的命令。

❸ 名为 rc 的 key 对应的 value 是返回值。如果这个值非零，Ansible 将会认为任务执行失败。

❹ 名为 stderr 的 key 对应的 value 是输出到标准错误的文本组成的单一字符串。

❺ 名为 stdout 的 key 对应的 value 是输出到标准输出的文本组成的单一字符串。

- ⑥ 名为 `stdout_lines` 的 key 对应的 value 是按换行符分割的输出到标准输出的文本。这是一个由输出中的每一行作为元素组成的列表。

如果你和 `command` 模块一起使用 `register` 语句，你很可能想要访问 `stdout` 的内容，就像例 4-4 所示。

例4-4 在task中获取命令的输出

```
- name: capture output of id command
  command: id -un
  register: login
- debug: msg="Logged in as user {{ login.stdout }}"
```

有时候，一个失败的 task 的输出信息也可能是有用的。可是如果 task 失败了，Ansible 就会停止对发生失败的主机继续执行后续 task。我们可以使用 `ignore_errors` 语句来解决这个问题。如例 4-5 所示，使用了 `ignore_errors` 语句后，Ansible 就不会在发生错误的时候终止运行了。

例4-5 忽略模块返回的错误

```
- name: Run myprog
  command: /opt/myprog
  register: result
  ignore_errors: True
- debug: var=result
```

Shell 模块输出的结果与 `command` 模块一致，但是其他模块的输出会包含不一样的 key。例 4-6 所示为安装一个以前不存在的软件包时，`apt` 模块的输出。

例4-6 安装新软件包时apt模块的输出

```
ok: [server1] => {
  "result": {
    "changed": true,
    "invocation": {
      "module_args": "name=nginx",
      "module_name": "apt"
    },
    "stderr": "",
    "stdout": "Reading package lists...\nBuilding dependency tree...",
    "stdout_lines": [
      "Reading package lists...",
      "Building dependency tree...",
      "Reading state information...",
      "Preparing to unpack .../nginx-common_1.4.6-1ubuntu3.1_all.deb ...",
      ...
      "Setting up nginx-core (1.4.6-1ubuntu3.1) ...",
    ]
  }
}
```

```
"Setting up nginx (1.4.6-1ubuntu3.1) ...",
"Processing triggers for libc-bin (2.19-0ubuntu6.3) ..."
]
}
}
```

访问变量中字典的 key

如果变量中包含字典，那么你可以使用点号（.）或者中括号（[]）来访问字典的 key。例 4-4 中就有一个变量引用使用的是点号：

```
{{ login.stdout }}
```

我们可以将其替换为中括号：

```
{{ login['stdout'] }}
```

该规则在多重引用时也生效，所以下面所有形式都是等价的：

```
ansible_eth1['ipv4']['address']
ansible_eth1['ipv4'].address
ansible_eth1.ipv4['address']
ansible_eth1.ipv4.address
```

一般情况下我更倾向于使用点号，除非 key 中包含不允许出现在变量名的字符，例如 key 中包括点、空格或者连字符。

Ansible 使用 Jinja2 实现变量解引。所以本话题更细节的内容可以查阅 Jinja2 文档中的变量章节 (<http://jinja.pocoo.org/docs/dev/templates/#variables>)。

例 4-7 列出了在软件包已经安装在目标主机上的情况下，apt 模块的输出。

例 4-7 已经安装软件包的情况下，apt 模块的输出

```
ok:[server1] => {
    "result": {
        "changed": false,
        "invocation": {
            "module_args": "name=nginx",
            "module_name": "apt"
        }
    }
}
```

需要注意的是，只有当软件包没有预先安装的情况下，输出中才有 `stdout`、`stderr`、`stdout_lines` 为 key 的三个字典。74



如果你的 playbook 使用了注册变量，那么无论模块是否改变了主机的状态，请确保你都是了解变量的内容的。否则，当你的 playbook 尝试访问注册变量中不存在的 key 时可能会导致失败。

fact

就像我们曾经看到过的，当 Ansible 运行 playbook 的时候，在开始执行第一个 task 之前会发生如下事件：

```
GATHERING FACTS ****
ok: [servername]
```

当 Ansible 采集 fact 的时候，它会连接到主机收集各种详细信息：CPU 架构、操作系统、IP 地址、内存信息、磁盘信息等。这些信息保存在被称作 fact 的变量中。fact 与其他变量的行为一模一样。

下面是能够打印每台服务器操作系统信息的简单 playbook：

```
- name: print out operating system
hosts: all
gather_facts: True
tasks:
- debug: var=ansible_distribution
```

在运行 Ubuntu 和 CentOS 的服务器上，它的输出如下所示：

```
PLAY [print out operating system] ****
GATHERING FACTS ****
ok: [server1]
ok: [server2]

TASK: [debug var=ansible_distribution] ****
ok: [server1] => {
    "ansible_distribution": "Ubuntu"
}
ok: [server2] => {
    "ansible_distribution": "CentOS"
}
PLAY RECAP ****
server1                  : ok=2      changed=0      unreachable=0      failed=0
server2                  : ok=2      changed=0      unreachable=0      failed=0
```

75 ➤ 你可以查阅 Ansible 官方文档 (http://docs.ansible.com/playbooks_variables.html#)

information-discovered-from-systems-facts) 来获取列有一些可用的 fact 的列表。我也在 GitHub 上维护了一个更全面的 fact 列表 (<https://github.com/lorin/ansible-quickref/blob/master/facts.rst>)。

查看与某台服务器关联的所有 fact

Ansible 使用一个名为 setup 的特殊模块来实现 fact 的收集。你不需要在你的 playbook 中去调用这个模块，因为 Ansible 会在采集 fact 时自动调用。不过，如果你像这样使用 ansible 命令行工具手动调用它：

```
$ ansible server1 -m setup
```

那么，Ansible 将会输出所有的 fact，如例 4-8 所示：

例4-8 setup模块的输出

```
server1 | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15",
            "192.168.4.10"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fefe:1e4d",
            "fe80::a00:27ff:fe67:bbf3"
        ],
    }
}
```

(下面略了很多的 fact)

注意该模块的返回值是一个字典，字典的 key 是 `ansible_facts`，而它的 value 是一个由实际 fact 的名字与值组成的字典。

查看 fact 子集

由于 Ansible 收集非常多的 fact，setup 模块支持 `filter` 参数，以帮助你通过指定 shell 通配符 (glob)^{注1} 来针对 fact 名进行过滤，例如：

```
$ ansible web -m setup -a 'filter=ansible_eth*'
```

输出如下所示：

```
web | success >> {
    "ansible_facts": {
        "ansible_eth0": {
```

注1：glob 就是 shell 用于对文件进行模式匹配的通配符（例如：`*.txt`）。

```
    "active": true,
    "device": "eth0",
    "ipv4": [
        "address": "10.0.2.15",
        "netmask": "255.255.255.0",
        "network": "10.0.2.0"
    ],
    "ipv6": [
        {
            "address": "fe80::a00:27ff:fefe:1e4d",
            "prefix": "64",
            "scope": "link"
        }
    ],
    "macaddress": "08:00:27:fe:le:4d",
    "module": "e1000",
    "mtu": 1500,
    "promisc": false,
    "type": "ether"
},
"ansible_eth1": {
    "active": true,
    "device": "eth1",
    "ipv4": [
        "address": "192.168.33.10",
        "netmask": "255.255.255.0",
        "network": "192.168.33.0"
    ],
    "ipv6": [
        {
            "address": "fe80::a00:27ff:fe23:ae8e",
            "prefix": "64",
            "scope": "link"
        }
    ],
    "macaddress": "08:00:27:23:ae:8e",
    "module": "e1000",
    "mtu": 1500,
    "promisc": false,
    "type": "ether"
}
},
"changed": false
}
```

任何模块都可以返回 fact

如果你仔细观察例 4-8 所示的输出，你会看到该输出是一个 key 为 `ansible_facts` 的字典。在返回值中使用 `ansible_facts` 字典是 Ansible 的特定语法。如果模块返回一个字典且包含名为 `ansible_facts` 的 key，那么 Ansible 将会根据对应的 value 创建相应的变量，并分配给相对应的主机。

对于返回 fact 的模块，并不需要使用注册变量，因为 Ansible 会自动为你创建。例如，下面的任务将会使用 `ec2_facts` 模块来获取 Amazon EC2^{注2} 中服务器的 fact，并输出实例 id。◀77

```
- name: get ec2 facts
  ec2_facts:
    - debug: var=ansible_ec2_instance_id
```

输出如下：

```
TASK: [debug var=ansible_ec2_instance_id] ****
ok: [myserver] => {
    "ansible_ec2_instance_id": "i-a3a2f866"
}
```

注意在这个例子中，我们在调用 `ec2_facts` 时并不需要使用 `register` 关键字，因为返回值是 fact。Ansible 所附带的模块中有一些是返回 facts 的。我们将会在第 13 章看到另外一个有此行为的模块，`docker` 模块。

本地 fact

Ansible 还提供了另外一种为某个主机设定 fact 的机制。你可以将一个或多个文件放置在目标主机的 `/etc/ansible/facts.d` 目录下。如果该目录中的文件是以下形式的，Ansible 会自动识别：

- `.ini` 格式。
- JSON 格式。
- 可以不加参数形式执行，并在标准输出中输出 JSON 的可执行文件。

以这种形式加载的 fact 是 key 为 `ansible_local` 的特殊变量。

我们来看一下实例。例 4-9 是一个 `.ini` 格式的 fact 文件。

注 2： 我们将在第 12 章更详细地讨论 Amazon EC2。

例4-9 /etc/ansible/facts.d/example.fact

```
[book]
title=Ansible: Up and Running
author=Lorin Hochstein
publisher=O'Reilly Media
```

如果我们将这个文件复制到远程主机的 */etc/ansible/facts.d/example.fact*, 我们就可以在 playbook 中访问 `ansible_local` 变量的内容：

```
78    - name: print ansible_local
      debug: var=ansible_local
    - name: print book title
      debug: msg="The title of the book is {{ ansible_local.example.book.title }}"
```

这些 task 的输出如下所示：

```
TASK: [print ansible_local] ****
ok: [server1] => {
    "ansible_local": {
        "example": {
            "book": {
                "author": "Lorin Hochstein",
                "publisher": "O'Reilly Media",
                "title": "Ansible: Up and Running"
            }
        }
    }
}

TASK: [print book title] ****
ok: [server1] => {
    "msg": "The title of the book is Ansible: Up and Running"
}
```

不过, `ansible_local` 变量值的结构需要格外注意。因为 fact 文件的名字是 *example.fact*, 所以 `ansible_local` 变量是一个字典, 且包含了一个名为 “example” 的 key。

使用 `set_fact` 定义新变量

Ansible 还允许你使用 `set_fact` 模块在 task 中设置 fact (实际上与定义一个新变量是一样的)。我总是喜欢在 `register` 关键字后立即使用 `set_fact`, 这样能让变量引用变得更简单。例 4-10 示范了如何利用 `set_fact` 使变量可以用 `snap` 引用, 而不需要使用更烦琐的 `snapshot_result.stdout` 来引用。

例4-10 使用set_fact来简化变量引用

```
- name: get snapshot id
  shell: >
    aws ec2 describe-snapshots --filters
      Name>tag:Name,Values=my-snapshot
    | jq --raw-output ".Snapshots[].SnapshotId"
  register: snap_result
- set_fact: snap={{ snap_result.stdout }}

- name: delete old snapshot
  command: aws ec2 delete-snapshot --snapshot-id "{{ snap }}"
```

内置变量

79

Ansible 定义了一些在 playbook 中永远可以访问的变量，如表 4-1 所示。

表4-1 内置变量

参数	描述
hostvars	字典，key 为 Ansible 主机的名字，value 为所有变量名与相应变量值映射组成的字典
inventory_hostname	当前主机被 Ansible 识别的名字
group_names	列表，由当前主机所属的所有群组组成
groups	字典，key 为 Ansible 群组名，value 为群组成员的主机名所组成的列表。包括 all 分组和 ungrouped 分组：{"all": [...], "web": [...], "ungrouped": [...]}
play_hosts	列表，成员是当前 play 涉及的主机的 inventory 主机名
ansible_version	字典，由 Ansible 版本信息组成：{"full": "1.8.2", "major": 1, "minor": 8, "revision": 2, "string": "1.8.2"}

其中 hostvars、inventory_hostname 和 groups 变量最为常用，下面就对它们详细展开讨论。

hostvars

在 Ansible 中，变量的作用域按照主机划分。只有针对特定主机讨论变量的值才有意义。

由于 Ansible 允许你针对一组主机来定义变量，所以变量与特定主机相关联的理念可能会让人困惑。这里还是用实例来说明更为形象：如果你在一个 play 中的 vars 区段定义了一个变量，那么就是在这个 play 中对一组主机定义了变量，但 Ansible 的实际做法其

实是对这个群组中的每一个主机创建一个变量的副本。

有时候，在某一个主机上运行的 task 可能会需要在另一台主机上定义的变量值。设想一下配置 Web 服务的场景：你为 Web 服务器创建的配置文件可能会包含数据库服务器的 `eth1` 网卡上的 IP 地址，且你无法预先知道这个 IP 地址。这个 IP 地址就可以使用数据库服务器的 `ansible_eth1.ipv4.address` 这个 fact。

80 ➤ 这类问题的解决方案就是使用 `hostvars` 变量。这个字典包含了在所有主机上定义的所有变量，并以 Ansible 识别的主机名作为 key。如果 Ansible 还未对主机采集 fact，那么除非启用了 fact 缓存^{注3}，否则就不能使用 `hostvars` 变量访问 fact。

还是回到我们的例子中，如果我们的数据库服务器是 `db.example.com`，那么我们可以在配置文件的模板中使用如下引用：

```
{{ hostvars['db.example.com'].ansible_eth1.ipv4.address }}
```

这个引用会返回 `db.example.com` 主机的 `ansible_eth1.ipv4.address` 的值。

inventory_hostname

`inventory_hostname` 是 Ansible 所识别的当前主机的主机名。如果你定义过别名，那么这里就是那个别名。例如，如果你的 inventory 包含如下一行：

```
server1 ansible_ssh_host=192.168.4.10
```

那么 `inventory_hostname` 将会是 `server1`。

利用 `hostvars` 和 `inventory_hostname` 变量，你可以输出与当前主机相关联的所有变量：

```
- debug: var=hostvars[inventory_hostname]
```

groups

当你想要访问一组主机的变量时，`groups` 变量会很有用。比如我们正在配置一台负载均衡服务器，我们的配置文件肯定需要 web 群组中所有服务器的 IP 地址。我们的配置文件将会包含如下片段：

```
backend web-backend
{% for host in groups.web %}
    server {{ host.inventory_hostname }} {{ host.ansible_default_ipv4.address }}
}{{ :80
% endfor %}
```

注 3：可以在第 9 章查看关于 fact 缓存的更多信息。

最终生成的文件如下：

```
backend web-backend
  server georgia.example.com 203.0.113.15:80
  server newhampshire.example.com 203.0.113.25:80
  server newjersey.example.com 203.0.113.38:80
```

在命令行设置变量

81

通过向 `ansible-playbook` 传入 `-e var=value` 参数设置的变量拥有最高优先级，也就意味着你可以通过这种方式覆盖已经定义的变量。例 4-11 演示了如何将名为 `token` 的变量的值无条件地设置为 12345。

例4-11 从命令行设置变量

```
$ ansible-playbook example.yml -e token=12345
```

如果你希望像带有命令行参数的 shell 脚本那样使用 `playbook`，可以使用 `ansible-playbook -e var=value` 方法。`-e` 参数允许你像传递参数一样传递变量。

例 4-12 是一个非常简单的 playbook，它会输出由变量定义的信息。

例4-12 greet.yml

```
- name: pass a message on the command line
hosts: localhost
vars:
  greeting: "you didn't specify a message"
tasks:
  - name: output a message
    debug: msg="{{ greeting }}"
```

如果按照如下方法调用：

```
$ ansible-playbook greet.yml -e greeting=hiya
```

输出将会是这样的：

```
PLAY [pass a message on the command line] ****
TASK: [output a message] ****
ok: [localhost] => {
    "msg": "hiya"
}

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0
```

如果你希望在变量中出现空格，那么你需要使用引号：

```
$ ansible-playbook greet.yml -e 'greeting="hi there"'
```

使用单引号将整个参数引起起来，像这样：'greeting="hi there"'，shell 就会将其解释为单一参数传给 Ansible。使用双引号引起变量的值，像这样："hi there"，Ansible 将会按单一字符串处理双引号内的信息。

Ansible 还允许你在命令行传入 @filename.yml 作为 -e 的参数，这种方法将会使用包含变量的文件，而不是在命令行直接传递变量。例如，如果我们有例 4-13 所示的文件：

例 4-13 greetvars.yml

```
greeting: hiya
```

那么我们可以像如下所示将这个文件在命令行传入：

```
$ ansible-playbook greet.yml -e @greetvars.yml
```

优先级

我们已经介绍了许多不同的定义变量的方法，那么就可能出现这样的情况：对某一主机的相同变量重复多次定义，且设定了不同的值。你需要尽可能地规避这种现象。但如果无法避免的话，一定要牢记 Ansible 的变量定义优先级规则。如果相同的变量被定义了多次，优先级会决定哪个值将最终被赋予该变量。

基本的优先级规则是：

1. (最高优先级) `ansible-playbook -e var=value`。
2. 这个优先级列表中没有提到的其他方法。
3. 通过 inventory 文件或者 YAML 文件定义的主机变量或群组变量。
4. Fact。
5. 在 role^{注4} 的 `defaults/main.yml` 文件中。

在本章中，我们讨论了许多不同的定义与访问变量和 fact 的方法。在下一章中，我们将聚焦于一个部署应用的现实例子。

注 4： 我们将在第 8 章讨论 role。

初识Mezzanine：我们的测试应用

我们在第 2 章讨论了 playbook 的基本编写方法。但是真实世界总是比编程语言教程中的介绍章节要复杂得多。所以，我们将会从头到尾完成一个完整的实例，在这个实例中，我们将部署一个具有一定复杂度的应用。

我们的实例应用是一个开源的内容管理系统（CMS），名字叫作 Mezzanine (<http://mezzanine.jupo.org>)，从本质上讲它有点像 WordPress。Mezzanine 是基于 Django 开发的。Django 是一个免费软件，一个面向 Web 应用的 Python 框架。

为什么向生产环境部署软件是一件复杂的事

在正式开始部署之前，我们先讨论一下在你的笔记本上以开发模式运行软件与在生产环境运行软件有什么区别。

Mezzanine 是一个很好的例子。在开发模式运行它比真正去部署它要容易得多。例 5-1 列举了在你的笔记本电脑上运行 Mezzanine 的所有步骤^{注1}。

例 5-1 以开发模式运行 Mezzanine

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install mezzanine
$ mezzanine-project myproject
$ cd myproject
$ python manage.py createdb
$ python manage.py runserver
```

它将会提示你回答几个问题。我对于每个 yes/no 问题都回答 “yes”，并且对于所有拥有

注 1：按照这个步骤操作将会向 virtualenv 中安装 Python 软件包。我们将在第 99 页中的“将 Mezzanine 和其他软件包安装到 virtualenv”一节讨论 virtualenv。

默认选项的问题都使用默认选项。我的整个交互界面如下所示：

```
You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'lorinhochstein'):
Email address: lorian@ansiblebook.com
Password:
Password (again):
Superuser created successfully.

A site record is required.
Please enter the domain and optional port in the format 'domain:port'.
For example 'localhost:8000' or 'www.example.com'.
Hit enter to use the default (127.0.0.1:8000):

Creating default site record: 127.0.0.1:8000 ...

Installed 2 object(s) from 1 fixture(s)

Would you like to install some initial demo pages?
Eg: About us, Contact form, Gallery. (yes/no): yes
```

最后，你会在终端上看到如下输出：

```
.....
      _d^^^^^b_
     .d'           ``b.
     .p'             `q.
     .d'             `b.
     .d'           `b.    * Mezzanine 3.1.10
     ::            ::    * Django 1.6.8
     ::            ::    * Python 2.7.6
     ::            ::    * SQLite 3.8.5
     `p.          .q'    * Darwin 14.0.0
     `p.          .q'
     `b.          .d'
     `q..         ..p'
     ^q.....p^
     .....

Validating models...

0 errors found
December 01, 2014 - 02:54:40
Django version 1.6.8, using settings 'mezzanine-example.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

如果你使用浏览器访问 `http://127.0.0.1:8000`，你会看到如图 5-1 所示的网页。

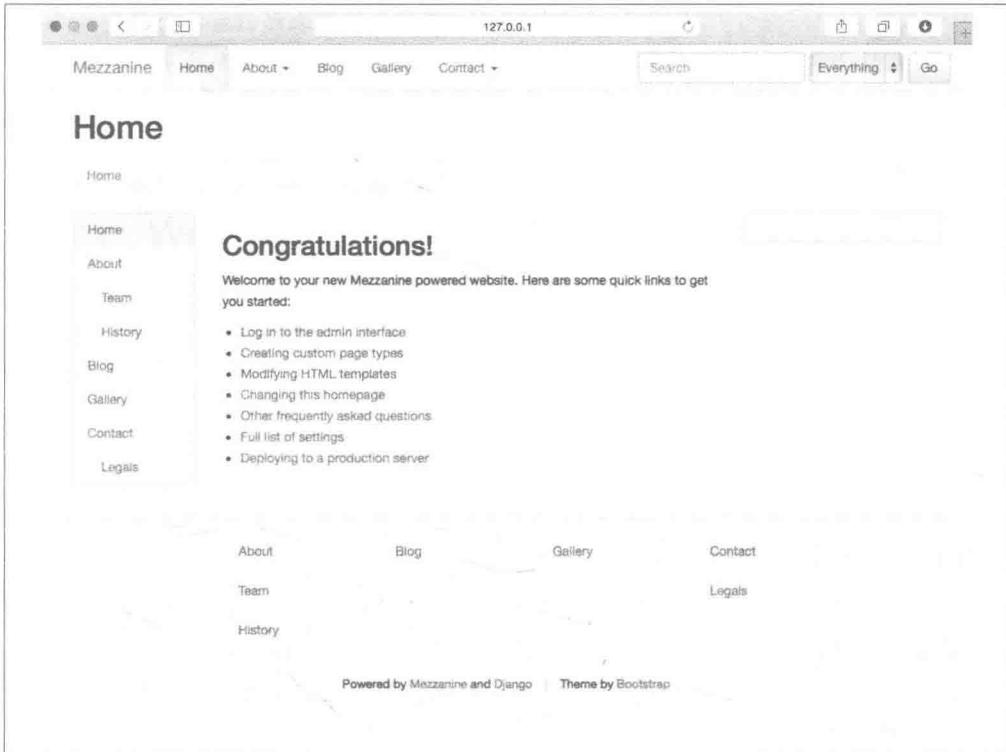


图5-1 全新安装后的Mezzanine

但是，将这个应用部署到生产环境可就是另一回事了。当你运行 `mezzanine-project` 命令的时候，Mezzanine 会在 `myproject/fabfile.py` 生成一个 Fabric (<http://www.fabfile.org/>) 部署脚本，你可以使用这个脚本将你的项目部署到生产服务器上。（Fabric 是一个基于 Python 的工具，它可以通过 SSH 自动运行任务。）这个脚本有 500 多行，而且这还不算部署过程中需要的配置文件。为什么向生产环境部署软件会如此复杂？我倒是很高兴你问这个问题。

在开发环境中，Mezzanine 进行了如下简单化操作（见图 5-2）：

- 系统将使用 SQLite 作为后端数据库，如果数据库不存在的话就会创建它们。
- 开发版 HTTP 服务器既要处理静态内容（图片、.CSS 文件、JavaScript），还要动态生成 HTML。
- 开发版 HTTP 服务器使用（不安全的）HTTP 协议而不是（安全的）HTTPS 协议。
- 开发版 HTTP 服务器在前台运行，直接接管你的终端窗口。

- 这个 HTTP 服务器的主机名只能是 127.0.0.1 (localhost)。

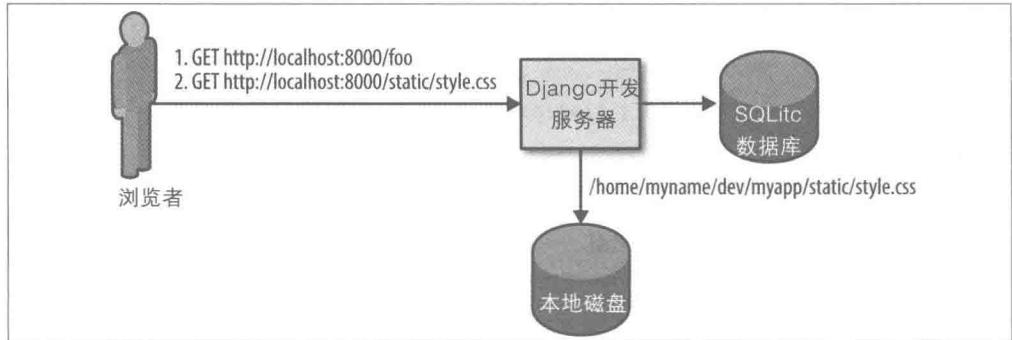


图5-2 开发模式的Django应用

现在，让我们来看看部署到生产环境的时候会发生什么。

PostgreSQL：数据库

SQLite 是单机版数据库软件，并没有服务器版。在生产环境中，我们希望运行服务器版本的数据库软件，因为这会对大规模并发请求有更好的支持。而且服务器版数据库还允许我们运行多个 HTTP 服务器用于负载均衡。这意味着我们需要部署一个数据库管理系统，例如 MySQL 或者 PostgreSQL（简称“Postgre”）。配置一台这样的数据库服务器需要很多工作。我们需要：

1. 安装数据库软件。
2. 确保数据库服务一直在运行。
3. 在数据库管理系统中创建数据库。
4. 为数据库系统创建用户并设置恰当的权限。
5. 在我们的 Mezzanine 应用中配置数据库用户和连接信息。

87

Gunicon：应用服务器

因为 Mezzanine 是基于 Django 的应用，你可以使用 Django 的 HTTP 服务器。在 Django 文档中将这个 HTTP 服务器称为开发服务器。下面是 Django 1.7 的文档中关于开发服务器的描述 (<https://docs.djangoproject.com/en/1.7/intro/tutorial01/#the-development-server>)：

不要在任何生产环境相关的场景下使用开发服务器。它是依照仅用于开发使用

的场景设计的。(我们的目的是开发一个 Web 框架, 而不是 Web 服务器。)

Django 实现了标准的 Web Server Gateway Interface (WSGI^{注2}), 所以任何支持 WSGI 的 Python HTTP 服务器, 也都适合运行像 Mezzanine 这样的 Django 应用。我们将会使用 Gunicorn。它是最流行的 HTTP WSGI 服务器之一, 同时也是 Mezzanine 部署脚本所选择的 HTTP 服务器。

nginx : Web 服务器

就像开发服务器所做的那样, Gunicorn 会执行我们的 Django 应用。但是 Gunicorn 并不能处理应用中涉及的静态内容, 如图片、.css 文件和 JavaScript 文件等。它们之所以被称为静态内容, 是因为与 Gunicorn 所处理的动态生成网页的程序相比, 它们不会发生改变。

尽管 Gunicorn 可以处理 TLS 加密, 但我们通常还是配置 nginx 来处理加密计算^{注3}。

如图 5-3 所示, 我们将使用 nginx 作为 Web 服务器处理静态内容和 TLS 加密。我们需要将 nginx 配置为 Gunicorn 的反向代理。如果是获取像 css 文件这样的静态内容的请求, nginx 将直接从本地文件系统中找到该静态文件并返回。否则, nginx 将会把请求代理到 Gunicorn。我们将会在 nginx 服务器本机上运行 Gunicorn, 这样 nginx 的代理就是向本地的 Gunicorn 服务发送一个 HTTP 请求。nginx 通过 URL 来判断是向 Gunicorn 返回本地文件还是代理请求。注意, 所有发送到 nginx 的请求都使用 (加密的) HTTPS 协议, 而所有 nginx 代理到 Gunicorn 的请求都使用 (未加密的) HTTP 协议。

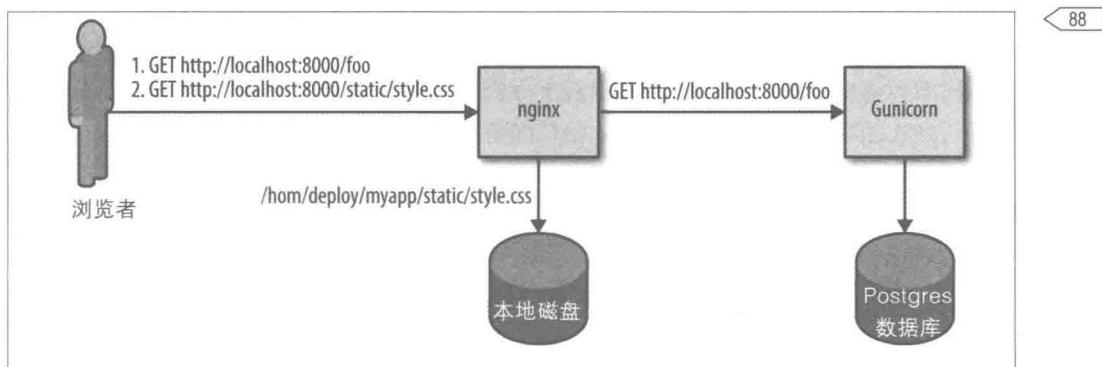


图5-3 nginx作反向代理

注2：WSGI 协议被记录在 Python Enhancement Proposal (PEP) 3333 中 (<https://www.python.org/dev/peps/pep-3333/>)。

注3：Gunicorn 0.17 添加了对 TLS 加密的支持。之前的版本中你必须使用像 nginx 这样的独立程序来处理加密。

Supervisor：进程管理器

当我们在开发模式运行的时候，我们会在终端的前台运行应用服务器。如果我们关闭终端，程序也将一同关闭。对于服务器应用，我们需要在后台运行它。这样的话，即便我们关闭了用于启动程序的终端，程序也不会关闭。

这类进程俗称为守护进程或者服务。我们需要以守护进程方式运行 Gunicorn，并且我们希望能够很容易地停止或重启它。有许多服务管理器可以实现这个需求。我们将会使用 Supervisor，因为这也是 Mezzanine 的部署脚本所使用的。

至此，你应该对向生产环境部署一个 Web 应用的步骤有大概的了解了。我们将在第 6 章详细讨论如何用 Ansible 实现这个部署过程。

使用Ansible部署Mezzanine

嗯，是时候写一个将 Mezzanine 部署到服务器上的 Ansible playbook 了。我们将从头到尾、一步一步地剖析这个过程。如果你是急于看剧透的人^{注1}，你可以在本章末尾的例 6-27 中看到完整的 playbook，在 GitHub (<https://github.com/lorin/ansiblebook/blob/master/ch06/playbooks/mezzanine.yml>) 上也有一份。在尝试运行它之前，一定要先看看 README (<https://github.com/lorin/ansiblebook/blob/master/ch06/README.md>)。

我已经尽最大努力使它保持与 Mezzanine 作者 Stephen McDonald 编写的原 Fabric 脚本相近。^{注2}

列出 playbook 中的 task

在我们深入到 playbook 的细节中去之前，让我们先对其有一个整体的认识。ansible-playbook 命令行工具支持一个叫作 `--list-tasks` 的参数。这个参数将会打印出这个 playbook 中所有任务的名字，这是一种方便的总结 playbook 将会做哪些事的方法。我们这样来使用它：

```
$ ansible-playbook --list-tasks mezzanine.yml
```

例 6-1 展示了例 6-27 中的 playbook `mezzanine.yml` 执行上述命令后的输出。

例 6-1 列出 Mezzanine playbook 中的 task

```
playbook: mezzanine.yml
```

```
play #1 (Deploy mezzanine on vagrant):
```

注 1： 我的妻子 Stacy 就有这样的癖好。

注 2： 你可以在 GitHub 上找到随 Mezzanine 一起发布的那个 Fabric 脚本 (https://github.com/stephenmcd/mezzanine/blob/master/mezzanine/project_template/fabfile.py)。

```

install apt packages
check out the repository on the host
install required python packages
install requirements.txt
create a user
create the database
generate the settings file
sync the database, apply migrations, collect static content
set the site id
set the admin password
set the gunicorn config file
set the supervisor config file
set the nginx config file
enable the nginx config file
remove the default nginx config file
ensure config path exists
create tls certificates
install poll twitter cron job

```

组织要部署的文件

正如我们之前讨论过的，Mezzanine 是基于 Django 编写的。在 Django 中，一个 Web 应用叫作一个 *project*。我们需要为我们的 project 取个名字，这里我取名为 *mezzanine-example*。

我们的 playbook 向 Vagrant 虚拟机中部署，并且会将文件部署到 Vagrant 账户的主目录。

/home/vagrant/mezzanine-example 是我们部署目标的最高层级目录。它也将作为 virtualenv 的目录，这意味着我们将要把所有相关的 Python 安装包安装到这个目录中。

/home/vagrant/mezzanine-example/project 将会用来放置所有从 GitHub 源代码仓库中复制出来的源代码。

变量与私密变量

如你在例 6-2 中所看到的，这个 playbook 只会定义很少的变量。

例 6-2 定义变量

```

vars:
  user: "{{ ansible_ssh_user }}"
  proj_name: mezzanine-example
  venv_home: "{{ ansible_env.HOME }}"
  venv_path: "{{ venv_home }}/{{ proj_name }}"

```

```
proj_dirname: project
proj_path: "{{ venv_path }}/{{ proj_dirname }}"
reqs_path: requirements.txt
manage: "{{ python }} {{ proj_path }}/manage.py"
live_hostname: 192.168.33.10.xip.io
domains:
  - 192.168.33.10.xip.io
  - www.192.168.33.10.xip.io
repo_url: git@github.com:lorin/mezzanine-example.git
gunicorn_port: 8000
locale: en_US.UTF-8
# Mezzanine 中的 Fabric 脚本中并不包含下面的变量
# 我在这里定义它们是为了方便起见
conf_path: /etc/nginx/conf
tls_enabled: True
python: "{{ venv_path }}/bin/python"
database_name: "{{ proj_name }}"
database_user: "{{ proj_name }}"
database_host: localhost
database_port: 5432
gunicorn_proc_name: mezzanine
vars_files:
  - secrets.yml
```

< 91

我尽量让绝大部分变量使用与 Mezzanine 的 Fabric 脚本中相同的变量名。我也添加了一些新的变量以让配置更清楚。例如，Fabric 脚本直接使用了 `proj_name` 作为数据库名和数据库用户名。而我更倾向于定义名为 `database_name` 和 `database_user` 的中间变量，并根据 `proj_name` 的值来定义这些中间变量。

这里有几点需要注意。首先是我们如何定义一个变量并基于另一个变量的值来设定它。例如，我们根据 `venv_home` 和 `proj_name` 变量的值来设定 `venv_path` 变量。

其次是我们如何在这些变量中引用 Ansible facts。例如，根据从每台主机上收集的 `ansible_env_fact` 来设定 `venv_home` 变量。

最后是我们如何在名为 `secrets.yml` 的独立文件中指定一部分变量。具体方法如下：

```
vars_files:
  - secrets.yml
```

这个文件包含像密码和令牌这样需要保密的认证信息。注意我的 GitHub 仓库中并没有包含这个文件，而是包含了一个名为 `secrets.yml.example` 的文件，内容如下所示：

```
db_pass: e79c9761d0b54698a83ff3f93769e309
```

```

admin_pass: 46041386be534591ad24902bf72071B
secret_key: b495a05c396843b6b47ac944a72c92ed
nevercache_key: b5d87bb4e17c483093296fa321056bdc
# 为了满足 Mezzanine 与 Twitter 结合功能要求，你需要先在 https://dev.twitter.com 上创建一个 Twitter 应用
#
# 获取与 Twitter 相结合的更详细资料可以访问 http://mezzanine.jupo.org/docs/twitter-integration.html
twitter_access_token_key: 80b557a3a8d14cb7a2b91d60398fb8ce
twitter_access_token_secret: 1974cf8419114bdd9d4ea3db7a210d90
twitter_consumer_key: 1f1c627530b34bb58701ac81ac3fad51
twitter_consumer_secret: 36515c2b60ee4ffb9d33d972a7ec350a

```

如果需要使用这个仓库中的版本的话，你需要将 `secrets.yml.example` 拷贝为 `secrets.yml`，并编辑其内容使之包含你站点的认证信息。还要注意，这个 `secrets.yml` 文件需要包含在 Git 仓库中的 `.gitignore` 文件中，以防止有人不小心提交了这些认证信息。

由于涉及安全风险，最好避免将未加密的认证信息提交到你的版本控制仓库中。这是维护授权信息的可选策略之一。我们也可以把它们作为环境变量来传递。第 7 章将会详细描述另外一种方案：使用 Ansible 的 `vault` 功能提交加密版本的 `secrets.yml`。

使用迭代（`with_items`）安装多个软件包

为了部署我们的 Mezzanine 应用，我们需要安装两种不同类型的软件包。第一类是一些系统级的软件包。由于我们将会部署到 Ubuntu 上，所以将会使用 `apt` 作为包管理工具来安装这些系统级软件。第二类是一些 Python 软件包，我们将使用 `pip` 来安装 Python 软件包。

因为系统级软件包是特意设计来与操作系统配合工作的，所以一般来说系统级软件包比 Python 软件包更容易处理。但是，系统软件包仓库一般不会包含我们所需要的最新版本的 Python 库，所以我们需要使用 Python 软件包来安装它们。这是一个在稳定性与最新版本、最强功能之间的权衡。

例 6-3 列出了我们将会用于安装系统软件包的 task

例 6-3 安装系统软件包

```

- name: install apt packages
  apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
  sudo: True
  with_items:
    - git
    - libjpeg-dev

```

```
- libpq-dev
- memcached
- nginx
- postgresql
- python-dev
- python-pip
- python-psycopg2
- python-setuptools
- python-virtualenv
- supervisor
```

这里有许多需要解读的内容。因为我们要安装多个软件包，所以我们使用了 Ansible 的迭代功能，`with_items` 语句。我们也可以逐个安装这些软件包，就像这样：

```
- name: install git
  apt: pkg=git

- name: install libjpeg-dev
  apt: pkg=libjpeg-dev

...
```

很明显，如果我们把所有的软件包组织成一个列表的方式，将更易于读写。当我们调用 `apt` 模块的时候，我们向它传递了 `{{ item }}`。这是一个占位变量，它会被 `with_items` 语句中列表的每一个元素分别替代。

 Ansible 总是使用 `item` 作为循环迭代变量的名字。

此外，在 `apt` 模块的案例中，使用 `with_items` 语句来安装多个软件包效率更高。这是由于 Ansible 会将整个软件包列表一起传递给 `apt` 模块，而模块只调用 `apt` 命令一次，将整个软件包列表作为需要安装的软件包传给该命令。许多模块（如 `apt`）原生设计就能像上面的行为一样处理迭代列表。如果模块并不原生支持列表，那么 Ansible 将会简单地多次调用该模块，列表中的每个元素都要调用一次。

你可以从下面的输出中看到，`apt` 模块可以智能地一次处理多个软件包。

```
TASK: [install apt packages] ****
ok: [web] => (item=git,libjpeg-dev,libpq-dev,memcached,nginx,postgresql,
python-dev,python-pip,python-psycopg2,python-setuptools,python-virtualenv,
supervisor)
```

相对的，pip 模块就不能智能地处理迭代列表。所以 Ansible 只能为列表中的每个元素都调用一次该模块，输出就像下面这样：

```
94 ┶
TASK: [install other python packages] ****
ok: [web] => (item=gunicorn)
ok: [web] => (item=setproctitle)
ok: [web] => (item=south)
ok: [web] => (item=psycopg2)
ok: [web] => (item=django-compressor)
ok: [web] => (item=python-memcached)
```

向 task 中添加 sudo 语句

在第 2 章的 playbook 范例中，我们希望整个 playbook 都以 root 身份运行，所以我们向 play 中添加了 sudo: True 语句。

在我们部署 Mezzanine 的时候，绝大部分 task 使用 SSH 到主机的用户身份运行就可以了，并不需要 root 权限。所以我们并不希望整个 play 都使用 sudo 运行，而是只选择需要的 task 来使用 sudo。

要实现这一点，只需要向希望使用 root 身份运行的 task 中添加 sudo: True，就像例 6-3 所示的那样。

更新 apt 缓存



本节中的所有范例命令都是运行在（Ubuntu）远程主机上的，而不是控制主机。

Ubuntu 会对 Ubuntu 软件包仓库中所有可用的 apt 软件包的名字维护一个缓存。假如我们需要安装名为 libssl-dev 的软件包，可以使用 apt-cache 命令查询本地缓存中具有该软件包的那个版本：

```
$ apt-cache policy libssl-dev
```

输出如例 6-4 所示。

例 6-4 apt-cache 的输出

```
libssl-dev:
```

```
Installed: (none)
Candidate: 1.0.1f-1ubuntu2.5
Version table:
  1.0.1f-1ubuntu2.5 0
    500 http://archive.ubuntu.com/ubuntu/ trusty-updates/main amd64 Packages
    500 http://security.ubuntu.com/ubuntu/ trusty-security/main amd64
Packages
  1.0.1f-1ubuntu2 0
    500 http://archive.ubuntu.com/ubuntu/ trusty/main amd64 Packages
```

如我们所见，这些软件包并没有在本地安装。根据本地缓存的信息，最新版本是 1.0.1f-ubuntu2.5。我们还能看到一些关于软件包仓库位置的信息。◀95

某些情况下，在 Ubuntu 项目发布了软件包的新版本时，会在软件包仓库中移除旧版本。如果 Ubuntu 服务器的本地 apt 缓存没有及时更新，那么它将会提示要安装的软件包在软件包仓库中不存在。

回到我们的范例，假如我们要尝试安装 libssl-dev 软件包：

```
$ apt-get install libssl-dev
```

如果 1.0.1f-ubuntu2.5 版本在软件包仓库上已经不存在了，我们将会看到如下错误：

```
Err http://archive.ubuntu.com/ubuntu/ trusty-updates/main libssl-dev amd64
1.0.1f-1ubuntu2.5
  404 Not Found [IP: 91.189.88.153 80]
Err http://security.ubuntu.com/ubuntu/ trusty-security/main libssl-dev amd64
1.0.1f-1ubuntu2.5
  404 Not Found [IP: 91.189.88.149 80]
Err http://security.ubuntu.com/ubuntu/ trusty-security/main libssl-doc all
1.0.1f-1ubuntu2.5
  404 Not Found [IP: 91.189.88.149 80]
E: Failed to fetch
http://security.ubuntu.com/ubuntu/pool/main/o/openssl/libssl-dev_1.0.1f-
1ubuntu2.5_amd64.deb
  404 Not Found [IP: 91.189.88.149 80]

E: Failed to fetch
http://security.ubuntu.com/ubuntu/pool/main/o/openssl/libssl-doc_1.0.1f-
1ubuntu2.5_all.deb
  404 Not Found [IP: 91.189.88.149 80]

E: Unable to fetch some archives, maybe run apt-get update or try with
--fix-missing?
```

在命令行中更新本地 apt 缓存的方法是运行 `apt-get update`。在我们使用 Ansible 的

apt 模块时，更新 apt 缓存的方法是在调用模块的时候传入 `update_cache=yes` 参数，就像例 6-3 中做的那样。

由于更新缓存需要消耗更多的时间，并且在排查错误时我们一般会连续运行 playbook 多次，因此我们可以通过在模块中使用 `cache_valid_time` 参数来避免缓存更新带来的时间消耗。这能通知模块只有在超过一定临界值的时候才需要更新缓存。例 6-3 中使用了 `cache_valid_time=3600`，这意味着只有距上一次更新缓存 3600 秒（1 小时）后才会再次更新缓存。

96 使用 Git 来 Check Out 项目源码

Mezzanine 可以在不编写任何定制代码的情况下使用，但它的特长之一就是基于 Django 平台开发。如果你了解 Python，应该知道 Django 是一个极好的 Web 应用平台。如果你仅仅需要一个 CMS 系统，可能像 WordPress 这样的系统更适合你。但是如果你想要编写一个包含 CMS 功能的定制应用，Mezzanine 是一个不错的选择。

作为开发环节的一部分，你需要从包含你的 Django 应用的 Git 仓库中 check out 源代码。如果用 Django 的术语来讲，那就是这个仓库中必须包含一个 *project*。我在 GitHub 上创建了一个包含 Django project 必备文件的仓库 (<https://github.com/lorin/mezzanine-example>)，也就是这个 playbook 所部署的 project。

我使用 Mezzanine 附带的 `mezzanine-project` 程序创建这些文件。就像这样：

```
$ mezzanine-project mezzanine-example
```

在我们的代码仓库中并没有任何定制的 Django 应用，仅仅是这个 *project* 所需要的文件而已。在一个正式的 Django 应用部署中，这个代码仓库将会包含若干子目录，而这些子目录中会包含另外的 Django 应用。

例 6-5 演示了我们如何使用 git 模块从 Git 仓库中 check out 源代码到远程主机上。

例 6-5 从 Git 仓库 Check Out 源代码

```
- name: check out the repository on the host
  git: repo={{ repo_url }} dest={{ proj_path }} accept_hostkey=yes
```

我将项目的代码仓库设置为公开，以便读者可以访问这个仓库。但是一般来说，你将会通过 SSH 从私有 Git 仓库中获取文件。为此，我设置了 `repo_url` 变量，以便使用下面的方式通过 SSH 克隆仓库：

```
repo_url: git@github.com:lorin/mezzanine-example.git
```

如果你在家中实验这个范例，运行这个 playbook 之前你必须具备以下条件：

1. 拥有一个 GitHub 账号。
2. 拥有一个分配给你的 GitHub 账号的 SSH 公钥。
3. 在你的控制主机上运行 SSH agent 并打开 agent forwarding。

可以在 `ansible.cfg` 中添加如下内容打开 agent forwarding：

```
[ssh_connection]
  ssh_args = -o ControlMaster=auto -o ControlPersist=60s -o ForwardAgent=yes
```

除了使用 `repo` 参数来指定仓库的 URL，以及用 `dest` 参数来指定仓库的目标路径之外，◀ 97
我们还使用了参数 `accept_hostkey`，这与 host key 检查有关。在附录 A 中有关于 SSH
agent forwarding 和 host key 检查的详细信息。

将 Mezzanine 和其他软件包安装到 virtualenv 中

就像本章前面提到的，我们将会使用 Python 软件包来安装一些软件，因为这样得到的软件比使用 `apt` 安装得到的版本更新。

我们可以使用 `root` 用户在系统级安装 Python 软件包，但是更好的实践方案是，将这些包安装到独立的环境中，以避免污染系统级 Python 软件包。在 Python 中，这种独立的软件包环境叫作 `virtualenv`。一个用户可以创建多个 `virtualenv` 并且可以向 `virtualenv` 中安装 Python 软件包而不需要 `root` 权限。

Ansible 的 `pip` 模块支持向 `virtualenv` 中安装软件包，并且还支持在没有可用的 `virtualenv` 时自动创建一个。例 6-6 列出了两个我们用来向 `virtualenv` 中安装 Python 软件包的 task，两个 task 都使用了 `pip` 模块，但使用方式不同。

例6-6 安装Python软件包

```
- name: install required python packages
  pip: name={{ item }} virtualenv={{ venv_path }}
  with_items:
    - gunicorn
    - setproctitle
    - south
    - psycopg2
    - django-compressor
    - python-memcached
```

```
- name: install requirements.txt
  pip: requirements={{ proj_path }}/{{ reqs_path }} virtualenv={{ venv_path }}
```

Python 项目的通常模式是使用名为 *requirements.txt* 的文件制定软件包依赖关系。嗯，我们仓库中的 Mezzanine 范例也包含一个 *requirements.txt* 文件，它的内容如例 6-7 所示。

例6-7 requirements.txt

```
Mezzanine==3.1.10
```

这个 *requirement.txt* 缺少了我们这个部署所需要的某些 Python 软件包，所以我们将作为独立的 task 来明确地指定安装它们。

需要注意的是，Mezzanine 的 Python 软件包在 *requirements.txt* 中被限制在特定版本（3.1.10），而其他软件包并没有被限定，我们将会获取它们的最新版本。如果我们不希望限制 Mezzanine 的版本，我们可以像下面这样将 Mezzanine 添加到软件包列表中：

```
- name: install python packages
  pip: name={{ item }} virtualenv={{ venv_path }}
  with_items:
    - mezzanine
    - gunicorn
    - setproctitle
    - south
    - psycopg2
    - django-compressor
    - python-memcached
```

相对的，如果我们希望限定所有软件包的版本，我们有几种可选的实现方法。我们可以通过 *requirements.txt* 文件来实现，这个文件包含软件包和依赖关系的信息。例 6-8 展示了一个范例文件。

例6-8 requirements.txt范例

```
Django==1.6.8
Mezzanine==3.1.10
Pillow==2.6.1
South==1.0.1
argparse==1.2.1
beautifulsoup4==4.1.3
bleach==1.4
django-appconf==0.6
django-compressor==1.4
filebrowser-safe==0.3.6
future==0.9.0
grappelli-safe==0.3.13
```

```
gunicorn==19.1.1
html5lib==0.999
oauthlib==0.7.2
psycopg2==2.5.4
python-memcached==1.53
pytz==2014.10
requests==2.4.3
requests-oauthlib==0.4.2
setproctitle==1.1.8
six==1.8.0
tzlocal==1.0
wsgiref==0.1.2
```

如果你有一个已经存在并安装过软件包的 virtualenv，你可以使用 `pip freeze` 命令来打印已经安装的软件包列表。例如，如果你的 virtualenv 在 `~/mezzanine-example` 目录下，就可以用如下所示的方法来激活 virtualenv 并输出 virtualenv 中的软件包：

```
$ source ~/mezzanine_example/bin/activate
$ pip freeze > requirements.txt
```

例 6-9 演示了在拥有 `requirements.txt` 文件的情况下，我们如何使用这个文件来安装软件包。

例6-9 通过requirements.txt安装软件包

```
- name: copy requirements.txt file
  copy: src=files/requirements.txt dest=~/requirements.txt
- name: install packages
  pip: requirements=~/requirements.txt virtualenv={{ venv_path }}
```

或者，我们可以通过列表同时指定软件和对应的版本，如例 6-10 所示。我们传递了一个字典的列表，并且使用 `item.name` 和 `item.version` 来解引元素。

例6-10 指定软件包的名称与版本

```
- name: python packages
  pip: name={{ item.name }} version={{ item.version }} virtualenv={{ venv_path }}
  with_items:
    - {name: mezzanine, version: 3.1.10 }
    - {name: gunicorn, version: 19.1.1 }
    - {name: setproctitle, version: 1.1.8 }
    - {name: south, version: 1.0.1 }
    - {name: psycopg2, version: 2.5.4 }
    - {name: django-compressor, version: 1.4 }
    - {name: python-memcached, version: 1.53 }
```

task 中的复杂参数：稍微跑个题

到目前为止，每当我们调用模块的时候，模块的参数都是作为字符串传入的。就比如例 6-10 中的 pip 范例，我们向 pip 模块传入一个字符串作为参数：

```
- name: install package with pip
  pip: name={{ item.name }} version={{ item.version }} virtualenv={{ venv_path }}
```

如果不喜欢文件中出现过长的行，我们可以使用第 30 页讨论过的 YAML 的折行功能将参数字符串分割成多行：

```
[100] - name: install package with pip
  pip:
    name={{ item.name }}
    version={{ item.version }}
    virtualenv={{ venv_path }}
```

Ansible 还为我们提供了另一个将模块调用分割成多行的选择。我们可以传递 key 为变量名的字典，而不是传递字符串参数。这意味着我们可以将例 6-10 变更为下面这样：

```
- name: install package with pip
  pip:
    name: "{{ item.name }}"
    version: "{{ item.version }}"
    virtualenv: "{{ venv_path }}"
```

基于字典传递参数的方法在调用拥有复杂参数的模块时非常有用。复杂参数就是指模块的参数是一个列表或者一个字典。用于在 Amazon EC2 上创建新服务器的 ec2 模块就是拥有复杂参数的模块的一个范例。例 6-11 演示了如何调用一个使用列表作为 group 的参数，同时使用字典作为 instance_tags 参数的模块。我们将会在第 12 章更详细地讨论这个模块。

例 6-11 调用一个具有复杂参数的模块

```
- name: create an ec2 instance
  ec2:
    image: ami-8ca1ce4
    instance_type: m3.medium
    key_name: mykey
    group:
      - web
      - ssh
    instance_tags:
      type: web
      env: production
```

你甚至可以混合使用多种方法，一些参数使用字符串，而其他的参数使用字典。在混用的时候，你需要使用 `args` 语句来指定那些字典变量。我们可以将之前的范例改写为如下形式：

```
- name: create an ec2 instance
  ec2: image=ami-8caalce4 instance_type=m3.medium key_name=mykey
  args:
    group:
      - web
      - ssh
  instance_tags:
    type: web
    env: production
```

如果你正在使用 `local_action` 语句（我们将在第 7 章详细讨论它），那么复杂参数的语法会有些细微的变化。你需要像下面这样添加 `module: <modulename>`。

```
- name: create an ec2 instance
  local_action:
    module: ec2
    image: ami-8caalce4
    instance_type: m3.medium
    key_name: mykey
    group:
      - web
      - ssh
  instance_tags:
    type: web
    env: production
```

101

同样，你仍然可以在使用 `local_action` 时混合使用简单参数与复杂参数：

```
- name: create an ec2 instance
  local_action: ec2 image=ami-8caalce4 instance_type=m3.medium key_name=mykey
  args:
    image: ami-8caalce4
    instance_type: m3.medium
    key_name: mykey
    group:
      - web
      - ssh
  instance_tags:
    type: web
    env: production
```



在使用 `file`、`copy` 和 `template` 等模块时，Ansible 允许你指定文件权限。文件权限常用八进制整数来表示。如果你将八进制数作为复杂参数，你必须在八进制整数前加 0，或者用引号引起起来作为字符串传入。

还是用范例来说明吧。我们可以注意到 `mode` 参数的值是以 0 开头的：

```
- name: copy index.html
  copy:
    src: files/index.html
    dest: /usr/share/nginx/html/index.html
    mode: "0644"
```

如果 `mode` 参数的值没有以 0 开头，也没有作为字符串被引号引起起来，Ansible 将会将其解释为十进制整数而不是八进制整数。这将会导致不能按照你期望的方式设置文件的权限。你可以在 GitHub (<https://github.com/ansible/ansible/issues/9196>) 上找到更多详细信息。

102> 如果你希望将参数分成多行，并且没有使用复杂参数的话，那么选用哪种形式就仅仅是个人喜好问题。一般来说我比较推荐字典形式（相对于多行字符串形式），但是在本书中每种形式都会用到。

创建数据库和数据库用户

当 Django 运行在开发模式的时候，它使用 SQLite 作为后端。如果数据库文件不存在的话，后端将会自动创建数据库文件。

当使用类似 Postgre 这样的数据库管理系统的時候，我们需要在 Postgre 中创建数据库，并创建这个数据库的所有者账户。稍后我们将为 Mezzanine 配置这个用户的认证信息。

Ansible 随附的 `postgresql_user` 模块和 `postgresql_db` 模块可以用于创建 Postgre 的用户和数据库。例 6-12 演示了如何在我们的 playbook 中调用这些模块。

例 6-12 创建数据库和数据库用户

```
- name: create a user
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"
    sudo: True
    sudo_user: postgres

- name: create the database
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"
```

```
encoding: UTF8
lc_ctype: "{{ locale }}"
lc_collate: "{{ locale }}"
template: template0
sudo: True
sudo_user: postgres
```

你可能已经注意到每个任务都使用了 `sudo: True` 和 `sudo_user: postgres`。当你在 Ubuntu 上安装 Postgre 的时候，安装程序创建了名为 `postgres` 的用户。该用户具有 Postgre 的管理权限。需要注意的是，默认情况下 `root` 账户并没有 Postgre 的管理权限。所以在 playbook 中，我们需要 `sudo` 到 `postgres` 用户以执行类似创建用户和数据库这样的管理任务。

当我们创建数据库的时候，我们设定了数据库相关的编码（UTF8）与 locale 类别（LC_TYPE、LC_COLLATE）。因为我们设定了区域（locale）信息，所以我们使用 `template0` 作为模板。^{注3}

从模板生成 local_settings.py 文件

Django 希望在名为 `settings.py` 的文件中找到与具体 project 相关的配置。Mezzanine 遵循 Django 的将配置分为两部分的规则：

- 在所有部署环境中都相同的配置 (`settings.py`)。
- 随着部署环境而变化的配置 (`local_settings.py`)。

对于在所有部署环境中都相同的配置，我们放在项目仓库中的 `settings.py` 文件中定义。你可以在 GitHub (<https://github.com/lorin/mezzanine-example/blob/master/settings.py>) 上找到这个文件。

如例 6-13 所示，`settings.py` 文件中包含一段加载 `local_settings.py` 文件的 Python 代码片段。如果 `local_settings.py` 文件不存在的话，Django 将抛出一个异常。

例6-13 加载本地配置

```
try:
    from local_settings import *
except ImportError as e:
    if "local_settings" not in str(e):
        raise e
```

注3： 如果想要获取关于数据库模板的更详细信息，请查阅 Postgres 的文档 (<http://www.postgresql.org/docs/9.4/static/manage-ag-templatebs.html>)。

此外，`.gitignore` 文件被配置为对 `local_settings.py` 文件忽略。这是由于开发者通常在本地开发时才会创建并配置这个文件。

我们需要创建一个 `local_settings.py` 文件并将它上传到远程主机上，这也是我们部署工作的一部分。例 6-14 列出了我们用于生成这个文件的 Jinja2 模板。

例 6-14 `local_settings.py.j2`

```
from __future__ import unicode_literals

SECRET_KEY = "{{ secret_key }}"
NEVERCACHE_KEY = "{{ nevercache_key }}"
ALLOWED_HOSTS = [% for domain in domains %]{{ domain }},{% endfor %}

DATABASES = {
    "default": {
        # 该配置的值需要以 "postgresql_psycopg2"、"mysql"、"sqlite3" 或者 "oracle" 结尾
        "ENGINE": "django.db.backends.postgresql_psycopg2",
        # 数据库的名字。但如果使用 sqlite3，则为数据库的路径
        "NAME": "{{ proj_name }}",
        # 使用 sqlite3 时，不使用这个配置
        "USER": "{{ proj_name }}",
        # 使用 sqlite3 时，不使用这个配置
        "PASSWORD": "{{ db_pass }}",
        # 对于本机，可以设置为空字符串。使用 sqlite3 时，不使用这个配置
        "HOST": "127.0.0.1",
        # 该配置的值默认为空字符串。使用 sqlite3 时，不使用这个配置
        "PORT": "",
    }
}
SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTOCOL", "https")

CACHE_MIDDLEWARE_SECONDS = 60

CACHE_MIDDLEWARE_KEY_PREFIX = "{{ proj_name }}"

CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.MemcachedCache",
        "LOCATION": "127.0.0.1:11211",
    }
}
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

该模板的大部分内容都比较直接易懂。在模板中使用了 `{{ variable }}` 语法来插入类似 `secret_key`、`nevercache_key`、`proj_name` 和 `db_pass` 等变量的值。唯一不容易理解逻

辑的就是例 6-15 所示的这一行：

例6-15 在Jinja2模板中使用for循环

```
ALLOWED_HOSTS = [{% for domain in domains %}"{{ domain }}",{% endfor %}]
```

如果我们回过头重新看看变量定义部分，可以看到有一个名为 `domains` 的变量，它的定义是这样的：

```
domains:  
- 192.168.33.10.xip.io  
- www.192.168.33.10.xip.io
```

我们的 Mezzanine 应用只会响应向 `domains` 变量中的主机名之一发起的请求，在我们的范例中是 `http://192.168.33.10.xip.io` 或者 `http://www.192.168.33.10.xip.io`。如果一个请求到达 Mezzanine，但是 host 头信息是这两个域名以外的域名，那么它将会返回“Bad Request(400).”

我们希望这一行最终成为如下的样子：

```
ALLOWED_HOSTS = ["192.168.33.10.xip.io", "www.192.168.33.10.xip.io"]
```

105

我们可以像例 6-15 所示那样使用 `for` 循环来达到这个目的。注意，它处理的结果并不完全和我们所期望的一样。实际上列表的结尾会多一个尾逗号，就像下面这样：

```
ALLOWED_HOSTS = ["192.168.33.10.xip.io", "www.192.168.33.10.xip.io",]
```

不过对于列表中的尾逗号，Python 会进行完美地处理，所以我们可以就这样不去管它。

什么是 xip.io

你可能已经注意到了，我们使用的域名看起来有点奇怪：`192.168.33.10.xip.io` 和 `www.192.168.33.10.xip.io`。它们是域名，但是里面还嵌入了 IP 地址。

访问网站时，我们一般是在浏览器中输入类似 `http://www.ansiblebook.com` 这样的域名来访问它们，而不是像 `http://54.225.155.135` 这样的 IP 地址。

当我们编写向 Vagrant 中部署 Mezzanine 的 playbook 时，我们希望使用可以被访问到的域名来配置应用。

问题是我们没有映射到 Vagrant 虚拟机 IP 地址的 DNS 记录，在这个范例中是 `192.168.33.10`。当然没人阻止我们设置一个相应的 DNS 记录。例如，我们可以创建 DNS 记录将 `mezzanine-internal.ansiblebook.com` 指向 `192.168.33.10`。

实际上，如果我们希望创建一条解析到私有 IP 地址的 DNS 记录，有一个很便捷的服务叫作 *xip.io*。*xip.io* 服务由 Basecamp 提供并且完全免费。使用这个服务的话，我们就不需要创建自己的 DNS 记录了。如果 *AAA.BBB.CCC.DDD* 是一个 IP 地址，那么 DNS 记录 *AAA.BBB.CCC.DDD.xip.io* 将会解析到 *AAA.BBB.CCC.DDD*。例如，*192.168.33.10.xip.io* 将会解析到 *192.168.33.10*。除此之外，*www.192.168.33.10.xip.io* 也解析到 *192.168.33.10*。

我认为 *xip.io* 是一个非常赞的工具，尤其是我想把 Web 应用部署到私有 IP 地址用于测试的时候。

亦或者，你可以通过向你本地主机的 */etc/hosts* 文件添加相应条目的方法达到这个目的。这个方法在没有网络连接的时候仍然可以工作。

让我们再诊查一下 Jinja2 的 `for` 循环语法。为了让它更易读，我们可以将它分为多行，如下所示：

```
ALLOWED_HOSTS = [  
    {% for domain in domains %}  
        "{{ domain }}",  
    {% endfor %}  
]
```

106 生成的配置文件将会像下面这样，它对于 Python 仍然是合法的：

```
ALLOWED_HOSTS = [  
    "192.168.33.10.xip.io",  
    "www.192.168.33.10.xip.io",  
]
```

需要注意的是，`for` 循环可以被 `{% endfor %}` 语句中断。此外还需要注意 `for` 语句和 `endfor` 语句需要使用 `{% %}` 括起来。这与用于变量替换的 `{{ }}` 有所不同。

所有在 `playbook` 中定义的变量和 `fact` 都可以在 Jinja2 模板中使用。所以我们并不需要显式地向模板传递变量。

运行 `django-manage` 命令

Django 应用使用一个特殊的脚本叫作 `manage.py` (<http://bit.ly/1FvJwnp>)。这个脚本负责向 Django 应用执行管理命令，例如：

- 创建数据库表。

- 应用数据库迁移。
- 将文件中的数据加载到数据库。
- 将数据库中的数据导出到文件。
- 将静态内容复制到相应的目录下。

除了 `manage.py` 支持的内置命令之外，Django 应用还可以添加自定义命令。Mezzanine 就添加了名为 `createdb` 的自定义命令。`createdb` 命令用于初始化数据库并将静态内容复制到相应位置。官方 Fabric 脚本的操作等同于如下命令：

```
$ manage.py createdb --noinput --nodata
```

Ansible 随附的 `django_manage` 模块就会调用 `manage.py` 命令。我们可以使用如下方法调用它：

```
- name: initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
```

不幸的是，Mezzanine 添加的 `createdb` 自定义命令并不是幂等性的。如果我们第二次调用它，将会产生如下错误：

```
TASK: [initialize the database] *****
failed: [web] => {"cmd": "python manage.py createdb --noinput --nodata",
"failed": true, "path": "/home/vagrant/mezzanine_example/bin:/usr/local/
sbin:/usr/local/bin:/usr/sbin: /usr/bin:/sbin:/bin:/usr/games:/usr/local/
games", "state": "absent", "syspath": ["", "/usr/lib/python2.7", "/usr/
lib/python2.7/plat-x86_64-linux-gnu", "/usr/lib/python2.7/lib-tk", "/usr/
lib/python2.7/lib-old", "/usr/lib/python2.7/lib-dynload", "/usr/local/lib/
python2.7/dist-packages", "/usr/lib/python2.7/dist-packages"]}
msg:
:stderr: CommandError: Database already created, you probably want the syncdb
or
migrate command
```

107

不幸中的万幸是，`createdb` 自定义命令等效于下面三个幂等性的 `manage.py` 内置命令。

syncdb

为没有被 South 管理的 Django model 创建数据库表。South (<http://south.readthedocs.org/en/latest/>) 是一个为 Django 应用进行数据库迁移的库。

migrate

为已经被 South 管理的 Django model 创建和更新数据库表。

collectstatic

将静态内容复制到相应目录下。

通过调用这些命令，我们可以得到一个幂等性的 task：

```
- name: sync the database, apply migrations, collect static content
  django_manage:
    command: "{{ item }}"
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
  with_items:
    - syncdb
    - migrate
    - collectstatic
```

在应用环境中运行自定义的 Python 脚本

为了初始化我们的应用，我们需要向数据库中应用两个变更。

1. 我们需要创建一个包含我们站点的域名（在我们的范例中，域名就是 `192.168.33.10.xip.io`）的 Site model 对象 (<https://docs.djangoproject.com/en/1.7/ref/contrib/sites/#django.contrib.sites.models.Site>)。
2. 我们需要设置管理员账户和密码。

尽管我们可以直接使用 SQL 命令来实现这些变更，但通常我们还是通过编写 Python 代码来实现。而且 Mezzanine 的 Fabric 脚本也是这样做的，所以我们也会这么做。

108 这里有两个比较难处理的地方。Python 脚本需要在我们已经创建的 virtualenv 环境中运行。同时，这个 Python 环境还需要被正确设置，以便能够让这个脚本导入在 `~/mezzanine_example/project` 下的 `settings.py` 文件。

大部分情况下，当我需要自定义一些 Python 代码的时候，我会编写一个定制的 Ansible 模块。然而就我所知，目前 Ansible 不允许你在 virtualenv 环境中运行模块，所以这个方法出局了。

我使用 `script` 模块作为替代方案。这个模块会复制自定义的脚本并执行它。我编写了两个脚本，一个用于设置 Site 记录，而另一个用于设置 admin 用户名和密码。

你可以向 `script` 模块传递命令行参数并解析它们，但是我还是决定使用环境变量来传递参数。我不希望通过命令行参数来传递密码（这会造成运行 `ps` 命令时，密码出现在进程列表中的现象）。而且，在脚本中解析环境变量比解析命令行参数更容易。



向 Ansible 的 task 传递环境变量

Ansible 允许你通过向 task 中添加 `environment` 语句，并传入包含环境变量名字与值的字典来设置环境变量。你可以向任何一个任务中添加 `environment` 语句，并不限定是 `script` 模块。

为了在 `virtualenv` 环境中运行这些脚本，我还需要设置 `path` 变量，以确保 `path` 中第一个可执行的 Python 是 `virtualenv` 中的那个。例 6-16 演示了我是如何调用这两个脚本的。

例 6-16 使用 script 模块来调用定制的 Python 代码

```
- name: set the site id
  script: scripts/setsite.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    WEBSITE_DOMAIN: "{{ live_hostname }}"
- name: set the admin password
  script: scripts/setadmin.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    ADMIN_PASSWORD: "{{ admin_pass }}"
```

这两个脚本展示在例 6-17 和例 6-18 中。我将它们放在 `scripts` 子目录中。

109

例 6-17 scripts/setsite.py

```
#!/usr/bin/env python
# 这是一个设置网站域名的脚本程序
# 本程序假定存在如下两个环境变量
#
# PROJECT_DIR: 项目目录（例如：~/projname）
# WEBSITE_DOMAIN: 网站的域名（例如：www.example.com）

import os
import sys

# 将项目目录添加到 PATH 系统环境变量中
proj_dir = os.path.expanduser(os.environ['PROJECT_DIR'])
sys.path.append(proj_dir)

os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
from django.conf import settings
from django.contrib.sites.models import Site

domain = os.environ['WEBSITE_DOMAIN']
Site.objects.filter(id=settings.SITE_ID).update(domain=domain)
```

```
Site.objects.get_or_create(domain=domain)

例6-18 scripts/setadmin.py
#!/usr/bin/env python
# 这是一个设置管理员密码的脚本程序
# 本程序假定存在如下两个环境变量
#
# PROJECT_DIR: 项目目录 (例如: ~/projname)
# ADMIN_PASSWORD: 管理员用户的密码

import os
import sys

# 将项目目录添加到 PATH 系统环境变量中
proj_dir = os.path.expanduser(os.environ['PROJECT_DIR'])
sys.path.append(proj_dir)

os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'

from mezzanine.utils.models import get_user_model
User = get_user_model()
u, _ = User.objects.get_or_create(username='admin')
u.is_staff = u.is_superuser = True
u.set_password(os.environ['ADMIN_PASSWORD'])
u.save()
```

110

设置服务的配置文件

下一步,我们为gunicorn(我们的应用服务器)、nginx(我们的Web服务器)和supervisor(我们的进程管理器)设置配置文件,如例6-19所示,gunicorn配置文件的模板如例6-21所示。

例6-19 设置配置文件

```
- name: set the gunicorn config file
  template: src=templates/gunicorn.conf.py.j2 dest={{ proj_path }}/gunicorn.conf.py

- name: set the supervisor config file
  template: src=templates/supervisor.conf.j2
  dest=/etc/supervisor/conf.d/mezzanine.conf
  sudo: True
  notify: restart supervisor

- name: set the nginx config file
  template: src=templates/nginx.conf.j2
  dest=/etc/nginx/sites-available/mezzanine.conf
  notify: restart nginx
  sudo: True
```

对于所有这三种情况，我们都使用模板来生成配置文件。supervisor 和 nginx 的进程是使用 root 启动的（尽管它们在运行的时候会降权为非 root 用户），所以我们需要 sudo 以便我们拥有足够的权限对它们的配置文件可写。

如果 supervisor 配置文件变更了，那么 Ansible 将会触发 restart supervisor handler。如果 nginx 配置文件变更了，那么 Ansible 将会触发 restart nginx handler。这两个 handler 如例 6-20 所示。

例 6-20 两个 handler

```
handlers:  
  - name: restart supervisor  
    supervisorctl: name=gunicorn_mezzanine state=restarted  
    sudo: True  
  
  - name: restart nginx  
    service: name=nginx state=restarted  
    sudo: True
```

例 6-21 templates/gunicorn.conf.py.j2

```
from __future__ import unicode_literals  
import multiprocessing  
  
bind = "127.0.0.1:{{ gunicorn_port }}"  
workers = multiprocessing.cpu_count() * 2 + 1  
loglevel = "error"  
proc_name = "{{ proj_name }}"
```

111

例 6-22 templates/supervisor.conf.j2

```
[group: {{proj_name}}]  
programs=gunicorn_{{ proj_name }}  
  
[program:gunicorn_{{ proj_name }}]  
command={{ venv_path }}/bin/gunicorn_django -c gunicorn.conf.py -p gunicorn.pid  
directory={{ proj_path }}  
user={{ user }}  
autostart=true  
autorestart=true  
redirect_stderr=true  
environment=LANG="{{ locale }}",LC_ALL="{{ locale }}",LC_LANG="{{ locale }}"
```

只有 nginx 配置文件的模板具有模板逻辑（除了变量替代以外），如例 6-23 所示。它具有条件逻辑：在 `tls_enabled` 变量设置为 `true` 的时候启用 TLS。你将会看到 `if` 语句像这样分散在模板中的各处：

```
{% if tls_enabled %}
```

```
...
{%- endif %}
```

nginx 配置文件的模板还使用了 Jinja2 的 join 过滤器 (filter) :

```
server_name {{ domains|join(",") }};
```

这段代码片段要求 domains 变量是一个列表。它将会把 domains 的成员连接在一起生成一个字符串，并由逗号分隔开。回忆一下在我们的范例中，domains 列表被定义为：

```
domains:
- 192.168.33.10.xip.io
- www.192.168.33.10.xip.io
```

当模板渲染后，本行将会变成如下这样：

```
server_name 192.168.33.10.xip.io, www.192.168.33.10.xip.io;
```

例6-23 templates/nginx.conf.j2

```
upstream {{ proj_name }} {
    server 127.0.0.1:{{ gunicorn_port }};
}

112 > server {
    listen 80;

    {% if tsl_enabled %}
    listen 443 ssl;
    {% endif %}
    server_name {{ domains|join(", ") }};
    client_max_body_size 10M;
    keepalive_timeout 15;

    {% if tsl_enabled %}
    ssl_certificate      conf/{{ proj_name }}.crt;
    ssl_certificate_key  conf/{{ proj_name }}.key;
    ssl_session_cache    shared:SSL:10m;
    ssl_session_timeout  10m;
    # 由于 ssl_ciphers 条目太长，不便于在书中展示。请到下面地址中查看完整文件
    # https://github.com/lorin/ansiblebook/ch06/playbooks/templates/nginx.conf.j2
    {% endif %}

    location / {
        proxy_redirect    off;
        proxy_set_header Host                      $host;
        proxy_set_header X-Real-IP                 $remote_addr;
        proxy_set_header X-Forwarded-For           $proxy_add_x_forwarded_for;
    }
}
```

```

proxy_set_header X-Forwarded-Protocol $scheme;
proxy_pass      http://{{ proj_name }};
}

location /static/ {
    root          {{ proj_path }};
    access_log    off;
    log_not_found off;
}

location /robots.txt {
    root          {{ proj_path }}/static;
    access_log    off;
    log_not_found off;
}

location /favicon.ico {
    root          {{ proj_path }}/static/img;
    access_log    off;
    log_not_found off;
}
}

```

启用 nginx 配置文件

113

nginx 配置文件的惯例是将你的配置文件放到 `/etc/nginx/sites-available` 中，并通过将它们符号连接（Symlink）到 `/etc/nginx/sites-enabled` 下来启用它们。

Mezzanine 的 Fabric 脚本将配置文件直接复制到 `sites-enabled` 下。但是我打算违背 Mezzanine 的做法，因为这将会给我创造一个使用 `file` 模块创建符号链接的机会。我们还需要删除 nginx 软件包创建在 `/etc/nginx/sites-enabled/default` 下的默认配置文件。

如例 6-24 所示，我们使用 `file` 模块创建符号链接并删除默认配置文件。这个模块可以用于创建目录、符号链接和空文件，删除文件、目录和符号链接，以及设置诸如权限和所属关系等属性。

例6-24 启用nginx配置文件

```

- name: enable the nginx config file
  file:
    src: /etc/nginx/sites-available/mezzanine.conf
    dest: /etc/nginx/sites-enabled/mezzanine.conf
    state: link
  notify: restart nginx
  sudo: True

```

```
- name: remove the default nginx config file
  file: path=/etc/nginx/sites-enabled/default state=absent
  notify: restart nginx
  sudo: True
```

安装 TLS 证书

我们的 playbook 定义了一个名为 `tls_enabled` 的变量。如果这个变量设置为 `true`，那么 playbook 将会安装 TLS 证书。在我们范例中，我们使用自签发证书。所以如果证书不存在，playbook 可以创建它。

在向生产环境部署时，你需要复制从证书权威机构获得的证书。

例 6-25 中列出了两个参与到配置 TLS 证书的 task。我们使用 `file` 模块来确保需要放置 TLS 证书的目录是存在的。

例 6-25 安装 TLS 证书

```
- name: ensure config path exists
  file: path={{ conf_path }} state=directory
  sudo: True
  when: ssl_enabled
- name: create ssl certificates
  command: >
    openssl req -new -x509 -nodes -out {{ proj_name }}.crt
    -keyout {{ proj_name }}.key -subj '/CN={{ domains[0] }}' -days 3650
    chdir={{ conf_path }}
    creates={{ conf_path }}/{{ proj_name }}.crt
  sudo: True
  when: ssl_enabled
  notify: restart nginx
```

需要注意的是，两个任务都包含了如下语句：

```
when: tls_enabled
```

如果 `tls_enabled` 的值是 `false`，那么 Ansible 将会跳过这个任务。

Ansible 并没有随附创建 TLS 证书的模块，所以为了创建自签发证书，我们需要使用 `command` 模块来调用 `openssl` 命令。由于命令非常长，我们使用 YAML 折行语法（详见第 30 页中的“折行”）来将命令分割为多行。

在命令的最后两行是传递给模块的附加参数。它们不是传递给命令行的。`chdir` 参数会在命令运行前改变所在目录。`creates` 参数则实现幂等性：Ansible 将会预先检查

`conf_path` } } / { { proj_name } }.crt 文件是否在主机上存在。如果文件已经存在，Ansible 将会跳过这个 task。

```
chdir={{ conf_path }}  
creates={{ conf_path }}/{{ proj_name }}.crt
```

安装 Twitter 计划任务

如果运行 `manage.py poll_twitter`，那么 Mezzanine 将会获取所配置账户的推文^{注4} 并将它们展示在首页上。

Mezzanine 随附的 Fabric 脚本通过安装一个每五分钟运行一次的计划任务来保证这些推文是实时更新的。

如果我们完全遵循 Fabric 的做法，我们需要将计划任务的脚本复制到 `/etc/cron.d` 目录下。我们可以利用 `template` 模块来实现这个行为。然而，Ansible 随附了 `cron` 模块来供我们创建和删除计划任务，我觉得这个方法更为优雅与简洁。例 6-26 演示了安装这个计划任务的 task。

例 6-26 安装用于从 Twitter 拉取推文的计划任务

115

```
- name: install poll twitter cron job  
cron: name="poll twitter" minute="*/5" user={{ user }} job="{{ manage }}  
poll_twitter"
```

如果你手动 SSH 到 Vagrant 虚拟机，并执行列出计划任务的命令 `crontab -l`，你将会看到相应的计划任务已经被安装了。当我使用 Vagrant 用户部署时，在我的 Vagrant 虚拟机上看到的计划任务如下所示：

```
#Ansible: poll twitter  
*/5 * * * * /home/vagrant/mezzanine-example/bin/python /home/vagrant/  
mezzanine-example/project/manage.py poll_twitter
```

特别需要注意的是第一行的注释，这是 Ansible 模块用于按照名字删除计划任务所使用的。如果你执行如下操作：

```
- name: remove cron job  
cron: name="poll twitter" state=absent
```

`cron` 模块将会寻找匹配名字的注释行并连带注释一起删除相应的计划任务。

注 4：Twitter 中将发表的内容称作 Tweet。在国内称作“推文”或者“推”。——译者注

playbook 全文

例 6-27 列出了历经众多考验的完整 playbook。

例 6-27 mezzanine.yml：完整的 playbook

```
---
```

```
- name: Deploy mezzanine
  hosts: web
  vars:
    user: "{{ ansible_ssh_user }}"
    proj_name: mezzanine-example
    venv_home: "{{ ansible_env.HOME }}"
    venv_path: "{{ venv_home }}/{{ proj_name }}"
    proj_dirname: project
    proj_path: "{{ venv_path }}/{{ proj_dirname }}"
    reqs_path: requirements.txt
    manage: "{{ python }} {{ proj_path }}/manage.py"
    live_hostname: 192.168.33.10.xip.io
    domains:
      - 192.168.33.10.xip.io
      - www.192.168.33.10.xip.io
    repo_url: git@github.com:lorin/mezzanine-example.git
    gunicorn_port: 8000
    locale: en_US.UTF-8
    # Mezzanine 中的 Fabric 脚本中并不包含下面的变量 y
    # 我在这里定义它们是为了方便起见
    conf_path: /etc/nginx/conf
    tsl_enabled: True
    python: "{{ venv_path }}/bin/python"
    database_name: "{{ proj_name }}"
    database_user: "{{ proj_name }}"
    database_host: localhost
    database_port: 5432
    gunicorn_proc_name: mezzanine
  vars_files:
    - secrets.yml
  tasks:
    - name: install apt packages
      apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
      sudo: True
      with_items:
        - git
        - libjpeg-dev
        - libpq-dev
        - memcached
```

```
- nginx
- postgresql
- python-dev
- python-pip
- python-psycopg2
- python-setuptools
- python-virtualenv
- supervisor

- name: check out the repository on the host
  git: repo={{ repo_url }} dest={{ proj_path }} accept_hostkey=yes

- name: install required python packages
  pip: name={{ item }} virtualenv={{ venv_path }}
  with_items:
    - gunicorn
    - setproctitle
    - south
    - psycopg2
    - django-compressor
    - python-memcached
- name: install requirements.txt
  pip: requirements={{ proj_path }}/{{ reqs_path }} virtualenv={{ venv_path }}

- name: create a user
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"
  sudo: True
  sudo_user: postgres

- name: create the database
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"
    encoding: UTF8
    lc_ctype: "{{ locale }}"
    lc_collate: "{{ locale }}"
    template: template0
  sudo: True
  sudo_user: postgres

- name: generate the settings file
  template:
    src:templates/local_settings.py.j2
    dest:"{{ proj_path }}/local_settings.py"
```

```
- name: sync the database, apply migrations, collect static content
  django_manage:
    command: "{{ item }}"
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
  with_items:
    - syncdb
    - migrate
    - collectstatic

- name: set the site id
  script: scripts/setsite.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    WEBSITE_DOMAIN: "{{ live_hostname }}"

- name: set the admin password
  script: scripts/setadmin.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    ADMIN_PASSWORD: "{{ admin_pass }}"

- name: set the gunicorn config file
  template:
    src:templates/gunicorn.conf.py.j2
    dest:"{{ proj_path }}/{gunicorn}.conf.py"

- name: set the supervisor config file
  template:
    src:templates/supervisor.conf.j2
    dest:/etc/supervisor/conf.d/mezzanine.conf
    sudo: True
  notify: restart supervisor

- name: set the nginx config file
  template:
    src:templates/nginx.conf.j2
    dest:/etc/nginx/sites-available/mezzanine.conf
  notify: restart nginx
  sudo: True

- name: enable the nginx config file
  file:
    src: /etc/nginx/sites-available/mezzanine.conf
    dest: /etc/nginx/sites-enabled/mezzanine.conf
```

```

    state: link
notify: restart nginx
sudo: True

- name: remove the default nginx config file
  file: path=/etc/nginx/sites-enabled/default state=absent
  notify: restart nginx
  sudo: True

- name: ensure config path exists
  file: path={{ conf_path }} state=directory
  sudo: True
  when: tsl_enabled

- name: create ssl certificates
  command: >
    openssl req -new -x509 -nodes -out {{ proj_name }}.crt
    -keyout {{ proj_name }}.key -subj '/CN={{ domains[0] }}' -days 3650
    chdir={{ conf_path }}
    creates={{ conf_path }}/{{ proj_name }}.crt
  sudo: True
  when: tsl_enabled
  notify: restart nginx

- name: install poll twitter cron job
  cron: name="poll twitter" minute="*/5" user={{ user }}
    job="{{ manage }} poll_twitter"

handlers:
- name: restart supervisor
  supervisorctl: name=gunicorn_mezzanine state=restarted
  sudo: True

- name: restart nginx
  service: name=nginx state=restarted
  sudo: True

```

在 Vagrant 虚拟机上运行 playbook

我们 playbook 中的 `live_hostname` 变量和 `domains` 变量假定了我们将要部署的主机可以通过 `192.168.33.10` 访问。如例 6-28 所示的 `Vagrantfile` 会为 Vagrant 虚拟机配置那个 IP 地址。

例6-28 Vagrantfile

```

VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

```

```
config.vm.box = "ubuntu/trusty64"
config.vm.network "private_network", ip: "192.168.33.10"
end
```

将 Mezzanine 部署到 Vagrant 虚拟机：

```
$ ansible-playbook mezzanine.yml
```

然后，你就可以通过如下 URL 中任意一个来访问崭新部署的 Mezzanine 网站了：

- *http://192.168.33.10.xip.io*
- *https://192.168.33.10.xip.io*
- *http://www.192.168.33.10.xip.io*
- *https://www.192.168.33.10.xip.io*

将 Mezzanine 部署到多台主机

在本章中，我们已经将 Mezzanine 完整地部署在单台主机上了。然而，实际情况下往往需要将数据库服务和 Web 服务分别部署到独立的主机上。在第 8 章中我们将会演示一个将数据库服务和 Web 服务部署在各自独立主机上的 playbook。

现在，你已经看到了部署一个使用 Mezzanine 的真实应用是什么样子了。在下一章，我们将会讨论一些本章范例未涉及的 Ansible 的高级功能。

复杂playbook

在上一章，我们从头到尾地编写了一个用于部署 Mezzanine CMS 系统的全功能的 Ansible playbook。这个范例使用了很多 Ansible 常见特性，但是并没有覆盖到全部特性。本章将会触及那些附加特性。正是这些特性使得 Ansible 有点像多啦 A 梦的四次元口袋的感觉。

在控制主机上运行 task

有时候，你会希望在控制主机（而不是远程主机）上运行一个特定的 task。Ansible 为 task 提供了 `local_action` 语句以支持这种使用方式。

假设我们打算安装 Mezzanine 的服务器才刚刚启动，如果我们过早运行 playbook，则会因为服务器还没有彻底启动完毕而导致报错。

我们可以将 `wait_for` 模块调用作为我们 playbook 的开始，使得在执行剩下的 playbook 之前先等待 SSH 服务器做好接受连接的准备。在这种情况下，我们希望这个模块在我们的笔记本电脑上执行，而不是远程主机上执行。

这个 playbook 的第一个 task 需要以此开始：

```
- name: wait for ssh server to be running
  local_action: wait_for port=22 host="{{ inventory_hostname }}"
    search_regex=OpenSSH
```

特别要注意的是，我们在这个 task 中引用的变量 `inventory_hostname`。该变量的值是远程主机的名字而不是 `localhost`。这是因为这些变量的范围仍然是远程主机，即使 task 正在本地执行。



如果你的 play 涉及多个远程主机，并且你使用了 `local_action` 模块，那么这个 task 将会被执行多次，对每台远程主机执行一次。你可以使用 `run_once` 来限制这个行为。`run_once` 将会在第 126 页的“只执行一次”中详细讨论。

在涉及的主机以外的机器上运行 task

有时候我们会希望运行与某主机相关联的 task，但是希望在另一台服务器上执行这个 task。这时可以使用 `delegate_to` 语句来在另一台主机上运行 task。

这个特性主要用于两个典型场景：

- 在报警系统中启用基于主机的报警，例如 Nagios。
- 向负载均衡器中添加一台主机，例如 HAProxy。

例如，我们想要为所有 web 群组的主机启动 Nagios 报警。假设在我们的 inventory 中有一个主机名为 `nagios.example.com` 的服务器运行着 Nagios。例 7-1 列出了一个使用 `delegate_to` 来完成这个操作的范例。

例 7-1 使用 `delegate_to` 配置 Nagios

```
- name: enable alerts for web servers
  hosts: web
  tasks:
    - name: enable alerts
      nagios: action=enable_alerts service=web host={{ inventory_hostname }}
      delegate_to: nagios.example.com
```

在这个范例中，Ansible 将会在 `nagios.example.com` 上执行这个 `nagios` task，但是在 play 中所引用的 `inventory_hostname` 变量的值是 `web` 主机。

如果想查看一个更详细的使用 `delegate_to` 的范例，可以到 Ansible 项目的范例 GitHub 仓库 (<https://github.com/ansible/ansible-examples>) 中查阅 `lamp_haproxy/rolling_update.yml`。

手动采集 fact

在我们开始运行 playbook 的时候，如果 SSH 服务器有可能还没有开始运行，那么我们就需要关闭显式的 fact 采集 (gathering)。否则，Ansible 会在运行第一个 task 之前尝试 SSH 到主机上采集 fact。由于我们仍然需要访问 fact (回忆一下，我们曾在 playbook 中用到了 `ansible_env fact`)，我们可以直接调用 `setup` 模块来让 Ansible 采集 fact，如例

7-2 所示。

123

例7-2 等待SSH服务器启动

```
- name: Deploy mezzanine
hosts: web
gather_facts: False
# vars 区段与 vars_files 区段没有在本例中列出
tasks:
  - name: wait for ssh server to be running
    local_action: wait_for port=22 host="{{ inventory_hostname }}"
      search_regex=OpenSSH

  - name: gather facts
    setup:
# 其余的 task 都放到这里
```

逐台主机运行

默认情况下，Ansible 会并行地在所有相关联主机上执行每一个 task。有时候你会希望逐台主机地执行 task。最典型的例子就是对负载均衡后面的应用服务器进行升级。通常来说，你会将应用服务器从负载均衡上摘除，更新它，然后再添加回去。但是你肯定不希望同时将所有应用服务器都从负载均衡上摘除，那样的话你的服务将会不可用。

你可以在 play 中使用 `serial` 语句告诉 Ansible 限制并行执行 play 的主机数量。例 7-3 列出了一个范例。在这个范例中，Ansible 会逐台从 Amazon EC2 的弹性负载均衡中移除主机，更新系统软件包，然后再添加回负载均衡（我们将在第 12 章详细讨论 Amazon EC2）。

例7-3 从负载均衡中移除主机并更新软件包

```
- name: upgrade packages on servers behind load balancer
hosts: myhosts
serial: 1
tasks:
  - name: get the ec2 instance id and elastic load balancer id
    ec2_facts:

  - name: take the host out of the elastic load balancer
    local_action: ec2_elb
    args:
      instance_id: "{{ ansible_ec2_instance_id }}"
      state: absent
```

```
- name: upgrade packages
  apt: update_cache=yes upgrade=yes

124 > - name: put the host back in the elastic load balancer
      local_action: ec2_elb
      args:
        instance_id: "{{ ansible_ec2_instance_id }}"
        state: present
        ec2_elbs: "{{ item }}"
      with_items: ec2_elbs
```

在我们的范例中，我们将 1 作为 serial 语句的参数传入，来告诉 Ansible 某时间段只在一台主机执行。如果我们传入 2，Ansible 将会同时在两台主机上执行。

一般来说，当 task 失败时，Ansible 会停止执行失败的那台主机上的任务，但是继续对其他主机执行。在负载均衡的场景中，你可能希望 Ansible 在所有主机都发生失败前让整个 play 都停止。否则，你将会面临所有主机都从负载均衡上摘除，并且全部执行失败，最终负载均衡上没有任何主机的局面。

你可以同时使用 serial 语句和 max_fail_percentage 语句来指定，最大失败主机比例达多少时 Ansible 就让整个 play 失败。例如，假设我们按如下所示配置指定最大失败比例是 25%：

```
- name: upgrade packages on servers behind load balancer
  hosts: myhosts
  serial: 1
  max_fail_percentage: 25
  tasks:
    # 具体 task 内容省略
```

如果我们在负载均衡后面有四台主机，并且有一台主机执行 task 失败，那么 Ansible 还将继续执行 play，因为还没有超过 25% 的阈值。然而，如果有第二台主机执行 task 失败了，Ansible 将会让整个 play 失败，因为已经有 50% 的主机执行 task 失败，超过了 25% 的阈值。如果你希望 Ansible 在任何主机出现 task 执行失败的时候都放弃执行，则需要将 max_fail_percentage 设置为 0。

只执行一次

有时候你可能想要某个 task 只执行一次，即使有多台主机与它相关联。例如，假设你有多台应用服务器运行在负载均衡后面，你希望执行一个数据库迁移，而且只需要在一台应用服务器上执行这个迁移。

你可以使用 `run_once` 语句来告诉 Ansible 只运行这个命令一次。

```
- name: run the database migrations
  command: /opt/run_migrations
  run_once: true
```

当你在涉及多台主机的 playbook 中使用 `local_action`, 并且想让本地 task 只执行一次 125 的时候, `run_once` 就会特别好用。

```
- name: run the task locally, only once
  local_action: command /opt/my-custom-command
  run_once: true
```

处理不利行为命令 : `changed_when` 和 `failed_when`

回忆一下第 6 章中, 我们避开调用自定义的 `createdb manage.py` 命令, 如例 7-4 所示, 因为这个调用不是幂等性的。

例7-4 调用django manage.py createdb

```
- name: initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
```

我们通过调用几个幂等性且作用与 `createdb` 相同的 `django manage.py` 命令来绕过这个问题。但是如果没有任何可以调用等同命令的模块又该怎么办呢? 答案就是使用 `changed_when` 语句和 `failed_when` 语句来改变 Ansible 对 task 是 `changed` 状态还是 `failed` 状态的认定。

首先, 我们需要了解该命令第一次运行时输出的样子, 以及它第二次运行时输出的样子。

回忆一下第 4 章中关于获取失败 task 输出的内容, 你添加一个 `register` 语句来将输出保存到一个变量, 以及一个 `failed_when: False` 语句, 以使得即便在模块返回失败的时候, task 的执行也不会被停止。然后添加一个 `debug` task 来输出变量, 并在最后使用 `fail` 语句使得 playbook 停止执行, 如例 7-5 所示。

例7-5 查看任务的输出

```
- name: initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
```

```
failed_when: False
register: result
- debug: var=result
- fail:
```

第二次调用时，这个 playbook 的输出如例 7-6 所示。

例7-6 当数据库已经被创建时的返回值

```
TASK: [debug var=result] ****
ok: [default] => {
    "result": {
        "cmd": "python manage.py createdb --noinput --nodata",
        "failed": false,
        "failed_when_result": false,
        "invocation": {
            "module_args": '',
            "module_name": "django_manage"
        },
        "msg": "\nstderr: CommandError: Database already created, you probably
want the syncdb or migrate command\n",
        "path":
        "/home/vagrant/mezzanine_example/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games",
        "state": "absent",
        "syspath": [
            '/',
            "/usr/lib/python2.7",
            "/usr/lib/python2.7/plat-x86_64-linux-gnu",
            "/usr/lib/python2.7/lib-tk",
            "/usr/lib/python2.7/lib-old",
            "/usr/lib/python2.7/lib-dynload",
            "/usr/local/lib/python2.7/dist-packages",
            "/usr/lib/python2.7/dist-packages"
        ]
    }
}
```

这就是任务多次运行时所发生的事。想要看到第一次运行发生了什么的话，只需要删除数据库然后执行 playbook 来重新创建它。最简单的办法就是使用 Ansible ad-Hoc 任务来删除数据库：

```
$ ansible default --sudo --sudo-user postgres -m postgresql_db -a \
"name=mezzanine_example state=absent"
```

现在我再次运行 playbook 的时候，输出将如例 7-7 所示。

例7-7 初次调用时的返回值

```
ASK: [debug var=result] ****
ok: [default] => {
    "result": {
        "app_path": "/home/vagrant/mezzanine_example/project",
        "changed": false,
        "cmd": "python manage.py createdb --noinput --nodata",
        "failed": false,
        "failed_when_result": false,
        "invocation": {
            "module_args": '',
            "module_name": "django_manage"
        },
        "out": "Creating tables ...\\nCreating table auth_permission\\nCreating table auth_group_permissions\\nCreating table auth_group\\nCreating table auth_user_groups\\nCreating table auth_user_user_permissions\\nCreating table auth_user\\nCreating table django_content_type\\nCreating table django_redirect\\nCreating table django_session\\nCreating table django_site\\nCreating table conf_setting\\nCreating table core_sitepermission_sites\\nCreating table core_sitepermission\\nCreating table generic_threadedcomment\\nCreating table generic_keyword\\nCreating table generic_assignedkeyword\\nCreating table generic_rating\\nCreating table blog_blogpost_related_posts\\nCreating table blog_blogpost_categories\\nCreating table blog_blogpost\\nCreating table blog_blogcategory\\nCreating table forms_form\\nCreating table forms_field\\nCreating table forms_formentry\\nCreating table forms_fieldentry\\nCreating table pages_page\\nCreating table pages_richtextpage\\nCreating table pages_link\\nCreating table galleries_gallery\\nCreating table galleries_galleryimage\\nCreating table twitter_query\\nCreating table twitter_tweet\\nCreating table south_migrationhistory\\nCreating table django_admin_log\\nCreating table django_comments\\nCreating table django_comment_flags\\n\\nCreating default site record: vagrant-ubuntu-trusty-64 ... \\n\\nInstalled 2 object(s) from 1 fixture(s)\\nInstalling custom SQL ...\\nInstalling indexes ...\\nInstalled 0 object(s) from 0 fixture(s)\\n\\nFaking initial migrations ...\\n\\n",
        "pythonpath": null,
        "settings": null,
        "virtualenv": "/home/vagrant/mezzanine_example"
    }
}
```

127

注意尽管确实改变了数据库的状态，changed 的值仍被设为 false。这是因为当运行的命令并不为模块熟知时，django_manage 模块总是返回 changed=false。

我们可以添加 changed_when 语句在 out 返回值中寻找 "Creating tables"，如例 7-8 所示。

例7-8 初次尝试添加changed_when

```
- name: initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
  register: result
  changed_when: '"Creating tables" in result.out'
```

128 这个方法的问题是：如果我们回顾例 7-6，我们会发现并没有 out 变量，而是会有 msg 变量。这意味着如果我们第二次执行这个 playbook，我们将会得到如下（不怎么有帮助的）错误：

```
TASK: [initialize the database] *****
fatal: [default] => error while evaluating conditional: "Creating tables" in
result.out
```

我们需要确保 Ansible 只有在 result.out 变量被定义的情况下才去获取它的值。一种方法是直接对变量是否定义进行检测：

```
changed_when: result.out is defined and "Creating tables" not in result.out
```

或者，如果 result.out 变量不存在，我们也可以使用 Jinja2 的 default 过滤器为 result.out 提供一个默认值：

```
changed_when: '"Creating tables" not in result.out|default("")'
```

又或者，我们可以通过简单地检测 failed 是否为 false：

```
changed_when: not result.failed and "Creating tables" not in result.out
```

我们还需要改变失败时的行为，因为我们并不希望 Ansible 仅仅因为 createdb 已经被调用过就认为任务执行失败了：

```
failed_when: result.failed and "Database already created" not in result.msg
```

这里的 failed 检测是 msg 变量存在的保障。最终幂等性 task 如例 7-9 所示。

例7-9 幂等性的manage.py createdb

```
- name: initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
  register: result
```

```
changed_when: not result.failed and "Creating tables" in result.out
failed_when: result.failed and "Database already created" not in result.msg
```

从主机获取 IP 地址

在我们的 playbook 中，我们用到的几个主机名源于 Web 服务器的 IP 地址。

```
live_hostname: 192.168.33.10.xip.io
domains:
  - 192.168.33.10.xip.io
  - www.192.168.33.10.xip.io
```

如果我们想要使用相同的模式，但不将 IP 地址硬编码到变量中又该如何做呢？这样一来，◀129如果 Web 服务器的 IP 地址变更了，我们就不需要修改我们的 playbook 了。

Ansible 获取每一台主机的 IP 地址并保存为 fact。每个接口具有一个关联的 Ansible fact。例如，有关接口 eth0 的详细信息存储在 `ansible_eth0 fact` 中，例 7-10 列出了这个例子。

例 7-10 `ansible_eth0 fact`

```
"ansible_eth0": {
    "active": true,
    "device": "eth0",
    "ipv4": {
        "address": "10.0.2.15",
        "netmask": "255.255.255.0",
        "network": "10.0.2.0"
    },
    "ipv6": [
        {
            "address": "fe80::a00:27ff:fefe:1e4d",
            "prefix": "64",
            "scope": "link"
        }
    ],
    "macaddress": "08:00:27:fe:1e:4d",
    "module": "e1000",
    "mtu": 1500,
    "promisc": false,
    "type": "ether"
}
```

我们的 Vagrant 虚拟机有两个接口，eth0 和 eth1。eth0 接口是一个私有接口，它的 IP 地址（10.0.2.15）并不可达。eth1 接口是我们在 `Vagrantfile` 中指定 IP 地址（192.168.33.10）的那个。

于是，我们可以像这样定义我们的变量：

```
live_hostname: "{{ ansible_eth1.ipv4.address }}.xip.io"
domains:
  - ansible_eth1.ipv4.address.xip.io
  - wwwansible_eth1.ipv4.address.xip.io
```

使用 Vault 加密敏感数据

我们的 Mezzanine 应用的 playbook 需要访问一些敏感信息，例如数据库和管理员的密码。
130> 在第 6 章中，我们通过这样的方法解决过这个问题：将这些敏感信息存储在一个名为 *secrets.yml* 的独立文件，并确保不会将这个文件推送到版本控制仓库中。

Ansible 提供了另外一种解决方法：我们可以向版本控制仓库提交一个 *secrets.yml* 文件的加密版本，而不是将其放在版本控制仓库之外。使用这种方法，即使我们的版本控制仓库被攻破，攻击者仍然无法访问 *secrets.yml* 文件的内容，除非他同时拿到了用于加密的密码。

`ansible-vault` 命令行工具允许你创建和编辑加密文件，`ansible-playbook` 可以自动识别并使用密码解密这些文件。

我们可以像这样加密一个已存在的文件：

```
$ ansible-vault encrypt secrets.yml
```

同样的，我们可以使用如下命令新建一个名为 *secrets.yml* 的加密文件：

```
$ ansible-vault create secrets.yml
```

你将被提示输入密码，然后 `ansible-vault` 将会启动一个文本编辑器以便于你编辑这个文件。它运行的编辑器是由 `$EDITOR` 环境变量指定的。如果这个变量未被定义，它将默认使用 `vim`。

例 7-11 所示为使用 `ansible-vault` 加密的文件内容的一个范例。

例 7-11 使用 `ansible-vault` 加密的文件的内容

```
$ANSIBLE_VAULT;1.1;AES256
34306434353230663665633539363736353836333936383931316434343030316366653331363262
6630633366383135386266333030393634303664613662350a623837663462393031626233376232
31613735376632333231626661663766626239333738356532393162303863393033303666383530
...
623466333434643133303838326465316233386334383364653231666263356236393833643438
6463666536653834303838303165646161366566326563306639643833165653436
```

你可以使用 play 中的 `vars_files` 区段像一般文件一样引用一个由 `ansible-vault` 加密的文件：如果我们加密 `secrets.yml` 文件，我们根本不需要修改例 6-27 中的 playbook。

我们唯一需要做的是要求 `ansible-playbook` 提示我们输入加密文件使用的密码，否则它会直接报错退出。只需要使用 `--ask-vault-pass` 参数就可以做到：

```
$ ansible-playbook mezzanine.yml --ask-vault-pass
```

你也可以将密码存储在文本文件中，并通过 `--vault-password-file` 参数告诉 `ansible-playbook` 密码文件的所在位置。

```
$ ansible-playbook mezzanine --vault-password-file ~/password.txt
```

如果 `--vault-password-file` 参数所指向的文件权限被设置为可执行，Ansible 将会执行这个文件并使用它的标准输出（Standard out）内容作为密码。这个特性允许你使用脚本向 Ansible 提供密码。◀ 131

表 7-1 列出了常用的 `ansible-vault` 命令

表7-1 ansible-vault命令

命令	描述
<code>ansible-vault encrypt file.yml</code>	加密明文文件 <code>file.yml</code>
<code>ansible-vault decrypt file.yml</code>	解密已加密的文件 <code>file.yml</code>
<code>ansible-vault view file.yml</code>	打印已加密文件 <code>file.yml</code> 的内容
<code>ansible-vault create file.yml</code>	创建新的加密文件 <code>file.yml</code>
<code>ansible-vault edit file.yml</code>	编辑已加密的文件 <code>file.yml</code>
<code>ansible-vault rekey file.yml</code>	变更已加密的文件 <code>file.yml</code> 的密码

通过模式匹配指定主机

目前为止，我们在 play 中的 `host` 参数中已经指定过单台主机或者群组了，比如：

```
hosts: web
```

除了单台主机和群组，你还可以指定模式匹配。截至目前，我们已经见过了会让 play 在所有主机上运行的 `all` 模式：

```
hosts: all
```

你可以使用 : 来指定两个群组的并集。比如我们想要指定所有 dev 群组和 staging 群组的主机：

```
hosts: dev:staging
```

你可以使用 :& 来指定群组之间的交集。例如，想要指定所有仿真环境中的数据库服务器，你可以这样做：

```
hosts: staging:&database
```

表 7-2 列出了 Ansible 支持的所有模式匹配。需要注意的是，正则表达式永远以 ~ 开头。

132 表7-2 Ansible 所支持的匹配模式

匹配行为	用法示例
所有主机	all
所有主机	*
并集	dev:staging
交集	staging:&database
排除	dev:!queue
通配符	*.example.com
数字范围	web[5:10]
正则表达式	~web\d\.example\.(com)

Ansible 支持多种匹配模式的组合使用——例如：

```
hosts: dev:staging:&database:!queue
```

限定某些主机运行

使用 -l hosts 或者 --limit hosts 参数会告诉 Ansible 只针对限定的主机运行 playbook，如例 7-12 所示。

例7-12 限制某些主机运行

```
$ ansible-playbook -l hosts playbook.yml  
$ ansible-playbook --limit hosts playbook.yml
```

你可以使用模式匹配语法来描述任意指定的主机组合，例如：

```
$ ansible-playbook -l 'staging:&database' playbook.yml
```

过滤器

过滤器是 Jinja2 模板引擎的特性。Ansible 除了使用 Jinja2 作为模板之外，还将其用于变量求值，所以除了模板之外，你也可以在 playbook 中在 {{ }} 内使用过滤器。过滤器的用法有点像 Unix 管道，可以想象为变量像通过管道一样通过过滤器。Jinja2 随附了一系列的内置过滤器 (<http://jinja.pocoo.org/docs/dev/templates/#builtin-filters>)。除此之外，Ansible 随附了它自己的过滤器来扩展 Jinja2 的过滤器 (http://docs.ansible.com/ansible/playbooks_variables.html#jinja2-filters)。

我们在这里只是涵盖了一个示例过滤器，要获取完整的过滤器列表，请查看官方的 Jinja2 和 Ansible 文档。

◀133

default 过滤器

default 过滤器非常有用。下面是一个 default 过滤器的实际范例：

```
"HOST": "{{ database_host | default('localhost') }}",
```

如果变量 database_host 已被定义，那么括号将会直接获取这个变量的值。相反，如果变量 database_host 没被定义，那么括号将取值为字符串 localhost。有些过滤器需要参数，有些则不需要。

我们在这里只是涵盖了一个示例过滤器，要获取完整的过滤器列表，请查看官方的 Jinja2 和 Ansible 文档。

用于注册变量的过滤器

比如说我们希望即便在某个 task 失败的情况下，也能继续运行这个 task 并打印它的输出。但是，如果 task 失败了，我们还是希望 Ansible 在我们打印输出后对那个主机失败。我们可以如例 7-13 所示将 failed 过滤器作为 failed_when 语句的参数。

例7-13 使用failed过滤器

```
- name: Run myprog
  command: /opt/myprog
  register: result
  ignore_errors: True
- debug: var=result
```

```
- debug: msg="Stop running the playbook if myprog failed"
  failed_when: result|failed
# 具体 task 内容省略
```

表 7-3 列出了一系列我们可以对注册变量检查状态的过滤器。

表7-3 任务返回值过滤器

名称	描述
failed	如果注册变量的值是任务 failed 则返回 True
changed	如果注册变量的值是任务 changed 则返回 True
success	如果注册变量的值是任务 succeeded 则返回 True
skipped	如果注册变量的值是任务 skipped 则返回 True

134 应用于文件路径的过滤器

表 7-4 列出了一些过滤器。这些过滤器在处理包含控制主机文件系统的路径的变量时十分有用。

表7-4 文件路径过滤器

basename	文件路径的文件名部分
dirname	文件路径中的目录
expanduser	将文件路径中的 ~ 替换为用户目录
realpath	处理符号链接后的文件实际路径

思考一下这段 playbook 片段：

```
vars:
  homepage: /usr/share/nginx/html/index.html
tasks:
- name: copy home page
  copy: src=files/index.html dest={{ homepage }}
```

留意一下这里引用了两次 *index.html*。第一次是定义 *homepage* 变量时；第二次是指定控制主机上文件的路径时。

basename 过滤器可以让我们从全路径中提取文件名部分：*index.html*。这就允许我们在编写 playbook 时不需要重复文件名^{注1}：

注 1：感谢 John Jarvis 对此处的建议。

```
vars:  
  homepage: /usr/share/nginx/html/index.html  
tasks:  
 - name: copy home page  
   copy: src=files/{{ homepage | basename }} dest={{ homepage }}
```

编写你自己的过滤器

回顾一下我们的 Mezzanine 范例，我们利用模板生成了 *local_settings.py* 文件。生成的文件中有一行如例 7-14 所示：

例7-14 模板生成的local_settings.py中的一行

```
ALLOWED_HOSTS = ["www.example.com", "example.com"]
```

我们有一个名为 *domains* 的变量，这个变量包含一个主机名的列表。我们起初是在模板中使用 `for` 循环来生成这一行，但是过滤器是更优雅的方案。◀135

`join` 是 Jinja2 的内置过滤器。它可以使用分隔符将一系列字符串连接成一个列表。可惜的是，它处理的结果并不完全是我们想要的。如果我们在模板中按照如下方式使用它：

```
ALLOWED_HOSTS = [{% domains|join(",") %}]
```

最后我们将会在文件中得到一个未被引号引起起来的字符串，如例 7-15 所示。

例7-15 未被正确引起起来的字符串

```
ALLOWED_HOSTS = [www.example.com, example.com]
```

如果有一个可以将列表中的字符串引起起来的 Jinja2 过滤器，如例 7-16 所示，那么模板生成的输出将同例 7-14 中的内容一样。

例7-16 使用过滤器将列表中的字符串引起起来

```
ALLOWED_HOSTS = [{% domains|surround_by_quote|join(", ") %}]
```

遗憾的是，并不存在我们想要的 `surround_by_quote` 过滤器。不过我们可以自己写一个。(实际上，孙翰菲已经在 Stack Overflow 上解决了这个问题，网址为 <http://stackoverflow.com/questions/15514365/add-quota-around-each-string-in-a-list-in-jinja2>)。

Ansible 会在我们放 `playbook` 的目录下的 `filter_plugins` 目录中查找自定义过滤器。

例 7-17 列出了上面讨论的过滤器的一种实现方式。

例7-17 filter_plugins/surround_by_quotes.py

```
# 源自 http://stackoverflow.com/a/15515929/742
```

```

def surround_by_quote(a_list):
    return ['"%s"' % an_element for an_element in a_list]

class FilterModule(object):
    def filters(self):
        return {'surround_by_quote': surround_by_quote}

```

surround_by_quote 函数定义了 Jinja2 过滤器。FilterModule 类定义了一个 filters 方法，这个方法返回由过滤器函数名和函数本身组成的字典。FilterModule 类是 Ansible 相关的代码，它使得 Jinja2 过滤器可以在 Ansible 中使用。

你也可以将过滤器插件放在 `/usr/share/ansible_plugins/filter_plugins` 目录下，或者你还可以将 `ANSIBLE_FILTER_PLUGINS` 环境变量设置为存放你插件的目录，通过这样的方法来自己指定目录。这些路径在附录 B 中都有收录。

lookup

在理想情况下，你的所有配置信息都会被作为 Ansible 变量保存，而且可以保存在 Ansible 允许你定义变量的各种地方（例如 playbook 中的 vars 区段、通过 vars_files 加载的文件、我们在第 3 章讨论过的 host_vars 目录和 group_vars 目录下的文件）。

可叹的是，真实世界要混乱得多。有时候，总有那么一点配置数据需要散落在其他地方。也许是在文本文件中或者 .csv 文件中。你肯定不希望将这些数据拷贝到 Ansible 变量文件中，因为那样你就需要维护两份一样的数据，而你信奉 DRY^{注2} 原则。或者也许这些数据根本不是作为文件来维护的，它们维护在类似 etcd³ 这样的 key-value 存储系统中。Ansible 有个名为 lookup 的功能，lookup 允许你从千奇百怪的来源读取配置数据，并在你的 playbook 和模板中使用这些数据。

Ansible 支持一套从不同来源获取数据的 lookup，如表 7-5 所示。

表 7-5 lookup

名称	描述
file	文件的内容
password	随机生成一个密码
pipe	本地命令执行的输出
env	环境变量

注 2：Don't Repeat Yourself，因 Andy Hunt 和 Dave Thomas 的神作 *The Pragmatic Programmer* 而普及的术语。
注 3：etcd 是一个分布式的 key-value 存储。它由 CoreOS 项目组维护 (<https://coreos.com/docs/etcd/>)。

续表

名称	描述
template	Jinja2 模板渲染的结果
csvfile	.csv 文件中的条目
dnstxt	DNS 的 TXT 记录
redis_kv	对 Redis 中的 key 进行查询
etcd	对 etcd 中的 key 进行查询

137

你可以通过调用 `lookup` 函数并传入两个参数来实现 `lookup` 调用。第一个参数是选用的 `lookup` 的名字组成的字符串；第二个参数是包含传递给 `lookup` 的一个或多个参数的字符串。例如，我们可以像下面这样调用 `file` `lookup`：

```
lookup('file', '/path/to/file.txt')
```

你可以在 `playbook` 中的 `{} {}` 里调用 `lookup`，或者你可以将它们放到模板中。

在本节中，我只对 `lookup` 可以做什么做一个简短的概述。Ansible 的官方文档 (http://docs.ansible.com/ansible/playbooks_lookups.html) 提供了更多关于如何使用 `lookup` 的细节。

Ansible 所有的 `lookup` 插件都是在控制主机（而不是远程主机上）上执行的。



file

比如说在你的控制主机上有一个文本文件，这个文本文件中包含有你希望复制到远程主机的 SSH 公钥。例 7-18 演示了如何使用 `file` `lookup` 来读取文件的内容并将其作为参数传递给模块。

例 7-18 使用 `file` `lookup`

```
- name: Add my public key as an EC2 key
  ec2_key: name=mykey key_material="{{ lookup('file',
  '/Users/lorinhochstein/.ssh/id_rsa.pub') }}"
```

你还可以在模板中调用 `lookup`。如果我们想要使用相同的技术来创建一个包含公钥文件内容的 `authorized_keys` 文件，我们可以创建一个调用 `lookup` 的 `Jinja2` 模板，如例 7-19 所示。

然后我们可以在 playbook 中调用 template 模块，如例 7-20 所示。

138

例 7-19 authorized_keys.j2

```
{{ lookup('file', '/Users/lorinhochstein/.ssh/id_rsa.pub') }}
```

例 7-20 生成authorized_keys 的 task

```
- name: copy authorized_host file  
  template: src=authorized_keys.j2 dest=/home/deploy/.ssh/authorized_keys
```

pipe

pipe lookup 在控制主机上调用一个外部程序，并将这个程序的输出打印到标准输出上。

例如，如果我们的 playbook 使用 git 进行版本控制，并且我们希望得到对最新的 git commit^{注4} 使用 SHA-1 算法的值，我们就可以使用 pipe lookup：

```
- name: get SHA of most recent commit  
  debug: msg="{{ lookup('pipe', 'git rev-parse HEAD') }}"
```

输出类似下面这样：

```
TASK: [get the sha of the current commit] ****  
ok: [myserver] => {  
    "msg": "e7748af0f040d58d61de1917980a210df419eae9"  
}
```

env

env lookup 实际就是获取在控制主机上的某个环境变量的值。我们可以像这样使用这个 lookup：

```
- name: get the current shell  
  debug: msg="{{ lookup('env', 'SHELL') }}"
```

因为我使用 zsh 作为我们 shell，所以当我运行它时输出是这样的：

```
TASK: [get the current shell] ****  
ok: [myserver] => {  
    "msg": "/bin/zsh"  
}
```

注 4： 或许你觉得这听起来毫无意义，别担心，这只是一个运行命令的范例而已。

password

password lookup 会随机生成一个密码，并且还会将这个密码写入到参数指定的文件中。例如，如果我们想要创建一个名为 *deploy* 的 Postgre 用户，并且希望随机生成 *deploy* 用户的密码，且将这个密码写入到控制主机的 *deploy-password.txt* 中，我们可以这样做：
139

```
- name: create deploy postgres user
  postgresql_user:
    name: deploy
    password: "{{ lookup('password', 'deploy-password.txt') }}"
```

template

template lookup 让你指定一个 Jinja2 模板文件，然后返回这个模板渲染的结果。如果我们有一个如例 7-21 所示的模板：

例7-21 message.j2

```
This host runs {{ ansible_distribution }}
```

并且我们定义一个这样的 task：

```
- name: output message from template
  debug: msg="{{ lookup('template','message.j2') }}"
```

那么我们将会看到类似如下的输出：

```
TASK: [output message from template] ****
ok: [myserver] => {
    "msg": "This host runs Ubuntu\n"
}
```

csvfile

csvfile lookup 可以从 *.csv* 文件中读取出一个条目。假设我们有一个如例 7-22 所示的 *.csv* 文件：

例7-22 users.csv

```
username,email
lorin,lorin@ansiblebook.com
john,john@example.com
sue,sue@example.org
```

如果我们想要使用 csvfile lookup 插件提取出 Sue 的电子邮件地址，我们需要这样调用 lookup 插件：

```
lookup('csvfile', 'sue file=users.csv delimiter=, col=1')
```

csvfile lookup 是一个使用多个参数调用 lookup 的好例子。下面列出了传递给插件的四个参数：

140

- sue
- file=users.csv
- delimiter=,
- col=1

你不需要对 lookup 插件的第一个参数指定一个名字，但是你需要为后面的附加参数指定名字。在这个 csvfile 的例子中，第一个参数是必须精确出现在表格中第 0 列的条目（也就是第一列，索引从 0 开始计数）。

其他的参数指定 .csv 文件的名字、分隔符以及需要将哪一列返回。在我们的例子中，我们想要实现的事情有：查看名为 *users.csv* 的文件；使用逗号作为分隔符来定位区域；寻找第一列的值是 *sue* 的那一行；返回第二列的值（即列 1，索引从 0 开始计数）。最后将会得到 *sue@example.org*。

如果我们想要查找的用户名被存储在名为 *username* 的变量中，我们可以使用 “+” 符号连接 *username* 字符串和其他的参数字符串，来构建完整的参数字符串：

```
lookup('csvfile', username + 'file=users.csv delimiter=, col=1')
```

dnstxt



dnstxt 模块需要你在控制主机上安装 Python 软件包 *dnspython*。

作为本书的读者，你很可能了解域名系统（DNS）是做什么的。但是考虑到少数读者可能不了解，所以这里还是简单介绍一下。DNS 是一种服务，它将类似 *ansiblebook.com* 这样的主机名翻译为类似 *64.99.80.30* 这样的 IP 地址。

DNS 是通过将一个主机名与一条或多条记录相关联来工作的。最经常使用的 DNS 记录类型是 A 记录和 CNAME 记录。A 记录将一个主机名与一个（或多个）IP 地址相关联；而 CNAME 记录则指定一个主机名是另一个主机名的别名。

DNS 协议还支持另一个与主机名相关的记录类型，叫作 *TXT* 记录。*TXT* 记录就是一个可以附加在主机名上的任意字符串。一旦你为一个主机名关联一条 *TXT* 记录，任何人都可以使用 DNS 客户端获取这段文本。

举个例子，*ansiblebook.com* 域是被我持有的，所以我可以为这个域^{注5}下所有主机名创建相关联的 *TXT* 记录。我为 *ansiblebook.com* 主机名关联了一条 *TXT* 记录，这条 *TXT* 记录中包含本书的 ISBN 号。你可以像例 7-23 中所示使用 *dig* 命令行工具查找 *TXT* 记录。

例 7-23 使用 *dig* 工具来查阅 *TXT* 记录

```
$ dig +short ansiblebook.com TXT
"isbn=978-1491915325"
```

dnstxt *lookup* 会向 DNS 服务器查询关于对应主机相关联的 *TXT* 记录。如果我们在 *playbook* 中创建如下 task：

```
- name: look up TXT record
  debug: msg="{{ lookup('dnstxt', 'ansiblebook.com') }}"
```

那么它的输出则类似如下所示的内容：

```
TASK: [look up TXT record] ****
ok: [myserver] => {
    "msg": "isbn=978-1491915325"
}
```

如果某一个主机拥有多个相关联的 *TXT* 记录，那么模块会将它们连在一起，并且每次调用时的连接顺序可能不同。例如，如果 *ansiblebook.com* 还有第二条 *TXT* 记录，内容为下面的文本：

```
author=lorin
```

那么 *dnstxt* *lookup* 将会随机返回下面两个结果之一：

- isbn=978-1491915325author=lorin
- author=lorinisbn=978-1491915325

redis_kv



redis_kv 模块需要你在控制主机上安装 Python 软件包 *redis*。

注 5： DNS 服务提供商通常提供一个 Web 界面来执行 DNS 相关的任务，例如创建 *TXT* 记录。

Redis 是一个非常流行的 key-value 存储。它常被用于缓存或者类似 Sidekiq 这样的消息队列系统的数据存储。你可以使用 `redis_kv` lookup 来获取一个 key 的 value。key 必须是一个字符串，如同是在使用 Redis `GET` 命令一样。

还是通过例子来理解，比如说我们在控制主机上运行着一个 Redis 服务器端，并且我们已经通过如下操作将名为 `weather` 的 key 对应的 value 设置为 `sunny`：

```
$ redis-cli SET weather sunny
```

如果我们在 playbook 中定义一个调用 Redis lookup 的 task：

```
- name: look up value in Redis
  debug: msg="{{ lookup('redis_kv', 'redis://localhost:6379,weather') }}"
```

输出将会如下所示：

```
TASK: [look up value in Redis] ****
ok: [myserver] => {
    "msg": "sunny"
}
```

如果没有指定 URL，这个模块会默认连接到 `redis://localhost:6379`。所以我们实际上可以这样调用这个模块（要注意 key 前面的逗号）：

```
lookup('redis_kv', ',weather')
```

etcd

etcd 是一个分布式 key-value 存储。它通常被用于保存配置信息或者用于实现服务发现。你可以使用 etcd lookup 来从 etcd 中获取指定 key 的 value。

举例来说，比如说我们在控制主机上运行了一个 etcd 的服务器端，并且我们已经通过如下操作将名为 `weather` 的 key 对应的 value 设置为 `cloudy`：

```
$ curl -L http://127.0.0.1:4001/v2/keys/weather -XPUT -d value=cloudy
```

如果我们在 playbook 中定义如下所示调用 etcd 插件的 task：

```
- name: look up value in etcd
  debug: msg="{{ lookup('etcd', 'weather') }}"
```

该 task 执行的输出如下所示：

```
TASK: [look up value in etcd] ****
ok: [localhost] => {
```

```
        "msg": "cloudy"
    }
```

默认情况下，`etcd` lookup 会在 `http://127.0.0.1:4001` 上查找 `etcd` 服务器，但是你可以在执行 `ansible-playbook` 之前通过设置 `ANSIBLE_ETCD_URL` 环境变量来更改这个设置。

编写你自己的 lookup 插件

< 143

如果已存在的 lookup 插件无法实现你需要的功能，你也可以编写自己的 lookup 插件。编写自定义 lookup 插件不在本书的范围之内。但如果你对此感兴趣，我推荐你看看 Ansible 随附的 lookup 插件 (<https://github.com/ansible/ansible/tree/devel/lib/ansible/plugins/lookup>) 的源代码。

如果你已经编写了自己的 lookup 插件，将它们放到如下目录中的任意一个里：

- 和你 playbook 并列的 `lookup_plugins` 目录。
- `/usr/share/ansible_plugins/lookup_plugins`。
- `ANSIBLE_LOOKUP_PLUGINS` 环境变量所指定的目录。

更复杂的循环

到本节为止，每当我们需要编写一个遍历对象列表的 task 的时候，我们都会使用 `with_items` 语句来指定对象的列表。尽管这是实现循环的最普通的方法，不过 Ansible 还支持其他机制来实现循环。表 7-6 是对各种循环结构的一个总结。

表7-6 循环结构总结

名称	输入	循环策略
<code>with_items</code>	列表	对列表元素进行循环
<code>with_lines</code>	要执行的命令	对命令输出结果进行逐行循环
<code>with_fileglob</code>	<code>glob</code>	对文件名进行循环
<code>with_first_found</code>	路径的列表	输入中第一个存在的文件
<code>with_dict</code>	字典	对字典元素进行循环
<code>with_flattened</code>	列表的列表	对所有列表的元素顺序循环
<code>with_indexed_items</code>	列表	单次迭代
<code>with_nested</code>	列表	循环嵌套
<code>with_random_choice</code>	列表	单次迭代

名称	输入	循环策略
with_sequence	整数数组	对数组进行循环
with_subelements	字典的列表	嵌套循环
144 with_together	列表的列表	对多个列表并行循环
with_inventory_hostnames	主机匹配模式	对匹配的主机进行循环

Ansible 官方文档 (http://docs.ansible.com/ansible/playbooks_loops.html) 对循环有非常细致和透彻的介绍，所以在这里我只演示几个例子帮助你感受一下它们是如何工作的。

with_lines

`with_lines` 循环结构会让你在控制主机上执行任意命令，并对命令的输出进行逐行迭代。

想象一下你有一个包含一系列名字的文件，你希望向每个名字发送一个 Slack^{注6} 消息。就好像这样：

```
Leslie Lamport
Silvio Micali
Shafi Goldwasser
Judea Pearl
```

例 7-24 演示了如何使用 `with_lines` 读取文件并逐行迭代它的内容。

例 7-24 使用 with_lines 构建循环

```
- name: Send out a slack message
slack:
  domain: example.slack.com
  token: "{{ slack_token }}"
  msg: "{{ item }}"
with_lines:
  - cat files/turing.txt
```

with_fileglob

`with_fileglob` 结构对于迭代控制主机上的一系列文件很有用。

例 7-25 演示了如何对 `/var/keys` 目录下，以及与你的 playbook 并列的 `keys` 目录下所有以 `.pub` 结尾的文件进行迭代。这个范例中我们使用了 file lookup 插件来获取文件的内容，

注 6：Slack 是向企业提供一体化消息 SaaS 服务的著名企业级 SaaS 厂商。国内的同类产品有工作宝和信鸽。——译者注

然后传递给 `authorized_key` 模块。

例7-25 使用`with_fileglob`来添加公钥

```
- name: add public keys to account
  authorized_key: user=deploy key="{{ lookup('file', item) }}"
  with_fileglob:
    - /var/keys/*.pub
    - keys/*.pub
```

with_dict

`with_dict`可以让你对字典而不是列表进行迭代。当你使用这种循环结构的时候，`item` 循环变量是一个含有如下两个 key 的字典。

`key`

字典中的一个 key。

`value`

字典中与上面的 `key` 相对应的 `value`。

还是用例子来说明，如果你的主机有一个接口叫作 `eth0`，那么就会有一个名为 `ansible_` `eth0` 的 Ansible fact。`ansible_` `eth0` 中名为 `ipv4` 的 key 对应了一个如下所示的字典：

```
{
  "address": "10.0.2.15",
  "netmask": "255.255.255.0",
  "network": "10.0.2.0"
}
```

我们像下面这样迭代字典并逐一输出相应的条目：

```
- name: iterate over ansible_eth0
  debug: msg={{ item.key }}={{ item.value }}
  with_dict: ansible_eth0.ipv4
```

输出如下所示：

```
TASK: [iterate over ansible_eth0] ****
ok: [myserver] => (item={'key': u'netmask', 'value': u'255.255.255.0'}) => {
  "item": {
    "key": "netmask",
    "value": "255.255.255.0"
  },
  "msg": "netmask=255.255.255.0"
}
```

146 ➤

```
ok: [myserver] => (item={'key': u'network', 'value': u'10.0.2.0'}) => {
    "item": {
        "key": "network",
        "value": "10.0.2.0"
    },
    "msg": "network=10.0.2.0"
}
ok: [myserver] => (item={'key': u'address', 'value': u'10.0.2.15'}) => {
    "item": {
        "key": "address",
        "value": "10.0.2.15"
    },
    "msg": "address=10.0.2.15"
}
```

将循环结构用作 lookup 插件

Ansible 实现了将循环结构作为 lookup 插件的特性。你只需要在 lookup 插件之前添加 with 就可以在循环结构中使用它。例如，我们可以利用 with_file 的形式重新编写例 7-18，结果如例 7-26 所示。

例 7-26 将 file lookup 用于循环

```
- name: Add my public key as an EC2 key
  ec2_key: name=mykey key_material="{{ item }}"
  with_file: /Users/lorinhochstein/.ssh/id_rsa.pub
```

通常情况下，只要 lookup 插件返回一个列表，就可以用作循环结构。同时这也是我将插件分别列于两个表中所使用的原则：表 7-5（返回字符串）和表 7-6（返回列表）。

我们在本章中讨论了很多基础功能。在下一章，我们将会讨论 *role*，它是一种用于组织 playbook 的便利机制。

role：扩展你的playbook

我最爱 Ansible 的地方就是可以灵活地向上或向下伸缩。我并不是指你所管理的主机数目的伸缩，而是你尝试自动化的工作的复杂度。

Ansible 擅长向下收缩是因为简单的 task 非常容易实现。Ansible 擅长向上扩展是因为它提供了将复杂任务分解为比较小的部分的机制。

在 Ansible 中，*role* 是将 playbook 分割为多个文件的主要机制。它大大简化了复杂 playbook 的编写，同时还使得它们非常易于复用。*role* 可以看成是你分配给一台或多台主机的某些事情。例如，你会把 *database role* 分配给那些数据库服务器的主机。

role 的基本构成

每个 Ansible 的 role 都会有一个名字，比如“database”。与 database role 相关的文件都放在 *roles/database* 目录中。这个目录中包含以下文件及目录。

roles/database/tasks/main.yml

task

roles/database/files/

保存着需要上传到主机的文件

roles/database/templates/

保存 Jinja2 模板文件

roles/database/handlers/main.yml

handler

roles/database/vars/main.yml

不应被覆盖的变量

148 > *roles/database/defaults/main.yml*

可以被覆盖的默认变量

roles/database/meta/main.yml

role 的依赖信息

每个单独的文件都是可选的。如果你的 role 并不包含 handler，那么并不需要准备空的 *handlers/main.yml* 文件。

Ansible 会到哪里查找我的 role

Ansible 将会到与你的 playbook 并列的 *roles* 目录下寻找 role。它也会在 */etc/ansible/roles* 中查找系统级的 role。你可以如例 8-1 所示，通过改变 *ansible.cfg* 文件 defaults 区段中 *roles_path* 的设置来定制系统级 role 的位置。

例 8-1 *ansible.cfg*：覆盖默认的 role 路径设置

```
[defaults]
roles_path = ~/ansible_roles
```

你也可以通过设置 *ANSIBLE_ROLES_PATH* 环境变量来覆盖这个设置，详见附录 B。

范例：Database 和 Mezzanine role

我们这次还是拿 Mezzanine playbook 开刀，使用 Ansible role 来实现这个 playbook。我们可以只创建一个名为“mezzanine”的 role，但是我还是决定将 Postgre 数据库的部署拆分到名为“database”的独立 role 中。这样会使得我们最终将数据库部署在与 Mezzanine 应用相独立的主机上的工作更为容易。

在你的 playbook 中使用 role

在我们进入如何定义 role 的细节之前，让我们先聊聊如何在 playbook 中将 role 分配给主机。

一旦我们定义好 database role 和 mezzanine role，向单台主机部署 Mezzanine 的 playbook 将变化为如例 8-2 所示。

例8-2 mezzanine-single-host.yml

```
- name: deploy mezzanine on vagrant
  hosts: web
  vars_files:
    - secrets.yml

  roles:
    - role: database
      database_name: "{{ mezzanine_proj_name }}"
      database_user: "{{ mezzanine_proj_name }}"

    - role: mezzanine
      live_hostname: 192.168.33.10.xip.io
      domains:
        - 192.168.33.10.xip.io
        - www.192.168.33.10.xip.io
```

当我们使用 role 的时候，在我们的 playbook 中将会增加一个 roles 区段。roles 区段需要列出一个所使用的 role 的列表。在我们的范例中，这个列表中包括两个 role : database 和 mezzanine。

注意一下我们在调用 role 的时候是如何传递变量的。在我们的范例中，我们为 database role 传递了 database_name 和 database_user 变量。如果这些变量在 role 中已经被定义了（不管是在 vars/main.yml 中还是 defaults/main.yml 中），那么变量的值将被传入的变量覆盖。

如果你不需要向 role 传递变量，那么可以像如下这样简单地指定 role 的名字：

```
roles:
  - database
  - mezzanine
```

定义了 database role 和 mezzanine role 之后，编写一个在多个不同的主机上部署 Web 应用与数据库服务的 playbook 变得更加简单。如例 8-3 所示的 playbook 在 db 主机上部署数据库，在 web 主机上部署 Web 服务。需要注意的是，这个 playbook 中包含两个独立的 play。

例8-3 mezzanine-across-hosts.yml

```
- name: deploy postgres on vagrant
  hosts: db
  vars_files:
    - secrets.yml
  roles:
    - role: database

  - name: deploy mezzanine on vagrant
    hosts: web
    vars_files:
      - secrets.yml
    roles:
      - role: mezzanine
```

```

database_name: "{{ mezzanine_proj_name }}"
database_user: "{{ mezzanine_proj_name }}"

- name: deploy mezzanine on vagrant
  hosts: web
  vars_files:
    - secrets.yml
  roles:
    - role: mezzanine
      database_host: "{{ hostvars.db.ansible_eth1.ipv4.address }}"
      live_hostname: 192.168.33.10.xip.io
      domains:
        - 192.168.33.10.xip.io
        - www.192.168.33.10.xip.io

```

Pre-Task 和 Post-Task

有时候，你希望调用你的 role 之前或者之后运行一些 task。比如说你在部署 Mezzanine 之前会希望先更新一下 apt 缓存，并希望在部署完成后，向 Slack 频道发送一个通知。

Ansible 允许你把在 role 之前执行的一系列 task 定义在 pre_tasks 区段，而在 role 之后执行的一系列 task 定义在 post_tasks 区段。例 8-4 列出了一个非常具有实践性的范例。

例 8-4 使用 pre-tasks 和 post-tasks

```

- name: deploy mezzanine on vagrant
  hosts: web
  vars_files:
    - secrets.yml
  pre_tasks:
    - name: update the apt cache
      apt: update_cache=yes
  roles:
    - role: mezzanine
      database_host: "{{ hostvars.db.ansible_eth1.ipv4.address }}"
      live_hostname: 192.168.33.10.xip.io
      domains:
        - 192.168.33.10.xip.io
        - www.192.168.33.10.xip.io
  post_tasks:
    - name: notify Slack that the servers have been updated
      local_action: >
        slack
        domain=acme.slack.com
        token={{ slack_token }}

```

```
msg="web server {{ inventory_hostname }} configured"
```

嗯，使用 role 的内容介绍到这里就差不多了。接下来让我们聊聊编写 role。

用于部署数据库的“database”role

我们的“database”role 的工作是安装 Postgre 并创建需要的数据库与数据库用户。

我们的 database role 包括如下文件。

- *roles/database/tasks/main.yml*
- *roles/database/defaults/main.yml*
- *roles/database/handlers/main.yml*
- *roles/database/files/pg_hba.conf*
- *roles/database/files/postgresql.conf*

151

这个 role 包含两个自定义 Postgre 配置文件。

postgresql.conf

修改了默认的 `listen_addresses` 配置项以使得 Postgre 可以接受来自于任意网卡的连接。Postgre 默认仅接受来自于本机的连接。如果我们想要让数据库在 Web 应用以外的独立主机上运行的话，这个默认行为将会导致服务不能正常工作。

pg_hba.conf

配置 Postgre 对通过网络的连接使用用户名与密码认证。



因为这些文件太大了，我就不在这里展示它们了。你可以在 GitHub (<https://github.com/lorin/ansiblebook>) 的 ch08 目录下的样例代码中找到它们。

例 8-5 列出了用于部署 Postgre 的 task。

例 8-5 *roles/database/tasks/main.yml*

```
- name: install apt packages
  apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
  sudo: True
  with_items:
    - libpq-dev
    - postgresql
    - python-psycopg2
```

```

- name: copy configuration file
  copy:
    src=postgresql.conf dest=/etc/postgresql/9.3/main/postgresql.conf
    owner=postgres group=postgres mode=0644
  sudo: True
  notify: restart postgres

- name: copy client authentication configuration file
  copy:
    src=pg_hba.conf dest=/etc/postgresql/9.3/main/pg_hba.conf
    owner=postgres group=postgres mode=0640
  sudo: True
  notify: restart postgres

- name: create a user
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"
  sudo: True
  sudo_user: postgres

- name: create the database
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"
    encoding: UTF8
    lc_ctype: "{{ locale }}"
    lc_collate: "{{ locale }}"
    template: template0
  sudo: True
  sudo_user: postgres

```

例 8-6 列出了 handler 文件。

例8-6 roles/databases/handlers/main.yml

```

- name: restart postgres
  service: name=postgresql state=restarted
  sudo: True

```

我们唯一指定的默认变量是数据库端口号，如例 8-7 所示。

例8-7 roles/database/defaults/main.yml

```
database_port: 5432
```

需要注意，我们这一系列 task 中引用了几个我们还没有在 role 中任何地方定义过的变量：

- database_name

- database_user
- db_pass
- locale

在例 8-2 和例 8-3 中，当我们调用 role 的时候，我们将 database_name 和 database_user 作为参数传入。我假定 db_pass 已经在 vars_files 区段中加载的 secrets.yml 文件中定义了。locale 变量很可能是每个主机都一样的变量并且可能在多个 role 或者 playbook 中用到，所以我在 group_vars/all 文件中定义了它。本书的范例代码中可以找到这个文件。

◀ 153

为什么会有两种在 role 中定义变量的方法

当 Ansible 最初引进对 role 的支持的时候，只有在 vars/main.yml 文件这一个地方可以定义 role 变量。在这个位置定义的变量比在 play 的 vars 区段中定义的变量具有更高的优先级。这意味着，除非你将变量作为参数显式地传递给 role，不然你就没法覆盖变量的值。

后来，Ansible 引入了默认 role 变量的概念。对，就是放在 defaults/main.yml 文件中的那个。这个类型的变量是在 role 中定义的，但是它是低优先级的，所以如果 playbook 中定义了另一个同名的变量，它的值将会被覆盖。

如果你认为你可能希望在 role 中变更变量的值，那么使用默认变量。如果你不希望变更，那么使用常规的变量就可以了。

用于部署 Mezzanine 的“mezzanine” role

“mezzanine” role 的工作是安装 Mezzanine。这还包含安装 Nginx 作为反向代理以及 Supervisor 作为进程监控。

下面是所涉及的文件。

- roles/mezzanine/defaults/main.yml
- roles/mezzanine/handlers/main.yml
- roles/mezzanine/tasks/django.yml
- roles/mezzanine/tasks/main.yml
- roles/mezzanine/tasks/nginx.yml
- roles/mezzanine/templates/gunicorn.conf.py.j2
- roles/mezzanine/templates/local_settings.py.filters.j2
- roles/mezzanine/templates/local_settings.py.j2

- *roles/mezzanine/templates/nginx.conf.j2*
- *roles/mezzanine/templates/supervisor.conf.j2*
- *roles/mezzanine/vars/main.yml*

如例 8-8 所示是我们为这个 role 定义的变量。你可能已经注意到了，我们将变量的名字都变更为以“mezzanine”开头。这是一个关于 role 变量的良好实践，因为 Ansible 在不同的 role 之间完全没有命名空间的概念。这意味着在其他 role 中定义的变量，或者在 playbook 中其他地方定义的变量，可以在任何地方被访问到。如果你意外地在两个不同的 role 中使用了相同的变量名，将会导致意料之外的行为。

例8-8 roles/mezzanine/vars/main.yml

```
# 用于 Mezzanine 的变量文件
mezzanine_user: "{{ ansible_ssh_user }}"
mezzanine_venv_home: "{{ ansible_env.HOME }}"
mezzanine_venv_path: "{{ mezzanine_venv_home }}/{{ mezzanine_proj_name }}"
mezzanine_repo_url: git@github.com:lorin/mezzanine-example.git
mezzanine_proj_dirname: project
mezzanine_proj_path: "{{ mezzanine_venv_path }}/{{ mezzanine_proj_dirname }}"
mezzanine_reqs_path: requirements.txt
mezzanine_conf_path: /etc/nginx/conf
mezzanine_python: "{{ mezzanine_venv_path }}/bin/python"
mezzanine_manage: "{{ mezzanine_python }} {{ mezzanine_proj_path }}/manage.py"
mezzanine_gunicorn_port: 8000
```

例 8-9 列出了为我们的 mezzanine role 定义的默认变量。在这个范例中，我们只有一个变量。当我编写默认变量的时候，我多半不会添加前缀。因为我们可能会希望在其他地方有意地覆盖它的值。

例8-9 roles/mezzanine/defaults/main.yml

```
ssl_enabled: True
```

因为 task 的列表相当长，所以我决定将它们分割为多个文件。如例 8-10 所示为我们 mezzanine role 的顶级 task 文件。在这个 task 中首先安装 apt 软件包，然后使用 include 语句来调用在同一目录下的两个其他的 task 文件。这两个文件罗列在例 8-11 和例 8-12 中。

例8-10 roles/mezzanine/tasks/main.yml

```
- name: install apt packages
  apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
  sudo: True
  with_items:
    - git
    - libjpeg-dev
    - libpq-dev
```

```

- memcached
- nginx
- python-dev
- python-pip
- python-psycopg2
- python-setuptools
- python-virtualenv
- supervisor

- include: django.yml

- include: nginx.yml

```

例8-11 roles/mezzanine/tasks/django.yml

```

- name: check out the repository on the host
  git:
    repo={{ mezzanine_repo_url }}
    dest={{ mezzanine_proj_path }}
    accept_hostkey=yes

- name: install required python packages
  pip: name={{ item }} virtualenv={{ mezzanine_venv_path }}
  with_items:
    - gunicorn
    - setproctitle
    - south
    - psycopg2
    - django-compressor
    - python-memcached

- name: install requirements.txt
  pip: >
    requirements={{ mezzanine_proj_path }}/{{ mezzanine_reqs_path }}
    virtualenv={{ mezzanine_venv_path }}

- name: generate the settings file
  template: src=local_settings.py.j2 dest={{ mezzanine_proj_path }}/local_settings.py

- name: sync the database, apply migrations, collect static content
  django_manage:
    command: "{{ item }}"
    app_path: "{{ mezzanine_proj_path }}"
    virtualenv: "{{ mezzanine_venv_path }}"
  with_items:
    - syncdb
    - migrate

```

156 >

```
- collectstatic

- name: set the site id
  script: scripts/setsite.py
  environment:
    PATH: "{{ mezzanine_venv_path }}/bin"
    PROJECT_DIR: "{{ mezzanine_proj_path }}"
    WEBSITE_DOMAIN: "{{ live_hostname }}"

- name: set the admin password
  script: scripts/setadmin.py
  environment:
    PATH: "{{ mezzanine_venv_path }}/bin"
    PROJECT_DIR: "{{ mezzanine_proj_path }}"
    ADMIN_PASSWORD: "{{ admin_pass }}"

- name: set the gunicorn config file
  template: src=gunicorn.conf.py.j2 dest={{ mezzanine_proj_path }}/gunicorn.conf.py

- name: set the supervisor config file
  template: src=supervisor.conf.j2 dest=/etc/supervisor/conf.d/mezzanine.conf
  sudo: True
  notify: restart supervisor

- name: ensure config path exists
  file: path={{ mezzanine_conf_path }} state=directory
  sudo: True
  when: ssl_enabled

- name: install poll twitter cron job
  cron: >
    name="poll twitter" minute="*/5" user={{ mezzanine_user }}
    job="{{ mezzanine_manage }} poll_twitter"
```

例8-12 roles/mezzanine/tasks/nginx.yml

```
- name: set the nginx config file
  template: src=nginx.conf.j2 dest=/etc/nginx/sites-available/mezzanine.conf
  notify: restart nginx
  sudo: True

- name: enable the nginx config file
  file:
    src: /etc/nginx/sites-available/mezzanine.conf
    dest: /etc/nginx/sites-enabled/mezzanine.conf
    state: link
  notify: restart nginx
  sudo: True
```

```

- name: remove the default nginx config file
  file: path=/etc/nginx/sites-enabled/default state=absent
  notify: restart nginx
  sudo: True

- name: create ssl certificates
  command: >
    openssl req -new -x509 -nodes -out {{ mezzanine_proj_name }}.crt
    -keyout {{ mezzanine_proj_name }}.key -subj '/CN={{ domains[0] }}' -days 3650
    chdir={{ mezzanine_conf_path }}
    creates={{ mezzanine_conf_path }}/{{ mezzanine_proj_name }}.crt
  sudo: True
  when: ssl_enabled
  notify: restart nginx

```

在 role 中定义 task 和在常规的 playbook 中定义 task 之间有一个很重要的区别，就是在使用 copy 模块或 template 模块的时候。 ◀157

在 role 中定义的 task 中调用 copy 模块的时候，Ansible 将会首先把 *rolename/files/* 目录作为需要复制的文件的位置进行检查。类似地，在 role 中定义的 task 中调用 template 模块的时候，Ansible 将会首先把 *rolename/templates/* 目录作为需要使用的模板的位置进行检查。

这意味着过去在 playbook 中像下面这样的 task：

```

- name: set the nginx config file
  template: src=templates/nginx.conf.j2 \
  dest=/etc/nginx/sites-available/mezzanine.conf

```

现在在 role 中调用将会变成这样（注意 src 参数的变化）：

```

- name: set the nginx config file
  template: src=nginx.conf.j2 dest=/etc/nginx/sites-available/mezzanine.conf
  notify: restart nginx

```

例 8-13 列出了 handler 文件。

例 8-13 roles/mezzanine/handlers/main.yml

```

- name: restart supervisor
  supervisorctl: name=gunicorn_mezzanine state=restarted
  sudo: True

- name: restart nginx
  service: name=nginx state=restarted
  sudo: True

```

尽管模板文件中一些变量名会有些变化，但总的来说模板文件基本和前面章节中的一样，所以我就不在这里示范它了。可以通过本书配套的范例代码来了解更多细节。

使用 ansible-galaxy 创建 role 文件与目录

Ansible 还随附另一个命令行工具 *ansible-galaxy*，而且我们至今还没有讨论到它。它的主要作用是下载那些由 Ansible 社区（本章稍后详述）分享的 role。但是它还可以用来生成 *scaffolding*。*scaffolding* 是指在 role 中调用的一套初始的文件与目录：

```
$ ansible-galaxy init -p playbooks/roles web
```

-p 参数告诉 ansible-galaxy 你的 role 目录的位置所在。如果你没有指定它，那么 role 文件将被创建在当前目录下。

运行上面的命令会创建如下文件与目录。

- 158
- *playbooks/roles/web/tasks/main.yml*
 - *playbooks/roles/web/handlers/main.yml*
 - *playbooks/roles/web/vars/main.yml*
 - *playbooks/roles/web/defaults/main.yml*
 - *playbooks/roles/web/meta/main.yml*
 - *playbooks/roles/web/files/*
 - *playbooks/roles/web/templates/*
 - *playbooks/roles/web/README.md*

从属 role

假设我们有两个 role，`web` 和 `database`。两个 role 都需要在主机上安装 NTP^{注1} 服务。我们可以在两个 role 中都指定安装 NTP 服务，但是这样做就会产生重复配置。我们可以创建一个单独的 `ntp` role，但是这样的话，我们将不得不牢记让 `web` role 或者 `database` role 对主机执行的时候，我们还必须得让 `ntp` role 也对主机执行。这样虽然避免了重复，但是易于出错，因为我们可能会忘记指定 `ntp` role。我们真正想要的效果是：无论何时 `web` role 或者 `database` role 对主机执行，`ntp` role 都将同时对主机执行。

Ansible 支持一个名为 *dependent role* 的特性来处理上面我们提到的场景。当你定义一个 role 的时候，你可以指定它依赖于一个或者多个其他的 role。Ansible 将会确保被指定为从属的 role 一定会被优先执行。

注 1：NTP 代表网络时间协议（Network Time Protocol），用于时钟同步。

继续回到我们的范例中。比方说我们创建一个 `ntp` role，这个 role 的作用是配置主机从 NTP 服务同步时间。Ansible 允许你向 dependent role 传递参数，所以我们也可以说假设我们将 NTP 服务的地址作为参数传递给 role。

如果我们想要指定 `web` role 依赖于 `ntp` role，可以创建 `roles/web/meta/main.yml` 文件，并像标准 role 定义一样，连带参数一起将 dependent role 列在该文件中，如例 8-14 所示。

例8-14 roles/web/meta/main.yml

```
dependencies:  
  - { role: ntp, ntp_server=ntp.ubuntu.com }
```

我们也可以指定多个 dependent role。例如，如果我们有 `django` role 用于建立 Django Web 服务器，并且我们想要指定 `nginx` 和 `memcached` 作为 dependent role，那么 `django` role 的依赖关系文件将如例 8-15 所示。

例8-15 roles/django/meta/main.html

```
dependencies:  
  - { role: web }  
  - { role: memcached }
```

如果想要了解 Ansible 如何评估 role 依赖关系的细节，可以查看官方文档 (http://docs.ansible.com/ansible/playbooks_roles.html#role-dependencies) 中关于 role 依赖关系的部分。

Ansible Galaxy

如果你需要部署一个开源软件系统到你的主机上，很可能有人已经编写好 Ansible 的 role 来完成这个工作了。尽管编写 Ansible 部署软件的脚本很容易，但是有些系统的部署还是挺烦人的。

不管你想要复用其他人已经写好的 role，还是你仅仅想看看别人是怎么解决你遇到的问题的，Ansible Galaxy 都可以帮助你排忧解难。Ansible Galaxy 是由 Ansible 社区维护的 Ansible role 开源仓库。这些 role 实际存储在 GitHub 上。

Web 界面

你可以在 Ansible Galaxy 的网站 (<https://galaxy.ansible.com/>) 浏览所有可用的 role。Galaxy 支持自由文本搜索和基于类别或贡献者来浏览。

命令行工具

命令行工具 `ansible-galaxy` 可以帮你从 Ansible Galaxy 下载 role。

安装一个 role

比如说我想要安装一个由 GitHub 用户 `bennojoy` 编写的名为 `ntp` 的 role。这个 role 将会配置主机从 NTP 服务器同步时钟。

通过 `install` 命令来安装 role。

```
$ ansible-galaxy install -p ./roles bennojoy.ntp
```

`ansible-galaxy` 程序默认会把 role 安装到系统级 role 所在的位置（参见 150 页的“Ansible 会到哪里查找我的 role”）。我们可以像前面范例中那样使用 `-p` 参数来改变位置。

160 安装 role 命令的输出如下所示：

```
downloading role 'ntp', owned by bennojoy
no version specified, installing master
- downloading role from https://github.com/bennojoy/ntp/archive/master.tar.gz
  - extracting bennojoy.ntp to ./roles/bennojoy.ntp
write_galaxy_install_info!
bennojoy.ntp was installed successfully
```

`ansible-galaxy` 将会把 role 文件安装到 `roles/bennojoy.ntp` 下。

Ansible 还会安装一些关于 role 安装的元数据到 `./roles/bennojoy.ntp/meta/.galaxy_install_info` 文件中。在我的机器上，这个文件内容如下：

```
{install_date: 'Sat Oct 4 20:12:58 2014', version: master}
```



`bennojoy.ntp` 这个 role 并没有指定版本号，所以版本就被简单地标记为“`master`”。有些 role 会指定类似 `1.2` 这样的版本号。

列出已安装的 role

你可以使用下面的命令列出已经安装的 role：

```
$ ansible-galaxy list
```

命令的输出如下：

```
bennojoy.ntp, master
```

卸载 role

可以使用 `remove` 命令卸载 role：

```
$ ansible-galaxy remove bennojoy.ntp
```

向 Galaxy 贡献你自己编写的 role

可以通过在 Ansible Galaxy 网站 (<https://galaxy.ansible.com/intro>) 上查看“如何分享我编写的 role (How To Share Roles You've Written)”来了解向社区贡献 role 的详细方法。因为所有的 role 都是被托管在 GitHub 上的，所以你需要有一个 GitHub 的账号。

到此为止，你应该已经理解了如何使用 role、如何编写你自己的 role 以及如何下载其他人编写的 role。role 是组织你 playbook 的神器。我一向使用 role 来编写 playbook，并且我强烈建议你们也这样做。

让Ansible快到飞起

在本章中，我们将会讨论缩短 Ansible 执行 playbook 时间的办法。

SSH Multiplexing 与 ControlPersist

如果你是从本书开头顺序读到这里的，那么你肯定已经知道了 Ansible 与服务器之间的主要通信机制使用的是 SSH。具体点说，Ansible 默认使用系统的 SSH 程序。

因为 SSH 协议运行在 TCP 协议的顶层，当你使用 SSH 与远程主机建立连接的时候，你需要创建一个新的 TCP 连接。客户端与服务器端在开始真正的通信之前，需要先协商连接。这个协商就是常说的“三次握手”，当然会消耗很小量的时间。

当 Ansible 运行 playbook 的时候，它会建立很多 SSH 连接来执行复制文件、运行命令这样的操作。Ansible 每次都会重新创建到主机的 SSH 连接，这当然需要付出三次握手的开销。

OpenSSH 是最常见的 SSH 实现并且如果你使用 Linux 或者 Mac OS X 的话，几乎可以肯定 OpenSSH 也是你本地主机上安装的 SSH 客户端。OpenSSH 支持一个优化，叫作 *SSH Multiplexing*，也被称作 *ControlPersist*。当你使用 SSH Multiplexing 的时候，多个连接到相同主机的 SSH 会话将会共享相同的 TCP 连接。这样，就只有第一次连接的时候需要进行 TCP 三次握手。

当你启用 Multiplexing 后：

- 你第一次尝试 SSH 连接到主机的时候，OpenSSH 创建一个主连接。
- OpenSSH 创建一个 UNIX 域套接字（又被称作控制套接字），通过主连接与远程主机相关联。

- 下一次你尝试 SSH 到主机的时候，OpenSSH 将使用控制套接字与远程主机通信，并不会创建新的 TCP 连接。

主连接的保存时间是可配置的，当这个时间用完的时候 SSH 客户端就会断开连接。Ansible 默认使用 60s。

手动启用 SSH Multiplexing

Ansible 会自动启动 SSH Multiplexing，但是为了让你对底层都做了什么有一个概念，我们还是逐步地手动启动 SSH Multiplexing 并使用它 SSH 到一台远程主机。

例 9-1 列出了一个 `~/.ssh/config` 文件中条目的范例。这个范例是为 `myserver.example.com` 主机配置的，并且已经配置启用 SSH Multiplexing。

例 9-1 启用 SSH Multiplexing 的 `ssh/config`

```
Host myserver.example.com
  ControlMaster auto
  ControlPath /tmp/%r@%h:%p
  ControlPersist 10m
```

`ControlMaster auto` 这一行启用了 SSH Multiplexing，并且它告诉 SSH 在主连接和控制套接字不存在的情况下，自动创建它们。

`ControlPath /tmp/%r@%h:%p` 这一行告诉 SSH 将 UNIX 域控制套接字文件放在文件系统中的什么位置。`%h` 是目标主机名字，`%r` 是远程登录用户名，`%p` 是口号号。如果我们使用 Ubuntu 用户 SSH 到这台主机：

```
$ ssh ubuntu@myserver.example.com
```

你第一次 SSH 到服务器的时候，SSH 会将控制套接字创建在 `/tmp/ubuntu@myserver.example.com:22`。

`ControlPersist 10m` 这一行告诉 SSH 如果 10 分钟内都没有 SSH 连接，那么就关闭主连接。

你可以使用 `-O check` 参数来查询主连接是否处于打开状态：

```
$ ssh -O check ubuntu@myserver.example.com
```

如果 Control Master 正在运行，那么输出如下所示：

```
Master running (pid=4388)
```

如果你使用 `ps 4388`，将查看到 Control Master 进程如下：

```
4388 ?? Ss      0:00.00 ssh: /tmp/ubuntu@myserver.example.com:22 [mux]
```

你也可以使用 `-O exit` 参数来中断主连接，如下所示：

```
$ ssh -O exit ubuntu@myserver.example.com
```

你可以在 `ssh_config` 的 man 手册中查看更多关于这些设置的细节。

我使用如下方法来测试 SSH 连接的速度：

```
$ time ssh ubuntu@myserver.example.com /bin/true
```

这条命令会统计创建到服务器的 SSH 连接并执行 `/bin/true` 程序的总时间。而 `/bin/true` 程序会直接返回返回值 0。

我第一次运行它的时候，计时部分的输出如下^{注1}：

```
0.01s user 0.01s system 2% cpu 0.913 total
```

我们真正关心的时间是总时间：`0.913 total`。它说明花费了 0.913s 来执行整个命令。（总时间有时候也被称为墙上时间 *wall-clock time*。之所以叫墙上时间，是因为它代表着有多少时间流逝了，就好像我们使用墙上的时钟来测量时间。）

第二次执行的时候，输出如下：

```
0.00s user 0.00s system 8% cpu 0.063 total
```

总时间下降到 0.063s。在初次连接后，每个 SSH 连接可以下降大约 0.85s。回忆一下，Ansible 每执行一个 task 至少使用两个 SSH 会话：其中一个负责将模块文件复制到主机上，另一个负责在主机上执行相应的文件^{注2}。这意味着 SSH Multiplexing 应该为你的 playbook 中运行的每个 task 节省大约一到两秒钟。

Ansible 中的 SSH Multiplexing 选项

在表 9-1 中列出了 Ansible 用于 SSH Multiplexing 的选项。

表9-1 Ansible的SSH Multiplexing选项

164

选项	值
ControlMaster	auto
ControlPath	<code>\$HOME/.ansible/cp/ansible-ssh-%h-%p-%r</code>
ControlPersist	60s

注 1：根据你使用的 Shell 和操作系统，输出的格式可能也有所不同。我是在 Mac OS X 上运行 zsh 的。

注 2：这些步骤可以使用本章稍后就会讨论的 pipelining 来优化。

我从来不去变更 Ansible 中 `ControlMaster` 或者 `ControlPersist` 的默认值。不过我曾经有过变更 `ControlPath` 选项值的需求。这是因为操作系统对于 UNIX 域控制套接字的路径有最大长度限制，如果 `ControlPath` 字符串太长，那么连接复用可能会不工作。最悲惨的是，Ansible 并不会告诉你 `ControlPath` 字符串太长了，它只会不使用 SSH Multiplexing 直接运行。

你可以在你的控制主机上使用 Ansible 所配置的 `ControlPath` 去手动尝试 SSH 来测试是否有这个问题：

```
$ CP=~/ansible/cp/ansible-ssh-%h-%p-%r  
$ ssh -o ControlMaster=auto -o ControlPersist=60s \  
-o ControlPath=$CP \  
ubuntu@ec2-203-0-113-12.compute-1.amazonaws.com \  
/bin/true
```

如果 `ControlPath` 太长了，你将会看到如例 9-2 所示的错误。

例9-2 `ControlPath`太长了

```
ControlPath  
"/Users/lorinhochstein/.ansible/cp/ansible-ssh-ec2-203-0-113-12.compute-1.amazonaws.com-22-ubuntu.KIwEKEsRzCKFABch"  
too long for Unix domain socket
```

当连接到 Amazon EC2 实例的时候就经常会发生这个问题，因为 EC2 使用很长的主机名。

绕过问题的办法就是配置 Ansible 使用更短的 `ControlPath`。官方文档 (http://docs.ansible.com/intro_configuration.html#control-path) 建议你在你的 `ansible.cfg` 文件中设置这个选项：

```
[ssh_connection]  
control_path = %(directory)s/%h-%r
```

Ansible 自动将 `%(directory)s` 替换为 `$HOME/.ansible/cp`。而使用双百分号 `(%%)` 是因为百分号在 `.ini` 格式的文件中是特殊字符，所以需要转义。

165



如果你启用了 SSH Multiplexing，并且你变更了 SSH 连接的配置，比如更改了 `ssh_args` 配置项，那么之前连接打开的控制套接字还存在的时候变更不会生效。

pipelining

回忆一下 Ansible 如何执行一个 task 的。

1. 它基于调用的模块生成一个 Python 脚本。
2. 然后它将 Python 脚本复制到主机上。
3. 最后，它执行这个 Python 脚本。

Ansible 支持一个优化，叫作 *pipelining*。pipelining 模式下 Ansible 执行 Python 脚本的时候并不会复制它，而是通过管道传递给 SSH 会话。因为这会让 Ansible 使用的 SSH 会话由两个减少到一个，所以可以节省时间。

启用 pipelining

pipelining 默认是关闭的，因为它需要依赖你的远程主机上的某些配置。但是它确实能加速执行，所以我很喜欢启用它。要启用 pipelining 的话，需要修改你的 *ansible.cfg* 文件，如例 9-3 所示。

例9-3 启用pipelining的ansible.cfg

```
[defaults]
pipelining = True
```

将主机配置为支持 pipelining

如果想要 pipelining 正常工作，你需要确保主机上的 */etc/sudoers* 文件中的 *requiretty* 没有启用。否则的话，当你运行 playbook 的时候，你会得到像例 9-4 这样的错误。

例9-4 当requiretty启用时候的错误

```
failed: [vagrant1] => {"failed": true, "parsed": false}
invalid output was: sudo: sorry, you must have a tty to run sudo
```

如果你的主机上的 *sudo* 配置为从 */etc/sudoers.d* 中读取文件，那么最简单的解决这个问题的办法是添加一个 *sudoers* 配置文件，并在这个配置文件中对于你 SSH 使用的用户禁用 *requiretty*。

如果 */etc/sudoers.d* 目录存在，那么你的主机应该支持在这个目录中添加 *sudoers* 配置文件。◀166
你可以使用 *ansible* 命令行工具来检查是否存在这个目录：

```
$ ansible vagrant -a "file /etc/sudoers.d"
```

如果目录存在，输出会如下所示：

```
vagrant1 | success | rc=0 >>
/etc/sudoers.d: directory

vagrant3 | success | rc=0 >>
/etc/sudoers.d: directory

vagrant2 | success | rc=0 >>
/etc/sudoers.d: directory
```

如果目录不存在，那么输出就会是这样：

```
vagrant3 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d' (No such file or
directory)

vagrant2 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d' (No such file or
directory)

vagrant1 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d' (No such file or
directory)
```

如果目录存在，那么可以创建如例 9-5 所示的模板文件。

例9-5 templates/disable-requiretty.j2

```
Defaults:{{ ansible_ssh_user }} !requiretty
```

然后运行如例 9-6 所示的 playbook，不过要记得把 myhosts 改成你的主机。别忘了在执行这个之前还是得先禁用 pipelining，不然你的 playbook 会直接报错退出。

例9-6 disable-requiretty.yml

```
- name: do not require tty for ssh-ing user
  hosts: myhosts
  sudo: True
  tasks:
    - name: Set a sudoers file to disable tty
      template: >
        src=templates/disable-requiretty.j2
        dest=/etc/sudoers.d/disable-requiretty
        owner=root group=root mode=0440
        validate="visudo -cf %s"
```

167

注意这里对 validate="visudo -cf %s" 的使用。在 283 页中的“验证文件”中讨论了为什么说在编辑 sudoers 文件时使用 validate 是个好主意。

fact 缓存

如果你的 play 没有引用任何 Ansible fact，那么你可以为这个 play 关闭 fact 采集。还记得吗？我们可以在 play 中使用 `gather_facts` 语句来禁用 fact 采集，例如：

```
- name: an example play that doesn't need facts
hosts: myhosts
gather_facts: False
tasks:
    # 具体 task 内容省略
```

你可以在 `ansible.cfg` 文件中添加如下配置来默认禁用 fact 采集：

```
[defaults]
gathering = explicit
```

如果你编写的 play 确实有引用 fact，你可以使用 fact 缓存。使用了 fact 缓存的情况下，Ansible 只会对一台主机采集一次 fact。即使你重新运行 playbook 或者执行不同的 playbook，只要连接到的是同样的主机，就不会重复进行 fact 采集。

如果 fact 缓存启用，Ansible 在初次连接到主机的时候会将 fact 保存在缓存中。对于随后 playbook 的运行，Ansible 就不再从远程主机获取 fact，而是从缓存中查询，直到缓存过期。

如果想要启用 fact 缓存，你必须将例 9-7 所列的这几行都添加到 `ansible.cfg` 文件中。`fact_caching_timeout` 的值是以秒为单位的，例子中使用 24 小时（86400 秒）作为过期时间。



如同所有基于缓存的解决方案一样，缓存永远具有出现脏数据的风险。某些类似 CPU 架构（保存在 `ansible_architecture` fact 中）这样的 fact 不太可能经常变化。另外一些类似由主机汇报的日期和时间（保存在 `ansible_data_time` fact 中）这样的 fact 肯定会经常变化。

如果你决定启用 fact 缓存，确保你自己清楚地了解在你的 playbook 中用到的可能会改变的 fact 被引用的频繁程度，并设置一个合适的 fact 缓存过期时间。如果你希望在运行 playbook 之前清除 fact 缓存，可以向 `ansible-playbook` 传递 `--flush-cache` 参数。

例9-7 启用fact缓存的ansible.cfg

```
[defaults]
gathering = smart
# 这里将超时时间设定为 24 小时。可以按照具体需要调整此值。
```

<168

```
fact_caching_timeout= 86400  
# 你必须在这里指定选用哪种 fact 缓存后端  
fact_caching = ...
```

在 *ansible.cfg* 文件中将 *gathering* 配置项设定为 “smart” 会让 Ansible 使用智能采集。这意味着 Ansible 将只在缓存中没有对应的数据或者缓存中数据已经过期的时候才会采集 fact。



如果你想要使用 fact 缓存的话，确保你的 playbook 中没有明确指定 *gather_facts:True* 或者 *gather_facts=False*。在配置文件中启用智能采集的时候，Ansible 将只当 fact 在缓存中不存在的时候才进行 fact 采集。

你必须在 *ansible.cfg* 中明确指定 *fact_caching* 实现，否则 Ansible 将在每次 playbook 运行之间不缓存 fact。

截止到编写本书时，fact 缓存的实现有 3 种。

- JSON 文件
- Redis
- Memcached

JSON 文件 fact 缓存后端

使用 JSON 文件作为 fact 缓存后端的时候，Ansible 将会把采集的 fact 写入你控制主机上的文件中。如果你的系统上已经存在了这个文件，那么 Ansible 将使用这个文件中的数据，而不再连接到主机去采集 fact。

添加如例 9-8 所示的设置到 *ansible.cfg* 文件中就可以启用 JSON 文件 fact 缓存后端。

例9-8 使用JSON文件fact缓存的ansible.cfg

```
[defaults]  
gathering = smart  
  
# 这里将超时时间设定为 24 小时。可以按照具体需要调整此值。  
fact_caching_timeout = 86400  
  
# 使用 JSON 文件作为缓存后端  
fact_caching = jsonfile  
fact_caching_connection = /tmp/ansible_fact_cache
```

使用 `fact_caching_connection` 配置项来指定 Ansible 应该保存包含 fact 的 JSON 文件到哪个目录。如果这个目录不存在，Ansible 将会创建它。

Ansible 使用文件修改时间来决定是否已经发生了 fact 缓存过期。

Redis fact 缓存后端

Redis 是一个流行的 key-value 数据存储，它经常被用于缓存。想要启用使用 Redis 后端的 fact 缓存，你需要进行如下操作。

1. 在你的控制主机上安装 Redis。
2. 确保 Redis 服务在控制主机上运行。
3. 安装 Python Redis 软件包。
4. 修改 `ansible.cfg` 来启用使用 Redis 的 fact 缓存。

例 9-9 展示了如何配置 `ansible.cfg` 来使用 Redis 做缓存后端。

例 9-9 使用 Redis fact 缓存的 `ansible.cfg`

```
[defaults]
gathering = smart

# 这里将超时时间设定为 24 小时。可以按照具体需要调整此值
fact_caching_timeout = 86400

fact_caching = redis
```

Ansible 需要在控制主机上安装 Python Redis 软件包，你可以使用 pip 来安装^{注3}。

```
$ pip install redis
```

你还必须安装 Redis 并确保它们在控制主机上运行。如果你正在使用 Mac OS X，你可以使用 Homebrew 来安装 Redis。如果你正在使用 Linux，那么可以使用原生的包管理工具来安装 Redis。

Memcached fact 缓存后端

170

Memcached 是另一个流行的 key-value 数据存储并且常用于缓存。想要启用使用 Memcached 后端的 fact 缓存，你需要如下操作。

注 3： 你可能会需要使用 sudo 或者激活 virtualenv，具体情况取决于你在控制主机上如何安装 Ansible。

1. 在你的控制主机上安装 Memcached。
2. 确保 Memcached 服务在你的控制主机上运行。
3. 安装 Python Memcached 软件包。
4. 修改 *ansible.cfg* 来启用使用 Memcached 的 fact 缓存。

例 9-10 展示了如何配置 *ansible.cfg* 来使用 Memcached 做缓存后端。

例9-10 使用Memcached。fact缓存的ansible.cfg

```
[defaults]
gathering = smart

# 这里将超时时间设定为 24 小时。可以按照具体需要调整此值
fact_caching_timeout = 86400

fact_caching = memcached
```

Ansible 需要在控制主机上安装 Python Memcached 软件包，你可以使用 pip 来安装它。你可能需要使用 sudo 或者激活一个 virtualenv，这取决于你在控制主机上是如何安装 Ansible 的。

```
$ pip install python-memcached
```

你还必须在你的控制主机上安装 Memcached 并确保它处于运行状态。如果你正在使用 Mac OS X，你可以使用 Homebrew 来安装 Memcached。如果你正在使用 Linux，那么可以使用原生包管理工具来安装 Memcached。

关于 fact 缓存的更多信息，请查阅官方文档 (http://docs.ansible.com/ansible/playbooks_variables.html#fact-caching)。

并行性

对于每个 task，Ansible 都会并行连接到主机去执行它们。但是并不是必须并行连接到所有主机。相反 Ansible 的并行度是由一个参数控制的，这个参数的默认值是 5。你可以从两种方法中任选一个来用于改变这个参数的默认值。

你可以设置 `ANSIBLE_FORKS` 环境变量，如例 9-11 所示。

171 例9-11 设置ANSIBLE_FORKS

```
$ export ANSIBLE_FORKS = 20
$ ansible-playbook playbook.yml
```

你可以修改 Ansible 配置文件 (*ansible.cfg*)：在 defaults 区段设置 `forks` 选项，如例 9-12 所示。

例9-12 通过*ansible.cfg*配置并发数量

```
[defaults]
forks = 20
```

加速模式

Ansible 支持一种名为加速模式 (*accelerated mode*) 的连接模式。这个特性要早于 pipelining，而且官方文档建议使用 pipelining 而不是加速模式，除非你的环境所限无法使用 pipelining。关于加速模式的详细信息还是查阅官方文档 (http://docs.ansible.com/ansible/playbooks_acceleration.html) 吧。

火球模式

火球模式 (*Fireball mode*) 是一个已被废弃的曾用于提高性能的 Ansible 特性。它已经被加速模式替代了。

你现在应该了解了如何配置 SSH Multiplexing、pipelining、fact 缓存和并发性来让你的 playbook 运行得更快。第 10 章，我们将会讨论编写你自己的 Ansible 模块的话题。

自定义模块

有时候，你会想要执行一个对于 command 或 shell 模块来说太过复杂的 task，并且也没有相应的模块可以完成你想做的事情。在这种情况下，你可能会想要编写你自己的模块。

以前我曾经编写过在网络地址转换（NAT）网关后获取公网 IP 地址的自定义模块，还编写过在 OpenStack 部署中初始化数据库的自定义模块。我还曾经考虑过要编写一个生成自签发 TLS 证书的自定义模块，尽管我并没有抽出时间去写。

自定义模块的另一个常见运用是在你想要通过 REST API 与第三方服务相互联系时。例如，GitHub 提供了叫作 Releases 的服务。这个服务可以让你把二进制文件附加到仓库中，这个服务就是通过 GitHub 的 API 来提供的。如果你的部署任务需要下载一个附加到 GitHub 私有仓库中的二进制文件，那么这部分逻辑就很适合在你的自定义模块中实现。

范例：检测远程服务器是否可达

比如说我们想要检测是否可以通过特定的端口连接到远程服务器。如果不能连接到远程服务器，那么我们希望 Ansible 将其看作一个错误并停止 play 的运行。



本章中我们将会开发的自定义模块是 `wait_for` 模块的简化版本。

使用 `script` 模块替代编写自己的模块

回忆一下我们在例 6-16 中如何使用 `script` 模块在远程主机上运行自定义脚本。有时候

使用 `script` 模块要比编写一个完整的 Ansible 模块简单得多。

我喜欢将这类脚本放到 `scripts` 文件夹中与 `playbook` 保存在一起。例如，我们可以创建一个名为 `playbooks/scripts/can_reach.sh` 的脚本文件。这个脚本可以接收主机的名字、需要连接的端口号以及尝试连接的超时时间作为参数。

```
can_reach.sh www.example.com 80 1
```

我们创建的这个脚本的内容如例 10-1 所示。

例 10-1 `can_reach.sh`

```
#!/bin/bash
host=$1
port=$2
timeout=$3

nc -z -w $timeout $host $port
```

我们可以使用如下方法调用它：

```
- name: run my custom script
  script: scripts/can_reach.sh www.example.com 80 1
```

切记你的脚本就像 Ansible 模块一样会在远程主机上执行。因此，你脚本中需要的任何程序都必须事先在远程主机上安装好。例如，你可以使用 Ruby 编写你的脚本，前提是远程主机上已经安装好 Ruby 并且脚本的第一行如下所示，正确地调用了 Ruby 解释器：

```
#!/usr/bin/ruby
```

模块形式的 `can_reach`

接下来，我们来将 `can_reach` 实现成真正的 Ansible 模块。这个模块可以被这样调用：

```
- name: check if host can reach the database server
  can_reach: host=db.example.com port=5432 timeout=1
```

这将会检测是否可以和 `db.example.com` 主机的 5432 端口建立 TCP 连接。如果不能建立连接，将在 1s 后超时。

我们将在本章余下部分都使用这个范例。

自定义模块该放到哪里

Ansible 会在 playbook 的 *library* 目录下寻找自定义模块。在我们的范例中，我们将 playbook 放到 *playbooks* 目录下，所以我们将自定义模块存入 *playbooks/library/can_reach*。

Ansible 如何调用模块

在真正开始实现模块之前，我们从头回顾一下 Ansible 是如何调用模块的。调用模块时，Ansible 执行如下操作。

1. 生成带有参数的独立 Python 脚本（仅限 Python 模块）。
2. 将模块复制到远程主机。
3. 在远程主机上创建参数文件（仅限非 Python 模块）。
4. 在远程主机上调用模块并传入参数文件作为参数。
5. 分析模块的标准输出。

让我们再详细分析一下其中的每一个步骤。

生成带有参数的独立 Python 脚本（仅限 Python 模块）

如果模块是使用 Python 编写的并且使用了 Ansible 提供的辅助代码（将会在稍后描述），那么 Ansible 将会生成注入了辅助代码和模块参数的自包含 Python 脚本。

将模块复制到远程主机

Ansible 将把生成的 Python 脚本（对于基于 Python 的模块）或者本地文件 *playbooks/library/can_reach*（对于非基于 Python 的模块）复制到远程主机中的临时目录下。如果你使用 *ubuntu* 用户接入远程主机，Ansible 将会把文件复制到如下路径：

```
/home/ubuntu/.ansible/tmp/ansible-tmp-1412459504.14-47728545618200/can_reach
```

在远程主机上创建参数文件（仅限非 Python 模块）

如果模块不是使用 Python 编写的，Ansible 将会在远程主机上创建一个文件，文件名如下所示：

```
/home/ubuntu/.ansible/tmp/ansible-tmp-1412459504.14-47728545618200/arguments
```

如果我们像这样调用模块：

```
[176] - name: check if host can reach the database server  
can_reach: host=db.example.com port=5432 timeout=1
```

那么参数文件将会包含如下内容：

```
host=db.example.com port=5432 timeout=1
```

我们可以告诉 Ansible 使用 JSON 格式为模块生成参数文件，只需要在 *playbooks/library/can_reach* 文件中添加如下一行即可：

```
# WANT_JSON
```

如果我们的模块配置为接收 JSON 输入，那么参数文件的内容类似如下：

```
{"host": "www.example.com", "port": "80", "timeout": "1"}
```

调用模块

Ansible 将会调用模块并把参数文件作为参数传入。如果是基于 Python 的模块，Ansible 会像如下这样执行它（将 */path/to/* 替换为实际路径）：

```
/path/to/can_reach
```

如果是非基于 Python 的模块，Ansible 将会查看模块的第 1 行来决定使用哪个解释器并像这样执行：

```
/path/to/interpreter /path/to/can_reach /path/to/arguments
```

假设 *can_reach* 模块是使用 Bash 脚本实现的并且第 1 行是：

```
#!/bin/bash
```

那么 Ansible 将会像下面这样做：

```
/bin/bash /path/to/can_reach /path/to/arguments
```

但即使这样也并不严格正确。Ansible 实际上会这样做：

```
/bin/sh -c 'LANG=en_US.UTF-8 LC_CTYPE=en_US.UTF-8 /bin/bash /path/to/can_  
reach \  
/path/to/arguments; rm -rf /path/to/ >/dev/null 2>&1'
```

你可以通过传递 *-vvv* 参数给 *ansible-playbook* 来看到 Ansible 调用确切的命令。

预期的输出

Ansible 要求模块输出 JSON 格式。例如：

```
{'changed': false, 'failed': true, 'msg': 'could not reach the host'}
```



在 1.8 版本之前，Ansible 支持一种简写输出格式，也被称为 *baby JSON*。*baby JSON* 的格式类似 `key=value` 的形式。在 1.8 版本中，Ansible 放弃对这个格式的支持。如我们稍后会看到的，如果你使用 Python 编写你的模块，Ansible 提供了一些辅助方法来帮助你简单地生成 JSON 输出。

177

Ansible 预期的输出变量

你的模块可以返回任何你喜欢的变量，但是 Ansible 会对几个必须返回的变量做特殊处理。

changed

所有的 Ansible 模块都应该返回 `changed` 变量。`changed` 变量是表示模块的执行是否会导致主机状态改变的布尔型变量。在 Ansible 运行的时候，不管状态是否发生改变，它都会把这个变量显示在输出中。如果 task 中含有 `notify` 语句来通知 handler，那么只有 `changed` 为 `true` 的时候通知才会被触发。

failed

如果模块未能执行完成，那么它应该返回 `failed=true`。Ansible 将会把这个 task 看作执行失败并且不会对发生失败的主机继续执行接下来的 task，除非这个 task 含有 `ignore_errors` 或者 `failed_when` 语句。

如果模块执行成功，你既可以返回 `failed=false`，也可以直接忽略这个变量。

msg

`msg` 变量用来添加描述模块失败原因的描述信息。

如果 task 失败了并且模块返回了 `msg` 变量，那么 Ansible 将会与其他变量稍有不同地输出这个变量。例如，一个模块返回：

```
{"failed": true, "msg": "could not reach www.example.com:81"}
```

那么在 Ansible 执行这个 task 失败的时候将会输出这样的内容：

```
failed: [vagrant1] => {"failed": true}
```

```
msg: could not reach www.example.com:81
```

178 使用 Python 来实现模块

如果你使用 Python 来实现你的自定义模块，Ansible 提供了叫作 `AnsibleModule` 的 Python 类来简化你的如下工作。

- 解析输入。
- 使用 JSON 格式返回输出。
- 调用外部程序。

实际上，在编写 Python 模块时，Ansible 将会把参数直接注入生成的 Python 文件中，而不需要你去解析单独的参数文件。我们将在本章后面内容讨论它是如何工作的。

我们将会通过创建一个 `can_reach` 文件来创建我们的 Python 模块。我们先看它的全部实现，然后再分解讲解（见例 10-2）。

例10-2 can_reach

```
#!/usr/bin/python

def can_reach(module, host, port, timeout):
    nc_path = module.get_bin_path('nc', required=True)❶
    args = [nc_path, "-z", "-w", str(timeout),
            host, str(port)]
    (rc, stdout, stderr) = module.run_command(args)❷
    return rc == 0

def main():
    module = AnsibleModule(❸
        argument_spec=dict(❹
            host=dict(required=True), ❺
            port=dict(required=True, type='int'),
            timeout=dict(required=False, type='int', default=3)❻
        ),
        supports_check_mode=True ❼
    )

    # 在检测模式下不做具体操作
    # 因为这个模块永远不会改变系统的状态，所以
    # 我们直接返回 changed=False
    if module.check_mode: ❽
        module.exit_json(changed=False)❾
```

```

host = module.params['host'] ⑩
port = module.params['port']
timeout = module.params['timeout']

if can_reach(module, host, port, timeout):
    module.exit_json(changed=False)
else:
    msg = "Could not reach %s:%s" % (host, port)
    module.fail_json(msg=msg) ⑪

from ansible.module_utils.basic import * ⑫
main()

```

179

- ① 获取外部程序的路径。
- ② 调用外部程序。
- ③ 实例化 AnsibleModule 辅助类。
- ④ 指定一系列合法的参数。
- ⑤ 一个必选参数。
- ⑥ 一个具有默认值的可选参数。
- ⑦ 指定本模块支持检测模式。
- ⑧ 测试一下看看模块是否运行于检测模式。
- ⑨ 成功退出，传递返回值。
- ⑩ 提取参数。
- ⑪ 退出失败，传递错误信息。
- ⑫ “import” AnsibleModule 辅助类。

解析参数

通过解析上面的范例很容易理解 AnsibleModule 处理参数的方法。回忆一下，我们的模块是像这样被调用的：

```
- name: check if host can reach the database server
  can_reach: host=db.example.com port=5432 timeout=1
```

让我们假设 host 参数和 port 参数是必选的，而 timeout 参数是一个默认值为 3s 的可选参数。

你通过传入 argument_spec 参数实例化 AnsibleModule 对象。argument_spec 是一个字典，它的 key 是参数名而 value 是包含参数信息的字典。

```
180 >     module = AnsibleModule(  
              argument_spec=dict(  
                  ...
```

在我们的范例中，我们声明了一个名为 host 的必选参数。如果我们在 task 中使用模块时没有传递这个参数，Ansible 将会报告一个错误。

```
          host=dict(required=True),
```

名为 timeout 的参数是可选的。除非你指定别的类型，不然 Ansible 会假定参数是字符串。我们的 timeout 变量是一个整数，所以我们指定类型为 int，为了让 Ansible 自动将它转为 Python 数字。如果 timeout 没有被指定，模块会假定它的值为 3：

```
          timeout=dict(required=False, type='int', default=3)
```

AnsibleModule 构造函数使用的参数不同于 argument_spec。在前面的范例中，我们为它添加了如下参数：

```
          supports_check_mode = True
```

这意味着我们的模块支持检测模式。我们将在本章稍后的内容解释检测模式。

访问参数

一旦你声明了 AnsibleModule 对象，你就可以像下面这样通过 params 字典来获取参数的值：

```
module = AnsibleModule(...)  
  
host = module.params["host"]  
port = module.params["port"]  
timeout = module.params["timeout"]
```

导入 AnsibleModule 辅助类

在接近模块的结尾，你会看到这个 import 语句：

```
from ansible.module_utils.basic import *
```

如果你曾经写过 Python 脚本，你很可能经常在脚本的开始看到 import，而不是结尾。然而，这其实是个假的 import 语句。它看起来很像传统的 Python import，但是行为有些不

太一样。

模块中的 import 语句和常规的行为不太一样，是因为 Ansible 只复制一个需要执行的单一 Python 文件到远程主机上。Ansible 通过将导入的代码直接包含到生成的 Python 文件中（有点像 C/C++ 中 #include 语句的行为）来模拟传统 Python 的 import 行为。

因为 Ansible 会使用代码替换 import 语句，这会导致你所编写模块中的行号与生成的 Python 文件中的行号并不一样。把 import 语句放到文件的底部，那么在它之上的所有行在模块中与在生成的文件中的行号就都一样了。在需要分析包含行号的 Python traceback 的时候，你就会发现这么做的意义。

<181

因为 import 语句的行为与传统 Python import 并不相同，你不应再像例 10-3 那样显式导入某个类了。尽管在传统的 Python 开发中，显式导入是个好习惯。

例 10-3 显式导入（千万别这么干）

```
from ansible.module_utils.basic import AnsibleModule
```

如果你使用显式导入，那么你将无法使用 Ansible 模块调试脚本。这是因为这些调试脚本会寻找包括 “*” 的指定字符串，并且如果它们找不到的话会报错失败。



早期版本的 Ansible 使用下面这行代替 import 语句来标记 Ansible 需要插入生成的辅助代码的位置。

```
#<<INCLUDE_ANSIBLE_MODULE_COMMON>>
```

参数选项

对于 Ansible 模块的每一个参数，你都可以指定一些选项，如表 10-1 所示。

表 10-1 参数选项

选项	描述
required	如果值是 true，那么该参数是必选参数
default	可选参数默认值
choices	参数可能值的列表
aliases	另一个你可以用来作为此参数别名的名字
type	参数类型。允许使用如下值：'str'、'list'、'dict'、'bool'、'int'、'float'

required

`required` 选项是唯一一个你应该对每个参数都指定的选项。如果它的值是 `true`，那么在用户没有指定该参数的情况下，Ansible 将会返回一个错误。

182> 在我们的 `can_reach` 模块范例中，`host` 和 `port` 是必选的，而 `timeout` 不是必选的。

default

对于设置了 `required=False` 的参数，你一般应该为其指定一个默认值。在我们的范例中：

```
timeout=dict(required=False, type='int', default=3)
```

如果用户像下面这样调用模块：

```
can_reach: host=www.example.com port=443
```

那么 `module.params["timeout"]` 将会包含值 3。

choices

`choices` 选项允许你将合法的参数限定在一个预定义的列表中。

思考下面范例中的 `distro` 参数：

```
distro=dict(required=True, choices=['ubuntu', 'centos', 'fedora'])
```

如果用户传递了一个未在列表中的参数，比如：

```
distro=suse
```

这将会导致 Ansible 抛出一个错误。

aliases

`aliases` 选项允许你使用不同的名字来指代相同的参数。例如，思考一下 `apt` 模块中的 `package` 参数：

```
module = AnsibleModule(
    argument_spec=dict(
        ...
        package = dict(default=None, aliases=['pkg', 'name'], type='list'),
    )
)
```

因为 `pkg` 和 `name` 是 `package` 参数的别名，所以下面的调用都是等价的：

```
- apt: package=vim  
- apt: name=vim  
- apt: pkg=vim
```

type

type 选项使你可以指定参数的类型。默认情况下，Ansible 假定所有参数都是字符串。

但是，你可以为参数指定类型，然后 Ansible 会将其转换到要求的类型。支持的类型有：183

- *str* (字符串)
- *list* (列表)
- *dict* (字典)
- *bool* (布尔型)
- *int* (整型)
- *float* (浮点型)

在我们的范例中，我们将 port 参数指定为 int 类型：

```
port=dict(required=True, type='int'),
```

当我们像下面这样从 params 字典中访问它时：

```
port = module.params['port']
```

port 变量的值将是一个整数。如果我们没有在声明 port 变量的时候将类型指定为 int，那么 module.params['port'] 的值将是一个字符串而不是 int。

列表是使用逗号分隔的。例如，我们有一个名为 foo 的模块，并且该模块有一个名为 colors 的列表参数：

```
colors=dict(required=True, type='list')
```

然后，你像这样传递一个 list：

```
foo: colors=red,green,blue
```

对于字典，你既可以使用 key=value 的形式并使用逗号分隔多对，也可以使用内联 JSON。

例如，如果你有一个名为 bar 的模块，并且该模块有一个名为 tags 的 dict 参数：

```
tags=dict(required=False, type='dict', default={})
```

那么，你可以像这样传递参数：

```
- bar: tags=env=staging,function=web
```

或者你也可以像这样传递参数：

```
- bar: tags={"env": "staging", "function": "web"}
```

Ansible 官方文档使用术语 *complex args* 作为传递给模块参数的列表和字典的统称。在第 102 页的“task 中的复杂参数：稍微跑个题”中详细说明了如何在 playbook 中传递这些类型的变量。

184

AnsibleModule 的初始化参数

AnsibleModule 的初始化方法有许多参数。唯一必须指定的参数就是 argument_spec。表 10-2 为 AnsibleModule 初始化参数。

表10-2 AnsibleModule初始化参数

参数	默认	描述
argument_spec	(None)	包含参数信息的字典
bypass_checks	False	如果值为 true，不检测任何参数约束
no_log	False	如果值为 true，此模块的任何行为都不记录日志
check_invalid_arguments	True	如果值为 true，用户传入未知参数时返回错误
mutually_exclusive	None	互斥参数组成的列表
required_together	None	必须一起出现的参数组成的列表
required_one_of	None	必须至少存在一个的参数组成的列表
add_file_common_args	False	支持 file 模块的参数
supports_check_mode	False	如果值为 true，表示模块支持检测模式

argument_spec

如在之前章节中的描述，这是一个包含本模块合法参数描述的字典。

no_log

当 Ansible 在主机上执行模块的时候，模块会将日志输出到 syslog 中。在 Ubuntu 中，syslog 的位置在 /var/log/syslog。

日志的输出类似下面这样：

```
Sep 28 02:31:47 vagrant-ubuntu-trusty-64 ansible-ping: Invoked with data=None
```

```

Sep 28 02:32:18 vagrant-ubuntu-trusty-64 ansible-apt: Invoked with dpkg_
options=force-confdef,force-confold upgrade=None force=False name=nginx
package=['nginx'] purge=False state=installed update_cache=True default_
release=None install_recommends=True deb=None cache_valid_time=None Sep 28
02:33:01 vagrant-ubuntu-trusty-64 ansible-file: Invoked with src=None
original_basename=None directory_mode=None force=False remote_src=None
selevel=None seuser=None recurse=False serole=None content=None
delimiter=None state=directory diff_peek=None mode=None regexp=None
owner=None group=None path=/etc/nginx/ssl backup=None validate=None
setype=None
Sep 28 02:33:01 vagrant-ubuntu-trusty-64 ansible-copy: Invoked with src=/
home/vagrant/.ansible/tmp/ansible-tmp-1411871581.19-43362494744716/source
directory_mode=None force=True remote_src=None dest=/etc/nginx/ssl/nginx.key
selevel=None seuser=None serole=None group=None content=NOT_LOGGING_PARAMETER
setype=None original_basename=nginx.key delimiter=None mode=0600 owner=root
regexp=None validate=None backup=False
Sep 28 02:33:01 vagrant-ubuntu-trusty-64 ansible-copy: Invoked with src=/
home/vagrant/.ansible/tmp/ansible-tmp-1411871581.31-95111161791436/source
directory_mode=None force=True remote_src=None dest=/etc/nginx/ssl/nginx.crt
selevel=None seuser=None serole=None group=None content=NOT_LOGGING_PARAMETER
setype=None original_basename=nginx.crt delimiter=None mode=None owner=None
regexp=None validate=None backup=False

```

如果模块接收敏感信息作为参数，你可能希望禁用日志记录。

想要将模块配置为不写入syslog，在AnsibleModule初始化时需要将no_log=True作为参数传入。

check_invalid_arguments

默认情况下，Ansible将会验证所有用户传给模块的参数是否是合法参数。你可以在AnsibleModule初始化时将check_invalid_arguments=False作为参数传入来禁用这个检查。

mutually_exclusive

mutually_exclusive参数是一个由参数组成的列表，列表中的参数不能在一次模块调用中一起被指定。

例如，lineinfile模块允许你添加一行到文件中。你可以使用insertbefore参数来指定插入在哪一行之前，或者使用insertafter参数指定插入在哪一行之后，但是你不能两个都指定。

因此，这个模块像以下这样指定这两个参数是互斥的：

```
mutually_exclusive=[['insertbefore', 'insertafter']]
```

required_one_of

required_one_of 参数是一个由参数组成的列表，这些参数中至少一个必须传给模块。

例如，用于安装 Python 软件包的 pip 模块，既接收软件包的名字，又接收包含一系列软件包的需求文件的名字。模块像下面这样指定这些必选参数：

```
required_one_of=[['name', 'requirements']]
```

186

add_file_common_args

很多模块都创建或者修改文件。用户常常希望指定结果文件的一些属性，就好像所有者、所属组以及文件权限。

你可以像下面这样调用 file 模块来设置这些参数：

```
- name: download a file
  get_url: url=http://www.example.com/myfile.dat dest=/tmp/myfile.dat

- name: set the permissions
  file: path=/tmp/myfile.dat owner=ubuntu mode=0600
```

为了简洁，Ansible 允许你指定一个模块接收与 file 模块一样的参数。这样你就可以直接把相关的参数传递给创建或者修改文件的模块来设置文件属性。例如：

```
- name: download a file
  get_url: url=http://www.example.com/myfile.dat dest=/tmp/myfile.dat \
  owner=ubuntu mode=0600
```

下面的配置可以指定模块应该接收这些参数：

```
add_file_common_args=True
```

AnsibleModule 模块提供了辅助方法协助你处理这些参数。

load_file_common_arguments 方法接收参数字典作为参数，并且返回一个包含所有与设置文件属性相关的参数组成的参数字典。

set_fs_attributes_if_different 方法接收文件参数字典及一个表示主机状态是否已经发生改变的布尔型值。这个方法将设置文件属性作为副作用，并且在主机状态发生改变时返回 true（不论初始参数就是 true，还是这个方法作为副作用的一部分改变了文件）。

如果你正在使用普通文件参数，那么不要显式指定参数。想要在你的代码中访问这些属性的话，像下面这样使用辅助方法来提取参数并设置文件属性：

```
module = AnsibleModule(  
    argument_spec=dict(  
        dest=dict(required=True),  
        ...  
    ),  
    add_file_common_args=True  
)  
  
# 如果模块导致主机状态改变，将“changed”的值置为True  
changed = do_module_stuff(param)  
file_args = module.load_file_common_arguments(module.params)  
  
changed = module.set_fs_attributes_if_different(file_args, changed)  
module.exit_json(changed=changed, ...)
```

187



Ansible 假定你的模块包含名为 `path` 和 `dest` 的参数，并且这些参数中包含到文件的路径。

bypass_checks

在 Ansible 模块执行之前，它会先检查所有的参数约束是否都满足了。如果没有全部满足的话就会返回一个错误。这些参数约束包括：

- 没有互斥参数同时出现。
- 标记为 `required` 选项的参数都存在。
- 被 `choices` 选项限制的参数有合法的值。
- 指定 `type` 的参数的值的类型与指定的 `type` 相符。
- 标记为 `required_together` 的参数同时存在。
- `required_one_of` 列表中的参数至少有一个存在。

你可以通过设置 `bypass_checks=True` 来禁用所有这些检查。

返回成功或失败

可以使用 `exit_json` 方法来返回成功。你应该总是将 `changed` 作为参数返回。除此之外，返回有意义信息的 `msg` 是一个良好的实践：

```
module = AnsibleModule(...)  
...  
module.exit_json(changed=False, msg="meaningful message goes here")
```

使用 `fail_json` 方法来宣告失败。你应该总是返回 `msg` 参数来向用户解释失败的原因：

```
module = AnsibleModule(...)  
...  
module.fail_json(msg="Out of disk space")
```

188 调用外部命令

`AnsibleModule` 类提供了 `run_command` 方法用于便利地调用外部程序。这个方法是对原生 Python 模块 `subprocess` 的一个封装。它接收如表 10-3 所示的参数。

表10-3 `run_command`的参数

参数	类型	默认值	描述
args (默认)	字符串或者字符串列表	(None)	需要执行的命令 (详见后面的章节)
check_rc	布尔型	False	如果值为 <code>true</code> , 在命令返回非零值的时候调用 <code>fail_json</code>
close_fds	布尔型	True	将 <code>close_fds</code> 作为参数传给 <code>subprocess.Popen</code>
executable	字符串 (程序的路径)	None	将 <code>executable</code> 作为参数传给 <code>subprocess.Popen</code>
data	字符串	None	将数据通过 <code>stdin</code> 传递给子进程
binary_data	布尔型	False	如果值为 <code>false</code> 并且存在 <code>data</code> 参数, 那么 Ansible 会在发送 <code>data</code> 后往 <code>stdin</code> 中添加一个换行符
path_prefix	字符串 (路径的列表)	None	需要预置到 PATH 环境变量的路径组成的冒号分隔的列表
cwd	字符串 (目录路径)	None	如果该参数被指定, Ansible 将在执行前切换到这个目录下
use_unsafe_shell	布尔型	False	详见后面章节

如果 `args` 像例 10-4 这样作为列表传入, 那么 Ansible 将会使用 `shell=False` 参数调用 `subprocess.Popen`。

例10-4 作为列表传递args参数

```
module = AnsibleModule(...)  
...  
module.run_command(['/usr/local/bin/myprog', '-i', 'myarg'])
```

如果 `args` 参数像例 10-5 这样作为字符串传入，那么 Ansible 的行为将取决于 `use_unsafe_shell` 的值。如果 `use_unsafe_shell` 的值为 `false`，那么 Ansible 会将 `args` 分割成列表然后使用 `shell=False` 参数调用 `subprocess.Popen`。如果 `use_unsafe_shell` 的值为 `true`，Ansible 会将 `args` 作为字符串传递给使用 `shell=True` 参数调用的 `subprocess.Popen`。^{注1}

189

例 10-5 作为字符串传递 args 参数

```
module = AnsibleModule(...)  
...  
module.run_command('/usr/local/bin/myprog -i myarg')
```

检测模式 (dry run)

Ansible 支持“检测模式 (check mode)”，通过向 `ansible-playbook` 传递 `-C` 或者 `--check` 参数来启用这个模式。它和其他很多工具支持的“dry run”模式类似。

当 Ansible 在检测模式下运行 playbook 时，它并不在主机上进行任何变更。相对地，它会直接报告每个 task 是否会变更主机的状态，不进行任何变更直接返回成功，或者返回一个错误。



支持检测模式

Ansible 模块必须被明确地配置支持检测模式。如果你将要编写你自己的模块，我建议你支持检测模式，这样你的模块才会真正成为 Ansible 大家庭中的一员。

如例 10-6 所示在 `AnsibleModule` 初始化方法中将 `supports_check_mode` 设置为 `true` 就可以告诉 Ansible 你的模块支持检测模式。

例 10-6 向 Ansible 告知模块支持检测模式

```
module = AnsibleModule(  
    argument_spec=dict(...),  
    supports_check_mode=True)
```

你的模块应当具有确认检测模式是否已启用的逻辑。确认的方法是如例 10-7 所示检查

注 1：想要获取更详细的关于 Python 标准库 `subprocess.Popen` 类的文档，请访问在线文档 (<https://docs.python.org/2/library/subprocess.html#subprocess.Popen>)。

`AnsibleModule` 对象的 “`check_mode`”^{注2} 属性的值。你仍然需要像正常情况一样调用 `exit_json` 和 `fail_json` 方法。

190 例 10-7 检查是否已启用检测模式

```
module = AnsibleModule(...)

...
if module.check_mode:
    # 检查一下该模块是否会造成状态的改变
    would_change = would_executing_this_module_change_something()
    module.exit_json(changed=would_change)
```

当运行在检测模式时，确保你的模块不会修改主机的状态这个关键的任务就交给你了。对，就是你！伟大的模块作者！

文档化你的模块

你应当遵循 Ansible 项目标准来文档化你的模块，这会帮助你正确生成你的模块的 HTML 文档并使 `ansible-doc` 程序可以展示你模块的文档。Ansible 使用基于 YAML 的特殊语法来实现模块文档化。

具体方法为：在你的模块起始的附近，定义一个包含文档的字符串变量 `DOCUMENTATION`，以及一个包含用法示例的字符串变量 `EXAMPLES`。

例 10-8 列出了我们 `can_reach` 模块的文档区段范例。

例 10-8 模块文档范例

```
DOCUMENTATION = '''

---
module: can_reach
short_description: Checks server reachability
description:
  - Checks if a remote server can be reached
version_added: "1.8"
options:
  host:
    description:
      - A DNS hostname or IP address
    required: true
  port:
    description:
      - The TCP port number
```

注 2： 好家伙，这句话有好多个检查！（作者原意是这一句话中用了好多单词“check”。翻译时为了便于理解将动词和名词翻译为不同的中文词，并且断为两句。不过看起来还是比较圆。——译者注）

```

required: true
timeout:
  description:
    - The amount of time try to connecting before giving up, in seconds
required: false
default: 3
flavor:
  description:
    - This is a made-up option to show how to specify choices.
required: false
choices: ["chocolate", "vanilla", "strawberry"]
aliases: ["flavour"]
default: chocolate
requirements: [netcat]
author: Lorin Hochstein
notes:
  - This is just an example to demonstrate how to write a module.
  - You probably want to use the M(wait_for) module instead.
...

```

```

EXAMPLES = '''
# 检查一下 SSH 是否在运行，使用默认的超时时间
- can_reach: host=myhost.example.com port=22

# 检查一下 Postgres 是否在运行并设置一个超时时间
- can_reach: host=db.example.com port=5432 timeout=1
...

```

Ansible 在文档中支持有限的标记。表 10-4 列出了 Ansible 文档工具支持的标记语法以及推荐使用标记的时机。

表10-4 文档标记

类型	语法范例	何时使用
URL	U(http://www.example.com)	URLs
模块	M(apt)	模块名
斜体字	I(port)	参数名
等宽字	C(/bin/bash)	文件或选项名

已有的 Ansible 模块是最好的文档范例来源。

调试你的模块

在 GitHub 中的 Ansible 代码仓库中包含几个允许你直接在本地主机上调用模块的脚本。这些脚本并不需要使用 `ansible` 或者 `ansible-playbook` 命令来运行。

复制 Ansible 代码仓库：

```
$ git clone https://github.com/ansible/ansible.git --recursive
```

192 设置你的环境变量以便于调用模块：

```
$ source ansible/hacking/env-setup
```

调用你的模块：

```
$ ansible/hacking/test-module -m /path/to/can_reach -a "host=example.com  
port=81"
```

因为 `example.com` 并没有监听 81 端口的服务，所以我们的模块应该会失败并返回有意义的错误信息，并且它确实是这样运作的：

```
* including generated source, if any, saving to:  
  /Users/lorinhighstein/.ansible_module_generated  
* this may offset any line numbers in tracebacks/debuggers!  
*****  
RAW OUTPUT  
{"msg": "Could not reach example.com:81", "failed": true}  
*****  
PARSED OUTPUT  
{  
    "failed": true,  
    "msg": "Could not reach example.com:81"  
}
```

按照上面输出的建议，当你运行 `test-module` 的时候，Ansible 会生成 Python 脚本并将其复制到 `~/.ansible_module_generated` 下。这是一个独立的脚本，你可以在你希望的时候直接执行它。调试脚本将会替换如下这行：

```
from ansible.module_utils.basic import *
```

替换为 `lib/ansible/module_utils/basic.py` 文件的内容。这个文件可以在 Ansible 的代码仓库中找到。

这个文件不接收任何参数。所以 Ansible 将参数像如下这样直接插入到文件中：

```
MODULE_ARGS = 'host=example.com port=91'
```

使用 Bash 实现模块

如果你要编写 Ansible 模块，我建议你使用 Python 编写它。原因正如本章前面我们所看到的，Ansible 为使用 Python 编写模块提供了辅助类。但是，你也可以使用其他语言来编写模块。也许你需要使用其他语言编写模块是因为你的模块依赖一个不是用 Python 实现的第三方库。又或者这个模块太简单了，使用 Bash 来编写是最简单的方法。又或者你只是喜欢使用 Ruby 编写脚本。

在本节，我们会使用 Bash 脚本从头到尾地完成一个实现模块的例子。它看起来会很像例 10-1 的实现。主要的区别是解析输入参数和生成 Ansible 所期望的输出。

<193

我将会使用 JSON 格式作为输入并使用名为 jq (<http://stedolan.github.io/jq/>) 的工具在命令行解析 JSON。这意味着在调用这个模块之前你将需要在主机上安装 jq。如例 10-9 所示为我们模块的完整 Bash 实现。

例 10-9 Bash 编写的 can_reach 模块

```
#!/bin/bash
# WANT_JSON

# 从文件中读取变量
host=`jq -r .host < $1`
port=`jq -r .port < $1`
timeout=`jq -r .timeout < $1`

# 检查是否可以连通主机
nc -z -w $timeout $host $port

# 基于检查结果输出成功或者失败
if [ $? -eq 0 ]; then
    echo '{"changed": false}'
else
    echo "{\"failed\": true, \"msg\": \"could not reach $host:$port\"}"
fi
```

我们在注释中添加了 WANT_JSON 来告诉 Ansible 使用 JSON 语法作为输入。

简化输入的 Bash 模块

为输入使用简化符号实现 Bash 模块是可能的。但我不建议使用这种方法，因为最简单的方法是使用内置的 `source`，但这种方法有潜在的安全风险。但是，如果你铁了心要这样做，我推荐看看 Jan-Piet Mens 的这篇 blog：“Shell scripts as Ansible modules” (<http://jpmens.net/2012/07/05/shell-scripts-as-ansible-modules/>)。

为 Bash 指定替代的位置

需要注意的是我们的模块假定 Bash 位于 `/bin/bash`。但是，并不是所有的系统都将可执行的 Bash 放在那个位置。你可以通过在把 Bash 安装在别处的主机上设置 `ansible_bash_interpreter` 变量来让 Ansible 到其他地方寻找 Bash 解释器。
194>

举例说明，比如说你有一台名为 `fileserver.example.com` 的 FreeBSD 主机，并且这台主机的 Bash 安装在 `/usr/local/bin/bash`。你可以通过创建包含如下内容的文件 `host_vars/fileserver.example.com` 来创建主机变量：

```
ansible_bash_interpreter: /usr/local/bin/bash
```

然后，当 Ansible 在 FreeBSD 主机上调用这个模块的时候，它将会使用 `/usr/local/bin/bash` 替代 `/bin/bash`。

Ansible 通过查找“she-bang”(`#!`)然后获取第 1 个元素的 basename 来决定使用哪个解释器。在我们的范例中，Ansible 会看到这样一行：

```
#!/bin/bash
```

Ansible 会寻找 `/bin/bash` 的 basename，也就是 `bash`。然后它会使用 `ansible_bash_interpreter` 变量，前提是用户定义了它。



基于 Ansible 寻找解释器的机制，如果你的 she-bang 叫作 `/usr/bin/env`，例如：

```
#!/usr/bin/env bash
```

Ansible 将会错误地识别解释器为 `env`，因为它会针对 `/usr/bin/env` 调用 basename 来识别解释器。

关键点是：不要在 she-bang 中调用 `env`。相对地，显式地指定解释器的位置并在需要的时候使用 `ansible_bash_interpreter` 重写（或者等效）。

范例模块

学习编写 Ansible 模块最好的办法是阅读 Ansible 随附模块的源代码。可以在 GitHub 上查看它们：`modules core` (<https://github.com/ansible/ansible-modules-core>) 和 `modules extras` (<https://github.com/ansible/ansible-modules-extras>)。

在本章中，我们介绍了如何使用 Python 和其他语言编写模块，以及如何利用 `script` 模块避免自己编写完整的模块。如果你真的编写了一个模块，我支持你将它提交到 Ansible 主项目中。

Vagrant

Vagrant 是一个非常棒的用于测试 Ansible playbook 的环境，这也是我在本书中一直使用它的原因，同时也是我经常使用 Vagrant 来测试我自己的 Ansible Playbook 的原因。

Vagrant 并不仅仅用于测试配置管理脚本。它最初设计用于创建可重复的开发环境。如果你曾经加入一个新的软件团队，并且曾花费若干天来寻找适合安装在你的笔记本电脑中，用于运行内部产品的开发版本的软件的话，你会发现 Vagrant 就是解救你痛苦的良药。Ansible playbook 是说明如何配置 Vagrant 虚拟机的绝佳方法，这样你团队的新人们第一天就可以成功地将 Vagrant 运行起来。

Vagrant 有一些我们还没有利用到的对 Ansible 的内置支持。在本章中，我们将介绍对于使用 Ansible 来配置 Vagrant 虚拟机有帮助的 Vagrant 特性。

对于 Vagrant 完整的介绍超出了本书的范围。如果你想要了解更多细节，可以参阅由 Vagrant 的作者 Mitchell Hashimoto 编写的 *Vagrant : Up and Running*。



便捷的 Vagrant 配置项

Vagrant 为虚拟机提供了很多配置项，其中有两项我发现在使用 Vagrant 进行测试的时候特别好用：设置特定的 IP 地址和启用 agent forwarding。

端口转发和私有 IP 地址

当你使用 `vagrant init` 命令创建一个新的 `Vagrantfile` 的时候，默认的网络配置允许你

通过 SSH 端口经由 localhost 转发到达 Vagrant 虚拟机。对于第一台你启动的 Vagrant 虚拟机，这个端口是 2222。你随后启动的每台 Vagrant 虚拟机都将使用不同的端口进行转发。因此，默认配置下唯一访问 Vagrant 虚拟机的方法就是通过 SSH 到 localhost 的 2222 端口。Vagrant 将它转发到 Vagrant 虚拟机的 22 端口。

默认配置并不适合测试基于 Web 的应用，因为 Web 应用将会监听一些我们无法访问的端口。

这里有两个方法绕过这个问题。一个方法就是告诉 Vagrant 去建立另一个转发端口。例如，如果在 Vagrant 虚拟机里你的 Web 应用监听 80 端口，你可以配置 Vagrant 将你本机的 8000 端口转发到 Vagrant 虚拟机的 80 端口。例 11-1 展示了如何通过编辑 Vagrantfile 实现这个端口转发的配置。

例 11-1 将本地 8000 端口转发到 Vagrant 虚拟机的 80 端口

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
    # 其他配置项省略

    config.vm.network :forwarded_port, host: 8000, guest: 80
end
```

端口转发可以解决问题，但是我发现为 Vagrant 虚拟机分配它自己的 IP 地址的方法更有用。利用这种方法时，Vagrant 虚拟机使用起来更像是在使用真正的远程服务器：我可以直接连接到 Vagrant 虚拟机的 IP 地址的 80 端口而不是连接到 localhost 的 8000 端口。

一个比较简单的方法是为 Vagrant 虚拟机分配私有 IP。例 11-2 展示了如何通过编辑 Vagrantfile 将 IP 地址 192.168.33.10 分配给 Vagrant 虚拟机。

例 11-2 为 Vagrant 虚拟机分配私有 IP

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
    # 其他配置项省略

    config.vm.network "private_network", ip: "192.168.33.10"
end
```

如果我们在 Vagrant 虚拟机中的 80 端口运行一个 Web 服务器，那么我们可以通过 <http://192.168.33.10> 来访问它。

这个配置使用了 Vagrant “private network”。这意味着 Vagrant 虚拟机只能被运行 Vagrant 的机器访问。你不能在另外一台物理服务器上连接到这个 IP 地址，即使它与运行 Vagrant 的机器在同一个网络中。但是，不一样的 Vagrant 虚拟机可以访问彼此。

更多细节可以查看 Vagrant 文档中关于不同的网络配置项一节。

启用 agent forwarding

如果你正在通过 SSH 检查一个远程 Git 仓库，并且你需要使用 agent forwarding，那么你必须配置你的 Vagrant 虚拟机以在通过 SSH 连接到 agent 的时候支持 agent forwarding。可以查看例 11-3 了解如何启用它。如果想要了解更多关于 agent forwarding 的细节，请参阅附录 A。

例11-3 启用agent forwarding

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # 其他配置项省略

  config.ssh.forward_agent = true

end
```

Ansible 置备器

Vagrant 有一个置备器 (*provisioner*) 的概念。置备器是一个 Vagrant 用来在虚拟机启动后配置它们的外部工具。除了 Ansible，Vagrant 也可以使用 Shell 脚本、Chef、Puppet、Salt、CFengine，甚至是 Docker 作为置备器。

如例 11-4 所示是为使用 Ansible 作为置备器进行配置的 Vagrantfile，并明确地指定使用 *playbook.yml* playbook。

例11-4 Vagrantfile

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"

  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "playbook.yml"
  end
end
```

198

置备器何时运行

你第一次执行 `vagrant up` 的时候，Vagrant 将会执行置备器并记录下置备器已经执行。如果你停止虚拟机并再次启动，Vagrant 会记得已经运行过置备器而不会第二次运行它。

你可以像这样强制 Vagrant 对一台运行着的虚拟机运行置备器：

```
$ vagrant provision
```

你可以调用如下命令重启一台虚拟机并在完成重启后运行置备器：

```
$ vagrant reload --provision
```

相似地，你像下面这样可以启动一台已经停止的虚拟机并让 Vagrant 在启动后运行置备器：

```
$ vagrant up --provision
```

由 Vagrant 生成 inventory

当 Vagrant 运行的时候，它生成一个名为 `.vagrant/provisioners/ansible/inventory/vagrant_ansi_le_inventory` 的 Ansible inventory 文件。如例 11-5 所示为在我们的范例中这个文件的内容：

例 11-5 `vagrant_ansible_inventory`

```
# Generated by Vagrant

default ansible_ssh_host=127.0.0.1 ansible_ssh_port=2202
```

需要注意的是，它使用 `default` 作为 inventory 主机名。当为 Vagrant 置备器编写 playbook 的时候，可以指定 `hosts: default` 或者 `hosts: all`。

199 如果你的 Vagrant 环境中有多台虚拟机那就更有意思了。这种情况下，你的 `Vagrantfile` 中也包含多个虚拟机。例如，如例 11-6 所示。

例 11-6 `Vagrantfile`（多台虚拟机）

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
    vagrant1.vm.provision "ansible" do |ansible|
      ansible.playbook = "playbook.yml"
    end
  end
  config.vm.define "vagrant2" do |vagrant2|
```

```

vagrant2.vm.box = "ubuntu/trusty64"
vagrant2.vm.provision "ansible" do |ansible|
  ansible.playbook = "playbook.yml"
end
end
config.vm.define "vagrant3" do |vagrant3|
  vagrant3.vm.box = "ubuntu/trusty64"
  vagrant3.vm.provision "ansible" do |ansible|
    ansible.playbook = "playbook.yml"
  end
end
end

```

生成的 inventory 文件将如例 11-7 所示。注意，Ansible 别名（vagrant1、vagrant2、vagrant3）与在 Vagrantfile 分配给 Vagrant 虚拟机的名字相吻合。

例11-7 vagrant_ansible_inventory（多台虚拟机）

```

# Generated by Vagrant

vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201

```

并行配置

在例 11-6 中，Vagrant 会为每台虚拟机都运行一次 ansible-playbook，并且还将使用 --limit 参数使得置备器每次只针对一台虚拟机运行。

唉！可惜以这种方式运行并不能利用 Ansible 针对多台主机并行执行 task 的能力。幸好我们可以通过配置我们的 Vagrantfile 仅在最后一台虚拟机启动时运行置备器，并同时告诉 Vagrant 不要向 Ansible 传递 --limit 参数来绕过这个问题。如例 11-8 所示为修改后的 playbook。

例11-8 Vagrantfile（多台虚拟机并行配置）

```

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # 为每台虚拟机使用相同的密钥
  config.ssh.insert_key = false

  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
  end
  config.vm.define "vagrant2" do |vagrant2|

```

```
vagrant2.vm.box = "ubuntu/trusty64"
end
config.vm.define "vagrant3" do |vagrant3|
  vagrant3.vm.box = "ubuntu/trusty64"
  vagrant3.vm.provision "ansible" do |ansible|
    ansible.limit = 'all'
    ansible.playbook = "playbook.yml"
  end
end
end
```

现在，当你第一次运行 `vagrant up` 时，它只会在 3 台虚拟机全部启动后再运行 Ansible 置备器。

从 Vagrant 的视角，只有最后一台虚拟机 `vagrant3` 拥有置备器，所以执行 `vagrant provision vagrant1` 或 `vagrant provision vagrant2` 都将没有效果。

如我们在 46 页的“准备工作：创建多台 Vagrant 虚拟机”中所讨论的，Vagrant 1.7 以上版本会默认为每台虚拟机使用不同的 SSH 公钥。如果我们想要并行配置虚拟机，我们需要配置 Vagrant 虚拟机都使用相同的 SSH 公钥。这也是例 11-8 中包含如下这行的原因：

```
config.ssh.insert_key = false
```

指定群组

为 Vagrant 虚拟机分配群组的功能特别有用，特别是在你复用的 `playbook` 引用了已存在群组的情况下。在例 11-9 中，我们分配 `vagrant1` 到 `web` 群组，`vagrant2` 到 `task` 群组，`vagrant3` 到 `redis` 群组。

例 11-9 Vagrantfile（多台虚拟机并使用群组）

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # 为每台虚拟机使用相同的密钥
  config.ssh.insert_key = false

  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
  end
  config.vm.define "vagrant2" do |vagrant2|
    vagrant2.vm.box = "ubuntu/trusty64"
  end
  config.vm.define "vagrant3" do |vagrant3|
    vagrant3.vm.box = "ubuntu/trusty64"
```

```
vagrant3.vm.provision "ansible" do |ansible|
  ansible.limit = 'all'
  ansible.playbook = "playbook.yml"
  ansible.groups = {
    "web" => ["vagrant1"],
    "task" => ["vagrant2"],
    "redis" => ["vagrant3"]
  }
end
end
```

如例 11-10 所示为 Vagrant 生成的 inventory 文件。

例 11-10 vagrant_ansible_inventory (多台虚拟机并使用群组)

```
# Generated by Vagrant

vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201

[web]
vagrant1

[task]
vagrant2

[redis]
vagrant3
```

本章是一个快速的（但是我希望有帮助的）关于如何最有效地结合使用 Vagrant 和 Ansible 的概述。Vagrant 对于 Ansible 作为置备器还支持许多本章没有涉及的选项。如果想要了解更多细节，可以查阅 Vagrant 的官方文档中关于 Ansible 置备器的章节 (<https://docs.vagrantup.com/v2/provisioning/ansible.html>)。

<202

Amazon EC2

Ansible 有很多功能可以让 IaaS (infrastructure-as-a-service) 云服务的使用更简单。本章节会聚焦于 Amazon EC2，因为它是最流行的 IaaS 云服务，同时也是我最熟悉的一个。当然，很多内容对其他 Ansible 所支持的云服务也同样适用。

Ansible 支持 EC2 的两种方式：

- 动态 inventory 插件，可以自动地更新你的 Ansible inventory 文件而不需要手动指定服务器。
- 支持在 EC2 上执行类似创建新的服务器等操作的模块。

在本节中，我们会讨论 EC2 动态 inventroy 插件和 EC2 模块。

什么是 IaaS 云

你可能已经在科技新闻中不厌其烦地“云计算”轰炸下感到麻木了。^{注 1} 我将讲述一下我所理解的 IaaS 云服务。

我们将以一个典型的用户与 IaaS 云服务的交互开始。

用户

我想要 5 台新的服务器，每台需要配备 2 个 CPU、4GB 内存以及 100GB 存储，并且运行 Ubuntu 14.04 操作系统。

注 1：国家标准与技术研究所（NIST-National Institute of Standards and Technology）对于云计算有一个非常棒的定义：*The NIST Definition of Cloud Computing*。

服务

请求已收到。您的请求编号是 432789。

用户

请求 432789 的当前状态是什么？

服务

你的服务器已经可以使用，IP 地址分别为：203.0.113.5、203.0.113.13、203.0.113.49、203.0.113.124、203.0.113.209。

用户

请求 432789 所关联的服务器已经不需要了。

服务

请求已收到，这些服务器已经被关闭。

IaaS 云服务是一个让用户可以随意配置（创建）新的服务器的服务。所有的 IaaS 云服务都是自助式的，这意味着用户将直接与软件服务交互，而不再需要去 IT 部门填表格。大部分的 IaaS 云服务都提供 3 种不同类型的接口用于帮助用户与系统交互。

- Web 接口。
- 命令行接口。
- REST API。

在 EC2 场景下，Web 接口被称为 AWS 管理控制台（AWS Management Console：<https://console.aws.amazon.com>），命令行接口被（毫无想象力地）称为 AWS 命令行界面（AWS Command-Line Interface：<https://aws.amazon.com/cli/>）。REST API 的文档记录在 Amazon 的官网上（[IaaS 云服务通常使用虚拟机来实现提供给用户的服务器，尽管你也可以使用裸机服务器（就是指用户直接在硬件上运行服务而不是在虚拟机里）或容器来构建 IaaS 云服务。SoftLayer 和 Rackspace 提供裸机服务，而 Amazon Elastic Compute Cloud、Google Compute Engine 和 Joyent clouds 则提供容器服务。](http://docs.aws.amazon.com/AWSEC2/latest/APIReference>Welcome.html）。</p>
</div>
<div data-bbox=)

绝大部分的 IaaS 云服务不仅提供了让你启动和销毁服务器的功能。比如说，它们通常为你提供配置存储服务，你可以利用这个服务在你的服务器上随意地附加或者删除这些磁盘。这种类型的存储通常被称为块存储。它们还提供了网络功能，

让你可以定义网络拓扑来描绘你的服务器是如何相互连接的，并且你还能定义防火墙规则来限制服务器的网络权限。

Amazon EC2 是最流行的 IaaS 公有云提供商，不过除了 EC2 还有很多其他的 IaaS 云服务。除了 EC2，Ansible 随附的模块也支持 Microsoft Azure、Digital Ocean、Google Compute Engine、Linode 以及 Rackspace，甚至是使用 OpenStack 和 VMWare vSphere 构建的云服务。

术语^{注 2}

205

EC2 公开了许多不同的概念。这些概念在本章出现的时候我会逐一解读。不过有 3 个术语我需要提前解释一下。

实例

EC2 的文档中使用实例 (*instance*) 这个术语来指代一台虚拟机，而且我也会在本章中使用这个术语。要牢牢记住，一个 EC2 实例在 Ansible 看来就是一台主机。

EC2 文档 (<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-lifecycle.html>) 中交替使用 *creating instance*、*launching instance* 和 *running instance* 这 3 个术语来描述创建一个新实例的过程。而 *starting instance* 却代表不同的意思——启动一个之前处于停止状态的实例。

Amazon 系统镜像

一个 Amazon 系统镜像 (AMI, Amazon Machine Image) 就是一个虚拟机镜像，它包含了一个已经安装好操作系统的文件系统。当你在 EC2 上创建一个实例的时候，你要通过指定 AMI 来选择你想要运行实例的操作系统，EC2 将会使用 AMI 来创建实例。

每个 AMI 都有一个关联的标识字符串，称为 *AMI ID*，它以“ami-”开头并且包含 8 个十六进制字符。例如，`ami-12345abc`。

标签

EC2 允许用户使用自定义元数据注释实例^{注 3}，这些自定义元数据称作标签 (*tag*)。标签就是一个字符串组成的 key-value 对。比如说，我们可以使用下面的标签来注释某个实例：

注 2： 本书中所有 AWS 相关术语的译文都尽量遵循 AWS 中文版的翻译。——译者注

注 3： 除了实例，你也可以向其他对象添加标签。例如 AMI、卷以及安全组。

```
Name=Staging database  
env=staging  
type=database
```

如果你已经在 AWS 管理控制台中给你的 EC2 实例进行了命名，那么你就在不知不觉中使用了标签功能。EC2 是使用标签功能来实现实例名的，这类标签的 key 是 Name，而对应的 Value 就是你为实例设置的名称。除此之外，Name 标签没有其他特别之处了。当然除了 Name 标签，你也可以配置管理控制台来显示其他标签的值。

206 标签不需要是唯一的，因此你可以给 100 个实例添加同样的标签。由于 Ansible 的 EC2 模块大量使用了标签，因此在本章中标签将被多次提到。

指定认证凭据

当你向 Amazon EC2 发送请求的时候，你需要指定认证凭据。如果你使用 AWS 管理控制台，那么使用你的用户名和密码来登录即可。然而，Ansible 所有与 EC2 的交互都是通过 EC2 API 的。API 不能使用用户名和密码作为认证凭据。相对地，API 使用两个字符串：*access key ID* 和 *secret access key*。

一般情况下，这些字符串像如下这样。

- 示例 EC2 access key ID : AKIAIOSFODNN7EXAMPLE
- 示例 EC2 secret access key : wJalrXUtnFEMI/K7MDENG/bPxRfCYEXAMPLEKEY

当你调用 EC2 相关的模块时，你需要将这些字符串作为模块参数传入。对于动态 inventory 插件，你可以在 *ec2.ini* 文件中（稍后将会详细讨论）指定认证凭据。但是，无论是 EC2 模块还是动态 inventory 插件都允许使用环境变量来指定认证凭据。如果你的控制机本身就是一个 Amazon EC2 实例的话，你可以使用访问控制与密码管理（IAM, *Identity and Access Management*）role，详见附录 C。

环境变量

尽管 Ansible 允许你明确地通过参数的方式将认证凭据传给模块，但是它也支持使用环境变量来设置 EC2 凭据。例 12-1 展示了如何设置这些环境变量。

例 12-1 设置 EC2 环境变量

```
# 不要忘记将具体值替换为你实际的认证信息  
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE  
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfCYEXAMPLEKEY  
export AWS_REGION=us-east-1
```



并不是所有的 Ansible EC2 模块都遵循 AWS_REGION 环境变量，所以我建议当调用 EC2 相关模块时始终明确地将 EC2 区域作为参数传入。本章的所有范例中，都明确地将区域作为参数传入。

<207

我建议使用环境变量是因为它允许你在使用 EC2 相关模块和 inventory 插件的时候不需要将认证凭据放到任何 Ansible 相关文件中。我把这些环境变量的设置放到了一个会话启动时自动运行的隐藏文件中。我使用的是 Zsh，所以对于我的情况这个文件是 `~/.zshrc`。如果你使用的是 Bash，你可能想要将环境变量的设置放到你的 `~/.profile` 文件^{注4}。如果你使用了 Bash 或 Zsh 以外的其他 Shell，那么我觉得你可能清楚自己应该修改哪个隐藏文件来设置这些环境变量了。

一旦你已经将凭据设置到环境变量中，你就可以在你的控制主机上调用 Ansible EC2 模块，同样也可以使用动态 inventory 插件。

配置文件

另一种使用环境变量的方式是将你的 EC2 凭据放在一个配置文件中。我们在之后马上就会讨论到，Ansible 使用了 Python 的 Boto 类库，因此它支持使用 Boto 的惯例将凭据维护在一个 Boto 配置文件中。在这我不再对这种文件格式做更多叙述了，你可以通过查阅 Boto 的配置文档了解更多详细信息 (http://boto.readthedocs.org/en/latest/boto_config_tut.html#boto-config)。

必要条件：Boto Python 库

Ansible 所有关于 EC2 的功能都需要你在控制主机上安装 Python Boto 类库（一个 Python 系统软件）。你需要如下操作来安装它：^{注5}

```
$ pip install boto
```

如果你已经有实例在 EC2 上运行了，那么你可以通过使用如例 12-2 所示的 Python 命令行交互来验证 Boto 是否安装正确以及你的凭据的正确性。

注 4： 或者它也许是 `~/.bashrc`？我并不了解 Bash 不同的隐藏文件之间的区别。`(~/.profile` 在非登录 Shell 中并不会被执行，而 `~/.bashrc` 不管是否登录 Shell 都会被执行。一般准则是：环境变量和登录时执行的程序放在 `profile` 系列文件中；而函数与别名定义放到 `bashrc` 系列文件中。当然你可以在足够了解 Bash 行为的前提下自己决定添加到哪个文件中。——译者注)

注 5： 你可能需要使用 `sudo` 或者激活 `virtualenv`，取决于你是如何安装 Ansible 的。

例 12-2 测试Boto和凭据

```
$ python
Python 2.7.6 (default, Sep 9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto.ec2
>>> conn = boto.ec2.connect_to_region("us-east-1")
>>> statuses = conn.get_all_instance_status()
>>> statuses
[]
```

208

动态 inventory

如果你的服务器已经在 EC2 上运行，而你不想在 Ansible inventory 文件中保存一份服务器信息的单独副本，因为这个文件会随着你创建或销毁服务器而随时发生变化。

借助于 Ansible 所支持的动态 inventory 功能可以直接从 EC2 上获取服务器信息，这样跟踪你的 EC2 服务器数据就变得十分简单了。Ansible 随附了一个支持 EC2 的动态 inventory 脚本，不过我建议你从 Ansible 的 GitHub 仓库^{注6} 获取一个最新的版本。

你需要如下两个文件。

ec2.py

实际的 inventory 脚本 (<https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/ec2.py>)

ec2.ini

inventory 脚本所需要的配置文件 (<https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/ec2.ini>)

在此之前，我们使用一个 *playbooks/host* 文件来担任我们的 inventory。现在我们要使用一个 *playbooks/inventory* 目录，将 *ec2.py* 和 *ec2.ini* 放到这个目录中，同时设置 *ec2.py* 为可执行文件。例 12-3 展示了一种实现方式。

例 12-3 安装EC2动态inventory脚本

```
$ cd playbooks/inventory
$ wget https://raw.githubusercontent.com/ansible/ansible/devel/plugins/inventory/ec2.py
$ wget https://raw.githubusercontent.com/ansible/ansible/devel/plugins/inventory/ec2.ini
$ chmod +x ec2.py
```

注 6： 呃，老实说，我根本不知道包管理工具把这个文件装到哪里去了。



如果在一个使用 Python 3.x 作为默认 Python 版本的 Linux 发行版（比如 Arch Linux）上运行 Ansible，那么 *ec2.py* 将无法在未经修改的情况下正常工作，因为它是一个 Python 2.x 脚本。

在这种情况下需要确认你的系统已经安装了 Python 2.x 版本，然后将 *ec2.py* 文件的第 1 行从：

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

修改为：

```
#!/usr/bin/env python2
```

如果你已经像之前章节中所描述的那样设置好了你的环境变量，那么你就可以直接通过 209 运行如下脚本来确认脚本是否可以正常工作。

```
$ ./ec2.py --list
```

脚本应该会输出你的每个 EC2 范例的信息，数据结构看起来像这样：

```
{
    "_meta": {
        "hostvars": {
            "ec2-203-0-113-75.compute-1.amazonaws.com": {
                "ec2_id": "i-i2345678",
                "ec2_instance_type": "c3.large",
                ...
            }
        }
    },
    "ec2": [
        "ec2-203-0-113-75.compute-1.amazonaws.com",
        ...
    ],
    "us-east-1": [
        "ec2-203-0-113-75.compute-1.amazonaws.com",
        ...
    ],
    "us-east-1a": [
        "ec2-203-0-113-75.compute-1.amazonaws.com",
        ...
    ],
    "i-12345678": [
        "ec2-203-0-113-75.compute-1.amazonaws.com",
    ],
    "key_mysshkeyname": [

```

```
"ec2-203-0-113-75.compute-1.amazonaws.com",
...
],
"security_group_ssh": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
],
"tag_Name_my_cool_server": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
],
"type_c3_large": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
]
}
```

210 ◀ inventory 缓存

当 Ansible 执行 EC2 动态 inventory 脚本时，这个脚本会向一个或多个 EC2 端点请求信息。由于这个过程需要花费一定的时间，因此这个脚本会在第一次被调用的时候将返回信息缓存在下面的文件中：

- `$HOME/.ansible/tmp/ansible-ec2.cache`
- `$HOME/.ansible/tmp/ansible-ec2.index`

在之后的调用中，这个动态 inventory 脚本将会使用这些缓存信息直到缓存过期。

你可以通过编辑 `ec2.ini` 配置文件中的 `cache_max_age` 配置项来修改这一行为。它的默认值是 300s（5 分钟）。如果你根本不希望使用缓存，可以将其设置为 0：

```
[ec2]
...
cache_max_age = 0
```

你也可以在调用 inventory 脚本的时候指定 `--refresh-cache` 参数来强制 inventory 脚本刷新缓存：

```
$ ./ec2.py --refresh-cache
```



如果你创建或销毁了实例，则 EC2 动态 inventory 脚本并不会直接反映这些变化，除非缓存过期或者你手动刷新缓存。

其他配置项

ec2.ini 文件还包含许多其他的配置项来控制动态 inventory 脚本的行为。由于文件本身已经利用注释良好地文档化了，在这里我就不再赘述了。

自动生成群组

EC2 动态 inventory 脚本会创建如表 12-1 所示的群组。

表 12-1 自动生成的EC2群组

< 211

类型	范例	Ansible 群组名
Instance	i-123456	i-123456
Instance type	c1.medium	type_c1_medium
Security group	ssh	security_group_ssh
Keypair	foo	key_foo
Region	us-east-1	us-east-1
Tag	env=staging	tag_env_staging
Availability zone	us-east-1b	us-east-1b
VPC	vpc-14dd1b70	vpc_id_vpc-14dd1b70
All ec2 instances	N/A	ec2

在群组名中只有字母、连字符和下画线是合法的。动态 inventory 脚本会自动将其他的字符转换成下画线。

举个例子，如果你的某个范例有这样一个标签：

```
Name=My cool server
```

Ansible 将会生成 tag_Name_my_cool_server 这样的组名。

使用标签定义动态群组

之前提到动态 inventory 脚本会自动基于某些数据创建群组，比如实例类型、安全组、密钥对和标签。EC2 的标签是用来创建 Ansible 群组的最简洁的方式，因为你可以按照自己喜欢的方式来定义它们。

比如说，你可以将所有的 Web 服务器都加上下面的标签：

```
type=web
```

Ansible 将会自动地创建一个名为 `tag_type_web` 的群组，它包含所有的具有 `name` 为 `type`, `value` 为 `web` 这样标签的服务器。

EC2 允许你为每个实例添加多个标签。比如说，如果你有单独的仿真环境和生产环境，那么你可以将生产环境的 Web 服务器加上下面的标签：

```
212 > env=production  
      type=web
```

现在你可以使用 `tag_env_production` 来代表生产环境的服务器，用 `tag_type_web` 代表 Web 服务器。如果你想要代表生产环境的 Web 服务器，你可以像下面这样使用 Ansible 的交集语法：

```
hosts: tag_env_production:&tag_type_web
```

把标签应用到现有资源

在理想情况下，你会在创建 EC2 实例的同时添加标签。然而，如果你使用 Ansible 来管理现有的 EC2 实例，你可能需要给一些正在运行的实例添加标签。Ansible 提供了一个 `ec2_tag` 模块，允许你给自己的实例添加标签。

举个例子，如果你想要给一个实例添加 `env=prodution` 和 `type=web` 这样的标签，你可以像例 12-4 那样写一个简单的 playbook。

例 12-4 为实例添加 EC2 标签

```
- name: Add tags to existing instances  
  hosts: localhost  
  vars:  
    web_production:  
      - i-123456  
      - i-234567  
    web_staging:  
      - i-ABCDEF  
      - i-333333  
  tasks:  
    - name: Tag production webservers  
      ec2_tag: resource={{ item }} region=us-west-1  
      args:  
        tags: { type: web, env: production }  
      with_items: web_production
```

```
- name: Tag staging webservers
  ec2_tag: resource={{ item }} region=us-west-1
  args:
    tags: { type: web, env: staging }
  with_items: web_staging
```

为了使 playbook 看起来更整洁，这个范例在指定标签的时候使用了 YAML 字典内联语法 (`{ type: web, env: production}`)。不过常规的 YAML 字典语法也可以正常工作：

```
tags:
  type: web
  env: production
```

更好听的群组名

◀213

就个人而言，我并不喜欢 `tag_type_web` 这个群组名。我更愿意称它们为 `web`。

如果想使用这样的组名，我们需要在 `playbooks/inventory` 目录中增加一个包含群组信息的新文件。这仅仅是一个传统的 Ansible inventory 文件，这里我们将它命名为 `playbooks/inventory/hosts`（见例 12-5）。

例 12-5 `playbooks/inventory/hosts`

```
[web:children]
tag_type_web
[tag_type_web]
```

一旦你完成上面的步骤，你就可以在你的 Ansible play 中使用 `web` 作为群组名了。



你必须在静态的 inventory 文件中定义一个空的 `tag_type_web` 群组，尽管动态 inventory 脚本也定义了这个群组。如果你没有这么做，Ansible 会报出下面的错误：

```
ERROR: child group is not defined: (tag_type_web)
```

EC2 Virtual Private Cloud (VPC) 和 EC2 Classic

早在 2006 年，Amazon 首次推出 EC2 服务的时候，所有的 EC2 实例都使用 flat 网络^{注7}直接通信。每一个 EC2 实例都有一个私有 IP 地址和一个公有 IP 地址。

注 7： AWS 的内部网络分割为多个子网，但是用户无法控制实例被分配到哪个子网下。

在 2009 年，Amazon 推出一个叫作 *Virtual Private Cloud* (VPC) 的新功能。VPC 允许用户控制他们实例之间的网络连接方式，以及需要与公网互通还是被隔离。Amazon 使用“VPC”这个术语来描述用户可以在 EC2 内部创建的虚拟网络。Amazon 使用“EC2-VPC”来表示部署在 VPC 中的实例，使用“EC2-Classic”来表示没有部署在 VPC 中的实例。

214

Amazon 积极地鼓励使用他们的 EC2-VPC 服务，比如，像 t2.micro 这种类型的实例仅在 EC2-VPC 中可用。根据你的 AWS 账号创建时间和你之前部署实例所在的 EC2 区域，你可能根本没有权限使用 EC2-Classic。^{注 8} 表 12-2 描述了哪种账号拥有使用 EC2-Classic 的权限。

表 12-2 我可以使用 EC2-Classic 吗

账户创建时间	是否可以使用 EC2-Classic
2013 年 3 月 18 日之前	之前可以，但是仅限在你之前使用过的区域
2013 年 3 月 18 日至 2014 年 12 月 4 日	有可能，不过也仅限在你之前使用过的区域
2013 年 12 月 4 日之后	不可以

支持使用 EC2-Classic 和仅可以使用 EC2-VPC 的主要区别是：当你创建新的 EC2 实例且没有明确将实例与一个 VPC ID 相关联的时候会发生什么。如果你的账号开启 EC2-Classic，那么新的实例就不会与任何 VPC 相关联。如果你的账号没有开启 EC2-Classic，那么新的实例就会与默认 VPC 相关联。

你需要关注这个区别的原因是：在 EC2-Classic 中，所有的实例都允许对外连接到互联网任何地址。而在 EC2-VPC 中，默认情况下实例无法对外连接到互联网。如果某个 VPC 实例需要对外连接，它必须与一个允许对外连接的安全组（security group）相关联。

出于本章的目的考虑，我只会使用 EC2-VPC 服务，因此我会将实例关联到一个已开启对外连接的安全组中。

配置 ansible.cfg 支持使用 EC2

当我使用 Ansible 来配置 EC2 实例时，我在 *ansible.cfg* 里面添加了下面几行：

```
[defaults]
remote_user = ubuntu
host_key_checking = False
```

注 8： 可以到 AWS 官网查询关于 VPC (http://docs.aws.amazon.com/zh_cn/AmazonVPC/latest/UserGuide/default-vpc.html#default-vpc-basics) 和是否你可以使用某个区域的 EC2-Classic (http://docs.aws.amazon.com/zh_cn/AmazonVPC/latest/UserGuide/default-vpc.html#detecting-platform) 的详细信息。

我一直在使用 Ubuntu 镜像，并且在这些镜像中你应该使用 *ubuntu* 用户来进行 SSH 操作。我还关闭了 host key 检查，因为我无法事先知道新实例上的 host key 是什么。^{注 9}

启动新的实例

`ec2` 模块允许用户在 EC2 上启动新的实例。这是 Ansible 最复杂的模块之一，因为它支持非常多的参数。

例 12-6 展示了一个用于启动 Ubuntu 14.04 EC2 实例的简单 playbook。

例 12-6 创建一个EC2实例的简单playbook

```
- name: Create an ubuntu instance on Amazon EC2
hosts: localhost
tasks:
- name: start the instance
  ec2:
    image: ami-8caalce4
    region: us-east-1
    instance_type: m3.medium
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { Name: ansiblebook, type: web, env: production }
```

让我们来过一遍这些参数的含义。

`image` 参数表示 Amazon 系统镜像 (AMI) 的 ID，必须在每次调用时指定。正如本章之前所述，镜像基本上是一个包含着已经安装好操作系统的文件系统。上面的范例中使用 *ami-8caalce4* 来代表一个安装了 64 位 Ubuntu 14.04 版本操作系统的镜像。

`region` 参数指定了所启动实例的地理区域。^{注 10}

`instance_type` 参数描述了实例的 CPU 核数、内存容量和存储容量。EC2 不允许用户选择任意的 CPU 核数、内存和存储的组合方式。相对地，Amazon 定义了一系列的实例类型。^{注 11} 在先前的例子中使用的 *t2.medium* 实例类型，是一个具有 1 核 CPU、3.75GB RAM 内存以及 4GB 基于 SSD 的存储的 64 位实例类型。

注 9： 可以通过向 EC2 查询实例的控制终端输出来获取 host key。我必须承认我从没这么做过，因为我一直没抽出时间来编写从控制终端输出中解析出 host key 的脚本。

注 10： 可以访问 AWS 官网 (http://docs.aws.amazon.com/zh_cn/general/latest/gr/rande.html#ec2_region) 来查询它支持区域的列表。

注 11： 有一个便利的（非官方）网站 (<http://www.ec2instances.info/>) 提供了一个列有所有可用的 EC2 实例类型的表格。



并不是所有的镜像都兼容全部的实例类型。我没有实际测试过是否 ami-8caa1ce4 镜像可以兼容 m3.medium 实例类型。请读者自行甄别。

216 key_name 参数表示一个 SSH 密钥对。Amazon 提供给用户登录到服务器的方式是使用 SSH 密钥对。在你启动第一台服务器之前，你必须创建一个新的 SSH 密钥对或者上传你之前创建的密钥对中的公钥。无论你是创建一个新的密钥对，还是上传现有的，你都必须对你的 SSH 密钥对进行命名。

group 参数表示与实例所关联的安全组列表。这些安全组决定了哪些进出的网络连接是被允许的。

instance_tags 参数用 EC2 的键值对标签的方式将元数据和实例关联在一起。在之前的范例中，我们设置了如下标签：

```
Name=ansiblebook  
type=web  
env=production
```

EC2 密钥对

在例 12-6 中，我们假定 Amazon 已经配置好了名为 mykey 的 SSH 密钥对。接下来看一下我们如何使用 Ansible 来创建新的密钥对。

创建新的密钥

当你创建一个新的密钥对时，Amazon 生成一个私钥和一个对应的公钥，然后将私钥发给你。Amazon 不会保存私钥的副本，因此你需要确保在生成之后保存好它。例 12-7 展示了如何使用 Ansible 来创建一个新的密钥。

例 12-7 创建一个新的SSH密钥对

```
- name: create a new keypair  
hosts: localhost  
tasks:  
  - name: create mykey  
    ec2_key: name=mykey region=us-west-1  
    register: keypair  
  
  - name: write the key to a file  
    copy:  
      dest: files/mykey.pem
```

```
content: "{{ keypair.key.private_key }}"
mode: 0600
when: keypair.changed
```

在例 12-7 中，我们调用 `ec2_key` 模块来创建一个新的密钥对。然后我们使用 `copy` 模块 <217> 的 `content` 参数来将 SSH 的私钥存在一个文件里。

如果模块创建了一个新的密钥对，那么注册的 `keypair` 变量中包含的值就看起来像这样：

```
"keypair": {
  "changed": true,
  "invocation": {
    "module_args": "name=mykey",
    "module_name": "ec2_key"
  },
  "key": {
    "fingerprint": "c5:33:74:84:63:2b:01:29:6f:14:a6:1c:7b:27:65:69:61:f0:e8:b9",
    "name": "mykey",
    "private_key": "-----BEGIN RSA PRIVATE KEY-----\nMIIEowIBAAKCAQEAjAJpvhY3QGKh
...
OPkCRPl8ZHKtShKESIsG3WC\n-----END RSA PRIVATE KEY-----"
  }
}
```

如果密钥对已经存在了，那么注册的 `keypair` 变量中包含的值就是这样的：

```
"keypair": {
  "changed": false,
  "invocation": {
    "module_args": "name=mykey",
    "module_name": "ec2_key"
  },
  "key": {
    "fingerprint": "c5:33:74:84:63:2b:01:29:6f:14:a6:1c:7b:27:65:69:61:f0:e8:b9",
    "name": "mykey"
  }
}
```

由于 `private_key` 变量的值在密钥已经存在的时候并不存在，所以我们需要添加一个 `when` 语句来确保只有在 private key 文件存在的时候才调用 `copy` 将其写到本地磁盘。

我们添加如下一行：

```
when: keypair.changed
```

当 `ec2_key` 模块调用后，只有状态改变的时候才将文件写到本地磁盘中（例如，创建了一个新的密钥）。我们还可以使用另外一个方法：检查 `private_key` 的值是否存在，如下

面所示：

```
218 - name: write the key to a file
      copy:
        dest: files/mykey.pem
        content: "{{ keypair.key.private_key }}"
        mode: 0600
      when: keypair.key.private_key is defined
```

我们使用了 Jinja2 的内置检验语法 `defined`^{注12} 来检查 `private_key` 是否存在。

上传已有密钥

如果你已经有一个 SSH 公钥了，你可以把它上传到 Amazon 并且与一个密钥对进行关联：

```
- name: create a keypair based on my ssh key
  hosts: localhost
  tasks:
    - name: upload public key
      ec2_key: name=mykey key_material="{{ item }}"
      with_file: ~/.ssh/id_rsa.pub
```

安全组

例 12-6 中假设 `web`、`SSH` 和 `outbound` 这 3 个安全组已经存在。我们可以在使用它们之前用 `ec2_group` 模块来确认这些安全组是否已经被创建。

安全组与防火墙规则类似：你能够指定关于谁能够连接这台服务器以及如何连接的规则。

在例 12-8 中，我们指定 `web` 安全组的规则为：在互联网上的任何人都可以通过 80 和 443 端口连接。对于 `SSH` 安全组，我们允许在互联网上的任何人连接 22 端口。对于 `outbound` 安全组，我们允许对外连接到互联网的任何地址。我们需要开启对外连接以便从互联网上下载软件包。

例 12-8 安全组

```
- name: web security group
  ec2_group:
    name: web
    description: allow http and https access
    rules:
      - proto: tcp
```

注 12：想要获取更多关于 Jinja2 检验语法的信息，可以查阅 Jinja2 文档中关于内置检验语法的部分 (<http://jinja.pocoo.org/docs/dev/templates/#defined>)。

```

from_port: 80
to_port: 80
cidr_ip: 0.0.0.0/0
- proto: tcp
  from_port: 443
  to_port: 443
  cidr_ip: 0.0.0.0/0

- name: ssh security group
ec2_group:
  name: ssh
  description: allow ssh access
  rules:
    - proto: tcp
      from_port: 22
      to_port: 22
      cidr_ip: 0.0.0.0/0

- name: outbound group
ec2_group:
  name: outbound
  description: allow outbound connections to the internet
  region: "{{ region }}"
  rules_egress:
    - proto: all
      cidr_ip: 0.0.0.0/0

```



如果你正在使用 EC2-Classic，那么你不需要指定 `outbound` 安全组，因为 EC2-Classic 不会限制实例对外连接。

如果你之前没有使用过安全组，那么需要对规则字典的参数进行一些说明。表 12-3 提供了一个关于安全组连接规则参数的速查摘要。

表 12-3 安全组规则参数

参数	描述
proto	IP 协议 (tcp、udp、icmp) 或者 “all”，all 允许所有的协议和端口
cidr_ip	允许连接的子网 IP 地址，使用 CIDR 表示法
from_port	允许的端口范围内的起始端口
to_port	允许的端口范围内的截止端口

允许的 IP 地址

安全组允许你限制哪个 IP 地址可以访问实例。你可以使用无类别域间路由（CIDR）表示法指定子网。`203.0.113.0/24`^{注13} 是一个使用 CIDR 表示法表示子网的范例。它表示 IP 地址的前 24 位必须与 `203.0.113.0` 的前 24 位完全匹配。人们有时候会仅以 “/24” 来表示以 /24 结尾的 CIDR。

/24 是一个很讨喜的值，因为它对应地址的前 3 个字节，即 `203.0.113`^{注14}。这意味着以 `203.0.113` 开头的 IP 地址都在这个子网中，也就是在 `203.0.113.0 ~ 203.0.113.255` 范围内任意的 IP 地址。

如果你指定 `0.0.0.0/0`，那意味着所有 IP 地址都允许连接。

安全组端口

有件事让我对 EC2 的安全组有些疑惑，那就是 `from port` 和 `to port` 表示法。EC2 允许你指定一个允许访问的端口范围。比如说，你想要表明你允许 `5900 ~ 5999` 区间的任何端口号进行 TCP 连接访问，你需要这样指定：

```
- port: tcp
  from_port: 5900
  to_port: 5999
  cidr_ip: 0.0.0.0/0
```

然而，我时常发现 `from/to` 表示法让人很困惑，因为我几乎从来没有指定过一个端口范围。

^{注15} 相对地，我通常希望启用非连续的端口，比如 80 和 443。因此，几乎在每一个案例中，`from_port` 和 `to_port` 参数都是相同的。

`ec2_group` 模块中有大量的其他参数，包括使用安全组 ID 指定入站规则和出站规则。你可以查阅模块的文档了解更多详细内容。

获取最新的 AMI

在例 12-6 中，我们用下面的方式明确地指定了 AMI。

```
image: ami-8caaice4
```

`221` 然而，如果你想使用最新的 Ubuntu 14.04 版本镜像，你不会想使用像上面的这种硬编码

^{注13}：这个范例正好对应了一个名为 TEST-NET-3 的特殊地址范围，它是为范例而预留的。它是 example.com 的 IP 子网。

^{注14}：/8、/16 以及 /24 子网非常适合做范例，因为相对于其他数字，比如说 /17 或者 /23，更容易计算。

^{注15}：聪明的读者可能已经注意到了，端口 `5900 ~ 5999` 通常被 VNC 远程桌面协议所使用。它是为数不多的让指定端口范围有意义的应用之一。

AMI 的方式。那是因为 Canonical^{注16} 频繁地对 Ubuntu 镜像进行更新，并且更新一次镜像，就会生成一个新的 AMI。只是因为 `ami-8caaice4` 在昨天还对应 Ubuntu 14.04 的最新 release 版本，但并不意味着明天它还是。

Ansible 随附了一个漂亮的小模块 `ubuntu_ami_search`（作者正是鄙人），可以用来获取一个给定操作系统版本所对应的 AMI。例 12-9 展示了这个运行中的模块。

例 12-9 获取最新的Ubuntu AMI

```
- name: Create an ubuntu instance on Amazon EC2
hosts: localhost
tasks:
  - name: Get the ubuntu trusty AMI
    ec2_ami_search: distro=ubuntu release=trusty region=us-west-1
    register: ubuntu_image

  - name: start the instance
    ec2:
      image: "{{ ubuntu_image.ami }}"
      instance_type: m3.medium
      key_name: mykey
      group: [web, ssh, outbound]
      instance_tags: { type: web, env: production }
```

目前，该模块只支持搜索 Ubuntu 的 AMI。

向群组中添加一个新的实例

有时候我喜欢写一个单独的 playbook 来启动一个实例，然后对这个实例继续运行 playbook 后面的内容。

不幸的是，在你运行 playbook 之前，主机还不存在。禁用动态 inventory 脚本的缓存功能在这并没有帮助，因为 Ansible 只会在 playbook 开始执行的时候才调用动态 inventory 脚本，这些都是在主机存在之前发生的。

你可以使用 `add_host` 模块创建一个向群组中添加实例的任务，如例 12-10 所示。

例 12-10 向群组中添加一个实例

```
- name: Create an ubuntu instance on Amazon EC2
hosts: localhost
tasks:
  - name: start the instance
    ec2:
```

注 16： Canonical 是运营 Ubuntu 项目的公司。

```

image: ami-8caalce4
instance_type: m3.medium
key_name: mykey
group: [web, ssh, outbound]
instance_tags: { type: web, env: production }
register: ec2

- name: add the instance to web and production groups
  add_host: hostname={{ item.public_dns_name }} groups=web,production
  with_items: ec2.instances

- name: do something to production webservers
  hosts: web:&production
  tasks:
    - ...

```

ec2 模块的返回类型

ec2 模块返回一个包含 3 个字段的字典，如表 12-4 所示。

表 12-4 ec2 模块的返回值

参数	描述
instance_ids	实例 ids 组成的列表
instances	实例字典组成的列表
tagged_instances	实例字典组成的列表

如果用户向 ec2 模块传入了 exact_count 参数，那么模块可能不会创建新的实例，如 231 页“创建实例的幂等性方法”中所描述的那样。在这种情况下，instance_ids 和 instances 字段仅会在模块创建新的实例的时候被填充。不过，tagged_instances 字段会包含所有匹配标签的实例的实例 dicts，无论实例是新创建的还是之前存在的。

实例 dict 包含的字段如表 12-5 所示。

223 >

表 12-5 实例dict的内容

参数	描述
id	实例 ID
ami_launch_index	保留的实例索引（0 ~ N-1 之间，假设启动了 N 个实例）
private_id	内部 IP 地址（EC2 外部无法访问）
private_dns_name	内部 DNS 名称（EC2 外部无法访问）

续表

参数	描述
public_ip	公有 IP 地址
public_dns_name	公有 DNS 名称
state_code	状态变化原因码
architecture	CPU 架构
image_id	AMI
key_name	密钥对名称
placement	实例启动的地理位置
kernel	AKI (Amazon 内核镜像)
ramdisk	ARI (Amazon ramdisk 镜像)
launch_time	实例启动时间
instance_type	实例类型
root_device_type	root 设备的类型 (临时存储, EBS)
root_device_name	root 设备名称
state	实例状态
hypervisor	Hypervisor 类型

想要了解这些字段含义的更多细节, 请查看 Boto^{注 17} 文档中 `boto.ec2.instance.Instance` 类 (<http://boto.readthedocs.org/en/latest/ref/ec2.html#boto.ec2.instance.Instance>) 或者查看描述 Amazon 命令行工具 `run-instances` 命令输出结果的相关文档 (<http://docs.aws.amazon.com/cli/latest/reference/ec2/run-instances.html#output>)。^{注 18}

◀224

注 17: Boto 是 Ansible 用来与 EC2 通信的 Python 库。

注 18: 命令行工具的文档在 <http://aws.amazon.com/cli/>。

等待服务器启动

即使像 EC2 这样技术十分卓越的 IaaS 云平台，创建新的实例仍然需要耗费一定的时间。也就是说，你并不能在提交了创建 EC2 实例的请求后就立刻对它执行 playbook。相对地，你需要等 EC2 实例启动好才可以。

ec2 模块支持一个 `wait` 参数。如果设置成“yes”，那么 ec2 任务会一直不返回直到实例变为运行状态：

```
- name: start the instance
  ec2:
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { type: web, env: production }
    wait: yes
    register: ec2
```

不幸的是，等待实例进入运行状态并不能足够确保可以对主机实际执行 playbook。你仍然需要等待直到实例已经几乎完成启动过程，SSH 服务器已经启动并可以接受接入连接。

`wait_for` 模块就是专门为这种场景设计的。下面展示了 ec2 模块和 `wait_for` 模块如何配合使用来启动一个实例并且等待实例接受 SSH 连接：

```
- name: start the instance
  ec2:
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { type: web, env: production }
    wait: yes
    register: ec2
  225 →
- name: wait for ssh server to be running
  wait_for: host={{ item.public_dns_name }} port=22 search_regex=OpenSSH
  with_items: ec2.instances
```

`wait_for` 的调用过程使用了 `search_regex` 参数在连接到主机之后查找 OpenSSH 字符串。这里 `regex` 利用了一个客观事实，即当 SSH 客户端首次连接的时候，一个功能完整的 SSH 服务器将返回一个字符串，如例 12-11 所示。

例 12-11 运行在Ubuntu上的SSH服务的初始响应

```
SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.4
```

我们可以仅仅调用 `wait_for` 模块检查用于接入连接的 22 端口是否被监听。然而，有时候一个 SSH 服务器几乎已经启动完成并且正在监听 22 端口，但是还没有具备完整的功能。等待初始响应能够确保只有在 SSH 服务器完全启动后，`wait_for` 模块才返回成功。

创建实例的幂等性方法

在 playbook 中调用 `ec2` 模块通常不是幂等性的。如果你已经多次执行了例 12-6，EC2 将会创建很多实例。

你可以在 `ec2` 模块中使用 `count_tag` 参数和 `exact_count` 参数来编写幂等性的 playbook。

比如说，我们想要编写一个 playbook 来启动 3 个实例。我们希望这个 playbook 是幂等性的，因此如果 3 个实例已经运行的话，我们希望 playbook 什么都不做，如例 12-12 所示。

例 12-12 幂等性实例创建过程

```
- name: start the instance
  ec2:
    image: ami-8caalce4
    instance_type: m3.medium
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { type: web, env: production }
    exact_count: 3
    count_tag: { type: web }
```

`exact_count:3` 参数告诉 Ansible 确认只有 3 个匹配 `count_tag` 所指定标签的实例在运行。在我们的范例中，我仅为 `count_tag` 参数指定了一个标签，不过它可以支持多个标签。

当第一次运行这个 playbook 的时候，Ansible 将检测当前有多少个被标记了 `type=web` 标签的实例在运行。假设没有任何实例运行，Ansible 将会创建 3 个新的实例并使用 `type=web` 和 `env=production` 标签来标记它们。◀226

当再次运行这个 playbook 的时候，Ansible 同样会检测当前有多少个被标记了 `type=web` 标签的实例在运行。它将会看到有 3 个实例在运行并且不会再启动新的实例。

全部加在一起

例 12-13 列出了创建 3 个 EC2 实例并将它们配置为 Web 服务器的 playbook。这个 playbook 是幂等性的，因此你可以放心地多次运行它，并且它只会在实例还没有创建的

时候创建新的实例。

注意一下我们是如何使用 `tagged_instances` 返回 ec2 模块的值的，而不是使用 `instances` 返回值。究其原因我们在本书 228 页的“ec2 模块的返回类型”中描述过。

例12-13 ec2-example.yml：完整的EC2 playbook

```
---
- name: launch webservers
  hosts: localhost
  vars:
    region: us-west-1
    instance_type: t2.micro
    count: 3
  tasks:
    - name: ec2 keypair
      ec2_key: name=mykey key_material="{{ item }}" region={{ region }}
      with_file: ~/.ssh/id_rsa.pub

    - name: web security group
      ec2_group:
        name: web
        description: allow http and https access
        region: "{{ region }}"
        rules:
          - proto: tcp
            from_port: 80
            to_port: 80
            cidr_ip: 0.0.0.0/0
          - proto: tcp
            from_port: 443
            to_port: 443
            cidr_ip: 0.0.0.0/0

    - name: ssh security group
      ec2_group:
        name: ssh
        description: allow ssh access
        region: "{{ region }}"
        rules:
          - proto: tcp
            from_port: 22
            to_port: 22
            cidr_ip: 0.0.0.0/0

    - name: outbound security group
      ec2_group:
```

```
name: outbound
description: allow outbound connections to the internet
region: "{{ region }}"
rules_egress:
  - proto: all
  cidr_ip: 0.0.0.0/0

- name: Get the ubuntu trusty AMI
  ec2_ami_search: distro=ubuntu release=trusty virt=hvm region={{ region }}
  register: ubuntu_image

- name: start the instances
  ec2:
    region: "{{ region }}"
    image: "{{ ubuntu_image.ami }}"
    instance_type: "{{ instance_type }}"
    key_name: mykey
    group: [web, ssh outbound]
    instance_tags: { Name: ansiblebook, type: web, env: production }
    exact_count: "{{ count }}"
    count_tag: { type: web }
    wait: yes
  register: ec2

- name: add the instance to web and production groups
  add_host: hostname={{ item.public_dns_name }} groups=web,production
  with_items: ec2.tagged_instances
  when: item.public_dns_name is defined

- name: wait for ssh server to be running
  wait_for: host={{ item.public_dns_name }} port=22 search_regex=OpenSSH
  with_items: ec2.tagged_instances
  when: item.public_dns_name is defined

- name: configure webservers
  hosts: web:&production
  sudo: True
  roles:
    - web
```

指定 Virtual Private Cloud

228

到目前为止，我们已经在默认的 Virtual Private Cloud (VPC) 上启动了我们的实例。Ansible 也支持创建新的 VPC 并且在其中启动实例。

什么是 VPC

可以把 VPC 想象成一个独立的网络。当你创建一个 VPC 后，你需要指定一个 IP 地址范围。它必须是专用地址范围（ $10.0.0.0/8$ 、 $172.16.0.0/12$ 或者 $192.168.0.0/16$ ）中某一个的子集。

你可以将你的 VPC 切割成多个子网，每个子网的 IP 范围是你整个 VPC IP 范围的子集。在例 12-14 中，VPC 的 IP 范围是 $10.0.0.0/16$ ，并且我们关联了两个子网： $10.0.0.0/24$ 和 $10.0.1.0/24$ 。

当你启动一个实例的时候，你会把它分配给 VPC 的某个子网。你可以配置你自己的子网，以便你的实例可以获得公有或私有 IP 地址。EC2 也允许你在子网之间为路由通信定义路由表，或者创建一个 Internet 网关在你的子网和互联网之间进行路由通信。

网络配置是一个很复杂的话题，已经超出了本书的范畴。你可以查看 Amazon 的 EC2 文档中关于 VPC 的部分 (http://docs.aws.amazon.com/zh_cn/AmazonVPC/latest/UserGuide/VPC_Introduction.html) 以了解更多详情。

例 12-14 展示了如何创建具有两个子网的 VPC。

例 12-14 create-vpc.yml：创建一个 VPC

```
- name: create a vpc
  ec2_vpc:
    region: us-west-1
    internet_gateway: True
    resource_tags: { Name: "Book example", env: production }
    cidr_block: 10.0.0.0/16
    subnets:
      - cidr: 10.0.0.0/24
        resource_tags:
          env: production
          tier: web
      - cidr: 10.0.1.0/24
        resource_tags:
          env: production
          tier: db
    route_tables:
      - subnets:
          - 10.0.0.0/24
```

```

- 10.0.1.0/24
routes:
- dest: 0.0.0.0/0
gw: igw

```

创建 VPC 是幂等性的。Ansible 基于 `resource_tags` 和 `cidr_block` 两个参数确定唯一的 VPC。如果没有 VPC 匹配资源标签和 CIDR 块将会创建一个新的 VPC。^{注 19}

坦白地说，例 12-14 从网络角度来看只是一个简单的例子，因为我们仅仅定义了两个可以连接到互联网的子网。一个更现实的例子是将会有一个可路由到互联网的子网，另一个无法路由到互联网的子网，并且我们还会为两个子网之间的路由通信设置一些规则。

例 12-15 列出了创建一个 VPC 并在其中启动实例的完整例子。

例 12-15 `ec2-vpc-example.yml`: 指定一个VPC的完整EC2 playbook

```

---
- name: launch webservers into a specific vpc
hosts: localhost
vars:
  instance_type: t2.micro
  count: 1
  region: us-west-1
tasks:
- name: create a vpc
  ec2_vpc:
    region: "{{ region }}"
    internet_gateway: True
    resource_tags: { Name: book, env: production }
    cidr_block: 10.0.0.0/16
    subnets:
      - cidr: 10.0.0.0/24
        resource_tags:
          env: production
          tier: web
      - cidr: 10.0.1.0/24
        resource_tags:
          env: production
          tier: db
    route_tables:
      - subnets:
          - 10.0.0.0/24
          - 10.0.1.0/24

```

^{注 19}：截止到本书编写完成的时候，Ansible 中的一个 bug (<https://github.com/ansible/ansible-modules-core/issues/122>) 仍会导致本模块每次被调用的时候都会错误地汇报状态被改变，即使它并没有创建 VPC。

```
routes:
  - dest: 0.0.0.0/0
    gw: igw
register: vpc

- set_fact: vpc_id={{ vpc.vpc_id }}

- name: set ec2 keypair
  ec2_key: name=mykey key_material="{{ item }}"
  with_file: ~/.ssh/id_rsa.pub

- name: web security group
  ec2_group:
    name: vpc-web
    region: "{{ region }}"
    description: allow http and https access
    vpc_id: "{{ vpc_id }}"
    rules:
      - proto: tcp
        from_port: 80
        to_port: 80
        cidr_ip: 0.0.0.0/0
      - proto: tcp
        from_port: 443
        to_port: 443
        cidr_ip: 0.0.0.0/0

- name: ssh security group
  ec2_group:
    name: vpc-ssh
    region: "{{ region }}"
    description: allow ssh access
    vpc_id: "{{ vpc_id }}"
    rules:
      - proto: tcp
        from_port: 22
        to_port: 22
        cidr_ip: 0.0.0.0/0

- name: outbound security group
  ec2_group:
    name: vpc-outbound
    description: allow outbound connections to the internet
    region: "{{ region }}"
    vpc_id: "{{ vpc_id }}"
    rules_egress:
```

```

- proto: all
  cidr_ip: 0.0.0.0/0

- name: Get the ubuntu trusty AMI
  ec2_ami_search: distro=ubuntu release=trusty virt=hvm region={{ region }}
  register: ubuntu_image

- name: start the instances
  ec2:
    image: "{{ ubuntu_image.ami }}"
    region: "{{ region }}"
    instance_type: "{{ instance_type }}"
    assign_public_ip: True
    key_name: mykey
    group: [vpc-web, vpc-ssh, vpc-outbound]
    instance_tags: { Name: book, type: web, env: production }
    exact_count: "{{ count }}"
    count_tag: { type: web }
    vpc_subnet_id: "{{ vpc.subnets[0].id}}"
    wait: yes
  register: ec2

- name: add the instance to web and production groups
  add_host: hostname={{ item.public_dns_name }} groups=web,production
  with_items: ec2.tagged_instances
  when: item.public_dns_name is defined

- name: wait for ssh server to be running
  wait_for: host={{ item.public_dns_name }} port=22 search_regex=OpenSSH
  with_items: ec2.tagged_instances
  when: item.public_dns_name is defined

- name: configure webservers
  hosts: web:&production
  sudo: True
  roles:
    - web

```

◀ 231



不幸的是，截止到本书编写完成的时候，Ansible 的 `ec2` 模块都无法处理在不同的 VPC 中具有相同名字的安全组这种情况。这意味着，我们不能在多个 VPC 中都定义一个 SSH 安全组，因为当我们启动一个实例的时候，模块将会尝试关联到所有的 SSH 安全组。在我的范例中，我为这些安全组定义了不同的名字。我希望在这个模块之后的版本中可以修复这个问题。

动态 inventory 和 VPC

通常情况下，当使用 VPC 时，你会将一些实例放在一个无法路由到互联网的私有的子网

中。当你这样做的时候，就不会为这些实例关联公网 IP 地址了。

在这种情况下，你可能想要在你的 VPC 内部的某个实例上运行 Ansible。Ansible 的动态 inventory 脚本是足够智能的，它为那些没有公网 IP 地址的 VPC 实例返回内部 IP 地址。

232 可以查看附录 C 了解如何在 VPC 内部使用 IAM role 来运行 Ansible，而不需要再将 EC2 凭据复制到实例中。

构建 AMI

可以有两种方式来让你使用 Ansible 创建自定义的 Amazon 系统镜像（AMI）。你可以使用 `ec2_ami` 模块，或者你也可以使用一个支持 Ansible 的第三方工具 Packer。

使用 `ec2_ami` 模块

`ec2_ami` 模块会为一个正在运行的实例创建快照并保存成一个 AMI。例 12-16 列出了一个此模块的实践案例。

例 12-16 使用 `ec2_ami` 模块创建一个 AMI

```
- name: create an AMI
  hosts: localhost
  vars:
    instance_id: i-dac5473b
  tasks:
    - name: create the AMI
      ec2_ami:
        name: web-nginx
        description: Ubuntu 14.04 with nginx installed
        instance_id: "{{ instance_id }}"
        wait: yes
      register: ami

    - name: output AMI details
      debug: var=ami
```

使用 Packer

`ec2_ami` 模块的工作只是还好，但是你还是需要写一些额外的代码来创建或终止实例。

有一个叫作 Packer (<https://www.packer.io>) 的开源工具，它可以为你自动地创建或终止实例。Packer 恰好也是由 Vagrant 的作者 Mitchell Hashimoto 开发的。

Packer 可以创建不同种类的镜像，并且可以与不同的配置管理工具配合使用。本章将会

聚焦于使用 Packer 与 Ansible 相配合来创建 AMI，不过你也可以使用 Packer 来为其他的 IaaS 云创建镜像，比如 Google Compute Engine、DigitalOcean 或者 OpenStack。它甚至可以用于创建 Vagrant 镜像以及 Docker 容器。它也支持其他的配置管理工具，比如 Chef、Puppet 和 Salt。

使用 Packer 的方法是：你需要创建一个 JSON 格式的配置文件，然后使用 packer 命令 ◀233 行工具通过配置文件来创建镜像。

例 12-17 列出了一个使用 Ansible 和 web role 创建一个 AMI 的 Packer 范例配置文件。

例 12-17 web.json

```
{  
  "builders": [  
    {  
      "type": "amazon-ebs",  
      "region": "us-west-1",  
      "source_ami": "ami-50120b15",  
      "instance_type": "t2.micro",  
      "ssh_username": "ubuntu",  
      "ami_name": "web-nginx-{{timestamp}}",  
      "tags": {  
        "Name": "web-nginx"  
      }  
    }  
  ],  
  "provisioners": [  
    {  
      "type": "shell",  
      "inline": [  
        "sleep 30",  
        "sudo apt-get update",  
        "sudo apt-get install -y ansible"  
      ]  
    },  
    {  
      "type": "ansible-local",  
      "playbook_file": "web-ami.yml",  
      "role_paths": [  
        "/Users/lorinhochstein/dev/ansiblebook/ch12/playbooks/roles/web"  
      ]  
    }  
  ]  
}
```

使用 packer build 命令来创建 AMI：

```
$ packer build web.json
```

输出如下所示：

```
==> amazon-ebs: Inspecting the source AMI...
==> amazon-ebs: Creating temporary keypair: packer 546919ba-cb97-4a9e-1c21-389633dc0779
==> amazon-ebs: Creating temporary security group for this instance...
...
==> amazon-ebs: Stopping the source instance...
==> amazon-ebs: Waiting for the instance to stop...
==> amazon-ebs: Creating the AMI: web-nginx-1416174010
    amazon-ebs: AMI: ami-963fa8fe
==> amazon-ebs: Waiting for AMI to become ready...
==> amazon-ebs: Adding tags to AMI (ami-963fa8fe)...
    amazon-ebs: Adding tag: "Name": "web-nginx"
==> amazon-ebs: Terminating the source AWS instance...
==> amazon-ebs: Deleting temporary security group...
==> amazon-ebs: Deleting temporary keypair...
Build 'amazon-ebs' finished.

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:

us-west-1: ami-963fa8fe
```

例 12-17 有两个区段：builders 和 provisioners。builders 区段表示被创建的镜像的类型。在我们的范例中，我们创建了一个以弹性块存储（EBS，Elastic Block Store）为后端的 Amazon 系统镜像，所以我们使用 amazon-ebs 作为 builder。

Packer 需要启动一个新的实例来创建 AMI，所以你需要为 Packer 配置通常创建一个实例时所使用的所有信息：EC2 区域、AMI 和 实例类型。Packer 不需要配置安全组，因为它会自动创建一个临时的安全组，并在完成后自动删除。和 Ansible 一样，Packer 也需要使用 SSH 来创建实例。因此，你需要在 Packer 的配置文件中指定 SSH 用户名。

你还需要告诉 Packer 你的实例的名字以及你想要应用在你的实例上的标签。因为 AMI 名字必须唯一，我们使用 {{timestamp}} 函数来插入一个 UNIX 时间戳。一个 UNIX 时间戳将日期和时间编码成距离 1970 年 1 月 1 日（UTC）的秒数。查看 Packer 文档了解关于 Packer 所支持函数的更多信息（<https://www.packer.io/docs/templates/configuration-templates.html>）。

由于 Packer 需要与 EC2 交互来创建 AMI，它也需要访问你的 EC2 的凭据。和 Ansible 一样，

Packer 可以从环境变量中读取你的 EC2 凭据，所以你不需要将它们明确地指定在配置文件里，尽管如果这样做的话也可以实现目的。

provisioners 区段表示在实例被保存成镜像之前，用来配置它的工具。Packer 支持一个本地的 Ansible 置备器：它会在实例本身运行 Ansible。这意味着 Packer 在运行之前需要将所有必要的 Ansible playbook 和相关文件复制到实例上，也意味着在它执行 Ansible 之前必须在实例上安装好 Ansible。

Packer 支持一个 Shell 置备器，它可以在实例上运行任意命令。例 12-17 使用这个置备器利用 Ubuntu apt 包来安装 Ansible。为了避免在操作系统完全启动之前就尝试去安装软件包引起竞争，在我们的范例中 Shell 置备器被配置为在安装 Ansible 前等候 30s。 ◀235

例 12-18 列出了我们用来配置实例的 web-ami.yml playbook。这是一个将 web role 应用到本地机器的简单 playbook。由于它使用了 web role，配置文件中必须明确地指定包含 web role 的目录位置，以便于 Packer 可以将 web role 的相关文件复制到实例中。

例 12-18 web-ami.yml

```
- name: configure a webserver as an ami
  hosts: localhost
  sudo: True
  roles:
    - web
```

除了选择性地复制 role，我们也可以告诉 Packer 直接复制我们的整个 playbooks 目录。在这种情况下，配置文件将会变成如例 12-19 所示。

例 12-19 web-pb.json 复制整个playbooks 目录

```
{
  "builders": [
    {
      "type": "amazon-ebs",
      "region": "us-west-1",
      "source_ami": "ami-50120b15",
      "instance_type": "t2.micro",
      "ssh_username": "ubuntu",
      "ami_name": "web-nginx-{{timestamp}}",
      "tags": {
        "Name": "web-nginx"
      }
    }
  ],
  "provisioners": [
    {
```

```
  "type": "shell",
  "inline": [
    "sleep 30",
    "sudo apt-get update",
    "sudo apt-get install -y ansible"
  ],
},
{
  "type": "ansible-local",
  "playbook_file": "web-ami.yml",
  "playbook_dir": "/Users/lorinhochstein/dev/ansiblebook/ch12/playbooks"
}
]
```

截止到本书编写完成的时候，Packer 还不支持 SSH agent forwarding。可以查看 Github (<https://github.com/mitchellh/packer/issues/1066>) 了解这个问题的当前状态。



Packer 还有很多我们在这没有提到的功能。请查看它的官方文档 (<https://www.packer.io/docs/>) 了解更多细节。

其他模块

Ansible 支持很多 EC2 的功能，也支持其他的 AWS 服务。例如，你可以使用 Ansible 的 `cloudformation` 模块来启动 CloudFormation 堆栈，使用 `s3` 模块将文件存入 S3，使用 `route53` 模块修改 DNS 记录，使用 `ec2_asg` 模块创建自动伸缩组，使用 `ec2_lc` 模块创建自动伸缩配置，等等。

使用 Ansible 管理 EC2 是一个足够大的话题，你甚至可以为它写一本完整的书。事实上，Yan Kurniawan 就正在写一本关于 Ansible 和 AWS 的书。在消化理解了本章的内容之后，你应该已经拥有了足够的知识，可以毫无障碍地学会那些其他的模块了。

Docker

Docker 项目如风暴般风靡整个 IT 行业。我甚至找不到任何其他的技术可以像 Docker 一样如此迅速地被社区所热捧。本章会阐述如何使用 Ansible 来创建 Docker 镜像以及部署 Docker 容器。

什么是容器

容器是虚拟化技术的一种形式。当你使用虚拟化技术在 GuestOS 中运行进程时，在运行于物理服务器上的 HostOS 中，GuestOS 中的进程是不可见的。尤其是，GuestOS 中的进程不能直接访问物理资源，即使这些进程拥有 root 权限也无法实现。

容器时常被称为操作系统级虚拟化以和硬件虚拟化技术相区别。

在硬件虚拟化中，一个名为 *hypervisor* 的程序虚拟化出了一台完整的物理主机，包含虚拟的 CPU、内存以及像硬盘和网卡这样的设备。由于整台机器都是虚拟的，所以硬件虚拟化是非常灵活的。特别是，你可以在虚拟机中运行一个与宿主机完全不同的操作系统（比如，你可以在一台操作系统为 RedHat Enterprise Linux 的宿主机中运行一个操作系统为 Windows Server 2012 的虚拟机），并且你可以像操作一台物理机一样暂停或恢复一台虚拟机。这种灵活性带来了需要对硬件进行虚拟化的额外的性能开销。

在操作系统级虚拟化中（比如容器），*guest* 进程被操作系统与宿主机隔离开。*guest* 进程与宿主机运行在相同的内核上。宿主机操作系统负责确保 *guest* 进程完全与宿主机隔离开。当运行一个像 Docker 这样的基于 Linux 的容器程序时，*guest* 进程也必须是 Linux 程序。不管怎样，容器的性能开销比硬件虚拟化要少得多，因为你只是运行了一个操作系统。特别是，进程在容器中的启动速度远比在虚拟机内部要快。

Docker 不仅仅是容器，可以把 Docker 想象成用容器堆砌成的平台。打个比方，容器之于 Docker，就好像虚拟机之于 IaaS 云一样。另外两个组成 Docker 的主要组件就是镜像格式和 Docker API。

你可以把 Docker 镜像想象成虚拟机镜像。一个 Docker 镜像包含一个已安装好操作系统的文件系统以及一些元数据。一个重要的不同点就是：Docker 镜像是分层的。你可以通过在一个已经存在的 Docker 镜像上创建一个新的 Docker 镜像并且通过增加、修改或删除某些文件来修改它。新生成的 Docker 镜像包含对原有 Docker 镜像的引用，以及新旧 Docker 镜像的文件系统差异。例如，官方的 Nginx docker 镜像 (<https://github.com/nginxinc/docker-nginx/blob/master/Dockerfile>) 是在官方的 Debian Wheezy 镜像的基础上创建的新层。这种分层的方法意味着 Docker 镜像比传统的虚拟机镜像要小得多，所以在网络上传输 Docker 镜像要比传输传统虚拟机镜像更快。Docker 项目维护了一个公共镜像仓库（Docker Hub：<https://hub.docker.com/>）。

Docker 也支持远程 API 调用，用来与第三方工具相互配合。特别说明的是，Ansible 的 docker 模块就是使用 Docker 的远程 API。

Docker 与 Ansible 配合案例

Docker 容器可以很便捷地将你的应用程序打包成一个易于向不同地方部署的单一的镜像。这也是为什么 Docker 项目要隐喻集装箱。Docker 的远程 API 简化了在 Docker 上运行的软件系统的自动化流程。

Ansible 可以在两个领域简化 Docker 的工作。其一就是 Docker 容器的编配。当你部署一个“Docker 化”的软件应用时，你通常会创建包含着不同服务的多个 Docker 容器。这些服务需要互相进行通信，因此你需要正确地连接合适的容器，并且确保它们按照正确的顺序启动。起初，Docker 项目并没有提供编配工具，因此出现了一些第三方工具来填补这个缺口。Ansible 就是为编配而编写的，因此它天然适合部署基于 Docker 的应用。

另一个领域是创建 Docker 镜像。官方创建 Docker 镜像的方法是编写名为 *Dockerfiles* 的特殊文本文件，它类似于 Shell 脚本。对于简单的镜像来说，Dockfile 工作只是还好。然而，当你开始创建更复杂镜像的时候，你马上就会想念 Ansible 提供的强大功能了。幸运的是，你可以使用 Ansible 来创建 playbook。

Docker 应用的生命周期

下面是基于 Docker 的应用的典型生命周期。

1. 在本地主机上创建 Docker 镜像。
2. 将 Docker 镜像从本地主机推送到镜像仓库。
3. 将 Docker 镜像从镜像仓库中下载到远程主机。
4. 在远程主机上启动 Docker 容器，在容器启动的时候将所需的配置信息传给它。

你通常会在本地主机上创建自己的 Docker 镜像，或者在一个支持创建 Docker 镜像的持续集成系统中，例如 Jenkins 或者 CircleCI。一旦你完成了镜像的创建，你就需要将它存储在某个方便远程主机下载的地方。

Docker 镜像通常会放在一个被称为 *registry* 的仓库里。Docker 项目维护一个叫作“*Docker Hub*”的官方仓库，它同时提供公有和私有的 Docker 镜像托管服务，并且 Docker 的命令行工具默认支持推送（push）镜像到仓库和从仓库中下载（pull）镜像。

你的 Docker 镜像一旦放到 registry 中，你就可以连接到远程主机并下载镜像，然后运行容器。需要注意的是，如果你尝试运行一个镜像不在本地的容器，Docker 会自动从 registry 中下载镜像，所以你不需要主动输入命令从 registry 中下载镜像。

当你使用 Ansible 来创建 Docker 镜像并在远程主机上启动容器的时候，应用的生命周期看起来像是这样：

1. 编写用来创建 Docker 镜像的 Ansible playbook。
2. 在本地主机上运行 playbook 来创建 Docker 镜像。
3. 将 Docker 镜像从本地主机推送到仓库。
4. 编写用来在远程主机上下载镜像，在远程主机上启动容器同时传入配置信息的 Ansible playbook。
5. 运行 Ansible playbook 来启动容器。

容器化我们的 Mezzanine 应用

< 240

我们将使用我们的 Mezzanine 范例并将它部署在 Docker 容器中。回想一下，我们的应用包含如下的服务。

- Postgres database（关系型数据库）
- Mezzanine（Web 应用）
- Memcached（用来提升性能的内存缓存）

- Nginx (Web 服务器)

我们可以把所有的服务部署到同一个容器里。然而，为了教学目的，如图 13-1 所示，我将会把每一个服务分别运行在独立的容器中。将每个服务部署到独立的容器中制造了一定的部署复杂性，但是会让我更好地来演示如何用 Ansible 和 Docker 配合来处理复杂的工作。

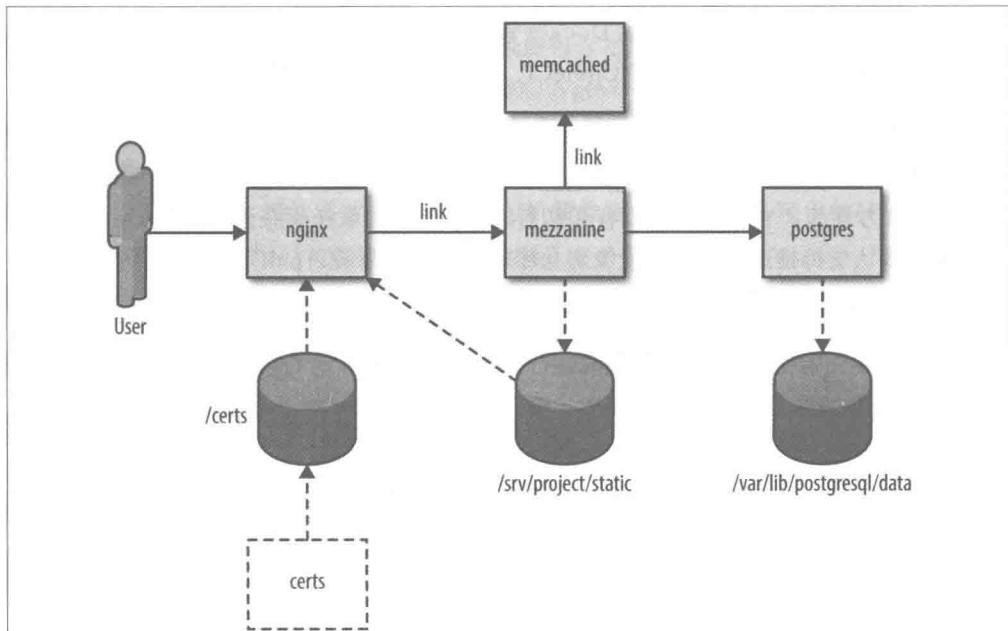


图13-1 用Docker容器部署Mezzanine应用

图 13-1 中的每个盒子表示运行一个服务的 Docker 容器。容器之间使用 TCP/IP 协议相互通信，用实线连接。Nginx 容器是唯一需要响应外部请求的容器。它将 Web 请求反向代理到 Mezzanine 应用，因此需要连接 Mezzanine 容器。Mezzanine 容器必须访问数据库，因此需要连接 Postgres 容器。Mezzanine 容器也需要连接 Memcached 容器以便于访问由 Memcached 提供的内存缓存来提升整体性能。

图中的圆柱体表示用于容器导入和导出的 Docker 数据卷。例如，Mezzanine 容器导出 /srv/project/static 数据卷，而 Nginx 容器则导入这个数据卷。

Nginx 服务必须提供静态内容，例如 JavaScript、CSS 和图片，也包括 Mezzanine 用户上传的文件。（回想一下 Mezzanine 是一个允许用户上传文件的（如图片）CMS 系统。）这些文件存在于 Mezzanine 容器中，而不是 Nginx 容器中。为了在容器之间共享文件，

我们配置 Mezzanine 容器将静态文件内容存储到数据卷中，然后我们将数据卷挂载到 Nginx 容器上。

只有运行在同一台主机上的容器才可以共享数据卷（在我们的部署中，*nginx* 和 *mezzanine*），在其他情况下我们都可以将容器分别部署到不同的主机上。在一个真实的部署场景中，我们倾向将 Memcached 容器与 Mezzanine 容器部署在同一台主机上，而把 Postgres 容器部署在其他的主机上。在我们的范例中，我需要使用容器链接技术（查看 247 页的“链接 Docker 容器”）将 Nginx 容器、Mezzanine 容器和 Memcached 容器链接在一起（故在图中添加 *link* 注释）。Mezzanine 容器将会通过 Postgres 容器暴露的端口与 Postgres 容器通信，为了演示两种互链容器的方法，我们将它们运行在一台主机上。

链接 Docker 容器

如果两个 Docker 容器运行在同一台主机上，你可以使用一种叫作容器链接 (*linking container*) 的技术，这样两个容器可以通过网络通信。链接是单向的，所以如果容器 *A* 被链接到容器 *B*，那么在容器 *A* 中的进程就可以连接容器 *B* 中运行的网络服务了。

Docker 会将特殊的环境变量注入某个容器。这些变量包括 IP 地址以及端口号，这样该容器可以访问其他容器中的服务，同时更新 */etc/hosts* 文件，这样容器就可以通过主机名访问其他容器。了解更多细节，可以查看官方 Docker 文档关于容器的部分 (<https://docs.docker.com/userguide/dockerlinks/>)。

最后，图中还有一个被标记为“certs”的虚线框。这是一个包含 TLS 证书的 Docker 数据卷。与其他容器不同的是，这个容器是停止的。它只是用来存储证书文件的。

使用 Ansible 创建 Docker 镜像

242

在本章中，我将会使用 Ansible 项目所推荐的方法来使用 Ansible 创建 Docker 镜像。简而言之，方法如下。

1. 使用官方基于 Ansible 的镜像，Ansible 已经被内置安装。
2. 在 Dockerfile 中，将 playbook 复制到镜像中。
3. 在 Dockerfile 中调用 Ansible。

值得注意的是，我们并不会使用 Ansible 来创建全部的镜像。在某种情况下，我们可以

在一个现成的地方使用已存在的镜像……呃……这个地方就是 Docker 仓库。另一些情况下，我们将会使用传统的 Dockerfile 方式来构建 Docker 镜像。

我们需要为图 13-1 中所描绘的每一个框创建一个 Docker 镜像。

Mezzanine

我们的 Mezzanine 容器镜像是最复杂的，并且我们将会使用 Ansible 来配置它。

官方的 Ansible 镜像托管在 Docker 仓库 (<https://hub.docker.com/>)。截止到本书编写完成时，有两个基础镜像可用：

- `ansible/centos7-ansible` (CentOS 7)
- `ansible/ubuntu14.04-ansible` (Ubuntu 14.04)

我们将会使用 Ubuntu 14.04 镜像。为了创建这个镜像，我需要创建一个 `mezzanine` 目录，目录包含如下文件。

- `Dockerfile`
- `ansible/mezzanine-container.yml`
- `ansible/files/gunicorn.conf.py`
- `ansible/files/local_settings.py`
- `ansible/files/scripts/setadmin.py`
- `ansible/files/scripts/setsite.py`

这些文件包括用来构建 Docker 镜像 Dockerfile、playbook 本身 (`mezzanine-container.yml`) 以及一些需要复制到镜像里的其他文件。

例 13-1 列出了构建 Mezzanine 镜像的 Dockerfile。

243

例 13-1 Mezzanine Dockerfile

```
FROM ansible/ubuntu14.04-ansible:stable
MAINTAINER Lorin Hochstein <lorin@ansiblebook.com>

ADD ansible /srv/ansible
WORKDIR /srv/ansible

RUN ansible-playbook mezzanine-container.yml -c local

VOLUME /srv/project/static

WORKDIR /srv/project
```

```
EXPOSE 8000
CMD ["gunicorn_django", "-c", "gunicorn.conf.py"]
```

我们先把 playbook 和相关的文件复制到容器中，然后执行 playbook。我们还为 `/srv/project/static` 目录创建一个挂载点，这个目录包含着由 Nginx 容器提供的静态内容。

最后，我们暴露出 8000 端口，并且指定 `gunicorn_django` 为容器的默认执行命令，它将会使用 Gunicorn 应用服务器来运行 Mezzanine。例 13-2 列出了我们用来配置容器的 playbook。

例 13-2 mezzanine-container.yml

```
- name: Create Mezzanine container
  hosts: local
  vars:
    mezzanine_repo_url: https://github.com/lorin/mezzanine-example.git
    mezzanine_proj_path: /srv/project
    mezzanine_reqs_path: requirements.txt
    script_path: /srv/scripts
  tasks:
    - name: install apt packages
      apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
      with_items:
        - git
        - gunicorn
        - libjpeg-dev
        - libpq-dev
        - python-dev
        - python-pip
        - python-psycopg2
        - python-setuptools

    - name: check out the repository on the host
      git:
        repo:"{{ mezzanine_repo_url }}"
        dest:"{{ mezzanine_proj_path }}"
        accept_hostkey=yes
        <244>

    - name: install required python packages
      pip: name={{ item }}
      with_items:
        - south
        - psycopg2
        - django-compressor
        - python-memcached
```

```
- name: install requirements.txt
  pip: requirements={{ mezzanine_proj_path }}/{{ mezzanine_reqs_path }}

- name: generate the settings file
  copy: src=files/local_settings.py dest={{ mezzanine_proj_path }}/
    local_settings.py

- name: set the gunicorn config file
  copy: src=files/gunicorn.conf.py dest={{ mezzanine_proj_path }}/gunicorn.conf.py

- name: collect static assets into the appropriate directory
  django_manage: command=collectstatic app_path={{ mezzanine_proj_path }}
    environment:
      # 如果私钥为空，将不会运行 collectstatic
      # 所以我们传入一个任意值
      SECRET_KEY: nonblanksecretkey

- name: script directory
  file: path={{ script_path }} state=directory

- name: copy scripts for setting site id and admin at launch time
  copy: src=files/scripts/{{ item }} dest={{ script_path }}/{{ item }} mode=0755
  with_items:
    - setadmin.py
    - setsite.py
```

例 13-2 的 playbook 和第 6 章里的 playbook 相似，但略有不同。

- 我们没有把第 5 章中讨论过的 Postgres、Nginx、Memcached 或者 Supervisor 安装到镜像里。
- 我们没有使用模板生成 *local_settings.py* 和 *gunicorn.conf.py* 文件。
- 我们没有运行 Django 的 *syncdb* 命令或 *migrate* 命令。
- 我们将 *setadmin.py* 和 *setsite.py* 脚本复制到容器里而不是执行它们。

245

我们并没有把其他的服务安装到镜像中，因为这些服务会被放到单独的容器中实现，除了 Supervisor。

为什么我们不再需要 Supervisor 服务

回顾一下，我们 Mezzanine 服务的部署过程原本是使用 Supervisor 来管理应用服务器（Gunicorn）。这意味着，Supervisor 负责启动和停止 Gunicorn 进程。

在我们的 Mezzanine Docker 容器中，我们不再需要单独的程序来负责 Gunicorn 进程的启动和停止。那是因为 Docker 本身就被设计成一个用来管理进程启动和停止的系统。

在没有使用 Docker 的情况下，我们会使用 Supervisor 来启动 Gunicorn：

```
$ supervisorctl start gunicorn_mezzanine
```

使用 Docker 情况下，我们启动一个包含 Gunicorn 的容器，然后我们使用 Ansible 来运行类似下面的命令：

```
$ docker run lorin/mezzanine:latest
```

我们没有使用模板来生成 *local_settings.py* 文件，这是因为当我们创建镜像的时候，我们并不知道设置应该是什么样的。比方说，我们不知道数据库的服务器地址、端口号、用户名和密码。即使我们知道，我们也不想将它们硬编码到镜像中，因为我们可能会在开发环境、仿真环境和生产环境中使用相同的镜像。

我们所需要的是一个服务发现机制，以便于我们可以在容器启动的时候确定所有的配置应该如何设置。有很多不同的方法可以实现服务发现，包括使用像 etcd、Consul、Apache ZooKeeper 或 Eureka 等服务发现工具。我们将会使用环境变量，因为 Docker 允许我们在启动容器的时候指定环境变量。如例 13-3 所示为我们在镜像中使用的 *local_settings.py* 文件的内容。

例 13-3 local_settings.py

```
from __future__ import unicode_literals
import os

SECRET_KEY = os.environ.get("SECRET_KEY", "")
NEVERCACHE_KEY = os.environ.get("NEVERCACHE_KEY", "")
ALLOWED_HOSTS = os.environ.get("ALLOWED_HOSTS", "")

DATABASES = {
    "default": {
        # 该配置的值需要以 "postgresql_psycopg2"、"mysql"、"sqlite3" 或者 "oracle" 结尾
        "ENGINE": "django.db.backends.postgresql_psycopg2",
        # 数据库的名字。但如果使用 sqlite3，则为数据库的路径。
        "NAME": os.environ.get("DATABASE_NAME", ""),
        # 使用 sqlite3 时，这个配置不使用
        "USER": os.environ.get("DATABASE_USER", ""),
        # 使用 sqlite3 时，这个配置不使用
        "PASSWORD": os.environ.get("DATABASE_PASSWORD", ""),
        # 对于本机，可以设置为空字符串。使用 sqlite3 时，这个配置不使用
        "HOST": os.environ.get("DATABASE_HOST", ""),
        # 该配置的值默认为空字符串。使用 sqlite3 时，这个配置不使用
        "PORT": os.environ.get("DATABASE_PORT", "")
    }
}

SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTOCOL", "https")
```

<246

```
CACHE_MIDDLEWARE_SECONDS = 60  
CACHE_MIDDLEWARE_KEY_PREFIX = "mezzanine"  
  
CACHES = {  
    "default": {  
        "BACKEND": "django.core.cache.backends.memcached.MemcachedCache",  
        "LOCATION": os.environ.get("MEMCACHED_LOCATION", "memcached:11211"),  
    }  
}  
  
SESSION_ENGINE = "django.contrib.sessions.backends.cache"  
  
TWITTER_ACCESS_TOKEN_KEY = os.environ.get("TWITTER_ACCESS_TOKEN_KEY ", "")  
TWITTER_ACCESS_TOKEN_SECRET = os.environ.get("TWITTER_ACCESS_TOKEN_SECRET ", "")  
TWITTER_CONSUMER_KEY = os.environ.get("TWITTER_CONSUMER_KEY ", "")  
TWITTER_CONSUMER_SECRET = os.environ.get("TWITTER_CONSUMER_SECRET ", "")  
TWITTER_DEFAULT_QUERY = "from:ansiblebook"
```

需要注意的是，在例 13-3 中的大部分设置都是通过调用 `os.environ.get` 来引用环境变量的。

对于大部分设置来说，如果环境变量不存在，我们并没有使用一个有意义的默认值。但是 Memcached 服务器的地址配置是一个例外：

```
"LOCATION": os.environ.get("MEMCACHED_LOCATION", "memcached:11211"),
```

这样的默认值正好可以处理我们使用容器链接技术的情况。如果我在运行时使用 `memcached` 这个名字链接 Memcached 容器，那么 Docker 将会自动把 `memcached` 的这个主机名解析到 Memcached 容器的 IP 地址。

例 13-4 列出了 Gunicorn 的配置文件。当然我们可能可以将端口号硬编码为 8000 以蒙混过关，但是我希望允许用户通过定义 `GUNICORN_PORT` 环境变量来覆盖这个配置。

247 例 13-4 `gunicorn.conf.py`

```
from __future__ import unicode_literals  
import multiprocessing  
import os  
  
bind = "0.0.0.0:{}".format(os.environ.get("GUNICORN_PORT", 8000))  
workers = multiprocessing.cpu_count() * 2 + 1  
loglevel = "error"  
proc_name = "mezzanine"  
  
setadmin.py 和 setsite.py 文件与原来在例 6-17 和例 6-18 中的保持不变。我们把这些文件复制到容器中以便于我们可以在部署的时候调用它们。在我们原来的 playbook 中，我们在部署的时候复制这些文件到远程主机上并执行它们，但是 Docker 目前还不支持可以
```

让我们在容器运行的时候将文件复制到容器中的简单方法。所以作为替代方案，我们只能在构建的时候将它们复制到镜像中。

其他的容器镜像

我们的 Mezzanine 范例中还使用了一些没有使用 Ansible 配置的其他 Docker 镜像。

Postgres

我们需要一个运行 Postgres 服务的镜像。幸运的是，在 Docker registry (https://registry.hub.docker.com/_/postgres/) 中有 Postgres 项目提供的官方镜像。我将使用官方镜像，所以就不需要自己再创建了。明确地，我会使用包含 Postgres 9.4 版本的镜像，它的官方镜像名称为 `postgres:9.4`。

Memcached

Memcached 并没有官方的镜像，不过使用 Dockerfile 来构建一个 Memcached 镜像是十分简单的，如例 13-5 所示。

例 13-5 Memcached 的 Dockerfile

```
FROM ubuntu:trusty
MAINTAINER lorin@ansiblebook.com

# 本文件基于 Digital Ocean 的教程：
# https://www.digitalocean.com/community/tutorials/docker-explained-how-to-
# create-docker-containers-running-memcached

# 更新默认应用仓库源列表
RUN apt-get update

# 安装 Memcached
RUN apt-get install -y memcached

# 暴露服务端口（默认：11211）
EXPOSE 11211

# 设置 Memcached 的默认运行参数
CMD ["-m", "128"]

# 设置运行 Memcached 守护进程的用户
USER daemon

# 设置 Memcached 二进制文件的启动程序
ENTRYPOINT memcached
```

248

Nginx

我们可以使用用于构建 Nginx 镜像的官方 Dockerfile (<https://github.com/dockerfile/nginx/blob/master/Dockerfile>)。我们需要让 Nginx 使用我们自己的配置文件，使其反向代理 Mezzanine 应用。官方 Nginx 镜像的配置允许我们将自定义的 `nginx.conf` 放到宿主机本地文件系统中并 mount 到容器中。然而，我更喜欢创建一个不需要依赖容器外配置文件的自包含 Docker 镜像。

我们可以基于官方的镜像并加入我们自定义的 Nginx 配置文件来构建一个新的镜像。如例 13-6 所示为相应的 Dockerfile，而如例 13-7 所示为我们所使用的自定义 Nginx 配置文件。

例13-6 用于自定义Nginx Docker镜像的Dockerfile

```
FROM nginx:1.7

RUN rm /etc/nginx/conf.d/default.conf \
    /etc/nginx/conf.d/example_ssl.conf
COPY nginx.conf /etc/nginx/conf.d/mezzanine.conf
```

例13-7 用于Nginx Docker镜像的nginx.conf

```
upstream mezzanine {
    server mezzanine:8000;
}

server {
    listen 80;

    listen 443 ssl;

    client_max_body_size 10M;
    keepalive_timeout 15;
    ssl_certificate      /certs/nginx.crt;
    ssl_certificate_key  /certs/nginx.key;
    ssl_session_cache    shared:SSL:10m;
    ssl_session_timeout  10m;
    ssl_ciphers          (too long to show here);
    ssl_prefer_server_ciphers on;

    location / {
        proxy_redirect      off;
        proxy_set_header    Host $host;
        proxy_set_header    X-Real-IP $remote_addr;
        proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header    X-Forwarded-Protocol $scheme;
    }
}
```

```

    proxy_pass      http://mezzanine;
}

location /static/ {
    root          /srv/project;
    access_log    off;
    log_not_found off;
}

location /robots.txt {
    root          /srv/project/static;
    access_log    off;
    log_not_found off;
}

location /favicon.ico {
    root          /srv/project/static/img;
    access_log    off;
    log_not_found off;
}
}

```

Nginx 并不原生支持从环境变量中获取配置，因此我们需要使用一个已知的静态内容地址的路径 (`/srv/project/static`)。我们指定 Mezzanine 服务的地址为 `mezzanine:8000`；当我们把 Nginx 容器链接到 Mezzanine 容器时，Docker 将会确保把主机名 `mezzanine` 解析到 Mezzanine 容器的 IP 地址。

certs

`certs` 的 Docker 镜像是一个文件，它包含着需要被 Nginx 服务所使用的 TLS 证书。在真实的场景下，我们会使用证书权威机构签发的证书。但是为了演示的目的，这个镜像的 Dockerfile 为 `http://192.168.59.103.xip.io` 生成了一个自签发证书，如例 13-8 所示。

例 13-8 certs 镜像的 Dockerfile

```

FROM ubuntu:trusty
MAINTAINER lorin@ansiblebook.com

# 为 192.168.59.103 创建自签发证书
RUN apt-get update
RUN apt-get install -y openssl

RUN mkdir /certs

WORKDIR /certs

RUN openssl req -new -x509 -nodes -out nginx.crt \

```

250

```
-keyout nginx.key -subj '/CN=192.168.59.103.xip.io' -days 3650  
VOLUME /certs
```

构建镜像

我没有使用 Ansible 本身来构建 Docker 镜像，而是用命令行的方式来构建。例如，如下命令用于构建 Mezzanine 镜像：

```
$ cd mezzanine  
$ docker build -t lorin/mezzanine
```

Ansible 也包含一个用来构建 Docker 镜像的模块，叫作 `docker_image`。但是，这个模块已经被弃用了，因为构建镜像并不适合使用像 Ansible 这样的工具。镜像构建是部署一个应用程序生命周期的一部分；构建 Docker 镜像以及将镜像推送到镜像仓库的工作应该属于持续集成系统的工作，而不是配置管理系统。

251 部署 Docker 化的应用



我们使用 `docker` 模块来部署应用。截止到本书编写完成的时候，Ansible 随附的 `docker` 模块仍然有一些已知问题。

- 与最新版本的 Docker 配合时 `volumes_from` 参数不工作。
- 不支持 Boot2Docker，一个用于在 Mac OS X 上运行 Docker 的常用工具。
- 不支持 `wait` 参数，我在本节的一些例子中会使用它。

在 Ansible 项目中，为上面所有的问题所提出的解决建议正在被等待审查。如果幸运的话，希望你读到这里的时候，这些问题都已经被修复了。此外，还有一个支持 `detach=no` 功能的请求在等待处理，它与范例中的 `wait=yes` 具有相同的行为。同时，我已经将一个已经修复了上述问题的 `docker` 模块的自定义版本放到示例代码仓库里面。这个文件为：`ch13/playbooks/library/docker.py`。

例 13-10 列出了在我们的 Mezzanine 部署中编配 Docker 容器的完整 playbook。敏感数据放在了其他文件中，如例 13-11 所示。你可以将其看作开发过程中的设置，因为所有的服务都运行在控制主机上。

需要注意的是，我们是在 Mac OS X 上使用 Boot2Docker 来运行它们的，因此 Docker 容器实际上是运行在一个虚拟机中，而不是在本地主机运行。这也意味着我可以调用

Docker 而不需要使用 root 权限。如果你在 Linux 上运行，你将需要使用 sudo 或者直接使用 root 来运行。

因为这是一个很大的 playbook，我们把它分解一下。

启动数据库容器

下面展示了我们如何启动容器来运行 Postgres 数据库。

```
- name: start the postgres container
  docker:
    image: postgres:9.4
    name: postgres
    publish_all_ports: True
    env:
      POSTGRES_USER: "{{ database_user }}"
      POSTGRES_PASSWORD: "{{ database_password }}"
```

无论你何时启动一个 Docker 容器，你都必须指定镜像。如果在本地没有安装 postgres:9.4 镜像，Docker 将会在它第一次运行的时候下载。我们指定 publish_all_ports 参数为 true，这样 Docker 会把容器所配置对外暴露的端口号开放出来，这里的端口号是 5432。◀ 252

这个容器是通过环境变量配置的，因此我们需要使用 env 传入有权限访问服务的用户名和密码。Postgres 镜像会自动创建一个与用户名相同的数据库。

获取数据库容器的 IP 地址和映射端口

当我们启动 Postgres 容器的时候，我们可以明确地将容器的数据库端口（5432）映射到已知的主机端口上（在 Nginx 容器上我们就是这么做的）。由于我们并没有这样做，Docker 将会在主机上选择一个随机的端口来与容器内部的 5432 端口建立映射关系。

接下来，在 playbook 中，我们需要知道这个随机端口是什么，因为我们需要在启动 Mezzanine 之前等待 Postgres 服务启动完成，并且我们还需要检测是否已经有进程在监听那个映射的端口了。

我们可以配置 Mezzanine 容器连接到所映射的端口上，但是我决定让 Mezzanine 容器连接 Postgres 容器 IP 地址上的 5432 端口。这样也就给了我一个演示如何获取 Docker 容器 IP 地址的理由。

当 Docker 模块启动一个或更多的容器的时候，它会将已启动的容器信息设置为 fact。这

意味着，我们不需要使用 `register` 语句来捕获模块调用结果；我们只需要知道包含着我们将要查找的容器信息所对应的名字。

包含信息的 fact 的名称是 `docker_containers`，这是包含容器信息的字典组成的列表。它与你使用 `docker inspect` 命令后看到的输出结果相同。如例 13-9 所示为启动 Postgres 容器后，`docker_containers` fact 值的范例。

例 13-9 启动 Postgres 容器后的 `docker_containers` fact

```
[  
  {  
    "AppArmorProfile": "",  
    "Args": [  
      "postgres"  
    ],  
    "Config": {  
      "AttachStderr": false,  
      "AttachStdin": false,  
      "AttachStdout": false,  
      "Cmd": [  
        "postgres"  
      ],  
      "CpuShares": 0,  
      "Cpuset": "",  
      "Domainname": "",  
      "Entrypoint": [  
        "/docker-entrypoint.sh"  
      ],  
      "Env": [  
        "POSTGRES_PASSWORD=password",  
        "POSTGRES_USER=mezzanine",  
        "PATH=/usr/lib/postgresql/9.4/bin:/usr/local/sbin:/usr/local/bin:  
/usr/sbin:/usr/bin:/sbin:/bin",  
        "LANG=en_US.utf8",  
        "PG_MAJOR=9.4",  
        "PG_VERSION=9.4.0-1.pgdg70+1",  
        "PGDATA=/var/lib/postgresql/data"  
      ],  
      "ExposedPorts": {  
        "5432/tcp": {}  
      },  
      "Hostname": "71f40ec4b58c",  
      "Image": "postgres",  
      "MacAddress": "",  
      "Memory": 0,  
    }  
}
```

```
"MemorySwap": 0,
"NetworkDisabled": false,
"OnBuild": null,
"OpenStdin": false,
"PortSpecs": null,
"StdinOnce": false,
"Tty": false,
"User": "",
"Volumes": {
    "/var/lib/postgresql/data": {}
},
"WorkingDir": ""

},
"Created": "2014-12-25T22:59:15.841107151Z",
"Driver": "aufs",
"ExecDriver": "native-0.2",
"HostConfig": {
    "Binds": null,
    "CapAdd": null,
    "CapDrop": null,
    "ContainerIDFile": "",
    "Devices": null,
    "Dns": null,
    "DnsSearch": null,
    "ExtraHosts": null,
    "IpcMode": "",
    "Links": null,
    "LxcConf": null,
    "NetworkMode": "",
    "PortBindings": {
        "5432/tcp": [
            {
                "HostIp": "0.0.0.0",
                "HostPort": ""
            }
        ]
    },
    "Privileged": false,
    "PublishAllPorts": false,
    "RestartPolicy": {
        "MaximumRetryCount": 0,
        "Name": ""
    },
    "SecurityOpt": null,
    "VolumesFrom": [
```

254

```
        "data-volume"
    ]
},
"HostnamePath": "/mnt/sda1/var/lib/docker/containers/71f40ec4b58c3176030274afb025fb3eb130fe79d4a6a69de473096f335e7eb/hostname",
"HostsPath": "/mnt/sda1/var/lib/docker/containers/71f40ec4b58c3176030274afb025fb3eb130fe79d4a6a69de473096f335e7eb/hosts",
"Id": "71f40ec4b58c3176030274afb025fb3eb130fe79d4a6a69de473096f335e7eb",
"Image": "b58a816df10fb20c956d39724001d4f2fabdddec50e0d9099510f0eb579ec8a45",
"MountLabel": "",
"Name": "/high_lovelace",
"NetworkSettings": {
    "Bridge": "docker0",
    "Gateway": "172.17.42.1",
    "IPAddress": "172.17.0.12",
    "IPPrefixLen": 16,
    "MacAddress": "02:42:ac:11:00:0c",
    "PortMapping": null,
    "Ports": {
        "5432/tcp": [
            {
                "HostIp": "0.0.0.0",
                "Hostport": "49153",
            }
        ]
    },
    "Path": "/docker-entrypoint.sh",
    "ProcessLabel": "",
    "ResolvConfPath": "/mnt/sda1/var/lib/docker/containers/71f40ec4b58c3176030274afb025fb3eb130fe79d4a6a69de473096f335e7eb/resolv.conf",
    "State": {
        "Error": "",
        "ExitCode": 0,
        "FinishedAt": "2001-01-01T00:00:00Z",
        "OOMKilled": false,
        "Paused": false,
        "Pid": 9625,
        "Restarting": false,
        "Running": true,
        "StartedAt": "2014-12-25T22:59:16.219732465Z"
    },
    "Volumes": {
        "/var/lib/postgresql/data": "/mnt/sda1/var/lib/docker/vfs/dir/4cccd3150c8d74b9b0feb56df928ac915599e12c3ab573cd4738a18fe3dc6f474"
    }
}
```

255

```
        },
        "VolumesRW": {
            "/var/lib/postgresql/data": true
        }
    }
]
```

如果你认真地读完了上面的输出，你可以看到 IP 地址和映射端口号是在上面数据结构的 NetworkSettings 部分：

```
"NetworkSettings": {
    "Bridge": "docker0",
    "Gateway": "172.17.42.1",
    "IPAddress": "172.17.0.12",
    "IPPrefixLen": 16,
    "MacAddress": "02:42:ac:11:00:0c",
    "PortMapping": null,
    "Ports": [
        "5432/tcp": [
            {
                "HostIp": "0.0.0.0",
                "HostPort": "49153"
            }
        ]
    ],
}
```

下面展示了我们如何将 IP 地址（172.17.0.17）和映射端口号（49153）提取出来并使用 set_fact 模块赋值给变量：[source, yaml+jinja]

```
- name: capture database ip address and mapped port
  set_fact:
    database_host: "{{ docker_containers[0].NetworkSettings.IPAddress }}"
    mapped_database_port: "{{ docker_containers[0].NetworkSettings.Ports['5432/tcp'][0].HostPort }}"
```

等待数据库启动

<256

官方的 Postgres Docker 镜像的文档（<https://github.com/dockerfile/nginx/blob/master/Dockerfile>）包含了如下的警告：

当 Postgres 在容器中启动的时候，如果没有任何数据库，那么 Postgres 会为你创建一个默认的数据库。虽然这是 Postgres 的预期行为，但是这意味着在这段时间内它无法接受传入的连接请求。当你使用自动化工具的时候可能会产生问题，比如 fig

这种同时启动多个容器的工具。

对于 `wait_for` 模块来说，这是一个极好的使用场景，它会阻塞 playbook 的执行过程直到服务接受 TCP 连接请求：

```
- name: wait for database to come up
  wait_for: host={{ docker_host }} port={{ mapped_database_port }}
```

注意 `docker_host` 变量指定运行 Docker 的主机的用法。下面部分展示了如何在 `vars` 区段定义这个变量。为了清晰起见我加了一个换行符，但是它们应该处于同一行。

```
docker_host: "{{ lookup('env', 'DOCKER_HOST') | regex_replace('^tcp://(.*):\d+$', '\\\\1') | default('localhost', true) }}"
```

这里的问题是 `docker_host` 的内容将会取决于你在哪里运行 Docker：是直接在你的控制主机上运行于 Linux 系统中？还是在 Mac OS X 上使用 Boot2Docker 在虚拟机中运行？

如果你在本地运行 Docker，那 `docker_host` 需要设置为 `localhost`。如果你使用 Boot2Docker 运行，那需要把它设置为虚拟机的 IP 地址。

如果你在运行 Boot2Docker，那么你需要定义一个名为 `DOCKER_HOST` 的环境变量。我的设置如下所示：

```
DOCKER_HOST=tcp://192.168.59.103:2375
```

如果 `DOCKER_HOST` 已经定义，我需要提取 `192.168.59.103` 这一部分。如果没有定义，那我希望它的默认值是 `localhost`。

我使用 `env` `lookup` 插件来取得 `DOCKER_HOST` 环境变量的值：

```
lookup('env', 'DOCKER_HOST')
```

为了提取 IP 地址，我使用了 `regex_replace` 过滤器，这是一个由 Ansible 定义的定制 Jinja2 过滤器，它允许你处理正则表达式（注意反斜线的数量）。

```
regex_replace('^tcp://(.*):\d+$', '\\1')
```

257 最后，如果 `DOCKER_HOST` 环境变量没有定义，我使用了标准的 Jinja2 过滤器 `default` 将变量 `docker_host` 的默认值设置为 `localhost`。由于 `env` `lookup` 会返回一个空字符串，因此我需要将 `default` 过滤器的第 2 个参数设置为 `true`，以保证过滤器可以正常工作。可以查阅 Jinja2 文档 (<http://jinja.pocoo.org/docs/dev/templates/#default>) 了解更多细节。

```
default('localhost', true)
```

初始化数据库

我们需要运行 Django 的 `syncdb` 和 `migrate` 命令来初始化数据库（在 Django 1.7 版本中，你只需要运行 `migrate` 命令即可，但是 Mezzanine 默认使用了 Django 1.6 版本）。

我们需要运行 Mezzanine 容器来执行这个操作，而不是运行 Gunicorn。我们想要传递恰当的 `syncdb` 和 `migrate` 命令，并且运行 `setsite.py` 和 `setadmin.py` 脚本来设置网站 ID 和管理员密码。

```
- name: initialize database
  docker:
    image: lorin/mezzanine:latest
    command: python manage.py {{ item }} --noinput
    wait: True
    env: "{{ mezzanine_env }}"
  with_items:
    - syncdb
    - migrate

- name: set the site id
  docker:
    image: lorin/mezzanine:latest
    command: /srv/scripts/setsite.py
    env: "{{ setsite_env.update(mezzanine_env) }}{{ setsite_env }}"
    wait: yes

- name: set the admin password
  docker:
    image: lorin/mezzanine:latest
    command: /srv/scripts/setadmin.py
    env: "{{ setadmin_env.update(mezzanine_env) }}{{ setadmin_env }}"
    wait: yes
```

我们使用 `command` 参数来指定 `syncdb` 和 `migrate` 命令。

我们使用 `wait` 参数来使模块在进程完成前一直阻塞。否则我们会在数据库还没有配置完成的情况下启动 Mezzanine，这可能会引起竞争。

需要注意 `env` 参数传递包含配置信息的环境变量的用法，这些配置信息包括如何连接数据库服务等。我将所有的环境变量都放入 `mezzanine_env` 变量中，它的定义如下所示：

```
mezzanine_env:
  SECRET_KEY: "{{ secret_key }}"
  NEVERCACHE_KEY: "{{ nevercache_key }}"
  ALLOWED_HOSTS: "*"
```

258

```
DATABASE_NAME: "{{ database_name }}"
DATABASE_USER: "{{ database_user }}"
DATABASE_PASSWORD: "{{ database_password }}"
DATABASE_HOST: "{{ database_host }}"
DATABASE_PORT: "{{ database_port }}"
GUNICORN_PORT: "{{ gunicorn_port }}"
```

当我需要设置网站 ID 时，我需要添加两个额外的环境变量，我把它们定义在 `setsite_env` 变量中：

```
setsite_env:
  PROJECT_DIR: "{{ project_dir }}"
  WEBSITE_DOMAIN: "{{ website_domain }}"
```

我们需要将 `mezzanine_env` 和 `setsite_env` 自动合并成一个单独的字典并传递 `env` 参数。

不幸的是，Ansible 中没有可以简单合并两个字典的方法，但是有一个替代方法。Jinja2 提供了一个 `update` 方法允许你将一个字典合并到另一个字典中。然而问题是调用该方法的时候并不会返回合并后的字典，它只是更新了字典的状态。因此，你需要先调用 `update` 方法，然后再对变量求值。`env` 参数最后的结果如下所示：

```
env: "{{ setsite_env.update(mezzanine_env) }}{{ setsite_env }}"
```

启动 Memcached 容器

Memcached 容器的启动十分直截了当。因为 Memcached 不需要任何配置信息，所以我们甚至不需要传入任何环境变量。我们也不需要开放任何端口，因为只有 Mezzanine 容器会通过链接来连接它。

```
- name: start the memcached container
  docker:
    image: lorin/memcached:latest
    name: memcached
```

启动 Mezzanine 容器

我们将 Mezzanine 容器链接到 Memcached 容器，并且通过环境变量传递配置信息。

我们还使用相同的镜像启动另一个容器来运行 cron，因为 Mezzanine 需要使用 cron 从 Twitter 更新信息：

```
- name: start the mezzanine container
  docker:
    image: lorin/mezzanine:latest
```

```
name: mezzanine
env: "{{ mezzanine_env }}"
links: memcached

- name: start the mezzanine cron job
  docker:
    image: lorin/mezzanine:latest
    name: mezzanine
    env: "{{ mezzanine_env }}"
    command: cron -f
```

启动证书容器

我们启动保存 TLS 证书的容器。之前提到这个容器并不会运行服务，但是我们需要启动它，以便于我们将数据卷挂载到 Nginx 容器中。

```
- name: start the cert container
  docker:
    image: lorin/certs:latest
    name: certs
```

启动 Nginx 容器

最后，我们启动 Nginx 容器。容器需要对外暴露 80 端口和 443 端口。它还需要挂载静态内容和 TLS 证书的数据卷。最后，我们把它链接到 Mezzanine 容器使得 Mezzanine 主机名可以解析到运行 Mezzanine 的容器。

```
- name: run nginx
  docker:
    image: lorin/nginx-mezzanine:latest
    ports:
      - 80:80
      - 443:443
  name: nginx
  volumes_from:
    - mezzanine
    - certs
  links: mezzanine
```

就是它啦！如果你是在本地的 Linux 主机上运行 Docker，你现在可以通过 `http://localhost` 和 `https://localhost` 访问 Mezzanine。如果你是在 Mac OS X 上运行 Boot2Docker，你应该能够通过你的 Boot2Docker 虚拟机的 IP 地址访问它。如下操作可以获得 Boot2Docker 的虚拟机 IP 地址：

```
boot2docker ip
```

260 在我的机器上，地址是：<http://192.168.59.103> 和 <https://192.168.59.103>，或者你可以使用 xip.io 工具通过 <https://192.168.59.103.xip.io/f1326.20> 访问。

完整的 playbook

例 13-10 列出了完整的 playbook，而例 13-11 列出了 *secrets.yml* 文件的内容。

例13-10 run-mezzanine.yml

```
#!/usr/bin/env ansible-playbook
---
- name: run mezzanine from containers
  hosts: localhost
  vars_files:
    - secrets.yml
  vars:
    # postgres 容器为数据库和用户使用相同的名字
    database_name: mezzanine
    database_user: mezzanine
    database_port: 5432
    gunicorn_port: 8000
    docker_host: "{{ lookup('env', 'DOCKER_HOST') | regex_replace('^\^tcp://(.*):\\\\d+$', '\\\\\\1') | default('localhost', true) }}"
    project_dir: /srv/project
    website_domain: "{{ docker_host }}.xip.io"
    mezzanine_env:
      SECRET_KEY: "{{ secret_key }}"
      NEVERCACHE_KEY: "{{ nevercache_key }}"
      ALLOWED_HOSTS: "*"
      DATABASE_NAME: "{{ database_name }}"
      DATABASE_USER: "{{ database_user }}"
      DATABASE_PASSWORD: "{{ database_password }}"
      DATABASE_HOST: "{{ database_host }}"
      DATABASE_PORT: "{{ database_port }}"
      GUNICORN_PORT: "{{ gunicorn_port }}"
    setadmin_env:
      PROJECT_DIR: "{{ project_dir }}"
      ADMIN_PASSWORD: "{{ admin_password }}"
    setsite_env:
      PROJECT_DIR: "{{ project_dir }}"
      WEBSITE_DOMAIN: "{{ website_domain }}"
  tasks:
    - name: start the postgres container
      docker:
```

```

image: postgres:9.4
name: postgres
publish_all_ports: True
env:
  POSTGRES_USER: "{{ database_user }}"
  POSTGRES_PASSWORD: "{{ database_password }}"
- name: capture database ip address and mapped port
  set_fact:
    database_host: "{{ docker_containers[0].NetworkSettings.IPAddress }}"
    mapped_database_port: "{{ docker_containers[0].NetworkSettings.Ports['5432/tcp'][0].HostPort}}"
- name: wait for database to come up
  wait_for: host={{ docker_host }} port={{ mapped_database_port }}
- name: initialize database
  docker:
    image: lorin/mezzanine:latest
    command: python manage.py {{ item }} --noinput
    wait: True
    env: "{{ mezzanine_env }}"
  with_items:
    - syncdb
    - migrate
  register: django
- name: set the site id
  docker:
    image: lorin/mezzanine:latest
    command: /srv/scripts/setsite.py
    env: "{{ setsite_env.update(mezzanine_env) }}{{ setsite_env }}"
    wait: yes
- name: set the admin password
  docker:
    image: lorin/mezzanine:latest
    command: /srv/scripts/setadmin.py
    env: "{{ setadmin_env.update(mezzanine_env) }}{{ setadmin_env }}"
    wait: yes
- name: start the memcached container
  docker:
    image: lorin/memcached:latest
    name: memcached
- name: start the mezzanine container

```

```
262 ➤
docker:
  image: lorin/mezzanine:latest
  name: mezzanine
  env: "{{ mezzanine_env }}"
  links: memcached

- name: start the mezzanine cron job
  docker:
    image: lorin/mezzanine:latest
    name: mezzanine
    env: "{{ mezzanine_env }}"
    command: cron -f

- name: start the cert container
  docker:
    image: lorin/certs:latest
    name: certs

- name: run nginx
  docker:
    image: lorin/nginx-mezzanine:latest
    ports:
      - 80:80
      - 443:443
    name: nginx
    volumes_from:
      - mezzanine
      - certs
    links: mezzanine
```

例13-11 secrets.yml

```
database_password: password
secret_key: randomsecretkey
nevercache_key: randomnevercachekey
admin_password: password
```

由于 Docker 项目还相对年轻，许多的工具仍然在不断变化，并且 Docker 的部署模式也在不断演变。许多新兴的工具很可能与使用 Ansible 编配容器的功能有所重叠。

甚至，即使有另一个工具崭露头角并最终统治了 Docker 编配的世界，我相信 Ansible 仍然会是开发人员与运维人员手中的神兵利器。

调试 Ansible playbook

错误总会发生，我们需要直面它。不管是 playbook 中的一个 bug，还是你控制主机上的配置文件包含一个错误的配置值，终究会有些事情出乱子。在最后这一章，我将会回顾那些可以帮助你追踪错误的技术。

调试 SSH 问题

有时候，Ansible 不能成功地与主机建立 SSH 连接。当这个问题出现的时候，看到 Ansible 传递给底层 SSH 客户端的完整参数会比较实用。因为这可以帮助你在命令行手动复现这个问题。

如果调用 `ansible-playbook` 的时候使用 `-vvv` 参数，你就可以看到 Ansible 调用的完整 SSH 命令。这会很利于调试。

例 14-1 展示了一个 Ansible 在执行复制文件的模块时的输出范例。

例 14-1 当详细信息参数启用时的输出范例

```
TASK: [copy SSL key] ****
<127.0.0.1> ESTABLISH CONNECTION FOR USER: vagrant
<127.0.0.1> ESTABLISH CONNECTION FOR USER: vagrant
<127.0.0.1> EXEC ['ssh', '-C', '-tt', '-q', '-o', 'ControlMaster=auto', '-o',
'ControlPersist=60s', '-o', 'ControlPath=/Users/lorinhochstein/.ansible/cp/
ansible-ssh-%h-%p-%r', '-o', 'Port=2222', '-o', u'IdentityFile="/Users/
lorinhochstein/.vagrant.d/insecure_private_key"', '-o', 'KbdInteractive
Authentication=no', '-o', 'PreferredAuthentications=gssapi-with-mic,gssapi-keyex,
hostbased,publickey', '-o', 'PasswordAuthentication=no', '-o', 'User=vagrant',
'-o', 'ConnectTimeout=10', u'127.0.0.1', u'/bin/sh -c \'sudo -k && sudo -H -S -p
"[sudo via ansible, key=ypkyixkznnvqmrbmlhezlnlujtdhrcoam] password: " -u root
/bin/sh -c \'\"\\\"\\\"echo SUDO-SUCCESS-ypkyixkznnvqmrbmlhezlnlujtdhrcoam; rc=0;
```

```
[ -r "/etc/nginx/ssl/nginx.key" ] || rc=2; [ -f "/etc/nginx/ssl/nginx.key" ] ||  
rc=1; [ -d "/etc/nginx/ssl/nginx.key" ] && echo 3 && exit 0; (/usr/bin/md5sum  
/etc/nginx/ssl/nginx.key 2>/dev/null) || (/sbin/md5sum -q /etc/nginx/ssl/nginx.  
key 2>/dev/null) || (/usr/bin/digest -a md5 /etc/nginx/ssl/nginx.key 2>/dev/null)  
|| (/sbin/md5 -q /etc/nginx/ssl/nginx.key 2>/dev/null) || (/usr/bin/md5 -n /etc/  
nginx/ssl/nginx.key 2>/dev/null) || (/bin/md5 -q /etc/nginx/ssl/nginx.key 2>/dev/  
null) || (/usr/bin/csum -h MD5 /etc/nginx/ssl/nginx.key 2>/dev/null) || (/bin/  
csum -h MD5 /etc/nginx/ssl/nginx.key 2>/dev/null) || (echo "${rc} /etc/nginx/ssl/  
nginx.key")\''\``\``]
```

当调试连接问题的时候，有时你可能需要使用 `-vvvv` 参数，借此观察 SSH 客户端抛出的错误信息。

例如，如果你的主机没有启动 SSH，你将会看到如下错误：

```
testserver | FAILED => SSH encountered an unknown error. The output was:  
OpenSSH_6.2p2, OSSLShim 0.9.8r 8 Dec 2011  
debug1: Reading configuration data /etc/ssh_config  
debug1: /etc/ssh_config line 20: Applying options for *  
debug1: /etc/ssh_config line 102: Applying options for *  
debug1: auto-mux: Trying existing master  
debug1: Control socket "/Users/lorinhochstein/.ansible/cp/ansible-ssh-  
127.0.0.1-2222-vagrant" does not exist  
debug2: ssh_connect: needpriv 0  
debug1: Connecting to 127.0.0.1 [127.0.0.1] port 2222.  
debug2: fd 3 setting _NONBLOCK  
debug1: connect to address 127.0.0.1 port 2222: Connection refused  
ssh: connect to host 127.0.0.1 port 2222: Connection refused
```

如果你开启了 host key 验证，并且 `~/.ssh/known_hosts` 中的 host key 与服务器上的 host key 并不匹配的时候，那么使用 `-vvvv` 的情况下将会输出如下错误：

```
@@@@@@@  
@     WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!      @  
@@@@@@@  
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!  
Someone could be eavesdropping on you right now (man-in-the-middle attack)!  
It is also possible that a host key has just been changed.  
The fingerprint for the RSA key sent by the remote host is  
c3:99:c2:8f:18:ef:68:fe:ca:86:a9:f5:95:9e:a7:23.  
Please contact your system administrator.  
Add correct host key in /Users/lorinhochstein/.ssh/known_hosts to get rid of  
this message.  
Offending RSA key in /Users/lorinhochstein/.ssh/known_hosts:1  
RSA host key for [127.0.0.1]:2222 has changed and you have requested strict
```

```
checking.  
Host key verification failed.
```

如果你遇到这种情况，你需要在 `~/.ssh/known_hosts` 文件中删除冲突的条目。

265

debug 模块

我们已经在本书中多次使用 `debug` 模块了。它可以称得上是 Ansible 版的 `print` 语句。如例 14-2 所示，你可以使用它输出变量的值或者任意字符串。

例 14-2 debug 模块实践范例

```
- debug: var=myvariable  
- debug: msg="The value of myvariable is {{ var }}"
```

如同我们在第 4 章中讨论到的，你可以通过如下调用打印出所有与当前主机相关联的变量的值：

```
- debug: var=hostvars[inventory_hostname]
```

assert 模块

`assert` 模块会在指定的条件不符合的时候返回错误并失败退出。例如，下面配置在没有 `eth1` 网卡情况下会让 `playbook` 直接返回失败：

```
- name: assert that eth1 interface exists  
  assert:  
    that: ansible_eth1 is defined
```

当调试 `playbook` 的时候，插入 `assert` 模块在违反某些假定条件的时候立刻失败，会给调试带来很大帮助。

如果你希望检查主机文件系统中一些文件的状态，那么你可以先调用 `stat` 模块，然后再对 `stat` 模块的返回值调用 `assert` 模块：

```
- name: stat /opt/foo  
  stat: path=/opt/foo  
  register: st  
  
- name: assert that /opt/foo is a directory  
  assert:  
    that: st.stat.isdir
```

`stat` 模块收集关于文件路径状态的各种信息。它的返回值是一个字典，该字典包含 `stat`

字段，对应的值如表 14-1 所示。

266

表14-1 stat模块的返回值

字段	描述
atime	路径的最后访问时间，使用 UNIX 时间戳格式
ctime	路径的创建时间 ^{注1} ，使用 UNIX 时间戳格式
dev	inode 所在的设备 ID 编号
exists	如果路径存在返回 true
gid	路径的所属组 ID 编号
inode	inode 号
isblk	如果路径为指定块设备文件返回 true
ischr	如果路径为指定字符设备文件返回 true
isdir	如果路径为目录返回 true
isfifo	如果路径为 FIFO（命名管道）返回 true
isgid	如果文件设置了 setgid 返回 true
islnk	如果路径是符号链接返回 true
isreg	如果路径是常规文件返回 true
issock	如果路径是 UNIX 域 socket 返回 true
isuid	如果文件设置了 setuid 返回 true
mode	字符串格式的八进制文件模式（例如 1777）
mtime	路径最后修改时间，使用 UNIX 时间戳格式
nlink	文件硬链接的数量
pw_name	文件所属者的登录名
rgrp	如果设置所属组可读权限返回 true
roth	如果设置其他人可读权限返回 true
rusr	如果设置用户可读权限返回 true

注 1：原作者在这里犯了一个常见的错误。ctime 应该是 change time，变更时间。UNIX/Linux 操作系统并不记录文件的创建时间。ctime 和 mtime 的区别是：ctime 为元数据变更时间，而 mtime 为文件内容修改时间。——译者注

字段	描述
size	如果是常规文件，返回字节单位的文件大小
uid	路径所属者的 uid
wgrp	如果设置所属组可写权限返回 true
woth	如果设置其他人可写权限返回 true
wusr	如果设置用户可写权限返回 true
xgrp	如果设置所属组可执行返回 true
xoth	如果设置其他人可执行返回 true
xusr	如果设置用户可执行返回 true

在执行前检查你的 playbook

ansible-playbook 命令支持一些参数来让你在执行 playbook 之前对它进行完整的检查。

语法检查

如例 14-3 所示的 --syntax-check 参数将会检查你的 playbook 语法是否有效，但是它并不会执行 playbook。

例 14-3 语法检查

```
$ ansible-playbook --syntax-check playbook.yml
```

列出主机

如例 14-4 所示的 --list-hosts 参数将会输出所有将会执行相应 playbook 的主机，但是它并不会执行 playbook。

例 14-4 列出主机

```
$ ansible-playbook --list-hosts playbook.yml
```



有时候你会得到如下的严重错误信息：

```
ERROR: provided hosts list is empty
```

在你的 inventory 中必须至少明确指定一台主机，否则你就会得到这个错误，即使你的 playbook 是对本地主机执行的也会如此。如果你的 inventory 起初是空的（也许是因为你使用动态 inventory 脚本，但是还没有上线任何主机），

你可以通过向你的 inventory 中显式地添加如下一行来绕过这个问题：

```
localhost ansible_connection=local
```

列出 task

如例 14-5 所示的 `--list-tasks` 参数将会输出你的 playbook 要执行的所有 task。它并不会真正执行 playbook。

例14-5 列出task

```
$ ansible-playbook --list-tasks playbook.yml
```

回忆一下，我们曾在例 6-1 中使用这个参数列出我们第一个 Mezzanine playbook 中的 task。

检测模式

`-C` 和 `--check` 参数会使得 Ansible 运行于检测模式（有时被称作 *dry-run*）。检测模式会告诉你 playbook 中的每个任务是否会修改主机的状态，但是并不对主机执行任何实际的修改。

```
$ ansible-playbook -C playbook.yml  
$ ansible-playbook --check playbook.yml
```

使用检测模式可能有一个小挑战：playbook 中可能会有些内容是需要在前面某些部分确定执行后才会成功执行的。使用检测模式运行例 6-27 的 playbook 将会得到例 14-6 所示的错误。这是因为有 task 依赖于之前在主机上安装 Git 程序的 task。

例14-6 检测模式在运行正确的playbook时报错

```
PLAY [Deploy mezzanine] ****  
GATHERING FACTS ****  
ok: [web]  
  
TASK: [install apt packages] ****  
changed: [web] => (item=git,libjpeg-dev,libpq-dev,memcached,nginx,postgresql,  
python-dev,python-pip,python-psycopg2, python-setuptools, python-virtualenv,  
supervisor)  
TASK: [check out the repository on the host] ****  
failed: [web] => {"failed": true}  
msg: Failed to find required executable git  
  
FATAL: all hosts have already failed -- aborting
```

关于模块如何实现检测模式的详细内容可以查阅第 10 章。

diff (显示文件差异)

-D 和 --diff 参数将会为任何变更远程主机状态的文件输出差异信息。将它与 --check 结合起来非常好用。将它们结合起来使用会展示正常运行情况下 Ansible 会如何修改文件。

```
$ ansible-playbook -D --check playbook.yml  
$ ansible-playbook --diff --check playbook.yml
```

如果 Ansible 会修改任何文件（例如，使用类似 copy、template 以及 lineinfile 这样的模块），那么它将会像下面这样使用 .diff 格式展示这些变化：

```
TASK: [set the gunicorn config file] ****  
--- before: /home/vagrant/mezzanine-example/project/gunicorn.conf.py  
+++ after: /Users/lorinhochstein/dev/ansiblebook/ch06/playbooks/templates/  
gunicorn.conf.py.j2  
@@ -1,7 +1,7 @@  
 from __future__ import unicode_literals  
 import multiprocessing  
  
 bind = "127.0.0.1:8000"  
 workers = multiprocessing.cpu_count() * 2 + 1  
-loglevel = "error"  
+loglevel = "warning"  
 proc_name = "mezzanine-example"
```

限制特定的 task 运行

有时候，你并不希望 Ansible 运行你 playbook 中的每一个 task，特别是在你第一次编写和调试某个 playbook 的时候。Ansible 提供了一些命令行选项来让你控制哪个 task 运行。

step

如例 14-7 所示的 --step 参数将会让 Ansible 在执行每一个 task 之前都提示你，就像下面这样：

```
Perform task: install packages (y/n/c):
```

你可以选择执行这个 task (y)，跳过它 (n)，或者告诉 Ansible 继续执行剩下的 playbook 并且不再提示你 (c)。

例 14-7 step

```
$ ansible-playbook --step playbook.yml
```

270

start-at-task

如例 14-8 所示的 `--start-at-task taskname` 参数告诉 Ansible 从指定的 task 开始运行 playbook，而不是从头开始运行。如果你的 playbook 因为某一个 task 中有 bug 而失败了，在你修复了这个 bug 后你希望从这个你修复的 task 开始再次运行 playbook 的时候，使用这个参数会非常便利。

例14-8 start-at-task

```
$ ansible-playbook --start-at-task="install packages" playbook.yml
```

tags

Ansible 允许你对一个 task 或者一个 play 添加一个或者多个 tags。例如，这里有个 play 具有名为 `foo` 的 tags，另外还有一个 task 具有名为 `bar` 和 `quux` 的 tags：

```
- hosts: myservers
  tags:
    - foo
  tasks:
    - name: install editors
      apt: name={{ item }}
      with_items:
        - vim
        - emacs
        - nano

    - name: run arbitrary command
      command: /opt/myprog
      tags:
        - bar
        - quux
```

使用 `-t tagnames` 或者 `--tags tagnames` 参数告诉 Ansible 仅运行具有特定 tags 的 task 或者 play。使用 `--skip-tags tagnames` 参数告诉 Ansible 跳过具有特定 tags 的 task 或者 play。参见例 14-9。

例14-9 运行或者跳过tags

```
$ ansible-playbook -t foo,bar playbook.yml
$ ansible-playbook --tags=foo,bar playbook.yml
$ ansible-playbook --skip-tags=baz,quux playbook.yml
```

本章的结束意味着我们的旅途也告一段落了。然而你与 Ansible 的旅途其实才刚刚开始。我希望你像我一样享受使用 Ansible 工作的过程，并且在下一次遇到对自动化工具具有明确需求的同事的时候，你能向他们演示 Ansible 如何让他们的生活美好起来。

SSH

因为 Ansible 使用 SSH 作为它的传输机制。你需要了解一些 SSH 的特性以便能够在使用 Ansible 的时候利用这些特性。

原生 SSH

默认情况下，Ansible 使用你操作系统上安装的原生 SSH 客户端。这意味着 Ansible 可以利用所有典型的 SSH 特性，包括 Kerberos 和 jump hosts。如果你在 `~/.ssh/config` 文件中有 SSH 设置自定义配置，Ansible 将会遵循这些设置。

SSH agent

有一个非常便利的程序叫作 `ssh-agent`。它可以简化 SSH 私钥相关操作。

当 `ssh-agent` 在你的机器上运行的时候，你可以使用 `ssh-add` 命令来添加私钥。

```
$ ssh-add /path/to/keyfile.pem
```



必须设置 `SSH_AUTH_SOCK` 环境变量，否则 `ssh-add` 命令将不能与 `ssh-agent` 通信。详见 280 页中“启动 `ssh-agent`”。

你可以在运行 `ssh-add` 程序时使用 `-L` 参数来查看哪些密钥已经被添加到 `agent` 中，如例 A-1 所示。这个范例中已经添加了两个密钥到 `agent` 中。

274 例A-1 列出添加到agent中的密钥

```
$ ssh-add -L
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDWAfog5tz4W9bPVbPDlNC8HWMfhjTgK0hpSZYI+clc
e3/pz5viqsHDQIjzSImoVzIOTV0t0IfE8qMkqEYk7igESccCy0zN9VnD6EfYVKEx1C+xqKLTZTEVuQn
d+4qyo222EAVkHm6bAhgyoA9nt9Um9WF00045yHZL2Do9ZKXTS4x0qeGF5vv7SiuKcsLj0RPcWcYqC
fYdrdUdRD9dFq7zFKmpCPJqNwDQDrXbgaT0e+H6cu2f4RrJLp88WY8voB3zJ7avv68e0gah82dovSgw
hcsZp4SycZSTy+WqZQhzLogaifvtdgdzaooxNtsm+qRvQJyHkwdoXR6nJgt /Users/lorinhochste
in/.ssh/id_rsa
ssh-rsa AAAAB3NzaC1yc2EAAAQEA6NF8iallvQVp22WDkTkrtvp9eWW6A8YVr+kz4TjGYe7
gHzIw+nInltGEFHxD8+v1I2YJ6oXevct1YeS0o9HzyN1Q9qgCgzUFtd0KLv6IedplqoPkcmF0aYet2P
kEdo3M1TBckFXPITAMzF8dJSIFo9d8Hfd0V0IAdx407PtixWKn5y2hMNG0zQPyUecp4pzC6kivAIhyf
HilFR61RGL+GPXQ2MWZWFYbAGjyiYJnAmCP3NOTd0jMZEEnDkbUvxhMmBYsdETk1rRgm+R4L0zFUGaHq
HDFIPKcF96hrucXzcWyLbIbEgE980HlnVYCzRdK8jlqm8tehUc9c9WhQ== insecure_private_key
```

当你尝试与远程主机创建连接的时候，如果 ssh-agent 程序正在运行，那么 SSH 客户端将会尝试使用存储在 ssh-agent 中的密钥与主机进行认证。

使用 SSH agent 有如下几个优点。

- SSH agent 让加密的 SSH 私钥的管理变得更容易。如果你使用一个加密的 SSH 私钥，那么这个私钥文件使用密码来进行保护。当你使用这个私钥与一台主机建立 SSH 连接的时候，你将被提示输入密码。使用加密的私钥的情况下，即使有人访问你的 SSH 私钥，只要他们没有同时拿到密码也是不能使用这个私钥的。如果你使用加密的 SSH 私钥并且没有使用 SSH agent，那么你将不得不在每次使用这个私钥的时候都输入加密密码。但如果你使用 SSH agent，那么仅在向 agent 添加私钥的时候需要输入私钥密码。
- 如果你正在使用 Ansible 管理的主机使用不同的 SSH 密钥，使用 SSH agent 将会简化你的 Ansible 配置文件。你不再需要像我们在例 1-1 中那样在你的主机上明确指定 `ansible_ssh_private_key_file`。
- 如果你需要从你的远程主机到另一台主机建立 SSH 连接（例如，通过 SSH 复制一个私有 Git 仓库），你可以利用 `agent forwarding`。使用 agent forwarding 时，你不需要将 SSH 私钥复制到远程主机上。我们将在后面解释 agent forwarding。

启动 ssh-agent

如何启动 SSH agent 会根据所运行的操作系统而变化。

275 Mac OS X

Mac OS X 已经预先配置好运行 ssh-agent，所以不需要额外的工作。

Linux

如果你的程序运行在 Linux 上，那么你将需要自己启动 ssh-agent 并确保环境变量已经正确设置好。如果你直接调用 ssh-agent，它将会输出你需要设置的环境变量。就像这样：

```
$ ssh-agent  
SSH_AUTH_SOCK=/tmp/ssh-YI7PBG1k0teo/agent.2547; export SSH_AUTH_SOCK;  
SSH_AGENT_PID=2548; export SSH_AGENT_PID;  
echo Agent pid 2548;
```

你可以像这样通过调用 ssh-agent 来自动输出这些环境变量：

```
$ eval $(ssh-agent)
```

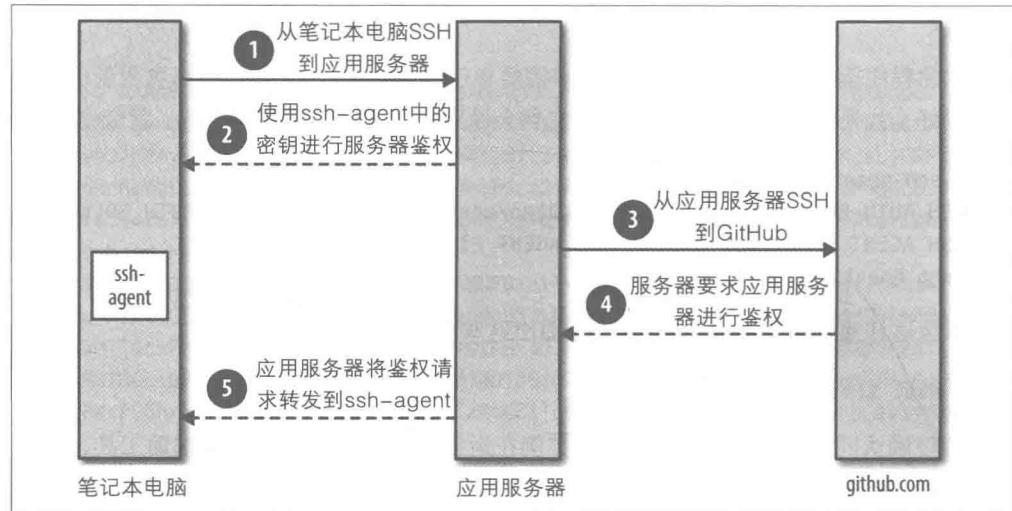
你还希望确认同时只有一个 ssh-agent 实例在运行。在 Linux 上有很多辅助工具，例如 *Keychain* 和 *Gnome Keyring*，可以帮你管理 ssh-agent 的启动，或者你可以通过修改你的 *.profile* 文件来确保 ssh-agent 在每个登录 Shell 只启动一次。为 ssh-agent 配置你的账户超出了本书的范围，所以我建议你查阅你的 Linux 发行版相关的文档来获取关于如何配置这些内容的详细信息。

agent forwarding

如果通过 SSH 来复制一个 Git 仓库，你将需要使用被你 Git 服务器认可的 SSH 私钥。我喜欢尽量避免复制 SSH 私钥到我的主机上，主要是为了限制在某台主机被攻破时的损害。

避免将 SSH 私钥到处复制的办法之一就是在你的本地主机上使用 ssh-agent 程序并打开 agent forwarding 功能。如果你在笔记本电脑中 SSH 到主机 A，并且你已经打开了 agent forwarding 功能，那么 agent forwarding 允许你使用存在你笔记本电脑中的私钥通过 SSH 从主机 A 连接到主机 B。

图 A-1 展示了 agent forwarding 的一个实践范例。比如说你想要使用 SSH 从 GitHub 上 check out 一个私有仓库。你已经在笔记本电脑上启动 ssh-agent，并且也已经使用 ssh-add 命令添加了你的私钥。



图A-1 agent forwarding实践范例

如果你手动 SSH 到应用服务器，你可以在调用 `ssh` 命令时添加 `-A` 参数，即可启用 agent forwarding：

```
$ ssh -A myuser@myappserver.example.com
```

在应用服务器上，你使用 SSH URL 来 check out 一个 Git 仓库：

```
$ git clone git@github.com:lorin/mezzanine-example.git
```

Git 将会通过 SSH 连接到 GitHub。GitHub 的 SSH 服务器将会尝试对应用服务器上的 SSH 客户端进行鉴权。应用服务器并不知道你的私钥。但是，因为你启用了 agent forwarding，应用服务器上的 SSH 客户端将会连回你笔记本电脑上的 `ssh-agent` 并由 `ssh-agent` 来处理这个鉴权。

在将 Ansible 与 agent forwarding 同时使用的时候，有几个问题你需要时刻牢记于心。

首先，你需要 Ansible 连接到远程主机的时候告诉它启用 agent forwarding，因为 Ansible 默认并不启用 agent forwarding。

你可以通过添加如下内容到控制主机的 `~/.ssh/config` 文件来对所有你 SSH 到的节点添加 agent forwarding：

```
Host *
  ForwardAgent yes
```

或者，如果你只希望对指定服务器启用 agent forwarding，可以添加如下内容：

```
Host appserver.example.com
  ForwardAgent yes
```

相对地，如果你只想要对 Ansible 启用 agent forwarding，那么你可以通过向 ssh_ 277 connection 区段的 ssh_args 参数添加它来编辑 ansible.cfg 文件：

```
[ssh_connection]
  ssh_args = -o ControlMaster=auto -o ControlPersist=60s -o ForwardAgent=yes
```

这里，我使用了更冗长的 -o ForwardAgent=yes 参数，而不是较短的 -A 参数，但是它们的作用是一样的。

ControlMaster 和 ControlPersist 是名为 *SSH Multiplexing* 的性能优化技术所需要的设置项。我默认打开它们，但是如果你覆盖了 ssh_args 变量，那么你需要显式地指定它们，否则你将会停用这个性能提升。我们在第 9 章中已经讨论过 SSH Multiplexing。

sudo 与 agent forwarding

当你启用 agent forwarding 的时候，远程主机会设置 SSH_AUTH_SOCK 环境变量，并且这个环境变量中包含一个指向 UNIX 域套接字的路径（例如，/tmp/ssh-FShDVu5924/agent.5924）。但是，如果你使用了 sudo，那么 SSH_AUTH_SOCK 环境变量不会继承，除非明确地配置 sudo 允许这个行为。

为了允许 SSH_AUTH_SOCK 变量在 sudo 到 root 用户的时候能够继承，我们可以将下面这行添加到 /etc/sudoers 文件或者（基于 Debian 的发行版，比如 Ubuntu）在 /etc/sudoers.d 目录下的一个独立文件中。

```
Defaults:root env_keep+=SSH_AUTH_SOCK
```

让我们就将它保存在 99-keep-ssh-auth-env 文件中吧，当然还要将这个文件放在我们本地主机的相应文件目录下。

验证文件

copy 和 template 模块支持 validate 语句。这个语句让你指定一个程序用于执行 Ansible 生成的文件。使用 %s 作为文件名的占位符。例如：

```
validate: visudo -cf %s
```

当 validate 语句存在，Ansible 会先将文件复制到临时目录然后运行指定的验证程序。如果验证程序返回成功 (0)，那么 Ansible 将会把文件从临时位置复制到恰当的目标位置。如果验证程序返回了非零返回码，Ansible 将会返回如下所示错误：

```

failed: [myhost] => {"checksum": "ac32f572f0a670c3579ac2864cc3069ee8a19
588","failed": true}
msg: failed to validate: rc:1 error:

FATAL: all hosts have already failed -- aborting

```

因为糟糕的 sudoers 文件会阻止我们作为 root 访问它，所以总是使用 visudo 程序来验证 sudoers 文件是一个好主意。关于无效的 sudoers 文件的警示故事，可以查阅 Ansible 项目代码贡献者 Jan-Piet Mens 的 blog：“Don't try this at office : /etc/sudoers” (<http://jpmens.net/2013/02/06/don-t-try-this-at-the-office-etc-sudoers/>)。

```

- name: copy the sudoers file so we can do agent forwarding
  copy:
    src: files/99-keep-ssh-auth-sock-env
    dest: /etc/sudoers.d/99-keep-ssh-auth-sock-env
    owner: root group=root mode=0440
    validate: visudo -cf %s

```

遗憾的是，目前并不支持作为一个非 root 用户 sudo 并使用 agent forwarding。例如，比如说我想要从 ubuntu 用户 sudo 为 deploy 用户。问题是 SSH_AUTH_SOCK 所指向的 UNIX 域套接字的所属者是 ubuntu 用户并且对于 deploy 用户不可读且不可写。

为了绕过这个问题，你可以每次调用 Git 模块的时候都使用 root 并且使用 file 模块变更文件的权限，如例 A-2 所示。

例A-2 使用root复制并变更权限

```

- name: verify the config is valid sudoers file
  local_action: command visudo -cf files/99-keep-ssh-auth-sock-env
  sudo: True

- name: copy the sudoers file so we can do agent forwarding
  copy:
    src=files/99-keep-ssh-auth-sock-env
    dest=/etc/sudoers.d/99-keep-ssh-auth-sock-env
    owner=root group=root mode=0440
    validate='visudo -cf %s'
  sudo: True

- name: check out my private git repository
  git: repo=git@github.com:lorin/mezzanine-example.git dest={{ proj_path }}
  sudo: True

- name: set file ownership
  file:
    path={{ proj_path }} state=directory recurse=yes

```

```
owner={{ user }} group={{ user }}  
sudo: True
```

Host Key

279

每台运行 SSH 服务器端的主机都拥有一个关联的 host key。Host key 就好像指纹一样是主机的唯一标识。Host key 的存在是为了防止中间人攻击。如果你正在通过 SSH 从 GitHub 复制一个 Git 仓库，你并不真的知道是否声称 `github.com` 的服务器是真的 GitHub 的服务器，还是使用 DNS 劫持假扮 `github.com` 的攻击者。Host key 允许你去检测声称是 `github.com` 的服务器是否是真的 `github.com`。这意味着你在尝试连接到主机之前需要先得到 host key（就像是一份签名的副本）。

Ansible 默认会检查 host key，但是你可以像下面这样在 `ansible.cfg` 文件中禁止这个行为：

```
[defaults]  
host_key_checking = False
```

在使用 `git` 模块的 `play` 时也会进行 host key 检查。让我们来回顾一下第 6 章中我们使用 `git` 模块的时候就有一个 `accept_hostkey` 参数：

```
- name: check out the repository on the host  
  git: repo={{ repo_url }} dest={{ proj_path }} accept_hostkey=yes
```

如果 host key 检查在主机上被启用且 Git 服务器的 SSH host key 不是我们已知的该主机的 host key，那么在使用 SSH 协议复制 Git 仓库时候 `git` 模块会暂停操作。

最简单的方法就是使用 `accept_hostkey` 参数告诉 Git 自动接收未知的 host key，这也是我们在例 6-5 中使用的方法。

很多人直接接收 host key，他们并不担心这些类型的中间人攻击。也就是我们之前在 `playbook` 中所做的，在调用 `git` 模块时指定 `accept_hostkey=yes` 作为参数。但是，如果你更有安全意识并且不希望自动接收 host key，那么你可以手动获取与验证 GitHub 的 host key，然后将合法的 host key 添加到全系统的 `/etc/ssh/known_hosts` 文件中或者添加到指定用户的 `~/.ssh/known_hosts` 文件中。

要手动验证 GitHub 的 SSH host key，你将需要使用某种带外通道来获取 Git 服务器的 SSH host key 指纹。如果你使用 GitHub 作为你的 Git 服务器，你可以在 GitHub 网站上找到它的 SSH key 的指纹 (<https://help.github.com/articles/what-are-github-s-ssh-key-fingerprints/>)。

在编写本书的时候，GitHub 的 RSA 指纹是 `16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48`。但是别那么相信我的话，还是去网站验证一下吧。

280 > 接下来，你需要获取完整的 SSH host key。你可以使用 ssh-keyscan 程序来获取名为 `github.com` 的主机的 host key。我喜欢将文件放在 Ansible 将会处理的 `files` 目录下，所以我们这么操作：

```
$ mkdir files  
$ ssh-keyscan github.com > files/known_hosts
```

输出如下所示：

```
github.com ssh-rsa  
AAAAB3NzaC1yc2EAAAQEAq2A7hRGmdnm9tUDb09IDSwBK6TbQa+PXYPCKPy6rbTrTtw7PH  
kccKrpp0yVhp5HdEIcKr6pLlVDBf0LX9QUsyCOV0wzfjIJNLGEYsdllJizHhb2mUjvSAHQqZETY  
P81eFzLQNnPht4EVVUh7VfDESU84KezmD5QlWpXLmvU31/yMf+Se8xhHTvKSCZIFImWwoG6mbUoW  
f9nzbIoaSjB+weqqUUpaaasXVal72J+UX2B+2RPW3RcT0e0zQgqlJL3RKrTJvdsjE3JEAvGq3LG  
HSZXy28G3skua2SmVi/w4yCE6gb0DqnTWlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
```

对于更偏执的人，`ssh-keyscan` 命令支持 `-H` 参数以在 `known_hosts` 文件中隐藏主机名。纵然有人可以访问你的已知 `known_hosts` 文件，但他们也没法知道具体主机名是什么。当使用这个参数的时候，输出会变成如下内容：

```
|1|BI+Z8H3hzbcmTWna9R4orrwrNrg=|wCxJf50pTQ83JFzyXG4aNlxEmzc= ssh-rsa AAAAB3N  
zaC1yc2EAAAQEAq2A7hRGmdnm9tUDb09IDSwBK6TbQa+PXYPCKPy6rbTrTtw7PHkccKrpp  
0yVhp5HdEIcKr6pLlVDBf0LX9QUsyCOV0wzfjIJNLGEYsdllJizHhb2mUjvSAHQqZETYP81eFzL  
QNnPht4EVVUh7VfDESU84KezmD5QlWpXLmvU31/yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzbI  
oSjB+weqqUUpaaasXVal72J+UX2B+2RPW3RcT0e0zQgqlJL3RKrTJvdsjE3JEAvGq3lGHSZXy28  
G3skua2SmVi/w4yCE6gb0DqnTWlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
```

然后你需要验证 `files/known_hosts` 文件中的 host key 与你在 GitHub 上找到的指纹相匹配。你可以使用 `ssh-keygen` 程序来进行这个检查：

```
$ ssh-keygen -lf files/known_hosts
```

输出应该与网站上公布的 RSA 指纹相符，好像这样：

```
2048 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48 github.com (RSA)
```

现在你可以确信你有 Git 服务器的正确的 host key。你可以使用 `copy` 模块将它复制到 `/etc/ssh/known_hosts`。

```
- name: copy system-wide known hosts  
  copy: src=files/known_hosts dest=/etc/ssh/known_hosts owner=root group=root  
        mode=0644
```

或者你可以复制到指定用户的 `~/.ssh/known_hosts` 目录下。例 A-3 展示了如何将控制主机上的 `known_hosts` 文件复制到远程主机上。

例A-3 添加已知主机的host key

```
- name: ensure the ~/.ssh directory exists
  file: path=~/ssh state=directory
- name: copy known hosts file
  copy: src=files/known_hosts dest=~/ssh/known_hosts mode=0600
```

281

即使禁用 key 检查，一个有问题的 host key 也可能产生问题

如果你已经在 Ansible 中通过设置 `ansible.cfg` 文件中的 `host_key_checking` 的值为 `false` 禁用了 host key 检查，并且 Ansible 尝试连接的主机的 host key 与你 `~/ssh/known_hosts` 文件中的相应 key 条目并不匹配，那么 agent forwarding 将不能正常工作。这种情况下尝试复制 Git 仓库将会导致如下所示的错误：

```
TASK: [check out the repository on the host]*****
failed: [web] => {"cmd": "/usr/bin/git ls-remote git@github.com:lorin/mezzanine-example.git -h refs/heads/HEAD", "failed": true, "rc": 128}
stderr: Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

msg: Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

FATAL: all hosts have already failed -- aborting
```

如果你正在使用 Vagrant，那么这种情况就可能发生。当你销毁一个 Vagrant 虚拟机，然后又创建了一个新的 Vagrant 虚拟机的时候，因为每次你创建的新 Vagrant 虚拟机的 host key 都会变化，所以会导致这个问题。你可以通过下面的命令来检查 agent forwarding 是否正常工作：

```
$ ansible web -a "ssh-add -l"
```

如果它正常工作，你将会得到如下的输出：

```
web | success | rc=0 >>
2048 e5:ec:48:d3:ec:5e:67:b0:22:32:6e:ab:dd:91:f9:cf /Users/lorinhochstein/.ssh/id_rsa (RSA)
```

如果它不工作，你将会得到如下输出：

```
web | FAILED | rc=2 >>
Could not open a connection to your authentication agent.
```

如果你遇到了这个问题，那么就需要删除你的`~/.ssh/known_hosts`文件中相对应的条目。

还有一点需要注意，由于SSH Multiplexing，Ansible会持续维护一个连接到主机的开放的SSH 60s，所以你需要等这个连接超时，否则就无法观察到修改`known_hosts`文件的效果。

282> 很明显，验证一个SSH的host key比盲目地接受它要增加很多工作。在绝大多数情况下，我们都需要在安全与便利之间做权衡。

默认设置

Ansible 定义了很多设置项。你可以通过修改 Ansible 配置文件或者环境变量来覆盖这些设置项的默认值。

配置文件被分为以下几个部分，详细内容如表 B-1 到表 B-4 所示。

- defaults
- ssh_connection
- paramiko
- accelerate

表B-1 defaults部分

配置名称	环境变量	默认值
hostfile	ANSIBLE_HOSTS	/etc/ansible/hosts
library	ANSIBLE_LIBRARY	(none)
roles_path	ANSIBLE_ROLES_PATH	/etc/ansible/roles
remote_tmp	ANSIBLE_REMOTE_TEMP	\$HOME/.ansible/tmp
module_name	(none)	command
pattern	(none)	*
forks	ANSIBLE_FORKS	5
module_args	ANSIBLE_MODULE_ARGS	(空字符串)
module_lang	ANSIBLE_MODULE_LANG	en_US.UTF-8
timeout	ANSIBLE_TIMEOUT	10

续表

配置名称	环境变量	默认值
poll_interval	ANSIBLE_POLL_INTERVAL	15
remote_user	ANSIBLE_REMOTE_USER	current user
ask_pass	ANSIBLE_ASK_PASS	false
private_key_file	ANSIBLE_PRIVATE_KEY_FILE	(none)
sudo_user	ANSIBLE_SUDO_USER	root
ask_sudo_pass	ANSIBLE_ASK_SUDO_PASS	false
remote_port	ANSIBLE_REMOTE_PORT	(none)
ask_vault_pass	ANSIBLE_ASK_VAULT_PASS	false
vault_password_file	ANSIBLE_VAULT_PASSWORD_FILE	(none)
ansible_managed	(none)	Ansible managed: {file} modified on %Y-%m-%d %H:%M:%S by {uid} on {host}
syslog_facility	ANSIBLE_SYSLOGFacility	LOG_USER
keep_remote_files	ANSIBLE_KEEP_REMOTE_FILES	true
sudo	ANSIBLE_SUDO	false
sudo_exe	ANSIBLE_SUDO_EXE	sudo
sudo_flags	ANSIBLE_SUDO_FLAGS	-H
hash_behaviour	ANSIBLE_HASH_BEHAVIOUR	replace
jinja2_extensions	ANSIBLE_JINJA2_EXTENSIONS	(none)
su_exe	ANSIBLE_SU_EXE	su
su	ANSIBLE_SU	false
su_flag	ANSIBLE_SU_FLAGS	(空字符串)
su_user	ANSIBLE_SU_USER	root
ask_su_pass	ANSIBLE_ASK_SU_PASS	false
gathering	ANSIBLE_GATHERING	implicit

续表

配置名称	环境变量	默认值
action_plugins	ANSIBLE_ACTION_PLUGINS	/usr/share/ansible_plugins/action_plugins
cache_plugins	ANSIBLE_CACHE_PLUGINS	/usr/share/ansible_plugins/cache_plugins
callback_plugins	ANSIBLE_CALLBACK_PLUGINS	/usr/share/ansible_plugins/callback_plugins
connection_plugins	ANSIBLE_CONNECTION_PLUGINS	/usr/share/ansible_plugins/connection_plugins
lookup_plugins	ANSIBLE_LOOKUP_PLUGINS	/usr/share/ansible_plugins/lookup_plugins
vars_plugins	ANSIBLE_VARS_PLUGINS	/usr/share/ansible_plugins/vars_plugins
filter_plugins	ANSIBLE_FILTER_PLUGINS	/usr/share/ansible_plugins/filter_plugins
log_path	ANSIBLE_LOG_PATH	(空字符串)
fact_caching	ANSIBLE_CACHE_PLUGIN	memory
fact_caching_connection	ANSIBLE_CACHE_PLUGIN_CONNECTION	(none)
fact_caching_prefix	ANSIBLE_CACHE_PLUGIN_PREFIX	ansible_facts
fact_caching_timeout	ANSIBLE_CACHE_PLUGIN_TIMEOUT	86400(秒)
force_color	ANSIBLE_FORCE_COLOR	(none)
nocolor	ANSIBLE_NOCOLOR	(none)
nocows	ANSIBLE_NOCOWS	(none)
display_skipped_hosts	DISPLAY_SKIPPED_HOSTS	true
error_on_undefined_vars	ANSIBLE_ERROR_ON_UNDEFINED_VARS	true
host_key_checking	ANSIBLE_HOST_KEY_CHECKING	true
system_warnings	ANSIBLE_SYSTEM_WARNINGS	true
deprecation_warnings	ANSIBLE_DEPRECATED_WARNINGS	true
callable_whitelist	ANSIBLE_CALLABLE_WHITELIST	(空列表)

配置名称	环境变量	默认值
command_warnings	ANSIBLE_COMMAND_WARNINGS	false
bin_ansible_callbacks	ANSIBLE_LOAD_CALLBACK_PLUGINS	false



如果你是使用包管理工具来安装 Ansible 的，那么插件的默认路径可能会与这里列出的不太一样。这是因为下游包管理工具可能会修改 Ansible 相关文件由上游 Ansible 项目指定的默认位置。

表B-2 ssh_connection部分

配置名称	环境变量	默认值
ssh_args	ANSIBLE_SSH_ARGS	-o ControlMaster=auto -o ControlPersist=60s -o ControlPath="\$ANSIBLE_SSH_CONTROL_PATH"
control_path	ANSIBLE_SSH_CONTROL_PATH	%(directory)s/ansible-ssh-%h-%p-%r
pipelining	ANSIBLE_SSH_PIPELINING	false
scp_if_ssh	ANSIBLE SCP IF SSH	false

Ansible 将 `_control_path` 配置文件中的 `%(directory)s` 替换为 `$HOME/.ansible/cp`。

表B-3 paramiko部分

配置名称	环境变量	默认值
record_host_keys	ANSIBLE_PARAMIKO_RECORD_HOST_KEYS	true
pty	ANSIBLE_PARAMIKO_PTY	true

表B-4 accelerate部分

配置名称	环境变量	默认值
accelerate_keys_dir	ACCELERATE_KEYS_DIR	\~/fireball.keys
accelerate_keys_dir_perms	ACCELERATE_KEYS_DIR_PERMS	700
accelerate_keys_file_perms	ACCELERATE_KEYS_FILE_PERMS	600
accelerate_multi_key	ACCELERATE_MULTI_KEY	false

为EC2证书使用IAM role

如果你要在 VPC 中使用 Ansible，你可以利用 Amazon 的访问控制和密码管理（IAM）role 来实现传递你的 EC2 证书到实例，甚至不需要设置任何环境变量。

Amazon 的 IAM role 允许你定义用户和群组以及控制用户和群组许可在 EC2 上进行的操作（例如，获取你运行中的实例的信息、创建实例、创建镜像）。你也可以分配 IAM role 给正在运行的实例，所以实际上你可以这样说：“这个实例可以启动其他实例。”

当你使用支持 IAM role 的客户端程序向 EC2 发起请求的时候，并且一个实例已经被 IAM role 授予权限，客户端将会从 EC2 实例元数据服务 (<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html>) 获取证书并使用它们向 EC2 服务终端发起请求。

你可以通过 Amazon Web Service (AWS) 管理控制台创建 IAM role，或者在命令行使用 AWS 命令行接口工具，或者是 AWS CLI (<http://aws.amazon.com/cn/cli/>)。

AWS 管理控制台

下面列出了使用 AWS 管理控制台是如何创建一个 IAM role “Power User Access”的。“Power User Access” 意味着它被允许使用 AWS 做几乎任何事情，除了修改 IAM 用户和群组。

1. 登录 AWS 管理控制台 (<https://console.aws.amazon.com>)。
2. 单击“Identity & Access Management”。
3. 单击左侧的“Roles”。

4. 单击“Create New Role”按钮。
5. 给定你的 role 名字。我喜欢使用“ansible”作为将要运行的 Ansible 实例的 role 名字。
6. 在“AWS Service Roles”中选择“Amazon EC2”。
7. 选择“Power User Access”。然后，Web 界面上将会向你展示策略名和策略文档，并会给你随意编辑它们的机会。默认策略名将会类似于 *PowerUserAccess-ansible-201411182152*，而策略文档则为类似例 C-1 的 JSON 字符串。
8. 单击“Next Step”。
9. 单击“Create Role”。

例C-1 IAM Power User策略文档

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "NotAction": "iam:*",
      "Resource": "*"
    }
  ]
}
```

当你通过 Web 界面创建 role 的时候，AWS 还会自动创建一个使用与 role 相同名字（例如，“ansible”）的实例配置并与和实例配置同名的 role 相关联。当你使用 ec2 模块创建一个实例的时候，如果你将实例配置的名字作为 `instance_profile_name` 参数，那么创建的实例将会拥有对应的 role 的权限。

命令行

你也可以使用 AWS CLI 工具创建 role 和实例配置，但是这需要更多工作要做。你需要：

1. 创建一个 role 并指定信任策略。信任策略描述了可以承担 role 的实体和 role 访问条件。
2. 创建一个描述这个角色允许做哪些事情的策略。在我们的例子中，我们希望创建一个与 power user 相同权限的策略。也就是说这个 role 可以执行任何 AWS 有关的操作，除了操作 IAM role 和群组。
3. 创建一个实例配置。

4. 为实例配置关联这个 role。

你需要首先创建两个 JSON 格式的 IAM 策略文件。信任策略如例 C-2 所示。这个信任策略与你通过 Web 界面创建 role 时候自动生成的一样。

role 策略描述了 role 允许做哪些操作，如例 C-3 所示。

例C-2 trust-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

例C-3 power-user.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "NotAction": "iam:*",
      "Resource": "*"
    }
  ]
}
```

例 C-4 展示了一旦你创建了如例 C-2 和例 C-3 所示的文件，如何在命令行创建实例配置，

例C-4 在命令行创建实例配置

```
# 确认 trust-policy.json 和 power-user.json 在当前目录下，或者
# 将 file:// 参数的值替换为全路径
```

```
$ aws iam create-role --role-name ansible --assume-role-policy-document \
  file://trust-policy.json
$ aws iam put-role-policy --role-name ansible --policy-name \
  PowerUserAccess-ansible-20141118 --policy-document file://power-user.json
```

292 > \$ aws iam create-instance-profile --instance-profile-name ansible
\$ aws iam add-role-to-instance-profile --instance-profile-name ansible \
--role-name ansible

如你所见，通过 Web 界面操作要简单得多。但是如果你想要自动化这些操作，那么你可以使用命令行来替代 Web 界面。更多细节请查阅 AWS 访问控制与密码管理用户文档 (<http://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html>)。

一旦你创建了实例配置，你就可以启用这个实例配置来启动 EC2 实例。你可以使用 ec2 模块并使用 instance_profile_name 参数来实现这个操作：

```
- name: launch an instance with iam role  
  ec2:  
    instance_profile_name: ansible  
    # 其他参数省略
```

如果你 SSH 到这个实例，你可以查询 EC2 元数据服务来确认这个实例是与 Ansible 配置相关联的。输出将会类似如下所示内容：

```
$ curl http://169.254.169.254/latest/meta-data/iam/info  
{  
  "Code" : "Success",  
  "LastUpdated" : "2014-11-17T02:44:03Z",  
  "InstanceProfileArn" : "arn:aws:iam::549704298184:instance-profile/ansible",  
  "InstanceProfileId" : "AIPAINM7F44YGDNIBHPYC"  
}
```

你也可以直接检查证书，尽管实际上并不需要你做什么。当 Ansible ec2 模块或者动态 inventory 脚本执行的时候，Boto 库将会自动获取证书：

```
$ curl http://169.254.169.254/latest/meta-data/iam/security-credentials/  
ansible  
{  
  "Code" : "Success",  
  "LastUpdated" : "2015-02-09T21:45:20Z",  
  "Type" : "AWS-HMAC",  
  "AccessKeyId" : "ASIAIYXCUETJPY42AC2Q",  
  "SecretAccessKey": "0Rp9gldiymIKH9+rFtWEx8BjGRteNTQSRnLnlmWq",  
  "Token" : "AQoDYXdzEGca4AMPC5W69pvtENpXjw79oH9...",  
  "Expiration" : "2015-02-10T04:10:36Z"  
}
```

这些证书是临时的，并且 Amazon 将会自动为你替换它们。

现在你可以使用这个实例作为你的控制主机而不需要通过环境变量来指定你的证书。Ansible 的 ec2 模块会从元数据服务自动获取证书。

术语

Alias (别名)

当 inventory 中主机的名字与主机实际的 hostname 不相符的时候，这个名字叫作 Alias。

AMI

Amazon Machine Image, Amazon Elastic Compute Cloud, 又被称作 EC2 中的虚拟机镜像。

Ansible, INC.

管理 Ansible 项目的公司。

Ansible Galaxy

由社区贡献的 Ansible role 的仓库 (<https://galaxy.ansible.com>)。

Ansible Tower

Ansible 公司售卖的商业软件。它包含一个基于 Web 的 Dashboard 和用于控制 Ansible 的 REST 接口。

Check Mode (检测模式)

运行 playbook 的一个可选模式。当 Ansible 执行 playbook 并且检测模式启用的时候，将不会对远程主机进行任何变更。相对地，它仅会报告每个 task 是否会变更主机的状态。有时候被称作 “dry run” 模式。

CIDR

Classless Inter-Domain Routing，一种 IP 地址范围的标记法。在定义 Amazon EC2 安全

组的时候会用到 CIDR。

Configuration Management (配置管理)

一个用于确保服务器处于正常工作状态的进程。

这个正常的状态我们指的是服务器应用的配置文件具有正确的内容、特定的文件正常存在、正确的服务在运行、期望的用户名存在、权限被正确设置等。

Convergence (收敛性)

配置管理系统的一种属性，具有这种属性的系统将会对一台服务器执行多次以让服务器接近期望的状态，每次执行都会让服务器更接近期望的状态。收敛性与 CFEngine 配置管理系统最相关。而 Ansible 在一次执行后就会将服务器置为期望状态，所以并不具备收敛性。

Complex arguments (复杂参数)

类型为列表或者字典的传递给模块的参数。

Container (容器)

一种服务器虚拟化的形式，这种形式的虚拟化在操作系统层实现，这种情况下虚拟机实例共享同一个内核。Docker 是目前最火爆的容器技术。

Control machine (控制主机)

你运行 Ansible 并用于控制远程主机的那台计算机。

Control socket (控制套接字)

当 SSH Multiplexing 启用的时候，SSH 客户端用于连接到远程主机的 UNIX 域套接字。

ControlPersist

与 SSH Multiplexing 同义。

Declarative (声明式)

一种编程语言的类型，程序员用其描述期望的输出，而不是如何计算输出的过程。Ansible 的 playbook 是声明式的。SQL 是另一个声明式语言的例子。与之对应的是面向过程的语言，例如 Java 和 Python。

Deployment (部署)

将你的软件发布到线上系统的过程。

DevOps

20世纪10年代中期逐渐流行起来的IT流行术语。

Dry run

详见 *Check mode*。

DSL

Domain-Specific Language (特定领域语言)。在使用 DSL 的系统中，用户与系统的交互是通过使用 DSL 编写文本文件，然后通过系统执行这些文件来实现的。DSL 没有通用编程语言那么强大，但是（如果经过良好的设计）会比通用编程语言更易于读写。Ansible 使用 YAML 语法来实现 DSL。

Dynamic inventory

在 playbook 执行时向 Ansible 提供主机和群组信息的来源。

EBS

Elastic block store。在 Amazon EC2，EBS 指的是可以加载到实例的持久化存储。

Fact

一种包含指定主机信息的变量。

Facter

Puppet 用来获取主机相关信息的工具。如果 Facter 被安装，Ansible 将在采集关于一台主机的 fact 的时候调用 Facter。

Glob

Glob 是 UNIX Shell 用来进行文件名匹配的一种模式。例如，*.txt 是一个将会匹配所有以 .txt 结尾的文件的 glob。

Group (群组)

有命名的主机集合。

Handler

与 task 类似，除非被通知触发，否则 handler 默认不会执行。这些通知在被配置为与 handler 相关联的 task 出现状态改变时被触发。

Host (主机)

一台被 Ansible 管理的远程服务器。

IAM

访问控制和密码管理（Identity and Access Management），Amazon EC2 的一个允许你管理用户与群组权限的特性。

Idempotent (幂等性)

如果一个操作被执行多次与被执行一次的效果相同，那么这个操作是幂等的。

Instance (实例)

一台虚拟机。这个术语一般被用来指代运行在基础设施即服务类型（IaaS）的云（如 Amazon EC2）中的虚拟机。

Inventory

主机与群组的列表。

Lookups

当 playbook 运行时，在控制主机上执行的用来获得一些 Ansible 需要的配置数据的代码。

Module (模块)

Module 是执行特定任务的 Ansible 脚本。例如包括创建一个用户账号和安装一个软件包，或者启动一个服务。绝大部分 Ansible 模块是幂等的。

Ohai

Chef 用来获取关于主机信息的工具。如果 Ohai 被安装，Ansible 将在采集关于主机的 fact 的时候调用 Ohai。

Orchestration (编配)

在一批服务器上按照定义好的顺序执行一系列任务。通常来说，Orchestration 是执行部

署的必需部分。

Pattern (模式)

用来描述 play 执行的对象主机的 Ansible 语法。

Play

将一系列主机与需要在主机上运行的任务列表相关联。

Playbook

一个 Ansible 脚本。它指定一系列 play 与一批 play 执行的对象主机。

Registered variable (注册变量)

在 task 中使用 register 语句创建的变量。

Role

用于将一系列 task、handlers、文件、模板和变量捆绑在一起的一种 Ansible 机制。

例如，一个 Nginx role 可能包含安装 Nginx 软件包的 task、生成 Nginx 配置文件、复制 TLS 证书文件和启动 Nginx 服务。

SSH Multiplexing

OpenSSH 的 SSH 客户端拥有的一个特性。这个特性可以在向相同机器创建多条 SSH 连接的时候降低 SSH 创建连接的时间。Ansible 默认使用 SSH Multiplexing 来提升性能。

Task

Ansible play 中的工作单元。一个 task 指定一个模块和相关参数，以及一个可选的名字与一些可选的参数。

TLS

Transport Layer Security，一种用于 Web 服务器与浏览器之间安全通信的协议。TLS 取代了一个名为 *Secure Socket Layer* (SSL) 的早期协议。很多人错误地使用 SSL 指代 TLS。

Transport

Ansible 用于连接到远程主机的协议与实现。默认的 transport 是 SSH。

Vault

Ansible 用于加密磁盘上的敏感信息的一种机制。通常用于在版本控制系统中安全地存储机密数据。

Vagrant

一个用于管理虚拟机的工具，专门用于开发者创建可重复的开发环境。

Virtualenv

用于安装 Python 软件包到一个可以被启用或者禁用的环境中的机制。它让用户可以在没有 root 权限的情况下安装 Python 软件包并且不会污染机器上的全局 Python 软件包库。

VPC

Virtual Private Cloud。被 Amazon EC2 使用的术语，用于描述可以用来为 EC2 实例创建的独立网络。

参考文献

- [ansible-aws] Kurniawan, Yan. *Ansible for AWS*. Leanpub, 2015 (forthcoming).
- [cloudsysadmin] Limoncelli, Thomas A.; Hogan, Christina J.; Chalup, Strata R. *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems*. Addison-Wesley Professional, 2014.
- [dataintensive] Kleppmann, Martin. *Designing Data-Intensive Applications*. O'Reilly Media, 2015.
- [nist] Mell, Peter; Grance, Timothy. *The NIST Definition of Cloud Computing*. NIST Special Publication 800-145, 2011.
- [openssh] *OpenSSH/Cookbook/Multiplexing*, Wikibooks, <http://en.wikibooks.org/w/index.php?title=OpenSSH/Cookbook/Multiplexing&oldid=2711287> October 28, 2014.
- [pragprog] Hunt, Andrew; Thomas, David. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
- [tastetest] Jaynes, Matt. *Taste Test: Puppet, Chef, Salt, Ansible*. Publisher, 2014.
- [vagrant] Hashimoto, Mitchell. *Vagrant: Up and Running*. O'Reilly Media, 2013.
- [webops] Shafer, Andrew Clay. *Agile Infrastructure in Web Operations: Keeping the Data on Time*. O'Reilly Media, 2010.

索引[†]

A

A records, 140
abstraction, layers of, 7
accelerate configurations, 286
accelerated mode, 10, 171
accelerate_keys_dir configuration, 286
accelerate_keys_dir_perms configuration, 286
accelerate_keys_file_perms configuration, 286
accelerate_multi_key configuration, 286
action_plugins configuration, 283
add_file_common_args, 184, 186
add_host, 65
agent forwarding
 for Vagrant, 197
 SSH, 275-278
 Sudo and, 277
aliases option, 182
aliases, host, 54
Amazon machine image (AMI), 205
 building, 232-236
 building with ec2_ami module, 232
 building with Packer, 232-236
 getting latest, 220
ami_launch_index, 222
Ansible
 advantages of, 5-8
 default settings, 283-286
 installation, 10
 naming, 2
 prerequisites for, 9
 setting up server for testing, 11-20
 simple use case with, 3
 simplicity/power of, 8
 uses of, 2

Ansible Galaxy, 8, 159
Ansible Tower, 10
Ansible, Inc., 8
ansible-galaxy, 157
ansible-pull, 6
ansible-vault commands, 130
ansible-vault create, 131
ansible-vault decrypt, 131
ansible-vault edit, 131
ansible-vault encrypt, 131
ansible-vault rekey, 131
ansible-vault view, 131
ansible.cfg, 16-20, 214
AnsibleModule helper class, importing, 180
AnsibleModule initializer method, 184-189
 arguments for, 184-187
 check mode (dry run), 189
 invoking external commands, 188
 returning success or failure, 187
ansible_*_interpreter, 49, 50
ansible_connection, 49
ansible_managed configuration, 283
ansible_managed variable, 25
ansible_python_interpreter, 50
ansible_python_interpreter parameter, 49
ansible_shell_type, 50
ansible_shell_type parameter, 49
ansible_ssh_host parameter, 49
ansible_ssh_pass parameter, 49
ansible_ssh_port parameter, 49
ansible_ssh_private_key_file parameter, 49
ansible_ssh_user parameter, 49
ansible_version variable, 79
application server (Gunicorn), 87

[†]：中文版书中切口处“□”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引所列页码为原英文版页码。

application servers, 1
Apt cache, updating, 94
apt module, 34, 92
args run_command argument, 188
arguments
 complex, in tasks, 99-102
 options, for custom modules, 181-183
 parsing, for custom modules, 179
arguments file, custom modules, 175
argument_spec, 184
argument_spec initializer argument, 184
ask_pass configuration, 283
ask_sudo_pass configuration, 283
ask_su_pass configuration, 283
ask_vault_pass configuration, 283
assert module, 265
atime return value, 265
auto-generated groups, EC2, 210
availability zone groups, 210
AWS CLI (command line) tool, 290-292
AWS management console, 289
awscli, 59
Azure, 3, 11

B

Baker, Bill, 55
bare metal servers, 204
basename filter, 134
Bash
 implementing custom modules in, 192
 specifying alternative location for, 193
behavioral inventory parameters, 49-51
 ansible_*_interpreter, 49
 ansible_connection, 49
 ansible_python_interpreter, 49
 ansible_shell_type, 49
 ansible_ssh_host, 49
 ansible_ssh_pass, 49
 ansible_ssh_port, 49
 ansible_ssh_private_key_file, 49
 ansible_ssh_user, 49
Bias, Randy, 55
binary_data run_command argument, 188
bin_ansible_callbacks configuration, 283
Book2Docker, 251
booleans, YAML syntax, 29
Boto (Python library), 207
Brugess, Mark, 7
builders, 234

bypass_checks, 184, 187

C

cache_plugins configuration, 283
caching, EC2 inventory, 210
callable_whitelist configuration, 283
callback_plugins configuration, 283
can_reach, 174
Capistrano, 2
Card, Orson Scott, 2
cattle (numbered hosts), 55
Celery task, 52
certificate container, Docker, 259
certs (Docker image), 249
cert_file variable, 41
CFEngine configuration management system, 7
changed (variable), 177
changed_when, 125-128
chdir parameter, 114
check mode, for debugging, 268
check mode/dry run (AnsibleModule initializer
 method), 189
check_invalid_arguments, 185
check_rc run_command argument, 188
Chef, 2
choices (option), 182
Chrome, 44
cidr_ip parameter, 219
close_fds run_command argument, 188
Cloudscaling, 55
CNAME records, 140
Cobbler, 59
collectstatic command, 107
command line, setting variables on, 81
command module, 18, 19, 72, 114, 173
command-line tool, 19, 75
command_warnings configuration, 283
comments, YAML syntax, 28
complex args, 183
complex playbooks, 121-146
 changed_when and failed_when clauses,
 125-128
 encrypting data with vault, 129-131
 filters, 132-143
 limiting which hosts run, 132
 lookups, 136-143
 looping constructs as lookup plug-ins, 146
 manual fact gathering, 122
 more complicated loops for, 143-146

patterns for specifying hosts, 131
retrieving IP address from host, 128
running task on control machine, 121
running task on machine other than host, 122
running task only once, 124
running tasks on one host at a time, 123
configuration files, for EC2 credentials, 207
configuration management, 2, 5
configuration management databases (CMDBs), 59
connection_plugins configuration, 283
constant-width documentation markup, 191
container linking, 241
container(s)
about, 237
Docker images, 247-250
linking, in Docker, 241
Mezzanine container image in Docker, 242-247
control machine, running tasks on, 121
ControlMaster, 163
ControlPath, 163
ControlPersist, 49, 161-164, 163
control_path connection, 286
convention, 24
convergence, 7, 7
copy module, 34, 277
cowsay program, 27
creating instances, 205
credentials, EC2, 206
csh, 50
csvfile lookup, 139
ctime return value, 265
custom homepage, 25
custom modules, 173-194
accessing parameters, 180
AnsibleModule initializer parameters, 184-189
argument options, 181-183
arguments files for, 175
can_reach as, 174
debugging, 191
documenting, 190
expected outputs, 176
for checking that remote server is reachable, 173
implementing in Bash, 192
implementing in Python, 178-190

importing the AnsibleModule helper class, 180
invoking, 175
learning to write from existing Ansible module source code, 194
output variables expected by Ansible, 177
parsing arguments, 179
proper directory for, 175
Python script, 175
using script modules, 174
cwd run_command argument, 188

D.

daemon, 88
data run_command argument, 188
database
creating in Mezzanine, 102
PostgreSQL, 86
role for deploying, 150-153
roles, 148-153
datastores, 1
debug module, 39, 70, 265
debugging
Ansible playbooks, 263-270
assert module, 265
check mode for, 268
checking playbook before execution, 267-269
custom modules, 191
debug module, 265
diff (show file changes) for, 269
limiting which tasks run, 269
list hosts for, 267
list tasks for, 268
SSH issues, 263
start-at-task taskname flag for, 270
step flag for, 269
syntax check for, 267
tags for, 270
declarative modules, 6
default argument options, 181
default filters, 133
default settings, Ansible, 283-286
default value (module option), 182
defaults, changing, 50
DeHaan, Michael, 2, 7
delegate_to module, 122
dependent roles, 158
deployment, 2

deprecation_warnings configuration, 283
dev return value, 265
development mode, production mode vs., 83-88
dict instance, 222
dictionary keys, 73
dictionary(-ies), YAML syntax, 30
diff (show file changes), 269
Digital Ocean, 3, 11, 204
display_skipped_hosts configuration, 283
Django, 40
 development mode, 102
 inventory with, 52-54
 Mezzanine and, 87, 106
 production, 52
 staging environment, 53
 vagrant environment, 53
django-manage commands, 106
DNS record, 140
dnstxt lookup, 140
Docker, 237
 application life cycle, 239
 building non-Ansible images, 250
 container images other than Ansible, 247-250
 deploying applications in, 251
 deploying Mezzanine in, 240-241
 image creation with Ansible, 242-247
 initializing database, 257
 linking containers in, 241
 Mezzanine container image in, 242-247
 playbook example, 260
 reasons for pairing with Ansible, 238
 retrieving database container IP address and
 mapped port, 252-255
 starting certificate container, 259
 starting Mezzanine container, 258
 starting the database container, 251
 starting the Memcached container, 258
 waiting for database to start up, 256-262
Docker Hub, 239
docker_host variable, 256
documentation
 for modules, 35
 of custom modules, 190
domain name system (DNS), 140
dry run/check mode (AnsibleModule initializer
 method), 189
dynamic inventory, 59-63, 60

add_host with, 66
and VPC, 231
auto-generated groups, EC2, 210
defining EC2 dynamic groups with tags, 211-213
EC2 and, 208-211
interface for script, 60
inventory caching, 210
per-existing scripts, 65
writing a script, 61-63

E

EC2, 3, 11, 59, 100, 203-236
 adding new instance to group, 221
 building AMIs, 232-236
 Classic, 213
 configuration files for credentials, 207
 configuring ansible.cfg for use with, 214
 defining dynamic groups with tags, 211-213
 dynamic inventory, 208-211
 dynamic inventory and VPC, 231
 environment variables for credentials, 206
 getting latest AMI, 220
 IAM roles with AWS CLI tool, 290-292
 IAM roles with AWS management console, 289
 idempotent playbooks for, 225-226
 inventory caching, 210
 key pairs, 216
 launching new instances, 215
 permitted IP addresses, 219
 Python Boto library for, 207
 return type of ec2 module, 222
 security group ports, 220
 security groups, 218
 specifying a VPC, 228-232
 specifying credentials in, 206
 terminology, 205
 using IAM roles for credentials, 289-292
 various modules for, 236
Virtual Private Cloud, 213
 waiting for server to come up, 224
ec2 module, 222
EC2-Classic, 213
EC2-VPC (Virtual Private Cloud), 213
ec2_ami module, 232
enable configuration, 36
encryption
 with vault, 129-131

Enders Game (book), 2
env lookup, 138
environment clause, 108
environment variables
 for EC2 credentials, 206
 setting with environment clause, 108
ERB, 40
error_on_undefined_vars configuration, 283
etcd lookup, 142
exact_count parameter, 222
executable config option, 51
executable documentation, 5
executable run_command argument, 188
execution time, reducing, 161-171
 accelerated mode, 171
 fact caching, 167-170
 Fireball mode, 171
 parallelism, 170
 pipelining, 165
 SSH multiplexing and ControlPersist,
 161-164
execution, checking playbook before, 267-269
exists return value, 265

F

Fabric, 2, 85, 113, 114
fact caching, 167-170
 JSON file backend, 168
 Memcached backend, 170
 Redis backend, 169
fact gathering, 28, 28
 and fact caching, 167-170
 manual, 122
fact(s), 74-78
 local, 77
 returned by modules, 76
 using set_fact to define new variable, 78
 viewing all associated with a server, 75
 viewing subsets, 75
fact_caching configuration, 283
fact_caching_connection configuration, 283
fact_caching_prefix configuration, 283
fact_caching_timeout configuration, 283
failed=true (variable), 177
failed_when, 125-128
file lookup, 137
file module, 34
file paths, filters for, 134
file, start of (YAML syntax), 28

filter parameter, 75
FilterModule class, 135
filters
 basename, 134
 changed, 133
 default, 133
 failed, 133
 file path, 134
 for file paths, 134
 for registered variables, 133
 in complex playbooks, 132-136
 skipped, 133
 success, 133
 writing your own, 134-136
filter_plugins configuration, 283
Fireball mode, 171
fish, 50
force_color configuration, 283
forks configuration, 283
from_port parameter, 219

G

gathering configuration, 283
GATHERING FACTS, 28
gid return value, 265
Git
 and SSH agent forwarding, 275-278
 for checking out Mezzanine project, 96
git module, 279
GitHub, 65
Google Compute Engine, 3, 11, 204, 204
group parameter, 216
group variable files, 58
group variables
 in own files, 57
 inside of inventory, 56
group(s)
 adding new EC2 instance to, 221
 assigning to Vagrant virtual machines,
 200-202
 inventory with, 51-56
 made up of other groups, 55
 numbered hosts, 55
groups variable, 79, 80
group_by, 67
group_names variable, 79
group_vars directory, 58
Gunicorn, 52, 87, 110, 245, 257

H

handlers, 36, 41-43
 notifying, 42
 pitfalls, 42
HAProxy, 122
hardware virtualization, 237
hash_behaviour configuration, 283
homepage, custom, 25
host keys, 279-282, 281
host state, tracking, 36
host variable files, 58
host(s), 45
 assigning roles to, 148-153
 configuring for pipelining, 165
 in own files, 57
 inside of inventory, 56
 limiting which ones run, 132
 patterns for specifying, 131
 retrieving IP address from, 128
hostfile configuration, 283
hostvars, 79
host_key_checking, 281, 283
host_vars directory, 58
HP Public Cloud, 11
hypervisor, 222

I

IAM (Identity and Access Management) roles
 using for EC2 credentials, 289-292
 with AWS CLI tool, 290-292
 with AWS management console, 289
idempotent commands
 collectstatic, 107
 migrate, 107
 syncdb, 107
idempotent modules, 6
idempotent playbooks, EC2, 225-226
image parameter, 215
images, Docker, 242-247
image_id, 222
infrastructure-as-a-service (IaaS) clouds, 203
 basics of, 203
 EC2, 203-236
initializer arguments
 add_file_common_args, 184
 argument_spec, 184
 bypass_checks, 184
 check_invalid_arguments, 184
 mutually_exclusive, 184

no_log, 184
required_one_of, 184
required_together, 184
supports_check_mode, 184
inode return value, 265
installation, Ansible, 10
instance groups, 210
instance profiles, 290
instance type groups, 210
instance(s)
 adding to group, 221
 EC2 definition, 205
 idempotent playbooks for, 225-226
 launching new, 215
instances module, 222
instance_ids module, 222
instance_type, 222
instance_type parameter, 215
inventory
 adding entries at runtime with add_host
 and group_by, 65-68
 aliases and ports, 54
 and inventory files, 45
 behavioral inventory parameters, 49-51
 breaking out into multiple files, 65
 Django app with, 52-54
 dynamic, 59-63
 for multiple Vagrant machines, 46-48
 group variables inside inventory, 56-59
 group variables inside their own files, 57
 groups of groups, 55
 hosts inside inventory, 56-59
 hosts inside their own files, 57
 numbered hosts, 55
 of servers, 45-68
 Vagrants generation of, 198
 with groups of hosts, 51-56
inventory caching, 210
inventory files, 17, 45
 dynamic, 60
 file format, 25
inventory_hostname, 79, 80
invoke lookups, 137
IP addresses
 in EC2, 219
 private, for Vagrant, 196
 retrieving from host, 128
isblk return value, 265
ischr return value, 265

isdir return value, 265
isfifo return value, 265
isgid return value, 265
islnk return value, 265
isreg return value, 265
issock return value, 265
isuid return value, 265
italics documentation markup, 191
iteration, 92

J

Jaynes, Matt, 9
Jinja2, 10, 24, 40, 105, 132
`jinja2_extensions` configuration, 283
Joyent, 204
JSON file fact-caching backend, 168
JSON, YAML equivalents, 28-30

K

Kay, Alan, 6
`keep_remote_files` configuration, 283
kernel, dict instance, 222
key pairs, EC2, 216
key variable, 145
keypair groups, 210
`key_file` variable, 41
`key_name`, 222
`key_name` parameter, 216

L

launching instances, 205
`launch_time`, 222
layers of abstraction, 7
Le Guin, Ursula K., 2
library configuration, 283
line folding, 30
Linode, 3, 11, 204
Linux, 10, 161, 275
list hosts, 267
list tasks, 268
`list(s)`, YAML syntax, 29
load balancers, 1
`load_file_common_arguments` method, 186
local facts, 77
local-hosts, 45
`local_action` clause, 101, 121
`local_settings.py`, 103
login variable, 70

`log_path` configuration, 283
`lookup(s)`, 136-136
 `csvfile`, 136
 `dnstxt`, 136
 `env`, 136
 `etcd`, 136
 `file`, 136
 `invoke`, 137
 `password`, 136
 `pipe`, 136
 `redis_kv`, 136
 `template`, 136
`lookup_plugins` configuration, 283
looping constructs, 143-146
 `with_dict`, 143
 `with_fileglob`, 143
 `with_first_found`, 143
 `with_flattened`, 143
 `with_indexed_items`, 143
 `with_inventory_hostnames`, 143
 `with_items`, 143
 `with_lines`, 143
 `with_nested`, 143
 `with_random_choice`, 143
 `with_sequence`, 143
 `with_subelements`, 143
 `with_together`, 143

M

Mac OS X, 161, 251, 275
`madule_args` configuration, 283
`manage.py`, 106
mappings, 30
`max_fail_percentage` clause, 124
Memcached
 Docker and, 247
 fact caching backend, 170
 starting the container, 258
memory-based caching systems, 1
message queues, 1
Mezzanine
 adding sudo clause to task, 94
 and PostgreSQL, 86
 application server, 87
 as test application, 83-88
 checking out project using Git, 96
 complex arguments in tasks, 99-102
 container image in Docker, 242-247
 creating database and database user, 102

deploying in Docker containers, 240-241
deploying on multiple machines, 119
deploying with Ansible, 89-119
development mode vs. production mode, 83-88
Django and, 87, 106
enabling nginx configuration, 113
Fabric scripts, 113
full playbook, 115
generating local_settings.py from template, 103
Gunicorn and, 87
installing into a virtualenv, 97-99
installing TLS certificates, 113
installing twitter cron job, 114
listing tasks in playbook, 89
Nginx and, 87
organization of deployed files, 90
process manager, 88
roles, 148-153, 153-157
running custom Python scripts, 107-111
running django-manage commands, 106
running playbook against a Vagrant machine, 118
setting service configuration files, 110
simplifications, 85
starting container in Docker, 258
Supervisor and, 88
updating Apt cache, 94
using iteration to install multiple packages, 92
variables and secret variables, 90-92
web server, 87
mezzanine-project command, 85
Microsoft, 55
Microsoft Azure, 11, 204
migrate command, 107
mode return value, 265
module documentation markup, 191
modules, 6, 35
 built-in, 6
 declarative, 6
 documentation, 35
 facts returned by, 76
 idempotent, 6
 in playbooks, 34
module_lang configuration, 283
module_name configuration, 283
msg (variable), 177

mtime return value, 265
multiple remote servers, 3
Mustache, 40
mutually_exclusive, 184, 185

N

Nagios, 122
name setting, 33
native SSH, 273
network address translation (NAT), 173
Nginx, 87
 Docker and, 248-249
 starting container in Docker, 259
nginx config file, 24, 40, 113
nlink return value, 265
nocolor configuration, 283
nocows configuration, 283
non-Python-based modules, 176
NoSQL databases, 1
no_log, 184
numbered hosts, 55

O

OpenSSH, 161
OpenStack APIs, 3, 204
operating system virtualization, 237
optional settings, 33
orchestration, 3
output variables, custom module, 177
outputs, custom module, 176

P

packages
 dependencies, 97
 Python, 92
 system level, 92
Packer, building AMI with, 232-236
parallel provisioning, 199
parallelism, 170
paramiko, 10, 50
paramiko configurations
 pty, 286
 record_host_keys, 286
parsing arguments, 179
password lookup, 138
path_prefix run_command argument, 188
pattern configuration, 283
pattern, for specifying hosts, 131

pets (numbered hosts), 55
ping module, 19
pip, 10, 92
pip freeze command, 99
pip module, 97
pipe lookup, 138
pipelining, 165
pipelining connection, 286
pipsi, 11
placement, dict instance, 222
play(s), 32
playbook(s)
 anatomy of, 31-35
 assigning roles to hosts in, 148-150
 basics of, 21-44
 checking before execution, 267-269
 configuring Vagrant for, 21
 custom homepage for, 25
 debugging, 263-270
 defining variables in, 69
 deploying Mezzanine against Vagrant
 machine, 118
 Docker, 260
 EC2, 221
 full Mezzanine playbook, 115
 generating TLS certificate, 38
 handlers, 41-43
 idempotent, with EC2, 225-226
 listing tasks in, 89
 Mezzanine deployment, 89-119
 modules in, 34
 nginx config file for, 24, 40
 plays in, 32
 reducing execution time of, 161-171
 running simple example, 26-28
 running the TLS support example, 43
 simple example, 22-28
 tasks in, 33
 TLS support example, 36-44
 tracking host state, 36
 webservers group for, 25
 YAML syntax for, 28-30
play_hosts variable, 79
poll_interval configuration, 283
port forwarding, 196
port(s)
 and hosts, 54
 for EC2 security groups, 220
post-tasks, 150

Postgres, 56, 102
 customized configuration files, 151
 database container, 251
 Docker image, 247
 retrieving database container IP address and
 mapped port, 252-255
 waiting for database to start up, 256-262
Postgres database, 52
PostgreSQL, 86
Power User Access, 289
powershell, 50
pre-tasks, 150
precedence rules, 82
private ip, 222
private networks, 197
private_dns_name, 222
private_key_file configuration, 283
process manager, 88
production environments, 52
production mode, development mode vs.,
 83-88
proto parameter, 219
provisioner, 3, 234
 Ansible, for Vagrant, 197
 provisioning in parallel, 199
provisioning, 3
proxies, reverse, 87
pty configuration, 286
public_dns_name, 222
public_ip, 222
pull-based, 5
Puppet, 2
push-based agents, 5
pw_name return value, 265
Python, 9
 Boto library for EC2, 207
 custom modules written in, 175
 module implementation in, 178-190
 package manager, 10
 packages, 92
 running custom scripts in Mezzanine,
 107-111
 virtualenv, 11
Python Memcached package, 170
Python Paramiko library, 63

Q

quoting, 39

R

RabbitMQ, 52, 56
Rackspace, 3, 11, 204, 204
ramdisk, 222
`record_host_keys` configuration, 286
Redis fact caching backend, 169
redis Python package, 141
`redis_kv` lookup, 142
redundancies, 1
region groups, 210
region parameter, 215
registered variables, 70, 133
registries, 239
remote hosts, 5
remote servers, 173
`remote_port` configuration, 283
`remote_tmp` configuration, 283
`remote_user` configuration, 283
required (option), 181
required_one_of, 184, 185
required_together initializer argument, 184
requires argument options, 181
reverse proxies, 87
`rgrp` return value, 265
Rocannons World (book), 2
roles, 8
 assigning to hosts in playbooks, 148-150
 basic structure, 147-160
 creating role files/directories with ansible-galaxy, 157
 database, 148-153
 defining variables in, 153
 dependent, 158
 directories for, 148
 for deploying database, 150-153
 for scaling up playbooks, 147-160
 IAM, 289-292
 in Ansible Galaxy, 159
 Mezzanine, 148-153, 153-157
 pre-tasks and post-tasks, 150
 roles_path configuration, 283
 root_device_name, 222
 root_device_type, 222
 roth return value, 265
 Ruby, 192
 running instances, 205
 run_command arguments
 args, 188
 binary_data, 188

 check_rc, 188
 close_fds, 188
 cwd, 188
 data, 188
 executable, 188
 path_prefix, 188
 use_unsafe_shell, 188
rusr return value, 265

S

Salt, 2
scalability, 6
scaling down, 6
`scp_if_ssh` connection, 286
script modules, 108, 174
secret variables, 90-92
security group rule parameters, 219
 cidr_ip, 219
 from_port, 219
 proto, 219
 to_port, 219
security groups, 210, 218, 220
sensitive data, encrypting, 129-131
sequences, 29
serial clause, 123
server(s)
 custom modules for checking reachability of, 173
 for testing, 11-20
 inventory of, 45-68
 viewing all facts associated with, 75
 waiting for, in EC2, 224
server_name variable, 41
service, 88
service configuration files, 110
service discovery mechanism, 245
service module, 34
setup module, 75
set_fact, 78
set_fs_attributes_if_different method, 186
shell module, 72, 173
shorthand input, Bash modules, 193
simplejson library, 5
size return value, 265
smart transport, 49
SoftLayer, 11, 204
software, development mode vs. production mode, 83-88
SQL databases, 1

- SSH
agent forwarding, 275-278
debugging, 263
features of, 273-282
host keys, 279-282
native, 273
ssh-agent, 273-275, 275
- SSH key pairs, 216
- SSH multiplexing, 161-164, 161
ControlMaster, 163
ControlPath, 163
ControlPersist, 163
manually enabling, 162
options in Ansible, 163
- ssh-agent program, 15
- ssh_args connection, 286
- ssh_connections
control_path, 286
pipelining, 286
scp_if_ssh, 286
ssh_args, 286
- SSL, TLS vs., 37
- staging environments, 52
- stale data, 167
- start at task, 270
- start of file, YAML syntax, 28
- stat module return values, 265
- state, dict instance, 222
- state_code, 222
- static assets, 87
- step flag, 269
- strings, YAML syntax, 28
- su configuration, 283
- subsets, fact, 75
- Sudo, 277
- sudo clause, 94
- sudo configuration, 283
- sudo setting, 33
- sudo_exe configuration, 283
- sudo_flags configuration, 283
- sudo_user configuration, 283
- Supervisor, 110
as Mezzanine process manager, 88
Docker alternative to, 245
- supports_check_mode initializer argument, 184
- surround_by_quote function, 135
- su_exe configuration, 283
- su_flag configuration, 283
- su_user configuration, 283
- syncdb command, 107
- syntax check, 267
- syntax, Ansible, 5
- syslog_facility configuration, 283
- system-level packages, 92
- system_warnings configuration, 283

T

- tag groups, 210
- tag(s)
added to task/play for debugging, 270
applying to existing EC2 resources, 212
defining EC2 dynamic groups with, 211-213
group names for, 213
in EC2, 205
- tagged_instances module, 222
- task queues, 1
- task(s), 3, 33, 35
adding sudo clause, 94
changed_when and failed_when clauses, 125-128
complex arguments in, 99-102
in playbooks, 33
limiting, for debugging, 269
listing in playbook, 89
running on control machine, 121
running on machine other than host, 122
running on one host at a time, 123
running only once, 124
- Taste Test: Puppet, Chef, Salt, Ansible (books), 9
- template lookup, 136, 139
- template module, 34, 277
- template, generating local_settings.py from, 103
- templates, 24
Django, 40
ERB, 40
Mustache, 40
- test server, Ansible
inventory files and, 15
setup, 11-20
simplifying with ansible.cfg file, 16-20
Vagrant for, 12-14
- testserver, 25
- timeout configuration, 283
- TLS
playbooks and, 36-44
SSL vs., 37
- TLS certificate

generating, 38
installing in Mezzanine, 113
TLS-supported playbook, 36-44
to_port parameter, 219
transports, 49
true, yes vs., 23
twitter cron job, 114
type argument options, 181
type option, 182

U

Ubuntu, 4, 92
images, 214
MAAS, 59
uid return value, 265
URL documentation markup, 191
use_unsafe_shell run_command argument, 188

V

Vagrant, 3, 105, 195-202
agent forwarding, 197
Ansible provisioner, 197
configuring for playbooks, 21
convenient configuration options for, 195-197
inventory for multiple machines, 46-48
inventory generated by, 198
port forwarding, 196
private IP addresses, 196
provisioner, 197
provisioning in parallel, 199
running Mezzanine playbook against, 118
setting up test server with, 12-14
specifying groups, 200-202
Vagrant environments, 52
vagrant status command, 61
validate clause, 277
value variable, 145
variables, 69-74
accessing dictionary keys in, 73
built-in, 79
custom module output, 177
defining in playbooks, 69
defining in roles, 153
defining with set_fact, 78
environment for EC2 credentials, 206
filters for registered, 133
groups variable, 80
hostvars, 79

in Mezzanine, 90-92
in playbook with TLS, 38-39
inventory_hostname, 80
precedence rules, 82
registering, 70-74
setting on command line, 81
viewing values of, 70
vars setting, 33
vars_plugins configuration, 283
vault, encrypting data with, 129-131
vault_password_file configuration, 283
version control, 17

Virtual Private Cloud (VPC), 213

basics, 228
dynamic inventory and, 231
specifying, 228-232

VirtualBox, 12

virtualenv, 97-99

virtualization

hardware, 237
operating system, 237

VMWare vSphere, 204

volumes_from parameter, 251

VPC groups, 210

W

wait_for module, 121, 256
Web Server Gateway Interface (WSGI), 87
web servers, 1
webservers group, 25
wgrp return value, 265
whoami command, 70
Windows, 50
with_dict looping construct, 145
with_fileglob looping construct, 144
with_flattened loop, 143
with_indexed_items loop, 143
with_inventory_hostnames loop, 143
with_items, 92
with_items loop, 143
with_lines looping construct, 144
with_nested loop, 143
with_random_choice loop, 143
with_sequence loop, 143
with_subelements loop, 143
with_together loop, 143
woth return value, 265
wusr return value, 265

X

xgrp return value, 265
xip.io, 105
xusr return value, 265

Y

YAML, 28-30
yes, true vs., 23

关于作者

Lorin Hochstein 从小在魁北克蒙特利尔长大，除了偶尔会说“闭上灯”这种话之外，从他的口音中你绝对猜不出他是加拿大人。他正在回归学术界：他已经在内布拉斯加林肯大学作为终身教职计算机科学与应用的副教授任教 2 年。此外，他还曾作为计算机科学家在南加州大学信息科学院工作 4 年。他在麦吉尔大学取得了计算机工程学学士学位，在波士顿大学取得了电子工程学的硕士学位，并在马里兰大学帕克分校取得了计算机科学的博士学位。他现在在 SendGrid 任职高级软件工程师，为 SenGrid 实验室开发新产品。

封面

刊载在《奔跑吧 Ansible》封面的动物是一只荷斯坦奶牛（原始牛）。在北美经常被简称为 Holstein，而在欧洲则被简称为 Friesian。这种牛源于现在的欧洲荷兰地区，主要培育的原因是为了培养大量食草的（因为草是当地最富饶的资源），并且又具有高产特征的黑白奶牛。荷斯坦奶牛于 1621—1664 年间被引进到美国，但美国的饲养者直到 19 世纪 30 年代才开始对这个品种感兴趣。

荷斯坦奶牛以它们身上大块的黑白印记和高产奶量而著名。黑白的花纹是由培育者的人工选择培育方式造成的。健康的小牛出生时一般在 90 ~ 100 磅之间，而成年荷斯坦奶牛则可以达到 1280 磅重 58 英寸高。这个品种的母牛普遍的生产周期是 13 ~ 15 个月，怀孕期为九个半月。

这个品种的牛平均每年产奶 2022 加仑，纯种的平均每年能达到 2146 加仑，一生当中的产奶量可达到 6898 加仑。

2000 年 9 月，一头叫 Hanoverhill Starbuck 的荷斯坦奶牛使这个品种成为了争论的焦点。因为人们用它死前一个月提取的冷冻纤维细胞克隆了 Starbuck 二代。这头克隆牛比原来的 Starbuck 晚生了 21 年 5 个月。

O'Reilly 封面上的很多动物都濒临灭绝；对于世界来说它们都非常重要。如果想要了解怎么去帮助它们，请登录 animal.oreilly.com。

奔跑吧Ansible

在当下百家争鸣的配置管理工具领域中，Ansible有着独特的优势：原生即轻量。你不需要在你要管理的机器上安装任何软件，并且它的学习曲线非常平缓。不管是想要将代码部署到生产环境的开发者，还是寻求更好的自动化解决方案的系统管理员，这本实用指南都会帮助你快速地在生产环境中使用这个工具。

作者Lorin Hochstein示范了如何编写playbook（Ansible中的配置管理脚本），如何管理远程服务器，并带你探索了这个工具的潜在法宝：内置声明模块。通过阅读本书，你会发现Ansible不仅拥有你所需要的功能，而且还异乎寻常地简单。

- 理解Ansible区别于其他配置管理系统的特点。
- 使用YAML语法编写你自己的playbook。
- 学习Ansible中的变量与fact。
- 部署一个复杂应用的完整范例。
- 使用role来简化与复用playbook。
- 使用SSH Multiplexing、pipelining和并行化来让playbook运行得更快。
- 将应用部署到Amazon EC2或其他云平台。
- 使用Ansible创建Docker镜像并部署Docker容器。

Lorin Hochstein任职于SendGrid实验室，是一名负责开发与部署新产品的高级软件工程师。他曾在Nimbis Services任职云服务首席架构师，还曾经是加州大学信息科学院的计算机科学家。Lorin拥有马里兰大学计算机科学博士学位。

SYSTEM ADMINISTRATION

图书分类: 系统配置管理

策划编辑: 张春雨

责任编辑: 付睿



Broadview®
WWW.BROADVIEW.COM.CN

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

“本书对Ansible的介绍非常精彩，对于Ansible新手与当前用户都是极佳的参考书。我曾以为我了解Ansible的方方面面，但阅读Lorin的这本书还是学到了很多新东西。”

——Matt Jaynes

Taste Test: Puppet vs Chef vs SaltStack vs Ansible 的作者

ISBN 978-7-121-27507-4



9 787121 275074 >

定价: 79.00元

[General Information]

书名=奔跑吧ANSIBLE

作者=(加)LORIN HOCHSTEIN著

页数=318

SS号=13918610

DX号=

出版日期=2016.01

出版社=北京电子工业出版社