



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Practical DevOps

Harness the power of DevOps to boost your skill set and make your IT organization perform better

Joakim Verona

[PACKT] open source*

PUBLISHING

community experience distilled

实用的 DevOps

利用 DevOps 的力量来提高您的技能组合并使您的 IT 组织表现更好

乔金·维罗纳



伯明翰 - 孟买

实用的 DevOps

版权所有 © 2016 Packt Publishing

版权所有。未经出版商事先书面许可，不得以任何形式或通过任何方式复制本书的任何部分、将其存储在检索系统中或传播，但在重要文章或评论中嵌入简短引述的情况下除外。

在准备本书时已尽一切努力确保所提供信息的准确性。但是，本书中包含的信息在出售时没有任何明示或暗示的保证。作者、Packt Publishing 及其经销商和分销商均不对本书直接或间接造成或据称造成的任何损害承担责任。

Packt Publishing 已尽力通过适当使用大写字母来提供有关本书中提及的所有公司和产品的商标信息。

但是，Packt Publishing 无法保证此信息的准确性。

首次发布：2016 年 2 月

生产编号：1100216

由 Packt Publishing Ltd 出版。

Livery Place
35 Livery Street
伯明翰 B3 2PB, 英国。

书号 978-1-78588-287-6

www.packtpub.com

学分

作者	项目协调员
乔金·维罗纳	桑奇塔曼达尔
审稿人	校对
赫德曼	安全编辑
马克斯曼德斯	索引器
委托编辑	赫曼吉尼巴里
维娜帕加雷	图形
收购编辑	柯克德佩尼亚
索娜莉·韦尔内卡	生产协调员
内容开发编辑	Shantanu N. Zagade
萨曼莎贡萨尔维斯	封面作品
技术编辑	Shantanu N. Zagade
侯赛因坎奇瓦拉	
文案编辑	
马杜苏丹·乌奇尔	

关于作者

Joakim Verona是一名顾问,擅长持续交付和 DevOps。自 1994 年以来,他一直从事系统开发的各个方面的工作。作为复杂多层系统 (如 Web 系统、多媒体系统和混合软件/硬件系统)的主要实施者,他积极做出贡献。他广泛的技术兴趣使他在 2004 年进入了新兴的 DevOps 领域,此后他一直留在该领域。

Joakim 在林雪平理工学院完成了计算机科学硕士学位。他还曾在银行和金融、电信、工业工程、新闻和出版以及游戏开发等各个行业担任过广泛任务的顾问。他还对敏捷领域感兴趣,是一名经过认证的 Scrum 大师、Scrum 产品负责人和 Java 专家。

我要感谢我的妻子玛丽,是她在本书写作过程中给了我灵感。我还要感谢多年来与我共事的所有人和公司,是他们让我能够从事我喜欢的工作!

关于审稿人

Per Hedman 是一位充满激情的开发人员,他是 DevOps 和持续交付的坚定支持者,并认为您应该赋予开发人员权力,他们应该对自己编写的代码负责。

他的职业是软件顾问,自 2000 年代初以来一直是开发和运营人员。

特别感谢我的妻子和我的两个女儿让我微笑。

Max Manders 是 FanDuel 的运营工程师,FanDuel 是在线每日奇幻运动的领导者。Max 之前曾在 Cloudreach 的运营中心工作,Cloudreach 是 Amazon Web Services 高级咨询合作伙伴。Max 充分利用了他过去的经验和技能,以促进 DevOps 的发展;他还致力于通过 Chef 和 Puppet 掌握 Ruby 并提倡基础架构即代码。

Max 是 Whiskey Web 的联合创始人和组织者,这是一个针对网络开发和运营社区的苏格兰会议。当他不编写代码或修补最新最好的监控和操作工具时,Max 喜欢威士忌以及演奏爵士乐和放克长号。Max 和他的妻子 Jo 以及他们的猫 Ziggy 和 Maggie 住在爱丁堡。

www.PacktPub.com

支持文件、电子书、折扣优惠等

如需与本书相关的支持文件和下载,请访问www.PacktPub.com。

您是否知道 Packt 提供每本已出版书籍的电子书版本,并提供 PDF 和 ePub 文件?您可以在 www.PacktPub.com 升级到电子书版本作为印刷书籍的客户,您有权享受电子书副本的折扣。请通过 service@packtpub.com 联系我们了解更多详情。

在www.PacktPub.com,您还可以阅读一系列免费技术文章,注册一系列免费时事通讯,并获得 Packt 书籍和电子书的独家折扣和优惠。



<https://www2.packtpub.com/books/subscription/packtlib>

您需要即时解决您的 IT 问题吗? PacktLib 是 Packt 的在线数字图书图书馆。在这里,您可以搜索、访问和阅读 Packt 的整个图书库。

为什么要订阅? · 可在

Packt 出版的每本书中完全搜索 · 复制和粘贴、打印和书签内容

· 按需并通过网络浏览器访问

Packt 账户持有人可免费访问如果您在www.PacktPub.com上拥

有 Packt 账户,您现在可以使用它来访问 PacktLib 并查看 9 本完全免费的书籍。只需使用您的登录凭据即可立即访问。

目录

<u>前言</u>	<u>七</u>
<u>第 1 章 :DevOps 和持续交付简介</u>	<u>DevOps 简介 多快才算快?</u>
	<small>15个</small>
敏捷之轮 谨防货物崇拜 敏捷谬误	1
DevOps 和 ITIL 总结	3
第 2 章 :Orbit 视角	5
DevOps	6
过程和持续交付 概述	7
开发人员 版本控制系统 构建	8
服务器 工件存储库 包管理器 测试环境 暂存/生成	8个
发布管理 Scrum、看板和交付管道 总结 一个完整的例子	9
识别瓶颈 总结	9
第 3 章 :DevOps 如何影响架构	11
总结	12
	13
	14
	14
	15
	16
	16
	17
	17
	18
	19
	20
	21
介绍软件架构	21
单体场景	22
架构经验法则	23
关注点分离	23
凝聚力原则	24

目录

耦合 回到单体场景 一个实际例子 三层系统 表现层 逻辑层 数据层 处理数据库迁移 滚动升级 Liquibase 中的 Hello world 变更日志文件 pom.xml 文件 手动安装 微服务插曲 – 康威定律 如何保持服务接口向前兼容微服务和数据层 DevOps、架构和弹性总结第 4 章:一切都是代码	24
源代码控制的必要性 源代码管理的历史 角色 和代码 哪个源代码管理系统?	40
关于源代码管理系统迁移的一句话 选择分支策略 分支问题区域 工件版本命名 选择客户端 设置基本 Git 服务器 共享身份验证 托管 Git 服务器 大型二进制文件 尝试不同的 Git 服务器实现 Docker intermission Gerrit 安装 git-review 包历史修正主义的价值	43
[二]	55

目录

拉取请求模型 GitLab 总结	57
第 5 章:构建代码我们为什么	57
要构建代码?	59
	61
构建系统的方方面面 Jenkins 构建服务	61
器 管理构建依赖关系 最终工件 FPM 作	62
弊 持续集成 持续交付 Jenkins 插件 主	63
机服务器 构建从属主机上的软件 触发	66
器 作业链和构建管道 看看 Jenkins 文件	67
系统布局 构建服务器和基础架构即代码	68
按依赖顺序构建 构建阶段 替代构建服务	69
器 整理质量度量 关于构建状态可视化	70
认真对待构建错误 稳健性总结 第 6 章:测	70
试代码手动测试 测试自动化的优缺点 单	72
元测试 JUnit 和特定的 JUnit JUnit示	72
例Mocking Test Coverage 自动化集成	73
测试Docker in automated testing	74
Arquillian	74
	75
	76
	76
	77
	77
	78
	78
	79
	79
	80
	80
	81
	81
	83
	83
	84
	84
	86
	86
	87
	87
	89
	89
	89
	89
	90
	90
	91
	91
	92

目录

性能测试 自动化验收测试 自动化 GUI 测试 在 Jenkins 中集成 Selenium	92
测试 JavaScript 测试 测试后端集成点 测试 驱动开发 REPL 驱动开发 完整的测试自动化场景手动	95
测试我们的 Web 应用程序 运行自动化测试 查找错误 测试演练 处理棘手的依赖关系 Docker 小结 第 7 章 部署	96
代码为什么会有这么多部署系统？	98
	100
	100
	101
	101
	102
	105
	105
	105
	105
	109
	110
	111
	111
配置基本操作系统 描述集群 将包传送到系统 虚拟化堆栈	112
	113
	114
	116
在客户端执行代码 关于练习的注意事项 傀儡师和傀儡特工 Ansible 的 托盘操作 使用 Chef 进行部署 使用 SaltStack 部署 Salt 与 Ansible 与 Puppet 与 PalletOps 执行模型	118
	118
	119
	120
	123
	124
	125
	127
流浪汉 使用 Docker 部署 比较表 云解决方案 AWS 蔚蓝 概括	128
	130
	131
	131
	132
	132
	133

目录

第 8 章:监控代码	135
Nagios	135
穆宁	141
神经节	145
石墨	150
日志处理	152
客户端日志记录库	153
麋鹿栈	155
概括	158
第 9 章:问题跟踪	159
问题跟踪器有什么用?	159
工作流程和问题的一些示例 我们需要从问题跟踪器中获得什么?	161
问题跟踪器扩散问题 所有跟踪器Bugzilla Trac Redmine	166
GitLab 问题跟踪器 Jira总结	167
168	168
174	174
182	182
188	188
192	192
194	194
第 10 章:物联网和 DevOps 介绍 IoT 和 DevOps 市场对 IoT 的未来	195
机器对机器通信 IoT 部署影响软件架构 IoT 部署安全 好的,但是 DevOps 和 IoT 又如何呢?	195
199	199
201	201
202	202
203	203
具有用于 DevOps 摘要索引的 IoT 设备的动手实验室	205
211	211
213	213

Machine Translated by Google

前言

近年来,DevOps 领域变得流行和普遍。它变得如此普遍以至于很容易忘记,在 2008 年之前,当 Patrick Debois 组织了第一届 DevOpsDays 会议时,几乎没有人听说过

这个单词。

DevOps 是“开发人员”和“运维”这两个词的合成词,它的含义是什么?为什么它会引起如此巨大的兴奋?

本书的使命就是回答这个看似简单的问题。

简短的回答是,DevOps 旨在将不同的社区(例如开发人员和运营社区)聚集在一起,成为一个更高效的整体。

这在书中也有所体现。它探索了许多在 DevOps 工作中有用的工具,拉近人们距离的工具总是优于那些在人与人之间创建人为边界的工具。我们用于软件开发的过程也是工具,因此很自然地包括与 DevOps 相关的各种敏捷思想流派的各个方面。

正如书名所暗示的那样,这本书的目标还在于实用性。

让我们在 DevOps 的土地上开始我们的旅程吧!

本书涵盖的内容第1章, DevOps 和持续交付简介,
介绍了 DevOps 的背景,并为 DevOps 如何适应更广阔的敏捷系统开发领域奠定了基础。

第2 章, Orbit 的视角,将帮助您了解我们在 DevOps 中使用的所有系统如何组合在一起,形成一个更大的整体。

前言

第3章, DevOps 如何影响架构,描述了软件架构的各个方面以及它们对我们戴着 DevOps 眼镜工作时的意义。

第4章,一切都是代码,解释了为什么一切都是代码,并且您需要某个地方来存储它。组织的源代码管理系统就是那个地方。

第5章,构建代码,解释了您如何需要系统来构建您的代码。
它们在本章中进行了描述。

第6章,测试代码,告诉您如果您要尽早并经常发布您的代码,您必须对其质量有信心。因此,您需要自动化回归测试。

第7章,部署代码,说明在构建和测试代码后,您需要如何将其部署到您的服务器,以便您的客户可以使用新开发的功能。

第8章,监控代码,介绍了如何使用您选择的部署解决方案将代码安全地部署到您的服务器;您需要监视它以确保它正常运行。

第9章,问题跟踪,着眼于用于处理组织内开发工作流程的系统,例如问题跟踪软件。在实施敏捷流程时,此类系统是一个重要的辅助工具。

第 10 章,物联网和 DevOps,描述了 DevOps 如何在新兴的物联网领域帮助我们]。

这本书你需要什么

本书包含许多实用示例。要完成这些示例,您需要一台最好装有基于 GNU/Linux 的操作系统 (例如 Fedora)的机器。

本书的读者对象 本书面向希望承担更大责任并

了解构建当今企业的基础架构如何工作的开发人员。它还适用于希望更好地支持其开发人员的操作人员。使用测试自动化的技术测试人员也包括在目标受众中。

前言

这本书主要是一本技术文本,带有实用示例,适合喜欢通过实施具体工作代码来学习的人。尽管如此,前两章的方法不太实用。它们为您提供了解其余章节背后动机所需的背景和概述。

约定在本书中,您会发现许

多区分不同类型信息的文本样式。以下是这些样式的一些示例及其含义的解释:

文本中的代码字、数据库表名称、文件夹名称、文件名、文件扩展名、路径名、虚拟 URL、用户输入和 Twitter 句柄显示如下：“在本地安装上安装git-review。”

一段代码设置如下：

```
私有 int 正值; void setPositiveValue(int
x){ this.positiveValue=x;

}

int getPositiveValue(){ 返回正值;

}
```

任何命令行输入或输出都写成如下：

```
docker run -d -p 4444:4444 --name selenium-hub 硒/集线器
```

新术语和重要单词以粗体显示。您在屏幕上看到的文字,例如,在菜单或对话框中,以如下文本形式出现：“我们可以使用修改按钮更改状态。”



警告或重要说明出现在这样的框中。



提示和技巧是这样出现的。

前言

读者反馈我们随时欢迎读者提供反馈。让我们知道您对这本书的看法，您喜欢或不喜欢的地方。读者反馈对我们很重要，因为它可以帮助我们开发出您真正能从中获得最大收益的作品。

要向我们发送一般反馈，只需发送电子邮件至feedback@packtpub.com，并在邮件主题中提及该书的标题。

如果您在某个主题上具有专业知识，并且有兴趣撰写或投稿一本书，请参阅我们的作者指南，网址为www.packtpub.com/authors。

客户支持

既然您是 Packt 书籍的骄傲拥有者，我们有很多东西可以帮助您从购买中获得最大收益。

下载示例代码

您可以从您的帐户下载示例代码文件，网址为<http://www.packtpub.com>对于您购买的所有 Packt Publishing 书籍。如果您在其他地方购买了本书，您可以访问<http://www.packtpub.com/support>并注册以便将文件直接通过电子邮件发送给您。

为了使代码示例在快速发展的 DevOps 世界中保持最新，本书的代码示例也可以在这个 GitHub 存储库中找到：<https://github.com/jave/practicaldevops.git>。

您可以按照以下步骤下载代码文件：

1. 使用您的电子邮件地址和密码登录或注册我们的网站。
2. 将鼠标指针悬停在顶部的支持选项卡上。
3. 单击代码下载和勘误表。
4. 在搜索框中输入书名。
5. 选择您要为其下载代码文件的书籍。
6. 从您购买本书的下拉菜单中选择。
7. 单击代码下载。

前言

下载文件后,请确保使用以下最新版本解压缩或提取文件夹:

- 适用于 Windows 的 WinRAR / 7-Zip
- 适用于 Mac 的 Zipeg / iZip / UnRarX
- 适用于 Linux 的 7-Zip / PeaZip

勘误表尽管

我们已竭尽全力确保内容的准确性,但错误还是时有发生。如果您在我们的一本书中发现错误 可能是文本或代码中的错误 如果您能向我们报告,我们将不胜感激。通过这样做,您可以使其他读者免于沮丧,并帮助我们改进本书的后续版本。如果您发现任何勘误,请访问<http://www.packtpub.com/submit-errata>,选择你的书,点击Errata Submission Form链接,然后输入你的勘误的细节。一旦您的勘误得到验证,您的提交将被接受,勘误将被上传到我们的网站或添加到该标题勘误部分下的任何现有勘误列表中。

要查看之前提交的勘误表,请转至<https://www.packtpub.com/books/content/support>并在搜索字段中输入书名。所需信息将出现在勘误表部分下。

海盗行为

互联网上版权材料的盗版是所有媒体的一个持续问题。在 Packt,我们非常重视版权和许可的保护。

如果您在互联网上发现任何形式的非法复制我们的作品,请立即向我们提供位置地址或网站名称,以便我们进行补救。

请通过copyright@packtpub.com与我们联系,并提供涉嫌盗版材料的链接。

感谢您帮助保护我们的作者以及我们为您带来的能力
有价值的内容。

问题如果您对本书的

任何方面有疑问,可以通过questions@packtpub.com联系我们,我们将尽最大努力
解决问题。

Machine Translated by Google

1

DevOps 简介和 持续交付

欢迎来到实用 DevOps！

本书的第一章将介绍 DevOps 的背景，并为 DevOps 如何适应更广阔的敏捷系统开发世界做准备。

DevOps 的一个重要部分是能够向您组织中的同事解释什么是 DevOps 以及什么不是。

你越快让每个人都登上 DevOps 列车，你就能越快到达执行实际技术实施的部分！

在本章中，我们将讨论以下主题：

- 介绍 DevOps
- 多快才算快？
- 敏捷之轮 · 货运狂热的敏捷谬论 ·
- DevOps 和 ITIL

介绍 DevOps

根据定义，DevOps 是一个跨越多个学科的领域。这是一个非常实用和动手的领域，但同时，你必须了解技术背景和非技术文化方面。本书涵盖了在您的组织中实施最佳 DevOps 所需的实用技能和软技能。

DevOps 和持续交付简介

“DevOps”这个词是“开发”和“运营”这两个词的组合。这个文字游戏已经向我们暗示了 DevOps 背后理念的基本性质。这是一种鼓励不同软件开发学科之间协作的实践。

DevOps 这个词的起源和 DevOps 运动的早期可以相当准确地追踪：Patrick Debois 是一名软件开发人员和顾问，在 IT 的许多领域都有经验。他对开发人员和运营人员之间的分歧感到沮丧。他试图让人们在会议上对这个问题感兴趣，但最初并没有太大兴趣。

2009 年，在 O'Reilly Velocity 会议上的一个广受好评的演讲：“每天 10 次以上的部署：Flickr 的开发和运营合作”。然后，Patrick 决定在比利时根特组织一个名为 DevOpsDays 的活动。这一次引起了很大的兴趣，会议取得了成功。“DevOpsDays”这个名字引起了共鸣，会议已经成为经常性的活动。DevOpsDays 在 Twitter 和各种互联网论坛上的对话中缩写为“DevOps”。

DevOps 运动源于敏捷软件开发原则。

敏捷宣言于 2001 年由许多人撰写，他们希望改善当时系统开发的现状并在软件开发行业中找到新的工作方式。以下是 Agile Manifesto 的摘录，它现在是经典文本，可在 Web 上获取，网址为 <http://agilemanifesto.org/>：

“个人和交互优于流程和工具

综合文档之上的工作软件

客户协作优于合同谈判

响应变化胜于遵循计划

也就是说，虽然右边的物品有价值，但我们更看重左边的物品。”

鉴于此，DevOps 可以说与第一原则相关，“个人和交互高于流程和工具”。

这可能被视为一种相当明显有益的工作方式。为什么我们还要陈述这个显而易见的事实呢？好吧，如果你曾经在任何大型组织工作过，你就会知道相反的原则似乎反而在起作用。一个组织的不同部分之间往往很容易形成壁垒，即使是在较小的组织中，起初似乎不可能形成这样的壁垒。

因此,DevOps 倾向于强调个人之间的互动非常重要,而技术可能有助于实现这些互动并拆除组织内部的壁垒。这似乎违反直觉,因为第一个原则有利于人与人之间的互动而不是工具,但我认为任何工具在使用时都会产生多种效果。

如果我们正确使用这些工具,它们可以促进敏捷工作场所的所有所需属性。

一个非常简单的例子可能是用于报告错误的系统的选择。开发团队和质量保证团队经常使用不同的系统来处理任务和错误。这会在团队之间造成不必要的摩擦,并在他们真正应该专注于一起工作时进一步分离他们。

运营团队可能反过来使用第三个系统来处理部署到组织服务器的请求。

另一方面,具有 DevOps 思维的工程师会立即将所有三个系统识别为具有相似属性的工作流系统。三个不同团队中的每个人都应该可以使用相同的系统,也许可以进行调整以针对不同的角色生成不同的视图。另一个好处是维护成本更低,因为三个系统被一个系统取代了。

DevOps 的另一个核心目标是自动化和持续交付。简而言之,将重复和乏味的任务自动化,可以为人工交互留出更多时间,从而创造真正的价值。

多快才算快?

DevOps 流程的周转必须很快。我们需要从更大的角度考虑上市时间,从更小的角度关注我们的任务。持续交付运动也持有这种思路。

与敏捷的许多事物一样,DevOps 和持续交付中的许多想法实际上是相同基本概念的不同名称。这两个概念之间确实没有任何争论。它们是同一枚硬币的两面。

DevOps 工程师致力于使企业流程更快、更高效、更可靠。尽可能避免容易出错的重复性体力劳动。

然而,在使用 DevOps 实施时很容易忘记目标。什么都不做更快对任何人都没有用。
相反,我们必须跟踪交付增加的业务价值。

DevOps 和持续交付简介

例如,增加组织中角色之间的沟通具有明显的价值。您的产品负责人可能想知道开发过程是如何进行的,并且很想看看。在这种情况下,能够快速有效地向测试环境交付代码的增量改进是很有用的。在测试环境中,相关利益相关者(例如产品所有者,当然还有质量保证团队)可以跟踪开发进度。

另一种看待它的方式是:如果你觉得自己因为不必要的等待而失去了注意力,那么你的流程或工具出了问题。如果您发现自己在编译期间观看机器人射击气球的视频,那么您的编译时间太长了!

对于在等待部署等时闲置的团队也是如此。这种怠速,当然比一个人的怠速还要昂贵。

机器人射击练习视频很有趣,软件开发也很鼓舞人心!
我们应该通过消除不必要的开销来帮助集中创造潜力。

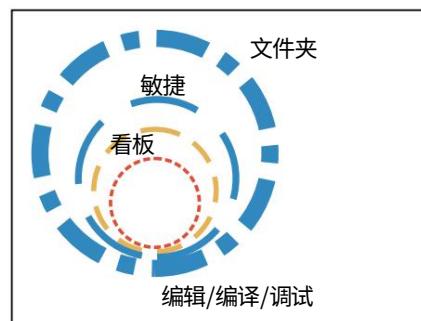


死亡射线激光机器人与您团队的生产力

敏捷之轮

敏捷开发中有几个不同的周期,从项目组合级别到 Scrum 和看板周期,再到持续集成周期。根据您正在使用的敏捷框架,对节奏工作的重点有所不同。看板强调 24 小时循环,在运营团队中很受欢迎。Scrum 周期可以在两到四个星期之间,并且经常被使用 Scrum 敏捷过程的开发团队使用。

更长的周期也很常见,在 Scaled Agile Framework 中称为Program Increments,它跨越几个 Scrum Sprint 周期。



敏捷之轮

DevOps 必须能够支持所有这些周期。鉴于 DevOps 的中心主题:敏捷组织中学科之间的合作,这是很自然的。

DevOps 最明显和可衡量的具体好处出现在更短的周期中,这反过来又使更长的周期更有效率。正如那句古老的格言所说,照顾好便士,英镑就会照顾好自己。

以下是 DevOps 何时可以使敏捷周期受益的一些示例:

- 由 DevOps 工程师维护的部署系统使 Scrum 周期结束时的交付更快、更高效。这些可以以两到四个星期的周期进行。

在主要由手工完成部署的组织中,部署时间可能需要几天。拥有这些低效部署流程的组织将从 DevOps 思维模式中受益匪浅。

- 看板周期是 24 小时,因此很明显,如果我们要成功使用看板,部署周期需要比这快得多。

一个设计良好的 DevOps 持续交付管道可以在几分钟内将代码从提交到代码存储库部署到生产,具体取决于更改的大小。

当心货物崇拜敏捷谬误

理查德·费曼 (Richard Feynman) 因其在量子物理学领域的工作而于 1965 年获得诺贝尔奖。他注意到科学家之间存在一种普遍行为,即他们遍历了科学的所有运动,但错过了科学过程中的一些核心、至关重要的成分。他将这种行为称为“货物崇拜科学”,因为它让人想起美拉尼西亚南海岛屿上的货物崇拜。这些货物崇拜是在第二次世界大战期间形成的,当时岛民们看到大型飞机载着有用的货物着陆。战争停止后,货物也不再来了。

岛民们开始模拟着陆跑道,按照他们观察到的美国军队所做的一切来让飞机着陆。



货运狂热的敏捷飞机

我们并没有以敏捷或面向 DevOps 的方式工作,仅仅因为我们有一个晨会,在那里我们喝咖啡并聊聊天气。我们没有 DevOps 管道只是因为我们有一个只有运营团队知道的 Puppet 实现。

跟踪我们的目标并不断质疑我们是否在做正确的事情以及是否仍在正确的轨道上,这一点非常重要。这是所有敏捷思想的核心。然而,这显然在实践中很难做到。很容易成为货物崇拜的追随者。

例如,在构建部署管道时,请记住我们首先构建它们的原因。目标是让人们以更少的工作更快地与新系统交互。反过来,这有助于不同角色的人更有效地相互交流,减少周转时间。

另一方面,如果我们构建的流水线只帮助一组人实现他们的目标,例如运维人员,那么我们就没有实现我们的基本目标。

虽然这不是一门精确的科学,但值得记住的是,敏捷周期,例如 Scrum 敏捷方法中的冲刺周期,通常有一种方法来处理这种情况。在 Scrum 中,这被称为 sprint 回顾,团队聚在一起讨论在 sprint 期间哪些进展顺利,哪些可以做得更好。在这里花一些时间来确保您在日常工作中做的是正确的事情。

这里的一个常见问题是冲刺回顾的输出并没有真正付诸行动。反过来,这可能是由于一个不幸的事实造成的,即所发现的问题实际上是由您没有很好沟通的组织的其他部分引起的。因此,这些问题在回顾中反复出现,始终没有解决。

如果您认识到您的团队处于这种情况,您将受益于 DevOps 方法,因为它强调组织中角色之间的合作。

总而言之,尝试在您的方法本身中使用敏捷方法中提供的机制。如果您正在使用 Scrum,请使用冲刺回顾机制来捕捉潜在的改进。话虽这么说,但不要把这些方法当作福音。找出适合您的方法。

DevOps 和 ITIL

本节解释 DevOps 和其他工作方式如何共存并融入一个更大的整体。

DevOps 非常适合敏捷或精益企业的许多框架。

Scaled Agile Framework,或 SAFe®,特别提到了 DevOps。自从 DevOps 起源于敏捷环境以来,不同敏捷实践和 DevOps 的支持者之间几乎没有任何分歧。不过,ITIL 的情况有点不同。

DevOps 和持续交付简介

ITIL,以前称为信息技术基础设施库,是许多大型成熟组织使用的实践。

ITIL 是一个将软件生命周期的许多方面形式化的大型框架。

虽然 DevOps 和持续交付认为我们交付给生产的变更集应该很小并且经常发生,但乍一看,ITIL 似乎持有相反的观点。应该指出的是,这不是真的。遗留系统通常是单一的,在这些情况下,您需要 ITIL 等流程来管理通常与大型单一系统相关的复杂变更。

如果您在大型组织中工作,那么您使用如此庞大的单一遗留系统的可能性非常高。

无论如何,ITIL 中描述的许多实践直接转化为相应的 DevOps 实践。ITIL 规定了配置管理系统和配置管理数据库。这些类型的系统也是 DevOps 不可或缺的一部分,本书将介绍其中的一些系统。

概括

本章简要概述了 DevOps 运动的背景。我们讨论了 DevOps 的历史及其在开发和运营以及敏捷运动中的根源。我们还研究了 ITIL 和 DevOps 如何在更大的组织中共存。探索了 cargo cult 反模式,我们讨论了如何避免它。您现在应该能够回答 DevOps 在哪些方面适合更大的敏捷环境以及敏捷开发的不同周期。

我们将逐渐转向更具技术和实践性的主题。下一章将概述我们在 DevOps 中倾向于关注的技术系统是什么样的。

2

从轨道上看

DevOps 流程和持续交付管道可能非常复杂。在开始实施之前 ,您需要掌握预期的最终结果。

本章将帮助您了解持续交付管道的各个系统如何组合在一起 ,形成一个更大的整体。

在本章中 ,我们将了解 :

- 概述 DevOps 流程、持续交付管道实施以及流程参与者
- 发布管理 · Scrum、看板
- 和交付管道 · 瓶颈

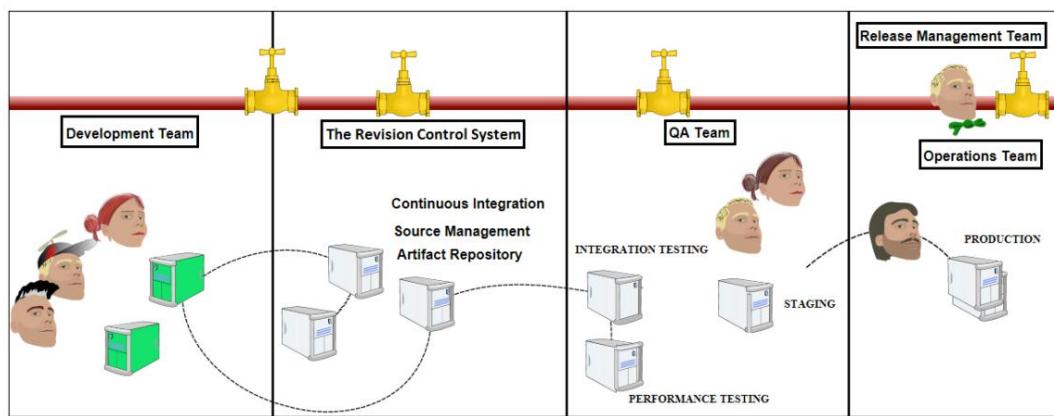
DevOps 过程和持续 交货 - 概述

以下持续交付管道的概览图片中有很多细节 ,您很可能无法阅读所有文本。暂时不要担心这个 ;随着我们的进行 ,我们将深入研究细节。

从轨道上看

就目前而言,了解当我们使用 DevOps 时,我们是在一个庞大而复杂的环境中处理大型而复杂的流程就足够了。

下图介绍了大型组织中持续交付管道的示例:



尽管这个形象的基本轮廓出人意料地经常适用,但与组织无关。当然,根据组织的规模和正在开发的产品的复杂性,存在差异。

链的早期部分,即开发人员环境和持续集成环境,通常非常相似。

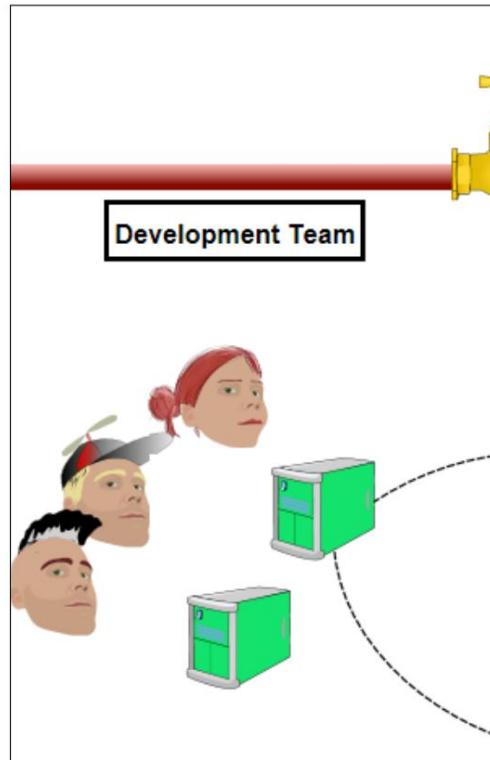
测试环境的数量和类型差异很大。生产环境也千差万别。

在以下部分中,我们将讨论持续交付管道的不同部分。

开发商

开发人员（图中最左侧）在他们的工作站上工作。
他们开发代码并需要许多工具来提高效率。

之前较大的持续交付管道概述中的以下详细信息说明了开发团队。



理想情况下，他们每个人都拥有类似生产的环境，可以在他们的工作站或笔记本电脑上本地使用。根据正在开发的软件类型，这实际上可能是可能的，但更常见的是模拟或模拟生产环境中难以复制的部分。

例如，外部支付系统或电话硬件等依赖项可能就是这种情况。

从轨道上看

当您使用 DevOps 时,您可能会或多或少地关注持续交付管道的这一部分,这取决于您在原始背景中强调的两个组成部分中的哪一个。如果您具有强大的开发人员背景,您会欣赏预先打包的开发人员环境带来的便利,并经常使用这些环境。这是一种合理的做法,否则开发人员可能会花费大量时间来创建他们的开发环境。例如,这种预先打包的环境可能包括特定版本的 Java 开发工具包和集成开发环境,例如 Eclipse。如果您使用 Python,您可能会打包特定的 Python 版本,等等。

请记住,我们本质上需要两个或多个单独维护的环境。上面的开发环境包含了我们需要的所有开发工具。这些将不会安装在测试或生产系统上。此外,开发人员还需要某种方式以类似于生产的方式部署他们的代码。

这可以是在开发人员的机器上运行 Vagrant 的虚拟机、在 AWS 上运行的云实例或 Docker 容器:有很多方法可以解决这个问题。



我个人的偏好是使用类似于生产环境的开发环境。例如,如果生产服务器运行 Red Hat Linux,开发机器可能运行 CentOS Linux 或 Fedora Linux。这很方便,因为您可以使用在本地生产中运行的许多相同软件,而且麻烦更少。使用 CentOS 或 Fedora 的妥协可能是由于 Red Hat 的许可证成本以及企业发行版通常落后于软件版本这一事实。

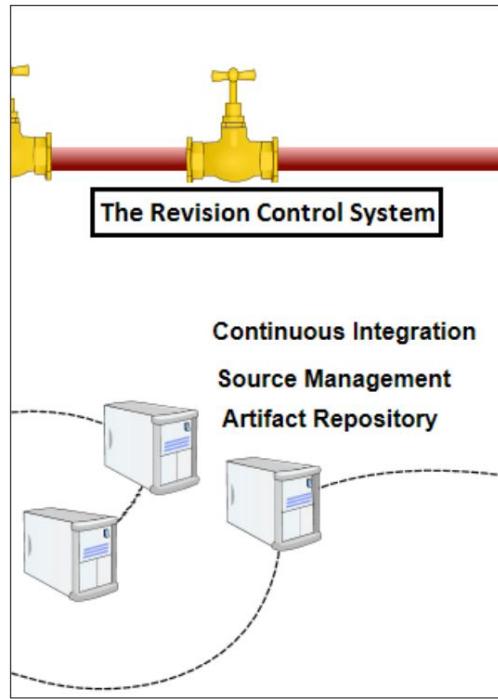
如果您在生产环境中运行 Windows 服务器,使用 Windows 开发机器可能也更方便。

修订控制系统

版本控制系统通常是开发环境的核心。

构成组织软件产品的代码存储在这里。在这里存储构成基础设施的配置也很常见。如果您从事硬件开发,设计也可能存储在版本控制系统中。

下图更详细地显示了在持续交付管道中处理代码、持续集成和工件存储的系统:



对于组织基础架构中如此重要的部分 ,产品选择几乎没有变化。如今 ,许多人使用 Git 或正在转向使用 Git ,尤其是那些使用生命周期即将结束的专有系统的人。

无论您在组织中使用何种版本控制系统 ,产品的选择只是全局的一方面。

您需要决定目录结构约定和分支策略
使用。

如果您有大量独立组件 ,您可能决定为每个组件使用一个单独的存储库。

由于版本控制系统是开发链的核心 ,它的许多细节将在第5章 “构建代码”中介绍。

构建服务器
构建服务器在概念

上很简单。它可能被视为一个美化的鸡蛋计时器 ,它可以定期或在不同的触发器上构建您的源代码。

从轨道上看

最常见的使用模式是让构建服务器监听版本控制系统中的变化。当注意到更改时,构建服务器会从版本控制系统更新其源代码的本地副本。然后,它构建源代码并执行可选测试以验证更改的质量。这个过程称为持续集成。它将在第5章“构建代码”中进行更详细的探讨。

与代码存储库的情况不同,尚未出现明显的赢家
构建服务器字段。

在本书中,我们将讨论 Jenkins,它是一种广泛用于构建服务器的开源解决方案。 Jenkins 开箱即用,为您提供简单而强大的体验。它也很容易安装。

工件存储库

当构建服务器验证了代码的质量并将其编译成可交付成果时,将编译后的二进制工件存储在存储库中是很有用的。
这通常与版本控制系统不同。

本质上,这些二进制代码存储库是可通过 HTTP 协议访问的文件系统。通常,它们提供搜索和索引以及存储元数据的功能,例如关于工件的各种类型标识符和版本信息。

在 Java 世界中,一个相当普遍的选择是 Sonatype Nexus。 Nexus 不仅限于 Java 工件,例如 Jars 或 Ears,还可以存储操作系统类型的工件,例如 RPM、适合 JavaScript 开发的工件等。

Amazon S3 是一个键值数据存储,可用于存储二进制工件。一些构建系统,例如 Atlassian Bamboo,可以使用 Amazon S3 来存储构件。S3 协议是开放的,并且有可以在您自己的网络中部署的开源实现。一种可能性是 Ceph 分布式文件系统,它提供与 S3 兼容的对象存储。

我们接下来要看的包管理器,其核心也是工件存储库。

包管理器

Linux 服务器通常采用原则上相似但在实践中有一些差异的系统进行部署。

类似 Red Hat 的系统使用一种称为 RPM 的包格式。类 Debian 系统使用.deb 格式,这是一种具有相似功能的不同包格式。然后可以使用从二进制存储库中获取它们的命令将可交付成果安装在服务器上。这些命令称为包管理器。

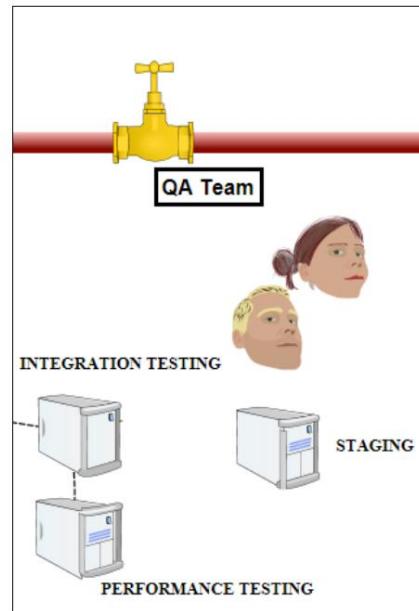
在 Red Hat 系统上,该命令称为 yum,或者最近称为 dnf。
在类 Debian 系统上,它称为 aptitude/dpkg。

这些包管理系统的最大好处是安装和升级包很容易;依赖项会自动安装。

如果您没有更高级的系统,远程登录每台服务器然后在每台服务器上键入 yum upgrade 是可行的。然后将从二进制存储库中获取并安装最新的软件包。当然,正如我们将看到的,我们确实有更先进的部署系统可用;因此,我们不需要执行手动升级。

测试环境在构建服务器将工件存储在二进制存储库中之后,可以从那里将它们安装到测试环境中。

下图更详细地显示了测试系统:



从轨道上看

测试环境通常应尽可能接近生产环境。
因此,最好使用与生产服务器相同的方法安装和配置它们。

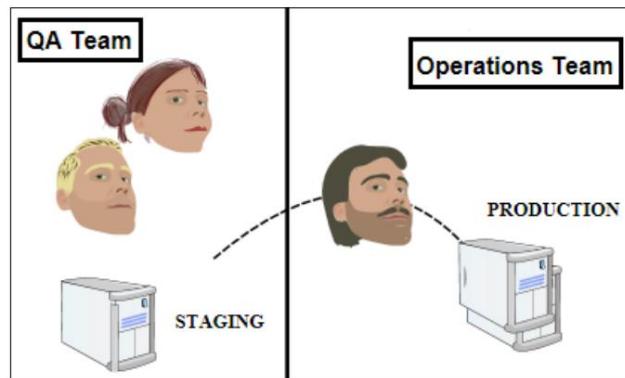
分期/生产

暂存环境是测试环境的最后一行。它们可以与生产环境互换。您在登台服务器上安装新版本,检查一切是否正常,然后换出旧的生产服务器并将其替换为登台服务器,然后这些服务器将成为新的生产服务器。这有时称为蓝绿部署策略。

如何执行这种部署方式的具体细节取决于所部署的产品。有时,不可能同时运行多个生产系统,通常是因为生产系统非常昂贵。

另一方面,我们可能在一个池中有数百个生产系统。然后我们可以逐步推出池中的新版本。已登录的用户将继续使用在他们登录的服务器上运行的版本。新用户登录到运行新版本软件的服务器。

更大的持续交付图像中的以下详细信息显示了涉及的最终系统和角色：



并非所有组织都有资源来维护生产质量的登台服务器,但在可能的情况下,这是处理升级的一种不错且安全的方法。

发布管理

到目前为止,我们假设发布过程大部分是自动的。这是使用 DevOps 的人梦寐以求的场景。

这个梦想场景是在现实世界中实现的挑战。原因之一是通常很难达到对自动化部署充满信心所需的测试自动化水平。另一个原因很简单,业务开发的节奏并不总是与技术开发的节奏相匹配。因此,有必要在发布过程中启用人为干预。

下图中使用水龙头来象征人际互动 在本例中,由专门的发布经理进行。



在实践中如何做到这一点各不相同,但部署系统通常有一种方法来支持如何描述在不同环境中使用哪些软件版本。

然后可以将集成测试环境设置为使用已部署到二进制构件存储库的最新版本。登台和生产服务器具有经过质量保证团队测试的特定版本。

Scrum、看板和交付管道

我们在本章中描述的持续交付管道如何支持 Scrum 和看板等敏捷流程?

Scrum 专注于冲刺周期,可以每两周或每月发生一次。可以说看板更侧重于更短的周期,每天都可能发生。

Scrum 和看板之间的哲学差异更深一些,但并不相互排斥。许多组织同时使用看板和 Scrum。

从轨道上看

从软件部署的角度来看,Scrum 和看板都很相似。两者都需要频繁的无障碍部署。从 DevOps 的角度来看,变更开始通过持续交付管道传播到测试系统,当它被认为足以开始该旅程时,就会传播到更远的地方。这可以根据主观测量或客观测量来判断,例如“所有单元测试都是绿色的”。

我们的管道可以管理以下两种类型的场景:

- 构建服务器支持生成我们做出决策所需的客观代码质量指标。这些决定可以自动做出,也可以作为人工决定的基础。
- 部署管道也可以手动定向。这可以通过问题管理系统、配置代码提交或两者来处理。

因此,再次从 DevOps 的角度来看,我们是否使用 Scrum、Scaled Agile Framework、Kanban 或精益或敏捷框架中的其他方法并不重要。即使是传统的瀑布流程也可以成功管理 DevOps 服务于所有人!

总结 一个完整的例子

到目前为止,我们已经粗略地涵盖了很多信息。

为了更清楚地说明这一点,让我们看一下具体更改在通过系统传播时会发生什么,使用示例:

- 开发团队被赋予了对组织系统进行更改的责任。更改围绕向身份验证系统添加新角色。这个看似简单的任务实际上很难,因为许多不同的系统都会受到变化的影响。
 - 为了让生活更轻松,决定将更改分解为几个更小的更改,这些更改将独立进行测试,并且主要由自动回归测试自动进行。
 - 第一个变化是向身份验证系统添加一个新角色,它是在开发人员机器上本地开发的,并尽最大努力进行本地测试。要真正知道它是否有效,开发人员需要访问在他或她的本地环境中不可用的系统;在本例中,是包含用户信息和角色的 LDAP 服务器。
-
- 如果使用测试驱动开发,甚至在编写任何实际代码之前就编写了一个失败的测试。在编写失败的测试后,编写使测试通过的新代码。

- 开发人员签入对组织修订控制的更改系统,一个 Git 存储库。
- 构建服务器获取更改并启动构建过程。单元测试后,更改被认为足够适合部署到二进制存储库,这是一个 Nexus 安装。
- 配置管理系统 Puppet 注意到有一个新的可用的身份验证组件的版本。集成测试服务器被描述为需要安装最新版本,因此 Puppet 继续安装新组件。
- 新组件的安装现在会触发自动回归测试。当这些都成功完成后,由质量人员进行手动测试保证小组开始。
- 质量保证团队批准变更。更改转移到登台服务器,最终验收测试开始。 · 验收测试阶段完成后,登台服务器切换到生产服务器,生产服务器成为新的登台服务器。

最后一步由组织的负载平衡服务器管理。

然后根据需要重复该过程。如您所见,发生了很多事情!

识别瓶颈

从前面的示例中可以明显看出,通过从开发到生产的管道传播的任何更改都会发生很多事情。重要的是这个过程要有效率。

与所有敏捷工作一样,跟踪您正在做的事情,并尝试找出问题区域。

当一切正常时,对代码存储库的提交应该会导致更改在 15 分钟的时间跨度内部署到集成测试服务器。

当事情运行不佳时,部署可能需要几天的时间来解决意想不到的麻烦。
以下是一些可能的原因:

- 数据库架构更改。 · 测试数据不符合预期。 · 部署取决于人员,并且该人员不可用。

从轨道上看

- 有与传播变化相关的不必要的繁文缛节。 · 您的更改不小,因此需要大量工作才能安全部署。这可能是因为您的架构基本上是一个整体。

我们将在接下来的章节中进一步研究这些挑战。

概括

在本章中,我们进一步研究了您在进行 DevOps 工作时通常使用的不同类型的系统和流程。我们对处于 DevOps 核心的持续交付过程有了更深入、更详细的了解。

接下来,我们将研究 DevOps 思维方式如何影响软件架构,以帮助我们实现更快、更精确的交付。

3

DevOps 如何影响 建筑学

软件架构是一个庞大的主题，在本书中，我们将重点关注对持续交付和 DevOps 影响最大的架构方面，反之亦然。

在本章中，我们将看到：

- 软件架构的各个方面及其对我们使用 DevOps 眼镜时的意义
- 基本术语和目标
- 反模式，例如单体
- 关注点分离的基本原则
- 三层系统和微服务

最后，我们总结了一些有关数据库迁移的实际问题。

数量不多，让我们开始吧！

介绍软件架构

我们将讨论 DevOps 如何影响我们应用程序的架构，而不是我们在本书其他地方讨论的软件部署系统的架构。

通常在讨论软件架构时，我们会想到软件的非功能需求。非功能性需求是指软件的不同特性，而不是对特定行为的需求。

DevOps 如何影响架构

功能需求可能是我们的系统应该能够处理信用卡交易。一个非功能性需求可能是系统应该能够每秒管理几个这样的信用卡交易。

以下是 DevOps 和 Continuous 的两个非功能性需求
软件架构交付地点：

- 我们需要能够经常部署小的更改 · 我们需要能够对我们的
更改质量充满信心

正常情况应该是我们能够在很短的时间内从开发人员的机器到生产环境部署小的改动。由于更改引起的意外问题而回滚更改应该很少发生。

那么,如果我们暂时将部署系统从等式中剔除,我们部署的软件系统的架构将受到怎样的影响?

单体场景理解有问题的架构可能导致持续交付问题的一种
方法是暂时考虑一个反例。

假设我们有一个具有许多不同功能的大型 Web 应用程序。
我们在应用程序中还有一个静态网站。整个 Web 应用程序部署为单个 Java EE 应用程序存档。
因此,当我们需要修复静态网站中的拼写错误时,我们需要重建整个 Web 应用程序存档并重新部署。

虽然这可能被视为一个愚蠢的例子,而且开明的读者永远不会做这样的事情,但我已经在现实世界中看到了这种反模式。作为 DevOps 工程师,这可能是我们可能需要解决的实际情况。

让我们分解一下这种纠缠不清的问题的后果。当我们想要纠正拼写错误时会发生什么?让我们来看看:

1. 我们知道我们要纠正哪个拼写错误,但是我们需要在哪个代码库中做呢?由于我们有一个整体,我们需要在我们的代码库的版本控制系统中创建一个分支。这个新分支对应于我们在生产环境中运行的代码。
2. 做分支,改正拼写错误。
3. 建立一个带有修正的新神器。给它一个新版本。
4. 将新工件部署到生产中。

好吧,乍一看这似乎还不算太糟糕。但也要考虑以下几点:

- 我们对包含整个业务关键系统的整体进行了更改。如果在我们部署新版本时出现问题,我们会在一分钟内损失收入。我们真的确定更改不会影响其他任何内容吗?
- 事实证明,我们并没有真正将更改仅限于更正拼写错误。在制作新工件时,我们还更改了许多版本字符串。但是更改版本字符串也应该是安全的,对吧?我们确定吗?

这里的要点是,我们已经花费了相当多的精力来确保更改确实是安全的。系统是如此复杂,以至于很难考虑变化的影响,即使它们可能微不足道。

现在,更改通常比简单的拼写更正更复杂。因此,我们需要对单体应用的所有更改执行部署链的所有方面,包括手动验证。

我们现在处在一个我们不愿去的地方。

体系结构经验法则有许多体系结构规则可以帮助我们了解如何处理以前的不良情况。

关注点分离

著名的荷兰计算机科学家 Edsger Dijkstra 在 1974 年的论文《科学思维的作用》中首次提到了他关于如何有效组织思维的想法。

他称这个想法为“关注点分离”。迄今为止,它可以说是软件设计中最重要的规则。还有许多其他众所周知的规则,但其中许多规则都遵循关注点分离的思想。基本原则很简单,我们应该分别考虑系统的不同方面。

凝聚力原则

在计算机科学中,内聚性是指软件模块的元素属于在一起的程度。

内聚性可以用来衡量模块中功能的相关程度。

一个模块最好有很强的内聚力。

我们可以看到,强内聚是分离原则的另一个方面的担忧。

耦合

耦合是指两个模块之间的依赖程度。我们总是希望模块之间的耦合度低。

同样,我们可以将耦合视为关注点分离原则的另一个方面。

高内聚低耦合的系统会自动分离关注点,反之亦然。

回到单体场景在前面的拼写更正场景中,很明显我们在关注点分离方面失败了。我们根本没有进行任何模块化,至少从部署的角度来看是这样。该系统似乎具有低内聚和高耦合的不良特征。

如果我们有一组单独的部署模块,我们的拼写更正很可能只会影响一个模块。更明显的是,部署更改是安全的。

当然,这在实践中应该如何实现会有所不同。在此特定示例中,拼写更正可能属于前端 Web 组件。最起码,这个前端组件可以和后端组件分开部署,有自己的生命周期。

但在现实世界中,我们可能不会幸运地总是能够影响我们工作所在组织使用的不同技术。

例如,前端可以使用具有其自身怪癖的专有内容管理系统来实现。如果遇到这种情况,明智的做法是跟踪此类系统造成成本。

一个实际例子

现在让我们看一下我们将在本书的其余部分处理的具体示例。在我们的例子中,我们为一个名为 Matangle 的组织工作。

该组织是一家软件即服务 (SaaS) 提供商,向学童销售教育游戏。

与任何此类提供商一样,我们很有可能拥有一个包含客户信息的数据库。我们将从这个数据库开始。

该组织的其他系统将随着我们的进展而充实,但这个初始系统非常适合我们的目的。

三层系统

Matangle 客户数据库是一个非常典型的三层、CRUD (创建、读取、更新和删除)类型的系统。这种软件架构模式已经使用了几十年,并且继续流行。这些类型的系统非常常见,您很可能会遇到它们,无论是遗留系统还是新设计。

在此图中,我们可以看到关注点分离的实际应用:



DevOps 如何影响架构

下面列出的三个层显示了我们的组织如何选择构建其系统的示例。

表示层

表示层将是一个使用 React Web 框架实现的 Web 前端。它将作为一组 JavaScript 和静态 HTML 文件进行部署。React 框架是相当新的。您的组织可能不使用 React，而是使用其他一些框架，例如 Angular。无论如何，从部署和构建的角度来看，大多数 JavaScript 框架都是相似的。

逻辑层

逻辑层是在 Java 平台上使用 Clojure 语言实现的后端。Java 平台在大型组织中非常普遍，而小型组织可能更喜欢基于 Ruby 或 Python 的其他平台。我们的示例基于 Clojure，包含了两个世界的一点点。

数据层在我们的例子中，数据库是

用 PostgreSQL 数据库系统实现的。

PostgreSQL 是一个关系数据库管理系统。虽然可以说不像 MySQL 安装那么普遍，但大型企业可能更喜欢 Oracle 数据库。

无论如何，PostgreSQL 都是一个健壮的系统，因此我们的示例组织选择了 PostgreSQL。

从 DevOps 的角度来看，三层模式看起来很有吸引力，至少表面上是这样。应该可以将更改分别部署到三个层中的每一层，这将使通过以下方式传播小更改变得简单

服务器。

但在实践中，数据层和逻辑层通常是紧密耦合的。表示层和逻辑层也是如此。为了避免这种情况，必须注意保持层之间的接口精简。使用众所周知的模式不一定是灵丹妙药。如果我们在设计系统时不小心，我们仍然会得到一个不受欢迎的单体系统。

处理数据库迁移

处理关系数据库中的更改需要特别考虑。

关系数据库存储数据和数据的结构。与升级程序二进制文件时出现的挑战相比,升级数据库模式会带来其他挑战。通常,当我们升级应用程序二进制文件时,我们会停止应用程序,升级它,然后再次启动它。我们并不真正关心应用程序状态。这是在应用程序之外处理的。

在升级数据库时,我们确实需要考虑状态,因为数据库包含的逻辑和结构相对较少,但包含的状态却很多。

为了描述数据库结构的变化,我们发出一个命令来改变结构。

应用更改前后的数据库结构应视为数据库的不同版本。我们如何跟踪数据库版本?



Liquibase 是一个数据库迁移系统,其核心是使用久经考验的方法。有很多这样的系统,通常每种编程语言至少有一个。 Liquibase 在 Java 世界中是众所周知的,甚至在 Java 世界中,也有几个以类似方式工作的替代品。 Flyway 是 Java 平台的另一个例子。

通常,数据库迁移系统采用以下方法的一些变体:

- 向存储数据库版本的数据库添加一个表。
- 跟踪数据库更改命令并将它们集中在一起
版本化的变更集。对于 Liquibase,这些更改存储在 XML 文件中。 Flyway 采用了一种略有不同的方法,其中变更集作为单独的 SQL 文件处理,或者偶尔作为单独的 Java 类处理,以进行更复杂的转换。
- 当 Liquibase 被要求升级数据库时,它会查看元数据表并确定要运行哪些变更集以使数据库与最新版本保持同步。

如前所述,许多数据库版本管理系统都是这样工作的。

它们的主要区别在于变更集的存储方式以及它们如何确定要运行的变更集。它们可能存储在一个 XML 文件中,例如 Liquibase,或者作为一组单独的 SQL 文件,例如 Flyway。后一种方法在本地系统中更为常见,并且具有一些优势。 Clojure 生态系统也至少有自己的一个类似的数据库迁移系统,称为 Migratus。

滚动升级

进行数据库迁移时要考虑的另一件事是所谓的滚动升级。当您不希望您的最终用户遇到任何停机时间或至少很少的停机时间时,这些类型的部署很常见。

下面是我们组织的客户数据库滚动升级的示例。

当我们开始时,我们有一个正在运行的系统,其中包含一个数据库和两台服务器。我们在两台服务器前面有一个负载均衡器。

我们将推出对数据库架构的更改,这也会影响服务器。我们将把数据库中的客户姓名字段拆分为两个单独的字段,名字和姓氏。

这是一个不兼容的更改。我们如何最大限度地减少停机时间?让我们看看解决方案:

1. 我们首先进行数据库迁移,创建两个新的名称字段,然后通过获取旧名称字段并通过在名称中找到空格字符将字段分成两半来填充这些新字段。

这是最初选择的名称编码,不是很好。这就是我们想要改变它的原因。

此更改到目前为止是向后兼容的,因为我们没有删除 name 字段;我们刚刚创建了两个到目前为止未使用的新字段。

2. 接下来,我们更改负载均衡器配置,使我们的两台服务器中的第二台不再可从外界访问。第一个服务器运行得很愉快,因为旧的名称字段仍然可以被旧的服务器代码访问。

3. 现在我们可以自由升级服务器二了,因为没有人使用它。

升级后,我们启动它,它也很高兴,因为它使用了两个新的数据库字段。

4. 此时,我们可以再次切换负载均衡器配置,使服务器一不可用,让服务器二联机。我们在服务器 1 离线时进行相同类型的升级。我们启动它,现在通过恢复我们原来的负载均衡器配置使两台服务器再次可访问。

现在,更改几乎已完全部署。唯一剩下的就是删除旧名称字段,因为没有服务器代码再使用它。

正如我们所见,滚动升级需要大量的提前工作才能正常运行。如果您的组织有的话,在自然停机期间安排升级要容易得多。国际组织可能没有任何合适的自然窗口来执行升级,然后滚动升级可能是唯一的选择。

Liquibase 中的你好世界

这是 Liquibase 关系数据库变更集处理程序的简单“hello world”样式示例。

要试用该示例,请解压缩源代码包并运行 Java 构建工具 Maven。

```
cd ch3-liquibase-helloworld  
mvn liquibase:更新
```

变更日志文件

以下是 Liquibase 变更日志文件的示例。

它使用数字标识符 1 和 2 定义了两个变更集或迁移:

- 变更集 1 创建一个名为 customer 的表,其中包含一个名为 name 的列
- 变更集 2 在名为 customer 的表中添加一个名为 address 的列

```
<?xml version= 1.0 encoding= utf-8 ?><databaseChangeLog  
xmlns= http://www.liquibase.org/xml/ns/dbchangelog xmlns:xsi= http://www.w3.org/2001/XMLSchema-  
instance xsi:schemaLocation= http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/  
xml/ns/dbchangelog/dbchangelog-3.0.xsd >
```

```
<changeSet id= 1 author= jave >  
  
<createTable 表名= 客户 >  
    <列名= id type= int /><列名= name  
        type= varchar(50) />  
</createTable> </  
changeSet>
```

DevOps 如何影响架构

```

<changeSet id= 2 author= jave >
    <addColumn tableName= 客户 >
        <column name= address type= varchar(255) />
    </addColumn> </
changeSet>

</databaseChangeLog>

```

pom.xml 文件

pom.xml文件是一个标准的Maven 项目模型文件,它定义了我们需要的 JDBC URL 等内容,以便我们可以连接到我们希望使用的数据库以及 Liquibase 插件的版本。

将创建H2 数据库文件/tmp/liquidhello.h2.db 。 H2是一个方便的内存数据库,适合测试。

这是 “liquibase hello world”示例的 pom.xml 文件:

```

<?xml version= 1.0 encoding= utf-8 ?><project xmlns= http://
maven.apache.org/POM/4.0.0 xmlns:xsi= http://www.w3.org/2001/XMLSchema-instance xsi:schemaLocation= http://
maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd >

<modelVersion>4.0.0</modelVersion>
<groupId>se.verona.liquibasehello</groupId><artifactId>liquibasehello</
artifactId> <version>1.0-SNAPSHOT</version> <build>

<插件> <插件>

    <groupId>org.liquibase</groupId> <artifactId>liquibase-maven-
    plugin</artifactId> <version>3.0.0-rc1</version> <configuration> <changeLogFile>src/
    main/资源/db-changelog.xml </changeLogFile> <driver>org.h2.Driver</driver>
    <url>jdbc:h2:liquidhello</url>

</配置> <依赖项> <依赖项>

    <groupId>com.h2database</groupId> <artifactId>h2</
    artifactId> <version>1.3.171</version>

</依赖>

```

```
</dependencies> </
plugin> </plugins> </build>
</project>
```

如果您运行代码并且一切正常,结果将是一个 H2 数据库文件。 H2 有一个简单的 Web 界面,您可以在其中验证数据库结构确实是您所期望的。

手动安装

在我们可以自动化一些东西之前,我们需要了解相应的手动过程。

在本书中,假设我们使用的是基于 Red Hat 的 Linux 发行版,例如 Fedora 或 CentOS。大多数 Linux 发行版在原则上都是相似的,除了用于包操作的命令集可能会有所不同。

对于练习,您可以使用物理服务器或安装在 VirtualBox 中的虚拟机。

首先我们需要 PostgreSQL 关系数据库。使用此命令:

```
dnf安装postgresql
```

这将检查是否已经安装了 PostgreSQL 服务器。否则,它将从远程 yum 存储库中获取 PostgreSQL 软件包并进行安装。因此,经过反思,许多可能涉及的手动步骤已经自动化。我们不需要编译软件,检查软件版本,安装依赖等等。所有这些都已经在 Fedora 项目的构建服务器上提前完成了,非常方便。

但是,对于我们自己组织的软件,我们最终需要自己模拟这种行为。

同样,我们还需要一个 Web 服务器,在本例中为 NGINX。要安装它,请使用以下命令:

```
dnf安装nginx
```

dnf命令取代了 Red Hat 派生发行版中的yum。它是对yum的兼容重写,保留了相同的命令行界面。

[DevOps 如何影响架构](#)

我们正在部署的软件,即 Matangle 客户关系数据库,在技术上不需要单独的数据库和网络服务器。名为 HTTP Kit 的 Web 服务器已包含在该软件的 Clojure 层中。

通常,在基于 Java、Python 等构建的服务器前面使用专用 Web 服务器。再次,主要原因是关注点分离问题;这次不是为了逻辑分离,而是为了性能、负载均衡、安全等非功能性需求。如今,基于 Java 的 Web 服务器可能完全能够提供静态内容,但基于 C 的原生 Web 服务器(如 Apache httpd 或 NGINX)仍然具有卓越的性能和更适中的内存需求。例如,使用前端 Web 服务器进行 SSL 加速或负载平衡也很常见。

现在,我们有一个数据库和一个网络服务器。此时,我们需要构建和部署我们组织的应用程序。

在您的开发机器上,在本书的解压源存档目录中执行以下步骤:

CD 频道 3/crm1

莱恩建立

我们现在已经生成了一个可用于部署和运行应用程序的 Java 存档。

试用应用程序:

莱恩奔跑

将浏览器指向控制台中显示的 URL 以查看 Web 用户界面。

我们如何在我们的服务器上正确部署应用程序?如果我们可以使用与安装数据库和网络服务器时相同的命令和机制,那就太好了。我们将在第7章“部署代码”中看到如何做到这一点。现在,只需从 shell 运行应用程序就足够了。

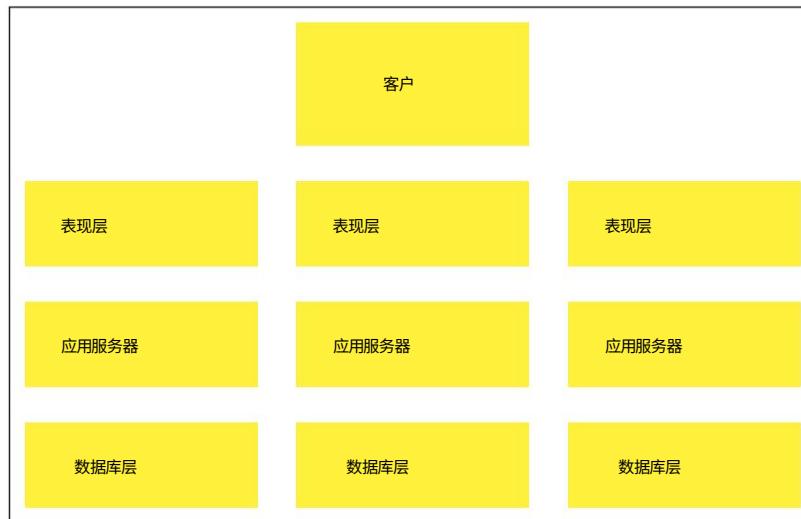
微服务微服务是一个最近的术语,

用于描述三层模式的逻辑层由几个较小的服务组成的系统,这些服务与语言无关的协议进行通信。

通常,与语言无关的协议是基于 HTTP 的,通常是 JSON REST,但这不是强制性的。协议层有几种可能性。

这种架构设计模式非常适合持续交付方法,因为正如我们所见,部署一组较小的独立服务比整体式服务更容易。

下面是微服务部署的示例:



随着我们的进行,我们将把我们的例子发展到微服务架构。

插曲 康威定律

1968 年,梅尔文·康威 (Melvin Conway) 提出了这样一种观点,即设计软件的组织结构最终会复制到软件的组织中。这称为康威定律。

例如,三层模式反映了许多组织的方式

IT 部门的结构:

- 数据库管理员团队,或简称 DBA 团队
- 后端开发团队 · 前端开发团队 ·
- 运营团队

好吧,这就是四个团队,但我们可以清楚地看到架构模式和组织之间的相似之处。

DevOps 如何影响架构

DevOps 的主要目标是将不同的角色聚集在一起,最好是在跨职能团队中。如果康威定律成立,这些团队的组织将反映在他们的设计中。

微服务模式恰好非常接近地反映了一个跨职能团队。

如何保持服务接口向前兼容

必须允许服务接口随时间发展。这是很自然的,因为组织的需求会随着时间的推移而变化,而服务接口在一定程度上反映了这一点。

我们怎样才能做到这一点?一种方法是使用有时称为Tolerant Reader 的模式。这只是意味着服务的消费者应该忽略它无法识别的数据。

这是一种非常适合 REST 实现的方法。

SOAP 是一种基于 XML 模式的定义服务的方法,更加严格。使用 SOAP,您通常不需要更改现有接口。接口被视为应该保持不变的契约。

 您无需更改接口,而是定义一个新的模式版本。现有的消费者要么必须实施新协议然后重新部署,要么生产者需要并行提供多个版本化的端点。这很麻烦,并且会在提供者和消费者之间产生不希望的更紧密的耦合。

虽然 DevOps 和持续交付的想法实际上并没有在强制要求如何完成事情方面做太多事情,但通常更喜欢最有效的方法。

在我们的例子中,可以说成本最低的方法是将实施变更的负担分摊到生产者和消费者身上。生产者无论如何都需要改变,而消费者需要接受实施 Tolerant Reader 模式的一次性成本。使用 SOAP 和 XML 可以做到这一点,但不如使用 REST 实现自然。这是 REST 实施在采用 DevOps 和持续交付的组织中更受欢迎的原因之一。

如何在实践中实现 Tolerant Reader 模式因所使用的平台而异。对于 JsonRest,通常将 JSON 结构解析为等效的特定于语言的结构就足够了。然后,您可以选择应用程序所需的部件。所有其他部分,无论是旧的还是新的,都将被忽略。这种方法的局限性在于生产者不能删除消费者所依赖的结构部分。添加新部件是可以的,因为它们将被忽略。

这再次给生产者带来了跟踪消费者意愿的负担。

在组织内部,这不一定是个大问题。生产者可以跟踪消费者最新阅读器代码的副本,并测试它们在生产者的构建阶段是否仍然有效。

对于广泛暴露于 Internet 的服务,这种方法实际上并不适用,在这种情况下,更严格的 SOAP 方法更为可取。

微服务和数据层

查看微服务的一种方式是,每个微服务都可能是一个独立的三层系统。不过,我们通常不会为每个微服务实现每一层。考虑到这一点,我们看到每个微服务都可以实现自己的数据层。好处是可能增加服务之间的分离。



不过,根据我的经验,更常见的做法是将组织的所有数据放入一个数据库或至少一个数据库类型中。这更常见,但不一定更好。

两种情况各有利弊。当系统彼此明显分开时,部署更改会更容易。另一方面,当一切都存储在同一个数据库中时,数据建模会更容易。

DevOps、架构和弹性

我们已经看到,从 DevOps 的角度来看,微服务架构具有许多理想的属性。DevOps 的一个重要目标是更快地将新功能交付给我们的用户。这是微服务提供的更多模块化的结果。

不过,那些担心微服务会通过提供没有缺点的完美解决方案而让生活变得无趣的人可以松一口气了。微服务确实提供了自己的挑战。

DevOps 如何影响架构

我们希望能够快速部署新代码,但我们也希望我们的软件可靠。

微服务在系统之间有更多的集成点,并且比单体系统更容易出现故障。

自动化测试对于 DevOps 非常重要,因此我们部署的更改质量好并且可以信赖。但是,这不是解决服务因其他原因突然停止工作的问题的解决方案。由于我们有更多使用微服务模式运行的服务,因此在统计上服务失败的可能性更大。

我们可以通过努力监控服务并在出现故障时采取适当的措施来部分缓解这个问题。这最好是自动化的。

在我们的客户数据库示例中,我们可以采用以下策略:

- 我们使用两个都运行我们的应用程序的应用程序服务器 · 该应用程序通过 JsonRest 提供一个特殊的监控界面 · 一个监控守护进程定期轮询这个监控界面
- 如果服务器停止工作,则重新配置负载平衡器,以便将有问题的服务器从服务器池中移除

这显然是一个简单的例子,但它可以说明我们在设计包含许多移动部件的弹性系统时所面临的挑战,以及它们可能如何影响架构决策。

为什么我们要提供我们自己的特定于应用程序的监控界面?由于监控的目的是让我们全面了解系统当前的健康状况,因此我们通常会监控应用程序堆栈的许多方面。

我们监控服务器 CPU 是否过载,是否有足够的可用磁盘和内存空间,以及基础应用程序服务器是否正在运行。但是,这可能不足以确定服务是否正常运行。

例如,该服务可能由于某种原因具有损坏的数据库访问配置。特定于服务的健康探测接口将尝试联系数据库并在返回结构中返回连接状态。

当然,如果您的组织可以就探测返回结构的通用格式达成一致是最好的。该结构还将取决于所使用的监控软件的类型。

概括

在本章中,我们从 DevOps 和持续交付的角度审视了软件架构这个庞大的主题。

我们了解了关注点分离规则可以采用的许多不同面孔。我们还开始为组织内的一个组件制定部署策略,即 Matangle 客户数据库。

我们深入研究了细节,例如如何从存储库安装软件以及如何管理数据库更改。我们还研究了高级主题,例如经典的三层系统和最近的微服务概念。

我们的下一站将处理如何管理源代码和建立源代码修订控制系统。

Machine Translated by Google

4

一切都是代码

一切都是代码,您需要在某个地方存储它。组织的源代码管理系统就是那个地方。

开发人员和运营人员为其不同类型的代码共享同一个中央存储。

托管中央代码存储库的方式有很多种:

- 您可以使用软件即服务解决方案,例如GitHub、Bitbucket 或GitLab。这可能具有成本效益并提供良好的可用性。
- 您可以使用AWS 或Rackspace 等云提供商来托管您的存储库。

某些类型的组织根本无法让他们的代码离开大楼。对他们来说,私人内部设置是最佳选择。

在本章中,我们将探索不同的选项,例如 Git,以及基于 Web 的 Git 前端,例如 Gerrit 和 GitLab。

此探索将帮助您找到满足您组织需求的 Git 托管解决方案。

在本章中,我们将开始体验 DevOps 领域的挑战之一 有太多的解决方案可供选择和探索!对于 DevOps 世界的核心源代码管理领域尤其如此。

因此,我们也将从用户的角度介绍软件虚拟化工具Docker,以便我们在探索中使用该工具。

一切都是代码

源代码控制的必要性美国作家特伦斯麦肯纳曾经说过,一切都是代码。

虽然人们可能不同意 McKenna 关于宇宙中的一切是否都可以表示为代码的观点,但出于 DevOps 的目的,实际上几乎一切都可以以编码形式表达,包括以下内容:

- 我们构建的应用程序 · 托管我们应用
用程序的基础设施 · 记录我们产品的文档

甚至运行我们应用程序的硬件也可以用软件的形式来表达。

鉴于代码的重要性,我们放置代码的位置 (源代码存储库)自然成为我们组织的核心。我们生产的几乎所有东西都会在其生命周期的某个时刻通过代码存储库。

源代码管理的历史

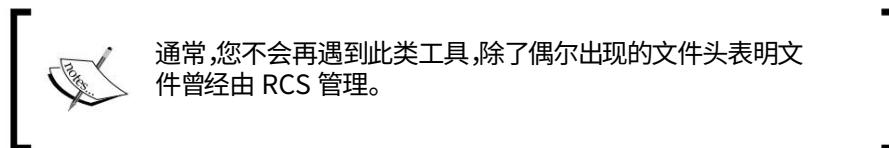
为了理解对源代码控制的核心需求,简要回顾一下源代码管理的发展历史会很有启发。这让我们深入了解我们自己可能需要的功能。一些例子如下:

- 将源代码的历史版本存储在单独的档案中。

这是最简单的形式,在某种程度上它仍然存在,许多免费软件项目提供旧版本的 tar 存档以供下载。

- 带有签入和签出功能的集中源代码管理。

在某些系统中,开发人员可以锁定文件以供独占使用。每个文件都是单独管理的。此类工具包括 RCS 和 SCCS。



- 一个集中存储,您可以在提交前合并。例子包括
并发版本系统(CVS)和Subversion。

特别是 Subversion 仍在大量使用。许多组织都有集中的工作流,而 Subversion 为他们很好地实现了这样的工作流。

- 去中心化商店。

在进化阶梯的每一步,我们都获得了更多的灵活性和并发性,以及更快、更高效的工作流程。我们还获得了更先进、更强大的枪支来射击自己的脚,我们需要牢记这一点!

目前,Git 是此类中最流行的工具,但还有许多其他类似的工具在使用,例如 Bazaar 和 Mercurial。

时间会证明 Git 及其底层数据模型是否会击退争夺源代码管理宝座的竞争者,这些竞争者无疑将在未来几年表现出来。

角色和代码从 DevOps 的角度来看,

利用源代码管理工具的自然交汇点很重要。许多不同的角色都可以在更广泛的意义上使用源代码管理。对于具有技术头脑的角色来说这样做更容易,但对于其他角色来说更难,例如项目管理。

开发人员与源代码管理息息相关。这是他们的面包和黄油。

运营人员也喜欢以代码、脚本和其他工件的形式管理基础设施的描述,正如我们将在接下来的章节中看到的那样。

此类基础结构描述符包括网络拓扑、应安装在特定服务器上的软件版本等。

质量保证人员可以将他们的自动化测试以编码形式存储在源代码存储库中。 Selenium 和 Junit 等软件测试框架就是如此。

但是,执行各种任务所需的手动步骤的文档存在问题。这与其说是技术问题,不如说是心理或文化问题。

虽然许多组织采用 wiki 解决方案,例如支持 Wikipedia 的 wiki 引擎,但共享驱动器和电子邮件中仍然有大量 Word 格式的文档。

一切都是代码

这使得某些角色很难找到和使用文档,而对其他角色来说则很容易。从 DevOps 的角度来看,这是令人遗憾的,应该努力让所有角色都能对组织中的文档进行良好和有用的访问。

根据所使用的 wiki 引擎,可以将所有 wiki 格式的文档存储在中央源代码存储库中。

哪个源代码管理系统?

那里有许多源代码管理 (SCM) 系统,由于 SCM 是开发的重要组成部分,因此这些系统的开发将继续进行。

然而,目前有一个占主导地位的系统,这个系统就是 Git。

Git 有一个有趣的故事:它是由 Linus Torvalds 发起的,目的是将 Linux 内核开发从 BitKeeper 转移出去,BitKeeper 是当时使用的专有系统。 BitKeeper 的许可证已更改,因此不再将其用于内核是不切实际的。

因此,Git 支持相当复杂的 Linux 内核开发工作流程,并且在基础技术层面上对大多数组织来说已经足够好了。

Git 与旧系统相比的主要优势在于它是一个分布式版本控制系统 (DVCS)。还有许多其他分布式版本控制系统,但 Git 是最普遍的一种。

分布式版本控制系统有几个优点,包括但不限于以下几点:

- 即使未连接到网络,也可以有效地使用 DVCS。您可以在乘坐火车或洲际航班时随身携带您的工作。
- 由于您不需要为每个操作都连接到服务器,因此在大多数情况下,DVCS 可能比替代方案更快。
- 您可以私下工作,直到您觉得您的工作已经准备就绪被分享。
- 可以同时使用多个远程登录,从而避免单点故障。

除 Git 之外的其他分布式版本控制系统包括：

- Bazaar：缩写为 bzr。Bazaar 得到 Canonical 的认可和支持，Canonical 是 Ubuntu 背后的公司。Launchpad 是 Canonical 的代码托管服务，支持 Bazaar。
- Mercurial：著名的开源项目，如 Firefox 和 OpenJDK 使用水银。它与 Git 大约同时出现。

Git 可能很复杂，但它通过快速和高效弥补了这一点。它可能很难理解，但可以通过使用支持不同任务的前端来变得更容易。

关于源代码管理系统迁移的一句话

我曾使用过许多源代码管理系统，也经历过从一种系统到另一种系统的多次转换。

有时，在执行迁移时，会花费大量时间来保持所有历史记录的完整性。对于某些系统来说，这种努力是值得的，例如古老的免费或开源项目。

对于许多组织而言，保留历史并不值得花费大量时间和精力。如果在某个时候需要旧版本，旧的源代码管理系统可以保持在线和参考。这包括从 Visual SourceSafe 和 ClearCase 的迁移。

不过有些迁移是微不足道的，例如从 Subversion 迁移到 Git。在这些情况下，不必牺牲历史准确性。

选择分支策略

使用部署到服务器的代码时，在整个组织内就分支策略达成一致非常重要。

分支策略是一种约定或一组规则，它描述了何时创建分支、如何命名它们、分支应该有什么用途等等。

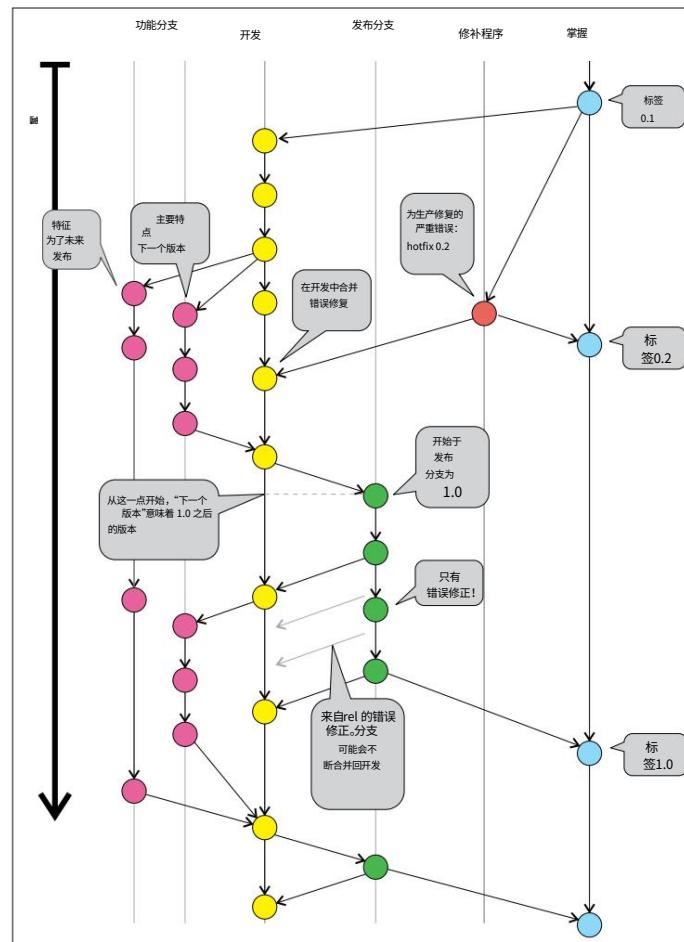
分支策略在与其他人一起工作时很重要，而在某种程度上，当你独自工作时，分支策略就不那么重要了，但它们仍然有一个目的。

一切都是代码

大多数源代码管理系统都没有规定特定的分支策略,Git 也没有。 SCM 只是为您提供了执行分支的基本机制。

使用 Git 和其他分布式版本控制系统,在本地使用功能分支通常非常便宜。功能或主题分支是用于跟踪有关特定功能、错误等的正在进行的开发的分支。这样,可以一起处理有关该功能的代码中的所有更改。

有许多众所周知的分支策略。 Vincent Driessen 形式化了一个名为Git flow 的分支策略,它有很多好的特性。对于某些人来说,Git 流程过于复杂,在这些情况下,它可以按比例缩小。有许多这样的缩小模型可用。这就是 Git 流的样子:



Git 流程看起来很复杂,那么让我们简单了解一下分支是干什么用的:

- master 分支只包含完成的工作。所有提交都被标记,因为它们代表发布。所有版本都来自 master。开发分支是开发下一个版本的地方。工作时到这里就完成了, develop 合并到 master。
- 我们为所有新功能使用单独的功能分支。功能分支是合发展。
- 当生产中出现毁灭性错误时,会创建一个修补程序分支创建错误修复的位置。然后将 hotfix 分支合并到 master, 并制作一个新的生产版本。

Git 流程是一种集中式模式,因此,它让人想起与 Subversion、CVS 等一起使用的工作流程。主要区别在于使用 Git 具有一些技术和效率相关的优势。

另一种策略称为分叉模式,其中每个开发人员都有一个中央存储库,在组织内部很少实际使用,除非,例如,雇用分包商等外部方。

分支问题区域

持续交付实践和分支策略之间存在争论的根源。一些持续交付方法提倡单个主分支,所有发布都从主分支进行。Git Flow 就是这样一种模型。

这简化了部署的某些方面,主要是因为分支图变得更简单了。这反过来导致简化测试,因为只有一个分支通向生产服务器。

另一方面,如果我们需要对已发布的代码执行错误修复,而大师有我们不想发布的新功能怎么办?当生产中的安装节奏慢于开发团队的发布节奏时,就会发生这种情况。这是一种不受欢迎的事态,但并不少见。

有两种处理此问题的基本方法:

- 创建错误修复分支并从中部署到生产环境;这样更简单
从某种意义上说,我们不会破坏开发流程。另一方面,此方法可能需要重复测试资源。他们可能需要反映分支策略。

一切都是代码

·功能切换:另一种对应用提出更严格要求的方法

开发人员正在切换功能。在此工作流程中,您将关闭尚未准备好用于生产的功能。通过这种方式,您可以发布最新的开发版本,包括错误修复以及将被禁用和停用的新功能。

没有硬性规定,教条主义在这种情况下也无济于事。

最好为这两种情况做好准备,并使用在给定情况下最适合您的方法。

以下是一些可以指导您做出选择的观察结果:当您的更改大部分是向后兼容的或全新的时,功能切换是很好的。否则,功能切换代码可能会变得过于复杂,以适应许多不同的情况。



这反过来又使测试复杂化。

错误修复分支使部署和测试复杂化。虽然直接分支和修复错误很简单,但新版本应该叫什么,我们在哪里测试它?测试资源很可能已经被正在进行的开发分支占用。

有些产品足够简单,我们可以根据需要即时创建测试环境,但对于更复杂的应用程序来说,情况很少。有时,测试资源非常稀缺,例如难以复制的第三方Web服务甚至是硬件资源。

工件版本命名

当您有较大的安装时,版本号变得很重要。

下面列出了版本命名的基本原则:

- 版本号应该单调增长,即变大 · 它们之间应该具有可比性,应该很容易看出哪个版本更新
- 对所有工件使用相同的方案

这通常转换为包含三个或四个部分的版本号：

- 第一个主要是主要更改 此处的更改表示代码中的主要更改◦ 第二个是次要更改,它向后 API 兼容◦ 第三个是错误修复◦ 第四个可以是内部版本号

虽然这看起来很简单,但它是一个足够复杂的领域,需要以 SemVer 或语义版本控制的形式创建标准化工作。完整的规范可以在<http://semver.org> 上阅读。

所有可安装的工件在源代码管理系统中都有一个正确的版本号和相应的标签,这很方便。

有些工具不是这样工作的。Java 构建工具 Maven 支持快照版本号。快照可以看作是最后一部分模糊的版本。但是,快照有一些缺点;例如,您无法立即看到工件对应于哪个源代码标签。这使得调试损坏的安装变得更加困难。

快照工件策略进一步违反了一个基本的测试原则:被测试的完全相同的二进制工件应该是部署到生产中的那个。

当您使用快照时,您改为测试快照直到完成测试,然后您发布具有真实版本号的工件,然后部署该工件。它不再是同一个神器。



虽然快照的基本思想当然是更改版本号不会产生实际后果,但墨菲定律指出当然会有后果。根据我的经验,墨菲在这种情况下是正确的,快照版本控制弊大于利。相反,使用内部版本号。

选择客户

Git 的优点之一是它不强制使用特定的客户端。

有几种可供选择,它们都相互兼容。大多数客户使用几个核心 Git 实现之一,这有利于稳定性和质量。

大多数当前的开发环境都很好地支持使用 Git。

一切都是代码

从这个意义上说,选择客户并不是我们真正需要做的事情。大多数客户端工作得很好,选择可以留给使用客户端的人的偏好。许多开发人员使用集成在其开发环境中的客户端,或命令行 Git 客户端。在处理操作任务时,通常首选命令行 Git 客户端,因为它在通过 SSH shell 远程工作时使用起来很方便。

我们实际上必须做出选择的一个例外是,当我们帮助组织中一般不熟悉源代码管理 (尤其是 Git) 的人员时。

在这些情况下,维护有关如何安装和使用简单 Git 客户端并为我们组织的服务器配置它的说明很有用。

该组织的 wiki 中的简单说明手册通常可以很好地解决这个问题。

设置一个基本的 Git 服务器

设置基本的 Git 服务器非常容易。虽然这在大型组织中很少见,但在转向更高级的解决方案之前这是一个很好的练习。

让我们首先详细说明我们将采取的步骤以及完成这些步骤所需的点点滴滴:

1. 一台有两个用户帐户的客户机。 git 和 ssh 包应该被安装。

SSH 协议的突出特点是作为其他协议的基础传输,Git 也是如此。

您需要手边的 SSH 公钥。如果由于某种原因您没有密钥,请使用 ssh-keygen 来创建它们。

[ 我们需要两个用户,因为我们将模拟两个用户与中央服务器交谈。为两个测试用户制作密钥。]

2. 运行 SSH 守护进程的服务器。

这可以是与您模拟两个不同客户端用户的机器相同的机器,也可以是另一台机器。

3. Git 服务器用户。

我们需要一个单独的 Git 用户来处理 Git 服务器功能。

现在,您需要将两个用户的公钥附加到authorized_keys文件,该文件位于各自用户的.ssh目录中。将密钥复制到此帐户。



您是否开始觉得所有这一切都很麻烦?这就是为什么我们稍后会研究简化此过程的解决方案。

4. 一个裸 Git 存储库。

裸 Git 存储库是 Git 的一个特性。它们只是 Git 存储库,只是没有工作副本,因此它们节省了一些空间。这是如何创建一个:

```
cd /opt/git mkdir  
project.git cd project.git  
git init --bare
```

5. 现在,尝试克隆、更改并推送到服务器。

让我们回顾一下解决方案:

- 这个解决方案并不能很好地扩展。

如果您只有几个人需要访问权限,那么创建新项目和添加新密钥的开销还算不错。这是一次性成本。如果您的组织中有很多人需要访问权限,那么此解决方案需要做的工作太多。

- 解决安全问题会更加麻烦。

虽然我的观点可能有些争议,即组织内部花费了太多工作来限制员工对系统的访问,但不可否认的是,您在设置中需要这种能力。

在此解决方案中,您需要为每个角色设置单独的 Git 服务器帐户,这会造成大量重复工作。Git 没有开箱即用的细粒度用户访问控制。

一切都是代码

共享身份验证在大多数组织中,有某种形式的中央服务器用于处理身份验证。LDAP 服务器是一个相当普遍的选择。虽然假设您的组织已经以某种方式处理了这个核心问题并且已经在运行某种 LDAP 服务器,但是为了测试目的设置 LDAP 服务器相对容易。

一种可能性是使用 389 服务器 (以 LDAP 服务器常用的端口命名),以及用于管理 LDAP 服务器的 phpLDAPadmin Web 应用程序。

拥有这个测试 LDAP 服务器设置对于本书的目的很有用,因为我们可以为我们将要研究的所有不同服务器使用相同的 LDAP 服务器。

托管 Git 服务器

许多组织无法使用托管在另一个组织内部的服务根本。

这些可能是政府组织或处理金钱的组织,例如银行、保险和游戏组织。

原因可能是合法的,或者可以这么说,仅仅是对让关键代码离开组织的大门感到紧张。

如果您没有这样的疑虑,那么使用提供私人帐户的托管服务 (例如 GitHub 或 GitLab)是非常合理的。

无论如何,使用 GitHub 或 GitLab 是学习使用 Git 和探索其可能性的便捷方式。

这两家供应商都很容易评估,因为他们提供免费帐户,您可以在其中了解服务及其提供的内容。看看您是否真的需要所有服务,或者您是否可以使用更简单的服务。

GitLab 和 GitHub 相对于纯 Git 提供的一些特性如下:

- 网络界面
- 带有内置 wiki 的文档工具 · 问题跟踪器
- 提交可视化

- 分支可视化
- 拉取请求工作流

虽然这些都是有用的功能,但您并不总是可以使用所提供的设施。例如,您可能已经拥有需要集成的 wiki、文档系统、问题跟踪器等。

那么,我们正在寻找的最重要的功能是那些与管理和可视化代码最密切相关的功能。

大型二进制文件

GitHub 和 GitLab 非常相似,但也有一些不同。其中之一源于这样一个事实:像 Git 这样的源代码系统传统上不太适合存储大型二进制文件。一直有其他方法,例如将文件路径存储到文件服务器中的纯文本文件中。

但是,如果您实际上拥有的文件在某种意义上等同于源文件,只是它们是二进制文件,并且您仍想对它们进行版本控制,该怎么办?此类文件类型可能包括图像文件、视频文件、音频文件等。现代网站越来越多地使用媒体文件,而这一领域通常是内容管理系统 (CMSSes) 的领域。CMS,无论多么好,与 DevOps 流程相比都有缺点,因此在普通源处理系统中存储媒体文件的吸引力很强。CMS 的缺点包括它们通常具有古怪或不存在的脚本功能。自动化,另一个本应真正包含在“DevOps”合成词中的词,因此对于 CMS 来说通常很困难。

当然,您可以只将二进制文件签入 Git,它将像处理任何其他文件一样处理。然后发生的情况是涉及服务器操作的 Git 操作突然变得迟钝。然后,Git 的一些主要优势 效率和速度 消失了。

这个问题的解决方案随着时间的推移而演变,但还没有明确的赢家。
参赛者如下:

- Git LFS,由 GitHub 支持
- Git Annex,由 GitLab 支持,但仅限于企业版

Git Annex 是其中比较成熟的。这两种解决方案都是开源的,并通过其插件机制作为 Git 的扩展实现。

一切都是代码

还有其他几个系统,说明这是Git目前状态下一个没有解决的痛点。 Git 附件在<http://git-annex.branchable.com/not/> 上有不同品种之间的比较。

如果您需要对媒体文件进行版本控制,您应该从探索 Git Annex 开始。它是用 Haskell 编写的,可以通过许多发行版的包系统获得。

还应注意,此类解决方案的主要好处是能够将媒体文件与相应代码一起进行版本控制。

使用代码时,您可以方便地检查代码版本之间的差异。检查媒体文件之间的差异更难,而且用处不大。

简而言之,Git Annex 使用了一种久经考验的数据逻辑问题解决方案:添加一个间接层。它通过将符号链接存储到存储库中的文件来实现。然后将二进制文件存储在文件系统中,并由本地工作区使用其他方式(例如 rsync)获取。当然,这涉及更多的工作来设置解决方案。



尝试不同的 Git 服务器实现

Git 的分布式特性使得为各种目的尝试不同的 Git 实现成为可能。无论服务器如何设置,客户端设置都是相似的。

您还可以同时运行多个解决方案。客户端并不过分复杂,因为 Git 被设计为在需要时处理多个远程。

码头工人中场休息

在第7章“部署代码”中,我们将看到一种令人兴奋的新方法来打包我们的应用程序,称为Docker。

在本章中,我们要解决类似的挑战。我们需要能够尝试几种不同的 Git 服务器实现,看看哪一种最适合我们的组织。

我们可以为此使用 Docker,因此我们将借此机会窥视 Docker 为我们提供的简化部署的可能性。

由于我们将进一步深入研究 Docker,因此在本章中我们将作弊并声称 Docker 用于下载和运行软件。虽然这种描述并非完全不真实,但 Docker 远不止于此。要开始使用 Docker,请按照以下步骤操作:

1. 首先,根据特定说明安装 Docker

你的操作系统。对于 Red Hat 衍生产品,它是一个简单的`dnf install docker-io`命令。



io 后缀可能看起来有点神秘,但是已经有一个实现桌面功能的 Docker 包,所以选择了 docker-io。



2.然后,需要运行 docker 服务:

`systemctl start docker`

`systemctl enable docker`

3. 我们需要另一个工具,Docker Compose,在撰写本文时,

没有为 Fedora 打包。如果您的软件包存储库中没有它,请按照此页面<https://docs.docker.com/compose/install/>上的说明进行操作。



Docker Compose 用于一起自动启动多个 Docker 打包的应用程序,例如数据库服务器和 Web 服务器,这是我们在 GitLab 示例中需要的。



Gerrit一个基

本的 Git 服务器足以满足多种用途。

不过,有时您需要对工作流程进行精确控制。

一个具体的例子是将更改合并到基础设施关键部分的配置代码中。虽然我认为 DevOps 的核心是不要在基础设施代码周围放置不必要的繁文缛节,但不可否认有时这是必要的。如果不出意外,开发人员可能会在提交对基础架构的更改时感到紧张,并希望更有经验的人来审查代码更改。

一切都是代码

Gerrit 是一个基于 Git 的代码审查工具,可以在这些情况下提供解决方案。简而言之,Gerrit 允许您设置规则以允许开发人员审查和批准其他开发人员对代码库所做的更改。这些可能是高级开发人员审查没有经验的开发人员所做的更改,或者更常见的情况,这只是代码上的更多关注通常对质量有好处。

Gerrit 基于 Java,并在底层使用基于 Java 的 Git 实现。

Gerrit 可以作为 Java WAR 文件下载,并具有集成的设置方法。

它需要一个关系数据库作为依赖项,但您可以选择使用一个集成的基于 Java 的 H2 数据库,该数据库足以评估 Gerrit。

一个更简单的方法是使用 Docker 来试用 Gerrit。Docker 注册表中心上有多个 Gerrit 映像可供选择。为这次评估活动选择了以下一个: <https://hub.docker.com/r/openfrontier/gerrit/>

要使用 Docker 运行 Gerrit 实例,请执行以下步骤:

1. 初始化并启动Gerrit:

```
docker run -d -p 8080:8080 -p 29418:29418 openfrontier/gerrit
```

2. 打开浏览器访问<http://<docker host url>:8080>

现在,我们可以尝试我们想要的代码审查功能。

安装 git-review 包

在本地安装上安装git-review :

```
sudo dnf 安装 git-review
```

这将为 Git 安装帮助应用程序以与 Gerrit 通信。它添加了一个新命令git-review,用于代替git push将更改推送 到 Gerrit Git 服务器。

历史修正主义的价值

当我们与团队中的其他人一起使用代码时,代码的历史变得比我们自己工作时更重要。文件更改的历史记录成为一种交流方式。这在工作时尤为重要

带有代码审查和 Gerrit 等代码审查工具。

代码更改也需要易于理解。因此,尽管可能违反直觉,但编辑更改的历史记录以使生成的历史记录更加清晰是有用的。

例如,考虑这样一种情况,您进行了一些更改,后来改变了主意并删除了它们。对于其他人来说,您进行了一组编辑然后删除了它们并不是有用的信息。另一种情况是,当您有一组提交时,如果它们是单个提交,则更容易理解。以这种方式将提交添加在一起在Git 文档中称为压缩。

另一种使历史复杂化的情况是,当您多次从上游中央存储库合并,并且合并提交被添加到历史中。在这种情况下,我们希望通过首先删除我们的本地更改,然后从上游存储库获取和应用更改,最后重新应用我们的本地更改来简化更改。这个过程称为变基。

压缩和变基都适用于 Gerrit。

更改应该是干净的,最好是一次提交。这不是 Gerrit 特有的东西;如果包装得当,审阅者会更容易理解您的更改。审查将基于此提交。

1. 首先,我们需要从 Git/Gerrit 服务器端获取最新的更改。我们在服务器端更改之上重新调整我们的更改:

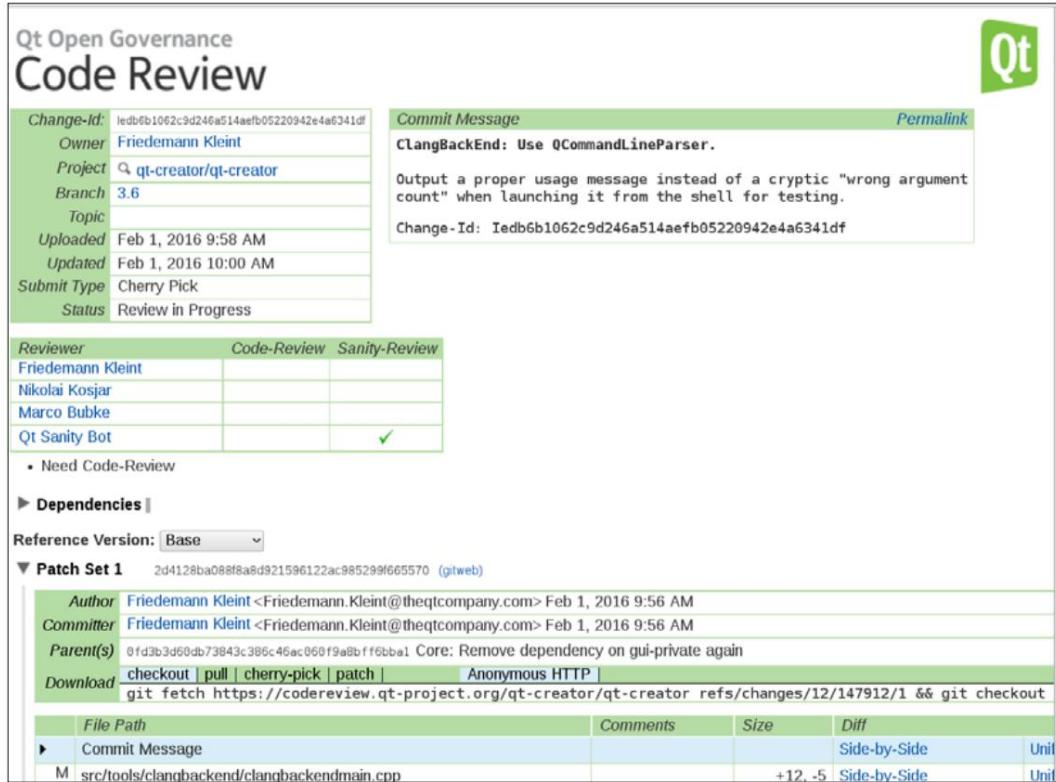
```
git pull --rebase origin master
```

2. 然后,我们通过压缩它们来完善我们的本地提交:

```
git rebase -i origin/master
```

[一切都是代码](#)

现在,让我们看一下 Gerrit Web 界面:



The screenshot shows the Gerrit Code Review interface for a project. The top navigation bar includes the Qt logo. The main content area displays a patch set with the following details:

- Change-Id:** Iedb6b1062c9d246a514aefb05220942e4a6341df
- Owner:** Friedemann Kleint
- Project:** qt-creator/qt-creator
- Branch:** 3.6
- Topic:** (empty)
- Uploaded:** Feb 1, 2016 9:58 AM
- Updated:** Feb 1, 2016 10:00 AM
- Submit Type:** Cherry Pick
- Status:** Review in Progress

Commit Message:

```
ClangBackend: Use QCommandLineParser.
Output a proper usage message instead of a cryptic "wrong argument count" when launching it from the shell for testing.
```

Reviewer:

Reviewer	Code-Review	Sanity-Review
Friedemann Kleint		
Nikolai Kosjar		
Marco Bubke		
Qt Sanity Bot		✓

• Need Code-Review

Dependencies:

Reference Version: Base

Patch Set 1

Author	Friedemann Kleint <Friedemann.Kleint@theqtcompany.com>	Feb 1, 2016 9:56 AM		
Committer	Friedemann Kleint <Friedemann.Kleint@theqtcompany.com>	Feb 1, 2016 9:56 AM		
Parent(s)	0f13b3d6db73843c386c46ac050f9a8bff6bb1 Core: Remove dependency on gui-private again			
Download	checkout pull cherry-pick patch Anonymous HTTP git fetch https://codereview.qt-project.org/qt-creator/qt-creator refs/changes/12/147912/1 && git checkout			
File Path	Comments	Size	Diff	Unif
▶ Commit Message			Side-by-Side	Unif
M src/tools/clangbackend/clangbackendmain.cpp		+12, -5	Side-by-Side	Unif

我们现在可以批准更改,并将其合并到 master 分支。

Gerrit 中有很多值得探索的地方,但这些是基本原则,应该足以作为评估的基础。

但是,结果值得麻烦吗?这些是我的观察:

- Gerrit 允许对敏感代码库进行细粒度访问。经授权人员审核后方可进行更改。

这是 Gerrit 的主要优势。如果您只是出于不明确的原因想要强制代码审查,请不要这样做。对所有相关人员来说,好处必须是明确的。与权威系统相比,最好就其他更非正式的代码审查方法达成一致。

- 如果 Gerrit 的替代方案是根本不允许访问代码库,甚至是只读访问,那么实施 Gerrit。

组织的某些部分可能过于紧张而不允许访问基础设施配置等内容。这通常是出于错误的原因。

您通常面临的问题不是人们对您的代码感兴趣;而是人们对您的代码感兴趣。恰恰相反。

有时,敏感密码会被签入代码,这被视为不允许访问源代码的理由。好吧,如果它很痛,就不要这样做。解决导致存储库中存在密码的问题。

拉取请求模型

围绕代码审查创建工作流的问题还有另一种解决方案:拉取请求模型,它已被 GitHub 流行起来。

在此模型中,除了存储库所有者之外,不允许推送到存储库。不过,其他开发人员可以分叉存储库,并在他们的分叉中进行更改。当他们完成更改后,他们可以提交拉取请求。然后,存储库所有者可以查看请求并选择将更改拉入主存储库。

这种模式的优点是简单易懂,很多开发者在GitHub上的众多开源项目中都有使用经验。

设置一个能够在本地处理拉取请求模型的系统将需要类似 GitHub 或 GitLab 的东西,我们将在接下来查看。

GitLab

在 Git 之上支持许多方便的功能。它是一个基于 Ruby 的庞大而复杂的软件系统。因此,安装起来可能很困难,要正确获取所有依赖项等等。

在<https://registry.hub>上有一个很好的适用于 GitLab 的 Docker Compose 文件。docker.com/u/sameersbn/gitlab/。如果您按照之前显示的 Docker 说明进行操作,包括安装docker-compose,那么现在启动本地 GitLab 实例非常简单:

```
mkdir gitlab cd  
gitlab wget https://  
raw.githubusercontent.com/sameersbn/docker-gitlab/master/docker-compose.yml docker-compose up
```

docker -compose命令将读取.yml文件并在默认演示配置中启动所有必需的服务。

一切都是代码

如果您阅读控制台窗口中的启动日志,您会注意到已经启动了三个独立的应用程序容器: gitlab postgresql1、gitlab redis1和gitlab gitlab1。

GitLab 容器包括 Ruby 基础 Web 应用程序和 Git 后端功能。Redis 是分布式键值存储,而 PostgreSQL 是关系数据库。

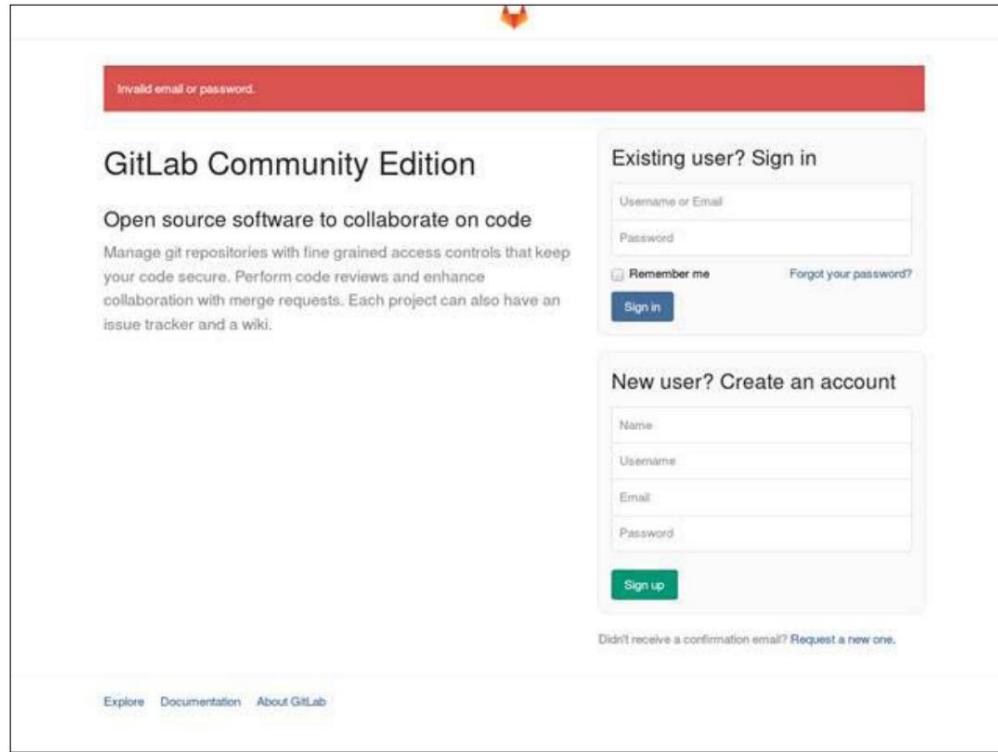
如果您习惯于设置复杂的服务器功能,您会感激我们使用docker-compose节省了大量时间。

docker-compose.yml文件在/srv/docker/gitlab中设置数据卷。

要登录 Web 用户界面,请使用 GitLab Docker 映像安装说明中提供的管理员密码。它们已在此处复制,但请注意,它们可能会随着 Docker 映像作者认为合适而更改:

- 用户名:root
- 密码:5iveL!fe

这是 GitLab Web 用户界面登录屏幕的屏幕截图:



尝试将项目从 GitHub 或本地私有项目导入到 GitLab 服务器。

看看 GitLab 如何可视化提交历史和分支。

在调查 GitLab 时,您可能会同意它提供了大量有趣的功能。

在评估功能时,重要的是要记住它们最终是否有可能被使用。 GitLab 或类似软件可以为您解决什么核心问题?

事实证明,GitLab 增加的主要价值是消除了 DevOps 工作流程中的瓶颈,具体表现为以下两个特性:

- 用户 ssh 密钥的管理 · 创建新的存储库

这些功能通常被认为是最有用的。

可视化功能也很有用,但 Git 客户端提供的客户端可视化对开发人员更有用。

概括

在本章中,我们探索了一些可用于管理组织源代码的选项。我们还调查了需要在 DevOps、版本编 号和分支策略中做出决策的领域。

在处理了纯形式的源代码之后,我们下一步将致力于将代码构建成有用的二进制工件。

Machine Translated by Google

5

构建代码

你需要一个系统来构建你的代码,你需要一个地方来构建它。

Jenkins 是一个灵活的开源构建服务器,可以随着您的需求而增长。
还将探索 Jenkins 的一些替代方案。

我们还将探讨不同的构建系统以及它们如何影响我们的 DevOps 工作。

我们为什么要构建代码?

大多数开发人员都熟悉构建代码的过程。然而,当我们在 DevOps 领域工作时,我们可能会遇到专门从事特定组件类型编程的开发人员不一定会遇到的问题。

出于本书的目的,我们将软件构建定义为将代码从一种形式塑造成另一种形式的过程。在这个过程中,可能会发生几件事:

- 将源代码编译为本机代码或虚拟机字节码,
取决于我们的生产平台。
- 代码检查:检查代码是否有错误并通过静态代码分析生成代码质量度量。“Linting”一词起源于一个名为 Lint 的程序,该程序开始随 Unix 操作系统的早期版本一起发布。该程序的目的是在语法上正确但包含可疑代码模式的程序中查找错误,这些错误可以通过与编译不同的过程来识别。
- 单元测试,通过以受控方式运行代码。 · 适合部署的工件的生成。

这是一项艰巨的任务!

构建代码

并非所有代码都会经历这些阶段中的每一个阶段。例如,解释型语言可能不需要编译,但它们可能会受益于质量检查。

构建系统的多方面

有许多构建系统在软件开发的历史中得到了发展。有时,可能感觉构建系统的数量多于编程语言的数量。

这是一个简短的列表,只是为了了解其中有多少:

- 对于 Java,有 Maven、Gradle 和 Ant
- 对于 C 和 C++,有许多不同风格的 Make
- 对于 Clojure,JVM 上的一种语言,有 Leiningen 和 Boot
来自 Maven
- 对于 JavaScript,有 Grunt
- 对于 Scala,有 sbt
- 对于 Ruby,我们有 Rake
- 最后,当然,我们有各种 shell 脚本

根据您组织的规模和您正在构建的产品类型,您可能会遇到任意数量的此类工具。为了让生活更有趣,组织发明自己的构建工具的情况并不少见。

作为对众多构建工具的复杂性的反应,通常还会出现对特定工具进行标准化的想法。如果您正在构建复杂的异构系统,这种方法效率不高。例如,使用 Grunt 构建 JavaScript 软件只是比使用 Maven 或 Make 更容易,使用 Maven 构建 C 代码效率不是很高,等等。通常,该工具的存在是有原因的。

通常,组织在单一生态系统上进行标准化,例如 Java 和 Maven 或 Ruby 和 Rake。除了用于主要代码库的构建系统之外,其他构建系统主要用于本机组件和第三方组件。

无论如何,我们不能假设我们在组织的代码库中只会遇到一个构建系统,也不能假设只有一种编程语言。

我发现这条规则在实践中很有用:开发人员应该可以检查代码并在他或她的本地开发机器上以最小的意外构建它。

这意味着我们应该标准化版本控制系统,并有一个单一的界面来在本地开始构建。

如果您要支持多个构建系统,这基本上意味着您需要将一个构建系统包装在另一个构建系统中。因此隐藏了构建的复杂性,并且允许同时使用多个构建系统。不熟悉特定构建的开发人员仍然可以期望检查它并相当轻松地构建它。

例如,Maven 适用于声明式 Java 构建。 Maven 还能够从 Maven 构建中启动其他构建。

这样,以 Java 为中心的组织中的开发人员可以期望以下命令行始终构建组织的组件之一:

mvn 全新安装

一个具体示例是使用 Nullsoft NSIS Windows 安装系统创建 Java 桌面应用程序安装程序。Java 组件是使用 Maven 构建的。当 Java 工件准备就绪时,Maven 调用 NSIS 安装程序脚本来生成一个独立的可执行文件,该可执行文件将在 Windows 上安装应用程序。

虽然现在 Java 桌面应用程序并不流行,但它们在某些领域仍然很流行。

Jenkins 构建服务器构建服务器本质上是一个基于各种触发器

构建软件的系统。

有几种可供选择。在本书中,我们将了解 Jenkins,它是一种用 Java 编写的流行构建服务器。

Jenkins 是 Hudson 构建服务器的一个分支。 Kohsuke Kawaguchi 是 Hudson 的主要贡献者,2010 年,在 Oracle 收购 Hudson 之后,他继续致力于 Jenkins 分支。如今,詹金斯显然是这两种菌株中更成功的一个。

Jenkins 对构建 Java 代码有特别的支持,但绝不仅限于构建 Java。

一开始设置基本的 Jenkins 服务器并不是特别困难。在 Fedora 中,您可以通过 dnf 安装它:

dnf 安装詹金斯

Jenkins 通过 systemd 作为服务处理:

systemctl 启动詹金斯

构建代码

您现在可以在`http://localhost:8080`查看 Web 界面：

The screenshot shows the Jenkins dashboard. On the left, there's a sidebar with links for 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. Below that is a 'Build Queue' section stating 'No builds in the queue.' Under 'Build Executor Status', it shows '1 Idle' and '2 Idle'. The main area is titled 'All' and displays a table of build jobs:

S	W	Name	Last Success	Last Failure	Last Duration
		emacs	50 min - #384	9 mo 8 days - #239	11 min
		fortune	4 days 6 hr - #2	N/A	81 ms
		inkscape	12 hr - #102	10 hr - #103	5 min 16 sec

Below the table, there are icons for 'S M L' and a 'Legend' with three RSS feed links: 'RSS for all', 'RSS for failures', and 'RSS for just latest builds'.

屏幕截图中的 Jenkins 实例已经定义了几个作业。 Jenkins 中的基本实体是作业定义,有多种类型可供选择。让我们在 Web 界面中创建一个简单的作业。为了简单起见,这项工作将只打印一个经典的 Unix 财富报价:

1. 创建一个Freestyle 项目类型的作业：

The screenshot shows the 'New Item' creation dialog. In the top right, there's a search bar and an 'OK' button. The 'Item name' field contains 'fortunes'. Below it, there are five options:

- Freestyle project**: This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
- Maven project**: Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- External Job**: This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).
- Multi-configuration project**: Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- Copy existing item**: Copy from

At the bottom is an 'OK' button.

2. 添加shell构建步骤。
3. 在 shell 条目 (命令)中,键入fortune:

The screenshot shows the Jenkins configuration interface for a project named 'fortune'. On the left, there's a sidebar with links like Back to Dashboard, Status, Changes, Workspace, Build Now, Delete Project, and Configure. The main area shows the project name 'fortune' and a description field. Under 'Advanced Project Options', there are checkboxes for Discard Old Builds, This build is parameterized, Disable Build, and Execute concurrent builds if necessary. The 'Source Code Management' section has 'None' selected. In the 'Build Triggers' section, three options are available: Build after other projects are built, Build periodically, and Poll SCM. The 'Build' section contains a single step: 'Execute shell' with the command 'fortune' entered. Below the build steps, there are sections for 'Post-build Actions' and buttons for 'Save' and 'Apply'.

每当您运行该作业时,都会在作业日志中打印一个财富报价。

构建代码

可以手动启动作业,您会找到作业执行的历史记录并可以检查每个作业日志。这保留了以前执行的历史记录,当您试图找出哪个更改破坏了构建以及如何破坏构建时,这非常方便。

要解决这个问题。

如果您没有fortune命令,请使用dnf install fortune-mod 安装它,或者您可以选择简单地运行date命令。这只会在构建日志中输出日期,而不是经典的引语和俏皮话。

管理构建依赖

在前面的简单示例中,我们将幸运饼干打印到构建日志中。

虽然这个练习在复杂性上无法与管理真实构建相提并论,但我们至少学会了安装和启动 Jenkins,如果您在安装fortune实用程序时遇到问题,您可以瞥见管理持续集成服务器的阴暗面:管理构建依赖项。

某些构建系统(例如 Maven 工具)的优点在于 Maven POM 文件包含对需要哪些构建依赖项的描述,如果它们不存在于构建服务器上,Maven 会自动获取它们。

Grunt 以类似的方式构建 JavaScript。有一个构建描述文件,其中包含构建所需的依赖项。Golang 构建甚至可以包含指向完成构建所需的 GitHub 存储库的链接。

C 和 C++ 构建以不同的方式呈现挑战。许多项目使用 GNU Autotools;其中之一是 Autoconf,它使自己适应主机上可用的依赖项,而不是描述它们需要哪些依赖项。因此,要构建文本编辑器 Emacs,您首先要运行一个配置脚本,该脚本确定构建系统上可用的众多潜在依赖项中的哪些。



如果缺少可选的依赖项,例如用于图像支持的图像库,则最终的可执行文件中将无法使用该可选功能。您仍然可以构建该程序,但您将无法获得您的构建机器未准备好的功能。

虽然如果您希望您的软件根据它应该在哪个系统上运行而在许多不同的配置中工作,这是一个有用的功能,但我们通常不希望我们的构建在企业环境中运行。在这种情况下,我们需要完全确定哪些功能最终可用。我们当然不希望在我们的生产服务器上以缺少功能的形式出现糟糕的惊喜。

RPM (Red Hat Package Manager 的缩写)系统用于从 Red Hat 派生的系统上,提供了解决此问题的方法。 RPM 系统的核心是一个称为规范文件的构建描述符文件,规范文件的简称。它列出了成功构建所需的构建依赖项以及使用的构建命令和配置选项等。由于 spec 文件本质上是一个基于宏的 shell 脚本,您可以使用它来构建多种类型的软件。 RPM 系统还认为构建源应该是原始的。 spec 文件可以通过在构建源代码之前修补源代码来调整源代码。

最终神器使用RPM 系统完成构建后,您

将得到一个RPM 文件,这是一种非常方便的基于Red Hat 操作系统的部署神器。

对于基于 Debian 的发行版,您会得到一个.deb文件。

Maven 构建的最终输出通常是企业存档,或简称为 EAR 文件。这包含 Java 企业应用程序。

这是我们稍后将部署到生产服务器的最终部署工件。

在本章中,我们关注构建部署所需的工件,在第7章部署代码中,我们讨论工件的最终部署。

然而,即使在构建我们的工件时,我们也需要了解如何部署它们。目前,我们将使用以下经验法则:操作系统级别的打包优于专门的打包。这是我个人的喜好,其他人可能不同意。

让我们简要讨论一下这个经验法则的背景以及备选方案。

作为一个具体示例,让我们考虑 Java EAR 的部署。通常,我们可以通过几种方式来做到这一点。这里有些例子:

- 通过机制将 EAR 文件部署为 RPM 包,并且
基本操作系统中可用的频道
- 通过 Java 应用程序可用的机制部署 EAR
服务器,例如 JBoss、WildFly 和 Glassfish

构建代码

从表面上看,使用特定于 Java 应用程序服务器的机制来部署 EAR 文件可能会更好,因为它无论如何都是特定于应用程序服务器的。如果您只做 Java 开发,那么这可能是一个合理的假设。然而,由于您无论如何都需要管理您的基本操作系统,因此您已经有了可以重用的可用部署方法。

此外,由于很可能您不仅在进行 Java 开发,而且至少还需要部署和管理 HTML 和 JavaScript,因此使用更通用的部署方法开始变得有意义。

我所经历过的几乎所有组织都拥有包含许多不同技术的复杂架构,并且这条经验法则在大多数情况下都适用。

唯一真正的例外是在 Unix 服务器与 Windows 服务器共存的混合环境中。在这些情况下,Unix 服务器通常会使用他们首选的包分发方法,而 Windows 服务器不得不一瘸一拐地使用某种自制的解决方案。这只是一种观察,而不是对这种情况的纵容。

使用 FPM 作弊

使用 spec 文件构建操作系统可交付成果(例如 RPM)是非常有用的知识。然而,有时您并不需要真实规范文件的严谨性。spec 文件毕竟是针对你自己不是发起者的场景优化的代码库。

有一个名为 FPM 的基于 Ruby 的工具,它可以直接从命令行生成适合构建的源 RPM。

该工具可在 GitHub 上获得,网址为<https://github.com/jordansissel/fpm>。

在 Fedora 上,您可以像这样安装 FPM:

百胜安装rubygems

百胜安装红宝石

yum 安装 ruby-devel gcc

宝石安装 fpm

这将安装一个包装 FPM Ruby 程序的 shell 脚本。

FPM 的一个有趣方面是它可以生成不同类型的包;受支持的类型包括 RPM 和 Debian。

这是制作 “hello world”shell 脚本的简单示例：

```
#!/bin/sh  
回声“你好世界！”
```

我们希望将 shell 脚本安装在 /usr/local/bin 中,因此在您的主目录中创建一个具有以下结构的目录：

```
$HOME/hello/usr/local/bin/hello.sh
```

使脚本可执行,然后打包：

```
chmod a+x usr/local/bin/hello.sh  
fpm -s dir -t rpm -n hello-world -v 1 -C 安装目录 usr
```

这将生成名称为hello-world且版本为1 的 RPM。

要测试包,我们可以先列出内容然后安装它：

```
rpm -qivp 你好世界.rpm  
rpm -ivh 你好世界.rpm
```

shell 脚本现在应该很好地安装在 /usr/local/bin 中。

FPM 是创建 RPM、Debian 和其他包类型的一种非常方便的方法。这有点像作弊！

持续集成

使用构建服务器的主要好处是实现持续集成。
每次检测到代码库中的更改时,都会启动测试新提交代码质量的构建。

由于可能有许多开发人员在处理代码库,每个人的版本都略有不同,因此查看所有不同的更改是否能正常协同工作很重要。这称为集成测试。如果集成测试相距太远,不同代码分支分歧太大的风险就会增加,合并就不再容易了。结果通常被称为“合并地狱”。由于分支之间的分歧,开发人员的本地更改应该如何合并到 master 分支不再明确。这种情况是非常不可取的。合并地狱的根本原因是心理上的,也许令人惊讶。为了将您的更改合并到主线,需要克服心理障碍。使用 DevOps 的一部分是让事情变得更容易,从而减少与提交更改等重要工作相关的感知成本。

构建代码

持续集成构建通常以比开发人员在本地执行的方式更严格的方式执行。这些构建需要更长的时间来执行,但由于如今高性能硬件并不那么昂贵,我们的构建服务器足够强大以应对这些构建。

如果构建速度足够快,不会让人觉得单调乏味,开发人员就会很乐意经常检查,并且会及早发现集成问题。

持续交付

在持续集成步骤成功完成后,您将拥有闪亮的新工件,可以将其部署到服务器。通常,这些测试环境设置为像生产服务器一样运行。

我们将在本书后面讨论部署系统的备选方案。

通常,构建服务器做的最后一件事是将成功构建的最终工件部署到工件存储库。从那里,部署服务器接管了将它们部署到应用程序服务器的责任。在 Java 世界中,Nexus 存储库管理器相当普遍。它支持除 Java 格式之外的其他格式,例如 JavaScript 工件和 RPM 的 Yum 通道。Nexus 现在也支持 Docker Registry API。

使用 Nexus 进行 RPM 分发只是一种选择。您可以相当轻松地使用 shell 脚本构建 Yum 通道。

詹金斯插件

Jenkins 有一个插件系统来为构建服务器添加功能。有许多不同的插件可用,它们可以从 Jenkins 网络界面中安装。其中许多甚至可以在不重新启动 Jenkins 的情况下安装。此屏幕截图显示了一些可用插件的列表:

第 5 章

The screenshot shows the Jenkins Plugin Manager interface. The top navigation bar includes links for 'Back to Dashboard' and 'Manage Jenkins'. Below the navigation is a search bar and a 'Filter:' input field. The main content area has tabs for 'Updates', 'Available' (which is selected), 'Installed', and 'Advanced'. A 'Name' and 'Version' header is present above a list of plugins. The list is titled 'Artifact Uploaders' and includes the following entries:

Name	Version
Appaloosa Plugin	1.4.2
Appetize.io Plugin	1.1.0
appthwack	1.9
ArtifactPromotionPlugin	0.3.5
Artifact Deployer Plug-in	0.33
aws-device-farm	1.9
AWS Lambda Plugin	0.3.0
AWS Elastic Beanstalk Deployment Plugin	0.0.3
Backlog plugin	1.10
Hudson Build-Publisher plugin	1.21
Capitomcat Plugin	0.1.0
Cloud Foundry Plugin	1.4.2
AWS CodeDeploy Plugin for Jenkins	1.7
Confluence Publisher	1.8
Crittercism dSYM Plugin	1.1
CRX Content Package Deployer Plugin	1.3.2
Deploy to container Plugin	1.10
Deploy to Websphere container Plugin	1.0
Xebialabs XL Deploy Plugin	5.0.0

At the bottom of the page are three buttons: 'Install without restart', 'Download now and install after restart', and a status message: 'Update information obtained: 6 hr 30 min ago'.

其中,我们需要 Git 插件来轮询我们的源代码存储库。

我们的示例组织选择了 Clojure 作为他们的构建,因此我们将安装 Leiningen 插件。

构建代码

主机服务器构建服务器

通常是组织中非常重要的机器。构建软件是处理器以及内存和磁盘密集型的。构建不应花费太长时间，因此您需要一台具有良好构建服务器规格的服务器，具有大量磁盘空间、处理器内核和 RAM。

构建服务器也有一种社交方面：正是在这里，许多不同的人和角色的代码第一次正确地集成在一起。如果服务器足够快，这方面的重要性就会增加。机器比人便宜，所以不要让这台机器成为你省钱的地方。

建立奴隶

要减少构建队列，您可以添加构建从站。主服务器将基于循环方案将构建发送给从服务器，或者将特定构建绑定到特定构建从服务器。

这样做的原因通常是某些构建对主机操作系统有一定的要求。

Build slaves 可用于提高并行构建的效率。它们还可以用于在不同的操作系统上构建软件。例如，对于使用 Windows 构建工具的组件，您可以拥有 Linux Jenkins 主服务器和 Windows 从属服务器。要为 Apple Mac 构建软件，让 Mac 构建从属很有用，特别是因为 Apple 对于在虚拟服务器上部署其操作系统有古怪的规则。

有几种方法可以将构建从站添加到 Jenkins 主站；请参阅 <https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds>。

从本质上讲，Jenkins master 必须有一种方法可以向 build slave 发出命令。这个命令通道可以是经典的 SSH 方法，Jenkins 有一个内置的 SSH 工具。您还可以通过从主服务器向从服务器下载 Java JNLP 客户端来启动 Jenkins 从服务器。如果 build slave 没有 SSH 服务器，这很有用。

关于交叉编译的说明虽然可以使用 Windows 构建从属,但有时使用 Linux 构建 Windows 软件实际上更容易。GCC 等 C 编译器可以配置为使用 MinGW 包执行交叉编译。

这是是否更容易很大程度上取决于正在构建的软件。

一个大系统通常包含许多不同的部分,其中一些部分可能包含适用于不同平台的本机代码。

以下是一些示例:

- 原生 android 组件
- 以 C 语言编写的原生服务器组件以提高效率
- 原生客户端组件,也以 C 或 C++ 编码以提高效率

本机代码的流行程度在一定程度上取决于您与之合作的组织的性质。电信产品通常有很多本地代码,例如编解码器和硬件接口代码。银行系统可能有本地代码的高速消息系统。

这方面的一个方面是能够在构建服务器上方便地构建所有正在使用的代码是很重要的。否则,某些代码往往只能在某些在某人办公桌下积尘的机器上构建。这是一个需要避免的风险。

您组织的系统需要什么,只有您自己知道。

主机上的软件根据构建的复杂性,您可能需要在构建服务器上安装许多不同类型的构建工具。请记住,Jenkins 主要用于触发构建,而不是自己执行构建。该工作委托给所使用的构建系统,例如 Maven 或 Make。

根据我的经验,拥有基于 Linux 的主机操作系统最为方便。

大多数构建系统都在分发存储库中可用,因此从那里安装它们非常方便。

要使构建服务器保持最新,您可以使用与使应用程序服务器保持最新相同的部署服务器。

[构建代码](#)

触发器

您可以使用计时器来触发构建,也可以轮询代码存储库以获取更改并在有更改时构建。

同时使用这两种方法可能很有用:

- 大多数时候都可以使用Git 存储库轮询,这样每次签入都会触发构建。 · 可以触发夜间构建,这比连续构建更严格

构建,因此需要更长的时间。由于这些构建发生在晚上,此时应该没有人工作,所以它们是否慢并不重要。

- 上游构建可以触发下游构建。

您还可以让一个作业的成功构建触发另一个作业。

作业链和构建管道

能够将作业链接在一起通常很有用。在最简单的形式中,这是通过在第一个作业成功完成时触发第二个作业来实现的。多个作业可以通过这种方式级联在一条链中。这样的构建链通常足以满足许多目的。有时,需要更好地可视化构建步骤以及更好地控制链的细节。

在 Jenkins 术语中,链中的第一个构建称为上游构建,第二个称为下游构建。

虽然这种链接构建的方式通常就足够了,但最终很可能需要更好地控制构建链。这样的构建链通常称为管道或工作流。

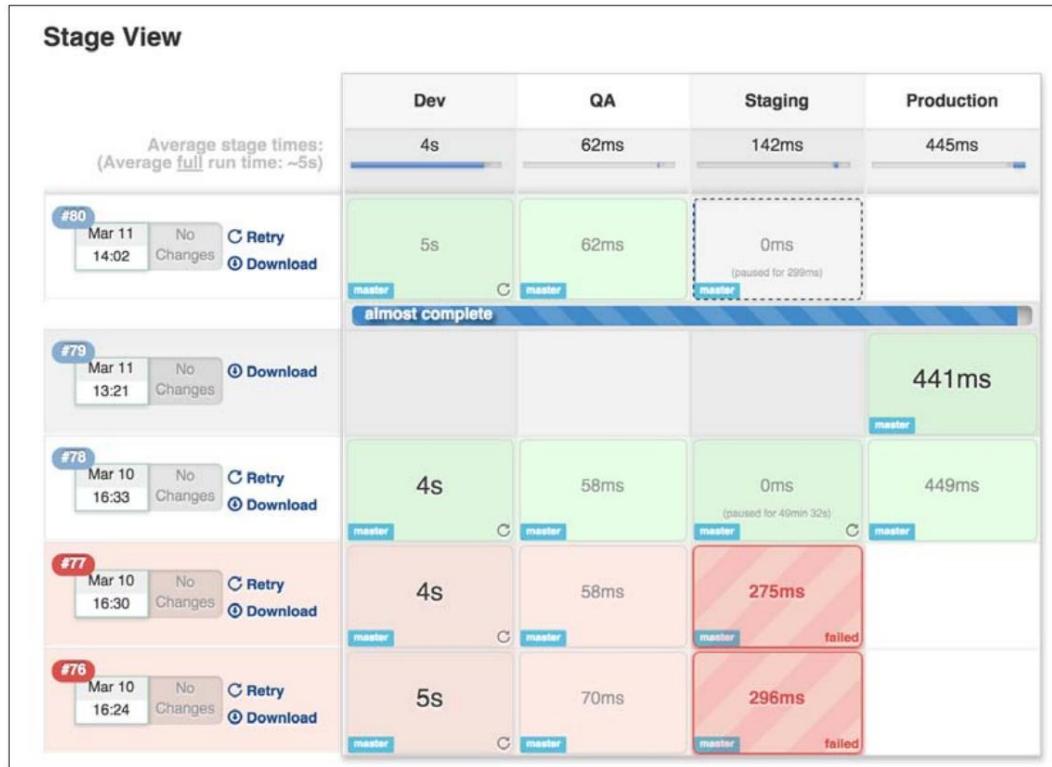
有很多插件可以为 Jenkins 创建改进的管道,而且有几个的事实表明在这方面确实有很大的改进愿望。

两个例子是多作业插件和工作流插件。

工作流插件更高级,并且还具有可以在 Groovy DSL 中描述而不是在 Web UI 中摆弄的优点。

工作流插件由 CloudBees 推广,他们是当今 Jenkins 的主要贡献者。

此处说明了一个示例工作流程：



当您查看工作流插件使用的 Groovy 构建脚本时,您可能会认为 Jenkins 基本上是一个具有 Web 界面的构建工具,您或多或少是正确的。

看看 Jenkins 文件系统布局

了解构建在文件系统中结束的位置通常很有用。

对于 Fedora 软件包,Jenkins 作业存储在这里：

/var/lib/詹金斯/工作

每个作业都有自己的目录,作业描述 XML 存储在这个目录中,还有一个名为 workspace 的构建目录。作业的 XML 文件可以备份到另一台服务器,以便在发生灾难性故障时能够重建 Jenkins 服务器。也有专门用于此目的的备份插件。

[构建代码](#)

构建会占用大量空间,因此有时您可能需要手动清理此空间。

当然,这不应该是正常情况。您应该将 Jenkins 配置为仅保留您有空间的构建数量。如果需要,您还可以配置配置管理工具以清除空间。

您可能需要深入研究文件系统的另一个原因是当构建神秘地失败时,您需要调试失败的原因。一个常见的原因是构建服务器状态不符合预期。例如,对于 Maven 构建,破坏的依赖关系可能会污染构建服务器上的本地存储库。

将服务器和基础设施构建为代码当我们讨论 Jenkins 文件结构时,注意到基于 GUI 的工具(例如 Jenkins)和基础设施应被描述为代码的 DevOps 公理之间经常发生的阻抗不匹配是很有用的。

理解这个问题的一种方法是,虽然 Jenkins 作业描述符是基于文本文件的,但这些文本文件并不是更改作业描述符的主要接口。

Web 界面是主要界面。这既是优势也是劣势。

使用 Jenkins 在现有构建之上创建临时解决方案很容易。您无需非常熟悉 Jenkins 即可完成有用的工作。

另一方面,Jenkins 开箱即用的体验缺少我们在编程世界中习惯的许多功能。在 Jenkins 中,继承甚至函数定义等基本功能都需要付出一些努力才能提供。

例如,GitLab 中的构建服务器功能采用了不同的方法。构建描述符从一开始就是代码。如果您不需要 Jenkins 提供的所有可能性,那么值得在 GitLab 中查看此功能。

按依赖顺序构建

许多构建工具都有构建树的概念,其中依赖项按照构建完成所需的顺序构建,因为构建的某些部分可能依赖于其他部分。

在 Make-like 工具中,这是明确描述的;例如,像这样:

```
a.out : 博科  
    博 : 公元前  
    合作 : 抄送
```

因此,为了构建a.out,必须首先构建bo和co。

在 Maven 等工具中,构建图派生自我们为工作设置的依赖关系。另一个 Java 构建工具 Gradle 在构建之前也会创建一个构建图。

Jenkins 支持在 Web 用户界面中可视化 Maven 构建的构建顺序,在 Maven 中称为反应器。

但是,此视图不适用于 Make 风格的构建。

The screenshot shows the Jenkins 'Modules' dashboard. On the left, there's a sidebar with links like 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Maven project', 'Configure', 'Modules', and 'Git Polling Log'. Below that is a 'Build History' section with a table of recent builds. The main area is titled 'Modules' and contains a table with columns: S, W, Name, Last Success, Last Failure, and Last Duration. The table lists several modules with their status (green for success, yellow for warning), last successful build time, last failure time, and duration.

S	W	Name	Last Success	Last Failure	Last Duration
		asd	4 hr 48 min - #2017	N/A	2.7 sec
		adminWeb	4 hr 48 min - #2017	N/A	5 sec
		alarm-handling-service	4 hr 48 min - #2017	N/A	1 sec
		alarm-to-client	4 hr 48 min - #2017	N/A	0.93 sec
		applet-client	4 hr 48 min - #2017	N/A	1 sec
		applet-client->download	4 hr 48 min - #2017	N/A	3 sec
		raspberry-xmlrpc-interface-servlet	4 hr 48 min - #2017	N/A	1.4 sec
		client-update-servlet	4 hr 48 min - #2017	N/A	0.68 sec
		customer-web-resources	4 hr 48 min - #2017	N/A	1.1 sec
		fencee-manager	4 hr 48 min - #2017	N/A	0.54 sec
		mmx	4 hr 48 min - #2017	N/A	10 sec
		mmx-ear	4 hr 48 min - #2017	N/A	1 min 56 sec
		mmx-RPM	4 hr 48 min - #2017	N/A	7.3 sec
		mmxCoreBeans-ejb3	4 hr 48 min - #2017	N/A	

构建阶段

Maven 构建工具的主要优点之一是它使构建标准化。

这对于大型组织非常有用,因为它不需要发明自己的构建标准。其他构建工具通常在如何实现各个构建阶段方面更为宽松。Maven 的刚性有其优点和缺点。有时,刚开始使用 Maven 的人会想起使用 Ant 等工具可以拥有的自由。

您可以使用任何工具来实施这些构建阶段,但如果工具本身不强制执行标准顺序:构建、测试和部署,则很难保持这种习惯。

构建代码

我们将在后面的章节中更详细地研究测试,但在这里我们应该注意测试阶段非常重要。持续集成服务器需要非常擅长捕捉错误,而自动化测试对于实现该目标非常重要。

替代构建服务器虽然 Jenkins 在我的经验中似乎在构建服务器场景中占据主导地位,但它绝不是唯一的。Travis CI 是一种托管解决方案,在开源项目中很受欢迎。Buildbot 是一个用 Python 编写并可配置的构建服务器。Go 服务器是另一个来自 ThoughtWorks 的服务器。

Bamboo 是 Atlassian 的产品。GitLab 现在还支持构建服务器功能。

在决定哪种构建服务器最适合您之前,请货比三家。

在评估不同的解决方案时,请注意供应商锁定的尝试。还要记住,构建服务器不会以任何方式取代对在开发人员机器上本地表现良好的构建的需求。

此外,作为一般经验法则,查看该工具是否可通过配置文件进行配置。虽然管理层往往对图形配置印象深刻,但开发人员和运营人员很少喜欢被迫使用只能通过图形用户界面进行配置的工具。

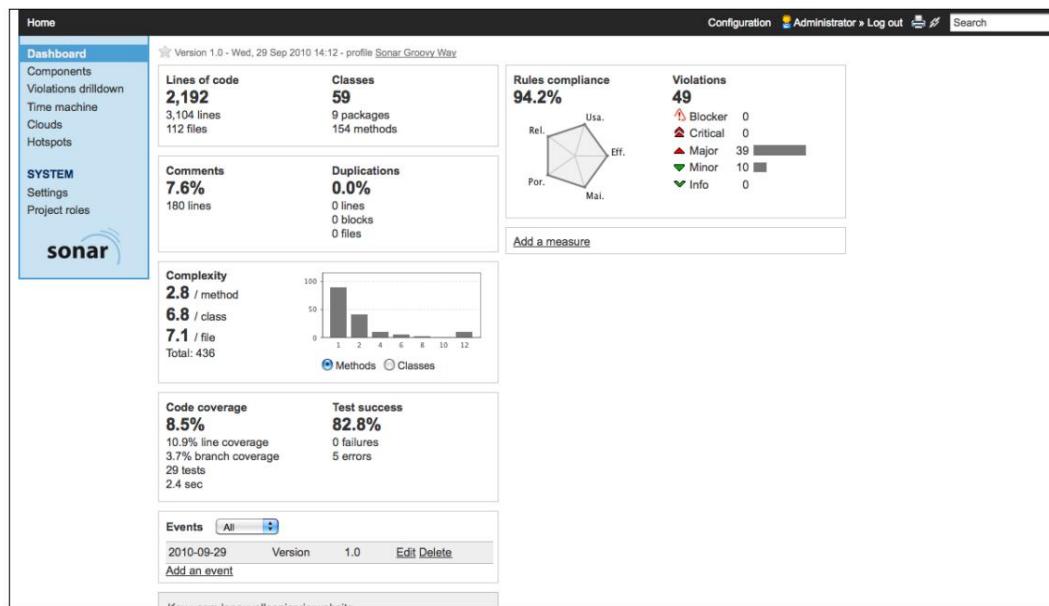
整理质量措施

构建服务器可以做的一件有用的事情是整理软件质量指标。Jenkins 对此有一些开箱即用的支持。执行 Java 单元测试,可以直接在作业页面上进行可视化。

另一个更高级的选项是使用 Sonar 代码质量可视化工具,如以下屏幕截图所示。Sonar 测试在构建阶段运行并传播到 Sonar 服务器,在那里它们被存储和可视化。

Sonar 服务器可以成为开发团队查看他们在改进代码库方面的努力成果的好方法。

实施 Sonar 服务器的缺点是它有时会减慢构建速度。建议每天一次在夜间构建中执行 Sonar 构建。



关于构建状态可视化构建服务器生成大量数据，这些数据适合在共享显示上进行可视化。例如，立即意识到构建失败是很有用的。

最简单的方法是在类似信息亭的配置中连接一个监视器，并使用一个指向您的构建服务器 Web 界面的 Web 浏览器。Jenkins 有许多插件可以提供适用于信息亭显示的简化作业概览。这些有时被称为信息辐射体。

将其他类型的硬件连接到构建状态也很常见，例如熔岩灯或彩色 LED 灯。

根据我的经验，这种显示可以使人们对构建服务器充满热情。不过，从长远来看，成功地展示一个有用的显示器比它最初看起来要棘手得多。屏幕可能会分散注意力。如果为了避免分心而将屏幕放在不容易看到的地方，那么显示的目的就落空了。

构建代码

熔岩灯与谨慎放置的屏幕相结合可能是一个有用的组合。熔岩灯通常不会点亮,因此不会分散注意力。当发生构建错误时,它会亮起,然后您知道您应该查看构建信息辐射器。熔岩灯甚至传达了一种建筑质量的历史记录形式。当熔岩灯亮起时,它会变暖,一段时间后,熔岩会在灯内移动。纠正错误后,灯会冷却下来,但热量会保留一段时间,因此熔岩会移动一段时间,这与纠正构建错误所花费的时间成正比。

认真对待构建错误

构建服务器可以根据需要发出错误和代码质量问题的信号;如果开发团队不关心这些问题,那么在通知和可视化方面的投资就白费了。

这不是单靠技术手段就能解决的。必须有一个每个人都同意的过程,达成共识的最简单方法是让这个过程对每个相关人员都有明显的好处。

问题的一部分是组织中的一切都在火上浇油。构建错误比生产错误更重要吗?如果代码质量措施估计需要数年时间才能提高代码库的质量,那么是否值得开始修复这些问题?

我们如何解决这些问题?

这里有一些想法:

- 不要过度使用代码质量指标。减少测试直到报告显示可修复的水平。在解决了最初的一组问题后,您可以再次添加测试。
- 确定问题的优先级。首先解决生产问题。然后修复构建错误。在问题得到解决之前,不要在损坏的代码之上提交新代码。

鲁棒性

虽然希望构建服务器成为持续交付管道中的焦点之一,但也要考虑构建和部署过程不应在构建服务器出现故障时停滞不前。出于这个原因,构建本身应该尽可能健壮并且可以在任何主机上重复。

第 5 章

这对于某些构建来说相当容易,例如 Maven 构建。即便如此,Maven 构建可能会出现许多使其不可移植的缺陷。

如果一个人没有幸运地在操作系统存储库中拥有所有可用的构建依赖项,那么基于 C 的构建可能很难移植。不过,鲁棒性通常是值得付出努力的。

概括

在本章中,我们快速浏览了构建代码的系统。

我们看过用 Jenkins 构建一个持续集成服务器。我们还检查了一些可能出现的问题,因为 DevOps 工程师的生活总是很有趣,但并不总是那么轻松。

在下一章中,我们将继续努力通过研究如何将测试集成到我们的工作流程中来生成最高质量的代码。

Machine Translated by Google

6

测试代码

如果我们要尽早并经常发布我们的代码,我们应该对其质量充满信心。因此,我们需要自动化回归测试。

本章探讨了一些软件测试框架,例如用于单元测试的JUnit和用于Web前端测试的Selenium。我们还将了解这些测试如何在我们的持续集成服务器 Jenkins 中运行,从而形成我们持续交付管道的第一部分。

测试对于软件质量来说非常重要,它本身就是一个很大的课题。

我们将在本章中关注这些主题:

- 如何使手动测试更容易且不易出错 · 各种类型的测试,例如单元测试,以及如何执行它们
在实践中
- 自动化系统集成测试

我们已经在上一章中了解了如何使用 Sonar 和 Jenkins 积累测试数据,我们将继续深入研究这个主题。

手动测试

即使测试自动化对 DevOps 的潜在好处比手动测试更大,但手动测试始终是软件开发的重要组成部分。

如果不出意外,我们将需要至少手动执行一次测试,以便使它们自动化。

测试代码

尤其是验收测试很难被替代,尽管已经有人尝试这样做。软件需求规格说明可能非常简洁,甚至对于开发实现这些需求的功能的人员来说也难以理解。在这些情况下,眼睛盯着球的质量保证人员是无价且不可替代的。

使手动测试更容易的事情与使自动化集成测试更容易的事情相同,因此在不同的测试策略之间可以实现协同作用。

为了拥有快乐的质量保证人员,您需要:

- 管理测试数据,主要是后端数据库的内容,以便测试重复运行时给出相同的结果
- 能够快速部署新代码以验证错误修复

这看起来很明显,但在实践中可能很难。也许您拥有无法复制到测试环境的大型生产数据库。也许它们包含需要受法律保护的最终用户数据。在这些情况下,在将数据部署到测试环境之前,您需要对数据进行去标识化处理并清除其中的任何个人详细信息。

每个组织都是不同的,所以除了 KISS 规则:“保持简单,愚蠢”之外,不可能在这个领域给出普遍有用的建议。

测试自动化的优缺点

当您与人交谈时,大多数人都对测试自动化的前景充满热情。想象一下等待我们所有的好处:

- 更高的软件质量 · 对我
- 们发布的软件将按预期运行的信心更高 · 减少单调乏味的繁琐的手动测试。

所有非常好的和令人向往的东西!

但在实践中,如果您花时间与拥有复杂多层产品的不同组织打交道,您会注意到人们在谈论测试自动化,但您也会注意到实践中测试自动化的可疑缺失。这是为什么?

如果您只是编译程序并在它们通过编译后部署它们,您可能会遇到糟糕的体验。软件测试对于程序在现实世界中可靠地运行是完全必要的。手动测试速度太慢,无法实现持续交付。因此,我们需要测试自动化才能成功实现持续交付。因此,让我们进一步调查围绕测试自动化的问题领域,看看我们是否可以找出改善情况的方法:

- 廉价测试的价值较低。

一个问题是生产成本相当低的测试自动化类型,单元测试,通常比其他类型的测试具有更低的感知价值。单元测试仍然是一种很好的测试类型,但手动测试可能会被认为在实践中暴露了更多的错误。然后可能会觉得没有必要编写单元测试。

- 很难创建与自动化相关的测试支架

集成测试。

虽然为单元测试编写测试支架或测试夹具并不是很困难,但随着测试支架变得更像生产,它往往会变得更难。这可能是因为缺乏硬件资源、许可、人力和

很快。

- 程序的功能随时间而变化,必须调整测试

因此,这需要时间和精力。

这使得测试自动化似乎只会让编写软件变得更加困难,而不会提供明显的好处。

在开发人员与运维人员没有密切关系的组织中尤其如此,即非面向 DevOps 的组织。如果其他人将不得不处理您的糟糕代码,而这些代码并没有真正按预期工作,那么开发人员就没有真正的成本。这不是一段健康的关系。这是 DevOps 旨在解决的核心问题。DevOps 方法遵循这个重复的规则:帮助不同角色的人更紧密地合作。在 Netflix 这样的组织中,敏捷团队全权负责其服务的成功、维护和中断。

- 很难编写在许多不同构建场景中可靠工作的健壮测试。

这样做的结果是开发人员倾向于在本地构建中禁用测试,以便他们可以不受干扰地使用分配给他们的功能。由于人们不参与测试,因此影响测试结果的更改逐渐出现,最终测试失败。

测试代码

构建服务器会发现构建错误,但现在没有人记得测试是如何进行的,可能需要几天时间才能修复测试错误。当测试失败时,构建显示将显示为红色,最终,人们将不再关心构建问题。其他人最终会解决问题。

- 只是很难编写好的自动化测试,期间。
确实很难创建良好的自动化集成测试。它也可能是有益的,因为您可以了解正在测试的系统的所有方面。
这些都是难题,尤其是因为它们大多源于人们的看法和关系。

没有万能药,但我建议采用以下策略:

- 利用人们对测试自动化的热情 · 不要设定不切实际的目标
- 循序渐进地工作

单元测试

单元测试是开发人员通常最关心的一种测试。
主要原因是,根据定义,单元测试测试系统中定义明确的部分,与其他部分隔离开来。因此,它们相对容易编写和使用。

许多构建系统都内置了对单元测试的支持,可以毫不费力地利用它。

例如,对于 Maven,有一个约定描述如何编写测试,以便构建系统可以找到它们、执行它们,并最终准备一份结果报告。编写测试基本上归结为编写测试方法,这些方法用源代码注释标记以将方法标记为测试。因为它们是普通的方法,所以它们可以做任何事情,但按照惯例,应该编写测试,以便它们不需要付出很大的努力即可运行。如果测试代码开始需要复杂的设置和运行时依赖项,我们就不再处理单元测试了。

在这里,单元测试和功能测试之间的区别可能会造成混淆。通常,相同的底层技术和库会在单元测试和功能测试之间重复使用。

这是一件好事,因为重用通常是好的,并且可以让您在从事另一个领域的工作时受益于您在一个领域的专业知识。尽管如此,它有时还是会让人感到困惑,时不时抬起眼睛看看自己是否在做正确的事情是值得的。

一般的 JUnit 和特别的 JUnit

您需要运行测试的东西。JUnit 是一个框架,允许您在 Java 代码中定义单元测试并运行它们。

JUnit 属于统称为xUnit 的测试框架系列。SUnit是这个家族的祖父,由 Kent Beck 于 1998 年为 Smalltalk 语言设计。

虽然 JUnit 是特定于 Java 的,但这些想法对于移植来说是足够通用的,例如,C#。C# 的相应测试框架被称为 NUnit,这有点缺乏想象力。N 源自 .NET,Microsoft 软件平台的名称。

在继续之前,我们需要以下一些术语。命名法并不特定于 JUnit,但我们将使用 JUnit 作为示例,以便更容易与定义相关联。

·**测试运行器**:测试运行器运行由xUnit 框架定义的测试。

JUnit 有一种从命令行运行单元测试的方法,而 Maven 使用一个名为 Surefire 的测试运行器。测试运行器还收集并报告测试结果。在 Surefire 的案例中,报告是 XML 格式的,这些报告可以由其他工具进一步处理,特别是可视化。

·**测试用例**:测试用例是最基本的测试定义类型。

创建测试用例的方式在不同的 JUnit 版本中略有不同。在早期版本中,您继承自 JUnit 基类;在最近的版本中,您只需要注释测试方法。这更好,因为 Java 不支持多重继承,您可能希望使用自己的继承层次结构而不是 JUnit 的继承层次结构。按照惯例,Surefire 还会定位类名中带有Test后缀的测试类。

·**测试夹具**:测试夹具是测试用例可以依赖的已知状态

on 以便测试可以具有明确定义的行为。创建这些是开发人员的责任。测试夹具有时也称为测试上下文。

测试代码

使用 JUnit,您通常使用@Before和@After注释来定义测试装置。不出所料,@Before 在测试用例之前运行,用于启动环境。 @After同样恢复状态如果有

需要。

有时, @Before和@After更形象地命名为Setup和Teardown 。由于使用了注解,该方法可以具有在该上下文中最直观的名称。

·**测试套件**:您可以将测试用例组合在测试套件中。测试套件通常是一组共享相同测试夹具的测试用例。

·**测试执行**:测试执行运行测试套件和测试用例。

在这里,所有先前的方面都结合在一起。找到测试套件和测试用例,创建适当的测试夹具,然后运行测试用例。

最后,收集并整理测试结果。

·**测试结果格式化器**:测试结果格式化器格式化测试结果输出以供人类使用。 JUnit 使用的格式非常通用,可以被其他不直接与 JUnit 关联的测试框架和格式化程序使用。因此,如果您有一些测试没有真正使用任何 xUnit 框架,您仍然可以通过提供测试结果 XML 文件在 Jenkins 中呈现测试结果而受益。由于文件格式为 XML,如果需要,您可以使用自己的工具生成它。

·**断言**:断言是xUnit 框架中的一种构造,可确保满足条件。如果不满足,则认为是错误,报测试错误。当断言失败时,测试用例通常也会终止。

JUnit 有许多可用的断言方法。以下是可用断言方法的示例:

· 检查两个对象是否相等: `assertEquals(str1, str2);`

· 检查条件是否为真:

```
assertTrue (val1 < val2);
```

· 检查条件是否为假:

```
assertFalse(val1 > val2);
```

一个 JUnit 示例

JUnit 得到 Java 构建工具的良好支持。一般而言,它将很好地用作 JUnit 测试框架的示例。

如果我们使用 Maven,按照惯例,它会期望在以下目录中找到测试用例:

```
/src/测试/java
```

嘲笑

模拟是指编写模拟资源以启用单元测试的做法。

有时,会使用“假”或“存根”等词。例如,使用来自数据库的 JSON 结构响应的中间件系统将“模拟”数据库后端以进行单元测试。否则,单元测试将需要数据库后端联机,可能还需要独占访问。这不方便。

Mockito 是 Java 的模拟框架,也已移植到 Python。

测试覆盖率

当您听到人们谈论单元测试时,他们通常会谈论测试覆盖率。

测试覆盖率是测试用例实际执行的应用程序代码库的百分比。

为了测量单元测试代码覆盖率,您需要执行测试并保持跟踪已执行或未执行的代码。

Cobertura 是用于 Java 的测试覆盖率测量实用程序,可以执行此操作。其他此类实用程序包括 jcoverage 和 Clover。

Cobertura 的工作方式是检测 Java 字节码,将自己的代码片段插入已编译的代码中。这些代码片段是在执行测试用例期间测量代码覆盖率时执行的。

通常假设 100% 的测试覆盖率是理想的。情况可能并非总是如此,人们应该意识到成本/收益的权衡。

测试代码

一个简单的反例是 Java 中的一个简单的 getter 方法：

```
私有 int 正值; void setPositiveValue(int  
x){ this.positiveValue=x;  
  
}  
  
int getPositiveValue(){ 返回正值;  
  
}
```

如果我们为这个方法编写测试用例,我们将获得更高的测试覆盖率。另一方面,在实践中,我们还没有取得任何成就。我们真正测试的唯一一件事是我们的 Java 实现没有错误。

另一方面,如果设置器更改为包括验证以检查该值不是负数,情况就会发生变化。只要方法包含某种逻辑,单元测试就很有用。

自动化集成测试

就所使用的基本技术而言,自动化集成测试在许多方面类似于单元测试。您可以使用相同的测试运行器和构建系统支持。与单元测试的主要区别在于涉及的模拟较少。

单元测试将简单地模拟从后端数据库返回的数据,而集成测试将使用真实数据库进行测试。数据库是您需要的测试资源类型以及它们可能出现的问题类型的一个很好的例子。

自动化集成测试可能非常棘手,您需要谨慎选择。

例如,如果您正在测试只读中间件适配器,例如用于数据库的 SOAP 适配器,则可以使用生产数据库副本进行测试。

您需要数据库内容是可预测和可重复的;否则,将很难编写和运行测试。

这里的附加值是我们使用的是生产数据副本。如果您要从头开始创建测试数据,它可能包含难以预测的数据,要求与手动测试相同。通过自动化集成测试,您需要比手动测试更多的自动化。对于数据库,这不必非常复杂。自动数据库备份和恢复是众所周知的操作。

自动化测试中的Docker

Docker 在构建自动化测试平台时非常方便。它添加了单元测试的一些功能,但是是在功能级别上。如果您的应用程序由集群中的多个服务器组件组成,您可以使用一组容器来模拟整个集群。Docker为集群提供了一个虚拟网络,明确了容器在网络层面是如何交互的。

Docker 还可以轻松地将容器恢复到已知状态。如果您的测试数据库位于 Docker 容器中,您可以轻松地将数据库恢复到与测试发生之前相同的状态。这类似于单元测试中After方法中的环境恢复。

Jenkins 持续集成服务器支持启动和停止容器,这在使用 Docker 测试自动化时非常有用。

使用 Docker Compose 运行您需要的容器也是一个有用的选项。

Docker 还很年轻,使用 Docker 进行测试自动化的某些方面可能需要不够优雅的胶水代码。

一个简单的示例可以是启动相互通信的数据库容器和应用程序服务器容器。启动容器的基本过程很简单,可以使用 shell 脚本或 Docker Compose 完成。

但是,既然我们要对已经启动的应用服务器容器进行测试,那么如何知道它是否正常启动呢?对于 WildFly 容器,除了观察日志输出中出现的字符串或可能轮询网络套接字之外,没有明显的方法来确定运行状态。无论如何,这些类型的 hack 都不是很优雅,而且编写起来很耗时。

不过,最终结果是值得付出努力的。

测试代码

阿奎利安

Arquillian 是测试工具的一个示例,它允许测试级别更接近集成测试,而不是单元测试和模拟允许的测试。 Arquillian 特定于 Java 应用程序服务器,例如 WildFly。 Arquillian 很有趣,因为它说明了在测试过程中如何更接近生产系统。您可以通过多种方式达到这样的近似值,而通往那里的道路充满了权衡取舍。

本书源码中有Arquillian的“hello world”风格演示
代码存档。

性能测试

性能测试是大型公共网站等开发的重要组成部分。

性能测试提出了与集成测试类似的挑战。我们需要一个类似于生产系统的测试系统,以便性能测试数据可用于预测实际生产系统性能。

最常用的性能测试是负载测试。通过负载测试,我们测量服务器的响应时间,同时性能测试软件为服务器生成合成请求。

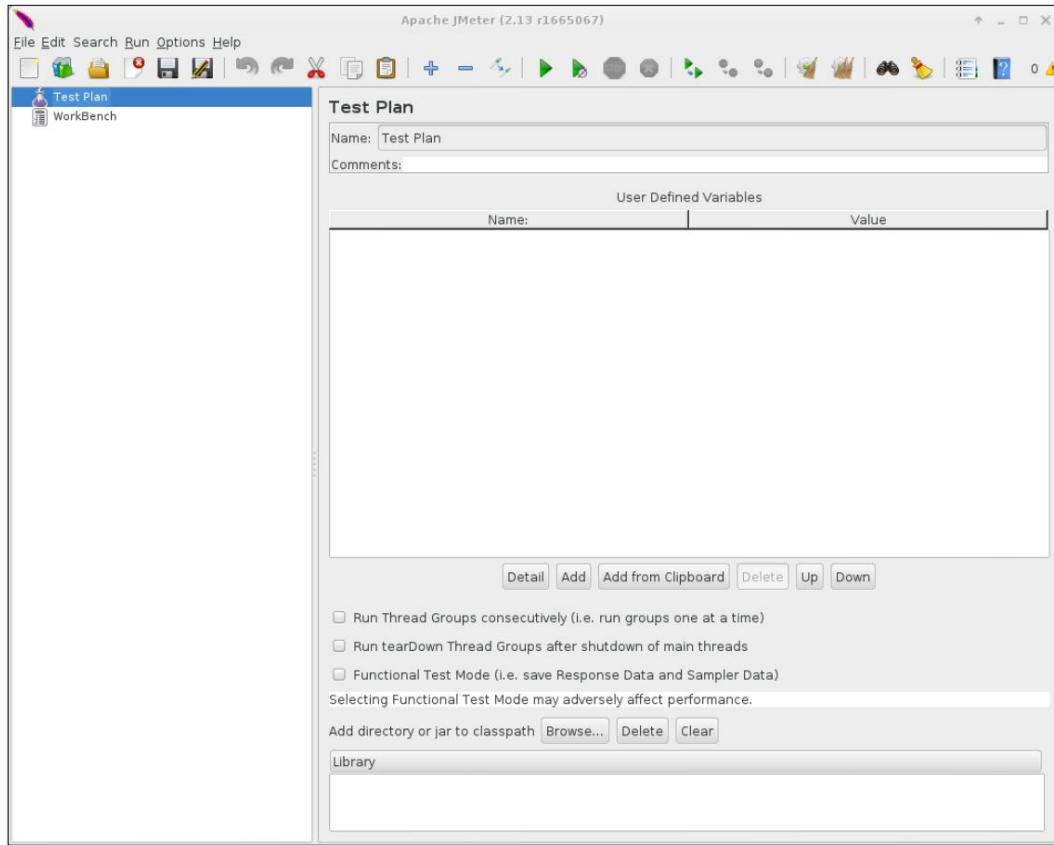
Apache JMeter 是一个用于测量性能的开源应用程序的示例。虽然它比其专有的同类产品(如 LoadRunner)更简单,但 JMeter 非常有用,而且简单并不是一件坏事。

JMeter 可以生成模拟负载并测量许多协议的响应时间,例如 HT、LDAP、SOAP 和 JDBC。

有一个 JMeter Maven 插件,因此您可以在构建过程中运行 JMeter。

JMeter 也可以在持续集成服务器中使用。 Jenkins 有一个插件,称为性能插件,可以执行 JMeter 测试场景。

理想情况下,持续集成服务器会将已构建的代码部署到类似生产的测试环境中。部署后,将执行性能测试并收集测试数据,如以下屏幕截图所示:



自动化验收测试

自动验收测试是一种确保您的测试从最终用户的角度来看是有效的方法。

Cucumber 是一个框架 ,其中测试用例以明文形式编写并与测试代码相关联。这称为行为驱动开发。 Cucumber 的最初实现是用 Ruby 编写的,但现在有许多不同语言的端口。

从 DevOps 的角度来看,Cucumber 的吸引力在于它旨在将不同的角色聚集在一起。 Cucumber 功能定义以对话式风格编写 ,无需编程技能即可实现。然后从描述中提取测试运行所需的硬数据并用于测试。

测试代码

虽然意图是好的,但在实施 Cucumber 时存在困难,这些困难可能不会立即显现出来。虽然行为规范的语言基本上是自由文本,但它们仍然需要简朴和形式化;否则,很难编写从描述中提取测试数据的匹配代码。这使得编写规范对本来应该编写规范的角色没有吸引力。然后发生的是程序员编写规范,他们经常不喜欢冗长而诉诸于编写普通的单元测试。

与许多事情一样,合作在这里至关重要。当开发人员和产品所有者一起以对每个相关人员都适用的方式编写规范时,Cucumber 可以发挥很好的作用。

现在,让我们看一下 Cucumber 的一个 “hello world”风格的小例子。

Cucumber 使用称为特征文件的纯文本文件,如下所示:

```
特征:加法
我想用袖珍计算器加法

场景:整数
* 我在计算器里输入了 4
* 我按添加
* 我在计算器中输入了 2
* 我按等于
* 屏幕上的结果应该是 6
```

功能描述与实现语言无关。描述 Cucumber 测试代码是在一个名为Gherkin 的词汇表中完成的。

如果您使用 Cucumber 的 Java 8 lambda 版本,测试步骤可能看起来像这样:

```
计算器计算;公共
MyStepdefs() {
    Given( 我已经在计算器中输入了 (\d+), (Integer i) -> {
        System.out.format( 输入的数字:%n\n , i); calc.push(i);});
```

```
When( I press (\w+), (String op) ->
    { System.out.format("operator entered: %n\n", op); calc.op(op); });
Then( 结果应该是 (\d+), (Integer i) -> { System.out.format("result: %n\n", i); assertThat(calc.result(), i); });
```

像往常一样,完整的代码可以在本书的源档案中找到。

这是一个简单的例子,但 Cucumber 的优点和缺点应该立即显而易见。功能描述具有很好的人类可读性。但是,您必须在测试代码中将字符串与正则表达式进行匹配。如果您的功能描述在措辞上有轻微的变化,您将不得不调整测试代码。

自动化图形用户界面测试

自动化 GUI 测试具有许多理想的特性,但也很困难。一个原因是用户界面在开发阶段往往会发生很大变化,并且按钮和控件会在 GUI 中移动。

老一代的 GUI 测试工具通常通过合成鼠标事件并将它们发送到 GUI 来工作。当一个按钮移动时,模拟的鼠标点击无处可去,测试失败。然后,随着 GUI 的变化来更新测试变得很昂贵。

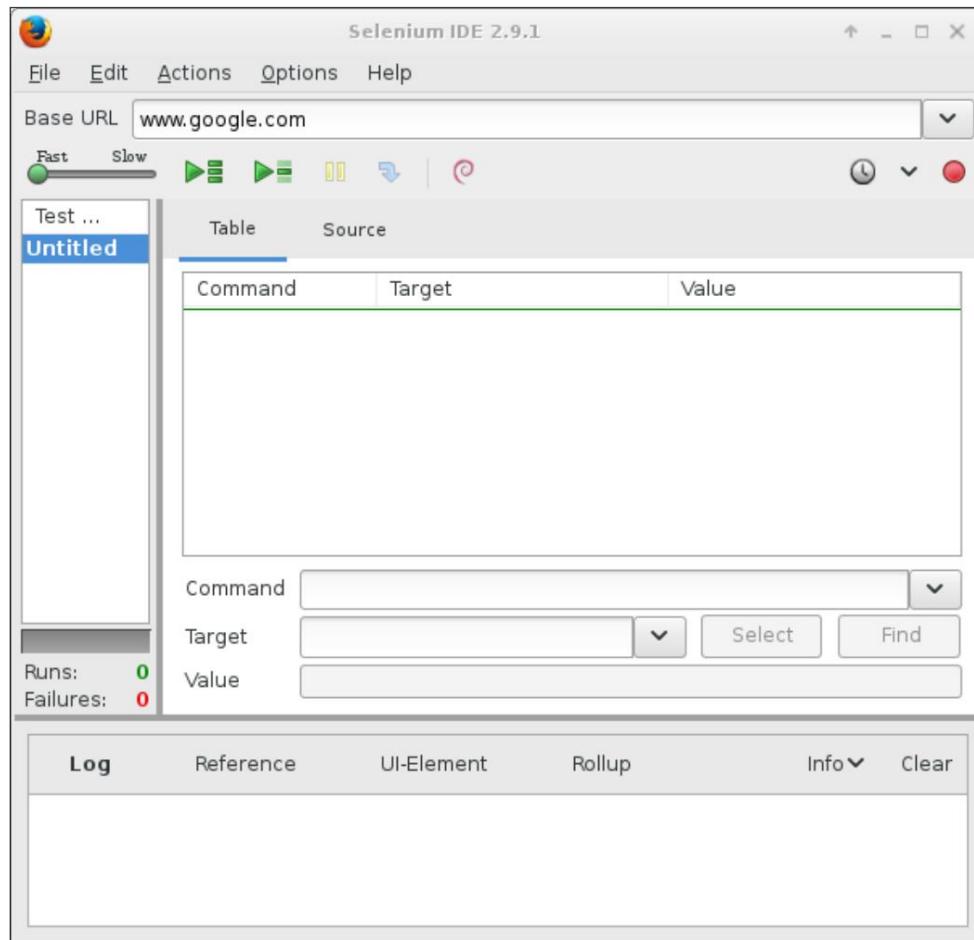
Selenium 是一个 Web UI 测试工具包,它使用一种不同的、更有效的方法。控制器配备了标识符,以便 Selenium 可以通过检查文档对象模型 (DOM) 而不是盲目地生成鼠标点击来找到控制器。

Selenium 在实践中工作得很好,并且多年来一直在发展。

Sikuli 测试框架采用了另一种方法。它使用计算机视觉框架 OpenCV 来帮助识别控制器,即使它们移动或改变外观也是如此。这对于测试本机应用程序 (例如游戏) 很有用。

测试代码

下面包含的屏幕截图来自 Selenium IDE。



在 Jenkins 中集成 Selenium 测试

Selenium 的工作方式是调用浏览器,将其指向运行应用程序的 Web 服务器,然后通过将自身集成到 JavaScript 和 DOM 层来远程控制浏览器。

开发测试时,可以使用两种基本方法:

- 在浏览器中记录用户交互,稍后保存生成的测试代码
重复使用
- 使用 Selenium 的测试 API 从头开始编写测试

许多开发人员更喜欢一开始就使用 Selenium API 将测试编写为代码,这可以与测试驱动的开发方法相结合。

不管测试是如何开发的,它们都需要在集成构建服务器中运行。

这意味着您需要在测试环境的某处安装浏览器。

这可能有点问题,因为构建服务器通常是无头的,也就是说,它们是不运行用户界面的服务器。

可以在构建服务器上的模拟桌面环境中包装浏览器。

更高级的解决方案是使用 Selenium Grid。顾名思义,Selenium Grid 提供了一个服务器,它提供了一些可供测试使用的浏览器实例。这使得并行运行多个测试以及提供一组不同的浏览器配置成为可能。

您可以从单一浏览器解决方案开始,然后在需要时迁移到 Selenium Grid 解决方案。

还有一个方便的 Docker 容器,它实现了 Selenium Grid。

JavaScript 测试

由于现在几乎每个产品通常都有 Web UI 实现,因此 JavaScript 测试框架值得特别提及:

- Karma 是用于 JavaScript 语言单元测试的测试运行器 · Jasmine 是一个类似 Cucumber 的行为测试框架 · Protractor 用于 AngularJS

Protractor 是一个不同的测试框架,在范围上类似于 Selenium,但针对流行的 JavaScript 用户界面框架 AngularJS 进行了优化。虽然似乎每天都有新的 Web 开发框架来来去去,但有趣的是,为什么在 Selenium 可用时存在像 Protractor 这样的测试框架,并且它也足够通用以测试 AngularJS 应用程序。

首先,Protractor 实际上在底层使用了 Selenium 网络驱动程序实现。

您可以用 JavaScript 编写 Protractor 测试,但如果不喜欢用 Java 等语言编写测试用例,也可以使用 JavaScript 为 Selenium 编写测试用例。

测试代码

事实证明,主要的好处是 Protractor 内化了有关 Angular 框架的知识,这是像 Selenium 这样的通用框架所无法真正拥有的。

AngularJS 有一个特定的模型/视图设置。其他框架使用其他设置,因为模型/视图设置不是 JavaScript 语言固有的东西 至少现在还不是。

Protractor 了解 Angular 的特性,因此可以更轻松地在具有特殊构造的测试代码中定位控制器。

测试后端集成点

后端功能 (例如 SOAP 和 REST 端点) 的自动化测试通常非常经济高效。后端接口往往相当稳定,因此相应的测试也比 GUI 测试需要更少的维护工作。

使用 soapUI 等工具也可以相当容易地编写测试,这些工具可用于编写和执行测试。这些测试也可以从命令行和 Maven 运行,这对于构建服务器上的持续集成非常有用。

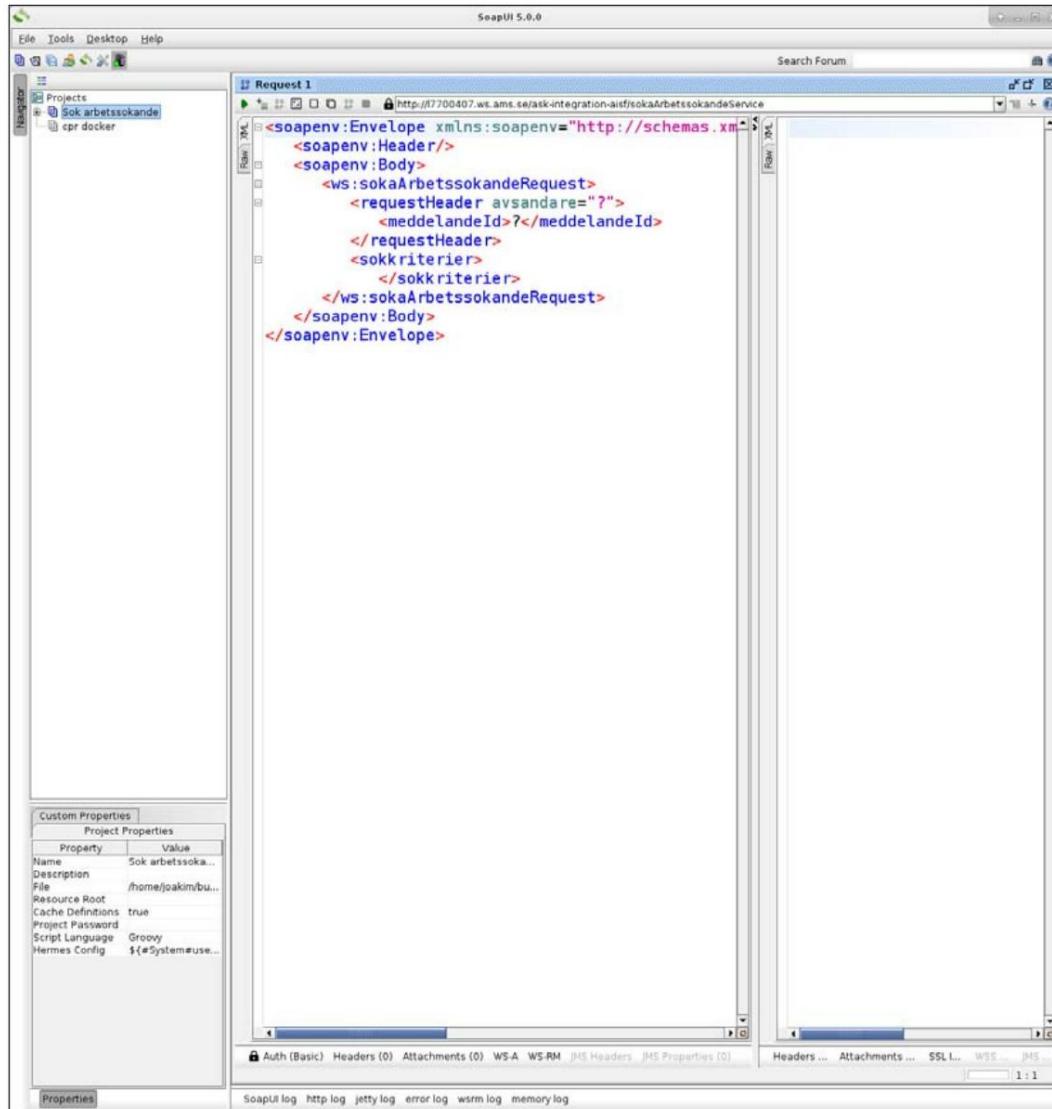
soapUI 是吸引多个不同角色的工具的一个很好的例子。构建测试用例的测试人员获得了一个结构良好的环境来编写测试并以交互方式运行它们。可以逐步构建测试。

开发人员可以在他们的构建中集成测试用例,而不必使用 GUI。
有 Maven 插件和命令行运行器。

命令行和 Maven 集成对于维护
也构建服务器。

此外,许可证是开源的,在单独的专有版本中增加了一些功能。开源性质使构建更加可靠。当构建失败时,由于许可证意外到期或浮动许可证用完而导致压力很大。

soapUI 工具有它的缺陷,但总的来说,它很灵活并且运行良好。
用户界面如下所示:



soapUI 用户界面相当简单。左侧有一个列出测试用例的树视图。可以选择单个测试或整个测试套件并运行它们。
结果显示在右侧区域中。

还值得注意的是,测试用例是用 XML 定义的。这使得将它们作为源代码存储库中的代码进行管理成为可能。这也使得有时可以在文本编辑器中编辑它们,例如,当我们需要执行全局搜索并替换已更改名称的标识符时 这正是我们在 DevOps 中喜欢的方式!

测试代码

测试驱动开发

测试驱动开发(TDD) 更加关注测试自动化。它因 90 年代的极限编程运动而流行起来。

TDD 通常被描述为一系列事件,如下所示:

·实施测试:顾名思义,您先编写测试,然后再编写代码。一种看待它的方式是你实现了要开发的代码的接口规范,然后通过编写代码来进步。为了能够编写测试,开发人员必须找到所有相关的需求规范、用例和用户故事。

将重点从编码转移到理解需求可能有利于正确实施它们。

·验证新测试是否失败:新添加的测试应该失败,因为目前还没有正确实现行为的方法,只有编写测试所需的存根和接口。运行测试并验证它是否失败。 ·编写实现测试功能的代码:我们编写的代码没有

但必须特别优雅或高效。最初,我们只是想让新测试通过。

·验证新测试是否与旧测试一起通过:当新测试通过时,我们就知道我们已经正确地实现了新功能。

由于旧测试也通过了,我们没有破坏现有功能。

·重构代码: “重构”一词具有数学根源。在编程,这意味着清理代码,除其他外,使其更易于理解和维护。我们需要重构,因为我们
在开发的早期作弊了。

TDD 是一种非常适合 DevOps 的开发风格,但不一定是唯一的一种。主要好处是您可以获得可用于持续集成测试的良好测试套件。

REPL 驱动的开发

虽然 REPL 驱动的开发不是一个广为人知的术语,但它是 我最喜欢的开发风格,并且对测试有特殊的影响。这种开发风格在使用解释型语言 (例如 Lisp、Python、Ruby 和 JavaScript) 时非常常见。

当您使用 Read Eval Print Loop (REPL) 时,您将编写独立且不依赖于全局状态的小函数。

即使您编写函数,也会对其进行测试。

这种开发风格与 TDD 略有不同。重点是编写没有副作用或副作用很少的小函数。这使得代码易于理解,而不是像在 TDD 中那样在编写功能代码之前编写测试用例。

您可以将这种开发方式与单元测试相结合。由于您也可以使用 REPL 驱动的开发来开发您的测试,因此这种组合是一种非常有效的策略。

完整的测试自动化场景

我们已经研究了许多使用测试自动化的不同方法。

将各个部分组装成一个有凝聚力的整体可能会让人望而生畏。

在本节中,我们将查看一个完整的测试自动化示例,从我们组织的用户数据库 Web 应用程序 Matangle 开始。

您可以在本书随附的源代码包中找到源代码。

该应用程序由以下几层组成:

- 网络前端
- JSON/REST 服务接口 · 应用后端层 · 数
据库层

测试代码在执行过程中会经历以下几个阶段:

- 后端代码的单元测试 · Web 前端的功能测
试,使用 Selenium web 执行
测试框架 · JSON/REST
接口的功能测试,使用 soapUI 执行

所有测试按顺序运行,当所有测试都成功时,结果可以用作决定查看应用程序堆栈是否足够健康以部署到测试环境的基础,其中手动测试

可以开始。

测试代码

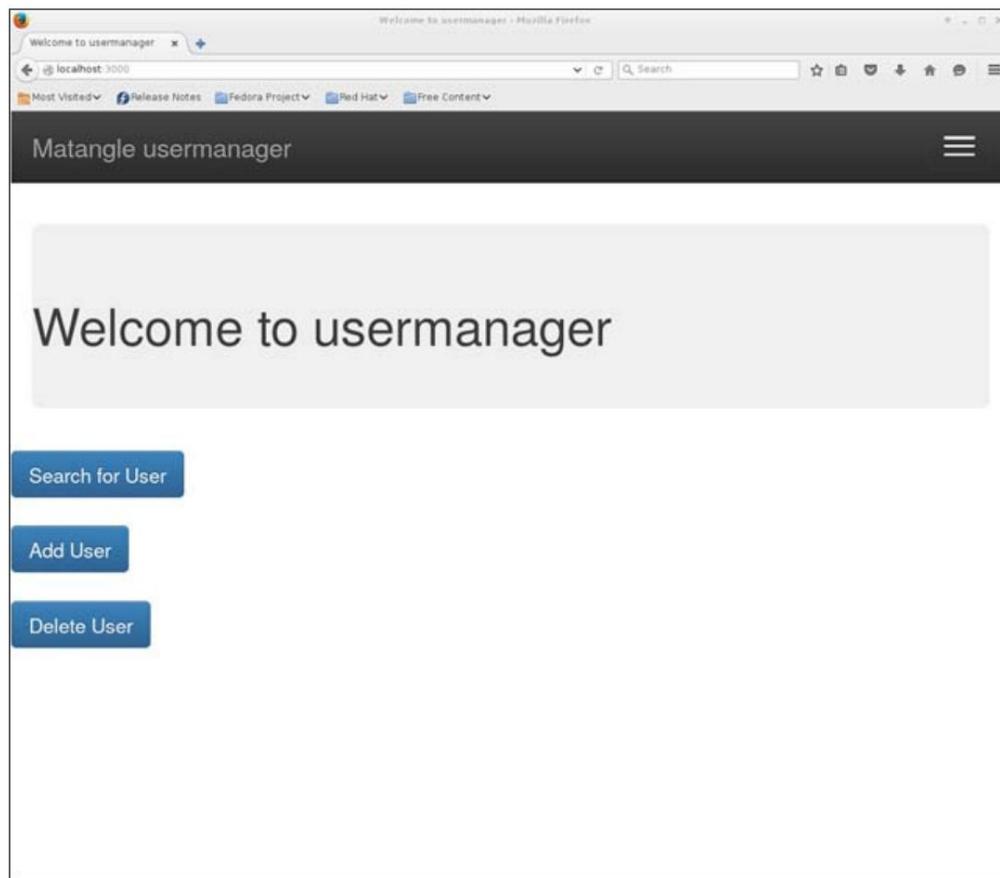
手动测试我们的网络应用程序

在我们可以自动化一些有用的东西之前,我们需要了解我们将自动化的细节。我们需要某种形式的测试计划。

下面,我们有一个 Web 应用程序的测试计划。它详细说明了如果没有可用的测试自动化,人工测试人员需要手动执行的步骤。它类似于真实的测试计划,除了真实的计划通常会有更多的手续。在我们的例子中,我们将直接进入相关测试的细节:

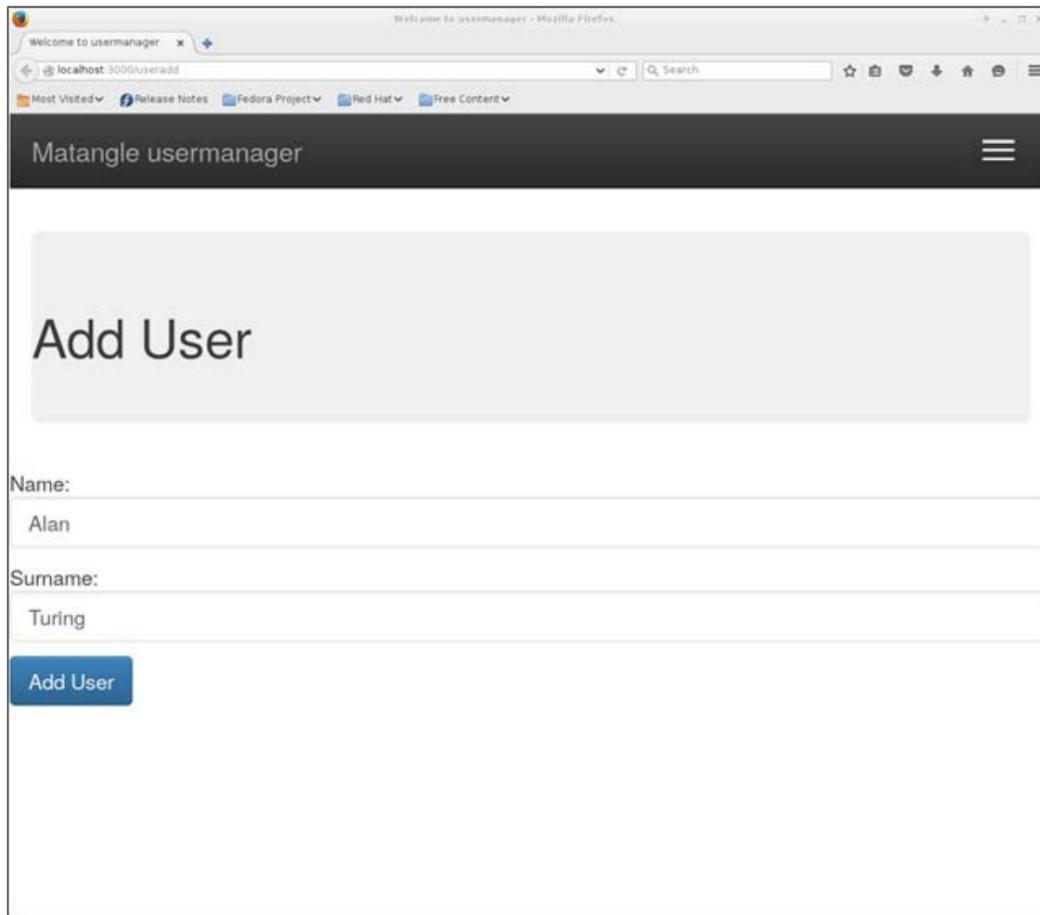
1. 开始新的测试。这会将数据库后端重置为已知状态并设置测试场景,以便手动测试可以从已知状态继续进行。

测试人员将浏览器指向应用程序的起始 URL:



2. 单击添加用户链接。

3. 添加用户：

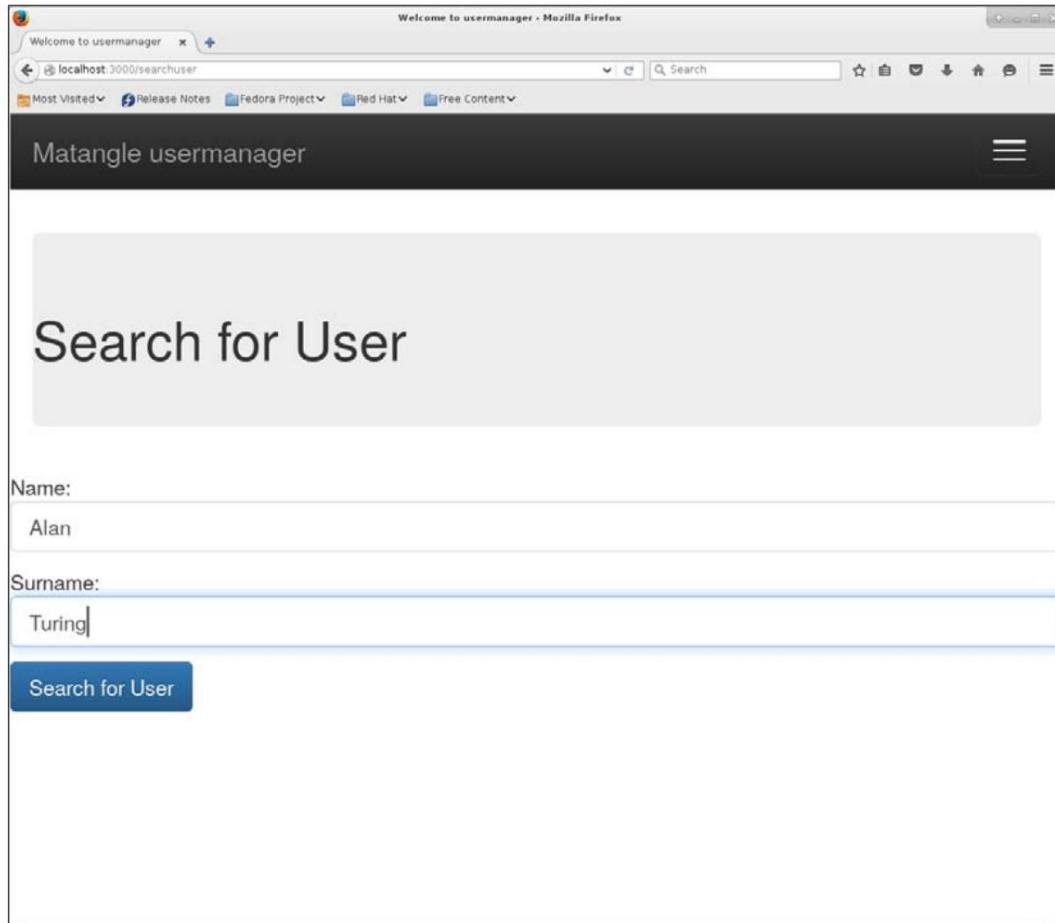


在我们的测试用例中输入用户名 Alan Turing。

4. 保存新用户。将显示一个成功页面。

测试代码

5. 通过执行搜索验证用户是否已正确添加：



单击“搜索用户”链接。寻找艾伦图灵。验证Alan是否出现在结果列表中。

虽然此时读者可能对应用程序的复杂性印象不深,但如果我们要能够自动化该场景,这就是我们需要处理的详细程度,我们在这里研究的正是这种复杂性。

运行自动化测试

该测试在源包中有多种形式。

要运行第一个 ,您需要安装 Firefox。

选择名为autotest_v1 的那个 ,然后从命令行运行它：

```
autotest_v1/bin/autotest.sh
```

如果一切顺利,您将看到一个 Firefox 窗口打开 ,您之前手动执行的测试将自动完成。您填写的值和您手动点击的链接都会自动完成。

这还不是万无一失的 ,因为您安装的 Firefox 版本可能不兼容 ,或者依赖项存在其他问题。
对此类问题的自然反应是依赖管理 ,我们很快就会看到使用 Docker 进行依赖管理的变体。

发现错误

现在我们将引入一个错误 ,让测试自动化系统找到它。

作为练习 ,在测试源中找到字符串 “Turing” 。将出现的其中一个更改为 “Tring” 或其他一些印刷错误。只换一个 ;否则 ,验证码会认为没有错误 ,一切正常 !

再次运行测试 ,发现错误被自动测试系统发现了。

测试演练

现在我们已经运行了测试并验证了它们是否有效。我们还验证了他们能够发现我们创建的错误。

实现是什么样的 ?代码很多 ,转载到书里用处不大。不过 ,对代码进行概述并查看一些代码片段是很用的。

打开autotest_v1/test/pom.xml文件。它是一个 Maven 项目对象模型文件 ,测试使用的所有插件都在这里设置。 Maven POM 文件是声明性的 XML 文件 ,测试步骤是逐步命令式指令 ,因此在后一种情况下 ,使用 Java。

测试代码

顶部有一个属性块，其中保留了依赖版本。没有必要打破版本；在这种情况下，它已被用于使 POM 文件的其余部分不那么依赖于版本：

```
<属性>
<junit.version>XXX</junit.version>
<selenium.version>XXX</selenium.version>
<cucumber.version>XXX</cucumber.version>
...
</属性>
```

以下是 JUnit、Selenium 和 Cucumber 的依赖项：

```
<依赖关系>
<groupId>junit</groupId>
<artifactId>junit</artifactId> <版本>$
{junit.version}</version> </dependency>

<依赖关系>
<groupId>org.seleniumhq.selenium</groupId>
<artifactId>selenium-java</artifactId> <版本>${selenium.version}
</version> </dependency>

<dependency>
<groupId>info.cukes</groupId>
<artifactId>cucumber-core</artifactId> <版本>$
{cucumber.version}</version> </dependency>

<依赖关系>
<groupId>info.cukes</groupId>
<artifactId>cucumber-java</artifactId> <版本>$
{cucumber.version}</version> </dependency>

<dependency>
<groupId>info.cukes</groupId>
<artifactId>cucumber-junit</artifactId> <版本>$
{cucumber.version}</version> </dependency>
```

要根据 Cucumber 方法定义测试,我们需要一个以人类可读语言描述测试步骤的功能文件。这个特性文件对应我们之前的手动测试的测试计划:

```
功能:管理用户
作为管理员
我希望能够
- 创建用户
- 搜索用户
- 删除用户

场景:创建命名用户
给定一个名为 “Alan”的用户
还有 “图灵”这个姓氏
当管理员单击 “添加用户”时
然后应该添加用户

场景:搜索指定用户
给定一个名为 “Alan”的用户
还有 “图灵”这个姓氏
当管理员单击 “搜索用户”时
然后应该显示用户 Alan Turing

场景:删除命名用户
给定一个名为 “Alan”的用户
还有 “图灵”这个姓氏
当管理员单击 “删除用户”时
那么用户应该被删除
```

特征文件主要是带有少量机器可读标记元素的纯文本。用正则解析场景的明文就看相应的测试代码了。

也可以将功能文件本地化为您自己团队中使用的语言。这可能很有用,因为功能文件可能是由不习惯英语的人编写的。

该功能需要实际的具体代码才能执行,因此您需要某种方式将功能绑定到代码。

测试代码

您需要一个带有一些注释的测试类,以使 Cucumber 和 JUnit 协同工作:

```
@RunWith(黄瓜类)
@Cucumber.Options( glue =
    matangle.glue.manageUser ,features =  features/
    manageUser.feature ,format={ pretty ,  html:target/Cucumber }

)
```

在此示例中,Cucumber 测试类的名称按照惯例具有 Step 后缀。

现在,您需要将测试方法绑定到特性场景,并且需要某种方法从特性描述中挑选出测试方法的参数。对于 Java Cucumber 实现,这主要是通过 Java 注释完成的。这些注释对应于特征文件中使用的关键字:

```
@Given( .+一个名为 (.+) 的用户) public void addUser(String
    name){
```

在这种情况下,不同的输入存储在成员变量中,直到整个用户界面事务准备就绪。操作顺序由它们在特征文件中出现的顺序决定。

为了说明 Cucumber 可以有不同的实现,本书的源代码包中还有一个 Clojure 示例。

到目前为止,我们已经看到我们需要一些用于 Selenium 和 Cucumber 的库来运行测试,以及 Cucumber 特性描述符如何绑定到我们的测试代码类中的方法。

下一步是检查 Cucumber 测试如何执行 Selenium 测试代码。

Cucumber 测试步骤大多在带有 View 后缀的类中调用带有 Selenium 实现细节的类。这不是技术上的必要,但它使测试步骤类更具可读性,因为 Selenium 框架的细节保存在一个单独的类中。

Selenium 框架负责测试代码和浏览器之间的通信。视图类是我们正在自动化的网页的抽象。

视图代码中有成员变量对应于 HTML 控制器。

您可以使用 Selenium 框架的注释来描述测试代码成员变量和 HTML 元素之间的绑定,如下所示:

```
@FindBy(id =  name ) private WebElement nameInput;
@FindBy(id =  surname ) private WebElement surnameInput;
```

然后,测试代码使用成员变量来自动执行人类测试人员使用测试计划遵循的相同步骤。将类划分为视图类和步骤类也使步骤类与测试计划的相似性更加明显。当涉及测试和质量保证的人员使用代码时,这种关注点分离很有用。

要发送字符串,您可以使用一种方法来模拟用户在键盘上的键入:

```
nameInput.clear();
nameInput.sendKeys(value);
```

有许多有用的方法,例如click(),它将模拟用户单击控件。

使用 Docker 处理棘手的依赖关系

因为我们在测试代码示例中使用了 Maven,所以它处理了除浏览器之外的所有代码依赖项。虽然您可以清楚地将 Firefox 等浏览器部署到与 Maven 兼容的存储库,并按照您的想法处理测试依赖性,但这通常不是浏览器处理此问题的方式。浏览器是挑剔的生物,在不同版本中表现出截然不同的行为。我们需要一种机制来运行许多不同的浏览器

不同的版本。

幸运的是,有这样一种机制,称为 Selenium Grid。由于 Selenium 具有可插入的驱动程序架构,您基本上可以在客户端服务器架构中对浏览器后端进行分层。

要使用 Selenium Grid,您必须首先确定您希望服务器部分如何运行。

虽然最简单的选择是使用在线提供商,但出于教学原因,这不是我们将在此处探讨的选项。

有一个autotest_seleniumgrid目录,其中包含一个包装器,用于使用 Docker 运行测试以启动本地 Selenium Grid。您可以通过运行包装器来试用该示例。

有关如何运行 Selenium Grid 的最新信息可在该项目的 GitHub 页面上找到。

Selenium Grid 具有分层架构,要设置它,您需要三个部分:

- 测试代码中的RemoteWebDriver实例。这将是界面到硒网格。
- Selenium Hub,可以看作是浏览器实例的代理。

测试代码

- Firefox 或 Chrome 网格节点。这些是 Hub 将代理的浏览器实例。

设置 RemoteWebDriver 的代码可能如下所示：

```
DesiredCapabilities 能力 = new DesiredCapabilities(); capabilities.setPlatform(Platform.LINUX);
capabilities.setBrowserName( "Firefox" );

capabilities.setVersion( 35 );
driver = new RemoteWebDriver( new
URL( "http://localhost:4444" ), capabilities );
```

该代码要求具有一组特定功能的浏览器实例。系统会尽力满足。

仅当有一个连接了 Firefox 节点的 Selenium Grid Hub 运行时，该代码才能工作。

以下是使用 Docker 包装启动 Selenium Hub 的方法：

```
docker run -d -p 4444:4444 --name selenium-hub 硒/集线器
```

以下是启动 Firefox 节点并将其附加到 Hub 的方法：

```
docker run -d --link selenium-hub:hub selenium/节点-firefox
```

概括

测试代码演练到此结束。当您阅读代码时，您可能只想使用所说明的想法的一个子集。也许 Cucumber 方法并不真正适合您，或者您更看重简洁明了的代码，而不是示例中使用的分层抽象。这是一种自然而合理的反应。采纳这些想法，以便它们为您的团队工作。此外，在决定适合您的方法时，请查看源包中可用的其他测试代码风格！

软件测试是一个庞大的主题，可以填满大量的内容。在本章中，我们调查了一些可用的不同类型的测试。我们还研究了在持续集成服务器中使用自动化软件测试的具体方法。我们使用了 Jenkins 和 Maven 以及 JUnit 和 JMeter。虽然这些工具是面向 Java 的，但这些概念很容易转换到其他环境。

现在我们已经构建并测试了我们的代码，我们将在下一章开始部署我们的代码。

7

部署代码

现在代码已经构建和测试完毕,我们需要将它部署到我们的服务器上,以便我们的客户可以使用新开发的功能!

这个领域有许多相互竞争的工具和选项,适合您和您的组织的工具和选项取决于您的需求。

我们将通过展示如何在不同场景中部署示例应用程序来探索 Puppet、Ansible、Salt、PalletOps 等。这些工具中的任何一个都有一个庞大的补充服务和工具生态系统,因此要掌握它绝非易事。

在整本书中,我们遇到了一些已经存在的不同部署系统的各个方面。我们查看了 RPM 和.deb 文件以及如何使用 rpm 命令构建它们。我们了解了各种 Java 工件以及 Maven 如何使用二进制存储库的想法,您可以在其中部署版本化的工件。

在本章中,我们将重点介绍安装二进制包及其使用配置管理系统的配置。

为什么会有这么多的部署系统?

关于软件包安装和在实际服务器上配置它们的选项多得令人眼花缭乱,更不用说部署客户端代码的所有方法了。

让我们首先检查一下我们试图解决的问题的基础知识。

部署代码

我们有一个典型的企业应用程序 ,其中包含许多不同的高级组件。我们不需要为了开始推理这个领域中存在的挑战而使场景过于复杂。

在我们的场景中 ,我们有 :

- 网络服务器
- 应用程序服务器 · 数据库
- 服务器

如果我们只有一台物理服务器 ,并且需要担心每年发布一次的这几个组件 ,我们可以手动安装软件并完成任务。这将是处理这种情况的最具成本效益的方法 ,尽管手动工作很无聊且容易出错。

但是 ,期望在现实中符合这种简化的发布周期是不合理的。大型组织更有可能拥有数百台服务器和应用程序 ,并且它们的部署方式各不相同 ,具有不同的要求。

管理现实世界显示的所有复杂性是困难的 ,因此有许多不同的解决方案以不同的方式做基本相同的事情开始变得有意义。

无论执行我们代码的基本单元是什么 ,无论是物理服务器、虚拟机、某种形式的容器技术 ,还是它们的组合 ,我们都面临着几个挑战需要应对。我们现在来看看它们。

配置基本操作系统

必须以某种方式处理基本操作系统的配置。

通常 ,我们的应用程序堆栈对基本操作系统有微妙或不那么微妙的要求。一些应用程序堆栈 ,如 Java、Python 或 Ruby ,使这些操作系统要求不太明显 ,因为这些技术竭尽全力提供跨平台功能。在其他时候 ,操作系统要求在更大程度上是显而易见的 ,例如当您使用低级混合硬件和软件集成时 ,这在电信行业中很常见。

有许多现有的解决方案可以解决这个基本问题。一些系统使用裸机（或裸机）方法，从头开始安装所需的操作系统，然后安装组织对其服务器所需的所有基本依赖项。此类系统包括，例如，Red Hat Satellite 和 Cobbler，它们以类似的方式工作，但更轻量级。

Cobbler 允许您使用 dhcpcd 通过网络启动物理机或虚拟机。然后 DHCP 服务器可以允许您提供符合网络引导的图像。

启动网络引导映像时，它会联系 Cobbler 以检索将要安装的软件包，以创建新的操作系统。例如，可以根据目标机器的网络 MAC 地址在服务器上决定安装哪些包。

今天非常流行的另一种方法是提供可以在机器之间重复使用的基本操作系统映像。 AWS、 Azure 或 OpenStack 等云系统以这种方式工作。当您向云系统请求新的虚拟机时，它是使用现有映像作为基础创建的。 Docker 等容器系统也以类似的方式工作，您可以在其中声明基础容器镜像，然后描述您想要为自己的镜像制定的更改。

描述集群必须有一种方法来描述集群。

如果您的组织只有一台机器和一个应用程序，那么您可能不需要描述应用程序的集群部署是什么样的。不幸的是（或者幸运的是，这取决于您的看法），现实情况通常是您的应用程序分布在一组虚拟或物理机器上。

我们在本章中使用的所有系统都以不同的方式支持这个想法。

Puppet 有一个广泛的系统，允许机器具有不同的角色，这反过来意味着一组包和配置。 Ansible 和 Salt 也有这些系统。基于容器的 Docker 系统有一个新兴的基础设施，用于描述连接在一起的容器集以及可以接受和部署此类集群描述符的 Docker 主机。

AWS 等云系统也有集群部署的方法和描述符。

簇描述符通常也用于描述应用层。

[部署代码](#)

将包传送到系统

必须有一种方法可以将包传送到系统。

许多应用程序可以作为包安装,配置管理系统将它们安装在目标系统上而不加修改。 RPM 和deb 等包系统具有有用的功能,例如通过为包中的所有文件提供校验和来验证包提供的文件在目标系统上未被篡改。出于安全原因和调试目的,这很有用。包交付通常使用操作系统设施完成,例如基于 Red Hat 的系统上的yum包通道,但有时,配置管理系统也可以使用自己的设施交付包和文件。这些工具通常与操作系统的包通道一起使用。

必须有一种方法来管理独立于已安装包的配置。

显然,配置管理系统应该能够管理我们应用程序的配置。这很复杂,因为配置方法在应用程序之间千差万别,不管为统一该领域所做的许多努力。

配置应用程序的最常见和最灵活的系统依赖于基于文本的配置文件。还有其他几种方法,例如使用提供 API 的应用程序来处理配置 (例如命令行界面) ,或者有时通过数据库设置来处理配置。

根据我的经验,基于文本文件的配置系统产生的麻烦最少,至少应该优先用于内部代码。有许多方法可以管理基于文本的配置。您可以在 Git 等源代码处理系统中管理它们。有许多工具可以简化损坏配置的调试,例如diff。如果手头紧,可以使用远程文本编辑器 (如 Emacs 或 Vi)直接在服务器上编辑配置。

通过数据库处理配置的灵活性要低得多。这可以说是一种反模式,通常发生在开发团队和运营团队之间心理分歧太大的组织中,而这正是我们旨在通过 DevOps 解决的问题。在数据库中处理配置会使应用程序堆栈更难运行。您甚至需要一个工作数据库来启动应用程序。

第七章

出于类似的原因,通过命令式命令行 API 管理配置设置也是一种可疑的做法,但有时会有所帮助,尤其是当 API 用于管理基于文本的底层配置时。许多配置管理系统,例如 Puppet,都依赖于能够管理声明式配置。如果我们通过其他机制(例如命令行命令式 API)管理配置状态,Puppet 将失去许多优势。

甚至管理基于文本的配置文件也可能很麻烦。应用程序可以通过多种方式发明自己的配置文件格式,但有一组流行的基本文件格式。此类文件格式包括 XML、YML、JSON 和 INI。

通常,配置文件不是静态的,因为如果它们是静态的,您可以像任何二进制工件一样使用包系统部署它们。

通常,应用程序配置文件需要基于某种模板文件,稍后实例化为适合部署应用程序的机器的形式。

一个例子可能是应用程序的数据库连接器描述符。如果将应用程序部署到测试环境,您希望连接器描述符指向测试数据库服务器。反之亦然,如果您要部署到生产服务器,您希望连接器指向生产数据库服务器。

另外,一些组织试图通过管理他们的 DNS 服务器来处理这种情况,例如示例数据库 DNS 别名 database.yourorg.com 根据环境解析到不同的服务器。域 yourorg.com 应该替换为您组织的详细信息,当然还有数据库服务器。

能够根据环境使用不同的 DNS 解析器是一种有用的策略。然而,开发人员可能很难在他或她自己的开发机器上使用等效机制。在开发机器上运行私有 DNS 服务器可能很困难,管理本地主机文件也很麻烦。在这些情况下,使应用程序在应用程序级别具有数据库主机和其他后端系统的可配置设置可能会更简单。

很多时候,可以完全忽略实际配置文件格式的细节,只依赖配置系统的模板管理系统。

这些通常通过对占位符使用特殊语法来工作,在为将部署应用程序的具体服务器创建具体配置文件时,占位符将被配置管理系统替换。您可以对所有基于文本的配置文件使用完全相同的基本思想,有时甚至可以对二进制文件使用完全相同的基本思想,尽管应尽可能避免此类黑客攻击。

部署代码

XML 格式具有可用于管理配置的工具和基础结构,而且 XML 确实是一种流行的配置文件格式。例如,有一种特殊的语言 XSLT 可以将 XML 从一种结构形式转换为另一种结构形式。这在某些情况下非常有用,但在实践中的使用比人们预期的要少。简单的模板宏替换方法会让您走得更远,并且具有适用于几乎所有基于文本的配置格式的额外好处。 XML 也相当冗长,这也使其在某些圈子中不受欢迎。 YML 可以被看作是对 XML 冗长的反应,并且可以完成与 XML 相同的许多事情,而且输入更少。

一些文本配置系统的另一个值得一提的有用特性是基本配置文件可以包含其他配置文件的想法。这方面的一个例子是标准的 Unix sudo 工具,它在 /etc/sudoers 文件中有其基本配置,但它允许通过包含目录 /etc/sudoers.d 中已安装的所有文件来进行本地自定义。

这非常方便,因为您可以提供一个新的 sudoer 文件,而不必太担心现有的配置。这允许更大程度的模块化,并且当应用程序允许时,它是一种方便的模式。

虚拟化堆栈拥有自己的内部服务器场的组织倾向于大量使用虚拟化来封装其应用程序的不同组件。

根据您的要求,有许多不同的解决方案。

虚拟化解决方案提供具有虚拟硬件 (例如网卡和 CPU) 的虚拟机。虚拟化和容器技术有时会混淆,因为它们有一些相似之处。

您可以使用虚拟化技术来模拟与您实际拥有的硬件完全不同的硬件。这通常称为仿真。

如果你想在你的开发机器上模拟手机硬件以便你可以测试你的移动应用程序,你可以使用虚拟化来模拟设备。底层硬件越接近目标平台,仿真器在仿真时的效率就越高。例如,您可以使用 QEMU 模拟器来模拟 Android 设备。如果您在基于 x86_64 的开发人员机器上模拟 Android x86_64 设备,则模拟将比在基于 x86_64 的开发人员机器上模拟基于 ARM 的 Android 设备更有效。

对于服务器虚拟化,您通常对仿真的可能性并不真正感兴趣。您反而对封装应用程序的服务器组件感兴趣。例如,如果服务器应用程序组件开始失控并消耗不合理数量的 CPU 时间或其他资源,您不希望整个物理机器完全停止工作。

这可以通过在具有 64 个内核的机器上创建一个可能具有两个内核的虚拟机来实现。只有两个内核会受到失控应用程序的影响。内存分配也是如此。

基于容器的技术提供与虚拟化技术类似的封装程度和对资源分配的控制。不过,容器通常不提供虚拟化的仿真功能。这不是问题,因为我们很少需要对服务器应用程序进行仿真。

抽象底层硬件并在不同竞争虚拟机之间仲裁硬件资源的组件称为管理程序。

管理程序可以直接在硬件上运行,在这种情况下它被称为裸机管理程序。否则,它在操作系统内核的帮助下在操作系统内部运行。

VMware 是专有的虚拟化解决方案,存在于桌面和服务器管理程序变体中。它在许多组织中得到很好的支持和使用。服务器变体有时会更改名称;目前,它称为 VMware ESX,它是一个裸机管理程序。

KVM 是 Linux 的虚拟化解决方案。它在 Linux 主机操作系统中运行。由于它是一个开源解决方案,它通常比专有解决方案便宜得多,因为没有每个实例的许可成本,因此在拥有大量虚拟化的组织中很受欢迎。

Xen 是另一种类型的虚拟化,除其他功能外,它还具有半虚拟化。半虚拟化建立在这样的想法之上,即如果可以使客户操作系统使用修改后的内核,它可以以更高的效率执行。通过这种方式,它介于完全 CPU 仿真(使用完全独立的内核版本)和基于容器的虚拟化(其中

使用主机内核。

VirtualBox 是 Oracle 的开源虚拟化解决方案。它在开发人员中很受欢迎,有时也用于服务器安装,但很少大规模使用。在开发机器上使用 Microsoft Windows 但希望在本地模拟 Linux 服务器环境的开发人员通常会发现 VirtualBox 很方便。

同样,在工作站上使用 Linux 的开发人员发现模拟 Windows 机器很有用。

部署代码

不同类型的虚拟化技术的共同点是它们都提供 API 以实现虚拟机管理的自动化。libvirt API 就是这样一种 API,可以与多种不同的底层管理程序一起使用,例如 KVM、QEMU、Xen 和 LXC

在客户端执行代码

此处描述的几个配置管理系统允许您重用节点描述符以在匹配的节点上执行代码。这有时很方便。例如,您可能想在所有面向公共 Internet 的 HTTP 服务器上运行目录列表命令,可能出于调试目的。

在Puppet生态系统中,这个命令执行系统叫做Marionette Collective,简称MCollective。

关于练习的说明使用 Docker 来管理基本操作系统来尝试

各种部署系统非常容易,我们将在其中进行实验。这是一种节省时间的方法,可以在开发和调试特定于特定部署系统的部署代码时使用。然后,此代码将用于在物理机或虚拟机上进行部署。

我们将首先尝试在本地部署模式下通常可用的每种不同的部署系统。更进一步,我们将看到如何模拟一个系统的完整部署,其中多个容器共同构成一个虚拟集群。

如果可能,我们将尝试使用官方 Docker 镜像,但有时没有,有时官方镜像消失,就像官方 Ansible 镜像一样。这就是快速发展的 DevOps 世界中的生活,无论好坏。

然而,应该注意的是,Docker 在模拟完整操作系统时有一些局限性。有时,容器必须以提升的特权模式运行。我们将在这些问题出现时加以处理。

还应该注意的是,许多人更喜欢 Vagrant 来进行这些类型的测试。

如果可能的话,我更喜欢使用 Docker,因为它轻量级、快速并且在大多数情况下足够用。



请记住,实际在生产中部署系统需要比我们在此提供的更多关注安全性和其他细节。

傀儡师和傀儡特工

Puppet 是一种部署解决方案,在大型组织中非常流行,并且是同类系统中的首批系统之一。

Puppet 由客户端/服务器解决方案组成,其中客户端节点定期与 Puppet 服务器进行检查,以查看是否需要在本地配置中更新任何内容。

Puppet 服务器称为Puppet master,在为各种 Puppet 组件选择的名称中有很多相似的文字游戏。

Puppet 在处理服务器场的复杂性方面提供了很大的灵活性,因此,该工具本身非常复杂。

这是 Puppet 客户端和
木偶大师:

1. Puppet 客户端决定是时候检查 Puppet Master 以发现任何新的配置更改。这可能是由于计时器或客户端操作员的手动干预所致。 Puppet 客户端和 master 之间的对话通常使用 SSL 加密。
2. Puppet 客户端提供其凭据,以便 Puppet Master 可以准确知道哪个客户端正在调用。管理客户端凭证是一个单独的问题。
3. Puppet master 确定客户端应该具有的配置
通过编译 Puppet 目录并将其发送给客户端。这涉及许多机制,并且特定设置不需要利用所有可能性。

Puppet 客户端同时具有基于角色和具体的配置是很常见的。基于角色的配置可以被继承。

4. Puppet master 在客户端运行必要的代码,使配置与 Puppet master 决定的配置相匹配。

从这个意义上说,Puppet 配置是声明性的。你声明一台机器应该有什么样的配置,Puppet 就会计算出如何从当前状态进入所需的客户端状态。

部署代码

Puppet 生态系统有利也有弊：

- Puppet 有一个很大的社区，并且有很多资源在 Puppet 的互联网。有很多不同的模块，如果您没有真正奇怪的组件要部署，那么很可能已经有一个为您的组件编写的现有模块，您可以根据需要进行修改。
- Puppet 需要 Puppet 客户端机器上的一些依赖项。

有时，这会引起问题。 Puppet 代理将需要一个 Ruby 运行时，它有时需要领先于您的发行版存储库中可用的 Ruby 版本。企业发行版往往版本落后。

- Puppet 配置的编写和测试可能很复杂。

Ansible Ansible 是

一种倾向于简单性的部署解决方案。

Ansible 架构是无代理的；它不需要像 Puppet 那样在客户端运行守护进程。相反，Ansible 服务器登录到 Ansible 节点并通过 SSH 发出命令以安装所需的配置。

虽然 Ansible 的无代理架构确实使事情变得更容易，但您需要在 Ansible 节点上安装 Python 解释器。 Ansible 对其代码运行所需的 Python 版本比 Puppet 对其 Ruby 代码运行所需的 Python 版本更为宽松，因此这种对 Python 可用的依赖在实践中并不是一个很大的麻烦。

与 Puppet 和其他软件一样，Ansible 专注于幂等的配置描述符。这基本上意味着描述符是声明性的，并且 Ansible 系统会计算出如何将服务器带到所需的状态。您可以重新运行配置运行，这将是安全的，对于命令式系统而言不一定如此。

让我们用我们之前讨论过的 Docker 方法来尝试 Ansible。

我们将使用为此目的开发的 williamyeh/ansible 图像，但应该可以使用任何 Ansible Docker 图像或完全不同的图像，我们稍后只需将 Ansible 添加到其中。

1. 使用以下语句创建一个 Dockerfile：

来自 williamyeh/ansible:centos7

2. 使用以下命令构建 Docker 容器：

泊坞窗构建。

这将下载图像并创建一个空的 Docker 容器,我们可以使用。

通常,当然,您会有一个更复杂的 Dockerfile,可以添加我们需要的东西,但在这种情况下,我们将以交互方式使用图像,因此我们将使用主机中的 Ansible 文件挂载目录,以便我们可以在主机上更改它们并轻松地重新运行它们。

3. 运行容器。

以下命令可用于运行容器。您将需要来自先前构建命令的哈希值: docker run -v `pwd`/ansible:/ansible -it <hash> bash

现在我们有一个提示,并且我们有可用的 Ansible。-v技巧是使主机文件系统的一部分对 Docker 来宾容器可见。

这些文件将在容器的/ansible目录中可见。

playbook.yml文件如下:

```
---
- 主机:本地主机
  变量:
    http_port: 80
    max_clients: 200
  远程用户:root
  任务:
    - name: 确保 apache 是最新版本 yum: name=httpd state=latest
```

这个 playbook 做的不多,但它演示了 Ansible playbook 的一些概念。

现在,我们可以尝试运行我们的 Ansible 剧本:

```
cd /ansible
ansible-playbook -i inventory playbook.yml           --connection=local --sudo
```

输出将如下所示:

```
播放 [本地主机] ****
*****
```

```
收集事实 ****
*****
```

```
好的 [本地主机]
```

部署代码

```
任务:[确保 apache 是最新版本] ****
*****
好的:[本地主机]

播放回顾 ****
*****
本地主机 :好的= 2 改变=0 无法到达=0
失败=0
```

运行任务以确保我们想要的状态。在本例中,我们希望使用yum安装Apache 的httpd ,并且我们希望httpd是最新版本。

为了继续我们的探索,我们可能想做更多的事情,比如自动启动服务。然而,在这里我们遇到了使用 Docker 模拟物理或虚拟主机的方法的局限性。毕竟,Docker 是一种容器技术,而不是成熟的虚拟化系统。在 Docker 的正常用例场景中,这无关紧要,但在我们的例子中,我们需要做一些变通办法才能继续。主要问题是systemd init 系统需要特别小心才能在容器中运行。 Red Hat 的开发人员已经找到了执行此操作的方法。以下是与 Red Hat 合作的 Vaclav Pavlin 对 Docker 镜像的略微修改版本:

从软呢帽

```
运行 yum -y 更新; yum clean all RUN yum install ansible
sudo RUN systemctl mask systemd-remount-fs.service dev-
hugepages.mount sys fs-fuse-connections.mount systemd-logind.service getty.target console getty.service RUN cp /usr/lib/
systemd/system/dbus.service /etc/systemd/system/; sed -i s/OOMScoreAdjust=-900// /etc/systemd/system/dbus.service
```

```
音量[ /sys/fs/cgroup , /run , /tmp ]
ENV 容器=docker
```

```
CMD[ /usr/sbin/init ]
```

环境变量container用于告诉systemd init 系统它在容器内运行并相应地运行。

为了使systemd能够在容器中工作,我们需要docker run的更多参数:

```
docker run -it --rm -v /sys/fs/cgroup:/sys/fs/cgroup:ro -v `pwd`/ansible:<hash>
```

容器使用systemd 启动,现在我们需要从不同的 shell 连接到正在运行的容器:

```
docker exec -it <哈希> bash
```

呸!仅仅为了让容器更逼真就需要做大量工作!另一方面,在我看来,使用虚拟机(例如VirtualBox)更加麻烦。当然,读者可能会有不同的决定。

现在,我们可以在容器内运行稍微高级一点的 Ansible playbook,如下所示:

```
---
- 主机:本地主机
  变量:
    http_port: 80
    max_clients: 200
  remote_user:根任务:
    - name: 确保 apache 是最新版本
      yum: name=httpd state=latest
    - name: 编写 apache 配置文件
      template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    - name: 重新启动 apache
      service: name=httpd state=restarted
  处理程序:
    - name: 重新启动 apache 服务
      service: name=httpd
        state=restarted
```

此示例建立在前一个示例的基础上,并向您展示如何:

- 安装包 · 编写模板
- 文件 · 处理服务的运行状态

该格式采用非常简单的 YAML 语法。

托盘操作

PalletOps 是一个高级部署系统,它结合了 Lisp 的声明功能和非常轻量级的服务器配置。

部署代码

PalletOps 将 Ansible 的无代理理念更进一步。不需要在要配置的节点上安装 Ruby 或 Python 解释器,您只需要安装ssh和bash。这些都是非常简单的要求。

PalletOps 将其 Lisp 定义的 DSL 编译为在从节点上执行的 Bash 代码。这些要求非常简单,您可以在非常小和简单的服务器上使用它 甚至是手机!

另一方面,虽然 Pallet 有许多称为crate 的支持模块,但数量少于 Puppet 或 Ansible。

使用 Chef 进行部署

Chef 是来自 Opscode 的基于 Ruby 的部署系统。

试用 Chef 非常容易;为了好玩,我们可以在 Docker 容器中完成它,这样我们的实验就不会污染我们的主机环境:

```
docker 运行 -it ubuntu
```

我们需要curl命令来继续下载 Chef 安装程序:

```
apt-get 安装卷曲  
curl -L https://www.opscode.com/chef/install.sh |狂欢
```

Chef 安装程序是使用 Chef 团队的一个名为omnibus 的工具构建的。我们的目标是试用名为Chef-solo 的 Chef 工具。验证该工具是否已安装:

```
厨师独奏-v
```

这将给出如下输出:

```
厨师:12.5.1
```

chef-solo的要点是能够在没有配置系统的完整基础设施 (例如客户端/服务器设置)的情况下运行配置脚本。这种类型的测试环境在使用配置系统时通常很有用,因为在开发要部署的配置时很难让所有的细节都按顺序工作。

Chef 更喜欢其文件的文件结构,并且可以从 GitHub 检索预滚动结构。您可以使用以下命令下载并解压缩它:

```
curl -L http://github.com/opscode/chef-repo/tarball/master -o master.tgz tar -zxf master.tgz mv chef-chef-repo* chef-repo rm master.tgz
```

您现在将为 Chef 说明书准备一个合适的文件结构,如下所示:

```
./cookbooks ./
cookbooks/README.md ./data_bags ./
data_bags/README.md ./
environments ./environments/
README.md ./README.md ./LICENSE ./
roles ./roles/README.md ./忽略
```

您将需要执行进一步的步骤以使一切正常工作,告诉厨师在哪里可以找到其食谱:

```
主目录.chef
echo cookbook_path [ /root/chef-repo/cookbooks ] > .chef/knife.rb
```

现在我们可以使用小刀工具创建一个配置模板,如下:

```
刀食谱创建 phpapp
```

使用 SaltStack 部署

SaltStack 是一个基于 Python 的部署解决方案。

Jackson Cage 为 Salt 提供了一个方便的 dockerized 测试环境。您可以使用以下内容启动它:

```
docker run -i -t --name=saltdocker_master_1 -h master -p 4505 -p 4506 \
-p 8080 -p 8081 -e SALT_NAME=master -e SALT_USE=master \
-v `pwd`/srv/salt:/srv/salt:rw jacksoncage/salt
```

这将创建一个包含 Salt master 和 Salt minion 的容器。

我们可以在容器内部创建一个 shell 以供我们进一步探索:

```
docker exec -i -t saltdocker_master_1 bash
```

我们需要一个配置来应用到我们的服务器。 Salt 将配置称为“状态”或 Salt 状态。

部署代码

在我们的例子中,我们想要安装一个具有这个简单 Salt 状态的 Apache 服务器:

顶部.sls:

根据:

```
* :
  - 网络服务器
```

网络服务器.sls:

```
阿帕奇2:          # 身份声明
  包装:          # 状态声明
    - 安装        # 函数声明
```

Salt 使用.yml文件作为其配置文件,类似于 Ansible 所做的。

文件top.sls声明所有匹配的节点都应该是webserver 类型。

Web 服务器状态声明应该安装apache2包,基本上就是这样。请注意,这将取决于分布。我们使用的 Salt Docker 测试镜像基于 Ubuntu,其中 Apache Web 服务器包称为apache2。例如,在 Fedora 上,Apache Web 服务器包被简称为httpd。

通过让 Salt 读取 Salt 状态并在本地应用它,运行命令一次以查看 Salt 的运行情况:

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

第一次运行会非常冗长,特别是因为我们启用了调试标志!

现在,让我们再次运行命令:

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

这也将会非常冗长,输出将以此结尾:

当地的:

```
编号:apache2
功能:pkg.installed
结果:正确
注释:包 apache2 已经安装。
开始时间:22:55:36.937634
持续时间:2267.167 毫秒
```

变化：

概括

成功:1

失败的： 0

总状态运行：

1个

现在，您可以退出容器并重新启动它。这将从上一次运行期间安装的 Apache 实例中清除容器。

这一次，我们将应用相同的状态，但使用消息队列方法而不是在本地应用状态：

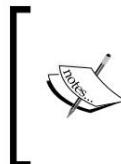
盐调用状态.highstate

这与之前使用的命令相同，只是我们省略了-local 标志。
您也可以尝试再次运行该命令并验证状态是否保持不变。

Salt 与 Ansible 与 Puppet 与 PalletOps 执行模型

虽然我们在本章中探索的配置系统有很多相似之处，但它们在客户端节点上执行代码的方式上有很大不同：

- 使用 Puppet，Puppet 代理向 Puppet 主机注册并打开通信通道以检索命令。这个过程定期重复，通常每三十分钟一次。



三十分钟并不快。当然，您可以为下一次运行所需的时间间隔配置一个较低的值。无论如何，Puppet 本质上使用的是拉模型。

客户必须检查以了解是否有可用的更改。

- Ansible 在需要时通过 SSH 推送更改。这是一个推送模型，但具有不同的实现。它使用一个 ZeroMQ 消息服务器，客户端连接到该服务器并监听有关更改的通知。这有点像 Puppet，但速度更快。
- Salt 使用推送模型，

[部署代码](#)

哪种方法最好是开发人员社区之间争论不休的领域。消息队列架构的支持者认为它更快,速度很重要。普通 SSH 方法的支持者声称它足够快并且简单性很重要。我倾向于后一种立场。事情往往会破裂,并且破裂的可能性随着复杂性而增加。

流浪汉

Vagrant 是虚拟机的配置系统。它旨在为开发人员创建虚拟机,但也可用于其他目的。

Vagrant 支持多个虚拟化提供商,而 VirtualBox 是一个很受开发人员欢迎的提供商。

首先,一些准备工作。根据您的发行版的说明安装vagrant。对于 Fedora,命令是这样的:

百胜安装“流浪者”

这将安装许多软件包。但是,当我们在 Fedora 上安装它时,我们会遇到一些问题。 Fedora Vagrant 软件包使用 libvirt 作为虚拟机提供者而不是 VirtualBox。这在许多情况下很有用,但在这种情况下,我们想使用 VirtualBox 作为提供者,这需要在 Fedora 上执行一些额外的步骤。如果您使用其他发行版,情况可能会有所不同。

首先,将 VirtualBox 存储库添加到您的 Fedora 安装中。然后我们就可以用 dnf 命令安装VirtualBox了,如下:

dnf 安装 VirtualBox

不过,VirtualBox 还没有准备好使用。它需要特殊的内核模块才能工作,因为它需要仲裁对低级资源的访问。VirtualBox 内核驱动程序不随 Linux 内核一起分发。与使用始终默认安装的内核驱动程序的便利性相比,在 Linux 源代码树之外管理 Linux 内核驱动程序总是有些不便。

VirtualBox 内核驱动程序可以作为需要安装的源模块安装

编译。这个过程可以通过 dkms 命令在一定程度上自动化,它会在安装新内核时根据需要重新编译驱动程序。另一种方法更简单且不易出错,是使用由您的发行版为您的内核编译的内核模块。如果您的发行版提供了内核模块,它应该会自动加载。否则,您可以尝试 modprobe vboxdrv。对于某些发行版,您可以通过调用 init.d 脚本来编译驱动程序,如下所示:

```
sudo /etc/init.d/vboxdrv setup
```

现在已经安装了 Vagrant 依赖项,我们可以启动一个 Vagrant 虚拟机。

以下命令将从模板创建 Vagrant 配置文件。我们稍后将能够更改此文件。基础镜像将是hashicorp/precise32,它又基于 Ubuntu。

```
vagrant init 哈希公司/precise32
```

现在,我们可以启动机器了:

```
流浪起来
```

如果一切顺利,我们现在应该有一个正在运行的vagrant虚拟机实例,但由于它是无头的,我们将看不到任何东西。

Vagrant 与 Docker 有一些相似之处。 Docker 使用可以扩展的基础镜像。 Vagrant 也允许这样做。在 Vagrant 词汇表中,基本图像称为框。

要连接到我们之前启动的 headless vagrant实例,我们可以使用以下命令:

```
流浪者 ssh
```

现在我们有一个ssh会话,我们可以在其中使用虚拟机。为此,Vagrant 已经完成了一些任务,例如为我们设置 SSH 通信通道的密钥。

Vagrant 还提供了一个配置系统,以便可以使用 Vagrant 机器描述符来重新创建一个完全从源代码配置的虚拟机。

这里是我们前期拿到的 Vagrantfile。为简洁起见,删除了评论。

```
Vagrant.configure(2) 执行 |config|
  config.vm.box = hashicorp/precise32
结尾
```

在 Vagrantfile 中添加一行,调用我们将提供的 bash 脚本:

```
Vagrant.configure( 2 ) 执行 |config|
  config.vm.box = hashicorp/precise32
  config.vm.provision :shell,路径: "bootstrap.sh"
结尾
```

部署代码

bootstrap.sh脚本将如下所示：

```
#!/usr/bin/env 庆典
```

易于获取更新

```
apt-get install -y apache2
```

这将在 Vagrant 管理的虚拟机中安装 Apache Web 服务器。

现在我们对 Vagrant 有了足够的了解,可以从 DevOps 视角：

- Vagrant 是一种主要为基于VirtualBox 的虚拟机管理配置的便捷方式。它非常适合测试。
- 配置方法并没有真正扩展到集群,也不是预期用例。
- 另一方面,一些配置系统如 Ansible 支持 Vagrant,所以 Vagrant 在测试我们的配置代码时非常有用。

使用 Docker 部署

最近的部署替代方案是 Docker,它具有几个非常有趣的特性。我们已经在本书中多次使用了 Docker。

即使您使用 Puppet 或 Ansible 等来部署您的产品,您也可以将 Docker 的功能用于测试自动化目的。

Docker 创建可在开发机器、测试环境和生产环境中使用的可重用容器的模型非常有吸引力。

在撰写本文时,Docker 开始对大型企业产生影响,但 Puppet 等解决方案占主导地位。

虽然众所周知如何构建大型 Puppet 或 Ansible 服务器群,但还不太了解如何构建大型基于 Docker 的服务器集群。

有几种新兴的解决方案,例如：

- Docker Swarm: Docker Swarm 与Docker Compose 兼容,这很吸引人。 Docker Swarm 由 Docker 社区维护。
- Kubernetes: Kubernetes 仿照 Google 的 Borg 集群软件,

这很有吸引力,因为它是谷歌庞大数据中心内部使用的经过良好测试的模型。 Kubernetes 与 Borg 不同,这一点必须牢记。目前尚不清楚 Kubernetes 是否提供与 Borg 相同的扩展方式。

比较表

每个人都喜欢为旧概念想出新词。虽然各种产品中的不同概念并不总是匹配,但制作一本字典来映射配置系统的不同术语是很有诱惑力的。

下面是这样一张术语对比图：

系统	木偶	Ansible的	托盘	盐
客户	代理人	节点	节点	奴才
服务器	掌握	服务器	服务器	掌握
配置	目录	剧本	箱	盐州

另外,这里有一张技术对比图：

系统	木偶	Ansible的	托盘	厨师	盐
无代理	不	是的	是的	是的	两个都
客户端依赖	红宝石	Python、sshd、bash	sshd, 庆典	红宝石,sshd,庆典	Python
语	红宝石	Python	Clojure 语言	红宝石	Python

云解决方案首先,我们必须

退一步看风景。我们可以使用云提供商,例如 AWS 或 Azure,也可以使用我们自己的内部云解决方案,例如 VMware 或 OpenStack。外部和内部云提供商或什至两者都有有效的论据,具体取决于您的组织。

某些类型的组织,例如政府机构,必须将有关公民的所有数据存储在他们自己的墙内。此类组织不能使用外部云提供商和服务,而必须构建自己的内部云等效项。

较小的私有组织可能会从使用外部云提供商中受益,但可能负担不起将所有资源都交给这样的提供商。他们可能会选择使用内部服务器来应对正常负载,并在峰值负载期间扩展到外部云提供商。

我们在这里描述的许多配置系统都支持云节点和本地节点的管理。例如,PalletOps 支持 AWS, Puppet 支持 Azure。 Ansible 支持许多不同的云服务。

部署代码**AWS**

Amazon Web Services 允许我们在亚马逊的集群上部署虚拟机镜像。您还可以部署 Docker 映像。按照以下步骤设置 AWS：

1. 注册一个 AWS 账户。注册是免费的,但即使是免费级别也需要信用卡号
2. 需要进行一些身份验证,这可以通过
自动挑战响应电话。
3. 用户验证过程完成后,您将能够登录 AWS 并使用 Web 控制台。



在我看来,AWS Web 控制台并不代表 Web 界面可用性的缩影,
但它完成了工作。
有很多选项,在我们的例子中,我们对虚拟机和 Docker 容器选项感
兴趣。

4. 转到EC2 网络和安全。您可以在此处创建稍后需要的管理密钥。

作为第一个示例,让我们创建 AWS 提供的默认容器示例,console-sample-app-static。要登录生成的服务器,您首先需要创建一个 SSH 密钥对并将您的公钥上传到 AWS。

单击所有步骤,您将获得一个小样本集群。最后的资源创建步骤可能会很慢,所
以这是喝杯咖啡的绝佳机会!

5. 现在,我们可以查看集群的详细信息并选择Web服务器容器。可以看到IP
地址。尝试在网络浏览器中打开它。

现在我们在 AWS 上有了一个工作账户,我们可以使用我们选择的配置管理系统
来管理它。

Azure

Azure 是微软的云平台。它可以托管 Linux 和 Microsoft 虚拟机。
虽然 AWS 是人们经常默认使用的服务,至少在 Linux 领域是这样,
但探索这些选项永远不会有坏处。 Azure 就是这样一种选择,目前
正在获得市场份额。

在 Azure 上创建虚拟机用于评估目的与在 AWS 上创建虚拟机相当。过程还算顺
利。

概括

在本章中,我们探讨了在部署我们构建的代码时可用的众多选项中的一些。有很多选择,这是有原因的。
部署是一个困难的主题,您可能会花费大量时间来找出最适合您的选项。

在下一章中,我们将探讨监控运行代码的主题。

Machine Translated by Google

8

监控代码

在上一章中,我们探讨了部署代码的方法。

既然已经使用您选择的部署解决方案将代码安全地部署到您的服务器,您需要监视它以确保它正常运行。

您可以花费大量时间为在开发过程中可能设想的多种故障模式做准备。最后,您的软件可能会因为您准备好的其他原因而崩溃。如果您的系统出现故障,您的组织可能会付出非常高昂的代价,无论是收入损失还是信誉损失,这最终可能是一回事。您需要尽快知道出了什么问题,以便处理这种情况。

考虑到服务停机的潜在负面影响,有许多替代解决方案可以解决从不同角度和观点监视已部署代码的问题域。

在本章中,我们将了解几个可供我们使用的选项。

Nagios

在本节中,我们将探讨用于整体服务器健康状况的 Nagios 监控解决方案。

Nagios 自 1999 年问世以来,拥有一个庞大的社区,可提供满足各种需求的插件和扩展。网络上有大量适用于 Nagios 的资源,如果您的组织需要,还可以提供商业支持。

由于它已经存在了很长时间并且许多组织都在使用它,Nagios 是网络监控的标准,其他解决方案可以与之相提并论。因此,它是我们穿越监控景观之旅的自然起点。

监控代码

Nagios这个名字是一个递归的首字母缩写词,这是黑客圈子里的一个传统。它代表“Nagios Ain't Gonna Insist On Sainthood”。最初,该产品名为NetSaint,但在商标纠纷期间该名称被否决,取而代之的是Nagios首字母缩略词。agios这个词在希腊语中也意味着天使,所以它是一个非常聪明的首字母缩略词。

下面是一些Nagios安装演示的截图。Nagios提供了很多视图,这里我们看到其中两个:

·所有主机组的服务概览:

Linux Servers (linux-servers)				Network Printers (network-printers)				Network Switches (switches)			
Host	Status	Services	Actions	Host	Status	Services	Actions	Host	Status	Services	Actions
localhost	UP	11 OK 5 CRITICAL		hpcolor	UP	1 OK 1 WARNING		netgear-prosafe	UP	1 OK 3 UNKNOWN	
exodia	UP	No matching services		gate	UP	No matching services					
gw	UP	4 OK 2 CRITICAL		localhost	UP	11 OK 5 CRITICAL					
mentor-xbmc	DOWN	6 CRITICAL									

所有主机的服务状态详细信息：

The screenshot shows the Nagios Core interface in a Firefox browser. The left sidebar contains navigation links for General, Current Status, Services, Host Groups, Service Groups, Problems, Reports, and System. The main content area displays the "Current Network Status" with a last update of Wed Nov 11 00:10:44 CET 2015. It includes sections for "Host Status Totals" and "Service Status Totals". Below this is the "Service Status Details For All Hosts" table, which lists 100 services across various hosts (gw, hpcolor, localhost). The table columns include Host, Service, Status, Last Check, Duration, Attempt, and Status Information. Many entries show critical errors, such as "/dev/hda1 Free Space" and "updates". The table has a yellow border and some rows are highlighted in yellow.

Host	Service	Status	Last Check	Duration	Attempt	Status Information
gw	/dev/hda1 Free Space	CRITICAL	11-11-2015 00:04:18	1271d 7h 6m 49s	3/3	DISK CRITICAL - /dev/hda1 is not accessible. No such file or directory
gw	CPU Load	OK	11-11-2015 00:08:18	135d 12h 53m 19s	1/3	OK - load average: 0.00, 0.01, 0.05
gw	Current Users	OK	11-11-2015 00:09:18	135d 12h 54m 19s	1/3	USERS OK - 3 users currently logged in
gw	Total Processes	OK	11-11-2015 00:03:18	135d 12h 54m 19s	1/3	PROCS OK: 110 processes
gw	Zombie Processes	OK	11-11-2015 00:02:18	135d 12h 51m 19s	1/3	PROCS OK: 0 processes with STATE = Z
gw	updates	CRITICAL	11-11-2015 00:04:18	1271d 7h 2m 53s	3/3	NRPE: Command 'check_updates' not defined
hpcolor	PING	OK	11-11-2015 00:09:18	1d 3h 1m 26s	1/3	PING OK - Packet loss = 0%, RTA = 0.30 ms
hpcolor	Printer Status	WARNING	11-11-2015 00:08:18	3d 13h 12m 26s	3/3	Out of Paper ("Fill p.. paper.")
localhost	Current Load	OK	11-11-2015 00:09:18	5d 1h 21m 26s	1/4	OK - load average: 0.70, 0.72, 0.67
localhost	Current Users	OK	11-11-2015 00:07:18	1441d 9h 7m 32s	1/4	USERS OK - 4 users currently logged in
localhost	HTTP	OK	11-11-2015 00:07:18	226d 0h 55m 14s	1/4	HTTP OK - HTTP/1.1 200 OK - 3933 bytes in 0.000 second response time
localhost	Midonkey	CRITICAL	11-11-2015 00:08:18	399d 2h 8m 56s	4/4	PROCS CRITICAL: 0 processes with command name 'midonkey'
localhost	PING	OK	11-11-2015 00:09:18	739d 19h 29m 22s	1/4	PING OK - Packet loss = 0%, RTA = 0.05 ms
localhost	Root Partition	CRITICAL	11-11-2015 00:08:18	56d 13h 3m 11s	4/4	DISK CRITICAL - free space: / 6200 MB (4% inode=95%)
localhost	SSH	OK	11-11-2015 00:08:18	739d 19h 27m 49s	1/4	SSH OK - OpenSSH_6.9 (protocol 2.0)
localhost	Swap Usage	OK	11-11-2015 00:09:18	1441d 9h 4m 25s	1/4	SWAP OK - 97% free (7574 MB out of 7887 MB)
localhost	Total Processes	CRITICAL	11-11-2015 00:08:18	30d 0h 43m 18s	4/4	PROCS CRITICAL: 726 processes with STATE = RSZDT
bitbee	OK	OK	11-11-2015 00:09:18	523d 17h 5m 22s	1/4	• bitbee.service - BitBee IRC/IM gateway
dovecot	OK	OK	11-11-2015 00:07:18	226d 5h 30m 23s	1/4	• dovecot.service - Dovecot IMAP/POP3 email server
fetchmail	OK	OK	11-11-2015 00:08:18	305d 1h 40m 7s	1/4	• fetchmail.service - SYSV: Fetchmail mail retrieval program
minecraft	CRITICAL	CRITICAL	11-11-2015 00:08:18	465d 7h 50m 31s	4/4	connect to address localhost and port 25565: Connection refused
test	OK	OK	11-11-2015 00:08:18	1440d 21h 3m 43s	1/4	127.0.0.1 10 20
						CHECK_UPDATES CRITICAL - your machine is running kernel 4.1.10-200

这些视图指出了一些示例问题，如下所示：

- 主机不可访问
- 其中一台主机中的磁盘已满

监控代码

主机有可用的挂起更新。现在让我们通过一个简单的场景来特别检查 Nagios 的基础知识和一般监控：

- 我们有一个 Nagios 主机来监控其他一些设备的运行状况
物理和虚拟主机。
- 被监控的主机之一,一个网络服务器,突然从
整个网络。Nagios 知道这一点,因为它被设置为每五分钟 ping Web 服务器一次,并在服务器没有及
时响应 ping 的情况下采取行动。
- Nagios 确定当此服务器关闭时要做什么,在本例中是向预定组中的每个人(在本例中为组织的管理员)
发送电子邮件。 · 恰好在 Matangle 办公室附近的 Joakim 是第一个回复电子邮件的人,然后去了
地窖里相当不专业的服务器室。网络服务器已经停止工作,因为它的电源单元充满了灰尘,这是在附近
的房间进行装修时堆积在那里的 锯石膏会产生大量灰尘。 Joakim 隽咒并发誓说,如果他只有
资源。

在这种情况下,您可能比 Joakim 更好地照顾您的服务器,但关键是会发生不可预见的事情,而 Nagios 有助于
解决这种情况。此外,虽然为您的服务器使用合适的数据中心可能会防止诸如设备灰尘和人员被网络电缆绊倒等
问题,但在严酷的复杂服务器部署世界中仍然存在许多奇怪和无法预料的故障模式。

Nagios 可以监控服务和主机,并且可以使用不同的方法来实现。最简单的方法在 Nagios 服务器上运行并查
询远程主机上的服务。

Nagios 区分两种基本的检查类别:

- 主动检查**:这些检查由 Nagios 发起。Nagios 可以定时执行插件中的代码。代码执行的结果决定了检
查是否失败。主动检查是最常见的,非常有用且易于设置许多类型的服务,例如 HTTP、SSH 和数据库。
- 被动检查**:这些源自 Nagios 以外的系统,并且系统通知 Nagios 而不是 Nagios 轮询它。这些在特殊情
况下很有用,例如,如果被监视的服务在防火墙后面,或者如果服务是异步的并且通过轮询来监视它效
率低下。

如果您习惯于基于文件的配置,那么配置 Nagios 相对简单,但是要获得基本功能,需要进行大量配置。

我们将使用 Docker Hub 映像cpuguy83/nagios 探索 Nagios。

该映像使用 Nagios 版本 3.5,该版本与撰写本文时 Fedora 存储库中可用的版本相同。有更高版本的 Nagios 可用,但许多人似乎仍然更喜欢版本 3 系列。 cpuguy83映像采用从 SourceForge 下载 Nagios tarball 并将其安装在映像中的方法。如果您愿意,可以使用相同的方法来安装您选择的 Nagios 版本。本书源代码中有一个替代的 Docker 文件,如果您使用它,则需要在本地构建镜像。

此语句启动 Nagios 服务器容器:

```
docker run -e -p      NAGIOSADMIN_USER=nagiosadmin -e NAGIOSADMIN_PASS=nagios  
80:30000 cpuguy83/nagios
```

验证 Nagios Web 界面在端口 30000 上是否可用;输入上面定义的用户名和密码以登录。

Nagios 用户界面的开箱即用配置允许您浏览主机和服务等内容,但一开始定义的唯一主机是本地主机,在本例中是运行 Nagios 本身的容器。我们现在将以单独的 Docker 容器的形式配置另一台主机进行监控。这将使我们能够强制关闭受监控的容器并验证 Nagios 容器是否注意到中断。

此命令将启动第二个以默认配置运行nginx的容器:

```
docker run -p 30001:80 nginx
```

通过转到端口 30001验证容器是否正在运行。您将看到一条简单的消息,欢迎使用 nginx! 。这或多或少就是这个容器开箱即用的全部功能,这对我们的测试来说已经足够好了。当然,如果您愿意,也可以使用物理主机。

要一起运行容器,我们可以在命令行上链接它们或使用 Docker 组合文件。这是此场景的 Docker compose 文件,也可在本书资源中找到:

```
nagios: 图  
片: mt-nagios 构建:  
- mt-nagios
```

监控代码

```
端口: -
80:30000
环境:
- NAGIOSADMIN_USER=nagiosadmin
- NAGIOSADMIN_PASS=nagios 卷:
./nagios:/etc/nagios nginx: 图片: nginx
```

Nagios 配置作为 Docker 卷安装在./nagios目录中。

Nagios 监控nginx容器所需的配置在源代码中可用,也包含在此处:

```
定义主机{
    姓名          常规主机 linux 服务器
    利用
    注册          0
    max_check_attempts 5
}

定义主机{
    利用          常规主机客户端 1
    host_name 地址
    192.168.200.15 contact_groups admins test client1

    笔记

} hostgroups.cfg define
hostgroup { hostgroup_name 测
    试组别名
    成员          测试服务器
    成员          客户1
}

services.cfg
#+BEGIN_SRC sh
# 定义服务(
    利用          通用服务测试组
    hostgroup_name
    service_description 平
    check_command   check_ping!200.0,20%!600.0,60%
)
#END_SRC
```

让我们看看当我们关闭nginx容器时会发生什么。

使用docker ps找出nginx容器的哈希值。然后用docker kill杀掉它。再次使用docker ps验证容器是否真的消失了。

现在,稍等片刻并重新加载 Nagios Web 用户界面。您应该看到 Nagios 已经提醒您这种情况。

这看起来类似于显示错误服务的早期屏幕截图。

现在,您想在 NGINX 宕机时收到一封电子邮件。然而,制作一个以万无一失的方式工作的示例并不简单,因为如今由于垃圾邮件,电子邮件变得更加复杂。您需要了解您的邮件服务器详细信息,例如 SMTP 服务器详细信息。这是您需要填写特定详细信息的框架:

您可以创建一个文件contacts.cfg来处理电子邮件,其内容如下:

```
定义联系人{ 联系人姓名
    利用 matangle-admin generic-
    别名 contact Nagios Admin pd-
    电子邮件 admin@matangle.com
}

定义联系人组{
    contactgroup_name 别名 管理员
    成员 Nagios 管理员 matange-admin
}
```

 如果您不更改此配置,邮件将最终发送到 pd-admin@matangle.com,这将被忽略。

Munin Munin

用于绘制服务器统计信息,例如内存使用情况,这对于了解整体服务器健康状况很有用。由于 Munin 随着时间的推移绘制统计数据,您可以看到资源分配趋势,这可以帮助您在问题变得严重之前发现问题。还有其他几个应用程序,如 Munin,可以创建图形;然而,与 Nagios 一样,Munin 是一个很好的起点。

它的设计易于使用和设置。开箱即用的体验让您只需很少的工作即可获得许多图表。

在北欧传说中,胡金和穆宁是两只漆黑的乌鸦。他们不知疲倦地在米德加尔特的世界各地飞来飞去,收集情报。千山万水后,他们回到神王奥丁身边,坐在他的肩膀上,将他们的遭遇一一诉说。Hugin这个名字来源于思想这个词,而Munin这个词来源于记忆这个词。

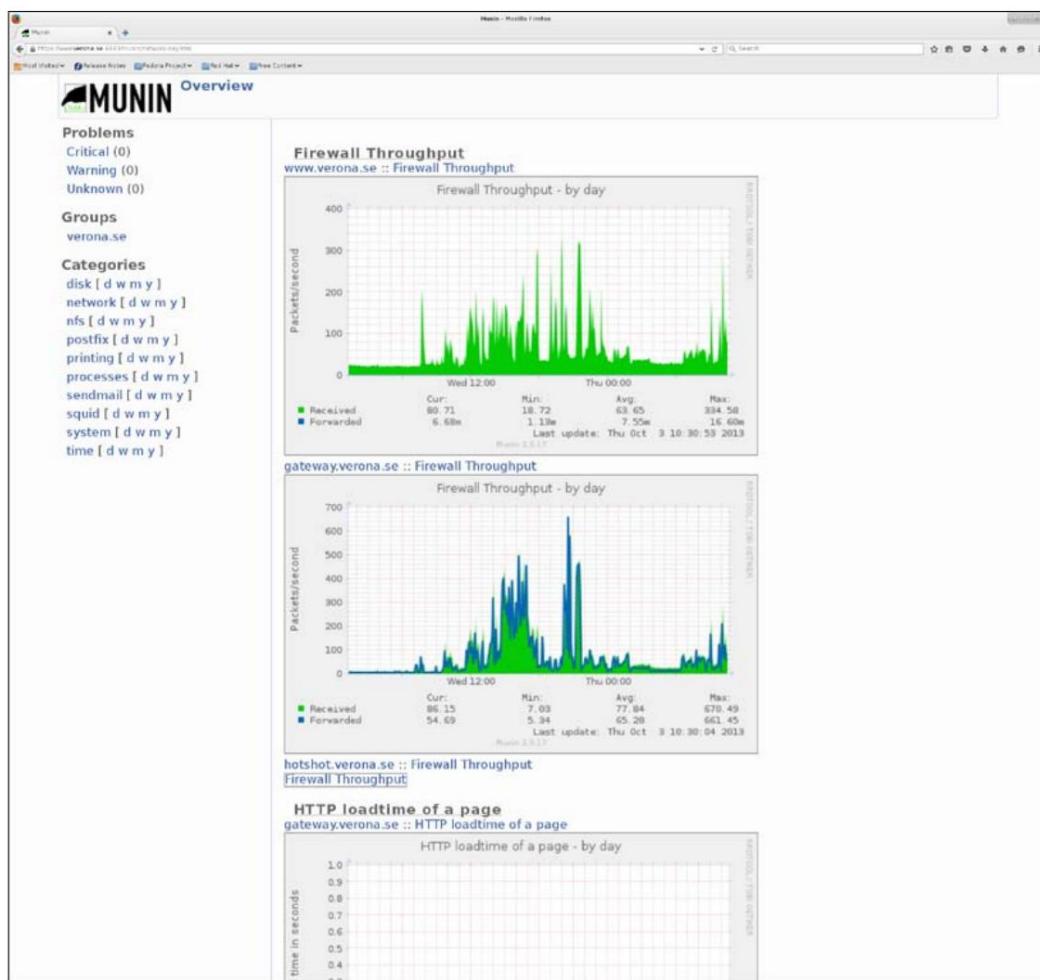
监控代码

Nagios 关注服务健康状况的高级特征（服务或主机是否处于二进制状态）,而 Munin 会跟踪它定期采样的统计数据,并绘制它们的图表。

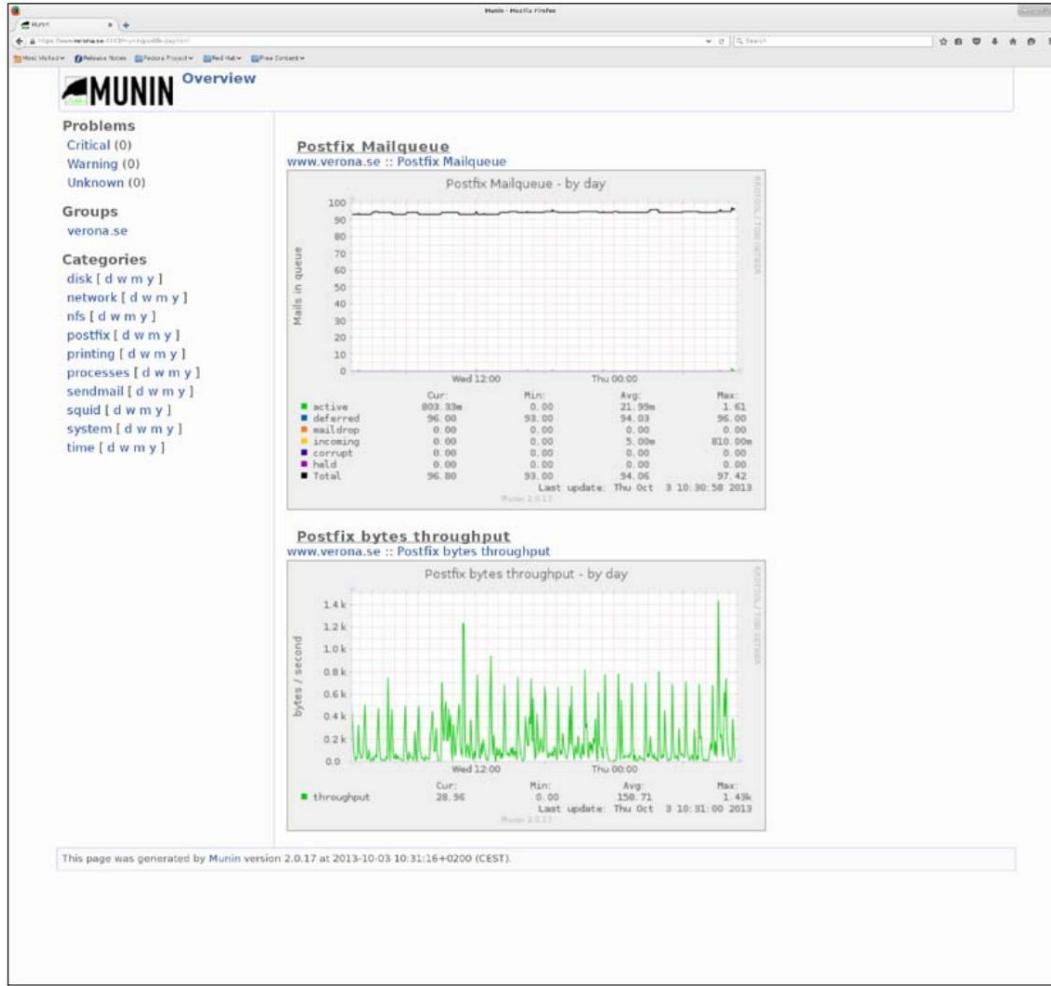
Munin 可以对许多不同类型的统计数据进行采样,从 CPU 和内存负载到您孩子的 Minecraft 服务器中的活跃用户数。也很容易制作插件,从您自己的服务中获取您想要的统计数据。

一幅图像可以说很多,一眼就能传达很多信息,因此绘制服务器的内存和处理器负载图表可以在出现问题时提前发出警告。

以下是 Munin 的示例屏幕截图,显示了 Matangle 总部防火墙安装的指标:



以下是该组织的smtpd (Postfix 安装)的一些指标：



与我们在本章中探索的任何监控系统一样,Munin 也具有面向网络的架构,如此处所述。它在设计上类似于 Nagios。

Munin的主要成分如下：

- 有一个中央服务器,Munin master,负责从 Munin 节点收集数据。 Munin 数据存储在名为RRD 的数据库系统中,RRD是Round-robin Database 的首字母缩写。 RRD 还对收集到的数据进行绘图。
- Munin 节点是安装在将被监控的服务器上的一个组件。 Munin master 连接到所有 Munin 节点并运行将数据返回给 master 的插件。

监控代码

为了尝试一下,我们将再次使用运行我们正在探索的服务 Munin 的 Docker 容器:

```
docker run -p 30005:80 lrivallain/munin:最新
```

Munin 第一次运行需要一段时间,所以在检查网页界面之前先等待一段时间。如果不希望等待,可以在容器中手动运行munin-update命令,如下所示。它显式地轮询所有 Munin 节点以获取统计信息。

```
docker run exec -it <哈希> bash  
su - munin --shell=/bin/bash  
/usr/share/munin/munin-更新
```

现在您应该能够看到在第一次更新期间创建的图表。如果让堆栈运行一段时间,您可以看到图形是如何发展的。

编写一个 Munin 插件来监视特定于您的应用程序堆栈的统计信息并不难。您可以制作 Munin 调用的 shell 脚本,以获取您想要跟踪的统计信息。

Munin 本身是用 Perl 编写的,但是只要符合非常简单的界面,您就可以用大多数语言编写 Munin 插件。

当使用配置参数调用时,程序应该返回一些元数据。
这样 Munin 就可以在图表上放置适当的标签。

这是一个示例图形配置:

```
graph_title 平均负载 graph_vlabel 负载  
load.label 负载
```

要发出数据,您只需将其打印到标准输出即可。

```
printf load.value cut -d     -f2 /  
proc/loadavg
```

这是一个示例脚本,它将绘制机器的平均负载图:

```
#!/bin/sh

配置中的 $1 案例)

猫<< EOM
graph_title 平均负载 graph_vlabel 负载
load.label 负载

EOM
出口 0;;
esac

printf load.value cut -d
-f2 /proc/loadavg
```

这个系统非常简单和可靠,您可以很容易地为您的应用程序实现它。您需要做的就是能够将您的统计信息打印到标准输出。

神经节

Ganglia 是大型集群的图形和监控解决方案。它可以在方便的概览显示中汇总信息。

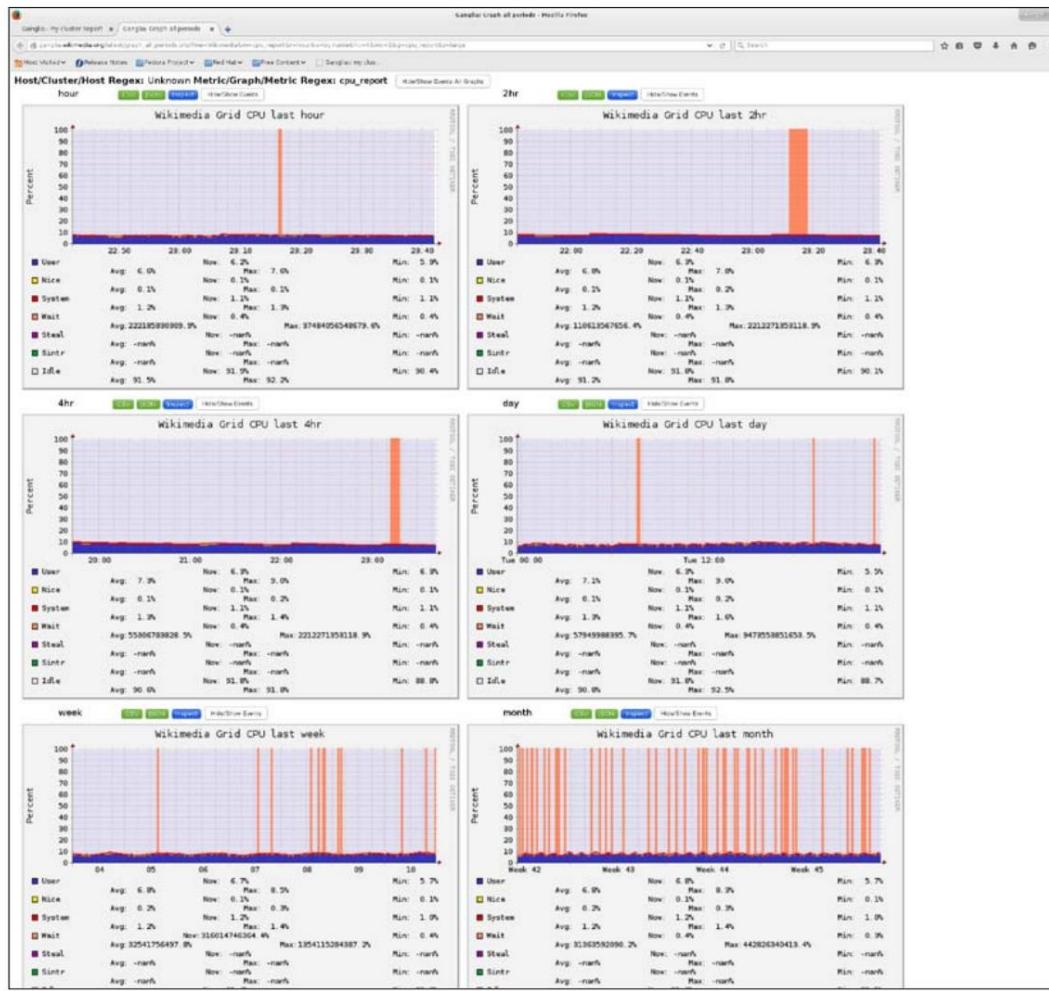
“神经节”一词是神经节的复数形式,是解剖学上的神经细胞簇。这个类比意味着 Ganglia 可以是您集群中的感觉神经细胞网络。

与 Munin 一样,Ganglia 也使用 RRD 进行数据库存储和绘图,因此这些图看起来与之前的 Munin 图类似。代码重用是一件好事!

Ganglia 在<http://ganglia.wikimedia.org/>上有一个有趣的在线演示。组织/最新/。

监控代码

Wikimedia 为维基百科提供媒体内容,它非常繁忙。因此,演示让我们很好地了解了 Ganglia 的功能,这在您自己的网络中很难以任何简单的方式获得。



第一页以图形格式显示可用数据的概览。您可以深入查看图表以获得其他视角。

例如,如果您针对 CPU 负载向下钻取其中一个应用程序集群图,您将获得集群中各个服务器的详细视图。您可以单击服务器节点以进一步向下钻取。

如果您拥有 Wikimedia 那种集群规模,您可以清楚地看到 Ganglia 概览和深入视图的吸引力。您既可以从轨道上查看,也可以轻松访问详细信息。

Ganglia 由以下组件组成:

- Gmond:这是Ganglia 监控守护程序的首字母缩写词。金蒙德是收集节点信息的服务。 Gmond 将需要安装在您希望 Ganglia 监控的每台服务器上。
- Gmetad:这代表Ganglia 元守护进程。 Gmetad 是一个运行在主节点上的守护进程,收集所有 Gmond 节点收集的信息。 Gmetad 守护进程也可以协同工作以在网络上分散负载。如果你有一个足够大的集群,拓扑确实开始看起来像一个神经细胞网络!
- RRD:循环数据库,与 Munin 在主节点上使用的工具相同,以适合图形的时间序列存储 Ganglia 的数据和可视化。
- 基于 PHP 的 Web 前端:显示主节点拥有的数据
收集和 RRD 为我们绘制了图表。

与 Munin 相比,Ganglia 多了一层,即 meta daemon。这个额外的层允许 Ganglia 通过在节点之间分配网络负载来扩展。

Ganglia 有一个网格概念,您可以将具有相似目的的集群组合在一起。例如,您可以将所有数据库服务器放在一个网格中。数据库服务器不需要有任何其他关系;他们可以服务于不同的应用程序。网格概念允许您一起查看所有数据库服务器的性能指标。

您可以以任何您喜欢的方式定义网格:按位置、按应用程序或您需要的任何其他逻辑分组来查看服务器会很方便。

您也可以再次使用 Docker 镜像在本地试用 Ganglia。

监控代码

来自 Docker Hub的wookietreiber/ganglia Docker 镜像提供了一些内置帮助：

docker 运行 wookietreiber/ganglia --help

用法:docker run wookietreiber/ganglia [opts]

运行一个 ganglia-web 容器。

-? | -h | -帮助 | --help --with-gmond 容器 打印此帮助并在里面运行
gmond

--without-gmond 容器 不要在里面运行gmond

--时区参数容器， 设置时区内

区域信息， 必须在 /usr/share/ 下面的路径
例如欧洲/柏林

在我们的例子中,我们将使用以下命令行参数运行图像：

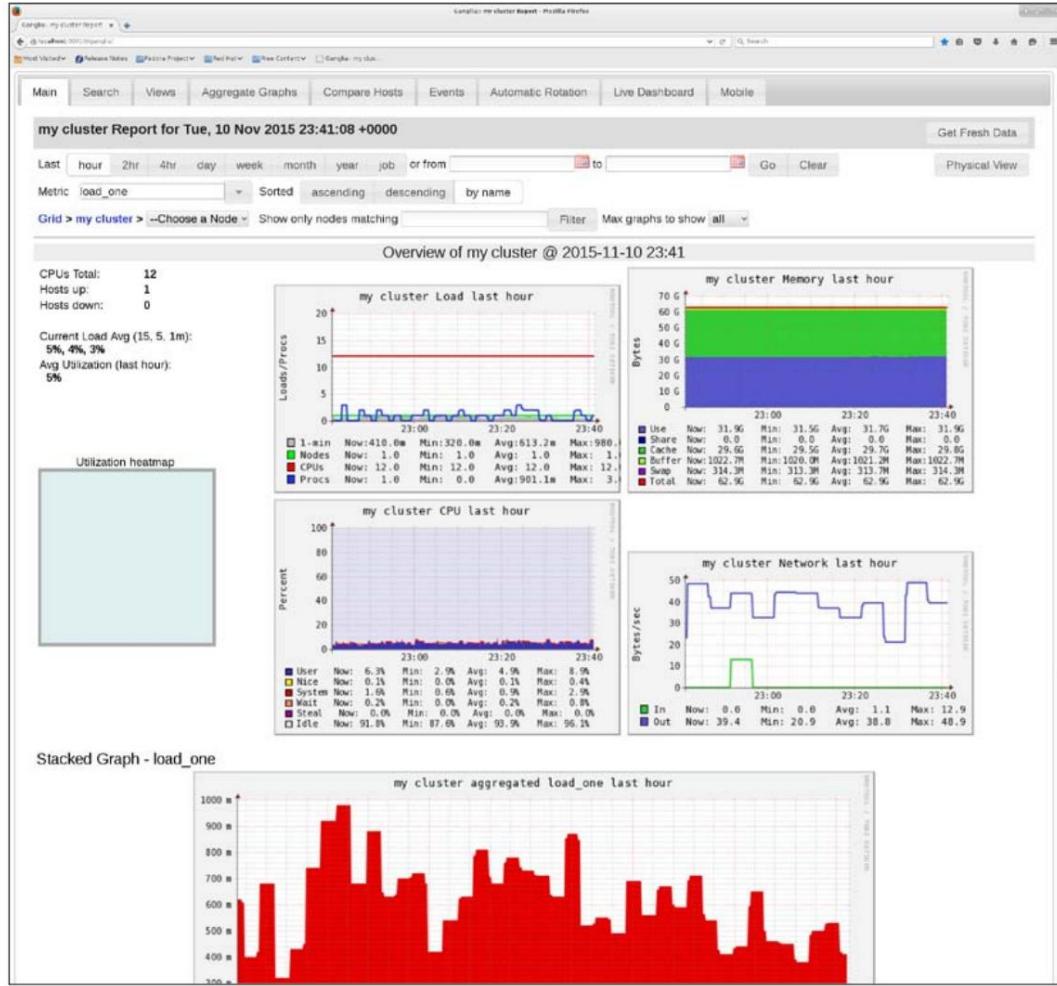
docker run -p 30010:80 wookietreiber/ganglia

此映像运行 Gmond 和 Gmetad。这意味着它将开箱即用地监控自己。稍后我们将添加一个单独的容器来运行 Gmetad。

现在 Ganglia 容器正在运行,您可以查看容器提供的 Web 界面：

<http://localhost:30010/ganglia/>

浏览器应显示类似于以下屏幕截图的视图：



Ganglia Gmond 节点在多播 IP 通道上相互通信。

这在大型集群中提供了冗余和易于配置，并且多播配置是默认的。您还可以在单播模式下配置 Ganglia。

监控代码

由于我们将只运行两个通信容器，在这种情况下我们实际上不会从多播配置中受益。

同样，我们将使用 Docker Compose 一起启动容器，以便单独的 Gmond 和 Gmetad 实例可以通信。我们可以启动三个都运行gmond 的容器。 Gmetad 守护进程将在默认配置中发现它们并将它们一起列在My Cluster 下：

gmetad:

图片:wookietreiber/ganglia 端口:

- “30010:80”

吉蒙德:

图片:wookietreiber/ganglia gmond2:图片:

wookietreiber/ganglia

当您访问<http://localhost:30010/ganglia/> 时，您应该会发现正在监视的新gmond实例。

石墨

虽然 Munin 很不错，因为它很健壮而且很容易上手，但它提供的图表只是偶尔更新一次，通常是每五分钟更新一次。因此，需要一种更接近实时的绘图工具。 Graphite 就是这样一种工具。

Graphite堆栈由以下三个主要部分组成。它类似于 Ganglia 和 Munin，但使用自己的组件实现。

- Graphite Web 组件，它是一个 Web 应用程序，呈现一个用户界面，该界面由在树状浏览器小部件中组织的图形和仪表板组成
- 收集指标的 Carbon 指标处理守护程序 · Whisper 时间序列数据库库

因此,Graphite 栈在实用性上与 Munin 和 Ganglia 相似。与 Munin 和 Ganglia 不同的是,它使用自己的时间序列库 Whisper,而不是 RRD。

有几个预打包的 Docker 映像可用于试用 Graphite。我们可以使用来自 Docker Hub 的 sitespeedio/graphite 镜像,如下所示:

```
docker run -it \ -p 30020:80 \ -p 2003:2003 \ sitespeedio/graphite
```

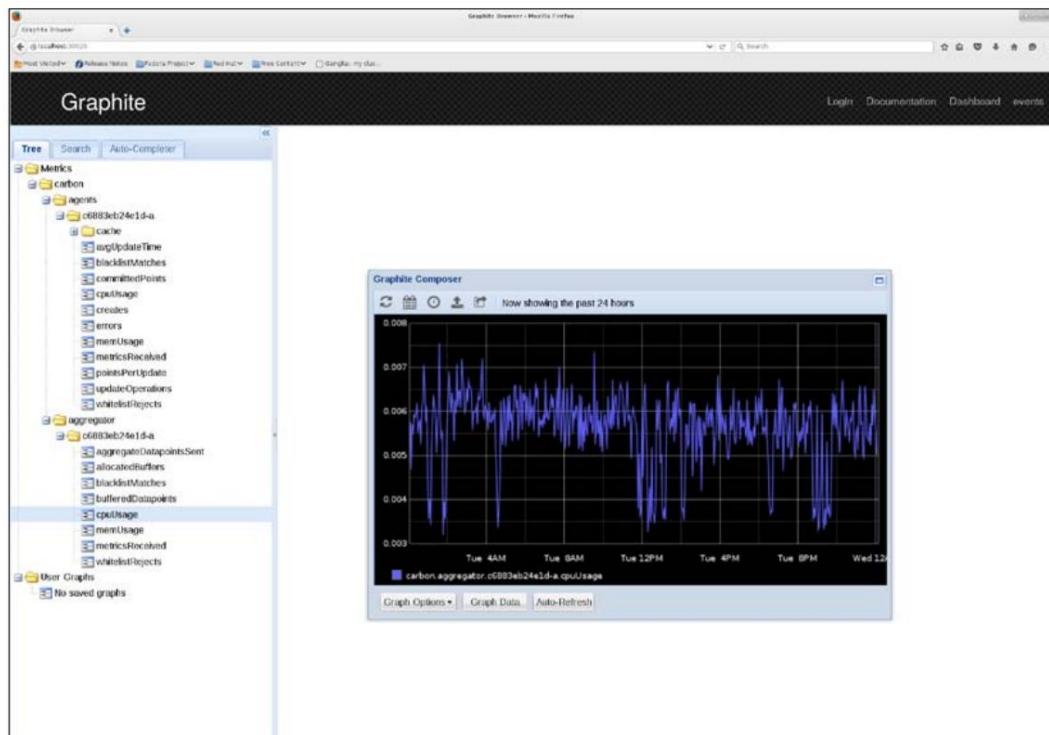
这将启动一个运行 Graphite 并带有 HTTP 基本身份验证的 Docker 容器。

您现在可以查看 Graphite 的用户界面。输入 “guest”作为用户名和密码。您可以通过为图像提供.htpasswd文件来更改用户名和密码。

如果您不更改上面 Docker 语句中的端口,则此 URL 应该有效:

<http://localhost:30020/>

浏览器窗口应显示类似于以下屏幕截图的视图:



[监控代码](#)

Graphite 具有可定制的用户界面,您可以在仪表板中将您感兴趣的图形组织在一起。还有一个完成小部件,可让您通过键入图形名称的前几个字符来查找命名图形。

日志处理

日志处理是一个非常重要的概念,我们将探讨许多选项中的一些,例如ELK (Elasticsearch、Logstash和Kibana)堆栈。

传统上,日志记录只是在代码中使用简单的打印语句来跟踪代码中的事件。这有时称为printf式调试,因为您使用跟踪来查看代码的行为方式,而不是使用常规调试器。

这是 C 语法中的一个简单示例。这个想法是我们想知道什么时候进入函数fn(x)以及参数x的值是什么:

```
void fn(char *x)
    { printf( DEBUG 进入 fn,x 是 %s\n ,x);
    ...
}
```

从控制台中的调试跟踪,您可以确定正在开发的程序是否按预期运行。

当然,你也想看看你的程序是否有严重的错误,并以更高的优先级报告:

```
printf( ERROR x 不能为空字符串\n );
```

这种调试方式有几个问题。当您想知道程序的行为方式时,它们很有用,但当您完成开发并想要部署代码时,它们就没那么有用了。

如今,基于上述经过时间验证的模式,有大量框架以多种不同方式支持日志记录。

日志框架主要通过定义标准和提供改进的功能来为 printf 风格的日志增加价值,例如:

- 不同的日志优先级,例如调试、警告、跟踪和错误。 · 过滤不同优先级的日志消息。您可能并不总是对调试跟踪感兴趣,但您可能总是对错误消息感兴趣。

- 记录到不同的目的地,例如文件、数据库或网络守护进程。这包括我们稍后将访问的 ELK 堆栈。
- 日志文件的循环和归档。可以存档旧日志文件。

时不时地,一个新的日志框架会弹出,所以日志问题领域似乎直到今天还远未穷尽。这种趋势是可以理解的,因为正确完成日志可以帮助您确定不再运行的网络服务失败的确切原因,当然,也可以帮助您确定您没有经常监督的任何复杂服务。日志记录也很难做到正确,因为过多的日志记录会降低服务的性能,而太少则无法帮助您确定失败的原因。因此,日志记录系统竭尽全力在日志记录的各种特征之间取得平衡。

客户端日志记录库

Log4j是一种流行的 Java 日志记录框架。其他语言有几个端口,例如:

- C 语言的 Log4c · JavaScript 语言的 Log4js
- Apache log4net,
- Microsoft .NET 框架的端口

存在其他几个端口,它们具有许多共享许多概念的不同日志记录框架。

由于仅针对 Java 平台就有许多日志记录框架,因此也有一些包装器日志记录框架,例如Apache Commons Logging或Simple Logging Facade for Java (SLF4J)。这些旨在允许您使用单个接口并在需要时换出底层日志记录框架实现。

Logback是 log4j 的继任者,并且与 ELK 堆栈兼容。

Log4j 是第一个 Java 日志记录框架,本质上为您提供了与以前的printf语句等效的功能,但具有更多的功能。

Log4j 使用三个主要结构:

- 记录器
- 附加程序
- 布局

监控代码

记录器是您用来访问记录方法的类。有一个记录方法可以调用每个日志严重性。记录器也是分层的。

这些概念最容易用一些示例代码来解释：

```
记录器记录器 = Logger.getLogger( se.matangle );
```

这为我们提供了一个特定于我们组织的记录器。通过这种方式，我们可以从源自我们可能正在使用的其他组织代码的消息中找出我们自己的记录器消息。这在 Java 企业环境中变得非常有用，您可能有许多库都使用 log4j 进行日志记录，并且您希望能够为每个库配置不同的日志级别。

我们可以同时使用多个记录器以获得更大粒度的日志。

不同的软件组件可以有不同的记录器。

这是一个视频录制应用程序的示例：

```
记录器 videologger = Logger.getLogger( se.matangle.video ); logger.warn( disk is dangerously
low )
```

我们可以使用几种不同的日志级别，为我们的日志增加更多的特异性：

```
videologger.error( 视频编码源过早用完 )
```

日志消息最终结束的地方在 log4j 术语中称为附加程序。有许多附加程序可用，例如控制台、文件、网络目标和记录器守护程序。

布局控制我们的日志消息的格式。它们允许使用类似 printf 格式的转义序列。

例如，配置为使用转换模式%r [%t] %-5p %c - %m%n 的类 PatternLayout 将创建以下示例输出：

```
176 [main] INFO se.matangle - 用户发现
```

以下是模式中字段的含义：

- 第一个字段是自开始以来经过的毫秒数
该程序
- 第二个字段是发出日志请求的线程 · 第三个字段是日志语句的
级别 · 第四个字段是与日志请求关联的记录器的名称 · -之后的
文本是语句的消息

Log4j 努力使日志记录的配置在应用程序外部。

通过外部化日志记录配置,开发人员可以通过将日志配置为转到文件或控制台来使日志记录在本地工作。稍后,当应用程序部署到生产服务器时,管理员可以将日志附加器配置为其他东西,例如我们稍后要讨论的 ELK 堆栈。这样,不需要更改代码,我们可以在部署时修改日志记录的行为和目标。

像 WildFly 这样的应用程序服务器提供了自己的配置系统,可以插入到 log4j 系统中。

如果您不使用应用程序服务器,较新版本的 log4j 支持许多不同的配置格式。这是一个 XML 样式的文件,它将在类路径中查找 log4j2-test.xml :

```
<?xml version= "1.0" encoding= "UTF-8" ?><Configuration  
status= "WARN" ><Appenders>  
  
    <Console name= "Console" target= "SYSTEM_OUT" >  
        <PatternLayout pattern= "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36}  
- %msg%n" /></Console></Appenders><Loggers>  
  
    <根级别= "错误" >  
        <AppenderRef ref= "控制台" />  
    </根>  
    </Loggers>  
</Configuration>
```

麋鹿栈

ELK 堆栈由以下组件组成:

- Logstash :这是一个类似于log4j 的开源日志记录框架。
Logstash 的服务器组件处理传入的日志。 · Elasticsearch :存储所有日志并对其进行索引,如名称
暗示,用于搜索。 ·
- Kibana :这是用于日志搜索和可视化的网络界面。

监控代码

为了了解它是如何工作的,我们将跟踪一条日志消息,从它在代码内部的起源位置,通过中间网络层,一直到网络运营商屏幕上的最终目的地:

1. 在代码深处,发生了Java异常。应用程序意外地无法将导入文件中存在的用户 ID 映射到数据库中的用户 ID。使用 log4j 记录以下错误: logger.error(cant find imported user id in database)

2. log4j 系统配置为将错误记录到 Logstash 后端,
如下:

```
log4j.appenders.applog=org.apache.log4j.net.SocketAppender log4j.appenders.applog.port=5000
log4j.appenders.applog.remoteHost=master log4j.appenders.applog.DatePattern= . yyyy-MM-
dd log4j .appenders.applog.layout=org.apache.log4j.PatternLayout
log4j.appenders.applog.layout.ConversionPattern=%d %-5p [%t] %c %M
```

```
- %m%on
log4j.rootLogger=警告,应用日志 log4j.logger.nl=调试
```

3. Logstash系统配置如下,监听log4j系统描述的端口。 Logstash 守护进程必须与您的应用程序分开启动,如下所示: input

```
{ log4j {
```

```
    模式=>服务器
    主机=> "0.0.0.0"
    端口 => 5000 类型 =>
        "log4j"
}
```

```
} output
{ stdout { codec => rubydebug } stdout {}
elasticsearch { cluster => "elasticsearch"
```

```
}
}
```

4. Elasticsearch引擎接收日志输出并使其可搜索。

5. 管理员可以打开Kibana GUI应用,查看日志
实时出现或搜索历史数据。

您可能认为这太复杂了,您是对的。但有趣的是,log4j 可以配置为支持这种复杂的场景以及更简单的场景,因此使用 log4j 或其兼容的竞争对手之一真的不会出错。

您可以从不使用任何网络日志记录开始,只使用普通日志文件。

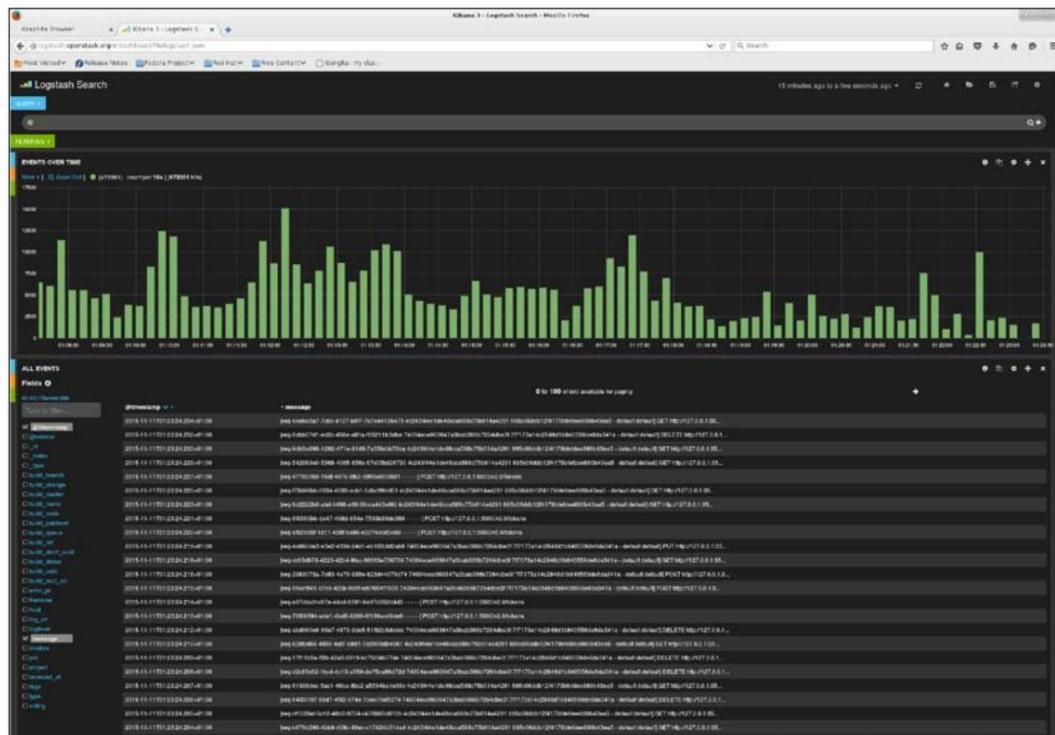
这对于许多情况来说已经足够了。如果以及何时需要 ELK 堆栈的额外功能,您可以稍后添加它。保留日志文件也很有用。如果高级日志工具由于某种原因失败,您总是可以求助于对日志文件使用传统的 Unix grep实用程序。

Kibana 有很多功能可以帮助您分析日志数据。您可以为您正在寻找的特定模式绘制数据图并过滤日志数据。

要在实践中尝试所有这些,我们可以使用官方的 Kibana 和 Elasticsearch Docker Hub 上可用的 Docker 图像:

```
docker run -d elasticsearch &&
docker run --link some-elasticsearch:elasticsearch -d kibana
```

如果一切顺利,我们将能够访问 Kibana 用户界面,如下所示:



[监控代码](#)

概括

在本章中,我们了解了一些可用于监控已部署代码的选项: Nagios 用于监控主机和服务的状态; Munin、Ganglia 和 Graphite 用于绘制来自我们主机的统计数据;以及 log4j 和 ELK 堆栈来跟踪日志信息。

在下一章中,我们将了解有助于开发组织内部工作流程的工具,例如问题跟踪器。

9

问题跟踪

在上一章中,我们研究了如何使用监控和日志处理解决方案来关注我们部署的代码。

在本章中,我们将研究处理组织内开发工作流程的系统,例如问题跟踪软件。这样的系统在实施敏捷流程时是一个重要的帮助。

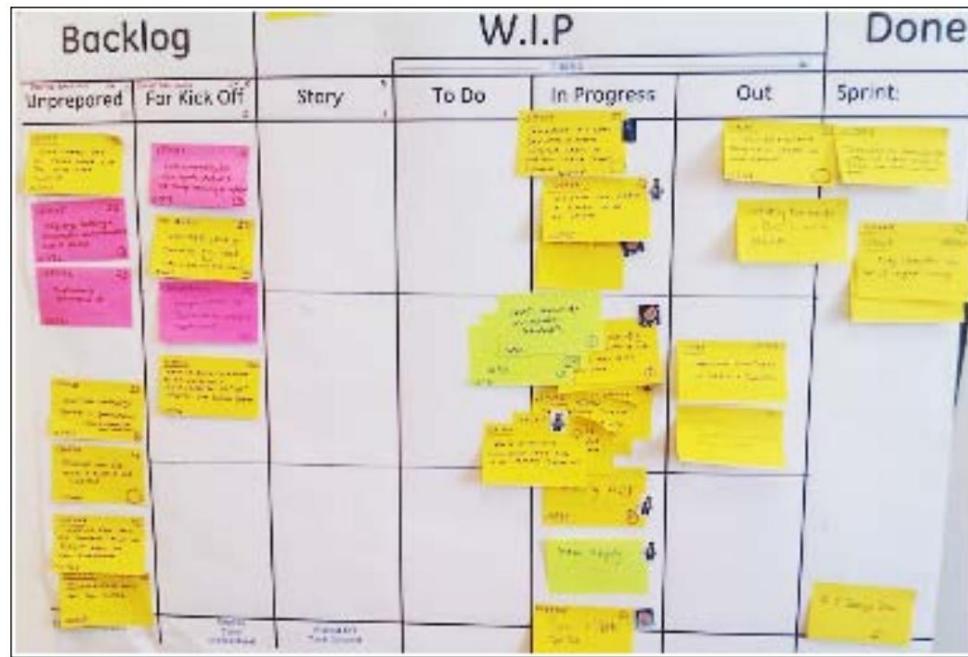
问题跟踪器有什么用?

从敏捷过程的角度来看,问题跟踪器用于帮助解决敏捷过程的细节问题。问题跟踪器处理的实体可能代表工作项、错误和问题。大多数敏捷流程都包含关于如何管理敏捷团队将要执行的任务的想法,其形式是在白板上贴便利贴或电子等价物。

在敏捷环境中工作时,手写便利贴上的问题板很常见。这是看板方法中的核心概念,因为看板在日语中实际上是招牌的意思。该板很好地概述了正在进行的工作,并且非常易于管理,因为您只需在板上四处移动便签纸即可表示工作流中的状态变化。

问题跟踪

只需重写您写在看板上的各种标记（例如车道）,即可轻松更改看板:



物理板在 Scrum 团队中也很常见。

另一方面,大多数基于网络的计算机化问题跟踪器提供了更好的细节,并且可以在远程工作时使用。这些问题跟踪器还可以帮助您记住流程中的步骤以及其他好处。

理想情况下,人们会同时想要一个物理板和一个问题跟踪器,但要使它们保持同步需要做很多工作。没有简单的方法来解决这个基本争论。

有些人使用投影仪来显示问题跟踪器,就好像它是一块板子一样,但它没有相同的触觉,并且聚集在投影图像或监视器周围与聚集在一块物理板周围是不同的。当团队成员想要讨论任务时,看板还具有随时可用和方便参考的优势。

从技术上讲,问题跟踪器通常实现为代表状态机的对象数据库,具有不同程度的复杂性。您使用 Web 界面或其他方式 (例如电子邮件或自动化) 创建问题;然后,这个issue因为人际交互经历了各种状态,最后这个issue以关闭的状态归档,以供日后参考。有时,由于来自另一个系统的交互,流程会继续进行。一个简单的例子可能是任务不再被视为有效,因为它已过期并自动关闭。

工作流程和问题的一些示例

尽管本章通篇使用术语“问题”,但某些系统使用术语“票证”或“错误”。从技术上讲,它们是同一回事。问题也可能表示待办事项、增强请求或任何其他类型的工作项。从技术上讲,增强与错误基本上是一样的,这可能有点违反直觉,但如果您将增强视为缺失的功能,它就会开始有意义。

一个问题有各种与之关联的元数据,这取决于它代表什么以及问题跟踪器支持什么。最基本的问题类型非常简单,但很有用。它具有以下基本属性:

·描述:这是问题的自由格式文本描述 ·报告者:这代表打开问题的人 ·分
配:这是应该处理该项目的人。

此外,它有两种状态:打开和关闭。

这通常是问题跟踪器提供的最低限度。如果将其与便利贴进行比较,唯一额外的元数据是报告者,您也可以删除它,跟踪器仍然可以使用。使用便利贴,您只需将便利贴放在板上或将便利贴取下即可处理打开和关闭状态。

当项目经理第一次遇到电子问题跟踪器时,往往会过度增加跟踪器的状态机和属性存储的复杂性。也就是说,显然有很多有用的信息可以添加到问题跟踪器中而不过分。

问题跟踪

除了前面提到的基本信息外,这些附加属性通常非常有用:

·**到期日**:问题预计得到解决的日期。 ·**里程碑**:里程碑是一种将问题组合成比单个问题更大的有用工作包的方法。例如,里程碑可以表示 Scrum 冲刺的输出。里程碑通常也有截止日期,如果您使用里程碑功能,通常不会使用个别问题的截止日期。

·**附件**:能够附上屏幕截图和问题的文档,这可能对处理问题的开发人员或验证它的测试人员有用。 ·**工作估算**:估算解决问题所需的预期工作支出可能很有用。这可用于规划目的。同样,包含在该问题上花费的实际时间的字段也可用于各种计算。另一方面,做估算总是很棘手,到头来可能会得不偿失。许多团队在他们的问题跟踪器中没有此类信息的情况下做得很好。

以下也是一些有用的状态,可用于比普通的打开和关闭状态更好地建模敏捷工作流。为清楚起见,此处重复打开和关闭状态:

·**开放**:问题已报告,目前还没有人处理 ·**进行中**:有人已分配到问题并致力于解决 ·**准备测试**:问题已完成,现在可以进行验证。它又被取消了 ·**测试**:有人被分配来测试实施

问题的

·**完成**:任务再次标记为就绪且未分配。完成状态用于标记问题以跟踪它们直到冲刺结束,例如,如果团队正在使用 Scrum 方法

·**已关闭**:该问题不再受监控,但仍保留以供参考

在最好的情况下,以有序的方式发布从一个状态到下一个状态的进展。在更现实的情况下,任务很可能从测试变为打开而不是完成,因为测试可能会揭示问题并未真正得到妥善解决。

我们需要问题跟踪器提供什么？

除了支持前面描述的基本工作流之外，我们还需要问题跟踪器做什么？有很多担忧，其中一些并不立即显现。下面列出了一些需要考虑的事项：

- 我们需要什么规模？

大多数工具在最多约 20 人的规模上运行良好，但除此之外，我们还需要考虑性能和许可要求。我们需要能够跟踪多少问题？有多少用户需要访问问题跟踪器？这些是我们可能会有的一些问题。

- 我们需要多少许可证？

在这方面，免费软件占了上风，因为专有软件的定价可能不直观。免费软件可以是免费的，带有可选的支持许可。

本章提到的大多数问题跟踪器都是免费软件，Jira 除外。

- 是否存在性能限制？

性能通常不是限制因素，因为大多数跟踪器使用生产就绪的数据库，如 PostgreSQL 或 MariaDB 作为后端数据库。本章中描述的大多数问题跟踪器在通常与组织内部问题跟踪器关联的规模上表现良好。

Bugzilla 已在处理大量问题并面向公共 Internet 的安装中得到证明。

尽管如此，最好还是评估您打算部署的系统的性能。也许您的预期用途的某些特殊性会触发一些性能问题。一个例子可能是大多数问题跟踪器使用关系数据库作为后端，如果需要进行递归查询，关系数据库并不是表示分层树结构的最佳选择。

在正常用例中，这不是问题，但如果您打算大规模使用深度嵌套的问题树，问题可能会浮出水面。

问题跟踪

- 有哪些支持选项可用,它们的质量如何?

在实际使用之前很难确定支持的质量,但这里有一些有助于评估的想法:

- 对于开源项目,支持取决于用户社区的规模。此外,通常还有可用的商业支持提供商。
- 商业问题跟踪器可以同时获得付费和社区支持。

- 我们能否使用异地托管的问题跟踪器?

有许多公司提供问题跟踪器。其中包括 Atlassian 的 Jira 跟踪器,它既可以安装在客户场所,也可以由 Atlassian 托管。托管问题跟踪器的另一个示例是 Trello, Inc. 的 Trello。

在您自己的网络中部署问题跟踪器通常并不难。就安装复杂性而言,它们属于本书中描述的较简单的系统。通常,您需要一个数据库后端和一个 Web 应用程序后端层。通常,最困难的部分是让身份验证服务器和邮件服务器集成正常工作。尽管如此,即使托管您自己的问题跟踪器安装并不难,但不可否认的是使用托管跟踪器更容易。

出于法律或其他原因,某些组织不能将有关其工作的数据泄露到自己的网络之外。对于这些组织,托管问题跟踪器不是一种选择。他们必须在他们的内部部署问题跟踪器

自己的网络。

- 问题工作流系统是否适应我们的需要?

一些系统,如 Jira,允许高度灵活的状态机来定义流程和集成编辑器来编辑它。其他人有更简约的流程。

适合您的解决方案取决于您的需求。有些人一开始想要一个非常复杂的流程,最后意识到他们只需要打开和关闭状态。其他人意识到他们需要复杂的流程来支持他们流程的复杂性。

可配置的流程在大型组织中很有用,因为它可以封装有关可能不太明显的流程的知识。例如,当错误被解决后,质量保证部门的 Mike 应该验证修复。这在小型组织中用处不大,因为无论如何每个人都应该知道该做什么。然而,在大型组织中,情况往往恰恰相反。在决定下一步应该由谁来处理任务以及做什么时,获得一些帮助是很有用的。

- 问题跟踪器是否支持我们选择敏捷方法？

本章提到的大多数问题跟踪器都有足够灵活的底层数据模型来表示任何类型的敏捷过程。从某种意义上说，您真正需要的只是一个可配置的状态机。当然，问题跟踪器可能会添加额外的特性来为特定的敏捷方法和工作流提供更好的支持。

这些支持敏捷方法的额外功能通常归结为两个主要类别：可视化和报告。虽然拥有这些很不错，但我的观点是过于关注这些功能，尤其是缺乏经验的团队领导者。可视化和报告确实增加了价值，但没有人们一开始认为的那么多。



在为问题跟踪器选择功能时请记住这一点。
确保您将实际使用您一直想要的令人印象深刻的报告。

- 该系统是否容易与我们组织中的其他系统集成？

问题跟踪器可能集成的系统示例包括代码存储库、单点登录、电子邮件系统等。例如，损坏的构建可能会导致构建服务器在问题跟踪器中自动生成一个票证，并将其分配给破坏构建的开发人员。

由于我们在这里主要关注开发工作流程，因此我们需要的主要集成是与代码存储库系统的集成。我们在这里探索的大多数问题跟踪器都以某种形式与代码存储库系统集成。

- 跟踪器可以扩展吗？

将某种形式的 API 作为问题跟踪器通常很有用

扩展点。API 可用于集成到其他系统，还可以自定义跟踪器，使其适合我们组织的流程。

也许您需要以 HTML 格式显示当前未解决的问题，以便在可公开查看的监视器上显示。这可以通过一个好的 API 来完成。也许您希望问题跟踪器触发部署系统中的部署。这也可以通过一个好的问题跟踪器 API 来完成。

一个好的 API 可以打开许多否则会被关闭的可能性
为我们。

问题跟踪

- 跟踪器是否提供多项目支持？

如果您有多个团队和项目,为每个项目设置单独的问题跟踪器可能会很有用。因此,问题跟踪器的一个有用特性是能够将其自身划分为多个子项目,每个子项目在系统内都有一个独立的问题跟踪器。

像往常一样,需要权衡取舍。如果您有许多独立的问题跟踪器,您如何将问题从一个跟踪器移动到另一个?当您有多个团队处理不同的任务集时,这是必需的;例如,开发团队和质量保证团队。从 DevOps 的角度来看,如果工具将团队拉得更远而不是拉近他们之间的距离,那就不好了。

因此,虽然每个团队在主系统中拥有自己的问题跟踪器是件好事,但也应该很好地支持将多个团队作为一个整体一起处理所有任务。

- 问题跟踪器是否支持多个客户端?

虽然通常不将功能视为选择问题跟踪器的关键验收要求,但通过多个客户端和界面访问问题跟踪器可能很方便。例如,开发人员在其集成开发环境中工作时能够访问问题跟踪器非常方便。

一些使用自由软件的组织更喜欢完全基于电子邮件的工作流程,例如 Debian 的调试。不过,此类要求在企业环境中很少见。

问题跟踪器扩散的问题

由于现在设置问题跟踪器在技术上非常容易,因此团队经常会设置自己的问题跟踪器来处理自己的问题和任务。

许多其他类型的工具(例如编辑器)都会发生这种情况,但编辑器属于开发人员的个人工具,其主要用例不是与其他人共享协作界面。因此,问题跟踪器激增是一个问题,而编辑器激增则不是。

导致问题跟踪器扩散问题的一个原因是大型组织可能会标准化一种实际上很少有人喜欢使用的跟踪器。

一个典型的原因是在决定问题跟踪器时只考虑了一种类型的团队的需求,例如质量保证团队或运营团队。另一个原因是只为问题跟踪器购买了有限数量的许可证,团队的其他成员将不得不自己凑合。

开发人员使用一种跟踪器,质量保证团队使用另一种,而运营团队使用另一种完全不同且不兼容的系统也很常见。

因此,虽然拥有许多不同的问题跟踪系统来做同样的事情对于整个组织而言并不是最佳选择,但对于能够在该领域选择自己的工具的团队来说,这可能被视为一种胜利。

同样,DevOps 的核心思想之一是让不同的团队和角色更紧密地联系在一起。相互不兼容的协作工具的激增在这方面无济于事。这主要发生在大型组织中,通常没有简单的解决方案。

如果您使用免费软件跟踪器,您至少可以摆脱有限许可问题。

当然,找到一个在各个方面都令所有人满意的系统要困难得多。它可能有助于限制选择标准的范围,这样问题跟踪器工具而不是管理,取悦实际将整天使用它的人。

这通常也意味着减少对报告漂亮的燃尽图的关注,而更多地关注实际完成工作。

所有跟踪器接下来,我们将探索一系

列不同的问题跟踪器系统。在您进行实际部署之前,它们都很容易试用。大多数都是免费的,但也提到了一些专有的替代品。

此处提到的所有跟踪器都出现在维基百科的比较页面上,网址为https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems。

问题跟踪

由于我们只能探索部分问题跟踪器,因此选择了以下问题跟踪器,因为它们因不同的设计选择而表现出差异。 Bugzilla 专为大规模面向公众的跟踪器而设计。 Trac 专为简单性和工具集成而设计。 Redmine 作为具有问题跟踪器的功能齐全的项目管理工具。选择 GitLab 跟踪器是为了简单,而 Git 集成和 Jira 是为了可用性。

我们从 Bugzilla 开始我们的问题跟踪器探索,因为它是最早的问题跟踪器之一并且仍然很受欢迎。这意味着它的流程非常成熟,因为它们已经在很长一段时间内证明了自己。

臭虫

Bugzilla 可以说是本章描述的所有问题跟踪器系统的鼻祖。 Bugzilla 自 1998 年以来一直存在,并被许多知名组织使用,例如 Red Hat、Linux 内核项目和 Mozilla 项目。

如果您曾经报告过这些产品之一的错误,那么您很可能已经遇到过 Bugzilla。

Bugzilla 是由 Mozilla 项目维护的免费软件,该项目还生产著名的 Firefox 浏览器。 Bugzilla 是用 Perl 编写的。

毫不奇怪,Bugzilla 专注于跟踪错误而不是处理其他问题类型。处理其他类型的任务,例如提供 wiki,需要单独完成。

许多组织使用 Bugzilla 作为可以报告问题的面向公众的工具。因此,Bugzilla 具有良好的安全特性。

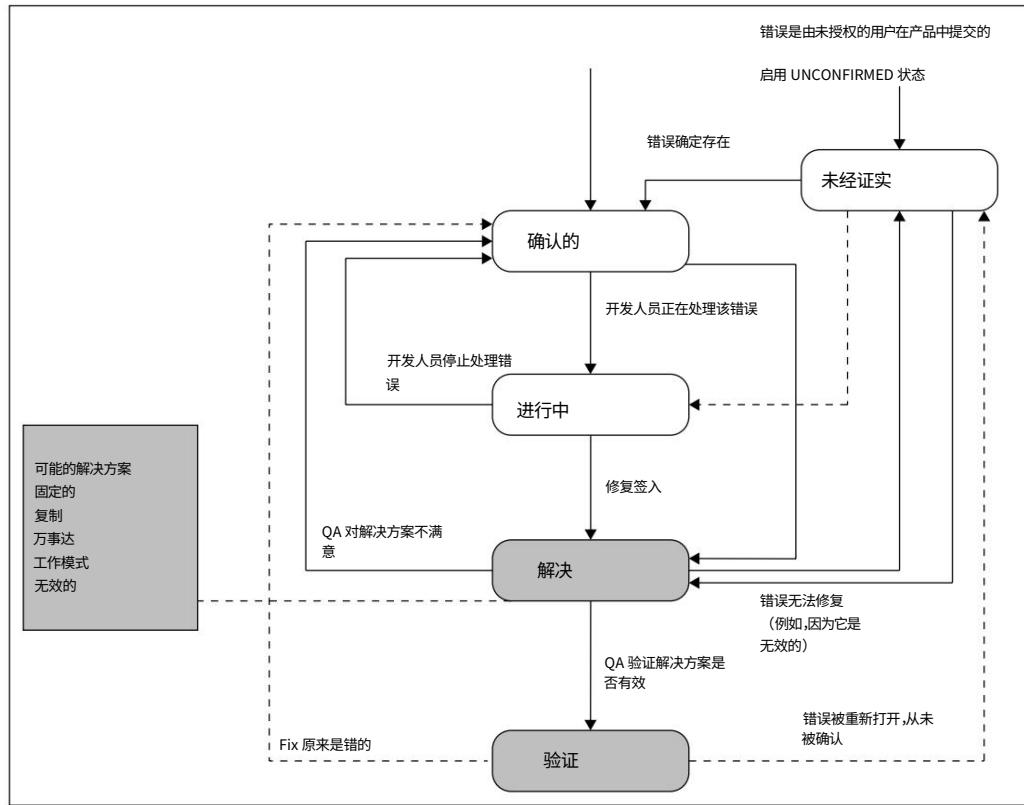
有一个演示安装,可以在https://landfill.bugzilla.org/bugzilla_tip/ 上测试 Bugzilla 开发版本的最新功能。

Bugzilla 支持自定义。可以向问题添加自定义字段并自定义工作流程。

Bugzilla 同时具有 XML-RPC 和 JSON REST API。

由于 Bugzilla 已经存在了很长时间,因此有许多可用的扩展和插件,包括替代客户端。

下图显示了 Bugzilla 错误状态机或工作流：



此工作流是可配置的,但默认状态如下:

- 未确认:非授权用户的新错误从未确认状态开始
- 已确认:如果确认错误值得调查,则将其放入确认状态
- 进行中:当开发人员开始解决错误时,使用进行中状态
- 已解决:当开发人员认为他们已经完成修复错误时,他们将其置于已解决状态
- 已验证:当质量保证团队同意时该错误已修复,将其置于已验证状态

问题跟踪

让我们现在试试 Bugzilla。

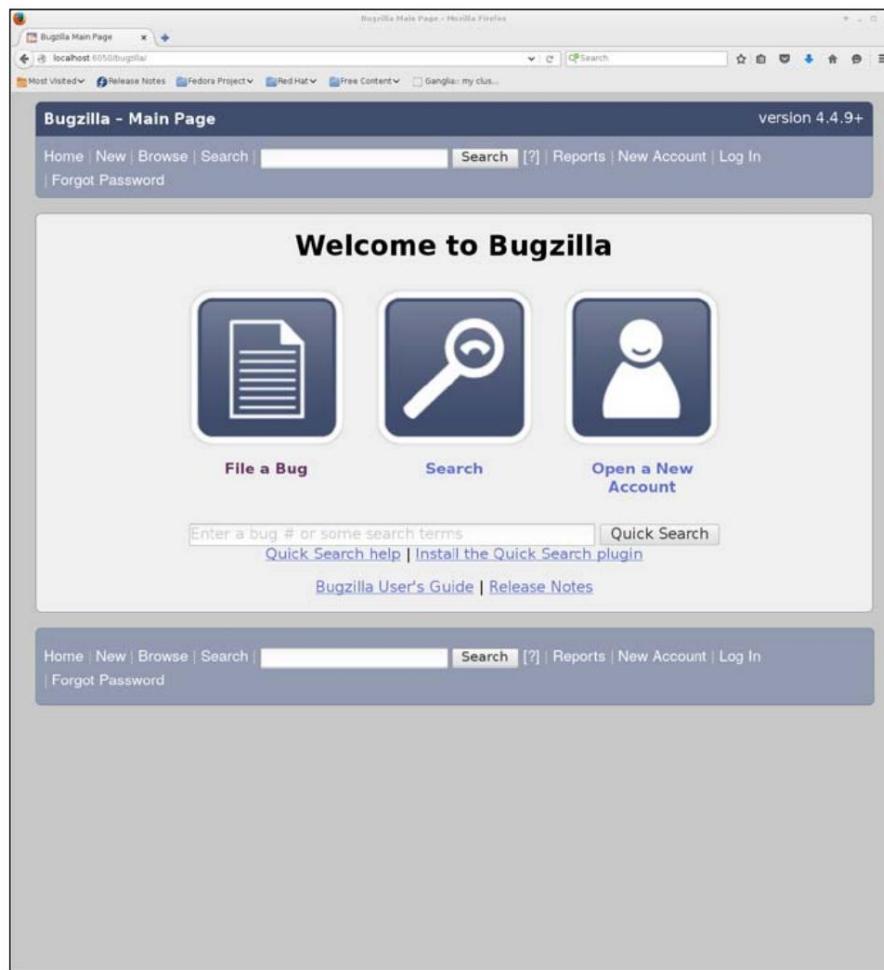
有几个 Docker hub 镜像可用于 Bugzilla,它很可能在您的 Linux 发行版的存储库中也可用。

这将在一个单独的容器中启动 Bugzilla,并完成一个合适的 MySQL 配置。端口是任意选择的：

```
docker run -p 6050:80 dklawren/docker-bugzilla
```

现在尝试访问界面,网址为http://localhost:6050。

你会看到这样的景象：



Bugzilla 要求您拥有一个帐户才能提交错误。您可以使用管理员帐户或创建一个新帐户。

可以使用管理员用户名admin@bugzilla.org登录,密码初始为password。现在查看以下创建和解决错误的步骤：

1. 选择开设新账户。这将提示您输入电子邮件地址。

带有链接的确认邮件将发送到该地址。确保链接中的端口号正确;否则,创建它。现在应该创建您的帐户。

2. 尝试使用File a Bug选项创建一个新错误。 Bugzilla 需要你

提供错误的产品和组件。有可用的TestProduct和TestComponent选项;我们可以使用它们来报告我们的错误：

The screenshot shows the Bugzilla 'Enter Bug' interface for the 'TestProduct' project. The 'Component' field is set to 'TestComponent'. The 'Summary' field contains 'a horrible bug in our product'. The 'Description' field contains 'Indeed a horrible bug, it should be fixed'. The 'Severity' dropdown is set to 'enhancement'. The 'OS' dropdown is set to 'Linux'. The 'Hardware' dropdown is set to 'PC'. The 'Version' dropdown is set to 'unspecified'. The 'Reporter' field shows 'admin@bugzilla.org'. The 'Show Advanced Fields' link is visible above the component field.

问题跟踪

3. 既然漏洞已经公开,我们就可以对其进行评论了。每次我们对错误发表评论时,我们也可以根据我们的访问级别将其移动到新状态。尝试添加一些评论,如以下屏幕截图所示:

The screenshot shows a web-based bug tracking system. At the top, there's a header bar with tabs like 'release Notes', 'Fedora Project', 'Red Hat', 'Free Content', and 'Ganglia: my clus...'. Below the header, a message says 'First Last Prev Next This bug is not in your last search results.' The main area displays a bug entry for 'Bug 1 - a horrible bug in our product'.

Bug 1 - a horrible bug in our product (edit)

Status: CONFIRMED ([edit](#)) **Reported:** 2015-11-29 21:01 UTC by [Admin](#)

Product: TestProduct **Modified:** 2015-11-29 21:02 UTC ([History](#))

Component: TestComponent **CC List:** Add me to CC list
0 users ([edit](#))

Version: unspecified **Hardware:** PC | Linux

Importance: enhancement **See Also:** ([add](#))

Assigned To: Admin ([edit](#))

URL: [input field]
Tags: [input field]

Depends on: [input field]
Blocks: [input field]

Show dependency [tree / graph](#)

Orig. Est.:	Current Est.:	Hours Worked:	Hours Left:	%Complete:	Gain:	Deadline:
0.0	0.0	0.0 + 0	0.0	0	0.0	[calendar icon]

Summarize time (including time for bugs blocking this bug)

Attachments
[Add an attachment](#) (proposed patch, testcase, etc.)

Additional Comments:

Status: CONFIRMED **Save Changes**

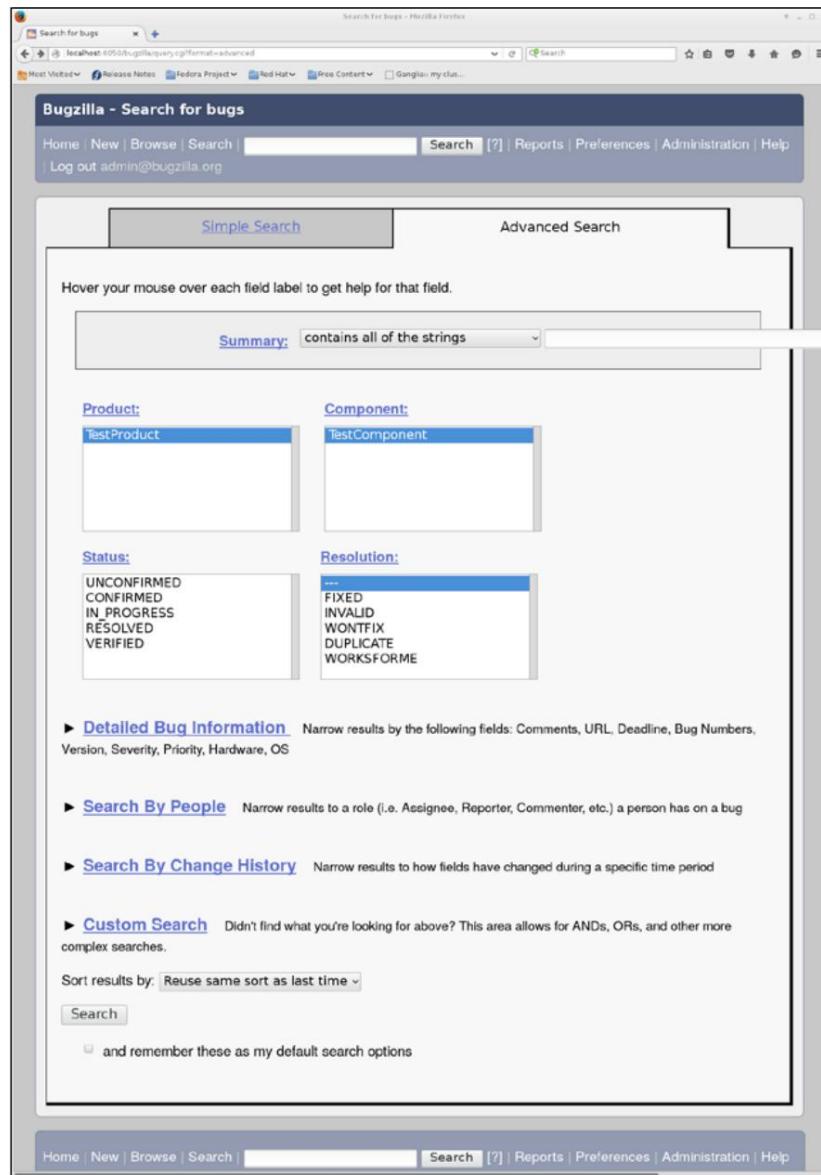
Mark as Duplicate

Admin 2015-11-29 21:01:51 UTC **Description** [[reply](#)] [-] [Collapse All Comments](#)
indeed a horrible bug, it should be fixed

Admin 2015-11-29 21:02:09 UTC **Comment 1** [[reply](#)] [-] [Expand All Comments](#)
we agree this bug is pretty bad

[Add Comment](#)

4. 尝试搜索错误。有一个简单的搜索页面和一个高级的搜索页面。高级页面如以下屏幕截图所示。它可以让您搜索大多数字段组合：



5. 最后,完成后,将错误标记为已解决。

问题跟踪

如您所见,Bugzilla 具有问题跟踪器应具有的所有功能。它适用于针对处理大量产品和组件中的大量错误而优化的工作流程。如果您对 Bugzilla 开箱即用的工作方式感到满意,那么开始使用 Bugzilla 是相当容易的。

定制需要一些额外的努力。

Trac

Trac 是一个相当容易设置和测试的问题跟踪器。 Trac 的主要吸引力在于它体积小,并且将多个系统集成在一个简洁的模型中。 Trac 是最早的问题跟踪器之一,它表明问题跟踪器、wiki 和存储库查看器可以有益地集成在一起。

Trac 是用 Python 编写的,并在 Edgewall 的免费软件许可下发布。

它最初是根据 GPL 许可的,但自 2005 年以来,它已根据 BSD 样式许可获得许可。

通过使用插件架构,Trac 是高度可配置的。有许多可用的插件,其中一些列在<https://trac-hacks.org> 上。

Trac 以保守的速度发展,许多用户对此非常满意。

Trac 还有一个 XML-RPC API,可用于读取和修改问题。

Trac 过去只与 Subversion 集成,但现在支持 Git。

Trac 还具有集成的 wiki 功能,这使它成为一个相当完整的系统。

有几个可用的 Docker hub Trac 映像,并且 Trac 通常也可以在 Linux 发行版的存储库中使用。

本书的代码包中有一个适用于 Trac 的 Dockerfile,我们也可以使用它。看看下面的命令:

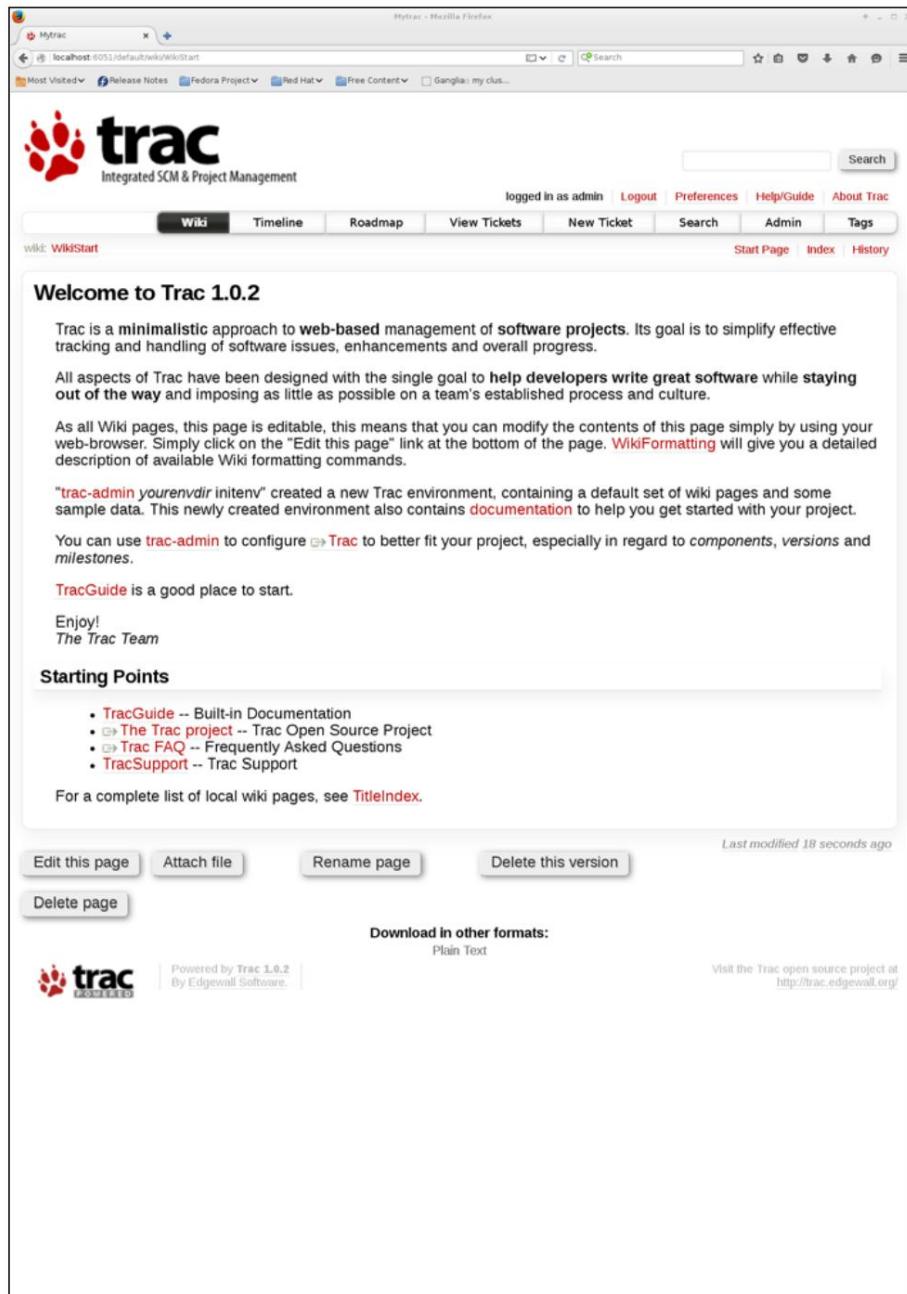
```
docker run -d -p 6051:8080 巴罗吉/trac:1.0.2
```

这将使 Trac 实例在端口 6051 上可用。让我们试一试:

- 访问<http://localhost:6051/>显示可用项目列表。
一开始,只有默认项目可用。

第9章

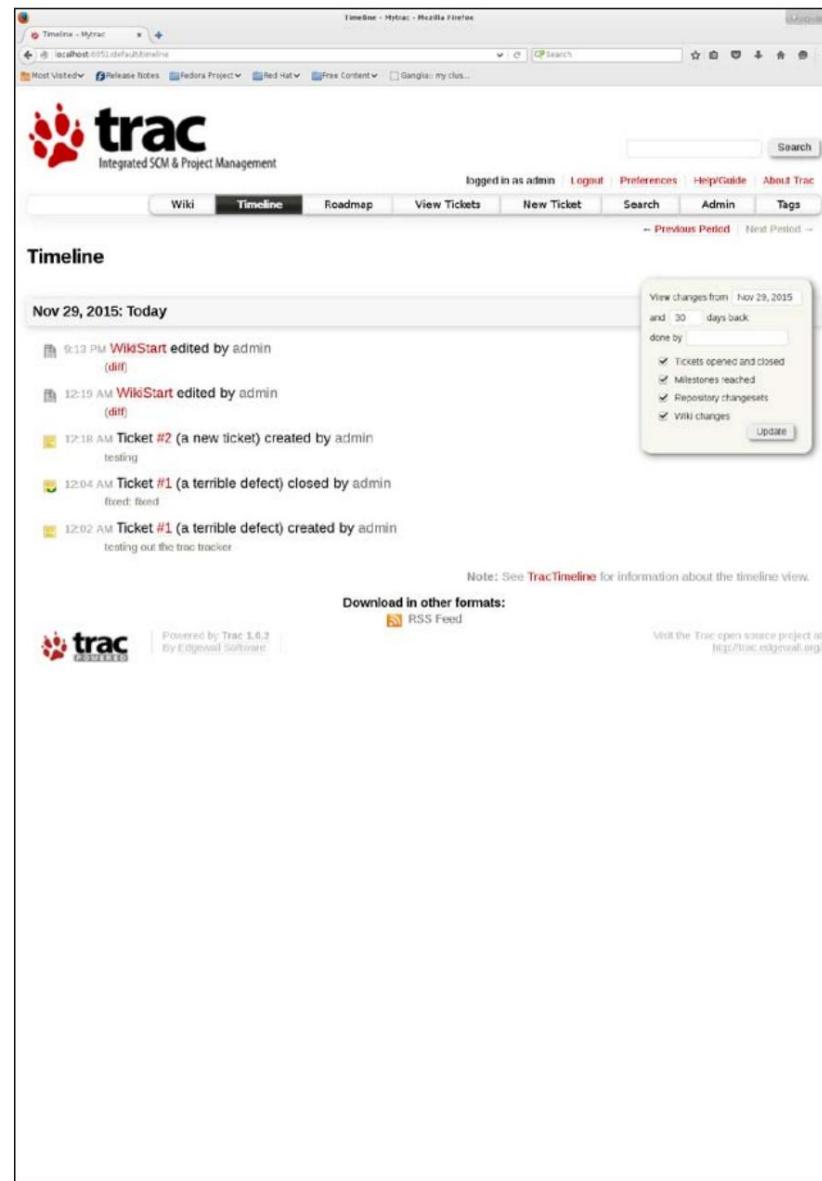
- 右侧的菜单行显示默认可用的基本功能,如以下屏幕截图所示:



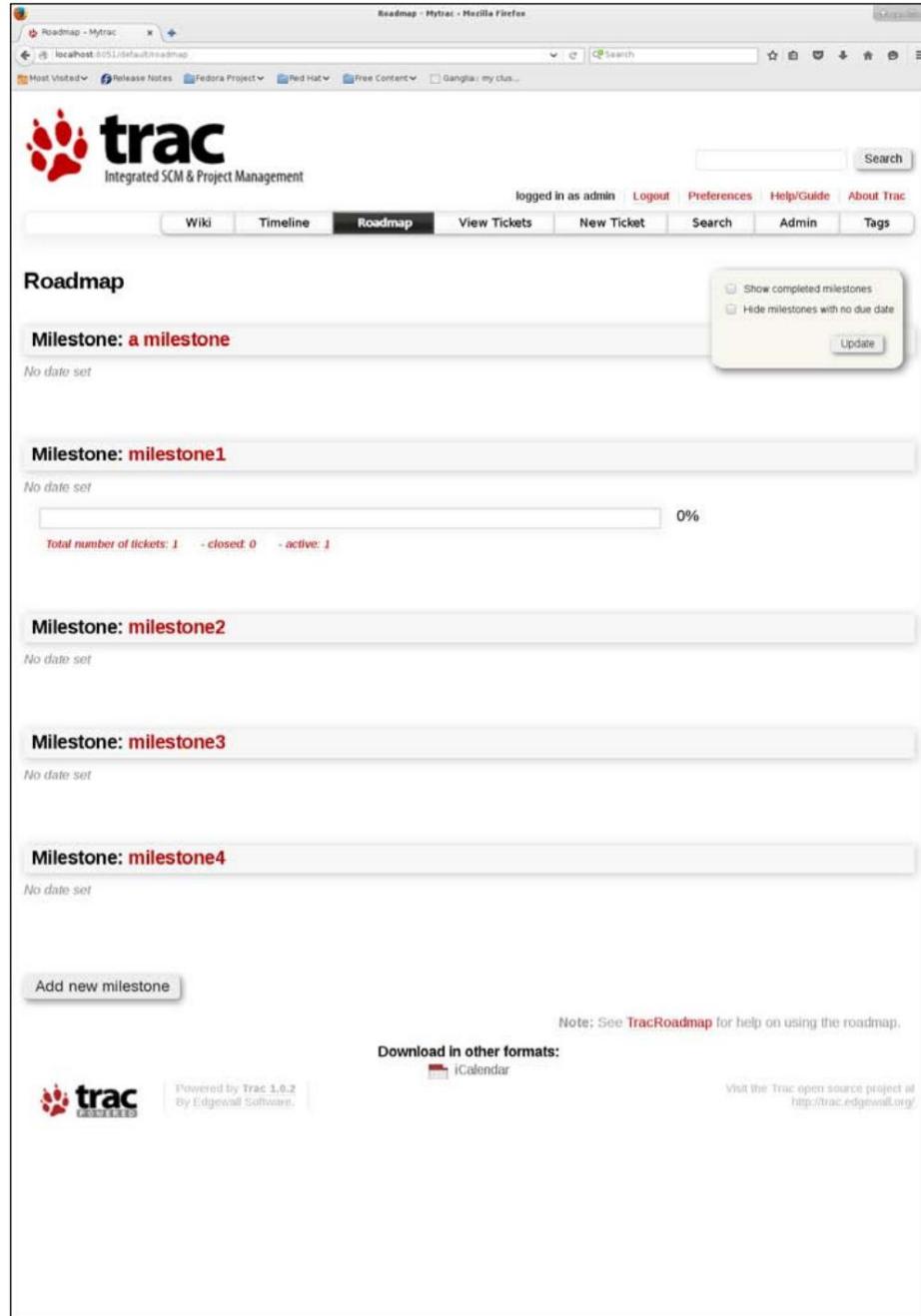
问题跟踪

· Wiki是一个传统的wiki,例如,您可以在其中编写对开发有用的文档。

·时间轴将显示 Trac 中发生的事件:



·路线图将显示按里程碑组织的工单：



问题跟踪

·查看门票,顾名思义,显示门票。有几种可能的报告,您可以创建新报告。这是一个非常有用的功能,可以让不同的团队和角色看到与他们相关的信息:

The screenshot shows the Trac web interface with the following details:

- Header:** Available Reports - My... | localhost:4051/default/report | Mozilla Firefox
- User Information:** logged in as admin | Logout | Preferences | Help/Guide | About Trac
- Main Navigation:** Wiki, Timeline, Roadmap, View Tickets (selected), New Ticket, Search, Admin, Tags
- Sub-navigation:** Available Reports | Custom Query
- Section:** Available Reports
- Buttons:** Show Descriptions, Clear
- List:** SQL reports and saved custom queries
 - {1} Active Tickets
 - {2} Active Tickets by Version
 - {3} Active Tickets by Milestone
 - {4} Accepted, Active Tickets by Owner
 - {5} Accepted, Active Tickets by Owner (Full Description)
 - {6} All Tickets By Milestone (including closed)
 - {7} My Tickets
 - {8} Active Tickets, Mine first
- Buttons:** % Edit, - Delete
- Links:** Create new report
- Note:** Note: See [TracReports](#) for help on using and creating reports.
- Download:** Download in other formats: RSS Feed, Comma-delimited Text, Tab-delimited Text
- Powered By:** trac | Powered by Trac 1.0.2 | By Edgewall Software.
- Project Information:** Visit the Trac open source project at <http://trac.edgewall.org/>

第9章

现在,尝试以管理员身份使用用户名和密码登录。

您现在获得一个名为New Ticket 的新菜单栏条目。尝试创建一张票。在您键入时,实时预览将在窗口的下部更新:

The screenshot shows the 'Create New Ticket' interface in a Mozilla Firefox browser. The URL is `localhost:8051/default/newticket`. The top navigation bar includes links for 'Wiki', 'Timeline', 'Roadmap', 'View Tickets', 'New Ticket' (which is highlighted in black), 'Search', 'Admin', and 'Tags'. A search bar is also present.

Create New Ticket

Properties

Summary: a new feature of great importance for the future of our product

Reporter: admin

Description:

B I A wysiwyg textarea You may use [WikiFormatting](#) here.
this feature really is pretty good and will be of great importance for the happiness of our customers.

Type: defect

Milestone:

Version:

Cc:

Owner: < default >

Priority: major

Component: component1

Keywords:

Parent Tickets:

I have files to attach to this ticket

[Preview](#) [Create ticket](#)

new defect (ticket not yet created)

a new feature of great importance for the future of our product

Reported by: admin Owned by: < default >
Priority: major Milestone:
Component: component1 Version:

Keywords:
Parent Tickets:

Description
this feature really is pretty good and will be of great importance for the happiness of our customers.

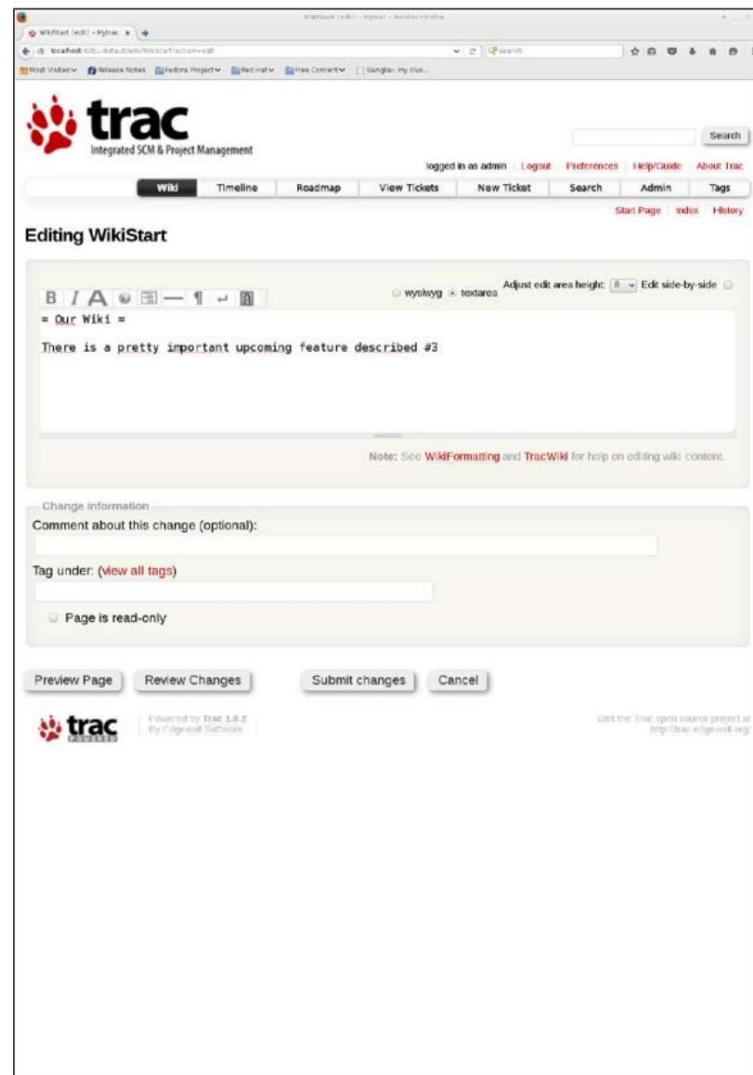
Note: See [TracTickets](#) for help on using tickets.

Powered by Trac 1.0.2 | Visit the Trac open source project at <http://trac.edgewall.org/>

问题跟踪

工单现在处于“新”状态。我们可以使用修改按钮更改状态。
尝试解决工单并对其发表评论。下一个状态将是“关闭”。

由于 Trac 的主要卖点之一是 Wiki、问题跟踪器和存储库的紧密集成，让我们尝试一下集成。编辑主要的 Wiki 页面。Trac 使用简单的 wiki 语法。创建一个新的标题，使用等号来标记它，如下面的截图所示：



当我们提交更改时,问题标识符变为可点击,我们可以访问该问题。

如果您在代码更改的提交消息中提到错误,Trac 也会在其更改集浏览器中将其设置为可点击。这是一个受 wiki 启发的简单方便的功能。

管理界面允许自定义 Trac 实例。您可以直接在界面中管理 Trac 中的许多实体,但不可能从这里管理 Trac 的各个方面。您需要在文件系统中配置以下一些方面:

- 用户
 - 组成部分 · 里程碑
 - 优先事项
 - 决议
 - 严重性 · 工
- 单类型

Trac 有一个可配置的状态机。默认情况下,新安装的状态机只比最小的打开/关闭状态机稍微复杂一点。

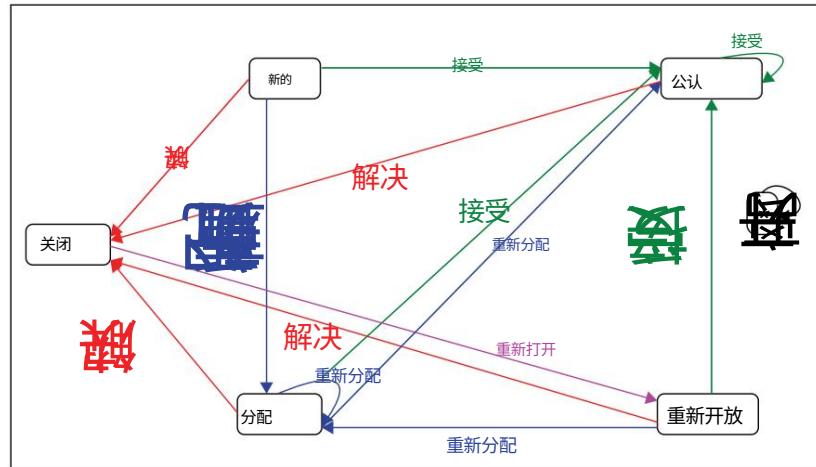
状态机如下:

- 新的
- 已分配 · 已接

受

问题跟踪

- 关闭
- 重新开放



Trac 发行版以.ini文件的形式提供了一组示例工作流配置,用于描述工作流状态和转换。目前无法在 GUI 中编辑工作流程。

我们对 Trac 问题跟踪器的研究到此结束。如果您喜欢简单性和工具集成,它很有吸引力。

Redmine

Redmine 是一个流行的问题跟踪器,用 Ruby on Rails 编写。它的设计与 Trac 有许多相似之处。

Redmine 也有两个分支:ChiliProject 和 OpenProject。虽然 ChiliProject 的开发停滞不前,但 OpenProject 仍在继续开发。

Redmine 是免费软件。它与 Trac 相似,因为它将多个工具集成到一个连贯的整体中。它也是基于网络的,就像我们在这里探索的其他问题跟踪器一样。

与 Trac 的开箱即用体验相比,Redmine 具有许多附加功能。其中一些如下:

- 可以从网络界面管理多个项目
- 可以使用甘特图和日历

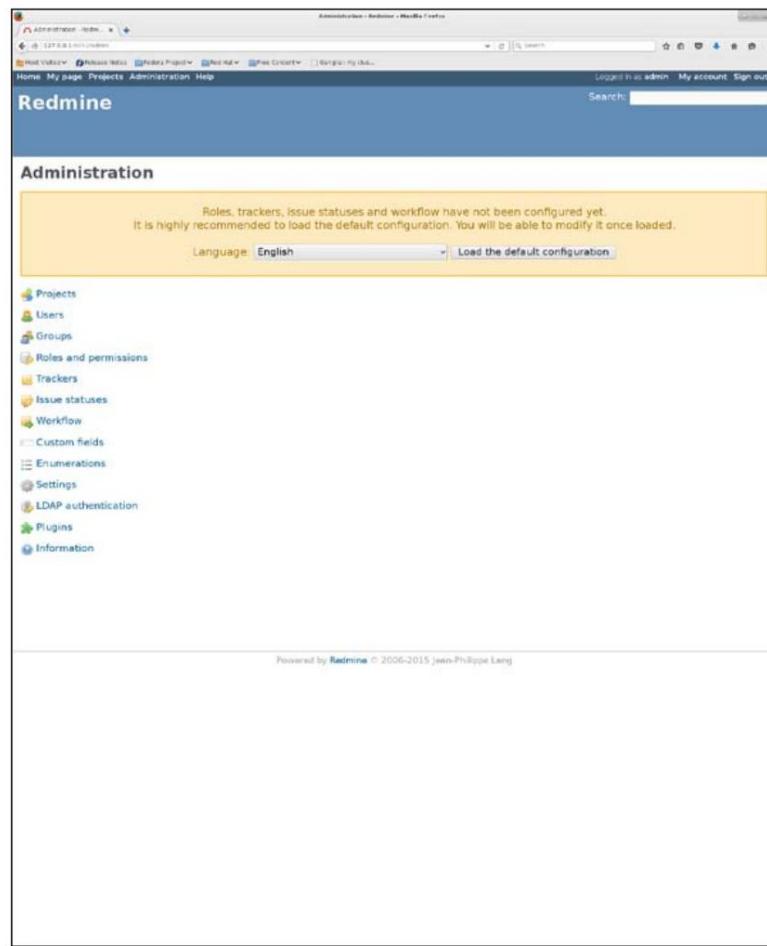
本书的代码包中有一个适用于 Redmine 的 Dockerfile,在 Docker 中心也有一个官方的 Redmine 镜像。看看这个命令：

```
docker run -d -p 6052:3000 红矿
```

这将在主机端口 6052 上启动一个可用的 Redmine 实例。该实例将使用 SQLite 数据库,因此它不会扩展到生产用途。如果您决定进行生产安装,您可以将 Redmine 容器配置为使用 Postgres 或 MariaDB 容器作为数据库。

您可以使用默认的用户名和密码登录,均为admin。

在继续之前,您需要手动初始化数据库。使用页面顶部的管理链接。按照页面上的说明进行操作:



问题跟踪

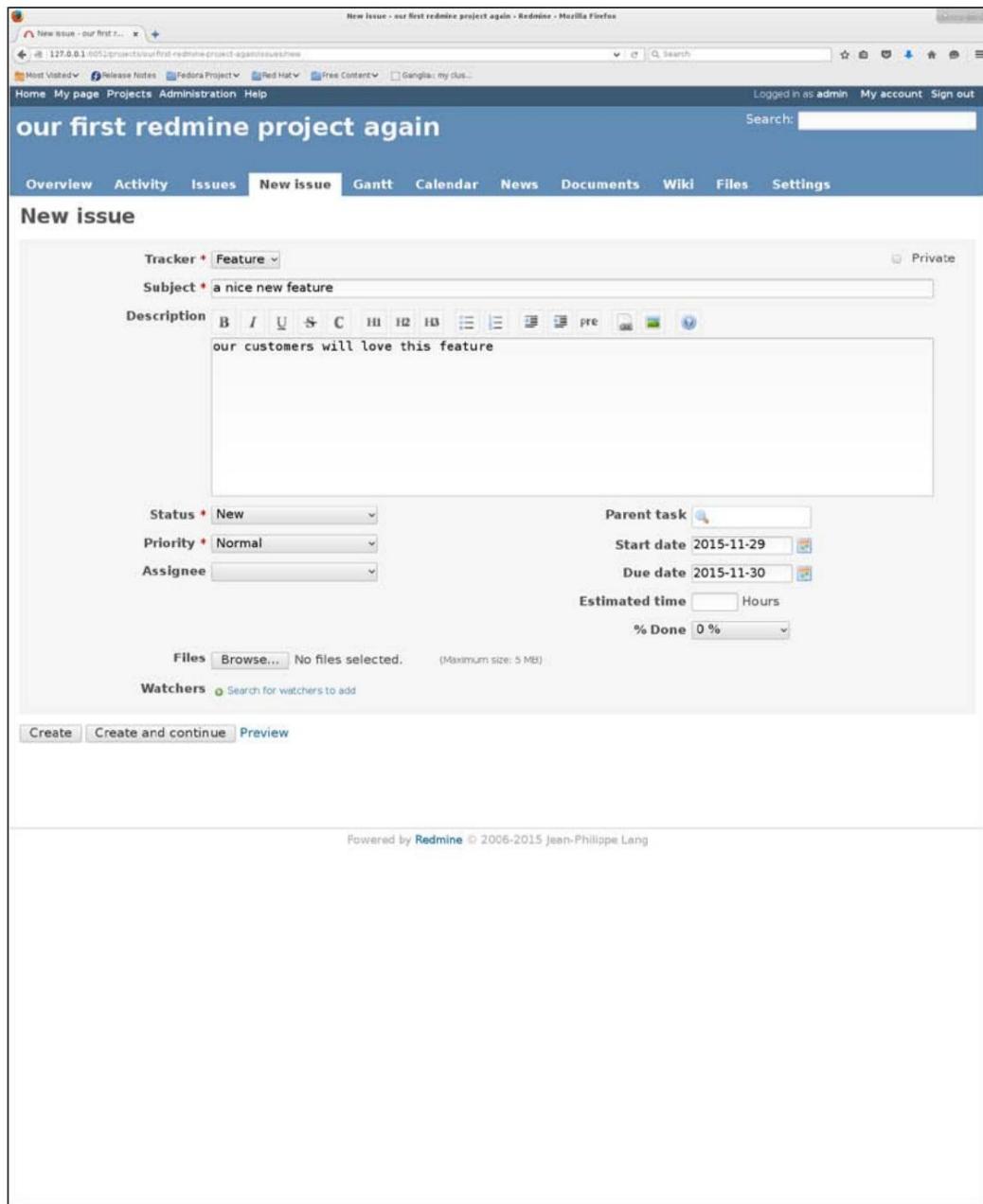
您可以从用户界面中创建一个新项目。如果不先按照前面的数据库初始化步骤进行，新的项目就没有必要的配置，也就无法创建issues。Redmine的新项目视图是

包含在下面的屏幕截图中：

The screenshot shows the 'New project' creation interface in Redmine. The 'Name' field contains 'our first redmine project'. The 'Identifier' field contains 'our-first-redmine-project'. The 'Homepage' field is empty. The 'Public' checkbox is checked. The 'Inherit members' checkbox is unchecked. Under 'Modules', the 'Issue tracking' checkbox is checked, while 'Documents', 'Repository', and 'Gantt' are unchecked. The 'Time tracking', 'Files', 'Forums', 'News', 'Wiki', and 'Calendar' checkboxes are checked. At the bottom, there are 'Create' and 'Create and continue' buttons.

第9章

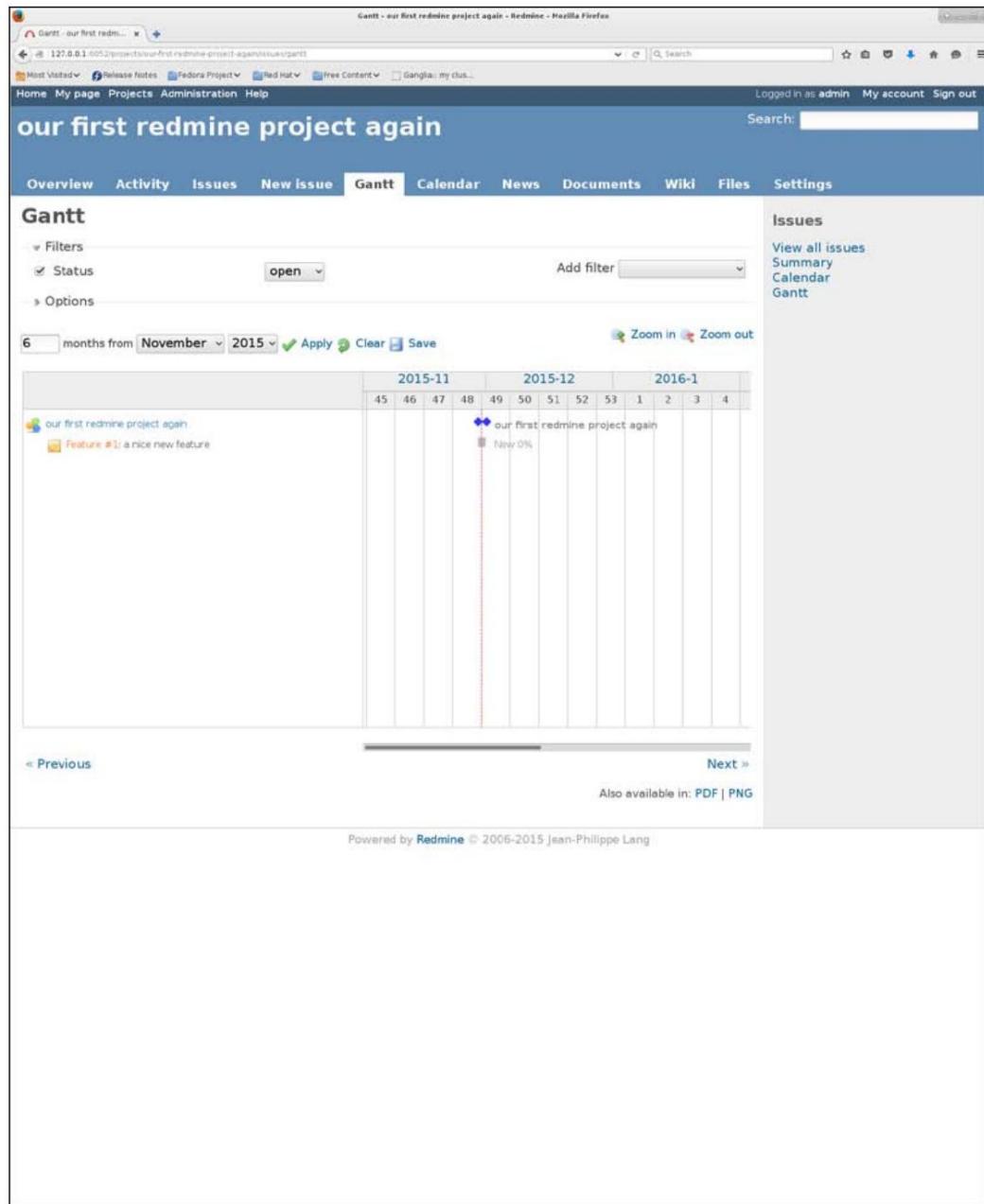
现在我们已经创建了一个项目，我们现在可以创建一个问题。使用 Redmine，您可以为您的问题选择一个类别：默认情况下，错误、功能和支持。让我们做一个新的功能问题：



问题跟踪

除了许多其他功能之外，还有甘特图视图和日历视图。

这是甘特图视图：



第9章

这是日历视图：

The screenshot shows the Redmine project management application's calendar interface. The title bar reads "Calendar - our first redmine project again - Redmine - Mozilla Firefox". The address bar shows the URL "127.0.0.1:4052/project/our-first-redmine-project-again/issues/calendar". The top navigation bar includes links for "Home", "My page", "Projects", "Administration", and "Help", along with a "Logged in as admin" message and "Sign out" link. A search bar is also present.

The main header "our first redmine project again" is displayed above the navigation menu which includes "Overview", "Activity", "Issues", "New issue", "Gantt", "Calendar" (which is currently selected), "News", "Documents", "Wiki", "Files", and "Settings".

The "Calendar" section has a "Filters" dropdown set to "Status" with "open" selected, and an "Add filter" button. It also shows the month and year as "November 2015" with "Apply" and "Clear" buttons. Navigation arrows for "October" and "December" are available.

The calendar grid spans from week 45 to week 49. The days of the week are labeled "Sunday" through "Saturday". Specific dates are highlighted with yellow boxes:

- Week 45: No specific highlights.
- Week 46: No specific highlights.
- Week 47: No specific highlights.
- Week 48: No specific highlights.
- Week 49:
 - Monday, November 29: Labeled "Feature #1: a nice new feature".
 - Tuesday, November 30: Labeled "Feature #1: a nice new feature".

A legend at the bottom left of the calendar area defines the symbols:

- Green diamond: "issue beginning this day"
- Red diamond: "issue ending this day"
- Red diamond with a green border: "issue beginning and ending this day"

The footer of the calendar section states "Powered by Redmine © 2006-2015 Jean-Philippe Lang".

问题跟踪

Redmine 问题具有以下默认状态机：

- 新的
- 进行中 · 已解决
- 反馈
- 关闭
- 被拒绝

总之,Redmine 是一个不错的跟踪器,具有许多不错的功能,并且建立在 Trac 的经验之上。

GitLab 问题跟踪器 正如人们所期望的那样,GitLab 的问题跟踪器与 Git 存储库管理系统集成得非常好。 GitLab 的问题跟踪器很漂亮,但在撰写本文时并不十分灵活。对于喜欢简单工具的团队来说,这可能已经足够好了。

GitLab 的开发速度很快,因此功能集可能会发生变化。 GitLab 问题跟踪器跟踪每个存储库的问题,因此如果您有许多存储库,可能很难同时获得所有问题的概览。

针对此问题的建议解决方案是创建一个单独的项目,其中收集了有关多个存储库的问题。

虽然 GitLab 问题跟踪器缺少竞争系统中的许多功能,但它确实有一个很好、灵活的 API,并且有一个 API 的命令行客户端,因此您可以方便地从 shell 中试用它。

测试 GitLab API 非常简单：

首先,安装 GitLab CLI

然后,设置以下两个描述端点的环境变量
GitLab API 和授权令牌：

- GITLAB_API_PRIVATE_TOKEN = <您项目的令牌> ·
- GITLAB_API_ENDPOINT = <http://gitlab.matangle.com:50003/api/v3>

现在您可以列出问题等等。有一个内置的帮助系统：

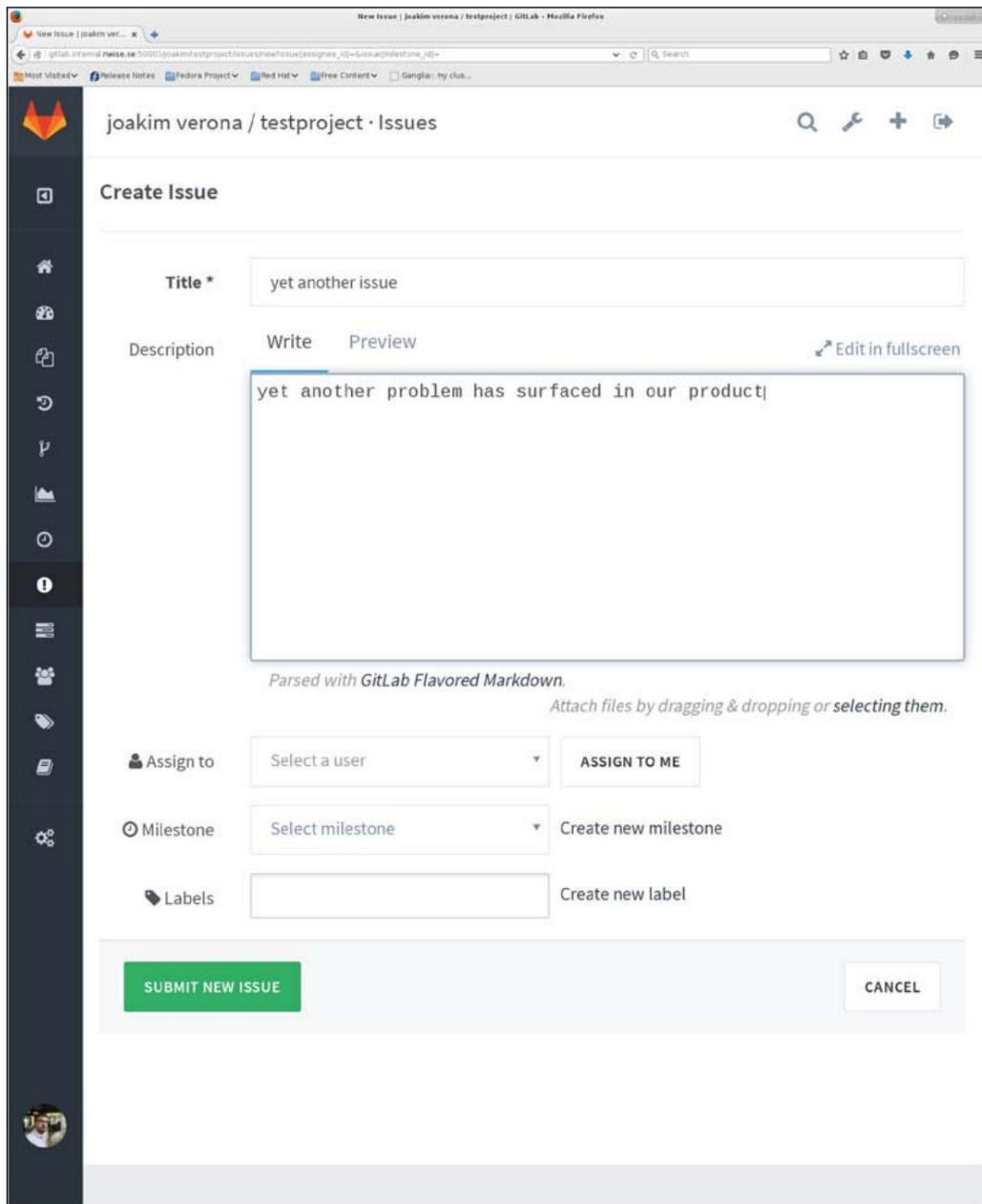
```
gitlab 帮助 Issues
+-----+
|       问题      |
+-----+
|关闭问题|
+-----+
|创建问题|
+-----+
|编辑问题|
+-----+
|问题      |
+-----+
|问题      |
+-----+
|重开问题|
+-----+
```

GitLab 问题只有两种常规状态：打开和关闭。这非常简单，但其想法是使用其他类型的元数据（例如标签）来描述问题。GitLab 问题跟踪器标签由文本和可与问题关联的背景颜色组成。可以关联多个标签。

还可以在问题描述中以 Markdown 格式嵌入待办事项列表。虽然标签和待办事项列表可以一起用于创建灵活的视图，但这并不能真正考虑到问题跟踪器状态机的传统用例。您必须记住自己需要做的事情：GitLab 问题跟踪器无法帮助您记住流程。小型团队可能会发现 GitLab 跟踪器中缺乏仪式感令人耳目一新，而且一旦您已经设置了 GitLab 就很容易设置您已经安装了它。

问题跟踪

用户界面非常直观。这是创建问题视图：



以下是该界面的一些功能：

- 分配给：可以分配为项目成员的人员。
- 标签：您可以拥有一组默认标签、创建自己的标签或混合使用。

以下是默认标签：

- 漏洞
- 确认
- 关键
- 讨论
- 文档
- 增强
- 建议◦ 支持

以下是可用于问题的一些属性：

- 里程碑：您可以将问题分组到里程碑中。里程碑描述了何时以时间线方式关闭问题。
- GitLab Flavored Markdown 支持： GitLab 支持 Markdown 和一些额外的标记，使 GitLab 中的不同实体更容易链接在一起。
- 附加文件

在 Bugzilla、Trac 和 Redmine 接口的复杂性之后，GitLab 问题跟踪器的严肃方法可能会让人感到耳目一新！

问题跟踪

吉拉

Jira 是 Atlassian 用 Java 编写的灵活的错误跟踪器。它是此处审查的唯一专有问题跟踪器。

虽然我更喜欢FLOSS (Free/Libre/Open Source Software) ,但Jira 的适应性很强,而且对于最多 15 人的小团队是免费的。与 Trac 和 GitLab 不同,wiki 和存储库查看器等不同模块是单独的产品。如果您需要 Wiki,您可以单独部署一个,而 Atlassian 自己的 Confluence Wiki 是集成最少的摩擦。Atlassian 还提供了一个名为Fisheye的代码浏览器,用于代码存储库的集成和浏览。

对于小型团队,Jira 许可很便宜,但很快就会变得更加昂贵。

默认情况下,Jira 的工作流程与 Bugzilla 工作流程非常相似:

- 开放 ·
- 进行中 · 已解决
- 关闭
- 重新开放

有几个可用的 Docker Hub 镜像;一个是cptactionhank/atlassian-jira 但请记住,Jira 不是免费软件。

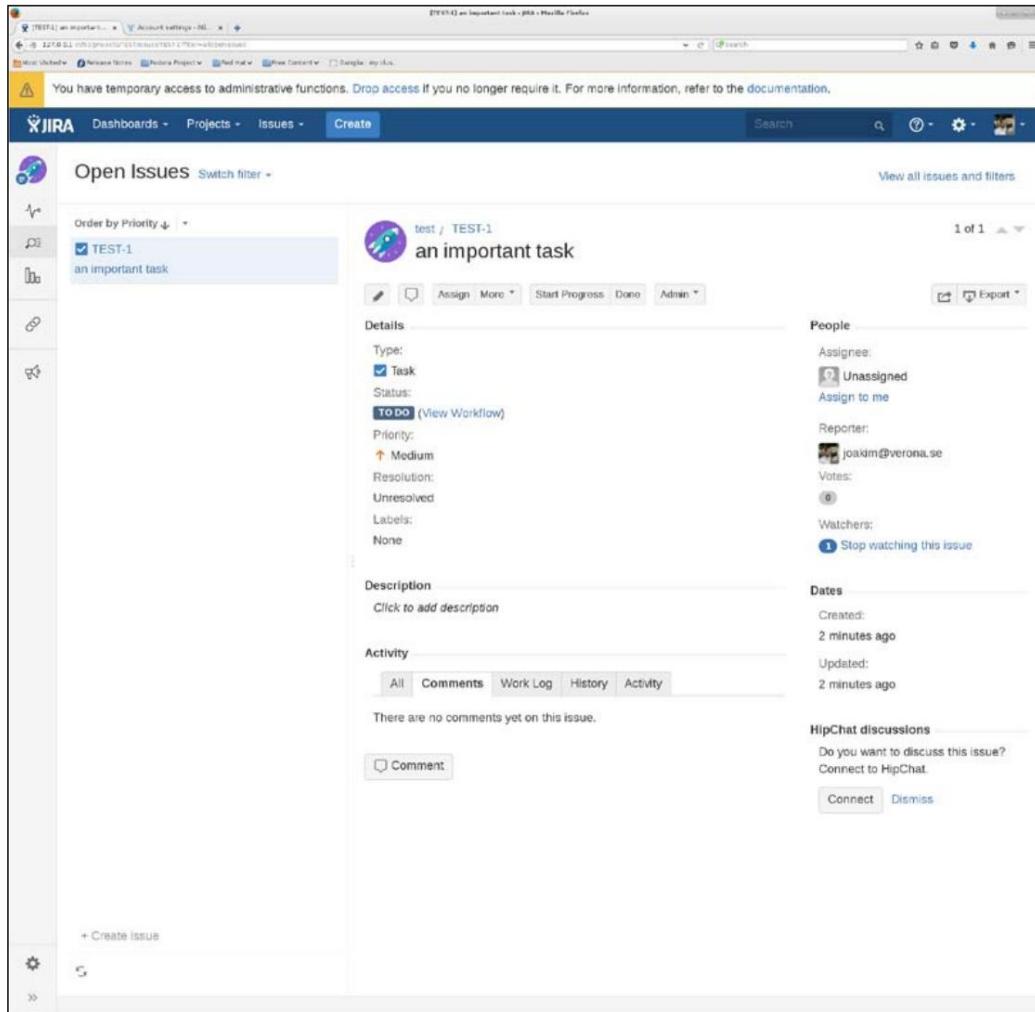
让我们现在试试 Jira:

```
docker run -p 6053:8080 cptactionhank/atlassian-jira:最新
```

可以在6053端口访问接口。Jira试用起来稍微复杂一些,因为在开始前需要先创建一个Atlassian账号。完成后使用您的帐户登录。

Jira 从一个简单的教程开始,您可以在其中创建项目和项目中的问题。如果您继续操作,结果视图将如下所示:

第9章



Jira 具有许多功能,其中许多功能可通过内置应用商店中的插件获得。如果您可以接受 Jira 不是免费软件,那么它可能是这里评论的问题跟踪器中外观最漂亮的。配置和管理起来也相当复杂。

问题跟踪

概括

在本章中,我们研究了可以在我们的组织中实施问题跟踪工作流支持的各种系统。和往常一样,有很多解决方案可供选择,尤其是在这个领域,这里的工具是每天都在使用的类型。

下一章将讨论更深奥的内容:DevOps 和物联网。

10

物联网和 DevOps

在上一章中,我们探讨了一些不同的工具选项,例如问题跟踪器,它们可用于帮助我们管理工作流。

本章将前瞻:DevOps如何在物联网这个新兴领域助力我们?

物联网 (简称 IoT)对 DevOps 提出了挑战。我们将探讨这些挑战是什么。

由于物联网不是一个明确定义的术语,我们将从一些背景开始。

介绍 IoT 和 DevOps

物联网一词是在 1990 年代后期创造的,据称是英国企业家凯文·阿什顿 (Kevin Ashton) 在研究 RFID 技术时创造的。 Kevin 在宝洁公司工作期间对使用 RFID 管理供应链产生了兴趣。

例如,RFID或射频 ID是您佩戴在钥匙链上并用于开门的小标签背后的技术。 RFID 标签是有趣事物的一个例子,在这种情况下,这些事物可以间接连接到互联网。当然,RFID 标签不仅限于开门,外形也不必局限于钥匙链上的标签。

物联网和 DevOps

RFID 标签包含一个大约 2 平方毫米的小芯片和一个线圈。当放置在阅读器附近时,线圈通过感应充电,芯片获得足够长的电力,以便将唯一标识符传输到阅读器的硬件。反过来,阅读器将标签的识别字符串发送到服务器,服务器决定是否要打开与阅读器关联的锁。服务器很可能连接到互联网,因此可以根据有关谁可以访问锁着的门的不断变化的情况,远程添加或删除与 RFID 标签相关联的标识符。几个不同的系统共生工作以达到预期的结果。

其他有趣的物联网技术包括被动二维码,可以通过摄像头扫描并向系统提供信息。较新的蓝牙低功耗技术提供了智能主动传感器。这种传感器可以运行长达一年

在锂电池上。

下面描述了两个用于操作锁的 RFID 标签:



“物联网”一词相当模糊。什么时候是“东西”，什么时候是电脑？或者两者兼而有之？

让我们以一台小型计算机为例，例如 Raspberry Pi；它是一个片上系统(SoC)，安装在信用卡大小的电路板上。它很小，但仍然是一台完整的计算机，并且功能强大到足以运行 Java 应用程序服务器、Web 服务器和 Puppet 代理。

与传统计算机相比，物联网设备受到各种方式的限制。

通常，它们是放置在难以访问、受限位置的嵌入式设备。那么，从 DevOps 的角度来看，这似乎是 IoT 的决定性特征：设备可能在不同方面受到限制。

我们不能使用我们在服务器或台式机上使用的每一种技术。限制可能是有限的内存、处理能力或访问，仅举几例。

以下是一些物联网设备的示例类型，您可能已经遇到或即将遇到：

·**智能手表**：如今，智能手表可以配备蓝牙和 Wi-Fi

连接并自动检测可用的升级。它可以下载新固件并根据用户交互进行自我升级。今天有许多智能手表可供选择，从功能强大的 Pebble 到 Android Wear 和 Apple Watch。

已经有带有移动网络连接的智能手表，最终，它们将变得司空见惯。·**键盘**：键盘可以有可升级的固件。键盘是

从某种意义上说，这实际上是物联网设备的一个很好的例子。它提供了许多传感器。

它可以运行软件，为最终连接到互联网的更强大的机器提供传感器读数。有带有开放固件的完全可编程键盘，例如 Ergodox，它将 Arduino 式板连接到键盘矩阵(<http://ergodox.org/>)。·**家庭自动化系统**：它们与互联网相连，可以

通过智能手机或台式电脑进行控制。Telldus TellStick 是此类设备的一个示例，可让您通过联网设备控制远程电源继电器。还有许多其他类似的系统。·**无线监控摄像头**：可以通过智能手机界面进行监控。有许多针对不同细分市场的供应商，例如

轴通信。

物联网和 DevOps

·**生物识别传感器**:此类传感器包括健身传感器,例如,脉搏计、身体秤和放置在身体上的加速度计。Withings 体重秤测量生物特征并将其上传到服务器,并允许您通过网站读取统计数据。智能手表和手机中也有加速度计,可以跟踪您的动作。

·**基于蓝牙的钥匙查找器**:如果您丢失了钥匙,您可以通过智能手机激活它们。

·**车载计算机**:它们可以处理从处理媒体和导航到管理车内物理控制系统(例如锁、窗和门)的所有事情。

·**音频和视频系统**:包括娱乐系统,例如网络音频播放器和视频流硬件,例如Google Chromecast 和 Apple TV。

·**智能手机**:无处不在的智能手机实际上是带有功能的小型电脑
将它们连接到更广泛的互联网的 3G 或 4G 调制解调器和 Wi-Fi。

·**嵌入存储卡中的具有 Wi-Fi 功能的计算机**:这些使它
可以将现有的 DSLR 相机转换为支持 Wi-Fi 的相机,可以自动将图像上传到服务器。

·**家庭或办公室的网络路由器**:这些实际上是小型服务器
通常可以从 ISP 端远程升级。

·**联网打印机**:如今这些打印机非常智能,可以与云服务交互,以便更轻松地从许多不同的设备进
行打印。

当然,这个清单可以继续下去,所有这些设备在今天的许多家庭中都很容易获得和使用。

有趣的是,从技术角度来看,提到的许多设备非常相似,即使它们用于完全不同的目的。大多数智能手机都使用 ARM CPU 架构的变体,这些架构可以与一些外围芯片变体一起获得跨制造商的认可。例如,MIPS 处理器在路由器硬件供应商中很受欢迎,而 Atmel RISC 处理器在嵌入式硬件实施者中很受欢迎。

基本的 ARM 芯片可供开发人员和爱好者使用,以便轻松制作原型。Raspberry Pi 基于 ARM,可用作专业用途的原型设计设备。Arduino 同样使 Atmel 架构设备可用于轻松制作硬
件原型。

这使得物联网创新的完美风暴成为可能：

- 这些设备很便宜,即使是小批量生产也很容易获得。 · 这些设备易于开发和获得原
型平台 · 开发环境相似,这使得它们更容易学习。很多时候,使用 GNU Compiler Collection 或
GCC。 · 在论坛、邮件列表、文档中提供大量支持,
- 等等。

对于 Raspberry Pi 等功能强大的设备,我们可以使用与服务器相同的方法和实践。Pi 设备可以是服务器,只
是不如传统服务器强大。对于 IoT 设备,无代理部署系统比需要代理的系统更适合。

更小的设备,例如 Arduinos 中使用的 Atmel 嵌入式 CPU,受到更多限制。通常,当特殊的引导加载程序代码
运行时,您编译新固件并在重启期间将它们部署到设备。然后设备通过 USB 连接到主机。

在开发过程中,可以通过连接一个单独的设备来自动上传固件,该设备重置原始设备并将其置于加载程序模
式。这在开发过程中可能有效,但在实际部署场景中并不划算,因为它会影响成本。这些是在使用 IoT 时可能会
影响 DevOps 的问题类型。在开发环境中,我们可能或多或少地使用我们习惯于开发服务器应用程序的方法,
也许需要一些额外的硬件。但是,从质量保证的角度来看,在与测试中使用的硬件不同的硬件上进行部署存在
风险。

根据市场的物联网的未来

据研究机构 Gartner 称,到 2016 年底,将有大约 64 亿物件连接到互联网:自 2015 年以来增长了 30%。

再远一些,到 2020 年底,估计将有 210 亿台联网设备。消费市场将负责最多的设备,而企业将负责最多的
支出。

物联网和 DevOps

暂称为 5G 移动网络的下一代无线网络预计将在 2020 年投放市场，在撰写本文时不会太远。即将推出的移动网络将具备适用于物联网设备的功能，甚至远远领先于当今的 4G 网络。

新移动网络标准的一些预计功能包括：

- 能够连接比当今网络更多的设备。这将使部署具有数十万个传感器的大规模传感器网络成为可能。
- 每秒数万兆比特的数据速率，用于数万个用户。在办公环境中，每个节点都将提供每秒 1Gb 的容量。
 - 极低的延迟，支持实时交互式应用程序。

无论 5G 计划在实践中的结果如何，可以安全地假设移动网络将快速发展，并且更多设备将直接连接到互联网。

让我们看看一些行业巨头是怎么说的：

- 我们从瑞典领先的移动网络硬件供应商爱立信开始。以下引自爱立信物联网网站：“Ericsson and the Internet of Things”

超过 40% 的全球移动流量在爱立信网络上运行。我们是业内最具创新精神的公司之一，拥有超过 35,000 项授权专利，我们正在通过降低企业创建现代通信技术支持的新解决方案的门槛来推动物联网的发展；通过打破行业之间的壁垒；并通过连接公司、个人和社会。展望未来，我们看到两个主要的机会领域：

行业转型：物联网正在成为一个又一个行业的关键因素，支持新型服务和应用，改变商业模式并创造新的市场。

不断发展的运营商角色：在新的物联网范式中，我们看到运营商可以扮演三种不同的可行角色。运营商选择扮演何种角色将取决于历史、雄心、市场先决条件和他们的未来前景等因素。”

第10章

爱立信还编写了一本很好的物联网概念视觉词典,他们称之为“漫画书”：http://www.alexandra.dk/uk/services/Publications/Documents/IoT_Comic_Book.pdf

- 网络巨头思科估计物联网将包括
到 2020 年将有 500 亿台设备连接到互联网。

思科更喜欢使用术语“万物互联”而不是物联网。该公司设想了许多以前未在此处提及的有趣应用。其中一些是：

- 智能垃圾桶,嵌入式传感器允许遥感垃圾桶的满满程度。然后可以改善废物管理的物流。
- 根据需求改变费率的停车计时器。
- 如果他或她生病了会通知用户的衣服。

所有这些系统都可以一起连接和管理。

- IBM 估计,到 2020 年将有 260 亿台设备连接到互联网,这可能有些保守。他们提供了一些已经部署或正在部署的应用程序示例：

- 使用广泛的传感器网络部署改善汽车制造商的物流
- 医院医疗保健情况下的更智能物流◦ 同样,更智能的物流和更准确的预测
火车旅行

不同的市场参与者对未来几年的预测略有不同。

无论如何,很明显,物联网在未来几年将呈近乎指数级增长。许多新的和令人兴奋的应用程序和机会将会出现。很明显,许多领域将受到这种增长的影响,自然包括 DevOps 领域。

机器对机器通信许多物联网设备将主要连接到其他机器。作为比例比较,我们假设每个人都有智能手机。这大约是 50 亿台设备。IoT 最终将包含至少 10 倍的设备 500 亿台设备。何时会发生这种情况在预测中略有不同,但我们最终会到达那里。

物联网和 DevOps

这种增长的驱动力之一是机器对机器的通信。

工厂已经在向更高程度的自动化迈进,而且这种趋势只会随着越来越多的选择变得可用而增长。通过广泛的传感器网络部署和大数据分析来不断改进流程,工厂内部的物流可以大大增加。

现代汽车几乎所有可能的功能都使用处理器,从灯光和仪表板功能到车窗升降器。下一步将是将汽车连接到互联网,并最终实现自动驾驶汽车,将所有传感器数据传递到中央协调服务器。

许多形式的旅行都可以从物联网中受益。

IoT 部署影响软件架构IoT 网络由许多设备组成,这些设备的固件版本可能不同。升级可能会及时分散,因为硬件可能在物理上不可用等等。这使得接口级别的兼容性变得很重要。由于小型联网传感器可能受内存和处理器限制,版本化二进制协议或简单的 REST 协议可能是首选。

为了允许具有不同硬件版本的事物在不同版本的端点进行通信,版本化协议也很有用。

大规模传感器部署可以受益于较少的对话协议和分层消息队列架构来异步处理事件。

物联网部署安全

安全是一个困难的话题,拥有大量连接到互联网而不是私人网络的设备并不能使情况变得更容易。许多消费类硬件设备(例如路由器)具有用于升级的接口,但也很容易被破解者利用。合法的服务设施因此成为后门。增加可用表面会增加潜在攻击向量的数量。

也许您从开发中认识到了其中一些反模式：

- 开发人员在代码中留下一条路,使他或她以后可以提交
将在服务器应用程序上下文中评估的代码。这个想法是,作为开发人员,您并不真正知道需要什么样的修补程序,也不知道在需要部署修补程序时是否有操作员可用。

那么为什么不在代码中留一个“后门”,方便我们需要时直接部署代码呢?当然,这里有很多问题。开发人员认为通常商定的部署过程不够高效,因此,破解者也可能很容易弄清楚如何使用后门。这种反模式比人们想象的更常见,唯一真正的补救措施是代码审查。

为破解者敞开大门从来都不是一件好事,你可以想象,如果可以利用自动驾驶汽车或热力发电厂的后门,那将是一场灾难。

- SQL 注入等意外攻击的发生大多是因为开发人员可能没有意识到该问题。

补救措施是了解问题并以避免问题的方式进行编码。

好的,但是 DevOps 和 IoT 又如何呢?

让我们退后一步。到目前为止,我们已经讨论了物联网的基础知识,它基本上是我们的普通互联网,但节点比我们通常认为可能的要多得多。我们还看到,在接下来的几年里,具有某种形式的互联网功能的设备数量将继续呈指数级增长。这种增长的大部分将出现在互联网的机器对机器部分。

但是,专注于快速交付的 DevOps 真的适合关键嵌入式设备的大型网络吗?

典型的反例是核设施或心脏起搏器等医疗设备中的 DevOps。但仅仅加快发布速度并不是 DevOps 的核心理念。它是通过让不同学科的人们更紧密地联系在一起做出更快、更正确的发布。

物联网和 DevOps

这意味着让类似生产的测试环境更接近开发人员,也更接近与他们一起工作的人。

这样描述,看起来 DevOps 确实可以用于传统上保守的行业。

但不应低估挑战:

- 嵌入式硬件设备的生命周期可能比那些传统的客户端-服务器计算机。不能期望消费者在每个产品周期都进行升级。同样,工业设备可能部署在更换成本高昂的地方。
- 物联网设备的故障模式多于台式计算机。
这使得测试更加困难。 · 在工业和企业部门,可追溯性和可审计性是重要的。例如,这与服务器上的部署相同,但物联网端点的数量比服务器多得多。 · 在传统的DevOps 中,我们可以处理小的变更集并将它们部署到我们的一部分用户。
如果更改以某种方式不起作用,我们可以进行修复并重新部署。如果一个网页对我们的已知用户子集呈现不佳并且可以快速修复,则只涉及很小的潜在风险。另一方面,如果即使是控制门或工业机器人等物的单个物联网设备出现故障,后果也可能是毁灭性的。

DevOps 在 IoT 领域面临着巨大的挑战,但替代方案并不一定更好。 DevOps 也是一个工具箱,你总是需要考虑你开箱即用的工具是否真的适合手头的工作。

我们仍然可以使用 DevOps 工具箱中的许多工具;我们只需要确保我们在做正确的事,而不是在不理解的情况下实施想法。

以下是一些建议:

- 只要您在您的测试实验室中,失败和快速周转是可以接受的 · 确保您的测试实验室与生产类似 · 不要只在您的实验室中拥有最新版本;容纳老年人
版本以及

带有物联网设备的动手实验室

开发运维

到目前为止,我们主要在抽象意义上讨论了 DevOps 和物联网以及物联网的未来。

为了亲身体验我们可以做什么,让我们制作一个连接到 Jenkins 服务器并显示构建状态的简单 IoT 设备。这样,我们就可以试用 IoT 设备并将其与我们的 DevOps 重点结合起来!

如果构建失败,状态显示将只显示一个闪烁的 LED。

该项目很简单,但可以由富有创造力的读者进行扩展。为本练习选择的 IoT 设备用途广泛,不仅可以使 LED 闪烁!

该项目将有助于说明物联网的一些可能性和挑战。

NodeMCU Amica 是一款基于乐鑫 ESP8266 芯片的小型可编程设备。除了基本的 ESP8266 芯片外,Amica 开发板还增加了使开发更容易的功能。

以下是设计的一些规格:

- 有一个 32 位 RISC CPU,即 Tensilica Xtensa LX106,运行频率为 80 MHZ。
- 它有一个 Wi-Fi 芯片,可以连接到我们的网络和 Jenkins 服务器。
- NodeMCU Amica 开发板有一个USB 插座,用于对固件进行编程和连接电源适配器。ESP8266芯片需要一个USB转串口适配器连接到USB接口,NodeMCU板上提供了这个。
- 该板有多个输入/输出端口,可以连接到某种硬件以可视化构建状态。最初,我们将保持简单,只使用连接到设备端口之一的板载 LED。 · NodeMCU 包含允许用Lua 语言编程的默认固件。Lua 是一种允许快速制作原型的高级语言。顺便说一句,它在游戏编程中很流行,这可能暗示了 Lua 的效率。

物联网和 DevOps

- 考虑到它提供的许多功能,该设备相当便宜:



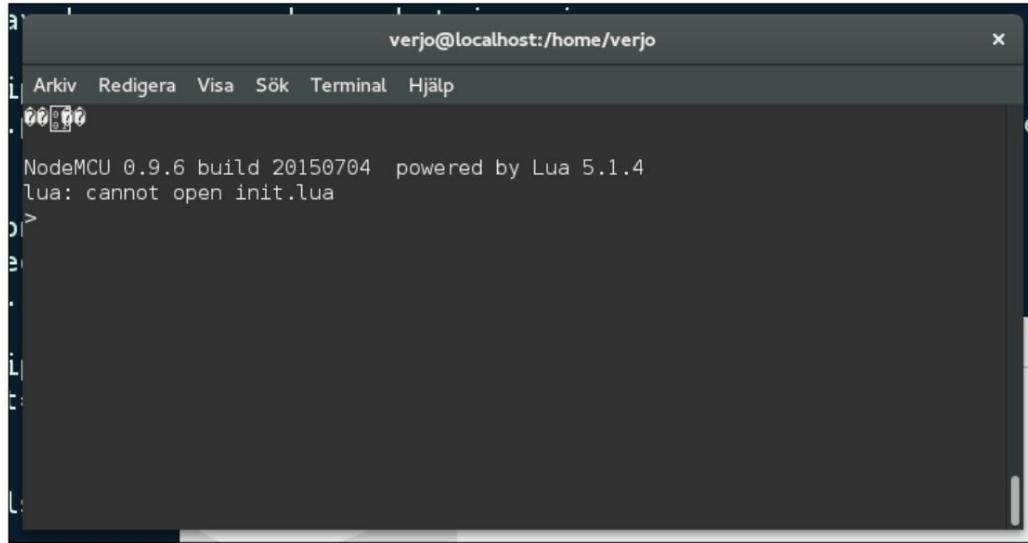
有很多选择可以从电子爱好者商店和互联网经销商处采购 NodeMCU Amica。

虽然 NodeMCU 不难获得,但从硬件的角度来看,该项目相当简单,如果事实证明这些设备更易于访问,也可以在实践中使用 Arduino 或 Raspberry Pi 进行。

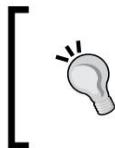
以下是开始使用 NodeMCU 的一些提示:

- NodeMCU 包含提供交互式 Lua 的固件
可以通过串行端口访问的解释器。您可以直接通过串行线路开发代码。在您的开发机器上安装串行通信软件。有很多选项,例如用于 Linux 的 Minicom 和用于 Windows 的 Putty。
- 使用串行设置9600 波特、八位、无奇偶校验和一位停止位。这通常缩写为 9600 8N1。
现在我们有了串行终端连接,将NodeMCU 连接到USB 端口,切换到终端,并确认您在终端窗口中得到提示。

如果您使用的是 Minicom,窗口将如下所示:



在开始编写代码之前,根据您的特定 NodeMCU 在出厂时的设置方式,可能需要将固件映像刻录到设备。如果在上一步有提示,则不需要刻录固件镜像。如果您需要图像中的更多功能,您可能想稍后再做。



要刻录新的固件映像,如果需要,请先从固件源存储库发布链接下载它。这些版本在 <https://github.com/nodemcu/nodemcu-firmware/releases> 提供

这是下载固件的示例wget命令。已发布的固件版本有整数和浮点形式,具体取决于您在数学函数方面的需要。基于整数的固件版本通常足以用于嵌入式应用程序:

```
 wget https://github.com/nodemcu/nodemcu-firmware/releases/download/0.9.6-dev_20150704/nodemcu_integer_0.9.6-dev_20150704.bin
```

[物联网和 DevOps](#)

您还可以直接在您的开发机器上本地从 GitHub 上的源构建固件映像,或者您可以使用在线构建服务根据您自己的规范为您构建固件。

在线构建服务位于<http://nodemcu-build.com/>。值得一试。
如果不出意外,构建统计图非常有趣。

现在您已经获得了合适的固件文件,您需要安装固件烧录实用程序,以便可以将固件映像文件上传到 NodeMCU:

```
git 克隆 https://github.com/themadinventor/esptool.git
```

按照存储库的自述文件中的安装说明进行操作。

如果您不想执行 README 中建议的系统范围安装,您可以从您的发行版安装 pyserial 依赖项并从 git-clone 目录运行该实用程序。

这是安装 pyserial 依赖项的示例命令:

```
sudo dnf 安装 pyserial
```

实际的固件上传命令需要一段时间才能完成,但会显示一个进度条,以便您了解发生了什么。

以下命令行是如何上传编写本文时最新的 0.9.6 固件的示例:

```
sudo python ./esptool.py --port /dev/ttyUSB0 write_flash 0x00000 nodemcu_integer_0.9.6-dev_20150704.bin
```

如果在连接 NodeMCU 时串行控制台出现乱码,您可能需要为固件刻录命令提供额外的参数:

```
sudo esptool.py --port=/dev/ttyUSB0 write_flash 0x0 nodemcu_integer_0.9.6-dev_20150704.bin -fs  
32m -fm dio -ff 40m
```

esptool 命令还有一些其他功能可用于验证设置:

```
sudo ./esptool.py read_mac  
正在连接...  
MAC: 18:fe:34:00:d7:21
```

```
sudo ./esptool.py flash_id
```

正在连接...

制造商 :e0

设备 :4016

上传固件后,重置 NodeMCU。

此时,您应该有一个带有 NodeMCU 问候提示的串行终端。

您可以使用工厂提供的 NodeMCU 固件或将新固件版本上传到设备来实现此状态。

现在,让我们先尝试一些 “hello world”风格的练习。

最初,我们只会使连接到 NodeMCU Amica 板的 GPIO 引脚 0 的 LED 闪烁。如果你有另一种类型的电路板,你需要弄清楚它是否有 LED 以及它连接到哪个输入/输出引脚,以防有。

当然,您也可以自己连接 LED。



请注意,电路板的某些变体将 LED 连接到 GPIO 引脚 3,而不是此处假设的引脚 0。



如果您的终端软件允许,您可以将程序作为文件上传到您的 NodeMCU,或者您可以直接在终端中输入代码。



NodeMCU 库的文档可从<http://www.nodemcu.com/docs/>获得,并提供了许多函数使用示例。



您可以先尝试点亮 LED:

```
gpio.write(0, gpio.LOW) -- 打开 LED
```

然后,使用以下命令关闭 LED:

```
gpio.write(0, gpio.HIGH) -- 关闭 LED
```

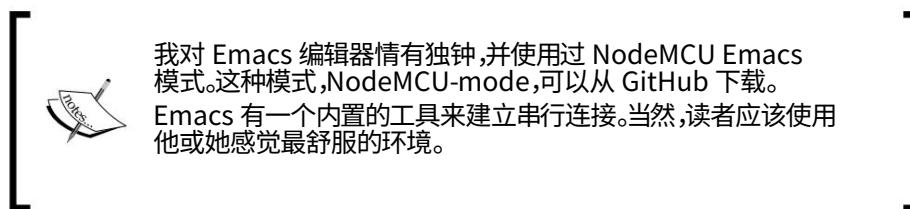
现在,您可以循环语句并穿插一些延迟:

```
while True:
    # 永远循环 gpio.write(0,
    #                      gpio.HIGH) -- 关闭 led tmr.delay(1000000) gpio.write(0, gpio.LOW) -- 打开 led
    # 等待一秒钟 tmr.delay(1000000)           -- 等一下
```

结尾

此时,您应该能够验证基本的工作设置。不过,直接在终端中输入代码有点原始。

NodeMCU 有许多不同的开发环境,可以改善开发体验。



在能够完成实验之前,我们需要一些额外的提示。

要连接到无线网络,请使用以下内容:

```
wifi.setmode(wifi.STATION)
wifi.sta.config( SSID , 密码 )
```

需要将SSID和密码替换为适合您网络的字符串。

如果 NodeMCU 正确连接到您的无线网络,此命令将打印它从网络的dhcpd服务器获取的 IP 地址:

```
打印 ( wifi.sta.getip () )
```

此代码段将连接到位于www.nodemcu.com的 HTTP 服务器并打印一个返回码:

```
conn=net.createConnection(net.TCP, false) conn:on( 接收 ,
function(conn, pl) print(pl) end)
conn:connect ( 80, "121.41.33.127" )
conn:send( GET / HTTP/1.1\r\n主机: www.nodemcu.com\r\n
.. 连接:保持活动状态\r\n接受:*\r\n\r\n\r\n )
```

您可能还需要一个定时器功能。此示例每 1000 毫秒打印一次hello world :

```
tmr.alarm(1, 1000, 1, function() print( hello world )  
结尾 )
```

在这里,Lua 的函数式范例得以体现,因为我们声明了一个匿名函数并将其作为参数发送给定时器函数。每隔1000毫秒就会调用匿名函数,也就是每秒。

要停止计时器,您可以键入:

```
tmr.停止 (1)
```

现在,您应该拥有自己完成实验的所有细节。
如果卡住了,可以参考本书源代码包中的代码。
快乐黑客!

概括

在最后一章中,我们了解了物联网这一新兴领域及其对 DevOps 的影响。除了对物联网的概述之外,我们还制作了一个连接到构建服务器并显示构建状态的硬件设备。

通过实际示例从抽象到具体,然后再回到抽象的想法一直是本书的主题。

在第1章DevOps 和持续交付简介中,我们了解了 DevOps 的背景及其在敏捷开发领域的起源。

在第2 章, Orbit 的视角,我们研究了持续交付管道的不同方面。

第3 章, DevOps 如何影响架构,深入探讨了软件架构领域以及 DevOps 的思想如何影响它。

在第4章 “一切皆代码”中,我们探讨了开发组织如何选择处理其重要资产 源代码。

第5 章,构建代码,介绍了构建系统的概念,例如 Make 和 Jenkins。我们探讨了它们在持续交付管道中的作用。

构建代码后,我们需要对其进行测试。这对于执行有效、无故障的发布至关重要。我们查看了第6章 “测试代码”中可用的一些测试选项。

物联网和 DevOps

在第7章部署代码中,我们探讨了最终将构建和测试代码部署到服务器的许多选项。

当我们的代码运行时,我们需要保持它运行。第8 章,监控代码,研究了确保代码正常运行的方法。

第9 章,问题跟踪,处理了一些可用的许多不同的问题跟踪器,它们可以帮助我们处理跟踪开发流程的复杂性。

这是本书的最后一章,这是一段漫长的旅程!

希望您和我一样享受这次旅行,祝您在 DevOps 的广阔领域中进一步探索取得成功!

指数

一个

敏捷开发周期 5 个替代
构建服务器 78

Ansible
大约 120 实
施 120-122 与 Puppet
127
Apache Commons Logging 153
apache log4net 153 appender,
log4j 154 架构规则 23 植物 24 内聚原
则 24 关注点分离 23 自动化验收测试
93-95 自动化 GUI 测试 95 自动化集
成测试 90

Bugzilla
关于 168 个

错误,创建 171、172 个错
误,解决 173、174 个错误
URL 168

使用 170,171 工
作流 169 构建链
74 构建依赖关系管
理 66,67 认真对待构建
错误 80 构建阶段 77、
78 构建服务器 76 构建
从属 72 构建状态可视
化 79 构建系统面 62,63

C

阿基利安 92
码头工人 91
AWS 132
蔚蓝 132

乙

后端集成点测试 98,99 行为驱动
开发 93 框 129 分支问题定义
45,46 分支策略选择 43-45

cargo cult 敏捷谬误 6,7 中央代
码存储库托管 39

Chef
about 124
deploying with 124, 125
client selecting 47, 48
client-side logging
libraries log4j 153 logback
153 cloud solutions 131
code building 61 defining
41

在客户端上执行 118 监控 135 比
较表 131

并发版本系统 (CVS) 40
持续交付 70
持续交付、DevOps 流程 9、10 工件存储库 14 构建服务
器 13 开发人员 11、12 包管理器 14 修订控制系统
12、13 暂存/生产 16 测试环境 15、16

CRUD (创建、读取、更新和删除) 25

丁

数据库迁移更改日志文件 29
处理 27 hello world，在
Liquibase 29 pom.xml
文件 30、31 升级、滚动 28 数据层 26 交
付管道 18 依赖顺序构建 76、77 部署系
统关于 111、112 基本操作系统，配置
112 集群，描述 113 个包，向系统交付 114、
115 DevOps 关于 1、35 和 ITIL 7、8 架构
35、36 敏捷周期的好处 5 快速周转流程 3、
4 概述 2、3 弹性 36 DevOps 流程瓶颈，
确定 19 持续交付 9、10

示例 18、19 分布式版
本控制系统的优点 42

芭莎 43
水星 43
Docker
about 52
defining 52, 53
deploying with 130
网址 53
码头群 130

乙

弹性搜索 155
ELK 箱约 155

弹性搜索 155
基巴纳 155
Logstash 155
工作 156、157
Ergodox
网址 197

F

功能切换 46 最终神器 67、
68
FLOSS (免费/自由/开源软件)
192
FPM
与 68、69 作弊

G

Ganglia 大
约 145 个组
件 147
Gmetad 147
金蒙德 147
基于 PHP 的 Web 前端 147
RRD 147
网址 145
Ganglia 监控解决方案
约 145 探索 146-149

Gerrit 定
义 53

git-review 包,安装 54 历史,定义 55-57
网址 54
小黄瓜 94
混帐 42
Git 附件约 51

网址 52
Git 流程 44
GitHub 特性 50
GitLab 约 57-59 特征 50
网址 57
GitLab 问题跟踪器 188-191
Git LFS 51
Git 服务器实施 52 设置 48,49
Gmetad 147
金蒙德 147
石墨约 150、151
碳度量处理守护进程 150 个组件 150
Graphite-Web 组件 150
Whisper 时序数据库库 150

H
在 73
一个托管 Git 服务器上
托管软件,定义 50,51
主机服务器 72 修补
程序分支 45 管理程序
117

信息辐射器 79 集成测试 69
物联网 (IoT) 大约 195 和 DevOps
195,196,203 设备 197,198 未来 199-201

物联网部署
影响软件架构 202 安全 202
用于 DevOps 动手实验室的 IoT
设备 205-210 问题跟踪器
about 159, 160

Bugzilla 168 注意事项 163-166
GitLab 问题跟踪器 188,189
Jira 192 扩散问题 166,167
红矿 182
Trac 174 问题跟踪系统关于 167 参考 167

杰
JavaScript 测试 97,98
詹金斯 61
Jenkins 构建服务器 63-66
Jenkins 文件系统布局 75,76
詹金斯插件 70,71
Jira 192,193 作品
业链 74
JUnit
约 83
断言 88
示例 89 一般
87,88 特别 87,88 测试
用例 87 测试执行 88 测试
夹具 87 测试结果格式化程序 88 测试运行程序 87 测试套件 88

钾
看板 17
基巴纳 155
库伯内斯 130

大号	基于 PHP 的 Web 前端 147 管道 74 实际示例 25 表示层 26 printf 样式调试 152
大型二进制文件 51.52 log4c 153 log4j 153 log4js 153 logback 153 日志处理 152 逻辑层 26	Program Increments 5 拉取请求模型定义 57
日志存储 155	Puppet 与 PalletOps 127
米	傀儡特工 119
机器对机器通信 201 手动安装 31.32 手动测试 83.84 微服务	Puppet 生态系统缺点 120 优点 120 傀儡师 119
关于 32 和数据层 35 模拟 89 单体场景 22-24	问 质量措施整理 78
穆宁 关于 141、142 实例 142 监控解决方案,探索 143-145	R 变基 55 Redmine about 182 使用 用 182-188 发布管理 17 REPL 驱动的开发 100.101 稳健性 80 角色定义 41
否	
纳吉欧斯 135 Nagios 监控解决方案关于 135 主动检查 138 探索 135-141 被动检查 138	RRD (循环数据库) 143、147
NodeMCU 阿米卡 205 N单元 87	小号
欧	盐与 Ansible 127 SaltStack about 125 deploying with 125, 126 敏捷 17 Scrum Sprint 周期 5
公共汽车 124	硒 83 硒测试 在 Jenkins 96.97 中集成 语义版本 网址 47
P	
PalletOps 大约 123 实施 124 半虚拟化 117 性能测试 92	

服务接口

维护向前兼容的 34 位共享认证

三层系统大约 25、26 数
据层 26 逻辑层 26 表
示层 26

定义 50

Simple Logging Facade for Java (SLF4J) 153 软件架构
21, 22 源代码控制需要 40 源代码管理示例 40 历史 40, 41 源代
码管理系统迁移, 定义 43 使用 42 压缩 55

宽容的读者 34

Trac

约 174
URL 174

使用 176-182

触发器 74

Ü

单元测试缺点
86

V

Vagrant 128-130 版本
命名定义 46, 47 原则
46 虚拟化堆栈
116-118

W

工作流程 74, 75 工作
流程和问题示例 161, 162

X

xUnit 87

颠覆 40

S单元 87

片上系统 (SoC) 197

吨

测试自动化缺点

84-86 优点

84-86

测试自动化场景

关于 101 自动

化测试, 运行 105 错误, 发现 105 测试演

练 105-108 棘手的依赖关系, 使用

Docker 处理 109 web 应用程序, 手动

测试 102-104

测试覆盖率 89, 90 测试驱动

开发 (TDD) 大约 100 事件序列 100

Machine Translated by Google



感谢您购买
实用的 DevOps

关于 Packt Publishing

Packt,读作“packed”,于2004年4月出版了第一本书《掌握phpMyAdmin以实现有效的MySQL管理》,随后继续专门出版关于特定技术和解决方案的高度集中的书籍。

我们的书籍和出版物分享了您的IT专业人员同行在调整和定制当今系统、应用程序和框架方面的经验。我们基于解决方案的书籍为您提供定制软件和技术以完成工作的知识和能力。Packt书籍比您过去看到的IT书籍更具体,更不笼统。我们独特的商业模式使我们能够为您提供更有针对性的信息,为您提供更多您需要了解的信息,以及更少您不知道的信息。

Packt是一家现代而独特的出版公司,专注于为开发人员、管理员和新手社区制作高质量的前沿书籍。

如需更多信息,请访问我们的网站www.packtpub.com。

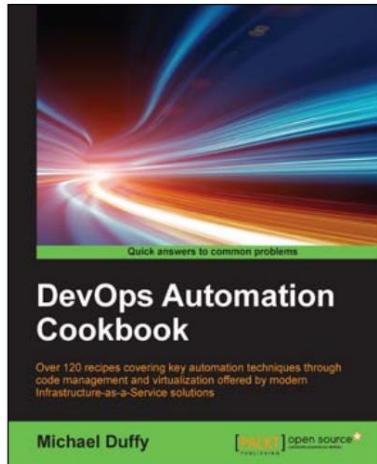
关于 Packt 开源

2010年,Packt推出了两个新品牌,Packt Open Source 和 Packt Enterprise,以继续专注于专业化。这本书是 Packt Open Source 品牌的一部分,该品牌是关于围绕开源许可证构建的软件的书籍的所在地,并为从高级开发人员到崭露头角的网页设计师的任何人提供信息。开源品牌还运行 Packt 的开源版税计划,根据该计划,Packt 向每个开源项目支付版税,以说明书籍的软件销售情况。

为 Packt 写作

我们欢迎对创作感兴趣的人的所有询问。书籍提案应发送至author@packtpub.com。如果您的图书创意仍处于早期阶段,并且您想在撰写正式的图书提案之前先进行讨论,那么请联系我们;我们的一位委托编辑将与您联系。

我们不仅在寻找已发表的作者;如果您有很强的技术能力但没有写作经验,我们经验丰富的编辑可以帮助您发展写作生涯,或者仅仅因为您的专业知识而获得一些额外奖励。



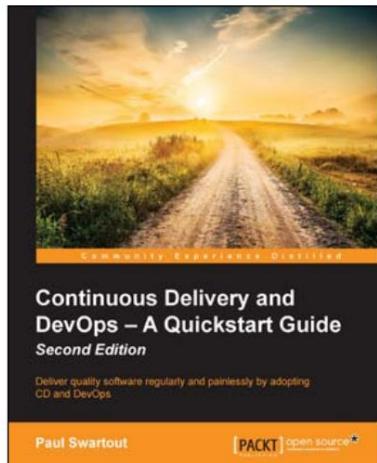
DevOps 自动化指南

书号:978-1-78439-282-6

平装本:334 页

超过 120 种方法涵盖了通过现代基础设施即服务解决方案提供的代码管理和虚拟化的关键自动化技术

1. 使用一些已经出现的强大工具,使系统管理员和开发人员能够控制和自动化复杂基础设施的管理、监控和创建。
2. 涵盖 DevOps 工程师可用的一些最激动人心的技术,并演示使用它们的多种技术。



持续交付和 DevOps – 快速入门指南

第二版

书号:978-1-78439-931-3

平装本:196 页

通过采用 CD 和 DevOps 定期、轻松地交付高质量的软件

1. 使用 DevOps 和持续交付
识别可能扼杀高质量软件交付并克服这些问题的潜在问题的方法。
2. 了解持续交付和 DevOps 如何与其他敏捷工具协同工作。
3. 充满插图和最佳实践的指南,可帮助您始终如一地交付高质量的软件。

请查看www.PacktPub.com以获取有关我们标题的信息



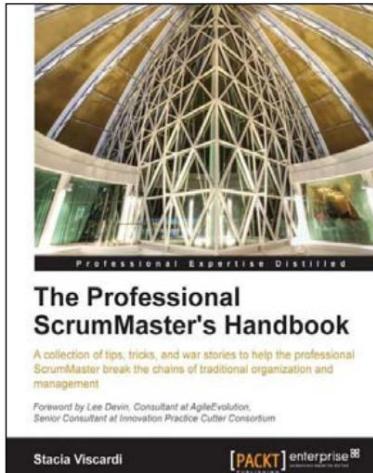
JIRA 敏捷基础

书号:978-1-78439-491-2

平装本:132 页

将敏捷的力量引入 Atlassian JIRA,并使用 Scrum 和看板高效运行您的项目

1. 通过将 JIRA Agile 与其他应用程序集成,轻松规划和管理项目。
2. 使用 Scrum 和 Kanban 以及敏捷方法提高团队的绩效。
3. 易于遵循的学习指南,用于安装 JIRA Agile 并了解它如何适应 Atlassian JIRA。



专业 ScrumMaster 的手册

书号:978-1-84968-802-4

平装本:336 页

帮助专业 ScrumMaster 打破传统组织和管理链条的提示、技巧和战争故事合集

1. 清单、问题和练习,让您像专业的 ScrumMaster 一样思考(和行动)。
2. 以轻松、无行话、风度翩翩的风格呈现。
3. 充满基于现实世界经验的想法、技巧和轶事。

请查看www.PacktPub.com以获取有关我们标题的信息